

# **Case Study Report: Architecture Evolution at Costco**

**Jeffrey M. Barnes**

December 2013

CMU-ISR-13-116

Institute for Software Research  
School of Computer Science  
Carnegie Mellon University  
Pittsburgh, PA 15213

This case study was carried out with financial support from the Software Engineering Institute. The data for the case study was collected at Costco Wholesale Corporation.

The views and conclusions expressed in this document are mine alone and should not be construed as representing the policies or opinions of Costco Wholesale Corporation, Carnegie Mellon University, or the Software Engineering Institute. Furthermore, where individual research participants are quoted, their comments are their own and should not be interpreted as representative of the policies or positions of their employer.

I would like to thank all four members of my thesis committee for their guidance and advice on this case study: Travis Breaux, Ipek Ozkaya, Kevin Sullivan, and especially my advisor, David Garlan. In addition, I would like to thank Jim Herbsleb for his advice on navigating the institutional review board process.

I would also like to express my deep and sincere gratitude to Shrikant Palkar at Costco for making this case study possible and ensuring its success. Finally, I would like to thank the anonymous interview participants for contributing their time.

**Keywords:** software architecture, software evolution, software engineering, specification, case study, qualitative research, content analysis

## **Abstract**

Many software systems eventually undergo changes to their basic architectural structure. As systems age, they often require redesign in order to accommodate new requirements, support new technologies, or respond to changing market conditions. At present, however, software architects lack tools to assist them in developing plans for carrying out such evolution.

In previous research, we have developed an approach to support architects in reasoning about evolution. Our approach is based on formally modeling and reasoning about possible evolution paths—sequences of transitional architectures leading from the current state to some desired target architecture.

To date, much of this work has been theoretical in character. We have focused our efforts on the definition and elaboration of a theoretical framework for architecture evolution, and have done relatively little work on applying this theoretical research in practical contexts.

This report presents a case study examining architecture evolution in depth in a real-world software organization: the IT division of a major global retailer. Based on content analysis of interview data and architectural documentation, the case study examines how practicing architects plan and reason about evolution, what challenges they face in doing so, and whether our modeling approach is adequate for capturing the concerns that arise in a real-world evolution. It finds that architects face a variety of pressing challenges in planning evolutions of major software systems, and use a variety of techniques to manage these challenges. It further finds that most of the major considerations that architects described in connection within a specific evolution can be successfully modeled using our approach, suggesting that the conceptual framework and specification languages that we have developed are expressive enough to be applicable to a real-world architecture evolution.



# Contents

<b>1 Problem</b>	<b>7</b>
1.1 Software architecture evolution . . . . .	7
1.2 Literature review . . . . .	9
1.3 Research questions . . . . .	9
1.4 Case study overview . . . . .	9
<b>2 Case selection</b>	<b>11</b>
2.1 About the case . . . . .	11
2.2 The architecture organization . . . . .	12
<b>3 Data collection</b>	<b>14</b>
3.1 Interview procedure . . . . .	14
3.2 Interview data . . . . .	15
3.3 Architectural documentation . . . . .	16
<b>4 Analysis</b>	<b>16</b>
4.1 Transcription . . . . .	16
4.2 Content analysis . . . . .	17
4.2.1 A very brief history of content analysis . . . . .	19
4.2.2 Qualitative versus quantitative content analysis . . . . .	19
4.2.3 Elements of a qualitative content analysis . . . . .	21
4.2.4 Coding frame . . . . .	22
4.2.5 Segmentation . . . . .	25
4.2.6 Pilot phase . . . . .	28
4.2.7 Main analysis phase . . . . .	28
4.3 Evolution model construction . . . . .	29
4.3.1 Initial and target architectures . . . . .	29
4.3.2 Evolution operators . . . . .	34
4.3.3 Constraints . . . . .	38
4.3.4 Evaluation functions . . . . .	46

<b>5 Findings</b>	<b>50</b>
5.1 Motives for evolution . . . . .	50
5.2 Causes of problems . . . . .	55
5.3 Consequences . . . . .	57
5.4 Challenges . . . . .	58
5.5 Approaches . . . . .	63
<b>6 Conclusions</b>	<b>68</b>
6.1 Answers to the research questions . . . . .	68
6.2 Quality considerations . . . . .	69
6.2.1 Reliability . . . . .	70
6.2.2 Validity . . . . .	75
<b>A Interview protocol</b>	<b>79</b>
A.1 Introductory consent script . . . . .	80
A.2 Collection of personal identifiers . . . . .	80
A.3 The participant's role and background . . . . .	81
A.4 Software architecture evolution at Costco . . . . .	81
A.5 Limitations of today's approaches to software architecture evolution . . . . .	81
A.6 Specifics about the evolution of particular software systems at Costco . . . . .	81
A.7 Conclusion . . . . .	82
<b>B Coding guide</b>	<b>82</b>
B.1 Content analysis 1: Learning how architects do evolution . . . . .	82
B.1.1 General principles . . . . .	82
B.1.2 Categories . . . . .	82
B.2 Content analysis 2: Modeling a real-world evolution . . . . .	94
B.2.1 General principles . . . . .	94
B.2.2 Categories . . . . .	94
<b>References</b>	<b>100</b>

# 1. Problem

Software architecture—the discipline of designing the high-level structure of a software system—is today widely recognized as an essential element of software engineering. However, one topic that current approaches to software architecture do not adequately address is software architecture evolution. Architectural change is commonplace in today’s software systems. As systems age, they often require redesign in order to accommodate new requirements, support new technologies, or respond to changing market conditions. At present, however, software architects lack tools to assist them in developing plans for carrying out such evolution.

In previous research [5], we have developed an approach to support software architects in reasoning about evolution. Our approach is based on formally modeling possible *evolution paths*—sequences of transitional architectures leading from the current state to some desired target architecture—and selecting among them on the basis of automatable analyses.

To date, much of this work has been theoretical in character. We have focused much of our effort on the definition and elaboration of a theoretical framework for software architecture evolution, and have done relatively little practical work on applying this theoretical research in practice in the context of a real-world software organization. Thus far, our only effort in the latter vein has been a single case study carried out at NASA’s Jet Propulsion Laboratory [3]. This case study focused primarily on the issues involved in adapting commercial modeling tools to work with our evolution modeling techniques; less attention was paid to general questions about the needs of real-world architects and the applicability of our approach.

This report presents a new case study focusing specifically on these questions. In addition to the difference in topical focus, the present case study is distinguished by a more methodical and rigorous research design. In particular:

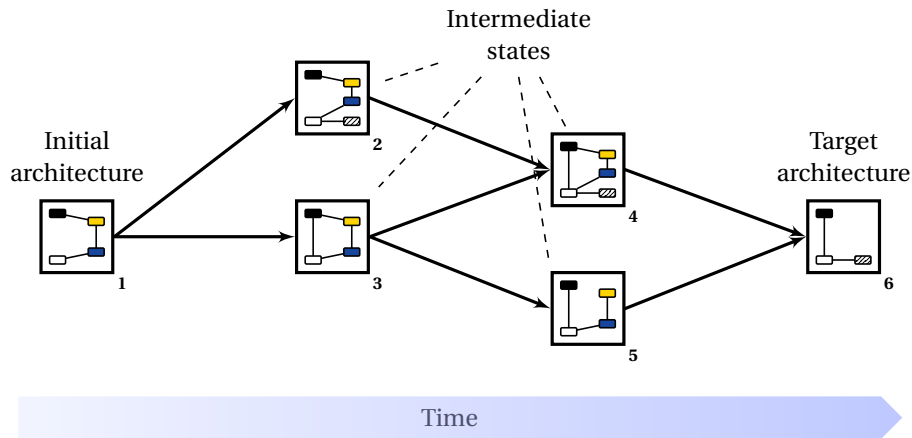
- The data in the present case study were collected through formal semistructured research interviews of practicing architects in a real-world software organization, supplemented by careful collection of related architectural documentation. In the previous case study, data collection was conducted informally during the course of an internship.
- The analysis is based on qualitative content analysis of the research data. The previous case study lacked any formal analysis.
- The findings of the case study are linked to the data by a rigorous case study design, with explicit consideration of validity and reliability. The conclusions of the previous case study were informal and impressionistic.

This case study report is organized as follows. The remainder of section 1 more formally defines the problem that this case study seeks to address by explicitly stating the research questions. It also presents additional necessary background on our previous research on architecture evolution. Section 2 describes the software organization that served as the subject of the case study. Section 3 describes the collection of data. Section 4 describes the analysis, including transcription of recorded interviews, content analysis, and creation of an evolution model. Section 5 describes the findings that resulted from the analysis. Section 6 concludes by returning to the research questions and showing how the case study addresses them.

## 1.1. Software architecture evolution

The full details of our approach are given elsewhere [5]. This section provides a brief summary of some of the key ideas necessary to understand the rest of this report.

Rearchitecting a software system can be a complex, multifaceted operation; often, an architect preparing to carry out an evolution must develop a plan to change the architecture and implementation of a system through a series of phased releases, ultimately leading to the target architecture. The architect’s task, therefore, is to develop a plan consisting of a series of



**Figure 1:** A depiction of an evolution graph. Each node in the graph is a complete architectural representation of the system. The edges of the graph represent possible evolutionary transitions. The architect’s task is to select the optimal path through the graph. This graph has only three paths: 1–2–4–6, 1–3–4–6, and 1–3–5–6.

architectural designs for intermediate states of the system as it evolves, beginning with the current state of the system and ending with the target state, together with a characterization of the engineering operations required to transform the system from its current architecture to the target architecture. This is the basis for the model of architecture evolution that underlies our research.

We assume that both the initial architecture (the current design of the system) and the target architecture (the intended design to which the system must evolve) are known. In fact, of course, this is often not the case. Especially in the case of an old, legacy system, design documents showing the initial architecture may be lost, fragmentary, or outdated, or they may never have existed at all. And similarly, the architect may be uncertain what the target architecture should be. However, other areas of research address these problems. The problem of learning the architecture of an existing system for which no architectural representation currently exists is addressed by research on *architecture reconstruction*, which provides tools for extracting an architectural representation of an implementation [29]. And the problem of determining a target architecture is addressed by research on *architectural design*, which provides guidance on designing a software architecture that meets a set of requirements [7]. Our research focuses instead on the problem of finding a way to evolve the system from a known initial architecture to a chosen target architecture—of charting a course from a known origin to a known destination.

Our approach is based on considering possible *evolution paths* from the initial architecture of the system (as it exists at the outset of the evolution) to the target architecture (the desired architecture that the system should have when the evolution is complete). Each such evolution path can be represented as a sequence of *transitional architectures* leading from the initial architecture to the target architecture. We can represent and relate these evolution paths within an *evolution graph* whose nodes represent transitional architectures and whose edges represent the possible transitions among them (figure 1). These transitions, in turn, may be understood as sequences of *evolution operators*—reusable architectural transformations such as *add adapter* or *migrate database*.

Once the evolution graph is defined, the next step is to apply analyses to select the optimal path—the one that best meets the evolution goals, while adhering to any relevant technical and business constraints, and subject to concerns such as cost and duration. To support the



architect in selecting a path, we provide two kinds of analysis: *evolution path constraints*, which define which paths are legal or permissible, and *path evaluation functions*, which provide quantitative assessments of qualities such as duration and cost. Operators and analyses are generally specific to particular domains of evolution; for example, an evolution of a desktop application to a cloud-computing platform will have different operators and analyses than an evolution of a thin-client/mainframe system to a tiered web services architecture.

These concepts are fairly simple, but there is a substantial formal framework supporting them. To formalize path constraints, for example, we have developed a language based on linear temporal logic. The use of formal methods has several advantages, the most important of which are precision (by using a formal approach, we minimize ambiguity, which helps to pin down what project stakeholders mean when they talk about the project) and automation (a formal approach allows us to develop tools to make it easier to plan and analyze the evolution).

## 1.2. Literature review

For a general review of related work on software architecture evolution, see our journal paper [5] or the recent literature reviews by Breivold et al. [14] and Jamshidi et al. [42]. For a review specifically on the topic of empirical research in software architecture evolution, see my conference paper on the Jet Propulsion Laboratory case study [3].

## 1.3. Research questions

This case study seeks to answer three basic research questions:

1. How do practicing architects in a real-world software organization plan and reason about evolution?
2. What difficulties do practicing architects face in planning and carrying out evolution?
3. How well can our modeling framework capture the concerns that arise in a real-world architecture evolution?

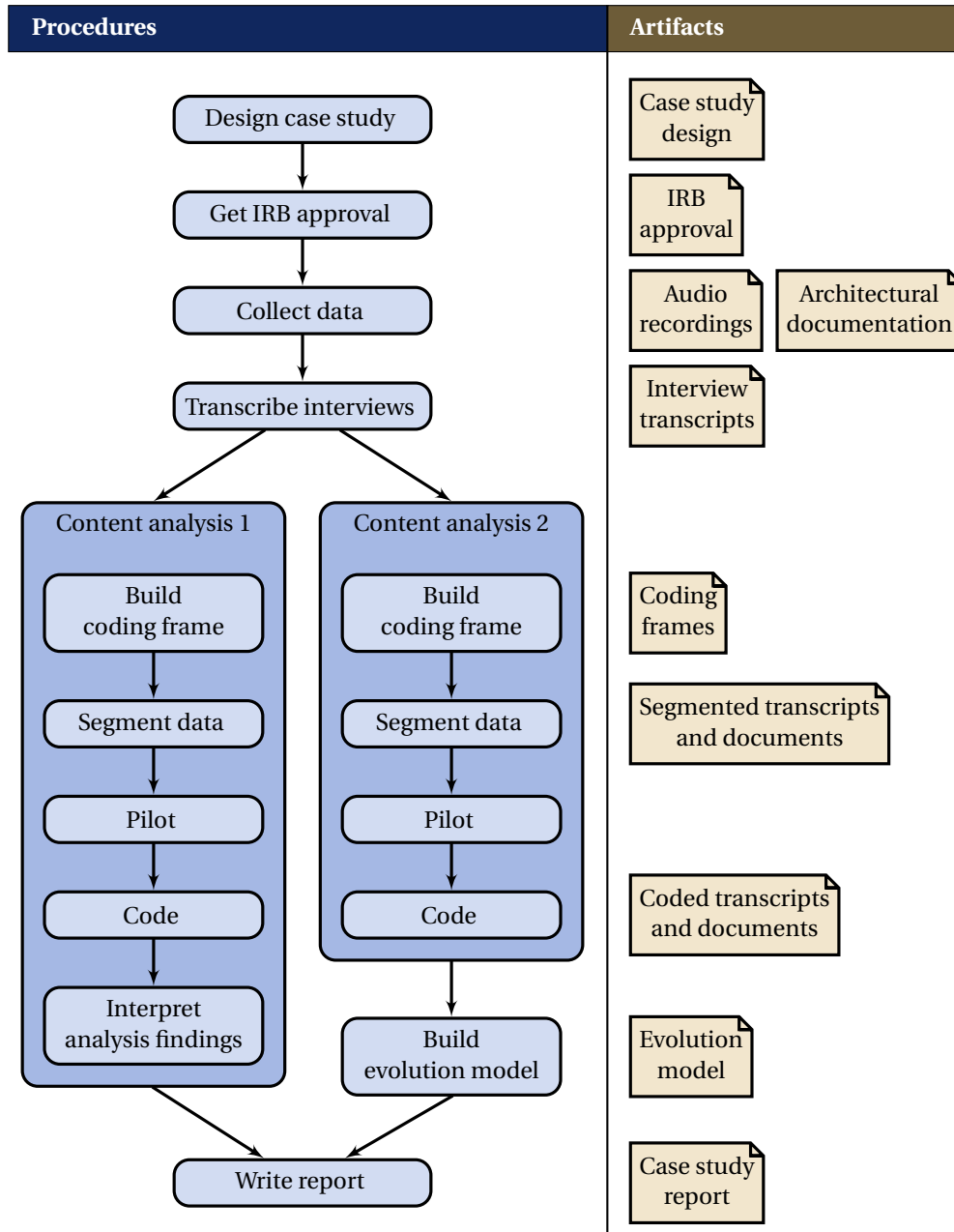
## 1.4. Case study overview

Figure 2 provides an overview of the design of this case study, which may be helpful to refer to while reading the rest of this report. This section explains the overall logic of the study—the procedures that link the research questions defined above to the eventual findings of the study.

The first step, of course, was case study design. The overall case study design is explained in this section, but see also section 2, which describes the organization that served as the subject of the case study. After designing the case study, we obtained approval to carry it out from Carnegie Mellon University’s institutional review board. For a brief discussion of ethical considerations in the collection of data from human research participants, see section 3.1.

With these design steps out of the way, the next phase of the case study was data collection. I spent two weeks conducting semistructured interviews with architects and examining a variety of architectural documentation. Section 3 describes the data collection procedures in detail. I transcribed the interviews soon after conducting them; see section 4.1.

The main analytical method used in this case study was content analysis. In fact, I conducted two content analyses in parallel: one examining research questions 1 and 2 as defined above, and one examining research question 3. I will explain the reasons for this decision in detail in section 4.2.4, but to summarize briefly, the use of two content analyses was motivated chiefly by the different characters of the research questions: research questions 1 and 2 are *descriptive* of the current state of practice of architecture evolution, while research question 3 is *evaluative* of our approach to evolution modeling. Research questions 1 and 2



**Figure 2:** An overview of the case study design.

can be answered directly through a content analysis of the interview data, while research question 3 will be answered through a two-stage analysis: a content analysis followed by a modeling phase. The content analysis will identify key architectural elements and evolution elements in the research data, and these results will feed into a modeling phase in which I construct a (partial) evolution model using our approach. The content analysis is explained in detail in section 4.2, and the modeling phase is described in section 4.3.

The conclusions of the case study appear in sections 5 and 6. Section 5 discusses the findings of content analysis 1, and section 6 discusses the overall conclusions of the case study (including both content analyses as well as the modeling procedure) with respect to the research questions defined above. Section 6 also discusses issues of case study quality: reliability and validity.

I should also note here that the final case study report was reviewed by Costco, which requested that I remove or edit certain passages containing information that they deemed to be confidential or sensitive. For example, the company requested that I remove all references to specific vendors associated with the company. I complied with all such requests. All of these changes were fairly minor and did not, in my judgment, materially influence the presentation of the overall findings of the case study.

## 2. Case selection

In this section, I describe the organization that served as the case for this case study. Section 2.1 gives general background on the company and explains why it was a suitable choice for this case study. Section 2.2 describes its architectural organization.

Some of the information that appears in this section, especially the information on the architectural organization, emerged from the data collected during the case study, but I include it here (rather than in the “Findings” section) because it helps to set the context for the rest of this report.

### 2.1. *About the case*

Costco Wholesale Corporation, founded in 1983, is the second-largest retailer in the United States [84] and the sixth-largest in the world [83]. In fiscal 2012, the company did \$97 billion in net sales [23]. Costco uses a warehouse club business model, meaning that it sells goods in large, wholesale quantities; thus, you can buy a 30-pack of toilet paper, but (unlike Wal-Mart or Target) not a 4-pack. Retail locations are called “warehouses” rather than “stores” and have a spartan decor, with concrete-floor aisles flanked by stacks of cardboard boxes. The company uses a membership-only model; only people who have purchased a membership may shop there. A one-year basic membership costs \$55 [23]. There are currently about 70 million members [24].

The company’s no-frills warehouses, volume purchasing, and membership model are part of what allows it to keep prices lower than competitors while remaining profitable. Other contributing factors are the company’s efficient distribution network, low marketing expenses, limited warehouse hours, rapid inventory turnover, and low payment processing fees (warehouses don’t accept credit cards other than American Express). Warehouses also have far fewer distinct items than typical discount retailers, greatly simplifying inventory management. A typical warehouse carries around 3,500 SKUs [23]; by comparison, a typical Wal-Mart Supercenter has 142,000 [12].

An important part of the company’s strategy is ensuring that its members trust the company to keep prices low. To that end, the company caps markup on all products at 15%, so customers know they’re never getting a bad deal [11].

Also remarkable are the company’s labor practices. The average hourly employee is paid \$20.89 an hour, in comparison with an average of \$12.67 at Wal-Mart [82]. (Executive salaries,



**Figure 3:** Interior of a Costco warehouse, Kawasaki, Kanagawa, Japan. *Photo credit: “Costco Japan” by Danny Choo. Reproduced from <http://www.flickr.com/photos/dannychoo/5455109003/> under the Creative Commons Attribution-ShareAlike 2.0 license: <http://creativecommons.org/licenses/by-sa/2.0/>.*

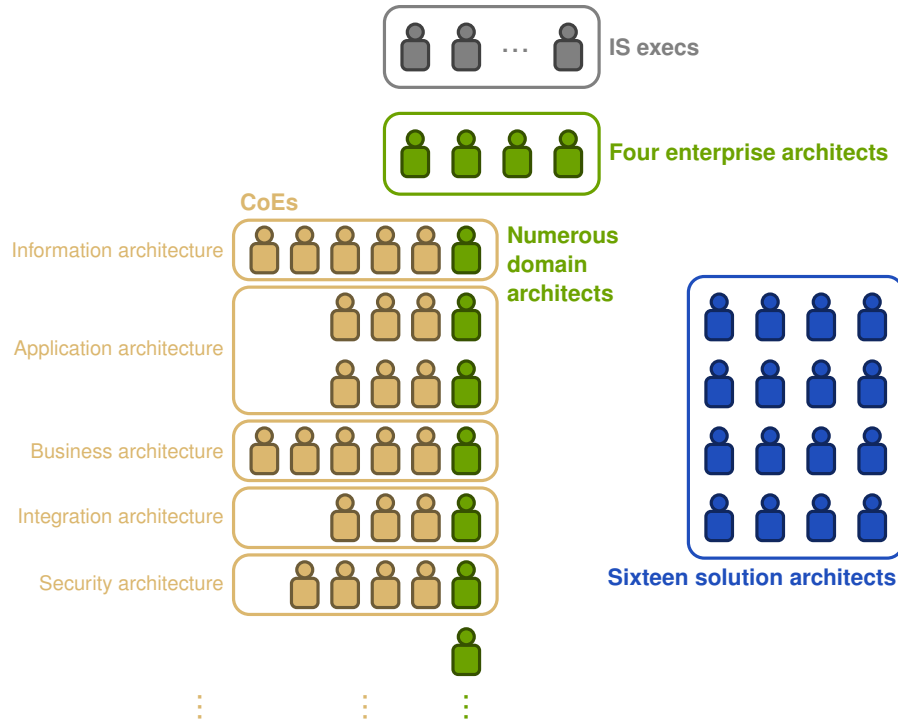
on the other hand, are low in comparison with other Fortune 500 companies.) In addition, workers enjoy much better job security than at other major retailers; even in the depths of the 2007–2009 recession, the company didn’t lay off employees.

Today Costco is a global retailer. It has about 627 locations worldwide: 449 in the United States, 85 in Canada, 33 in Mexico, 24 in the United Kingdom, 15 in Japan, 9 in South Korea, 9 in Taiwan, and 3 in Australia [24].

Over the years, Costco has accumulated a patchwork of legacy software systems. Until recently, the company had always built almost all of its software systems, even systems that are more typically purchased as off-the-shelf packages, such as the company’s core financial systems. A few years ago, the company embarked on an extensive and prolonged modernization effort to revamp many of its key software systems, which were growing archaic. A few of these systems were simply rebuilt in modern languages with modern engineering techniques, but for the most part, the company has transitioned from its tradition of homegrown software to a posture of buying off-the-shelf software whenever possible. Thus, many of the company’s old, custom-built systems are being replaced with off-the-shelf products. As a result of this far-reaching modernization effort, architectural-level evolution of software systems is now pervasive there. The contemporaneous overhaul of so many core systems has posed significant integration and communication challenges. These factors made the company an appealing candidate for a case study on architecture evolution.

## *2.2. The architecture organization*

Understanding the organizational structure of the architecture group that was the subject of this case study will be helpful for contextualizing the rest of this report. The architecture group underwent a significant reorganization several years ago, at the beginning of the modernization effort described in section 2.1. Until recently, architects were scattered throughout the IT department; people in architectural roles were associated with particular teams or projects, and there was no central architecture organization, nor even regular communication among the architects. This made it difficult to understand the dependencies among systems and to



**Figure 4:** The organizational structure of Costco’s architects.

diagnose integration problems, so a central enterprise architecture group was established to define and maintain enterprise standards and to manage and harmonize architectural practices across the organization.

In figure 4, I have diagrammed my understanding of the current state of the architecture organization, based on the interviews I conducted. At the top of the architecture organization are four enterprise architects (EAs). Each of these four individuals has a broad set of responsibilities, but they all have their own focuses. One of them, for example, is primarily responsible for infrastructure; another has oversight over process issues. The EAs are together responsible for defining strategy and making policies that affect the organization as a whole. They are tasked with ensuring that the company’s software systems are architecturally consistent—to guard against the kind of architectural incongruities, integration troubles, and communication problems that beleaguered the company years ago, before a central architecture organization was created. One of the ways they do this is by leading the Enterprise Architecture Review Board, which must approve major architectural decisions to ensure that they are in conformance with corporate strategy and practices.

Under the EAs are a number of domain architects. These domain architects also do enterprise architecture, in the sense that they also set standards and provide guidance that is applicable to the organization as a whole. Each domain architect has responsibility over a single domain of expertise; thus, there are information architects, infrastructure architects, business architects, application architects, security architects, integration architects, and so on.

When this new enterprise architecture group was formed, there was concern that it would become an ivory tower, estranged from the experiences of the people actually building the software, dispensing impractical guidance from on high. To avoid this, centers of excellence (CoEs) were established as a way of keeping the work of the domain architects grounded

in reality. Each domain architect, generally speaking, leads a CoE that has some decision-making responsibility within that architect's domain. Thus, the business architect leads a business architecture CoE, an application architect leads an application architecture CoE, and so on. The CoEs are led by the domain architects, but they are staffed by engineers and other personnel who have hands-on roles in building, maintaining, or managing systems. These people have practical, day-to-day domain expertise that complements and grounds the domain architect's broader, theoretical domain expertise. For example, the business architecture CoE, which is led by the business architect, is staffed by a business process analyst, a data steward, a business analyst, a service engineer, and a couple of product analysts.

Together, the four EAs and the domain architects form the enterprise architecture side of the architecture organization, support by these CoEs. The other side of the architecture organization is the solution architects, who are responsible for specific projects. A solution architect is usually involved with a few projects at a time. It is the solution architect who actually creates the architectural designs for individual projects, in accordance with the practices approved by the enterprise architecture group, and with guidance from domain architects with relevant expertise. The solution architect gathers project requirements, defines the high-level design for a project (which will be validated by the Enterprise Architecture Review Board), hands it off to a lead developer for lower-level design, and oversees architectural matters as the solution is implemented and released. There are sixteen solution architects in total.

Together these three groups—enterprise architects, domain architects, and solution architects—constitute the architecture group. The EAs provide leadership and define strategies and policies that affect the entire organization, the domain architects work with their CoEs to define practices and provide guidance within their respective domains, and the solution architects work in support of individual projects.

### 3. Data collection

Data collection was carried out during a two-week visit to Costco's headquarters in Issaquah, Washington. Two main types of data were collected: interview data and written architectural documentation. The following sections describe the collection of these data.

Section 3.1 describes the interview procedure that guided the collection of interview data. Section 3.2 describes the interview data that was thus collected. Section 3.3 describes the architectural documentation.

#### 3.1. Interview procedure

This case study adhered to a *semistructured* interviewing discipline, which means that although the interviews were guided by an explicit interview protocol that defined the general topics that the interviews would examine, I was free to devise new questions on the fly to explore interviewees' knowledge at will. This is in contrast to a *structured* interview, in which every question is scripted and the interviewer is strongly constrained, or an *unstructured* interview, in which the interviewer is completely unconstrained and the interview has no set format.

The choice of semistructured interviewing was the most appropriate for this type of research, as I had some familiarity with the domain and a strong sense of the kind of information I was seeking, but I needed the freedom to explore participants' responses and investigate closely those topics on which a particular participant might have especially deep knowledge. Unstructured interviews are most useful for exploring a topic about which very little is known, while structured interviews are most useful when the focus of the research is on directly

comparing responses across participants (as is generally the case in quantitative interview research).

Because this case study involved interviews with human subjects, it was subject to institutional review board approval. Before beginning data collection, I secured approval from Carnegie Mellon University's IRB. The IRB application included the aforementioned interview protocol, details on the purposes and procedures of the case study as a whole, and explanations of the measures that would be taken to protect the privacy of the human participants in the research and the confidentiality of the data.

The interview protocol is reproduced in appendix A. It includes an introductory script to secure informed consent followed by a series of topics to be covered: the participant's background, the practice of architecture evolution, limitations of current approaches to managing evolution, and specifics about the evolution of particular software systems.

Interviews varied in length from participant to participant, depending on participants' availability and their relevance to the research questions. As I wrote in the IRB application:

For example, if I'm interviewing an experienced software architect with expansive knowledge of Costco's software systems, I might want to interview her or him for a full hour. But if I'm interviewing a junior tester who happens to be very busy and can only spare a short amount of time, I might conduct a 15-minute interview instead. We estimate that all interviews will be between 15 and 90 minutes long.

I captured audio recordings of all interviews. Recording interviews has a number of advantages. First, it provides the researcher with an accurate and complete record of each interview, so that the researcher is not forced to rely solely on his own notes and memories, which are inevitably inaccurate and incomplete. Second, it frees the interviewer from taking excessive, meticulously complete notes during the course of the interview, which can seriously encumber the flow of dialogue. Third, it bolsters the validity of the research; when a researcher has only his memories and notes to go on, it is easier for him to impose his own biases on interview responses. Fourth, it enables transcription and content analysis of interview responses, which is of particular relevance to this research (see section 4.1).

Of course, recording interviews also raises ethical and technical issues. Ethically, capturing recordings of interviews requires explicit consent. I therefore included a clear explanation of the nature and purpose of the recording in the introductory consent script that I read at the beginning of each interview, and I offered each interviewee the option to opt out of being recorded. (In fact, all of the participants in the study consented to be recorded.) Another ethical issue is data security. The IRB considers audio recordings of interviews to be sensitive research data and generally requires that they be stored securely. To that end, I ensured that all copies of audio recordings were secured either physically or electronically during the research. I was the only person who had access to the recordings. Upon completion of this research, all audio recordings will be destroyed.

Interviewers who rely on audio recorders must anticipate the possibility of technical difficulties. I used two recording devices to capture each interview to reduce the possibility of failure. This also proved helpful later for transcription, since segments of speech that were inaudible on one recording were sometimes clearer on the other. Section 4.1 will discuss transcription further.

### *3.2. Interview data*

I interviewed six participants for this case study in eight interview sessions. Some participants were interviewed more than once, and also in some interview sessions multiple participants were present.

The median interview duration was 44.4 minutes. The mean duration was 40.5 minutes. Because each interview was recorded on two devices, this amounted to a total of 10.8 hours

of data (40.5 minutes/recording  $\times$  2 recordings/participant  $\times$  8 participants = 10.8 hours). As mentioned above, the two recordings were not identical; some segments of speech that were unclear on one recording were clearer on the other. Thus, I had to use all 10.8 hours of recorded data in transcribing the interviews.

### 3.3. *Architectural documentation*

In addition to interviewing personnel, I was also permitted to access many architectural documents pertaining to the company's software systems. These were a very useful source of data and particularly useful as a complement to the interview data that I collected.

Particularly significant examples of documents to which I had access include:

- Thirty-five "architectural decision" documents, each of which captured an architectural decision that had been approved by the enterprise architecture team in accordance with a defined process. Each of these documents described the background and justification for the decision, as well as alternatives considered. The procedure for publishing these architectural decisions had been implemented in 2010, so all of these documents were from the period 2010–2013.
- Twenty-one "strategy" documents describing, in more general terms, strategic initiatives from an architectural standpoint. These were all from 2011–2013.
- Two detailed architectural design documents describing a specific evolution, the evolution of Costco's point-of-sale system, which would become the focus of the modeling phase of the case study (section 4.3).

These documents were useful because they provided clear, explicit articulation of the architectural reasoning and decision-making process. However, because the documents that were available to me were all relatively recent, they did not provide detailed historical information. Thus, these documents were most useful as a complement to the interview data, which provided a better historical perspective.

## 4. **Analysis**

This section describes the analytical procedures of the case study in detail. Section 4.1 explains how the interviews were transcribed. Section 4.2 explains the content analysis, which constituted the main analytical phase of the case study. Finally, section 4.3 describes the construction of an evolution model to evaluate the applicability of our modeling approach to a real-world evolution.

### 4.1. *Transcription*

I fully transcribed all eight of the interviews I conducted. Transcription of research interviews has a number of advantages. First, it allows much easier reference to the interview contents, allowing the researcher to peruse the collected interview data freely without the need to listen to long stretches of audio to find the desired information. Indeed, unlike an audio recording, a transcript is fully searchable, allowing the researcher to instantly find significant passages. In addition, and even more importantly, transcription enables various kinds of analysis, including content analysis (section 4.2).

In the social sciences literature, there is a great deal of discussion on methods and best practices for transcription of research interviews. I reviewed a number of such sources before beginning the transcription process. One of the most basic decisions that a researcher must make when transcribing interviews is whether to try to capture all the nuances and quirks of verbal communication, such as pauses, pronunciation idiosyncrasies, stutters, false starts, and nonverbal utterances like "um" and "ah," or whether to instead adapt the material to the



written form, eliding stutters and verbal tics, standardizing grammar, and adding punctuation. In the literature, these are sometimes called *denaturalized* and *naturalized* approaches to transcription.<sup>1</sup> Of course, these are really two ends of a spectrum, and often it is most appropriate to take a middle course. In addition, it is also possible for a transcription system to be naturalized in some respects but denaturalized in others, for example by including stutters and nonverbal utterances but also inserting punctuation. But in the extreme case, highly naturalized transcriptions can read much like written, edited prose, while highly denaturalized transcriptions often adopt abstruse notation systems to capture the nuances of spoken language, resulting in a transcript that bears little resemblance to ordinary natural language. Figure 5 presents an example of a passage from one of the interviews that I transcribed twice, once in a denaturalized style and once in a naturalized style, to illustrate the difference between these approaches.

