Architecture Design in Global and Model-Centric Software Development

Werner Heijstek

Architecture Design in Global and Model-Centric Software Development

Proefschrift

ter verkrijging van
de graad van Doctor aan de Universiteit Leiden,
op gezag van Rector Magnificus Prof.mr. P.F. van der Heijden
volgens besluit van het College voor Promoties
te verdedigen op woensdag 5 december 2012
klokke 10:00 uur

door

Werner Heijstek
geboren te Dordrecht
in 1982

Promotiecommissie

Promotor	:	Prof. dr. J.N. Kok	
Copromotor	:	Dr. M.R.V. Chaudron	
Overige Leden	:	Prof. dr. Th.H.W. Bäck	
		Prof. dr. P. Runeson	(*Lund University*)
		Prof. dr. ir. P. Avgeriou	(*University of Groningen*)

The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

Parts of this research were supported by the Accelerated Delivery Center (ACD) of Capgemini the Netherlands.

Parts of this research were supported by the Israeli Ministry of Foreign Affairs Division for Cultural and Scientific Affairs via the Netherlands Organization for International Cooperation in Higher Education (NUFFIC).

Parts of this research were supported by the Platform Outsourcing the Netherlands (PON).

Images used on the cover:

(1) *"Carte des Indes orientales I. feuille, dans la quelle on represente les Indes deca la Riviere de Ganges, le Golfe de Bengale, Siam, Malacca, Sumatra dressee par Mr. de Tobie Mayer de la Societe Geograph."* (1748)

(2) *"AIM UML Class diagram."* from "The caBIG™ Annotation and Image Markup Project" D. S. Channin et al. Journal of Digital Imaging Vol. 23 Issue 2 (2010)

Typeset by LaTeX, printed by Ipskamp Drukkers.
ISBN: 978-94-6191-525-2

"Einstein argued that there must be simplified explanations of nature, because God is not capricious or arbitrary. No such faith comforts the software engineer. Much of the complexity that he must master is arbitrary complexity, forced without rhyme or reason by the many human institutions and systems to which his interfaces must conform. These differ from interface to interface, and from time to time, not because of necessity but only because they were designed by different people, rather than by God."

— Frederick O. Brooks, Jr.

Contents

Chapter 1

Introduction

In this chapter the concepts central to this dissertation as well as the study motivation and objectives are discussed. In addition, the research approach and associated research methods are outlined.

1.1 Central Concepts

At the first NATO Software Engineering Conference in 1968 in Garmisch, Germany, it was established that a more structured approach to software development was required to battle the "software crisis" – software of poor quality resulting from late and canceled projects in which large software systems were built:

> *"The general admission of the existence of the software failure in this group of responsible people is the most refreshing experience I have had in a number of years, because the admission of shortcomings is the primary condition for improvement."*
> — E.W. Dijkstra (Naur and Randell, 1968)

Software development was thenceforth regarded as a profession ("software engineering"). While not completely new, this neologism reflected a desire to approach the design and implementation of software as systematically and rigorous as civil engineers go about constructing a bridge or a skyscraper.

The concepts central to this dissertation are all aimed at contributing to this strive towards more structured development of software systems: Software architecture (clearly an analogy derived from the civil engineering discipline) strives to the structured design of software systems — much like its namesake counterpart. The Rational Unified Process (RUP) is a collection of best practices in an integrated process framework. The closely related Unified Modeling Language (UML) aims to standardize software modeling to better align stakeholder requirements. In line with UML, Model-Driven

Development (MDD) introduces the notion of abstraction from implementation details by means of modeling with high-level development languages.

Global Software Development (GSD), software development taking place at geographically separated locations, is driven by considerations such as development speed, software quality and a sheer lack of educated software engineers (notably the case for the Netherlands). GSD is very common. Salger (2009) even refers to GSD as *"the new standard mode of software development."* In the following sections GSD, software architecture, RUP, UML and MDD will be elaborated on in more detail.

1.1.1 Global Software Development

Global software development (GSD)[1] can be defined as:

> *"Software work undertaken at geographically separated locations across national boundaries in a coordinated fashion involving real time (synchronous) and asynchronous interaction"* (Sahay et al., 2003)

Many motivations exist for geographically distributing software development activities. A popular and commonly mentioned argument is development cost reduction (cf. Šmite et al., 2010): Developer wages in emerging economies are much lower than they are in Western Europe and the United States (the main sources of GSD work). Other motivations, often mentioned in combination with or even secondary to cost reductions, include (mostly derived from Conchúir et al., 2009):

- *Improved software design modularization*
  Various distributed software development teams can work on different components of a single software system. This enhances component cohesion and (by definition) decreases coupling (Ebert and De Neve, 2001, Grinter et al., 1999) — both seen as desirable traits from the perspective of software quality (Eder et al., 1994).

- *Leveraging time zone shifts*
  By adopting the so-called "follow-the-sun" development model, more hours can be worked on a working day. In theory, this enables schedule compression (Carmel and Agarwal, 2001, Herbsleb and Grinter, 1999b, Herbsleb et al., 2000).

[1] GSD is synonymous to *distributed* software development and global software engineering (GSE) — both often encountered in scientific literature. Outside scientific literature, sourcing and (the more specific term) outsourcing are often incorrectly used to denote GSD. Where outsourcing may refer to any work done by external entities, GSD particularly pertains to the outsourcing of software development work. Another commonly encountered hyponym of outsourcing is *offshoring*. Offshoring specifically defines the distance to the outsourcing destination to be significant (overseas). The neologisms farshoring and (antonym) nearshoring both aim to clarify the relative degree of this distance.

- *Access to larger skilled labor pool*
  Many organizations are confronted with limited software engineer availability and are forced to look beyond their regional or national borders. GSD provides the opportunity to utilize the vast amounts of software engineering graduates in countries such as India and Brazil (Carmel and Agarwal, 2001, Herbsleb et al., 2000).

- *Closer proximity to market and client*
  GSD can be employed to gain proximity to a market or client which may be beneficial for several reasons: A software firm may seek to create a more "intimate relationship" with a client (Porter, 1985). The objective might be to be better able to localize software for local markets (Grinter et al., 1999, Herbsleb et al., 2000). Another reason might be that an organization maneuvers into a (local) position for merger or acquisition.

- *Innovation and shared best practice*
  Software engineers from diverse backgrounds could share their approaches and bring diverse and novel solutions to problems.

Not all these potential benefits materialize: Due to significant overhead in communication, travel cost, additional resources required for governance and e.g. turnover at the offshore site (Carmel and Tjia, 2005), potential cost reductions are often not realized (Conchúir et al., 2009). Instead of increasing software development speed, time zone differences introduce a set of problems due to limited windows for collaboration (Carmel and Agarwal, 2001, Holmström et al., 2006, Conchúir et al., 2009). And according to Conchúir et al. (2009), *"[e]mployees who feel threatened by low-wage colleagues are unlikely to share more than necessary to get the job done,"* thereby mitigating any potential benefits related to shared best practice. Nevertheless, GSD is increasingly the rule rather than the exception.

While Sahay et al. explicitly include the crossing of national boundaries in their definition, some of the problems associated with GSD arise when software engineering project team members are separated as little as 30 meters (Allen, 1977). The "global" scale of GSD is regarded as adding additional complexity as it is found to introduce three different notions of distance (Carmel and Agarwal, 2001, Ågerfalk et al., 2005, Holmström et al., 2006):

- *Geographical distance*
  refers to the physical distance between the development locations. As this distance increases, the opportunity for co-located teamwork reduces and also becomes progressively harder to organize.

- *Temporal distance*
  refers to time zone differences between development sites. Team working hour overlap (in terms of potential for synchronous collaboration) decreases as this

distance increases. Note that geographical and temporal distance are only related if the geographical distance has a longitudinal component.

- *Socio-cultural distance*
  refers to differences in beliefs, norms, values and customs. As this distance increases it becomes increasingly difficult to communicate effectively due to misunderstandings resulting from misinterpretations of language and behavior. Distances between different national and organizational cultures can be measured by plotting them on a set of dimensions including individualism (the degree to which a culture is individualistic or more group-oriented) and "power distance" (the degree to which a culture is hierarchical or more egalitarian) (Hofstede, 1984, Trompenaars and Prud'homme van Reine, 2004).

Solutions for the problems associated with distance so far include (Herbsleb and Grinter, 1999a, Carmel and Agarwal, 2001, Herbsleb et al., 2005):

- *Strategic selection of the offshore location*
  To overcome one or more distances the offshore location can be selected based on limited geographical distance (e.g. Amsterdam — Warsaw), limited temporal distance (e.g. Amsterdam — Capetown) or limited socio-cultural distance (e.g. Amsterdam — Tel Aviv). In some instances, all three distances can be overcome to some extent (e.g. Seattle — Vancouver).

- *Limit the need for intense collaboration*
  To limit the impact of the three distances, one can divide the system development work in such a way that different development sites are not required to collaborate much.

- *Assign a liaison*
  To reduce the socio-cultural divide between development teams, a liaison can be appointed who regularly physically visits the locations.

- *Plan so that synchronous communication takes place frequently*
  Asynchronous communication methods such as e-mail and fora introduce delays and offer limited opportunity to ensure messages have been properly understood. Increasing temporal distance makes for a more challenging process to plan meetings by means of telephone or a video-link. Not all synchronous communication means are equally useful. Socio-cultural distance hampers direct communicated to a lesser extent when richer communication media such as video are used (Isaacs and Tang, 1994).

1.1.2 Software Architecture

Much in line with the ambition to mimic the professional approach of the engineering disciplines, the subdiscipline of software architecture specifically focuses on the design

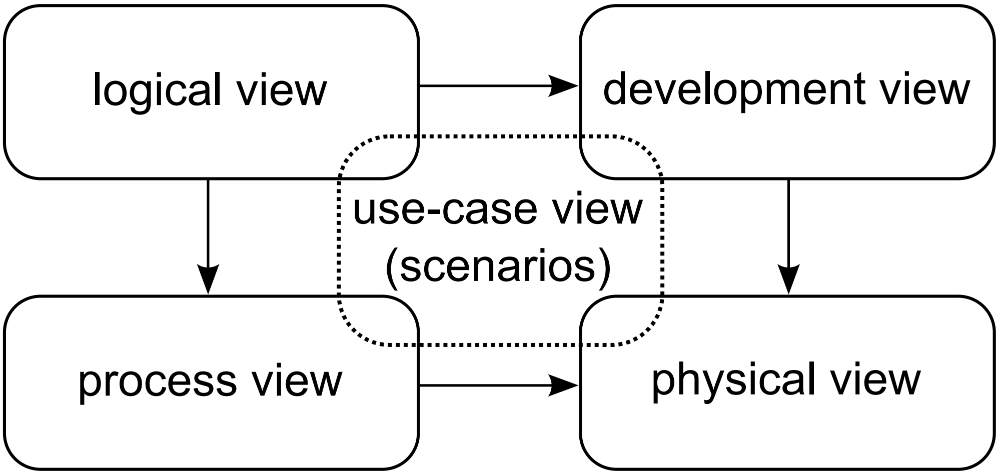


Figure 1.1: *The* 4 + 1*-view model*

of the main structure of a software system using tried and tested principles. While the term software architecture has been around as long as the term software engineering, only in the past two decades or so has it received much attention as an academic field of study. The many definitions that have been proposed since the early stages of this "software architecture renaissance" (Kogut and Clements, 1994) have in common the notion of components and their interconnections derived from Perry and Wolf (1992). For example, a commonly accepted definition of software architecture is given in ISO/IEC/IEEE standard 42010 (ISO/IEC/IEEE, 2011) (a standard that has recently superseded IEEE 1471 (IEEE, 2000)): Software architecture is

> *"[t]he fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution."*(ISO/IEC/IEEE, 2011)

Because it concerns design at the software component level (as opposed to the design of those components themselves), software architecture is a pivotal vehicle to address and guarantee non-functional requirements such as security, maintainability, extendability and portability. Since the interest in software architecture research has increased, several important concepts were introduced. First, the influential 4 + 1-view model (Figure 1.1, Kruchten, 1995) expounded that, for representational clarity and the purpose of completeness, a software architecture is to be described according to predefined views. These views are defined so that they each accommodate the different issues that stakeholders have. The feedback that these stakeholders are then able to give is thought to benefit the fitness and other general design aspects of the software architecture.

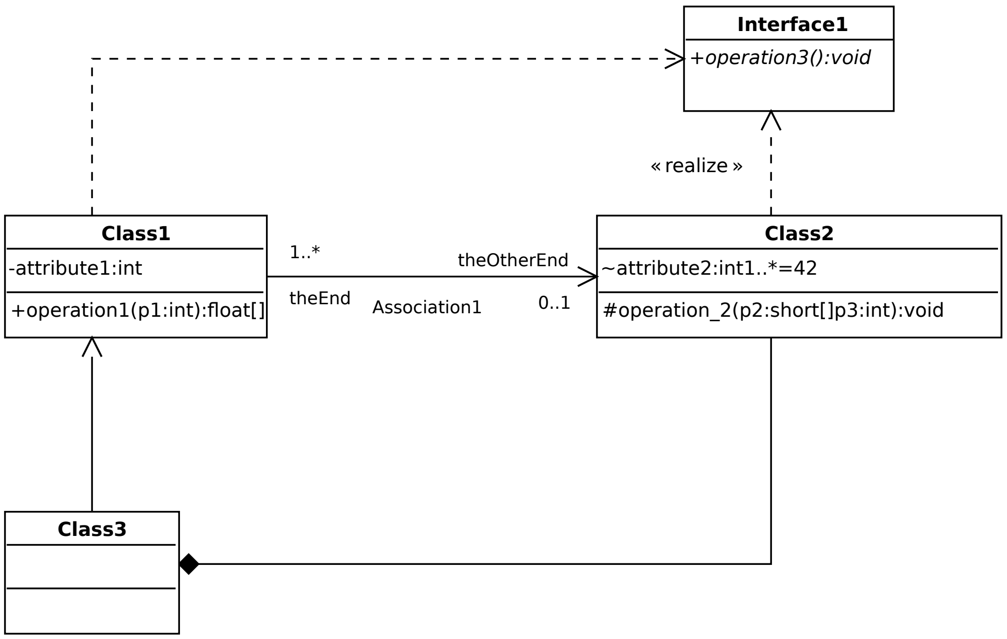


Figure 1.2: *Example of a UML class diagram*

Second, software architecture representation is thought to benefit from the inclusion of a clear rationale for its design (e.g. Bratthall et al., 2000). As a result, software architecture is seen as a collection of "design decisions". If the rationale for these design decisions changes (because of requirement changes or changes in the system's environment), so might the eventual software architecture design.

Third, software architecture is no longer only thought of as a representation of the prescription of the organization of the system components *to be* (an artifact). Software architecture also comprises activities related to the process of designing and communicating that architecture as well as ensuring eventual compliance (a process).

1.1.3 The Unified Modeling Language

The Unified Modeling Language (UML) is a graphical modeling language geared towards modeling object-oriented software systems (Rumbaugh et al., 1990). The first version (1.0) of the UML was the result of combining the object modeling technique (OMT, Rumbaugh et al., 1990), object-oriented software engineering (OOSE, Jacobson et al., 1992) and the modeling language Booch (Booch, 1995a). In 1997, it was proposed as a standard to the Object Media Group (OMG) which it became several years later (ISO/IEC, 2005). The most recent version of the UML (Object Management Group, 2011) was released in August 2011 and consists of 14 diagram types which are divided into structural and behavioral diagrams. The most commonly used diagram types are (Dobing and Parsons, 2006):

- The *class diagram*
  A structure diagram that outlines the relations between the classes (entities) in a system. An example of a class diagram is depicted in Figure 1.2.

- The *use case diagram*
  A structure diagram that depicts an overview of a system in terms of the relation between various required usages of a system.

- The *sequence diagram*
  A behavior diagram in which a sequence of messages between instantiations of objects are modeled.

UML is widely used throughout industry and has abundant tool support. In addition, UML is prescribed to be used in RUP (see Section 1.1.4). In practice, UML is used for a variety of purposes. Ordered from informal to formal in the sense of diagram completeness and adherence to the UML standard, these are:

- As a *sketch*
  Developers can use the notational elements of UML to quickly draw part of a system for comprehension and communication purposes. For sketches, UML elements such as classes and actors might be used in combination with informal or domain-specific constructs (Cherubini et al., 2007).

- For *communication of system design*
  By modeling parts of a system one can explain e.g. how a system component is supposed to function. Depending on how much of a system is modeled, this approach can be a form of *model-centric development*.

- As a *blueprint*
  In this case, most system analysis and design has been done and the resulting set of UML diagrams then is to be used for implementation. This type of development approach is referred to as model-centric development. UML is likely to be used as a blueprint in the context of GSD where design and coding activities take place at different geographical locations.

- As a *programming language*
  UML diagrams can be used to generate code. The UML diagrams must strictly adhere to a predefined syntax. This type of development approach is referred to as Model-Driven Development (MDD, see Section 1.1.5).

1.1.4 The Rational Unified Process

The Rational Unified Process (RUP) is an adaptable, architecture-centric, risk-driven process framework that is commonly used in software engineering practice. It provides a disciplined and iterative approach to the assignment of tasks and responsibilities in software development projects. RUP is offered as a set of tools of which the most important is a hyperlinked knowledge base with sample artifacts and detailed descriptions for many types of activities of the software engineering process.

RUP is the product of a development process at Rational Software in the 1980s and the 1990s, based on Boehm's spiral model (Boehm, 1986, 1988) and a development process at the Swedish company Objectory AB, based on the Objectory Process developed by Jacobson (Jacobson, 1987, Jacobson et al., 1992). The merged process — initially called the *Rational Objectory Process* — was described in detail by Kruchten (2003b). Currently, the framework is owned by IBM[2] and is offered as a part of the IBM *Rational Method Composer*[3] that allows customization of the process.

In RUP, software engineering processes are organized into phases and iterations. A project consists of four phases which correspond with the first four main stages of the waterfall model: requirements definition, system and software design, implementation and unit testing, and integration and system testing (Ghezzi et al., 2002):

- During the *inception phase* the business case and the financial forecast are created as well as a use-case model, a risk assessment and project description.

- The *elaboration phase* is used to perform problem domain analysis and to shape the architecture.

- During the *construction phase* the development and integration of components are the central activities.

- Finally, the *transition phase* the software system that is developed will be implemented at the client's organization.

In RUP, the effort that is spent on activities is categorized into nine "disciplines". These disciplines are depicted in the iconic "RUP Hump" diagram (Figure 1.3):

1. The *business modeling* discipline is concerned with activities that bridge business and software engineering in order to understand business needs and to translate them to software solutions.

2. The *requirements* discipline is concerned with elicitation and organization of functionality and non-functional demands and aims to create a description for what the system should do.

[2]https://www-01.ibm.com/software/awdtools/rup/
[3]https://www-01.ibm.com/software/awdtools/rmc/

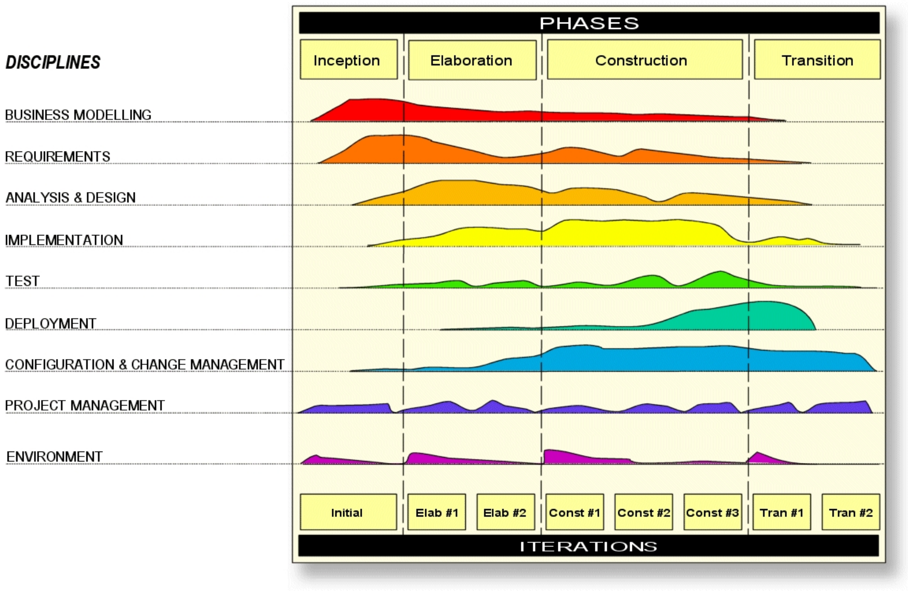


Figure 1.3: *A recent version of the RUP "hump" diagram*

3. The *analysis and design* discipline is concerned with the mapping of requirements to a formal design. The resulting design model acts as a input to the implementation. A modeling language such as UML can be used to design classes and structure them into packages with well-defined interfaces.

4. By means of activities that are part of the *implementation* discipline, the actual implementation of the components is made, either by reuse or by creation of new components.

5. The *test* discipline serves to verify the completeness and correctness of the implementation of the requirements. This discipline is also responsible for the elicitation of defects and their respective fixes.

6. The *deployment* discipline is concerned with product releases and end-user delivery of these releases.

7. Activities that fall in the *configuration and change management* discipline deal with change requests regarding project artifacts and models and version control of these changes.

8. The *project management* discipline focuses on progress monitoring of iterations through collection and analysis of metrics, planning iterations and management of risk.

9. The *environment* discipline aims at activities that facilitate the configuration of a project and project support in general by means of tools and supporting processes.

1.1.5 Model-Driven Development

Model-Driven Development (MDD, Selic, 2003) is a development paradigm in which models (instead of code) are the central development artifacts. By working with abstractions (models), the complexities of lower-level implementations are handled by a separate component (often a model-interpreter or code-generator). The fundamental idea behind MDD is not new but rather the natural continuation of the trend of raising the level of abstraction at which software is developed (Atkinson and Kühne, 2003). The core concepts behind MDD are:

Table 1.1: *MDD provides a common language that augments the individual representations each software engineering stakeholder uses*

stakeholder	**commonly used representation method**	**new representation method**
Client	natural language	Domain-Specific Modeling Language
Business Analyst	BPMN	
Requirements Engineer	use cases	
Designer	UML	
Software Architect	ADL	
Programmer	source code	

1. *Model-centrism*
   In MDD models, rather than code, are to be treated as first-class entities. This entails that software is designed and implemented using (often domain-specific) models as a primary vehicle.

2. *Code generation*
   To enable model-centrality over code, (significant) portions of code of the software implementation are generated from models.

3. *Model reuse*
   The use of models that are domain-specific implies that models can be reused for other software systems within the same domain.

An important (hypothesized) benefit of MDD is that the use of models as a central language increases the involvement of stakeholders as they all understand the models. This presumably is particularly so when the applied modeling language is tailored to describe a certain domain. Such a modeling language is called a Domain-Specific Language (DSL, Van Deursen et al., 2000). In practice, such a modeling language is often a subset of UML that is extended with domain-specific stereotypes. Such a DSL essentially replaces the "native language" of team members of different disciplines (Table 1.1). Each stakeholder is expected to understand and to express himself using the DSL. A programmer and the client — whose "native languages" are very different in code-centric development — both use the same constructs and can therefore directly communicate with one another.

1.2 Problem Statement

GSD projects are associated with increases in risk and complexity which aggravate their difficulty and failure rate (Sharma and Seshagiri, 2006). A common approach to GSD is a type of "transfer by development stage" (Mockus and Weiss, 2001) where requirements gathering and architecture design activities take place at a different geographical location than the implementation-related activities. To ensure compliance to software requirements, the software implementation must adhere to the software design. Much software design is thought to be disseminated informally. The possibilities for informal interaction between team members in GSD settings, however, is limited. In addition, offshore developers are often not able to directly contact a member of the design team due to geographical distance. Synchronous communication is often difficult due to time zone differences. Even if an architect can be contacted, communication can be hampered by socio-cultural differences such as language barriers.

Nevertheless, developer understanding of software design and its rationale is not only believed to benefit software quality (Soloway et al., 1988, Tilley and Huang, 2003, Hayes, 2003, Kotlarsky et al., 2008) but is imperative to ensure that a software implementation satisfies its requirements.

In summation: While software design is more challenging to communicate in a GSD context we are unsure how software design is developed, represented, communicated and coordinated in the context of GSD. In addition, we are unsure what the effects are on GSD of increasingly popular MDD approaches. In these approaches, design plays an even more central role.

1.3 Research Objectives

The objective of this dissertation is to investigate the role of software architecture design in global software development so to improve the success rate of GSD projects in terms of delivery on-time and within budget. The main research question is:

> *How can software architecture design be effectively represented, disseminated and coordinated in the context of model-centric and model-driven global software development ?*

This question is addressed in the context of custom software development as explicitly opposed to product software development. An important characteristic of custom software development is that the majority of its software architecture is project-specific and that team members mainly collaborate in the context of a single project.

To address the main research question, software architecture design is explored both as a process and as an artifact in the context of global, *model-centric* software development. In addition, as a case of (very) model-centric development, the effects of application of *model-driven* software development tools and techniques in the context of GSD are studied. We regard MDD as a special case of model-centric software development in which models are more centric — to the extent that they surpass source code as the primary development artifact. To this end, three sub-questions have been defined:

RQ1 How is software architecture represented, disseminated and coordinated in the context of global software development? *(software architecture as a process)*

RQ2 How can we design software architecture documentation so that it is understood well by developers in the context of global software development? *(software architecture as an artifact)*

RQ3 How does the application of model-driven development tools and techniques affect the problems associated with global software development? *(model-driven development)*

1.4 Research Methodology

Empiricism (from Ancient Greek — ἐμπειρία) is a philosophical doctrine that holds that knowledge is derived from experience (Locke, Berkeley, Hume; see e.g. Russell, 1945). Contrastingly, the doctrine of rationalism (Descartes, Leibniz, Spinoza; see e.g. Russell, 1945) maintains that knowledge is gained independently of experience. The empirical research paradigm aims to corroborate the presence of a causal relation between a cause and its hypothesized effect. Analysis of empirical observations relies on inductive reasoning — the practice of inferring principles or rules from observed facts.

In this section and the next, a distinction is made between empirical research methodologies, which offer a comprehensive and structured approach to address research objectives, and data collection techniques, which are used in the context of a particular research method. The empirical research methodologies in this thesis include case studies, grounded theory and a controlled experiment.

1.4.1 Case Studies

According to Runeson et al. (2012), a case study is *''an empirical enquiry that draws on multiple sources of evidence to investigate one instance (or a small number o instances) of a contemporary software engineering phenomenon within its real-life context, especially when the boundary between phenomenon and context cannot be clearly specified''*. The majority of studies reported on in this dissertation, apply the case study approach. Yin defines three types of case studies of which the explanatory case study (understanding the relation between a phenomenon and its causes) and exploratory case study (analyzing patterns in collected data) are used in this dissertation. The case study method is used in chapters 2, 3, 4 and 7. In designing, analyzing and reporting the findings of these case studies, the guidelines as prescribed by Stake (1995), Yin (2002) and Runeson and Höst (2009) were applied.

1.4.2 Controlled Experiments

An experiment is a formal, rigorous and controlled investigation in which the relation between an effect and a cause is addressed (Wohlin et al., 2000). In a controlled experiment the effect of a treatment on an experimental group is tested and compared to a control group for which the treatment was absent. If the experiment design is in line with the study objective and executed in line with methodological guidelines (such as described in Wohlin et al., 2000), obtained results are regarded as strong evidence. A disadvantage of using experiments is that because of the required exclusion of other variables (alternative explanations) the experimental setup has little semblance of industrial reality. This potentially limits the generalizability of results. A controlled experiment was used in Chapter 6.

1.4.3 Grounded Theory

Grounded theory is an analysis method that is geared towards theory development. Grounded theory (Strauss and Corbin, 1990) is rooted in the data that has been collected while observing a phenomenon by means of induction. An example of the associated method of data collection and analysis for a software project is schematically outlined in Figure 1.4. Grounded theory can be used to analyze data collected as part of a case study. The principles of grounded theory were followed in chapters 5 and 8.

1.5 Data Collection Techniques

The empirical paradigm requires observation of the subject – preferably in its natural habitat. Therefore, industrial software development projects were used as a source for most studies reported on in this dissertation. Obtaining valid industrial data requires the application of a set of procedures that include (but is by no means limited to)

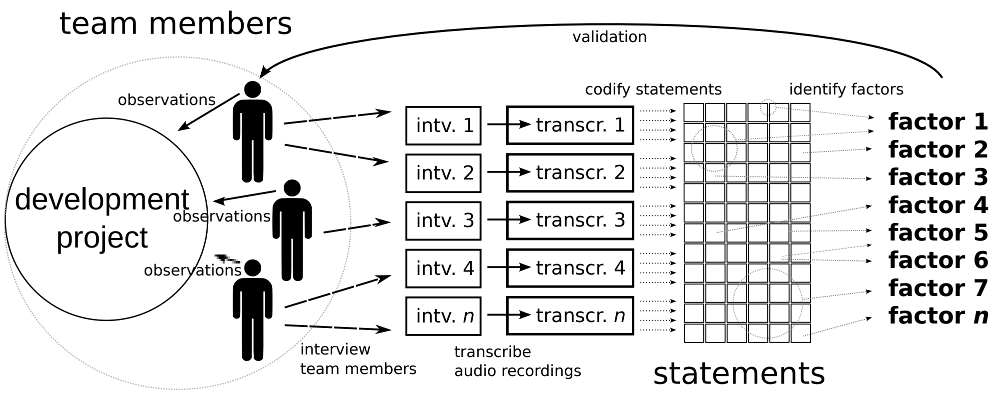


Figure 1.4: *The grounded theory data collection and analysis method – applied to a software project*

diligent inter- and intra-organization networking to obtain access to team members and to gain permission to collect data, developing intelligent searching heuristics to mine software repositories, adopting to the myriad of digital platforms that exists and careful examination and obtaining means for triangulation (Basili and Weiss, 1984). While this work can be rather labor-intensive, analysis of valid software engineering data from industrial practice is necessary to support hypotheses and theories in the academic field. Conversely, conclusions based on analysis of careful observations lead to insights that are also valuable to industry. The data collection techniques employed in this dissertation include interviews, document analysis and software repository mining.

1.5.1 Software Repository Mining

Software repository mining is a technique that involves obtaining information from software repositories such as software configuration and change management systems (SCCMSs), defect tracking systems or time registration systems. All sorts of data are stored in repositories and all can be mined. Data typically mined includes source lines of code, software models, defect data, requirements, change requests, particular documents such as the software architecture description and system and project characteristics such as functional size estimation, cost structure, project planning and information pertaining to team member time registration.

Depending on the system and the nature of the information needed, various different approaches might be required to mine a software repository. More structured data requirements such as the build-up of the code of a particular component over all revisions requires a different strategy. The commonly used SCCMS Subversion (SVN) provides multiple interfaces and therefore lends itself well to access via a scripting language such as Perl[4] or GNU Bash[5]. However, depending on particular (often client-specific) technology requirements, a software development organization might use various different SCCMSs from which a researcher might need similar data. IBM's ClearCase[6] also permits the use of Perl but requires different handling from SVN. For Microsoft .Net[7] projects, the designated SCCMS often is Team Foundation Server[8], which only permits very limited requests. Each repository mining situation therefore requires a specific approach dependent on the data needed and the type of system being mined. Specific data collection methods have been elaborated on in each separate chapter.

1.5.2 Document Analysis

In the context of this dissertation, document analysis pertains to the structured dissection of software engineering documentation including process descriptions, software architecture design documentation, project management reports and post-mortem project reviews. For comparative analysis, abstractions of concepts are created on the basis of which documents can be compared. In the case of project management reports and post-mortem project reviews, documents are essentially treated as a software repository from which data is gathered.

1.5.3 Interviews

The interview is a qualitative data collection technique that, *"seeks to cover both a factual and a meaning level"* (Kvale, 1996). Interviews are used when data needs to be collected about phenomena that cannot be obtained using quantitative measures. The type of interview used for data collection in the context of this dissertation is the "qualitative interview" which is *"a sort of guided conversation"* (McNamara, 1999). The interviews are standardized in the sense that similar questions are asked of each interviewee (depending on that person's role) and open-ended in the sense that there is ample room for interviewees to elaborate. This type of interview is also referred to as *semi-structured* or *focused*.

1.5.4 Data Sources

For this dissertation, data (artifacts, other project-specific data and narratives and opinions from employees) were obtained from nine large, international software development organizations.

[4] http://www.perl.org/
[5] https://www.gnu.org/software/bash/
[6] https://www-01.ibm.com/software/awdtools/clearcase/
[7] https://www.microsoft.com/net
[8] http://msdn.microsoft.com/en-us/vstudio/ff637362.aspx

Projects A and B (studied in chapter 3) were executed by the same organization (which is one of the three organizations from Chapter 2). Projects C and D (studied in Chapter 3) and cases A, B and C (studied in Chapter 4) are all GSD projects and were all executed by the same two organizations. Furthermore, project C (Chapter 3) is used for different analyses in Chapters 7 and 8. Data sources are elaborated on in more detail, in each chapter.

1.6 Contributions and Outline

The contribution of this dissertation is threefold. First, this dissertation provides sound empirical evidence about the necessity of careful and structured dissemination and coordination of software architecture design in the context of global software development.

Second, this dissertation provides recommendations regarding representation of software architecture design in the context of global software development.

Third, this dissertation provides sound empirical evidence that suggests that application of MDD tools and techniques significantly changes the traditional software architecture process and that this might be beneficial in the context of global software development. Additionally, recommendations for the application of MDD tools and techniques are provided.

This dissertation is structured as follows:

Chapter 2 — Comparison of Industrial Process Descriptions for GSD: In this chapter, the main objective is to explore how software development process descriptions used by three software development organizations are tailored to accommodate for GSD (**RQ1**). Parts of this chapter were published earlier (Heijstek et al., 2010).

Chapter 3 — Architecture and Design Process Evaluation Through Effort Visualization: In this chapter, the main objective is to assess visually how resource allocation for global software development differs from co-located software development (**RQ1**). Parts of this chapter were published earlier (Heijstek and Chaudron, 2007, 2008a,b).

Chapter 4 — A Multiple Case Study of Coordination of Software Architecture Design in GSD: In this chapter, the main objective is to chart how software architecture is coordinated and disseminated in three large cases of industrial, custom, global software development (**RQ1**).

Chapter 5 — A Theory of Coordination of Software Architecture Design in GSD: In this chapter, the main objective is to relate and reflect on the factors that shape how software architecture is disseminated and coordinated in large, industrial, custom, global software development projects (**RQ1**).

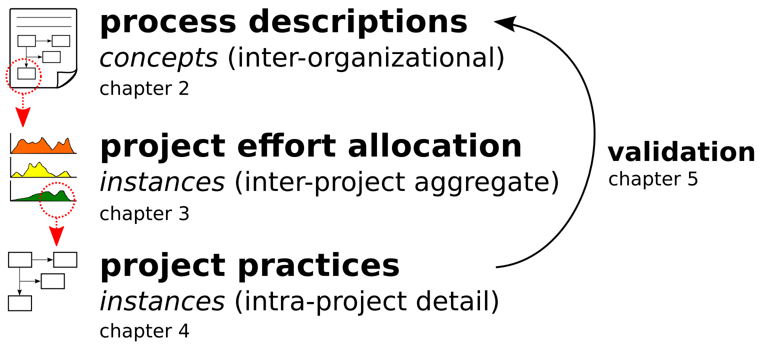

process descriptions
concepts (inter-organizational)
chapter 2
project effort allocation
instances (inter-project aggregate)
chapter 3
project practices
instances (intra-project detail)
chapter 4
validation
chapter 5

Figure 1.5: *Organization of the four chapters that address RQ1*

Chapter 6 — Experimental Analysis of Representation of Software Architecture Design: In this chapter a controlled experiment is discussed that addresses how software developers comprehend software architecture representations (**RQ2**). Parts of this chapter were published earlier (Heijstek et al., 2011).

Chapter 7 — Contrasting Model-Driven Development with Code-Centric Development: The aim of this chapter is to explore how the characteristics of a large scale, industrial model-driven development project in the context of global software development compare to non-MDD projects (**RQ3**). Parts of this chapter were published earlier (Heijstek and Chaudron, 2009).

Chapter 8 — Analysis of the Consequences of Model-Driven Development for GSD: In this chapter, the main objective is to assess how the application of MDD tools and techniques impact the problems associated with Global Software Development (**RQ3**). Parts of this chapter were published earlier (Heijstek and Chaudron, 2010)

Chapter 9 contains a summary of the findings, the conclusions, an outline of future work and a reflection on the research process. The relation between in the four chapters that address RQ1 is visualized in Figure 1.5.

1.7 Publications

This is a chronological list of publications that were (co-)authored during this doctoral research:

1. Werner Heijstek and Michel R. V. Chaudron (2007) **Effort distribution in model-based development.** *In Proceedings of the 2nd Workshop on Model Size Metrics (MSM 2007) pages 26–38, Nashville, Tennessee, USA*

2. Werner Heijstek and Michel R. V. Chaudron (2008) **Exploring Effort Distribution in RUP Projects.** *In Proceedings of the 2nd International Symposium on Software Engineering and Measurement (ESEM 2008) page 359, Kaiserslautern, Germany*

3. Werner Heijstek and Michel R. V. Chaudron (2008) **Evaluating RUP Software Development Processes Through Visualization of Effort Distribution.** *In Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2008) pages 266–273, Parma, Italy*

4. Werner Heijstek and Michel R. V. Chaudron (2009) **Empirical Investigations of Model Size, Complexity and Effort in Large Scale, Distributed Model-Driven Development Processes — A Case Study.** *In Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009) pages 113–120, Patras, Greece*

5. Werner Heijstek, Michel R. V. Chaudron, Libing Qiu and Christian C. Schouten (2010) **A Comparison of Industrial Process Descriptions for Global Custom Software Development.** *In Proceedings of the 5th International Conference on Global Software Engineering (ICGSE 2010) pages 277–284, Princeton, New Jersey, USA*

6. Werner Heijstek and Michel R. V. Chaudron (2010) **The Impact of Model-Driven Development on the Software Architecture Process.** *In Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2010) pages 333–341, Lille, France*

7. Christoph J. Stettina and Werner Heijstek (2011) **Five Agile Factors: Helping Self-Management to Self-Reflect.** *In Proceedings of the 18th European System & Software Process Improvement and Innovation Conference (EUROSPI 2011) pages 84–96, Roskilde, Denmark*

8. Jorge A. Osorio, Michel R. V. Chaudron and Werner Heijstek (2011) **An Empirical Study into the Benefits of Using an Iterative Development Process Versus a Waterfall Process.** *In Proceedings of the 37th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2011) pages 453–460, Helsinki, Finland*

9. Werner Heijstek, Thomas Kühne and Michel R.V. Chaudron **Experimental Analysis of Textual and Graphical Representations for Software Architecture Design.** *In Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2011) pages 167–176, Banff, Alberta, Canada*

10. Hugo H. Schoonewille, Werner Heijstek, Michel R.V. Chaudron and Thomas Kühne (2011) **A Cognitive Perspective on Developer Comprehension of Software Design Documentation.** *In proceedings of the 29th ACM International Conference on Design of Communication (SIGDOC 2011) pages 211–218, Pisa, Italy*

11. Christoph J. Stettina and Werner Heijstek (2011) **Necessary and Neglected? An Empirical Study of Internal Documentation in Agile Software Development Teams.** *In Proceedings of the 29th ACM International Conference on Design of Communication (SIGDOC 2011) pages 159–166, Pisa, Italy*

12. Christoph J. Stettina, Werner Heijstek and Tor Erlend Fægri (2012) **Documentation Work in Agile Teams: The Role of Documentation Formalism in Achieving a Sustainable Practice.** *In Proceedings of the AGILE Conference 2012 Dallas, Texas*

13. Michel R. V. Chaudron and Werner Heijstek (2012) **Quality Assurance for UML Modeling** *In Proceedings of the Fifth International Conference on Frontiers of Information Technology, Applications and Tools (FITAT 2012), Ulaanbaatar, Mongolia*

14. Michel R. V. Chaudron, Werner Heijstek and Ariadi Nugroho (2012) **How Effective is UML Modeling? — An Empirical Perspective on Costs and Benefits** *Journal of Software and Systems Modeling* vol. 11, issue 4, pages 571–580

The following manuscripts were under review at the time of writing this dissertation:

15. Ana M. Fernández-Sáez, Peter Hendriks, Werner Heijstek and Michel R. V. Chaudron (2012) **The Role of Domain-Knowledge in Understanding Activity Diagrams — An Experiment**

16. Rut Torres Vargas, Seher Altinay Soyer, Werner Heijstek and Michel R. V. Chaudron (2012) **A Developer Perspective on the Role of Software Architecture Documentation in Global Software Development**

17. Erik Jan Philippo, Werner Heijstek, Bas Kruiswijk and Michel R. V. Chaudron (2012) **Requirement Ambiguity Not as Important as Expected — Results of an Empirical Evaluation**

Chapter 2

A Comparison of Industrial Process Descriptions for Global Custom Software Development

In this chapter, we analyze how organizations address global software development-specific issues on the process level. To this end, we conduct a comparative analysis of industrial process descriptions for global software development.

This chapter is based on the following publication:

Werner Heijstek, Michel R. V. Chaudron, Libing Qiu and Christian C. Schouten (2010) **A Comparison of Industrial Process Descriptions for Global Custom Software Development.** *In Proceedings of the 5th International Conference on Global Software Engineering (ICGSE 2010) pages 277–284, Princeton, New Jersey, USA*

2.1 Introduction

Some of the pitfalls associated with GSD, such as the lack of a structured process, unclear tasks, roles and responsibilities, knowledge sharing concerns and general communication issues can be alleviated by using a process description which explicitly addresses these issues (Prikladnicki et al., 2006). A process description is an instance of the description of a software process model. Such a model represents:

"a networked sequence of activities, objects, transformations, and events that

embody strategies for accomplishing software evolution. Such models can be used to develop more precise and formalized descriptions of software life cycle activities" (Marciniak, 2002)

Software Life Cycle Processes are defined in more detail in an international standard (ISO/IEC, 2008). It is generally agreed upon that working according to a well-defined development process is key to software engineering in general (Parnas and Clements, 1985, Royce, 1970). Nevertheless, it is unclear whether organizations tailor their process descriptions for GSD and of what they are comprised.

We report the findings of a comparative study of the GSD process descriptions used for custom software development of three industrial organizations. Section 2.2 describes the objectives of the study, Section 2.3 describes related work and Section 2.4 outlines the method of the study. Section 2.5 reports on the results which are then discussed in Section 2.6. Finally, Sections 2.7 and 2.8 present the conclusions and elaborate on future work.

2.2 Objectives

In this chapter, we address **RQ1** (Section 1.3). This exploratory research question aims (in part) to uncover how software architecture is coordinated in the context of global software development. However, little is known about prescribed processes for GSD in industrial practice in general. Therefore, in order to address the coordination of the software architecture process, we must first examine the GSD process as a whole. To this end, we turn to process descriptions of such processes: What are they comprised of? How are they made? How are they used in practice? Are software process descriptions tailored for GSD-specific issues? What aspects of GSD are focused on? The motivation behind this study is to explore *how software development process descriptions are tailored to accommodate for GSD*. This study analyzes the content of, motivation for and use of three process descriptions currently used in GSD projects within three different organizations. The contribution to scientific GSD literature of this study is three-fold and is divided up into three sub-questions which we will shortly motivate here.

1. The first sub-question relates to the extent to which process descriptions are tailored for GSD in different organizations: *How do different process descriptions for GSD compare?* This question deals with the content of the process descriptions: What is described in each of the process descriptions for GSD? What common and distinct elements can be identified?

2. The second sub-question relates to the rationale behind the build-up of the process descriptions, why is a specific GSD process description made and what is the rationale for including or omitting certain elements? The second sub-question is: *What is the organizational rationale behind the design of a GSD-specific process description?*

3. Lastly, we investigate: *How are these process descriptions meant to be used in actual development projects?*

2.3 Related Work

The lack of process structure is a commonly reported source of GSD process frustration (Salger, 2009). This includes lacking a definition of work units, such as design documentation (Hussey and Hall, 2007), lacking prescriptions on methods of bundling work units (Cusumano, 2008) and lacking prescriptions on procedures for knowledge management (Hussey and Hall, 2007). No previous studies have focused specifically on industrial GSD process descriptions.

The role of process descriptions in GSD has been discussed but not empirically validated in industrial practice. In their report on application of GSD in the large, Battin et al. (2001) discuss various issues, one of which is "differing development process". This issue poses the challenge of coordinating between various development sites that follow different processes. The authors prescribe three solutions for this problem. First, Battin et al. argue not to impose a common process to let each team produce results immediately. Second, they propose to come up with a set of common work products and vocabulary and to make a mapping between the common work products and the specific deliverables of the individual development center. Third, to split a system up in tested subsystems that can be developed independently by each development center. This division by modularity or chunking of work items is one of the strategies that can be taken to global software development (Mockus and Weiss, 2001). However, this strategy can only be applied if both the organization and the software system architecture allow modularization. If not, increased process commonality is imperative.

Coordination in GSD becomes an issue because of process non-uniformities. An example of process non-uniformities is variation in definitions which may cause mismatched expectations. Also, a mismatch in common milestones at one location may affect other development sites, but is often not communicated early enough. In addition, different time zones may lead to more frequent work handovers (Mockus and Herbsleb, 2001).

Commonly regarded as the largest sources of risk in GSD are a lack of clarity and resulting project delay due to strenuous communication across development sites. The decreased amount of opportunity for informal team communication in GSD aggravates the inherent communicative difficulties posed by the distinctly different experience, training, professional backgrounds, cultures and native languages of various team members. This problem is further aggravated by the often rapid changes in team composition on each development site (Herbsleb et al., 2000).

In their systematic review of 170 GSD studies, Jiménez et al. (2009) list ten success factors. At least seven of these, such as 'establishment of an effective communication mechanism' and 'application of maturity models' are related to process descriptions.

In addition, all seven best practices listed in a recent survey of empirical GSD studies literature (Šmite et al., 2010) are facilitated by a process description.

Process descriptions include processes and activities related to knowledge management. In the context of GSD, knowledge management issues become exponentially pertinent as knowledge is spread over development sites and coordination of this knowledge can prove to be difficult (Desouza and Evaristo, 2004). And although the more central role of tools such as the use of distributed software configuration and change management systems (SCCMS) (Carmel, 1999, Grinter, 1997) formalize, and thereby unify work methods, the use of these tools needs to be enforced by a common process.

2.4 Method

In this section the research environment and the study approach are discussed.

2.4.1 Research Environment

We obtained access to process descriptions of three different software development organizations that work with an offshore subsidiary to develop software. The organizations will be referred to as JKL, ABC and XYZ. The first two organizations are large and established information technology service providers that operate on a world-wide scale while the latter is a comparatively small, Dutch Information Technology (IT) service supplier. Table 2.1 outlines relevant data including organization age, size, process scope and organizational maturity (defined as the attained level on the Capability Maturity Model (CMMI, Chrissis et al., 2003). Table 2.1 also mentions whether an organization offshores only to subsidiaries that are part of its own organization ("intra-organizational" offshoring) or whether it also makes use of the services of external organizations ("inter-organizational" offshoring).

Table 2.1: *Organizations Under Study*

Org.	Age (yrs.)	Employ.	Offshore Location	Process Used in	CMMI level	Dev. Process	Offshore Model
JKL	50	50,000	India	Netherlands	3	RUP	*inter*-organiz.
ABC	50	100,000	India	world-wide	2	RUP	intra-organiz.
XYZ	7	70	S. Africa	Netherlands[1]	2	RUP	intra-organiz.

[1] Organization is *only* active in the Netherlands

2.4.2 Approach

The main unit of analysis was the GSD process description document. We studied the target audience, the methods described, the way the process was outlined, the level of detail used and we distilled commonalities and distinctions in the descriptions. As part of the analysis, the workflow and activities of the process descriptions have been remodeled using a uniform third-party modeling language. We used the Business Process Model and Notation language (BPMN, Object Management Group, 2009). Using a common, visual-oriented, modeling language such as BPMN eases comparison of verbal descriptions and notational snippets. An example of one of these translations can be seen in Figure 2.1. After this initial analysis, we interviewed the designers of the process descriptions. The main purpose of these semi-structured interviews was to clarify the findings of the analysis, to understand the process and organization of the "process development" and to gauge the extent to which the descriptions are used in practice. To this end, the interviews were divided into six common sections:

1. *Document purpose*
   Why was the document made?

2. *Communication of the process*
   How is the process communicated to its audience?

3. *Process Construction & Maintenance*
   How is the process made? How is it maintained?

4. *Process management*
   Who is responsible for the process?

5. *Process in practice*
   How is the process description intended to be used?

6. *Document versioning*
   What is the version history of the document we analyzed? Are future versions planned?

In addition, we reserved a section of the interview to discuss the specific process description of a particular organization.

2.5 Results

All processes are based on, or at least rely on, terminology from RUP. The RUP has enjoyed a widespread popularity and is often used in industrial practice. As a result, terminology used in the RUP is commonly understood and used.

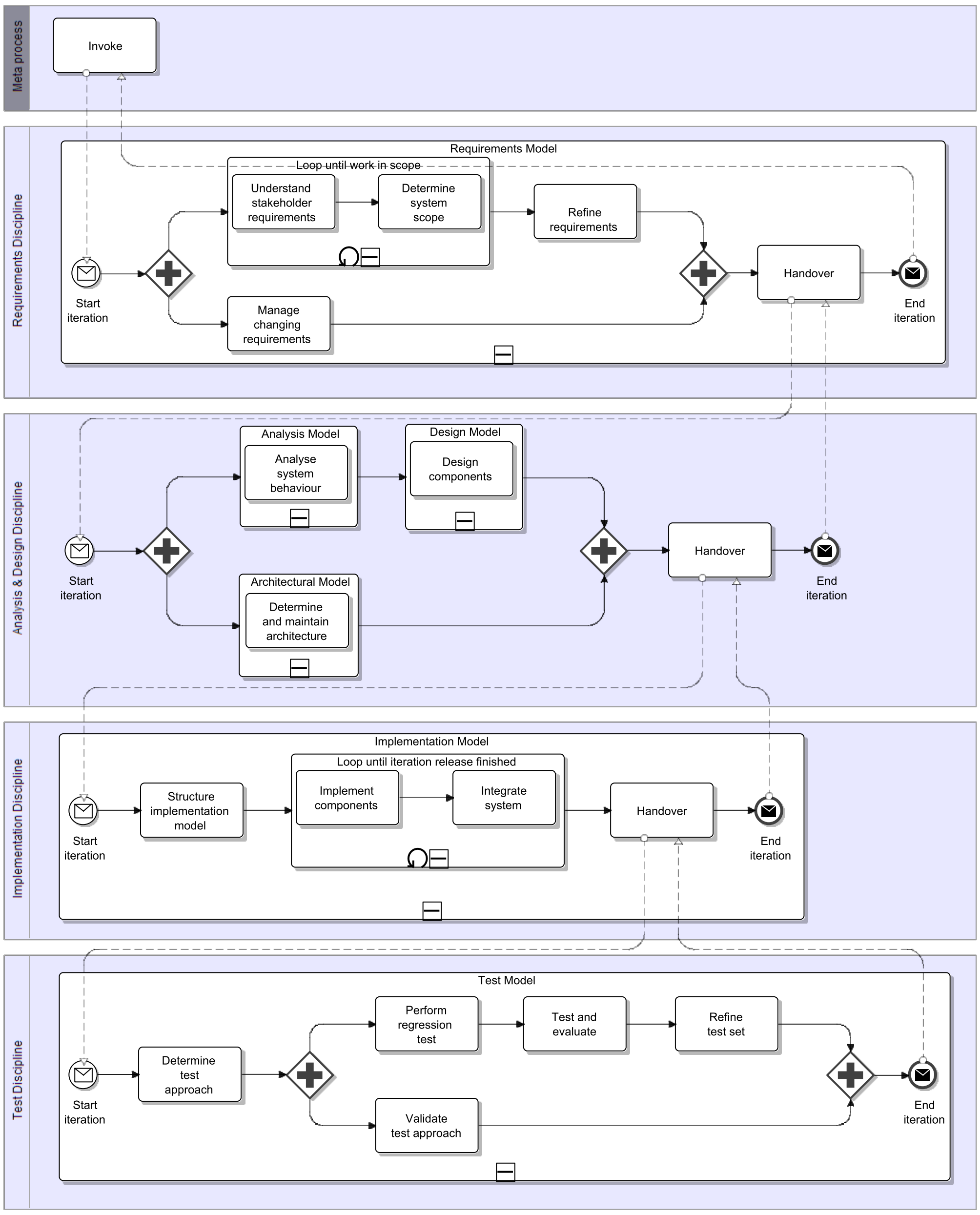


Figure 2.1: *BPMN diagram for GSD process description of case "XYZ"*

Of the three process descriptions, two were written as chiefly verbal-oriented documents and one as a (detailed) set of slides. XYZ chose the slide format to be able to use the presentation as part of the on-boarding course for new project members or project leaders. XYZ shares the same documentation regarding their process description with all stakeholders, including all team members and clients. In contrast, ABC reserves their process description for higher project management and JKL, while also aiming to only write for higher project management, keeps their process description confidential. Only XYZ uses detailed UML notation to decrease the chances of ambiguity due to the wider audience of their process description. In the case of ABC, additional documentation was used to supplement the "non-onshore" software development process description. The other two organizations tailored their common process descriptions to specifically address GSD-specific issues. All three process specifications are in use at the Dutch subsidiary of each organization.

At the time of writing, JKL's process description was in the later stages of being completely reviewed and renewed. We analyzed a release candidate of version 1.0 which was at that time in use at several projects in the organization. ABC wrote their GSD process description in 2008 and did not intend on updating the GSD aspect of the process description but in 2009 embarked upon a multi-year, organization-wide campaign to further formalize development practices. XYZ's process description was made over the course of several years and reviews or updates were not planned in the foreseeable future.

2.5.1 Process Sections

We identified 13 different, coherent groups of information in the process descriptions. These groups were identified by defining and naming a group based on a set of coherent topics, looking for the same set of coherent topics in the second and third process descriptions and redefining and renaming (often a subset of the initial set of coherent topics) until a set was consistent for as many process descriptions a possible. A summary of these groups can be found in Table 2.2. Only three topics were common to all process descriptions, specifically (1) a description of workflow, (2) an overview of deliverables and (3) a classification of involved roles. In addition, two out of three process descriptions contained sections on (4) activities, (5) organizational objectives, (6) tools and (7) quality control. The other six identified topics were unique to the process description of ABC, namely (8) a description of change request processes, (9) risk analysis and management, (10) communication protocol, (11) resource planning, (12) customer value and (13) knowledge management (Qiu, 2009).

Only two out of three process descriptions describe a detailed process workflow and objectives. ABC approaches their GSD process description as an additive set of rules and practices to any process description and therefore, lacks a step-by-step workflow description. As both JKL and ABC are larger, multi-national organizations, there is a stronger focus on organizational objectives. XYZ is more flexible and defines objectives per project. Remarkable observations include that while all three organizations find the CM process a key element of their process description, only ABC prescribes a CM process in their process description. JKL did not yet formalize their CM process and XYZ uses a CM tool set which enforces a specific process. Furthermore, JKL does not mention tooling because the organization does not make use of standardized tooling and XYZ describes itself as too small to need to formalize knowledge management procedures. Also, only ABC prescribes a resource planning method. Resource planning for GSD is different from common resource planning because additional effort has to be calculated for communication, travel, increased quality of documentation and knowledge management (Zopf, 2009). GSD is known to add extra risk factors such as missing knowledge or know how and misunderstanding because of language deficiencies, different cultural backgrounds or employee turnover (Zopf, 2009). Only ABC describes risk analysis in their process description.

2.5.2 General Process

The ABC process description mainly highlights the headlines of the process. Most of the description deals with project management issues and deliverable specifications. In the introduction of its process description, ABC defines a project manager which is "usually" located off-site and an overall project manager which is "usually" located on-site. Most of the sections containing project management activities include a reference as to which role is responsible for those activities, leading to an estimation of the distribution of activities on-site versus off-site. For the activities of the software development process, a table has been drafted of clear goals and guidelines regarding the distribution of activities on-site versus off-site. ABC's process description is clear on which activities are performed by which actors on which shore.

