Technische Universität München

Fakultät für Informatik

Master's Thesis

Software Architecture Design for Globally Distributed Development Teams

Lutz Küderli

Aufgabensteller:	Prof. Dr. Dr. h.c. Manfred Broy
Betreuer:	Dipl.-Inf. Gerd Beneken
Abgabedatum:	15.11.2005

Ich versichere, dass ich diese Diplomarbeit selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel verwendet habe.

München, den 14.11.2005 ....

Lutz Küderli

Abstract

This thesis discusses the aspect of software architecture in the context of geographically distributed development. It is based upon a research project that was conducted by Siemens Corporate Research in Princeton, NJ, USA, where a central team managed a distributed development project with six remote teams spread across the globe.

In the first chapters, an introduction into distributed development and software architecture is provided, followed by a description of the project as well as a number of architecture-related issues that were encountered. These are analyzed and categorized. In a next step, a criteria catalogue is compiled that contains a number of goals that emerged as important factors for the success of such an undertaking. Questions and metrics are attached to these goals by employing the Goal Question Metrics approach.

In a final step, an approach to assess the overall architecture is provided with the help of a scorecard. The set of metrics provides the architect and the project management with a tool to identify risks and evaluate architectural alternatives.

Table of Contents

Table of Figures

Chapter 1
Introduction

1.1 Motivation

Today, the globally distributed development of software is a reality for many IT companies and there are two factors play major roles in this process. On the one hand, companies have grown into multi-national corporations by expanding and acquiring other businesses and subsequently, the expertise and the knowledge are distributed worldwide. Experts who live in different countries have to work together and cooperate in order to create successful products. On the other hand, outsourcing and offshoring are used more widely by IT companies than ever. As the pool of candidates available who possess knowledge about software design and development increases in countries outside of Western Europe and North America, the possibilities to bring costs down because of lower wages in other countries and to shorten development cycles by being able to use a larger work force are generating more interest.

While this opens new possibilities to corporations, many new challenges have to be mastered as well. The complexity of software development for large systems is very high to start with already, and conducting it in a distributed fashion adds an additional layer to it. Therefore, processes and practices have to be adapted to be able to deal with the distributed environment. While research of this topic has many aspects, this thesis is going to focus on the architecture of systems built on a global scale. We presume that the advantage a well-defined and properly designed architecture provides is one of the factors critical to the success of a software product, and has not been researched in enough detail so far for distributed development. This observation is confirmed by a multitude of studies and spectacular failures that illustrate that design and implementation of software architectures constitute a very complex problem even for systems developed in a co-located fashion.

In distributed development environments, the need for consistent, well-founded, and well-documented software architectures is even larger than in projects that are developed at a single site. As (Clements et al., 2002a) point out: one of the main reasons for having an overall architecture is that it serves as a mean of communication between the shareholders who are involved in a project. It is a concept to clarify the requirements and resolve misconceptions that arise during the course of a project. In globally distributed teams, the need for a formal channel of communication is even higher. As informal communication between team members plays a less important role and there is less communication between teams in general, the need for a common foundation for understanding each other is even larger.

Another important point is the impact of misunderstandings or misconceptions. If the architecture has not been designed well enough, and problems arise out of this, it is even harder and more expensive to fix them when the different stakeholders are not co-located. For software engineering in general, this issue has been described in (Boehm, 2002) and Boehm's earlier works.

Consequently, it is crucial to evaluate the architecture according to a set of criteria. If one looks at the architecture of a system, one can predict certain properties of the system before it has been designed in detail and built. Therefore, well-specified and well-chosen criteria are a step forward in minimizing the project risks and judging the quality of the architecture itself as well as the implementation resulting from it.

In this thesis, we will use the term Architecture instead of the more specific term Software Architecture, except where noted otherwise.

1.2 The Global Studio Project

1.2.1 Background

This thesis refers to a research project carried out by Siemens Corporate Research in Princeton, New Jersey, USA. The architecture group of the software engineering department is conducting an ongoing study to evaluate distributed development processes with a central team in Princeton and development teams located at universities in varying countries. In the iteration described herein, five universities in the USA, Ireland, Germany, and India participated. The main research goals are to find out to which extent the artifacts specified by the central team provided were sufficient for the distributed teams to deal with, and what overhead of communication was necessary to resolve issues where the specifications where ambiguous. Another objective was to discover which criteria can be applied to figure out whether the expected or predicted results were going to be reached before actually getting to this point, i.e. an early warning system for project risks.

Although the project was conducted by computer science and software engineering students who did not have substantial practical experience and could not commit to the project full-time because of other academic obligations, the results bear still some validity for real-life projects. Additionally, the controlled setting in which the project was conducted enabled it to manage the parameters of the experiment. This is different from the industrial environments many papers refer to, as only data can be analyzed, because for obvious reasons the commercial success of the project has to take precedence over the research goals.

To ensure that as many data points as possible were collected, direct communication between the development teams was discouraged. All communication was conducted between the central team and the particular development teams or at least forwarded by the central team.

1.2.2 MSLite

The MSLite project had the goal of creating a building management station that offers a common interface to manage multiple building systems. While the output of the project was never intended to produce a commercially successful result, other business units of Siemens were interested in the experiences gathered during the execution of the project.

The requirements and architecture of the MSLite project are described in more detail in Chapter 1. The project plan for the MSLite project can be found in (Paulish, 2001). Basically, it consists of a three-tier architecture including a presentation layer, a middle tier responsible for the logic of the system, and a subsystem in the bottom layer that simulated field systems such as doors and smoke detectors.

The scope of the project was scaled down to basic functionality. Only a proof of concept of a working system was intended to be the outcome. Requirements such as security and performance played only a minor role.

1.3 Related Work

The previous research efforts on the issues investigated in this thesis can be divided into two major approaches: On the one hand, a large body of research exists on designing, evaluating, and documenting architecture, even though the field of software architecture studies is a relatively young one With regards to the first approach, Parnas (Parnas, 1972) described the basic principles of information hiding and subsystem decomposition; Brooks (Brooks, 1975) and others already remarked in the 1970s that the quality of a software system depends strongly on the coherency of its design. The systematic approach to designing and reviewing architecture, however, gained momentum in the early nineties (Garlan and Perry, 1994; Perry and Wolf, 1992). Not only did a substantial amount of work was done, but also some of the most influential practices have been defined at Carnegie Mellon University, particularly at the Software Engineering Institute. Major contributions include (Bass et al., 2003; Shaw and Garlan, 1996). Another good, practical approach can be found in (Hofmeister et al., 1999). The depiction and documentation of software architecture using views are shown in (Kruchten, 1995)A good introduction into the distributed project management from an architectural point of view is given in (Paulish, 2001); a standard book about working in a global context is (O'Hara-Devereaux and Johansen, 1994).

On the other hand, collaborative work – in co-located as well as in distributed environments – is another research focus of software engineering. Jim Herbsleb has conducted extensive empirical studies on collaborative work in the software engineering field and commented on them (Herbsleb and Grinter, 1999b; Herbsleb and Moitra, 2001).

1.4 Research Direction and Methodology

1.4.1 Goal of this Thesis

This thesis aims to join two issues. The first issue it deals with is the topic of how to evaluate software architectures. Research is performed to find out which traditional methods and measurements exist on how to evaluate architecture. A set of criteria is formulated on how these methods have to be adapted so that these principles can be used to judge the architecture for projects that are developed in a distributed fashion.

The second issue is empirical research conducted by investigating the MSLite project. We are particularly interested in the design decisions made in the beginning of the project, and in how the issue of distributed development was viewed back then. During the course of the project, it is interesting to see what effect these architectural decisions had on the success of the assignment, and which problems arose because of the decisions made in the beginning. From these observations, we derive the architecture-related issues that influenced the project the most, and analyze these in further detail.

The goal of this research is to gain a better understanding about the impact of architectural decisions on the quality of work delivered by the distributed teams. We establish a set of criteria, resulting from the latter issue gathered above, and consisting of questions and metrics. These metrics are used to measure the suitability of an architecture for a distributed development project.

This thesis does not discuss the specific details of how to create the architecture for a distributed development. Rather, the focus of our thesis is to judge an already existing architecture. Nevertheless, the architect can use the criteria established here to support architectural decisions. Additionally, the work done in this thesis can be used to research an approach on how to design architectures in a distributed fashion.

1.4.2 Empirical data

To evaluate the data gathered during the MSLite project, we had access to a variety of resources. The main source of knowledge are the assorted documents that were produced during the project and stored in a central repository under version control, such as project plans and architectural specifications. These documents were supplemented by our personal experiences as members of the central team.

Archiving of all email traffic between the central team and the development teams, as well as the emails between the members of the individual teams helped to gain insight into the communication structures and contents. Additional artifacts include monthly phone interviews conducted by a Harvard University Ph.D. student, weekly surveys for the team members and the supply managers, an additional post-mortem questionnaire, and the code repository and the design documents that were produced by the teams during the course of the project.

The post-mortem questionnaire was mainly related to the communication between the participants of the project, as well as other interaction, awareness issues and similar factors. The collected data was analyzed by Klarissa Chang and Jim Herbsleb from Carnegie Mellon University; the results of their work are shown in Appendix A.

The controlled setting of the environment of the MSLite project allowed to controll various parameters that are usually given and cannot be changed. These include:

- The number of remote teams, as well as their locations could be decided upon by SCR without focusing on a concrete project.
- The scope and complexity of the project and its requirements could be revised according to the abilities and skills of the involved teams.
- The problem domain could be chosen with regard to the research effort, instead of being dependent on business goals.
- The solution domain, that means all technical issues, as well as the infrastructure, could be decided upon by the central team without any regard to the remote teams.
- The actual output did not have to be of a concrete value to Siemens.

Two major factors, however, could not be controlled by the research team at SCR and have to be considered when evaluating the data:

- The academic schedule of the university teams involved into the project only allocated a limited amount of time to their assignments – the number of hours they put in weekly, as well as the overall time span they participated in it, typically one semester.
- The amount of resources SCR was able to assign to the project during its early phases was limited. Due to the mandated schedules of the members, they had to start delivering some results even though they had no clear objectives.

1.4.3 Limitations to the Controlled Environment

One has to take into account that this kind of research project has some differences to a "real-life" project. These include:

- The stakes are much higher if the project is supposed to fulfill a real business need. This leads to a higher pressure on all participants, as well as to more support by the upper levels of management, preferred allocation of resources, and usually a better overall motivation.
- As mentioned above, the students could not commit themselves full-time to the project. While this could have been mitigated in part by scaling down the scope of the project, two issues cannot be dealt with so easily. First, the experience with the employed technologies, tools and processes does not grown in a linear fashion. Usually, in the beginning of a project, participants have enough time to get familiar with these. In the case here, the students had to learn the usage of this tools and digest the information while the project was already well on its way. Secondly, the students' motivation was weaker, as they could not focus completely on this project, but had to fulfill other academic commitments (and sometimes part-time jobs) as well.
- Additionally, the students have to face the issue that while the project leaders are located at SCR, their performance (as reflected by their grades for this course) is rated by the professor or teaching assistant at their university. This can lead to conflicts of interest. One team, for example, was concerned that only a working implementation would enable them to receive good grades. The central team at SCR, while interested in a working implementation, was focused on a good specification to build upon. In distributed development in general, however, there is always the potential for conflict between the managers who are on site and the team of architects located somewhere else, reducing the significance of this issue somewhat.
- Projects with students face the possible impediment that the students' supervisors do not agree with the work done by the students. First of all, they might not like the software engineering practices or processes, as they are handled and mandated by the central team. Furthermore, it can happen that the teaching staff feels that the educational aspect of the course does not carry

enough weight compared to the goal of delivering successful work product. In the MSLite project, however, we found that this concern was unsubstantiated, as the supervisors at all involved universities let the students work very independently and were content with their role as advisors.

Chapter 2
Challenges of Distributed Development

2.1 Introduction and Definition

While software engineering is a large and complex problem by itself, distributed development adds an additional number of new problems. These concern mostly the issues of communication and coordination. Other major issues are cultural differences and politics that play a role when different sites are involved. This chapter aims to give an introduction into these issues, as well as providing some solutions for problems that are frequently encountered. Many of the basic concepts about distributed developed mentioned and explained in sections 2.2, 2.3, and 2.4 are either taken from or based on (Gersmann, 2005).

A distributed software engineering project is an undertaking where the stakeholders are located at geographically different locations, and spend a substantial amount there (instead of traveling and working at one location together). The reasons why organizations conduct software engineering products in a distributed fashion in the first place are outlined in section 1.1.

2.2 Classification

The actors' physical distribution can be used to determine the distance between the actors involved in the process is. Several distinct scenarios of physical distance are described in (Evaristo and Priklandnicki 2003):

- **Same-physical location** scenario in which the team is co-located in the same building, e.g. in an outsourcing project where the project team is co-located with the customer (who may represent the user as well in this case) for the time span of the project;
- **Cross-town** scenario, where a local contractor could be involved in the project;

- '**No time shift** scenario', where a team is physical distant but synchronous communication does not pose problems if the locations reside in the same time zone;
- **Continental** scenario, where a team is distributed but the physical distance and time difference is moderate;
- **Global** scenario, where a team is distributed to a large degree and located in at least two different continents.

From these scenarios, the notion of temporal and physical distance can be derived. It is important to differentiate between those two instances of distance because they have different implications. While having a large physical distance but small temporal distance hinders only face-to-face communication, it still leaves opportunities for synchronous communication using modern communication tools.

2.3 Communication and Coordination Issues

One general observation when working with multiples teams has been made by (Brooks, 1975) and confirmed many times later on (Curtis et al., 1988): If no mitigation strategies are employed, the communication overhead between teams and team members rises exponentially according to the number of people. Distributed development intensifies this problem, as communication that is not conducted face-to-face poses additional challenges. An example for these is the fact that conducting communication across different time zones leads to delays. When considering that software engineers spend about 40% of their time waiting for resources or doing work unrelated to their main task (Perry et al., 1994), this becomes even more obvious. Informal communications between colleagues meeting in the office do not take place across sites. The efficiency of this kind of conversations has been studied in detail, e.g. in (Weinberg, 1998).

One of the most daunting problems is the missing knowledge transfer between sites. While specifications and rules can be codified into written documents relatively easily, this task is much harder to accomplish when then teams are distributed. The lack of informal communication and the tendency of groups not to share information with people perceived as “outsiders” create knowledge imbalances. This is often related to a lack of awareness to what the other teams are doing.

The communication problems can become even tougher if the teams at different sites have a different cultural background. The possibilities of miscommunications because of culture-related conflicts are very large, but it is often hard to identify the exact problem sources beforehand. (O'Hara-Devereaux and Johansen, 1994) provide a number of examples. A very important reason can be found by examining the role of the context of a conversation or other interaction. While most Europeans and North Americans do not value this context very high (“They say what they think”), lots of information are communicated non-verbally in many Asian countries.

Additionally, the coordination and control of the developers by the project management become more difficult. As the hierarchies are not clearly visible, leaders may not be able to recognize progress and problems at other sites. Furthermore, employees tend to orientate themselves to higher-ranking staff at their own location instead of following the establish chain of command. This concern can be intensified by political issues that arise between the management of different sites.

2.4 Technical Issues

The technical issues arising from distributed development are mainly related to the tools used for the communication between teams and the joint work. Communication is hindered by the shortcomings and improper use of instruments such videoconferencing systems, email and instant messaging solutions.

While the tool support for the implementation phase, such as automated builds and tests, version control, and issue tracking systems is fairly advanced, the idea of concurrent and cooperative work at the same artifact at the same time by people located at different sites is still an exception rather than the rule. While *Computer Supported Collaborative Work* (*CSCW*, see (Grudin, 1994) for an overview about the subject) has been researched in detail, many concepts fail to be adapted outside of academic environments (Markus and Connolly, 1990).

Additional technical problems include problems setting up the communications structures do to different protocols or versions teams at different sites employ. The initial setup of a common infrastructure can require a substantial amount of resources, particularly if the teams have not worked together before. Problems can lead to reluctance to use the tools specified by the project management, leading to a heterogeneous environment that brings additional communication issues and mismatches with it.

2.5 Conway's Law

Conway's Law (Conway, 1968) states that *"Organizations which design systems are constrained to produce designs which are copies of the communication structures of these organizations."* In other words, the outcome and the architecture of a software development project reflect the internal structure of the organization. For distributed development, this often-observed characteristic implicates that the final product will have distinct separations between its parts according to the geographical distribution of the teams. Therefore, the architecture has to be designed with the team structure in mind. If this is not the case, the system will exhibit the characteristic mentioned above, but unintentionally instead of planned.

(Herbsleb and Grinter, 1999a) investigate the relationship of architecture and Conway's law in detail. They conclude that a good design is vital, but that amongst others, the following points have to be taken into account as well:

- Only well-understood projects should be split up for development in a distributed fashion.
- Decisions should be recorded and made available to all project participants, creating a knowledge repository.
- Get-together meetings by traveling to the various sites should be conducted as early as possible.

2.6 Lessons from Open Source Software Development

Open Source Software (OSS) has gained a large amount of attention during recent years. It follows the concept that the code of software is provided to the public for free, under licenses such as the GNU General public license (GPL, 2005), or the BSD License (BSD License, 2005).

The proponents of developing software in such a fashion claim that the quality is higher and has a positive relationship to the the availability of the code. Open access leads to more people looking at it, thus detecting more mistakes. The famous *Linus's law* states (Raymond, 2001): *"given enough eyeballs, all bugs are shallow"*.

It is interesting to see, however, that the major part of successful applications developed as open source projects is part of a category of software where the developers have a lot of domain knowledge. Consequently, flourishing products are editors (Emacs), IDEs (Eclipse), operating systems (BSD, Linux), web servers (Apache) and compilers (GCC). In other areas, closed-source products are often much more successful (such as graphical design, large-scale business applications or video games) than their open-source counterparts. One can speculate that the success and alleged low rate of defects is related to this fact. As all developers are familiar with the domain they are operating in, the requirements elicitation process is usually much less formal.