This choice is not an inconsequential one. A denaturalized transcription system includes more information, but it does so at the risk of obscuring the content. A naturalized transcription system reads much more easily, but it does so at the cost of losing information that might inform interpretation of the text. These issues have long been a topic of debate, but a good general principle is that such choices should be driven by the purpose of the analysis [56; 65; 67; 70; 75]. In this case, we are interested in the interviews solely for their information content; we are interested in what is said, not how it is said. As a result, I adopted a highly naturalized form of transcription; in figure 5, the right column is an excerpt from my actual transcript.

The transcription process itself was methodical and painstaking. As mentioned in section 3.2, two recordings were made of each interview. During the transcription process, I listened to each recording at least twice to ensure transcription accuracy and minimize the number of passages that had to be marked inaudible. (Because the two recorders were differently positioned, usually the interviewer was more clearly audible in one and the participant was clearer in the other.) After all eight interviews were transcribed, I went through a final editing phase to ensure consistent orthography across interviews. In total, transcribing all eight interviews took approximately fifty hours of work.

## 4.2. Content analysis

As mentioned previously, the main analytical method used in this case study to address the research questions posed in section 1.3 is *content analysis*. Content analysis is a research method, commonly used in the social sciences, for extracting meaning from text. A fairly standard definition of *content analysis* is that given by Krippendorff [48, p. 24]: “a research technique for making replicable and valid inferences from texts (or other meaningful matter) to the contexts of their use.”

Content analysis has been seldom used in software engineering research and almost never used in software architecture research. Because of this, many readers may be unfamiliar with it, so I present here an overview of the method and a fairly detailed discussion of relevant methodological considerations in adopting it.

Section 4.2.1 presents a brief history of content analysis. Section 4.2.2 discusses the difference between qualitative and quantitative content analysis and explains why this case study uses the former. Section 4.2.3 describes the elements of a qualitative content analysis. Finally, sections 4.2.4–4.2.7 describe how content analysis was used in this case study.

---

<sup>1</sup>Unfortunately, these useful terms have been compromised by inconsistent usage in the literature. When Bucholtz [15] introduced the terms in 2000, she defined naturalized transcription as transcription that privileges written over oral discourse features, producing a “literacized” text. Denaturalized transcription, in Bucholtz’s original terminology, was transcription that prioritizes faithfulness to oral language through the use of verbatim transcription, including discourse markers, repetitions, repairs, and so on. However, in a 2005 paper, Oliver et al. [67] reversed these terms (perhaps unintentionally), using the term *naturalized* to describe what Bucholtz had called denaturalized transcription and vice versa. Usage of the terms in the literature is now inconsistent, with some authors [41; 65] using Bucholtz’s original meanings for the terms and others [60; 62] following Oliver et al. This report does the former.

**Researcher:** Pt okay. ·hhhhh Um: (0.2) is ther:e- (0.2) is there ↑anything ↓you:: ↑wish you 'ad known at ↓the: (0.2) ↑outset 'v this:: uh (0.2) project, er any sort of lessons you've ·hh learned ↓that you wish:: (0.2) uh you'd known (·) a couple a years ago?=-or a few years ago?=-

**Participant:** =·hhhhh Um hhheh- ↑heh (1.8) >I d'know, it's tough t' say=I< mean thih- (·) Prob'ly th' biggest things would be: tuh understa:nd, (0.5) at th' beginning:, just how disjointed th' retail world would be at ↑this point, (0.8) so we wouldn't hafta: 've tried t' say: (·) an' (·) understa:nd everything th't we would wan' a system t' do at th' be↑ginning? (0.8) An' jus' go back t' th' concept th't we're doing now: of let's jus' put in a founda:tional system that's open t' the re:st?=-rather 'n tryin' tuh (0.3) solve all these

(0.6)

**R:** =[Mmhhh.]

**P:** [ques:tions] at the ti:me?=-which (·) even now don't have an↑swers for 'em? ·hh Uh I think that was parta what happened in kinda those multiple itera:tions as we tried t' so:lve 'n' could never git that ·hh compelling business reason dow:n, so it's taken so lo::ng f'r us t' do this?

**Researcher:** Is there anything you wish you had known at the outset of this project, or any lessons you've learned that you wish you'd known a couple of years ago or a few years ago?

**Participant:** [*chuckles*] I don't know. It's tough to say. Probably the biggest things would be to understand, at the beginning, just how disjointed the retail world would be at this point, so we wouldn't have to have tried to say and understand everything that we would want a system to do at the beginning. Just go back to the concept that we're doing now: let's just put in a foundational system that's open to the rest, rather than trying to solve all these questions at the time, which even now don't have answers for them. I think that was part of what happened in those multiple iterations as we tried to solve and could never get that compelling business reason down, so it's taken so long for us to do this.

**Figure 5:** Denaturalized versus naturalized transcription. At left is a verbatim, highly denaturalized transcription of a 52-second segment of an interview. This notation, invented by Jefferson [43], is very widely used in fields such as conversation analysis where it is crucial to capture the subtleties of oral communication. Without a legend, much of the notation may be difficult to decipher, but note the fine details: subtle variations in word pronunciations (e.g., “to” versus “tuh” versus “t”); careful measurements of breaks in speech (the parenthesized decimals are pause durations in seconds); indicators of stress and tone; and even audible inhalations and exhalations. Such details would be useless in the present case study, however, and would only obscure the content, which is what we care about here. The naturalized transcription at right hides these details, standardizes spelling and to some extent grammar, elides false starts and discourse markers such as “um” and “sort of,” and adds correct punctuation. (In the Jefferson-style transcript on the left, the punctuation indicates intonation rather than demarking clauses.) This necessarily involves some interpretation and judgment, but it results in a transcript whose content is much more accessible.

### 4.2.1 A very brief history of content analysis

The history of content analysis can be traced back centuries; for an overview of this early history, see Krippendorff [48, ch. 1]. Content analysis in its modern form, however, came into being during the Second World War, when the renowned social scientist Harold Lasswell, acting as the chief of the United States government's Experimental Division for the Study of War-Time Communications, applied content analysis to Axis propaganda [73, ch. 6]. Lasswell standardized techniques that are still used in content analysis today, such as pilot testing, the use of formal sampling criteria, and assessments of reliability based on interrater agreement. In 1941, content analysis was the focus of a major conference on mass communication that was attended by many leading scholars [86]. In 1952, Bernard Berelson published the first major textbook on content analysis [10].

The ensuing decades saw ever-increasing adoption of content analysis by researchers [66, ch. 2]. The method was adapted to varied applications in many different disciplines, from readability analyses [32] to authorship attribution [64] to studies of television violence [22]. Today, content analysis is a major research method in the social sciences and a number of other disciplines, and many textbooks and articles give guidance on the method and its application to various disciplines.

### 4.2.2 Qualitative versus quantitative content analysis

Content analysis has spawned many variants as it has been applied to many different fields, but the oldest and most enduring division is between *quantitative* and *qualitative* content analysis. In his 1952 book on content analysis, Berelson defined content analysis as “a research technique for the objective, systematic, and quantitative description of the manifest content of communication” [10, p. 18]. But it did not take long at all for critics to find fault with this definition, particularly with the words *quantitative* and *manifest*; in the very same year, Kracauer published a critical response to Berelson, entitled “The Challenge of Qualitative Content Analysis” [45], that argued that meaning is often complex, context-dependent, and manifest, and that quantitative approaches are insufficient for analysis of such meaning.

This interchange defined the contours of a debate that continues to this day. Quantitative content analysis has retained its status as the dominant form of the method, and there has been an ongoing debate about the rigor, validity, and appropriateness of qualitative alternatives. Advocates of qualitative content analysis argue that a qualitative approach is useful and necessary for analyzing the content of texts where meaning is highly subjective and context-dependent and quantitative measurements are ill suited to describing meaning. Meanwhile, critics of qualitative content analysis have expressed skepticism about its reliability and its methodological rigor.

Indeed, to a significant degree, mainstream quantitative content analysts continue to be fairly dismissive of qualitative content analysis. Krippendorff's *Content Analysis: An Introduction to Its Methodology*, the leading text on the method, equates qualitative content analyses with what Krippendorff calls “text-driven content analyses” [48, § 14.1.1]—analyses that are motivated by the availability of texts rather than by epistemic research questions—which he dismisses as “fishing expeditions” [p. 355]. “Qualitative content analysts,” he writes, “typically stop at their own interpretations of texts” [p. 357]. However, he grants that “qualitative approaches to text interpretation are not incompatible with content analysis” [p. 89].

Neuendorf's *The Content Analysis Guidebook*, another major text on the method, is even more dismissive, devoting only one sentence to the topic of qualitative content analysis: “Although some authors maintain that a nonquantitative (i.e., ‘qualitative’) content analysis is feasible, that is not the view presented in this book” [66, p. 14].

It may be true that early examples of qualitative content analysis lacked some of the methodological rigors of the quantitative method. However, great strides have been made in developing qualitative content analysis into a rigorous research methodology with a disci-

plined process and with careful consideration of validity and reliability.

The methodologist most responsible for these strides is Philipp Mayring, who in 1983 published (in German) an influential textbook on qualitative content analysis that standardized the discipline and introduced methods for evaluating reliability and validity. The textbook is now in its eleventh edition [59]. However, it was not until 2000 that any of Mayring's writing on qualitative content analysis was translated into English, and even then only a short journal article [57]. (The textbook remains untranslated to date.) As a result, qualitative content analysis was, for a long time, much more widely adopted by German researchers than by Anglophones.

This is now changing. Adoption of qualitative content analysis in the English-speaking research community has increased in recent years and is likely to continue increasing. One of the first English-language books on qualitative content analysis was published last year by a German methodologist, Margrit Schreier [77].

Qualitative content analysis, in its modern, systematized form, has a number of qualities that make it more suitable than quantitative content analysis for our purposes here:

- Qualitative content analysis is intended for use in cases where *interpretation* is necessary to analyze the data—when the meaning of the material is not standardized, is context-dependent, or would likely be understood differently by different readers. A different way of putting this is that qualitative content analysis specifically seeks to examine the *latent* meaning of texts rather than only the *manifest* content. (Quantitative content analysis is often applied to latent content as well, but qualitative content analysis is designed specifically to facilitate reliable and explicit interpretation of texts.) Schreier [77, p. 2] explains this point with an example based on analysis of magazine advertisements. If a study seeks to find out information about the number of women and men who appear in magazine advertisements, then quantitative content analysis would be more suitable than qualitative content analysis, because very little interpretation is required to determine whether the persons in a picture are female or male. But if a study seeks to determine whether women in magazine advertisements are more likely to be placed in trivial contexts than men, qualitative content analysis would be a highly suitable method, because there is a significant interpretative step required to determine whether a given context is “trivial” (different readers might well disagree). Qualitative content analysis provides a method for managing this ambiguity in a valid and reliable way.

In the present study, we are explicitly concerned with interpretation of the language that architects use in their discourse on evolution. The meaning of language in architecture is not sufficiently standardized to justify a classical approach in which we simply apply our own unexamined interpretations to the data; indeed, even basic terms like *architecture* and *evolution* mean very different things to different people. The use of qualitative content analysis allows us to deal in an explicit and rigorous way with such differences and helps us to avoid imposing our own biases onto the interpretation of the data.

In addition, one of the particular goals of this study is to translate, in a replicable way, the language that real-world architects use when talking about evolution into the modeling constructs of our approach to architecture evolution. This is merely a special kind of interpretation, and qualitative content analysis is well suited to help us with it.

- In quantitative content analysis, the research questions are addressed via frequency counts (or quantitative measures derived from frequency counts through methods such as cross-tabulation or factor analysis); counting things (words, codes, categories, etc.) is almost invariably a core part of the analysis.

In qualitative content analysis, frequency counts and statistical methods *may* be used (and often are)—after all, the method necessarily involves unitizing the material and

coding it based on a finite set of predefined categories, so it's quite natural to tally the results once the material is coded. But in qualitative content analysis, it is equally acceptable to conduct a textual (nonfrequency) analysis, using the codes and categories as a way to achieve reliability and avoid bias, rather than as figures to be tallied.

The research questions that interest us in this study are not quantitative in character (although frequency data may be helpful in answering them—and indeed we will make use of frequency data in our analysis). In addition, a quantitative analysis approach is most appropriate when there are many subjects to be compared—when the goal is to compare or relate results collected from many individuals (individual people, individual companies, individual texts, etc.). Our content analysis, on the other hand, is conducted within the context of a case study—indeed, a single-case study. There is only one “subject” of the case study: Costco. (There were six human participants—the interviewees—but they are not the subjects of the case study. Rather, their role is as informants who can supply data about the true subject of the study, Costco.) Thus a qualitative analysis is much more appropriate here.

- Qualitative content analysis has a more flexible method than quantitative content analysis. In quantitative content analysis, the procedure is fairly fixed. There are many variants of the method, but each variant has a prescribed set of steps, with only certain kinds of deviations permitted.

Qualitative content analysis has historically been much looser. Critics of qualitative content analysis have regarded this as a flaw, while advocates have regarded it as a strength. Here, we are using a particularly well-defined form of qualitative content analysis that does have a process with a fixed set of steps. But even so, much more variation is permitted in the execution of those steps than in classical quantitative analysis. For example, the coding frame may be defined in a very data-driven way, or in a very concept-driven way; reliability may be assessed by a variety of different methods [77, p. 17].

For us, flexibility is useful because it makes it easier to adapt the approach to our own research context. There are a couple of particularly significant ways in which our use of content analysis differs from typical uses. One has already been mentioned: one component of our study involves translating the data into a formal model based on our previous research on software architecture evolution. This kind of interpretation is formally similar to what typically goes on in content analysis, but it is sufficiently different that it necessitates special consideration.

The second way in which our use of content analysis deviates from the norm is that it occurs within the context of a single-case study. Content analysis (especially quantitative content analysis, but even qualitative content analysis) is typically used to relate or compare multiple subjects of study, rather than to describe only one subject in detail. Of course, the use of content analysis in a case study is far from unprecedented, but nonetheless it requires some degree of deviation from the standard process.

In reality, the qualitative/quantitative division in content analysis is not a dichotomy, but a continuum [38]. Many qualitative content analyses have highly quantitative aspects (such as the use of statistical methods, as mentioned earlier), and many classically quantitative content analyses deal with latent meaning, involve rich textual interpretation, and adopt methodological variations. However, the above discussion makes clear that qualitative content analysis is much more suitable for our research questions than quantitative content analysis. The next section describes the specifics of the qualitative content analysis method.

### **4.2.3 Elements of a qualitative content analysis**

Schreier [77] describes a qualitative content analysis as comprising these main steps:

1. **Define the research questions.** In the first step, I defined the research questions listed in section 1.3. Long before data collection began, I had already defined the broad research questions that the study aimed to address. But the research questions in section 1.3 are more narrowly and specifically tailored to be questions that are answerable with the actual data collected.

2. **Build a coding frame.** Coding—an analysis technique in which researchers annotate segments of text with descriptive codes, categories, or labels—is probably the most well known and widely used of all qualitative analysis methods. There are many methods for coding, which vary greatly in their purpose, use, and approach [74].

In qualitative content analysis, coding is used as a way of reducing the data and focusing the analysis on the research questions. Coding is done in accordance with a coding frame, which defines the codes that may be applied to segments of data. Defining this coding frame is the first step in a qualitative content analysis (once the research questions have been determined and data have been collected).

There is a particular structure to which a coding frame for qualitative content analysis should adhere. A coding frame is built around a set of *categories*, which define the dimensions of interest; within each category is a set of mutually exclusive subcategories. (Subcategories may in turn have their own subcategories, resulting in a hierarchical structure.) The construction of the coding frame circumscribes the analysis. Content analysis is a reductive process; the coding frame determines which information will be excluded from the analysis and which is important enough to be included.

3. **Divide the material into coding units.** In content analysis, codes are not applied at will to arbitrary segments of text, as in some coding methods. Rather, the text is first divided into *coding units*, such that each unit may be coded with at most one subcategory of a main category in the coding frame.

This is done for a few reasons. First, it forces the researcher to analyze all the material, segment by segment, avoiding the risk that the researcher will inadvertently ignore portions of the text or devote too much attention to the passages that are most striking (or that fit into a preconceived narrative). Second, it helps to keep the analysis directed toward answering the research questions. Third, it facilitates double-coding as a means of demonstrating reliability.

4. **Try out the coding frame.** Before the main coding phase begins, a pilot phase is recommended in which the researcher applies the coding frame to a portion of the material. This can reveal problems that would otherwise crop up during the main analysis.

5. **Evaluate the trial coding and finalize the coding frame.** After the pilot phase, the coding frame is assessed for its reliability and validity and modified as necessary in preparation for the main analysis. If necessary, a second pilot phase may be conducted at this point.

6. **Carry out the main analysis.** Coding in the main analysis is conducted much as in the pilot phase, except all material is now coded.

7. **Interpret the findings.** There are various techniques that can be applied to further analyze the results of a content analysis, and various ways that the findings of a content analysis can be presented. The ultimate goal is to answer the research questions in a way that is valid and reliable.

The following sections describe how these steps were carried out in this case study.

#### 4.2.4 Coding frame

The coding frame that I developed is reproduced in its entirety in appendix B. In this section, I explain how it was constructed.

The construction of the coding frame is one of the most crucial steps in a content analysis. After all, the coding frame defines the rules that govern the interpretation of the text. Thus, the construction of the coding frame defines the shape of the rest of the content analysis, including the segmentation of the text into units, the coding itself, and the tabulation and reporting of the findings.

As mentioned in section 4.2.3, there is a particular structure to which a coding frame for a qualitative content analysis should adhere [77]. The starting point for a coding frame is a set of *main categories*, which define the dimensions or aspects on which the analysis will focus. Within each main category is a set of *subcategories* that specify what may be said about the aspect represented by the main category. For example, in a study investigating attitudes about the Iraq War, there might be a main category called “Attitudes about the morality of going to war in Iraq” with subcategories such as “Morally necessary,” “Morally justifiable,” “Morally unclear,” and “Morally wrong.” In effect, the main categories are analogous to the variables in a quantitative study, and the subcategories are analogous to the levels of those variables. Coding frames vary greatly in complexity; there may be one or many main categories, and there may also be multiple levels of hierarchy, with subcategories containing their own further subcategories. (Note that in content analysis, *category* is effectively synonymous with *code*, even though these terms are importantly different in, e.g., grounded theory.)

For the present work, it is useful to observe that the research questions in section 1.3 are of two very different characters. On the one hand, we have a number of questions about how evolution happens in the real world today—what methods architects use to plan and carry out evolution, what challenges they face, what problems they encounter, and so on. These research questions are *descriptive* of architecture as it is practiced today. They will be answered directly through a content analysis of the interview data.

On the other hand, there is the final research question, which asks whether our approach to architecture evolution is suitable for representing the concerns of a real-world evolution. This research question is *evaluative* of our approach. This question will be answered via a two-step process in which we first apply content analysis to that portion of the research data which pertains to a specific evolution (the evolution of the point-of-sale system), then use the results of that content analysis in support of a modeling effort in which we construct a model of the evolution in accordance with our approach.

It is therefore useful to describe the analysis as actually comprising two entirely separate qualitative content analyses, one targeted at the descriptive research questions and one targeted at the evaluative research question. This is justified by the following distinctions between the two content analyses:

- The two content analyses examine different bodies of material. The analysis addressing the descriptive research questions will examine the entirety of the interview data (and nothing else). The analysis addressing the evaluative research question will examine the fraction of the interview data that pertains to the evolution of the point-of-sale system, along with collected architectural documentation pertaining to that evolution.
- The character of the material examined will be different. The analysis addressing the descriptive questions will deal exclusively with interview data, while the analysis addressing the evaluative question also must examine architectural documentation, some of which is in pictorial rather than textual format. This will complicate the segmentation and coding of the material.
- The coding unit for the two content analyses will be very different. In the analysis addressing the descriptive questions, the interview material will be segmented thematically, such that each coding unit is a passage of text (ranging in length from a phrase to several sentences) addressing one topic of interest. In the analysis addressing the evaluative question, the coding unit will often be a single word representing a particular architectural element. Section 4.2.5 will say more on this.

- The findings will be used in different ways. The findings with respect to the descriptive questions will be reported directly, while the analysis addressing the evaluative question is a precursor to a second phase, in which we model the evolution using the results from the content analysis.

For these reasons, I will henceforth describe the analysis as incorporating two separate qualitative content analyses, one addressing the descriptive research questions (“content analysis 1”) and one addressing the evaluative research question (“content analysis 2”).

The differences between these two content analyses necessitate that their coding frames likewise be constructed differently. It is helpful here to consider a distinction that is drawn in the methodological literature on qualitative content analysis. Both Mayring [57; 58] and Schreier [77, pp. 84–94], in their treatments of qualitative content analysis, identify two main approaches for developing a coding frame: *concept-driven* (or deductive) and *data-driven* (or inductive) category development. With a concept-driven strategy, categories are defined a priori, without reference to the data. Instead of being derived from the text, categories are based on preexisting theory, prior research, the format of the interviews, or other similar considerations. With a data-driven strategy, categories are based on the collected data—developed through progressive summarization of relevant passages of text or other similarly bottom-up strategies. Most qualitative content analyses, Schreier suggests, will use a combination of concept-driven and data-driven strategies. Schreier emphasizes that the strategy adopted for developing a coding frame should be one that is suited to the goals of the analysis. However, as a general recommendation, she writes that a concept-driven strategy is most appropriate for testing hypotheses or establishing comparisons with prior work, while a data-driven strategy is most appropriate for detailed description of material [77, pp. 105–106].

I adopted a principally data-driven strategy for content analysis 1 and a principally concept-driven strategy for content analysis 2. For the descriptive research questions, the purpose of the content analysis is to describe architects’ perceptions and experiences regarding architecture evolution. Because the purpose of this content analysis is to describe the material in detail, a primarily data-driven approach is most suitable.

Therefore, after defining the top-level categories based on the research questions (“Evolution motives,” “Challenges,” etc.), I defined their subcategories using a data-driven strategy. To do so, I made a pass through the entirety of the interview data, marking and paraphrasing any passages that appeared relevant to any of the top-level categories. For example, the following is an excerpt from an architect’s reflection on a previous effort to implement approaches that he and his colleagues had read about in well-known books on service-oriented architecture:

One of the problems that I’ve seen is that it is very cumbersome. Very good, very sophisticated, but when you are new to this, when you are really not intelligent enough to make those decisions, make those choices, when you have to go through a very elaborate process, it can be very counterproductive. And indeed that’s what we saw. When this method is introduced to the world, there’s a lot of confusion about: Why do we have to do all these things? I personally participated in some meetings too and saw this whole process as very cumbersome, very confusing, and frankly the result of that is not very good.

Next to this passage, I wrote, “challenge: available approaches cumbersome.” After marking all the interview data in this way, I consolidated topically similar marked passages into subcategories. For example, the above passage was combined with several others, marked with paraphrases such as “challenge: no resources for information” and “challenge: lack of needed tools,” into a category called “Challenges: Inadequate guidance and tool support.”

Note that a concept-driven strategy would have worked very poorly here. Instead of developing categories from the data inductively, I *could* have predefined a set of challenges that I hypothesized architects might face and used those as the subcategories of “Challenges.” But since my goal was to find out what challenges architects actually face—not to test hypotheses



about specific challenges that they are believed to face—this would have been inappropriate. In brainstorming hypothetical challenges, I would undoubtedly miss some *actual* challenges that were reflected in the data, thereby injudiciously disregarding a portion of the material.

For content analysis 2, on the other hand, concept-driven category development is perfectly appropriate. The goal of this content analysis is not to describe textual material in an open-ended way, but rather to evaluate the suitability of a particular, preexisting model with respect to a specific evolution. This preexisting model forms the basis for the coding frame.

Since its output will be used to produce an evolution model, content analysis 2 must serve to identify the elements that will appear in the evolution model. Thus, the coding units in content analysis 2 are descriptions of the *architectural elements* (components, connectors, etc.) and *evolution elements* (constraints, operators, etc.) that will appear in the model. Section 4.2.5 will explain this further. The first output that the content analysis must yield is the proper *identification* of these elements. That is, for each element described by a coding unit, we want to determine through the content analysis how that element should appear in the model—whether it should be characterized as a component, connector, constraint, operator, or some other type of element. Thus, the first part of our coding frame is a classification scheme for elements of the architecture evolution, with categories such as “Classification: Component,” “Classification: Constraint,” and so on.

The second output that we want to produce with respect to these identified elements is what phases of evolution they appear in. In particular, for each element, we would like to know whether it is present in the initial architecture, and whether it is present in the target architecture. The second part of the coding frame for content analysis 2, therefore, comprises the categories “Presence in initial architecture: Present,” “Presence in initial architecture: Absent,” “Presence in target architecture: Present,” and “Presence in target architecture: Absent.” Using this part of the coding frame, we can systematically determine whether each identified element should appear in the initial and target state of the evolution model. (We could extend this idea further, adding additional categories for the intermediate states of the evolution in addition to the initial and final states. However, for the purposes of this case study, I decided to keep the coding frame as simple as possible.)

The advantage of using content analysis to guide the construction of the model (rather than just constructing a model based on informal impressions and methods, as is more typical in software architecture research) is that the model will be directly and reliably tied to the research data, giving us more confidence that the conclusions we draw from the model are adequately supported by the data and are not influenced by researcher bias. Because the segmentation (i.e., the identification of model elements) and coding (i.e., the classification of model elements) are conducted according to well-defined procedures, we can be more certain that, for example, a model element that we identify as a connector really does represent an actual connector in the system as described in the research data, and that we haven’t inadvertently omitted or overlooked elements that ought to appear in the model.

The coding guide itself was written in a format typical of coding guides for content analysis. (For a discussion of the components of a good coding guide, see Schreier [77, pp. 94–104].) For each category, the coding guide defines: the name of the category, an abbreviation for the category (to be used when marking the coding sheet), a detailed description of the category (including an explanation of what it encompasses and guidelines for determining inclusion in the category), and usually an example of the category and sometimes a list of indicators to serve as decision aids. The full coding guide that I produced appears, as noted earlier, in appendix B.

#### 4.2.5 Segmentation

With the coding frame defined, the next step is to segment the material into *coding units* so that the categories of the coding frame can be applied to those coding units in the main analysis phase. Schreier [77, pp. 127–129] lists three reasons why it is important to segment

**Researcher:** When you say you generally have an idea of where you want to go and how you want to get there—you just can't necessarily execute on it as easily as you'd like to—how do you develop that plan? How do you figure out where you want to go and how you want to get there? Do you have processes to help you, or is it mostly just intuition and experience?

**Participant:** <sup>A9</sup>(Mostly intuition and experience, yeah.) <sup>A10</sup>(You look to the industry to see what's going on), <sup>A11</sup>(but ultimately, in this particular line of business—and I've worked in software development companies and retailers and laboratory environments, and I can tell you that in this business, you tend to lean toward simplicity. The next system you build is going to last twenty years, and it needs to be maintainable, and everybody needs to be able to capitalize on it and expand and extend when necessary, so you really don't try to get too crazy with it. We're not launching shuttles here, we're selling mayonnaise and toilet paper, so let's keep it in perspective.)

A9: Approaches: Experience and intuition

A10: Approaches: Industry practices

A11: Approaches: Rules of thumb and informal strategies

**Figure 6:** A segmented and coded passage from an interview transcript.

the text before coding it (rather than simply coding the text *ad libitum*, without explicitly demarcated coding units, as is done in many other forms of coding [74]). First, segmentation helps ensure that all relevant material is taken into account in the analysis by forcing the researcher to identify explicitly which portions of the material are relevant. Second, because the delineation of coding units is dependent on the definition of the coding frame, segmentation requires the researcher to have clear and explicit research goals. Third, segmentation allows the reliability of the coding frame to be assessed by having a portion of the material be coded by two different coders, or by the same coder at two different points in time, and comparing the results.

As with the construction of the coding frame, the segmentation of the material is handled separately for content analysis 1 and content analysis 2. For content analysis 1, the interview material was segmented *thematically*. That is, the material was divided into segments of sufficiently fine granularity that each coding unit pertained to one topic—where *topic* is interpreted with respect to the coding frame, so that one category is applicable to each coding unit. Figure 6 shows an interview segment as it was segmented and later coded.

Because of differences in how participants talked about the various topics under study, this segmentation process resulted in substantial variation in segment length. That is, some coding units were only a few words long, while others spanned several paragraphs of text. For example, participants sometimes expounded at considerable length on the approaches that they used to manage evolution, going into substantial detail on the specifics of these approaches. Their remarks about other topics tended to be briefer. As a result, segments pertaining to approaches were sometimes much lengthier than segments on other categories.

Of course, this effect is not purely an artifact of the participants' communication style; it is also a result of how the coding frame was defined. Had we wished to use the content analysis to explore elements of specific approaches, for example, we could have defined more specific categories for that purpose, such as "Approaches: Business architecture: Capability map" instead of just "Approaches: Business architecture." Elaborating the coding frame in this way would have resulted in more consistently compact coding units.

Segmentation was much more complex for content analysis 2 than for content analysis 1. The objective of content analysis 2 was to methodically and reliably identify the architectural elements and evolution elements described in the source material—that is, to identify and

distinguish among components, connectors, evolution operators, evolution constraints, and so on, so that they could be included in an evolution model based on the content analysis.

One challenge that this posed was how to deal with multiple mentions of the same element. To correctly identify and code an element, we must consider *all* its mentions throughout the source material (along with their contexts). This stands in contrast to the piecemeal interpretative process that is more typical of content analysis, in which we proceed through the relevant passages of a text in sequence, coding each passage in isolation.

Nonetheless, the need to consider multiple mentions of a referent together in order to determine a code is not incompatible with content analysis; it merely requires more careful selection of the coding units. In particular, we can consider multiple mentions of a single referent (e.g., all references to the corporate integration architecture) to together constitute a single coding unit—one that happens not to be contiguous. Krippendorff [48, § 5.2.2], in his well-known text on content analysis, discusses situations where noncontiguous coding units may be helpful:

The text included in any one recording unit need not be contiguous. Suppose an analyst samples fictional narratives with the aim of studying the populations of characters occurring in them. [...] In a typical narrative [...], the characters are rarely dealt with one at a time, or one per paragraph, for example. They tend to interact and evolve over the course of the narrative, and information about them emerges in bits and pieces, often becoming clear only toward the end. To be fair to the nature of narratives, the analyst cannot possibly identify one unit of text with each character. Thus information about a recording unit may be distributed throughout a text. Once the analyst has described the recording units, it is these descriptions, the categories to which they are assigned, that are later compared, analyzed, summarized, and used as the basis for intended inferences.

(*Recording unit* is another term for *coding unit*.) An architectural element or evolution element is similar, in this respect, to a character in a narrative. It may be mentioned many times throughout the material, in a variety of contexts that provide different information about it, all of which must be understood together to obtain a complete view.

Another segmentation challenge posed by content analysis 2 is that not all of the “text” is strictly textual; in addition to the interview material and the written documentation, there are a number of diagrams that must be considered in the analysis. Fortunately, this is not a real problem as far as content analysis is concerned. Content analysis has long been used for analyzing more than just prose. Indeed, Berelson’s groundbreaking 1952 textbook [10] opens by defining the object of content analysis as “that body of meanings through symbols (verbal, musical, pictorial, plastic, gestural) which makes up [a] communication” [p. 13] and includes a section on content analysis of “non-verbal communications” [pp. 108–113]. Content analysis has been applied to everything from films [25] to photos of models in magazines [28] to children’s drawings and cereal boxes [61]. (For general guidance on content analysis of visual data, see Ball & Smith [1, ch. 2] and Bell [8].) Content analysis of architectural diagrams, then, is not such a stretch. At a basic level, we can segment and code diagrammatic elements—boxes, lines, clouds, whatever—in much the same way that we can segment and code phrases and paragraphs.

So there’s no theoretical barrier to including architectural diagrams in our content analysis. But even so, the heterogeneity of the data being analyzed does create some practical challenges. Perhaps most significantly, a coding unit in this analysis is something much more complex and multifaceted than a coding unit in a typical content analysis. While in a typical content analysis a coding unit is simply a phrase or a passage, here a coding unit is an aggregation of words, phrases, and passages (occurring in both transcribed speech and written architectural documentation) as well as diagrammatic elements. This makes segmentation much more complicated, and it also makes coding itself a much knottier undertaking,

since properly categorizing a single coding unit now requires consideration of textual and diagrammatic elements that are spread throughout the material, along with their context.

#### 4.2.6 Pilot phase

As noted in section 4.2.3, a pilot phase is an important part of a qualitative content analysis, allowing any problems with the coding frame or the segmentation of the text to be ironed out before the main analysis phase. Thus, after completing a draft of the coding frame and finishing the definition of the coding units, I applied the preliminary coding frame to the segmented text, noting any points of difficulty or confusion. I then modified the coding frame as well as the definitions of the coding units as appropriate. Ultimately, all of the changes I made at this stage were relatively minor: adjusting boundaries between coding units, clarifying descriptions of categories in the coding frame, and so on. The coding frame that appears in appendix B is the final version, which I used for the main analysis.

#### 4.2.7 Main analysis phase

In the main analysis phase, I categorized each coding unit in accordance with the coding guide I had developed. I carried out the coding for content analysis 1 and content analysis 2 on separate occasions—again, treating them as two distinct content analyses even though they are part of a single case study.

One coding consideration worth noting is how much surrounding context should be considered while assigning categories. Many authors on content analysis recommend explicit definition of a *context unit* (in addition to the definition of a *coding unit* and a *unit of analysis*) that defines the scope of the context to be considered when carrying out the coding [48, § 5.2.3; 77, pp. 133–134]. Based on the results of the pilot phase, I had determined that in content analysis 1, coding units could be categorized accurately based on consideration of a relatively narrow surrounding context, consisting of at most a paragraph or so before and after the context unit itself. Indeed, many context units were effectively self-contained; their meaning was clear even without consideration of surrounding context. In content analysis 2, however, much broader context was often required to accurately determine proper categorization of the coding units. This is not too surprising, since the coding units in content analysis 2 consisted of a set of isolated words, phrases, and diagrammatic elements spread throughout the data. In practical terms, this meant that proper categorization of the coding units in content analysis 2 sometimes required consideration of broad contexts that amounted to almost the entirety of the source material, and occasionally even some documents external to the data being analyzed, as described in appendix B.2.1. In all cases, however, the categorization of the coding units was guided solely by the research data and the coding frame; I did not permit myself to refer to any notes that I had previously taken or to the categories I had assigned in the pilot phase.

