Architecture design in global and model-centric software development

Heijstek, W.

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

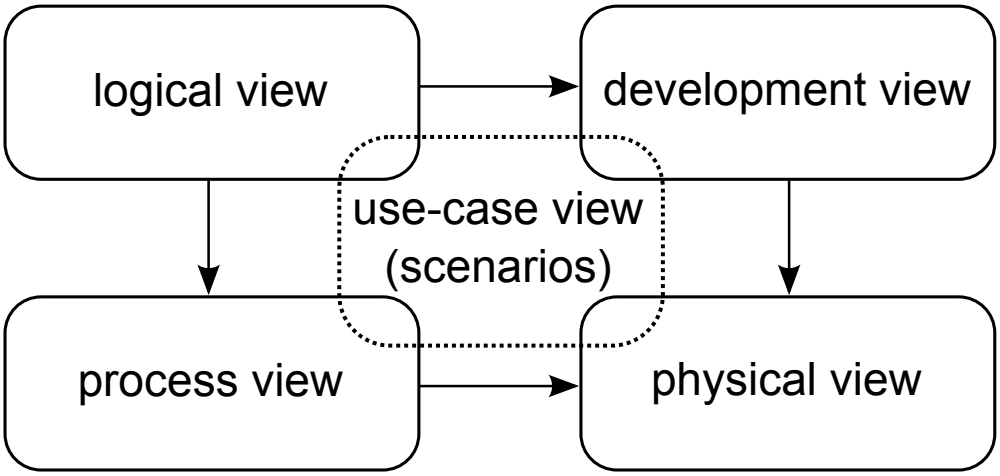

Figure 1.1: *The* 4 + 1*-view model*

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.