It should be noted that many of the largest open source projects are backed by a large company or other institution providing them with resources. Being open source does not make a project adhere to a certain development scheme, but almost all open source projects have in common that their contributors are distributed worldwide, and rarely have face-to-face conversations with each other. Somehow, the open source community seems to manage the distributed development approach quite well. While this method seems chaotic at first, and this impression is intensified by works such as describing the open source structures as "bazaar" (Raymond, 2001), large open source projects have clearly defined internal structures, processes and roles. It can be argued (Bauer and Pizka, 2003) that these processes are closely related to the architecture. A selection of the best solution is likely because many people are involved in the incorporation of changes and decisions about the architecture, This is a stark contrast to the hierarchical structures of companies, where decisions are made by a small circle of stakeholders, or sometimes only by a single person, the architect.

Additionally, source projects employ most often, a strong tool support for testing and automated builds. This, together with the exchange of ideas, suggestions and bug reports via mailing lists that are usually open to everybody lead to a very transparent development process.

For the distributed development approach within companies, the way open source software projects proceed shows some very promising leads, even though not all concepts can be transferred directly.

2.7 Mitigation Approaches

A number of software engineering strategies and concepts can be used to tackle to problems related to distributed development. Some of these are mentioned in the next chapters. While they are not restricted to distributed projects, but are all employed in co-located environments as well, they can be employed to solve some major issues that are mentioned concerning the MSLite project.

2.7.1 Dealing with Domain Knowledge

Two types of domain knowledge can be distinguished among developers: the knowledge about the application domain as well as technical knowledge about the solution domain. It is imperative that the architects of the system have a thorough understanding about the application domain or at the least, work in close collaboration with a domain expert, as well as exhibit in-depth technical knowledge about the solution domain.

In distributed development teams, both kinds of knowledge often differ signifigantly. In Siemens projects, the application domain experts are usually located in the business units marketing the product, while the solution domain experts are located in these business units, as well in outsourced development business units, and are sometimes recruited from special divisions such as SCR. In addition, the complexity of the projects leads to the collaboration of specialized subdivisions where even the application domain knowledge is distributed among locations. This leads to the challenging task of bringing together a large number of developers from different cultures, with different amounts of problem and solution domain knowledge.

2.7.2 Interface Issues

The definition and implementation of interfaces is usually one of the main and most daunting problems when compared to co-located development. While the teams can usually coordinate their internal tasks in a more or less consistent manner, the communication between the components developed by different teams is an issue in almost every major project. The integration of these components often creates a large number of the problems for the people responsible for doing this. The number of solutions to deal with this task reflects its importance and the problems associated with it.

However, a number of process-related strategies have been introduced. They can be classified in problem avoidance and problem detection. Problem detection includes the iterative approach of building the whole system, as problems will occur on a smaller scope and can be fixed more easily than those taking place when the whole system is integrated at once. This process is taken to its extreme when using *Continuous Integration*, where the integration build and all associated tasks are done at least once a week and optimally, at least once a day. That way, errors introduced into the code can be recognized much quicker. Other methods of detection problems with interface mismatches are walkthroughs and code inspections, where the specifications and the actual code itself are explained by the authors and investigated by a team of other project participants. Automated tests can be used to detect problems with interfaces. Good test cases provide a high rate of detection for these kinds of issues. Due to the nature of an interface mismatch in which two teams are involved, a problem arises during testing since the integration test is already conducted during a late stage of the implementation, causing relatively high costs in fixing errors. If the problems are recognized e.g. during a walkthrough of the design documents, less effort has to be invested.

Problem avoidance is done by specifying the architecture as good as possible. Proponents of this approach suggest using an *Interface Description Language* or *Interface Definition Language (IDL).* These languages are used to formalize the architectural design by using an unambiguous, simple syntax. By employing an IDL, the misunderstandings about the call semantics can be minimized.

2.7.3 Vertical Slicing and Feature-Driven Approaches

One possible solution to interfaces issues is the so-called *vertical slicing* of the product development (Martin, 1999), suggesting to divide the project into slices that contain sets of functionality. While the initial idea of slicing is to build the slices in a primary sequential approach, it is possible to parallelize much of this process after the initial requirements elicitation and high-level design have been finished and some initial slices containing core functionality haven been implemented.

If a system is developed in such as fashion, development teams with enough domain knowledge and technical expertise develop the functionality delegated to them in a framework context and integrate the result with the slices developed by the other teams. The advantage of this strategy is obvious: the developers can focus on their core competencies and the architecture of their part of the system is usually developed by themselves. Additionally, the first slice(s) can be used to tackle hard problems and prove major characteristics (Paulish, 2001). The concept of a implementing one piece of functionality completely through all layers to find out whether the architecture is feasible and the project environment is working is also called *tracer bullet* (Hunt and Thomas, 1999).

The disadvantages of this approach originate mainly from the difficulties of dealing with the communication between the components that were implemented. As the functionality is developed in an iterative fashion and slices do not necessarily correspond with subsystems, not all interface definitions are stable. These changing interface issues can usually resolved by using a well-documented process and techniques such as branching the source code in the version control system and configuration management, or by versioning the interfaces. Still, in distributed requirement it is much harder to change an interface than when developing a project in a co-located fashion, as pointed out in section 4.6.6. One issue is the time difference between teams that often does not allow for changes being done in a coordinated fashion, another one are the communication problems arising when something goes wrong.

When implementing a product in this manner, certain aspects of *feature-driven development* (Palmer and Felsing, 2002) move into the spotlight. The fine-grained design and the implementation are assigned to so-called *feature teams*, which take over the ownership of the code responsible for the feature they are supposed to implement. In a practice report by (Karlsson et al., 2000), a large number of advantages for this kind of setup are mentioned, such as ruling out the "feature creep", better motivation, and claiming that interfaces between modules can be resolved within feature teams. In distributed development, the use of feature teams provide for less interfaces mismatches, and to a better understanding of how certain parts of the system operate.

For example, each team writes the GUI for the feature assigned to them, avoiding mismatches between this and the way the feature of the system is actually executed and consequently, the communication overhead will be reduced.

Additionally, in many cases, domain experts can be distributed to the specific teams or at least communicate more closely with them. One example in the MSLite system is the GUI for the L&R subsystem (cf. section 4.2). SCR decided to hand out this task to the IIITB team, which implemented other parts of the Presentation subsystem as well. Had the development strategy been oriented according to a feature-oriented fashion, the MU team would have been responsible for the L&R subsystem itself as well. In this project, the domain knowledge needed for these tasks could have been provided by SCR. If there had been requirements for more functionality, however, both teams probably would have needed to consult a domain expert. This would have been different if one team had been responsible for both components.

Apart from problems with feature teams in general, such as the degradation of the modules over time, this approach creates a number of new issues that becomes particularly obvious when considering the layered architecture as exhibited by the MSLite project. The majority of features consist of passing a command from the Presentation subsystem to the SOM or from the FSS to the SOM and to the Presentation layer; there are bottlenecks the flow of data has to pass. If the system had been in implemented in a feature-oriented fashion, concerns such as persistence, authentication and others would all have crosscut the implementation. This would have led to a larger communication overhead, or led to duplicate work. The architect has to deal with a tradeoff here: would these disadvantages have outweighed the benefits? Considering the fact that the amount of communication already posed a problem, this approach would probably not have been useful for the MSLite project.

Had the system implemented in microkernel-like fashion, consisting of a core framework and plug-ins for the features such as persistence, a feature-driven approach would have been more useful. Architectural styles such as blackboard and repository patterns (Buschmann et al., 1996) have the advantage that the main interactions between features and subsystems happen only with the central components, therefore reducing the coordination overhead.

Recapitulating, following a feature-driven approach as opposed to following the module structure is useful for distributed development, but only if the architecture allows for it. Usually, at least some core modules already have to exist in addition to an architecture that follows a repository-like pattern.

2.7.4 Service Oriented Architectures

One focus of a number of research initiatives and industrial support is the so-called *Service Oriented Architecture (SOA)* design (Papazoglou, 2003). SOAs adhere to the following principles:

- Their coupling is very loose (as compared to the level of objects or components).
- Their dependencies are usually satisfied only at runtime (as compared to components, where the dependencies have to be dealt with during the built).
- The scope of their communication is often between enterprises.
- They are technology neutral.
- Their design is contract-based; this contract is published.

Usually, SOAs are used when communication between different entities has to be performed, a service consumer, a service provider, and a registry, which supports the consumer in finding the provider that is required. Some features of service-orientation, however, make a lot of sense in distributed development as well.

Using contracts creates a highly visible set of interface definitions. Therefore, the points of interaction between the different development sites are reflected in the contracts of the architecture. As every feature that is provided by one team is specified and provided over a registry, the control of call semantics can be controlled very well by the architect.

The advantage of a SOA over a repository pattern stems from the fact that with a repository, the responsibilities cannot be defined clearly. The team responsible for the structure of the central subsystem specifies the way the data flows in and out it. This module has limited control over the usage of the data outside of it, however, and acts only as an information broker. In a SOA, the contract is placed between the consumer and provider themselves.

Chapter 3
Software Architecture

3.1 Introduction and Definition

The notion of software architecture that is used today stems from the seventies (cf. section 1.3). The word "architecture" was introduced, and the idea of splitting a system into smaller parts to reduce its complexity was investigated in detail. During the nineties, the impact that software architecture had was given a higher visibility. The rise of object-oriented languages pushed the subject further into the spotlight. Many of today's technologies require extensive knowledge about fundamental architectural issues, such as the three-tier-architecture employed by J2EE and .NET.

This chapter provides an overview about the definition of software architecture, what it contains, and how it can be evaluated. To write about this subject, we need to find a definition first. As pointed out in (Avritzer and Weyuker, 1998): *"The answer is harder than one might expect, especially given that there is an extensive literature on the subject. Many authors deal with the question by essentially ignoring it. It is left as an undefined term with the implication that everyone knows what it means, and therefore there is no need to spell it out."*

Nevertheless, a definition has to be introduced here to create a basis for discussion. The Software Engineering Insitute at Carnegie Mellon University provides an exhaustive list of definitions on its website (How Define, 2005). We chose the following definition from (Jazayeri et al., 2000): *"Software architecture is a set of concepts and design decisions about the structure and texture of a software that must be made prior to concurrent engineering to enable effective satisfaction of architecturally significant explicit functional and quality requirements and implicit requirements of the product family, the problem, and the solution domains."*

Even if this definition was created with respect to product lines, it stresses the most important points:

- Software architecture does not only describe the subsystems and their interaction, but the basic concepts and ideas that can be found across the whole system.
- The aspects of collaboration and concurrent work are introduced.
- An architecture is based on three kinds of requirements: functional and nonfunctional requirements (nonfunctional requirements are often called quality attributes) and implicit requirements.
- Additionally, this specification is based on IEEE 1471 (IEEE Recommended practice for architectural description of software-intensive systems, 2000).

Many studies have shown (Clements and Northrop, 1996; Garlan and Shaw, 1994) that a good software architecture is one of the most important factors during the lifecycle of a software project. The success of the project depends strongly on it because it lays the foundations for further work concerning the system. A bad architecture harms almost all other activities that are going to follow.

3.2 Purpose

According to (Garlan and Perry, 1994), an architecture is required for a software system under development because:

- It makes the system understandable through a high level of abstraction.
- It exploits patterns to solve commonly encountered problems.

(Bass et al., 2003) break the first of these points down into four sub-issues:

- It provides a vehicle for communication among stakeholders.
- It is the manifestation of the earliest design decisions about a system.
- It is a transferable, reusable abstraction of a system.
- The tradeoffs that are made – and there always are tradeoffs – are analyzed and documented clearly.

The first point mainly concerns the communication with the users about the requirements of the system, and keeping the customer of the software informed about the progress of the project. While the architect makes major design decisions, his actions are ultimately driven by the management decisions made earlier on.

The second point is the most important one in the context of this thesis. An architecture codifies design decisions that are taken during the early stages of a project. Therefore, fundamental behavior, work distribution, and the schedule derive directly from it. Additionally, the design decisions taken now with respect to the *quality attributes* of a system are going to be realized. A quality attribute, often called a nonfunctional requirement, represents an obligation the system has to fulfill, though not in a functional manner. Typical examples for quality attributes are modifiability, performance, scalability, and similar features.

Another very important result of the creation of an architecture is the identification of risky elements within the system, enabling the architect to think about mitigation strategies early on.

It is important to keep in mind that an architectural specification has to cover all the details that the architect and the project manager expect to find represented in the implementation. Underspecified parts of the system will be specified by somebody else along the way – either explicitly or implicitly. The process of creating an architecture is an iterative one – as the fundamental design decisions are made, the requirements can refined, and out of this, the design is improved again.

3.3 Fundamental concepts

3.3.1 Quality attributes

The architecture of a software system is driven by so-called *Quality Attributes* (often called *nonfunctional requirements*). They describe properties of the system that describe not what the system has to do, but other qualities of the system, such as stability, usability, extensibility, and others. An exhaustive list can be found in (ISO, 1991). It should be noted that the quality attributes are mainly orthogonal to functional requirements. Just listing the name of a quality attribute such as modifiability is not enough; a way of describing them in more detail are methods such as quality attribute scenarios (Bass et al., 2003).

Quality attributes are a major influence for an architecture, because they can be used as a foundation for deriving *tactics* (design decisions that are influential in the control of a quality attribute response) and *architectural styles* (descriptions of element and relation types together with a set of constraints on how they are used). The actual functional requirements are not the focus of establishing the architectural specification. This is a reason why the domain expertise of an architect is usually not a major concern when starting a project. He will gain an understanding of it while dealing with the requirements. Not all quality attributes will reflect in the architecture, though. Large parts of the usability quality attribute, for example, are not a direct input to the architecture.

The quality attribute based approach is not the only one that can be employed to establish an initial architecture of the system from the requirements. Traditional object-oriented methods identify the entities in the domain and model them according to inheritance hierarchies (Booch, 1991). There are many other approaches, such as scenario-based ones (de Bruijn and van Vliet, 2001) to create an architectural design.

The advantage of using a quality attribute based approach in the context of this thesis links to the fact that the *distributability* of the development process can be considered as a quality attribute. Emanating on this, we can establish the strategies outlined in the following as measures that are derived from this QA.

3.3.2 Different Views

The needs for different viewpoints arise out of the need that there are multiple perspectives from which a system can be examined. Every of these perspectives has a different focus on the structures of the software, and is addressed to a different group of stakeholders. This implicates that when comparing two views, each has to omit details that are present in the other one. They depend on each other, however, as the relationships between them have to be consistent.

There is no consensus about the number and type of views that should be employed when describing an architecture. The choices are made according to the quality attributes of the system and the needs of the stakeholders. A problem is that very similar views have often been given different names by different researchers. In (Kruchten, 1995), the so-called 4+1 view model of architecture is described as following:

- *The* **logical** *view describes the design's object model when an object-oriented design method is used. To design an application that is very data-driven, you can use an alternative approach to develop some other form of logical view, such as an entity-relationship diagram.*
- *The* **process** *view describes the design's concurrency and synchronization aspects.*
- *The* **physical** *view describes the mapping of the software onto the hardware and reflects its distributed aspect.*
- *The* development *view describes the software's static organization in its development environment.*
- To illustrate these views, **scenarios** can be employed, which represent a fifth view.

3.4 Architectural Evaluation

3.4.1 Purpose

The reasons for conducting an architectural evaluation are twofold: On the one hand, an early assessment can uncover problems that are going to cause problems throughout the further course of the project. At the time the review is conducted, changes can be introduced at less cost than e.g. during the implementation phase. It therefore serves a tool to assess the risks associated with a project. On the other hand, an architectural review can be done with a relatively small effort, if it is compared e.g. with a code review. This relates to the fact that the architectural specification is typically less extensive than the system itself.

The goal of an architectural assessment is to verify that the requirements for the system are covered sufficiently in the architectural specifications. Therefore, the evaluators check whether the decisions the architect made to reflect the requirements were correct. A related objective is to make sure that conflicting requirements were resolved properly and did not lead to inconsistencies within the architecture.

Additionally, it shows whether the architecture was documented well enough, as the assessors will not have been involved in the design as much as the architects and have to base their analysis upon the documents provide by them.

Usually, an architectural evaluation uncovers problems that were not visible during the requirements gathering and the definition of the business goals. Therefore, it provides the stakeholders with a better understanding about the proposed system, as an architecture enables the them to predict certain properties of the system before it has been built (Clements et al., 1995).

3.4.2 Analysis Approaches

A number of different approaches and methods to conduct an architectural evaluation have been proposed. These include (Kurpjuweit, 2001):

- **Active Design Reviews** (Parnas and Weiss, 1985) are based on checklists and questionnaires
- The **Software Architecture Analysis Method (SAAM)** (Kazman et al., 1994) is a scenario based approach that addresses the quality attribute of modifiability.
- A **Software Architecture Evaluation Model** (Duenas et al., 1998) provides a framework that incorporates architectural metrics for architecture analysis.
- **AT&T Best Current Practices: Software Architecture Validation** (Architecture Validation, 1991) constitutes another checklist-based approach
- **MITRE's Architecture Quality Assessment** (Hilliard et al., 1997) is question-based approach with over 200 metrics.

All of these approaches have a different focus of their investigation. An overview about the different measuring and question techniques that can be employed is given in Table 1 (Bass et al., 2003). Two basic options exist to examine an architecture: One is based upon objective measurements, such as the number of classes, the other lets reviewers judge certain aspects of the system under development.