After completing an initial phase of coding—that is, properly categorizing each coding unit in accordance with the coding guide—I set the data aside for a period of time, then started over and coded the data a second time. The purpose of this second phase of coding is to assess the reliability (more specifically, the stability) of the coding procedure. I will discuss this further in section 6.2.1. Schreier [77, p. 146] recommends that at least 10–14 days should elapse between successive codings by the same researcher, to ensure that during the recoding, the researcher does not remember the categories he selected in the initial coding. I allowed 17 days to elapse between the initial coding and recoding in the case of content analysis 2, and over a month in the case of content analysis 1. (Subjectively, I found that this interval was indeed long enough that I did not remember individual category selections from the initial coding.)

Once two rounds of coding had been completed, I reconciled the results. For each discrepancy that existed between the initial coding and the recoding, I carefully examined the

coding unit, the surrounding context, and the full descriptions of both categories (the one assigned in the initial coding and the one assigned in the recoding) in the coding guide, then chose whichever of the two was most appropriate. This final categorization served the basis for the subsequent interpretation of the results as well as (in the case of content analysis 2) the modeling phase.

### 4.3. Evolution model construction

The final analytical phase involved the construction of a (partial) evolution model based on the results of content analysis 2. This construction proceeded in stages. First, I constructed initial and target architectures, directly using the results of content analysis 2. Second, I specified a number of evolution operators relevant to the evolution, including several that had been specifically mentioned in the data. Third, I specified the constraints that had been identified in content analysis 2. Finally, I specified evaluation functions for the evolution concerns that had been identified by the content analysis. I will now describe each of these steps in detail.

#### 4.3.1 Initial and target architectures

Our approach to architecture evolution is based on modeling evolution states, but it is not tied to a particular modeling language or tool. Almost any modeling language can be used with our approach in principle, provided that it permits representation of architectural structure and is amenable to the sort of analysis that our approach entails. In previous work, we have demonstrated how our approach can be applied with a diverse array of modeling languages: Acme [36], UML [5], SysML [3], and PDDL [6]. Thus, the first step in constructing an evolution model in this case study was selecting a modeling language.

I selected Acme as the modeling language for the case study. Acme is an architecture description language developed at Carnegie Mellon University for representation of component-and-connector architectures [35]. Acme was appealing for this purpose because it is designed specifically for systematic, semantically rich representation of software architectures, and it has an extensive set of features for that purpose, such as support for decomposition of architectural elements and definition of rich hierarchies of element types. By contrast, other modeling languages we have used in past work, such as UML, aspire to a much broader set of purposes. Consequently, using such languages with our approach requires great care to be taken in establishing explicit conventions for their use, as we have seen in our prior work [3; 5; 6].

In addition, Acme is associated with a mature, full-featured modeling tool called AcmeStudio [76]. Like Acme, AcmeStudio is designed specifically for representation of component-and-connector architectures, and its feature set is designed to make that particular task easy. By contrast, UML tools tend to have a surfeit of unneeded features that get in the way of expeditious component-and-connector modeling.

The content analysis had identified all the significant architectural elements to be modeled and had further categorized them as components, connectors, systems, and so on. In addition, it had identified which of these elements appeared in the initial architecture, and which appeared in the target architecture. With this work done, constructing initial and target architectures in Acme was fairly straightforward. Specifically, the classification categories in the coding frame were realized in Acme as follows:


- *Components* were realized as Acme components.
- *Connectors* were realized as Acme connectors.
- *Ports and roles* (attachment relations) were realized as Acme ports and roles.
- *Systems* were realized as Acme components. A component type called SystemT was defined to distinguish systems from other components. In the architectural diagrams

that appear in this section, systems appear as gray, rounded rectangles.

- *Groupings* were rendered as dashed rectangles using AcmeStudio annotations. In AcmeStudio, annotations change the visual appearance of an architectural diagram, but they do not become part of the Acme structural representation and they cannot be analyzed or constrained. They are often used to group related elements (e.g., to identify layers or tiers) and add supplemental information.
- *Containment relations* were achieved through Acme representations. Representations are a feature in Acme for representing decomposition. A representation of an element is a definition of its internal structure.

Of course, the content analysis was not sufficient, on its own, to specify the initial and target architectures fully. It defined and classified all the key elements. But because the interviews and architectural documentation were fairly high-level, they seldom descended to the level of ports and roles, for example. Although the coding guide included a category for ports and roles, most of the ports and roles in the system were never explicitly mentioned. (Acme requires ports and roles to be specified fully. Thus, while in casual conversation one might simply speak of a component *A* being connected to component *B* by a connector *C*, Acme requires the roles of connector *C* to be explicitly defined, and attached to ports on components *A* and *B* that are also explicitly defined.) Similarly, we did not attempt to identify element types in the content analysis; thus, these had to be introduced in the modeling phase.

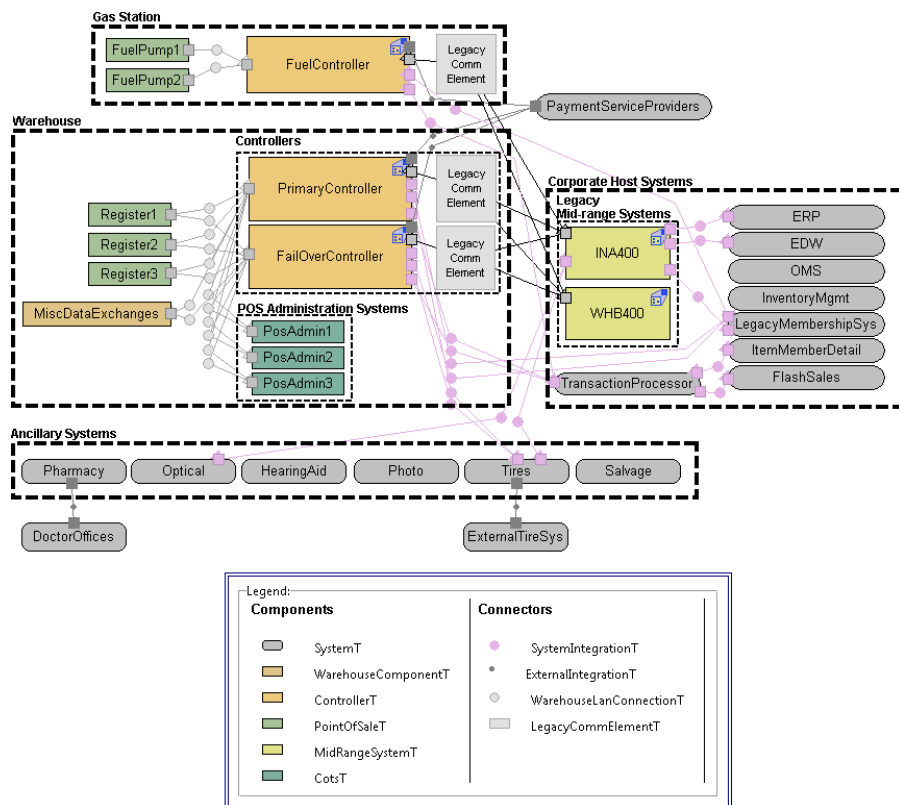
However, these low-level decisions were all fairly straightforward and inconsequential. All the major decisions—what the major components of the system were, how they should be connected, how the system as a whole was structured—had already been made via the content analysis. At least subjectively, then, we can say that the content analysis succeeded in systematizing and formalizing the major decisions involved in architecture representation, even though it did not itself amount to a comprehensive representation of the architecture.

The initial and target architecture of the system are shown in figures 7 and 8. Not shown in these figures are system substructures modeled using Acme representations. Elements with representations defined are indicated by AcmeStudio's  symbol. Expounding the architectural details of the point-of-sale evolution is beyond the scope of this case study report. Nonetheless, it will be useful to have a passing acquaintance with the system, so I now provide a brief explanation.

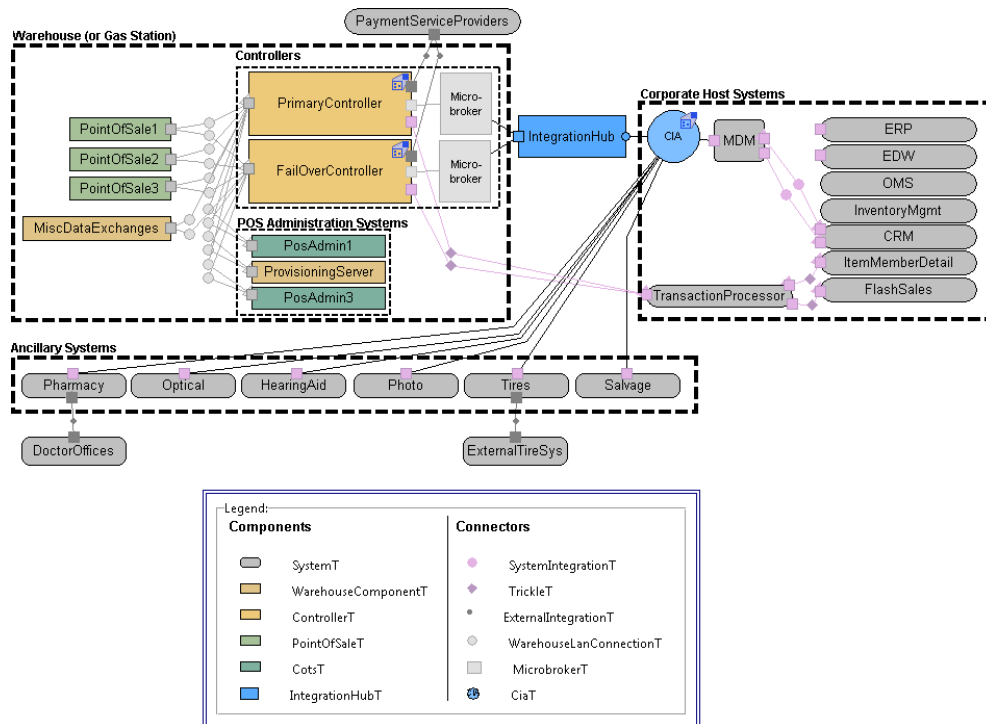
Figures 7 and 8 are each divided into several major groups of elements, represented by dashed rectangles, which correspond to the groupings identified in the content analysis. The core point-of-sale system is within the warehouse. The POS element itself does not appear in these diagrams, because it is inside the controller components. It would appear in a decomposition view of the controller components, if such a view were included in this report. The main goal of the point-of-sale evolution is to replace the legacy component at the core of the point-of-sale system, an off-the-shelf point-of-sale package, with another, more modern off-the-shelf package.

At the same time, however, a number of other things are changing in systems with which the point-of-sale system integrates, and these changes will affect the operation of the point-of-sale system significantly. For example, a number of new elements are being introduced to facilitate communication among systems, including a new integration architecture element, an integration hub, and a Master Data Management system. These elements will serve to decouple various systems, to unify various communication paradigms in use among systems, and to reduce redundancy. Another significant change is that various legacy systems will eventually be disused (e.g., the mid-range systems that appear in figure 7) or replaced (e.g., the legacy membership system will be replaced with a new CRM system).

Although representing the initial and target architectures based on the results of the content analysis was generally straightforward, there were a few difficulties that did arise. These difficulties may suggest areas for future research or methodological refinement, so I



**Figure 7:** An AcmeStudio representation of the initial architecture of the point-of-sale system, constructed from the content analysis results.



**Figure 8:** An AcmeStudio representation of the target architecture of the point-of-sale system, constructed from the content analysis results.

discuss them now:

- The exclusive use of a single (component-and-connector) architectural view type was somewhat limiting. The use of multiple view types might have helpfully enriched the model. In particular, supplementing the component-and-connector view with a deployment view might have helped clarify certain aspects of architectural structure. Figures 7 and 8 represent physical boundaries (e.g., the boundary between the warehouse and the corporate host systems) primarily using groupings. However, using a deployment view would have permitted us to depict such boundaries more explicitly and precisely. In addition, it would have allowed us to depict which software elements were allocated to which physical devices. This would have eliminated some redundancy. For example, all three of the controllers in figure 7 have the same internal substructure; with a deployment view, this could have been shown with less duplication by depicting individual software elements as being allocated to multiple controllers. In previous work [5], we have demonstrated that our approach can be used with multiple view types. However, here we have used only one view type so as to simplify the content analysis and permit the use of Acme, which does not support multiple architectural views.
- There were several elements about which very little information was provided. For example, the order management system, which appears as “OMS” in figures 7 and 8, was mentioned only a couple of times, and then only in passing. I did not receive any information about which other elements it was connected to or about its exact relationship with the point-of-sale system. In fact, because so little information was provided, it might have been best to omit it from the model, except that I wanted to ensure the model was derived from the content analysis with as few tweaks as possible. There were other elements about which significant, but still incomplete, information



was provided. For example, the TransactionProcessor element (which is responsible for processing data from point-of-sale transactions) was discussed in somewhat more detail than the order management system, but still not quite enough detail to be certain of its correct placement. For example, it might actually be hosted on one of the mid-range computers. The general problem here is that the data does not provide an exhaustive description each element. In interviews, architects speak only of the elements that are particularly salient; and even architectural documentation focuses on certain elements at the expense of others. Neither provides a comprehensive topological picture. With more time, I could have remedied this through follow-ups with the study participants. But I also think that the difficulty of getting an accurate and comprehensive picture of an architecture suggests an opportunity for future research into better methods of architecture elicitation.

- I had some difficulty deciding how to represent multiplicity. When it was convenient to do so, I used multiple element instances; for example, figure 7 shows three registers and a couple of fuel pumps. (The actual number of registers per warehouse and fuel pumps per gas station is quite a bit higher.) At other times, showing multiplicity was impractical. How can we adequately capture the point that there are hundreds of individual warehouses across the country? We could simply add an annotation—“Warehouse (627 instances)” —but this is a somewhat crude method. In an intermediate state, there is no good way to show a transitional point in which some warehouses are using a legacy system and some others are using its modernized successor.
- The target architecture was necessarily somewhat speculative, because there were certain aspects of its structure that architects weren't yet sure about. For example, a long-term goal is to increase integration between the point-of-sale system and the company's ancillary systems, but it's not yet certain precisely when or how this will happen. In the model in figure 8, I depict these ancillary systems as being fully integrated through the integration architecture, but there are other possibilities that might be just as likely. To some extent this kind of uncertainty is unavoidable, but it also suggests the possibility of capturing the uncertainty in the model itself and reasoning about the trade-offs among different possibilities.
- A subtler difficulty had to do with the way that people describe architectural structure when speaking informally. In formal architecture description, we are very precise about how we describe relationships among architectural elements. Formally, to say that a controller is connected to a mid-range computer is completely different from saying that the point-of-sale system hosted on that controller is connected to a socket server hosted on the mid-range computer. (In fact, in our architectural notation, only the former is directly representable. It is not possible for a connector to attach directly to subcomponents of two different components.)

In everyday language, though, these can amount to the same thing. Thus, when one person describes a connection between the mid-range systems and the warehouse, another person explains that the controller is connected to the mid-range systems, and a third explains that the warehouse has a link to the corporate hosts, they are all referring to the very same connector—even though they are describing it in different ways, and indeed in our content analysis, such utterances were unitized as though they were different entities. In fact, this is not a fictitious example; the content analysis had several different coding units such as “midrange-warehouse,” “controller-midrange,” “hosts-POS,” and several others that all ended up being realized through the same connector in the model.

One way of understanding such examples is as instances of a very common linguistic phenomenon called synecdoche, in which a part of something refers to the whole or vice versa, as in the phrase “all hands on deck,” where “hands” refers to sailors. Similarly,

in the example above, we could interpret “warehouse” as synecdochically referring to the controller or vice versa.

But there are other ways besides this in which casual language treats architectural structure imprecisely. Sometimes when architects appear to be describing a connection between two components *A* and *B*, they are actually describing a communication pathway comprising a series of connectors, in which *A* and *B* are the endpoints and the communication goes through a number of intermediaries, so *A* is connected to *C*, *C* to *D*, and *D* to *B*. For example, one architect told me that the CRM system has “an integration point to” the point-of-sale system in the target architecture. In the content analysis, I identified this as a connector between the point-of-sale system and the CRM system. But while reviewing the architectural documentation during the modeling effort, I came to believe that in fact there is no direct link between these two systems in the target architecture; rather, this communication goes through a series of intermediaries: the integration hub, the integration architecture, and the MDM system. Such complexities suggest a need for further research on how humans talk about architecture and how to elicit precise architectural information from practitioners.

#### 4.3.2 Evolution operators

Because the point-of-sale evolution is a fairly large-scale evolution with many parts and aspects, architects tended to speak of the evolution in terms of its general goals and major stages, rather than the specific individual operations that will be required to carry it out. Nonetheless, the content analysis did identify a few evolution operations at a finer level of granularity, namely:

1. Removing an existing legacy element
2. Connecting the point-of-sale system to a new system
3. Doing in-house customization of an off-the-shelf point-of-sale system
4. Paying the vendor to customize an off-the-shelf point-of-sale system
5. Upgrading the operating system of the warehouse controllers
6. Deploying new point-of-sale software to the warehouse controllers
7. Eliminating the fuel controllers (so that gas stations share warehouse controllers)
8. Replacing nightly polling of transaction logs with a “trickle poll” setup that provides a steady stream of updates throughout the day
9. Installing an off-the-shelf product in the warehouse
10. Modifying an off-the-shelf package to meet the company’s needs
11. Replacing legacy communication for file transfers with the managed file transfer capabilities provided by the company’s new integration architecture
12. Replacing the data queues used by the socket server with message queues
13. Moving functionalities and responsibilities from a legacy system to the modernized system that replaces it

Obviously, this handful of operators is not sufficient, on its own, to describe all the changes in the point-of-sale evolution—to take us from figure 7 to 8. Because of the scale and complexity of the point-of-sale evolution, a fairly large number of operators would need to be defined to accomplish this. But the operators mentioned in the data are a good basis on which to evaluate our modeling approach. These operators are likely to be no less complex and no less difficult to model than typical evolution operators. On the contrary, it seems reasonable to suppose that the operators that were specifically mentioned by architects in a high-level discussion of an evolution might be more complex on average than the typical evolution

operator, if anything. Certainly, the operators in the list above seem to represent a diverse array of different sorts of operators. Let us thus proceed to consider how these evolution operators could be modeled in our approach.

In our previous work, we have used a few different approaches to model evolution operators. In a journal paper [5, § 3.6], we introduced a pseudocode-like notation for describing architectural transformations. In a case study based on the modeling tool MagicDraw, I used scripts written in the Groovy scripting language to interface with MagicDraw's model manipulation API, allowing the operators to be applied automatically by the tool [3]. In recent work studying automated planning for architecture evolution, we formalized evolution operators as actions in PDDL, a specification language for automated planning [6]. And other approaches are possible too; for example, as we have mentioned in previous work [3; 5], an existing transformation language such as QVT could be used to model evolution operators as well. The following discussion uses the pseudocode-like notation we introduced previously, because it is quite readable and is not tied to a particular modeling language (and hence can be used with Acme, unlike the other mentioned possibilities).

In our approach, an operator comprises three parts: a description of the structural transformations it effects, some preconditions, and additional information to support analysis. In the following examples, I show only structural transformations and preconditions. In section 4.3.4, I will show how operators can be augmented with additional information to support analyses.

1. **Removing an existing legacy element** is a straightforward deletion operator, which is not hard to model at all:

```
operator removeElement(e) {
  transformations {
    delete e;
  }
}
```

2. **Connecting the point-of-sale system to a new system** is also very simple; it amounts to the creation of a new connector between the point-of-sale system and the new system. To avoid ambiguity, the operator should take as parameters the specific ports to join, rather than just the components.

```
operator integrateWithSystem(clientPort, systemPort) {
  transformations {
    Connector c = create Connector : SystemIntegrationT;
    attach clientPort to c.Client;
    attach systemPort to c.Sys;
  }
  preconditions {
    declaresType(clientPort, SystemCallT) and
    declaresType(systemPort, SystemIntegrationPointT)
  }
}
```

3. **Doing in-house customization of an off-the-shelf point-of-sale system** is not an operator that has any structural effects on the architecture, because this operator captures code-level modifications that do not manifest themselves at the architectural level. Nonetheless, modeling this operator is important for reasoning about the evolution, because we do want to ensure that the off-the-shelf point-of-sale system does get appropriately customized at some point during the evolution, and we may want to reason about the time that carrying out the modifications takes, or the money it costs. To make

this possible, we can simply define an operator that sets a property called *customized* on the point-of-sale system to true.

```
operator customizePackageInHouse(componentToCustomize) {
  transformations {
    set componentToCustomize.customized = true;
  }
  preconditions {
    declaresType(componentToCustomize, CotsT)
  }
}
```

4. **Paying the vendor to customize an off-the-shelf point-of-sale system** is identical to the previous operator, except that it may have different analytical properties. For example, paying the vendor to do the customization might be costlier but faster than doing it in-house. Including this kind of information in the definition of the operator permits analysis of trade-offs.
5. **Upgrading the operating system of the warehouse controllers** is similar to the two preceding operators, in that it has no structural effect, but it does take time and money and it is important to ensure that it happens. In this case, we might set an “osUpgraded” property on the controller components to true. Because this is so similar to the last example, I will not provide the specification here.
6. **Deploying new point-of-sale software to the warehouse controllers** has the structural effect of deleting the component that represents the old point-of-sale software and creating a component for the new point-of-sale software in its place. In the model, we can represent this very simply by renaming the appropriate component. (In our operator specification language, the “transformations” block is exclusively for modeling purposes; it specifies the changes necessary to transform the model so as to properly reflect the effects of the operator. It need not somehow capture the reality of the operation; that is, we do not need to delete the legacy component and then create an identical component with a different name in its place, merely to satisfy some tenuous notion of fidelity to the reality of the operation. Instead, the “reality” of the operation—its duration, its cost, its difficulty—is captured through the use of analytical properties.)

```
operator deployNewPos(oldPos) {
  transformations {
    set oldPos.name = "NewPOS";
  }
  preconditions {
    declaresType(oldPos, CotsT) and oldPos.name == "OldPOS" and
    oldPos.osUpgraded
  }
}
```

There a couple of interesting points to note about this example. First, it uses the effects of the operating-system upgrade operator we just discussed as a precondition. Second, this operator would look quite different if our model included a physical view of the system as well as a component-and-connector view, because in such a model deployment could be represented directly: the deployment of a piece of software to a machine can be represented by a deployment relationship between a software component and a physical computer. In this case, the specification of the operator would be a bit more complex, but it would also provide a richer description of the architectural effects. See our previous paper [5] for examples.

7. **Eliminating the fuel controllers (so that gas stations share warehouse controllers)** is a rather complex operator conceptually. The idea here is that while in the current system gas stations have their own point-of-sale systems with their own controllers, the point-of-sale evolution will unify this setup with the main warehouse point-of-sale system, eliminating the need for gas stations to have their own controllers. However, despite the conceptual complexities, this operator is representationally simple. In the target state (figure 8), the warehouse point-of-sale system and the gas station system are unified. All that is necessary to reach this state representationally is to delete all the gas station components. Because we have already shown a deletion operator, I do not provide the operator specification here.

There are alternative ways that we could have chosen to represent the unification of the two systems—for example, by retaining the fuel pumps in the model, deleting the fuel controller, and hooking up the fuel pumps with the warehouse controllers. This would be marginally more complicated to specify, but still very manageable.

8. **Replacing nightly polling of transaction logs with a “trickle poll” setup that provides a steady stream of updates throughout the day** is another conceptually complex but representationally simple operator. In the model, the only effect of this operator is that the type of a connector changes.

```
operator implementTricklePoll(oldConn) {
  transformations {
    remove type oldConn : SystemIntegrationT;
    add type oldConn : TrickleT;
  }
  preconditions {
    declaresType(oldConn, SystemIntegrationT);
  }
}
```

9. **Installing an off-the-shelf product in the warehouse** is a straightforward component creation. The details of the operator specification depend to some degree on the specifics of the installation. For example, should the new component be at the top level of the model, or inside a representation? Does the new component need to be connected to anything, or will that be achieved via separate operator? But in any case, the operator will be quite straightforward to specify, so I do not provide a sample definition here.
10. **Modifying an off-the-shelf package to meet the company’s needs** is similar to many of the operators we have seen, entailing a property change in the model but no structural modifications.
11. **Replacing legacy communication for file transfers with the managed file transfer capabilities provided by the company’s new integration architecture** is only slightly more complex than the examples we have seen. In this operator, the legacy connector used to allow the warehouse controllers to interact with the corporate host systems is disused and replaced with a connection to the integration hub.

```
operator disuseLegacyFileTransferComm(legacyComm, integrationHub) {
  transformations {
    for (Port p : legacyComm.Ctlr.attachedPorts) {
      remove type p : ControllerLegacyCommPortT;
      add type p : MicrobrokerHookupT;
      Component microbroker = create Component : MicrobrokerT;
      attach p to microbroker.Controller;
      attach integrationHub.IntegrationPoint
```

```

        to microbroker.IntegrationHub;
    }
    delete legacyComm;
}
preconditions {
    declaresType(legacyComm, LegacyCommElementT) and
    declaresType(integrationHub, IntegrationHubT) and
    size(legacyComm.Ctrl.attachedPorts) = 1 and
    forall p in legacyComm.Ctrl.attachedPorts |
        declaresType(p, ControllerLegacyCommPortT);
}
}
}

```

12. **Replacing the data queues used by the socket server with message queues** involves not only replacing the data queues themselves, but also modifying the code of the components that use the queues. But in the model, the only thing that changes is the type of the relevant connectors.

```

operator replaceDataQueues(subsystem) {
    transformations {
        for (Connector c : subsystem.components) {
            if (declaresType(c, DataQueueT)) {
                remove type c : DataQueueT;
                add type c : MessageQueueT;
            }
        }
    }
}
}

```

13. **Moving functionalities and responsibilities from a legacy system to the modernized system that replaces it** is another example of an operator that has no structural effects, but can be represented effectively by changes in properties on components.

These operators are remarkable for how straightforward and easy they were to specify. Even the operators that seemed conceptually complex turned out to be simple to define. Although we should be careful of drawing too strong a conclusion from this limited set, this suggests that the operators that arise in real-world evolution may tend to be fairly simple in many cases, and that operator specification is not as difficult as might be feared.

### 4.3.3 Constraints

Sixteen constraints were explicitly mentioned in the data and identified through the content analysis. As was the case with operators, this is certainly not a complete list of constraints relevant to the point-of-sale evolution. We could undoubtedly come up with many other plausibly applicable constraints. But by focusing our modeling efforts on constraints that were explicitly mentioned by architects as being significant in the point-of-sale evolution, we hope to validate the suitability of our constraint specification method for modeling the most relevant and salient constraints in a real-world architecture evolution.

In segmenting and coding the data, I took a fairly inclusive view of what it means for something to be a “constraint.” I included in this category everything that took the form of a constraint somehow governing the evolution, without considering whether it was an architecture-level constraint or a lower-level constraint, whether it was a constraint on the evolution as a whole or only particular states, or whether it was amenable to representation via our approach.

1. One architect I interviewed strongly emphasized the importance of minimizing the impact of the system change on its users:

Their first and absolute foremost goal is to create almost a zero-impact change in our warehouses. They don't want to have to retrain a bunch of people and deploy a bunch of new hardware. There will be an OS upgrade and then there will be a software upgrade, and that generally takes a few hours overnight. People basically leave, they come in the next day, and they have a whole new system and they should notice about this much change [*indicates a small amount*], so there isn't a big training effort now to get people back into the swing of things.

This is a difficult constraint to capture using our methods, because in large part, it is not architectural. That is, avoiding disruption to users has a great deal to do with the specifics of exactly how the change is carried out, and details like user interface elements, and not much to do with architectural structure. In fact, the same architect made a similar point later in that interview, noting that carrying out the evolution would require attention to minute, nonarchitectural details such as the color of the labels on the keys of the cash registers:

The things that boggle my mind we talk about in this project—well, we need to know what color code to put on the key cap. I say, oh, yeah, I never really thought about that. But they're training people that identify sections of that keyboard. So you learn a little bit more every time you turn around. It's kind of interesting. It's like, ugh, these darn labels—I forgot all about them! [*laughs*] And really, is that something the architect is concerned about? Probably not.

But to the extent that this “zero-impact change” constraint does implicate architectural considerations, we should be able to model it. For example, if we wished to model a simple structural constraint like “The number of cash registers in the warehouse remains constant throughout the evolution,” we could easily do so via the temporal formula

$$\{s\} \square \text{equalRegisters}(\text{system}, s.\text{system}),$$

where *equalRegisters* is a binary predicate over architectural states, definable in Acme's architecture constraint language, Armani, by:

```
Design Analysis registerCount(s : System) : int =
    size({ select c in s.Components | declaresType(c, PointOfSaleT) });
Design Analysis equalRegisters(s1, s2 : System) : boolean =
    registerCount(s1) == registerCount(s2);
```

(Incidentally, this constraint actually fails for the evolution shown in figures 7 and 8, because the fuel pumps present in the initial architecture are no longer present in the target architecture, so some fine-tuning would need to be done to adapt the above constraint to the conventions we have adopted to represent the unification of the fuel and warehouse point-of-sale systems.) Obviously, this captures only one tiny aspect of what it means to have “zero-impact change” in the warehouse; again, for the most part this has to be managed at a nonarchitectural level. But another example of something that could be modeled is that the actual deployment of the software should occur overnight, not during the day. In previous work [6], we have examined some of the issues involved in representing constraints about which operators are permitted to happen on which dates.

2. The architectural documentation noted that all modifications to the new POS package were to be performed by an external vendor.

Recall that in section 4.3.2, we defined different operators for doing in-house customization and having a vendor do the customization. Enforcing this constraint therefore amounts to ensuring that the *customizePackageInHouse* operator is never applied to the new POS component.

Interestingly, in our constraint specification language this is not as straightforward as it sounds. Our constraint language is designed for reasoning about intermediate *states*—but the operators that are executed are part of the *transitions* among those states, and are therefore not directly accessible to our constraint specification language. One way to remedy this would be to modify our constraint language to support reasoning about transitions. In principle, this seems simple; one might imagine we could simply represent a constraint like “The *customizePackageInHouse* operator is never applied to the new POS component” by some formula along the lines of

$$\square \neg \text{customizePackageInHouse}(\text{NewPOS}).$$

However, adding support for reasoning about transitions in a principled way entails significant theoretical complexities. A number of logics have been developed that permit reasoning about both states and transitions, such as Chaki et al.’s State/Event Linear Temporal Logic (SE-LTL) [17]. Constructs from such logics could likely be adapted to our constraint language. While this idea is promising and could potentially enrich our language in useful ways, working out the details and evaluating the properties of the resulting constraint language are well beyond the scope of this report.

In the absence of such a language extension, we could achieve the same effect by modifying the *customizePackageInHouse* operator to set a property on the customized package called *hasBeenCustomizedInHouse* to true. We could then very easily represent the desired constraint by asserting that *hasBeenCustomizedInHouse* should never be true:

$$\square \text{newPosNotCustomizedInHouse}(\text{system}),$$

where *newPosNotCustomizedInHouse* may be defined as an architectural predicate in Armani by

```
Design Analysis newPosNotCustomizedInHouse(s : System) : boolean =
  forall c1 : component in s.components |
    forall r1 : representation in c1.representations |
      forall c2 : component in r1.components |
        forall r2 : representation in c2.representations |
          forall c3 : component in r2.components |
            c3.name == "NewPOS" ->
              !c3.hasBeenCustomizedInHouse;
```

Most of the complexity in this Armani predicate is because the new POS component is nested three levels deep in our model: a component inside a component inside a component.

3. Another very similar constraint arising from the architectural documentation was one that said that all configuration and development work to integrate the point-of-sale system with an off-the-shelf microbroker component would be performed by the vendor. Because this is so similar to the previous constraint, I do not provide a formalization here.
4. One architect mentioned availability constraints as being significant to the point-of-sale system:

We shoot for five 9s uptime, which is like twenty minutes of unscheduled downtime a year. We’re not looking at lots of amounts of time. One blue



screen of death, and we're beyond that.

Of course, accurately predicting system availability from architectural structure is a challenging problem—one that has received significant research attention in its own right [37]. It would be interesting to study how well our approach could capture architectural reliability models that have appeared in previous research, but that is well beyond the scope of this case study.

As a simple example, consider a case in which we are interested solely in the reliability of the controllers in the warehouse. A warehouse has two redundant controllers: a primary and a fail-over. Suppose we annotate each of these two components with a property indicating its expected failure rate (e.g., *failureRate* = 0.001 to indicate that the component will be down 0.1% of the time.). If we assume that the two controllers fail independently, we can calculate the overall failure rate of the system by multiplying the failure rates of the two controllers. (Of course, independence will not hold in practice. If the warehouse loses power, both controllers will fail. But it is a reasonable simplifying assumption for our demonstration here.) The constraint that availability never drops below 99.999% then becomes

$$\square \text{highReliability}(\text{system}),$$

where *highReliability* is defined in Armani by

```
Design Analysis highReliability(s : System) : boolean =
  s.PrimaryController.failRate * s.FailOverController.failRate <= 0.00001;
```

- The architectural documentation on the point-of-sale system noted that a desired goal was to implement communication through the company's new integration architecture "for all participating systems." That is, all communication between components attached to the integration architecture should eventually use the integration architecture for communication. This is easy to represent in our temporal constraint language:

$$\diamond \square \text{intArchUsedWhenAvailable}(\text{system}),$$

where *intArchUsedWhenAvailable* is an architectural predicate that checks whether all the participating systems are communicating exclusively through the integration architecture. One way of interpreting this is that there are no connectors directly linking one participating system to another. We can define a participating system as one that connects to the integration architecture element, the integration hub, or MDM. In this case, the Armani predicate is:

```
Design Analysis isParticipatingComponent(c : Component) : boolean =
  connected(CIA, c) or connected(IntegrationHub, c) or attached(MDM, c);
Design Analysis intArchUsedWhenAvailable(s : System) : boolean =
  forall c1, c2 : component in s.components |
    isParticipatingComponent(c1) and isParticipatingComponent(c2) ->
      !connected(c1, c2);
```

- One constraint mentioned in both the interview data and the documentation was the need to align the work being carried out by the company's own engineers with the work being carried out by the vendor:

Coordinating the work of [the vendor] with internal project support resources will require close coordination of work to ensure deliverables are supported for each of the tests required and deployment activity.