The process description submitted by organization XYZ, is less formal in structure, but not less formal in description. The software development process has been clearly defined and specified into activities. Roles and tasks are separated. And while project management activities have also been defined and specified into separate activities, they have not been integrated with the software development activities. This loose connection between project management and software development is illustrative of the informal structure of the process description. Another possible explanation for this lack of integration could be that XYZ is less experienced in GSD and generally less mature than the other two organizations. This process description makes no difference between which activities are performed onshore and, which are performed offshore as XYZ uses the same process description at both locations.

2.5.3 Additional Documentation

In addition to the documentation we analyzed, for at least one organization, XYZ, we found that the CM process was enforced by a tool. This tool obliges team members to follow certain steps while registering change requests. Moreover, all three organizations make use of an additional set of discipline-specific instructions which are placed on an internal wiki or other type of intranet site. For example, detailed methods for system design and modeling are described. Another example is a set of best practices for setting up a workshop to facilitate requirement elicitation. In all three organizations, these pages are regularly updated by experts.

Table 2.2: *Comparison of GSD process descriptions*

		CASE JKL	CASE XYZ	CASE ABC
Overall	*size*	44 pages	54 slides	44 pages
	content types	text; UML & workflow diagrams; tables	text; UML diagrams; other diagrams	text; other diagrams, tables
	language	English	English	English
	audience	project management	team members, client	engagement management
Workflow	*steps*	yes	yes	no
	lev. of detail	high level of detail (UML)	high level of detail (UML)	low level of detail
	described	roles; responsibilities; deliverables; requirements	responsibilities per role; deliverables per phase	responsibilities per role; deliverables and requirements per step
Activities	*descr. meth.*	step-by-step	absent	a list of important points
	described	activity flows; acceptance criteria; activities per project type	nothing	activity lists for situations; acceptance criteria; on- and off-site activities
Objectives		objectives for steps; general objective of process description	absent ('these differ per project')	objectives for steps; objectives for steps
Deliverables	*descr. meth.*	described with activities description	described as comments in UML diagrams	described in activities description
	described	responsible staff; list acceptance criteria; references links in activities description; product hand-over process; update deliverables process	responsible staff as actors in UML diagrams; list acceptance criteria & show sample deliverables	responsible staff; list acceptance criteria; templates links in table; set deadline for hand-over
Role Classific.	*descr. meth.*	responsibilities per role	roles as actors in UML diagram	responsibilities per role
	described	describe responsibilities in activities description; assign deliverables in table; assign tasks in table	responsibilities in UML diagrams; related deliverables in UML diagrams	list responsibilities ; related deliverables; tasks in activities description

(continued on next page...)

Table 2.2: *Comparison of GSD process descriptions (continued)*

		CASE JKL	CASE XYZ	CASE ABC
Chg. Req. Process	*descr. meth.*	absent ('not yet formalized)	absent ('process captured in a tool')	change process outlined
	described	nothing	nothing	description of change related activities; description of change related staffs; description of change related tools
Risk Assessm.	*descr. meth.*	requirement	none	list important risk management activities
	described	'require update risk log'	nothing	list requirements of doing RA&M; assign tools for certain RA&M activities
Communic. Protocol	*descr. meth.*	none ('promote informal communication')	none ('organization is too small')	description of communication related activities
	described	nothing	nothing	responsible roles for communication activities; requirement for communication plan; requirement for communication document
Tools	*descr. meth.*	none ('does not use standard tooling')	short tool descriptions	tools are linked to tasks
	described	nothing	list of possible tools	recommended tools; Tasks
Quality Control	*descr. meth.*	quality control per activity	none ('differs per project')	separate quality control section ('but just for reference')
	described	list acceptance criteria; tasks of responsible roles	nothing	list acceptance criteria in activities description; tasks of responsible roles
Resource Plann.	*descr. meth.*	none ('no standard method')	none ('differs per project')	key activities of resourcing planning
	described	nothing	nothing	list requirements of resource planning; assign tasks to responsible people
Customer Value	*descr. meth.*	none ('but we focus on customer intimacy')	enhancing customer text ('shared view of enhancing the customer value')	
	described	nothing	nothing	organizational attitude towards customer value; activities for enhancing customer value

(continued on next page...)

Table 2.2: *Comparison of GSD process descriptions (continued)*

		CASE JKL	CASE XYZ	CASE ABC
Knowledge Mgmt.	*descr. meth.*	none ('separate org.-wide process specification')	none ('organization size is too small')	important activities of knowledge management
	described	nothing	nothing	organization's attitude of knowledge sharing; list activities of knowledge management

2.6 Discussion

In the following subsections, we discuss our findings regarding process design comparison, process rationale, intended process use and process maintenance.

2.6.1 Process Design Comparison

We found very different approaches of process description design for the three cases. The most visible are the description methods. ABC mainly uses lists and tables, JKL uses text supported by various types of diagrams including UML and free-form diagrams, and XYZ uses UML diagrams almost exclusively. Also, the level of detail of the process descriptions varies strongly. JKL provides a detailed process description in which process steps are clearly outlined. XYZ provides less detail and ABC provides almost no detail regarding process steps and focuses chiefly on the possible pitfalls of GSD. While these are three different organizations, these process descriptions are to be used for similar types of custom software development projects by software developers of similar education level and expertise. We did not find any particular reason for the choices for using models over text or vice-versa. While answering questions such as, *"Why did you use a UML activity chart to model this process?"*, process designers generally presumed that their chosen method was the only logical method to convey a specific process step or best practice. We observe that, at least in the organizations we studied, the choice for inclusion of specific elements and the methods to describe these elements is at least in part dependent on the expertise and professional background of the process engineers.

2.6.2 Process Rationale

We observed various reasons for designing a GSD process. Various interviewees within the same organization gave different answers to the question, "Why was a process description made for GSD?". Among the reasons were a desire for "repeatability of approach" (prescriptive) to be able to better predict the development process, to use as course material for on-boarding of new project team members (descriptive) and organizational maturity, to e.g. obtain a certain CMMI level.

Project Management Activity Integration

A distinctive feature was the extent to which the steps of the development process are integrated with project management activities. ABC separated both, providing for a strict separation of tasks and responsibilities. JKL, on the other hand, chose to fully integrate these processes. In the case of JKL, a separation of tasks and responsibilities was achieved by clearly describing the actors within the process description and by providing swim lane diagrams of the sub-processes. The information released by connecting the software development process and the surrounding project management activities can be seen as additional information regarding the process. A more mature organization, e.g. in terms of obtained level of CMMI certification, links various types of activities to one another. The actual process maturity and the intended audience of a GSD process specification dictate the extent to which development process and project management activities are linked. Project management and especially program management is less interested in development activities as it is in management activities but does need to understand how both integrate. XYZ's level of integration of development and management processes can be placed in the middle between companies ABC and JKL. XYZ clearly defined a process view of the project management activities and provided starting points to connect these elements, but a full integration is not achieved.

Tailoring Processes for GSD

With regard to the overall goal of this study to understand how software development processes descriptions are altered to tailor for GSD, we note that the extent to which the process descriptions have been particularly tailored for GSD differs. The larger organizations seem to take different strategies at GSD. JKL focuses on formal processes, whereas ABC intends to shift more responsibility to the individual project manager. This is remarkable as in their respective interviews, process management noted that JKL's organizational strategy is to focus on customer intimacy, whereas ABC aspires to attain operational excellence. And while GSD is a central and an increasingly important activity for both organizations, their organizational strategies are not (yet) apparent from their GSD process descriptions.

2.6.3 Intended Process Use

The intended use of the prescribed process does not necessarily correspond with the provided level of detail. For example, ABC provides a vast list of best practices but only intends these to be used as a reference, whereas JKL expects projects to follow the process as prescribed in the documentation. We have summarized the organization's approaches to process description use in Table 2.3.

Table 2.3: *Approach to Process Description Use*

	Process Level of Detail	GSD Focus	Process Use	Intended Audience
JKL	high	limited	obligatory	project leader
XYZ	medium	limited	obligatory	all team members
ABC	low	limited	facultative	project leader

If we define a process description plainly as a description of a process, ABC provides a minimal amount of description. However, it provides the most elaborate description of GSD practices, only to merely recommend its use to project leaders. JKL provides a very detailed account of process steps, a more limited focus on GSD specific procedures but obliges project managers to use the description. XYZ provides a description of process steps, the level of detail of which can be placed between that of the process descriptions of JKL and ABC and focuses only sparsely on possible GSD issues while requiring all project members to know the documented process. As shown in Table 2.1, the process scope of ABC is world-wide. ABC's set of rules and best practices, while elaborate, is set up so it can be used with very tailored software development processes. By letting a project manager free to set up a customer development process but providing him with a detailed set of guidelines, ABC combines flexibility and the benefits of applying best practices. This is, however, as mentioned, not in line with ABC's organizational objective of achieving "operational excellence". ABC is in the process of defining a more rigorous process description in which its current GSD process description will be merged.

Both larger organizations, JKL and ABC, have set up departments, which are responsible for (re-)engineering, publishing and distribution of process descriptions. These departments periodically review the existing processes, not only for custom software development (generally Java and Microsoft .Net development) but also for software development for business intelligence systems and Enterprise Resource Planning (ERP) systems such as SAP.

2.6.4 Process Maintenance

Both larger organizations contain business consulting and process engineering department, which are actively involved in (re-)designing processes and process descriptions.

The Dutch subsidiary of JKL uses a system where feedback is continuously asked of project management. Feedback on processes is also incorporated in the standard post-mortem analysis of a software development project. ABC uses a more top-down oriented approach, where an international team of specialists reviews and re-engineers process descriptions.

2.7 Validity

The lack of related work investigating GSD process descriptions in industrial practice warrants an exploratory study. As no industrial, GSD-specific process descriptions are currently available in literature, the three process descriptions we obtained can only be compared to each other. In addition, two organizations in our analysis predominantly engage in intra-organizational GSD while one organization (JKL) also engages in inter-organizational GSD. This might influence the way their respective process description has been designed. As one might expect is required for inter-organizational collaboration, a more detailed process description is required. However, this was not mentioned as a reason for the provided level of detail. In addition, we found that some parts of the processes are captured outside the process description documents we analyzed. (e.g. the CM process of one organization is captured in a CM tool set). These external elements were not available for this study.

2.8 Conclusions and Future Work

The conclusions are structured as answers to the three sub-research questions:

1. *How do different process descriptions for GSD compare?*
   All studied processes are based on, or at least rely on terminology from RUP. The level of detail of the process descriptions varies strongly. Also, the extent to which the process descriptions have been particularly tailored for GSD differs strongly. Particularly the larger organizations seem to take strategies at GSD that are different from one another.

2. *What is the organizational rationale behind the design of a GSD-specific process description?*
   The rationale behind the GSD process description format, structure and content are said to be derived from organizational objectives. Our analysis did not confirm this. GSD process descriptions are made by a multi-disciplinary group of consultants, and it is not yet clear how the processes are used in practice. Design of the process descriptions seems to be partly dependent on the expertise and professional background of the process designers. Other important influences on the design and intended use of the process descriptions are the size of an organization and organizational maturity.

3. *How are these process descriptions meant to be used in actual development projects?* The intended audience is sometimes explicitly project management and other times explicitly the entire project team. The use of the process description is facultative in one organization while it is obligatory in others. We found that the intended use of the prescribed process does not necessarily correspond with the provided level of detail.

In order to increase our understanding of GSD process descriptions, the use of the studied process descriptions in actual GSD projects must be studied. Are the specific alterations that organizations make to cope with GSD-specific issues followed by projects in practice? How does this influence project success? Furthermore, the process of engineering a GSD process is not yet clear and the specific impact of GSD process descriptions on the development process is also to be investigated.

Chapter 3

Global Architecture and Design Process Evaluation Through Effort Visualization

The objective of this chapter is to evaluate how resources (person-hours) are allocated in global software development projects and co-located projects. To this end, patterns in process resource allocation in general and in architecture and design processes in particular, are analyzed by means of effort distribution visualization. We collected data from four large-scale industrial software development projects. Data is obtained from various sources within these projects.

This chapter is based on the following publications:

- Werner Heijstek and Michel R. V. Chaudron (2007) **Effort distribution in model-based development.** *In Proceedings of the 2nd Workshop on Model Size Metrics (MSM 2007) pages 26–38, Nashville, Tennessee, USA*
- Werner Heijstek and Michel R. V. Chaudron (2008) **Evaluating RUP Software Development Processes Through Visualization of Effort Distribution.** *In Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2008) pages 266–273, Parma, Italy*
- Werner Heijstek and Michel R. V. Chaudron (2008) **Exploring Effort Distribution in RUP Projects.** *In Proceedings of the 2nd International Symposium on Software Engineering and Measurement (ESEM 2008) page 359, Kaiserslautern, Germany*

3.1 Introduction and Objectives

Software architecture and design are both processes and artifacts. As processes, software architecture and design influence one another. If perceived from a chronological stance, a first version of a software architecture is often designed before a more detailed design specification is made. However, such a design specification might lead to insights that in turn influence a newer instance of the software architecture.

While these process interactions are unique for each software project, one may expect to find patterns in process interactions in software projects. These patterns may be expected to be similar for custom software development projects.

The developers of the RUP (Kruchten, 2003b) defined a set of processes and created visualizations of the effort distribution between these processes in a diagram that would later be referred to as the "RUP Hump Chart" (Figure 1.3). In this chapter, we will use a similar visualization to investigate how resource allocation for distributed software development differs from co-located software development. Specifically, we will address **RQ1** (Section 1.3). This exploratory research question aims (in part) to uncover how software architecture is coordinated in the context of global software development. We will therefore specifically analyze the role of the "analysis and design" discipline in these visualizations.

Research regarding distribution of effort in software processes is commonly found in literature on software estimation and planning: (Milicic and Wohlin, 2004, Iwata et al., 2006, Baldassarre et al., 2006, Menzies et al., 2006). However, a large portion of the research in that area deals with estimating the total amount of effort needed for a project for specific conditions or development methods such as reuse of code (Lopez-Martin et al., 2006) or use-case based requirement specifications (Braz and Vergilio, 2006). What an effective distribution of effort over disciplines is, remains unaddressed in literature. Important reasons are a lack of data on software process in general and problems with regards to comparability of data in particular. This study focuses on effort distribution over the lifespan of industrial, custom software development processes. It does so for three reasons:

First, effort distribution is studied to improve our understanding of project dynamics from a resource perspective. Visualization of software engineering process effort distribution aides in analyzing process dynamics such as the effects of a chosen iteration strategy. Furthermore, such visualizations provide insights into the interaction between the resources spent on disciplines such as implementation and testing or requirements and analysis and design. These insights could, for example, lead to improved project planning practices in terms of a better resource allocation — which implies cost reduction.

Second, analysis of effort distribution is necessary in order to develop a method for project management to gain insight in resource allocation. The effort perspective provides an objective overview of what is happening (or has happened) in a software development project. Team members might work on several projects at the same time. By focusing on the hours spent on specific tasks in a particular project, the dynamics of the tasks that are executed can be better understood in isolation than by using observation. In addition, observations from visualization of effort data during a project elicit trends and provide a view of the progress of a project.

Third, effort distribution visualizations are presented to follow up on earlier work on effort visualization. A figure that is commonly referred to in the context of project planning is the "hump" figure used in the documentation for the RUP that depicts the effort that would be spent during a project on each of the nine disciplines RUP prescribes. Port et al. (2005) attempted to validate the RUP hump diagram earlier by means of student experiments. They conclude that the visualization of their data was similar to the RUP hump image. Contrastingly, this study presents data that was obtained from industrial practice to empirically validate the hump image. The work of Port et al. has been followed up before (Heijstek and Chaudron, 2007) albeit on an aggregate level. This study examines individual projects.

Another study in which RUP humps are redrawn based on a project data has been executed by Hindle et al. (2010). In their study, a variety of existing artifacts is used to draw RUP Humps of two major open source projects. Hindle et al. concluded that for both projects, the humps *"allowed [them] to find interesting requirements- and analysis- related behaviors"*. In particular, they found that they could uncover important events — such as major component re-designs — that would not have shown up in common project metric overviews. The visualizations made by Hindle et al. focused on product software (e.g. they used 14 and 9-year timescales) whereas in this work, we focus on greenfield[1] project-based software development. Timescales in this domain are measured in months.

The structure of this chapter is as follows: The following section (3.2) will elaborate on related work regarding RUP "humps". Section 3.3 explains the research method and Section 3.4 outlines the results. Section 3.5 explains the threats to validity, Section 3.6 contains a discussion of the findings. Finally, Section 3.7 contains our conclusions and future work.

3.2 Related Work

In this section "RUP humps" and related studies are discussed.

3.2.1 RUP Humps

The term RUP "hump" refers to a graph of effort spent over time during a particular discipline. The RUP hump chart consists of a collection of humps for all RUP dis-

[1] also referred to as bespoke software development: the development of software "from scratch" as opposed to e.g. product software development, based on prior releases

ciplines. This diagram was created in 1993 during a workshop on architecture and process (Kruchten, 2003a) and was inspired upon work by Booch (1995b) and Boehm (1986, 1988). It has been part of the Rational Objectory Process after reviews by Dyrhage and Bylund and moved on to play a more important role in the RUP in 1998 when it served as the opening page for the digital version of the process (Kruchten, 2003a). Its final form was published by Kruchten in 1998 (Kruchten, 2003b). An older version was later used by Jacobson et al. (1999) and an altered version was used by Royce (1998). A recent version of the RUP chart is depicted in Figure 1.3. Over the years this diagram has become increasingly connected with RUP in such a manner that it is sometimes perceived as a logo for the process. IBM refers to the RUP Humps as the *"widely recognized RUP Lifecycle Diagram"* (O'Neill, 2007). The chart has been spread widely over the Internet. A known misconception about the hump chart is, that it is based on empirical assessment of actual projects rather than on the educated guess of Kruchten.

> *"...I always insisted that these humps were just illustrative, as well as the number and duration of iterations shown on the horizontal axis, but many people wanted to read much more meaning in that diagram than I intended. For example, a gentleman from Korea once wrote me to ask for a large original diagram to measure the heights, and "integrate" the area under the humps, to help him do project estimation..."* (Kruchten, 2003a)

3.2.2 Other Related Work

Port et al. (2005) tried to empirically validate the RUP hump chart. They assessed the effort spent in a group of 26 student projects which served as an introduction to software engineering. The projects had a lead time of 24 weeks. The students participating were all graduate-level students at the University of Southern California's Center for Software Engineering. All projects were structured around the CS577 Model-Based Architecting and Software Engineering (MBASE) guidelines (Boehm et al., 1999). In their research, Port et al. create a mapping from the CS577 effort reporting categories to the RUP disciplines and they note that, although CS577 projects are representative of RUP projects, they *"do not strictly follow all the RUP guidelines."* Their finding was that *"CS577 projects generally follow the suggested RUP activity level distributions with remarkably few departures."* An important difference between the experiments conducted by Port et al. and the study in this chapter is that their effort was already reported in terms of RUP disciplines. An effort mapping was therefore not necessary.

Hindle et al. (2010) report on a study in which they employ visualizations that are consistent with the RUP hump chart. Their objective is software process recovery. Hindle et al. used data from mailing-list archives, version control systems and bug-tracker systems to draw what they refer to as Recovered Unified Process Views (RUPVs). In their study, they present two cases. First, they draw RUP Humps of the development process of the open source operating system "FreeBSD"[2] over a period of 16 years. Second, they reconstruct the development process of SQLite[3] over a period of 10 years. Lacking precise effort data, they use techniques such as word-bags and topic analysis to reconstruct discipline activity. Hindle et al. concluded that the humps *"allowed [them] to find interesting requirements- and analysis- related behaviors* and that they *"were able to find important events that would not have shown up in a commits per month signal."* Hindle et al. note that their humps are not only useful for project managers, who can use the visualizations in their dashboards, but also for (new) developers, unfamiliar with the project culture or consultants or investors want to gain an overview of a project's processes. For this study, we specifically use large industrial software development projects as opposed to open source projects concerned with product software development.

3.3 Methods

In this section, the methods used for our study, are outlined. Detailed hour registration data was collected from the software development department of a large IT service provider (organization ABC from Chapter 2). This data was visualized, consistent with the RUP hump chart and these visualizations were analyzed. Finally, senior project members were confronted with the process visualizations. The following paragraphs describe the research environment, the data collection process, visualization process and the validation of the data.

3.3.1 Project Context

Data is collected from four industrial projects developed by a single software organization. Within this organization, specific departments offer services to software projects. These services include estimation and measurement, "assembly line" support (e.g. development environment configuration), process coaching, tool support and infrastructure support. The estimation and measurement department is responsible for quantitative analysis of projects before, during and after execution. The assembly line department offers "continuous integration" services with regard to software development. Process coaches are responsible for providing help and training to project department members to help them to work more efficiently, more effectively and according to RUP specifications. The process coach uses the output of the estimation and measurement department, the assembly line and interviews with project members to assess the status of the project and to seek for areas of improvement. The tool support department is responsible for the tools that are used for supporting the services. Tools for version control, change and defect tracking and modeling of requirements and

[2]http://www.freebsd.org/
[3]http://www.sqlite.org

design are supported by this part of the facility. The infrastructure department is responsible for offering the technical capabilities to make use of all services. Besides supporting computer hardware and being responsible for project hardware and backups, the infrastructure department configures and maintains virtual environments for project members to work in. The business modeling discipline is not used within the software development organization as this part of projects is done by a different organization.

3.3.2 Data Collection

Data was primarily gathered by means of extracting data from the applications CA Clarity[4] (an hour registration system), Open Workbench[5] (a front-end to Clarity), IBM ClearQuest[6] (a defect tracking system) and the log files of source lines of code (SLOC) counters. These data were triangulated by examining various other sources of electronic data: As a first check, project documentation stored in the software configuration and change management system (SCCMS, IBM ClearCase[7]) systems such as management summaries and memos were consulted. Incomplete or inconsistent data was later compared to the measurement reports created by the Estimation and Measurement department of which backups are kept on the development department's own servers. These servers contain information on both current and past projects in which the development department's services were used. If ambiguities regarding project data still exist after consulting the prescribed administration systems, the informal project documentation and the measurement assessments, the project's process coach and project manager were consulted.

3.3.3 Visualizing Effort Data

Visual representations were made by automated interpretation of effort information that was entered by project members into Clarity. We created a custom view for the effort data so that the columns task type, task description, effort in person-hours, starting-date and ending-date and task effort for each week of the project were listed in that particular order. The ordering of the items was hierarchical so that a project consists of phases, phases consist of iterations, iterations consist of disciplines and disciplines consist of tasks. The data structure of the log files is depicted in a class diagram in Figure 3.1. These log files were analyzed by means of a set of GNU Bash[8] and Python[9] scripts that counted the amount of time and effort that were spent on the

[4]http://www.ca.com/us/project-portfolio-management.aspx
[5]http://sourceforge.net/projects/openworkbench/
[6]https://www-01.ibm.com/software/awdtools/clearquest/
[7]https://www-01.ibm.com/software/awdtools/clearcase/
[8]https://www.gnu.org/s/bash/
[9]http://python.org/

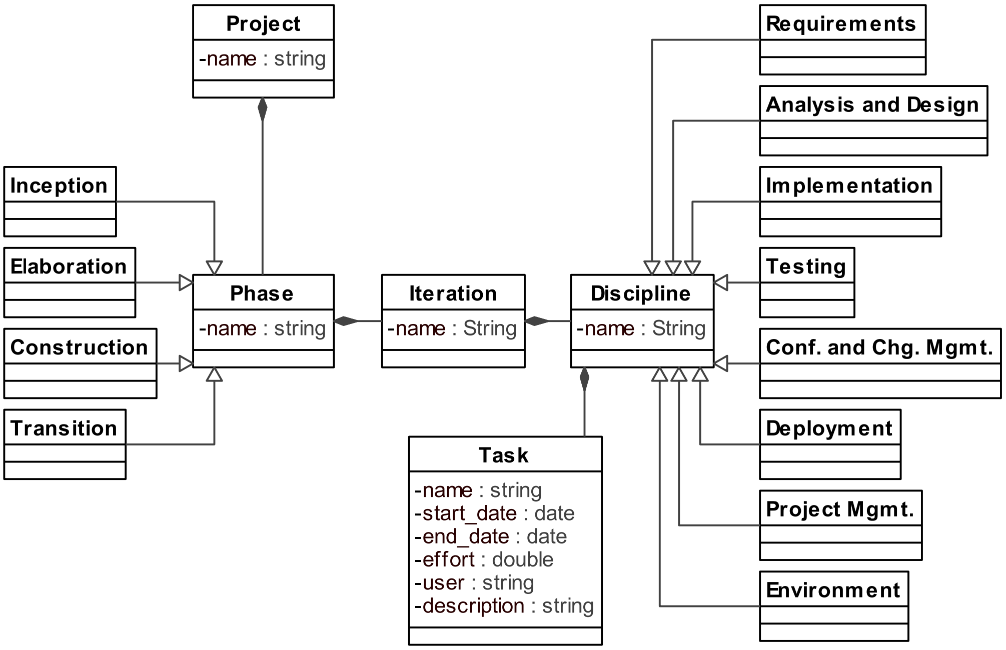


Figure 3.1: *Class diagram depicting the structure of the examined effort log files*

task-level. Then, both time and effort data were normalized and data points in the form of

$$x = \left(\frac{\text{task effort}}{\text{discipline effort}} \right) \text{ and } y = \left(\frac{\text{task time}}{\text{project time}} \right) \tag{3.1}$$

The data points for each discipline were then visualized by means of R (R Development Core Team, 2011) package `ggplot2` (Wickham, 2009).

3.3.4 Validation

After the projects were finalized, the process visualizations were validated with senior project members such as the project leader and the configuration manager. Also, the estimation and measurement department members were asked to elaborate on the humps. Questions asked during these unstructured interviews include:

- To what extent can you recognize the development strategy in the image?
- What other factors influence the visualization?
- Can you explain the reason for the amount and length of the phases and iterations?
- Would you find it useful to see these images during a project?

- Why is there a certain anomaly or unusual effort peak depicted in a certain phase or iteration?

3.4 Results

For this study, four projects were analyzed. All projects were executed by the same IT organization but for different clients, in different domains, under different circumstances and with different team members. The projects took place over a period of 7 years. RUP was adhered to as strictly as possible as this is stimulated by the IT organization in general and by the process coaches, discussed earlier, in particular. Also, the project leader as well as the other team members already had experience with using RUP. The IT organization emphasizes cooperation with the client and therefore primarily defines success in terms of client satisfaction. In the standard post-mortem analysis client interview process, on average, all projects scored over 4 on a scale of 1 to 5. Table 3.1 contains an overview of relevant project characteristics. The following subsections will describe the effort distribution visualizations, elicit the striking phenomena that can be observed from these images and try to explain these occurrences for each of the projects. Lastly, the explanations for each phenomenon given by involved project seniors will be described.

Table 3.1: *Project Characteristics*

	project A	**project B**	**project C**	**project D**
application type	webshop	administration	administration	search
domain	education	insurance	financial	government
dev. language	.Net	.Net	Java	.Net
func. points	866	912	2,000	600
person-hours	8,941	11,492	53,837	14,536
peak staff (FTE)[1]	7	12	28	13
sched. pressure	yes	no	no	yes
cost structure	time–material	fixed price	fixed price	fixed price
offshore	no	no	yes	yes

[1] Full-Time Equivalent

3.4.1 Project A

Project A consisted of building a web-enabled content management system and business-to-business web shop. The client was from the educational domain. The project employed 13 to 15 people with a peak of seven full–time equivalents during the construction phase. 866 Function points were realized in 8,941 person–hours and resulted in 80,000 source lines of code. During the execution of the project, the requirements changed and were expanded to a great extent. At the start of the project, the project manager had 4.5 years of experience in managing IT projects. Project A was executed under schedule pressure due to time limitations. Project A used some agile practices such as daily stand–up meetings and writing code with the responsible testers, the end users and the designers in the same room. The reason for applying these practices was the volatility of the requirements. The effort distribution visualization for project A is depicted in Figure 3.2. The horizontal axis was scaled to fit the entire

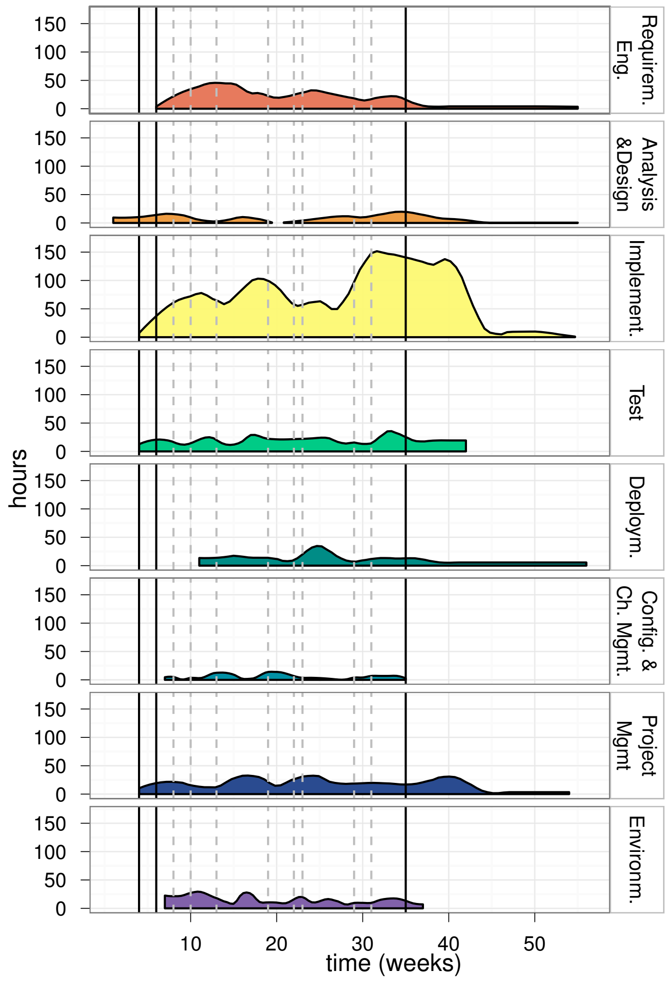


Figure 3.2: *Effort distribution visualization for project A*

project span. The black vertical lines on the vertical axis represent the phase delimiters and the gray, dotted lines represent the iteration delimiters.

In Figure 3.2, many iterations can be identified of which the varying length and the long third (construction) phase, are the most striking features. In the interviews, the amount of iterations showed to be correct whereas the length of the iteration was not always correct. The amount of iterations was said to be relatively high because of the volatility of the requirements. The different iteration lengths are the result of the fact that the effort logging database was used for hour registration and that this registration served directly as the basis for invoices for the client. Because there were so many iterations and because there was a certain amount of schedule pressure during the construction phases, setting the exact dates for each iteration was not a top priority. The phases were confirmed to be correct.

In interviews with project leaders, it emerged that explanations for the resource allocation as represented by the RUP humps were sometimes not directly related to software-related events. For example: in all disciplines, sudden drops of effort can be seen. The drop in all disciplines around 30 weeks can be attributed to a national holiday. Other drops were associated with team training or illness of team members.

The fact that, for example, the analysis and design and implementation disciplines are still in a peak around 40 weeks, makes the apparent ending of the project around that time seem abrupt. The project leader confirmed that, during development, the system was tested in the production environment as a result of problems with simulating the production environment. The effort spent in the last 10 weeks of the project is not logged as a result of the schedule pressure.

When compared to the original RUP hump chart in Figure 1.3, the visualization of effort distribution of project A shows distinct differences with regard to how analysis and design effort is spent. In the original RUP hump the analysis and design effort peaks early and peaks several times later, albeit somewhat lower, as the design is reworked in hypothetical, consecutive iterations. In Figure 3.2 we see that more effort is spent on analysis and design around week 35 (the beginning of the transition phase) than in the initial stages of the project. Besides changes in requirements that have fundamental impact on the design of the application, this could indicate that the first construction iterations were based on a poor design which was reworked later. The latter was the case: Rework in the design was caused by architectural decisions which were not confirmed. This rework is a pattern that can be clearly deduced from the visualization. The peak of implementation effort that can be seen around week 40 is a direct effect of this architecture redesign.

An important remark during the interview with the project leader of project A was that the effort for the requirements, analysis and design and implementation disciplines could essentially be combined in one hump to more accurately represent the spending of person-hours. A team member with the role of programmer who participated in a requirement specification workshop, recorded his or her working hours as effort spent on implementation. This finding implies that the RUP defined roles and disciplines were not always strictly used as separate entities from an effort registration perspective. The reason for this was the cost structure for project A which required every hour to be recorded according to price. In this cost structure it is more important to know the amount of hours that a certain team member worked because different team members have different rates associated with them.

3.4.2 Project B

During the execution of project B a car insurance application was built. The final application had to interface with various, already existing databases. At the peak of the project, during the construction phase, 12 FTE were working in the project simultaneously. During the transition phase, this amount was reduced to 0.5 FTE. Before project execution, 912 function points were counted. At the time of application deployment, a total of 62,000 SLOCs were delivered. The project produced more lines of code than predicted due to client–induced limitations on the architecture, a complex application front-end which could not be expressed in function points and a range of client change requests during the project which caused the functionality of the system to expand. Project B was a fixed price project. Figure 3.3 displays a visualization of how effort was distributed over the RUP disciplines for project B. As was the case for project A, all disciplines but the business analysis discipline were used.

The overall project trend was recognizable for the project leader who was interviewed about the visualization results. The project started as a traditional RUP project but at an early stage, the client thought the tempo of the project was too high. Consequently, staff was reduced. The effects of this decision is clearly visible in the visualization at around week 15 as requirement, analysis and design and implementation effort dips.

A striking feature of Figure 3.3 is the low amount of effort that is spent in the second half of the project. The effort spent in this last — transition — phase is spent by one person who works on the project 50 percent FTE. The long transition period was attributed to client change requests (RFCs), infrastructure problems at the client deployment site and dependence on other projects which were executed by different organizations at other locations. In the transition phase, most effort is attributed to the implementation stage. This, however, is not correct as the 0.5 FTE assigned to the project was responsible for multiple disciplines. Although time was spent on requirements, analysis and design, implementation, testing and deployment, this person choose to attribute all effort spent to the implementation discipline, due to time constraints. This portrays the central role that the implementation discipline plays and how this discipline is used as a default for effort logging under time pressure.

Figure 3.3 displays a large amount of phases and iterations. Not all phases depict real phases. Instead, some phases were used for registering hours for impact analysis and changes. The iterations were all recognized by the project leader. For example, during the construction phase, six iterations were executed in which six sets of use-cases were implemented.

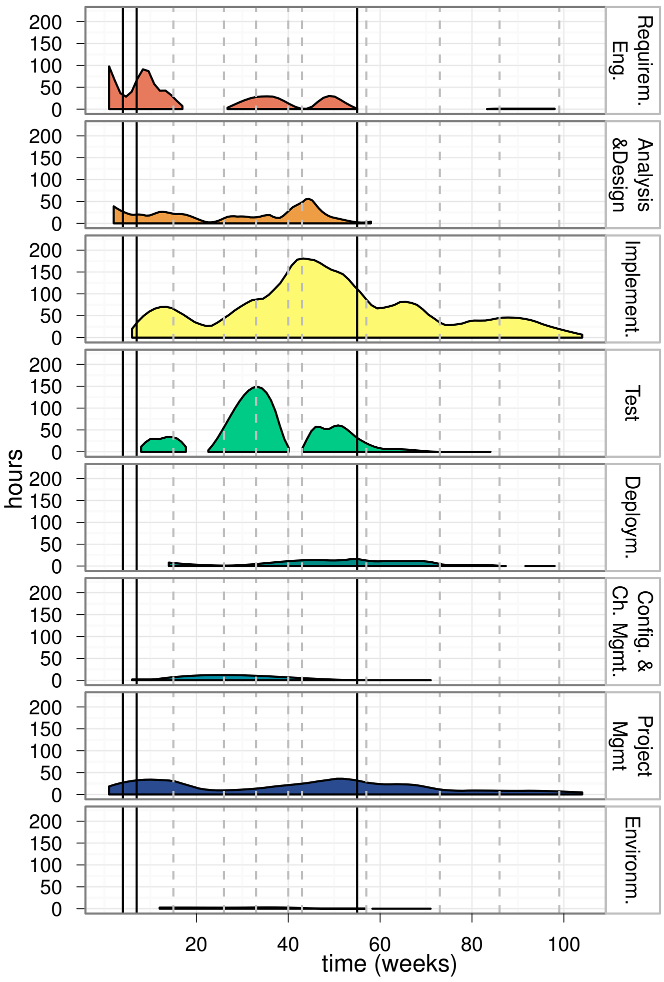


Figure 3.3: *Effort distribution visualization for project B*

3.4.3 Project C

The objective of project C was to develop a web-enabled registration system for various types of financial products. The client was a large, international financial organization. Project C employed model-driven development techniques and was partly executed offshore. During the two years the project ran, it had various difficulties both due to the complexity and novelty of the model-driven development tools and techniques and due to problems associated with offshore development. The requirements for this project were not particularly volatile. However, the software architecture was not very stable and difficulties existed with communicating the architecture to the offshore development location. The effort distribution visualization for project C is depicted in

Figure 3.4. In this visualization, only six disciplines used by the project team are shown.

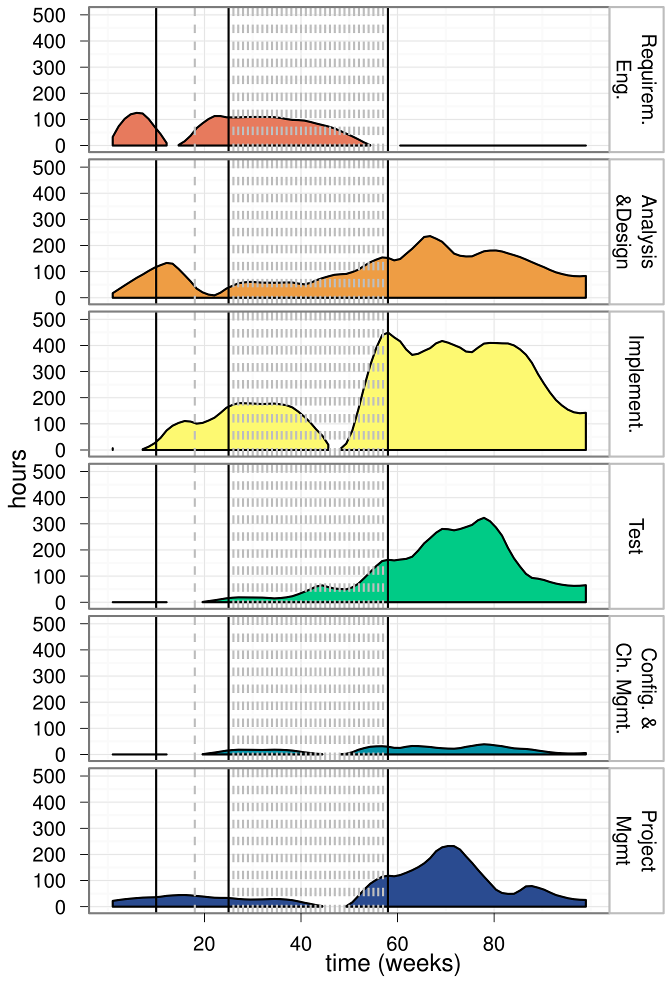


Figure 3.4: *Effort distribution visualization for project C*

The business modeling, deployment and environment disciplines are not used. The developed system was never deployed at the client site and the environment discipline was not seen as necessary.

The most striking feature in the RUP humps for project C are the 33 iterations that comprise the third (construction) phase. While RUP was used as a development method, during the construction phase, a Kanban approach (Ohno, 1988, Poppendieck and Poppendieck, 2003) was used. As a result, iterations were weekly. In the second (elaboration) phase, the only other iteration line can be seen. The Kanban approach also explains why requirements engineering effort is almost as high as it is in the first two phases, during most of the construction phase.

The effort distribution of project C, is further removed from the 'reference humps' than Projects A and B were. The reasons for this seem to be both the model-driven nature of this project and the fact that an offshore development team (GSD) was employed. Three reasons lead to this observation:

First, the code was generated from models. The effort logged as analysis and design can therefore be regarded as implementation effort. The effort spent on implementation should be split between effort spent on the code for the project and the code of the code generator. The effort data was not detailed enough to allow to deduce this distinction in the data.

Second, most effort analysis and design and implementation effort seems to be spent during the last (transition) phase. Part of these phenomena can be explained by the fact that the model-driven development paradigm was new for the development team. However, project management and team members mostly explain these late effort spikes as having to do with issues related to the difficulties of collaborating the with offshore development team. The project ran over time and over budget to a serious extent and had to ramp up development effort during the transition phase. Consequently, technically, the transition phase was another construction phase. The project management effort spikes in the last phase are explained to be a direct result of solving communication problems with the offshore team. The complexities of the meta-model used for code generation, were the reason that a only very few knowledgeable experts in the (onshore) development team were available. As they were all senior developers, they constituted costly resources that the contractor preferred not to be consulted too extensively. This inhibited offshore developers from attaining an understanding at a similar expert level. As a consequence, elaborate onshore guidance was needed throughout the project. Onshore support became even more vital when development was ramped up in the later phases of the project as more questions arose. The relatively high test effort, overall, was explained to be a result of examining consistency of implementation with the meta-model, or architecture compliance checking.

3.4.4 Project D

The objective of project D was to build a document retrieval system that was to be used by various governmental organizations. The employed development methodology was RUP. The project employed transfer-by-development stage offshoring (Mockus and Weiss, 2001) where different development stages are executed at different locations, sequentially. As a result, the inception and elaboration phases were executed by a small onshore team and implementation was executed by an offshore team of developers. The effort distribution visualization for project D is depicted in Figure 3.5. Disciplines missing from the effort registration of Project D were business analysis, environment and configuration and change management. Environment and configuration and

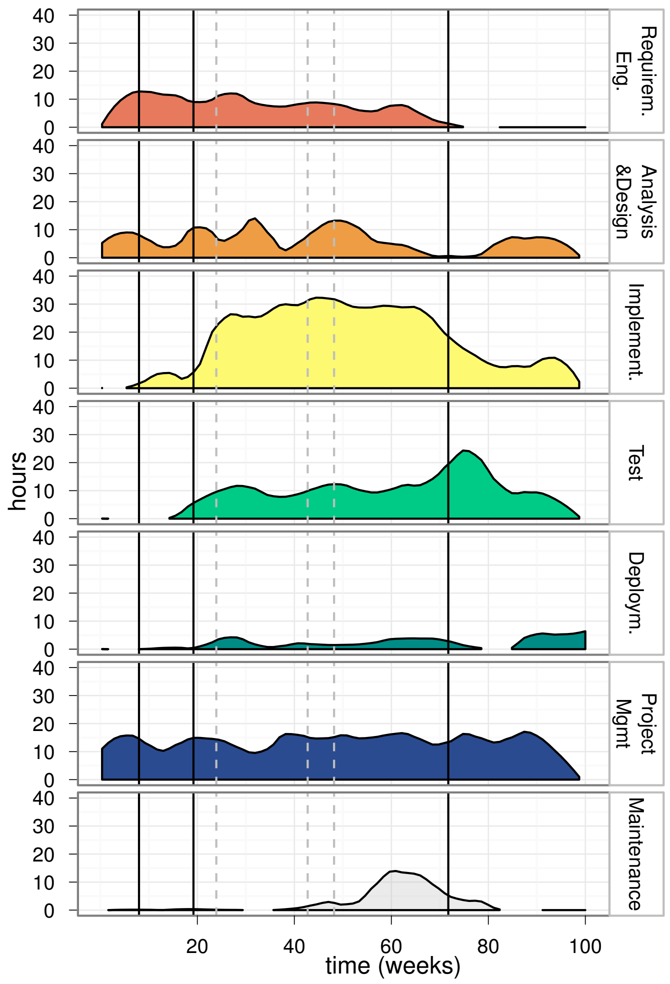


Figure 3.5: *Effort distribution visualization for project D*

change management effort were said to be merged with project management effort. The maintenance discipline is not an official RUP discipline. It was used for this project as the contractor was also set to maintain the system they were building. Therefore, it was seen as important to implement the system ensuring maintainability. Activities to that end were separately logged as if they were part of a maintenance discipline. Most effort spent on this discipline is implementation and analysis and design effort.

A striking feature is that the iterations in the third phase have an unequal length. This is the result of a feature-oriented iteration strategy. Not all features were of equal size and as a result, iteration length differed. In this project, a substantial amount of effort was required to ensure that the offshore development team complied with the architecture made onshore. This impact of the use of an offshore development team is visible in various parts of the hump image. First, the analysis and design discipline peaks in the fourth (transition) phase. It also runs until the end of the project. These features are caused by the effort that was spent communicating and altering the architecture design. Second, the test discipline hump is relatively large. The explanation for this phenomenon is that a significant amount of time was spent checking architecture design compliance. Third, even when accounting for the environment and configuration and change management effort being merged with project management effort, this discipline has an unusually large hump associated with it. Various changes in the offshore team composition required project management effort to increase. On the one hand, new team members needed to be made familiar with the project and the system's architecture. On the other hand, due to these team composition changes, the project became delayed and had to be re-planned.

3.5 Threats to Validity

Correctness of hour registration data directly influences data visualization. For example, the iteration mismatches and the sudden end of the effort distribution data that can be seen in project A is an artifact of the hour registration rather than a process change. These effects were uncovered by validating the visualizations with project leaders. Remarks of project team members include that the distinction between requirements and analysis and design disciplines is not always clear when logging effort data. We manually reclassified task descriptions and moved tasks between these two disciplines when necessary. In the case of project A, the testing discipline is also confused with requirements and analysis and design at some occasions. We again reclassified tasks based on their description when needed and validated this process with at least one project team member. Project B registered extra phases for change management (CM) and changes for administrative purposes. Also in project B, the implementation discipline was used to attribute effort to that in reality was spent on other disciplines. This merge was the result of time constraints. This mismatch poses a possible treat to validity. Because of these mismatches, the data in the hour registration system can not always be used as they are and should be reclassified and validated before they can be used for process analysis.

3.6 Discussion

The visualizations gave an insightful impression of spending of person-hours. Certain patterns were clearly visible. An example of such a pattern is the pattern seen in project A: The rework that had to be done on a poor design and had clear implications on various disciplines later in the project. This is a clear and understandable example of under-spending resources during the design phase which force a project to spend extra resources on the design in a later stage of the project. Contrastingly, if too large an amount of effort would have been spent on the design, the project would have been forced to spend less time on subsequent disciplines to prevent project overrun. Therefore, finding a balance between the amount of resources to be spent on disciplines prevents that the resource allocation is dictated by shortages. Another pattern that could be observed was the impact of distributed teams in projects C and D. Observed influences of employing offshore development teams on effort distribution were increased analysis and design for communication of architecture and increased test effort for increased testing of architecture compliance.

RUP's distinction between engineering and supporting disciplines is not a good predictor of which disciplines are thought to be most important to log. The disciplines that are most often missing from an effort registration system are business modeling, deployment and environment — of which the latter two are defined as engineering disciplines. The business analysis discipline was missing from all four of the cases under study because this particular task is not part of the expertise of this department. When projects run late, project management effort increased because of the required rescheduling of the work that remains to be done. Also, the client needs to be managed more intensively. This change is either logged in detail or it leads to a situation in which little to no effort is logged because of schedule pressure. Non-standard development approaches have significant impact on the shapes of the humps. An example is the model-driven development approach taken in project C. In this project, modeling is synonymous with implementing. The definitions of the analysis and design and implementation disciplines are therefore less clear.

According to the project leaders, all four effort distribution visualizations give an accurate indication of how effort was actually spent globally. However, RUP's flexibility led to differences in how effort was recorded. From the feedback during the interviews it became apparent that formal arrangements regarding expenditure, such as cost–structure, influence effort registration. This problem should be accounted for in future exploration and analysis of effort registration data. Using separate applications for logging effort data for analysis and for billing purposes can help to increase data comparability. However, effort registration is often not a priority in commercial software development. The objectives of scientific analysis of a software engineering project and the objectives of the project itself are conflicting. The prime objectives of a software project are to deliver relevant and functional software in a timely manner. Contrastingly, the benefits of scientific research, such as in this case, quantitative post–mortem project analysis, are not directly relevant to the client. Also, such analysis does not guarantee results and if it does, those results may be difficult to operationalize on the short term and so they constitute a long–term investment. Logging effort distribution poses other problems such as the challenge of defining what type of effort should be logged or the possibility that team members may see detailed logging of their activities as intrusive and a threat to their privacy.

3.7 Conclusions and Future Work

In this chapter, we followed up on a method for visualization of software development process effort and adapted it to provide a view on how resources are allocated in large-scale, custom software development projects. Both distributed and co-located projects were used. Evidence was found for aberrant distribution of analysis and design effort in projects in which offshore development teams are employed. These aberrations are related to unclarities related to communication and coordination of software architecture.

The visualizations of how effort was distributed over RUP disciplines were seen as useful in the sense that they can play a role in verifying to what extent resources should have been spent. As one project leader put it: *"The [RUP hump] image should not yield any surprises [at any given time during project execution]."* The visualizations are mainly dependent on a few factors such as the type of project in terms of cost or billing structure and the definitions of RUP disciplines used. In the organization in which cases A, B, C and D were executed, the visualizations are to be used as a standard extension to the tools used for post mortem project analysis.

More data is needed in order to categorize software development processes. Collecting data that was recorded in a uniform manner can help us determine patterns of effort distribution and to relate these patterns to various project specific success or failure related factors. Comparing average RUP humps for organizations can give insights in typical decisions taken in terms of project management style or the implicit organizational attitudes with regard to the software engineering process and to what extent these have a structural impact on project results.

The RUP hump plots can be extended to include a cumulative effort plot per discipline and plots of the cumulative number of source line of code, defects found and functionality realized over time measured in, for example, function points or use case points. The plots can then be used during project execution to analyze the project status and they can also become a part of a standard project post-mortem.

Chapter 4

A Multiple Case Study of Dissemination and Coordination of Software Architecture Design in Global Software Development

In this chapter an analysis is presented of software architecture dissemination and coordination practices in the context of large scale, global software development. To this end, a theoretical framework for the software architecture dissemination and coordination process is outlined. Using this framework, the software architecture dissemination and coordination processes of three cases of global software development are analyzed.

4.1 Introduction

Software architecture is an important artifact by means of which non-functional software qualities such as security, maintainability, extendability and portability can be addressed and guaranteed. Software architecture is disseminated to software developers in different ways depending on project characteristics such as project size, software complexity and the technological and organizational maturity of the client. Two main strategies to architecture knowledge dissemination can be discerned:

- A personalisation strategy, where, *"knowledge is closely tied to the person who developed it and is shared mainly through direct person-to-person contacts"* (Hansen et al., 1999).

- A codification strategy, *"centers on the computer and where knowledge is carefully codified and stored in databases, where it can be accessed and used easily by anyone in the company"* (Hansen et al., 1999).

Software architecture is generally disseminated through a mix of both these strategies. This is referred to as a *hybrid strategy* to knowledge sharing (Desouza et al., 2006). In traditional waterfall-approaches, this hybrid strategy will have a large element of codification. In agile (Highsmith and Fowler, 2001) approaches, the personalisation strategy is generally more dominant. In both cases, architecture documentation plays a central role. Software architecture documentation facilitates stakeholder communication and is instrumental in ensuring that essential design principles are adhered to by the code. Developer understanding of software architecture and its rationale is believed to be beneficial for software quality. Most development methodologies prescribe creation of a document that holds software architecture information. For example, in RUP, a methodology widely applied in industry, architecture is central and captured in a Software Architecture Document (SAD) for which a detailed template is used. This template is structured around Kruchten's 4+1 layer view on software architecture (Kruchten, 1995). While we are aware that this document is very often created in RUP projects, we are unsure what role such architecture documentation has in the software development practice. For example, how much of developer knowledge regarding a software architecture stems from this software architecture documentation as opposed to knowledge obtained from other sources such as colleagues? The advent GSD complicates informal personal communication in general and dissemination of software design in particular. A common approach to GSD is a type of "transfer by development stage" (Mockus and Weiss, 2001) where requirements gathering and analysis and design activities take place at a different geographical location compared to where the implementation work and unit testing activities are carried out. Complicating matters further, offshore developers are often unable to directly contact members of the architecture team due to geographical separation. Synchronous communication is often difficult due to time zone differences. Even if an architect can be contacted, communication can be hampered by socio-cultural differences and language barriers. GSD effectively mitigates informal communication while this type of communication is generally seen as an important element to disseminate architectural knowledge. This chapter reports on three case studies conducted to elicit architecture dissemination practices and the role of software architecture documentation in sizable GSD projects.

This chapter is structured as follows: The study objective is discussed in Section 4.2. Subsequently, related work and the research methodology are addressed in Sections 4.3 and 4.4. Cases A, B and C are discussed in Sections 4.5, 4.6 and 4.7, respectively. Conclusions and Future Work are outlined in Section 4.8.

4.2 Objectives

Understanding the processes and practices used for dissemination of architecture in GSD projects yields potential benefits in terms of more unequivocally conveying design decisions fundamental to a software system. Better understanding of a system's intended architecture by developers is of manifold importance. It directly benefits a project in terms of compliance to functional and non-functional requirements and thereby avoids expensive rework. It also benefits the structural integrity of the software in terms of aspects such a security, maintainability and portability.

This chapter addresses **RQ1** (Section 1.3). In Chapter 2 we analyzed software development process descriptions and in Chapter 3 we analyzed industrial instances of software development processes. In this chapter, we use a more detailed, qualitative approach to analyze industrial instances of software development processes. We aim to understand the process of software architecture dissemination in the context of global software development and therefore pose the following research question:

> *How is software architecture coordinated and disseminated in large, industrial, custom, global software development projects?*

As knowledge dissemination takes place via multiple methods we need to define sub-questions:

1. How is software architecture design and dissemination organized?
2. How is software architecture documentation used?
3. What is the role of the architect(s) during the software development life cycle ?
4. How is architecture compliance organized?

To this end we conducted three case studies of large, industrial custom GSD projects for which the architecture design was made in the Netherlands and the implementation was made in India.

4.3 Related Work

In this section we specifically address related work on the role of documentation in software develop processes, software architecture understanding and knowledge transfer in GSD.

4.3.1 The Role of Documentation

Work by Huysman and Wulf (2005) outlines that people have a preference to utilize personal networks over electronic networks to obtain knowledge in the context of sharing experience. However, as mentioned, software architecture documentation plays a role in development methods of different natures. Studies reporting on the use of software documentation during software development in general are few and report mixed results. In studies analyzing developer preferences regarding documentation, Forward and Lethbridge (2002) and Lethbridge et al. (2003) find a preference for simple and powerful documentation and conclude that documentation is an important tool for communication, even if it is not up-to-date.

Already in earlier case studies, such as a study by Walz et al. (1993), we find a general reluctance for creating documentation. In general, documentation is seen as distracting from the actual work of developing software. The popularity of agile development approaches attests to this in part. Because the second of the four main "commandments" mentioned in the agile manifesto (Highsmith and Fowler, 2001) reads that, *"working software [is valued over] over comprehensive documentation,"* agile development methodologies are generally seen as document-light. As a result, popular agile derivatives such as Scrum (Schwaber and Beedle, 2001) prefer direct communication over documentation (Abrahamsson et al., 2003, Dybå and Dingsøyr, 2008, Clear, 2003, Rubin and Rubin, 2011) and therefore tend to be interpreted (and applied) as documentation-averse (Stettina and Heijstek, 2011a). While in their manual for Scrum software development, Pries and Quigley (2010) emphasize that, *"the scrum approach is not document-averse but, rather, seeks a leaner solution to formulating the requirements for the product,"* adopting an agile approach to software development often implies that little to no documentation is developed. In fact, agile development is described as an antonym for so called "document-driven" software development (Sillitti et al., 2005). In one of the few contributions to understanding of documentation in agile projects, Clear (2003) points at the behavior of students and observes documentation being perceived to be something external. Instead of being produced in-line with the system as natural part of the development process, documentation was often hurriedly pieced together at the end of a project.

The small role that documentation plays in agile development is underlined by Dybå and Dingsøyr (2008). In their thorough structured literature review on evidence on agile software development practices, they find that documentation is addressed in just one of the identified studies. So little documentation is apparently used in agile projects that in an international study of agile teams, practitioners noted that they found that documentation was important but that too little of it was available in their projects (Stettina and Heijstek, 2011b). This ambivalent position towards documentation might be well explained by Parnas' observation that, *"The solution is neither to add more documentation nor to abandon documentation — it is to get better documentation,"* (Ågerfalk and Fitzgerald, 2006). In fact, as noted, even code-centric, lightweight methodologies advise to create a form of architectural documentation (Smith, 2001).