	Technique	**Generality**	**Detail Level**	**Phase**
Questioning Techniques	Questionnaire	General	Coarse	Early
	Checklist	Domain-specific	Varied	Middle
	Scenarios	System-specific	Medium	Middle
Measuring Techniques	Metrics	General or domain-specific	Fine	Middle
	Experiments (Prototypes, Simulations)	Domain-specific	Varied	Early

Table 1. Overview of assessment techniques

We will describe the **Architecture Tradeoff Analysis Method (ATAM)** (Clements et al., 2002b) in detail in the following. Its goal is *"to assess the consequences of architectural decision alternatives in light of quality attribute requirements."* This can be broken down into the points of

- *Discover risks – alternatives that might create future problems in some quality attribute.*
- *Discover sensitivity points – alternatives for which a slight change makes a significant difference in some quality attribute.*
- *Discover tradeoffs – decisions affecting more than one quality attribute.*

To reach these goals, an ATAM is conducted as a review where the stakeholders meet. It should be noted that when developing in a distributed fashion, the task of arranging a meeting between them could become a difficult undertaking by itself. Participants from both the evaluation team as well as the development team should be motivated to participate actively. Roles should be clearly defined in advance. An overview about the process is given in Figure 1.

The review starts by a presentation of the requirements and the proposed architecture. It has to be made clear how the architecture addresses the quality attributes stated in the requirements specification. Ultimately, this leads to a mapping of the business goals to the architecture.

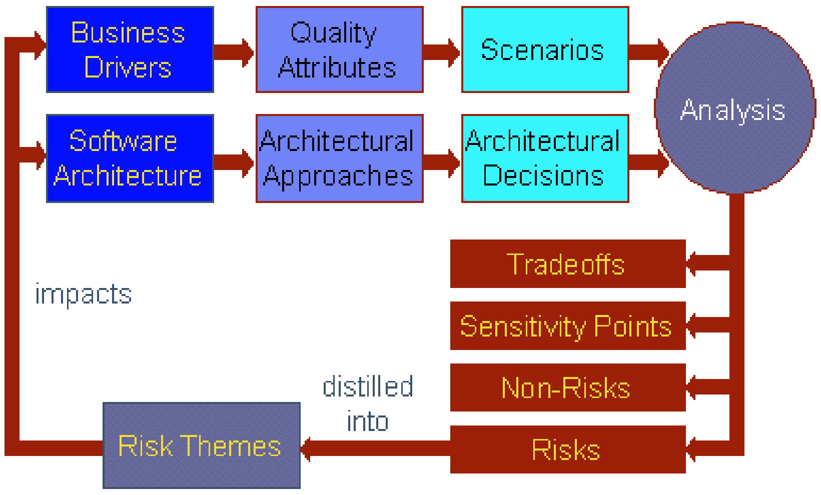


Figure 1. ATAM conceptual flow

In a next step, a so-called *quality attribute utility tree* is created to identify and prioritize the QAs (cf. Figure 2). Then, these attributes are used as a basis for the discussion of the architectural approaches that are to be employed. This is also useful to recognize the risks associated with them.

Then, a number of scenarios are created. They represent the stakeholders' interest in the system and deal with anticipated actions and issues as well as unanticipated stresses. These scenarios are then prioritized and compared with the utility tree. If there are scenarios that are not covered sufficiently by the utility tree, they are investigated in depth, risks are identified, and the information is written down.

At the end of the ATAM, a report is created that contains the decisions and observations made during its course. The following points are ideally reached as an outcome:

- There is an improved stakeholder buy-in.
- The documentation of the architecture has been improved.
- Missing requirements are discovered.
- Risks that were unknown so far are realized.
- Recommendations to mitigate risks are made.

Chapter 4
The MSLite Project

4.1 Background and Requirements

4.1.1 Business Goals

The following paragraph from the requirements document describes the objective of the system (Gersmann, 2005):

The MSLite system is a building management station that offers a common interface to manage multiple building systems that currently run in different platforms and have different management stations. For the purpose of the experiment project a Field System Simulator will mimic an integrated network of field events. A Management Station Adapter will be used to subscribe to these events and translate them into some common event protocol that can then be understood by the Management Station (MSLite). The Adapter will not be implemented in this iteration of the system. That means that the only supported field system for now is the Field System Simulator. The Management Stations basic operation must allow the user to view the current state of these event objects, issue commands which must be sent to the appropriate field event and, to receive, view and acknowledge any field event which issues an alarm.

The main business goals are stated in the following (Gersmann, 2005):

- Reduce total development costs for the management stations (replace 8 with 1).
- Broaden market base by being an open general-purpose management station that can be used with a wide variety of field systems (including eventually third party).
- Build a system with modern technologies that provides excellent user experience to satisfy advanced expectations by SBT customers.

For the MSLite system, the same business goals apply. The scope of the functionality, however, was reduced considerably, establishing a basic prototype to prove the feasibility of the basic architecture and highlights critical aspects of the development process as well as of the technical feasibility of some architectural design decisions. One focus of the project is the use of .NET as a framework and C# as the programming language, as the experience with these technologies in large projects is still much more limited than e.g. with Java or C++. The modeling of the flow of the events through the system constitutes an additional issue illustrated with a prototype.

4.1.2 Research Goals

For Siemens, distributed development has become on of the most demanding challenges when creating and maintaining software. When building a new product in such a fashion, its failure can have dire consequences for a whole business unit. Therefore, this issue is one of the areas to which the software engineering department of SCR devotes the most resources.

The main factor distinguishing MSLite from most other projects is the fact that it was set up with a set of research goals in mind. That is why there not only exists a set of business goals for the system, but a whole set of research goals as well which had to be fulfilled to make the project a success. These goals were stated in the project plan (Paulish, 2001):

- Given that the technical artifacts that are given to the remote component development teams are not adequate in specifying the precise work to be done, in what ways are they deficient?
- What strategies do the remote teams employ (and to what effect) to compensate for the deficiencies found in the received technical artifacts?
- What are the early warning signs that an issue is imminent? Can communication patterns, for example, between the central and remote teams be used to predict future component integration problems?

A complete list of the research goals can be found in Appendix B.

When investigating the project, it should always be remembered that some trade-offs had to be made when judging the business goals against the research goals, which is not the cause with a "real-life" business.

4.1.3 Requirements and Quality Attributes

The functional requirements for the system are listed in detail in (Gersmann, 2005). They are split up in the requirements for the *MSLite* system and for the *Field System Simulator*. The MSLite system is responsible for the authentication of users, deals with the change of value events that are created by the field system elements, issues commands to them, creates and manages alarms and is responsible for displaying a graphical representation of the field system elements and their hierarchy to the users.

The Field System Simulator is responsible for simulating the field system events, which can be configured and are read and saved from/to a persistent storage. Additionally, alarms can be managed by the FSS.

The following (Figure 2) quality attributes were identified during a Quality Attribute Workshop at CMU:

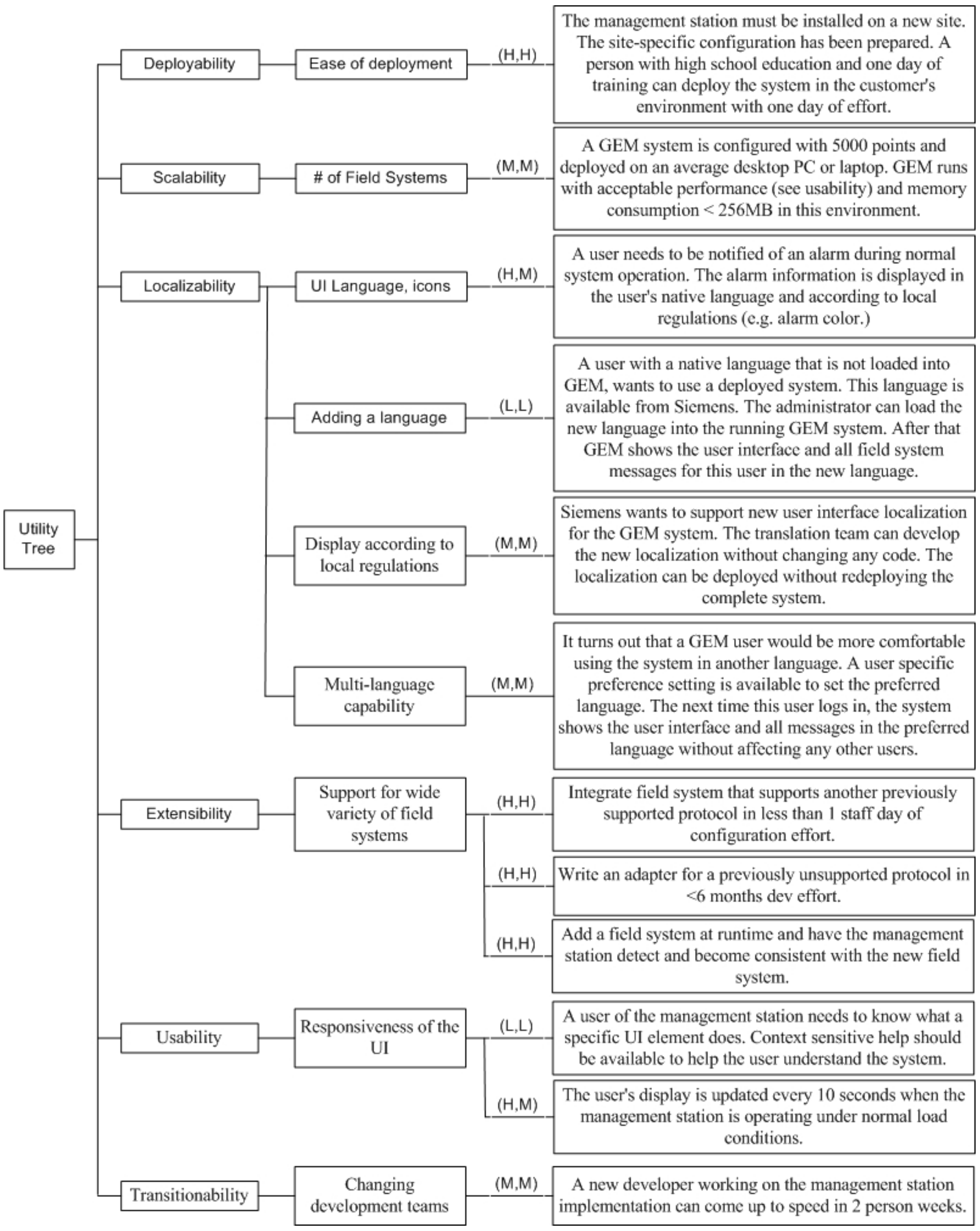

Utility Tree
Deployability
Ease of deployment
(H,H)
The management station must be installed on a new site. The site-specific configuration has been prepared. A person with high school education and one day of training can deploy the system in the customer's environment with one day of effort.
Scalability
# of Field Systems
(M,M)
A GEM system is configured with 5000 points and deployed on an average desktop PC or laptop. GEM runs with acceptable performance (see usability) and memory consumption < 256MB in this environment.
Localizability
UI Language, icons
(H,M)
A user needs to be notified of an alarm during normal system operation. The alarm information is displayed in the user's native language and according to local regulations (e.g. alarm color.)
Adding a language
(L,L)
A user with a native language that is not loaded into GEM, wants to use a deployed system. This language is available from Siemens. The administrator can load the new language into the running GEM system. After that GEM shows the user interface and all field system messages for this user in the new language.
Display according to local regulations
(M,M)
Siemens wants to support new user interface localization for the GEM system. The translation team can develop the new localization without changing any code. The localization can be deployed without redeploying the complete system.
Multi-language capability
(M,M)
It turns out that a GEM user would be more comfortable using the system in another language. A user specific preference setting is available to set the preferred language. The next time this user logs in, the system shows the user interface and all messages in the preferred language without affecting any other users.
Extensibility
Support for wide variety of field systems
(H,H)
Integrate field system that supports another previously supported protocol in less than 1 staff day of configuration effort.
(H,H)
Write an adapter for a previously unsupported protocol in <6 months dev effort.
(H,H)
Add a field system at runtime and have the management station detect and become consistent with the new field system.
Usability
Responsiveness of the UI
(L,L)
A user of the management station needs to know what a specific UI element does. Context sensitive help should be available to help the user understand the system.
(H,M)
The user's display is updated every 10 seconds when the management station is operating under normal load conditions.
Transitionability
Changing development teams
(M,M)
A new developer working on the management station implementation can come up to speed in 2 person weeks.

Figure 2. The QAW Utility Tree

It is interesting to note that distributability of development was not stated as an explicit quality attribute. This was revised in the list of quality attributes created for the second iteration of the MSLite project.

4.2 Architecture Design Overview and Rationale

Figure 3 depicts the component and connector view of the MSLite system. It is developed as a three-tier architecture, as described for example in (Fowler, 2002). In the "real" system, the data source level consists of the field system devices, such as lights, temperature sensors and doors, which are connected to the middle tier via a *Field System Adaptor*. In the MSLite system, these devices and the adaptor are emulated by a so-called *Field System Simulator.* It is used to test the back-end functionality by publishing BACNet-compliant notifications. To create event notifications, it reads the simulation data from pre-defined XML files.

The middle tier consists of a subsystem called System Object Model (SOM), which implements the core business logic of the MSLite system. Additionally, this tier contains the Logic and Reaction component (L&R). It is responsible for maintaining a set of conditions and reactions. When a certain condition is met, a reaction will be executed. The SOM communicates with the Presentation subsystem and the FSS using .NET remoting.

The *Presentation* subsystem (Figure 4) contains the user interface and its backend logic. The *UIFramework* provides a container for the components embedded in it and provides additional services to them. These UI components consist of ASP.NET pages and controls and constitute the interface between the user and the system.

The UIBackend contains the delegate components. These are used to communicate with the SOM and provide the backend logic for the UI components.

For a more detailed description of the MSLite architecture, refer to (Gersmann, 2005).