Obviously, we lack sufficient detail about the development schedule to model this constraint accurately. But we can discuss how it could be modeled in principle. First,

modeling the code drops from the vendor accurately would probably require us to introduce a notion of real-world calendar time into the model. That is, we would need to capture not only the order in which evolution steps occur, but also when they occur (e.g., January 2014). With this information in the model, we could then coordinate certain operators to occur before or after scheduled vendor code drops.

We have investigated some of the issues involved in modeling real-world time in previous work [6]. In the present example, the simplest way of incorporating calendar time would be to add a system property called *monthsElapsed* that each operator increments appropriately. For example, an operator that takes two months to carry out would increment *monthsElapsed* by 2. (This presumes a sequential model, where one operation is carried out at a time, which may be unrealistic for a large software organization. Concurrently occurring operators are possible, but more difficult to model.) Then, if we know that a certain operation can't be completed until a certain code drop from the vendor has been received, which is expected to come in the sixth month of the evolution, we could define a precondition on that operator: *monthsElapsed* > 6.

7. The architectural documentation mentions the issue of certification of payment systems:

Electronic Payment Support services are not expected to change with the upgrade to [the new POS system]. Any new processes or equipment will require recertification with the EPS vendor.

The second sentence of the above constitutes a constraint. If there is some candidate evolution path in which point-of-sale processes or equipment will change, that change will require recertification.

The most straightforward way to model this constraint is probably as a precondition on the operator that would modify the certified equipment. For example, if we have an operator such as *modifyCashRegisters*, it should have as a precondition that the *recertifyEps* operator has already happened to ensure that the modifications to the cash registers have been approved.

8. The architectural documentation notes that the replacement of the legacy communication facilities between the warehouse controller and the corporate hosts will require modifying the new point-of-sale package to support scheduled or on-demand data transfers. This is a straightforward ordering constraint: before the controller can be connected to the integration hub, the point-of-sale package must be modified to support scheduled or on-demand data transfers. Ordering constraints are very easy to model using our approach. One method is to do define the constraint very simply by

$$\square \text{posUpdatedIfNecessary}(\text{system}),$$

where *posUpdatedIfNecessary* is defined by

```
Design Analysis posUpdatedIfNecessary(s : System) : boolean =
  forall ctr : ControllerT in s.components |
    connected(ctr, IntegrationHub) ->
      ctr.hasBeenModifiedForScheduledDataTransfer;
```

The *hasBeenModifiedForScheduledDataTransfer* property would be set by the operator that completes the necessary modifications to the point-of-sale package.

9. Another ordering constraint arose from the interview data: the legacy communication facility between the warehouse controller and the corporate hosts can't be removed from the system until trickle poll has been implemented. Again, ordering constraints are easy to model using our approach. One way of doing so is by defining the constraint

as

$\square$  *legacyCommPresentUnlessTricklePresent(system)*,

where *legacyCommPresentUnlessTricklePresent* is defined in Armani by

```
Design Analysis legacyCommPresentUnlessTricklePresent(s : System) : boolean =
  !(forall ctr : ControllerT in s.components |
    exists trickle : TrickleT in s.connectors | attached(trickle, ctr)) ->
  forall ctr : ControllerT in s.components |
    exists legComm : LegacyCommElementT in s.connectors |
      attached(legComm, ctr)
```

10. Immediately after describing the above constraint, the architect went on to discuss another: the aforementioned legacy communication facility shouldn't be removed in *any* warehouse until *all* warehouses are on the new point-of-sale package. This is a somewhat more problematic constraint. Because we have chosen to model only a single illustrative warehouse, we can't represent constraints that require consideration of differences among different warehouses. However, if we enriched the architectural model with multiple warehouses, we could define a constraint very similarly to in the previous example:

$\square$  *legacyCommPresentUnlessAceEverywhere(system)*,

where *legacyCommPresentUnlessAceEverywhere* is defined by

```
Design Analysis legacyCommPresentUnlessAceEverywhere(s : System) : boolean =
  !(forall ctr : ControllerT in s.components |
    exists r1 : Representation in ctr.representations |
      exists c1 : Component in r1.components |
        exists r2 : Representation in c1.representations |
          exists c2 : Component in r2.components |
            c2.name == "NewPOS") ->
  forall ctr : ControllerT in s.components |
    exists legComm : LegacyCommElementT in s.connectors |
      attached(legComm, ctr)
```

This works because if the model included multiple warehouses, then the universal quantification would include all controllers in all warehouses. (The extra complexity here is merely because we have to dig deep inside the representation of a controller to find out whether it has been upgraded to the new POS package, as in constraint 2, not because there are multiple warehouses.)

11. The architectural documentation notes that during the evolution, support needs to be provided “for the current POS system to ensure uninterrupted operation throughout the transition” to the new POS package, and that “transition strategies must be carefully crafted for each system dependency to avoid creating gaps in functionality or data integrity.”

“Uninterrupted operation” and functionality are not properties that directly emerge from the architectural model. The best we can do is to model more concrete properties that bear on this constraint, such as: “Throughout the evolution, all registers shall remain connected to the point-of-sale system.” A structural constraint such as this is easily representable by a formula such as

$\square$  *allRegistersConnected(system)*

where *allRegistersConnected* is defined by

```

Design Analysis allRegistersConnected(s : System) : boolean =
  forall reg : PointOfSaleT in s.components |
    exists ctrl : Component in s.components |
      exists ctrlRep : Representation in ctrl.representations |
        exists pos : SystemT in ctrlRep.components |
          pos.name == "POS" and connected(reg, ctrl);

```

12. The architectural documentation indicates that the new point-of-sale system should never be hooked up to the socket server. Such a constraint can be modeled easily by

□ *newPosNotConnectedToSocketServer(system),*

with *newPosNotConnectedToSocketServer* defined by

```

Design Analysis newPosNotConnectedToSocketServer(s : System) : boolean =
  forall ctrl : Component in s.components |
    forall ctrlRep : Representation in ctrl.representations |
      forall pos : Component in ctrlRep.components |
        forall posRep : Representation in pos.representations |
          forall c : Component in posRep.components |
            c.name == "NewPOS" -> !connected(ctrl, WHB400);

```

13. One architect told me that the intent of the evolution was “to make the new system work exactly the same way as the old system”—that is, the final system should have exactly the same functionality as the initial system. This is too vague to model directly, but as with other constraints we have seen, it is easily formalizable provided that we define it more narrowly in terms of architectural structure. The details, of course, depend on how precisely we choose to define it, but this constraint is quite similar in spirit to constraint 1 above; therefore I will not go into specifics here.
14. The architectural documentation notes:

With the introduction of new back-office systems (e.g. CRM, ERP), the need for adhering to a SOA enterprise model becomes acute. All back-office systems will be compliant (or made so) with our SOA strategy to ensure interoperability.

This is too high-level a requirement to model directly. The referenced SOA strategy undoubtedly encompasses a number of specific elements, some of which might be suitable for representation as constraints. However, I did not collect any details on the company’s SOA strategy, so we do not have sufficient data to model specific constraints. As an example, though, one common constraint in SOA systems is that all communication must go through a centralized enterprise bus. A very similar constraint is given as constraint 5 above.

15. One participant mentioned that there are certain constraints on the user interface to which the system must adhere. Although a point of sale has a simple, text-only user interface, there are still requirements on the way information should be displayed to the cashier and printed on receipts. However, such considerations are probably too low-level to be included in an architecture evolution model; they are not really architectural in nature. See constraint 1 above for further discussion.
16. The architectural documentation notes:

The proposed solution assumes that all peripheral devices and systems requiring POS connectivity in the Warehouse will use software and methods provided by [an off-the-shelf integration framework].

This is a structural constraint that must hold in the final evolution state, so it takes the form

$$\diamond \Box \text{intFmwkUsedForAllWarehouseComm}(\text{system}).$$

In the target architecture, this constraint is enforced by interposing the off-the-shelf integration framework between the element within the POS system that represents the new POS package and the POS system port that leads to the rest of the warehouse. Thus, we can define *intFmwkUsedForAllWarehouseComm* to hold when the new POS package is not directly connected to the warehouse LAN (i.e., it has no *WarehouseDeviceLanPortT*) and instead uses the off-the-shelf integration framework.

```
Design Analysis intFmwkUsedForAllWarehouseComm(s : System) : boolean =
  forall ctr : Component in s.components |
    forall ctrRep : Representation in ctr.representations |
      forall pos : Component in ctrRep.components |
        forall posRep : Representation in pos.representations |
          forall posPkg : Component in posRep.components |
            (posPkg.name == "NewPOS" ->
              !(exists p : WarehouseDeviceLanPortT in posPkg.ports) and
                (exists c : Component in posRep.components |
                  connected(c, posPkg)));
```

Let us summarize the preceding discussion. Of the sixteen constraints that the content analysis identified:

- Four (constraints 5, 9, 12, and 16) are easily representable as evolution path constraints in purely structural terms.
- Three (constraints 2, 3, and 8) are easily representable as evolution path constraints once we define appropriate architectural properties to be set by operators.
- Four (constraints 4, 6, 11, and 14) are very high-level constraints that we lack the necessary detail to model in full, but are representable in principle.
- Three (constraints 1, 13, and 15) are low-level, chiefly nonarchitectural constraints, but are representable to the extent they implicate architectural considerations.
- One (constraint 7) is representable as an operator precondition.
- One (constraint 10) is representable only with significant modifications to the model.

The exact figures are not important. In many cases, there was some degree of choice in representing the constraints. Ordering constraints, for example, can typically be represented either as evolution path constraints or as preconditions.

What is important is the overall result: all the architectural constraints captured in the content analysis are representable using our approach, albeit with varying degrees of difficulty. We have seen some specific areas where our constraint language might be able to benefit from further enrichment, such as support for reasoning directly about the transitions in an evolution graph in addition to the states. But we did not encounter any architecture evolution constraints that could not be modeled at all.

Another interesting observation we can make pertains to the structure of the constraints that we modeled above. The preceding discussion included ten temporal formulas capturing evolution path constraints. Of these ten, seven could be expressed in the form  $\Box \phi$  for some architectural predicate  $\phi$ , meaning that they could be expressed as architectural constraints that were required to hold in each state of the evolution. Another two took the form  $\diamond \Box \phi$ , which (in a finite-state model such as ours) amounts to an architectural constraint on the final evolution state. Only one, constraint 1, required us to use the rigid-variable operator that we introduced in our previous work, and even that took the simplest possible form:  $\{s\} \Box \phi$ ,

where  $\phi$  is a binary architectural predicate. This suggests that real-world evolution path constraints may tend to be reasonably simple when expressed as temporal formulas. This is an encouraging result for the usability and tractability of our modeling approach, since it suggests that definition of path constraints may not generally require sophisticated reasoning about complex temporal formulas.

#### 4.3.4 Evaluation functions

As we have done with operators and constraints, we now describe each dimension of concern that was identified in the content analysis and how it could be modeled as an evolution path evaluation function.

This section follows the convention set by our earlier work [5], in which evaluation functions are defined as functions in JavaScript (more precisely, ECMAScript, edition 5.1 [30]) and operator properties are defined in JavaScript Object Notation [26].

1. The first dimension of concern identified in the content analysis was *cost*. Cost was mentioned several times as a relevant dimension of concern. In our approach, cost can be modeled as a property of operators. Each operator can be assigned a property describing its estimated cost in dollars. For example, if we know that the *replaceDataQueues* operator costs 500 dollars to carry out, we can add a property to the definition of the operator:

```
operator replaceDataQueues(subsystem) {
  transformations { ... }
  analysis { "costInDollars": 500 }
}
```

Of course, in some situations, cost may be more complex—depending, for example, on the operator parameters—but this is the basic idea. Then, an operator analysis can simply add up all the operators within an evolution path to get an estimate of the overall cost of the path.

```
function analyzeCost(states, transitions) {
  var total = 0;
  transitions.forEach(function (transition) {
    transition.operators.forEach(function (operator) {
      total += operator.analysis.costInDollars;
    });
  });
  return total;
}
```

The only difficult task is accurately estimating the costs of the operators. However, this is outside the scope of my research. There is a very large body of work on cost estimation in software engineering, so we may simply assume that the costs of the operators can be determined through traditional methods.

2. Another dimension of concern that arose in the content analysis was *effort* or development time. From a modeling standpoint, effort is extremely similar to cost. As with cost, we can define effort as a property of an operator (whose value can be determined through traditional effort estimation methods), then define an evaluation function that adds the effort values of all the operators in an evolution path. The only difference is that the unit of measurement is person-hours rather than dollars.
3. Another significant dimension of concern mentioned by architects is the overall *positive value* resulting from the evolution—the return on investment. This dimension is somewhat ephemeral; it's often unclear how to measure or quantify the benefits of a

software system. But to the extent that such benefits can be quantified—say, in terms of expected revenue for the company—we can analyze them using our approach.

One way of doing this is to define the benefit that each operator is expected to yield—for example, we might be able to say that adding the “trickle poll” functionality mentioned earlier will be worth some certain amount of money to the business. If we take this approach, a benefit analysis will look virtually identical to the cost analysis just discussed.

Another approach is to associate benefits with states rather than transitions. After all, operators don’t directly provide benefits; operators are beneficial only because of their effects on the state of the system architecture. Thus, it might be more naturally to associate with each state a property describing the benefit or utility that that state provides. (Alternatively, a more sophisticated approach would be to derive a state’s utility from its architectural structure, calculating the utility of an evolution state in terms of the benefits that its architectural elements provide.)

Defining the benefits arising from evolution states permits a number of different kinds of benefit analyses. Most simply, we can measure the expected benefit when we reach the target state:

```
function analyzeEventualBenefit(states, transitions) {
  var finalState = states.pop();
  return finalState.benefit;
}
```

But more sophisticated analyses are also possible. For example, if we know how long we will be in each state, we can multiply benefits by state durations to obtain a measure of accrued benefit over time:

```
function analyzeBenefitAccruedDuringEvolution(states, transitions) {
  return states.reduce(function (total, state) {
    return total + state.benefit * state.duration;
  }, 0);
}
```

4. A dimension of concern mentioned in both the interviews and the architectural documentation was *performance*. As one architect explained:

At this point, I’m just kind of concerned about performance. This thing has to be lightning-fast, and we don’t know what it’s going to look like yet.

There is a substantial body of research on architectural analyses of performance [2]. Our approach is sufficiently general to accommodate any analysis that derives a performance estimate from a standard representation of architectural structure. To illustrate this, I will present a simple performance analysis, in which we estimate the latency between a cash register in the warehouse and the transaction-processing host system by adding up the latencies of the connectors between them. This analysis considers only the target state; a more sophisticated analysis might calculate some weighted average over all the states in a path.

*// Uses Dijkstra's algorithm to find the length of the shortest path from PointOfSale1  
// to TransactionProcessor in the final state, where "shortest" is measured in terms  
// of total latency. In order to make this specification as brief as possible, I have  
// been deliberately inefficient in my implementation of this algorithm; there are a  
// number of obvious optimizations that could be made.*

```
function analyzeFinalLatency(states, transitions) {
  var finalSystem = states.pop().system;
  var components = finalSystem.components;
```

```

var register = finalSystem.PointOfSale1;
var txProc = finalSystem.TransactionProcessor;

// Simple implementation of Dijkstra's algorithm
components.forEach(function (u) { u.distance = Infinity; });
register.distance = 0;
while (true) {
    components.sort(function (u, v) { return v.distance - u.distance; });
    var u = components.pop(); // Component with smallest distance
    if (u == txProc || !isFinite(u.distance)) return u.distance;

    // Traverse to each component v that is connected to u
    u.ports.forEach(function (port) {
        port.attachedRoles.forEach(function (role) {
            var connector = finalSystem.parent(role);
            connector.roles.forEach(function (role) {
                role.attachedPorts.forEach(function (port) {
                    var v = finalSystem.parent(port);
                    var newDistance = u.distance + connector.distance;
                    if (newDistance < v.distance) v.distance = newDistance;
                });
            });
        });
    });
}

```

5. Another concern that was mentioned was *implementation details*. As one architect explained:

When you're doing something that is very outwardly facing, like a point-of-sale system, you have to be concerned about all of the training and all of the implementation details—there's new wiring, there's new routers, there's new switches, there's new devices, there's new keypads...

Although this was identified by the content analysis as a dimension of concern, it is not really an architectural one. Indeed, implementation details are almost by definition nonarchitectural. Thus, this concern is, for the most part, beyond the scope of what we aim to analyze in our approach. To the extent that these implementation details implicate architectural considerations, they may be analyzable, but otherwise we treat them as too low-level to consider as part of the architectural analysis.

6. The architectural documentation notes:

We will endeavor to perform the minimum necessary modifications to the current system to minimize investment in legacy technology [...].

There are a number of ways we could define an analysis to measure the effort spent on *modifying legacy technology*. A simple one is to count the number of operators that involve legacy components. Of course, this requires a definition of "legacy components;" we could hard-code a list of legacy components, but more elegant is to define a legacy component as one which is present in the initial architecture but not the target architecture. Here is an evaluation function which implements this approach:

```

// Counts the number of operators in the transition path that have a "legacy
// component" (as defined above) as at least one of their parameters

```



```

function countLegacyOperators(states, transitions) {
  // Returns a function that tests whether a component is the same as a
  function sameComp(a) {
    // Testing for equality isn't sufficient because the same component can be
    // represented by different objects in different states, so we check whether
    // the components have the same type and name
    return function (b) { return a.type == b.type && a.name == b.name; }
  }

  // Get the set of "legacy components"
  var initialComponents = states[0].system.components;
  var finalComponents = states.pop().system.components;
  var legacyComponents = initialComponents.filter(function (initComp) {
    return !finalComponents.some(sameComp(initComp));
  });

  // Count the number of operators applied to "legacy components"
  var count = 0;
  transitions.forEach(function (transition) {
    transition.operators.forEach(function (operator) {
      if (operator.parameters.some(function (param) {
        return legacyComponents.some(sameComp(param));
      })) count++;
    });
  });
  return count;
}

```

Note that this measure will certainly never be zero; some work involving legacy components is clearly necessary to carry out the evolution. But defining the evaluation function in this way gives us a way to *penalize* excessive reworking of legacy systems by minimizing the number of operators applied to legacy components.

7. The final dimension of concern identified by the content analysis was *flexibility*. Both architects I spoke with about the point-of-sale system indicated that a key goal of the evolution is making the system flexible and open to future changes.

System flexibility is difficult to estimate based on an architectural model. There is an existing body of research on architectural analysis of the flexibility, evolvability, or changeability of software systems [9; 13; 27; 50].

I did not attempt to implement any architectural analyses for flexibility as part of this case study. To the extent that flexibility is determinable from standard architectural models, it is analyzable using our approach. However, many existing methods for architectural analysis of flexibility are not based chiefly on an architectural model, but instead rely heavily on procedures such as definition of change scenarios [9; 50] or interviews with stakeholders [27]. Such methods lie well beyond the scope of the kind of analysis our approach aims to support.

With a number of evaluation functions defined, we could go on to define a measure of *overall utility* as a weighted composite of these primitive functions. To illustrate how this would work, let us create a very simple composite evaluation function, defining utility as benefits minus costs:

```

function evaluateOverallUtility(states, transitions) {
  return analyzeBenefitAccruedDuringEvolution(states, transitions)
    - analyzeCost(states, transitions);
}

```

}

Let us summarize the preceding discussion. Of the seven dimensions of concern identified by the content analysis:

- Five (concerns 1, 2, 3, 4, 6) can be modeled as evaluation functions with little trouble.
- One (concern 5: implementation details) refers to low-level, nonarchitectural details that are not relevant to an architectural analysis.
- One (concern 7: flexibility) is an architectural concern that cannot be easily analyzed using our approach.

## 5. Findings

In section 4, we discussed the case study’s analysis procedures, including the use of two content analyses: one (content analysis 2) feeding into an evolution modeling phase and one (content analysis 1) to be interpreted directly by conventional means. In section 4.3, we discussed how the modeling phase following content analysis 2 was carried out and what its findings were. In this section, we examine the findings of content analysis 1. In section 6, we will synthesize these findings and explain how they address the research questions defined at the outset of the study.

The following subsections correspond to the top-level categories of content analysis 1: “Motives for evolution” (section 5.1), “Causes of problems” (section 5.2), “Consequences” (section 5.3), “Challenges” (section 5.4), and “Approaches” (section 5.5). It may be helpful to refer to the coding guide in appendix B.1 while reading the following findings.

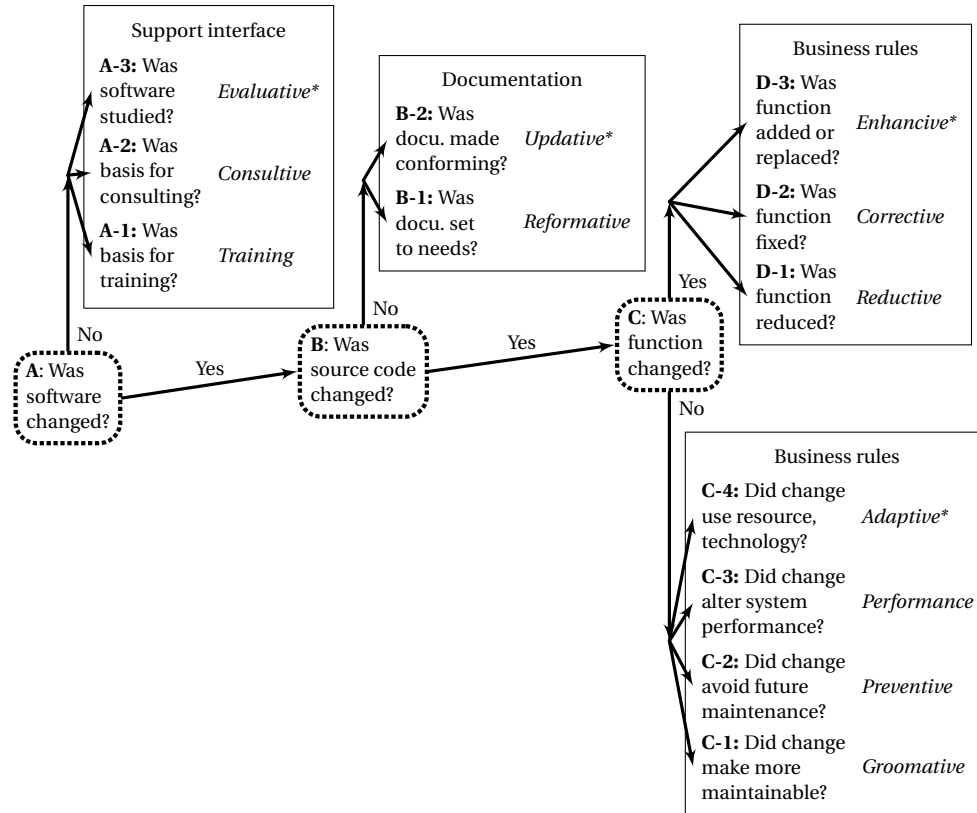
### 5.1. *Motives for evolution*

The first topic that the content analysis examined was the impetuses that motivate evolution—the catalysts that drive an organization to carry out major changes to its software systems. Of course, there is a great deal of previous research on reasons for software changes. One of the first influential taxonomies of software change was the one proposed by Swanson [85] in 1976, at the second International Conference on Software Engineering. Swanson identified three main kinds of software maintenance: (1) *corrective maintenance*, designed to address failures of a software system; (2) *adaptive maintenance*, designed to respond to changes in the environment; and (3) *perfective maintenance*, designed to improve a software system that is already working adequately. In 1980, Lientz & Swanson [52] reported on a large survey of software organizations in which they measured (among many other things) the amount of effort spent on these three types of maintenance, finding that more than 50% of maintenance effort was perfective. Swanson’s taxonomy was so influential that it was incorporated, many years later, into ISO/IEC 14764 [40, § 6.2], an international standard on software maintenance, with the addition of a fourth type: *preventive maintenance*.<sup>2</sup>

A more modern ontology of software maintenance is that of Chapin et al. [19]. Chapin et al.’s classification system is structured as a decision tree (fig. 9). The ontology includes documentation changes and changes to the software support interface in addition to changes to the software system itself. We are concerned only with the latter, so only the rightmost part of Chapin et al.’s decision tree is relevant to us, namely the following seven categories of maintenance: *enhancive* (adding or replacing functionality), *corrective* (fixing functionality), *reductive* (removing functionality), *adaptive* (changing technology or resources used by the

---

<sup>2</sup>The addition of preventive maintenance to this typology is often incorrectly attributed to Swanson himself. Many sources cite the term “preventive maintenance” as appearing in Swanson’s 1976 paper or Lientz & Swanson’s 1980 report, but in fact it appears in neither. For a well-researched discussion of the addition of preventive maintenance to Swanson’s taxonomy, see Chapin [18].



**Figure 9:** Chapin et al.'s [19] classification system for software evolution and software maintenance. This diagram is adapted from figure 2 in Chapin et al.'s paper. The decision tree is read from left to right, bottom to top. Type names are shown in italics. An asterisk indicates the default type within a cluster.

system), *performance* (altering system performance), *preventive* (avoiding future maintenance), and *groomative* (improving maintainability).

Very little research has been done on motivations for *architectural* evolution specifically, as opposed to software maintenance generally. Of course, it is true that many of the motivations for low-level software maintenance can also be motivations for architectural-level software evolution and vice versa, but there are likely some important differences, which previous work has rarely examined. Williams & Carver [88] developed a taxonomy of software architecture changes (called the Software Architecture Change Characterization Scheme) that categorizes architectural changes along a number of dimensions including a change's motivation, source, importance, and so on. Their taxonomy incorporates only two classes of motivation for architecture changes: changes motivated by the need for an *enhancement*, and changes in response to a *defect* [§ 5.2.1]. Besides being reductive in the extreme, this dichotomy has another problem: Williams & Carver's taxonomy is based on a literature review of work on software maintenance generally (such as the aforementioned work by Lientz & Swanson and Chapin) rather than any research specifically focusing on architectural-level change. Thus, while it purports to be a taxonomy of architectural change, it is not actually based on architecture research. Similarly, Jamshidi et al. [42, table 6], in their classification framework for "architecture-centric software evolution," borrow the software maintenance typology from ISO/IEC 14764 for their "Need for Evolution" dimension.

However, there is at least one empirical study that has examined causes of architectural-

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Add features	12	4	3
Improve interoperability	9	3	2
Modernize technology	5	4	3
Keep apace of business needs	3	3	3
Improve reliability	3	2	2
Improve flexibility	3	1	1
Improve performance	2	2	2

**Table 1:** Stated motivations for architecture evolution

level evolution specifically. Ozkaya et al. [68, § 4.2.1], in an interview-based survey of nine software architecture projects, collected data on the causes of the evolutions that the survey participants described. Responses included new features, market change, technology obsolescence, and scalability.

Clearly, more research investigating motivations for architectural change is needed.

In our own previous work, we have—in the course of providing background motivation for our research—put forth several implicit hypotheses about why architecture evolution happens. Nearly every paper we have published on the topic of architecture evolution opens with some variation on the following assertion:

Software architecture evolution is a phenomenon that occurs in virtually all software systems of significant size and longevity. As a software system ages, it often needs to be structurally redesigned to support new features, incorporate new technologies, adapt to changing market conditions, or meet new architectural quality requirements. In addition, many systems over the years tend to accrue a patchwork of architectural workarounds, makeshift adapters, and other degradations in architectural quality, requiring some sort of architectural overhaul to address. At present, however, software architects have few tools to help them plan and carry out these kinds of architecture evolution. [3; see also 4–6; 34]

This passage enumerates several possible causes of architecture evolution: new feature requirements, changing technology, the economic environment, quality concerns, and accumulated technical debt. But these explanations, while plausible, are based on our own general impressions, not on any data about how architects actually think about software evolution and the causes that motivate it.

Of course, our objective in the present work is by no means to develop a general taxonomy of motivations for architecture evolution. Clearly, data from a single case study at a single organization would provide insufficient evidence to support such a taxonomy even were it our wish to construct one. But what this data can do is help inform us as to how practicing architects in a real software organization think about and describe the causes of evolution, and it can also illuminate the rest of the case study by properly setting the context. With that said, I now proceed to describe the findings of the content analysis with respect to motives for evolution.

The interviewees I spoke with mentioned a variety of motivations for evolution, which I grouped into seven categories. These are listed in table 1.

Table 1 illustrates the basic format that all the frequency tallies from the content analysis will use. The subcategories of the relevant category from the coding frame are listed, along with the frequencies with which they appeared in the coded data. The frequency of a category

can be reasonably measured in any of three ways in this study: by counting the number of individual coding units to which the category is assigned, by counting the number of interviews in which the category occurred at all, or by counting the number of participants who had any statements to which the category was assigned. (There is not a one-to-one correspondence between interviews and participants because some participants took part in multiple interviews, and in some interviews two participants were present.) None of these measures is perfect. Probably the most customary choice would be the number of interviews, since content analysts most often count their results with respect to the unit of analysis, which in this case is an interview. However, the divisions between interviews are not particularly meaningful here. For example, in one case an interview was ended on Monday afternoon so that the participant could head home at the end of the workday, then resumed on Tuesday; this was treated as two separate interview sessions. So a more natural measure might be the number of participants, rather than the number of interviews. Both of these measures, though, are quite coarse, since the total number of participants was rather low. A more fine-grained measure is the number of coding units, but this has problems too. The coding units in this study varied dramatically in length, ranging from a short phrase to multiple paragraphs, so different coding units may not carry equal weight. Although none of these measures is perfect, in concert they give a very good sense of how often the categories occurred.

Several of the categories that appear in table 1—improving flexibility, improving performance, improving reliability, and improving interoperability—fall under the general heading of improving architectural qualities. Thus, if I were to summarize these results in a sentence, I would say that most of the evolutions that architects described arose from one of three general motivations: adding new features, modernizing old technology, or improving a system's architectural qualities.

Even at that high level, though, the lines between these broad classes of motivation are blurry. For example, many of the features that were added were themselves motivated by quality concerns. One of the evolution goals classified as a new feature was improving order status information for customers who purchase items online. But the underlying reason for the addition of this new feature was, of course, to improve the usability of the system for customers. Similarly, the underlying reason for technology modernization is often to improve reliability or maintainability. In the content analysis, I dealt with this by defining explicit rules for the categorization of such dilemmatic coding units (appendix B.1). But it is important to bear these ambiguities in mind when interpreting the results.

In the remainder of this section, we will examine more closely the motives that participants articulated in their discussions of architecture evolution. As noted in table 1, the most frequently mentioned motive was to add a new feature. A wide range of specific features were mentioned: personalization features for members, harmonization of the member experience across international borders, legal compliance features such as tax calculations, implementation of support for EMV (“Chip and PIN”) credit cards, near-real-time transmittal of transaction data, customer access to order status information, improved sales forecasting, reconciliation of customer data across databases, and allowing memberships to be sold in blocks.

The second most frequently mentioned motive for evolving a system was to improve interoperability. This was by far the most frequently mentioned among the motives pertaining to improvements of architectural qualities. This suggests that interoperability of systems may be a particularly important architectural quality where evolution is concerned—or at least that it was very much on the minds of the architects I interviewed. For example, one architect summarized the main motivation for the point-of-sale evolution that we studied in detail in section 4.3 as follows:

We're going from a very proprietary, inflexible point-of-sale to a much more open, easily adapted point-of-sale that can talk to most systems, so that fits in with what we're trying to do overall in modernization.

The third most frequent evolution motive was technology modernization. It is not surprising that replacement of outdated technology should be highly relevant to a company with a decades-old IT organization. Several times, architects mentioned their desire to move away from specific obsolescent technologies. But technology modernization can also be relevant to customer-facing systems in the warehouse, as one architect explained:

One example I always use is we currently have endcaps that have a juicer on them and we've got a TV/VCR combo playing on this looped commercial. In today's world, that doesn't really make sense. Why don't we just have a digital monitor there hooking into some cloud-based information? Now we're running some digital content. When we change that endcap and, say, move stuff around, we click a button, we change to the new digital content. Much more useful in today's world.

None of the four remaining categories was mentioned more than thrice. Three of those four categories describe quality improvements as motivations for architecture evolution—specifically, improvements to a system's reliability, flexibility, or performance. Were we to consider them together as a single, merged category, improvements to architectural qualities (including the three just mentioned plus interoperability) would be the single most frequently mentioned motivation for evolution, beating feature additions.

The remaining category, “Keep apace of business needs,” describes evolutions that are driven by the rapid pace of change within the company or within the industry. A good example of this category is the way in which one architect describes how the company's overall modernization efforts arose:

The company was bursting at the seams in its IT infrastructure and systems. The feeling was from the IT management that our current architecture and application systems would not scale to meet the growing demand of our growing business [...]. We've been growing both in the footprint of the number of warehouses that we have as well as, with the exception of one year, every year the warehouses are selling more at the same location. So it's a growth in every direction. And then we're also looking to grow the international market larger. That's going to put demand on our systems as well. And then from a business perspective, like many companies, I'm sure, the users felt like it just takes too long to deliver IT projects. Pretty common two dynamics. Those two major factors came together to decide we need to do a major modernization here at Costco.

In summary, architects mentioned several different kinds of reasons for the architecture evolutions they discussed (including past evolutions with which they had had experience, evolutions that were in progress, and evolutions that had been considered but not undertaken at the time of the study). The great majority fell into three broad classes: evolutions motivated by a need to improve architectural qualities such as interoperability and reliability, evolutions motivated by new feature requirements, and evolutions driven by a desire for technology modernization.