4.3.2 Software Architecture Understanding

Software architecture can be complex and empirical studies show that developers generally understand mostly "their own" specific components of the application (Curtis et al., 1988). Reasons for a lack of understanding of the software architecture as a whole can be time constraints, poor documentation or a high turnover of staff. These factors can lead to absence of a culture of collective code ownership (Nordberg III, 2003), which is desirable to attain because it yields benefits in terms of software quality and team building. Also, documentation of the architecture is essential for maintenance activities which typically involve different engineers from the ones who developed the system (hence non-verbal transfer of design is needed). Software architecture, *"captures and preserves designer intentions about system structure, thereby providing a defense against design decay as a system ages, and it is the key to achieving intellectual control over the enormous complexity of a sophisticated system"* (Hofmeister, 2000). Existing literature fails to confirm that architecture documentation is used in the maintenance phase (de Souza et al., 2005).

In addition, the effectiveness of using UML to disseminate software architecture design is not yet clear. UML is a commonly used language for software design representation. In their standard work on documenting software architectures, Clements et al. (2002) provide strategies for applying UML for representing various architecture constructs but also note that for e.g. the "component and connector construct", *"all of the strategies exhibit some form of semantic incompleteness or mismatch."*

4.3.3 Knowledge Transfer in GSD

Dissemination of software architecture design is thought to benefit from frequent informal communication, preferably with those who have intimate knowledge of that architecture. Practitioners also feel the need for informal communication (Schneider et al., 2008) and informal communication has been found to be very important (Herbsleb and Grinter, 1999a, Damian et al., 2007, Nielson, 1998), particularly in organizations with fast-changing environments (Kraut and Streeter, 1995, Galbraith, 1977). Even though various solutions for distant informal communication such as instant messaging and video techniques (Fish et al., 1993) exist, Herbsleb et al. (2001) found that informal communication is very different for local versus remote site communication and that people at remote sites are found to be difficult to contact. The use of collaborative tool sets seems to mitigate some of the negative impacts introduced by geographical, temporal and socio-cultural distances (Nguyen et al., 2008). However, these tools are not always implemented or used to their full extent. Codification strategies (Hansen et al., 1999) for sharing software architecture information (Babar et al., 2007) are therefore likely to be more commonly used in GSD settings. This places demands on the clarity, completeness and consistency of software architecture documentation.

4.4 Research Method

In this section, the method for this study is described. This study is reported on in two chapters. This chapter contains a multiple case study. In Chapter 5, we present a synthesis of the factors involved in software architecture dissemination and coordination in the context in GSD as derived from these cases as well as recommendations and distilled best practices.

4.4.1 Data Collection

We applied data source triangulation (Stake, 1995) by collecting data by means of several methods. We employed software repository mining to obtain project data such as functional size estimation, cost structure, project planning and information pertaining to team member time registration. The software repositories furthermore gave us insight in the contents of the requirements, software architecture and design documentation and supplementary specifications. These repositories often contained useful information regarding team communication in the form of forums, discussions on defects reports or change requests and saved e-mails. In addition, we conducted a series of on-site, structured and semi-structured interviews with team members who were or had been active in each one of the projects. For these interviews we spent several weeks in India. We obtained a copy of the SAD which we reviewed. We then interviewed the primary architect, located at the onshore location, asking for:

- the coordination of the software architecture process
- methods employed for dissemination of software design and architecture within the team
- the role of the architect(s) in the development life cycle and the processes
- clarifications regarding the SAD
- reflection upon the above mentioned topics

In the next paragraph we explain the justification for the interview topics in more detail.

4.4.2 Interview Design

We developed a theoretical framework that is based on the "correction system" described by Shannon and Weaver (1949). This framework is depicted in Figure 4.1. In this theoretical framework, we depict our perception of the relation between an architect and a developer in the context of dissemination of software architecture. We structure the framework around the formats or states in which software architecture exist. These states are:

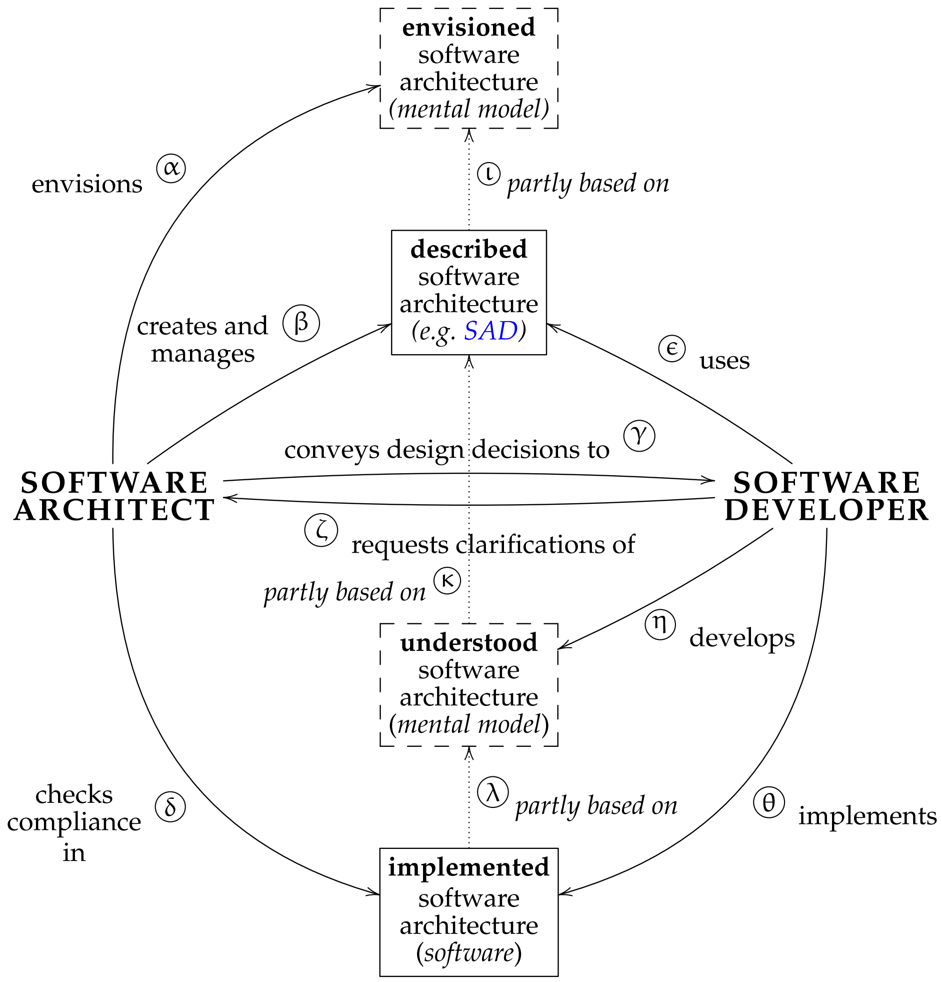

envisioned software architecture *(mental model)*
envisions α
ι *partly based on*
described software architecture *(e.g. SAD)*
creates and manages β
ϵ uses
conveys design decisions to γ
SOFTWARE ARCHITECT
SOFTWARE DEVELOPER
ζ requests clarifications of
partly based on κ
η develops
understood software architecture *(mental model)*
checks compliance in δ
λ *partly based on*
θ implements
implemented software architecture *(software)*

Figure 4.1: *Theoretical framework of software architecture design dissemination*

1. **envisioned architecture** — a *concept* that takes the form of a mental model that exists in a software architect's mind
2. **described architecture** — an *artifact* that takes the form of an architectural representation (such as found in an SAD)
3. **understood architecture** — a *concept* that takes the form of a mental model that exists in a software developer's mind
4. **implemented architecture** — an *artifact* that takes the form of software

The interview questions were targeted to the relations between the concepts and the artifacts. An overview of the question topics related to each link is depicted in Table 4.1. In addition, we asked the architect (1) which parts of the architecture document were important to understand for all developers and (2) to explain some of the most fundamental design decisions with which every developer should be familiar. Project managers were questioned regarding project characteristics, on topics on which they were knowledgeable and we verified information obtained in interviews with other team members. Interviews were taken in closed meeting rooms and recorded. All interviews took place face-to-face and we traveled to India for several weeks to be able to conduct extensive interviews locally. An overview of the team members we interviewed can be found in Table 4.2. Interviewees were asked for their permission for the audio recordings. The recordings were treated confidentially. The average interview length was approximately 60 minutes.

4.4.3 Data Analysis

For analysis, we employed two different techniques: (1) We followed the principles of grounded theory (Strauss and Corbin, 1990) and (2) an adapted form of shared mental model measurement. The use of these methods is described in more detail in the following sections.

4.4.4 Grounded Theory

Audio recordings were transcribed and labeled as follows: We summarized each point that the subject would make into a single sentence. Often participants use multiple sentences to convey a single point and sometimes multiple points are made in a single sentence. These points would be separated into multiple sentences. Transcriptions were done in such a way that each sentence in the transcription would contain a single point in such a way that the sentence could be properly understood in isolation. This one sentence would then be tagged by means of one or multiple tags. The use of these tags is inspired on their use on Twitter[1]. For example:

> *"In my experience, described software architecture, the SAD, is little adhered to by offshore development teams* `#sad_compliance` `#sad_relevance.` *"*

Advantages of using this text-based method are the reliability and modifiability of the data and the abundance of tools that can be used to analyze the data. We employed GNU Emacs[2], GNU Bash[3] and a wide selection of "GNU utilities", all stable and highly customizable, open source and cross-platform tools. For example, a continuously

[1] http://twitter.com/

[2] http://www.gnu.org/software/emacs/

[3] http://www.gnu.org/software/bash/

Table 4.1: *Interview Questions as Derived from Theoretical Framework (Figure 4.1)*

link	core question	interview topics
α	How does a software architect decide on a software architecture ?	**(out of scope)**
β	How does a software architect decide what to describe in an SAD?	architecture design process, intended audience, usefulness of SAD, templates, use of text, UML and box-and-line diagrams, perceptions of SAD quality
γ	How does a software architect disseminate architectural knowledge (to developers)?	dissemination process, perceived understandability of architecture and clarity of SAD, factors in understandability
δ	How does a software architect verify architectural compliance?	compliance methods, compliance in practice, factors in compliance
ϵ	How does a developer use an SAD?	perceived role of SAD, consultation frequency during the project lifecycle, developer input to SAD
ζ	How does a developer request clarifications?	dissemination process, feedback coordination
η	How does a developer arrive at an understanding of SA?	communication methods, perceptions of SAD quality
θ	To what extent does a developer implementation adhere to the intended software architecture?	compliance measurement methods
ι	Which parts of the envisioned architecture are noted in the SAD?	use of templates, architect preferences
κ	Which parts of the SAD are understood by a developer?	prescribed developer knowledge, mental model measurement
λ	To what extent are developers able to implement the intended software architecture?	**(out of scope)**

updated overview of all labels currently used in all interviews, including a count of how often they appear, can be obtained using the following simple Bash script:

```
while true;
do clear;
cat /interviewfiles/*
| awk '{for ( i = 1; i<=NF; i++ ) if ( $i ~ /^#\w/ ) _[$i]++ } END{ for ( i in
    _ ) printf "%s (%d)\n",i,_[i]}'
| sort -u
| pr --columns 4 -w 200 -l 80;
sleep 2;
done
```

For traceability purposes, a separate text file for each transcription was used. No more than two interviews were transcribed per day to avoid mistakes due to fatigue. After transcribing all case-related interviews, double labels were removed. After correction for typographic errors, 132 labels remained. Of these labels, 57 were only used once. We then used grep[4] to extract and group all sentences related to a single label like so:

```
  grep -r "#label" /interviewfiles/case_a/*
```

This grouping provided us with a comprehensive view on each label. We could then proceed to identify the main concepts related to the label. In grounded theory, this is referred to as reduction. This resulted in the following seven different categories:

- case-specific problems and generalizability
- architecture development process
- architecture dissemination and clarification process
- software architecture document
- architecture compliance
- shared mental model deviations
- best practices

4.4.5 Shared Mental Model Analysis

Shared mental models (SMMs) provide teams *"a common set of expectations that enable accurate and timely predictions of approaching needs and issues"* (Cannon-Bowers et al., 2001). Holt (2002) describes software architecture as a mental model shared among

[4] http://www.gnu.org/software/grep/

the people responsible for software. SMMs have been found to positively impact work-team adaptability and performance (Cannon-Bowers et al., 2001, Mathieu et al., 2000). There is little empirical evidence regarding how SMMs affect coordination in more asynchronous and geographically distributed collaboration (Espinosa et al., 2001). To address SMMs, in all interviews, we addressed the following set of topics:

- General project characteristics (e.g. *How far along is the project?*)
- The role of the architects (e.g. *Who is responsible for architecture compliance?*)
- Important design decisions in the system to be build (e.g. *What are the most important design decisions in the architecture?*)

In the following sections, we present and discuss the case findings, structured along these categories. The last step of grounded theory, integration of the concepts by means of induction, will be discussed in Chapter 5.

4.4.6 Case Studies

Three case studies of global, custom software development projects were executed. Each project was executed by the same global IT service provider. Each project had a team in the Netherlands and a team in India. All clients were Dutch. A summary of relevant project characteristics as well as the interviewed team members can be found in Table 4.2. The onshore organization (the front office) is responsible for the project and hires developers in the offshore organization (the back office). The back office is a so-called "cost center" which means that it is paid for the amount of hours that they work. The front office takes the risk for the project in that they take the loss when a fixed-price project goes over budget. Conversely, they take the profit when a project is a success. The following three sections contain the three case analyses.

4.5 Case A

The goal of the project that makes up case A is (1) the expansion of an existing system for indexing and (2) for making information of various governmental organizations searchable. The existing system held in the order of millions of documents and processes millions of queries per month. The new system was to replace the old and was to introduce some new functionality such as moving the responsibility for the (technical) presentation of search results to the client and capabilities for separate management of knowledge models and thesauri. The interfaces to the existing information sources were to be maintained. The system is structured around Microsoft SharePoint (Table 4.2). The system was to be delivered as Software as a Service (SaaS, Papazoglou and Georgakopoulos, 2003). The application is to be maintained by the same organization that is building the system. Functional maintenance, however, will be the (organizational) responsibility of the client.

Table 4.2: *Case Characteristics*

	Case A	**Case B**	**Case C**
client domain	government	private	private
functional size	34 use cases	70 use cases	800 function pts.
plan. duration	10 months	3 months	4 months
process meth.	RUP	RUP	Agile / Scrum
budget	€ 800,000	€ 210,000	€ 400,000
offshore dev's	6	5	4
technology	.Net + Microsoft FAST Search	.Net + Microsoft Office SharePoint Server 2007	.Net
GSD type	*transfer by development stage* (low level design & implementation in India)	*transfer by development stage* (low level design & implementation in India)	*transfer by development stage* (low level design & implementation in India)
project objective	expansion of an existing system for indexing and making searchable information from Dutch government organizations	centralization of a human resources portal for a large, multinational industrial firm	rebuild of an existing Visual Basic 6 application
team members interviewed onshore	delivery mgr. architect project mgr. arch. reviewer test lead	delivery mgr. architect arch. reviewer	delivery mgr. project mgr.
team members interviewed offshore	project mgr. #1 project mgr. #2 architect developer #1 developer #2 developer #3 developer #4	project mgr. sr. developer developer	sr. developer developer

4.5.1 Case-Specific Problems and Generalizability

We first sketch an overview of the main problems that were encountered during the development life cycle:

- For this project, a hard deadline and a fixed budget were agreed upon with the client. Time pressure was high because the strict required date of delivery was overly ambitious according to project leaders and other team members at both the onshore and the offshore location.
- No proof of concept (POC) was built for the solution for this project due to time and budgetary constraints. Various offshore team members referred to how useful a POC would have been to better understand the solution and therewith prevent certain delays.
- Requirements were changed late during the project. Specifically, an authentication system for all external interfaces was required.
- The system under development was to embed a proprietary, external search component for which external training from a third party was required. The training was difficult for team members as not all requirements regarding this component were clear at the time of the training.

With regard to the external validity of any insights we obtain from this particular case, we note that late requirements and the problems regarding a lack of a solid design and POC that arise from schedule pressure, are not at all uncommon. On this matter, offshore or back-office architect (BOA) noted that this project, *"was a typical high-time pressure project."* The front office architect (FOA) remarked that the problems that he encountered during this project are very similar to the problems he encountered in other GSD projects in which he either worked or heard about. The offshore project leader insisted that the project adhered to internal quality regulations and that it scored a three out of three on quality audit.

4.5.2 Architecture Development Process

Originally, the offshore team was to deliver the project's software architect in order to save costs. A FOA was intended to be associated with the project part-time to function as a coach for the BOA. This BOA visited the onsite (or: client-) location and had three weeks to develop an architecture. However, the onshore project leader and FOA decided that the quality of the work of the BOA was insufficient and that the FOA was to finish the architecture. The offshore team presented a different view on why the BOA was not able to create an architecture during his time in the Netherlands: *"The time I spent in [onshore] was not enough because of the use of an external component that was completely new for us. In addition, the learning time that was planned during the bid phase was*

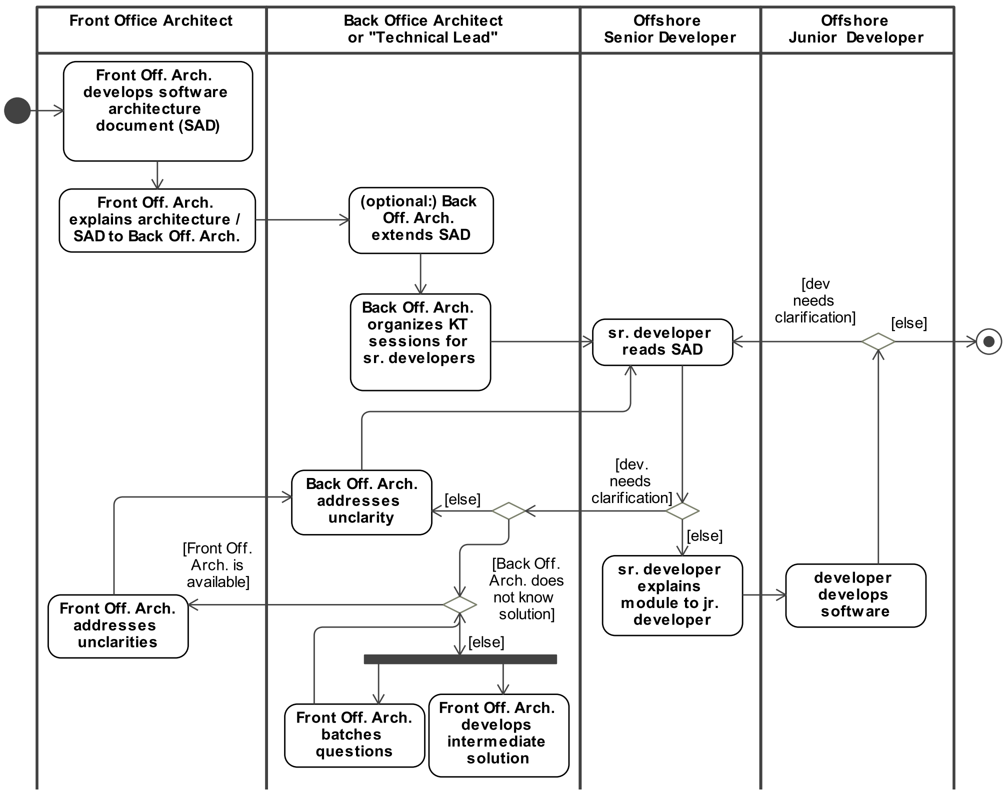

Front Office Architect
Back Office Architect or "Technical Lead"
Offshore Senior Developer
Offshore Junior Developer
Front Off. Arch. develops software architecture document (SAD)
Front Off. Arch. explains architecture / SAD to Back Off. Arch.
(optional:) Back Off. Arch. extends SAD
Back Off. Arch. organizes KT sessions for sr. developers
sr. developer reads SAD
[dev needs clarification]
[else]
Back Off. Arch. addresses unclarity
[else]
[dev. needs clarification]
[else]
[Front Off. Arch. is available]
[Back Off. Arch. does not know solution]
sr. developer explains module to jr. developer
developer develops software
Front Off. Arch. addresses unclarities
[else]
Front Off. Arch. batches questions
Front Off. Arch. develops intermediate solution

Figure 4.2: *Communication of the software architecture design from the onshore to the offshore location*

not enough." (BOA). The Dutch side of the project explained this as the incapability of the architect to create an architecture. The BOA and the offshore project leader explain that, *"the lack of knowledge of [a specific search engine] was also a problem as we had to design an architecture with a black box in the middle."* It is difficult to make out which team's narrative is right. In any case, with the shift of architecture responsibility, some of the tasks that were originally intended for the offshore team therewith moved to from India to the Netherlands. However, no extra budget was allocated to the FOA for these tasks.

4.5.3 Architecture Dissemination and Clarification Process

The process used to communicate the architecture from the FOA to the development team at the offshore location is outlined in the activity diagram in Figure 4.2.

Initial Architecture Communication Process

The architecture design was initially communicated from the FOA to the BOA and the development team by means of this SAD and two knowledge transfer sessions. The first of these sessions took place between the FOA and the BOA by means of a video connection, right after the delivery of the SAD. The BOA then held a knowledge transfer session with the development team, in knowledge transfer session, the BOA gave a high-level overview of the system. Subsequently, the development team obtained the SAD and then a second session took place with the BOA to discuss details and answers of developers and to explain how to proceed. The offshore project leader noted that the FOA was never involved in these sessions, he only talked to developers later, by means of video-communication sessions based on reviews of their code. A senior developer noted that, his previous projects were similar in terms of dissemination of architecture. A developer describes that, *"the architecture was discussed in a session, then we read through the documentation and came back for questions."* About this second meeting, another developer remarked that, *"in the SAD not all diagrams were written down using UML, that provided us with some difficulties but because we had [a] meeting with the FOA, he could explain to us what he meant [in those diagrams]."* A developer noted that, *"for most knowledge we needed, the knowledge transfer sessions we had were [...] enough."*

Feedback Process

The BOA works as a proxy between the FOA and the offshore developers. This situation was said to reduce the time that would otherwise be claimed of the FOA. However, the limited knowledge of the architecture of the BOA rendered him a message proxy. One developer would explain that, *"[our] team lead is too busy to properly communicate software architecture decisions — he has no time for coaching."* As a result, developers lack a deep understanding of the system's design. According to the offshore project leader, regular video meeting sessions directly between client and offshore team, took place. These sessions were not meant for the development team as the client was not seen as technically mature enough to talk directly to developers. A developer noted that, *"in my previous projects, we would have direct contact with our clients in the US — that is much better."* Codified knowledge does seem to play a major role in this project. One developer noted that, *"for a problem, I will first use the documentation and then talk to either the BOA or the FOA."* The FOA's perception of the team interaction on the offshore location is that, *"the technical team lead in India is less communicative with his team of developers than I am with him."* The FOA suspected that information got lost in the communication to developers through the BOA. As an explanation, the FOA suggested a lack of time for explanation or reviewing or a general lack of expertise on the part of the BOA. However, regarding the meetings that took place between the FOA and the BOA, the FOA claimed that *"[he] never said that he did not understand something."*

Requirements Maturity

A senior developer noted that, *"often, we worked more than 12 hours,"* and explained that, *"that had to do with unclarity regarding the functionality that the client requested."* As noted in Section 4.5.1, this project experienced problems with late changing requirements. The client was not the only source of late requirement changes as another example of late requirement changes resulted from poor requirement analysis. The BOA explained that:

> *"In the bid phase, one requirement pertaining to the reading of an input string was formatted as a simple one-line statement which later turned out to be a small project in itself — namely the construction of a parser."*

The requirement problems can partly be attributed to the level of technical maturity within the client organization. A developer described the client as, *"totally functional, not technical."*

4.5.4 The Software Architecture Document

All team members agree that the SAD is, *"very important,"* because, according to a developer, *"it is the base of the software and when it is in place, it is useful to a developer [. . .] with it, better software is made."*

The supplier organization provides an SAD template that contains extensive guidelines based on the RUP SAD template. Regarding this guideline, the FOA noted that, *"the SAD [of this project] was based on [this] template."* The FOA was critical of this template, though. He noted that he finds it allows too much freedom. While some sections of the SAD template had been filled out by the BOA, almost the entire SAD had been written by the FOA. The BOA explained that, *"there was little involvement from the client with the architecture."* According to the FOA, the software architecture template that is prescribed by his organization does not prescribe how to disseminate a software architecture design

In the next sections, we will discuss the purpose and the intended audience, the use and content of the SAD, perceptions on its quality and additional software architecture documentation used in this case.

Purpose and Intended Audience of the SAD

Regarding the purpose of the SAD, the FOA noted that, *"in essence,"* the client does not read the SAD. He added that the SAD audience is inherently a technical one and should have a basic level of technical knowledge to understand the SAD. Surprisingly, we found a rather extensive use of the Dutch language in certain UML models in the SAD. All use cases and the conceptual architecture models were written in Dutch. When confronted with this observation, the FOA explained that the use of Dutch was easier for client communication. The FOA noted that, *"translations are done in India"* and pointed out that, there exists a list with translations in the SAD. When interviewing the offshore development team, we did find this language issue to be a hindrance to the developers. A developer noted:

> *"another problem that we have is a language problem, for example in the SAD, parts were in Dutch, we use Google Translate to translate that text to English and if Google is wrong, that would be a problem."*

The development team did not seem to be aware of why Dutch was used in the SAD. One developer noted that, *"[the SAD] was in Dutch because the FOA was Dutch."* Answers regarding the choice of languages in the SAD were polite but hinted at frustration. A developer said, *"although the FOA speaks both English and Dutch, he used Dutch to create certain SAD diagrams[. Still,] it is not his job to translate Dutch to English."* The question who is responsible for translations remained unanswered. Another developer explained that, *"I would have preferred to have the documentation in English."* A senior developer noted that *"we cannot use Google Translate or a professional translator because of the technical nature of the project — we, however, [were forced to] use Google Translate."* The FOA said that, even if he was not pressured for time, he would still prefer to use Dutch for the parts of the SAD *"so not to confuse the client with different terminology."* The FOA noted that he is sure the BOA understood the SAD.

Use of the SAD

Developers said that they refer more often to the functional requirements described in use case documentation, than to the SAD. The available documentation is said to be a first point of reference for the developers who explicitly note they consult documentation before they ask the BOA or FOA. There seems to be a difference in the perception of when the SAD is used. For this particular project, the FOA noted that he did not expect the SAD to play a big role during the construction phase of the project. Yet architecturally significant components such as a locking mechanism, were added during the very late construction phases of the project. And the FOA explained earlier that, *"the SAD helped the developers to understand the [architecture layers], the different ideas in the presentation layer and the web services."* As a result, the offshore developers noted that the SAD played a major role during the construction phase of this project. For every new construction iteration, the requirements are gathered for another external connection of the system. The SAD is then needed to understand the flow of information to and from this new connection. Some developers explicitly noted: *"I even use the SAD later in the development stages."* Developers thus expected much more from the SAD than the FOA aimed to communicate with it. This is a mismatch in the perceived role of the SAD.

SAD Contents

Problems regarding the SAD that were mentioned include a remark by the offshore project leader who noted that he found lacking a clear description of *"the interaction with the external parties."* After analyzing the SAD, we observed that only a part of this model was visible. This implied that without having access to the digital source of the image, not all diagram elements in the component model could be seen. According to the FOA, this had to do with the manner in which the image was exported from the modeling tool, IBM Rational Enterprise Architect. As a solution, the FOA advised that the model source file could be opened with this tool. However, the development team lacked the licenses to use the modeling tool.

Most diagrams in the SAD were non-UML ("box-and-line" types of) diagrams. No legend was used for most of these diagrams. The offshore project leader noted that he preferred the use of UML in SADs because he found it to be *"a universal language."* Also, most developers explained that they preferred the use of UML in the SAD. One developer directly linked the limited use of UML in the SAD to project problems: *"In the SAD not all diagrams [were modeled using UML], that provided us with some difficulties but because we had two to three meetings with the FOA, he could explain to us what he meant with that."* Regarding the freedom the FOA took to develop the SAD we also found that certain protocols are described in more details than others. The FOA said that he choose himself which protocols prescribed in the SAD warranted further explanation. Regarding the choice for mentioning and explaining design patterns in the SAD, the FOA noted that he assumed that they were understood and that, *"some were particularly addressed."* An external architecture reviewer, however, noted that he found that the employed design decisions were not properly, if at all, described in the SAD: *"The design patterns in [this] SAD are not described in enough detail and seem to have been added just to show that some thought went into design patterns rather than to be used as a guide for developers."*

Perceptions of SAD Quality

Whether an SAD is good enough for a developer to use is influenced by what a developer is accustomed to. One developer noted that, *"compared to my other projects the documentation for [this project] was very good, everything was very well described in detail and that is needed when you need to build a low level design based on it."* Other developers noted that, *"this SAD is one of the better ones I have seen,"* and, *"the SAD in [this project] is better than the SADs I had to use in previous projects."* In contrast, a senior developer noted that, *"I have seen better SADs than the one we use in [this project]."*

One developer found that the incompleteness of the SAD led to a greater involvement of developers in the design of the architecture:

> *"Previous projects in which I was involved usually did not involve so many external components and were less complex, The SAD used in [this project] therefore is much*

> *better than the SAD in [my] previous projects because we were much involved in its creation and therefore we know how the architecture has evolved to its current form to understand the architecture much better."*

The desire for increased inclusiveness in software architecture design is a recurring theme in all three cases and will be discussed in more detail in the discussion section.

Additional Architecture Documentation

During the third construction iteration another design document, the *Supplementary Specification* (SS) was created to capture the late requirements and their impact upon the architecture. According to its definition, the SS artifact *"captures system requirements that are not readily captured in behavioral requirements artifacts such as use-case specifications"* (Kruchten, 2003b). The SS is normally prescribed to be used as input for the architectural analysis activity that leads to the SAD, in the inception phase. However, in this case, the SS is used as an update of the SAD. The SS contained information about logging, authentication, authorization and performance. The sources for the SS were the existing SAD and the Service Level Agreement (SLA) made for the project (which was to be delivered as a service). The FOA summarizes the rationale of the SS as follows: *"One of the goals of the SS is to prove to the client that we did certain things [because] in essence, they do not read the SAD."* A senior developer noted that, *"the supplementary specifications are created for the maintainers."*

4.5.5 Architecture Compliance

The non-functional requirements that an architecture addresses can only be guaranteed if an architecture is implemented according to plan. In this project, however, the FOA complained that, *"the developers think it is good enough when their work resembles the architecture,"* and explained that, *"as an architect for an offshore team, you have far less control over this."* A senior developer noted that, *"code should fully adhere to the architecture,"* and that even, *"when you find that some wrong decisions were made earlier and the architecture should then be readjusted,"* still, *"the architecture document should be leading."* When, during the interview, confronted with a choice between working code and architectural compliance, often developers would initially state that adherence to architecture is more important than working code. When pushed, developers would often admit that in practice, this plays out differently. Says one developer: *"this is somewhat diplomatic as when you find the architecture is not correct, you change the architecture and then align the code with this new architecture."* An architecture reviewer noted that an important reason for creating an SAD is to *"cover your ass."* In this sense, an SAD serves as an overview of non-functional requirements that where addressed that a project team can show to the client. In this case, not the code, but the documentation serves as a proof of work.

Architecture Understanding

Regarding developer knowledge of the SAD, the BOA explained that, *"knowledge was divided up per [external system] that was added — at least two people were knowledgeable per [system]."* This confirms findings by Curtis et al. (1988) who found that developers mainly understand "their" component. For example, when one developer was asked about certain parts of the architecture, he replied, *"I work with "the pushing scenario" so my knowledge is mostly confined to that functionality."*

Code Reviews

The FOA is seen by the offshore developers first and foremost as the person who knows how the system should work. To ensure that the system meets set requirements, developers must obtain information from the FOA. The FOA, in turn, needs to check the extent to which that information is understood. This check is done by means of code reviews (Figure 4.1, line δ). Code reviews are explained to be the only means of checking whether design decisions have been adhered to. Code reviews are prescribed by the development process to be first done by fellow developers, second by a senior developer (peer reviews), third by the BOA and fourth by the FOA. In this project the FOA did not have enough time to review all code. In total, he reviewed approximately half of all code. Due to resource constraints and tight schedules, the development team did not have enough time to perform code reviews. For example, no code reviews took place during the first construction iteration. Not all developers agreed that the code review process was adhered to strictly. Some developers referred to the *"incidental internal code reviews"* that would take place at the peer review level. A developer was dissatisfied with the code review procedures and noted:

> *"I would say that in this project the development team had less understanding of the architecture and of what to do than we had in my previous projects as there, a code review would be automatically done at check-in time, so we would immediately get feedback line-by-line, at that detail and those were mistakes you would not make again."*

Here, a connection between code review procedures and understanding of software architecture is made. Developers also imply that stricter code reviews would infer that their work or at least the quality thereof is important. Developers did not have a feeling that all their work was checked in detail. A developer noted, *"the FOA does not do code reviews on any scheduled time, when he had time he reviewed code and provides review comments more regarding architecture compliance than naming conventions."*

4.5.6 Shared Mental Model Deviations

We found the following differences in perceptions between team members in this project. First, we found differences regarding the software architecture process:

- Developers feel that they only interact with the FOA regarding software architecture matters. The FOA does not share this view: *"[more than half of my time I spend on] implementation related problems that are not concerned with the software architecture."*

- Some developers saw the BOA as the main source of information concerning software architecture while others saw the FOA in that role.

- There were great differences in how the development process was perceived. For example, some developers claimed that significant time losses occurred due to waiting for the onshore location while others claimed that, *"we never had to wait for something in particular from the front office."* This can be explained by the fact that not all developers were involved in discussions with the front office.

Second, we found differences regarding the artifacts created in the development process:

- Offshore developers found that they completely adhered to the design principles while the onshore team members did not share this view at all. (Developer: *"The code as it exists now is completely compliant to the SAD."* Another developer: *"currently the implementation is fully architecturally compliant."*)

- Some developers noted that the SAD was complete, some said it was not complete at all. This could be explained by the respective components that these developers "owned." One developer's component might have been more completely described in the SAD than another developer's component.

Third, we found differences regarding the project as a whole:

- Some developers remarked that the project was going to be delivered on time and that the system was almost done. The project was delivered 6 months too late and at the onshore location, the team members were much better aware that this would happen.

- The project is deemed to be a success by some offshore team members and *"very problematic"* by others. Location did not play an important role in the divergent opinions regarding this topic. The onshore team members consistently mention poor architectural compliance as an important reason for the project failure.

The FOA explained that every developer should at least have an understanding of the design patterns and the SAD chapter in which the "implementation view" (Kruchten, 1995) was outlined. This chapter contains information on design packages, the component model, process flows and architecture, naming and modeling guidelines. In each interview, we asked two questions pertaining to this content:

- *What are the most significant design decisions in the architecture?*

- *What are the most important parts of the SAD?*

Very different answers to this question were given:

- FOA: *"The most important design decisions in [this project] are the application of pushing and the use of chain of responsibilities [. . .] another important decision is the use of a manager so that the system is so flexible so that we can easily add a new connection."*
- Developer: *"The most important design decisions are the functional and non-functional requirements of the client and those decisions pertaining to the external systems that we are using and also the infrastructure."*
- Developer: *" The web service part was the most important part of the architecture."*
- Developer: *"Another important aspect of the architecture was that the flow of the system has to be able to be changed as easy as possible."*
- Developer: *"Other important aspects of the design are logging and exception handling because we have seven external connections."*
- Developer: *"The use of web services was another fundamental architectural design decision that allows us to not tightly couple the connections to the system to allow easy extendability"*

These answers always corresponded to the element of the system with which the developer was concerned. The FOA concluded that he, *"would give developer knowledge of the architecture a 6 (just enough) but there is a clear difference between the quality of individual developers."*

4.6 Case B

The objective of this project was to integrate various Human Resources (HR) systems into a single system. To this end, 70 use cases were implemented, 50 of which were electronic forms that were custom built in Microsoft InfoPath. The team members in this project were all different people from the team members for cases A and C. One exception is the offshore project leader who was briefly involved in the inception phase of case A. The client was a large, multinational industrial firm. Key characteristics of this case are summarized in Table 4.2.

4.6.1 Case-Specific Problems and Generalizability

Again, we first sketch an overview of the main problems that were encountered during the development life cycle of this case.

- This project dealt with changing requirements from the client. For example, many of the InfoPath forms that needed to be developed were already designed when the use cases changed a lot. Most of the form development then had to be redone. In addition, the system's architecture was impacted by these changing requirements.

- Significant delay was introduced due to difficulties related to deployment of the developed software. Various issues were explained to be the cause for these problems by different team members. The FOA noted that, *"the offshore team claimed that it worked in their environment,"* but that, *"they had little regard for the fact that we had to implement it in another environment."* A developer claimed that the cause for the problems was poor collaboration with the front office. He explained that, *"we cannot do deployment on our own as it takes place outside of our vision — we cannot visualize or imagine why things don't work in the production environment — what has been missed, why it is failing."* The end result was that the development team was not able to package their software in such a way that it could be delivered to the client. The front office therefore packaged and deployed the software.

- The development was severely hampered by three experienced developers leaving for other companies. The offshore project leader explained that SharePoint experience was a skill for which employers were willing to pay salaries that were 20 to 30 percent higher. Often changing employers (partly due to being offered such raises) is not uncommon in the competitive and fast-growing software development sector in India. According to the architect, when they knew they would leave, these developers were not very motivated anymore. As a result, they did not properly hand over their work. In addition, not enough information was codified. A senior developer explained: *"When the previous senior left for another company he gave the project a week notice, I came in two days before he left and on those two days he was very busy arranging his departure from the company so not much knowledge transfer happened in that time."* The FOA: *"this cost us a lot of expensive onshore resources."*

As with Case A, regarding the external validity of any insights we obtain from this case, we note that changing requirements and the problems associated with changing team composition in the offshore development team due to people switching companies, are common. The FOA noted that, *"the problems that we had in [this project] were typical."*

4.6.2 Architecture Development Process

As for the previous case, the intention for this project was to have the architecture created by the offshore team. The architecture responsibility was offered to the back office. Before the project commenced, however, the offshore organization made clear that they wanted the front office to develop the architecture. The reason seems to have been a lack of experienced architects available at the time. About the availability of developers that are able to create a proper software architecture in the back office, the architect was very clear: *"[Some people in our organization will have you believe that] if you just create a use case model, a set of non-functional requirements, some instructions about what it has to do, storyboards and some screenshots, [the back office] should figure out by themselves how to create sub-systems and decide how to build it — but we have found in past projects that that capacity is not available, also not with the seniors — who are good technical specialists but setting up an architecture is just not what they do."* In contrast, the offshore project leader made clear that, given availability, *"there is no reason why an Indian architect cannot talk to a Dutch client from India."*

There was little time available to create an architecture. The architect explained that, *"often clients have an idea of the solution that they want to have implemented and ask us to implement that solution — we then get little time to check such an architecture — this rarely leads to something good and often we need to rework the architecture during the implementation phase."* The offshore project leader expounded on her contradicting view by explaining that, *"I don't feel that the quality of the architecture suffers under grave budget constraints — perhaps the SAD suffers but I have seen good architecture frameworks made under pressure."* Not so for this architecture, though, as she continued to explain that, *"the architecture [that was] made [was] not very mature."* The initial lack of maturity of the architecture was confirmed by developers as they complained that they had to rework a large batch of InfoPath forms as the use cases and architecture were fundamentally changed during development. This problem could partly be attributed to the requirement changes described earlier.

In conclusion we observe that the architecture was not ready when it was communicated to the back office. Another mistake that was made here, was that all 50 use cases were developed immediately while limited knowledge was available regarding the target technology, InfoPath. The offshore project leader: *"In the first few weeks of a project, we [normally] go very slow deliberately — we don't aim for the sky — In [case A, for example, they] started ramping up very slowly — first [they] let developers create a small set of use cases so that [they] could assess the quality of the architecture but also the quality of the people."* Lastly, the offshore development team was not satisfied with the architecture design. A prominent critique of that design was that it was not robust enough for the client requirements. A senior developer explained that, *"SharePoint has a limitation on the amount of entries in a list — that should be no more than 10,000 — in a single month, the client overrun this by 5,000 — this should have been known by the architect."*

4.6.3 Architecture Dissemination and Clarification Process

The process of communicating the architecture from the architect to the development team in the back office is very similar to the activity diagram that described the same process for case A (Figure 4.2). One difference is that no back office architect was involved, but that the most senior developer took a similar role. The other difference is that the SAD was not extended by anyone in the back office. The knowledge transfer sessions took place by video conference and were only attended by the most senior developer — referred to as the "technical lead." The senior developer did not like the use of conferencing tools: *"communication on the architectural level should take place face to face because it is so fundamental."* The offshore project leader also complained about the architecture dissemination process and added that she did not know why the architect did not travel to India as she prefers that. The front office explained that the architect did not travel to India due to budgetary constraints. The offshore project lead responded by noting that that would have been an investment that would surely have paid off.

Developer Hierarchy

Similar to case A, the front office complained about the use of a single point of contact to the back office team. The architect complained that they, like in case A, had no say in the composition of the offshore team and that as a result, *"the developers were [. . .] not very visible to us from the front office — we got a point of contact offshore, but not a list of developers."* He continued to explain that, *"in bigger projects, Indian teams make use of a senior developer as a proxy and this does not work very well as the senior developer will understand you but [not] the junior developers."* One of the problems that resulted from this was that the front office had no influence in work distribution strategies. They noted that work was poorly distributed. The architect: *"A substantial part of [this project] consists of electronic forms which can thus be set up in a similar way - however, these forms were distributed by the senior in India and three different developers worked on these forms [and they were so dissimilar that] we could see by the implementation of the form, who made it."*

The use of a senior developers and a technical lead as proxies for the rest of the development team was also found in case A. A senior developer from this project explained that *"junior developers should not be given or even know the architecture because it is too much pressure and they will get confused — they are still learning and architecture is too much of a challenge for them — juniors should just implement whatever they are told too in the manner that is told to them."* He continued to explain that, *"As a senior developer you are a sort of translator between the architect and the junior developer."* He added that this hierarchical nature of work distribution is commonly found in software projects. The offshore project leader defended in a different way the use of a technical lead and senior developer as a proxy. She noted that, *"[while] it might be partly rooted in [our] culture; as long as a development team is bigger than three or four people, I don't see any other way than to ask questions from junior to senior developer to technical lead to front office in order to reduce the workload and to avoid double questions."* There might be more than one way to avoid double questions. Also, a question that is asked multiple times might serve as a bellwether for possible unclarities or complexities in the architecture design. Only when the more senior developers all left the project did the architect communicate to the developers directly. One developer noted that he liked this direct communication. The hierarchical nature of the team composition was still maintained, though, as even in this new situation, the most senior of the junior developers would note that *"the developers communicated directly the onshore architect but they would have less communication with him than I would have, because they would usually ask me a question first."*

Architecture Understanding

The clarification process, the steps that were taken when junior developers had questions, was similar to the description in Figure 4.2. A developer explained that, *"the SAD was provided to us and gone through with the technical lead — we then knew what was needed and did not use the SAD."* The front office noted that because the development team was not at all visible to them, they were not sure that they understood the architecture design.

We find that the offshore development team did not find understanding of the architecture and architectural compliance as important as the front office found it to be. The architect explained that, *"because of a lack of understanding in the Indian team, [I] was e-mailing whole chunks of self-written and self-found source code to outline what [we] meant — that is too much detail in my opinion."* He explained that this happened in all of the three offshored project in which he was involved as an architect. In contrast, a developer clearly outlined that he did not use the SAD as *"it was not so important for my own component."* This disregard for the relation that a "developer's functionality" has with the rest of the system, is a recurring theme in this case as well as the previous case. The architect referred to this problem as well: *"The mentality I always see with juniors in India is "I am here to execute a job, tell me what to build and I'll build it" — the idea that they also have to study the overarching structure is very rarely adopted."* As a result, the back office team admitted that only during the later stages in the project did they understand the software architecture.

The Notion of One Team

The notion of forming a single team, despite the geographical, temporal and socio-cultural distances is regarded as a best practice in literature (Lee et al., 2006) and in the case organization. The idea behind this notion is that communication technology is used to organize a team to simulate that it is co-located. In practice, however, it is not easy to accomplish "one team" as it mainly constitutes a feeling among team members — which is not straightforward to foster. For example, the architect explained that, *"we are supposed to be 'one team' but when problems arise such as a missed deadline, or who did not deliver what or are the specifications correct or hasn't something been tested well, and before you know it is one team versus the other."* He adds that, *"this feeling remains dominant in* *the rest of the project."* As noted in the previous section, the development team was not at all visible to the front office, which made creating a single team a challenge from the outset. A senior developer put across that, *"when the architecture is not developed offshore, it is not so clear what the architecture rationale is — you are just asked to implement, this model, this model and this model — if we knew how these models came to be, we can also contribute our ideas."* This is in concordance with remarks made by various back office team members in case A. Developers seem to find it important to be included in the architecture design process. They feel more like full partners than in a situation in which they are just told what to do. This contrasts with the front office view that the backoffice team simply needs to be told what to do to be able to function as team members. This follows for example, from this quote from the architect: *"The SAD does not contain many motivations for the design decisions that it describes [. . .] we most often do include these decisions but not so much for India, as they do not care much but more for the client."*

4.6.4 Software Architecture Document

In this project, as in Case A, limited time was available to create the SAD upfront. According to the architect, the SAD was not written with offshore development in mind. A GSD SAD, he notes, should have more explicit diagrams for clarity. However, Case B's SAD is "too immature" to allow for such distinctions. In hindsight, the level of detail of the SAD was too low. Initially, the front-office team offered to model up to the class-level and to make sequence diagrams. The architect said that the offshore senior developer *"laughed and said that was really not necessary."* Now, the architect notes, it turned out that we had better done that. At the end of the project, the SAD no longer reflected the implemented architecture — this phenomenon is referred to in literature as "architectural drift" (Rosik et al., 2010). The architect noted that he, at the time of the interview, still intended to (rudimentary) update the SAD.

Role of the SAD

The architect found that in GSD, it is important to make things very explicit. The SAD is an important means towards this end. The offshore project manager likened the SAD to be a *"semi-contractual document"* in relation to the client. Still, the architect is unsure about the role of the SAD in practice. He stresses that he thinks it is "very important" for a developer to understand the overarching structure of the architecture and the place of his own part of the system in that whole. He nevertheless adds that he observes that junior developers do not read the SAD. Apart from the hierarchical nature of the organization, he feels that an SAD might be too voluminous and complex for most developers in the offshore team. As a result, he notices, developers (at most) read the parts that are relevant to their component. From experiences in earlier projects, he notes that he found that visual materials, such as a clickable demo, work well. He adds that documents are just not read when they are too thick. He admits that the SAD might be more important in GSD but that it is a very limited vehicle for dissemination of design. The role of the SAD is less that of the primary conveyor of design decisions and more one of a reference document. From that perspective, the architect notes that, *"a super-well written SAD will also really help a lot [towards successful GSD projects, but] mostly as a reference."*

SAD Quality

The offshore project leader found the SAD to be incomplete and of low quality. She explains that the SAD is incomplete even though she feels that, *"not all aspects of the SAD as it is derived from the RUP, are needed to be filled out to be able to communicate it from onshore to offshore."* She finds the quality of SAD depends both on an architect's experience with architecture and his experience with collaboration with offshore teams. She adds that Dutch SADs suffer from language issues: *"The English in the SADs written by Dutch architects is often weird."* Also, offshore team members complained about the quality of the SAD: *"The SAD is not as good as SADs we used in other projects,"* and, *"it was not up to the mark."*

4.6.5 Architecture Compliance

According to the architect, the implemented architecture only resembled the prescribed architecture. He found that understanding of the prescribed software architecture was low among developers. He particularly found that developers did not test their own work, did not keep to coding standards and did little to ensure that their components work with other components. In line with these observations, one developer explicitly noted that he found architecture unimportant: *"The system was not complex enough to warrant a big role for architecture."* As a result, architecture compliance was low. A senior developer remarked that the implementation was *"fully compliant to the architecture."* Another developer said that the system was approximately 80 percent compliant to the prescribed architecture. The architect maintains that it was much lower.

Architecture Understanding

The architect finds that developers focus more on appearances than on "the inner workings." He adds that the idea that the overarching structure must be understood, is rarely adopted. He illustrated his point as follows:

> *"The offshore team is positioned far away on an island, and we send them a package of paper — the SAD. That creates an attitude: "What we have build may be interpreted as what you specified in that SAD, so we are done." [In my opinion, this does not stem from cultural differences at all.] The exact same attitude would*

> *exist if you would send a Dutch team to an island to collaborate with another onshore Dutch team."*

He added that according to him, the hierarchical nature of offshore development organizational culture did harm architectural compliance. Developers are, in his view, not encouraged to look beyond their own component. The architect: *"If I'd have to rate the knowledge that the offshore developers [in this project] had of the architecture I'd give them 6 out of 10."*

Code Review

Within the offshore team, the senior developers reviewed, as one explained, *"most of the code that the juniors write."* He explained that they specifically check whether, *"the layers that we want are used correctly."* Not enough time was available for full code reviews. Exceptions were made for very complex functions. In addition, a sample of the written code was reviewed by the onshore team. When mistakes were found, developers were told to repair similar problems for the entire code.

4.6.6 Shared Mental Model Deviations

When asked to describe the fundamental design decisions of the system architecture, the architect mentioned:

- The design of multi-language support
- Communication with external databases
- The interface with other external systems
- The translation of the client department workflow to the system

One developer noted that the most fundamental design decision were "long list policies." This was a design decisions specifically related to "his" components. A senior developer stated that, *"there are many important design decisions such as handling of the list, integrating InfoPath and integrating the company facebook."* Again, these components were that developer's "own components." I already discussed the discrepancies in the perceptions on the extent to which the implemented architecture complied to the prescribed architecture. The offshore team thought compliance was high to perfect while the onshore team thought the system was not compliant at all.

4.7 Case C

The objective of this project was to rebuild an existing Visual Basic 6 application in .Net. The size of the application was estimated to be approximately 800 function points.

Important characteristics of this case are summarized in Table 4.2. Compared to the cases A and B, this case had the smallest development team — only four offshore developers were involved. Where both cases A and B followed the RUP development process, case C used a distinctively agile approach. Another distinct aspect of this project is that a Model-Driven Development (MDD) approach was used to generate significant parts of the code.

4.7.1 Case-Specific Problems and Generalizability

The use of an existing MDD framework, implies that instead of one, multiple systems are being developed (Heijstek and Chaudron, 2010). The existing code generator needs the be developed in parallel to the system that is required by the client. One developer explained it by noting that, *"when we find a mistake while working with the framework, we fix it for the project, but also for the framework. This introduced extra effort."* In particular, at the start of the project, many things were changed to the framework, this led to many defects in the developed system and required extra effort to repair in both the generator and the meta-model of the system that was being built. In addition, not all developers were experienced with the MDD paradigm. As a result, at least one developer made changes to generated code. This code got overwritten after another generation cycle. He noted that he *"learned this the hard way."* This project had no problems with late requirements because, they say, of the weekly interaction with the client. Late requirements were expected and anticipated by using an agile design process (described in Section 4.7.2). This was possible in part because the complexity of this project was lower than that of cases A and B. GSD that employs MDD tools and techniques and that used agile development methods, is not very common. Growing bodies of literature exist on the application of agile methods in GSD projects (e.g. Taylor et al., 2006) and the integration of MDD in agile development approaches (Zhang and Patel, 2011). Studies on the application of MDD in a GSD (e.g. Heijstek and Chaudron, 2010, 2009) setting are rarer. Studies of the combination of all three paradigms are (to the best of our knowledge) unavailable in literature. The results obtained from analysis of this case are therefore hard to generalize and mainly serve to provide contrast to the more "traditional" cases A and B. This is a valuable addition to the analysis also because all three project were executed by the same organization.

4.7.2 Architecture Development

This project lacks a team member that was explicitly assigned the role of software architect. The project leader explains that there are relatively few complicated patterns in the system. In addition, the size of the project stems more from the quantity of the screens than from the complexity of the screens. The project had a start-up phase of three to four weeks in which scope was agreed upon with client and a high-level functionality document was made by using smart use case points. This document contains functional and non-function requirements. During these weeks, the non-functional requirements where reviewed to see whether it was possible to use the MDD approach they used earlier without having to change it fundamentally. They found this was the case.

Model-Driven Development Approach

The project makes use of a home-grown MDD platform that had been used on several earlier projects. This platform is based on Model-Driven Architecture (MDA) and therefore uses UML models as input. The class and use case models must adhere to strict modeling guidelines. By making use of stereotypes that are related to classes and use cases, patterns are used to generate code. The UML diagrams are translated to the XML Metadata Interchange format (XMI, Object Management Group, 2007) which is in turn used as input for the code generator.

In previous projects that used this MDD approach, approximately eight hours were needed to implement a single function point. Although little data is available in literature, this is at least a factor 1.5 to 2 faster than comparable traditional projects (which do not employ code generation techniques)[5]. The project leader, who also led these previous projects, explained that for this project, they were even faster as most people in both the on- and offshore teams were involved in the previous projects.

Applying MDD tools and techniques is limited to applications that can be described using the defined domain language. A trade-off needs to be made between flexibility and code generation coverage. The less "custom" functionality (that is, functionality that cannot be described using the domain language) is required, the higher the proportion of code that can be generated. A developer explains this by noting that, *"In an earlier project, lots of business rules were needed, many security constraints and lots of complications – [our framework] would not have been useful for such a project."*

Design Development Process

Implementation, design and testing is done in short iterations that last one week. In each iteration, a predetermined set of functionality is designed. In the next week, this functionality is implemented. The week thereafter, this functionality is tested.

[5]In the dataset accompanying ISBSG Benchmark Release 9 (International Software Benchmarking Standards Group, 2006), the average amount of (adjusted) function points per (normalized) person-hour for all 1001 "new development" projects was 15.3 h/fp. Jones (2000) reports the average productivity of 1065 projects (of which 505 are new and the rest are enhancement projects) in the US was 14.3 (h/fp). The smaller the data sets are, the higher the productivity seems to be become: Premraj et al. (2005) report that the productivity of 401 projects completed between 1997 and 2003 in Finland to be 4.3 h/fp. Tsunoda et al. (2009) report the average productivity of 211 enterprise application development projects in Japan to be 6.7 h/fp. However, the company average (for the organization from which all three cases in this chapter are derived), calculated from a dataset of 51 software development projects, is 20.5 h/fp. In an (unpublished) dataset in our possession containing 40 software development projects executed at another large Dutch organization, the average productivity is 27.7 h/fp.

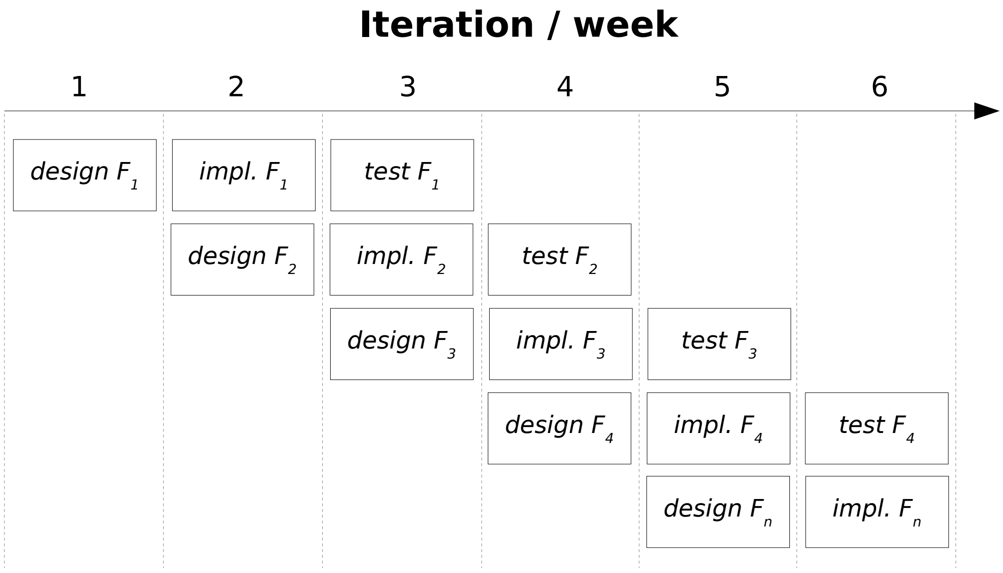


Figure 4.3: *Case 3 iteration strategy*

During every iteration (except the first two and the last two) one set of functionality is designed, another is implemented and yet another is tested. This strategy is depicted in Figure 4.3. This way, 17 iterations have taken place.