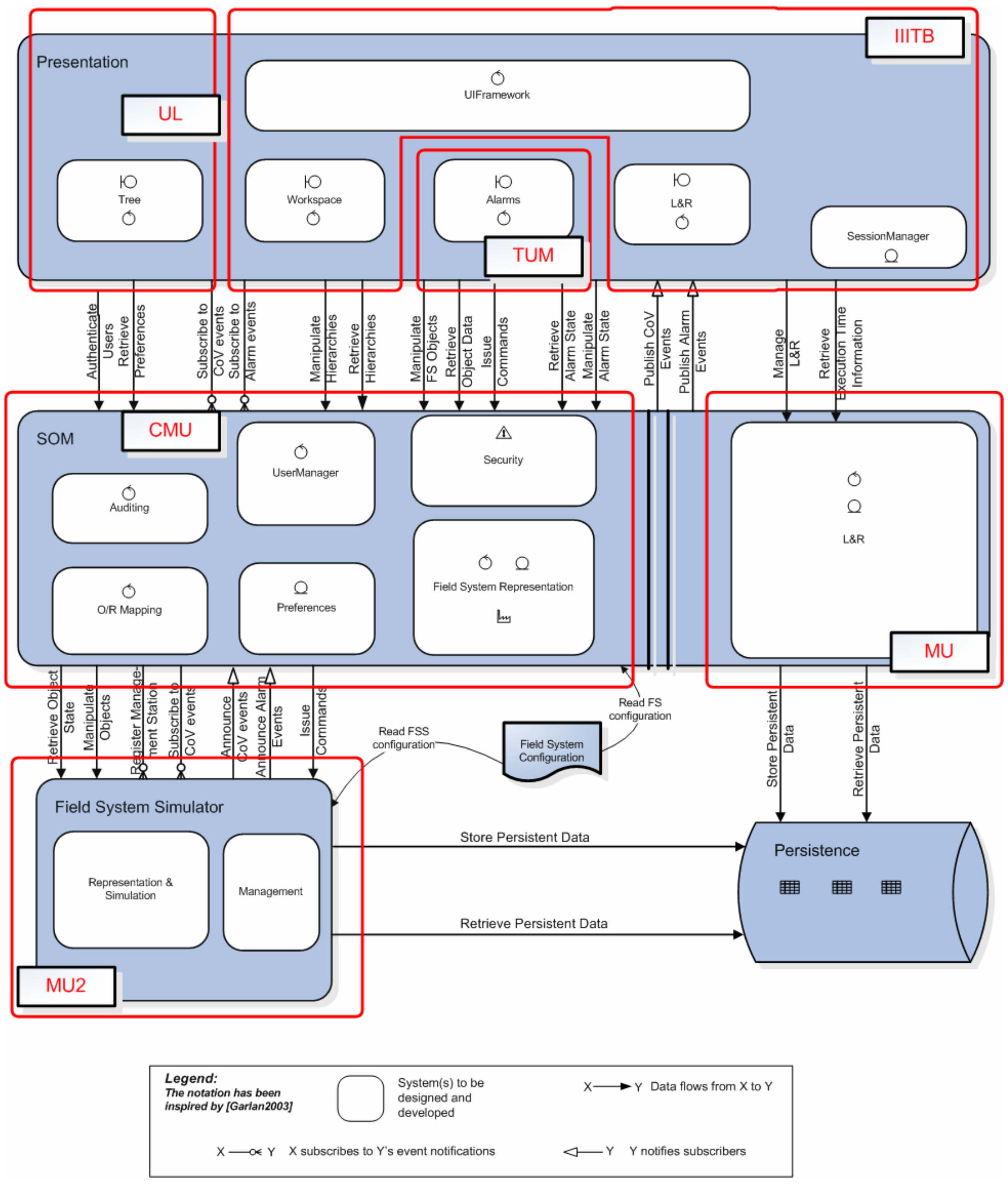


Figure 3. C&C view of the MSLite Architecture

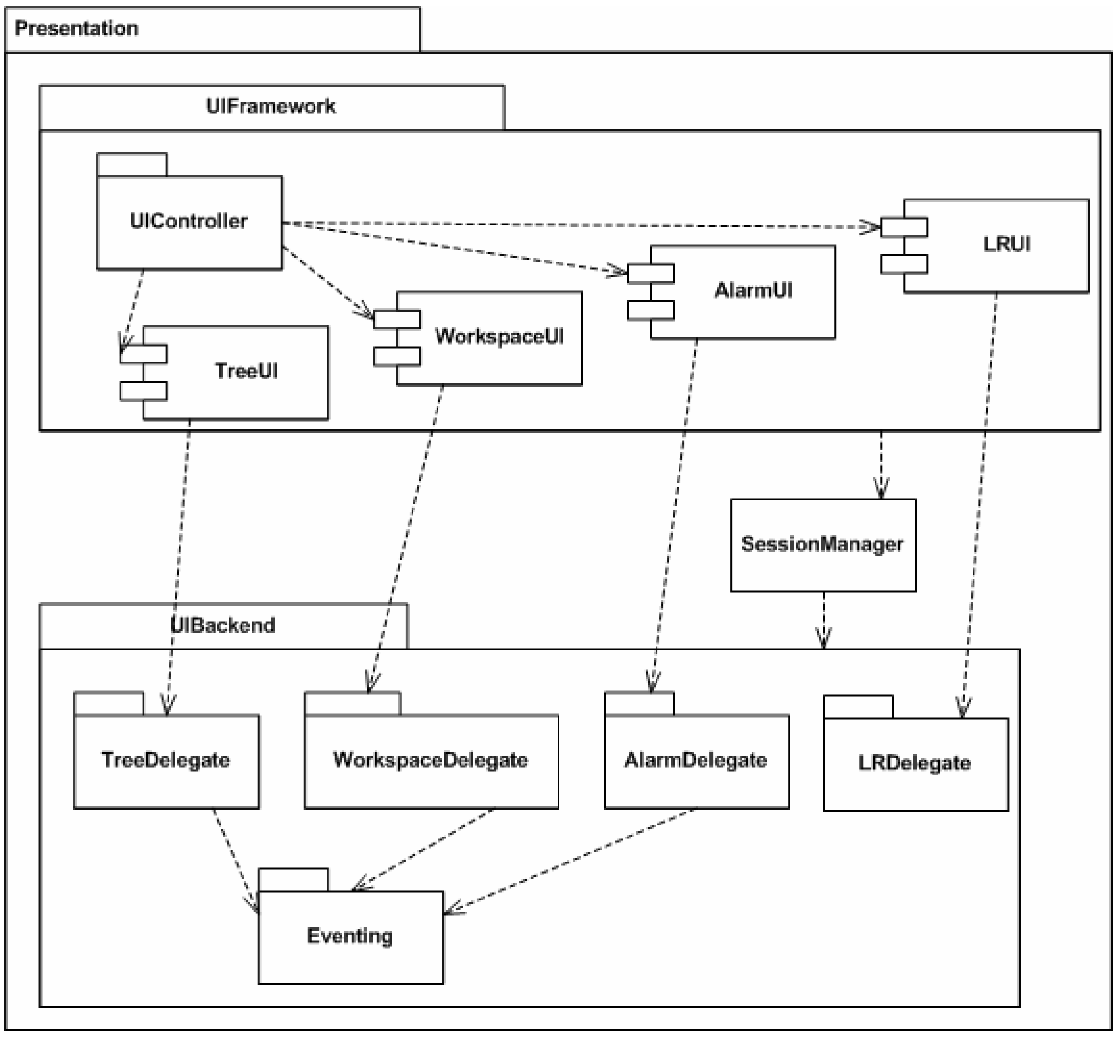


Figure 4. Presentation subsystem modules

4.3 Team Organization

Teams at six different locations were involved in the project. The central team at Siemens Corporate Research in Princeton consisted of four members, two of which worked full-time on the MSLite project. From Monmouth University, also in New Jersey, two teams (MU and MU2) of five and ten students contributed. In Pittsburgh, four Software Engineering Masters students participated from Carnegie Mellon University (CMU), along with three students in Munich from Technische Universität München (TUM). In India, there were four Siemens interns from the International Institute of

Technology Bangalore (IIITB). In Ireland, from the University of Limerick (UL), SCR recruited five Software Engineering Masters students.

The members of the ten-person team from Monmouth University were originally split up into two teams, but after two months into the project, it became clear that the Field Systems Simulator should be developed as a single component by one group. Since the central team considered the development workload of this subsystem to be too much for four to five people, it decided to merge the teams and assign the FSS to them. The L&R component was then reassigned to the remaining 5-person team from Monmouth University. Each team's responsibilities are highlighted in red in Figure 3.

Since the CMU students were the only ones who already had significant job experience in the Software Engineering field, the central team assigned to them the responsibility for the SOM subsystem. As part of this task, they would have to create a domain model, one of the most difficult tasks in the project.

Given that the team from IIITB had already some experience with development using ASP.NET, they were assigned the responsibility for building the WorkspaceUI and its corresponding delegate was handed over to them. The . As for the TUM group, since it consisted only of three members, they were assigned the Eventing subsystem, which was deemed to be of smaller scope. Lastly, the UL team had the task of designing and implementing a hierarchical (tree-like) view of the field system devices.

So-called *supplier managers* were established to serve as a point of contact for the students when they had questions. These individuals also coordinated communication between the central team and the respective remote team. An overview about the structure of the organization for the MSLite project is illustrated in Figure 5.

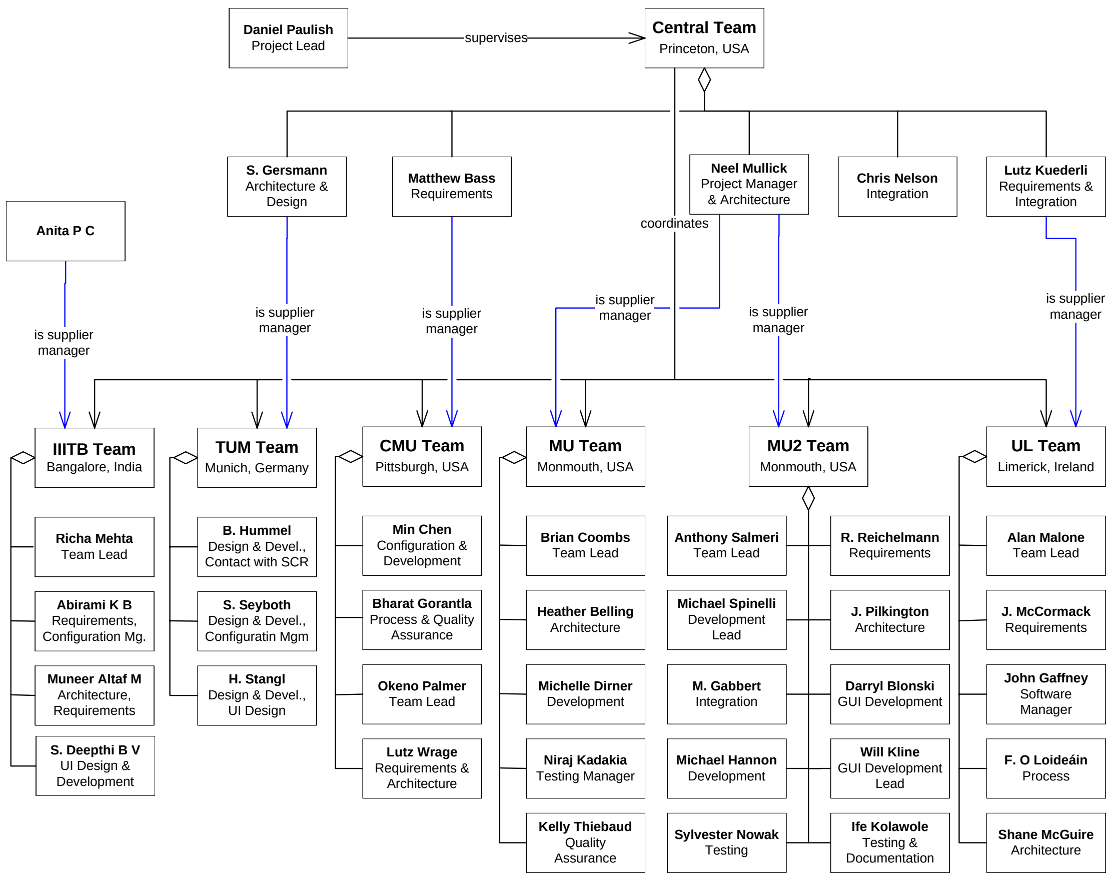


Figure 5. Organization Chart for the MSLite Project

4.4 Process Model and Quality Assurance

The project was originally intended to follow the so-called Model-Driven Rapid Application Development (MD.RAD) process which was developed at Siemens Corporate Research (Sangwan and Masticola, 2005). However, after the teams started working on the project, it became obvious that due to the lack of resources at SCR and timeframe for the project, no proper instantiation of the process could be defined. Therefore, the project followed an ad-hoc process that only made use of some ideas from the MD.RAD process.

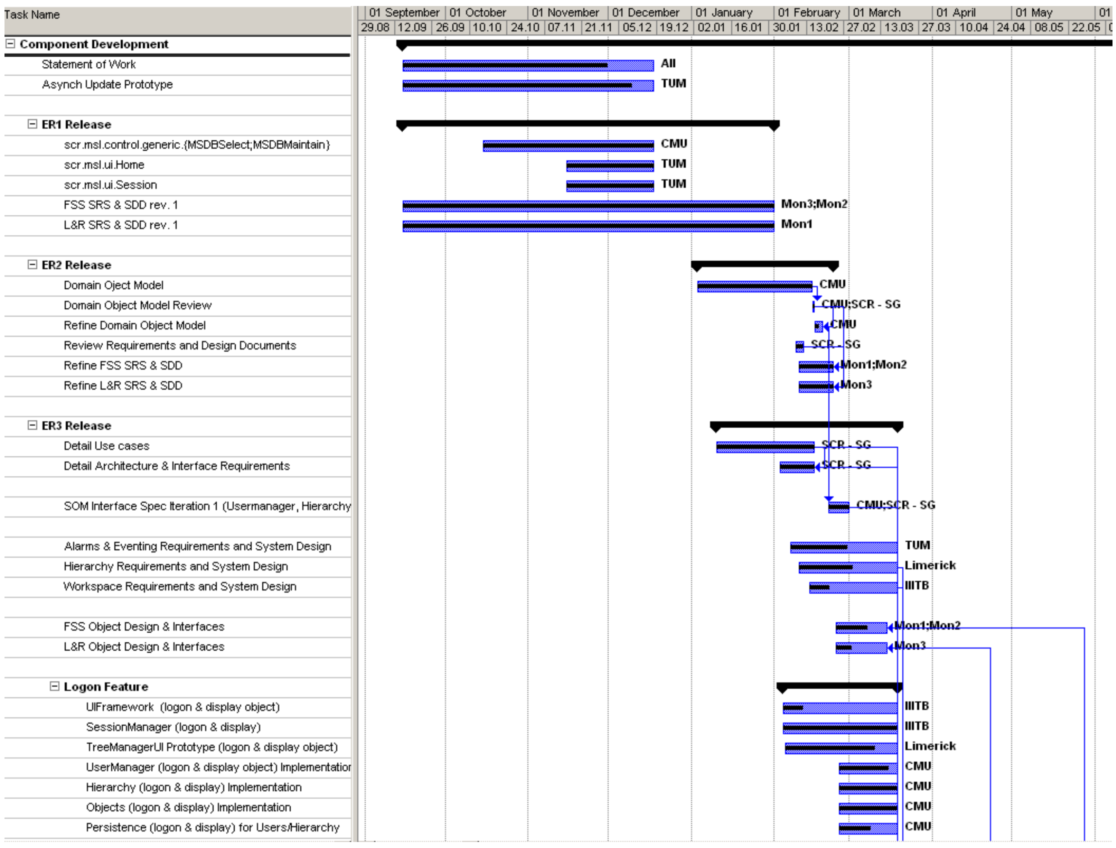
The process employed an iterative approach: refining the requirements and the design in every iteration, as well as implementing the functionality. *Engineering Releases (ERs)* were milestones set that specified the due date for the deliverables in every iteration. It should be noted that while all teams had the same scheduled ERs, the deliverables differed because of the team's status (determined from factors such as their start date and academic commitments) as well as the dissimilar features and structures of their work packages. The exact dates and deliverables are specified in the following section.

4.5 Project Timeline

The following timeline is provided in the following to give an overview about the time frame available to the project. Figure 6 is an excerpt from the project plan, demonstrating the dependencies between the tasks.

- 10-Nov-04: Release Logon Module
- 15-Dec-04: Release Statement of Work
- 01-Jan-05: Stefan Gersmann joins the central team
- 26-Jan-05: **ER1 Release**
    - FSS SRS and SDD by MU2
    - L&R SRS and SDD by MU
- 16-Feb-05: **ER2 Release**
    - Domain Object Model by CMU
    - Refined FSS SRS and SDD by MU2
    - Refined L&R SRS and SDD by MU
- 01-Mar-05: Lutz Küderli joins the central team
- 23-Mar-05: **ER3 Release**
    - Detailed Use Cases by SCR
    - Detailed Architecture by SCR
    - SOM Interface Specification by CMU
    - Alarms & Eventing SRS and SDD by TUM
    - Hierarchy SRS and SDD by UL
    - Workspace SRS and SDD by IIITB
    - FSS Object Design by MU2
    - L&R Object Design by MU

    - Logon feature release
- 20-Apr-05: **ER4 Release**
    - Integrated SRS and Architecture specifications by SCR
    - L&R implemented and tested by MU
    - FSS implemented and tested by MU2
    - Eventing implemented and tested by TUM
    - SOM iteration 1 implemented and tested by CMU
    - Workspace implementation by IIITB and TreeUI by UL not on time
- 30-May-05: **ER5 Release**
- 18-Jul-05: **ER6 Release (partially cancelled)**



Task Name	
⊟ **Component Development**	
Statement of Work	All
Asynch Update Prototype	TUM
⊟ **ER1 Release**	
scr.msl.control.generic.{MSDBSelect;MSDBMaintain}	CMU
scr.msl.ui.Home	TUM
scr.msl.ui.Session	TUM
FSS SRS & SDD rev. 1	Mon3;Mon2
L&R SRS & SDD rev. 1	Mon1
⊟ **ER2 Release**	
Domain Oject Model	CMU
Domain Object Model Review	CMU;SCR - SG
Refine Domain Object Model	CMU
Review Requirements and Design Documents	SCR - SG
Refine FSS SRS & SDD	Mon1;Mon2
Refine L&R SRS & SDD	Mon3
⊟ **ER3 Release**	
Detail Use cases	SCR - SG
Detail Architecture & Interface Requirements	SCR - SG
SOM Interface Spec Iteration 1 (Usermanager, Hierarchy	CMU;SCR - SG
Alarms & Eventing Requirements and System Design	TUM
Hierarchy Requirements and System Design	Limerick
Workspace Requirements and System Design	IIITB
FSS Object Design & Interfaces	Mon1;Mon2
L&R Object Design & Interfaces	Mon3
⊟ **Logon Feature**	
UIFramework (logon & display object)	IIITB
SessionManager (logon & display)	IIITB
TreeManagerUI Prototype (logon & display object)	Limerick
UserManager (logon & display object) Implementatior	CMU
Hierarchy (logon & display) Implementation	CMU
Objects (logon & display) Implementation	CMU
Persistence (logon & display) for Users/Hierarchy	CMU

Figure 6. Project Plan Sample

4.6 Experiences and Observations

4.6.1 Inconsistent or Unclear Architectural Specifications

Traditionally, some major problems occur when transforming the architectural description to the low-level design and then the implementation. The design choices made by the developers when they encounter under-specified, incorrect, or conflicting specifications or when they have trouble understanding it can have serious consequences on the outcome of the project.

Another set of inconsistencies stems from the different ideas developers have about the way a system is built. In the MSLite project, the CMU team defined an interface for the MU team to implement against, since their implementation was not visible to the MU team. The MU team, however, asked why certain classes that they had to use were not available, because they wanted to implement their code directly using the classes CMU developed, instead of implementing them on their own.

Other problems can arise when the architectural specification is not detailed enough and the method for refining the design or implementing a component is ambiguous. The behavior of the system under certain circumstances is sometimes unspecified and cannot be deduced from the requirements. One question the CMU team had for the central team concerned the behavior of the alarms that were triggered by the FSS: *"If the alarm condition changes back to normal, what should happen to the alarm object? Should the status change to 'closed', should the alarm be deleted, or something else?"* At the time this issue was recognized, no other team had encountered it yet, and they just assumed that one of the options presented above was correct. "If the team had not brought this issue to the other teams attention early on, this could have led to an *architectural mismatch*. Garlan et. al (Garlan et al., 1995) state that the reason for such issues arises from different assumptions the developers make about two separate entities in the system, which leads to problems e. g. when one of the components invokes the other one.

Changes to the architecture during the project caused difficulties as well. In the beginning, for example, the authentication was specified to be implemented using tokens, which were created by the SOM and passed to the components in the presentation layer. To perform an action at the SOM, the token had to be passed as credentials. Later on, when the architecture was modified, the session manager in the presentation subsystem became responsible for this task. After this design change, there was some confusion between the teams, as some were already adhering to the new way, while others had not updated their components yet.

Problems can occur when the architects make assumptions about the knowledge and abilities of the distributed team. Without the ease and efficiency of face-to-face interactions to resolve these problems, the central team spent a great deal of effort dealing with concerns such as the following:

At one point during the course of the project, the central team realized that the MU team was not sure about the purpose of an interface, as opposed to a class. In one e-mail, a team member writes, *"I am also new to using interfaces and can not seem to use them properly"*. After recognizing this issue, future communication about the architecture was enhanced to suit the team.

The development teams face a decision between four basic options every time one of these issues comes up:

1. Communicate with the person in charge of the architecture to clarify the open issues. This choice can lead to a fast resolution of the issue if the responsible person is easily accessible and knows the answer. However, if the communication does not work out that well or the architect has to think about the problem himself, it can take a long time and the progress of the team can be stalled.
2. Try to gather as much information as possible from the available resources, such as requirements specifications or experts on the site, then draw some best-guess conclusions and work from those. If the team has experienced a slow response to their requests in the past, they are more inclined to follow this approach.
3. Make up an ad-hoc solution to enable some progress to be made that can be shown as a evidence of effort to superiors.
4. Do nothing and hope the problem will go away by itself.