These results—and especially the relative frequencies with which the categories occurred—are probably somewhat specific to Costco. Costco is a decades-old company with a mature IT organization—one that is at present undergoing significant growth. Thus, it is not surprising that much of the evolution effort there is focused on modernizing obsolescent systems and ensuring that the architecture can meet the business's performance and reliability needs, for example. Were we to repeat this case study in a different kind of company—a five-year-old start-up, for example—the results would likely be different. A small, young organization would probably be much less driven by issues of technology modernization and scale, and perhaps more driven by feature development or other kinds of evolution. (Evolution in general would likely also constitute a lower proportion of software development activities,

with greenfield development of new software systems being more prominent.) Thus, in generalizing these results, and those of the following sections, it is important that we keep in mind what special qualities of the case under study may have influenced the results, and to scope our generalization accordingly. We will return to the issues of generalization and external validity in section 6.2.2.

Finally, it should be emphasized that the fact that certain causes don't appear in table 1 does not mean that such causes are not important. For example, the fact that technical debt, or the architectural messiness of legacy systems, does not appear here doesn't mean that that isn't an important factor in evolution for the company. On the contrary, it certainly is. However, the participants in the study did not generally speak in those terms in their impromptu explanations of reasons for evolution—or to the extent that they did, those remarks were categorized under “Evolution motives: Modernize technology” and thus grouped in with other considerations pertaining to the need to modernize technology. Thus, care should be taken in drawing negative conclusions from these results—conclusions that unnamed causes are *not* important.

The categories that appear in table 1 are those that naturally emerged from the data. (Recall that the definition of the subcategories in content analysis 1 was almost entirely data-driven, or inductive, not concept-driven, or deductive.) The results here thus do several things: they highlight some important classes of evolution motives that demonstrably do occur in a real-world organization, they allow comparison of the relative frequency with which those motives were reported, and they show how architects in the case study described the causes of evolution. But they don't permit us to infer anything about the frequency or importance of causes that were not explicitly mentioned.

## 5.2. *Causes of problems*

The second major category of the content analysis was causes of problems—circumstances that architects described as leading to specific negative outcomes during the course of an evolution project. (I also coded the consequences of such circumstances; see section 5.3.)

Of all the major categories, “Causes of problems” was the least frequent, and “Consequences” was the second least frequent. This means that participants spoke less about problems than they did about motives, challenges, and approaches. There are a few possible reasons for this. One possibility is that participants were reticent to speak about the company's failures; it would be understandable that architects might want to put their best foot forward in representing the company to an outsider, especially a researcher who was going to make the results of his study public. There may be an element of truth to this. However, my impression was that the people I spoke with were generally quite candid with me. Many of them were quite frank in discussing disagreements that they had had with other architects, or cultural factors that they believed were problematic. Many of them also expressed a sincere desire to facilitate the case study and to help me obtain the best results possible. It seems unlikely that problems were deliberately concealed.

That said, it is certainly possible that participants phrased their answers diplomatically to avoid casting their employer in an unduly negative light. For example, if an architect believed that decision X had been a mistake, he might not directly say, “We made a mistake with decision X and ran into trouble because of that.” Instead, he might say, “Making good decisions about X has been particularly challenging for us.” But if he phrased it this way, the statement would be coded as a challenge instead of as a cause of a problem.

This brings us to another factor that could have contributed to the low number of coding units: the coding frame. The coding frame provided specific guidance for distinguishing between causes of problems and challenges: only if a circumstance caused a specifically articulated negative outcome should it be categorized under “Causes of problems;” otherwise, it should be categorized under “Challenges.” Perhaps this criterion was too constraining, and greater liberties should have been taken in reading between the lines to distinguish

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Cultural inertia	4	2	2
Architects spread across the organization	2	1	1
Addressing too many problems up front	2	1	1
Lack of experience	1	1	1
Forking an off-the-shelf package	1	1	1
Ill-timed changes	1	1	1

**Table 2:** Stated causes of evolution problems

between general challenges and causes of specific problems. Alternatively, perhaps these two categories should have been merged to avoid the issue entirely.

A final possibility is that there genuinely have not been many poor decisions that resulted in adverse consequences. Certainly, the company has a mature IT organization with many experienced, thoughtful, and skilled architects. Perhaps they have successfully averted or mitigated issues that might otherwise have caused significant problems.

Because there were few coding units identified as causes of problems, each subcategory had only a handful of occurrences—and in some cases, only one. The frequencies are given in table 2.

The most common category—in fact, the only one with more than two occurrences—is “Cultural inertia.” This category refers to situations in which aspects of corporate culture hinder fast-paced changes. Two specific cultural factors were mentioned as causing problems. The first was the company’s build culture; the organization has a long history of building nearly all its software systems from the ground up rather than buying off-the-shelf products, which is somewhat rare in the retail industry. As one participant explained:

For fifteen years, Costco was a build shop. We built our general ledger, we built accounts receivable, our core financial systems, core membership systems. These, of course, due to the timing, were built on older systems—archaic systems now. [...] and when we started modernization, for whatever reason, we were talking more about COTS, but kind of in our DNA, we were still thinking *build*.

This tension, he said, led to some friction as the company transitioned from being a “build shop” to making significant use of off-the-shelf packages.

The second cultural issue that was mentioned was what one architect called “a culture of consensus.” Participants explained that great care is taken to gather input from all relevant stakeholders before making decisions that are significant to the company. That is, decision making tends to be consensus-based, as opposed to the top-down, command-and-control management style that prevails in some organizations. Participants said that this culture of consensus was a good thing for the most part, but that it sometimes got in the way of rapid progress. One participant put it this way:

We’re very sensitive to how people feel about things. And that’s wonderful under normal circumstances—that’s why I work here—but it’s very tricky in times of serious change.

Aside from cultural inertia, five other causes were cited as resulting in adverse consequences. Each was mentioned by only a single participant in a single interview, so I will not dwell on them at length, but I will briefly describe each one:



	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Wasted effort	6	4	4
Delays in evolving	2	2	2
Limited upgradability	1	1	1
Lost sales	1	1	1

**Table 3:** Stated consequences of evolution problems

- **Architects spread across the organization.** One participant mentioned communication difficulties that existed several years ago and were resolved by the reorganization described in section 2.2.
- **Addressing too many problems up front.** One architect recounted a situation in which an evolution effort had been delayed by initial difficulties in understanding and anticipating problems that might arise during the evolution. He argued that they should have spent less time considering specific problems, and instead designed a system that would be flexible and resilient in the face of unanticipated difficulties.
- **Lack of experience.** One participant attributed some of the evolution difficulties that the company faced in the past to inexperience in dealing with the new technologies, approaches, and ideas that the company has adopted, along with a general “lack of organizational maturity.”
- **Forking an off-the-shelf package.** One architect recalled a mistake that had been made years ago, in which they took an off-the-shelf package and customized its source code to meet the company’s needs. This prevented them from upgrading the package with newer versions later.
- **Ill-timed changes.** This final problem was not one that happened at Costco, but instead an issue that arose at another retailer that a participant was familiar with. The retailer rolled out a major software change a week before Black Friday, the beginning of the Christmas shopping season. The rollout went poorly, and the retailer sustained significant financial losses due to the timing of the change.

### 5.3. Consequences

“Consequences” is the sister category to “Causes of problems.” It describes the bad outcomes that resulted from issues such as those discussed in section 5.2. The frequencies from the content analysis appear in table 3. (See section 5.2 for a discussion of why there are so few occurrences of the “Consequences” category as a whole.)

By far the most commonly reported negative outcome was wasted effort. This category encompasses a range of specific repercussions, including unnecessary work carried out by engineers, needless complications during the course of an evolution, rollback of a failed evolution, and backtracking along an evolution path. It is chiefly this class of problems that our research on architecture evolution seeks to address.

Aside from wasted effort, delays in evolving were also occasionally mentioned as a negative consequence. This category captures situations in which an evolution took longer to get under way than it would have if the company had acted more decisively.

The remaining two categories are consequences that arose from specific incidents described in section 5.2. “Limited upgradability” was the outcome of the story in which the company bought an off-the-shelf package and modified its source code, making it impossible to install further upgrades. “Lost sales” was the result of another retailer’s bad choice to roll

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Communication, coordination, and integration challenges	22	7	5
Dealing with rapid change and anticipating the future	13	6	4
Business justification	9	6	5
Dealing with legacy systems	9	5	5
Scalability, reliability, and performance	8	2	3
Managing the expectations and experience of users and stakeholders	7	4	5
Managing people	7	3	3
Inadequate guidance and tool support	6	4	4
Cultural challenges	5	3	3
Lack of expertise or experience	5	3	3
Divergent understandings of architecture	4	2	2
Managing scope and cost	3	3	3
Dealing with surprises	3	2	2

**Table 4:** Stated challenges of architecture evolution

out major changes just before Black Friday, costing that retailer significant revenue.

#### 5.4. Challenges

An interview-based case study provides a unique opportunity to learn about the challenges that real-world architects face. Software architecture research, including our research, is often driven by researchers’ beliefs about what will help practicing architects. These beliefs are often founded on our subjective impressions, generalizations from our own experiences, and informal conversations with practitioners. However, it is important to ground these beliefs with real empirical data whenever possible, and one of the best ways to do so reliably is to ask architects about the challenges they face in the context of a research interview.

“Challenges” should be understood broadly here. I collected data on all kinds of challenges that architects face in evolving systems—not just the kinds of challenges that appear to be addressable through new tools or approaches. Encouraging architects to speak freely about the challenges that they face, rather than simply asking them about the narrow classes of challenges that our research is designed to address, gives us the broadest and least biased picture of architects’ needs, and helps us to accurately understand the role that approaches like ours can play in addressing a realistic subset of the challenges that architects face.

Frequencies appear in table 4. The most common code was “Communication, coordination, and integration challenges,” an expansively defined category that includes a broad range of difficulties, including challenges of communication among people, challenges in coordinating efforts, and challenges in integrating systems. When developing the coding frame, I chose to group these somewhat disparate topics together under a single category because issues of organizational communication are often closely tied to issues of software integration. With 22 occurrences, this category was extremely common—the second most

frequently applied category in the entire coding frame. Broadly speaking, this suggests that participants view communication and coordination challenges as a very important consideration in architecture evolution efforts. The following commentary from one participant provides a useful overview of some of the broad communication and coordination issues the company has faced, and how issues of organizational communication are tied to software integration:

Our systems, say, five years ago were very siloed, and so there wasn't a gigantic need for things like service bus or deep integrations or anything, because these systems were custom-written. [...] these were siloed systems that did not need to cross system lines or people lines or even departmental lines. It used to be [...] where if I ran the depots—the warehouses—at Costco, I would pick up the phone. There were maybe twenty people in the depot IT team. They would take my order (it was very request-based), they would program it up, they'd run it over to me, we'd do user acceptance testing, we may or may not have done any QA, certainly no perf testing other than rudimentary, and then I would roll it out, and I'd probably be the guy that trained you, and I would do the break fix and all that, and then future requests.

We don't do that now. Now it's very cross-matrix on the person side. So architecturally, we've got to worry much more about things like integrations. We've got to worry much more about hardware and software standards, because those are going to be on shared systems. Those are going to be in shared environments. So we've got to make use of that. Also, the implementation groups don't get to decide what the architecture looks like. We centralized architecture and planning on some [...] teams, so in some ways those other teams are a little bit more implementors, and a little bit more custodial in what they have to do.

A litany of specific communication and coordination challenges were named by participants: appropriate use of documentation, ensuring conformance between architecture and implementation, reliability issues with particular messaging technologies, coordination challenges with an off-site development team, challenges in soliciting input from stakeholders, and challenges in communicating in the face of rapid growth of the IT organization (particularly since the IT organization is spread across different buildings).

Aside from these specific challenges, each mentioned once or twice by a single participant, there was one significant theme that emerged from multiple participants over multiple interviews: integration issues among simultaneously evolving systems. Because so many different systems are evolving as part of the company's overall modernization effort, integration issues among those systems arise continually. Architects have to understand not only the future state of their own system, but also that of the other systems with which their system interoperates. One architect working on the point-of-sale evolution studied in section 4.3 explained:

It's not enough to say I'm going to upgrade my point-of-sale system, which is a huge and daunting task in and of itself here at Costco. When you talk about all of the dependencies and all of the different components that are going to be interacting with point-of-sale, now you've got a unfathomable challenge in front of you, and it really takes a lot of focus to keep from getting into trouble.

After communication and coordination issues, the next most frequently occurring category was “Dealing with rapid change and anticipating the future.” This, too, is an umbrella category that encompasses a few different meanings: planning ahead in the face of rapid change, seeing the long-term ramifications of decisions, estimating future effort, adapting industry advances to the company, and realizing a long-term vision. What all these topics have in common is that they describe challenges involved in making good decisions in a rapidly changing environment. The high frequency with which this category appeared in the data suggests that this is viewed as a particularly important challenge.

Indeed, participants were remarkably consistent in how they described dealing with rapid change as a challenge of architecture evolution. One participant said:

You really have to have that ability to shift gears and change directions quickly, because the technology landscape changes so fast, and when you're on a project that's going to run three to five years, changes are inevitable.

Another architect made the same point, emphasizing the rapid pace of change in the retail industry:

Two years in the retail industry from what's happening is a long time, but two years from trying to upgrade a system for a retailer the size of Costco is actually a fairly quick time. You're making decisions that say: In two years, what are we going to be doing? And that used to be OK twenty-five years ago; "OK, yeah, we're scanning now" was about the biggest change you had. Now the changes are so massive that it's difficult to say what is going to be happening in two years. Two years ago, MCX no one had talked about.

"MCX" here refers to Merchant Customer Exchange, a recent joint venture by a consortium of retailers to create a mobile payment platform to compete with other offerings such as Google Wallet and Isis. A third interviewee used a more relatable example to make a similar point:

This has been a very explosive growth for IT in the last ten years. When I started ten years ago, there was no such thing as Facebook and Twitter. People didn't use web-based e-mail programs; it was all Outlook and that type of stuff.

Thus, there seems to be broad agreement by participants that dealing with fast-paced change is a particularly important challenge.

The third most frequent category of challenge was "Business justification," which captures challenges in justifying architectural decisions in business terms, ensuring that software systems support business goals, understanding business needs, and so on. One participant characterized executive-level support for architecture as the biggest single issue that the industry faces with respect to architecture evolution, while emphasizing that this is not such a big problem at Costco:

I think in terms of architecture, what I've seen the biggest challenges are executive buy-in on the architecture function in general, because when you throw architects in at the beginning (goodness, I hope it's at the beginning) of a project, you're adding time to it, and when you've got a request from the business, or you're comped on time to market, then there's not a whole lot of effort that you want to put into initial architecture. We've been fortunate that we've got CIO buy-in of what we're doing. But I think that's probably the largest obstacle, and when I meet with other people who run EA and solutions architectures groups from other companies, they always say that that's the largest challenge they face.

Another participant viewed the issue from a different angle, lamenting a lack of business acumen on the part of senior engineers:

I observed back around 2000 that really senior technologists did not have that appreciation of the business. They could write a system for you in their sleep, but they really didn't understand why, and they didn't really understand, again, the far-reaching ramifications into the business, into the bottom line, into the P&L and everything that goes with business.

"Dealing with legacy systems" is a reasonably self-explanatory category. A large complex of legacy systems has been built up over a period of decades, and managing and evolving these systems presents significant challenges, as was explained to me in an interview with a pair of architects:

**Participant 1:** One of the sad things that Costco came to realize is that the applications that were built [...] here over time—over the twenty-five or thirty years or so that the mid-range systems were here—really grew organically, and there wasn't comprehensive architectural thinking going on in the early days, to the point where you had a big ball of string. And now it's so brittle that the simplest changes are taking months and months, and it's just not really good.

**Participant 2:** Part of the problem is understanding the existing architecture that exists, even if it wasn't architected by design.

**P1:** Yeah, and I would say that's probably optimistic—understanding the architecture. Really there was no architecture. [...] We got an analysis tool that we put onto our system and we had it do a drawing of all of the components and the relationships, and honestly it looked like a ball of yarn—the big red dots all around the outside with lines going everywhere. We said, well, that's the problem right there.

This can be viewed quite naturally as an example of technical debt. In failing to manage the overall design of its early systems as they were built, maintained, and evolved over the years, the company has been accruing a kind of architectural debt—debt which is now coming due as the company struggles to make major changes to these systems.

The fifth category, “Scalability, reliability, and performance,” captures challenges in maintaining desired architectural qualities when evolving large-scale software systems. It should be noted that all eight occurrences of this category came from just two interviews, both of which were on the topic of the point-of-sale system. This raises the question of why scaling challenges weren't mentioned in other interviews. One possibility is that scalability, reliability, and performance issues are more important for the point-of-sale system than for other software systems. However, there is no reason to expect this is the case. A more plausible explanation is that these quality concerns are mostly relevant in the context of specific systems, and the point-of-sale system was the only system I investigated in sufficient depth to bring these challenges to light.

A number of specific challenges were mentioned within this category: managing the practical issues involved in deploying a major change to an outwardly facing system such as point-of-sale, achieving adequate performance from a new communication technology, assuring system reliability in the face of Internet outages, adequately testing the point-of-sale system, achieving the network bandwidth necessary for certain features, and selecting off-the-shelf software that can scale to the company's needs.

Under the next category, “Managing the expectations and experience of users and stakeholders,” participants mentioned a number of specific challenges: minimizing the degree to which software changes impact employees in the warehouse, dealing with unrealistic demands from users and unrealistic expectations, making progress visible to stakeholders, dealing with user dissatisfaction with delivered products, and avoiding negative effects on customer perceptions. On that last point, one architect explained (when asked whether they had identified any specific areas of risk or uncertainty at the outset of the point-of-sale evolution):

I think the risks are far more changing what Costco does and how that might affect the warehouse of the future, having a negative effect on how we do stuff [...]. We have a very simple, as we said, point-of-sale system, where things get swiped and go through the front end. Yet we still have long lines, and there's a perception from members that it takes a while to purchase, even though that's more perception than reality, although we know there's a lot of crossover between those. So there's kind of a risk of: OK, we're going to put this whole new system in—are we going to change our functionality such that the ability of us to keep that flow going through is going to be changed? Are we going to put stuff out there that isn't really accepted?

Complementary to the topic of managing users and stakeholders is the topic of “Managing people,” which is the next category in the list. This category captures challenges pertaining to the management of personnel. Specific challenges that were mentioned include motivating employees, dealing with employee turnover, and addressing employee uncertainty about job roles in the face of rapid technological change. We will return to some of these points in section 5.5, when we discuss the approaches that participants use to manage these challenges.

The next category, “Inadequate guidance and tool support,” was applied whenever architects expressed that they lacked good resources to turn to for information, or that available tools and approaches were insufficient. I was somewhat surprised that this category occurred relatively infrequently; it was only the eighth most frequently mentioned challenge. Indeed, I was frankly hoping that this category would occur more frequently, because that would provide stronger evidence that architects need better tools and approaches to manage evolution—a basic premise of our research.

Not only was this category relatively infrequent, but when it did occur, it did not point specifically to a need for better evolution planning tools. Rather, the challenges that architects named within this category pertained to a broad range of issues that have little to do with architectural planning. Specific challenges that were mentioned were the difficulty of getting information about legacy systems, a lack of clear guidance from the enterprise architecture team on specific initiatives, a need for better tools for sharing architectural information within the organization, challenges in disseminating architectural best practices, and the cumbersomeness of existing approaches to designing service-oriented architectures. None of these are problems that our work is tailored to address.

Of course, the fact that architects did not expressly cite a need for better evolution planning tools as a challenge does not mean that such tools cannot be helpful to architects. But it does demand due consideration. I will postpone resolution of this point to a more detailed discussion in section 6.

The “Cultural challenges” category is very closely related to the “Causes of problems: Cultural inertia” category that was described in section 5.2. Recall that the difference between “Causes of problems” and “Challenges” is that causes of problems result in some adverse effect that is specifically named, while challenges are described in more general terms. Thus, “Challenges: Cultural challenges” and “Causes of problems: Cultural inertia” are very closely related indeed; the only difference between them is whether the named cultural issue was the cause of an explicitly articulated adverse consequence. It is not surprising, therefore, that the same cultural issues that were named as causes of problems in section 5.2 were also named as general cultural challenges here: the conflict between a long-standing culture of building homegrown software and a new strategy of buying off-the-shelf packages, and the tension between a culture of consensus and the need to make rapid decisions.

The “Lack of expertise or experience” category arose mostly in reference to specific technologies or tools that engineers or architects lacked experience with. For example, one participant mentioned engineers’ lack of experience with identity management technologies as a challenge; another mentioned architects’ lack of familiarity with a particular tool for managing architectural information.

“Divergent understandings of architecture” captures challenges arising from conflicts in different people’s conceptions of the role of architecture. One participant explained this challenge by way of analogy:

I think part of the challenge, especially within Costco, is getting everyone to understand what architecture’s trying to do. If you correlate enterprise architecture with an architect building a building, we’re not saying, “Go buy a Kohler toilet.” (If you know Kohler, they’re a brand that makes toilets.) [...] We’re saying, “In this building, there needs to be a toilet.” So solution architects, business, based on your capabilities, pick the best toilet. But we need a toilet. [...] So it’s getting everyone to understand that we’re not here to say, “Here’s what you need to do;

	Frequency (coding units)	Frequency (interviews)	Frequency (participants)
Communication and coordination practices	25	7	6
Organizational strategies and structures	17	5	5
Phased development	15	4	3
Drawing from expert sources	13	5	4
Rules of thumb and informal strategies	13	5	4
Tools	12	5	5
Anticipating the future	8	5	4
Prototypes, pilots, etc.	8	4	4
Training	8	4	4
Business architecture	8	3	3
Formal approaches	6	4	4
Experience and intuition	6	3	2
Process	4	3	3
Industry practices	3	3	3
Considering alternatives	1	1	1

**Table 5:** Stated approaches for architecture evolution

go do it,” and then step back and wash our hands. We’re here to say, [...] “Here’s how we do stuff at Costco. [...] Here’s how we’re going to do services. Here’s how applications are going to interact. So when you’re planning for your solutions ahead, use those.” We’re not saying, “OK, we’re going to do a web app that only supports Android, and we’re going to have Java *x.y.z* as our version, or we’re always going to be Windows 8, or we’re going to be *x*.” That’s not what we’re trying to say. [...] So getting that understanding across has been kind of the biggest challenge.

The final two categories, “Managing scope and cost” and “Dealing with surprises,” each had three occurrences.

### 5.5. Approaches

This final section of findings from content analysis 1 answers the question: what approaches do architects in a real-world software organization currently use to manage major evolution efforts?

At the beginning of the study, I conjectured that architects currently have few practical approaches for managing major evolutions, and that I would find that architects therefore have little to say about the techniques they use to plan and carry out evolution. But in fact “Approaches” turned out to be the most frequently occurring of all the major categories. Not only that, but the coding units in this category were longer, on average, than the coding units in any of the others. Thus, architects frequently and at great length about the methods they use to manage evolution.

As with the other categories, the topic of this category was construed quite broadly for coding purposes. That is, the “Approaches” category includes not only concrete techniques

such as tools and formal methods, but also very informal strategies and rules of thumb. Thus, the mere fact that the “Approaches” category occurred frequently does not, by itself, contradict the hypothesis that architects lack formal techniques for managing software evolution. That question can be addressed only once we have examined the subcategories, which we will now proceed to do. The frequencies are given in table 5.

As usual, we begin with the most frequent category. And just as “Communication, coordination, and integration challenges” was the most frequently occurring subcategory of “Challenges,” so is “Communication and coordination practices” the most frequent subcategory of “Approaches”—indeed, the most frequently occurring subcategory in the entire coding frame. That is, the architects in this study spent more time talking about communication, coordination, and integration than about any other topic. This suggests that communication and coordination issues are tremendously important in managing architecture evolution. One participant explained that communication is a very important part of an architect’s role:

A lot of this is human dynamics. That’s really the glue that holds it all together. It doesn’t matter how much process you have, it’s the business relationships and your ability as an architect to navigate the political waters. It’s going outside the perfect world of academic process and saying, I need to go have these conversations at these times to get these decisions made by the stakeholders that actually have the power and ability to make them. That’s where the art meets the science. The science says we can take a TOGAF framework or an ITIL framework or a Zachman framework and you have these deliverables by these people on these days, but the art comes in knowing how to get those people to sign off on them at those times, remove all the race conditions, because the reality is: things don’t go as smooth as you’d like them to in the order you’d like them to.

Another architect made a similar point:

A lot of this is just doing what works for you. It’s really more about communication than architecting systems. That collaboration aspect, I think, is absolutely paramount to the success of an architect. You have to talk to lots of people all the time. The drawings don’t have to be that precise, as long as they communicate the information.

Participants mentioned a variety of communication and coordination practices that can facilitate architecture evolution, including prioritization of projects, appropriate use of architectural documentation, coordination on dates of delivery for internal services, organizational standardization on certain technologies, alignment of parallel efforts, providing guidance on best practices within the organization, understanding political implications of architectural decisions, and maintaining a feedback loop between architects and developers.

The second subcategory, “Organizational strategies and structures,” captures architects’ explanations of the organizational structure of the IT department and how it supports architectural efforts. Because I have already described the organizational structure in detail in section 2.2, I will not say more about this here.

There were 15 occurrences of the category “Phased development,” which captures situations in which architects described breaking projects down into manageable phases or stages. As one architect explained, planning a large, complex evolution in phases is often a practical necessity: “you can’t just Big Bang something like that.” This category is significant because it describes the same basic approach that we have taken in our research: breaking down a large evolution into smaller steps that can be more easily understood and managed (or conversely, building up small operations into a large evolution). The high frequency with which architects described this technique in the interviews suggests that in this respect, architects are already using the kind of stage-based reasoning that our approach is based on. This is an encouraging sign for the realism and practicality of our approach.



Under the next category, “Drawing from expert sources,” participants identified a range of people that they relied on as sources of expertise on particular technologies. One source of expertise that has become important for the company in recent years is consultants. As the company has increased its reliance on off-the-shelf technologies as opposed to homegrown software, it has had to increase its reliance on consultants with experience in those technologies at the same time. But this category also includes other kinds of expert sources, including employees within the company who are experts in particular technologies, as well as books providing guidance on unfamiliar technologies or architectural styles.

The next category, “Rules of thumb and informal strategies,” was used to code the informal rules and guidelines that architects use to help make decisions (except when such rules and guidelines would fit better in another category). The following is a list of such strategies and rules of thumb that were mentioned by interviewees as guiding their decisions:

- Design systems with simplicity as a goal; build a solid, simple foundation that can be extended later.
- Anticipate possible future changes and design in such a way that those changes are easy.
- When possible, prefer to rent (storage space, e-mail service, etc.) rather than own.
- Minimize project risk by making sure the plans for future systems are solid before you depend on them in a project.
- Consistency across projects is important. Create and adhere to enterprise standards.
- Adopt open standards when possible. But when an open standard is very new and immature, fall back on frameworks that are established in the market as an interim solution.
- When the company doesn’t yet have accepted standards for a technology, try to harvest from what successful teams within the company have already done; take the artifacts from the successful project and elevate them into enterprise standards.

Some of these are strategies that the company has adopted recently (e.g., renting rather than owning reflects the company’s adoption of cloud-computing services); others are much more generally applicable.

The next category, “Tools,” is an interesting one because of its surprisingly high frequency. I expected to find that architects have few tools to help them manage evolutions. But when I asked them what, if any, tools they use to plan evolution, they responded by describing a variety of tools, making this a rather frequently occurring category. More specifically, the following kinds of tools were mentioned during the interviews:

- An architectural analysis tool for mapping relationships among components in an existing system
- A tool for managing architecture artifacts across projects
- A tool for transforming source code in legacy programming languages into Java
- A software process management tool
- A tool for building customizable project templates
- A service component used to aid in service configuration
- A code generation tool that transforms UML diagrams into WSDL descriptions and Java interfaces

This is quite a broad array of tools. But none of them are actually *architecture evolution tools* in the sense in which we mean it. Many of them have nothing to do with architecture, and many of them have nothing to do with evolution. But they are all tools that may be useful in planning or carrying out an architecture evolution. This suggests that we should perhaps be more cautious when motivating our research with claims such as, “At present,

software architects have few tools to help them plan and carry out these kinds of architecture evolution.” In fact, architects *do* have tools that help them to plan and carry out evolution; what they don’t have is tools to help them develop an end-to-end plan for rearchitecting a system.

The category “Anticipating the future” was applied whenever an interviewee described anticipating future developments as an approach for better planning evolution. One participant said that he considered consideration of long-term ramifications to be a key quality that separates skilled architects from novice architects: “One of the things, I think, good architects do well is they can see the far-reaching implications of decisions that get made.”

“Prototypes, pilots, etc.” encompasses a number of different concepts. One architect explained the difference between two of these concepts: proofs of concept and pilots.

There’s two different concepts. One’s called a proof of concept, and the other one’s a pilot. Proof of concepts don’t necessarily go anywhere. Right now, we have a proof of concept running in Chandler, Arizona, of what we’re calling Personal Shopper. Costco provides a device, you go in, scan your member card, and you pull the device off, scan all your items as you shop. So you still shop the same way, other than before you put it in your cart, you scan it. When you get done, you go up to the front, hand the device to a attendant, they close that, you scan your member card at a payment station, it pulls that order in, you pay and go. So now you’re eliminated from the scans. That’s a proof of concept. All we’re doing is proving that that model works at Costco. If we decide to go forward with it, it won’t necessarily be with those devices, it won’t necessarily be with that company, but that concept of a retailer-provided device to scan items has now been proven for Costco.

[...]

A pilot is the initial installation of what you believe is going to be your final product. It might not be the final, but it’s what that product is.

Aside from proofs of concept and pilots, participants mentioned a number of other concepts that are similar in character: reference applications to serve as examples of best practices, project templates that can be customized for use in real projects, and case studies and mock applications used to gain experience with new technologies. Speaking generally, it would appear that doing some kind of trial run to gear up before launching a real project or product is viewed as an important risk mitigation strategy.

“Training” has become increasingly important as the company has adopted new technologies in the course of its modernization efforts. Training is particularly important for Costco because the company, as a rule, doesn’t lay off unneeded employees—an exceptional policy for a major retailer. Thus, personnel with unneeded skills must, ipso facto, be retrained on new technology. One participant explained how this can benefit both the workers and the company, using the example of network administrators who had been trained on a productivity suite that was being retired:

What are you going to do? At first they were very resistant. But if you looked at what their day-to-day jobs were—I don’t know if you’ve ever done any network admin—it’s kind of grunt work. Oh, [...] my profile’s corrupted. Or please reset this or whatever. It’s low-end work. [...] So what we had to do is retrain or refigure those guys to be more collaboration experts and be a little bit more directional in focus instead of just low-end network admins. So in effect what happens is they become (which I think is one of the hottest things in the industry right now) vendor managers or software-as-a-service vendor partners, which is actually kind of a jump up than being the [system] admin, of which there’s fifty million out there.

“Business architecture” was mentioned by multiple participants as a significant topic. One of the CoEs within the architecture organization is a business architecture CoE, led by a dedicated business architect. Business architecture encompasses a range of practices, none of them particularly relevant to evolution specifically, so I do not discuss this topic further here.

The next category, “Formal approaches” is an interesting one. As with “Tools,” I expected there would be few instances of this category, conjecturing that architects lack tools and formal approaches for evolution. But also as with “Tools,” it turned out that architects did name a number of approaches—just not ones that were specific to evolution. Specifically, the Rational Unified Process (RUP) and TOGAF were both mentioned multiple times by different architects. Participants tended to be fairly critical of RUP, describing it as too heavyweight to meet the company’s needs.

The category “Experience and intuition” was applied whenever participants described relying on their own seasoned judgment in making architectural decisions. Several times, interviewees described the skills that they had gained through years of experience, or explained how experienced architects can avoid mistakes that junior architects are more likely to make.

The “Process” category was applied to all descriptions of software process. One participant gave a particularly detailed description of how the architect’s role permeates the entire software process, from initial conception to final delivery and beyond, which I reproduce in abridged form here:

Starting out right from day 1: Do we have a business case? Do we know what we’re building, why we’re building, who we’re building for? And is there an actual return on investment, or is it just a perceived one? Do the cost/benefit analyses actually add up? [...]

From there, once you’ve proved that yes, this really is some value here, then do a technical feasibility test. Can you even do this? Are we talking about teleportation, or are we talking about building a system based on known commodities?

Once you’ve technically determined it’s feasible, then you start fleshing out the requirements. Requirements isn’t just going to the stakeholder and saying, hey, what do you want us to build? It’s working with all the different teams that *may* be impacted—all the dotted lines. [...]

Once you flesh those out, then your job as solution architect is to go to (in terms of Costco) the high-level design. Let’s make sure we figure out what are all the components, what are all the integration points, what are the protocols we use to communicate across these things. [...]

Once you get through design, in Costco’s world, we would then hand it off to a lead developer or implementor, depending on what type of solution, for macro and micro design. In between that step, though, we go through Enterprise Architecture Review Board, which validates the highest level: Did what the solution architect designed make sense? Does it fit into the long-term strategy where Costco’s going? [...] So macro and micro happens.

[...] then you make sure the solution is implemented as intended on budget and on cost. A lot of people say, OK, I’m designed, I’m done. The trick to architecture is: if you just say what to build and you walk away and never look back, you never get the as-intended piece, and just because you build something doesn’t mean there’s actually any value. [...] So then you’re shepherding, especially the solution architect, through that implementation phase, through macro/micro design, actually into coding or the infrastructure spin-up, establishing the protocols, making sure the parts of the network are talking, making sure training is done, making sure it’s handed off to support, and shepherding it all the way through deployment. It’s not just, OK, the product is delivered. Well, no, it’s not delivered until someone actually uses the system. Making sure the actual end users have

something, whatever that may be—hardware, software, process.

And then you're still not done yet. You now need to go back and triage. What are the lessons learned from this process? [...] Take the lessons learned, and feed that back to the other projects, because remember the trick to architecture. It's not do once and forget. It's do once and take the patterns and antipatterns and re leverage them into the next solution [...]