4.7.3 Architecture Dissemination and Clarification Process

Because of the use of MDD, the architecture used for this project is mostly defined by the design choices on the MDD framework level. The architecture is proven and mature as it was used for previous projects. The project leader therefore referred to the project as, "not such a high-risk project." Even though it is agile and that is because of the fact that the MDD approach is used, it has a proven architecture. In fact, this stable architecture is prerequisite to enable code generation.

Various interviewees involved in cases A and B were familiar with the MDD approach that was used by this project. The offshore project leader involved in case B explained that mature and reusable architectures such as the one used in case C, are easier for offshore development as it is clear how these need to be implemented.

Design Communication Process

Since a significant part of the architecture is already implemented in the framework that enables code generation, most design communication was related to lower-level design. The senior developer of the offshore team was involved in the weekly requirements gathering workshop by means of video conferencing. A design for implementation of these requirements would be first made onshore and then disseminated to the offshore team. The project leader explained that from his experience with offshore GSD projects, he noticed that, *"too much is being 'sent' from the onshore team to the offshore team and it is checked too little whether the offshore team has understood everything correctly."* As a result, in this project a method was used that the team called "continuous verification." The essence of the method is that short bursts of design information are sent offshore — mostly by means of video conferencing. The offshore team is then asked to summarize the design information sent and to report back to the onshore team. This way, a verification can take place whether the design information was sent and received as intended. A by-effect is that the onshore team members need to carefully phrase their design information and that the offshore team members need to listen well and ask more questions. For all the benefits that frequent communication offers in this case, at least one developer experienced similar waiting problems found in cases A and B. That developer explained that not all developers are usually part of all communication calls with the front-office. As a result, until there is time to address one of that developer's problems, he has to make assumptions and, *"wait and when my assumption turns out to be wrong, I have to rework my solution."* Still, the developer preferred the communication methods used in this project, over the methods he used in the previous projects in which he was involved.

Agile and Model-Driven Communication

The offshore project leader explained why he feels agile practices aid in communicating design:

> *"I contend that agile is a better way of approaching GSD projects as daily communication between the shores means that you can't play "hide and seek any more" — in addition, because of the short iteration cycles, you are being forced to know every single moment what is happening — when you need to give a demo to the client every week, you will make sure that it works. Nobody likes to look like a fool."*

The project leader has experience in non-agile GSD projects and explains that, *"the main disadvantage of these big RUP GSD projects is that you give offshore developers a chance to disappear underwater without knowing what they do and without feedback loops - effectively creating more than one team."* There is, however, a limit to the size of the projects you can tackle with such intensive communication required. The project leader noted that for this 800 function point particular project, they were "stretching it".

The offshore developers were very positive about the use of MDD in particular. One remarked that, *"we were only two to three months into the project and most of the important parts of the application were ready."* The same developer found it much easier to communicate using a model as opposed to talking about code with colleagues and the client. This MDD-related benefit is found in other studies as well (e.g. Heijstek and Chaudron, 2010). The frequent interaction with the client was also regarded as useful.

A developer explained that the increased client interaction led to a reduced total defect count. As a result, the development team could focus on new functionality.

Team Composition

The project leader explained that he found many GSD projects, especially those staffed with many skilled and experienced technicians, are unsuccessful. He explains that, *"most people in onshore development teams are selected based on their technical skills while these matter to a lesser extent in GSD projects than communication skills, being able to delegate, having trust in other team members."* He explains that he staffs his onshore team with young people who are relatively inexperienced and who are willing to learn. In addition, dedicated meetings are planned in which both onshore and offshore team members are required to review the development process.

4.7.4 Software Architecture Document

As most of the architecture was defined in the MDD framework, no SAD was used. The project leader planned to have an SAD made after system delivery, for the client. Instead of using a custom SAD, developers have to understand the architecture of the MDD platform. Most team members had experience with the framework. One developer explained that he learned how the MDD framework works during a three week "on-boarding training" that was organized by colleagues that were already knowledgeable regarding the framework. He noted that he was not a very good modeler at the start of the project but that he learned quickly on the job. He said that the use of MDD made the development process more formal and that, *"there are many things we have to do in a specific way."* He found that this made things simpler.

4.7.5 Architecture Compliance

As the project had no designated software architect, the project leader asked an external MDD expert to do a review of the code at several points during the implementation of the system. Compliance of handwritten additions to the code was found to be good.

4.7.6 Shared Mental Model Deviations

No explicit shared mental model deviation could be found during the interview. Some disagreement existed as to the percentage of code that was generated versus the percentage of hand-written code. One developer reckoned it to be 30 to 50 percent of the code where another claimed it to be 60 to 70 percent of the code.

4.8 Conclusions and Future Work

The software architecture design and coordination processes are important factors that determine GSD project success. We found that two of the cases were seen as unsuccessful projects. In case A, the onshore and offshore team have very different perspectives on how successful their project was. In both cases A and B, onshore team members consistently mention poor architectural compliance as an important reason for project slowdown. Case C's team application of code generation techniques automated much of the architecture compliance. Much fewer project problems were reported and the project was seen as successful by both onshore and offshore team members.

The rest of the conclusion is structured around the sub-research questions posed in Section 4.2.

4.8.1 How is Software Architecture Design and Dissemination Organized?

The main finding is that *specifically the dissemination of software architecture does not seem to be formalized while this might benefit the development process*. The SAD seems to be typically created onshore. A cost-related incentive exist to have software architecture created offshore. This failed in cases A and B. The main reason given was the novelty of software architecture as a discipline and the resulting lack of experienced architects at the offshore location. After the architecture is codified it is sent offshore to a senior software developer. This senior software developer then acts as a proxy for work distribution for more junior developers. If architecture-related questions arise, junior developers are expected to ask their senior peers. Developers continue working based on assumptions while they wait for an answer. This leads to re-work and project delay. In addition, this proxy work method obfuscates the intentions of the onshore team. Vice-versa, it "hides" the offshore development team for the onshore team. Direct interaction between all team members is seen as beneficial for architecture understanding. It is, however, hard to organize in larger teams. In case C we find evidence that in an MDD context, when architecture plays a smaller role (because it is (1) proven, (2) not complex and (3) mostly already implemented to enable code generation), less problems arise associated with architecture dissemination.

4.8.2 How Is Software Architecture Documentation Used?

The main findings are that (1) *the SAD is intended to be used extensively but developers use the SAD sparingly if at all* and that (2) *while a template is used for development of the SAD, architects enjoy (and use) a large degree of freedom in choosing software architecture representation*. The SAD has multiple audiences, chiefly the developers and the client. A first version of the SAD is typically for the developers. An updated version of the SAD is created so that it covers all client requirements. Perceptions on what or who is the main source for architecture information, differ widely. The onshore team find the SAD the main source of information while developers use the SAD to a limited extent. Some developers rated the SAD quality in cases A and B to be low. When used, the SAD does not seem to be specifically tailored for GSD.

4.8.3 What is the Role of the Architect in the Development Life Cycle ?

This question is harder to answer. The main finding seems to be that *the role of the architect in GSD is not clearly defined*. This role is therefore open to interpretation. From the previous question, it follows that according to the offshore team members, the architect is the main source of information regarding the architecture. In addition, the architect is also seen as a hurdle in terms of accepting developed source code. Architect travel was noted to be a best practice by most interviewees. However, in none of the cases, onshore team members traveled to the offshore location to coach the development team regarding the SAD.

4.8.4 How is Architecture Compliance Organized?

The main finding is that *while developers say that they find architecture very important, they seem to be mostly knowledgeable about "their own" component*. Code reviews are used to check architectural compliance. Developers and architects often disagree on whether code is compliant. The prescribed code review process is that (1) developers review each others code first, (2) that code goes to the offshore technical lead and then (3) to the onshore architect who should mainly look at architecture compliance. The onshore architect normally only does elaborate code reviews when there is a feeling that there is a problem. We find that architects do not find it part of their responsibilities to "micro manage" implementations to attain compliance. Interestingly, developers all note that they would like to be more involved in the design process at both the architectural and lower design level. Onshore team members experiment with offshore team member inclusiveness but are generally negative about this.

Chapter 5

A Theory of Dissemination and Coordination of Software Architecture Design in Global Software Development

In this chapter an analysis is presented of software architecture coordination and dissemination practices in the context of large scale, global software development. To this end, the case findings discussed in Chapter 4 are contrasted by means of a synthesis of the results of interviews with a group of experts. In addition, recommendations for software architecture documentation and dissemination are outlined.

5.1 Introduction and Objectives

In the previous chapter, we analyzed three specific industrial cases for dissemination of software architecture design in the context of offshore software development. In this chapter we outline and discuss the best practices distilled from these cases. In addition, this chapter presents a synthesis of the factors that underlie the phenomena that were observed in the previous chapter. In other words: we set out to explain *why* these best practices seem to work. The resulting grounded theory provides an answer to research question **RQ1** (Section 1.3).

We aim to improve the process of software architecture dissemination in the context of global software development and therefore pose the following research question:

How do the factors that shape how software architecture is disseminated and coordinated in large, industrial, custom, global software development projects relate?

To this end we use the results of the three case studies of large, industrial custom GSD projects presented in Chapter 4. In this chapter, we compare these case findings to a synthesis of a series of interviews with industrial experts in the fields of software architecture and GSD from various organizations. These were not the same people as those interviewed for the case studies presented in Chapter 4.

The outline of this chapter is as follows. Section 5.2 describes data collection and sections 5.3 and 5.4 outline the main factors and their main implications. Finally, sections 5.5 and 5.6 discuss best practices and conclusions and future work.

5.2 Data Collection

In addition to the case-related interviews that were reported on in Chapter 4, a group of 19 software architecture experts were interviewed at various organizations involved in offshore software development (GSD). These semi-structured interviews were all conducted on-site and face-to-face (in India). All participants were drawn from three major, international IT organizations. They were either senior developers, software architects or project managers. An overview of their function titles and experience is presented in Table 5.1. This particular group of respondents was not associated with the cases described in Chapter 4. The topics discussed during the interviews were similar to those discussed in the case-specific interviews. However, instead of concentrating on a particular case, the interviewees were specifically encouraged to reflect on their entire body of practical software development experience. The transcriptions of these interviews were used to reflect on the themes that were uncovered in the case analyses in Chapter 4. During the interview and the interview analysis, the concepts relating to case-specific phenomena (such as shared mental model measurements) were omitted as the focus of this chapter is relating concepts to factors beyond the case-level. We obtained our respondent population by means of introductions through Chain-Referral Sampling (CRS — also known as “snowball sampling”) (Heckathorn, 1997, 2002). CRS is normally used to obtain access to hard-to-find subjects in hidden populations (such as in studies regarding substance dependence as in e.g. Wang et al., 2005). While software professionals are not necessarily imperceivable in hidden populations, CRS helps to deal with some of the problems common to research in industrial software engineering practice. As software professionals, especially architects, tend to be expensive resources, their time for non-productive hours is limited. This problem is compounded by the fact that cost reductions are often part of the motivation to employ global software development practices and people are expected to deliver at short notice. The resulting work pressure at the offshore development location can therefore be relatively high. Being introduced by a potential interviewee’s peer increases the chances of getting a

Table 5.1: *Overview of expert interview respondent characteristics*

function type	function title	total experience (years)	years in role
developers	senior developer	7	
	senior developer	6	4
	senior developer	6	1
architects	solution architect	7	5
	enterprise architect	8	8
	enterprise architect	8	5
	senior technical architect	16	9
	solution architect		7
	technical architect	13	3
	senior technical architect	11	8
	senior technical architect		9
	lead architect	10	3
project managers	project manager	8	4
	project manager	11	2
	project manager	9	4
	project manager	12	8
	project manager	11	1
	project manager	9	4
	project manager	12	8
	average experience	*11*	

meeting accepted. In addition, in relationship-oriented cultures, it is easier to obtain interviews by means of a personal introduction. An excerpt of the relation between the respondents involved in this study is depicted in Figure 5.1.

5.3 Theory Building

In this section, we discuss the factors that play a role in the process of software architecture design, coordination and dissemination. Based on the concepts that were identified from the labels and their respective narrative as discussed in the three cases, we arrived at the grounded theory depicted in Figure 5.2. Three main drivers were discerned that (eventually) negatively influence project success in terms of schedule and budget overrun:

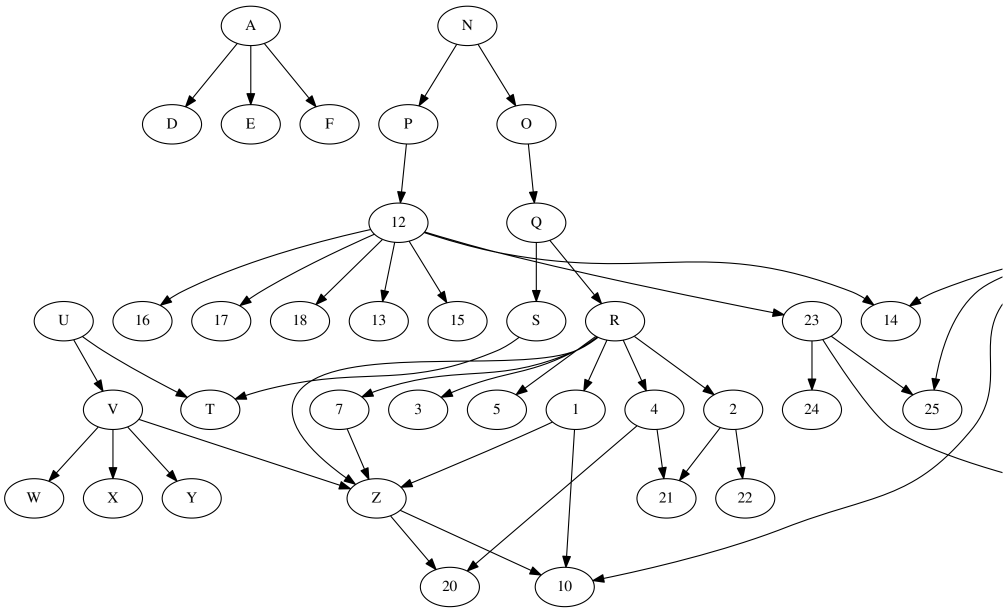


Figure 5.1: *Excerpt of (anonymized) chain-referral sampling graph of interview respondents*

1. The strong **implementation focus** of software development project management prematurely forces projects into the construction phase.

2. A **knowledge gap** exists between the onshore and offshore location regarding software architecture and its role during the software development life cycle.

3. **Cost reduction** forces a move of responsibilities towards the offshore software development location. This compounds the "knowledge gap" problems as less resources are available for knowledge improvement (training) and more work is required of less experienced team members. In addition, the added value of activities related to implementation is more tangible than that of design-related activities. As a result, the "'implementation focus" problem is aggravated.

In the following sections, each of these main drivers will be discussed.

5.3.1 Cost Reduction

Cost reduction is the most important driver for offshoring software development work (e.g. Šmite et al., 2010, Carmel and Agarwal, 2001, Ebert and De Neve, 2001). However, it remains to be seen whether cost reduction benefits materialize. Ample evidence suggests that GSD increases development cost (e.g. Conchúir et al., 2006, Herbsleb et al., 2001, Ebert et al., 2001, Espinosa and Carmel, 2003). Nevertheless,

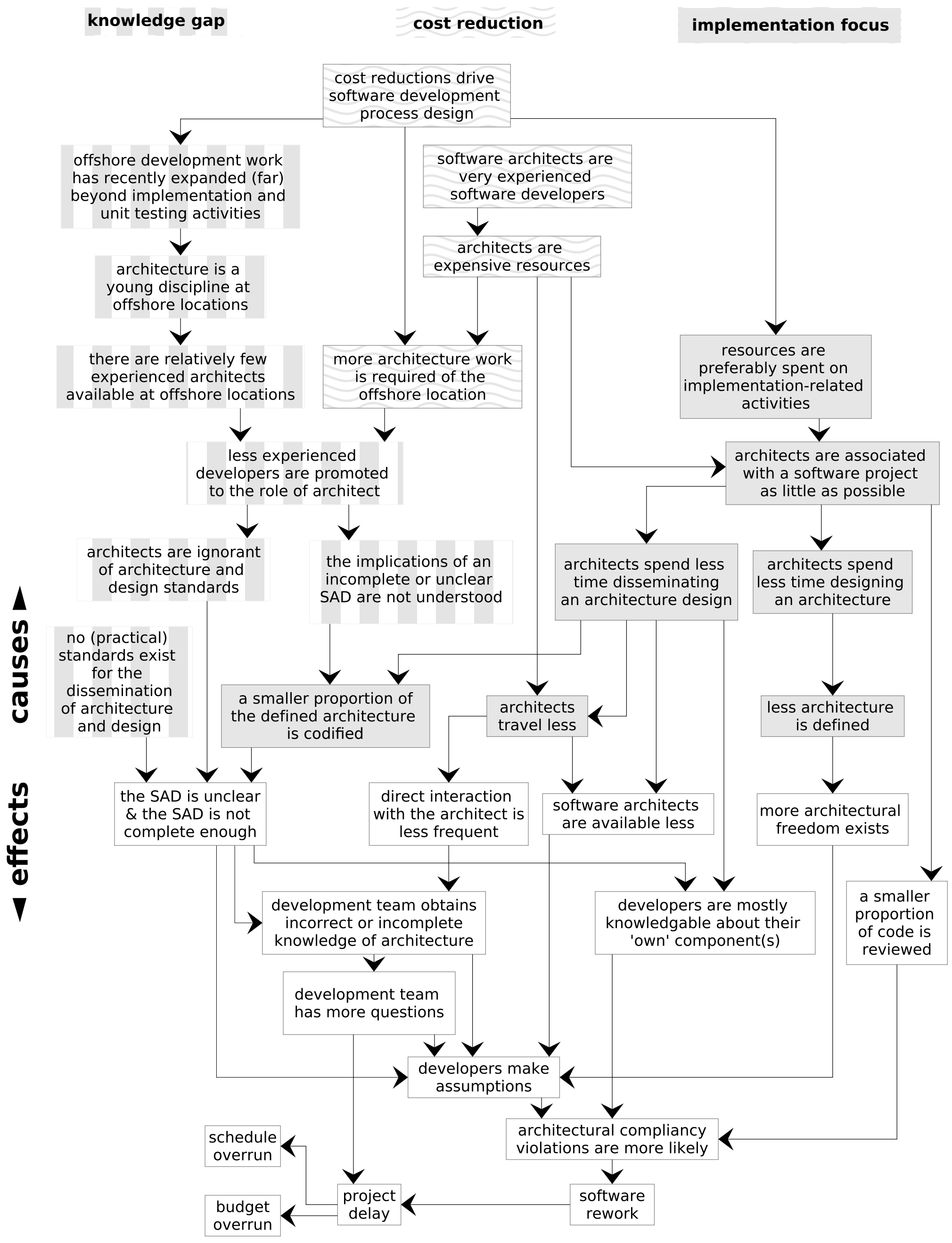

knowledge gap
cost reduction
implementation focus
cost reductions drive software development process design
offshore development work has recently expanded (far) beyond implementation and unit testing activities
software architects are very experienced software developers
architects are expensive resources
architecture is a young discipline at offshore locations
there are relatively few experienced architects available at offshore locations
more architecture work is required of the offshore location
resources are preferably spent on implementation-related activities
less experienced developers are promoted to the role of architect
architects are associated with a software project as little as possible
architects are ignorant of architecture and design standards
the implications of an incomplete or unclear SAD are not understood
architects spend less time disseminating an architecture design
architects spend less time designing an architecture
no (practical) standards exist for the dissemination of architecture and design
a smaller proportion of the defined architecture is codified
architects travel less
less architecture is defined
causes
effects
the SAD is unclear & the SAD is not complete enough
direct interaction with the architect is less frequent
software architects are available less
more architectural freedom exists
development team obtains incorrect or incomplete knowledge of architecture
developers are mostly knowledgable about their 'own' component(s)
a smaller proportion of code is reviewed
development team has more questions
developers make assumptions
architectural compliancy violations are more likely
schedule overrun
budget overrun
project delay
software rework

Figure 5.2: *Concept relation graph (⟶ denotes cause and effect)*

we find that the development organization from which cases A, B and C were derived, strives towards minimizing onshore project involvement. As software architects generally are experienced software developers, they tend to be expensive resources. Offshoring software architecture activities is therefore thought to potentially yield sizable cost reductions. The result of this insight is that more architecture work is required of the offshore location.

5.3.2 Knowledge Gap

In earlier projects, before the year 2005, only coding and unit testing work was entrusted to offshore development teams. The onshore location was responsible for requirements engineering, high and low-level design, integration testing and deployment. A recent development is that organizations require projects to offshore more of these "onshore activities" because of perceived potential development cost reductions. Gradually, unit testing, low-level component design and regression testing were added to the portfolio of activities that is commonly offshored.

Consequently, software architecture is a young discipline at this organization's offshore location in particular and at offshore development locations in general. There are therefore relatively few experienced architects available at offshore locations. To be able to meet demand, less experienced software developers are promoted to the role of software architect. An onshore architect typically has over ten years of software development experience. Offshore developers with just two years of experience have been promoted to the software architect role. As a result, offshore architects are aware of architecture design standards to a lesser extent.

5.3.3 Implementation Focus

Software development projects are limited by resources. Allocation of these limited resources over the various software development life cycle phases and activities is a well studied topic. Simulations are used to study the dynamics of resource allocation (e.g. Kellner et al., 1999) and theoretical models have been proposed to optimize resource allocation (e.g. Yiftachel et al., 2011).

Data from industry shows that resources are preferably spent on implementation related activities. Significantly more effort is spent in the construction phase of industrial projects than prescribed in theoretical models (Heijstek and Chaudron, 2007, Yang et al., 2008). Literature does not elaborate on the reasons for this phenomenon. We found a similar tendency towards coding-related activities at the expense of design activities in cases A, B and C. For example, the management tendency to push to move to the implementation phases as soon as possible (and limiting time spent on architecture design and dissemination) is referred to by various team members from cases A and B, as WHISCY or "**WH**y **IS** no-one **C**oding **Y**et?".

Cost reductions often limit an architect's association with a project to the architecture design (elaboration) phase. The focus on implementation shortens that design phase and thereby shortens even more the time an architect is involved with a software project. As a result, architects spend not only less time designing an architecture but also less time on disseminating that architecture design. The direct implications are that less architecture is defined and less architecture is codified. In addition, architects get less opportunity to travel to the offshore location.

5.4 Implications

We found the main problems caused by the drivers discussed in Section 5.3 to be:

- an *unclear and incomplete* software architecture document (knowledge transfer problem),
- that software architects are *available less* time per project (both a knowledge transfer and a control problem),
- that *less direct interaction* with the software architect is possible (both a knowledge transfer and a control problem),
- that *more architectural freedom* exists for developers and (control problem),
- that *less code is reviewed* (control problem).

These problems lead to incorrect and incomplete knowledge of the software architecture. Consequently, developers make assumptions and are mostly knowledgeable about their "own" components. Therefore, software architecture compliance violations are more likely to take place. This, finally, leads to software rework which in turn causes project delays in terms of schedule and budget overrun.

5.5 Recommendations and Best Practices

This section outlines a set of recommendations for coordination and dissemination of software architecture design in the context of GSD. The recommendations are structured as follows: First, the general recommendation are derived from the three main drivers that were discussed in Section 5.3. Second, a set of best practices is discussed that is distilled from the three cases analyzed in Chapter 4.

5.5.1 General Recommendations

First, *architecture should be recognized to be a first-class development concept*. All participants agree that architecture adherence is important. However, often developer knowledge of architecture is incorrect and incomplete and limited resources are allocated to checking architecture compliance during development. The added value of activities related to implementation is more tangible than that of design-related activities. This is an illusion. To adhere to non-functional requirements, extensive rework is often needed towards the end of a project. Therefore, making architecture a central concept (as is at the core of RUP) this type of rework can be largely prevented.

Second, *offshore software developers need to be trained to increase their understanding of software architecture*. The offshore shortage of software architects leads to limited understanding of the importance of software architecture.

Third, *a process needs to be in place for both initial dissemination of architecture design as well as the feedback process that follows it*. In the next section, we will describe some best practices regarding dissemination of architecture design as well as architecture implementation-related feedback. Such best practices should be institutionalized in a process so that common pitfalls may be avoided.

Fourth, *increased allocation of resources in (1) architecture design and representation and (2) guidance during the implementation phases from the onshore architect, is likely to lower budget overrun*. Conversely, we found, limiting onshore design time and architect availability results in budget overruns. Determining how many additional resources should be spent on software architecture design, representation and dissemination is not straightforward. As a guideline, in traditional (non-agile) development projects, between 70 and 80 percent of the architecture should be designed and represented when implementation commences. All architecturally significant use cases must be addressed in that initial architecture design.

5.5.2 Best Practices

Best practices are categorized in architecture design development, architecture design dissemination, the SAD, the architecture feedback process and architecture compliance.

Architecture Design Development

One strategy to ensure that the offshore team is more knowledgeable regarding software architecture design is to write the SAD together with the offshore technical team lead. For both cases A and B, in hindsight, the architects found that the SAD was not mature enough to be transferred to the offshore team. For a typical custom software development project, the majority of the architecture should therefore be stable before the construction phase. To this end, the “biggest mistakes” have to be removed in the elaboration phase. At the very least a POC of the architecture should be made during the inception phase. Investing some extra time into maturing the architecture and creating a more detailed SAD pays off during the construction phase in terms of fewer comments required during code reviews. This also amounts to fewer defects and less required rework.

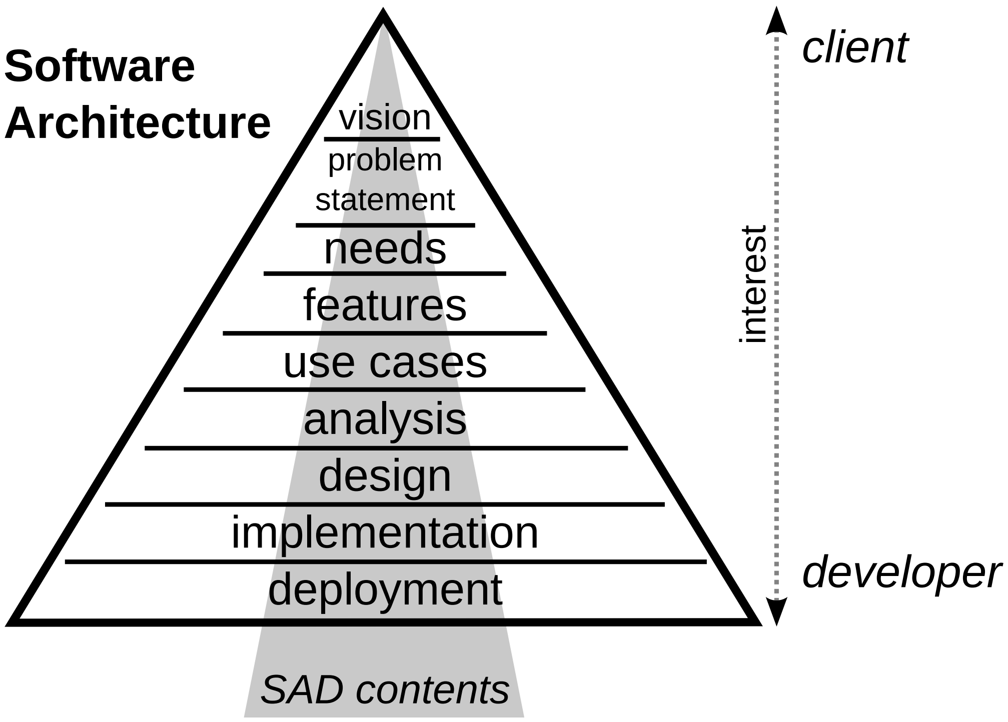


Figure 5.3: *Contents of an SAD and associated stakeholder interest*

Various parts of the same SAD address different audiences (Figure 5.3). A more technical stakeholder is generally interested in the bottom levels of the pyramid whereas a client would be more interested in the top layers. Apart from merely mentioning the information related to each layer, the key to a high quality SAD is the traceability between the layers.

Architecture Design Dissemination

The majority of respondents from all three cases agreed that every onshore architect should visit the offshore team at least once, to transfer in person the SAD and the explain its contents to the offshore development team. One developer explained that it is important for the architect to explain in person the SAD to a development team so that all developers, *"understand the dept or importance of specific requirements — this is something you cannot do from a document alone and it prevents that the solution moves slightly in a different way."* A slightly cheaper solution is to have at least one senior developer of the offshore team to travel to the onshore location during the initial software architecture development phase.

Successful dissemination of architecture requires that multiple communication channels are used to disseminate a single message. In addition to the SAD, the use of personal video conferencing sessions, telephone, e-mail and synchronous messaging should be used to repeat that same message. The frequency of use of these tools should be high. The objective for an architect is to give a lot of guidance after the SAD has been introduced so that all principles are properly understood.

Case C (Chapter 4) used a method which they called "continuous verification." The essence of the method is that short bursts of design information are sent offshore — mostly by means of video conferencing. The offshore team is then asked to summarize the design information sent and to report back to the onshore team. This way, a verification can take place if the design information was sent and received as intended. A by-effect is that the onshore team members need to carefully phrase their design information and that the offshore team members need to listen well and ask critical questions about what is unclear to them.

The iteration approach used in Case C requires a lot of interaction between team members in general and the onshore and offshore locations in particular. However, this strategy requires a client that is able and willing to have (very) frequent meetings throughout the project.

The SAD

As elaborated on earlier, SADs needs to both be clearer and more detailed than they were, found in e.g. cases A and B, because less contact between the development team and the architect takes place in a GSD context. However, an SAD that is too thick is less likely to be read in its entirety. The use of a common template for architecture representation is seen a beneficial. It requires architects to codify information they might otherwise have omitted. The use of UML is not seen by architects as essential for clear architecture representation. Architects argue that one, *"should use what gives most clarity."* However, that architects seem to take intended audience(s) into limited account and that offshore developers do seem to share a preference for UML. It might therefore be beneficial to take the use of UML for architecture representation into consideration. At least a legend should be used to avoid unclarity about the meaning of specific elements used in box-and-line diagrams.

Architecture information should be codified before and during the construction phase. For project-based software development, organizations, budget quickly "disappears" after a system is delivered. In addition, significant architectural drift (Rosik et al., 2010) takes place during the construction phase. Moreover, as employee turnover is relatively high (particularly in Indian development organizations), team members and their knowledge regularly disappear from a project. Therefore documentation in general and the SAD in particular should be kept up-to-date during the project.

Architecture Feedback Process

Almost all developers in cases A an B complained about delays due to the unavailability of software architecture information. Somebody with intimate knowledge of the software architecture should therefore always be available albeit not necessarily physically. This is difficult to explain to project leaders, given that cost reductions are a strong driver for GSD and that software architects are expensive resources due to their seniority. In addition to the trip that an architect should make to explain the SAD, the architect is recommended by some respondents to again travel to the offshore location during the first code reviews session. About this trip, the offshore project leader from Case A explained that, *"this provides a motivation for the developers — it tells them that somebody cares for them, sits besides them and helps them resolve issues."* To create a feeling of working in a single team, a method was used in Case C to make video conferencing sessions more like actual conversations — such as the ones at a "local coffee machine." Team members made it a habit to not get directly to business at the start of such a session: *"Each video conference we spend about ten minutes talking about private things."* In addition he notes that it is important to enrich the conversation with information that is contextual such as *"funny clothing that someone was wearing or [perhaps] spend some time talking about a mistake you made."* The goal of these habits is to build up a relationship like you would with local team members.

Architecture Compliance

Less complex architectures are less difficult to disseminate and are more easily adhered to. For architectures implemented in a GSD setting it is therefore even more pertinent to limit component coupling and increase component cohesion. For very complex architectures, the cost reductions that GSD potentially offers are likely to be offset by the time it costs to represent and disseminate architecture design and attain architectural compliance.

5.6 Conclusions and Future Work

Knowledge codification has the potential to mitigate some of the problems that are the result of the distances that GSD introduces. The opportunity to develop quality documentation, however, is limited. Cost reductions, a focus on implementation-related activities and a general lack of knowledge about software architecture lead to poor architecture design, coordination and dissemination. The lack of knowledge regarding software architecture is a genuine problem in the sense that it is a hurdle to be overcome if GSD projects are to make architecture a central concern. However, cost reductions at the expense of architecture design, coordination and dissemination as well as the tendency to give priority to implementation-related activities, provide a stark contrast with the claim made by all respondents that software architecture is such an important aspect of software development.

Chapter 6

Experimental Analysis of Textual and Graphical Representations for Software Architecture Design

In this chapter the results of a study on the use of software architecture documentation is described. First, the effectiveness of text-dominant versus diagram-dominant architecture descriptions are explored by means of an experiment. Second, developer characteristics that benefit architecture representation understanding are investigated.

This chapter is based on the following publication:

Werner Heijstek, Thomas Kühne and Michel R.V. Chaudron **Experimental Analysis of Textual and Graphical Representations for Software Architecture Design.** *In Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM 2011) pages 167–176, Banff, Alberta, Canada*

6.1 Introduction

Software architecture documentation facilitates stakeholder communication and is instrumental in ensuring that essential design principles are adhered to by the source code. Software architecture documentation *"captures and preserves designer intentions about system structure, thereby providing a defense against design decay as a system ages, and* *it is the key to achieving intellectual control over the enormous complexity of a sophisticated system"* (Hofmeister, 2000). However, software architecture and design knowledge management is challenging in the context of GSD (Ali et al., 2010). In such a scenario, complete and unambiguous architecture design documentation constitutes an indispensable complement to informal communication (Curtis et al., 1988, Lee et al., 2006).

However also in co-located development, using an iterative development process, reliable and effective documentation is highly desirable. While co-located teams better support spontaneous and informal communication, there is a danger that code is not implemented according to the principles as laid out in documentation artifacts. If the design documentation is not treated as the ultimate reference and does not succeed in allowing answers to common questions to be derived, there is a danger that the resulting system will be based on inconsistent interpretations and assumptions: *"The best architecture is worthless if the code doesn't follow it"* (Clements and Shaw, 2009). In their landmark study on the state of the practice in software architecture research, Shaw and Clements (2006) formulate some promising areas in which significant opportunities exist for new contributions in software architecture research. Among others, they discuss the need to find the right language to represent architectures and finding ways to assure conformance between architecture and code. Documentation of the architecture is also essential for maintenance activities which typically involve different engineers from the ones who originally developed the system.

In this chapter we report on an experiment we conducted on media type effectiveness for documenting software architecture designs. The outline of this chapter is as follows. Section 7.3 contains an overview of related work. The study objective is outlined in Section 6.3. In Section 6.4, the experimental design is explained and Section 6.5 contains an overview and discussion of the results. The threats to validity are discussed in Section 6.6. Finally, recommendations are given in Section 6.7 and Section 6.8 describes our conclusions and future work.

6.2 Related Work

The related work for this study spans different sub-fields of software engineering. In the following paragraphs we will discuss related work on software architecture representation in practice, quality of documentation, the use of UML for architectural representations, the use of design documentation, multimedia learning and related experimental analysis of software design representations.

6.2.1 Software Architecture Representation in Practice

In their survey of 11 industrial systems Soni et al. (1995) found that a combination of informal and semi-formal techniques was used to describe software architectures.

They found that informal diagrams, tables and natural language text with naming conventions are used to describe many of the software structures not described in functional decomposition diagrams. They note that, "even when a formal notation is used, it is often supplemented with informal and incomplete diagrams, in order to enhance the understanding of the formal model." Soni et al. did not find this surprising as rigorous architecture description techniques were not yet available at the time. However, it is common that the software architecture description of systems is informal and based on "boxes and lines" types of notation (Abowd et al., 1995, Soni et al., 1995). A recent study of 57 industrial software architecture documents (Heijstek and Chaudron, 2011) confirms that software architecture is still described using a variety of media without an apparent systematic approach to media usage. The limitations of this style of representation led to the taxonomic separation between software design and software architecture (Eden and Kazman, 2003). It also led to the development of several methods and frameworks for defining and representing architectures (e.g. Bachmann et al., 2000, IEEE, 2000, Bachmann et al., 2000, Clements et al., 2002, Jansen and Bosch, 2005, Taylor et al., 2009). It is, however, unknown which of these styles are more effective to allow developers to correctly understand the intended architecture. As a result, little is known about how to produce more effective documentation. Bengtsson and Bosch (1999) describe an industrial case in which they were involved in designing the architecture. They note that they, *"found it hard to capture the essence of the architecture."* They also note that only because of the co-located nature of the project, they were able to *"overcome the problems with the written documentation."*

Agile methods (Highsmith and Fowler, 2001) recommend to make "lean" documentation, suggesting that documentation should only include information that is used. But even such code-centric, light-weight methodologies employ a form of architectural documentation (Smith, 2001). Agile methods have been introduced in more rigorous methodologies such as RUP (Hirsch, 2002, Pollice, 2001), partly in an attempt to incorporate increased formality regarding documentation. Developers do seem to prefer less documentation. This is supported by the findings of Forward and Lethbridge (2002) and Lethbridge et al. (2003) who studied the use and usefulness of documentation. The authors find a preference for simple and powerful documentation and conclude that documentation is an important tool for communication, even if it is not up to date. A recent survey by Stettina and Heijstek (2011b) of 79 agile software development professionals in 8 teams in 13 different countries, found that the majority of agile developers find documentation important or even very important, but also that too little documentation is available in their projects.

6.2.2 Use of UML for Architectural Representations

Many current architecture description methods recommend the use of UML diagrams for representing a software architecture. Hofmeister et al. (1999) present results of their action research study into using UML for representation of a system's architecture. They found that UML worked well for describing important aspects typically described in software architecture documentation (such as the static structure of the architecture) and not so well for constructs (such as protocols and a general sequence of activities). A more recent study of the suitability of using UML to model software architectures (Medvidovic et al., 2002) reports that, *"UML lacks direct support for modeling and exploiting architectural styles, explicit software connectors, and local and global architectural constraints."*

There is certainly no standard way of creating architectural diagrams with the UML. UML allows its users a large degree of freedom (Nugroho and Chaudron, 2008). Systems are generally modeled incompletely and varying levels of detail are applied (Lange, 2006). UML standards are often applied loosely (Lange et al., 2003). All these aspects have been found to negatively contribute to the quality of software (Nugroho and Chaudron, 2009).

Results from an experiment by Tilley and Huang (2003) suggest that the UML's efficacy in support of program understanding is limited by factors such as ill-defined syntax and semantics, spatial layout, and domain knowledge. This chapter contributes to the understanding whether UML diagrams fulfill the expectation to represent precise and effective architecture documentation.

6.2.3 Use of Design Documentation

An observational study by Dekel and Herbsleb (2007) found that software teams improvised representations, incurring orientation difficulties and leading to an increased reliance on memory. Documented designs were therefore not useful without additional contextual information. We have to assume that this might be true to some extent for documented design decisions in practice. Design decisions are often the result of one-on-one meetings (LaToza et al., 2006). A study involving interviews and a survey by Cherubini et al. (2007) found that many modeled design decisions are lost. Studies that find that developers avoid using design documents when possible (Herbsleb and Moitra, 2001, LaToza et al., 2006, Kraut and Streeter, 1995, Müller and Tichy, 2001) may be construed as an indication that we know little about how to produce the best possible software architecture design documentation. In their study of how software developers use diagrams in software documentation Hungerford et al. (2004) found that search patterns that rapidly switched between two different diagrammatic representations are most effective. They note that, *"these findings support the cognitive theory thesis that how an individual processes information impacts processing success."* Empirical studies show that developers mostly understand only "their" specific components of the application (Curtis et al., 1988). Holt (2002) even advocates a "law of maximal ignorance" which he summarizes as follows: *"Don't learn more [about an architecture] than you need to get the job done."* Due to time pressure, Holt notes, developers barely have enough time to get acquainted with a system. Scanniello et al. (2010) experimentally evaluated the use of design documentation that outlines design patterns on maintenance activities performed on source code. They found that the effort and efficiency significantly improved when design pattern were properly documented and provided to the subjects. In their mixed-method study of software engineers in practice, Lethbridge et al. (2003) found that software documentation is frequently out of date, too voluminous, poorly written and unfathomable and that documentation processes ("much mandated documentation") can be inefficient and ineffective. They find that, "[a] considerable fraction of documentation is untrustworthy" but also that *"[A]rchitecture and other abstract documentation information is often valid or at least provides historical guidance that can be useful for maintainers."* More than 40 percent of respondents note that they find that software architecture documentation is rarely, if at all, updated after changes have been made to a software system. Most respondents agreed with the statement: "Documentation is always outdated relative to the current state of a software system." Lethbridge et al. conclude that we need to better understand the various roles of software documentation and more closely match our prescribed processes to fit those roles. We specifically focus on the role of architecture documentation to support implementation work. Observational studies show that developers use documentation as little as 3 percent of their time (Lethbridge et al., 2003). More recently, Stettina and Heijstek (2011b) studied the use of documentation in Agile teams and found that the majority of developers found documentation important to very important and that they also found that too little documentation was available. In addition, they found it difficult to locate this (internal) documentation.

6.2.4 Experimental Analysis of Software Design Representations

Various experiments have been conducted to investigate the efficacy of software design representations. These experiments concentrated on measurement of the strength of a particular representation type to convey certain design properties under certain circumstances. For example, Lange and Chaudron (2006) used an experiment to investigate how developers deal with inconsistencies in UML diagrams. They found that defects often remain undetected and cause misinterpretations. Gemino and Wand (2003) advocate the use of evaluating modeling techniques based on models of learning based on Mayer's cognitive theory of multimedia learning (Mayer, 2009). In a later study, Gemino and Wand (2005) compared two different visualizations of Entity-Relation Diagram (ERD) types (Chen, 1976). They found that, *"clarity within [a] model may be more important than the apparent complexity of [that] model when a model is used for developing domain understanding."* In this study, we compare textual and diagrammatic models as this combination is found to be most commonly used in industrial practice. Another relevant related experiment is the comparison of comprehensibility of UML class diagrams versus ERD in the context of comprehension, maintenance and verification by De Lucia et al. (2010). They found that using UML class diagrams, subjects scored higher on comprehension. In this experiment, we use UML diagrams.

Knodel et al. (2008) conducted an experiment to study the role of graphical elements in architecture representations. They found that specific visualizations, such as neighbor highlighting, and an information overlay panel had a significant impact on developer comprehension. In our experiment, we did not use an interactive architecture visualization tool and could therefore not incorporate their findings to increase the efficacy of our experimental material. Formal notations (such as the Object Constraint Language (OCL, Warmer and Kleppe, 1998)) can also be used to make software architecture documentation more unequivocal. In fact, Briand et al. (2005) found evidence that if developers are well trained OCL might be a more effective annotation for UML models than text. OCL is sparsely used in industrial practice.

6.3 Objectives

In this chapter, we address **RQ2** (Section 1.3). This question aims to find representation methods for software architecture design that are understood by developers in the context of GSD. Tilley (2009) concludes an overview of "findings and lessons learned related to documenting software systems with views from numerous projects spanning 15 years of research and practice" by noting: *"The question of when graphical documentation is more effective than other forms of documentation (e.g. textual), and for which types of users, remains open."* We describe our main objective according to the goal definition template provided in the GQM paradigm (Basili et al., 1994):

> We **ANALYZE** the effectiveness of diagrams and text for representing software architecture designs **FOR THE PURPOSE OF** improving the quality of architecture documentation **FROM THE PERSPECTIVE OF** communication between software architects and developers **IN THE CONTEXT OF** project-based custom software development.

We therefore pose the following research questions:

1. *Are diagrams using a visual notation better suited to communicate software architecture design than textual representations?*

2. *How do software developers comprehend software architecture representations?*

3. *How do developers deal with missing and conflicting information in software architecture representations?*

4. *To what extent do developers make assumptions or fill in gaps in software architecture representations?*

Diagrams are widely used during designing and it appears plausible that their visual representation is an effective medium to communicate software design. Diagrams contain the essential information in an easy to overview, easy to process, two-dimensional arrangement. Albers (2004) asserts that diagrams should allow developers to find answers quickly because diagrams contain less noise. In particular architectural principles pertaining to the topology of an architecture should be obvious in a diagram whereas they will have to be inferred from text. Moreover, diagrams should transcendent socio-cultural and language idioms because they are largely independent from natural language. For all the aforementioned reasons it can be expected that developers may have a preference for diagrammatic as opposed to textual documents. Yet, to the best of our knowledge no empirical study has ever attempted to confirm or refute these assumptions. In this study, we focus on natural language text because it is most commonly used in industrial practice. The experiment was designed to test the following hypotheses:

H1$_1$ *Diagrams are better than text at conveying software design to software developers.*

H2$_1$ *Diagrams are better than text at conveying topology-related design information to software developers.*

6.4 Experimental Design

Potential methods for testing our hypotheses include user surveys and controlled experiments. We dismissed the first because it generally only applies when more is known with regard to the variables that need to be controlled. Moreover, there is an element of uncertainty as to whether users correctly evaluate which media type is more effective (e.g. due to subjectivity). An "in the field" study, observing software architects while they are performing actual work might yield reliable data. However, it is difficult to get access to companies for such an in-depth analysis.

Hence, we designed an experiment in which we measure media effectiveness by varying the media dominance during a series of design document presentations. We define media effectiveness as the extent to which a medium can convey the modeled design information it contains in such a way that a developer understands it correctly. Therefore, we measured media effectiveness on the basis of how well participants were able to extract the intended design information from two documents that described one architecture design. One document contained text and the other diagrams. The variables under study are summarized in Table 6.1.

During the experiment, we used three methods to collect data:

1. We used two questionnaires to obtain participant-specific information.

2. We filmed participants during a set of tasks to understand when and from which medium they obtained answers to our questions.

Table 6.1: *Experiment Variables*

Question Characteristics		*variable type*	*values*	*source*
1.	QUESTION NATURE	nominal (binary)	$\in$ {topological, non-topological}	experimental design
2.	MEDIA DOMINANCE	nominal (binary)	$\in$ {diagram, text}	experimental design
3.	ANSWER LOCATION	nominal (quaternary)	$\in$ {diagram, text, both, neither}	experimental design
Developer Characteristics				
4.	LINGUISTIC DISTANCE	ratio (continuous)	$\frac{1}{\text{language score}}$ (see Table 6.6)	preliminary questionnaire
5.	EXPERIENCE	ordinal (quaternary)	$\in \{1,2,3,4\}$ (see Par. 6.5.7)	preliminary questionnaire
6.	MODELING SKILL	interval (discrete)	$\in \{1,\dots,7\}$	preliminary questionnaire
Developer Performance				
7.	MEDIA PREFERENCE	nominal (binary)	$\frac{1}{n}\sum_{i=1}^{n} q_i\_\text{percentage\_diagram}$ (q for question)	analysis of video
8.	MEDIA SWITCHES	ratio (continuous)	$\frac{1}{n}\sum_{i=1}^{n} q_i\_\text{switches}$	analysis of video
9.	TIME PER QUESTION	ratio (continuous)	$\frac{1}{n}\sum_{i=1}^{n} q_i\_\text{total\_time}$	analysis of video
10.	USED ONE MEDIUM	ratio (discrete)	$\sum_{i=1}^{n} q_i\_\text{only\_used\_one\_medium}$ (true if $q_i\_\text{perc\_diagram} = 0 \vee 100$)	analysis of video
11.	ANSWERS CORRECT	ratio (discrete)	$\in \{0,\dots,13\}$	analysis of video
12.	FELL FOR FALSE FRIENDS	ratio (discrete)	$\in \{0,1,2\}$	analysis of video
Developer Opinion				
13.	PERCEIVED EFFECTIVENESS OF MEDIA	interval (discrete)	$\in \{-6,\dots,6\}$ perc_eff_diagram − perc_eff_text (both interval $\in \{1,\dots,7\}$)	post-experimental questionnaire

3. We requested the participants to think out loud while answering questions about the system.

Filming the participants had several benefits. First, we obtained a wealth of information regarding participant behavior during the experiment. We could, for instance, determine which media type was consulted first, last, with what frequency and how often participants switched between media types. The recordings are likely to prove useful in the future for extracting new data from the data set according to different research questions or the measurement of other, newly identified variables.

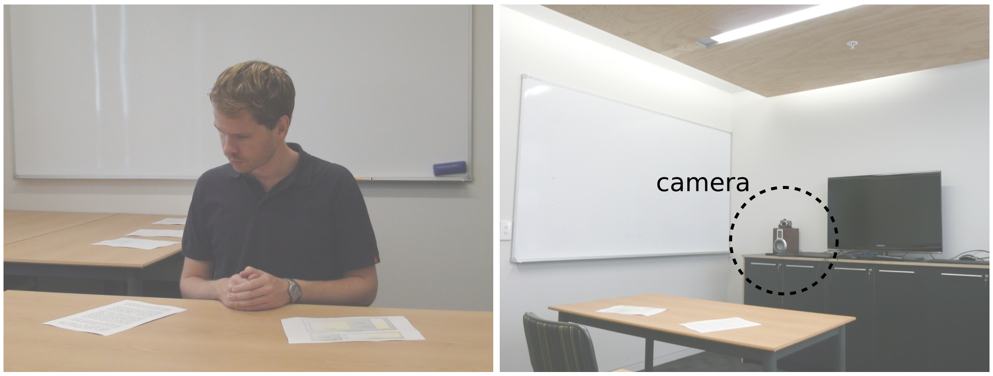
We documented our experiment design with an experiment protocol in which the experiment environment, process and measurement methods are outlined in detail. This documentation enabled consistency for running the experiment at different locations and will likewise facilitate future replication. For the diagrammatic representation of architectures we chose to use UML, which is the de facto standard for representing software designs.

6.4.1 Experiment Planning

The study was executed between February and August 2010. After an initial literature study, a general study design was created. We then initiated the human ethics approval process and started with the design of materials, questions and the experiment protocol. We ran several test sessions and subsequently refined the protocol, material and questions. The first 15 instances of the experiment were conducted in Wellington. While coding the first videos, we organized the second run in Leiden. Lastly, we focused mainly on the professional developers in the Dutch organizations. After 47 participant sessions (approx. 1.5 hour per session, including preparation), two weeks were spent on video coding and data entry (approximately three hours per participant).

6.4.2 Data Collection Process

We applied quota sampling (Wohlin et al., 2000) to obtain at least one professional developer for every three students who participated in the experiment. For both groups we applied convenience sampling. In Wellington, students participated voluntarily and went into a draw for a prize of NZ$50. In Leiden students participated both voluntarily and through a mandatory part of a course. Subjects from industrial organizations in Wellington participated voluntarily. All students but not all professionals had used UML during their studies. All participants had previous experience with UML. The study (which we referred to as "design study" towards the students) was mentioned during various software engineering lectures in Wellington. In Leiden, M.Sc. students who were enrolled in a research methodology course were required to participate. Student performance on the experiment did not influence their grade. In Leiden, absence from the experiment would be reflected in the course grade, though. Each



(a) Participant Station (b) Recording Environment

Figure 6.1: *Experiment environment*

participant was surveyed in the same way: the participant was welcomed and seated behind a table on which two blank pieces of paper were taped at 35 cm apart (see Figure 6.1(a)). A camera was placed so that it was able to accurately record movements of the participant's head. However, the environment was set up in a way so that the camera non-intrusively appeared as part of general audio and video equipment. This was done in order to prevent participants from getting overly nervous or self-conscious. The experimenter sat behind the participant so that the participant could concentrate on the task. This set-up also discouraged the participant from interacting with the experimenter. The set-up of the experimental environment was described in detail in the experiment protocol to minimize potential differences between the various locations at which the experiment took place. Generally, various participants were planned after one another. After a participant entered the room, he or she[1] was asked to sign a consent form and to fill out a preliminary questionnaire. This questionnaire consisted of 10 questions covering academic and industrial software modeling experience, a self-assessment of modeling skills, recent software architecture experience and demographics such as age, gender, (highest attained) level of education and native language. When designing the questionnaires, we followed standard guidelines (such as those described in Oppenheim, 1966). Next, it was explained that the objective of the study was to understand the use of software architecture design documentation and that a series of questions regarding such documentation would be asked. The participant was told that he could use the presented information in any way he deemed fit and that the experimenter did not know the correct answer to the question. In addition, the participant was requested to verbalize his thought process while using the architecture documentation. The first architecture design

[1]Most participants were male, we will continue referring to participants as males.

document presented was meant to serve as an example question only. The main purpose of this example question was to put the participant at ease and to acquaint him with the experiment protocol. The procedure of the experiment was carried out as follows: Architectural design documents would be placed on the participant's desk. Each architectural design was split up into two documents, one page contained a textual description and a another page contained a diagram. Both papers contained information about the same architecture. After placing the documents at their precisely defined positions, the context of the design would be briefly introduced. For instance, "This architecture describes the architecture for a booking system for flights." This introduction was made because in real software development scenarios, developers are aware of the domain or the high-level objectives of a system. Participants were then verbally asked questions about the design — such as *"Can component X directly authenticate users?."* Participants could then ask for a question to be repeated but no other requests such as clarifications would be accepted. After a participant had answered, the answer was repeated by the experimenter for verification purposes and the next question was asked.

Participants were given no feedback as to the correctness of their answer. The experiment was double-blind as neither participant nor experimenter was aware of the correct answer or in which of the documents the answer had to be found. As a result, the participant would not change his behavior according to his record of correct or incorrect answers during the experiment. Four architectures were used and three questions were asked per architecture — not counting the introductory example.

6.4.3 Material and Question design

Each set of architecture documentation consisted of a pair of sheets of which one contained text and one contained a diagram. After the last question, the participant was handed a second questionnaire, asking him to rate the extent to which he perceived the two media types used as being effective and the degree to which he understood all notational elements. All five text-diagram pairs (four pairs and one example pair) were inspired by industrial software architecture diagrams (SADs) documents in our possession. The documents were altered in such way as to:

- enable a uniform notation across cases,
- enable separation from larger design documents and
- enable translation to English where needed.

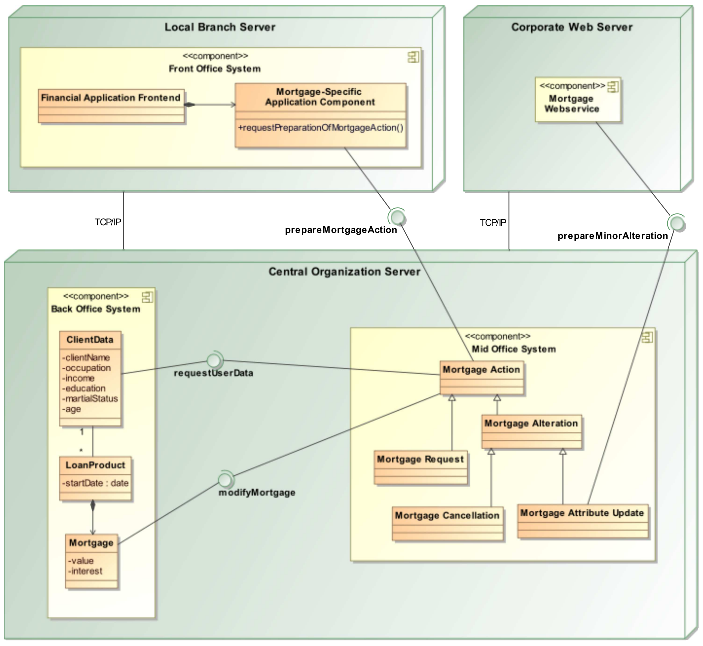
We focused on the ability of participants to extract design information from both grammatically and syntactically correct diagrams and texts. We corrected ambiguous constructs in original documentation and attempted to attain an overall coherent visual style plus lucid textual descriptions. All diagrams represented structural views of the system. We used UML 2 component and deployment diagrams. An example of one of the (verbose) diagrams is depicted in Figure 6.2(a). The text consisted of a description of the architectural component in natural language text. An example can be found in Figure 6.2(b). This text was assembled to be in concordance with industrial SADs. The non-verbose version of the text contained fewer details regarding the model and was about half as long as the verbose version. Questions and design documents were carefully tuned to each other in order to allow a multitude of subsequent analyses. Examples of questions are: *"Is system x the only component that may modify attribute y?"* and *"Through what node does system x connect to system y?."* The questions can be found in Table 6.4.

We define a set T that holds all design information described in the text sheet and a set D that holds all design information described in the diagram sheet. All design information is then described by $T \cup D$. We designed the questions so that the answer to some questions can be found in the intersection $T \cap D$, some answers can only be found in the complement $T \setminus D$ and some answers can only be found in the complement $D \setminus T$. Finally, some questions are not to be found in either set $(\sim (T \cup D))$. The distribution of the answers for our questions is described in Table 6.4.