Although the architecture of a system describes a model of the system and many implementation details must always be left to the developers to determine, when developers stray far from the architectural path, this can lead to severe consequences for the whole system. In the MSLite project, the structure of the Presentation subsystem was underspecified in the original architecture documentation. Thus, while the central team expected the designated team to create a thorough specification and structure for this subsystem, the team members simply began implementing it with ASP pages and without any visible structure. The central team architects had expected a much stricter encapsulation of the subsystems.

This problem was a serious issue throughout the MSLite project, because the architectural description of the system was significantly underspecified when the work was distributed to the teams. Originally, it was expected that the design would be refined during subsequent iterations in the project. Instead, this strategy led to confusion and incorrect design strategies, causing delays and bad morale among the developers.

In distributed environments, the issue of asking the architect for clarification is a much larger problem than in co-located environments, because a formal request for clarification – even if it is only an e-mail – must be made. Additionally, team members have a harder time admitting that they did not understand a specification over e-mail, especially those sent to a mailing list. This problem is intensified by the fact that under these circumstances, the architect and the developers usually do not have a close relationship, as they rarely have face-to-face interactions.

Interviews conducted with the project participants even showed that communication with the central team accounted only for 16 per cent of the problem resolution activities (Figure 7).

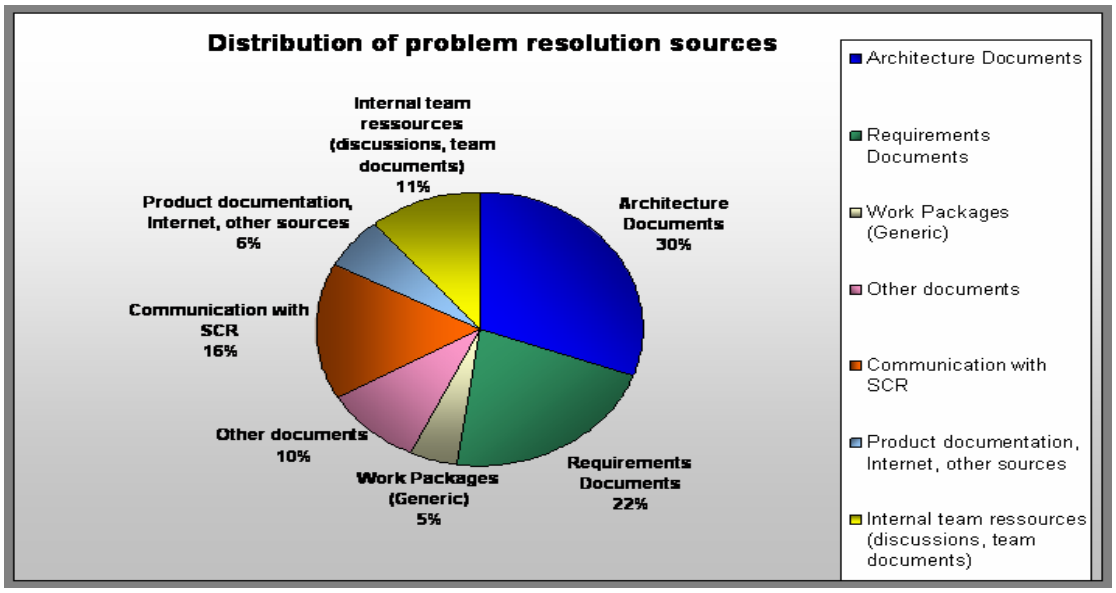


Figure 7. Distribution of problem resolution sources

In the MSLite project, it was often observed that teams chose the variants 3 and 4 of the above-listed options. Interestingly, there was a negative correlation with the skill levels (as judged by the central team) of the development teams. The highly skilled CMU and TUM teams asked lot of questions about issues they encountered, because they were confident enough to believe that the problem was within the specifications and not because of their lack of understanding. The less-skilled teams, however, were hesitant to inquire about concerns, possibly fearing that this would be interpreted as a weakness. Therefore, parts of the system were designed and implemented without adhering to the architectural specification, and a considerable number of problems had to be later corrected.

The straightforward solution for this problem is: specifications should be clear, complete, and consistent. Nevertheless, in practice this goal is not always achievable, and in fact, a less-than-perfect specification is the norm. Additionally, very good specifications can still be misinterpreted by other project members. A number of methods are available to guarantee that such ambiguities and misunderstandings are uncovered early on in a project.

In the MSLite project, tool support was used to guarantee a basic conformance to the specifications. By making exhaustive use of configuration management and automated builds, the central team was notified every time the system did not compile after a developer had checked in, modified or added source code or resources. This helped to address the most fundamental issues that occurred when code did not comply with interfaces defined by other teams. Additionally, the central team mandated that every artifact had to be checked into the central repository, creating a sense of accountability for the various documents and other resources.

4.6.2 Lack of Enforcement and Acceptance Criteria

A related problem consisted of the lack of enforcement applied by the central team. While milestones were set for the different iterations and certain artifacts were deliverables for these milestones, two issues were not resolved to the satisfaction of all participants. On the one hand, there was no effective way of measuring the quality of the artifacts. While the central team gave feedback by annotating documents and issuing tickets for problems in the source code, it was not evident when a team had really fulfilled the acceptance criteria for this milestone.

While we encouraged using a central repository which was located at SCR in Princeton for all project-related documents, the teams were hesitant to check-in intermediate results, preferring to deliver the artifacts only at the specified delivery date. This can lead to bug-prone results that are hard to fix. For example, we specified as one task for the IIITB team to submit a subsystem specification. After this document was not delivered at the planned deadline, we requested it during a phone conference and learned only then that they did not know what a subsystem specification was. If feedback had been given earlier, the central team could have provided a more in-depth explanation. Another time, we received a code delivery that did not comply to the system architecture at all. The central team was aware that some parts of the overall architecture were underspecified, but it had not been prepared for outright non-compliance. The problem it faced at that time was two-fold: On the one hand, there were no exact guidelines that specified what a correct result would be. On the other hand, while it is somewhat easier to judge the conformance of the code implementation by using automated builds, tests, coding conventions checkers etc., no real criteria existed to judge the quality of a document.

This lack of acceptance criteria for artifacts, particularly for non-source code, made it difficult to assess their quality and, even more importantly, to communicate the feedback to the remote teams. Consequently, the enforcement of standards and principles became much more difficult. When the teams were not provided hard criteria for how their work can be judged, some tended to deliver work products of inferior quality.

While this is true of a co-located development environment as well, the missing feedback loop encourages this kind of behavior. If the teams are physically distant from the coordinating instance, it is much harder to judge the progress and the status of the work package on which the team is working.

4.6.3 Different Levels of Skills and Experience

A large problem arouse from the different levels of education and experience the members of the teams had. At the beginning of the project, it was very hard for the central team to assess the abilities of the teams. Nobody at SCR had worked with any of the students at the remote sites before, or had been in extensive personal contact with them. This led to a distribution of the work packages to teams based on insufficient data.

Interestingly, the members of the individual teams all seemed to be quite homogenous – the intra-group differences were perceived as relatively small, as far as it could be judged by the central team. Over the course of the project, however, it became obvious that the skill levels of the teams varied by a larger extent than expected. Two possible reasons for these observations are the amount of industry work the individuals have already been involved in and the focus of their studies. The quality of the produced artifacts was evaluated by the central team and ranked on a scale from zero to 10; the results of this are shown in Figure 8.

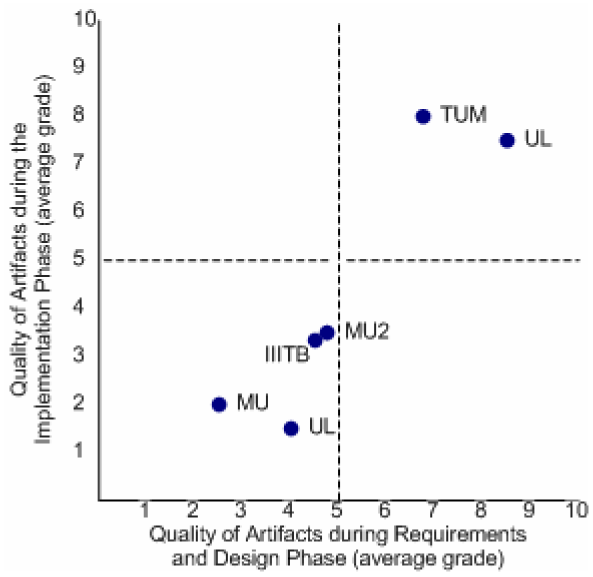


Figure 8. Skill levels of the distributed teams

While the level of the technical understanding of the technologies employed in the project differed, the resulting issues had far less impact on the overall project than the lack of knowledge about software engineering techniques and procedures. Most of the teams had not gathered exhaustive experience with .NET before, but due to the skills in other object-oriented languages such as Java, it did not have a strong negative influence. The missing familiarity with writing specifications, using tools such as the configuration management system, and the perceived lack of an overall picture of the project led to bigger problems.

We noticed that two teams focused only on their own components, trying just to fulfill the requirements for their part without being overly interested in the system as a whole. It was a lot more difficult to communicate changes or clarifications to these teams, and additionally the amount of issues raised by them was higher. The distributed approach supported this kind of behavior, as informal communication between members of separate teams was nonexistent.

The more experienced teams were comfortable with a high-level description of the components they were responsible for, even eager to take the general specification and write the lower-level one themselves. Other teams, however, left the central team with the impression that they would have liked to receive a specification going into the smallest detail that could be transformed into actual code without any additional work. When the original architecture had been designed, this aspect had not been taken into account. The level of detail was about the same for all parts of the system. In hindsight, the differences between the teams should have been taken into account. That way, every group would have been provided with the level of detail that would have fit their abilities.

4.6.4 Communication Issues

The central team had two basic options: it could either enforce a certain way of communication between itself and the development teams, or it could try to adapt to the approach the people at the remote location are used to. In the MSLite project, the use of mailing lists instead of direct mails was mandated, as well as the usage of a central repository for all project artifacts. At first, the central team decided to conduct almost all communication on an as-needed basis.

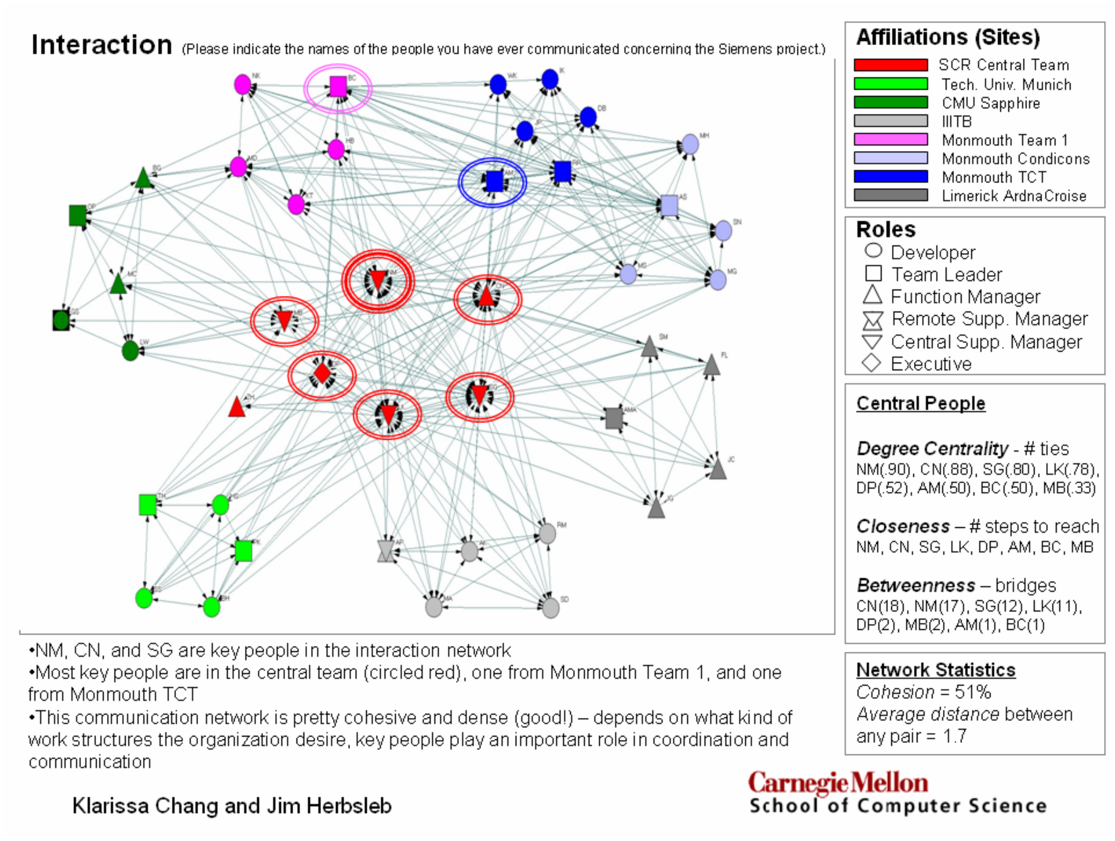
One problem with distributed teams is that due to the different time zones, unscheduled synchronous communication is hard or even impossible (the central team had – apart from scheduled meetings – no overlapping business hours with the IIITB team). Additionally, project participants whose first language is not English preferred asynchronous communications sometimes, as they probably felt that it is easier to express their thoughts in a written form. This combination led to a flow of communication that was sometimes perceived as too slow by the central team. While some project participants, including central team members, would have liked to use Instant Messaging solutions as one channel of communication, it was not used due to Siemens company policy and technical issues. The MU2 team used Instant Messaging extensively within their group and reported good experiences with it.

Another problem was the lack of personal meetings – except for the exceptions that are mentioned below, the contact consisted solely of phone conferences and email. While one central team member had visited the TUM and UL teams to create a personal rapport, he was not involved in later stages of the project any longer. The two full-time central team members did only know the TUM team members in person, having met them once before. When two central team members met with the teams from Monmouth University halfway during the project, all involved teams parted with a much better understanding of the project and each other's abilities and intentions. One participant said in an interview afterwards *"Meeting with Siemens last Friday in person set lots of things straight."*

4.6.5 Distribution of Work Does not Match Communication

During the course of the project, the central team realized that there were some issues caused by misunderstandings between the CMU and the MU teams. Because the central team felt that it would provide too much of an overhead to act as the mediator, it allowed direct communication between these teams, as long as it was conducted via the mailing lists so that the monitoring by the central team and the researchers was still possible. Opening this channel of communication proved to be successful and led to a resolution of the problem that would have taken longer if the central team had been the messenger between the parties.

An interesting issue arises out of this: when seeking the optimal way to communicate, it would have been much easier if the MU and MU2 teams would have communicated directly, as they were geographically located in the same place. This would probably have led to a much fast answer concerning some of the problems. These two teams, however, worked on completely unrelated modules, where no direct dependencies existed, as it can be observed in Figure 9. One could argue that the two teams should have been assigned more closely coupled subsystems, but on the other hand, the abilities of the teams did not match the profile required for this.



Interaction (Please indicate the names of the people you have ever communicated concerning the Siemens project.)

Affiliations (Sites)

🟥	SCR Central Team
🟩	Tech. Univ. Munich
🟩	CMU Sapphire
⬜	IIITB
🟪	Monmouth Team 1
⬜	Monmouth Condicons
🟦	Monmouth TCT
⬛	Limerick ArdnaCroise

Roles

- ○ Developer
- □ Team Leader
- △ Function Manager
- ⧖ Remote Supp. Manager
- ▽ Central Supp. Manager
- ◇ Executive

Central People

Degree Centrality - # ties
NM(.90), CN(.88), SG(.80), LK(.78), DP(.52), AM(.50), BC(.50), MB(.33)

Closeness – # steps to reach
NM, CN, SG, LK, DP, AM, BC, MB

Betweenness – bridges
CN(18), NM(17), SG(12), LK(11), DP(2), MB(2), AM(1), BC(1)

Network Statistics
Cohesion = 51%
Average distance between any pair = 1.7

•NM, CN, and SG are key people in the interaction network
•Most key people are in the central team (circled red), one from Monmouth Team 1, and one from Monmouth TCT
•This communication network is pretty cohesive and dense (good!) – depends on what kind of work structures the organization desire, key people play an important role in coordination and communication

Klarissa Chang and Jim Herbsleb

Carnegie Mellon
School of Computer Science

Figure 9. Interactions between participants in the MSLite project

Nevertheless, it was observed that the fact that the CMU and the central team were located in the same time zone created a very positive feedback loop. Because of the complexity of the SOM subsystem, more queries were conducted and those could be resolved faster than it could have been done if the teams would have had a time lag of half a day.

4.6.6 Changing Interfaces

When developing following a strictly waterfall-like process model, the need for interface changes should arise rarely. In reality, however, they occur frequently, and when developing following agile methods, the interfaces are modified all the time as the development of the software evolves. Therefore, changes to the interfaces have to be taken into account and should follow a process that is well understood by everyone involved.

In the MSLite project, this became an issue when some of the interfaces provided by the SOM subsystem had to be changed. The architects realized that the authentication mechanism, which worked by issuing a token to the presentation subsystem that in turn had to pass this token to the SOM with every remote call, was too complicated and had to be replaced.