The “Industry practices” category is similar to “Drawing from expert sources,” in the sense that it describes situations in which architects seek guidance from others to make better decisions. In this case, however, that guidance comes from other retailers instead of from the company's engineers, consultants, or vendors. Three different participants mentioned ways in which industry trends have informed their engineering efforts.

The final category, “Considering alternatives,” was intended to capture situations in which architects described consideration of alternative plans as a strategy in architecture evolution. However, the category had only one occurrence:

We have to always, I think, have plan A and plan B. Going into a lot of the work that we're doing now, I say, well, I would love to have this SOA service environment, but understanding that that's still very much in its infancy, here I have plan B, which is let's just continue on with what we know using legacy systems and legacy technologies.

Although consideration of alternatives did not emerge as a significant theme of the interviews, there is other evidence to suggest that alternatives are often considered in the course of architectural evolution efforts. Specifically, the architectural documentation I examined often included explicit and fairly detailed discussion of alternative possibilities and justification for the selected option.

## 6. Conclusions

### 6.1. *Answers to the research questions*

We now return to the research questions of section 1.3 and document explicitly how they are answered by this case study.

1. **How do practicing architects in a real-world software organization plan and reason about evolution?** This question was addressed by content analysis 1, particularly the “Approaches” category of the content analysis (and to a lesser extent the “Evolution motives” category, which describes the significant catalysts that architects mentioned as driving evolution). We explored the results in detail in section 5.5, but to summarize, we found that architects have a remarkably wide range of strategies and approaches that they use to manage architecture evolution. The most prominent category of approaches in the interview data was communication and coordination practices, suggesting that issues of communication and coordination are very important in managing architecture evolution. Other frequently mentioned approaches for managing architecture evolution included the use of organizational strategies and structures, breaking down software development into manageable phases, taking advantage of expert sources such as consultants and books, and using various rules of thumb and informal strategies.

A surprising result was that architects said they already have useful tools for managing architecture evolution. These tools, however, are quite different from the kind of tool envisioned in our work. Many of the tools they mentioned were not specifically architectural at all—for example, software process management tools or service configuration tools.

Another interesting result is that architects are already reasoning about evolution in terms of development phases, with explicit consideration of evolution alternatives. This suggests that our approach, which is based on modeling potential evolution paths comprising sequences of discrete evolution states, is compatible with the way architects already think about evolution.

- 2. What difficulties do practicing architects face in planning and carrying out evolution?** This research question was also addressed by content analysis 1—particularly by the “Causes of problems,” “Consequences,” and “Challenges” categories. We described the results in detail in sections 5.2, 5.3, and 5.4. In this summary, I will focus on the “Challenges” category because it occurred far more frequently than “Causes of problems” and “Consequences” and therefore provides a broader sample of difficulties and more robust information on their relevant frequencies.

By far the most commonly mentioned category of challenges was “Communication, coordination, and integration challenges.” This corresponds neatly to the result mentioned above that “Communication and coordination practices” was the most frequent subcategory of “Approaches.” This reinforces the conclusion that communication and coordination issues are extraordinarily important in managing architecture evolution in practice. Specific challenges mentioned within this subcategory include documentation challenges, challenges of architecture conformance, and challenges in coordinating with an off-site development team, among others.

Other significant classes of challenges identified by the content analysis include anticipating future developments, justifying decisions in business terms, dealing with legacy systems, assuring architectural qualities such as scalability and reliability, and managing stakeholder expectations.

Understanding the challenges that architects face helps us to position our research with respect to the current state of architectural practice. Our work has at least a small role to play in many of these challenges. For example, one of the goals of our work is to support reasoning about integration activities during the course of an evolution, which might help forestall certain integration problems. But clearly there are some categories, such as “Managing people” and “Cultural challenges,” to which our work is irrelevant.

- 3. How well can our modeling framework capture the concerns that arise in a real-world architecture evolution?** This research question was addressed by content analysis 2 and the evolution model that was constructed based on it. The construction of the evolution model is described in detail in section 4.3. The great majority of the operators, constraints, and evaluation functions identified by the content analysis could be modeled using our approach, although a few of them were not suitable for modeling, often because they were too low-level (pertaining to implementation details rather than architecture) or too high-level (encompassing a broad range of specific considerations that were not understood in sufficient detail to model).

These results are generally encouraging, although there were a few minor surprises. However, understanding the significance of these results requires us to critically assess their reliability and validity. We turn to this topic in the following section.

## 6.2. *Quality considerations*

There are two broad quality dimensions that are of paramount importance in evaluating qualitative research: validity and reliability. In general terms, an instrument is said to be *valid* to the extent that it captures what it sets out to capture, and *reliable* to the extent that it yields data that is free of error. In the following subsections, we will evaluate and discuss the reliability and validity of this case study.

### 6.2.1 Reliability

The most well-known treatment of reliability and its application to content analysis is that of Krippendorff [48]. Krippendorff distinguishes among three kinds of reliability:

- **Stability**, the degree to which repeated applications of the method will produce the same result
- **Reproducibility**, the degree to which application of the method by other analysts working under different conditions would yield the same result
- **Accuracy**, the degree to which a method conforms to a standard

These three types are listed above in order of increasing strength; that is, reproducibility is a stronger notion of reliability than stability is, and accuracy is the strongest of the three. However, accuracy is only occasionally relevant in content analysis, because it presumes that there is some preexisting gold standard to use as a benchmark, such as judgments by a panel of experienced content analysts. However, in many content analyses, especially those exploring new domains or topics, there is no such gold standard to use. That is certainly the case here. In addition, as Krippendorff [48, p. 272] explains, accuracy is a problematic concept for content analysis even in principle: “Because interpretations can be compared only with other interpretations, attempts to measure accuracy presuppose the privileging of some interpretations over others, and this puts any claims regarding precision or accuracy on epistemologically shaky grounds.” Thus, the two main forms of reliability which are normally relevant to content analysis are stability and reproducibility.

We begin with stability. The most direct way of assessing the stability of an instrument is through the use of a test-retest procedure, in which the instrument is reapplied to the same data to see if the same results emerge. In fact, stability is sometimes called “test-retest reliability,” or simply “repeatability.” I incorporated a test-retest procedure into the design of the content analysis. For each of the two content analysis, I conducted a second round of coding at least 17 days after the initial coding. (As mentioned in section 4.2.7, Schreier [77, p. 146] recommends that at least 10–14 days elapse between successive codings by the same researcher.) Once two rounds of coding have been completed, the results can be compared to evaluate the degree to which they agree. This provides a measure of intrarater agreement (also known by other similar names such as intracoder reliability, intra-observer variability, etc.).

Intrarater agreement can be quantified using the same metrics that are used to measure interrater agreement in studies with multiple coders. There are a large number of such metrics [31, ch. 18; 69], but there are four that are most popular in content analysis:

- The simplest and most obvious metric for quantifying intrarater or interrater agreement is percent agreement, which is simply the percentage of coding units that were categorized the same way in both codings. For example, in our content analysis 1, there were 306 coding units total across the eight interviews, of which 280 were categorized the same way in the initial coding and the recoding; thus, the percent agreement was  $280/306 = 91.5\%$ . This metric is intuitive and easy to calculate, but it is also problematic. In particular, it does not account for agreement that would occur merely by chance. Thus, the metric is biased in favor of coding frames with a small number of categories, since by chance alone, there would be a higher rate of agreement on a two-category coding frame than on a forty-category coding frame.
- To address this problem, Scott, in a 1955 paper titled “Reliability of Content Analysis: The Case of Nominal Scale Coding” [79], introduced a new interrater agreement measure called  $\pi$ , which corrects for both the number of categories in the coding frame and

the frequency with which they are used. It is defined by

$$\pi = \frac{P_o - P_e}{1 - P_e},$$

where  $P_o$  is the percent agreement as defined above and  $P_e$  is the percent agreement to be expected on the basis of chance. Informally speaking,  $\pi$  measures the degree to which interrater agreement exceeds the agreement that would be expected by chance. A  $\pi$  value of 1 indicates complete agreement, a zero value indicates that the agreement is no better than chance, and a negative value indicates agreement worse than chance. The expected agreement  $P_e$  is calculated by  $P_e = \sum_i p_i^2$ , where  $p_i$  is the proportion of responses that fall in category  $i$  and the sum is taken over all categories in the coding frame.

- A very similar metric was introduced by Cohen [21] in 1960. Like Scott's  $\pi$ , Cohen's  $\kappa$  is defined by the formula  $(P_o - P_e)/(1 - P_e)$ , but  $P_e$ , the percent agreement expected by chance, is calculated differently. Specifically, Cohen takes  $P_e = \sum_i p'_i p''_i$ , where  $p'_i$  is the proportion of coder 1's responses that fall in category  $i$  and  $p''_i$  is the proportion of coder 2's responses that fall in category  $i$ , so  $p'_i p''_i$  represents the probability that both coders would assign category  $i$  by chance.
- Perhaps the most widely accepted reliability coefficient in content analysis is Krippendorff's  $\alpha$ , first introduced by Krippendorff in 1970 [46] and subsequently developed over time [48, p. 278]. Like Scott's  $\pi$  and Cohen's  $\kappa$ , Krippendorff's  $\alpha$  can be defined as  $(P_o - P_e)/(1 - P_e)$  and thus also ranges from 0 (zero reliability, or chance agreement) to 1 (perfect agreement), with negative values possible for systematic disagreement. Again, the difference is in the calculation of  $P_e$ .

Krippendorff's  $\alpha$  is the most flexible of the three reliability coefficients, able to accommodate arbitrarily many coders; multiple types of data (i.e., not just nominal data but also ordinal, interval, etc.); and missing data. But in the simple case of nominal data with two coders (or one coder who coded the data twice) and no missing values,  $P_e$  may be calculated by

$$P_e = \sum_i \frac{n_i(n_i - 1)}{n(n - 1)},$$

where  $n$  is the total number of responses and  $n_i$  is the number of responses that fall in category  $i$ .

The differences among the use of these metrics are subtle, and some methodologists have expressed strong opinions about their appropriateness or inappropriateness for various uses [47]. For our purposes, the distinctions among them are unimportant, since  $\pi$ ,  $\kappa$ , and  $\alpha$  are all approximately equal for our data. As Krippendorff [47, p. 423] explains,  $\pi$  and  $\alpha$  converge as the sample size grows, and  $\pi$  and  $\kappa$  converge when coders agree well on the marginal distributions of categories. In this study, the sample size is large (there are many coding units), and both codings were carried out by a single person, so it is unsurprising that the marginal distributions of the categories are similar.

Our intracoder reliability figures appear in table 6. These measures were calculated with ReCal2, an online utility for calculating reliability coefficients [33], and a subset of the measures was verified by hand in Microsoft Excel. For content analysis 1, only one set of measures is shown, since each coding unit could be assigned exactly one category from anywhere in the coding frame. In content analysis 1, there were 306 coding units and 45 categories (i.e., the total number of subcategories that appear in appendix B.1.2, excluding the residual categories "mot-?", "cau-?", "con-?", "cha-?", and "app-?", which were never used). For content analysis 2, on the other hand, three sets of reliability measures are shown, one for each main category. This is because each coding unit in content analysis 2 could be assigned

	% agreement	$\pi$	$\kappa$	$\alpha$
Content analysis 1	91.5%	0.912	0.912	0.912
Content analysis 2: “Classification”	94.7%	0.936	0.936	0.936
Content analysis 2: “Presence in initial architecture”	91.8%	0.874	0.874	0.874
Content analysis 2: “Presence in target architecture”	93.8%	0.900	0.900	0.900

**Table 6:** Intracoder reliability measures: percent agreement, Scott’s  $\pi$ , Cohen’s  $\kappa$ , Krippendorff’s  $\alpha$

three categories—one subcategory of each main category. There were 208 coding units in content analysis 2. There were ten subcategories of the “Classification” category (including the residual category “c-?,” which *was* occasionally applied, unlike the residual categories in content analysis 1). There were three effective subcategories of the “Presence in initial architecture” category, and likewise for “Presence in target architecture”; in addition to the two that appear in appendix B.2.2 (“Present” and “Absent”), “Not coded” was treated as a third, implicit subcategory for the purpose of the reliability analysis. (Per the coding guide, coding units identified as corresponding to software elements such as components and connectors were to be coded as present or absent in the initial and target architectures, while coding units identified as corresponding to non-software elements such as constraints and containment relations were not to be so coded. However, it turns out that I failed to adhere to this protocol consistently. In particular, I often incorrectly treated containment relations as though they were software elements, and coded them for their presence or absence in the initial and target architectures. In fact, this particular coding error accounted for a large portion of the disagreements in content analysis 2, depressing the reliability figures in table 6.)

All twelve of the reliability coefficients in table 6 are between 0.87 and 0.94. But how are we to interpret these figures? Well, there are no universally accepted thresholds, but a number of methodologists have put forth guidelines. Krippendorff [48, p. 325] recommends  $\alpha = 0.8$  as a reasonable cutoff, with figures as low as  $\alpha = 0.667$  acceptable when drawing tentative conclusions and in cases where reliability is less important. Another well-known recommendation is that of Fleiss et al. [31, p. 604], who describe  $\kappa > 0.75$  as indicating “excellent” agreement,  $0.40 \leq \kappa \leq 0.75$  as indicating “fair to good” agreement, and  $\kappa < 0.40$  as “poor.” Various other rules of thumb have been proposed [39, § 6.2.1; 66, p. 143].

There are a couple of problems with directly applying any of these well-known and widely used guidelines here. First, these guidelines are intended for assessing interrater, not intrarater, reliability. It seems reasonable to hold intrarater reliability to a higher standard than interrater reliability, since one is more likely to agree with oneself (even one’s past self) than with another person.

Second, these guidelines are intended primarily for use in quantitative research, particularly quantitative content analysis. Arguably different standards should be applied to qualitative content analysis. Schreier [77, p. 173] writes:

In qualitative research, you will typically be dealing with meaning that requires a certain amount of interpretation. Also, coding frames in [qualitative content analysis] are often extensive, containing a large number of categories. In interpreting your coefficients of agreement, you should therefore never use guidelines from quantitative research as a cut-off criterion [...], as in: ‘Oh, my kappa coefficient is below 0.40—it looks like I will have to scrap these categories’. Instead, what looks like a low coefficient should make you take another closer look both at



your material and your coding frame. Perhaps, considering your material and the number of your categories, a comparatively low coefficient of agreement is acceptable—this is simply the best you can do.

Schreier may be incorrect in suggesting that a large number of categories justifies laxer standards for reliability coefficients, since chance-adjusted coefficients such as  $\pi$ ,  $\kappa$ , and  $\alpha$  already account for the number of categories. However, the degree of interpretation required to apply a coding frame is a very good reason to treat qualitative content analysis differently from quantitative content analysis. Even Neuendorf [66, p. 146], who is dismissive of qualitative content analysis generally, argues that content analyses that require significant interpretation should be subject to more relaxed reliability standards: “Objectivity is a much tougher criterion to achieve with latent than with manifest variables, and for this reason, we expect variables measuring latent content to receive generally lower reliability scores.”

Although this discussion has not yielded any precise thresholds to anchor our interpretation of the reliability figures in table 6, it should by this point be clear that coefficients of 0.87 to 0.94 are fairly high by almost any standard. Even considering the fact that these are coefficients of intrarater, not interrater, agreement, it seems reasonable to conclude that we have adequately demonstrated stability.

This leaves the other main reliability criterion relevant to content analysis: reproducibility. Most commonly, reproducibility is measured through interrater agreement. In this sense, reproducibility is synonymous with terms such as “interrater reliability” and “intersubjectivity.” Of course, with a single-coder study design, interrater reliability cannot be assessed. Schreier [77, p. 198] recommends that in a qualitative content analysis in which a single researcher is coding the data alone, a second round of coding (at least 10–14 days after the main coding) should be used as a substitute for using multiple coders to assess reliability, implying that intrarater agreement may be used as a substitute for interrater agreement in single-coder content analyses and that stability alone is a sufficient indicator of reliability in such cases. Ritsert [72, pp. 70–71] takes a similar position: “In an analysis by an individual, the important possibility of intersubjective agreement as to the coding process is precluded, but the methods of ‘split half’ or ‘test retest’ can still provide an individual with information on the consistency and reliability of his judgments.”<sup>3</sup>

On the other hand, Krippendorff [48, p. 272] argues that stability “is too weak to serve as a reliability measure in content analysis. It cannot respond to individually stable idiosyncrasies, prejudices, ideological commitments, closed-mindedness, or consistent misinterpretations of given coding instructions and texts.” On similar grounds, Neuendorf [66, p. 142] writes that “at least two coders need to participate in any human-coding content analysis.”

It is thus fair to say that there is no consensus in the content analysis literature about the adequacy of stability as a measure of reliability, and it would be misleading to assert that a single-coder content analysis design with stability as the primary reliability measure is uncontentious. However, there are certainly methodologists who do consider it an acceptable choice, and indeed content analyses with single-coder designs are common in the literature.

However, even in the absence of intercoder agreement as a reliability measure, reproducibility and intersubjectivity remain important goals in principle. A content analysis should be reproducible at least in principle, even if no test of interrater agreement was carried out in practice. Fortunately, there are other ways of getting at this quality in the absence of multiple coders. A particularly helpful perspective comes from Steinke [81, § 3], who argues that qualitative research needs to be evaluated differently from quantitative research: “For qualitative research, unlike quantitative studies, the requirement of inter-subject *verifiability* cannot be applied. [...] What is appropriate in qualitative research is the requirement to produce an inter-subjective *comprehensibility* of the research process on the basis of which

<sup>3</sup>“Zwar fällt bei einer Auswertung durch einen einzelnen die bedeutsame Möglichkeit intersubjektiver Verständigung über den Codierprozeß weg, aber die Methoden des ‘split half’ oder des ‘test retest’ können auch einem einzelnen Auskunft über Konsistenz und Verlässlichkeit seiner Urteile verschaffen.” Translation from German mine.

an evaluation of results can take place.” Steinke goes on to suggest that this intersubjective comprehensibility can be demonstrated in three ways. First, and most importantly, the research process should be thoroughly documented so that “an external public is given the opportunity to follow the investigation step by step and to evaluate the research process.” Second, interpretations can be cross-checked in groups. Steinke writes that a strong form of this method is peer debriefing, “where a project is discussed with colleagues who are not working on the same project.” Third, the use of codified procedures can contribute greatly to intersubjectivity.

All three of these methods were used in abundance in this case study. If you have reached this point in this lengthy case study report, the thoroughness with which the research process was documented is not in question. In addition, in the process of planning and carrying out this study, I consulted with many colleagues in my department who were uninvolved with the research. Finally, the content analysis was conducted in accordance with a rigorously constructed, comprehensively defined coding frame, which is reproduced in its entirety in appendix B. In Steinke’s terms, the detailed description of the research procedure and the use and publication of codified procedures permits readers to evaluate the research process and hence the results on their own terms. The publication of the coding frame also serves a more direct replicability purpose: other researchers can adopt the coding frame and apply it to other data to assess the extent to which the coding frame itself is generalizable. (However, because the coding frame is heavily data-driven, it would likely need to be adapted to be effective for use with other data. For example, in this study, we identified “bridge,” “broker,” “bus,” “queue,” and “transfer” as indicators signifying connector elements, but a different data set might use different words, such as “channel,” “link,” “pipe,” and “stream.”)

Before concluding this discussion of reliability, it is useful to examine briefly how reliability is treated in typical content analyses in the literature. After all, our aim in applying content analysis in this case study is to adopt a methodology that is commonly used in other disciplines—not to advance the current state of practice in content analysis. Thus, irrespective of various methodologists’ opinions about how reliability issues should ideally be handled and reported, it makes sense to see whether our treatment of reliability is in line with common practice.

Indeed, it does not take much investigation to see that the current state of practice with respect to the treatment of reliability in content analysis is unimpressive. There have been several surveys investigating the treatment of reliability in published content analyses. The most recent of which I am aware is Lombard et al.’s 2002 content analysis of content analyses [54], which found that only 69% of content analyses indexed in *Communication Abstracts* between 1994 and 1998 included any discussion of intercoder reliability. In those that did, only four sentences, on average, were devoted to discussion and reporting of reliability. Only 41% of the studies specifically reported reliability information for one or more variables, 6% had tables with reliability information, and 2% described how the reliability was computed. These results were broadly in line with earlier surveys. The state of practice may have improved somewhat since Lombard et al.’s survey, but it is still easy to find content analyses with no discussion of reliability.

Incidentally, since Lombard et al.’s survey was itself a content analysis, Lombard et al. reported their own reliability figures: percentage agreement, Scott’s  $\pi$ , Cohen’s  $\kappa$ , and Krippendorff’s  $\alpha$ . It later turned out that due to bugs in the software they had used to calculate these figures, they had themselves reported erroneous reliability values [55]—an amusing illustration of the difficulty of getting reliability reporting right. (I verified many of my reliability figures by hand to avoid any similar embarrassment.)

In light of such results, it seems that our detailed treatment of reliability here puts us ahead of the current state of practice in typical content analyses, even in the absence of any measurement of intercoder reliability.

### 6.2.2 Validity

The situation with validity is more muddled than that with reliability. There are just three main types of reliability that are relevant to content analysis—stability, reproducibility, and accuracy—and reliability can be easily quantified via well-accepted measures of coding agreement. But there is a bewildering array of flavors of validity: internal validity, external validity, construct validity, content validity, face validity, social validity, criterion validity, instrumental validity, sampling validity, semantic validity, structural validity, functional validity, correlative validity, concurrent validity, convergent validity, discriminant validity, predictive validity, ecological validity. All of these, and others, have been described as relevant to content analysis. To evaluate the validity of our content analysis, it is necessary to untangle this knot of concepts and focus on the elements of validity most relevant to this research.

Krippendorff [48, ch. 13] identifies three broad categories of validity at the highest level: face validity, social validity, and empirical validity.

**Face validity.** Krippendorff [48, p. 330] describes face validity as “the gatekeeper for all other kinds of validity,” so it is a good place to start. Face validity is the extent to which an instrument appears on its face to measure what it purports to measure. Face validity thus appeals to a commonsensical evaluation of the plausibility of a study. Neuendorf [66, p. 115] explains:

It’s instructive to take a “WYSIWYG” (what you see is what you get) approach to face validity. If we say we’re measuring verbal aggression, then we expect to see measures of yelling, insulting, harassing, and the like. We do not expect to find measures of lying; although a negative verbal behavior, it doesn’t seem to fit the “aggression” portion of the concept.

In the context of a content analysis such as this one, face validity may be assessed by considering whether the coding procedure appears to correspond with the concept being measured—the research questions. In section 6.1, we examined how our research questions are addressed by the findings of the content analysis (and the modeling phase that followed the content analysis). In addition, the coding guide appears in appendix B. By direct examination of the coding guide, it can be seen that the coding frame appears on its face to capture the concepts it sets out to capture.

Face validity, however, is a low bar—the mere *appearance* of measuring the correct concept. Weber [87, p. 19] calls it “perhaps the weakest form of validity.” We now continue to examine other kinds of validity relevant to content analysis.

**Social validity.** The term “social validity” was introduced in 1978 by Wolf [89] in the field of applied behavior analysis. In its original sense, the term refers to how well a social intervention is accepted by those who are meant to benefit from it. This sense is relevant only to interventionist methodologies such as action research.

In content analysis, social validity has a rather different meaning. (For a discussion of how a variety of meanings of “social validity” arose subsequent to the initial introduction of the term, see Schwartz & Baer [78].) A useful definition is given by Riffe et al. [71, p. 137], who describe social validity as hinging on “the degree to which the content analysis categories created by researchers have relevance and meaning beyond an academic audience.” Social validity thus gets at the *importance* of research.

While social validity is a criterion that has seldom (if ever) been applied to software engineering research, the idea of ensuring that research is relevant to the real world is an important one. We have motivated our work here in terms of its relevance to practicing software architects. Software architects today, we have argued, face substantial challenges in planning and executing major evolutions of software systems, and our work aims to help address a subset of these challenges. Our work thus has social validity to the extent it succeeds in having relevance to software practitioners.

**Empirical validity.** Krippendorff uses “empirical validity” as an umbrella term to encompass a number of specific types of validity. In general, empirical validity gets at how well the various inferences made within a research process are supported by available evidence and established theory, and how well the results can withstand challenges based on additional data or new observations. Assessments of empirical validity are based on rational scientific considerations, as opposed to appearances (face validity) or social implications (social validity). Empirical validity is a broad concept with a number of distinct facets, which we will now proceed to examine.

There are several conventionally recognized types of validity that fall under the heading of empirical validity: content validity, construct validity, and criterion validity. Although some methodologists have undertaken critical examinations of challenges in applying these (most notably, Krippendorff [48, ch. 13], after introducing and explaining these concepts, introduces his own typology of validation for quantitative content analysis), this trio of validities remains very popular in the literature and continues to be embraced by many content analysis methodologists [66, pp. 115–118; 77, p. 185].

**Content validity.** Content validity is the extent to which an instrument captures all the features of the concept it is intended to measure. As Schreier [77, pp. 186–191] notes, in a qualitative content analysis, content validity is of relevance to *concept-driven* (deductive) coding frames. Data-driven (inductive) coding frames are instead concerned with face validity. (The reason for this should be obvious. When the categories are generated from the data, instead of being derived from a theory, there is no expectation that they should capture any particular theoretical concept; instead, the only concern is that they faithfully describe the data. But when categories are derived from a theoretical concept, content validity is a concern, because it needs to be shown that the categories adequately cover the concepts they purport to cover.) In this study, only content analysis 2 used a deductive coding frame; the coding frame for content analysis 1 was data-driven. Here, the issue of content validity is a fairly straightforward one, because the categories for content analysis 2 were taken directly from the key concepts in our research—evolution operators, evolution constraints, and evolution path analyses (called dimensions of concern in the coding frame)—as well as traditional software architecture concepts such as components and connectors. (We consider here only the “Classification” main category of content analysis 2; the other two main categories are trivial and straightforward.)

For this study, then, the issue of content validity hinges on whether the category definitions in appendix B.2.2 adequately capture the concepts that they claim to capture. The descriptions of general software architecture concepts such as components, connectors, and attachment relations in the coding guide were based heavily on a widely used text on architectural representation, *Documenting Software Architectures* [20]. For example, compare the description of the “Connector” category in content analysis 2 with selected passages from the text of *Documenting Software Architectures* in table 7. Descriptions of the categories capturing concepts from our research (operators, constraints, analyses) were similarly based on our previous own descriptions of those concepts.

**Construct validity.** Construct validity is “the extent to which a particular measure relates to other measures consistent with theoretically derived hypotheses concerning the concepts (or constructs) that are being measured” [16, p. 23]. As Krippendorff [48, p. 331] explains, this concept “acknowledges that many concepts in the social sciences—such as self-esteem, alienation, and ethnic prejudice—are abstract and cannot be observed directly.” Neuendorf [66, p. 117] observes, “Although many scholars cite the need for the establishment of construct validity [...], good examples of the process are relatively few.”

In this study, we are measuring the concepts we wish to measure rather directly; we are

This category should be applied to software elements that are best characterized as <i>connectors</i> .	—
A connector represents a pathway of interaction between two or more components.	A <b>connector</b> is a runtime pathway of interaction between two or more components. [20, p. 128]
Examples of kinds of connectors include pipes, streams, buses, message queues, and other kinds of communication channels.	Simple examples of connectors are service invocation, asynchronous message queues, event multicast, and pipes that represent asynchronous, order-preserving data streams. [20, p. 128]
Although connectors are often thought of as simple, some connectors are quite complex, and connectors can even have an internal substructure, just as components can. Thus, do not categorize an element as a component simply because it seems complex and bulky.	Like components, complex connectors may in turn be decomposed into collections of components and connectors that describe the architectural substructure of those connectors. [20, p. 128]
Instead, the determination of whether an element is a component or a connector should be guided by its function. If it is principally a computational or data-processing element, it should be categorized as a component. If its principal role is to facilitate communication between components, it should be categorized as a connector.	If a component's primary purpose is to mediate interaction between a set of components, consider representing it as a connector. Such components are often best modeled as part of the communication infrastructure. [20, p. 129]

**Table 7:** The description for the “Connector” category in content analysis 2 (left column), juxtaposed with selected excerpts from *Documenting Software Architectures* [20] (right column). Basing our category definitions on widely accepted descriptions of the concepts under study bolsters their content validity.

not using easily observable qualities as a proxy for difficult-to-observe qualities. Thus, construct validity is of little relevance here.

**Criterion validity.** A procedure has criterion validity (also called instrumental validity or pragmatic validity) “if it can be shown that observations match those generated by an alternative procedure that is itself accepted as valid” [44, p. 22]. Criterion validity may be divided into concurrent validity and predictive validity. Concurrent validity applies when an instrument correlates well with a measure of a related construct administered at the same time, while predictive validity applies when an instrument is a good predictor of some future criterion measure.

Like construct validity, criterion validity is of limited relevance here. There are no accepted procedures for measuring the qualities that our coding frame seeks to get at, and hence no basis for assessing or discussing criterion validity.

There is one final type of validity that is highly relevant to a content analysis such as this one: external validity,<sup>4</sup> or generalizability. Determining the appropriate scope of inference—the degree to which results can be generalized—is one of the most challenging and most important aspects of the validation of a content analysis that seeks to have meaning beyond the immediate context of the data it is based on, or of a case study that aims to have implications beyond the case it examines. Since this work is both a content analysis and a case study (or more precisely a case study that incorporates a content analysis), the problem of

<sup>4</sup>The appearance of “external validity” here may lead you to wonder why the corresponding term “internal validity” appears nowhere in this section. Internal validity is the extent to which a study adequately demonstrates the causal relationships it purports to demonstrate. Since we are not making any causal claims—this is a descriptive and evaluative case study, not an explanatory one—there is no issue of internal validity.

generalizability is a particularly acute one.

There are a few points that are helpful to bear in mind here. First, generalizability is not all-or-nothing. A study is not “generalizable” or “ungeneralizable.” Rather, a study is generalizable to a certain extent—or, even more precisely, in certain respects and to certain contexts. The question is not whether this case study is generalizable, but to what extent (in what respects, to which contexts) it is generalizable.

Second, a case study is not generalizable in the same sense that a study based on statistical sampling is generalizable. In a typical observational study or controlled experiment—say, in epidemiology—generalization is straightforward. The study is conducted on a sample of some population, and the study is generalizable to the extent that the sample is representative of the population (which can be demonstrated by showing that the sampling was done in a valid way, e.g., randomly). In a case study, the goal is not statistical generalizability, but instead some notion of *transferability* or *analytic generalizability*.<sup>5</sup> The case studied is unique, but the findings of the case study can still be applied to other contexts.

Finally, there is another sense in which all-or-nothing thinking is unhelpful in reasoning about generalizability. Not only are there shades of gray in the generalizability of the case study as a whole, but also different results of a single case study may be generalizable in different ways. Some results may be highly specific to the context of the case study, while other may be readily transferable to other cases.

With these points in mind, it is possible to examine the issue of generalizability with respect to our results. In evaluating analytic generalizability or transferability, it is necessary to consider what special characteristics of the case may have influenced the results of the study. For example, in this case study, “Dealing with legacy systems” was among the several most significant challenges of architecture evolution mentioned by participants. However, these legacy challenges have a great deal to do with the history of the specific company under study. At companies whose software systems have a similar history—companies with large, complex software systems that date back decades, built on mainframes using now-archaic technologies—this result would be transferable. And in fact, there are many companies with such a history. But at a company whose software systems have a very different history, the result would likely be quite different. Consider Amazon.com, for example. Amazon.com certainly has some challenges with legacy systems, but they are quite different in character to those that Costco faces. Amazon.com is a significantly younger company, and its very oldest systems are at worst dated, not obsolete. And of course, a brand-new start-up company would have no legacy challenges at all.

---

<sup>5</sup>There is considerable disagreement about the proper handling of generalization in case studies and qualitative research generally. A summary of different schools of thought is given by Seale [80, ch. 8]. There are two particularly significant such schools, which I will summarize in this footnote.

The first camp holds that the proper criterion for generalizability of qualitative research is *transferability*. The goal of a case study, these methodologists say, is not to discover some kind of law that is of general applicability to the world. Rather, it is to achieve a rich understanding of a single case through thick, naturalistic description, such that the results of the case study can be transferred on a case-by-case basis to other contexts where they may be applicable. If we accept this notion of transferability, then transferring the results of a case study requires not only an understanding of the context of the case study itself, but also an understanding of the context to which the results are to be transferred. Significant methodologists in this camp include Lincoln & Guba [53] and LeCompte & Goetz [51].

In the second camp are methodologists who espouse some notion of theoretical generalization. Most prominent among these is Yin [90, pp. 38–39], who argues that a case study can serve as a way of validating a previously developed theory, an approach which he calls *analytic generalization*. Theoretical notions of case study generalizability are stronger than transferability because they generate or provide support for theories that have broad applicability well beyond the immediate context of the case study. However, some methodologists have criticized theoretical notions of generalizability due to the difficulty of deriving broad theoretical results from individual cases. Seale [80, p. 112], for example, is critical of theoretical conceptions of generalization, arguing that transferability “is a more secure basis for good work.”

Of course, these viewpoints are not mutually exclusive. Some methodologists advocate consideration of both transferability and analytic generalizability [49, pp. 260–265; 63, p. 279]. In my discussion of the generalizability of this case study, I have avoided taking a stand on the proper understanding of generalization in qualitative research. But it is useful to keep these schools of thought in mind when considering how this work might be generalizable.

On the other hand, the number one architecture evolution challenge that emerged in this study was “Communication, coordination, and integration challenges.” There is no clear a priori reason to expect particularly acute communication, coordination, and integration challenges to arise in this case. On the contrary, the organization studied has taken significant measures in recent years to ameliorate communication, coordination, and integration difficulties, but still there are significant challenges. Indeed, on a theoretical basis, we might expect that challenges in integrating systems and communicating with relevant personnel are highly relevant to architecture evolutions in general. Thus, we might reasonably say that our result on the prominence of communication, coordination, and integration challenges is more analytically generalizable than that on dealing with legacy systems.

Of particular interest is the generalizability of our results on the applicability of our approach to modeling a real-world evolution. Generalizability is a particularly crucial question here, because not only does this result emerge from a study at a single company, but it emerges from a study of just a single software system. To what extent is this result generalizable?