When designing questions, we limited ourselves to questions which could be answered with information available either in text or via diagrams. We designed the architecture representations and the questions so not to rely on detailed knowledge of UML semantics. The experiment design process involved determining and listing the most important design information conveyed in the design, creating a set of questions relating to this information and validating the questions by means of trials. Overly complicated or ambiguous questions were disregarded or rephrased. We created a total of nine architecture pairs before selecting the final four, which were selected based on ease of understanding. Half the question set consisted of open questions. Answering open questions is comparatively more difficult because a) more information has to be gathered and b) it is not as clear as to whether further consultation of another medium is required once a partial answer has been formulated. Open questions provide more insight into how easily a participant is satisfied with partial information.

With a view to our hypothesis $\mathtt{H2}_1$, five questions were designed to address design information of a topological nature. Per architecture, we only asked three questions. We left more difficult questions for last as they required increased use of the media and might create a learning effect for subsequent easier questions.

Each architecture was described using one page of text and one page with diagrams. We created two versions of each architecture description. One version contained a verbose diagram and non-verbose (content reduced) text and the second version contained verbose text and a non-verbose (content reduced) diagram. We refer to the former as a diagram-dominant representation and the latter as a text-dominant representation. We created non-verbose versions of diagrams and text by removing elements or sentences from the complete ones. Each participant was presented two diagram-dominant representations and two text-dominant representations making sure that the two verbose diagrams were not placed on the same side of the desk each



(a) verbose diagram

The system described in this diagram provides support for creating new mortgages and alteration of existing mortgages.

The design aims to separate the complexities of the business logic from the Financial Application Frontend by bundling all mortgage-related services on a central Mid Office System. This system provides services for the setup of all `mortgage actions'.

The Front Office Component hosts a Financial Application Frontend which contains a Mortgage-specific Application Component. Due to concerns regarding decreased Back Office availability, mortgage action requests may have a maximum size of 300 kilobytes.

The Mortgage Webservice provides an additional method to update mortgage attributes. This service only connects to an interface provided by the Mortgage Attribute Update specialization.

(b) non-verbose text

Figure 6.2: *Example design*

Table 6.2: *Media Dominance*

architecture		α	β	γ	δ
medium		**t d**	**t d**	**t d**	**t d**
experiment version	*A*	V n	V n	n V	n V
	B	n V	n V	V n	V n

t = text / **d** = diagram
V = Verbose / **n** = non-verbose

time. Ergo, out of every four participants, every first participant received the ordering pair α (non-verbose diagram on his left side), pair β (non-verbose diagram, right), pair γ (verbose diagram, left), pair δ (verbose diagram, right). The questions were the same for diagram- and text-dominant representations of the architecture. The distribution of media dominance is summarized in Table 6.2.

6.4.4 Ordering Process

We eliminated a potential bias caused by a possible tendency of participants to start reading the information on a particular side (e.g. the left hand side for Western participants) by changing the position of the diagram for every pair presented. In order to prevent participant preferences for a media type, based on the verbosity of the medium, medium-verbosity was also balanced. Furthermore, the ordering of the architecture pairs used was changed for every participant so that results for any particular architecture pair (particularly the first and last pairs particularly pairs α and δ) would not be influenced by effects due to the participant learning, getting tired or getting bored.

6.4.5 Data Coding

We obtained information regarding the amount of time subjects looked at media by manual coding of the video recording. We counted switches between media, which media the participants looks at first and last. An excerpt of the manner in which the videos were coded is depicted in Table 6.3.

To ensure consistency of extraction of data from the video recordings, we used a set of guidelines for coding. For example, the first timing measurement started at the moment the question was spoken and timing stops when the participant mentions the core element of his answer. To ensure data-entry consistency, we used scripts to transfer timing information from spreadsheets to the database. We then performed consistency checks on the final data employed by means of a set of semi-formal consistency checks such as

$$\forall \mathit{switches}\, [\mathtt{mod}(\mathit{switches}) = 0 \Rightarrow \mathit{medium}_{\mathit{start}} = \mathit{medium}_{\mathit{end}}]$$

Table 6.3: *Example of video coding log for a single question*

question number	diagram or text (or answer)	video timing (mark[1])	relative timing (seconds)	looked at diagr. (seconds)	looked at text (seconds)
$\gamma 2$	d	594.9			
	t	610.5	15.6	15.6	
	d	615.2	4.7		4.7
	answer	619.1	3.9	3.9	
answer given: "no"					

[1] from beginning of video file

Derived metrics:

- *total time looked at diagram:* 19.5 seconds (80.58% of total time)
- *total time looked at text:* 4.7 seconds (19.42% of total time)
- *switches between media:* 2
- *started at:* diagram
- *ended at:* diagram

which were executed by means of a set of SQL queries. In addition, we recalculated all derived values (e.g. percentages) in the database.

6.5 Results and Discussion

In this section, we discuss the results of the experiment. A total of 47 subjects participated of which 35 were male and 12 female. The average age was 27, ranging between 21 and 41. The participants came from two universities and four different organizations:

- 12 BSc. and BSc-hons. students, 1 MSc. student and 1 Ph.D. student at the School of Engineering and Computer Science (ECS) at Victoria University Wellington;
- 1 BSc. student, 20 MSc. students and 1 Ph.D. candidate at the Leiden Institute of Advanced Computer Science (LIACS) at Leiden University;
- 12 developers from the field of custom software development at various different organizations in New Zealand and the Netherlands, including Capgemini the Netherlands, Infoprofs and ASR Insurances.

Because of non-normal distributions, we use the non-parametric Mann-Whitney U test for comparison between groups and Kendall's τ for bi-variate correlation analysis. As expected, professionals are significantly older and reported significantly more academic ($U = 105$, $z = -2.9$, $p < 0.01$) and industrial experience ($U = 107$, $z = -2.6$, $p < 0.01$) than the students.

In the following sections we will evaluate the data obtained with respect to our hypotheses.

6.5.1 Media Effectiveness

We evaluated media effectiveness in terms of how well participants were able to extract information from the documents i.e. in terms of the amount of correct answers given. The distribution of the correctness of the answers to the 13 questions we posed is leptokurtic and left-skewed. A Shapiro-Wilk (S-W) normality test confirms that the distribution of correct answers (slightly) deviates from normality ($p = 0.04$). We therefore resort to non-parametric tests for statistical analysis.

24 Participants worked with version A of the experiment materials and 23 participants worked with version B. Media dominance was distributed equally over both versions of the experiment (see Table 6.2) to be able to examine the effect of media dominance. We used a Mann-Whitney U test to check if there were no significant differences between the measurements obtained for both versions of the experiment for all variables under study.

We tested for any potential advantage of diagram-dominant versus text-dominant document pairs. Descriptive statistics for the amount of given answers that were correct for text-dominant versus diagram-dominant architecture descriptions are depicted in Table 6.5. Histograms for the number of correct answers given for each treatment are depicted in Figures 6.3(a) and 6.3(b).

A visualization of the differences of the distribution of answers is depicted in Figure 6.4. Surprisingly, we found that neither diagram- nor text-dominance had a significant effect on correct answers given nor on the time spent on answering questions. We therefore reject $\mathtt{H1}_1$. Diagrams were not more effective, despite a substantial number of participants whose first language was not English (77 percent of participants). We will further discuss the role of language in Subsection 6.5.7.

We could not observe significant amounts of initial media preference or average media preference either. These findings do not change if we look at a subset of the questions: For the four questions to which the answer could be found only in either medium, no medium type proved to be more effective in terms of causing more correct answers. The findings did not change either when we looked at professionals and students separately. This contradicted the first hypothesis: diagrams were not preferred over text.

The treatment showed no clear pattern. We therefore looked for patterns in the data. Analysis of the respondent behavior led to the identification of two groups: One group

Table 6.4: *Experiment Questions*

pair		question	type	nature	answer location (for experiment version)	
					A	**B**
ex		*Where is information x stored?*[1]	open	non-topological	$T \cap D$	$T \cap D$
α	1.	*Which external system is a source of information regarding x?*	open	topological	$T \cap D$	$T \cap D$
	2.	*What type of service is service x?*	open	non-topological	$T \setminus D$	$D \setminus T$
	3.	*How does system x search in system y?*	open	topological	$T \cap D$	$T \cap D$
β	1.	*Is input x accepted by system y?*	closed	non-topological	$T \setminus D$	$D \setminus T$
	2.	*Can system x directly provide functionality y?*	closed	topological	$T \cap D$	$T \cap D$
	3.	*Can message type x be ignored by system y?*	closed	non-topological	$\sim (T \cup D)$	$\sim (T \cup D)$
γ	1.	*Name one of the responsibilities of system x*	open	non-topological	$D \setminus T$	$T \setminus D$
	2.	*Is system x the only component that may modify attribute y?*	closed	topological	$T \cap D$	$T \cap D$
	3.	*Is there a limitation on functionality x when requested from system y?*	closed	non-topological	$\sim (T \cup D)$	$\sim (T \cup D)$
δ	1.	*What type of messages are sent from system x to system y?*	open	non-topological	$D \setminus T$	$T \setminus D$
	2.	*Through what node does system x connect to system y?*	open	topological	$T \cap D$	$T \cap D$
	3.	*Is the communication between system x and system y secure?*	closed	non-topological	$\sim (T \cup D)$	$\sim (T \cup D)$

[1] system and component names are anonymized to enable reuse of the experiment material

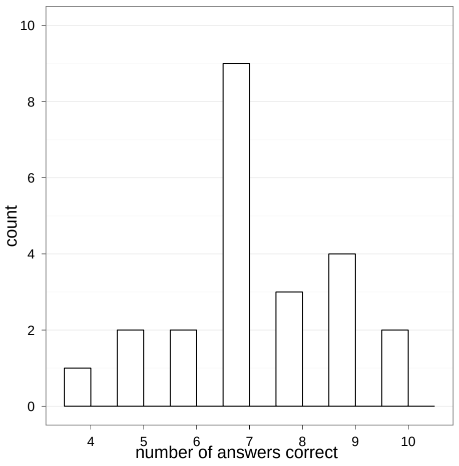
Table 6.5: *Descriptive statistics for the amount of given answers that were correct for text-dominant versus diagram-dominant architecture descriptions*

Treatment	**Answers Correct** (out of 12)		
	median $\tilde{x}$	mean $\bar{x}$	std. dev. σ
text–dominant[1]	8	7.70	1.54
diagram–dominant[2]	7	7.35	1.52

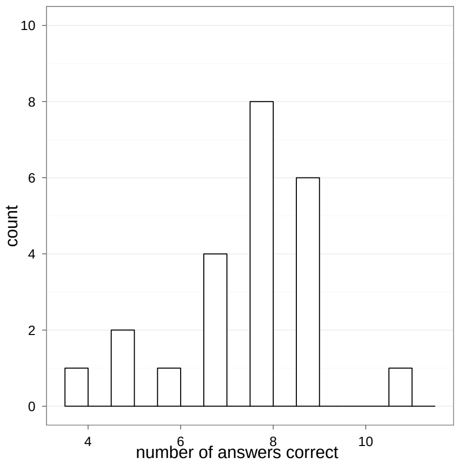
[1] version A of $(\alpha + \beta)$ + version B of $(\gamma + \delta)$

[2] version B of $(\alpha + \beta)$ + version A of $(\gamma + \delta)$

(for dominance distribution, also see Table 6.2)



(a) Diagram-dominant treatment



(b) Text-dominant treatment

Figure 6.3: *Histograms for the amount of correct answers per treatment*

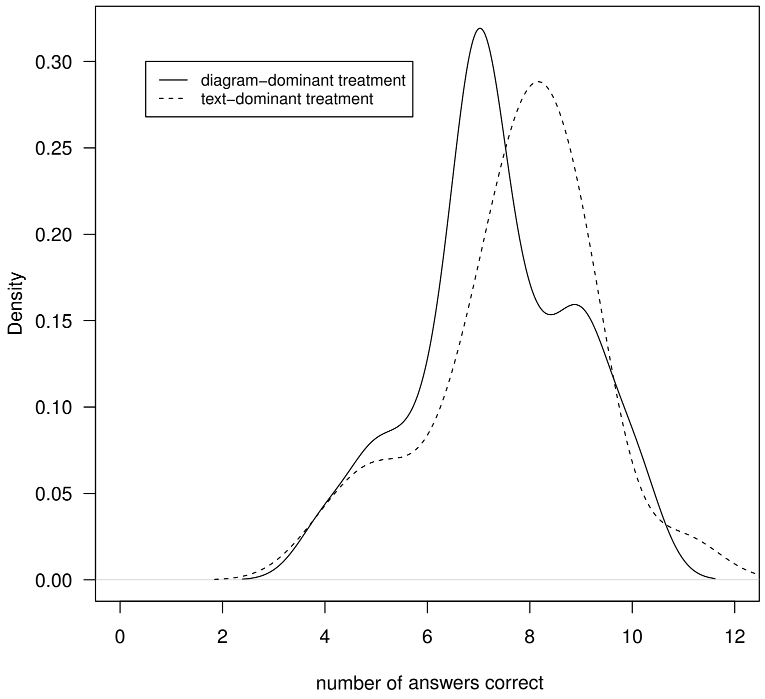


Figure 6.4: *Density plot for amount of correct answers per treatment*

(62 percent of the participants) predominantly used diagrams to answer questions, the other (the remaining 38 percent of the participants) used text more intensively. The group that uses diagrams more often, thinks that diagrams are more effective, answers faster and switches media more often and more frequently only resorted to diagrams to answer a question. The other group used text more often, was more experienced and scored better. Note that while the latter group suggests that participants who preferred text scored better, this does not imply a general advantage in effectiveness for text: participants who used the text-dominant architecture descriptions did not score better, as we mentioned earlier.

Using the post-experimental questionnaire, participants were asked to rate the effectiveness of both media on a 7-point Likert scale. Participants who prefer diagrams are significantly more likely to perceive the effectiveness of diagrams to be higher than participants who attribute a comparable score to their perception of the effectiveness of text. As mentioned, the diagram-preferring group of participants, who was faster ($\tau = -0.265, p \leq 0.05$) and rated their media type of preference as the most effective ($\tau = -0.265, p \leq 0.05$), did not score better in terms of correct answers, i.e. where not, in fact, more effective. In fact, we found that those participants who predominantly use text, score significantly better ($\tau = 0.281, p \leq 0.05$). So, participants who prefer text make a realistic judgment about the effectiveness of text. Participants who prefer diagrams, on the other hand, significantly overrate the effectiveness of diagrams. As a result, the group that preferred diagrams not only scored lower, but thought they had used the more effective media type. Diagrams seem to offer a specific group of developers a false sense of confidence. This group was more willing to provide an answer based on inferencing from the diagram than to carefully examine the text. In addition, we found that those with more experience more often resort to text to answer a question ($\tau = 0.273, p \leq 0.05$).

In the following section we investigate whether questions relating to topological architecture properties are better catered for by either media type ($\mathtt{H2}_1$).

6.5.2 Media Effectiveness for Topological Properties

In this section we explore whether diagrams are more effective at conveying topological design information to software developers. We define topological properties as the design information that is related to the static and structural dependencies between subsystems. Given that these topological properties lend themselves well to visualization, we expect that diagrams are the favored source for answering such questions. However, we could not observe that participants used diagrams more often to answer questions addressing properties of a topological nature (see Table 6.4). We therefore reject $\mathtt{H2}_1$. Contrastingly, we found that in comparison to non-topological questions, participants significantly more often used the diagram when they provided their answer ($U = 6.5$, $z = -1.797$, $p \leq 0.05$).

Media dominance had no influence on participant behavior for most of the five topology-related questions, with the exception of two questions: We found significant differences in the amount of correct answers given for questions $\alpha 3$ ($U = 191.5$, $z = -2.64$, $p < 0.01$) and $\delta 2$ ($U = 230$, $z = -2.03$, $p < 0.05$) (of the type *"How does X search in Y"* and *"Through what node does X connect to the Y?"* respectively). The answer to either question could be found in the diagram and the text in both document versions.

6.5.3 Media Preference

Media preference denotes the characteristic of an inherent preference of a person to prefer to work with either textual or graphical representations. Analyzing the data from this perspective, we found that participants prefer the diagram as their first source of information: in 10 out of 13 questions, participants first started examining the diagram. This count includes participants who only briefly gaze over the diagram before examining the text. Media dominance does not significantly influence this behavior. A possible explanation is that diagrams are used to get an initial global overview of a system. This is in line with the "Visual Information-Seeking Mantra" that Shneiderman (1996) describes. In his work, Shneiderman outlines the steps followed in visual information retrieval. He summarizes his principle in a mantra: *"Overview first, zoom and filter, then details-on-demand."*

When aggregating the usage pattern of media for all questions, 98 percent of all participants used both media types to arrive at answers. Only one participant relied exclusively on diagrams for answering all questions. When analyzing answers to individual questions, 27 percent of the 611 answers given (47 participants × 13 questions), were based on the use of only one medium. In contrast, five participants (10 percent) always used both media to answer a question. The latter group might be classified as a "thorough" group, not only because of their comprehensive media usage but because this group scored higher ($\tau = -0.312, p \leq 0.01$). The latter fact is not a result of the group consisting mainly of experts. Participants of this group were not more experienced nor was it composed mainly of professional developers. Also, these participants had slower response times ($\tau = -0.464, p \leq 0.001$) further corroborating the notion of "thoroughness." Note that "non-thorough" participants had no particular reason to work quickly as no time limit was imposed.

Interestingly, we found that "thorough" participants were more likely to start and end with looking at a diagram. In contrast, another group that scored better than average, predominantly used text. This latter group was composed of experienced participants. So, "thoroughness" does not correlate with media preference but "experience" does. A potential explanation for that latter observation is that industrial practice could have made experienced developers diagram-averse in the sense that they prefer understanding all text accompanying a diagram. Perhaps the low quality of diagrams they had do deal with in the past created low expectations as to the utility of diagrams in general. By using an eye tracker, Yusuf et al. (2007) also found that more experienced developers read diagrams differently.

6.5.4 Guesses and Suppositions

For four questions, the answer could be found in either the text or in the diagram, depending on the version of the experiment a participant obtained. For each of these four questions ($\alpha 2, \beta 1, \gamma 1$ and $\delta 1$), we looked at the amount of participants that used *only* the medium that did not provide enough information to answer the question. These participants did not switch to the medium from which the answer could be derived. Note that nothing inhibited these participants from spending as much time as they needed to answer the questions. These participants therefore seemed to prefer guessing over continued consideration of the presented material. For all participants, 10 out of a 188 (4 questions × 47 participants) or 5.3 percent of all questions, were given based on a single medium from which the answer could not be derived. Only in one of these 10 cases did a participant guess the correct answer. This was for question β 1 which was the only closed question of this type. This participant therefore had a "fifty-fifty" chance of guessing the right answer. These 10 answers were given by 9 different participants. So, 19 percent of participants demonstrably and needlessly guessed the answer to at least one question. Two of these participants were experienced software developers who are active in industry. Note that "switching between media" in the context of this experiment refers to the act of slightly moving one's head from left to right or vice versa. As architecture representations are often vast and dispersed over various sources, in practice (much) more effort is likely to be needed to understand a given aspect of a software architecture.

To be able to find a measure of the extent to which developers are satisfied with incorrect information, we inserted two "false friends" into the experiment design. These "false friends" constitute information that resembles a correct answer but would be easy to distinguish as incorrect information, if given sufficient attention would be given to detail. In the experiment design, the answers to three questions could not be derived from either medium (for both versions of the experiment). In two of these cases, we inserted these "false friends." For question β 3 (*"Can message type x be ignored by system y?"*), one of the classes contained the method "`+ignoreMessage()`." That particular class had no relation to external systems. For question γ 3 (*"Is there a limitation on functionality x when requested from system y?"*), the text contained the phrase "requests may have a maximum size of 300 kilobytes." Request volume and constraints on individual requests are unrelated, most participants correctly observed. For these questions, we took into consideration why a participant gave a specific answer (by means of the think aloud protocol) to be able to tell whether a "false friend" was the cause for the specific answer given. In univariate analysis, we found that those with more industrial experience, are less likely to fall for a false friend ($\tau = -0.333, p \leq 0.01$). These participants are less quickly satisfied with information that only resembles a correct answer. We found that the participants who demonstrably and needlessly guessed the answer to at least one question, were not more likely to fall for a false friend (Mann-Whitney $p = 0.88$).

6.5.5 A Case Against Overlap

While it is commonly accepted that images are far better remembered than text (e.g. McDaniel and Pressley, 1987), there are various restrictions to their use. For example, the third of the "ten commandments of picture facilitation", of Levin et al. (1987) (which were more recently validated by Carney and Levin (2002)) reads:

> *"Pictures shalt not be used in the presence of "heavenly" bodies of prose. If the text is highly memorable to begin with, there is no need to add pictures."*

For software design representations, this would imply that a diagram should only be used when a textual description is insufficient. Moreover, if a textual description is very clear (*"heavenly"*), diagrams should be avoided to prevent confusion. We reported that we found that neither text nor diagram-dominant descriptions are more efficient in communicating software architecture design. However, for five questions, we represented similar information in both media ($T \cap D$ - also see Table 6.4). Indeed, we found in industrial reality often an overlap of the information presented in diagrammatic and textual models (Heijstek and Chaudron, 2011). Given Levin's commandment, due to confusion, participants could have scored lower for questions to which the answer could be derived from both media (the example question and questions $\alpha1$, $\alpha3$, $\beta2$, $\gamma2$ and $\delta2$), compared to the other questions. The score per question varies too much to be able to compare whether this was the case. When we consider whether participants who predominantly used the text or the diagram, we found that this is not related to whether they answered these questions correctly.

6.5.6 Conflicting Information

In industrial reality, developers are confronted with design documentation that is mostly incomplete and often inconsistent. Both incompleteness and inconsistency in UML diagrams have been addressed by empirical research. For example, Lange and Chaudron (2006) studied the effect of defects in UML models on developer comprehension. In their controlled experiments with a group of 159 students and industrial practitioners, they found that defects in UML models often remain undetected and cause misinterpretations. In addition, they found no implicit consensus about the interpretation of undetected defects and conclude that defects in UML models are potential risks that can cause misinterpretation and miscommunication. Another interesting finding of this study is that the presence of domain knowledge strongly decreased the detection rate for a defect type. Domain knowledge might therefore lead people to more quickly fill in omissions based on assumed domain knowledge. Nugroho (2009) found that a higher level of detail in a UML model significantly improves correctness and efficiency of subjects in comprehending UML models. He also found that models with a lower level of detail were more often misunderstood or misinterpreted. Balzer (1991) reported that harsh consistency constraints on design in practice are often removed in favor of flexibility. The architecture used to support the first question, the example question, contained conflicting information between the diagrammatic and textual representations. By analyzing the answer and participant behavior, we could determine which medium type was more dominant for participants. Out of all participants, 85 percent examined both media. The other 15 percent used only the diagram (all of these participants on average preferred diagrams for all subsequent questions). Out of those who used both media for the first question, we found that only 35 percent preferred the answer that could be deduced from the text. We should note that the example architecture was text-dominant for all participants. Many participants noted that they found the inconsistency and deliberately choose the diagram. When confronted with conflicting information, developers seem to decide that the information presented in a diagram is more authoritative than the textual information.

6.5.7 Participant Characteristics as Performance Predictors

Gemino and Wand (2003) recommend examining three antecedents of knowledge construction in empirical evaluation of model representations, based on Oei et al.

(1992): (a) content, (b) presentation and (c) model viewer characteristics. Now that we have established that no media type was more effective in communicating architecture information, we set out to investigate: To what extent can developer (or: model viewer) characteristics explain the variance in the amount of correct answers given? In this section we investigate which participant characteristics can be used as predictors of performance. We employed a multiple regression analysis using the backward elimination method (Hocking, 1976). We considered the following variables:

1. *Experience* (Explained in Section 6.5.7)
2. *Media Preference* (Coded as diagram = 0, text = 1; Explained in Table 6.1)
3. *Media Exclusion* (i.e. how often a participant only used one medium)
4. *Diagram Working History* (i.e. how long it has been since the participant last worked with software design models)
5. *Media Inclusion* (i.e. average switches between media) (Also explained in Table 6.1)
6. *Self-Rated Modeling Skill* (Explained in Table 6.1)
7. *Average Time per Question* (Explained in Table 6.1)
8. *Linguistic Distance* (Explained in Section 6.5.7)

The first and last participant characteristics are explained next in more detail.

Linguistic Distance

For linguistic distance we adopted a measure reported by Chiswick and Miller (2004) based on a study by Hart-Gonzalez and Lindemann (1993). Chiswick and Miller pose the question: *How difficult is it for individuals who know language A to learn languages* B_1 *through* B_i*, where there are* i *other languages?* They go on to state that *"if it is more difficult to learn language* B_1*, than it is to learn language* B_2*, it can be said that language* B_1 *is more "distant" from A than language* B_2*."* A list of the languages encountered in the experiment and their associated linguistic distances to English can be found in Table 6.6.

We found that univariately, this measure significantly correlates with the amount of correct answers given ($\tau = -0.289$, $P \leq 0.05$). This implies that the further a participant's native language is removed from English, the fewer correct answers are given. Language distance, therefore, is important and should be minimized. This implies that diagrams were unable to bridge language barriers. In addition we found that participants whose language has a certain minimum distance away from English were significantly more likely to switch between media.

Table 6.6: *Language Grouping & Distance*

language[1]	*n*	score[2]	family	distance[3]
English	13	-	Indo-European	0
Romanian	1	3.00	Indo-European	0.33
Dutch	20	2.75	Indo-European	0.36
German	1	2.25	Indo-European	0.44
Spanish	1	2.25	Indo-European	0.44
Farsi	1	2.00	Indo-European	0.50
Bulgarian	1	2.00	Indo-European	0.50
Tagalog	1	2.00	Austroeasian	0.50
Bengali	1	1.75	Indo-European	0.57
Mandarin	4	1.50	Sino-Tibetan	0.67
Arabic	1	1.50	Afroasiatic	0.67
Chaouia	1	-	Afroasiatic	-
Nyanja	1	-	Niger-Congo	-

[1] self-reported by participant (“Native Language”)
[2] reported in (Hart-Gonzalez and Lindemann, 1993)
[3] inverse of score ($\frac{1}{\text{score}}$) (Chiswick and Miller, 2004)

Participant Experience

In this paragraph, we explain the metrics we used to quantify participant experience. We used the proposed ordinal classes of participant experience by Höst et al. (2005):

- **E1:** undergraduate student with less than 3 months recent industrial experience
- **E2:** graduate student with less than 3 months recent industrial experience
- **E3:** academic with less than 3 months recent industrial experience
- **E4:** any person with recent industrial experience, between 3 months and 2 years

This metric is a compound, composed of the values we collected for academic and industrial experience (see Table 6.1). A comparison of the distribution of these three variables is depicted in Figure 6.5. In this figure, one can observe that the metric proposed by Höst et al. (the middle line) is a proper compound of academic and industrial experience for our participant.

The values for experience we obtained are not normally distributed (leptokurtic, right skewed, S-W $p \leq 0.001$). We use Kendall’s τ (Noether, 1981) for bivariate statistical analysis. We found that the Höst et al. index of experience individually correlates with the amount of correct answers a participant gave ($\tau = 0.31, p = 0.1$). Experience,

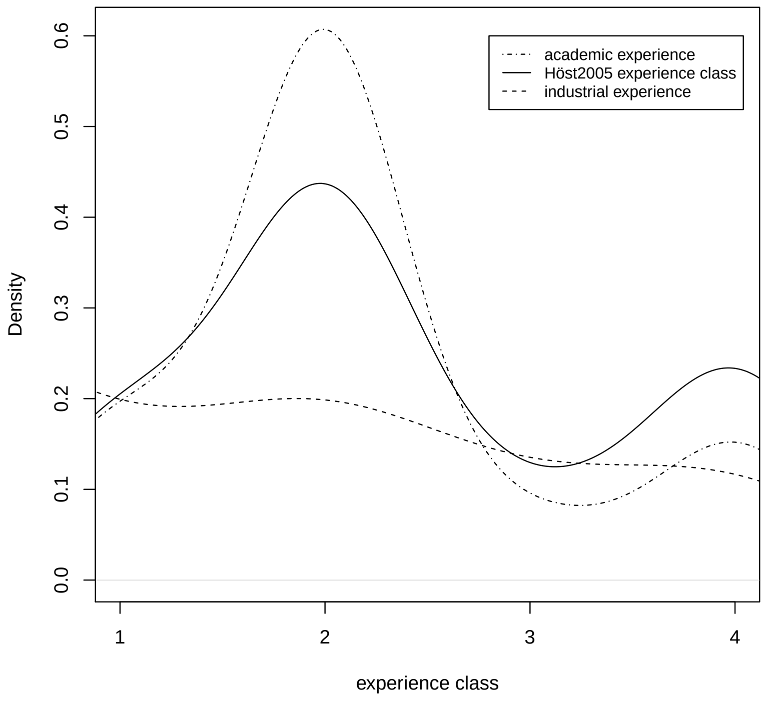


Figure 6.5: *Density plot for academic experience, industrial experience and Höst experience class of participants*

however, does not relate to the speed with which a participant was able to answer or the amount of times he switched between media.

Multivariate Analysis Results

The results of our multivariate analysis are summarized in Table 6.7. The multivariate analysis results in a model that features four significant variables, namely media preference, linguistic distance, experience and self-rated modeling skill. The model is highly significant ($p \leq 0.001$) and explains 50 percent of the variability of correct answers given. We were able to rule out multicolliniarity among the resulting predictors; the variance inflation factors (VIF) are not substantially higher than 1 (Bowerman and Richard, 1990). No individual subject seemed to influence the model as e.g. no case has a standardized residual larger than 2. We found that the coefficient for language distance is negative, implying that the further a participant's language is from English, the fewer correct answers the participant will give. Remember that media preference is coded as diagram = 0, text = 1. Therefore, this confirms the earlier result that a textual media preference was beneficial for correctly answering the questions. As expected, experience was also beneficial for providing correct answers.

Table 6.7: *Multivariate regression for "Amount of Correct Answers"*

Model[*]	**Unstdized. Coeff.**		**Std. Coeff.**	**t**	**Sig.**	**Correlations**			**Collinearity**	
	B	Std. Error	β			Zero-order	Partial	Part	Tolerance	VIF
(Constant)	5.959	0.760		7.842	0					
language distance	−3.288	0.927	−0.411	−3.548	0.001	−0.370	−0.499	−0.406	0.972	1.028
media preference	1.236	0.418	0.343	2.960	0.005	−0.362	0.443	0.338	0.971	1.029
modeling skill	0.442	0.150	0.342	2.949	0.005	0.294	0.432	0.337	0.974	1.027
experience	0.499	0.188	0.309	2.653	0.012	0.409	0.395	0.303	0.965	1.036

[*] Model Summary: $R^2 = 0.503; p \leq 0.001$

The fact that self-reported assessment of modeling skill turned out to be a predictor is consistent with similar findings that show that self-reported ability correlates with e.g. actual mechanical and spatial ability (Hegarty and Just, 1993). In order to be able to see which predictor was the strongest, we standardized coefficients (which are measured in standard deviation units). We found that language distance had the greatest contribution to the model that predicts the performance of participants. Those participants whose native language was further away from English had greater difficulty understanding the software architecture designs.

6.6 Threats to validity

In the following, we discuss a number of factors that may have influenced the results obtained from the experiment in a way that prevents them from being indicative for other contexts as well. We categorized our potential threats to validity based on Wohlin et al. (2000).

6.6.1 Internal Validity

We addressed problems relating to maturation i.e. increasing familiarity with a problem, in several ways: We interacted with participants by means of a strict interaction protocol so to be able to react to questions or remarks in a uniform way. We thus prevented participants to use the experimenter as an additional source of information. We also changed the order in which participants received the architectural descriptions (which also prevented effects due to fatigue).

We minimized the threat of "diffusion of treatment" by asking participants not to discuss the experiment with their colleagues or fellow students. No materials could be taken by participants from the experiment environment.

Threats related to instrumentation were addressed by using a protocol for the use of the camera. On the different locations where we conducted the experiment, we "camouflaged" the camera by placing it inconspicuously among other equipment in its surroundings (as in Figure 6.1(b)). Whenever the camera needed to be handled, it was done while no participant was in the room.

A related potential threat is that participants behave differently because they are aware of the experiment situation. Either stress or the "Hawthorne effect" could introduce a negative or positive bias respectively. In order to address such factors, we introduced an example question to allow participants to get used to the environment. We believe that participants found the questions challenging enough to fully concentrate on the design documentation, i.e. practically became oblivious to the experiment situation. This is supported by our analysis of the video material. The experiment protocol prescribed interaction while documentation was switched in order to keep participants occupied. We equivocated the use of a camera and by only mentioning the session would be "recorded" without providing further detail how this would be done.

Another possible threat is that various studies of the process of solving problems reported that verbalization of this process improves problem-solving performance (e.g. Johnson and Shih-Ping, 1999, Flaherty, 1975). Therefore, this might have influenced the extent to which participants were able to provide a correct answer. In addition, the intensity and contents of the verbalization varied strongly among participants. Not all participants will have benefited from the performance increases that verbalization yields. Measures taken to diminish the described threats to validity are (1) not facing the participant, (2) not answering participant questions, (3) making use of an interaction protocol to limit the amount of possible responses that can be given to a participant. The interaction protocol contains a process for stimulating thinking out loud.

The phrasing of the questions might have been suggestive in terms of which medium is likely to provide the answer. This might have introduced a bias for media preference and effectiveness, which may or may not become directly apparent. Our findings do not suggest any such bias at all but due to the complexity of the subject we cannot rule it out for every question.

Finally, one might argue that the separation of diagrams and text into two pieces of paper might have invited participants to always use both. We had to create a sufficient amount of physical separation between the sheets in order to be able to distinguish their use in the video material. However, it is not clear as to whether less physical separation would have made a difference and certainly some participants almost exclusively used diagrams, demonstrating the potential to ignore one of the documents if that was felt to be appropriate.

6.6.2 External Validity

The extent to which graduate, undergraduate and doctoral students are representative of professional software developers and the threat of interaction of selection and treatment respectively, have been addressed in various other studies (e.g. Arisholm and Sjøberg, 2004, Briand et al., 2005, Höst et al., 2000). The students who participated in this experiment were all exposed to software modeling in general and the notation used for the diagrams (UML) in particular during various courses at the bachelor or master level. In addition, more than a quarter of participants were actual professional software developers (more than one for every three students). We maintain that for understanding the software architecture designs we used, expert knowledge of the UML is not a prerequisite for extracting information from the diagrams needed for answering our questions.

In other words, we do not know whether these participants would not have done equally well, if they had used the diagrams more often. If experience with low-quality diagrams really played a role in our experiment then this particular finding would imply "media adversity of good participants" rather than on "media effectiveness for text-centric developers." Note, however, that overall we detected no better effectiveness for text, so the potential bias did not affect our overall results.

The findings were based on fragments of architectural descriptions that were closely based on samples of such descriptions taken from industrial SADs. Hence, these are considered to be representative for the class of project that use a mix of text and diagrams.

The threat of apprehension was mitigated by assuring anonymity and by explaining the recordings could only be accessed by the involved researchers (whose names were mentioned) and would be destroyed after data collection. Also, no time limit was imposed upon participants. We avoided hypothesis guessing by keeping the participants uninformed about the study objective.

6.6.3 Conclusion Validity

Reliability of measures was ensured by means of testing the used cameras and determining proper positioning to be able to clearly obtain head and eye gaze movement. This information was documented in the experiment protocol. A coding protocol was established for coding the videos and a second researcher re-coded videos to randomly assess accurateness. Database entry was semi-automated and subject to rigorous consistency checks. Threats regarding random irrelevancies were addressed by choosing quiet locations to conduct experiments. The experiment materials and environment were tested before real sessions commenced. The validity of the applied statistical tests (e.g. the assumptions and correct application), subject selection and the data collection process were discussed in previous sections.

6.7 Recommendations

This section contains an overview of the recommendation that follow from the findings of this study.

- The use of both text and diagrams is needed. The use of only one medium is not recommended, not even when the nature of the design decision to be communicated is topological. Also for topological information, for which diagrams intuitively seems to be the preferred communication medium, a textual description of the topology should be added.

- Another important reason for not defaulting to the use of only diagrams are that diagrams do not seem to bridge linguistic distance. In addition, they also potentially induce a false sense of confidence in developers in the sense that they make assumptions.

- When using diagrams, make clear how these should be read. In an earlier study, Holsanova et al. (2009) found that an "integrated format with spatial contiguity between text and illustrations facilitates integration." Employing this method would imply annotating a diagram with a description of how to read it, either in natural text or using a more formal notation such as OCL.

- Consider your audience when engaged in GSD: Make sure that documentation is unambiguous.

- Do not use ambiguous constructs. Use similar terminology in both diagrams and text.

- Developers who are better at modeling, read diagrams better. UML training or training regarding conceptual modeling might be beneficial for developer understanding of architecture representation.

6.8 Conclusions and Future Work

Architectural design documentation is essential for communicating an architect's intentions. In current practice such documentation consists of a mix of diagrams and textual descriptions but their creation is not informed by solid knowledge about how the documentation is perceived by developers. We therefore conducted a controlled experiment in order to evaluate the merits of different mixes of diagrammatic and textual descriptions in which we tried to approximate industrial reality. One of the results is that neither diagrams nor textual descriptions proved to be significantly more efficient in terms of communicating software architecture design decisions. Another unexpected result is that diagrams are not more suited to convey design decisions of a topological nature. Remarkably, participants who predominantly used text, scored significantly better; not just overall but with respect to topology-related questions as well. Also, diagrams were not able to alleviate the difficulties participants with a native language other than English had in extracting information from the documentation. In combination, these findings question the role of diagrams in software architecture documentation.

However, while diagrams were not superior regarding media effectiveness they still seem to perform a special role. Participants were more likely to use diagrams as their first source. They were more likely to look at the diagram at the very moment when they provided answers to questions of a topological nature. Interestingly, thorough developers tend to start and end with diagrams. More research is required to fully understand how text and diagrams could complement each other, in particular with respect to topological system properties.

Our analysis discovered two emerging group characterizations. One group predominantly utilized diagrams, was faster and overrated the effectiveness of diagrams; the other group was more experienced and preferred text. Further analysis needs to be performed in order to understand these groups, so to be able to specifically cater for them in the creation of software architecture documentation.

Finally, by conducting a multivariate regression analysis we identified developer characteristics that can be used as developer performance predictors: linguistic distance, media preference, experience and self-rated modeling skill. The participants who performed best had a native language close to English, used more text than diagrams, were more experienced and rated their modeling skill to be relatively high.

Summarizing, while our experiment and subsequent analyses produced some very interesting concrete findings, we feel that their ultimate value lies in the impetus they provide to perform further research to better understand the effectiveness and roles of media types in software architecture descriptions.

Chapter 7

Contrasting Model-Driven Development with Code-Centric Development

MDD is seen as the natural continuation of the trend of raising the level of abstraction at which software is developed. Consequently, in the past decade, there has been increasing interest in MDD in both industry and academia. In addition, MDD is emergent in GSD projects. Nevertheless its impact on the development process in large-scale, industrial practice is not yet clear and empirical validations of adoption of MDD tools and techniques are scarce. This chapter therefore addresses how the characteristics of a large scale, industrial model-driven development project in the context of global software development compare to non-MDD projects. We specifically focus on the quantification of process metrics.

This chapter is based on the following publication:

Werner Heijstek and Michel R. V. Chaudron (2009) **Empirical Investigations of Model Size, Complexity and Effort in Large Scale, Distributed Model-Driven Development Processes — A Case Study.** *In Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2009) pages 113–120, Patras, Greece*

7.1 Introduction

In the past decade, there has been increasing interest in MDD (Selic, 2003) in industry and academia. MDD is also emergent in GSD projects (Jiménez et al., 2009). The quality and productivity benefits claimed for the use of MDD triggered many studies to advance MDD practices. Nonetheless, few empirical studies are available that study the impact of applying MDD on industrial software development As a result, the general impact on software development of using MDD is unclear.

The structure of this chapter is as follows: Sections 7.2 and 7.3 elaborate on the study objective and related work. Section 7.4 explains the case study design. Section 7.5 discusses the results and the conclusions and future work are presented in Section 7.6.

7.2 Objectives

In this chapter, we address **RQ3** (Section 1.3). This question aims to explain how increasing model-centrality impacts the problems associated with GSD. To this end, the differences between code-centric and model-driven software development are analyzed.

In this chapter in particular, we aim to add to the limited experience reports in which the specificities of MDD cases in general and cases of MDD adopted in GSD context in particular, are reported. As established, it is important to investigate industrial cases of MDD to add to the scarce literature. Such investigation is also beneficial for benchmarking purposes and to evaluate the impact of the process and techniques used in general. To these ends, we pose the following research question:

> *How do the characteristics of a large scale, industrial model-driven development project in the context of global software development compare to non-MDD projects?*

This question is divided into sub-questions regarding key characteristics regarding the software development process in general and models in particular:

1. What types of diagrams are used?
2. How is effort distributed over the “classic” development phases?
3. How big and how complex are these models?
4. How does model size grow over time?
5. Do model size and complexity impact defect count?

7.3 Related Work

The main hypothesized benefits to be gained from adoption of MDD are

1. increase in productivity,
2. improved code-quality,
3. improved re-usability and
4. improved maintainability.

The main two principles that enable these benefits are (1) provision of better abstraction techniques and (2) facilitation of automation (Mohagheghi and Dehlen, 2008, Staron, 2006, Kleppe et al., 2003). The software architect fulfills a central role in ensuring that these potential benefits are actually obtained. The software architect is instrumental in enabling correct model transformations such as code generation. His objective of complying to non-functional requirements implies a careful consideration of available modeling case tools and a leading role in the design and application of a Domain Specific Language (DSL) (Van Deursen et al., 2000).

We focus on productivity. In this section, an overview is presented of related work. First, we elaborate on the state of empirical research in MDD tools and technique application. Second, the impact on productivity is discussed.

7.3.1 State of Empirical Research

Empirical evidence regarding the benefits of application of MDD tools and techniques is sparse. Literature regarding MDD in large-scale, industrial projects often describes processes in which legacy systems are reverse engineered to MDA (e.g. Anda and Hansen, 2006, Reus et al., 2006, Fleurey et al., 2007). Reports are mostly qualitative (Staron, 2006, Raistrick, 2004, Baker et al., 2005).

An extensive review of literature regarding MDD (published between 2000 and 2007) was executed by Mohagheghi and Dehlen (2008), the results of which have been summarized by Hutchinson et al. (2011):

- Most studies of the 25 selected papers were experience reports from single projects;
- MDD was applied in a wide variety of organizations; methods of code generation varied;
- MDD techniques are very dependent on tooling;
- productivity impact varied *widely* and more empirical studies that evaluate MDD are needed.

Concluding their work, Mohagheghi and Dehlen explicitly recommend that *"future work for evaluation of MDE should focus on performing more empirical studies, improving data collection and analyzing MDE practices."*

7.3.2 Impact on Productivity

Most empirical studies regarding MDD address questions regarding efficiency (White et al., 2005). Still, only few studies offer enough data to quantify and baseline productivity (and quality) in industrial MDD projects (Shirtz et al., 2007, Weigert et al., 2007). Anecdotal evidence in literature claims that adoption of MDD has hampered productivity as much as 27 percent (MODELWARE D5.3-1, 2006) and improved productivity as much as 800 percent (Baker et al., 2005). A case study by MacDonald et al. (2005) of modification of a legacy system using MDD found that development lead time increased due to "workarounds required to integrate with legacy systems." This directly impacts the work of a software architect. Furthermore, they found as many defects as the authors would have expected with "traditional development." Moreover, these defects were more difficult to find and repair in the models because of difficulties in tracing errors from the compiler directly back to the model without using the generated code as a reference. Suffice to say that technical support of this type of development process is demanding. The study did not use a fully functional executable model. Also, it was hard to maintain platform independence due to work methods and a lack of generic libraries.

In their recent multi-method study of the state of the practice of MDD, Hutchinson et al. (2011) specifically addressed the perceived impact of MDD activities on productivity and maintainability. They found that the largest impact was not code generation or meta-model reuse but *"the use of models for understanding a problem at an abstract level."* The second greatest impact was thought to be *"use of models for team communication."*

7.4 Case Study Design

In this section we describe the case study design.

7.4.1 Context

We examine a project in which a system was defined, designed and built for supporting the mid-office processes of the mortgage business. The client was a large financial institution that operates globally and the contractor is the Dutch subsidiary of an international IT service provider. The department responsible for development of the system has extensive experience with building tens of software systems for the financial sector as well as experience with global software development.

A function point analysis that was based on the requirements was executed in the early stages of the project. It reported a total of 1,973 function points to be built. During the execution of the project, various change requests have been made. A total of 32 team members worked on the project of which only a few did not work on this particular project full-time. This corresponds to 28 full-time equivalents (FTE). Only four team members had experience with a previous project in which MDD techniques were applied. Total project duration was 24 months. RUP was used as development process. The RUP is an adaptable process framework that is architecture-centric and risk-driven and can be used for iterative software development (Kruchten, 2003b). The project was carried out distributedly. A team of six developers and six testers worked in India. Modeling was done in the Netherlands by a team of four designers, development was done at both locations and testing was done in India. The Dutch project leader was the main point of contact to the client in the Netherlands.

7.4.2 Specificities of the MDD Approach

In the development process, a DSL with strict modeling guidelines is used. These guidelines address the dynamic aspects of the system and are based on UML 2. The guidelines are developed without code generation in mind. The rationale behind using UML for this reference model was that UML is more widely known than other suitable candidates such as the Business Process Modeling Notation (BPMN). Model consistency is enforced in two ways. First, the developers are restricted by the constraints imposed by the UML meta-model. Second, a model validator is used. This validator checks syntax and conformance to the UML meta-model. When code is generated, the models are validated first. However, complete validity of the models is not so much the goal as a working result. Source code is generated by using a code generator. This generator is realized through a combination of open source libraries. During the project, developers work at extending and enhancing the code generator. A general overview of system components is depicted in Figure 7.1. The system consists of two parts. Part one is a complex web-based system for user interaction, this system contains a web service client. Part two is a web service that enables existing systems to request information. The system domain model is formally modeled in UML and completely generated into a Java implementation. The model syntax consists of

- classes and properties,
- property types and their names and documentation,
- associations between classes and
- required fields and constraints on classes.

Inside these entities, no other behavior is modeled. The classes contain no operations. Screens that are deemed suitable to be modeled such as “input screens” and “selection

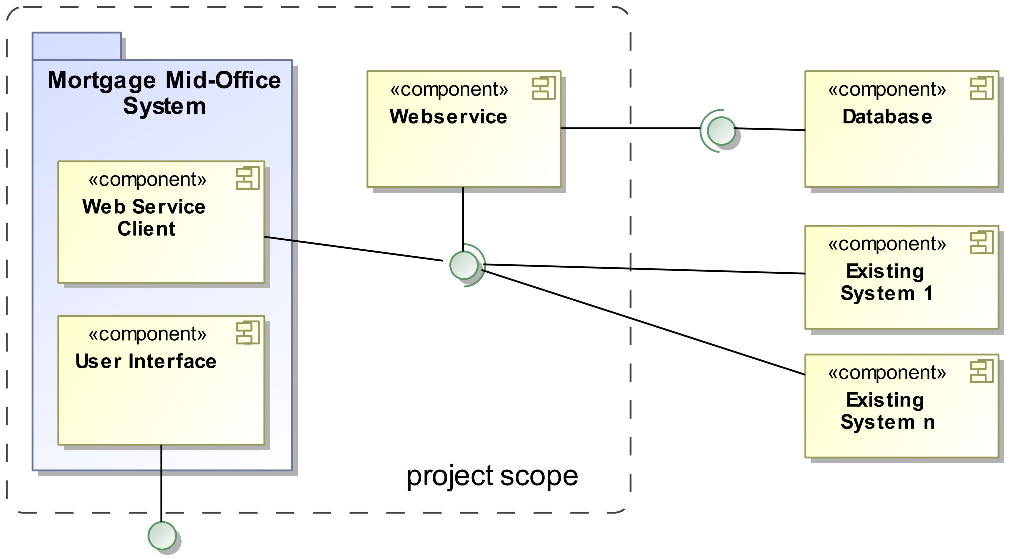


Figure 7.1: *Overview of case system components*

screens", are described in UML and completely generated to source code. For more complex, custom screens, the syntax of the DSL does not suffice. These screen are fully or partly hand-coded. The web service client is completely generated from the UML model. Some parts of the business logic layer can be fully generated from the UML model. Other parts are fully hand-coded.

Initially the target implementation language was a high level, business-oriented programming language that would have been relatively easy to maintain. Due to limitations imposed by using this language, later in the project it was decided that Java 2 Enterprise Edition was to be generated from the models. Both The Spring Framework[1] and Hibernate (Bauer and King, 2004) are used for target development and the tools used are Eclipse[2], JBoss[3] and MagicDraw[4]. The final application is to operate on the IBM WebSphere[5] platform and will use a DB2[6] database. Approximately 90 percent of the code is generated, the remaining 10 percent is written "by hand".

[1]http://www.springsource.org/
[2]http://www.eclipse.org/
[3]http://www.jboss.org/
[4]https://www.magicdraw.com/
[5]http://www-01.ibm.com/software/websphere/
[6]http://www-01.ibm.com/software/data/db2/

7.4.3 Data collection

We collected quantitative data regarding process and models from various sources. The models were collected from a Subversion repository. Effort and defect data was collected from SourceForge Enterprise Edition. For this study, we use the notion of "diagram groups." Each screen in the application that was created consisted of a set of diagrams. After collecting all diagrams from Subversion, metrics were extracted using SDMetrics (Wüst, 2009). This process was automated using a set of Bash and Perl scripts. Metric data is available on a per diagram basis whereas effort and defect data was only available on a per-diagram group basis. Therefore, the resulting metric files were aggregated per diagram group so that effort and defect data per model could be combined. In this project each diagram group was contained in a separate file.

7.5 Results

UML diagrams were created using MagicDraw 14.5[7]. The DSL specification required diagrams to be grouped together. A total of 119 diagram groups contain a total of 386 diagrams. A bar chart of the UML diagram types (Figure 7.2) shows that activity diagrams are most abundantly used, followed by class and use case diagrams. The reason for the plenitude of activity diagrams is that the development of the models is user interface centric. This means that the process flow of the process that the system will support is captured in the activity diagrams as a set of screens. The process flow of modeling is chosen so that during maintenance changes in the business process can easily be translated into changes to the models. In total, 104 diagram groups contain one or more activity diagrams, 32 diagrams groups together contain 150 class diagrams and all use case diagrams are spread over just two diagram groups. The average model consists of one or two activity diagrams and zero or one class diagrams. Because activity and class diagrams are the most important (and most prevalent) diagram types, we will focus on these during the remainder of the study. In the next sections, we will address model size, model complexity, development effort from various perspectives, defects and changes and defect discovery over time.

7.5.1 Model Size

Model size metrics have been proposed in many studies (e.g. Marchesi, 1998, Genero et al., 2002, Kim and Boldyreff, 2002). Empirical findings regarding model size metrics have been reported fairly scarcely. Most often, class diagram size metrics are reported (e.g. Marchesi, 1998, Lange and Chaudron, 2005, Lange, 2006, Egyed, 2007, Costagliola et al., 2005, Nugroho and Lange, 2007). To establish the size of a model, we summed all the size elements of all diagrams that were used in a model. Definitions of both

[7]http://www.magicdraw.com/

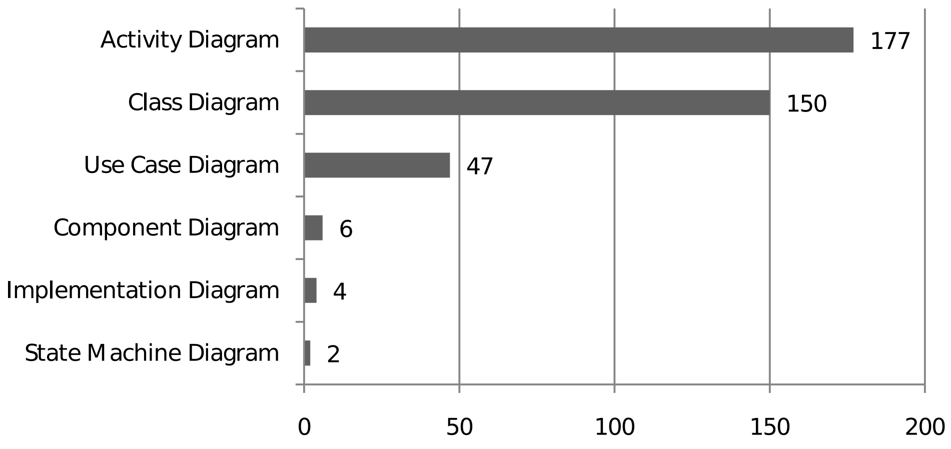


Figure 7.2: *Frequency of UML diagram type use*

activity and class diagram size metrics are presented in Table 7.1. Code size, measured in source lines of code, is an often-used metric to track progress in non-MDD projects. The organization that builds this software, usually estimates and tracks the lines of code as a means of tracking progress. In this case, model size metrics were used. A visualization of size metrics over time is depicted in Figure 7.3(a). As MDD was not formerly employed, the actual progress could not yet be benchmarked to other projects within the same organization for these specific model metrics. The average curve of the growth of metrics over time is plotted in Figure 7.3(b). As is the case with cumulative SLOC visualizations, a fairly gradual sigmoid curve can be discerned. However, the majority of model elements (approximately 72 percent) seems to have been created around development week 20, at approximately a third of the development process. In the same figure, a cumulative plot of revisions over time (as obtained from the repository log) is plotted over the same time period. From this plot, it can be deduced that the amount of revisions per week does not taper off after week 20. In fact, a slight increase can be observed. This implies that most model elements were already in place early in the development process and that model elements are mostly altered. This is consistent with the idea that MDD enables early prototyping and that the majority of development time can then be used for fine-tuning the implementation.

7.5.2 Model Complexity

Model complexity is defined by the sum of the complexity of the activity diagrams and the coupling of the class diagrams as they appear together in a single diagram group. Some complexity diagram metrics for class diagrams, such as *number of methods*,

Table 7.1: *Model Size Metrics*

ACTIVITY DIAGRAMS[1]	
Actions	The number of actions of the activity. Includes actions in all activity groups (partitions, interruptible regions, expansion regions, structured activities including conditional, loop, and sequence nodes), and their subgroups and sub-subgroups.
ObjectNodes	The number of object nodes of the activity. Counts data store, central buffer, and activity parameter nodes in all activity groups and their subgroups.
Pins	The number of pins on nodes of the activity. Counts all input, output, and value pins on all nodes and groups of the activity.
ControlNodes	The number of control nodes of the activity. Control nodes are initial, activity final, flow final, join, fork, decision, and merge nodes. The metric also counts control nodes in all activity groups and their subgroups.
Partitions	The number of activity partitions in the activity.
Groups	The number of activity groups or regions of the activity. Counts interruptible and expansion regions, structured activities, conditional, loop, and sequence nodes, at all levels of nesting.
CLASS DIAGRAMS[1]	
Classes	The number of classes on the diagram.
NumAttr	The number of attributes in the class. Also known as the Number of Variables per class (Lorenz and Kidd, 1994).

[1] source: SDMetrics 2.2 User Manual (Wüst, 2011)

were not applicable to the class diagrams designed by this project due to modeling conventions (no methods were used). Instead, we used coupling measures to denote class diagram complexity. Descriptions of both activity and class diagram complexity and coupling metrics are presented in Table 7.2. Diagram group complexity is defined as the average complexity per diagram type:

$$\text{complexity}_{\text{model}} = \frac{\text{complexity}_{\text{activity diagram}}}{\sum \text{diagrams}_{\text{activty}}} + \frac{\text{coupling}_{\text{class diagram}}}{\sum \text{diagrams}_{\text{class}}} \qquad (7.1)$$

Table 7.2: *Model Complexity Metrics*

ACTIVITY DIAGRAMS (COMPLEXITY)[1]	
ControlFlows	The number of control flows of the activity.
ObjectFlows	The number of object flows of the activity.
Guards	The number of guards defined on object and control flows of the activity.
CLASS DIAGRAMS (COUPLING)[1]	
Dep_Out	The number of elements on which this class depends.
Dep_In	The number of elements that depend on this class.
NumAssEl_ssc	The number of associated elements in the same namespace as the class.
NumAssEl_sb	The number of associated elements in the same scope branch as the class.
NumAssEl_nsb	The number of associated elements not in the same scope branch as the class.
EC_Attr	The number of times the class is externally used as attribute type. This is a version of OAEC+AAEC (Briand et al., 1999).
IC_Attr	The number of attributes in the class having another class or interface as their type. This is a version of OAIC+AAIC (Briand et al., 1999) and also known as Data Abstraction Coupling) (Li and Henry, 1993).
EC_Par	The number of times the class is externally used as parameter type. This is a version of OMEC+AMEC (Briand et al., 1999).