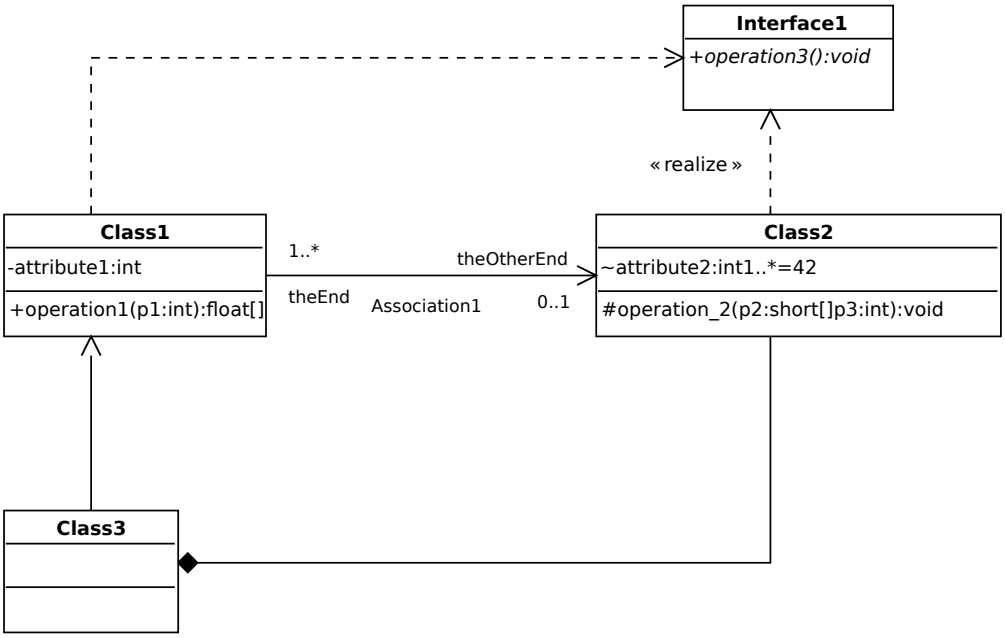

Figure 1.2: *Example of a UML class diagram*

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/

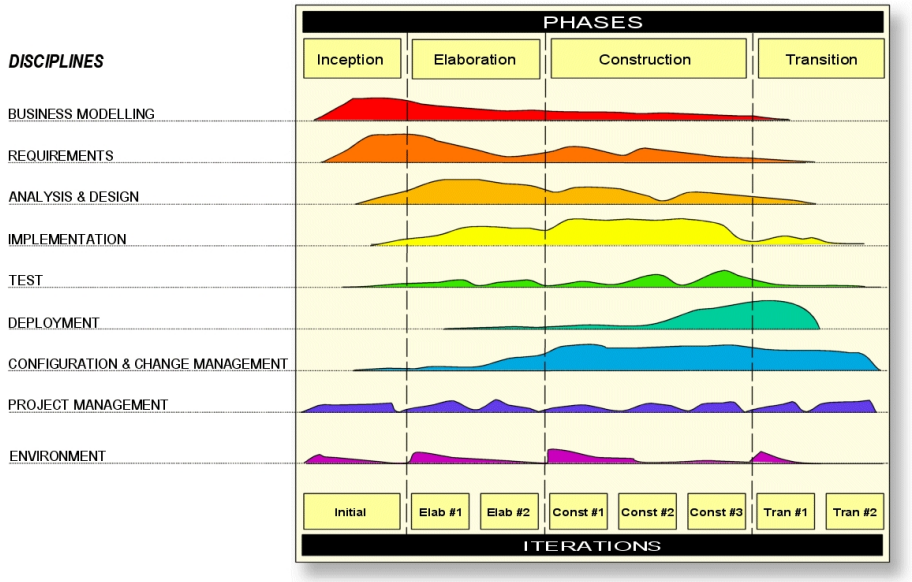

Figure 1.3: *A recent version of the RUP "hump" diagram*

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)

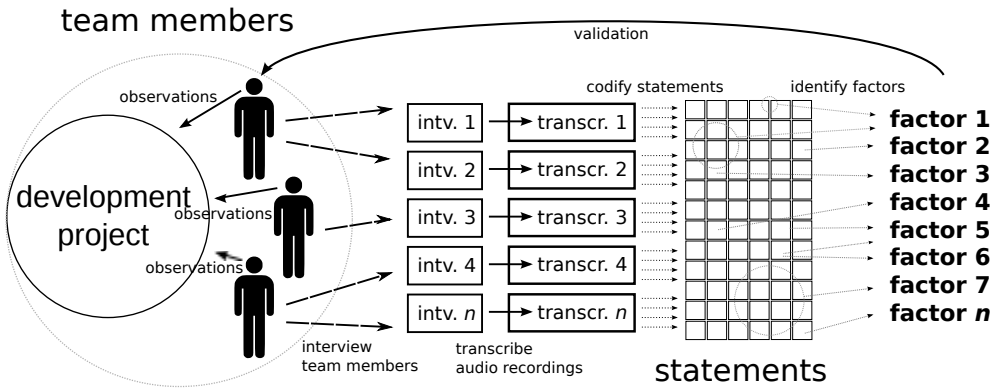

Figure 1.4: *The grounded theory data collection and analysis method – applied to a software project*

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**).

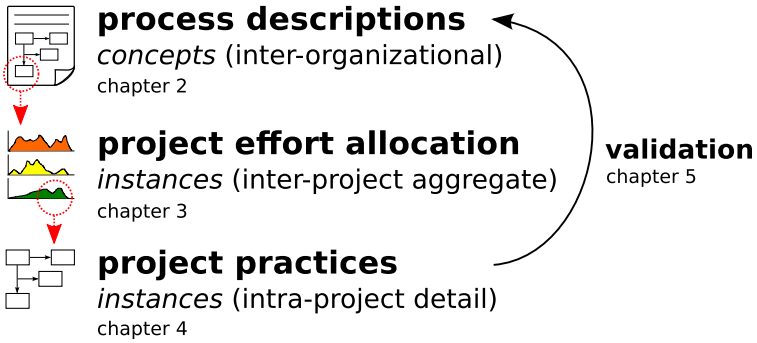

Figure 1.5: *Organization of the four chapters that address RQ1*

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**