Again, it is important to keep in mind how generalization works for case studies. There are some aspects of the case under study that do limit generalization in certain respects. For example, the point-of-sale system is one that can be understood fairly well from a component-and-connector architectural perspective. Our coding frame and modeling approach took advantage of this fact. The categories of the coding frame are based on the assumption of a component-and-connector model, and we produced only a component-and-connector view during the modeling process. (Indeed, the architecture description language we used to model the evolution, Acme, is designed exclusively for component-and-component representations.) In a different system where alternate architectural view types are important, the coding frame and the modeling procedures would have to be revised accordingly.

But the overall result—that our approach can capture the main elements of a real-world evolution—seems to be one that has a good deal of generalizability. There was no a priori theoretical reason to believe that the point-of-sale evolution would be particularly easy to model using our approach. We picked that evolution because it was of a reasonable size for the purposes of the case study, and because it was easy to get access to key personnel involved with architecting the system, not because of any special properties that would render it more amenable to our approach. Thus, the main evaluative result seems to have fairly strong external validity. But of course this generalizability has limits. For example, it would be questionable to transfer this result to evolutions of a very different scale (extremely large and complex evolutions, or small code-level evolutions), or to evolutions with special properties that we have theoretical reasons to believe might be difficult to model (e.g., significant elements of uncertainty). This is not to say that such evolutions could not be modeled using our approach—only that this case study does not clearly demonstrate that they can be. Ultimately, case study generalization involves a clear understanding of relevant theory, careful attention to the specifics of both the case being generalized and the case to which the result is to be transferred, and a good deal of judgment.

## A. Interview protocol

This appendix reproduces the interview protocol as approved by Carnegie Mellon University's institutional review board.

---

In a semistructured interview protocol, the interviewer (rather than rigidly adhering to a predefined schedule of questions) has the flexibility to explore participant responses by asking clarifying or follow-up questions on the spot. As a result, the exact set of questions cannot be known in advance. Rather, this protocol provides guidance on the overall form of the interview, the topics to be covered, and examples of key questions.

This protocol document is structured as a list of topics that we intend to cover in our interviews of software engineers at Costco. The questions listed under each topic are illustrative of the kinds of questions we plan to ask, but by no means exhaustive. In addition, the order of topics may be adjusted, or even entire topics excluded, depending on the individual being interviewed. (For example, a newly hired employee probably won't know much about the architecture evolution challenges that Costco has faced in the past, so questions on that topic would be skipped.) Obviously, some portions of the protocol are fixed: consent must always be obtained at the outset of the interview, and the "Conclusion" section will always be last.

### *A.1. Introductory consent script*

My name is Jeffrey Barnes, and I am a PhD student working with Professor David Garlan at Carnegie Mellon University. We are conducting a research study to learn about the architectural evolution of software systems at Costco. As part of our information-gathering process, I would like to interview you to learn about Costco's software systems and software evolution practices. This interview will take approximately \_\_\_ minutes. Your participation in this research is completely voluntary, and you may withdraw from the study at any time.

With your permission, I will collect your name and contact information so that I can contact you later if I have follow-up questions. However, when we publish the results of this study, we will not use your name; the data that we obtain from this interview will be anonymized to protect your identity.

You should avoid using first names or specific names when providing information for this study.

Before we begin, I need to verify that you're eligible to participate in the study. Are you at least 18 years of age?

[If no, abort the interview.]

Great. Do you have any questions about the study as I've described it?

[Answer any questions that the participant asks.]

Do you consent to participate in this study?

[If no, abort the interview.]

With your permission, I would like to take an audio recording of our conversation to ensure I capture it accurately. The audio recording will be kept private, so that only my research advisor and I have access to it. Is it OK if I record this interview?

[If no, conduct the interview without audio recording.]

### *A.2. Collection of personal identifiers*

I'd like to get your contact information so I can contact you later if I have any follow-up questions later or need to clarify something. Would you mind giving me your name, e-mail address, and phone number?

[Collect information from the participant. If the participant does not wish to provide complete contact information, but does wish to continue the interview, proceed without collecting information.]



### *A.3. The participant's role and background*

*Questions such as the following are necessary to understand the context for a participant's observations and descriptions.*

- What's your job title?
- Can you explain what your role is within the organization?
- How long have you been working at Costco?
- What are the main software systems that you work on?
- What are the main kinds of tasks you're responsible for performing on a day-to-day basis?

### *A.4. Software architecture evolution at Costco*

*We will use questions such as the following to learn how software architecture evolution is planned and carried out at Costco today.*

- As I mentioned earlier, my research is on the topic of software architecture evolution, so I'm interested in learning about how software engineers and architects at Costco plan and carry out major evolution of software systems. Can you tell me about any significant evolutions you've been involved with that would fit that description?
- Were the architectural changes necessary to achieve this evolution planned out in detail in advance, or were the changes instead made on an as-needed basis, without any overarching plan?
- What process was used to develop this evolution plan?
- Were any alternative plans considered?

### *A.5. Limitations of today's approaches to software architecture evolution*

*With questions such as the following, we are seeking evidence supporting (or refuting) our hypothesis that today's software architects are in need of better models and tools to help plan and carry out evolutions.*

- What are some of the major challenges that you've faced in trying to plan or carry out the kinds of evolutions that we've been discussing?
- Do you feel that software architects could benefit from better tools for planning these kinds of evolutions, or do you think today's approaches are generally adequate?
- Are there specific areas where you think better tool support would be especially helpful? Are there particular tasks involved in planning an architecture evolution that you think are especially suitable for automation?

### *A.6. Specifics about the evolution of particular software systems at Costco*

*In order to model Costco's software systems accurately, we need to obtain specific information about the systems' architectural structure and evolution. The following questions illustrate the kinds of questions we will ask, although the specific systems and components named in these sample questions are fictitious.*

- Can you give me an overview of the architecture of the inventory management system?
- How does the ordering module interact with the billing subsystem?
- In which version of the system was the new database adapter you mentioned added?
- Were there any particular constraints you had to adhere to as you were restructuring this subsystem?

- What specific changes are planned for this system in the coming months?
- What are the main reasons for migrating this data store from MySQL to Amazon S3?

### A.7. Conclusion

Those are all the questions I have for you today. Do you have any other questions for me about my research or this case study?

Here's my contact information in case you want to contact me regarding this research in the future. Thank you very much for your time.

## B. Coding guide

As explained in section 4.2.4, the content analysis in this case study was broken down into two separate content analyses, each with its own procedures and goals (and thus each with its own coding guide). The first content analysis addresses the “descriptive” questions of the case study, those that address how architects plan and carry out evolution today. The second content analysis addresses the “evaluative” research question, which seeks to assess the suitability of our approach to architecture evolution by using the output of the content analysis for the construction of an evolution model. The coding guides for these two content analyses are given in this appendix. A few of the category descriptions have been edited slightly to elide information deemed to be confidential.

### B.1. Content analysis 1: Learning how architects do evolution

#### B.1.1 General principles

Each coding unit should be assigned exactly one category. No coding unit may be left uncategorized, and two categories may not be assigned to a single coding unit.

Top-level categories may not be assigned to coding units directly; instead, select the appropriate subcategory. That is, never assign the category “Approaches” to a coding unit. Instead, pick the appropriate subcategory, such as “Approaches: Phased development.”

Each main category has a *residual* subcategory, for example “Approaches: Other.” These residual categories should be used very sparingly if at all. Try to assign the best possible specific category to each coding unit, even if there is no perfect fit; only use the residual categories if there really is no suitable category.

In selecting a category for a given coding unit, consider the coding unit itself as well as its immediate context—up to a couple of paragraphs before and after the coding unit.

#### B.1.2 Categories

---

##### **Evolution motives**

Abbreviation: mot

*Description:* Subcategories of this category should be applied to descriptions of impetuses that have motivated software evolution at Costco. These may be stated in terms of goals of the target system (e.g., an evolution occurs because the target system will have improved performance) or inadequacies of the initial system (e.g., an evolution occurs because the existing system has inadequate performance). This category may be applied regardless of whether the evolution described is one that has already happened, one that may yet occur, or one that was considered but not carried out.

---

<p>Add features Abbreviation: mot-fea</p>	<p><i>Description:</i> This category should be applied when an aim of evolution is to implement new functionality or new features.</p> <p>The line between a new feature and a quality improvement can sometimes be fuzzy. For example, an architect may describe a new feature that requires improvements to system interoperability to implement. The “Add features” subcategory should be used whenever a specific feature is being described; reserve the other subcategories, such as “Improve interoperability,” for when the interviewee describes those goals as the primary reason for evolving.</p> <p><i>Example:</i> “Currently we don’t have any way to tell from looking at our pharmacy patient files, our optical patient files, our membership roll, our customers that we’ve reached out to try to sell memberships to, we don’t have any way to say: Is the same person in there five times? There’s no unique, one person we know, ‘Hey, that’s this guy here.’ They need that.”</p>
<p>Modernize technology Abbreviation: mot-mod</p>	<p><i>Description:</i> This category should be applied when a stated aim of evolution is to adopt modern technology or abandon outdated technology.</p> <p>When outdated technology is discussed as a general challenge rather than as motivation for a specific evolution, use “Challenges: Dealing with legacy systems” instead.</p> <p><i>Indicators:</i> The initial system is described as “dated,” or irrelevant to “today’s world,” or the interviewee names a specific technology that an evolution aims to disuse.</p> <p><i>Example:</i> “Eventually, I want this [mid-range system] stuff to all go away. I don’t want to have to mess with [that] anymore”</p>
<p>Keep apace of business needs Abbreviation: mot-pac</p>	<p><i>Description:</i> This category should be applied when an evolution is driven by the rapid pace of change within the industry or within the company.</p> <p>Note that the “Challenges” category has a similar subcategory: “Dealing with rapid change and anticipating the future.” When deciding between these two categories, consider whether the interviewee is describing a reason for a specific evolution or a general challenge that has arisen.</p> <p><i>Indicators:</i> An evolution is described as being motivated by the “accelerating” pace of change, the “growing demand” of the business, or a need to “support the business curve.”</p> <p><i>Example:</i> “The feeling was [...] that our current architecture and application systems would not scale to meet the growing demand of our growing business, because Costco’s an extreme-growth company.”</p>

<p><b>Improve flexibility</b> Abbreviation: mot-flx</p>	<p><i>Description:</i> This category should be applied when an aim of evolution is to make the system more flexible, or to ameliorate inflexibility in the current system. This category should be applied when flexibility in general is described as a goal—when an evolution is described as aiming to open up future possibilities generally. For the specific goal of increasing the system’s ability to interoperate or integrate with other systems, use the code “Evolution motives: Improve interoperability.”</p> <p><i>Indicators:</i> An evolution is described as providing “flexibility,” making the system more “extensible,” or creating future “opportunities.”</p> <p><i>Example:</i> “It was becoming quite brittle. And an opportunity had been presented to us to update to an object-oriented system that would provide some flexibility for anything that we might choose to do someday.”</p>
<p><b>Improve performance</b> Abbreviation: mot-prf</p>	<p><i>Description:</i> This category should be applied when an aim of evolution is to improve system performance.</p> <p>When performance is discussed as a general challenge rather than as motivation for a specific evolution, use “Challenges: Scalability, reliability, and performance” instead.</p> <p><i>Example:</i> “We’re going to [a new point-of-sale package], which is basically the C++ version [...]. It’s supposed to be a little faster”</p>
<p><b>Improve reliability</b> Abbreviation: mot-rel</p>	<p><i>Description:</i> This category should be applied when an aim of evolution is to improve the availability, robustness, or resilience of the system.</p> <p>When reliability is discussed as a general challenge rather than as motivation for a specific evolution, use “Challenges: Scalability, reliability, and performance” instead.</p> <p><i>Example:</i> “Now we’re moving more into the highly available, somewhat more resilient message-based architecture, and I think that’s going to be a good thing for us.”</p>
<p><b>Improve interoperability</b> Abbreviation: mot-xop</p>	<p><i>Description:</i> This category should be applied when an aim of evolution is to make integration or interoperation among systems easier.</p> <p>When interoperability is discussed as a general challenge rather than as motivation for a specific evolution, use “Challenges: Communication, coordination, and integration challenges” instead.</p> <p><i>Indicators:</i> The current system is described as being too “proprietary,” or a stated goal is to make it easier to “hook to,” “work with,” “talk with,” or “share information” with other systems.</p> <p><i>Example:</i> “We’re going to try and make those systems work with each other and talk with each other and share information and data back and forth.”</p>
<p><b>Other</b> Abbreviation: mot-?</p>	<p><i>Description:</i> This category should be applied to evolution motives that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.</p>

<p><b>Causes of problems</b> Abbreviation: cau</p>	<p><i>Description:</i> Subcategories of this category should be applied to descriptions of circumstances that caused problems during the course of an evolution project. Architects often describe such circumstances in conjunction with their adverse effects, so “Causes of problems” codes and “Consequences” codes often occur in close proximity.</p> <p>Undesirable circumstances that have not caused any specifically articulated problems do not fall under this category but may fall under the “Challenges” category if they meet its criteria.</p>
<p><b>Lack of experience</b> Abbreviation: cau-exp</p>	<p><i>Description:</i> This category should be applied when a problem was caused by inexperience (or a failure to learn from experience), whether on the part of the architects planning an evolution or on the part of the engineers carrying it out.</p> <p>When inexperience is discussed as a challenge in general rather than as the cause of a specific adverse consequence, use “Challenges: Lack of expertise or experience” instead.</p> <p><i>Example:</i> “The bulk of the people—especially, say, five years ago—that work in Costco IT—the bulk of them came up through our warehouses. So there’s not a lot of ex-consultants running around here to have seen a million and one things. These are people who have seen one way of doing things. And then to say, you’ve got to do something new, and by the way you’ve never experienced that before, mistakes are going to get made.”</p>
<p><b>Architects spread across the organization</b> Abbreviation: cau-org</p>	<p><i>Description:</i> This category should be applied when a problem was caused by architects being spread across many different groups within the organization, impeding communication among architects and complicating diagnosis of system failures.</p> <p><i>Example:</i> “Our forms environment just couldn’t work. Why? Because the architecture was spread across different groups”</p>
<p><b>Addressing too many problems up front</b> Abbreviation: cau-prb</p>	<p><i>Description:</i> This category should be applied when a problem was caused by attempting to solve all the problems of an evolution at its outset, rather than developing a basic plan that allows for adaptation to unforeseen circumstances.</p> <p><i>Example:</i> “Let’s just put in a foundational system that’s open to the rest, rather than trying to solve all these questions at the time, which even now don’t have answers for them. I think that was part of what happened in those multiple iterations as we tried to solve and could never get that compelling business reason down”</p>
<p><b>Cultural inertia</b> Abbreviation: cau-cul</p>	<p><i>Description:</i> This category should be applied when a problem was caused by cultural factors that inhibited change.</p> <p><i>Example:</i> “When we started modernization, for whatever reason, we were talking more about COTS, but kind of in our DNA, we were still thinking <i>build</i>.”</p>
<p><b>Forking an off-the-shelf package</b> Abbreviation: cau-frk</p>	<p><i>Description:</i> This category should be applied when a problem was caused by adopting and modifying an off-the-shelf software package, making it difficult to incorporate further updates from the vendor.</p> <p><i>Example:</i> “We bought a package [...], and we did a very bad thing: we took the source code, we actually customized the source code for our need”</p>

<p>Ill-timed changes Abbreviation: cau-tim</p>	<p><i>Description:</i> This category should be applied when a problem was caused by changing the system at an inopportune time.</p> <p><i>Example:</i> “I don’t know if you’ve ever heard of Finish Line; they’re an athletic footwear dealer, online mainly. They piloted new e-comm and web software a week before Black Friday—the Friday after Thanksgiving. Not the best of times to pilot or make any changes to your website. It did not work too well.”</p>
<p>Other Abbreviation: cau-?</p>	<p><i>Description:</i> This category should be applied to causes that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.</p>
<hr/>	
<p><b>Consequences</b> Abbreviation: con</p>	<p><i>Description:</i> Subcategories of this category should be applied to adverse consequences that arose due to missteps in an evolution project. Architects often describe problems in conjunction with their causes, so “Causes of problems” codes and “Consequences” codes often occur in close proximity.</p>
<p>Lost sales Abbreviation: con-sal</p>	<p><i>Description:</i> This category should be applied when problems during an evolution resulted in financial loss.</p> <p><i>Example:</i> “They figure it cost them about six million dollars in lost sales and things.”</p>
<p>Wasted effort Abbreviation: con-eff</p>	<p><i>Description:</i> This category should be applied when missteps during an evolution resulted in unnecessary effort or unnecessary complications, or when an evolution is rolled back and abandoned, or when the direction or basic approach of the evolution must be changed while it is being carried out due to unforeseen problems.</p> <p><i>Example:</i> “We ran employees through literally tens of thousands of hours of Java training, but then they were not doing Java development.”</p>
<p>Delays in evolving Abbreviation: con-dly</p>	<p><i>Description:</i> This category should be applied when difficulties in planning an evolution delay the evolution’s inception.</p> <p><i>Example:</i> “I think that was part of what happened in those multiple iterations as we tried to solve and could never get that compelling business reason down, so it’s taken so long for us to do this.”</p>
<p>Limited upgradability Abbreviation: con-upg</p>	<p><i>Description:</i> This category should be applied when missteps in an evolution result in a system that is difficult to upgrade further.</p> <p><i>Example:</i> “[...] which rendered that package to be not upgradable for any future changes.”</p>
<p>Other Abbreviation: con-?</p>	<p><i>Description:</i> This category should be applied to consequences that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.</p>
<hr/>	
<p><b>Challenges</b> Abbreviation: cha</p>	<p><i>Description:</i> Subcategories of this category should be applied to descriptions of challenges that Costco, or similar companies, or software organizations in general, face (or have faced in the past) when planning or carrying out evolution.</p> <p>This category should not be used for specific challenges that motivated evolution (these should instead be coded with subcategories of “Evolution motives”), nor for specific circumstances that caused problems in past efforts (these should be coded with subcategories of “Causes of problems”).</p>

---

**Cultural challenges**  
Abbreviation: cha-cul

*Description:* This category should be applied to descriptions of corporate-culture issues that present challenges, such as clashes between the corporate culture and the needs of the business, cultural resistance to needed innovations, or a need to contravene the organizational culture in order to accomplish goals.

*Indicators:* The interviewee, in describing a challenge that Costco faces or has faced, uses words like “culture” or “cultural,” or describes a tension between what the company says and what it does.

*Example:* “Frankly that’s been, I think, the biggest challenge of the modernization that we’ve undertaken, because [...] we have a culture of consensus here [...]. We don’t do a lot of empowering some group to say how it’s going to be, and then they just lay it out. That doesn’t happen. Anything that we do has to be very collaborative if it’s going to change the way people operate and whatnot. Frankly, we’re still in the throes of getting that to work well.”

---

**Business justification**  
Abbreviation: cha-bus

*Description:* This category should be applied to challenges in justifying architectural decisions (or the need for architecture, or the role of architects) in business terms (or to business executives or enterprise architects); challenges in managing the expectations of business executives; challenges caused by engineers having insufficient business knowledge to make good decisions; challenges in ensuring that products support business goals; and challenges in understanding business needs.

*Indicators:* The interviewee discusses challenges relating to executive or business “buy-in” or “sponsorship,” the difficulty of explaining architectural needs “to the business,” business “concern” about architectural decision, or the attitudes of the “management,” or the interviewee questions whether businesses are “ready” to do architecture, or the interviewee observes that engineers lack an “appreciation of the business.”

*Example:* “I think in terms of architecture, what I’ve seen the biggest challenges are executive buy-in on the architecture function in general, because when you throw architects in at the beginning (goodness, I hope it’s at the beginning) of a project, you’re adding time to it [...]. We’ve been fortunate that we’ve got CIO buy-in of what we’re doing. But I think that’s probably the largest obstacle, and when I meet with [architects] from other companies, they always say that that’s the largest challenge they face.”

---

<p>Communication, coordination, and integration challenges Abbreviation: cha-com</p>	<p><i>Description:</i> This category should be applied to challenges in integrating systems or facilitating communication among systems, to challenges of communication and coordination among people or teams, to challenges in managing multiple simultaneous initiatives, to challenges in appropriate use of documentation and standards, and to challenges of architecture conformance.</p> <p>The challenge of integrating with legacy systems specifically should instead be coded “Challenges: Dealing with legacy systems.”</p> <p>When system integration is discussed as the impetus for a specific evolution rather than as a general challenge, use “Evolution motives: Improve interoperability.”</p> <p>When organizational communication issues are discussed as the cause of a specific adverse consequence rather than as a general challenge, use “Causes of problems: Architects spread across the organization.”</p> <p><i>Example:</i> “One of the other challenges is just that we’ve grown now. We’re in this building; the other half of IT is over on the corporate campus. You can’t get all of IT together and have a presentation type of thing. [...] That’s been a challenge. The communication is the biggest part.”</p>
<p>Lack of expertise or experience Abbreviation: cha-exp</p>	<p><i>Description:</i> This category should be applied to challenges arising from a lack of expertise or experience on the part of architects or engineers (or immaturity on the part of the organization as a whole), including general inexperience, a lack of knowledge about certain domains, or a lack of familiarity with particular tools. This category should also be applied to discussions of the difficulty of training people or the difficulty of learning new technologies.</p> <p>When inexperience directly causes a specifically articulated problem in the course of an evolution, use “Causes of problems: Lack of experience” instead.</p> <p><i>Example:</i> “And the tools around that, really, we’re still very immature with.”</p>
<p>Dealing with rapid change and anticipating the future Abbreviation: cha-cha</p>	<p><i>Description:</i> This category should be applied to challenges pertaining to planning ahead in the face of rapid change, seeing the long-term ramifications of decisions, dealing with extreme change within the company or throughout the industry, estimating future effort, adapting advances in the industry to Costco, or realizing a long-term vision.</p> <p>When rapid change is discussed as the impetus for a specific evolution rather than as a general challenge, use “Evolution motives: Keep apace of business needs” instead.</p> <p><i>Indicators:</i> Instances of this category often refer to “extreme” or “massive” change; describe the company as moving at a very fast pace; or use metaphors such as shifting sand, a shifting landscape, or a narrow window of opportunity.</p> <p><i>Example:</i> “The biggest challenge really, in my mind, is: How do you make sure that you are relevant? The wheel is moving, you know. Costco’s situation, I think, is one of the more extreme kind of conditions. We are trying to change so much in such a short time.”</p>



<p>Dealing with legacy systems Abbreviation: cha-leg</p>	<p><i>Description:</i> This category should be applied to challenges that arise in dealing with legacy systems, including architectural problems with legacy systems, challenges in understanding legacy systems, challenges in bringing legacy vendor systems in-house, and difficulties in upgrading legacy systems (such as running a legacy system alongside a modernized system).</p> <p>When the difficulty of dealing with legacy systems is discussed as the impetus for a specific evolution rather than as a general challenge, use “Evolution motives: Modernize technology” instead.</p> <p><i>Example:</i> “The applications that were built [...] here over time—over the twenty-five or thirty years or so that the mid-range systems were here—really grew organically, and there wasn’t comprehensive architectural thinking going on in the early days, to the point where you had a big ball of string. And now it’s so brittle that [...] the simplest changes are taking months and months”</p>
<p>Scalability, reliability, and performance Abbreviation: cha-sca</p>	<p><i>Description:</i> This category should be applied to challenges in managing large volumes of data; carrying out large implementations; meeting the needs of a company of Costco’s size; ensuring transmission of data over the Internet; verifying the reliability of critical systems (e.g., through QA); and providing adequate system performance.</p> <p>When reliability or performance is discussed as the impetus for a specific evolution rather than as a general challenge, use “Evolution motives: Improve reliability” or “Evolution motives: Improve performance” instead.</p> <p><i>Example:</i> “It’s very limited to the options for a retailer of our size what we can use for that point-of-sale. Especially when the project started, and even somewhat now, Windows is always looked at as less than ideal as a stable platform for running point-of-sale, both from stability and uptime during the day, as well as the need for constant patches and those kind of things. So when you limit that away and look at other systems, other operating system types, to run it, there’s even fewer that are out there.”</p>
<p>Dealing with surprises Abbreviation: cha-sur</p>	<p><i>Description:</i> This category should be applied to unexpected challenges that arise in the midst of an evolution, such as new requirements or unexpected infrastructure needs. If the unexpected challenge was a significant misstep that resulted in a specifically mentioned adverse consequence, use the appropriate subcategory of “Causes of problems” or “Consequences” instead.</p> <p><i>Example:</i> “The reality is: things don’t go as smooth as you’d like them to in the order you’d like them to. Sometimes development may be backed up, in which case you need to go renegotiate when does your project get started versus the projects that are in play. Capital expenditures: maybe we need to purchase extra infrastructure that we didn’t realize needed to be purchased up front.”</p>
<p>Managing scope and cost Abbreviation: cha-cst</p>	<p><i>Description:</i> This category should be applied to challenges in scoping a project or keeping costs low.</p> <p>Challenges in reconciling large expenses with a cultural of frugality should instead be coded “Challenges: Cultural challenges.”</p> <p><i>Example:</i> “The biggest concern retailers have about [alternative payment methods] is the fees they have to pay as part of them, if you know what an interchange fee is. Basically there’s a percentage you have to pay. [...] that’s been a continuing challenge for the retailers.”</p>

<p>Inadequate guidance and tool support Abbreviation: cha-gui</p>	<p><i>Description:</i> This category should be applied when architects report that they do not have good resources to turn to for information, or that they have no good source of architectural guidance, or that they need better tools, or that available approaches are insufficient, or that they do not have enough guidance on projects from more senior people in the company.</p> <p><i>Example:</i> “You can get shades-of-gray answers all over the place. Nobody’s put a fine point on any aspect of it, so you can interpret what you will. I was telling somebody the other day, just go google <i>system context diagram</i>, and see how many flavors of that you get: like a thousand different models. [...] I think again, the challenge is there is no definitive resource.”</p>
<p>Divergent understandings of architecture Abbreviation: cha-und</p>	<p><i>Description:</i> This category should be applied to challenges that arise when there are people who have misunderstandings of architecture, disagreements about the role of architecture, or negative perceptions about architecture (or specific varieties of architecture).</p> <p>If the challenge as described is specific to dealing with business executives, use the category “Challenges: Business justification” instead.</p> <p><i>Example:</i> “In general—not just Costco, but in general—there is a particular image that comes to a lot of people’s mind when you say ‘enterprise architecture.’ Some people go into, ‘Oh, man, that’s a really needed thing that we’ve needed for a long time.’ The other end of the spectrum is a bunch of ivory-tower impediments to progress.”</p>
<p>Managing people Abbreviation: cha-ppl</p>	<p><i>Description:</i> This category should be applied to challenges pertaining to the management of personnel, including dealing with employee uncertainty, retraining personnel with outdated expertise, motivating workers, and dealing with employee turnover.</p> <p><i>Example:</i> “We’re rooting out [our old] productivity suite, and we’re going with [a different provider]. [...] Now, we had [administrators for the old system]. So what are you going to do? At first they were very resistant.”</p>
<p>Managing the expectations and experience of users and stakeholders Abbreviation: cha-usr</p>	<p><i>Description:</i> This category should be applied to challenges in managing users’ and stakeholders’ expectations about software systems or the software development process, as well as to challenges in ensuring that stakeholders and users of systems have a good experience. Included in this category are topics such as managing unrealistic expectations, understanding the political impacts of decisions, minimizing the impact of a system change on its users, and training users on a new system.</p> <p><i>Example:</i> “The problem is in the user community, they feel like once they have your attention, they have to get everything they can out of you, otherwise they might not hear from you again for three years, and that’s not going to help them. They come and they want the kitchen sink and everything, and it’s very hard to tell them, we can’t do that”</p>
<p>Other Abbreviation: cha-?</p>	<p><i>Description:</i> This category should be applied to challenges that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.</p>

<p><b>Approaches</b> Abbreviation: app</p>	<p><i>Description:</i> Subcategories of this category should be applied to descriptions of approaches, methods, principles, structures, and tools that architects use (or have used, or have considered using) to help them plan and carry out evolutions. This includes everything from formal processes to very informal rules of thumb. It does not, however, include specific tactics or operations such as “introduce abstraction layer” or “wrap legacy component”.</p> <p>Interviewees often spoke at considerable length and in significant detail about the approaches they use, to a much greater degree than is true for the other major categories in this coding frame. In addition, some of the subcategories of this category are rather broad topically. For example, “Challenges: Organizational strategies and structures” is a fairly broad topic, and interviewees often expound on the topic of organizational structure at considerable length. As a result, subcategories of this category will often apply to quite long segments of text, to a greater degree than is true of the other major categories, which usually apply to shorter passages, rarely longer than a few sentences.</p>
<p><b>Experience and intuition</b> Abbreviation: app-int</p>	<p><i>Description:</i> This category should be applied to descriptions of architects relying on their own experience, intuition, or judgment to make evolution decisions, or of architects learning from their experiences.</p> <p><i>Example:</i> “I would say that my twenty-five years of experience in retail—I have a very unique perspective on what is realistic, what works, and how to stay away from the fringes and do what’s right for the company.”</p>
<p><b>Drawing from expert sources</b> Abbreviation: app-exp</p>	<p><i>Description:</i> This category should be applied to descriptions of drawing from expert sources to gain knowledge necessary to plan or carry out an evolution. Expert sources included experienced engineers within the company, outside consultants, product vendors, and books.</p> <p><i>Example:</i> “Our current direction in Costco for these migrations between current state to these packaged solutions is leveraging a lot of external vendors [...] We are looking at them to provide. ‘From [your] own best practices, how do you guys do that? We’re not going to question you. You should tell us: How do you get this thing done, and how do you ensure there’s knowledge transfer and support and all that stuff?’”</p>
<p><b>Industry practices</b> Abbreviation: app-ind</p>	<p><i>Description:</i> This category should be applied when Costco draws on the experiences and expertise of other companies in the industry.</p> <p><i>Example:</i> “We’ve got a mobile app. We’ve got different types of mobile checkout people want to look at. How do we do that in the best way? So we’re looking at what are the trends in the industry in mobile, both with our big competitors like Wal-Mart and Target, as well as [smaller chains].”</p>
<p><b>Phased development</b> Abbreviation: app-pha</p>	<p><i>Description:</i> This category should be used when the development or delivery of a project is broken into manageable phases or stages, or when an incremental or gradual transition strategy is used.</p> <p><i>Example:</i> “I did a very iterative approach. I divided the project into three evolutions. Each evolution basically took on a chunk of our scope.”</p>

<b>Tools</b> Abbreviation: app-too	<p><i>Description:</i> This category should be applied to mentions of tools used in planning or carrying out evolution. (These need not be—and generally will not be—architecture evolution tools as such, but rather tools that architects and engineers use in the course of planning or carrying out evolution, including communication tools, code generation tools, reverse engineering tools, and process planning tools.)</p> <p><i>Example:</i> “We got an analysis tool that we put onto our system and we had it do a drawing of all of the components and the relationships”</p>
<b>Formal approaches</b> Abbreviation: app-frm	<p><i>Description:</i> This category should be applied to mentions of established processes, methods, and frameworks for software development, such as TOGAF and the Rational Unified Process.</p> <p><i>Example:</i> “Rather than simply utilize EA group and the knowledge and experience we have [...], we’re trying to use a couple additional tools, one of which is TOGAF. We’re not strictly following that, but that’s the basis of our architecture foundation. The core of that is the ADM process.”</p>
<b>Prototypes, pilots, etc.</b> Abbreviation: app-pil	<p><i>Description:</i> This category should be applied to descriptions of the use of prototypes, proofs of concept, pilots, reference applications, project templates, case studies, mock applications, and technical demonstrations for trying out or demonstrating innovations.</p> <p><i>Example:</i> “For those architectural decisions, we develop guidance and references, and then further down we create a reference application that actually implements the decisions and the guidance that we are telling people to follow.”</p>
<b>Training</b> Abbreviation: app-trn	<p><i>Description:</i> This category should be applied to approaches for the training and mentoring of architects and engineers (including the retraining of engineers with unneeded skills).</p> <p><i>Example:</i> “I went through the ISA training and got certified on that”</p>
<b>Business architecture</b> Abbreviation: app-bus	<p><i>Description:</i> This category should be applied to descriptions of the use of business architecture practices, or the consideration of business concerns in planning architectural operations more generally, or thinking in terms of business capabilities.</p> <p><i>Example:</i> “Instead of all these very virtual or kind of theoretical categories to rank a service, we just look at business capability. Services are supposed to be aligned with the business when we build a service, so why don’t we just go straight to the source? So what we ended up doing is looking at business capability from the business architecture team.”</p>
<b>Communication and coordination practices</b> Abbreviation: app-com	<p><i>Description:</i> This category should be applied to descriptions of approaches for facilitating communication and coordination within the organization, including coordinating efforts, allocating tasks, providing guidance, documenting decisions, communicating with stakeholders, and incorporating feedback.</p> <p>When the discussion centers on the challenges of communication and coordination rather than approaches for communication and coordination, use the category “Challenges: Communication, coordination, and integration challenges” instead.</p> <p><i>Example:</i> “It’s really more about communication than architecting systems. That collaboration aspect, I think, is absolutely paramount to the success of an architect. You have to talk to lots of people all the time. The drawings don’t have to be that precise, as long as they communicate the information.”</p>

<p>Considering alternatives Abbreviation: app-alt</p>	<p><i>Description:</i> This category should be applied when architects consider alternative plans or backup plans for evolving a system.</p> <p><i>Example:</i> “We have to always, I think, have plan A and plan B. Going into a lot of the work that we’re doing now, I say, well, I would love to have this SOA service environment, but understanding that that’s still very much in its infancy, here I have plan B, which is let’s just continue on with what we know using legacy systems and legacy technologies.”</p>
<p>Anticipating the future Abbreviation: app-fut</p>	<p><i>Description:</i> This category should be applied to descriptions of anticipating future developments as an approach for better planning evolution.</p> <p>When anticipating the future is cited as a challenge rather than as a strategy, use the category “Challenge: Dealing with rapid change and anticipating the future” instead.</p> <p><i>Example:</i> “Besides point-of-sale, we’re modernizing our loyalty system, our CRM system. We’re modernizing all those parts and pieces. As we modernize those, then we look to that future where they all have an interaction with each other.”</p>
<p>Rules of thumb and informal strategies Abbreviation: app-rot</p>	<p><i>Description:</i> This category should be applied whenever an interviewee articulates a general rule of thumb or informal strategy for planning or carrying out evolution, such as “lean toward simplicity,” “prefer open standards,” or “identify areas of uncertainty.”</p> <p><i>Example:</i> “Consistency is very important in my book. I’m less about which standard’s a better standard than the other—rather that these have to be enterprise standards—everybody should do it the same way.”</p>
<p>Organizational strategies and structures Abbreviation: app-org</p>	<p><i>Description:</i> This category should be applied to descriptions of the organizational structures and strategies that Costco uses in architecting systems. This includes discussions of architects’ and engineers’ roles and the structure, function, and formation of groups and teams.</p> <p><i>Example:</i> “On that EA side, as we talked about, there’s the domain architects, which span each pillar: information, integration, business, mobility, security, . . . . Large number of people over on that side. We’re supposed to be focused, as SA, on the implementation or the project level, and they’re supposed to be focused on more of the strategy and the domain expertise for us to go to for implementation to make sure we’re using best practices in the security domain, for example, or infrastructure domain, or some domain that an individual architect may not be versed in”</p>
<p>Process Abbreviation: app-pro</p>	<p><i>Description:</i> This category should be applied to descriptions of software process at Costco, including descriptions of stages of software development and discussions of project life cycle.</p> <p><i>Example:</i> “There’s this SDM (solution delivery methodology) out there, which is basically the end-to-end life cycle of all the people involved in a project. Start-up is the first phase; that’s when you engage the PM and do as we talked about, the pre-requirements gathering. Then it goes to solution outline, which finalizes in the EARB. From there you go to macro design. Then you go to micro design, implementation, deployment, and close-down.”</p>
<p>Other Abbreviation: app-?</p>	<p><i>Description:</i> This category should be applied to approaches that do not fit into any of the other categories. Use this category sparingly if at all. Residual categories such as this should be used only when none of the specific categories fit.</p>

## B.2. Content analysis 2: Modeling a real-world evolution

### B.2.1 General principles

Each coding unit should be assigned exactly one subcategory of *each* applicable main category. (Note that this differs from the procedure in content analysis 1, in which each coding unit should be assigned only one category *total*.) The “Classification” main category is applicable to all coding units. Thus, all coding units should be assigned exactly one subcategory of the “Classification” main category. The other main categories are applicable to coding units that describe architectural elements. Thus, coding units that have been classified as components, connectors, ports or roles, systems, or groupings should be assigned exactly one subcategory of “Presence in initial architecture” and one subcategory of “Presence in target architecture.” Those that have been classified as containment relations, evolution operations, evolution constraints, dimensions of concerns, or “Other” should not be coded with respect to the “Presence in initial architecture” and “Presence in target architecture” categories.