[1] source: SDMetrics 2.2 User Manual (Wüst, 2011)

As expected, there exists a positive correlation between diagram group size and average diagram size (Table 7.5.3). This implies that diagram groups that contain more diagrams also contain bigger diagrams. In addition, as diagram group size increases, the average complexity per diagram also increases. This means that certain diagram groups receive more attention than others and might imply that some diagram groups are more important than other models. Not surprisingly, the greater the average diagram size in a diagram group is, the greater the complexity of the diagrams becomes. This underlines our finding that larger diagram groups contain more complex diagrams.

7.5.3 Development Effort

In this section we elaborate on the effort data recorded for the case. The following sections contain an analysis of model development effort, development phase effort and model change effort.

Model Effort

The elaboration phase was executed in three large iterations. The construction phase is executed in many iterations that last one week each. The amount of effort spent per project phase is depicted in Figure 7.4(a). In the construction phase, about 40 percent of effort is spent on development of the models. The remaining effort is spent on the generator and coding. Of the 10,000 hours spent in the construction phase, 500 hours were spent on changes. The amount of effort spent on the models, and the effort types we could distinguish from the data are shown in Figure 7.4(b). As can be seen, a substantial amount of time is spent on adding functionality to the code generator. This effort is disregarded for the analysis of the effort spent on each diagram group. Interesting is that about 9 percent of the time is spent on issue resolution, and 2 percent is spent on changes. This is a relatively low amount of effort. Of all effort, 59 percent could be traced back to a specific diagram. The amount of effort spent on development or modeling does not correlate with model size. Only the effort spent on testing correlates with the amount of defects found. Analyzing the relation between model complexity and effort, we found that, the longer a diagram group is worked on, the more complex the activity diagrams are. Contrastingly, development time does not seem to be related to class diagram complexity.

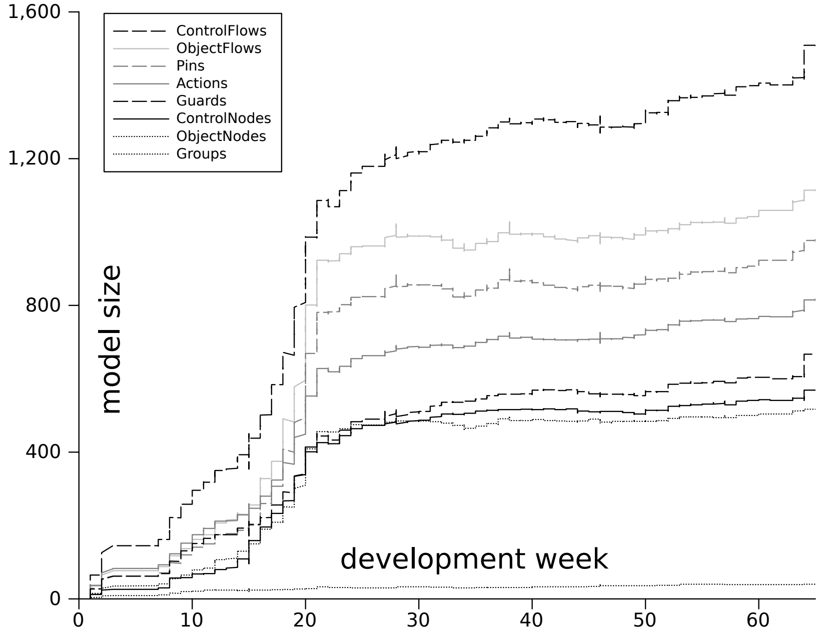
Phase Effort

We compared the effort spent per phase and the length of the phase to the averages of 17 RUP projects that were executed by the same organization (Figure 7.5). Some observations can be made regarding this visualization. First, the inception phase of MDD is quite similar to the other projects. This is most likely because the inception phase of an MDD project is not necessarily different from any other type of project. Second, during the elaboration phase, significantly more effort is spent. This is likely to be caused by a team size increase. Because MDD requires much modeling, more developers are needed at an earlier stage in the project. The team size increase that traditionally takes place at the start of the construction in this MDD project took place in the elaboration phase. Third, the elaboration phase lasted significantly longer. In the interviews, we found three explanations for this phenomenon:

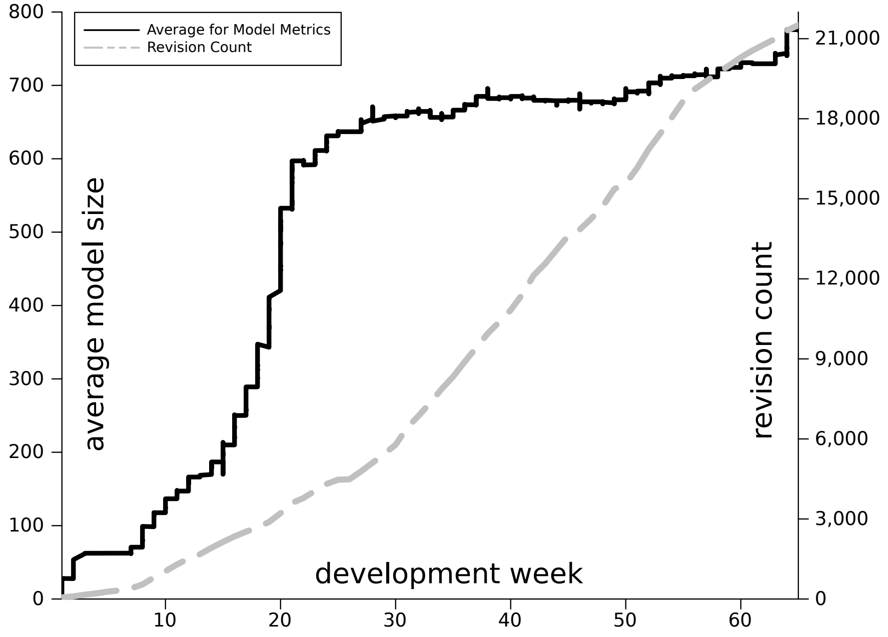
1. the initial design of many of the models is seen as a design activity rather than an implementation activity

2. the switch from the higher level target language to Java, which caused a delay

3. general learning effects of introducing MDD on a large scale

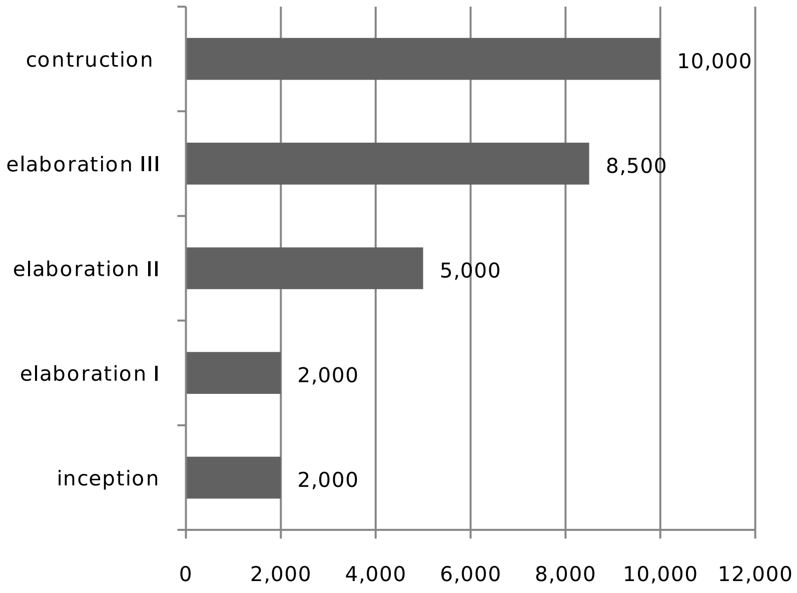


(a) Cumulative model size metrics over time

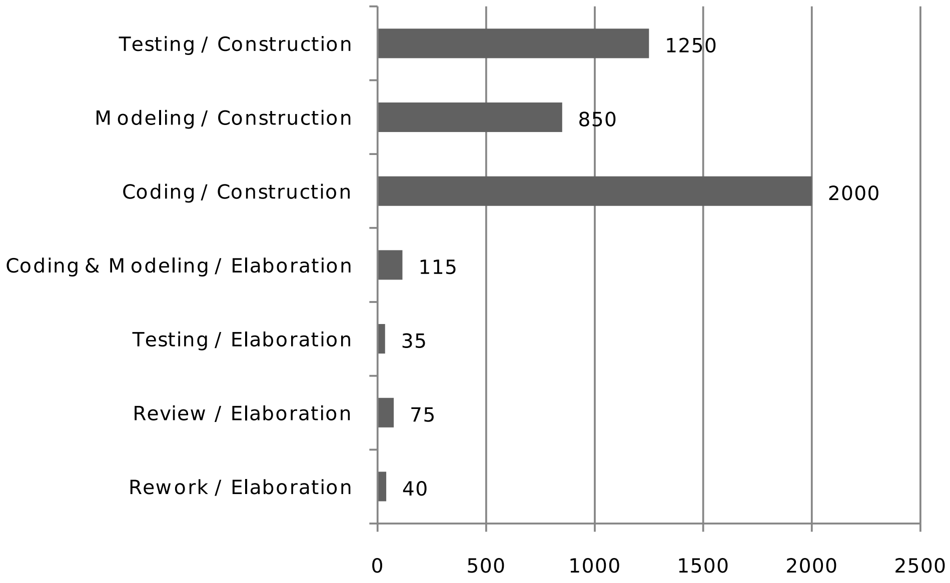


(b) Cumulative average model size metrics over time versus revisions over time

Figure 7.3: *Model metrics and revision count over time*



(a) Hours per development phase



(b) Effort related to model types

Figure 7.4: *Project effort distribution on phase and model level*

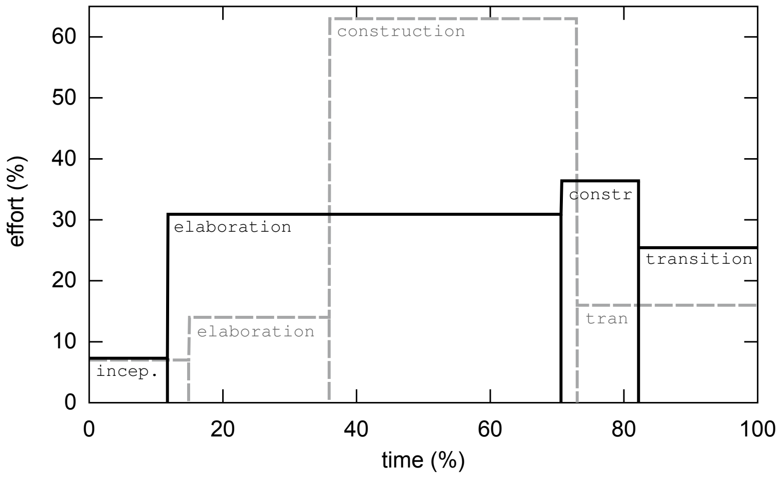


Figure 7.5: *Effort and duration per phase (normalized at 100 percent). The black line represents the case. The dotted gray line in the background represents the average for a set of 17 projects similar in size and complexity*

Table 7.3: *Bi-Variate Correlation Matrix for Common Process Metrics in MDD Context*

Spearman's Rank Correlation Coefficient (ρ)												
		defect count	defect priority	model size	diagram size	model complexity	class diagram size	activity diagram size	defect closing time	activity diagram complexity	number of diagrams	class coupling
defect count	ρ	1	0.176	0.258	0.188	0.103	0.189	0.227	0.294(*)	0.348(**)	0.226	0.093
	p	.	0.191	0.052	0.166	0.452	0.518	0.095	0.026	0.009	0.090	0.753
	N	57	57	57	56	56	14	55	57	55	57	14
defect priority	ρ	0.176	1	0.118	−0.019	0.012	−0.336	−0.020	0.289(*)	−0.032	0.169	−0.440
	p	0.191	.	0.382	0.891	0.928	0.240	0.885	0.029	0.817	0.209	0.115
	N	57	57	57	56	56	14	55	57	55	57	14
model size	ρ	0.258	0.118	1	0.364(**)	0.398(**)	0.428(*)	0.918(**)	−0.075	0.850(**)	0.738(**)	0.613(**)
	p	0.052	0.382	.	0	0	0.023	0	0.577	0	0	0.001
	N	57	57	119	103	103	28	103	57	103	119	28
diagram size	ρ	0.188	−0.019	0.364(**)	1	0.947(**)	−0.175	0.518(**)	−0.021	0.599(**)	−0.451(**)	0.181
	p	0.166	0.891	0	.	0	0.374	0	0.876	0	0	0.356
	N	56	56	103	103	103	28	102	56	102	103	28
model complexity	ρ	0.103	0.012	0.398(**)	0.947(**)	1	−0.190	0.553(**)	−0.064	0.545(**)	−0.456(**)	0.142
	p	0.452	0.928	0	0	.	0.334	0	0.639	0	0	0.470
	N	56	56	103	103	103	28	102	56	102	103	28
class diagram size	ρ	0.189	−0.336	0.428(*)	−0.175	−0.190	1	0.082	0.164	0.158	0.670(**)	0.641(**)
	p	0.518	0.240	0.023	0.374	0.334	.	0.680	0.575	0.421	0	0
	N	14	14	28	28	28	28	28	14	28	28	28

* Correlation is significant at α = 0.05 / ** Correlation is significant at α = 0.01

(continued on next page...)

Spearman's Rank Correlation Coefficient (ρ) (continued)

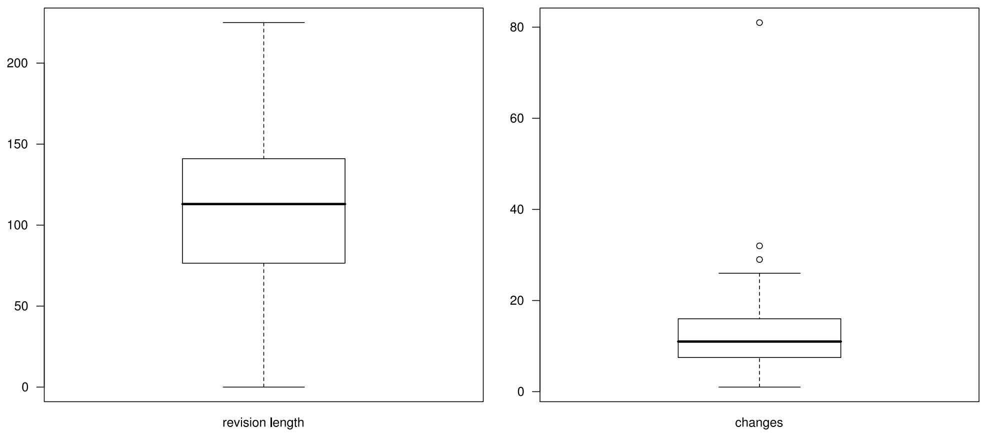
		defect count	defect priority	model size	diagram size	model complexity	class diagram size	activity diagram size	defect closing time	activity diagram complexity	number of diagrams	class coupling
activity diagram size	ρ	0.227	−0.020	0.918(**)	0.518(**)	0.553(**)	0.082	1	−0.117	0.940(**)	0.391(**)	0.332
	p	0.095	0.885	0	0	0	0.680	.	0.395	0	0	0.085
	N	55	55	103	102	102	28	103	55	103	103	28
defect closing time	ρ	0.294(*)	0.289(*)	−0.075	−0.021	−0.064	0.164	−0.117	1	−0.009	0.010	−0.065
	p	0.026	0.029	0.577	0.876	0.639	0.575	0.395	.	0.945	0.939	0.826
	N	57	57	57	56	56	14	55	57	55	57	14
act. diag. complex.	ρ	0.348(**)	−0.032	0.850(**)	0.599(**)	0.545(**)	0.158	0.940(**)	−0.009	1	0.347(**)	0.273
	p	0.009	0.817	0	0	0	0.421	0	0.945	.	0	0.159
	N	55	55	103	102	102	28	103	55	103	103	28
number of diag.s	ρ	0.226	0.169	0.738(**)	- 0.451(**)	- 0.456(**)	0.670(**)	0.391(**)	0.010	0.347(**)	1	0.405(*)
	p	0.090	0.209	0	0	0	0	0	0.939	0	.	0.032
	N	57	57	119	103	103	28	103	57	103	119	28
class coupling	ρ	0.093	−0.440	0.613(**)	0.181	0.142	0.641(**)	0.332	−0.065	0.273	0.405(*)	1
	p	0.753	0.115	0.001	0.356	0.470	0	0.085	0.826	0.159	0.032	.
	N	14	14	28	28	28	28	28	14	28	28	28

* Correlation is significant at $\alpha = 0.05$ / ** Correlation is significant at $\alpha = 0.01$

Table 7.4: *Bi-Variate Correlation Matrix for Defect Priority and Changes*

Spearman's Rank Correlation Coefficient (ρ)														
		devel. time	changes	number of diagrams	model size	diagram size	model complexity	defect closing time	defect count	defect priority	activity diagram size	activity diagram complexity	class diagram size	class coupling
devel. time	ρ	1	0.769(**)	0.378(**)	0.473(**)	0.215(*)	0.142	0.189	0.653(**)	0.162	0.366(**)	0.440(**)	0.101	0.308
	p	.	0	0	0	0.034	0.164	0.158	0	0.229	0	0	0.648	0.152
	N	107	107	107	107	98	98	57	57	57	97	97	23	23
changes	ρ	0.769(**)	1	0.614(**)	0.634(**)	0.078	0.013	0.138	0.493(**)	0.132	0.451(**)	0.489(**)	0.121	0.544(**)
	p	0	.	0	0	0.445	0.899	0.306	0	0.328	0	0	0.581	0.007
	N	107	107	107	107	98	98	57	57	57	97	97	23	23

* Correlation is significant at $\alpha = 0.05$ / ** Correlation is significant at $\alpha = 0.01$



(a) Revision length (days)   (b) Changes per model

Figure 7.6: *Software configuration and change management system usage*

Change Effort

The amount of changes per diagram were measured by the amount of version updates found in Subversion that were directly related to that diagram. On average, a model had 12.6 versions associated with it. A total of 1,308 change commits to the Subversion repository were associated with a model out of a grand total of 9,035 commits (14.5 percent). The reason for the substantial difference between model-related and non-model-related commits is that the repository contains all documentation regarding the project including status reports and other kinds of management specific files. The files are altered, and subsequently checked-in, frequently. Also, the amount of development time per model was measured as the difference between the dates of the first and the last model related change, measured in days. A summary of the measurements for revision length is depicted in Figure 7.6(a). The average amount of calendar days during which a model was revised was 111. The total amount of days during which all models were altered is 230 days.

7.5.4 Defects and Changes

Defects were stored in a centralized defect tracking system. All team members were able to add defects to the database. Per defect, a unique id, title and description were recorded as well as a priority. Furthermore, defect submission time, closing time and the last modified time were recorded. Lastly, defect status (Assigned, Closed, . . .) and defect type were recorded. Six different defect types were used, namely: defects related to deployment, development, generation, modeling, requirements and testing.

A total of 631 defects was registered. Of these defects, 81 percent was directly related to a model. A total of 80 models (or 68.4 percent) had one or more defects associated with them. In this subset, on average, 6.4 defects were found per model. These are pre-release defects. At defect submission time, a defect priority is assigned to the defect report on a scale of 1 (high priority) to 5 (low priority). The mean priority of the defects related to a model is 1.9 whereas the mean priority for a defect that is not directly related to a model is 2.35. This indicates that it is generally seen as more important to solve defects related to a model than to resolve defects that are not related to a model – underlining model centrality. The defects that are not related to a model mainly have to do with the code generator.

We find that the defect count per model positively correlates with defect closing time (Table 7.4). This implies that models with a relatively larger amount of defects, have a higher average defect repair time. This is an intuitive finding as an increase in the number of defects in a single model or diagram can increase the complexity of the repair process and thereby delay a fix.

The finding that models with a relatively larger amount of defects, have a higher average defect repair time is in line with maintenance for source code. A counter-intuitive finding is that while both development time and the amount of changes correlate positively with model size but that model size did not correlate with defect count. This leads us to conclude that larger models are changed more often and worked on longer but do not necessarily contain more defects. However, models that are changed often do contain more defects. The reason for this relation could be that fixing a defect induces extra changes. However, the reverse could also be true, namely that models changed more often contain more defects as a result of an increased amount of changes.

The earlier finding that certain diagram groups receive more attention than others and might imply that some diagram groups are more important than other models is not confirmed by the average defect priority of the diagram because it does not correlate with model size. Bigger models do not contain defects that, on average, are seen as more pressing to resolve. Also, model size does not correlate with defect closing time. We expected a negative correlation between these two variables because larger models are more complex and this could adversely impact the time needed to repair a defect.

Furthermore, bigger models do not contain more defects. While appearing counter-intuitive, this is in line with the Theory of Relative Defect Proneness (Koru et al., 2009) (recently confirmed for closed-source software; Koru et al., 2010).

7.5.5 Defect Discovery

We plotted the cumulative defects found over normalized time for the case (labeled "Project Alpha") and 10 projects that were executed by the same IT organization (Figure 7.7) to enable visual comparison. The lines were smoothed using a Bezier algorithm for readability purposes. In the image, we see a clear difference between the case

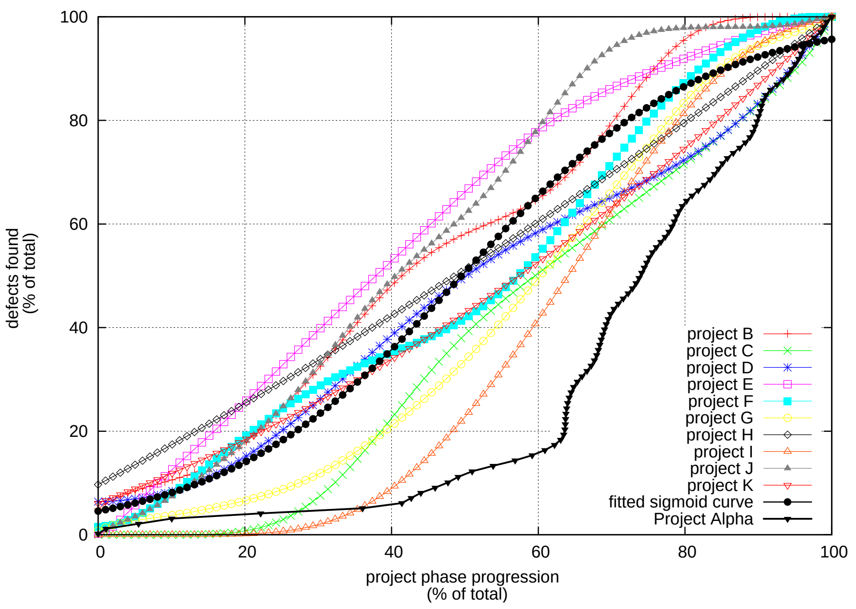


Figure 7.7: *Cumulative defect discovery over time (case is labeled "Project Alpha")*

(Project Alpha in the graph) and the other projects. For example, when 40 percent of the project time has passed only around 10 percent of the defects are reported whereas, on average for conventional projects, 35 percent of defects were found. We found three reasons for this relative slow defect discovery rate. First, ineffectivity in finding defects due to a learning curve that interviewees associate with introducing MDD. Second, because MDD was applied, much effort was spent in creating the models and the generator at first, only later, when the hand coded part of the system was developed, were real defects reported. Initial defects mostly involved the code generator. Third, the standard quality assurance process was not yet tailored for MDD. Finding defects in later stages in a project is commonly regarded as undesirable.

However, the amount of defects found per function point is equal to or less than[8] 0.32. The amount of function points has increased since the initial function point analysis that was executed before the project started development, due to change requests. The average of defects found per function point for 22 similar sized, non-MDD development projects that were executed at the same organization is 0.52. This implies that for projects on average approximately 396 less defects are reported for MDD. This is a drastic decrease in the amount of defects found. A possible explanation

[8]We use the initial functional point count for this calculation. The amount of function points that is implemented is expected to be higher than the initial functional point count.

is the (much) larger proportion of the development effort that is spent on model improvement compared to model build-up.

7.6 Conclusions and Future Work

The main objective for this study was to report on the specific characteristics of a large scale, industrial MDD project and to asses what the impact was of using MDD tools and techniques compared to non-MDD development.

Adopting MDD tools and techniques fundamentally impacts the software development process in general and the analysis and design phases in particular. Because almost all code was directly generated from diagrams, models were first-class citizens (France and Rumpe, 2007, Balasubramanian et al., 2006) in the software development process followed in this case.

Three striking differences in the development process were found. First, 59 percent of all effort was spent on developing the model. That is significantly more than the time spent on code development in classical software development. Second, most model elements were already implemented at one third of the development process. The remaining development time was spent on altering the models. Third, 40 percent fewer defects were found when compared to projects of similar size.

Diagrams and code are fundamentally different and therefore not easily compared as we found absent, for example, a positive relation between model size and complexity on the one hand and defects on the other — relations that have often been observed in source code. Also, larger diagrams were changed more often and worked on longer but did not necessarily contain more defects.

Chapter 8

Analysis of the Consequences of Model-Driven Development for Global Software Development

The promotion of models over code to first-class entities is a central theme of Model-Driven Development (MDD). In theory, this has a profound impact on the architectural process and the work of the software architect. MDD is emergent in GSD projects and the main challenges in GSD include difficulties to share knowledge, to align tasks and to obtain and maintain a shared mental model. In theory, MDD has the potential to mitigate some of these difficulties. In this chapter we aim to understand (1) how the application of MDD tools and techniques affects the architectural process and (2) how this relates to the problems commonly associated with GSD.

This chapter is based on the following publication:

Werner Heijstek and Michel R. V. Chaudron (2010) **The Impact of Model-Driven Development on the Software Architecture Process.** *In Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications (SEAA 2010) pages 333–341, Lille, France*

8.1 Introduction

Recent studies focus on decision making in the process of software architecting (e.g. Kruchten et al., 2005). As a result, the practice of software architecting and the position of the software architect in the software development process are under investigation (Clements et al., 2007, Farenhorst et al., 2009). Also in Global Software Development (GSD) projects, where software architecture is often transferred from one development location to the other, the role of the software architect is not clearly defined. In distributed settings, software architects have to interact with software developers through (often great) geographical, temporal and socio-cultural distances. So far, it has been unclear how these challenges are coped with in industrial practice.

At the same time, MDD tools and techniques claim improved team communication, better (or even automatic) architecture compliance and resulting productivity gains. Application of these tools demands a shift in boundaries between traditional roles in the software development process as the code is no longer the central artifact – the model is. This would have a profound impact on the architectural process and the work of the software architect who has to work with a different set of tools, a different vocabulary, a different type of development team. The architect also has an increased responsibility to maintain consistency throughout the development process.

MDD is emergent in GSD projects (Jiménez et al., 2009). The main challenges in GSD include sharing knowledge, to align tasks and to obtain and maintain a shared mental model (Cannon-Bowers et al., 2001, Espinosa et al., 2001). In theory, MDD has the potential to mitigate some of these difficulties. Using models as the central artifact enables teams to use software architecture and design to direct team composition, development process and even communication structures. In this chapter, we aim to understand (1) how the use of MDD tools and techniques affects the architectural process and (2) how these impact the problems commonly associated with GSD.

The outline of this chapter is as follows: Section 8.2 contains an overview of the study objective and the data collection and analysis methods. Section 8.3 outlines related work. Sections 8.4 and 8.5 discuss the results and the specific impact of MDD on GSD. Finally, Section 8.6 contains conclusions and future work.

8.2 Objectives and Data collection and Analysis Methods

In this chapter, we address **RQ3** (Section 1.3). One of the main implications that Šmite et al. list in their recent structured literature review of empirical evidence in GSD is that there exists a *"gap regarding in-depth empirical investigations addressing particular aspects of software engineering."* They continue to note that *"thus, future research ought to evaluate different practices, methods and techniques rather than mainly focus on managerial problem-oriented lessons learned."* We have motivated that existing studies hint at the positive influence that MDD could have on the GSD process. Therefore, it is not only necessary to understand what the impact is of using MDD tools and techniques in the context of GSD in general. Specifically, we will address the impact in the context of the problems associated with GSD. The research question addressed in this chapter can therefore be formulated as follows:

> *How does the application of model-driven development tools and techniques impact the problems associated with Global Software Development?*

The subject of our analysis is the case outlined in Chapter 7. To analyze our data, we apply the grounded theory approach. We used semi-structured interviews to survey a subset of the project team members. We interviewed both the project manager and lead architect before, during and after the project. The lead architect was interviewed extensively six times over the course of two years. In the first interviews, the structure of the project and the approach were discussed. The high amount of interviews was needed because the contracting organization was not used to manage MDD projects and therefore had limited insight in the approach and progress of the project. We also extensively interviewed other team members including designers, developers, lead developers, project leaders, a test manager, a system analyst and an estimation & measurement officer involved in sizing and tracking the project.

During the interviews we asked questions regarding the impact of the application of MDD on the activities of the participant and on the process of software development in general. All questions were directed at (1) identifying every possible architectural process-related differences with a non-MDD project and (2) finding all possible confounding factors. An audio-recording was made of all interviews. The next step involved transcribing and coding the audio recordings. All separate statements made by the subject were collected in a list. We then marked each statement that related to MDD. After this initial coding process, we grouped the statements and identified (formulated) a common impact factor that best described all statements in one group. We then removed and merged duplicate and overlapping impact factors and we established whether the impact was either (1) caused by the application of MDD, (2) the cause of MDD and other, non-MDD, factors or (3) most likely not the cause of MDD. We then confronted the interview participants with these distilled lists to validate our interpretations. We repeated the coding process and updated factor descriptions in the same way.

8.3 Related Work

This section addresses related work regarding the generic impact of MDD on the software architecture process and the (potential) benefits of application of MDD in the context of GSD.

8.3.1 General Impact on the Software Architecture Process

A commonly adopted framework for MDD is Model-Driven Architecture (MDA, Kleppe et al., 2003, Object Management Group, 2003b,a). MDA is a specific MDD approach that employs UML, MOF (Object Management Group, 2006) and XMI (Object Management Group, 2007). A study by The Middleware Company (2003) of the application of MDA specifically mentions *"architectural advantages."* The study explains that by application of MDA, an architect is forced to spend more time designing an architecture due to the necessity to also model high-level domain entities. The Middleware Company argues that increased upfront design effort reduces *"the possibility of introducing architectural flaws into your system later in the development life cycle."* The study further reports on an experiment in which the same application is developed by two teams of developers. One team applies MDA and one team does not. The MDA team finished their development ahead of schedule and significantly faster. Advantages of the application of MDD reported, include increased ease of communication of the design (including to the client) and consistency between design and code. Both are closely related to the core activities of a software architect. According to a survey by Staron (2006), the main aims for adopters of MDD were: improving quality by increasing understanding, improving communication within development team and traceability throughout software development artifacts (models). These three expected benefits are directly related to the responsibilities of a software architect. Application of MDD is therefore expected to alter the role of the software architect. Farenhorst et al. (2009) list five categories of architecting activities. At least three categories are expected to be impacted by adopting MDD. First, communication of architecture design is expected to be easier because models are the dominant artifact throughout the project. Second, quality assessment will likely be more important because of the more formal nature of MDD. Lastly, a stronger focus is expected on documentation due to the use of a Domain-Specific Language (DSL). A DSL is a modeling language that, by design, is particularly fit to express concepts related to a specific field or e.g. a branch of business.

8.3.2 MDD in Global Software Development

On the surface, MDD appears to have the potential to mitigate some of the difficulties that are associated with GSD. Using models as the central artifact would enable teams to use software architecture and design to direct team composition, development process and even communication structures. The main challenges in GSD include sharing knowledge, to align tasks and to obtain and maintain a shared mental model (Cannon-Bowers et al., 2001, Espinosa et al., 2001). The sources of these challenges are to be found in the three types of distance that are introduced in GSD projects: geographical distance, temporal distance and socio-cultural distance.

The model-centric nature of MDD and some of the requirements that lie at its foundation could have a positive influence on the problems associated with GSD.

Limited related work exists that explicitly addresses this conjecture. Nevertheless, evidence of the potential positive influence of MDD on GSD can be found in various studies. For example, that the higher level of abstraction inherent to MDD facilitates more effective stakeholder communication is explicitly mentioned in the context of GSD in many other studies. For example:

- *"explicit, shareable models and descriptions [...] facilitate collaboration between developers on an abstract level"* (Pahl, 2005),
- *" great advantages of [our MDD language] in the context of global software development [include] a common understanding of the software being developed"* (Heistracher et al., 2006),
- *"advantages of a unified, model-driven approach to requirements elicitation include significantly improved communication"* (Berenbach and Gall, 2006).

In addition, the advantages of the use of common tools on GSD is often referred to. For example:

- *"[MDD] tool vendors develop more and more tools to be applied during architecture development."* (Spanjers et al., 2006);
- *"Enforcing common tools and processes makes collaboration much easier "* (Lings et al., 2007);

Only limited more detailed evidence is available. For example, Lester and Wilkie (2004) present an empirical evaluation of the selection of a commercial CASE tool that supports UML in the context of a large GSD project. Lester and Wilkie specifically aim to address the problem that *"the lack of synchronization between design models and source code, for a development team working in different time zones, can lead to strained relationships between the geographically disparate sites."* MDD brought forth a host of tools that support model-code correspondence in general and code generation in particular.

Another example of more detailed analysis of the role of MDD-related tools and techniques is the work by Clerc et al. (2007). In this study, a case is reported in which distributed team organization was formed strictly along the lines of an architectural design — including dependency and subsystem-related constraints on communication. The authors claim that this helped GSD-related problems they define as *"difficulty to build a team."*

Application of MDD requires the early and complete definition of an architectural framework. This is in line with the requirement that an architecture must be sufficiently mature to be able to distribute team composition, development process and communication structures (Mullick et al., 2006, Conway, 1968). In line with the previous finding, Clerc et al. (2007) also found that *"alignment via architecture"* is beneficial. They found that in the same case, *"alignment of tasks and responsibilities [was] mainly done via the architecture,"* they go on to exemplify this by noting that, *"[r]equirements [were]* *assigned to subsystems, which [had] dedicated resources assigned."* Apart from a similar requirement regarding early architecture maturity, this model-centric method of project management is much in line with the premise of MDD in which models are even more central.

Another specific study of benefits of MDD techniques to GSD problems was executed by Andaloussi and Braun (2006) who outline their experiences in developing a model-driven test framework based on the the UML 2 testing profile (Schieferdecker et al., 2003). In their effort, Andaloussi and Braun specifically sought to obtain communication benefits for GSD teams: *"The advantage is to represent the system and its tests through one single notation."* In preliminary findings of a case study in which they implemented their framework, they found that through using the test framework they *"[overcame] the language barriers in releasing the test specification from [a] textual description filled with buzzwords and jargon."* In addition, they note that they made *"nearshore more independent from offshore, in avoiding initial training phases and requiring only standard skills (UML, U2TP and TTCN-3)."* Their use of a DSL made it easier to distribute the architecture in small components, which in turn increased comprehensibility.

8.4 Results

In this section, the results of the analysis of the case (as outlined in Chapter 7) are discussed. Structured around the three core concepts behind MDD (Section 1.1.5), we will discuss the influences of application of MDD on the GSD that were found in the project under study. We will particularly address the ramifications for the software architecture process. An overview of the implications of adoption of MDD in GSD on the software architecture process is presented in Figure 8.1.

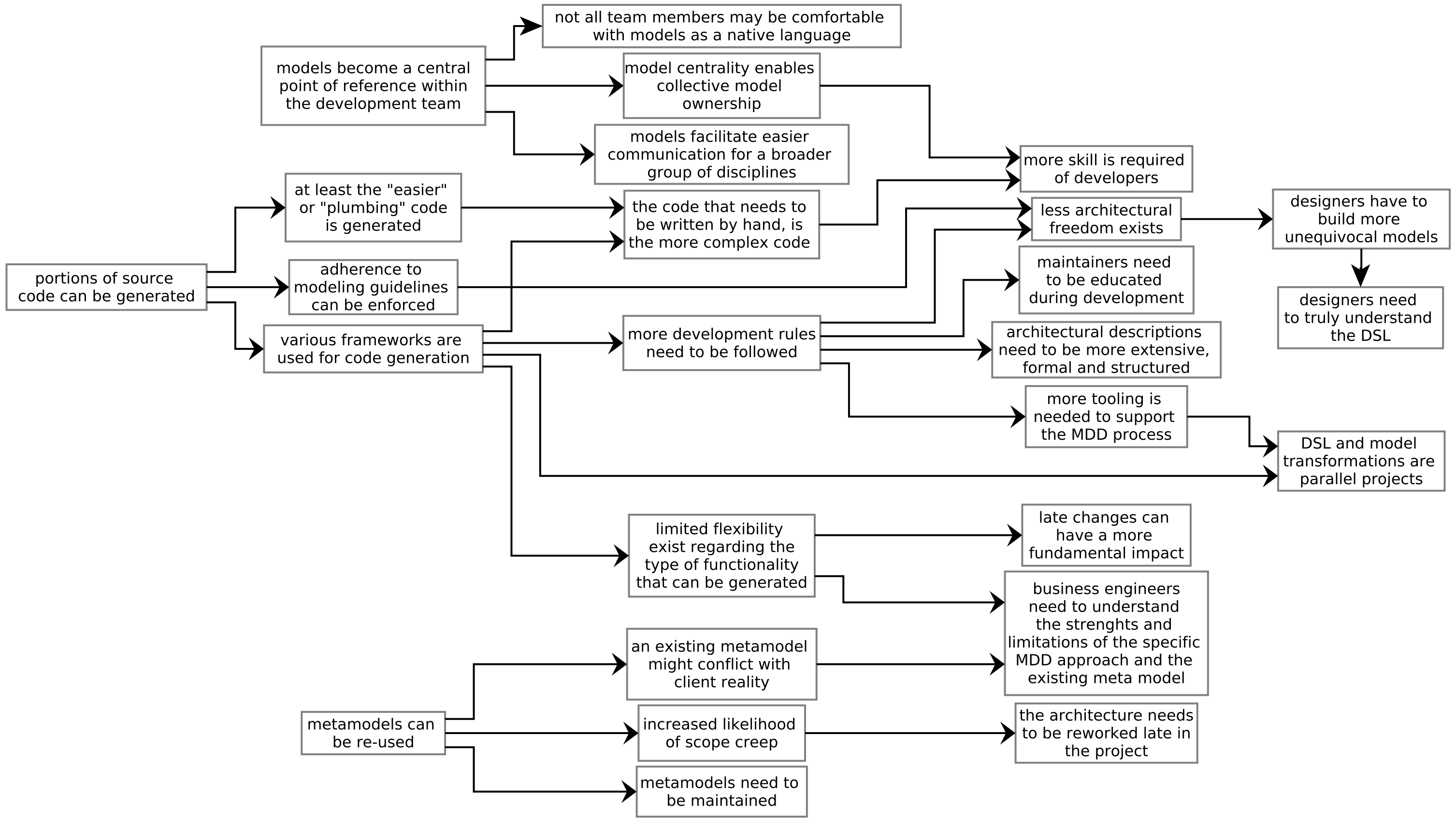

not all team members may be comfortable with models as a native language
models become a central point of reference within the development team
model centrality enables collective model ownership
models facilitate easier communication for a broader group of disciplines
more skill is required of developers
at least the "easier" or "plumbing" code is generated
the code that needs to be written by hand, is the more complex code
less architectural freedom exists
designers have to build more unequivocal models
portions of source code can be generated
adherence to modeling guidelines can be enforced
maintainers need to be educated during development
designers need to truly understand the DSL
various frameworks are used for code generation
more development rules need to be followed
architectural descriptions need to be more extensive, formal and structured
more tooling is needed to support the MDD process
DSL and model transformations are parallel projects
limited flexibility exist regarding the type of functionality that can be generated
late changes can have a more fundamental impact
business engineers need to understand the strenghts and limitations of the specific MDD approach and the existing meta model
an existing metamodel might conflict with client reality
metamodels can be re-used
increased likelihood of scope creep
the architecture needs to be reworked late in the project
metamodels need to be maintained

Figure 8.1: *Factor integration graph (⟶ denotes cause and effect)*

8.4.1 Model-Centrism

Source code is not easy to communicate even between those who understand the programming language. In addition, source code represents one of many representations of the system to be built. In traditional software development, each stakeholder has his own preferred representational conventions to describe the system. In MDD, models, rather than code, are treated as first-class entities. As models are abstractions of more technical details, the potential proportion of team members that understand the models is larger.

All team members noted that intra-team communication was much easier because the models were used as a single point of reference was used. Models were numbered and requirements engineers, the architect and even project management would refer to that same model number to discuss an particular issue that was relevant to their role. As a result, use of models as a common language eases communication and enables a larger group of stakeholders to participate in implementation-related discussions. The strongest evidence exist in literature for this particular effect of application of MDD.

A substantial amount of the architectural process is spend on communication of design and architectural decisions. By introducing models as a common language that is used throughout the development process, this time-consuming undertaking becomes less laborious. The project team members that were interviewed acknowledge that technical discussions related to aspects of the system were easier to conduct than they were used to in non-MDD projects. Reasons consistently mentioned for the discussion benefits were that the models could be used as a basis for discussion and because more different team members of different disciplines were familiar with the models. This translated to fewer traveling back and forth between the offshore and onshore locations than normally would be the case in projects of similar size and complexity.

In the following sections we discuss two other effects that model-centrality had on the case. First, the transition to models as a common language is easier for some disciplines than for others. Particularly, business analysis were reluctant to work with software CASE tools. Second, as more stakeholders are directly involved with the models (now the central development entities), "collective ownership" becomes an important development concept.

Common Language as a Challenge

In the case, a group of requirement engineers were only willing to participate in modeling their use-cases more formally on the condition that they would not be concerned with what they referred to as "programming." However, the models they made were directly generated to code, code that would be directly used in the system and which would become the vast majority of the final code of which the system would be comprised. Another example are a group of business analysts who refused to work with a UML case tool as they regarded it "technical work". They eventually left the project.

The technical possibilities and potential advantages in terms of synergy offered by consistent use of diagrams from early requirement workshops to the generation of a working software system are evident. Much is lost and misunderstood in translation of requirements to architecture to design to source code. And while MDD does not completely remove the need for translation, the use of models as a central language at least limits it. In practice, this implies that more team members need to be able to communicate in a common language. Nevertheless, a clear distinction between business related activities and IT-related activities is still often made in software engineering practice. Requirements engineers do not write source code, developers do not bother with domain models and a project manager might not be up to speed with specific testing techniques employed. However, the central use of models requires team members from all disciplines to work with the same language, concepts and tools.

Collective Model Ownership

The notion of "collective ownership" stems from the rules of Extreme Programming ((XP), Beck, 1999). It encompasses the notion that team members are collectively responsible for various aspects of the system under development, specifically system design. The key benefit that this practice aspires to obtain is the elimination of bottlenecks for changes to certain aspects of the system. In addition, people that are bottle necks may leave a project at any time, taking with them valuable information regarding an aspect of the system only they had deep understanding of. Developers tend to prefer to be responsible for their part of the system. Because less code is written and this code is more complex, developers must more often work with code that they did not author themselves. It is not possible to couple a developer to a particular use case. In fact, no developer should have objections to working at another use case or to have somebody change code related to a use case they initially authored.

The architect explained that all designers need to be able to develop and expand most models. While some of the more complex models were still assigned to one or two designers, the majority of models were worked on by a variety of team members. Various team members explained that they enjoyed working on various different models. They were also convinced that model quality improved because of this practice. Having more team members working at the same model increased the chance of spotting defects. Developers explained that they were not used to work with each others code and that they needed to get used to having other people work with "their" models and code.

An additional benefit of collective code ownership is that it facilitates increased contact between the programmer and the designer. However, the diagrams were never branched so enable that designers could work in parallel. In this case, no tool support existed for model version control. Although some tools are available (e.g. EMF

Compare (Brun and Pierantonio, 2008) and DSMDiff (Lin et al., 2007)), these are not easily integrated with existing MDD tools.

8.4.2 Code Generation

In this case, a source code generator was developed to generate code. Using this generator, a significant portions of source code (90 percent) of source code was generated. We found this to have four direct consequences:

First, at least all the "easy" code is generated. All of the "hand-written" code, therefore, is more difficult to write. This requires more skilled developers. Second, standardization on mature software components as well as integrating generated code and hand-written code, requires the use of a variety of frameworks. Increasing the amount of frameworks involved in the software architecture requires an increasing understanding of the complexity of the interaction between these frameworks. This requires even more skilled programmers. Third, modeling can no longer be done haphazardly (Lange et al., 2006). On the contrary, adherence to modeling guidelines must be enforced if models are to serve as the basis for code generation. Fourth, if code is generated then a code generator needs to be developed and maintained in parallel with the original project. The generator is a separate project with its own stakeholders.

These four consequences lead to a variety of implications. The first and second consequence directly require more skilled developers. The use of a variety of frameworks and a code generator require more structured models, a more formal development process, documentation and an increase in tooling. The use of these frameworks also limits the flexibility regarding the type of functionality that can be generated. In addition, developer compliance to architectural rules is no longer optional. Furthermore, the increase in development rules require software maintainers to be involved in the implementation process at an earlier stage than would have been the case in a traditional development process. The fourth implication, the generator being a separate project, greatly increases project complexity. These implications are discussed in the following sections.

More Skilled Developers are Required

During the project, several junior developers were not able to cope with the complexity of the code that had to be developed. These developers had to be replaced by more capable or experienced developers.

Aspects of a system that lend themselves particularly well for code generation are data related constructs such as CRUD[1]-functionality. Much of the more straightforward code has therefore already been generated. In addition, to enable code generation and to attain this level of abstraction, a substantial set of frameworks is used. Understanding how these framework interact can be a difficult process. As a result, not

[1] Create, Read, Update and Delete

all developers that would normally work on implementation of a system of similar complexity are able to cope with the more complex use cases or exceptions. In short, highly skilled developers are needed to implement the more complex parts — which form the majority of the "hand-work."

An architect is responsible for communicating the more complex build-up of frameworks that is chosen for MDD development and therefore has spend more time to train new developers and to evaluate whether they are up to the task.

Strict Quality Assurance for Modeling

Developers explained they had less freedom to interpret designs and architectural constraints due to the central role of the models and the strict guidelines that needed to be adhered to in order to guarantee the system could be generated correctly.

Tools that enforce adherence to architectural rules are not commonly used in industrial practice. For an architect it is therefore important to check architecture adherence throughout the development process. In MDD, adherence to architectural rules takes less effort because (1) modeling is done more formally, (2) architecture is more formally defined and thus easier adhered to and (3) less steps of translation take place as models are directly translated to code by a code generator. In addition, the code generator used in this project was equipped with a model validator, the model equivalent of a code parser. This validator checks syntactical adherence and provides some level of quality check. In the case, model verification was done by the architect.

More Extensive and Structured Architectural Descriptions are Required

The set of architectural artifacts used in the project is larger and more detailed than found in similar (non-MDD) projects of equal size. The sources of architectural knowledge available during the project are detailed in Table 8.1.

Next to the sources in Table 8.1, architectural knowledge exists which is not captured in any artifact but the system itself. In interviews, project members refer to design decisions which are visible in the models but which are not explicitly documented. Project management did not allow for the time to explicitly document all design decisions due to time constraints.

Since more detailed descriptions of use cases are required in early stages of the project, documentation is reviewed more often. The central role of a document such as the modeling guidelines implies that more team members use and comment on contents. This requires more formal and complete descriptions which is better structured. Architectural documentation in the project was updated more frequently and up until later stages in the process in comparison to non-MDD projects.

Table 8.1: *Architectural Artifacts Available in the Case Project*

artifact	description	contents
Software Architecture Document (SAD)	The most important architectural knowledge is described in this document. The SAD is used by all team members except the testers. The author is the software architect.	Description of actors and development tools; List of architecturally significant use cases and their realizations (use case view); Overview of logical layers (logical view); Definition of communication and process principles that are relevant for the software architecture (process view); Description of the distribution of the system over various nodes and its interaction with a selection of surrounding systems (deployment view); Architecture of the source code — layering, frameworks and best practices (implementation view); Description of transformation of the UML design model to various data aspects of the software architecture (data view)
Supplementary Specification (SS)	Requirements outside of the requirements described in the use cases.	Quality requirements of interfaces; Additional system requirements
Interface Documentation	Per interface documentation. The author is the system analyst.	Request message specification; Reply message specification; Web Services Description Language (WSDL) specifications; List of related Use Cases
Modeling Guidelines	A tool-independent description of how to describe functional requirements of an IT system using UML. The author is the software architect.	Naming and ordering of model elements; Data modeling guidelines; User interaction modeling guidelines (flows, sub flows, authorization, use of data, use of services, decisions, constraints, composite operations); UML Profile Reference
Wiki	The Wiki of SourceForge Enterprise Edition 4.4 is used. Authors include most project members.	Tips and tricks to set-up your code environment correctly and how to solve problems; An overview of the release cycles of all parallel working teams is managed; An overview of how to work with the release process
Separate Model Documentation	A set of documents which elaborate on some of the elements from the meta-model that are only used in specific models. The authors are the maintainers of the respective models.	Style guide for the screens associated with this use case; Data sources overview for use case
Design Decisions	Elaboration of certain design decisions. This document is based on the modeling guidelines and was created before the system was built. The author is the System analyst.	How deep packages are nested; How certain functionality is split up

More Tooling Is Needed to Support the MDD Process

From their literature review of MDD, Mohagheghi and Dehlen (2008) conclude that suitable tools are of fundamental importance for MDD to succeed. These tools must be selected carefully for fitness to meet requirements and must fit into an organization's existing chain of tools. Factors in deciding on tooling for software development include a trade-off between the standard tooling used by the development organization, specific project requirements and the wishes of the client and possibly the maintenance organization. Primarily responsible for this process is the software architect. The use of MDA requires more tooling than a non-MDD development process. As in most development processes, an MDD project requires a configuration and change management system, requirement-, defect-, time- and change tracking systems and modeling-, development- and testing environments. However, a set of requirements are added to the tool selection process for supporting the DSL or reference model and extra environment for supporting the generator.

In addition to selecting candidate case tools and evaluation, team members must be trained to work with new tools. Traditionally, an architect will prescribe the use of only a subset of the functionality offered by the tooling, limit the use of the tool. Team member's use of the tooling must therefore be monitored. As described earlier, the project must deal with team members resisting to use particular tooling and perhaps needs to convince the client that a lesser known tool is indeed a proper solution. Finally, one of the lessons learned from the case is that adopting the use of an existing code generator for large-scale application of MDD is not feasible for large scale, specific applications. As meta-model functionality changes, the generator and validator need to be altered.

The extra tooling employed in an MDD process make that an architect spends more time investigating, testing and explaining development tools. A rapid pace of development and the fragmented offering of state of the art MDD case tools requires an extensive evaluation process as a part of the inception of any MDD project.

Designers Have to Build More Unequivocal Models

The architect found that he continuously needed to support and correct the model designers to learn to work with the DSL, the modeling tool and the validator. Designers struggled to understand the implications of their design choices and found it difficult to create models fit for code generation. The architect played a central role in the continuous training that designers required. Being located onshore, providing this training to the offshore development team was quite a challenge. Daily intensive (video) training sessions were held.

In traditional software development, designers build a set of diagrams to convey certain key aspects of a system. Requirements are translated to a technical solution according to architectural rules. It is often up to developers how to precisely implement an aspect described by a design. The UML offers a great degree of freedom. In practice, this freedom leads to inconsistent, incomplete and otherwise ambivalent diagrams (Lange et al., 2003). The work of a modeler in MDD is different in the sense that it is not only to communicate functionality but also to directly implement that functionality by modeling. This implies that traditional trade-offs regarding design effort and detail and completeness of diagrams are no longer made. There are far fewer solutions that are correct.

Generator is a Parallel Software Development Project

The architect found that parallel development of a code generator quickly grew into a separate project with its own architecture, stakeholders and e.g. defect management system.

The applicability of code generation as a development method is limited to how specific the system requirements are. An "off-the-shelf" set of model transformations is rarely capable of generating exactly what a specific client wants. A specific methodology bundled with a code generator only allows for very specific applications to be built, in which case the client has little to say about the software architecture. Therefore, to facilitate the specific requirements of a large corporate client for a sizable system, model transformations must evolve with both models and the code. Consequently, in addition to the development of the software system that is central in the project, a software architect is responsible for development and maintenance of a code generator. As the main system, the code generator has its own requirements, architecture, design and code. In the project, a separate team of developers was responsible for development and maintenance of the code generator. The main influence of this practice was found to be that requirement changes have a larger impact on the development process. The impact of changes has to be checked in great detail so the impact analysis of a change requests is more detailed. However, according to the architect, a side effect of needing to more carefully examine changes was that the impact of changes was very clear and potential problems and defects are spotted much earlier. This prevented rework and thereby saved time.

Late Changes Can Have a More Fundamental Impact

The use of code generation comes at the cost of greater standardization. If certain functionality is not supported by a meta-model, it can only be generated by extending the meta-model, the model transformations and possibly the DSL. This is potentially more time-intensive than adding the functionality by hand. Late changes to the meta-model may therefore require a disproportional amount of effort in modifying existing models or generated code. To prevent major rework, any requirement that impacts the meta-model must be clear upfront. The architect should make sure that before commencing modeling, the meta-model is as mature as needed.

In this case, at a late stage in the project, a specific requirement regarding navigation through the graphical user interface was discovered. In an earlier version of the architecture it was prescribed that to navigate from one screen to another, that those two screens had to be explicitly connected to each other in the model. Marking every navigation step with an arrow implies that an increase in the amount of screens that can reach each other exponentially increases the amount of arrows that needed to be drawn in the models. Because it was not made clear that most of the screens would require the possibility to navigate to one another, that particular aspect of the architecture would

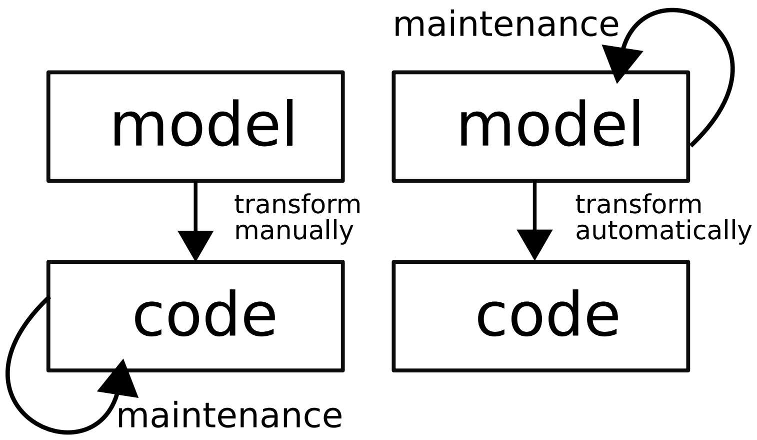


Figure 8.2: *MDD versus non-MDD maintenance (adapted from Van Vliet, 2008)*

probably have been redesigned at an early stage in the project. In a traditional project, this problem would probably be solved at the code level. In the case of this project, the screen navigation had to be altered at the model level and even required changes to the architecture, reference model and model transformations. The considerable amount of effort that had to be spent to rework all models after the late revision of fundamental aspects of the meta-model, indicates that it is imperative to find all requirements that significantly impact the meta-model *before* taking up modeling. Cabot and Yu (2008) argue for extending of MDD methods with improved requirements techniques.

Maintainers Need to be Involved During Development

Maintenance of software constructed using MDD tools and techniques is different in that not the code, but the models need to be altered 8.2. To ensure that models and system stay in-sync, it is imperative that maintainers are trained to understand the DSL, the models, the model transformations, the generation process employed and the integration between the generated code and the hand-written code.

It is essential that post-release changes are applied consistent with the MDD process used during development. Therefore, intimate knowledge of the meta-model buildup and the code generator as well as the frameworks involved is required from the maintenance staff. This knowledge is best obtained by close involvement of the development process.