As the project was following a continuous integration process, it was a main rule to have a working build at all times. This could not have been guaranteed if either the team responsible for the SOM or the Presentation subsystem team had changed the interfaces first and waited for the respective other one to follow up later. Instead, a branch in the repository was created where the CMU team conducted the changes first, then let the other team change their implementation, and then merged the branch back into the trunk of the configuration management. This guaranteed that the source code in the trunk could be compiled at all times, enabling other teams to continue with their work without any interferences. It took a substantial effort for the central team to define this process and to communicate the process itself and its rationale to the remote teams. They did not immediately see the need for such a process and felt uneasy about it at first, but after getting used to it, the process worked and reduced the energy the central team had to invest into the code integration. One main problem was that the remote teams without an exhaustive software engineering background did not know about the basic principles of branching and merging, leading to a lot of communication where the central team members had to explain these basic version control features. It has to be taken into account, however, that the amount of code produced in this venture cannot be compared to large-scale "real life" development, where the sheer extent of the project could cause considerable problems by itself.

It was difficult to set up and define the details of the above-mentioned process for the central team and the remote teams at first, but with an adequate amount and quality of documentation, these problems will probably less of an issue in the future.

4.6.7 Schedule of Teams Differs from Optimal Schedule

In most large software development undertakings, the optimal distribution of software engineers during the lifecycle of the project follows approximately a gauss distribution function. In the early stages, only a core team of domain experts, architects, and requirement elicitation specialists is required. After the coarse design has been done, more people are needed for a finer-grained design, and even more for the actual implementation of the software. The system tests and integration during the final stages require fewer personnel again.

In the MSLite project, the teams already joined very early, when much of the initial work that had to be conducted at SCR had not been done yet. To use these available resources, tasks where handed out before they had been finalized. Some of the decisions made then changed as the design of the architecture and the refinement of the requirements went ahead. This resulted in work that had to be redone. If the teams would have joined later or the central team had started to conduct work earlier, these problems would have been avoided.

This issue is not confined to a research project. Typically, when large-scale software development that spans across multiple business units in an organization is started by the management, one tries to staff it with a large body of personnel in the hope of achieving faster results. Additionally, politics have to be considered, as every involved party is interested in taking a lead role in the project. Therefore, they are interested in placing their people in the core team that is responsible for major design decisions, inflating its size.

4.6.8 Further Observations

There are some other observations related to the distributed setup of the project. They highlight the problematic aspects of distributing the development of software in general, but are not directly related to the design of the architecture or issues related to it. They are briefly listed below.

- **Lack of a project kick-off**. Some external constraints, such as the different academic calendars of the student teams, imposed restrictions upon the schedule of the whole project, and it was not possible to let all teams start with the project at the same time (which is unlikely for real projects as well). A formal project kick-off, at least separately for each team, would have helped. During the kick-off meeting, which should been conducted preferably on-site or at least during a videoconference, an overview about the process which the central team wanted to employ as well as about the general project could have been given. This would have created a feeling of involvement with the teams and given them a much better impression of what the central team expected, than starting with reading of documents could do.

- **Unstructured feedback by the central team.** There existed no clear guidelines what kind of feedback the central team was going to give after the distributed teams submitted project artifacts. Sometimes, those were reviews by commenting on the received Word documents, often, emails were sent back, and on other occasions, an informal feedback was given during a phone conference. While this could be interpreted as an as-needed way of conducting business, it made it harder for the teams to understand how the quality of the artifacts was perceived by the central team. For the future, a more formal approach is suggested.
- **Information and communication overload for the central team.** The decision to channel all communication between the teams through the central team facilitated a good understanding of the activities that were going on, and it makes the analysis of the data for the research associated with the project easier. As seen in Figure 9, the communication between the different teams – with the exception of the teams from Monmouth University, which were geographically co-located – followed this communication pattern throughout the project for the most part. Nevertheless, because of this, the central team had to deal with an enormous amount of communication. This led to delays and added an additional layer of complexity.
- **Upper-level management issue harder to explain.** When the management made decisions such as the removal of features or changes in the schedule, the central team had to communicate and explain these to the distributed teams. The rationale for those kinds of decisions is harder to understand for remote than for co-located teams, as there is much less insight into the political issues at the central location.
- **Cultural differences** are always a problem when teams from multiple countries work on the same project. Typical examples include the confidence the teams have in their abilities and in admitting problems. We observed for example, that in the weekly survey, the team from India was the only one who stated to have very high confidence in the successful design and implementation of their

component throughout the course of the project (cf. Figure 10), while it was judged as underperforming by the central team.

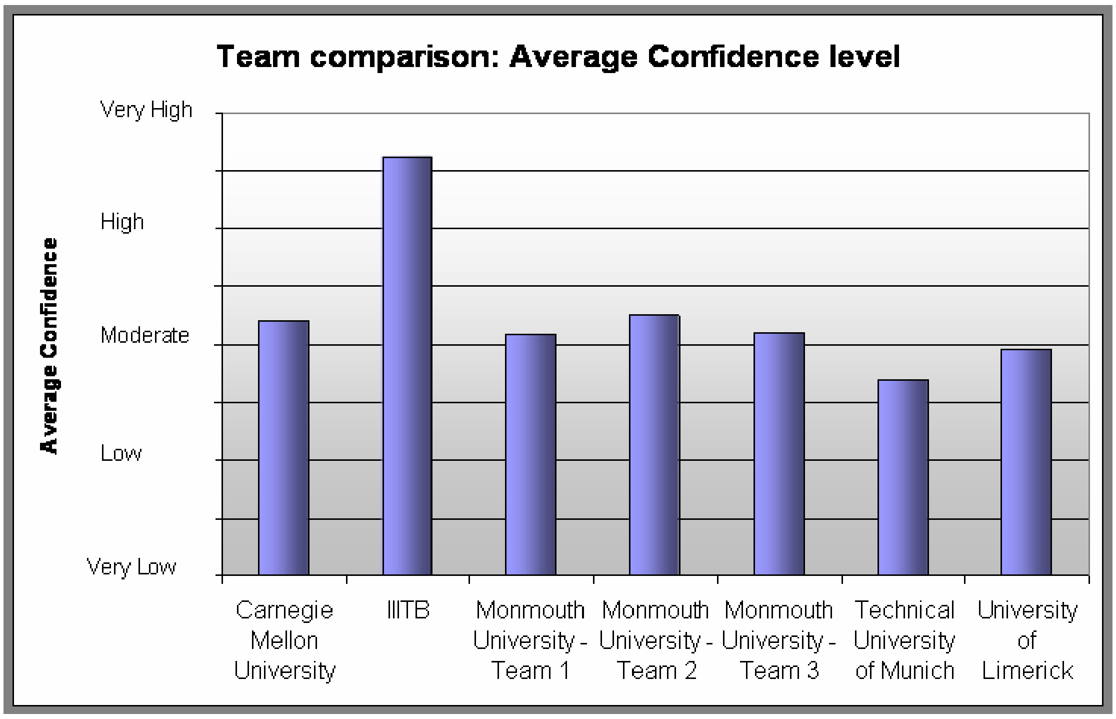


Figure 10. Average Confidence Level stated by Teams

4.7 Hypotheses

Based on the experiences listed in the last section, we are going to use this part of the thesis to put down some initial hypotheses. We are going to use these in Chapter 1 to extract the goals an architecture for projects developed in a distributed fashion should match.

4.7.1 Work Package Structure vs. Subsystem Design

During the course of the project, it became obvious that in some cases, a match of the organizational structure and the partition of the software design itself are obvious and can be conducted with minimal overhead. In many cases, though, it is imperative that this assignment has to be considered carefully. Therefore, we need a metric that enables us to assess the quality of such an assignment. This can be derived from observations such as the one in section 4.6.5. We expect that there is a correlation between the skill sets of the teams and the work allocation.

4.7.2 Interface Issues

Issues arising out of interface mismatches and communication failures concerning interfaces were expected from the beginning of the project and regarded as a major focus of the research aspect of the MSLite project. During the course of the experiment, it became obvious that this assumption would be validated; experiences are described in sections 4.6.5 and 4.6.6. Therefore, a metric is required that examines the relationships between the locations and other variables concerning the different teams, as those are supposedly influencing this interface issues. Additionally, we estimate that the structure of the architecture concerning the visibility of certain aspects of the different modules is influencing the number of problems arising from interface mismatch issues. Another concern is the communication structures between the teams and their relationship to the interface definitions. We assume that if the communication channels are similar to the flow of information within the system under development, less problems are going to be encountered during the implementation phase.

4.7.3 Enforcement

We noticed that it is hard to enforce and check the progress of the involved teams (cf. section 4.6.2). Therefore, we assume that the architecture has to support inherent means of doing automatic as well as manual checks whether the implementation corresponds with the architectural specification. We assume that this is going to lead to reduced costs of communication and fixing errors that derive from differences between specification and the work that was actually done. This can be investigated by employing a metric that gives a value for the possibility of enforcement of the architectural specification during the later stages of the project.

4.7.4 Communication Issues

We observed that failures in communicating the intentions of the architecture were a major setback for the project (cf. section 4.6.1). We hold the opinion that the architectural specification has to be formulated in an unambiguous way, in distributed development even more so than in co-located development. By establishing a metric to check whether the architecture can be communicated clearly, and whether changes to it can be made without difficulties, the architect can save time and effort when establishing the specifications for the project.

Chapter 5
Criteria, Metrics, and Measurement

There are two possible outcomes: If the result confirms the hypothesis, then you have made a measurement. If the result is contrary to the hypothesis, then you have made a discovery. –Enrico Fermi

5.1 Criteria

There are a number of different scenarios, in which the need for the evaluation of a software development project or the software itself arises. ISO/IEC 9126 (ISO, 1991) gives an overview about a variety of evaluation requirements, such as

- a user or a user's business unit could evaluate the suitability of a software product using metrics for quality in use;
- an acquirer could evaluate a software product against criterion values of external measures of functionality, reliability, usability and efficiency, or of quality in use;
- a maintainer could evaluate a software product using metrics for maintainability;
- a person responsible for implementing the software in different environments could evaluate a software product using metrics for portability;
- a developer could evaluate a software product against criterion values using internal measures of any of the quality characteristics.

In general, one usually wants to measure whether the *quality* of software is sufficient. Further work describes in detail the conformance to requirements (Crosby, 1979) or the fitness for use (Juran and Gryna, 1970). In this thesis, we are particularly interested in the quality requirements of a system's architecture. The stakeholders who require measuring the quality are usually the architect and the project manager. The architect has to make sure that he makes the right architectural decisions, while the project manager has to evaluate whether the architecture conforms to the resources he can commit to the project.

By now, it should be clear that the design of an architecture that is going to be developed in a distributed fashion will have to deal with additional problems, and that certain issues concerning the design are becoming more important compared to the co-located approach. Therefore, we have to establish a set of criteria that can be used to evaluate such an architecture. According to (Clements et al., 2002a), two main questions have to be answered:

- Is this architecture suitable for the system for which it was designed?
- Which of two or more competing architectures is the most suitable one for the system at hand?

Investigating the architecture of a system enables us to predict certain properties before it has been build. One of the most important reasons why criteria have to be applied is to establish an objective way of conducting risk management. If an architectural evaluation is performed during the early phases of a development effort, these criteria can be used to decide whether the architecture is suitable for the problem at hand. As it is much cheaper to find out the discovered problems at such an early stage, a thorough architectural review can save a substantial amount of money.

It often happens that an architecture is designed and implemented (or at least partially implemented) without expecting a distributed development approach, but the changing circumstances influence the project in such a fashion that this has to change. Then, a set of criteria provides a lead for evaluating the existing architecture, showing where problems might arise in the future, which parts have to be changed or even whether some important aspects have been left out.

There are two approaches of dealing with evaluation criteria. The first one is not to quantify them, but to use them only as guidelines to examine critical parts of the system. The review process itself is the most important part, as it enables the participants to discover problems which were not found before or which had not been deemed as severe. The expected outcome is a consensus of the strengths and weaknesses of the architecture and an increased awareness for problematic issues. A typical example is the *Architecture Tradeoff Analysis Method (ATAM)* suggested by the SEI (Clements et al., 2002a). The authors sum this approach up as follows:

"*An architecture evaluation doesn't tell you 'yes' or 'no,' 'good' or 'bad,' or '6.75 out of 10.' It tells you where you are at risk.*"

Other methodologies use metrics to quantify the architecture according to certain criteria. If a review is conducted, it is possible for the reviewers to attach a level of severity to the observations they made. It has to be considered, though, that estimates performed in the early stages of a project are often far off the actual values which are measured out later (Boehm, 2002).

5.2 The Goal Question Metric Approach

One very common way of measuring certain attributes is the *Goal Question Metric Approach (GQM)* described in (Basili, 1992). Its core philosophy "is based upon the assumption that for an organization to measure in a purposeful way it must first specify the goals for itself and its projects, then it must trace those goals to the data that are intended to define those goals operationally, and finally provide a framework for interpreting the data with respect to the stated goals."

Therefore, one has first to define clear goals that specify the intent of why the examination is conducted. A goal has a *purpose*, deals with an *issue*, has an *objective*, and is seen from the *viewpoint* of a specific group of people who have an interest in the project.

In our case, the general goal

• has the *purpose* to	analyze and evaluate
• according to the *issue* of	the distribution of work to separate locations
• with the *objective* of	improving the software architecture
• from the *viewpoint*	of all stakeholders, particularly the architect.

This general goal is broken down into sub-issues in the following sections.

We first define a number of goals that the architecture should meet in order to be suitable for distributed development. Afterwards, criteria are listed which are used to assess the goals, and metrics are proposed to measure to which degree an architecture complies with the criteria.

5.3 Metrics

After we have stated the questions, they have to be answered by employing concrete sets of measurable attributes. If a theory or a model is not proven by measuring the output, it cannot be validated, or as stated by Tom DeMarco (DeMarco, 1986), *"You cannot control what you cannot measure"*. The need for a disciplined approach toward measurement and metrics has been researched thoroughly, e.g. in (Goldenson et al., 1999). Metrics can be applied to

- Estimate project budget and schedule.
- Evaluate individual productivity and quality.
- Evaluate project productivity and quality.
- Evaluate software quality.

Every metric that one intends to use in a project has to be investigated carefully. While it is an advantage to have tangible numbers instead of making decisions based on opinions and gut feelings, a bad measurement can skew a metric, creating a false sense of security.

It is possible to classify metrics into two different areas, *product* metrics and *process* metrics. While product metrics deal with the output of the project, such as specifications and code, process metrics are concerned with the development process, such as the type of process, or the skills of the software engineers.

Another important distinction can be found between *subjective* and *objective* metrics. An objective metric will result in the same output if measured by two different evaluators, while a subjective one can produce dissimilar outcomes. Examples are the number of *Lines of Code* (*LoC*) that is independent from the review team, and the outcome of an ATAM review, which is strongly related to the individual assessors. Finally, there are *primitive* and *computed* metrics (Grady and Caswell, 1987). Computed metrics consist of the combination of other metrics, such as LOC per month.

5.3.1 Scales

There are four basic types of scales that can be used to rank different measurement approaches, a description can be found e.g. in (Kan, 2002):

- A *nominal scale* classifies data as belonging into a certain group. An example is assigning people to groups according to the color of their hair (blonde, brown, black, red and others). There is no ordering among those, i.e. blonde>brown does not make sense.
- When using an *ordinal scale*, the data is ordered but the differences between values are neglected. For example, one can divide the English language skills of people into groups of native, fluent, good, basic, and none.
- An *interval scale* is ordered as well, and the scale is constant. There is, however, no natural zero. The usual example is temperature when it is measured in Fahrenheit or Celsius.
- A *ratio scale* is an interval scale with a natural zero. Examples include weight in Kilograms, distance in Kilometers, and temperature measured in Kelvin.

While a ratio or interval scale seems to be a "better" indicator for many issues, since it is more specific, one has to be careful not to feel a false sense of security because of the apparently accurate values. An example is an assessment of a certain feature of a software system with five reviewers. Four of them consider the system correct in principle, but not ready for productive use yet, while one evaluator deems it to be of high quality. When using a nominal scale, four would vote "not acceptable", one would vote "acceptable". If one would use a grading scale from 0 to 10, four assessors give out grade 4, while one hands out a 9. It is tempting to determine the average value, resulting in a 5. This indicates average quality. The first scale gives an independent decision maker probably a better picture of the situation.

5.3.2 Measurement process

A measurement process follows the steps outlined in Figure 11 (McAndrews, 1993). During the planning phase, the scope has to be identified. During the planning phase, the objectives are defined, such as the object that has to be measured as well as the procedure that is going to be used to gather the data and the metrics for evaluating it. This corresponds to the GQM approach, where first the goals (the scope) is identified, and the questions and the metrics define the process.

During the implementation phase, the actual data gathering is conducted. After this, the data is analyzed according to the metrics and brought into a presentable form. This refined data can be used to improve the process that had been employed.

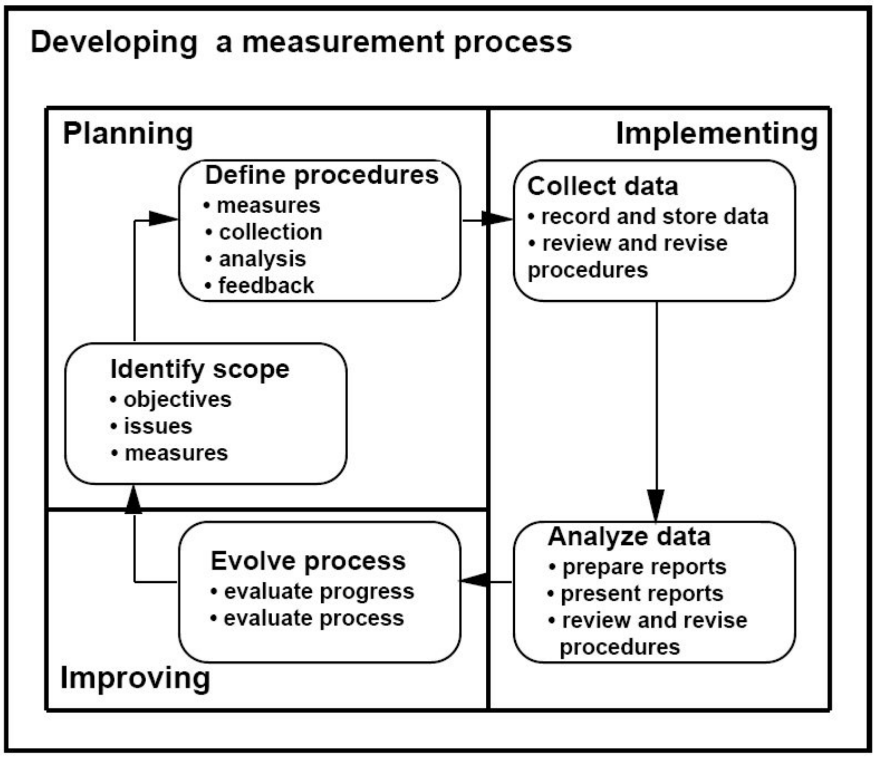


Figure 11. Software Measurement Process Architecture

5.3.3 Limitations

While measuring data can bring new insights for the stakeholders involved in a software engineering project, some risks are stated in the following:

- When the output of a measurement is wrong, this leads to a false sense of security. If none had been conducted at all, people would be more skeptical of a process or issue and watch out for problems more closely. Having a positive measurement changes that.