Because the coding units in this content analysis are not contiguous—each coding unit consists of a set of isolated fragments of content spread throughout the data—substantial surrounding context may be necessary to accurately categorize a coding unit. This surrounding context certainly extends to each page on which the coding unit occurs in the architectural documentation, as well as a couple of paragraphs before and after each occurrence of the coding unit in the interview transcripts. In a few cases where categorization is particularly difficult, it may be appropriate to go beyond the research data and consider external sources of information. Specifically, the following sources may be considered if necessary to accurately judge the proper categorization of a coding unit:

- The socket server diagram that I received in connection with the interviews
- The cash recycler integration architecture document that I received in connection with the interviews
- Publicly available documentation pertaining to specific commercially available software products, obtained from vendor websites (in the case of coding units corresponding to off-the-shelf products)

However, the architectural documentation and interview transcripts that form the research data should be the principal consideration. The external documents listed above should be used only as necessary to settle close calls in cases of ambiguity.

### B.2.2 Categories

---

**Classification**

Abbreviation: c

*Description:* Subcategories of this category are used to identify what type of entity (software element, constraint, or dimension of concern) a coding unit refers to. Every coding unit should fall into one of these subcategories; therefore, each coding unit should be assigned exactly one subcategory of this category.

If a coding unit fails to correspond to any of the subcategories, categorize it as “Classification: Other.” If there appear to be multiple subcategories that could apply to a coding unit, read the descriptions of the subcategories below carefully for discussion of the distinctions among them, so that you can choose the single most appropriate classification.

---

---

**Component**

Abbreviation: c-cmp

*Description:* This category should be applied to software elements that are best characterized as *components*. A component represents a computational element within a software system. Examples of kinds of components include software packages, services, data stores, and data processing elements. A component has a well-defined identity, a well-defined boundary, and a well-defined interface by which it interoperates with other software elements.

The “Component” category may be confused with several other categories, notably “Connector,” “System,” and “Grouping.” See the definitions of these other categories below for discussion of the distinctions.

*Indicators:* A software element that is described as an “application,” “controller,” “file,” “handler,” “log,” “product,” or “server” is usually a component. However, many components are not described in such terms, and a few elements that *are* described in these terms are better classified as systems or connectors. Thus, components must ultimately be identified based on their general characteristics, as described above.

An off-the-shelf package should always be classified as a component (unless it exists chiefly to facilitate communication, in which case it is a connector).

In graphical diagrams, components are often depicted as boxes. However, boxes are used for a great many other purpose, including representation of connectors, systems, and groupings, so this is by no means a reliable indicator of a component.

---

---

**Connector**  
Abbreviation: c-cnn

*Description:* This category should be applied to software elements that are best characterized as *connectors*. A connector represents a pathway of interaction between two or more components. Examples of kinds of connectors include pipes, streams, buses, message queues, and other kinds of communication channels. Although connectors are often thought of as simple, some connectors are quite complex, and connectors can even have an internal substructure, just as components can.

Thus, do not categorize an element as a component simply because it seems complex and bulky. Instead, the determination of whether an element is a component or a connector should be guided by its function. If it is principally a computational or data-processing element, it should be categorized as a component. If its principal role is to facilitate communication between components, it should be categorized as a connector.

*Indicators:* A software element that is described as a “bridge,” “broker,” “bus,” “queue,” or “transfer” is almost certainly a connector. However, connectors are not always described in such terms and often must be identified based on their general characteristics, as described above.

In prose (spoken or written), connectors are often not discussed in explicit terms as first-class entities, but instead appear implicitly as relationships between components. For example, if a component is described as “talking to” another component, the phrase “talking to” is evidence of a connector between the two components. A coding unit that captures a relation between components (or systems or groupings) should always be classified as either a connector or a containment relation.

In graphical diagrams, components are often depicted as lines (or arrows). In fact, a line between two components in a diagram almost always represents a connector. However, the converse is not true. Connectors can be diagrammatically represented in many other ways besides as lines—including as boxes—so the fact that an element appears as something other than a line in a diagram is not evidence that it is not a connector.

---

**Port or role**  
Abbreviation: c-att

*Description:* This category should be applied to coding units that express an attachment relation between a component (or system or grouping) and a connector. (This could be characterized architecturally as a port, a role, or a combination of a port attached to a role.)

Application of this category is relatively straightforward. When a coding unit expresses a relationship between a component (or system or grouping) and connector, that unit should be either coded as “Port or role” (if the relationship is one of component-connector attachment) or as “Containment relation” (if the relationship is one of containment).

---



---

**System**

Abbreviation: c-sys

*Description:* This category should be applied when a coding unit refers to a complete software system.

The distinction between a system and a component can be hazy, since components may themselves be fairly large and complex and may contain other software elements. However, a software system is understood to operate as a complete and relatively independent whole (although it may have dependencies on other systems with which it interoperates), while a component is intended to operate as one piece of a larger system.

In some cases, though, the distinction between a system and a component is merely a matter of perspective. For example, an off-the-shelf point-of-sale package could certainly be considered a system from the perspective of its developers, the package vendor, but from Costco's perspective, it is just one component of Costco's point-of-sale system. Thus, this application should be coded as a component rather than a system, since we are taking the perspective of Costco.

Topologically, systems are equivalent to components. Thus, connectors, which normally connect components to components, may also connect systems to systems, or components to systems. Indeed, a system may be regarded as a type of component. The distinction is made in this coding frame simply because it facilitates understanding of system boundaries and areas of responsibility.

Also murky is the distinction between a system and a grouping. In general, a system is a concrete package of software elements that have been deliberately composed together into a unified whole. The *grouping* category below is for more nebulous collections of elements. Such groupings may be used to associate elements logically, to express physical boundaries, or to demark software layers or tiers.

*Indicators:* Systems are often easy to recognize simply because they are called "systems"—for example, the point-of-sale *system*, or the membership *system*. However, this indicator is not foolproof; the term *system* sometimes appears in descriptions of things that might be better categorized as components. Notably, off-the-shelf packages should always be classified as components (or connectors), even though they might be regarded as systems under other circumstances.

---

<b>Grouping</b> Abbreviation: c-grp	<p><i>Description:</i> This category should be applied to coding units that express logical groupings of software elements, such as software layers or tiers. It should also be applied to coding units that express the physical boundaries within which software elements may be contained. For example, architects frequently distinguish between software elements that are contained within individual Costco warehouses, and those that are part of the central corporate infrastructure. In this case, <i>warehouse</i> defines a physical boundary and should be considered a grouping.</p> <p>Sometimes it may be unclear whether to categorize a unit as a grouping or a component. Components, after all, can (like groupings) contain other software elements. However, components exist as specifically identifiable elements within the software; they are discrete elements with a well-defined interface, intended to interoperate with other software elements. Groupings are more nebulous, and are often purely logical; in other words, it may be useful to use them to group related software elements together for purposes of discussion and analysis, but they may not have any real presence in the software.</p> <p>It may also be possible to confuse groupings with systems. See the definition of the “System” category above for a discussion of the distinction.</p> <p>Groupings, like systems, are topologically equivalent to components. See the description of the “System” category for an explanation of this point.</p>
<b>Containment relation</b> Abbreviation: c-cnt	<p><i>Description:</i> This category should be applied to coding units that express a containment relation between two software elements—that is, when a coding unit expressing a relationship between two software elements implies that one of them is contained within the other.</p> <p><i>Indicators:</i> In diagrams, containment is easy to recognize because it is almost always represented by physical containment; for example, one box (representing a system) contains another, smaller box (representing a component).</p> <p>Containment relations are not so clearly manifest in prose. However, a containment relation is often described in terms of the contained element being “in” the containing element, or the containing element “having” the contained element.</p>
<b>Evolution operation</b> Abbreviation: c-eop	<p><i>Description:</i> This category should be applied to coding units that describe an operation that may be carried out during the evolution. Evolution operations can be simple—such as adding, removing, or modifying individual software elements—or moderately complex, such as interposing a bridging element between two components to facilitate communication, or replacing a network of point-to-point connectors with a bus.</p>

<p><b>Evolution constraint</b> Abbreviation: c-cns</p>	<p><i>Description:</i> This category should be applied to coding units that express a constraint on the evolution of the system. Examples of kinds of evolution constraints including ordering constraints on evolution operations, architectural constraints characterizing the structure that the system must have at specific points within the evolution, timing constraints expressing when operations must be carried out, and organizational constraints expressing which organizational elements carry out which tasks.</p> <p>Constraints should be easy to distinguish from all of the other categories here except perhaps “Dimension of concern.” After all, constraints and dimensions of concern both represent qualities that are desired of an evolution. The difference is that constraints express compulsory requirements that an evolution must have in order to be valid. Dimensions of concern represent qualities that lie along a range or spectrum, with one end of the range being preferable to the other, and points in the middle of the range being intermediate between them. Thus, “availability” generally is a dimension of concern, while “The system must maintain 99% availability at all times” is a constraint.</p>
<p><b>Dimension of concern</b> Abbreviation: c-cnc</p>	<p><i>Description:</i> This category should be applied to coding units that capture a dimension of concern relevant to the evolution of the system—something that could, at least in principle, be quantified and used as an optimization criterion in planning the evolution. Examples of possible dimensions of concern include cost, effort, evolution duration, and architectural qualities such as performance or reliability.</p> <p>For the difference between constraints and dimensions of concern, see the definition of the “Evolution constraint” category above.</p>
<p><b>Other</b> Abbreviation: c-?</p>	<p><i>Description:</i> This category should be applied to coding units that do not fall into any of the above categories. Categorize coding unit in the most appropriate category above if at all possible; use this residual category only if none of the above categories are at all applicable.</p>
<p><b>Presence in initial architecture</b> Abbreviation: init</p>	<p><i>Description:</i> Subcategories of this category are used to indicate whether a software element is present in the initial architecture of the system. Note that only coding units corresponding to software elements should be assigned a subcategory of this category. Coding units not corresponding to software elements (i.e., coding units tagged “c-cnt,” “c-eop,” “c-cns,” “c-cnc,” or “c-?”) should not be assigned a subcategory of this category, since these concepts (constraints, operations, etc.) are not local to individual evolution states. However, any coding unit that has been identified as a software element (i.e., a coding unit classified “c-cmp,” “c-cnn,” “c-att,” “c-sys,” or “c-grp”) should be assigned exactly one subcategory of this category.</p> <p>The “initial architecture” in this case is the structure of the system at the outset of the point-of-sale evolution, before any of the modernizations described in the material had been effected.</p>
<p><b>Present</b> Abbreviation: init-p</p>	<p><i>Description:</i> This category should be applied when the software element is present in the initial architecture, at the outset of the evolution.</p>
<p><b>Absent</b> Abbreviation: init-a</p>	<p><i>Description:</i> This category should be applied when the software element is not yet present in the initial architecture, at the outset of the evolution.</p>

<b>Presence in target architecture</b> Abbreviation: targ	<i>Description:</i> Subcategories of this category are used to indicate whether a software element is present in the initial architecture of the system. The same rules apply here as to the “Presence in initial architecture category”: software elements should be assigned exactly one subcategory, and other coding units should not be assigned any. The “target architecture” is the structure of the system at the conclusion of the point-of-sale evolution, once all currently planned changes have been made. (However, speculative changes that are described as possibly occurring in the distant future, but for which no concrete plans currently exist, should not be considered as part of this evolution.)
<b>Present</b> Abbreviation: targ-p	<i>Description:</i> This category should be applied when the software element is present in the target architecture, at the conclusion of the evolution.
<b>Absent</b> Abbreviation: targ-a	<i>Description:</i> This category should be applied when the software element is no longer present in the target architecture, at the conclusion of the evolution.

## References

- [1] M. S. Ball, G. W. H. Smith (1992). *Analyzing Visual Data*. Sage. ISBN 0-8039-3434-3.
- [2] S. Balsamo, A. Di Marco, P. Inverardi, M. Simeoni (2004). “Model-Based Performance Prediction in Software Development: A Survey.” *IEEE Transactions on Software Engineering* **30**(5): 295–310. doi:10.1109/TSE.2004.9
- [3] J. M. Barnes (2012). “NASA’s Advanced Multimission Operations System: A Case Study in Software Architecture Evolution.” Proceedings of the International ACM SIGSOFT Conference on the Quality of Software Architectures (QoSA), pp. 3–12. ACM. ISBN 978-1-4503-1346-9. doi:10.1145/2304696.2304700
- [4] J. M. Barnes, D. Garlan (2013). “Challenges in Developing a Software Architecture Evolution Tool as a Plug-In.” Proceedings of the Workshop on Developing Tools as Plug-Ins (TOPI), pp. 13–18. IEEE. ISBN 978-1-4673-6288-7.
- [5] J. M. Barnes, D. Garlan, B. Schmerl (in press). “Evolution Styles: Foundations and Models for Software Architecture Evolution.” *Software and Systems Modeling*. doi:10.1007/s10270-012-0301-9
- [6] J. M. Barnes, A. Pandey, D. Garlan (2013). “Automated Planning for Software Architecture Evolution.” Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). To appear.
- [7] L. Bass, P. Clements, R. Kazman (2003). *Software Architecture in Practice*. Addison–Wesley, 2nd edn. ISBN 0-321-15495-9.
- [8] P. Bell (2001). “Content Analysis of Visual Images.” In *Handbook of Visual Analysis* (eds. T. van Leeuwen, C. Jewitt), pp. 10–34. Sage. ISBN 0-7619-6476-2.
- [9] P. Bengtsson, N. Lassing, J. Bosch, H. van Vliet (2004). “Architecture-Level Modifiability Analysis (ALMA).” *Journal of Systems & Software* **69**: 129–147. doi:10.1016/S0164-1212(03)00080-3
- [10] B. Berelson (1952). *Content Analysis in Communication Research*. Free Press.
- [11] M. Beyman (2012). “No-Frills Retail Revolution Leads to Costco Wholesale Shopping Craze.” *NBCNews.com*. <http://www.nbcnews.com/business/no-frills-retail-revolution-leads-costco-wholesale-shopping-craze-734271>
- [12] M. Boyle, S. Shannon (2011). “Wal-Mart Brings Back 8,500 Products in Bid to End U.S. Slump.” *Bloomberg.com*. <http://www.bloomberg.com/news/2011-04-11/wal-mart-brings-back-8-500-products-in-bid-to-end-u-s-slump.html>

- [13] R. Brcina, M. Riebisch (2008). "Architecting for Evolvability by Means of Traceability and Features." Proceedings of the International ERCIM Workshop on Software Evolution and Evolvability (Evol), pp. 72–81. IEEE. ISBN 978-1-4244-2776-5. doi:10.1109/ASEW.2008.4686323
- [14] H. P. Breivold, I. Crnkovic, M. Larsson (2012). "A Systematic Review of Software Architecture Evolution Research." *Information and Software Technology* **54**(1): 16–40. doi:10.1016/j.infsof.2011.06.002
- [15] M. Bucholtz (2000). "The Politics of Transcription." *Journal of Pragmatics* **32**(10): 1439–1465. doi:10.1016/S0378-2166(99)00094-6
- [16] E. G. Carmines, R. A. Zeller (1979). *Reliability and Validity Assessment*. No. 07-017 in Quantitative Applications in the Social Sciences, Sage. ISBN 0-8039-1371-0.
- [17] S. Chaki et al. (2004). "State/Event-Based Software Model Checking." Proceedings of the International Conference on Integrated Formal Methods (IFM), LNCS, vol. 2999, pp. 128–147. Springer. ISBN 978-3-540-24756-2. doi:10.1007/978-3-540-24756-2\_8
- [18] N. Chapin (2000). "Do We Know What Preventive Maintenance Is?" Proceedings of the International Conference on Software Maintenance (ICSM), pp. 15–17. IEEE. ISBN 0-7695-0753-0. doi:10.1109/ICSM.2000.882970
- [19] N. Chapin et al. (2001). "Types of Software Evolution and Software Maintenance." *Journal of Software Maintenance and Evolution: Research and Practice* **13**(1): 3–30. doi:10.1002/smr.220
- [20] P. Clements et al. (2011). *Documenting Software Architectures: Views and Beyond*. Addison–Wesley, 2nd edn. ISBN 0-321-55268-7.
- [21] J. Cohen (1960). "A Coefficient of Agreement for Nominal Scales." *Educational and Psychological Measurement* **20**(1): 37–46. doi:10.1177/001316446002000104
- [22] G. A. Cornstock, E. A. Rubinstein, eds. (1972). *Television and Social Behavior: Reports and Papers*, vol. I. National Institute of Mental Health. <http://www.eric.ed.gov/ERICWebPortal/detail?accno=ED059623>
- [23] Costco (2012). "Form 10-K for the Fiscal Year Ended September 2, 2012." SEC Accession No. 0001193125-12-428890. <http://edgar.secdatabase.com/1732/119312512428890/filing-main.htm>
- [24] Costco (2013). "Form 10-Q for the Quarterly Period Ended May 12, 2013." SEC Accession No. 0001445305-13-001463. <http://edgar.secdatabase.com/2430/144530513001463/filing-main.htm>
- [25] G. Cowan, M. O'Brien (1990). "Gender and Survival vs. Death in Slasher Films: A Content Analysis." *Sex Roles* **23**(3/4): 187–196. doi:10.1007/BF00289865
- [26] D. Crockford (2006). "The application/json Media Type for JavaScript Object Notation (JSON)." RFC 4627, IETF. <http://www.ietf.org/rfc/rfc4627>
- [27] C. Del Rosso, A. Maccari (2007). "Assessing the Architectonics of Large, Software-Intensive Systems Using a Knowledge-Based Approach." Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA), paper 2. IEEE. ISBN 0-7695-2744-2. doi:10.1109/WICSA.2007.17
- [28] H. Dixon et al. (2008). "Portrayal of Tanning, Clothing Fashion and Shade Use in Australian Women's Magazines, 1987–2005." *Health Education Research* **23**(5): 791–802. doi:10.1093/her/cym057
- [29] S. Ducasse, D. Pollet (2009). "Software Architecture Reconstruction: A Process-Oriented Taxonomy." *IEEE Transactions on Software Engineering* **35**: 573–591. doi:10.1109/TSE.2009.19
- [30] Ecma International (2011). *Standard ECMA-262: ECMAScript Language Specification*, 5.1 edn. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>
- [31] J. L. Fleiss, B. Levin, M. C. Paik (2003). *Statistical Methods for Rates and Proportions*. Wiley, 3rd edn. ISBN 0-471-52629-0.
- [32] R. Flesch (1948). "A New Readability Yardstick." *Journal of Applied Psychology*, **32**(3). doi:10.1037/h0057532
- [33] D. G. Freelon (2010). "ReCal: Intercoder Reliability Calculation as a Web Service." *International Journal of Internet Science* **5**(1): 20–33. [http://www.ijis.net/ijis5\\_1/ijis5\\_1\\_freelon\\_pre.html](http://www.ijis.net/ijis5_1/ijis5_1_freelon_pre.html)

- [34] D. Garlan, J. M. Barnes, B. Schmerl, O. Celiku (2009). "Evolution Styles: Foundations and Tool Support for Software Architecture Evolution." Proceedings of the Joint Working IEEE/IFIP Conference on Software Architecture & European Conference on Software Architecture (WICSA/ECSA), pp. 131–140. IEEE. ISBN 978-1-4244-4984-2. doi:10.1109/WICSA.2009.5290799
- [35] D. Garlan, R. Monroe, D. Wile (1997). "Acme: An Architecture Description Interchange Language." Proceedings of the Conference of the Centre for Advanced Studies on Collaborative Research (CASCON), pp. 169–183. ACM. doi:10.1145/1925805.1925814
- [36] D. Garlan, B. Schmerl (2009). "Ævol: A Tool for Defining and Planning Architecture Evolution." Proceedings of the International Conference on Software Engineering (ICSE), pp. 591–594. IEEE. ISBN 978-1-4244-3452-7. doi:10.1109/ICSE.2009.5070563
- [37] S. S. Gokhale (2007). "Architecture-Based Software Reliability Analysis: Overview and Limitations." *IEEE Transactions on Dependable and Secure Computing* **4**(1): 32–40. doi:10.1109/TDSC.2007.4
- [38] N. Groeben, R. Rustemeyer (1994). "On the Integration of Quantitative and Qualitative Methodological Paradigms (Based on the Example of Content Analysis)." In *Trends and Perspectives in Empirical Social Research* (eds. I. Borg, P. P. Mohler), pp. 308–326. Walter de Gruyter. ISBN 3-11-014312-7.
- [39] K. L. Gwet (2012). *Handbook of Inter-Rater Reliability*. Advanced Analytics, 3rd edn. ISBN 978-0-9708062-7-7.
- [40] ISO (2006). *International Standard ISO/IEC 14764: Software Engineering—Software Life Cycle Processes—Maintenance*. 2nd edn. ISBN 0-7381-4961-6.
- [41] A. Jaffe (2012). "Transcription in Practice: Nonstandard Orthography." In *Orthography as Social Action: Scripts, Spelling, Identity and Power* (eds. A. Jaffe, J. Androutsopoulos, M. Sebba, S. Johnson), pp. 203–224. De Gruyter. ISBN 978-1-61451-136-6.
- [42] P. Jamshidi, M. Ghafari, A. Ahmad, C. Pahl (2013). "A Framework for Classifying and Comparing Architecture-Centric Software Evolution Research." Proceedings of the European Conference on Software Maintenance and Reengineering (CSMR), pp. 305–314. IEEE. ISBN 978-0-7695-4948-4. doi:10.1109/CSMR.2013.39
- [43] G. Jefferson (2004). "Glossary of Transcript Symbols with an Introduction." In *Conversation Analysis: Studies from the First Generation* (ed. G. H. Lerner), pp. 13–31. John Benjamins. ISBN 90-272-5367-6.
- [44] J. Kirk, M. L. Miller (1986). *Reliability and Validity in Qualitative Research*. Sage. ISBN 0-8039-2560-3.
- [45] S. Kracauer (1952). "The Challenge of Qualitative Content Analysis." *Public Opinion Quarterly* **16**(4): 631–642. doi:10.1086/266427
- [46] K. Krippendorff (1970). "Bivariate Agreement Coefficients for Reliability Data." *Sociological Methodology* **2**: 139–150.
- [47] K. Krippendorff (2004). "Reliability in Content Analysis: Some Common Misconceptions and Recommendations." *Human Communication Research* **30**(3): 411–433. doi:10.1111/j.1468-2958.2004.tb00738.x
- [48] K. Krippendorff (2013). *Content Analysis: An Introduction to Its Methodology*. Sage, 3rd edn. ISBN 1-4129-8315-0.
- [49] S. Kvale, S. Brinkmann (2009). *InterViews: Learning the Craft of Qualitative Research Interviewing*. Sage, 2nd edn. ISBN 978-0-7619-2542-2.
- [50] N. Lassing, D. Rijssenbrij, H. van Vliet (1999). "Towards a Broader View on Software Architecture Analysis of Flexibility." Proceedings of the Asia Pacific Software Engineering Conference (APSEC), pp. 238–245. IEEE. ISBN 0-7695-0509-0. doi:10.1109/APSEC.1999.809608
- [51] M. D. LeCompte, L. P. Goetz (1982). "Problems of Reliability and Validity in Ethnographic Research." *Review of Educational Research* **52**(1): 31–60. doi:10.3102/00346543052001031
- [52] B. P. Lientz, E. B. Swanson (1980). *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations*. Addison-Wesley. ISBN 0-201-04205-3.

- [53] Y. S. Lincoln, E. G. Guba (1985). *Naturalistic Inquiry*. Sage. ISBN 0-8039-2431-3.
- [54] M. Lombard, J. Snyder-Duch, C. C. Bracken (2002). "Content Analysis in Mass Communication: Assessment and Reporting of Intercoder Reliability." *Human Communication Research* **28**(4): 587–604. doi:10.1111/j.1468-2958.2002.tb00826.x
- [55] M. Lombard, J. Snyder-Duch, C. C. Bracken (2003). "Correction." *Human Communication Research* **29**(3): 469–472. doi:10.1111/j.1468-2958.2003.tb00850.x
- [56] L. M. MacLean, M. Meyer, A. Estable (2004). "Improving Accuracy of Transcripts in Qualitative Research." *Qualitative Health Research* **14**(1): 113–123. doi:10.1177/1049732303259804
- [57] P. Mayring (2000). "Qualitative Content Analysis." *Forum: Qualitative Social Research*, **1**(2). <http://nbn-resolving.de/urn:nbn:de:0114-fqs0002204>
- [58] P. Mayring (2002). "Qualitative Content Analysis—Research Instrument or Mode of Interpretation?" In *The Role of the Researcher in Qualitative Psychology* (ed. M. Kiegelmann), pp. 140–149. Ingeborg Huber. ISBN 3-9806975-3-3. <http://psydok.sulb.uni-saarland.de/volltexte/2007/943/>
- [59] P. Mayring (2010). *Qualitative Inhaltsanalyse: Grundlagen und Techniken*. Beltz, 11th edn. ISBN 978-3-407-25533-4.
- [60] E. McLellan-Lemal (2008). "Transcribing Data for Team-Based Research." In *Handbook for Team-Based Qualitative Research* (eds. G. Guest, K. M. MacQueen), pp. 101–118. AltaMira. ISBN 0-7591-0910-9.
- [61] J. U. McNeal, M. F. Ji (2003). "Children's Visual Memory of Packaging." *Journal of Consumer Marketing* **20**(5): 400–427. doi:10.1108/07363760310489652
- [62] I. Mero-Jaffe (2011). "'Is That What I Said?' Interview Transcript Approval by Participants: An Aspect of Ethics in Qualitative Research." *International Journal of Qualitative Methods* **10**(3): 231–247. <http://ejournals.library.ualberta.ca/index.php/IJQM/article/view/8449>
- [63] M. B. Miles, A. M. Huberman (1994). *Qualitative Data Analysis: An Expanded Sourcebook*. Sage, 2nd edn. ISBN 0-8039-5540-5.
- [64] F. Mosteller, D. L. Wallace (1963). "Inference in an Authorship Problem." *Journal of the American Statistical Association* **58**(302): 275–309. doi:10.1080/01621459.1963.10500849
- [65] N. Müller, J. S. Damico (2002). "A Transcription Toolkit: Theoretical and Clinical Considerations." *Clinical Linguistics & Phonetics* **16**(5): 299–316. doi:10.1080/02699200210135901
- [66] K. A. Neuendorf (2002). *The Content Analysis Guidebook*. Sage. ISBN 0-7619-1978-3.
- [67] D. G. Oliver, J. M. Serovich, T. L. Mason (2005). "Constraints and Opportunities with Interview Transcription: Towards Reflection in Qualitative Research." *Social Forces* **84**(2): 1273–1289. doi:10.1353/sof.2006.0023
- [68] I. Ozkaya, P. Wallin, J. Axelsson (2010). "Architecture Knowledge Management during System Evolution—Observations from Practitioners." Proceedings of the Workshop on Sharing and Reusing Architectural Knowledge (SHARK), pp. 52–59. ACM. ISBN 978-1-60558-967-1. doi:10.1145/1833335.1833343
- [69] R. Popping (1988). "On Agreement Indices for Nominal Data." In *Sociometric Research* (eds. W. E. Saris, I. N. Gallhofer), vol. 1, pp. 90–105. St. Martin's Press. ISBN 0-312-00419-2.
- [70] W. R. Powers (2005). *Transcription Techniques for the Spoken Word*. AltaMira. ISBN 0-7591-0843-9.
- [71] D. Riffe, S. Lacy, F. G. Fico (1998). *Analyzing Media Messages: Using Quantitative Content Analysis in Research*. Lawrence Erlbaum Associates. ISBN 0-8058-2018-3.
- [72] J. Ritsert (1972). *Inhaltsanalyse und Ideologiekritik: Ein Versuch über kritische Sozialforschung*. Athenäum. ISBN 3-7610-5801-2.
- [73] E. M. Rogers (1997). *A History of Communication Study: A Biographical Approach*. Free Press, 1st paperback edn. ISBN 0-684-84001-4.
- [74] J. Saldaña (2013). *The Coding Manual for Qualitative Researchers*. Sage, 2nd edn. ISBN 978-1-4462-4737-2.

- [75] M. Sandelowski (1994). "Notes on Transcription." *Research in Nursing & Health* **17**(4): 311–314. doi:10.1002/nur.4770170410
- [76] B. Schmerl, D. Garlan (2004). "AcmeStudio: Supporting Style-Centered Architecture Development." Proceedings of the International Conference on Software Engineering (ICSE), pp. 704–705. IEEE. ISBN 0-7695-2163-0. doi:10.1109/ICSE.2004.1317497
- [77] M. Schreier (2012). *Qualitative Content Analysis in Practice*. Sage. ISBN 978-1-84920-593-1.
- [78] I. S. Schwartz, D. M. Baer (1991). "Social Validity Assessments: Is Current Practice State of the Art?" *Journal of Applied Behavior Analysis* **24**(2): 189–204. doi:10.1901/jaba.1991.24-189
- [79] W. A. Scott (1955). "Reliability of Content Analysis: The Case of Nominal Scale Coding." *Public Opinion Quarterly* **19**(3): 321–325. doi:10.1086/266577
- [80] C. Seale (1999). *The Quality of Qualitative Research*. Sage. ISBN 0-7619-5597-6.
- [81] I. Steinke (2004). "Quality Criteria in Qualitative Research." In *A Companion to Qualitative Research* (eds. U. Flick, E. von Kardorff, I. Steinke), pp. 184–190. Sage. ISBN 0-7619-7375-3.
- [82] B. Stone (2013). "Costco CEO Craig Jelinek Leads the Cheapest, Happiest Company in the World." *Bloomberg Businessweek*. <http://www.businessweek.com/articles/2013-06-06/costco-ceo-craig-jelinek-leads-the-cheapest-happiest-company-in-the-world>
- [83] STORES Media (2013). "2012 Top 250 Global Retailers." <http://www.stores.org/2012/Top-250-List>
- [84] STORES Media (2013). "2013 Top 100 Retailers." [http://www.stores.org/2013/Top-100-Retailers?order=field\\_revenue\\_value&sort=desc](http://www.stores.org/2013/Top-100-Retailers?order=field_revenue_value&sort=desc)
- [85] E. B. Swanson (1976). "The Dimensions of Maintenance." Proceedings of the International Conference on Software Engineering (ICSE), pp. 492–497. IEEE.
- [86] D. Waples, ed. (1942). *Print, Radio, and Film in a Democracy*. Univ. of Chicago Press.
- [87] R. P. Weber (1985). *Basic Content Analysis*. No. 07-049 in Quantitative Applications in the Social Sciences, Sage. ISBN 0-8039-2448-8.
- [88] B. J. Williams, J. C. Carver (2010). "Characterizing Software Architecture Changes: A Systematic Review." *Information and Software Technology* **52**(1): 31–51. doi:10.1016/j.infsof.2009.07.002
- [89] M. M. Wolf (1978). "Social Validity: The Case for Subjective Measurement, or How Applied Behavior Analysis Is Finding Its Heart." *Journal of Applied Behavior Analysis* **11**(2): 203–214. doi:10.1901/jaba.1978.11-203
- [90] R. K. Yin (2009). *Case Study Research: Design and Methods*. Sage, 4th edn. ISBN 978-1-4129-6099-1.