8.4.3 Model Reuse

Domain-specific models can be reused for new software systems within the same domain. For this case, a meta-model was created before the project started. We found three consequences of the use of that meta-model:

First, the objective was to have that meta-model expanded and kept separate from this project so that it could be used in future projects. This requires external stakeholders that ensure that the meta-model stays generalizable. No resources were allocated for such a role. Second, this existing meta-model, which was created by domain experts, did not represent the client's perception on that same domain. Third, if a client initially only requires a subset of functionality that an existing meta-model offers, it might seem tempting to expand a system's scope to include "things the meta-model already can do." The second and third consequences are discussed in more detail in the next sections.

Increased Likelihood of Scope Creep

In this case, project management regularly budgeted client requests for specific meta-model functionality at zero hours of person effort. In this case, severe project time and budget overruns could in great part be ascribed to this practice.

Scope creep occurs when system functionality expands beyond the initial project objectives. Any software development project will meet changing requirements and generally, project management evaluates whether a change is in scope before accepting it as part of the original system or whether it should be treated as a additional functionality. The use of MDD can make scope creep more likely for two reasons: First, extending functionality can be easier than in non-MDD development. This specifically pertain changes already supported by the meta-model. Second, an existing meta-model may contain more functionality than specified in the requirements, making it even easier to generate new functionality. This impacts the discussion between software supplier and client whether added or changed functionality is part of the original system requirements or if it should be treated as a change request. It might be easy from a technical perspective to generate functionality that is beyond project scope. However, the impact of added functionality extends beyond the technical implementation. Extra functionality requires increased test and documentation effort. Working beyond project scope furthermore requires a supplementary iteration of the analysis of the business modeling as added functionality might impact existing business processes of systems in the environment and could imply the inclusion of additional interfaces. In addition, not all code in an MDD project is generated and a part of the newly generated code might need to be amended by hand. This is costly, time consuming and it might well add to the complexity of the system.

The organizational impact of introduced features beyond initial project scope could also include additional training of future users or an extension of the pool of future users which in turn might impact other requirements.

An Existing Meta-Model Might Conflict with Client Reality

A benefit of MDD is that an existing meta-model can be used to quickly deploy applications within a certain domain. In the case of the project, a pre-existing meta-model of a specific aspect of the Dutch mortgage domain was the main motivation of applying MDD. This model was created in concordance with business analysts with extensive experience in the Dutch mortgage domain. However, an existing domain model might not correctly represent a domain in the way the client perceives it. Many assumptions made by experts in this domain regarding business processes and product-composition were not completely consistent with the business approach of the client. This either stemmed from incorrect assumptions or from domain evolution. Redevelopment of the meta-model lengthened development time and hampered potential productivity improvements from the use of an existing meta-model. In deciding between a detailed and a more generic meta-model describing a certain domain, the latter approach could be more feasible. The biggest gains of MDD can therefore be expected in stable domains (with limited domain evolution) or in domains in which much commonality exists. This same problem can occur at the DSL-level. The strong link between the DSL and the domain benefits development by domain experts, but backfires when that domain evolves. Various studies propose methods for addressing domain model evolution (Deng et al., 2006, Sprinkle and Karsai, 2004).

8.5 Impact of MDD on GSD

In this section, we discuss the advantages and the disadvantages that the identified impacts of use of MDD have on GSD. In their recent structured literature review of empirical studies in GSD, Šmite et al. (2010) give an overview of GSD challenges and the best practices that are so far known. In Table 8.2, these are linked to the MDD impacts that were discussed in the previous sections. The first two columns in this table have been taken from Šmite et al.

Most of the best-practices associated with GSD are directly affected by process changes that are found with use of MDD tools and techniques. As discussed earlier, many studies hypothesized that the communication benefits that a DSL entails would benefit the GSD process. The increased communication efficiency that was found in the case enforces — or at least positively impacts — many of the best practices in the overview of Šmite et al.. A DSL provided for a common language and therefore mitigated some of the problems associated with what is commonly regarded as the toughest of the three distances (Herbsleb et al., 2000): socio-cultural distance. However, models also made for the richer communication that GSD needs. In addition, the close interaction with the client through use of a DSL, enabled the incremental short-cycle development that is beneficial for GSD.

However beneficial all these communication-related benefits are, the evidence that

MDD mitigates some of the problems of GSD reaches further. The requirement of shared model ownership implies that a centralized project repository — essential in GSD — had to be employed. Furthermore, use of MDD required a more extensive and more explicitly defined architecture. This enabled easier task distribution based on architectural decoupling, which is one of the most concrete best practices for GSD (Herbsleb et al., 2000). Also, an increased reliance on tools that accompanies the use of MDD enabled a more reliable infrastructure through a more formal method of working. Still, integrating these tools in the existing chain of tools was a challenge and so was training people to use them. MDD was not found to have any direct impact on the "synchronous interaction" best practice that Šmite et al. listed. While MDD does not require this type of interaction it does not inhibit it either. A potential drawback of MDD when used in GSD is the training. Programmers, designers but also management needs to be educated so to understand the MDD paradigm. When assembling an offshore team it proved be difficult to assess the extent to which candidates had the required skills. Furthermore, the short iteration cycles mentioned in Table 8.2 are needed to address the problem of process unclarity or the lack of awareness of either the process followed or current process status (Espinosa et al., 2001, Levesque et al., 2001, Carmel, 1999, Mockus and Herbsleb, 2001). Some organizations tailor their process prescriptions to cater for GSD (Heijstek et al., 2010) but this does not solve the awareness problem. MDD requires that a strict process is followed. This process was defined early in the elaboration phase in concordance with the entire team. As the model transformations would evolve with the diagrams, offshore team members were aware of the status of their work, the work of the onshore team and the status of the project as a whole.

8.6 Conclusions and Future Work

Using MDD tools and techniques fundamentally impacts the software development process in general and the analysis and design phases in particular.

All team members in this case elaborated on how the use of models as a common language eased communication between team members in general and between on- and offshore teams in particular. In addition, models enabled a larger group of stakeholders to participate in implementation-related discussions. This translated to fewer traveling back and forth between the offshore and onshore locations than is normally the case in projects of similar size and complexity.

We found that the use of a common language mitigated some of the problems associated with what is commonly regarded as the toughest of the three distances (Herbsleb et al., 2000): socio-cultural distance. In addition, MDD techniques in general and shared model ownership in particular forces more frequent interaction between more team members.

While MDD enforces most GSD best practices that are currently known in literature, some drawbacks exist. Staffing requirements include team members that are willing to work with models and model CASE tools and highly skilled developers. However, in GSD contexts, it is not always possible for an architect to influence development team composition. In addition, the application of MDD requires more formal working procedures in terms of e.g. more extensive and detailed design documentation and models that strictly adhere to modeling guidelines. The architect played a central role in the continuous training that team members required. Being located onshore, providing this training to the offshore development team was quite challenge.

Table 8.2: *Relating GSD best practices (Šmite et al., 2010) to MDD-related practices*

Practices	Advantages	Impacted by MDD	GSD Impact
• F2F meetings • temporal collocation • exchange visits	• Trust • cohesiveness • effective teamwork	☒	• Intra team communication was said to be more efficient with MDD because of DSL • Less travel was required since communication was clearer
• Centralized project repository • common configuration management tool support	• Awareness • process transparency	☒	• Central repository was required for collective model ownership • More tools were used
• Effective and frequent synchronous communication	• Trust • cohesiveness	☒	• Communication was said to be more efficient with MDD because of DSL
• Reliable infrastructure • rich communication media	• Effective communication	☒	• Models made for richer communication as they were included in meetings. • More tools were needed. These allow stricter work procedures. The introduction of new tools also introduces some uncertainty. • The development process was more formal.
• Synchronous interaction	• Effective teamwork	☐	
• Task distribution based on architectural decoupling and low dependencies across remote locations	• Effective teamwork	☒	• A greater proportion of the architecture was explicitly defined • Any architectural decoupling was more straightforward to enforce as the implementation was closer to the architecture • Requirements were clearer
• Incremental short-cycle development	• Early feedback, capability evaluation	☒	• Code generation enables faster development • Earlier feedback is obtained because of closer client interaction through the DSL

Chapter 9

Conclusions

This dissertation addresses how software architecture and design is to be represented, disseminated and coordinated in the context of global software development. To this end, the role of software architecture (as a process as well as an artifact) was empirically assessed in various industrial contexts. In addition, the special case of model-driven software development was addressed. This chapter contains a summary and integration of the findings that were presented in the previous chapters as well as an outline of future work.

9.1 Summary of Findings

In this section, the empirical evidence collected throughout this dissertation is summarized and used to address the research questions central to this dissertation (Section 1.3).

9.1.1 RQ1: How is Software Architecture Represented, Disseminated and Coordinated in the Context of Global Software Development?

This dissertation set out outlining how organizations tailor software development process descriptions for the challenges that GSD introduces, from a process perspective (Chapter 2). Investigating how software development process descriptions are tailored to accommodate for GSD, we found that the process approach to GSD is dependent on organization size, maturity, intended use of the description and the expertise and experience of the process engineers.

Subsequently, we presented and demonstrated our method to visualize GSD processes consistent with an iconic process visualization (Chapter 3). These visualizations uncovered aberrant distribution of analysis and design effort which were the result of unclarities in the processes of communication and coordination of software architecture.

We then explicitly analyzed the role of software architecture design in the context in global software development by means of three case studies (Chapter 4). We found that some problems relating to software architecture dissemination and coordination processes led to poor architectural compliance. This, in turn, led to project overruns. The dissemination of software architecture as well as the role of the software architect are not formalized even though this might very well have benefited the development process. An important benefit of the application of MDD tools and techniques is that most of the software architecture is generated. As a result, architecture compliance improves. While this mitigates the problems associated with disseminating software architecture design, it introduces the problem of teaching developers to work with a Domain-Specific Language (DSL), associated tooling and the MDD approach in general.

Finally, we validated the findings from the case studies by means of a series of interviews with experts (Chapter 5). We then integrated the factors that influence how software architecture design is coordinated and disseminated and identified three main drivers to explain these factors:

1. First, the strong implementation focus of software development project management prematurely forces projects into the construction phase.
2. Second, a knowledge gap exists between the onshore and offshore location regarding software architecture and its role during the software development life cycle.
3. Third, cost reduction forces a move of responsibilities towards the offshore software development location. This compounds the "knowledge gap" problems as less resources are available for knowledge improvement (training) and more work is required of less experienced team members. In addition, the added value of activities related to implementation is more tangible than that of design-related activities. As a result, the "'implementation focus" problem is aggravated.

9.1.2 RQ2: How can we design software architecture documentation so that it is understood well by developers in the context of global software development?

We designed and executed an experiment in which we evaluated how software developers comprehend software architecture representations from the perspective of diagram-dominant versus text-dominant representations (Chapter 6). We found that neither diagrams nor textual descriptions are significantly more efficient in terms of communicating software architecture design. In addition, we found that diagrams were not able to alleviate the difficulties participants with a native language other than English had in extracting information from the documentation. However, while diagrams were not superior regarding media effectiveness they still seemed to perform a special role. Participants were more likely to use diagrams as their first source. They were more likely to look at the diagram at the very moment when they provided answers to questions of a topological nature.

Finally, we identified developer characteristics that can be used as developer performance predictors: linguistic distance, media preference, experience and self-rated modeling skill. The participants who performed best had a native language close to English, looked at text more than at diagrams, were more experienced and rated their own modeling skills to be relatively high. We conclude that, contrary to current industrial practice, architecture documentation should be specifically tailored for its audience in terms of the developer's experience and native language and the general readability of the text.

9.1.3 RQ3: How does the application of model-driven development tools and techniques relate to the problems associated with global software development?

We analyzed how the characteristics of a large scale, industrial model-driven development project in the context of global software development compare to non-MDD projects (Chapter 7). In MDD, models instead of code are the central development artifact. We found that the same logic applied to code cannot be applied to models: First, the majority of development effort was spend on developing the models. That is significantly more than the time spent on code development in classical software development. Second, most model elements were already present at one third of the development process. The remaining development time was spent on altering the models. In addition, 40 percent less defects were found in the MDD case when compared to projects of similar size. Models and code are fundamentally different and therefore not easily compared as we found absent, for example, a positive relation between model size and model complexity on the one hand and model defects on the other - relations that have often been observed in source code. Also, larger diagrams were changed more often and worked on longer but did not necessarily contain more defects.

We then analyzed how the application of MDD tools and techniques specifically impact the problems associated with Global Software Development (Chapter 8). We found that the use of models as a common language mitigated some of the problems associated with socio-cultural distance and also resulted in fewer traveling back and forth between the offshore and onshore locations. In addition, MDD techniques in general and shared model ownership in particular forces more frequent interaction between more team members.

Finally, an important implication of the use of code generation is that the software

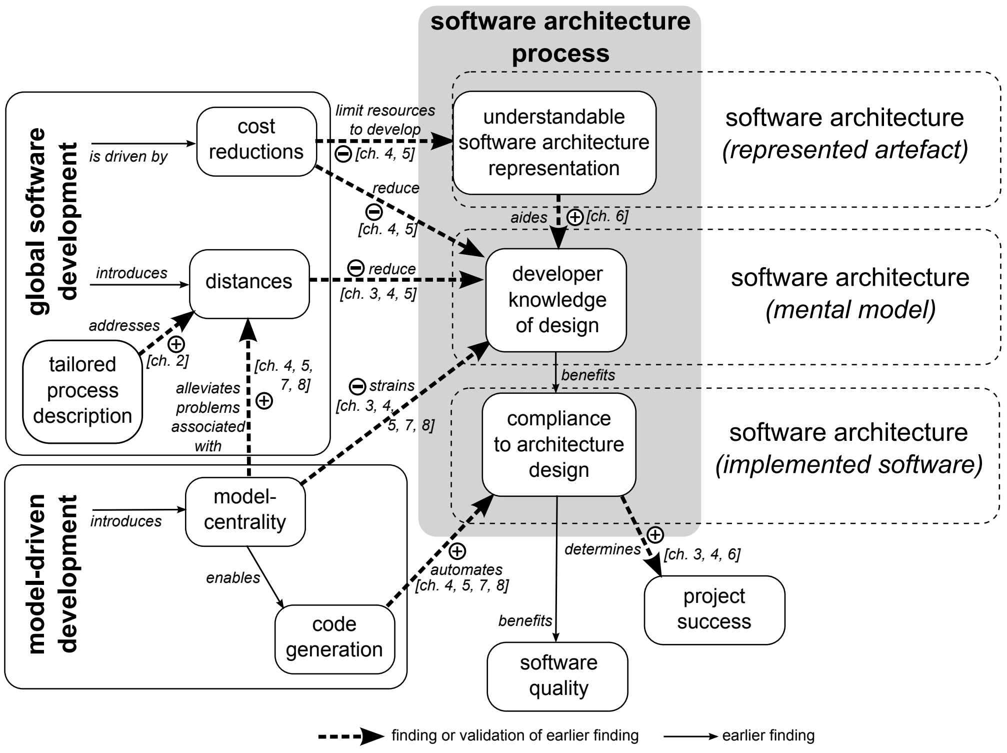

software architecture process
global software development
is driven by
cost reductions
introduces
distances
addresses
⊕ [ch. 2]
tailored process description
alleviates problems associated with
⊕ [ch. 4, 5, 7, 8]
limit resources to develop
⊖ [ch. 4, 5]
reduce
⊖ [ch. 4, 5]
⊖ reduce [ch. 3, 4, 5]
⊖ strains [ch. 3, 4, 5, 7, 8]
understandable software architecture representation
software architecture (represented artefact)
aides ⊕ [ch. 6]
developer knowledge of design
software architecture (mental model)
benefits
compliance to architecture design
software architecture (implemented software)
model-driven development
introduces
model-centrality
enables
code generation
⊕ automates [ch. 4, 5, 7, 8]
determines ⊕ [ch. 3, 4, 6]
project success
benefits
software quality
finding or validation of earlier finding
earlier finding

Figure 9.1: *Cause-effect diagram integrating the main findings of the study*

architecture is generated. Software architecture compliance is therefore automated. Nevertheless, the application of MDD required more formal artifacts in terms of e.g. more extensive and detailed design documentation and models that strictly adhere to modeling guidelines. The architect played a central role in the continuous additional training that team members required.

9.2 Contributions

The main contributions of this study and their interrelations are visualized in Figure 9.1. First, this dissertation establishes that *the software architecture process is significantly different in the context of GSD* when compared to co-located development. Second, we have demonstrated a clear link between architecture compliance and project success in terms of limiting rework. Third, we have presented evidence that the *cost reductions inherent to GSD limit the resources available for software architecture design and representation*. Fourth, we have found how developer knowledge of software architecture design:

1. can be positively influenced by improving diagrams and text in software archi-

tecture representation and training developers;

2. is limited by the socio-cultural, geographical and temporal distances in that GSD introduces;

3. is hampered by cost reductions which limit the resources available for coordination of software architecture;

4. is less important in the context of MDD because the architecture is a stable framework that is already implemented to enable code generation. Instead, however, developers need to learn to work with a new development paradigm, new tooling and a DSL.

Fifth, we found evidence for the principle that code generation increases the extent to which a software implementation complies to its intended architecture as much of the architecture is generated.

An important contribution that is not explicitly modeled in Figure 9.1 is that the use of models as a central development artifact is fundamentally different from using code. The logic or intuitions regarding effort and amount of defects that we have when it comes to source code cannot be applied to models. However, the implications of application of MDD tools and techniques are not yet clearly understood and therefore warrant further study.

9.3 Recommendations to Industry

Following from the findings of this dissertation, we formulate five recommendations:

1. *The problems associated with GSD should be specifically addressed in software development process descriptions*
   Such an addition might be as informal as a (concrete) list of best practices. Problems that were encountered and solved in a specific project by means of technology or a (set of) best practice(s) should find their way to the process description so that other projects may benefit in the sense that similar mistakes are not repeated. Given that the majority of problems in GSD is thought to benefit from intra-team member communication-related practices, the intended audience of such an augmented process description should be all team members, rather than project management.

2. *The processes of dissemination and coordination of software architecture must be explicitly formalized*

   - Preferably, one or more experienced offshore developers should be involved in the development of the architecture under guidance of an experienced onshore architect.

- The most fundamental aspects of the architecture design should be developed before starting offshore construction.
- If budget allows, the development of an architecture POC in which new, complex or otherwise unknown functionality is addressed, is recommended.
- The most knowledgeable architect should travel to the offshore location at least once, preferably when the construction phase commences.
- When disseminating software design, validate that the receiving party made the correct interpretation.
- The availability of the principal (onshore) architect should only be limited to the extent that the knowledge and experience of the principal offshore developer (the "technical lead") allows this. Any cost savings from limited association of the "expensive" onshore architect are unlikely to offset the costs incurred by rework as a result of architectural noncompliance.
- Offshore developers should be able to directly contact the software architect — preferably in a group so that effective use can be made of the architect's time. These sessions should be planned regularly to avoid developers having to batch their questions.
- Developers should be coached to understand that knowledge of the role of "their" component in relation to other components matters to the extent that it determines the quality of their work.

3. *Increase allocation of resources in architecture design*
   Upfront investment in architecture design is likely to lower budget overrun due to having rework a faulty architecture implementation.

4. *Create unambiguous and concise software architecture documentation that is specifically tailored to its intended audience*
   Pay particular attention to the use of both text and diagrams, even for topological information and annotating each (non-UML) diagram with a description of how to read it. Additionally, investments in UML training for developers benefits developer understanding of architecture representation.

5. *Consider application of MDD tools and techniques to reduce the negative impact of communication-related GSD problems*
   The use of models as a common language eases communication between on- and offshore teams and enables a larger group of stakeholders to participate in implementation-related discussions.

9.4 Future Work

As with all good research, the value of our findings in great part lies in the impetus they provide to perform further research. We describe relevant directions for future work in the next sections.

9.4.1 In-dept Evaluation of the Role of Documentation in GSD

We argued that software documentation plays a more central role in GSD than in co-located software development. In industrial practice, however, documentation is often regarded as a by-product and information is preferably shared directly and informally between people — hence the popularity of documentation-light Agile approaches. We take the point of view that not more or less documentation must be created, but *better* documentation. Nevertheless, advanced methods of documentation generation, an increase in the level of technical maturity of clients and the advent of team wikis do question the role of documentation in the software development process. Future work in this direction should aim to quantify the usefulness of documentation by, for instance, mapping the match between information need and availability and evaluating the extent to which parts of existing documentation is perceived as adding value. The outcome of such a study would be more lean documentation templates, a knowledge support system and/or best practices for knowledge codification practices. The action research paradigm is likely to be a suitable research method to address this topic.

9.4.2 Quantifying the Relation Between Developer Architecture Design Understanding and Software Quality

This dissertation provides evidence which supports recommending upfront investments in architecture development and representation to ensure developer architecture design understanding. While we take the stance that understanding of software architecture design is beneficial for architecture compliance, we are unsure about the mechanism underlying the nature of this relation. For example, how much should a developer know about an architecture to ensure his software is compliant? All of it? That seems uneconomical. Is it perhaps enough to understand the relation between "his" components and neighboring parts of the system? To summarize: How much better would developers implement if they know more about architecture? Such a topic could be addressed by means of a quasi-experiment in the sense that a researcher could invest in educating a selected group of developers with regards to the software architecture.

9.4.3 Facilitating industrial application of MDD

This dissertation contains evidence for the significant impact of shifting from code to models as central development artifacts. We also know that industrial application of MDD is slow and that limited evidence exists for MDD's potential benefits. Various reasons have been offered to why MDD seems used so little and so often to no avail. Reasons include unrealistic expectations, the problematic offering of MDD tooling and a general lack of understanding of the concept of MDD. Future work should preferably investigate industrial cases of MDD to collect factors that contribute to successful industrial application.

Samenvatting

Software speelt een sleutelrol in ons leven. Software is niet alleen verantwoordelijk voor het functioneren van zowel onze digitale als onze fysieke infrastructuur maar is zelfs van groot belang in onze dagelijkse inter-persoonlijke communicatie: We interacteren met organisaties en onze overheden met behulp van software en we vertrouwen niet alleen op de software in onze wekker en digitale videorecorder maar ook op de software in de MRI scanner in het ziekenhuis of in de treinbeveiliging.

Deze ontwikkeling leidt tot een vraag naar steeds grotere en complexere software-systemen enerzijds en toenemende kwaliteitseisen anderzijds. Om niet-functionele systeemeisen (als veiligheid, onderhoudbaarheid en duurzaamheid) te kunnen garanderen is het zaak om de interactie van systeemcomponenten op een hoger niveau van abstractie te ontwerpen. Dit hogere niveau van abstractie wordt software-architectuur genoemd — in lijn met de bouwkundige analogie waar “software engineering”, een gangbare aanduiding voor softwareontwikkeling, aan refereert. Methoden, technieken en processen die samenhangen met software-architectuur stellen softwareontwikkelaars in staat om te gaan met toenemende systeemcomplexiteit en stengere kwaliteitseisen. In het wetenschappelijke onderzoek heeft software-architectuur de afgelopen 15 jaar daarom veel aandacht gekregen.

De voordelen van het werken met abstracties en de directere link die daardoor gelegd kan worden met het specifieke domein waarbinnen een bepaald software systeem ontwikkeld wordt, maakt dat modellen een steeds centrale rol spelen binnen softwareontwikkeling. Een veel toegepast concept waarbinnen modellen centraal staan is modelgedreven softwareontwikkeling (MDD). In dit ontwikkelparadigma staan modellen, vaak ontwikkeld in domeinspecifieke talen, centraal. Afhankelijk van de ondersteunende software en specifieke methoden die worden toegepast kan executeerbare code worden gegenereerd uit abstracte modellen of worden deze modellen zelf “uitgevoerd” als waren ze software. De praktische relatie tussen software-architectuur van software-systemen die in meer of mindere mate van de grond af aan ontwikkeld worden (ook wel custom of “greenfield” softwareontwikkeling) en modelgedreven ontwikkeling is onduidelijk.

Offshoring (ook wel outsourcing of gedistribueerde softwareontwikkeling) is weer een andere, meer recente ontwikkeling die van grote invloed is op de manier waarop software wordt ontwikkeld. Bij deze vorm van softwareontwikkeling wordt software gebouwd op meerdere, geografisch gescheiden locaties tegelijk. Aanvankelijk werden kostenbesparingen verwacht van de lagere lonen van softwareontwikkelaars in landen als Canada en India. Later leek offshoring ook een oplossing voor het tekort aan softwareontwikkelaars door de steeds grotere vraag naar software-systemen.

In een gedistribueerd softwareontwikkelproject vinden de verschillende ontwikkelfasen typisch plaats op verschillende, door significante geografische afstand gescheiden locaties. Het verzamelen van systeemeisen en ontwikkelen van de architectuur vind meestal plaats op een andere locatie dan waar het systeem daadwerkelijk wordt geïmplementeerd (geprogrammeerd). Van de implementatie van software architectuur in een niet-gedistribueerde omgeving weten we dat de mate waarin de implementatie van een architectuur voldoet aan het ontwerp, in hoge mate afhankelijk is van de manier waarop en de mate waarin de software-architect softwareontwikkelaars ondersteunt bij hun werk. Hoewel de beoogde software-architectuur vrijwel altijd wordt beschreven in een document (SAD), lijken softwareontwikkelaars verder veel te leren over de beoogde architectuurimplementatie door informele communicatie met andere teamleden die veel weten van de architectuur. Geografische, temporale en socio-culturele afstanden maken formele (geplande) communicatie in gedistribueerde softwareontwikkeling echter lastig. Informele communicatie is daarom nog lastiger te organiseren. Het is onduidelijk of documentatie in deze situaties nu een grotere rol speelt of dat software nu misschien minder goed aan architectuurspecificaties voldoet.

De centrale doelstelling in dit proefschrift is onderzoeken *hoe software-architectuur op een effectieve manier kan worden gerepresenteerd, overgedragen en gecoördineerd in de context van gedistribueerde en model-centrische softwareontwikkeling*. De onderzoeksvragen die we hebben afgeleid uit deze doelstelling zijn alle geënt op empirische onderzoeksmethoden. In de context van softwareontwikkeling wil dit zeggen dat we zoveel mogelijk gebruik maken van gegevens uit de bedrijfspraktijk. De belangrijkste bijdragen van dit proefschrift zijn:

- de vaststelling dat de processen omtrent software-architectuur flink afwijken als niet-gedistribueerde en gedistribueerde softwareontwikkeling worden vergeleken, vooral dat er in een relatief laat stadium nog veel werk wordt verricht aan de architectuur van systemen,

- dat er een sterk, negatief verband lijkt te bestaan tussen de mate waarin de implementatie van software-architectuur voldoet aan zijn ontwerp en de mate waarin een softwareontwikkelproject op tijd wordt opgeleverd,

- dat de kostenbesparingen inherent aan de motivatie om gedistribueerd software te ontwikkelen direct beperkingen lijken op te leveren voor de kwaliteit van de ontwikkelde architectuur en haar representatie,

- dat het begrip van softwareontwikkelaars van software-architectuur
  - positief beïnvloed kan worden door diagrammen en tekst in software-architectuur-documenten te verbeteren en door softwareontwikkelaars te scholen in het gebruik van UML,
  - gelimiteerd wordt door de socio-culturele, geografische en temporale afstanden die door gedistribueerde softwareontwikkeling worden geïntroduceerd,
  - wordt verslechterd door kostenreducties, specifiek doordat er minder wordt geïnvesteerd in coördinatie van het software-architectuur-proces,
  - minder belangrijk is in modelgedreven softwareontwikkeling maar dat ontwikkelaars desalniettemin moeten leren werken met een nieuw softwareontwikkelparadigma, nieuwe software tools én de domeinspecifieke taal,
- dat software-implementaties beter voldoen aan de voorgeschreven architectuur als modelgedreven softwareontwikkeling wordt toegepast omdat de software-architectuur vast wordt gelegd om generatie van broncode mogelijk te maken.

Een bijkomstige bevinding is dat het gebruik van modellen als centrale ontwikkelartefacten fundamenteel verschilt van het klassieke model waarin programmacode deze rol vervult. De verschillen zelf en de implicaties van deze verschillende softwareontwikkelparadigma's zijn echter nog onduidelijk en nodigen uitdrukkelijk uit tot nader onderzoek.

Curriculum Vitae

Werner Heijstek was born in 1982 in Dordrecht, the Netherlands. He studied Computer Science and obtained his M.Sc in ICT in Business at Leiden University in 2007. From October 2007, he worked as a Ph.D. candidate and lecturer at the Leiden Institute of Advanced Computer Science (LIACS) at the Leiden University Faculty of Science. He worked within the Software Engineering Group under supervision of Prof. Dr. Michel R.V. Chaudron.

From November 2006 to October 2012, Werner was affiliated with the Accelerated Delivery Center (ADC) of Capgemini the Netherlands. From October 2008 until January 2009, Werner was a visiting researcher at the Caesarea Edmond Benjamin de Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science and the Department of Management Information Systems of Haifa University (Israel) in cooperation with Dr. Irit Hadar. From February until May 2010, Werner was a visiting researcher at the School of Engineering and Computer Science at Victoria University of Wellington (New Zealand) in cooperation with Prof. Dr. Thomas Kühne. Twice, Werner was a visiting researcher at Capgemini India in Mumbai in cooperation with Supriya Sonawane (in April 2007 and November 2010).

Before, during and after his studies Werner worked for several smaller and larger IT firms including KPN Mobile, Accenture Technology Services and Cyco Software. From October 2012, Werner works as a management consultant at the Software Improvement Group in Amsterdam.

Acknowledgments

Due to the empirical nature of my research and because I was lucky enough to travel much, I owe thanks to many people. First and foremost I thank Emiel Zandvliet for the generous grant that supported our research.

I would specifically like to thank my old LIACS roommate Ariadi Nugroho, who often helped me with problems and who always made for good conversation. In addition, I would like to thank Christoph for our pleasant out-of-group collaborations. I would also like to thank my colleagues Hafeez, Dave, Ana, Ramin, Bilal and Peng. I enjoyed teaching, supporting and supervising the various cohorts of the ICT in Business and Computer Science programs that studied at LIACS while I was there. I thank "my students" Martine, Marin, Stefan, Simon, Libing, Dimitris, Li and Erik Jan. I would like to thank Hugo in particular for his help during the execution of the software architecture design documentation experiment.

The people at Capgemini's ADC were always generous with their time and advice. For that I thank all members of the Dutch and Indian ADC and in particular Peter Bink, Peter Eichelsheim, Jan Vogel, Ronald Oudshoorn, Supriya Sonawane, Ron Meulenbroek, Jan Wiggers, Peter van Egdom, Anita Becattini and Hester Spee.

For my visit to Israel, I thank Irit Hadar, Peleg Yiftachel and the Caesarea Edmond Benjamin de Rothschild Foundation Institute for Interdisciplinary Applications of Computer Science and the Department of Management Information Systems of Haifa University for hosting me. I thank the Israeli Ministry of Foreign Affairs Division for Cultural and Scientific Affairs and the The Netherlands Organization for International Cooperation in Higher Education (NUFFIC) for their generous travel grant. For my visits to India, I thank Supriya Sonawane and my kind colleagues at the Capgemini ADC India in Mumbai for hosting me on two occasions. I thank the Platform Outsourcing the Netherlands (PON) for their travel grant. For my visit to New Zealand, I thank Thomas Kühne for his extensive support during my stay, Craig Anslow, the School of Engineering and Computer Science at Victoria University of Wellington and professor Stephen Levine. In addition, I would like to thank all involved organizations and all software architects, developers, designers, project leaders, testers and other people that I interviewed or that participated in my experiments. Thank you for your time and openness.

Mom and dad, thank you for your love and support, for keeping me in school and for always being there. Kavita, thanks for making sure that I started and completed this project. You were also responsible for many diversions though. Remember making me jump from planes and bridges, dive with sharks and race through missile attacks from Gaza? The coolest diversion has surely been Adam. Little man, while you did not exactly make this process easier, you certainly made it *much* more fun.

Werner Heijstek
Leiden, November 2012

List of Illustrations

Figures

Tables

Titles in the IPA Dissertation Series since 2006

E. Dolstra. *The Purely Functional Software Deployment Model*. Faculty of Science, UU. 2006-01

R.J. Corin. *Analysis Models for Security Protocols*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-02

P.R.A. Verbaan. *The Computational Complexity of Evolving Systems*. Faculty of Science, UU. 2006-03

K.L. Man and R.R.H. Schiffelers. *Formal Specification and Analysis of Hybrid Systems*. Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2006-04

M. Kyas. *Verifying OCL Specifications of UML Models: Tool Support and Compositionality*. Faculty of Mathematics and Natural Sciences, UL. 2006-05

M. Hendriks. *Model Checking Timed Automata - Techniques and Applications*. Faculty of Science, Mathematics and Computer Science, RU. 2006-06

J. Ketema. *Böhm-Like Trees for Rewriting*. Faculty of Sciences, VUA. 2006-07

C.-B. Breunesse. *On JML: topics in tool-assisted verification of JML programs*. Faculty of Science, Mathematics and Computer Science, RU. 2006-08

B. Markvoort. *Towards Hybrid Molecular Simulations*. Faculty of Biomedical Engineering, TU/e. 2006-09

S.G.R. Nijssen. *Mining Structured Data*. Faculty of Mathematics and Natural Sciences, UL. 2006-10

G. Russello. *Separation and Adaptation of Concerns in a Shared Data Space*. Faculty of Mathematics and Computer Science, TU/e. 2006-11

L. Cheung. *Reconciling Nondeterministic and Probabilistic Choices*. Faculty of Science, Mathematics and Computer Science, RU. 2006-12

B. Badban. *Verification techniques for Extensions of Equality Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2006-13

A.J. Mooij. *Constructive formal methods and protocol standardization*. Faculty of Mathematics and Computer Science, TU/e. 2006-14

T. Krilavicius. *Hybrid Techniques for Hybrid Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-15

M.E. Warnier. *Language Based Security for Java and JML*. Faculty of Science, Mathematics and Computer Science, RU. 2006-16

V. Sundramoorthy. *At Home In Service Discovery*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2006-17

B. Gebremichael. *Expressivity of Timed Automata Models*. Faculty of Science, Mathematics and Computer Science, RU. 2006-18

L.C.M. van Gool. *Formalising Interface Specifications*. Faculty of Mathematics and Computer Science, TU/e. 2006-19

C.J.F. Cremers. *Scyther - Semantics and Verification of Security Protocols*. Faculty of Mathematics and Computer Science, TU/e. 2006-20

J.V. Guillen Scholten. *Mobile Channels for Exogenous Coordination of Distributed Systems: Semantics, Implementation and Composition*. Faculty of Mathematics and Natural Sciences, UL. 2006-21

H.A. de Jong. *Flexible Heterogeneous Software Systems*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-01

N.K. Kavaldjiev. *A run-time reconfigurable Network-on-Chip for streaming DSP applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-02

M. van Veelen. *Considerations on Modeling for Early Detection of Abnormalities in Locally Autonomous Distributed Systems*. Faculty of Mathematics and Computing Sciences, RUG. 2007-03

T.D. Vu. *Semantics and Applications of Process and Program Algebra*. Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2007-04

L. Brandán Briones. *Theories for Model-based Testing: Real-time and Coverage*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-05

I. Loeb. *Natural Deduction: Sharing by Presentation*. Faculty of Science, Mathematics and Computer Science, RU. 2007-06

M.W.A. Streppel. *Multifunctional Geometric Data Structures*. Faculty of Mathematics and Computer Science, TU/e. 2007-07

N. Trčka. *Silent Steps in Transition Systems and Markov Chains*. Faculty of Mathematics and Computer Science, TU/e. 2007-08

R. Brinkman. *Searching in encrypted data*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-09

A. van Weelden. *Putting types to good use*. Faculty of Science, Mathematics and Computer Science, RU. 2007-10

J.A.R. Noppen. *Imperfect Information in Software Development Processes*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2007-11

R. Boumen. *Integration and Test plans for Complex Manufacturing Systems*. Faculty of Mechanical Engineering, TU/e. 2007-12

A.J. Wijs. *What to do Next?: Analysing and Optimising System Behaviour in Time*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2007-13

C.F.J. Lange. *Assessing and Improving the Quality of Modeling: A Series of Empirical Studies about the UML*. Faculty of Mathematics and Computer Science, TU/e. 2007-14

T. van der Storm. *Component-based Configuration, Integration and Delivery*. Faculty of Natural Sciences, Mathematics, and Computer Science,UvA. 2007-15

B.S. Graaf. *Model-Driven Evolution of Software Architectures*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2007-16

A.H.J. Mathijssen. *Logical Calculi for Reasoning with Binding*. Faculty of Mathematics and Computer Science, TU/e. 2007-17

D. Jarnikov. *QoS framework for Video Streaming in Home Networks*. Faculty of Mathematics and Computer Science, TU/e. 2007-18

M. A. Abam. *New Data Structures and Algorithms for Mobile Data*. Faculty of Mathematics and Computer Science, TU/e. 2007-19

W. Pieters. *La Volonté Machinale: Understanding the Electronic Voting Controversy*. Faculty of Science, Mathematics and Computer Science, RU. 2008-01

A.L. de Groot. *Practical Automaton Proofs in PVS*. Faculty of Science, Mathematics and Computer Science, RU. 2008-02

M. Bruntink. *Renovation of Idiomatic Crosscutting Concerns in Embedded Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-03

A.M. Marin. *An Integrated System to Manage Crosscutting Concerns in Source Code*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2008-04

N.C.W.M. Braspenning. *Model-based Integration and Testing of High-tech Multi-disciplinary Systems*. Faculty of Mechanical Engineering, TU/e. 2008-05

M. Bravenboer. *Exercises in Free Syntax: Syntax Definition, Parsing, and Assimilation of Language Conglomerates*. Faculty of Science, UU. 2008-06

M. Torabi Dashti. *Keeping Fairness Alive: Design and Formal Verification of Optimistic Fair Exchange Protocols*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2008-07

I.S.M. de Jong. *Integration and Test Strategies for Complex Manufacturing Machines*. Faculty of Mechanical Engineering, TU/e. 2008-08

I. Hasuo. *Tracing Anonymity with Coalgebras*. Faculty of Science, Mathematics and Computer Science, RU. 2008-09

L.G.W.A. Cleophas. *Tree Algorithms: Two Taxonomies and a Toolkit*. Faculty of Mathematics and Computer Science, TU/e. 2008-10

I.S. Zapreev. *Model Checking Markov Chains: Techniques and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-11

M. Farshi. *A Theoretical and Experimental Study of Geometric Networks*. Faculty of Mathematics and Computer Science, TU/e. 2008-12

G. Gulesir. *Evolvable Behavior Specifications Using Context-Sensitive Wildcards*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-13

F.D. Garcia. *Formal and Computational Cryptography: Protocols, Hashes and Commitments*. Faculty of Science, Mathematics and Computer Science, RU. 2008-14

P. E. A. Dürr. *Resource-based Verification for Robust Composition of Aspects*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-15

E.M. Bortnik. *Formal Methods in Support of SMC Design*. Faculty of Mechanical Engineering, TU/e. 2008-16

R.H. Mak. *Design and Performance Analysis of Data-Independent Stream Processing Systems*. Faculty of Mathematics and Computer Science, TU/e. 2008-17

M. van der Horst. *Scalable Block Processing Algorithms*. Faculty of Mathematics and Computer Science, TU/e. 2008-18

C.M. Gray. *Algorithms for Fat Objects: Decompositions and Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-19

J.R. Calamé. *Testing Reactive Systems with Data - Enumerative Methods and Constraint Solving*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-20

E. Mumford. *Drawing Graphs for Cartographic Applications*. Faculty of Mathematics and Computer Science, TU/e. 2008-21

E.H. de Graaf. *Mining Semi-structured Data, Theoretical and Experimental Aspects of Pattern Evaluation*. Faculty of Mathematics and Natural Sciences, UL. 2008-22

R. Brijder. *Models of Natural Computation: Gene Assembly and Membrane Systems*. Faculty of Mathematics and Natural Sciences, UL. 2008-23

A. Koprowski. *Termination of Rewriting and Its Certification*. Faculty of Mathematics and Computer Science, TU/e. 2008-24

U. Khadim. *Process Algebras for Hybrid Systems: Comparison and Development*. Faculty of Mathematics and Computer Science, TU/e. 2008-25

J. Markovski. *Real and Stochastic Time in Process Algebras for Performance Evaluation*. Faculty of Mathematics and Computer Science, TU/e. 2008-26

H. Kastenberg. *Graph-Based Software Specification and Verification*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-27

I.R. Buhan. *Cryptographic Keys from Noisy Data Theory and Applications*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-28

R.S. Marin-Perianu. *Wireless Sensor Networks in Motion: Clustering Algorithms for Service Discovery and Provisioning*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2008-29

M.H.G. Verhoef. *Modeling and Validating Distributed Embedded Real-Time Control Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2009-01

M. de Mol. *Reasoning about Functional Programs: Sparkle, a proof assistant for*

Clean. Faculty of Science, Mathematics and Computer Science, RU. 2009-02

M. Lormans. *Managing Requirements Evolution*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-03

M.P.W.J. van Osch. *Automated Model-based Testing of Hybrid Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-04

H. Sozer. *Architecting Fault-Tolerant Software Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-05

M.J. van Weerdenburg. *Efficient Rewriting Techniques*. Faculty of Mathematics and Computer Science, TU/e. 2009-06

H.H. Hansen. *Coalgebraic Modelling: Applications in Automata Theory and Modal Logic*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-07

A. Mesbah. *Analysis and Testing of Ajax-based Single-page Web Applications*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-08

A.L. Rodriguez Yakushev. *Towards Getting Generic Programming Ready for Prime Time*. Faculty of Science, UU. 2009-9

K.R. Olmos Joffré. *Strategies for Context Sensitive Program Transformation*. Faculty of Science, UU. 2009-10

J.A.G.M. van den Berg. *Reasoning about Java programs in PVS using JML*. Faculty of Science, Mathematics and Computer Science, RU. 2009-11

M.G. Khatib. *MEMS-Based Storage Devices. Integration in Energy-Constrained Mobile Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-12

S.G.M. Cornelissen. *Evaluating Dynamic Analysis Techniques for Program Comprehension*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2009-13

D. Bolzoni. *Revisiting Anomaly-based Network Intrusion Detection Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-14

H.L. Jonker. *Security Matters: Privacy in Voting and Fairness in Digital Exchange*. Faculty of Mathematics and Computer Science, TU/e. 2009-15

M.R. Czenko. *TuLiP - Reshaping Trust Management*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-16

T. Chen. *Clocks, Dice and Processes*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2009-17

C. Kaliszyk. *Correctness and Availability: Building Computer Algebra on top of Proof Assistants and making Proof Assistants available over the Web*. Faculty of Science, Mathematics and Computer Science, RU. 2009-18

R.S.S. O'Connor. *Incompleteness & Completeness: Formalizing Logic and Analysis in Type Theory*. Faculty of Science, Mathematics and Computer Science, RU. 2009-19

B. Ploeger. *Improved Verification Methods for Concurrent Systems*. Faculty of Mathematics and Computer Science, TU/e. 2009-20

T. Han. *Diagnosis, Synthesis and Analysis of Probabilistic Models*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-21

R. Li. *Mixed-Integer Evolution Strategies for Parameter Optimization and Their Applications to Medical Image Analysis*. Faculty of Mathematics and Natural Sciences, UL. 2009-22

J.H.P. Kwisthout. *The Computational Complexity of Probabilistic Networks*. Faculty of Science, UU. 2009-23

T.K. Cocx. *Algorithmic Tools for Data-Oriented Law Enforcement*. Faculty of Mathematics and Natural Sciences, UL. 2009-24

A.I. Baars. *Embedded Compilers*. Faculty of Science, UU. 2009-25

M.A.C. Dekker. *Flexible Access Control for Dynamic Collaborative Environments*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2009-26

J.F.J. Laros. *Metrics and Visualisation for Crime Analysis and Genomics*. Faculty of Mathematics and Natural Sciences, UL. 2009-27

C.J. Boogerd. *Focusing Automatic Code Inspections*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2010-01

M.R. Neuhäußer. *Model Checking Nondeterministic and Randomly Timed Systems*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-02

J. Endrullis. *Termination and Productivity*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-03

T. Staijen. *Graph-Based Specification and Verification for Aspect-Oriented Languages*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2010-04

Y. Wang. *Epistemic Modelling and Protocol Dynamics*. Faculty of Science, UvA. 2010-05

J.K. Berendsen. *Abstraction, Prices and Probability in Model Checking Timed Automata*. Faculty of Science, Mathematics and Computer Science, RU. 2010-06

A. Nugroho. *The Effects of UML Modeling on the Quality of Software*. Faculty of Mathematics and Natural Sciences, UL. 2010-07

A. Silva. *Kleene Coalgebra*. Faculty of Science, Mathematics and Computer Science, RU. 2010-08

J.S. de Bruin. *Service-Oriented Discovery of Knowledge - Foundations, Implementations and Applications*. Faculty of Mathematics and Natural Sciences, UL. 2010-09

D. Costa. *Formal Models for Component Connectors*. Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2010-10

M.M. Jaghoori. *Time at Your Service: Schedulability Analysis of Real-Time and Distributed Services*. Faculty of Mathematics and Natural Sciences, UL. 2010-11

R. Bakhshi. *Gossiping Models: Formal Analysis of Epidemic Protocols*. Faculty of Sciences, Department of Computer Science, VUA. 2011-01

B.J. Arnoldus. *An Illumination of the Template Enigma: Software Code Generation with Templates*. Faculty of Mathematics and Computer Science, TU/e. 2011-02

E. Zambon. *Towards Optimal IT Availability Planning: Methods and Tools*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-03

L. Astefanoaei. *An Executable Theory of Multi-Agent Systems Refinement*. Faculty of Mathematics and Natural Sciences, UL. 2011-04

J. Proença. *Synchronous coordination of distributed components*. Faculty of Mathematics and Natural Sciences, UL. 2011-05

A. Moralı. *IT Architecture-Based Confidentiality Risk Assessment in Networks of Organizations*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-06

M. van der Bijl. *On changing models in Model-Based Testing*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-07

C. Krause. *Reconfigurable Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-08

M.E. Andrés. *Quantitative Analysis of Information Leakage in Probabilistic and Nondeterministic Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2011-09

M. Atif. *Formal Modeling and Verification of Distributed Failure Detectors*. Faculty of Mathematics and Computer Science, TU/e. 2011-10

P.J.A. van Tilburg. *From Computability to Executability – A process-theoretic view on automata theory*. Faculty of Mathematics and Computer Science, TU/e. 2011-11

Z. Protic. *Configuration management for models: Generic methods for model comparison and model co-evolution*. Faculty of Mathematics and Computer Science, TU/e. 2011-12

S. Georgievska. *Probability and Hiding in Concurrent Processes*. Faculty of Mathematics and Computer Science, TU/e. 2011-13

S. Malakuti. *Event Composition Model: Achieving Naturalness in Runtime Enforcement*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2011-14

M. Raffelsieper. *Cell Libraries and Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-15

C.P. Tsirogiannis. *Analysis of Flow and Visibility on Triangulated Terrains*. Faculty of Mathematics and Computer Science, TU/e. 2011-16

Y.-J. Moon. *Stochastic Models for Quality of Service of Component Connectors*. Faculty of Mathematics and Natural Sciences, UL. 2011-17

R. Middelkoop. *Capturing and Exploiting Abstract Views of States in OO Verification*. Faculty of Mathematics and Computer Science, TU/e. 2011-18

M.F. van Amstel. *Assessing and Improving the Quality of Model Transformations*. Faculty of Mathematics and Computer Science, TU/e. 2011-19

A.N. Tamalet. *Towards Correct Programs in Practice*. Faculty of Science, Mathematics and Computer Science, RU. 2011-20

H.J.S. Basten. *Ambiguity Detection for Programming Language Grammars*. Faculty of Science, UvA. 2011-21

Bibliography

G. Abowd, R. Allen, and D. Garlan. Formalizing style to understand descriptions of software architecture. *ACM Transactions on Software Engineering and Methodology*, 4 (4):319–364, Oct. 1995. DOI `10.1145/226241.226244`. (cited on page 105).

P. Abrahamsson, J. Warsta, M. T. Siponen, and J. Ronkainen. New directions on agile methods: a comparative analysis. In *Proceedings of the 25th International Conference on Software Engineering*, pages 244–254, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-1877-X. DOI `10.1109/ICSE.2003.1201204`. (cited on page 58).

M. J. Albers. Signal to noise ratio of information in documentation. In *Proceedings of the 22nd annual International Conference on Design of Communication*, pages 41–44, New York, NY, USA, 2004. ACM. ISBN 1-58113-809-1. DOI `10.1145/1026533.1026546`. (cited on page 109).

N. Ali, S. Beecham, and I. Mistrik. Architectural knowledge management in global software development: A review. In *Proceedings of the International Conference on Global Software Engineering*, pages 347–352, Los Alamitos, CA, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4122-8. DOI `10.1109/ICGSE.2010.48`. (cited on page 104).

T. J. Allen. *Managing the Flow of Technology*. MIT Press, 1977. ISBN 978-0262510271. (cited on page 3).

B. Anda and K. Hansen. A case study on the application of UML in legacy development. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering*, pages 124–133. ACM, 2006. ISBN 1-59593-218-6. DOI `10.1145/1159733.1159754`. (cited on page 137).

B. S. Andaloussi and A. Braun. A test specification method for software interoperability tests in offshore scenarios: A case study. In *Proceedings of the International Conference on Global Software Engineering*, pages 169–178, Los Alamitos, CA, USA, 2006. IEEE Computer Society. DOI `10.1109/ICGSE.2006.261230`. (cited on page 162).

E. Arisholm and D. I. K. Sjøberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Transactions on Software Engineering*, 30(8):521–534, 2004. DOI `10.1109/TSE.2004.43`. (cited on page 131).

C. Atkinson and T. Kühne. Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41, 2003. DOI `10.1109/MS.2003.1231149`. (cited on page 10).

M. Babar, R. de Boer, T. Dingsøyr, and R. Farenhorst. Architectural knowledge management strategies: Approaches in research and industry. In *Proceedings of the 29th International Conference on Software Engineering Workshops*, page 2. IEEE Computer Society, 2007. DOI `10.1109/SHARK-ADI.2007.3`. (cited on page 59).

F. Bachmann, L. Bass, J. Carriere, P. Clements, and D. Garlan. Software architecture documentation in practice: Documenting architectural layers. Technical Report CMU/SEI-2000-SR-004, Carnegie Mellon Software Engineering Institute, March 2000. (cited on page 105).

P. Baker, S. Loh, and F. Weil. Model-driven engineering in a large industrial context — Motorola case study. In L. C. Briand and C. Williams, editors, *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 476–491. Springer, 2005. ISBN 3-540-29010-9. DOI `10.1007/11557432_36`. (cited on pages 137 and 138).

K. Balasubramanian, A. Gokhale, G. Karsai, J. Sztipanovits, and S. Neema. Developing applications using model-driven design environments. *Computer*, 39(2):33–40, 2006. DOI `10.1109/MC.2006.54`. (cited on page 155).

M. T. Baldassarre, N. Boffoli, D. Caivano, and G. Visaggio. Speed: Software project effort evaluator based on dynamic-calibration. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 272–273, Los Alamitos, CA, USA, 2006. IEEE Computer Society. DOI `10.1109/ICSM.2006.62`. (cited on page 38).

R. Balzer. Tolerating inconsistency. In *Proceedings of the 13th International Conference on Software Engineering*, pages 158–165, 1991. DOI `10.1109/ICSE.1991.130638`. (cited on page 125).

V. R. Basili and D. M. Weiss. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering*, 10(6):728–738, 1984. (cited on page 14).

V. R. Basili, G. Caldiera, and H. D. Rombach. The goal question metric approach. In *Encyclopedia of Software Engineering*, pages 528–532. Wiley, 1994. (cited on page 108).

R. D. Battin, R. Crocker, J. Kreidler, and K. Subramanian. Leveraging resources in global software development. *IEEE Software*, 18(2):70–77, 2001. DOI `10.1109/52.914750`. (cited on page 23).

C. Bauer and G. King. *Hibernate in Action*. Manning Publications, 8 2004. ISBN 978-1932394153. (cited on page 140).

K. Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999. ISBN 978-0201616415. (cited on page 165).

P. Bengtsson and J. Bosch. Haemo dialysis software architecture design experiences. In *Proceedings of the 21st International Conference on Software Engineering*, pages 516–525. IEEE Computer Society, 1999. DOI `10.1145/302405.302684`. (cited on page 105).

B. Berenbach and M. Gall. Toward a unified model for requirements engineering. In *Proceedings of the International Conference on Global Software Engineering*, pages 237–238, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN 0-7695-2663-2. DOI `10.1109/ICGSE.2006.261238`. (cited on page 161).

B. W. Boehm. A spiral model of software development and enhancement. *SIGSOFT Software Engineering Notes*, 11(4):14–24, 1986. DOI `10.1145/12944.12948`. (cited on pages 8 and 40).

B. W. Boehm. A spiral model of software development and enhancement. *Computer*, 21 (5):61–72, 1988. DOI `10.1109/2.59`. (cited on pages 8 and 40).

B. W. Boehm, M. Abi-Antoun, A. Brown, N. Mehta, and D. Port. Guidelines for the life cycle objectives (LCO) and the life cycle architecture (LCA) deliverables for model-based architecting and software engineering (MBASE). Technical Report USC-CSE-98-519, University of Southern California, Los Angeles, CA, 90089, February 1999. URL `http://sunset.usc.edu/publications/TECHRPTS/1998/usccse98-519/usccse98-519.pdf`. (cited on page 40).

G. Booch. *Object solutions: managing the object-oriented project*. Addison Wesley Longman Publishing Co., Inc., 1995a. ISBN 978-0805305944. (cited on page 6).

G. Booch. *Object Solutions: Managing the Object-Oriented Project*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995b. ISBN 0-8053-0594-7. (cited on page 40).

B. Bowerman and T. Richard. *Linear Statistical Models: An Applied Approach*. PWS-Kent Publishing Co., 1990. ISBN 978-0534380182. (cited on page 128).

L. Bratthall, E. Johansson, and B. Regnell. Is a design rationale vital when predicting change impact? — A controlled experiment on software architecture evolution. *Product Focused Software Process Improvement*, pages 77–100, 2000. DOI `http://dx.doi.org/10.1007/978-3-540-45051-1_14`. (cited on page 6).

M. R. Braz and S. R. Vergilio. Software effort estimation based on use cases. In *Proceedings of the 30th Annual International Computer Software and Applications Conference*, volume 1, pages 221–228, Los Alamitos, CA, USA, 2006. IEEE Computer Society. DOI `10.1109/COMPSAC.2006.77`. (cited on page 38).

L. C. Briand, J. W. Daly, and J. K. Wüst. A unified framework for coupling measurement in object-oriented systems. *IEEE Transactions on Software Engineering*, 25(1):91–121, Jan. 1999. DOI `10.1109/32.748920`. (cited on page 144).

L. C. Briand, Y. Labiche, M. D. Penta, and H. D. Yan-Bondoc. An experimental investigation of formality in UML-based development. *IEEE Transactions on Software Engineering*, 31(10):833–849, 2005. DOI `10.1109/TSE.2005.105`. (cited on pages 108 and 131).

C. Brun and A. Pierantonio. Model differences in the eclipse modeling framework. *UPGRADE, The European Journal for the Informatics Professional*, IX(2):29–34, April 2008. URL `http://www.cepis.org/upgrade/files/2008-II-pierantonio.pdf`. (cited on page 166).

J. Cabot and E. Yu. Improving requirements specifications in model driven development processes. In M. R. Chaudron, editor, *Proceedings of the First International Workshop on Challenges in Model-Driven Software Engineering*, volume 5421 of *Lecture Notes in Computer Science*, pages 36–40. Springer, 2008. ISBN 978-3-642-01647-9. DOI `10.1007/978-3-642-01648-6_4`. (cited on page 171).

J. Cannon-Bowers, E. Salas, and S. Converse. Shared mental models in expert team decision-making. *Environmental Effects of Cognitive Abilities*, pages 221–245, 2001. DOI `10.1016/S0898-1221(00)90067-1`. (cited on pages 64, 65, 158 and 160).

E. Carmel. *Global Software Teams: Collaborating Across Borders and Time Zones*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1999. ISBN 0-13-924218-X. (cited on pages 24 and 174).

E. Carmel and R. Agarwal. Tactical approaches for alleviating distance in global software development. *IEEE Software*, 18(2):22–29, 2001. DOI `10.1109/52.914734`. (cited on pages 2, 3, 4 and 94).

E. Carmel and P. Tjia. *Offshoring Information Technology: Sourcing and Outsourcing to a Global Workforce*. Cambridge University Press, 2005. ISBN 978-0-521-84355-3. (cited on page 3).

R. Carney and J. Levin. Pictorial illustrations still improve students' learning from text. *Educational Psychology Review*, 14(1):5–26, 2002. (cited on page 124).

P. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, 1976. (cited on page 107).

M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko. Let's go to the whiteboard: how and why software developers use drawings. In *Proceedings of the SIGCHI Conference on Human factors in Computing Systems*, pages 557–566, New York, NY, 2007. ACM. DOI 10.1145/1240624.1240714. (cited on pages 7 and 106).

B. R. Chiswick and P. W. Miller. Linguistic distance: A quantitative measure of the distance between English and other languages. Technical Report IZA DP No. 1246, Institute for the Study of Labor (IZA), August 2004. (cited on pages 126 and 127).

M. B. Chrissis, M. Konrad, and S. Shrum. *CMMI Guidelines for Process Integration and Product Improvement*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 0321154967. (cited on page 24).

T. Clear. Documentation and agile methods: striking a balance. *ACM SIGCSE Bulletin*, 35(2):12–13, 2003. DOI 10.1145/782941.782949. (cited on page 58).

P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional, 2002. ISBN 0201703726. (cited on pages 59 and 105).

P. Clements, R. Kazman, M. Klein, D. Devesh, S. Reddy, and P. Verma. The duties, skills, and knowledge of software architects. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 44–47, 2007. DOI 10.1109/WICSA.2007.41. (cited on page 158).

P. C. Clements and M. Shaw. "the golden age of software architecture" revisited. *IEEE Software*, 26(4):70–72, 2009. DOI 10.1109/MS.2009.83. (cited on page 104).

V. Clerc, P. Lago, and H. van Vliet. Global software development: Are architectural rules the answer? In *Proceedings of the IEEE International Conference on Global Software Engineering*, pages 225–234, 2007. DOI 10.1109/ICGSE.2007.21. (cited on page 161).

E. Ó. Conchúir, H. Holmström, P. J. Ågerfalk., and B. Fitzgerald. Exploring the assumed benefits of global software development. In *Proceedings of the International Conference on Global Software Engineering*, pages 159–168, 2006. DOI 10.1109/ICGSE.2006.261229. (cited on page 94).

E. O. Conchúir, P. J. Ågerfalk, H. H. Olsson, and B. Fitzgerald. Global software development: where are the benefits? *Communications of the ACM*, 52(8):127–131, Aug. 2009. DOI 10.1145/1536616.1536648. (cited on pages 2 and 3).

M. Conway. How do committees invent? *Datamation*, 14(4):28–31, 1968. (cited on page 161).

G. Costagliola, F. Ferrucci, G. Tortora, and G. Vitiello. Class point: An approach for the size estimation of object-oriented systems. *IEEE Transactions on Software Engineering*, 31(1):52–74, 2005. DOI 10.1109/TSE.2005.5. (cited on page 141).

B. Curtis, H. Krasner, and N. Iscoe. A field study of the software design process for large systems. *Communications of the ACM*, 31(11):1268–1287, Nov. 1988. DOI 10.1145/50087.50089. (cited on pages 59, 74, 104 and 106).

M. A. Cusumano. Managing software development in globally distributed teams. *Communications of the ACM*, 51(2):15–17, 2008. DOI 10.1145/1314215.1314218. (cited on page 23).

D. Damian, L. Izquierdo, J. Singer, and I. Kwan. Awareness in the wild: Why communication breakdowns occur. In *Proceedings of the Second IEEE International Conference on Global Software Engineering*, pages 81–90, 2007. DOI 10.1109/ICGSE.2007.13. (cited on page 59).

A. De Lucia, C. Gravino, R. Oliveto, and G. Tortora. An experimental comparison of ER and UML class diagrams for data modelling. *Empirical Software Engineering*, pages 1–38, 2010. DOI 10.1007/s10664-009-9127-7. (cited on page 107).

S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the 23rd International Conference on Design of Communication*, pages 68–75, September 2005. DOI 10.1145/1085313.1085331. (cited on page 59).

U. Dekel and J. D. Herbsleb. Notation and representation in collaborative object-oriented design: an observational study. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, volume 42, pages 261–280, New York, NY, USA, 2007. ACM. DOI 10.1145/1297105.1297047. (cited on page 106).

G. Deng, G. Lenz, and D. Schmidt. Addressing domain evolution challenges in software product lines. In J.-M. Bruel, editor, *Proceedings of the Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 247–261, 2006. ISBN 978-3-540-31780-7. DOI 10.1007/11663430_26. (cited on page 173).

K. C. Desouza and J. R. Evaristo. Managing knowledge in distributed projects. *Communications of the ACM*, 47(4):87–91, 2004. DOI 10.1145/975817.975823. (cited on page 24).

K. C. Desouza, Y. Awazu, and P. Baloh. Managing knowledge in global software development efforts: Issues and practices. *IEEE Software*, 23:30–37, 2006. DOI 10.1109/MS.2006.135. (cited on page 56).

B. Dobing and J. Parsons. How UML is used. *Communications of the ACM*, 49(5):109–113, 2006. (cited on page 7).

T. Dybå and T. Dingsøyr. Empirical studies of agile software development: A systematic review. *Information and Software Technology*, 50(9-10):833–859, August 2008. DOI 10.1016/j.infsof.2008.01.006. (cited on page 58).

C. Ebert and P. De Neve. Surviving global software development. *IEEE Software*, 18(2): 62–69, 2001. DOI 10.1109/52.914748. (cited on pages 2 and 94).

C. Ebert, C. H. Parro, R. Suttels, and H. Kolarczyk. Improving validation activities in a global software development. In *Proceedings of the 23rd international Conference on Software Engineering*, pages 545–554, 2001. DOI 10.1109/ICSE.2001.919129. (cited on page 94).

A. H. Eden and R. Kazman. Architecture, design, implementation. In *Proceedings of the 25th International Conference on Software Engineering*, pages 149–159, Piscataway, NJ, USA, 2003. IEEE CS. DOI 10.1109/ICSE.2003.1201196. (cited on page 105).

J. Eder, G. Kappel, and M. Schrefl. Coupling and cohesion in object-oriented systems. *Technical Report, University of Klagenfurt, Austria*, pages 1–34, 1994. (cited on page 2).

A. Egyed. Fixing inconsistencies in UML design models. In *Proceedings of the 29th International Conference on Software Engineering*, pages 292–301. IEEE Computer Society, 2007. DOI 10.1109/ICSE.2007.38. (cited on page 141).

A. J. Espinosa, R. E. Kraut, J. F. Lerch, S. A. Slaughter, J. D. Herbsleb, and A. Mockus. Shared mental models and coordination in large-scale, distributed software development. In *Proceedings of the Twenty-Second International Conference on Information Systems*, pages 513–518, New Orleans, LA, 2001. URL http://www.cs.cmu.edu/afs/cs/Web/People/jdh/collaboratory/research_papers/ICIS_2001.pdf. (cited on pages 65, 158, 160 and 174).

J. A. Espinosa and E. Carmel. Modeling coordination costs due to time separation in global software teams. In *Proceedings of the International Workshop on Global Software Development*, pages 64–68, 2003. URL http://gsd2003.cs.uvic.ca/upload/Espinosa.pdf. (cited on page 94).

R. Farenhorst, J. F. Hoorn, P. Lago, and H. van Vliet. What architects do and what they need to share knowledge. Technical Report Technical Report IR-IMSE-003, VU University Amsterdam, April 2009. (cited on pages 158 and 160).

R. Fish, R. Kraut, R. Root, and R. Rice. Video as a technology for informal communication. *Communications of the ACM*, 36(1):48–61, 1993. DOI 10.1145/142750.142755. (cited on page 59).

E. Flaherty. The thinking aloud technique and problem solving ability. *The Journal of Educational Research*, pages 223–225, 1975. (cited on page 131).

F. Fleurey, E. Breton, B. Baudry, A. Nicolas, and J.-M. Jézéquel. Model-driven engineering for software migration in a large industrial context. In G. Engels, B. Opdyke, D. C. Schmidt, and F. Weil, editors, *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 4735 of *Lecture Notes in Computer Science*, pages 482–497. Springer, 2007. ISBN 978-3-540-75208-0. DOI 10.1007/978-3-540-75209-7_33. (cited on page 137).

A. Forward and T. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the ACM Symposium on Document Engineering*, pages 26–33, 2002. DOI 10.1145/585058.585065. (cited on pages 58 and 105).

R. France and B. Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of the Future of Software Engineering*, pages 37–54, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-2829-5. DOI 10.1109/FOSE.2007.14. (cited on page 155).

J. Galbraith. *Organization design*. Addison-Wesley Pub. Co., Reading, Mass, 1977. ISBN 0201025582. (cited on page 59).

A. Gemino and Y. Wand. Evaluating modeling techniques based on models of learning. *Communications of the ACM*, 46:79–84, October 2003. DOI 10.1145/944217.944243. (cited on pages 107 and 125).

A. Gemino and Y. Wand. Complexity and clarity in conceptual modeling: Comparison of mandatory and optional properties. *Data & Knowledge Engineering*, 55(3):301–326, 2005. DOI 10.1016/j.datak.2004.12.009. (cited on page 107).

M. Genero, L. Jiménez, and M. Piattini. A controlled experiment for validating class diagram structural complexity metrics. In Z. Bellahsène, D. Patel, and C. Rolland, editors, *Proceedings of the 8th International Conference on Object-Oriented Information Systems*, volume 2425 of *Lecture Notes in Computer Science*, pages 483–487, 2002. ISBN 978-3-540-44087-1. DOI 10.1007/3-540-46102-7_40. (cited on page 141).

C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2002. ISBN 0133056996. (cited on page 8).

R. Grinter, J. Herbsleb, and D. Perry. The geography of coordination: dealing with distance in r&d work. In *Proceedings of the International ACM SIGGROUP Conference on Supporting Group Work*, pages 306–315, 1999. DOI 10.1145/320297.320333. (cited on pages 2 and 3).

R. E. Grinter. Doing software development: occasions for automation and formalisation. In *Proceedings of the fifth conference on European Conference on Computer-Supported Cooperative Work*, pages 173–188, Norwell, MA, USA, 1997. Kluwer Academic Publishers. ISBN 0-7923-4638-6. URL http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.106.3437&amp;rep=rep1&amp;type=pdf. (cited on page 24).

M. Hansen, N. Nohria, and T. Tierney. What's your strategy for managing knowledge? *Harvard Business Review*, 77(2):106–116, 1999. (cited on pages 55, 56 and 59).

L. Hart-Gonzalez and S. Lindemann. Expected achievement in speaking proficiency. Technical report, School of Language Studies, Foreign Services Institute, United States Department of State, 1993. (cited on pages 126 and 127).

J. H. Hayes. Do you like Piña Coladas? How improved communication can improve software quality. *IEEE Software*, 20(1):90–92, 2003. DOI `10.1109/MS.2003.1159036`. (cited on page 11).

D. Heckathorn. Respondent-driven sampling: a new approach to the study of hidden populations. *Social Problems*, 44(2):174–199, 1997. (cited on page 92).

D. Heckathorn. Respondent-driven sampling II: deriving valid population estimates from chain-referral samples of hidden populations. *Social Problems*, pages 11–34, 2002. (cited on page 92).

M. Hegarty and M. A. Just. Constructing mental models of machines from text and diagrams. *Journal of Memory and Language*, 32(6):717–742, December 1993. (cited on page 130).

W. Heijstek and M. R. V. Chaudron. Effort distribution in model-based development. In *Proceedings of the 2nd workshop on Model Size Metrics*, pages 26–38, October 2007. URL `http://www.win.tue.nl/~clange/MSM2007/MSM2007.pdf`. (cited on pages 16, 39 and 96).

W. Heijstek and M. R. V. Chaudron. Exploring effort distribution in RUP projects. In *Proceedings of the 2nd International Symposium on Empirical Software Engineering and Measurement*, page 359, 2008a. DOI `10.1145/1414004.1414083`. (cited on page 16).

W. Heijstek and M. R. V. Chaudron. Evaluating RUP software development processes through visualization of effort distribution. In *Proceedings of the 34th Euromicro Conference on Software Engineering and Advanced Applications*, pages 266–273, 2008b. DOI `10.1109/SEAA.2008.43`. (cited on page 16).

W. Heijstek and M. R. V. Chaudron. Empirical investigations of model size, complexity and effort in large scale, distributed model driven development processes — a case study. In *Proceedings of the 35th Euromicro Conference on Software Engineering and Advanced Applications*, pages 113–120, 2009. DOI `10.1109/SEAA.2009.70`. (cited on pages 17 and 84).

W. Heijstek and M. R. V. Chaudron. The impact of model driven development on the software architecture process. In *Proceedings of the 36th Euromicro Conference on Software Engineering and Advanced Applications*, pages 333–341, 2010. DOI `10.1109/SEAA.2010.63`. (cited on pages 17, 84 and 87).

W. Heijstek and M. R. V. Chaudron. On the use of UML diagrams in industrial software architecture documents. Technical Report TR2011-02, Leiden Institute of Advanced Computer Science (LIACS) / Faculty of Science, Leiden University, 2011. (cited on pages 105 and 124).

W. Heijstek, M. R. Chaudron, L. Qiu, and C. C. Schouten. A comparison of industrial process descriptions for global custom software development. In *Proceedings of the International Conference on Global Software Engineering*, pages 277–284, Los Alamitos, CA, USA, 2010. IEEE Computer Society. ISBN 978-0-7695-4122-8. DOI 10.1109/ICGSE.2010.39. (cited on pages 16 and 174).

W. Heijstek, T. Kühne, and M. R. V. Chaudron. Experimental analysis of textual and graphical representations for software architecture design. In *Proceedings of the 5th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 167–176, September 2011. DOI 10.1109/ESEM.2011.25. (cited on page 17).

T. Heistracher, T. Kurz, G. Marcon, and C. Masuch. Collaborative software engineering with a digital ecosystem. In *Proceedings of the International Conference on Global Software Engineering*, pages 119–126, 2006. ISBN 0-7695-2663-2. (cited on page 161).

J. D. Herbsleb and R. E. Grinter. Architectures, coordination, and distance: Conway's law and beyond. *IEEE Software*, 16(5):63–70, 1999a. DOI 10.1109/52.795103. (cited on pages 4 and 59).

J. D. Herbsleb and R. E. Grinter. Splitting the organization and integrating the code: Conway's law revisited. In *Proceedings of the 21st International Conference on Software Engineering*, pages 85–95, Los Angeles, CA, USA, 1999b. URL http://portal.acm.org/citation.cfm?id=302405.302455. (cited on page 2).

J. D. Herbsleb and D. Moitra. Global software development. *IEEE Software*, 18(2):16–20, March/April 2001. DOI 10.1109/52.914732. (cited on page 106).

J. D. Herbsleb, A. Mockus, T. A. Finholt., and R. E. Grinter. Distance, dependencies, and delay in a global collaboration. In *Proceedings of the 2000 ACM conference on Computer supported cooperative work*, pages 319–328, New York, NY, USA, 2000. ACM. ISBN 1-58113-222-0. DOI 10.1145/358916.359003. (cited on pages 2, 3, 23, 173 and 174).

J. D. Herbsleb, A. Mockus, T. A. Finholt, and R. E. Grinter. An empirical study of global software development: distance and speed. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 81–90, 2001. URL http://mockus.org/papers/speed.pdf. (cited on pages 59 and 94).

J. D. Herbsleb, D. J. Paulish, and M. Bass. Global software development at siemens: experience from nine projects. In *Proceedings of the International Conference on Software Engineering*, pages 524–533. ACM, 2005. DOI 10.1109/ICSE.2005.1553598. (cited on page 4).

J. Highsmith and M. Fowler. The agile manifesto. *Software Development Magazine*, 9(8): 29–30, 2001. (cited on pages 56, 58 and 105).

A. Hindle, M. Godfrey, and R. Holt. Software process recovery using recovered unified process views. In *Proceedings of the IEEE International Conference on Software Maintenance*, pages 1–10, 2010. DOI 10.1109/ICSM.2010.5609670. (cited on pages 39, 40 and 41).

M. Hirsch. Making RUP agile. In *Proceedings of the OOPSLA 2002 Practitioners Reports*, pages 1–8, New York, NY, USA, 2002. ACM. ISBN 1-58113-471-1. DOI 10.1145/604251.604254. (cited on page 105).

R. R. Hocking. The analysis and selection of variables in linear regression. *Biometrics*, 32(1):1–49, March 1976. (cited on page 126).

C. Hofmeister. Technical Committee (TC) 2 — SOFTWARE: Theory and Practice — Aims and Scopes of Workgroup (WG) 2.10 — Software Architecture. Technical Report WG210, International Federation for Information Processing (IFIP), 2000. URL http://www.ifip.org/bulletin/bulltcs/tc2_aim.htm#wg210. (cited on pages 59 and 104).

C. Hofmeister, R. L. Nord, and D. Soni. Describing software architecture with UML. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture*, volume 140, pages 145–160. Kluwer, 1999. ISBN 0-7923-8453-9. URL www.users.abo.fi/lpetre/SA10/paper99.pdf. (cited on page 105).

G. Hofstede. *Culture's Consequences: International Differences in Work-related Values*. Sage Publications, Inc, 1984. ISBN 0-8039-1306-0. (cited on page 4).

H. Holmström, E. Ó. Conchúir, P. J. Ågerfalk, and B. Fitzgerald. Global software development challenges: A case study on temporal, geographical and socio-cultural distance. In *Proceedings of the First IEEE International Conference on Global Software Engineering*, pages 3–11. IEEE, October 2006. DOI 10.1109/ICGSE.2006.261210. (cited on page 3).

J. Holsanova, N. Holmberg, and K. Holmqvist. Reading information graphics: The role of spatial contiguity and dual attentional guidance. *Applied Cognitive Psychology*, 23 (9):1215–1226, 2009. (cited on page 132).

R. Holt. Software architecture as a shared mental model. In *Proceedings of the ASERC Workhop on Software Architecture, University of Alberta*, 2002. URL http://plg.uwaterloo.ca/~holt/papers/sw-arch-mental-model-020314c-1.pdf. (cited on pages 64 and 106).

M. Höst, B. Regnell, and C. Wohlin. Using students as subjects—a comparative study of students and professionals in lead-time impact assessment. *Empirical Software Engineering*, 5(3):201–214, 2000. DOI 10.1023/A:1026586415054. (cited on page 131).

M. Höst, C. Wohlin, and T. Thelin. Experimental context classification: incentives and experience of subjects. In *Proceedings of the 27th international conference on Software engineering*, pages 470–478, New York, NY, USA, 2005. ACM. ISBN 1-59593-963-2. DOI 10.1145/1062455.1062539. (cited on page 127).

B. C. Hungerford, A. R. Hevner, and R. W. Collins. Reviewing software diagrams: A cognitive study. *IEEE Transactions on Software Engineering*, 30(2):82–96, 2004. URL http://csdl.computer.org/comp/trans/ts/2004/02/e0082abs.htm. (cited on page 106).

J. M. Hussey and S. E. Hall. *Managing Global Development Risk*. Auerbach Publications, 1st edition, November 2007. ISBN 978-1420055207. (cited on page 23).

J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical assessment of MDE in industry. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 471–480, 2011. DOI 10.1145/1985793.1985858. (cited on pages 137 and 138).

M. Huysman and V. Wulf. IT to support knowledge sharing in communities, towards a social capital analysis. *Journal of Information Technology*, 21(1):40–51, 2005. DOI 10.1057/palgrave.jit.2000053. (cited on page 57).

IEEE. 1471-2000: Recommended practice for architectural description of software-intensive systems. Technical report, IEEE, 2000. (cited on pages 5 and 105).

International Software Benchmarking Standards Group. The benchmark-release 9. Technical report, ISBSG, Warrandyte, Australia, 2006. URL http://www.isbsg.org/. (cited on page 85).

E. Isaacs and J. Tang. What video can and cannot do for collaboration: a case study. In *Proceedings of the First ACM International Conference on Multimedia*, volume 2, pages 63–73. ACM Press, 1994. DOI 10.1145/166266.166289. (cited on page 4).

ISO/IEC. 19501:2005 Information technology — Open Distributed Processing — Unified Modeling Language (UML) Version 1.4.2. Technical report, ISO/IEC, 2005. (cited on page 6).

ISO/IEC. 12007:2008 Systems and software engineering — Software Life Cycle Processes. Technical report, ISO/IEC, 2008. (cited on page 22).

ISO/IEC/IEEE. 42010:2011 Systems and software engineering — Architecture description. Technical report, ISO/IEC/IEEE, Oct. 2011. (cited on page 5).

K. Iwata, T. Nakashima, Y. Anan, and N. Ishii. Improving accuracy of multiple regression analysis for effort prediction model. In *Proceedings of the first IEEE/ACIS International Workshop on Component-Based Software Engineering, Software Architecture and Reuse (COMSAR 2006)*, volume 1, pages 48–55, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN 0-7695-2613-6. DOI 10.1109/ICIS-COMSAR.2006.46. (cited on page 38).

I. Jacobson. Object-oriented development in an industrial environment. In *Proceedings of the Conference on Object-oriented programming systems, languages and applications*, pages 183–191, New York, NY, USA, 1987. ACM Press. ISBN 0-89791-247-0. DOI 10.1145/38765.38824. (cited on page 8).

I. Jacobson, M. Christenson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering, A Use Case Driven Approach*. Addison-Wesley, Reading, Massachusetts, USA, July 1992. ISBN 978-0201544350. (cited on pages 6 and 8).

I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0-201-57169-2. (cited on page 40).

A. Jansen and J. Bosch. Software architecture as a set of architectural design decisions. In *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture*, pages 109–120. IEEE Computer Society, 2005. ISBN 0-7695-2548-2. DOI 10.1109/WICSA.2005.61. (cited on page 105).

M. Jiménez, M. Piattini, and A. Vizcaíno. Challenges and improvements in distributed software development: A systematic review. *Advances in Software Engineering*, pages 1–15, 2009. DOI 10.1155/2009/710971. (cited on pages 23, 136 and 158).

S. Johnson and C. Shih-Ping. The effect of thinking aloud pair problem solving (TAPPS) on the troubleshooting ability of aviation technician students. *Journal of Industrial Teacher Education*, 37(1):7–25, 1999. URL http://scholar.lib.vt.edu/ejournals/JITE/v37n1/john.html. (cited on page 131).

C. Jones. *Software Assessments, Benchmarks, and Best Practices*. Addison-Wesley Longman Publishing Co., Inc., 2000. ISBN 978-0201485424. (cited on page 85).

M. I. Kellner, R. J. Madachy, and D. M. Raffo. Software process simulation modeling: why? what? how? *Journal of Systems and Software*, 46(2):91–105, 1999. DOI 10.1016/S0164-1212(99)00003-5. (cited on page 96).

H. Kim and C. Boldyreff. Developing software metrics applicable to UML models. In *Proceedings of the 6th ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, 2002. URL http://alarcos.inf-cr.uclm.es/qaoose2002/docs/QAOOSE-Kim-Bol.pdf. (cited on page 141).

A. G. Kleppe, J. Warmer, and W. Bast. *MDA Explained: The Model Driven Architecture: Practice and Promise*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003. ISBN 032119442X. (cited on pages 137 and 160).

J. Knodel, D. Muthig, and M. Naab. An experiment on the role of graphical elements in architecture visualization. *Empirical Software Engineering*, 13(6):693–726, 2008. DOI 10.1007/s10664-008-9069-5. (cited on page 108).

P. Kogut and P. Clements. The software architecture renaissance. *Crosstalk-The Journal of Defense Software Engineering*, 7(11):1–5, 1994. URL `http://www.sei.cmu.edu/library/assets/sw_arch_renaissance.pdf`. (cited on page 5).

A. G. Koru, D. Zhang, K. El Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, March-April 2009. DOI `10.1109/TSE.2008.90`. (cited on page 153).

G. Koru, H. Liu, D. Zhang, and K. Emam. Testing the theory of relative defect proneness for closed-source software. *Empirical Software Engineering*, 15(6):577–598, Dec. 2010. DOI `10.1007/s10664-010-9132-x`. (cited on page 153).

J. Kotlarsky, P. C. van Fenema, and l. P. Willcocks. Developing a knowledge-based perspective on coordination: the case of global software projects. *Information and Management*, 45:96–108, 2008. DOI `10.1016/j.im.2008.01.001`. (cited on page 11).

R. E. Kraut and L. A. Streeter. Coordination in software development. *Communications of the ACM*, 38(3):69–81, March 1995. DOI `10.1145/203330.203345`. (cited on pages 59 and 106).

P. Kruchten. The 4 + 1 view model of architecture. *IEEE Software*, 12(6):42–50, Nov. 1995. DOI `10.1109/52.469759`. (cited on pages 5, 56 and 75).

P. Kruchten. A brief history of the RUP “hump chart”. Technical report, University of British Columbia, 2003a. (cited on page 40).

P. Kruchten. *The Rational Unified Process: An Introduction*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003b. ISBN 0321197704. (cited on pages 8, 38, 40, 73 and 139).

P. Kruchten, P. Lago, H. van Vliet, and T. Wolf. Building up and exploiting architectural knowledge. In *Proceedings of the Working IEEE/IFIP Conference on Software Architecture*, pages 291–292. IEEE Computer Society, 2005. ISBN 0-7695-2548-2. DOI `10.1109/WICSA.2005.19`. (cited on page 158).

S. Kvale. *InterViews: An Introduction to Qualitative Research Interviewing*. Sage Publications, 1996. ISBN 978-0803958203. (cited on page 15).

C. F. J. Lange. Model size matters. In T. Kühne, editor, *Proceedings of the MoDELS 2006 Workshops*, volume 4364 of *Lecture Notes in Computer Science*, pages 211–216. Springer, 2006. ISBN 978-3-540-69488-5. DOI `10.1007/978-3-540-69489-2_26`. (cited on pages 106 and 141).

C. F. J. Lange and M. R. V. Chaudron. Managing model quality in UML-based software development. In *Proceedings of the 13th IEEE International Workshop on Software Technology and Engineering Practice*, pages 7–16. IEEE Computer Society, 2005. ISBN 0-7695-2639-X. DOI `10.1109/STEP.2005.16`. (cited on page 141).

C. F. J. Lange and M. R. V. Chaudron. Effects of defects in UML models: an experimental investigation. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pages 401–411. ACM, 2006. ISBN 1-59593-375-1. DOI 10.1145/1134285.1134341. (cited on pages 107 and 125).

C. F. J. Lange, M. R. V. Chaudron, J. Muskens, L. J. Somers, and H. M. Dortmans. An empirical investigation in quantifying inconsistency and incompleteness of UML designs. In *Proceedings of the Workshop on Consistency Problems in UML-based Software Development*, pages 26–34, 2003. (cited on pages 106 and 169).

C. F. J. Lange, M. R. V. Chaudron, and J. Muskens. In practice: UML software architecture and design description. *IEEE Software*, 23(2):40–46, 2006. DOI 10.1109/MS.2006.50. (cited on page 166).

T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, pages 492–501, Shanghai, China, 2006. ACM. ISBN 1-59593-375-1. DOI 10.1145/1134355. (cited on page 106).

G. Lee, W. DeLone, and J. Espinosa. Ambidextrous coping strategies in globally distributed software development projects. *Communications of the ACM*, 49(10):35–40, 2006. (cited on pages 80 and 104).

N. Lester and F. Wilkie. Evaluating UML tool support for effective coordination and communication across geographically disparate sites. In *Proceedings of the International Workshop on Software Technology and Engineering Practice*, pages 57–64, Los Alamitos, CA, USA, 2004. IEEE Computer Society. ISBN 0-7695-2293-9. DOI 10.1109/STEP.2004.10. (cited on page 161).

T. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, 2003. DOI 10.1109/MS.2003.1241364. (cited on pages 58, 105 and 107).

L. L. Levesque, J. M. Wilson, and D. R. Wholey. Cognitive divergence and shared mental models in software development project teams. *Journal of Organizational Behavior*, 22:135–144, 2001. (cited on page 174).

J. R. Levin, G. J. Anglin, and R. N. Carney. *The Psychology of Illustration: I. Basic Research*, chapter On empirically validating functions of pictures in prose, pages 51—85. Springer, New York, 1987. (cited on page 124).

W. Li and S. Henry. Object-oriented metrics that predict maintainability. *The Journal of Systems and Software*, 23(2):111–122, Nov. 1993. (cited on page 144).

Y. Lin, J. Gray, and F. Jouault. DSMDiff: a differentiation tool for domain-specific models. *European Journal of Information Systems*, 16(4):349–361, 2007. DOI doi:10.1057/palgrave.ejis.3000685. (cited on page 166).

B. Lings, B. Lundell, P. J. Ågerfalk, and B. Fitzgerald. A reference model for successful distributed development of software systems. In *Proceedings of the Second IEEE International Conference on Global Software Engineering*, pages 130–139, 2007. DOI 10.1109/ICGSE.2007.6. (cited on page 161).

C. Lopez-Martin, C. Yanez-Marquez, and A. Gutierrez-Tornes. A fuzzy logic model based upon reused and new & changed code for software development effort estimation at personal level. In *Proceedings of the 15th International Conference on Computing*, pages 298–303, Los Alamitos, CA, USA, 2006. IEEE Computer Society. ISBN 0-7695-2708-6. DOI 10.1109/CIC.2006.5. (cited on page 38).

M. Lorenz and J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall, Englewood Cliffs, 1994. ISBN 0-13-179292-X. (cited on page 143).

A. MacDonald, D. M. Russell, and B. Atchison. Model-driven development within a legacy system: An industry experience report. In *Proceedings of the Australian Software Engineering Conference*, pages 14–22, Los Alamitos, CA, USA, 2005. IEEE Computer Society. ISBN 0-7695-2257-2. DOI 10.1109/ASWEC.2005.32. (cited on page 138).

M. Marchesi. OOA metrics for the unified modeling language. In *Proceedings of the 2nd Euromicro Conference on Software Maintenance and Reengineering*, pages 67–73. IEEE Computer Society, 1998. ISBN 0-8186-8421-6. URL http://csdl.computer.org/comp/proceedings/csmr/1998/8421/00/84210067abs.htm. (cited on page 141).

J. Marciniak, editor. *Encyclopedia of software engineering*. John Wiley & Sons, Inc., 2002. ISBN 978-0471028956. DOI 10.1002/0471028959. (cited on page 22).

J. Mathieu, T. Heffner, G. Goodwin, E. Salas, and J. Cannon-Bowers. The influence of shared mental models on team process and performance. *Journal of Applied Psychology*, 85(2):273, 2000. DOI 10.1037/0021-9010.85.2.273. (cited on page 65).

R. E. Mayer. *Multimedia Learning*. Cambridge University Press, second edition, 2009. ISBN 978-0521735353. (cited on page 107).

M. McDaniel and M. Pressley. *Imagery and Related Mnemonic Processes: Theories, Individual Differences, and Applications*. Springer-Verlag Publishing, 1987. ISBN 978-1461291114. (cited on page 124).

C. McNamara. General guidelines for conducting interviews. Technical report, Authenticity Consulting, LLC, Minneapolis, MN, 1999. URL http://managementhelp.org/businessresearch/interviews.htm. (cited on page 15).

N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, and J. E. Robbins. Modeling software architectures in the Unified Modeling Language. *ACM Transaction on Software Engineering and Methodology*, 11(1):2–57, Jan. 2002. DOI 10.1145/504087.504088. (cited on page 106).

T. Menzies, Z. Chen, J. Hihn, and K. Lum. Selecting best practices for effort estimation. *IEEE Transactions on Software Engineering*, 32(11):883–895, 2006. DOI 10.1109/TSE.2006.114. (cited on page 38).

D. Milicic and C. Wohlin. Distribution patterns of effort estimations. In *Proceedings of the 30th Euromicro Conference on Software Engineering and Advanced Applications*, pages 422–429, Los Alamitos, CA, USA, 2004. IEEE Computer Society. DOI 10.1109/EURMIC.2004.1333398. (cited on page 38).

A. Mockus and J. D. Herbsleb. Challenges of global software development. In *Proceedings of the IEEE International Symposium on Software Metrics*, page 182, Los Alamitos, CA, USA, 2001. IEEE Computer Society. DOI 10.1109/METRIC.2001.915526. (cited on pages 23 and 174).

A. Mockus and D. M. Weiss. Globalization by chunking: A quantitative approach. *IEEE Software*, 18(2):30–37, 2001. DOI 10.1109/52.914737. (cited on pages 11, 23, 50 and 56).

MODELWARE D5.3-1. Industrial ROI, Assessment, and Feedback — Master Document. Technical Report Revision 2.2, Master Document, 2006. URL http://www.modelware-ist.org. (cited on page 138).

P. Mohagheghi and V. Dehlen. Where is the proof? — A review of experiences from applying MDE in industry. In I. Schieferdecker and A. Hartman, editors, *Proceedings of the Fourth European Conference on Model Driven Architecture: Foundations and Applications*, volume 5095 of *Lecture Notes in Computer Science*, pages 432–443. Springer, 2008. ISBN 978-3-540-69095-5. DOI 10.1007/978-3-540-69100-6_31. (cited on pages 137, 138 and 168).

M. Müller and W. Tichy. Case study: extreme programming in a university environment. In *Proceedings of the 23rd international Conference on Software Engineering*, pages 537–544, 2001. DOI 10.1109/ICSE.2001.919128. (cited on page 106).

N. Mullick, M. Bass, Z. Houda, P. Paulish, and M. Cataldo. Siemens global studio project: Experiences adopting an integrated GSD infrastructure. In *Proceedings of the IEEE international conference on Global Software Engineering*, pages 203–212, 2006. URL www.casos.cs.cmu.edu/publications/papers/cataldo_p4.pdf. (cited on page 161).

P. Naur and B. Randell, editors. *Software Engineering: Report of a conference sponsored by the NATO Science Committee, Garmisch, Germany*, October 1968. Brussels, Scientific Affairs Division, NATO. URL http://homepages.cs.ncl.ac.uk/brian.randell/NATO/nato1968.PDF. (cited on page 1).

T. Nguyen, T. Wolf, and D. Damian. Global software development and delay: Does distance still matter? In *Proceedings of the IEEE International Conference on Global Software Engineering*, pages 45–54, 2008. DOI 10.1109/ICGSE.2008.39. (cited on page 59).

C. Nielson. An empirical examination of the role of "closeness" in industrial buyer-seller relationships. *European Journal of Marketing*, 32(5/6):441–463, 1998. DOI 10.1108/03090569810215812. (cited on page 59).

G. E. Noether. Why Kendall tau? *Teaching Statistics*, 3(2):41–43, 1981. DOI 10.1111/j.1467-9639.1981.tb00422.x. (cited on page 127).

M. E. Nordberg III. Managing code ownership. *IEEE Software*, 20:26–33, 2003. DOI 10.1109/MS.2003.1184163. (cited on page 59).

A. Nugroho. Level of detail in UML models and its impact on model comprehension: A controlled experiment. *Information & Software Technology*, 51(12):1670–1685, 2009. DOI 10.1016/j.infsof.2009.04.007. (cited on page 125).

A. Nugroho and M. R. V. Chaudron. A survey into the rigor of UML use and its perceived impact on quality and productivity. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 90–99, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-971-5. DOI 10.1145/1414004.1414020. (cited on page 106).

A. Nugroho and M. R. V. Chaudron. Evaluating the impact of UML modeling on software quality: An industrial case study. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, pages 181–195, Denver, CO, USA, 2009. Springer. ISBN 978-3-642-04424-3. DOI 10.1007/978-3-642-04425-0. (cited on page 106).

A. Nugroho and C. F. J. Lange. On the relation between class-count and modeling effort. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems*, volume 5002, pages 93–104. Springer, 2007. ISBN 978-3-540-69069-6. DOI 10.1007/978-3-540-69073-3_11. (cited on page 141).

Object Management Group. Technical Guide to Model Driven Architecture: The MDA Guide v1.0.1. Technical report, Object Management Group, June 2003a. URL http://www.omg.org/cgi-bin/apps/doc?omg/03-06-01.pdf. (cited on page 160).

Object Management Group. *MDA Guide Version 1.0.1*. Object Management Group, Framingham, Massachusetts, June 2003b. (cited on page 160).

Object Management Group. Meta Object Facility (MOF) Core Specification Version 2.0. Technical report, Object Management Group, 2006. URL http://www.omg.org/spec/MOF/. (cited on page 160).

Object Management Group. *XML Metadata Interchange (XMI)*. Object Management Group, 2007. URL http://www.omg.org/spec/XMI/. (cited on pages 85 and 160).

Object Management Group. Business Process Model and Notation (BPMN) Version 1.2. Technical Report formal/2009-01-03, Object Management Group, 140 Kendrick Street, Building A, Suite 300, Needham, MA 02494 USA, January 2009. URL http://www.omg.org/spec/BPMN/1.2. (cited on page 25).

Object Management Group. Unified modeling language superstructure. Technical Report formal/2011-08-05, Object Management Group, August 2011. URL http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF. (cited on page 6).

J. Oei, L. Van Hemmen, E. Falkenberg, and S. Brinkkemper. The meta model hierarchy: a framework for information systems concepts and techniques. Technical Report No. 92-17, Department of Informatics, Faculty of Mathematics and Informatics, Katholieke Universiteit, Nijmegen, the Netherlands, July 1992. (cited on page 125).

T. Ohno. *Toyota Production System: Beyond Large-Scale Production*. Productivity Press, March 1988. ISBN 978-0915299140. (cited on page 49).

A. N. Oppenheim. *Questionnaire Design, Interviewing and Attitude Measurement*. Heinemann Educational Books, 1966. ISBN 0-8264-5176-4. (cited on page 112).

C. O'Neill. Delivering systems faster with less risk: The macro-iterative dimension of RUP. *The Rational Edge*, September 2007. URL https://www.ibm.com/developerworks/rational/library/sep07/oneill/?ca=drs. (cited on page 40).

C. Pahl. Layered ontological modelling for web service-oriented model-driven architecture. In A. Hartman and D. Kreische, editors, *Model Driven Architecture — Foundations and Applications*, volume 3748 of *Lecture Notes in Computer Science*, pages 88–102. Springer, 2005. ISBN 978-3-540-30026-7. DOI 10.1007/11581741_8. (cited on page 161).

M. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented computing. *Communications of the ACM*, 46(10):25–28, 2003. DOI 10.1145/944217.944233. (cited on page 65).

D. Parnas and P. Clements. A rational design process: How and why to fake it. In H. Ehrig, C. Floyd, M. Nivat, and J. Thatcher, editors, *Formal Methods and Software Development*, volume 186 of *Lecture Notes in Computer Science*, pages 80–100. Springer, 1985. ISBN 978-3-540-15199-9. DOI 10.1007/3-540-15199-0_6. (cited on page 22).

D. Perry and A. Wolf. Foundations for the study of software architecture. *ACM SIGSOFT Software Engineering Notes*, 17(4):40–52, 1992. DOI 10.1145/141874.141884. (cited on page 5).

G. Pollice. Using the RUP for small projects: Expanding upon eXtreme Programming. Technical report, Rational Software, 2001. (cited on page 105).

M. Poppendieck and T. Poppendieck. *Lean software development: An agile toolkit*. Addison-Wesley Professional, 2003. ISBN 978-0321150783. (cited on page 49).

D. Port, Z. Chen, and P. Kruchten. An empirical validation of the RUP "hump" diagram. In *Proceedings of the 4th International Symposium on Empirical Software Engineering*, 2005. (cited on pages 39 and 40).

M. E. Porter. *Competitive Advantage: Creating and Sustaining Superior Performance*. Free Press, New York, 1985. ISBN 0029250900. (cited on page 3).

R. Premraj, M. Shepperd, B. Kitchenham, and P. Forselius. An empirical analysis of software productivity over time. In *Proceedings of the 11th IEEE International Symposium on Software Metrics*, pages 10–37, 2005. DOI 10.1109/METRICS.2005.8. (cited on page 85).

K. H. Pries and J. M. Quigley. *Scrum project management*. CRC Press, 2010. ISBN 978-1439825150. URL http://www.crcpress.com/product/isbn/9781439825150. (cited on page 58).

R. Prikladnicki, J. L. N. Audy, and R. Evaristo. A reference model for global software development: Findings from a case study. *International Conference on Global Software Engineering*, pages 18–28, 2006. DOI 10.1109/ICGSE.2006.261212. (cited on page 21).

L. Qiu. A comparison of process descriptions and specifications in the offshored software development industry. Master's thesis, M.Sc. ICT in Business, Leiden University, August 2009. (cited on page 27).

R Development Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2011. URL http://www.R-project.org/. ISBN 3-900051-07-0. (cited on page 43).

P. J. Ågerfalk and B. Fitzgerald. Introduction: Special issue: Flexible and distributed software processes: old petunias in new bowls? *Communications of the ACM*, 49(10): 26–34, Oct. 2006. DOI 10.1145/1164394.1164416. (cited on page 58).

P. J. Ågerfalk, B. Fitzgerald, H. Holmström, B. Lings, B. Lundell, and E. Ó. Conchúir. A framework for considering opportunities and threats in distributed software development. In *Proceedings of the International Workshop on Distributed Software Development*, pages 47–61, 2005. URL http://www.idi.ntnu.no/grupper/su/bibliography/pdf/OpenSource/print_DiSD05-Agerfalk-etal_CamReady.pdf. (cited on page 3).

C. Raistrick. Applying MDA and UML in the development of a healthcare system. In N. J. Nunes, B. Selic, A. R. da Silva, and J. A. T. Álvarez, editors, *Proceedings of the UML 2004 Satellite Activities*, volume 3297 of *Lecture Notes in Computer Science*, pages 203–218. Springer, 2004. ISBN 3-540-25081-6. DOI 10.1007/978-3-540-31797-5_21. (cited on page 137).

T. Reus, H. Geers, and A. van Deursen. Harvesting software systems for MDA-based reengineering. In A. Rensink and J. Warmer, editors, *Proceedings of the Second European Conference on Model Driven Architecture - Foundations and Applications*, volume 4066 of *Lecture Notes in Computer Science*, pages 213–225. Springer, 2006. ISBN 3-540-35909-5. DOI 10.1007/11787044_17. (cited on page 137).

J. Rosik, A. L. Gear, J. Buckley, M. A. Babar, and D. Connolly. Assessing architectural drift in commercial software development: a case study. *Software — Practice and Experience*, 12:63–86, November 2010. DOI 10.1002/spe.999. (cited on pages 81 and 100).

W. Royce. Managing the development of large software systems. In *Proceedings of IEEE WESCON*, volume 26, pages 1–9, 1970. URL www.cs.huji.ac.il/~feit/sem/se09/Waterfall.pdf. (cited on page 22).

W. Royce. *Software Project Management: A Unified Framework*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1998. ISBN 0-201-30958-0. (cited on page 40).

E. Rubin and H. Rubin. Supporting agile software development through active documentation. *Requirements Engineering*, 16:117–132, 2011. DOI 10.1007/s00766-010-0113-9. (cited on page 58).

J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented modeling and design*, volume 38. Prentice hall, 1990. ISBN 978-0136298410. (cited on page 6).

P. Runeson and M. Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–161, April 2009. DOI 10.1007/s10664-008-9102-8. (cited on page 13).

P. Runeson, M. Höst, A. Rainer, and B. Regnell. *Case Study Research in Software Engineering — Guidelines and Examples*. John Wiley & Sons, 1 edition, April 2012. ISBN 978-1-118-10435-4. (cited on page 13).

B. Russell. *A History of Western Philosophy*. Simon & Schuster/Touchstone, 1945. ISBN 978-0671201586. (cited on page 12).

S. Sahay, B. Nicholson, and K. S. *Global IT Outsourcing: Software Development Across Borders*. Cambridge University Press, 2003. ISBN 978-0521039482. (cited on pages 2 and 3).

F. Salger. On the use of handover checkpoints to manage the global software development process. In R. Meersman, P. Herrero, and T. S. Dillon, editors, *Proceedings of the OTM Workshops*, volume 5872 of *Lecture Notes in Computer Science*, pages 267–276. Springer, 2009. ISBN 978-3-642-05289-7. DOI 10.1007/978-3-642-05290-3. (cited on pages 2 and 23).

G. Scanniello, C. Gravino, M. Risi, and G. Tortora. A controlled experiment for assessing the contribution of design pattern documentation on software maintenance. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–4, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0039-1. DOI 10.1145/1852786.1852853. (cited on page 106).

I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch. The UML 2.0 testing profile and its relation to TTCN-3. In D. Hogrefe and A. Wiles, editors, *Proceedings of the 15th IFIP international conference on Testing of communicating systems*, volume 2644 of *Lecture Notes in Computer Science*, pages 79–94. Springer, 2003. ISBN 978-3-540-40123-0. DOI 10.1007/3-540-44830-6_7. (cited on page 162).

K. Schneider, K. Stapel, and E. Knauss. Beyond documents: visualizing informal communication. In *Proceedings of Requirements Engineering Visualization*, pages 31–40, 2008. DOI 10.1109/REV.2008.1. (cited on page 59).

K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2001. ISBN 0130676349. (cited on page 58).

B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003. DOI 10.1109/MS.2003.1231146. (cited on pages 10 and 136).

C. Shannon and W. Weaver. A mathematical theory of communication. *University of Illinois Press, Urbana, Ill*, pages 3–55, 1949. DOI 10.1145/584091.584093. (cited on page 60).

S. Sharma and G. Seshagiri. Point/counterpoint: Making global software development work – GSD: Not a business necessity, but a march of folly. *IEEE Software*, 23(5): 62–65, 2006. DOI 10.1109/MS.2006.138. (cited on page 11).

M. Shaw and P. C. Clements. The golden age of software architecture. *IEEE Software*, 23(2):31–39, 2006. DOI 10.1109/MS.2006.58. (cited on page 104).

D. Shirtz, M. Kazakov, and Y. Shaham-Gafni. Adopting model driven development in a large financial organization. In D. H. Akehurst, R. Vogel, and R. F. Paige, editors, *Proceedings of the Third European Conference on Model Driven Architecture: Foundations and Applications*, volume 4530 of *Lecture Notes in Computer Science*, pages 172–183. Springer, 2007. ISBN 978-3-540-72900-6. DOI 10.1007/978-3-540-72901-3_13. (cited on page 138).

B. Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, pages 336–343, 1996. DOI 10.1109/VL.1996.545307. (cited on page 122).

A. Sillitti, M. Ceschi, B. Russo, and G. Succi. Managing uncertainty in requirements: A survey in documentation-driven and agile companies. In *Proceedings of the 11th IEEE International Symposium on Software Metrics*, pages 10–17, Washington, DC, USA, 2005. IEEE Computer Society. ISBN 0-7695-2371-4. DOI `10.1109/METRICS.2005.29`. (cited on page 58).

J. Smith. A comparison of RUP and XP. Technical Report TP167, Rational Software, May 2001. (cited on pages 58 and 105).

E. Soloway, R. Lampert, S. Letovsky, D. Littman, and J. Pinto. Designing documentation to compensate for delocalized plans. *Communications of the ACM*, 31(11):1259–1267, 1988. DOI `10.1145/50087.50088`. (cited on page 11).

D. Soni, R. L. Nord, and C. Hofmeister. Software architecture in industrial applications. In *Proceedings of the 17th International Conference on Software Engineering*, pages 196–206. ACM Press, 1995. DOI `10.1145/225014.225033`. (cited on pages 104 and 105).

H. Spanjers, M. ter Huurne, B. Graaf, M. Lormans, D. Bendas, and R. van Solingen. Tool support for distributed software engineering. In *Proceedings of the International Conference on Global Software Engineering*, pages 187–198, 2006. DOI `10.1109/ICGSE.2006.261232`. (cited on page 161).

J. Sprinkle and G. Karsai. A domain-specific visual language for domain model evolution. *Journal of Visual Languages & Computing*, 15(3-4):291–307, 2004. DOI `10.1016/j.jvlc.2004.01.006`. (cited on page 173).

R. Stake. *The Art of Case Study Research*. Sage Publications, Inc, 1995. ISBN 978-0803957671. (cited on pages 13 and 60).

M. Staron. Adopting model driven software development in industry — A case study at two companies. In O. Nierstrasz, J. Whittle, D. Harel, and G. Reggio, editors, *Proceedings of the 9th International Conference on Model Driven Engineering Languages and Systems*, volume 4199 of *Lecture Notes in Computer Science*, pages 57–72. Springer, 2006. ISBN 3-540-45772-0. DOI `10.1007/11880240_5`. (cited on pages 137 and 160).

C. J. Stettina and W. Heijstek. Five agile factors: Helping self-management to self-reflect. In R. O'Connor, J. Pries-Heje, and R. Messnarz, editors, *Proceedings of the 18th European Conference on Systems, Software and Service Process Improvement*, pages 84–96, Roskilde, Denmark, June 2011a. DOI `10.1007/978-3-642-22206-1_8`. (cited on page 58).

C. J. Stettina and W. Heijstek. Necessary and neglected? an empirical study of internal documentation in agile software development teams. In *Proceedings of the 29th ACM International Conference on Design of Communication (SIGDOC 2011)*, pages 159–166, October 2011b. DOI `10.1145/2038476.2038509`. (cited on pages 58, 105 and 107).

A. C. Strauss and J. Corbin. *Basics of Qualitative Research: Grounded Theory Procedures and Techniques*. Sage Publications, Inc, 2nd edition, September 1990. ISBN 978-0803932500. (cited on pages 13 and 62).

P. Taylor, D. Greer, P. Sage, G. Coleman, K. McDaid, and F. Keenan. Do agile GSD experience reports help the practitioner? In *Proceedings of the 2006 international workshop on Global software development for the practitioner*, pages 87–93, 2006. DOI 10.1145/1138506.1138526. (cited on page 84).

R. N. Taylor, N. Medvidovic, and E. M. Dashofy. *Software Architecture: Foundations, Theory, and Practice*. Wiley, January 2009. ISBN 978-0470167748. (cited on page 105).

The Middleware Company. Model driven development for J2EE utilizing a model driven architecture (MDA) approach: Productivity analysis. Technical report, The Middleware Company, June 2003. URL http://www.omg.org/mda/mda_files/MDA_Comparison-TMC_final.pdf. (cited on page 160).

S. Tilley. Documenting software systems with views VI: lessons learned from 15 years of research & practice. In *Proceedings of the 27th ACM international Conference on Design of Communication*, pages 239–244, 2009. DOI 10.1145/1621995.1622043. (cited on page 108).

S. Tilley and S. Huang. A qualitative assessment of the efficacy of UML diagrams as a form of graphical documentation in aiding program understanding. In *Proceedings of the 21st annual international conference on Documentation*, pages 184–191, October 2003. ISBN 1-58113-696-X. DOI 10.1145/944868.944908. (cited on pages 11 and 106).

F. Trompenaars and P. Prud'homme van Reine. *Managing Change Across Corporate Cultures*. Capstone Publishing Ltd., 2004. ISBN 978-1841125787. (cited on page 4).

M. Tsunoda, A. Monden, H. Yadohisa, N. Kikuchi, and K. Matsumoto. Software development productivity of Japanese enterprise applications. *Information Technology and Management*, 10(4):193–205, Dec. 2009. DOI 10.1007/s10799-009-0050-9. (cited on page 85).

A. Van Deursen, P. Klint, and J. Visser. Domain-specific languages: An annotated bibliography. *ACM Sigplan Notices*, 35(6):26–36, June 2000. DOI 10.1145/352029.352035. (cited on pages 11 and 137).

H. van Vliet. *Software Engineering: Principles and Practice*. Wiley, 3rd edition, June 2008. ISBN 978-0470031469. (cited on pages 171 and 194).

D. Šmite, C. Wohlin, T. Gorschek, and R. Feldt. Empirical evidence in global software engineering: a systematic review. *Empirical Software Engineering*, 15(1):91–118, 2010. DOI 10.1007/s10664-009-9123-y. (cited on pages 2, 24, 94, 158, 173, 174, 176 and 195).

D. B. Walz, J. J. Elam, and B. Curtis. Inside a software design team: knowledge acquisition, sharing, and integration. *Communications of the ACM*, 36:63–77, October 1993. DOI 10.1145/163430.163447. (cited on page 58).

J. Wang, R. G. Carlson, R. S. Falck, H. A. Siegal, A. Rahman, and L. Li. Respondent-driven sampling to recruit MDMA users: a methodological assessment. *Drug and Alcohol Dependence*, 78(2):147–157, May 2005. DOI 10.1016/j.drugalcdep.2004.10.011. (cited on page 92).

J. Warmer and A. Kleppe. *The Object Constraint Language: Precise Modeling With UML (Addison-Wesley Object Technology Series)*. Addison-Wesley Professional, 1998. ISBN 978-0201379402. (cited on page 108).

T. Weigert, F. Weil, K. Marth, P. Baker, C. Jervis, P. Dietz, Y. Gui, A. van den Berg, K. Fleer, D. Nelson, M. Wells, and B. E. Mastenbrook. Experiences in deploying model-driven engineering. In E. Gaudin, E. Najm, and R. Reed, editors, *Proceedings of the 13th International SDL Forum on Design for Dependable Systems*, volume 4745 of *Lecture Notes in Computer Science*, pages 35–53. Springer, 2007. ISBN 978-3-540-74983-7. DOI 10.1007/978-3-540-74984-4_3. (cited on page 138).

J. White, D. C. Schmidt, and A. S. Gokhale. Simplifying autonomic enterprise java bean applications via model-driven development: A case study. In L. C. Briand and C. Williams, editors, *Proceedings of the ACM/IEEE 8th International Conference on Model Driven Engineering Languages and Systems*, volume 3713 of *Lecture Notes in Computer Science*, pages 601–615. Springer, 2005. ISBN 3-540-29010-9. DOI 10.1007/11557432_45. (cited on page 138).

H. Wickham. *ggplot2: elegant graphics for data analysis*. Springer, 2009. ISBN 978-0-387-98140-6. URL http://had.co.nz/ggplot2/book. (cited on page 43).

C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén. *Experimentation in Software Engineering: An Introduction*. Kluwer Academic Publishers, Norwell, MA, USA, 2000. ISBN 0-7923-8682-5. (cited on pages 13, 111 and 130).

J. Wüst. The software design metrics tool for the UML 2.11, 2009. URL http://www.sdmetrics.com/. (cited on page 141).

J. Wüst. *SDMetrics User Manual V2.2*. In der Lache 17, Zellertal, Germany, February 2011. URL www.sdmetrics.com/down/SDMetricsManual.pdf. (cited on pages 143 and 144).

Y. Yang, M. He, M. Li, Q. Wang, and B. W. Boehm. Phase distribution of software development effort. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 61–69, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-971-5. DOI 10.1145/1414004.1414016. (cited on page 96).

P. Yiftachel, I. Hadar, D. Peled, E. Farchi, and D. Goldwasser. The study of resource allocation among software development phases: An economics-based approach. *Advances in Software Engineering*, art. 579292, 2011. DOI 10.1155/2011/579292. (cited on page 96).

R. K. Yin. *Case Study Research: Design and Methods, Third Edition, Applied Social Research Methods Series, Vol 5*. Sage Publications, Inc, 3rd edition, December 2002. ISBN 0-761925538. (cited on page 13).

S. Yusuf, H. Kagdi, and J. Maletic. Assessing the comprehension of UML class diagrams via eye tracking. In *Proceedings of the 15th IEEE International Conference on Program Comprehension*, pages 113–122, 2007. DOI 10.1109/ICPC.2007.10. (cited on page 123).

Y. Zhang and S. Patel. Agile model-driven development in practice. *IEEE Software*, 28 (2):84–91, March 2011. DOI 10.1109/MS.2010.85. (cited on page 84).

S. Zopf. Success factors for globally distributed projects. *Software Process: Improvement and Practice*, 14(6):355–359, 2009. DOI 10.1002/spip.v14:6. (cited on page 28).

Glossary

ADL Architecture Description Language

BOA Back Office Architect

BPMN Business Process Model and Notation

CASE Computer-Aided Software Engineering

CM Change Management

CMMI Capability Maturity Model Integration

CRS Chain Referral Sampling

DSL Domain-Specific Language

ERD Entity-Relation Diagram

ERP Enterprise Resource Planning

FOA Front Office Architect

FTE Full-Time Equivalent

GNU GNU's not Unix!

GQM Goal-Question-Metric

GSD Global Software Development

HR Human Resources

IT Information Technology

MDA Model-Driven Architecture

MDD Model-Driven Development

MDE Model-Driven Engineering

MOF Meta Object Framework

OCL Object Constraint Language

POC Proof Of Concept

RFC Request For Change

RUP Rational Unified Process

SAD Software Architecture Document

SCCMS Software Configuration and Change Management System

SLA Service-Level Agreement

SLOC Source Lines of Code

SQL Structured Query Language

SS Supplementary Specification

SVN Subversion

UML Unified Modeling Language

XMI XML Metadata Interchange

XML Extensible Markup Language