- The motivation of software engineers may become lower if they feel that they are not judged by a personal opinion of other people, but by simple numbers that are the output of a rigid method.
- There is always a tendency of the persons whose work is going to be judged to create output that gives a favorable result instead of the one that makes the most sense. A trivial example would be a developer who knows that the lines of code are seen as an important measurement may be tempted to use longer programming constructs instead of simpler ones.
- A major problem with many software metrics is that they do not have a sufficient experimental validation. Particularly the validity of subjective measurements has to be confirmed by empirical studies. That is the case with the metrics introduced in this thesis as well.

Chapter 6
Criteria Catalogue

A goal is a dream with a deadline.

–Harvey MacKay

6.1 Goal Definition

6.1.1 The Problem of Feedback

As the quality of an architecture that is implemented in a distributed fashion should be of the same quality as one developed in a classical co-developed manner, the criteria which are used for architecture in general have to be applied to the architecture at hand as well. The architect is always concerned with performance, communication issues, etc.

One issue is much more relevant than in co-located development, however: when concerned with distributed development, the architects have to deal with a much worse feedback loop than in classical approaches. They have to deal with the following problems:

- It is difficult to obtain status updates from remote sites except for formal, scheduled reports.
- When people do not get into contact with co-workers who work on different parts of the project, they tend to see their part of the work as the only important one. The teams have set their focus only on milestones that were put down in the project plan, seeing the fulfillment of deliveries on a certain date as the only output that is expected of them. The general idea for the whole system often takes a back seat compared to the component the team is working on.
- There is the challenge of having to deal with different cultures of communication, when a single way of using certain channels of communication would simplify the process.
- When there are changes to the architecture, it is hard to make sure that all the teams are in the same state.

From these issues, a number of goals can be deducted:

Goal 1	A feedback loop shall enable the architect and the project manager to understand the status of the project.
Goal 2	The remote teams shall have a clear, unambiguous set of objectives that they have to meet.
Goal 3	The criteria for these objectives shall be measurable.

The following table maps these goals to a number of questions:

	Goal 1	**Goal 2**	**Goal 3**
How does the architecture provide for ways to check against the interfaces?		X	
How well does the architecture ensure that for every aspect of the system, one team takes responsibility, and can be held accountable?	X	X	X
How well does the architecture ensure that the implementation can be checked against the architectural specification?		X	X

Table 2. Goals and Question about the Feedback Loop

6.1.2 Subsystem Decomposition and Team Organization

Another set of issues comes up because the distribution of the workforce and the subsystem decomposition of the architecture should match. The reason for this is the inherent tight coupling within subsystems and loose coupling between them. To assign one component to two teams would create an enormous amount of dependencies and potential causes for errors. For that reason, even co-located teams hardly work on the same component. Main problems include

- The distribution of the experts to the teams is uneven.
- The distribution of the number of people in the teams does not correspond to the size of the subsystems.
- The dependencies of the subsystem do not correspond to the best geographical distribution, with related components developed close by.

- The flexibility of the developer to shift work packages and personnel between teams is seriously decreased.

Therefore, an architecture for a software system developed in a distributed fashion has to meet the following goals:

Goal 4	It shall be flexible enough to cope with an uneven distribution of skills.
Goal 5	It shall be flexible enough to cope with uneven team sizes.
Goal 6	More specifically, it shall match the team structure at hand.
Goal 7	It shall be possible to change the mapping between work packages and teams to a certain extent.
Goal 8	Tasks requiring coordination shall be developed at the same location or close by whenever possible.

Again, we use a table to show the correlation between goals and questions. The first question concerns all goals except Goal 8. It should be noted that the question is of the same importance for every goal.

	Goal 4	**Goal 5**	**Goal 6**	**Goal 7**	**Goal 8**
How does the architecture specification minimize the level of skills and knowledge required for implementing the subsystems?	X	X	X	X	
How good do the interfaces match the team distribution?	X		X		X
How many dependencies does the system have?			X		
How good is the relationship between flexibility in the work package breakdown and a loosely coupled system?			X	X	

Table 3. Goals and Questions about the Subsystem Decomposition

6.1.3 Communication Issues

A large number of issues are related to the inefficient and error-prone communication channels that are necessary in distributed projects, because the interactions cannot be conducted face-to-face.

- Inconsistencies in the architectural specifications take more effort and time to be sorted out than in co-located development.
- Developers are more hesitant to ask questions from remote locations.
- The coordinating team has to deal with more issues.
- Communications between remote teams (without using the central team as a proxy) are almost impossible to follow for the central team.

The following goals arise from these points:

Goal 9	Communication between the teams shall be minimal.
Goal 10	Communication shall follow structured rules to be as efficient as possible.
Goal 11	It shall be possible to communicate the architecture to the teams very clearly and unambiguously.

The table below maps these goals to corresponding questions.

	Goal 9	**Goal 10**	**Goal 11**
How can one minimize the communication between the subsystem developers?	X	X	
Are architectural styles/patterns employed to solve standard problems?		X	X
Is the architecture based on a reference architecture or a similar project?	X		X
How good is the overall communicability of the architecture?	X	X	X
Is the specification comprehensive enough, but not overly detailed?	X		X

Table 4. Goals and Questions about the Communication Issues

6.2 Conceptual Integrity

There are some authors, such as Fred Brooks (Brooks, 1975), who see the conceptual integrity of a system as the single most important factor when creating an architecture, describing it as:

"I will contend that conceptual integrity is the most important consideration in system design. It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas."

It is almost universally believed that a well-defined overall structure helps to extend and maintain an existing system (Gibson and Senn, 1989). When developing large, distributed systems, it becomes increasingly difficult to guarantee that the whole system is consistent. A number of papers, e.g. (Bratthall et al., 2002) point out how architectural consistency can be achieved on a global scale.

In distributed development, the problem is two-faced: on the one hand, multiple teams at different locations are certain to follow different approaches to solving problems. As an example, teams might have different priorities when implementing a feature. While one of them is accustomed to delivering the best possible performance, another team could be used to delivering software that can be easily maintained. (This could be measured as well, e.g. by ordering each team to create a prototype and to execute a walkthrough or a code review on both. This would uncover differences in the working procedure between both parties.) An ideal architectural specification would explain every issue in detail and unambiguously enough that one would have to ignore it to create a subsystem that violates the architectural consistency. Such a specification is usually only possible with a prohibitive amount of effort. Additionally, changing requirements often lead to changing specifications throughout the project, therefore it is one of the architect's main responsibilities to guarantee architectural consistency during the course of the project.

On the other hand, distributed development has an even larger need for a consistent architecture than a co-located development project, as a consistent architecture is a main factor in reducing the communication overhead. This has been explained in (Parnas, 1972), by stating that *"development time should be shortened because separate groups would work on each module with little need for* *communication".* Therefore, one has to find ways to guarantee architectural consistency, even though the dynamics of a distributed project drive it into the opposite direction, threatening to lead to an unstructured product.

As the concept of conceptual integrity is not formally defined, measuring it constitutes a difficult task. (Hsi, 2004) described a method of establishing a metric by excavating a computing application's ontology from the user interface and then applying graph theory concepts to identify core concepts of the application.

This technique is only of limited value for the problem at hand, however. While some user interface prototypes may exist at an early design stage, the quality of a software architecture should be measured during the design phases to make decisions about it. Therefore, we need to examine metrics that can be used before the user interface exists.

One way of judging whether the architecture is consistent in itself is by determining the existence of a reference model for the project at hand. If this reference model is used exhaustively, the likelihood that the architecture is coherent is high. Another metric is established from an evaluation of the rules and regulations that are part of the architectural documentation and are used to enforce concepts in the whole project. One example is the consistency of the error handling. If this is described in the specifications in sufficient detail, this part of the project will be consistent (assuming that these rules are enforced properly).

To create a formal metric for these kinds of observations is hardly possible, as the only basis for judging issues such as the consistency of the specifications is a textual representation in almost every case. While it is be possible to record this in a semi-structured manner, this is usually not done for large systems because the costs are too high. Therefore, the evaluators of the architecture have to put down their subjective impression of the qualities listed above. It is then possible to rank architectural alternatives and to detect differences in the attitudes of the assessors towards the architecture.

6.3 Enforceability / Verifiability / Measurability

General Question: How well can the architectural integrity be enforced for remote teams? How easy is it to verify that the work done by the teams corresponds to the specifications?

During the lifecycle of the development of a system, the architecture group depends on regular feedback by the distributed development teams. In order to track progress, identify risks, and to assess the conformance of the implementation to the architectural description, the architects need to know what work has been done, which problems were encountered, and whether the specifications were unambiguous.

To assess the conformance of the implementation, a number of different methods can be used:

- Test, including system and unit tests as well as an integration test
- Walkthroughs and code inspections
- Tests against Architectural Description Languages (ADLs) – this is hardly done outside of academic research projects
- Test against Interface Definition Languages (IDLs), such as the one specified as part of the CORBA OMG standard (CORBA, 2005).

There are two ways of implementing this: one consists of using automated checks and technical restrictions, and heavy tools support. The other way is to do it manually and let a person assess the work product. When inspecting artifacts that do not consist of source code, but rather are written in a natural language, it is almost impossible to use automated tools.

A first step in assuring that the implementation can be checked against the architectural description is making sure that a complete description exists. While this seems obvious, it is not a given in many projects. Additionally, it can lead to conflicts with many agile approaches, as the architecture can be substantially changed during the course of the project. If this happens, the implementation has to conform to the most recent architectural description. In a distributed environment, this means that all project participants must be constantly updated with the most recent architecture, leading to a large communication overhead. This is particularly important considering the fact that in many large projects, the resources are not allocated according to the optimal schedule. Often, there are many people already involved in the early stages of a project, as different business units and departments are joining it (cf. section 4.6.7).

A rather new development is the use of Aspect Oriented Programming (AOP; (Kiczales, 1996)) for checking the consistency of the code against the specification. Using AOP, one extracts so-called cross-cutting concerns that are distributed throughout many subsystems instead of just one. A typical example for these kinds of concern is the logging functionality almost every software system provides. By specifying *pointcuts*, it becomes possible to introduce special code at specified places in the application, e.g. when certain methods are called or variables accessed. To use AOP to check adherence to the specification, for example one could introduce a pointcut to monitor a particularly important variable. Then, a method can be used to check whether this variable was accessed in the correct context.

A very insightful paper (Bratthall et al., 2002) points out how the enforcement can be supported by enabling the usage of a very restricted set of tools, platforms, and technologies. By lowering the complexity of possible choices, the implementation Converges to consistency.

6.3.1 Interface Verification

Question: How does the architectural description provide for ways to check against the interfaces?
Metrics: The interfaces should be defined in a way that allows them to be checked automatically or at least semi-automatically as part of a well-defined process. This is achieved by checking whether the architectural description defines the interfaces properly. To find out whether this is true, the evaluator verifies whether every interface has been described as

- An entity in an Interface Definition Languages,
- By employing *design by contract* (Meyer, 1992), a contract that has to be satisfied by the subsystem in question has been specified, or
- By Test cases that have already been written for it.

While Architectural Description Languages seem to be a promising approach to formalization of the design of software systems, they have not yet found widespread use outside of academic environments. An overview about these languages can be found in (Medvidovic and Taylor, 2000). IDLs are utilized much more often in "real-life" projects, but it should be noted that by checking against an IDL, only the syntactical accuracy of a system is verified. It is not possible to test whether the semantics of the interface are correct, i.e. whether the functionality that one subsystem is going to provide to another is given. Therefore, IDL use can only be one part of interface checks.

Design by contract (DBC) is based on the principle that the caller of a routine meets certain *preconditions*, in addition to fulfill a set of *postconditions* that will be true after the method finishes. When these conditions, or *invariants,* are accomplished, the caller can be sure that the correct result will be returned. No checking whether the result makes sense has to be done, as this is guaranteed by the contract specification already. The advantage of employing DBC (which is dependent on whether the programming language implements this feature) can be found in the fact that because every entity has a formally, clearly specified contract to adhere to, the verification whether it fulfills this contract can be done rather easily. Therefore, it allows the architect to test the quality of the work delivered.

A qualitative metric for this issue evades definition, as e.g. the quality of the test or the completeness of the contract cannot be described in an unambiguous and absolute way. One could imagine a metric that relates the number of interfaces described in the above manner relative to the number of all interfaces that are part of the system.

The most useful measurement is to use an ordinal scale and compare two different parts of the system or two prototypes of the architecture with each other. The architect has to decide then which of the alternatives represents the better one. If more than two options exist, all can be ranked relative to the others.

6.3.2 Separation of Concerns

It is general knowledge that when a large product is built and a problem is discovered during the later stages of the project, such as the integration itself or the integration tests, it can become difficult to track down the problem to a certain part of the system.

In a distributed development environment, this issue can be extended to the question of accountability and responsibility. When teams are working under the impression that they can successfully argue their case of not being liable for a certain set of problems, it is hard to create an effective enforcement policy. Therefore, one goal of an architecture for distributed development it to divide the system into separate entities with respect to different teams. If this is taken to the extreme, one can make sure that different teams compile different executables – the different parts of the software only depend on each other during runtime instead of during compile time. Some of these actions are described in detail in section 6.4.4 as well.

Another related issue is the problem of missing trust. In a co-located environment, the people developing a product know each other personally and can create a sense of trust among them. If a member of such a team causes a problem somebody else on the same location has to deal with, he is aware that he will have to meet this person face-to-face. When teams are distributed all over the globe and one knows that his actions may influence a member of another team, but that he will never have to explain himself in person, the individual will have a less significant feeling of commitment and responsibility.

Question: How well does the architecture ensure that for every aspect of the system, one team takes responsibility, and can be held accountable?
Metric: While most measures for this purpose are process-related, such as having regular status update meetings and holding managers responsible for requirements and features, there are some aspects where the architecture can support this. One part of the architectural description should be a table that lists all components of the system in the left-hand column and the responsible team in the right-hand column. If one component is assigned to more than one team, the issue should be looked into in detail. The same applies if no team is assigned the responsibility for a certain component. In these cases, the architecture should be restructured so that there is a n:1 mapping between components and teams. If this is not possible, the architect at least can consider this distribution of responsibilities and estimate potential cases for future problems.

Component	Team
Component A	Team 1
Component B	Team 1
Component C	Team 2
Component C	Team 3
Component D	
Component E	Team 4

Table 5. Responsibility Matrix

As a formal metric, the percentage of components assigned to two or more teams as well as that of unassigned subsystems should be deducted from 100 percent, resulting in a metric where a higher value points to a better score (on a rational scale). In the above table, where Component D is unassigned and Component C is assigned to two teams, this results in a score of 60 per cent.

The results of this measurement can be used to compare several architectural variants. It is, however, possible to compare two different systems, as the resulting value does not depend on the structure of the system – the separation of subsystems should be independent from the application as well as the solution domain.

6.3.3 Check of the Implementation against the Specification

During every software project, checks have to be in place to ensure that the artifacts created during the project as well as the final product conform to the specifications agreed upon upfront with the customer. An important prerequisite for this is a complete architectural specification. If this document is not complete or is ambiguous, developers are likely to implement the system on implicit assumptions about the nature of the project, instead of following the specifications.

Question: How well does the architecture ensure the conformance of the implementation to the architectural specification?
Metric: The metric for this question is defined as a checklist of issues, each of which is an indicator that the implementation can be checked against the architectural description during various stages of the project (cf. Table 6). The status of each archictectural element throughout the system as a whole can be quantified by supplying

a fraction. One could e.g. imagine that only critical parts of the architecture are specified with IDLs, putting a 0.5 instead of a 1 into the second column. It therefore describes a kind of measured value. The third column indicates a value quantifying the architectural element's importance – as subjectively estimated by the project manager and the architect. This modifier depends on two main factors: one is the experience the assessors gathered with similar projects in the past; the second one is concerning the environment and the specific project properties at hand. It is apparent that choosing such a modifier correctly requires an experienced architect.

The result of the multiplications shows up in the last column; the sum of this column indicates a value for all issues. This metric can be used to compare two projects with each other, as the influence factors are independent from the size of the system. When using this information to compare system, one has to be mindful of several issues. While a higher score indicates that this system's conformance to the specifications is more easily checkable, the quality of the employed techniques could usually only be judged in a very subjective manner. Additionally, modifiers can skew the impression completely. E.g., a system that is completely specified by using contract-based design techniques can be depicted as less easy to check against specifications than one that has some automated builds and checks if the modifiers are used accordingly.

Issue	**Used**	**Modifier**	**Result**
Design by contract			
Use of IDLs			
Use of ADLs			
Use of AOP			
Tool support for checking the model against the implementation			
Simple interface structures (such as Java interfaces)			
Technical restrictions, e.g. automated builds and checks			
Use of a reference model or architecture			
Project-specific criteria added by the architect			
Sum			

Table 6. Checklist for the implementation conformance

Example from the MSLite project: The MSLite project had no formal specifications. All documentation was handed out in the form of informal text and UML diagrams. While the design was based on the standard three-tier pattern, it did not follow any thorough reference architecture designs, as the system being developed was attempting to solve an uncommon problem, so no information could be gained from academic research based on the completion of similar projects.

A mechanism for automated builds was deployed during the project. The successful build of the system was logged and unit test for separate modules were conducted. No automated integration test that checks whether the components work together at runtime was used. Therefore, mismatches between the different subsystems would were only noticed when major issues existed.

Issue	Used	Modifier	Result
Contract-based design	0	1	0
Use of an IDL	0	1	0
Use of an ADL	0	0.2	0
Use of AOP	0	1	0
Tool support for checking the model against the implementation	0	1	0
Simple interface structures (such as Java interfaces)	0.2	1	0.2
Technical restriction such as automated builds and checks	0.2	1	0.2
Use of a reference model or architecture	0.2	1	0.2
Sum			**0.6**

Table 7. Implementation conformance checks in the MSLite project

This results in a score 0.6 points out of 7.2 possible points. While we did not employ this metric in another project, this number suggests that the ability of the architects to test whether the implementation conforms to the requirements without resolving to a manual approach was difficult. This impression can be confirmed by anecdotal evidence from the central team for the MSLite project.

The issues listed in the table are relevanr to different phases of the project lifecycle. While it is possible to verify the suitability of using a reference model early in the design phase, the effectiveness of AOP practices can only be observed when the system is implemented. Nevertheless, as experiences with similar projects provide helpful indicators and prototypes of some of the early application functionality, it is possible to judge the benefits of employing techniques that are used in the later stages of the project.

6.4 Match of Architecture and Organizational Structure

Fundamental question: When mapping parts of the system to development sites and teams, is it possible to do so without having to separate coherent parts?

One of the most important decisions in a distributed project is determining an appropriate division of work among the locations that are involved. The two most obvious factors are the available work force and the specific skill set of the developers. The architect has to consider breaking the system down into work packages which are assigned to the various teams. There are two choices available to the architect:

- Create the overall architecture without respect to the specifics of the development teams, and afterwards look for subsystems that are appropriate for assignments to teams, taking dependencies and interface issues into account.
- Design the architecture from ground up with the number of teams and their properties in mind, making the team structure a key quality attribute.

Good reasons exist for both approaches: The first allows the architect to focus on runtime properties and other qualities not related to the development. The second can lead to development better suited to the challenges of distributing the workforce, but holds the danger that the focus on the aforementioned qualities is lost. In very large projects where more than three diverse teams are involved, however, an architecture where the distributed aspects do not play a major role is of dubious quality, compare e.g. (Paulish, 2001). Obviously, a hybrid approach can be chosen as well. One could imagine that a number of teams are already established, while for some parts of the project, the exact teams are not known yet.

In a good number projects, while the team structure is taken into account to some extent, most of the architecture design is conducted using the first method, and the architecture is developed regardless of the nature of the development teams. The following evaluation supports both methods: When the architecture already exists and the project has to be divided into work packages, it aids the decision of how to achieve a fit between the demand the implementation brings with it and the teams. For the second approach, it helps the architect to decide how to design system boundaries, interfaces and connectors with respect to the development teams.

To judge whether the team structure matches the architecture, two separate assessments have to be carried out first. The abilities of the teams have to be evaluated with respect to their experience in the problem and the solution domain, the culture of the company or business unit they are part of, the familiarity with the process that is going to be applied, and whether they have worked in distributed environments before. The results are typically shown in a skill matrix. (Bruegge and Dutoit, 2003) suggest distinguishing three types of entries: primary skill, secondary skill, and interest. To use an even more fine-grained method of rating the skills is often not possible, as the skill evaluators do not know all project participants good enough to make statements that are more precise. This plays an even larger role in distributed development, where the architect and the project manager may never meet the people working on a remote site before the project starts.

The architect has to document the fundamentals for subsystems or features. This depends on the kind of development approach the project is going to use. Out of these two sources of information, the matching has to be investigated. The following sections detail the prerequisites for the subsystems.

6.4.1 Different Sets of Skill and Domain Knowledge

In every development project, the skills and the experiences of the involved developers vary. In distributed development, these differences are intensified by the following factors:

- The teams usually belong to different organizational units. This means they are used to different tools as well as different processes and approaches of tackling problems. Examples include using different configuration management and design tools, using a different development process (e.g. the Rational Unified Process vs. Extreme Programming), and the communication of problems explicitly vs. those identified during status meetings. Often, the projects developed in different business units have drastically different application domains.
- The skill transfer between the team members is more difficult. When the projects members are located in the same building, it happens regularly that questions are answered in person, and that implicit as well as explicit knowledge are easily transferred. When there is a distance between the teams, this exchange becomes more complicated. Additionally, different time zones lead to the problem that even questions that could be easily answered by an expert in a certain field can take a day to get resolved. While these difficulties are easily overcome when the technical aspects of the solution domain are learned, the problem domain usually contains such an amount of implicit knowledge that there will always be a substantial need for communication.
- The educational background varies to a larger degree. One could imagine that one team consists only of German citizens with computer science degrees, and another group is composed of Indian professionals with a lot of work experience, but a less formal education. Both are at least somewhat familiar with the same basic principles of software engineering, but there is much room for differing expectations from the other side. On the other hand, the software engineers can estimate the skills by finding out about the educational and working background of their colleagues. While this can lead to misunderstandings and wrong impressions, the issue is more predominant when working with people from another country. The language barrier may also hinder communication.

Question: How does the architecture specification minimize the skills and the knowledge required for implementing the subsystems?

Metric: First, we have to judge whether the architecture allow teams with a narrow field of domain knowledge to work in an effective manner. Therefore, we investigate if the architecture is partitioned in a fashion that isolates this understanding within the boundaries of a part of a work package, preferably within a subsystem as well. To create a metric, we have to document the (often only implicitly present) domain knowledge and divide it into parts that correlate with the involved teams. As a next step, we identify the parts of the domain knowledge that relate to the subsystem decomposition. We can use this to gather an interval metric by specifying the overlap between the teams' knowledge and the work that is assigned to them.

The metric is established by creating a list of skills, for example by taking them out of the skill matrix mentioned above, and additionally, a list containing the domain knowledge issues. To separate this knowledge into discrete chunks is much harder than doing the same for the technical aspects of the project. Domain knowledge is often explicit and encompasses an area that is not clearly defined. The architect has to work closely with a domain expert to extract these parts and compile a list of them.

As a next step, we create a matrix containing the skills and the domain knowledge in the rows and the subsystems in the columns. We mark every cell that matches the combination. The sum of marked cells in one column gives a number for each subsystem. If the number is low, the set of skills needed for this subsystem is relatively low, while a high number indicates that a well-rounded team should take over this task. As the numbers that come out of this metric are not normalized, they cannot be used to compare two projects with each other, but rather can be used to compare different subsystem decompositions of the same product.

Table 8 was created for the MSLite project. It can be observed that the SOM requires the strongest overall skill set. The requirements for the other subsystems do not vary by such a large degree.

	FSS	SOM	L&R	Presentation
Problem domain				
Building management	X	X		X
Solution domain				
C#	X	X	X	X
GUI design	X			X
nHibernate		X	X	
Databases		X	X	
Remoting	X	X	X	X
Sum	**4**	**5**	**4**	**4**

Table 8. Skill/Knowlede Matrix for the MSLite project

In Table 8, this was done for the MSLite project. It can be observed that the SOM requires the strongest overall skill set. The requirements for the other subsystems do not vary by such a large degree.

When determining values by using this metric, it should be remembered that low numbers for all interfaces are an indicator for a clear separation of concerns. They do not provide an automatic best way to distribute the work on the teams, as the sets of skills needed in a certain configuration for specific subsystems may not exist in any team.

6.4.2 Interface Match and Dependencies

In a distributed development environment, the role of unambiguous boundaries – that means they are explicitly stated and understood by all stakeholders involved – and the scope of subsystems, work packages, and features are even more important than in co-located development. First, as already mentioned, the cost of clarifying issues afterwards is punitive; secondly, the coordination between teams is a lot higher. This is the hypothesis formulated in section 4.7.2, based on the experiences from the MSLite project (cf. section 4.6.6).

One can distinguish between various types of boundaries (Espinosa et al., 2002) in software development projects, such as

- **Geographic Boundaries** are the most visible borders between people working in a distributed project, as they are clearly separated by location.
- **Functional Boundaries** exist between groups of people taking on a certain function in a project. Often, cross-functional teams face a completely new set of challenges.
- **Identity Boundaries** become visible when team members are attached to multiple projects. It often happens that a software developer still plays a role in the assignment he was engaged before when joining a new project. This can lead to problems performing in the new one.
- **Organizational Boundaries** are often underestimated as all personnel involved in a project usually belong to the same company. Nevertheless, when considering outsourcing or the collaboration of different business units, it becomes obvious that the affiliation with a sub-group plays an important factor. Typically, however, organizational and geographic boundaries are similar.

While the last three types of boundaries play an important role in co-located projects as well, they become an even larger issue in distributed development. As the project manager and the lead architect cannot be on site everywhere, invisible structures and hierarchies tend to replace the official ones, creating “unofficial” communication channels.

An experience we made in the MSLite project, though it cannot be applied directly to a business environment, showed where the organizational boundaries became apparent in the MSLite project when investigating the attitude of the MU team. They were obviously more attached to their university than to SCR, valuing the existence of a running, albeit incomplete system higher than a complete specification. This led to a conflict of interest with SCR, which had different priorities.

When considering the mapping of the organizational structures to the architecture of the system under development, a number of criteria can be used to create a good match, including

- Domain knowledge.
- Experience of the personnel.
- Team size.
- The location of the team – whether it is located close by (or in the same time zone) as another team, or other stakeholders, such as the customer.

The issue of domain knowledge is already discussed in detail in section 2.7 and will not be restated here. The experience of the employees refers to the understanding of software engineering – implicit as well as explicit knowledge.

In every project, the architects are particularly concerned to minimize the number of interfaces, knowing that each of them is a potential cause for problems. In distributed development, the pressure to do this is even higher, as pointed out in section 6.3.2.

Obviously, it is not possible to get rid of all interfaces within a project. The next step deals with the interfaces in a manner that creates a minimum of interactions and clarifications between the teams. The last phase consists of finding an approach that selects the teams that have the least difficulties in dealing with resolving the issue without major intervention by the central team.

Question: How well do the interfaces match the team distribution?
Metric: (This metric can only be used if the work package breakdown has already been determined.) We create a matrix, putting all combinations of two teams in the rows and the following properties in the columns

- Cultural match
- Combined software engineering experience
- Overlapping domain knowledge
- Similarity of time zones
- Combined experience with distributed development

Every column is given a modifier according to the importance the architect and the project manager estimate that the property is going to have on the outcome of the project. One fills a value between one (specifying the least amount) and five (specifying the best fit) into the cells, multiplying it with the modifier belonging to this column. For the combined experiences, the skill matrices used for setting up the project (cf. section 6.4) can be used. Then for every pair of teams, all properties are added up in a last column, resulting in a value we define as an *Interaction Value (IV)*.

Teams	Cultural match	(x1)	Combined software engineering experience	(x2)	Overlapping domain knowledge	(x1)	Combined experience with distributed development	(x3)	Similarity of time zones	(x1)	Interaction Value
MU, MU2	5	**5**	1	**2**	1	**2**	1	**3**	5	**1**	**13**
TUM,CMU	3	**3**	4	**4**	2	**2**	2	**6**	2	**3**	**18**
SCR, IIITB	2	**2**	3	**6**	2	**2**	2	**6**	1	**1**	**18**
UL, IIITB	2	**2**	2	**4**	1	**1**	1	**1**	1	**1**	**9**

Table 9. Interaction Value Table

In Table 9, some relationships between teams involved in the MSLite project are shown. It is obvious that the score for the interaction between the teams from UL and IIITB is much lower than e.g. between the TUM and CMU students. This correlates with our experiences, as the cooperation within the Presentation subsystem did not work smoothly.

After this procedure, we create a table, specifying all interfaces and the teams that are involved for each of them. When substituting these for IVs and adding up all of them, a number results that indicates the ability of the projects participants to resolve interface mismatches. When changing the teams responsible for the subsystems, the table changes, resulting in a new value. A higher value shows an improvement.

This metric only shows how the teams within a project can be distributed to subsystems with a given set of interfaces. Using this method to compare projects with each other is not possible, as the figure contains the number of interfaces. While this could be normalized dividing by the number of interfaces, factors such as the numbers of interfaces per subsystem that can vary by a large extent make it an unreliable method for this purpose.

One of the most interesting outcomes of applying this measurement to a software architecture is that it becomes possible to determine the cost of an interface. It gives the architect one tool to estimate the probability that issues are going to arise from creating an interface between two subsystems assigned to separate teams, and can thus be used to plan ahead for delays and possible inconsistencies.

6.4.3 Build-Time Dependencies

Another important issue when designing the interfaces connecting subsystems deals with the dependencies between them. Often, the proper function of one subsystem depends on the other, but not vice versa – this is particularly obvious with many layered systems, where the top layers depend on the layers below them, but not the other way round. An example from the MSLite project is the FSS, which is independent from the other parts of the system. The interface issues that are described above deal with the interaction needed to guarantee that the interfaces by the teams match. The metric described here is regarding the issue that teams depend on each other for implemented work packages.

While it is always possible to write test drivers to simulate parts of the system that have not been implemented yet, this can lead to a huge effort when simulating complex behavior of the system. Additionally, it might not be possible to define the complete system in all details beforehand, but state that some parts of the architecture are going to be specified concerning to certain implementation details.

The whole problem is aggravated by the distributed environment. When a dependency that was not anticipated by the architects emerges during the course of the project, the effort needed for the dependent team that group, clarify the dependencies that have to be satisfied, and reviewing the changes is much higher than conducting this on site in a more informal environment. Therefore, it is the goal of the architect to assure in advance that the number of dependencies and the likelihood of unexpected dependencies arising later on are as low as possible.

The chance that unexpected dependencies occur cannot be judged by employing a formal measurement. It depends mainly on the experience of the architect, the complexity of the system, and the number of known dependencies (i.e., if a lot of functionality in a subsystem relies on another, the probability that there are more is rather large) – one more reason to reduce the overall number.

Question: How many dependencies does the system have?
Metric: This metric is simple, as it can be done in a straightforward, quantitative manner by assigning an experienced software engineer to count all dependencies within the project. Unlike most others mentioned previously, this metric *can* be used to compare two different projects with each other, as the number of dependencies is going to be a

direct influence for the work distribution, no matter how the rest of the project is structured.

As an additional data point, the number of dependencies per subsystem should be shown separately. That way, the architect has a good indicator for parts of the system under development that should be examined in more detail to reduce the number of dependencies.

6.4.4 Information Hiding

One basic hypothesis we deducted from the MSLite project can be summed by the following statement:

Assumption: The more communication between the components of the system in development, the more communication is necessary between the development teams.

The reason for this hypothesis can be found in the fact that every separate unit of communication that is exchanged between the components can be misinterpreted by the receiver. Every time such a mismatch occurs, communication between the developments teams is necessary to resolve this issue. This problem became particularly obvious when we observed the interactions between the different teams that had to implement the Presentation subsystem. As the number of different connectors within this layer was high, the teams had many questions about their interaction that had to be resolved by the architects of the system.

While similar to the interface issues described above, the problem is slightly different. When looking at modules, the focus of the description is on the functional separation, i.e. the tasks that the modules have to fulfill. The connectors in this case represent the flow of data during runtime.

The assumption above implies that the number of connectors between subsystems developed by different teams should be as low as possible. This relates to the principle of information hiding (Yourdon and Constantine, 1975). When minimizing the number of connectors, the amount of information one subsystem has to have about another one is minimized as well, which also leads to loose coupling between components.

This in turn leads to the conclusion that the principle of information hiding between the modules should be transferred to a similar approach of dealing with information between teams as well. If information is hidden within a module, it only has to be known to the team working on the module. In case the information is exchanged with other modules – as depicted explicitly in the Component & Connector (C&C) viewpoint, the teams responsible for the subsystems have to access the same information and probably exchange information as well. For a detailed background into the C&C view, see (Bass et al., 2003) and (Clements et al., 2002a).

Question: How much communication has to take place between the subsystem developers?
Metric: If the above hypothesis is true, then the communication between the developers of subsystems can be directly derived from a metric employed for measuring the coupling between modules. Such a metric can be found in (Lindvall et al., 2002). The authors classify the modules into different types and base their metric upon interaction between them and the involved type of the module (such as libraries). The measurement results in a number for each module that can be used by the architect to detect where a large number of interactions are going to take place, both during the runtime of the system as well as between the development teams that are assigned to the module.

To fit this model to our context, interactions that happen between modules that are both assigned to the same team can be deducted. The numbers that result as the output of the procedure are useful for finding problems with the design and comparing prototypes. As couplings are dependent on influence factors such as the technology that is used, the metric should not be used to compare two unrelated development projects with each other.

6.5 Flexibility of Work Distribution

General Question: How much effort does it take to transfer part of the work assigned to one team to another, without moving any team members to the other site?

In every development project, one has to deal with unexpected delays and obstacles. Often, it becomes obvious that tasks assigned to a certain team are going to take up considerably more time than was initially expected, while other parts of the development are making progress according to the schedule. One approach often used by project managers to resolve this problem is to transfer people from one team to another, anticipating that there is correlation between the number of people and project progress. This does not always apply – Brooks (Brooks, 1975) argues strongly against assigning more people to a project that is late, stating that *"Adding manpower to a late software project makes it later."* While Brooks writes about bringing in fresh people to a project, not about reassigning them internally, there are some serious issues with this course of action, which are amplified by the nature of distributed development:

- The cost of relocating staff is very high.
- Employees from a different location have less insight into other parts of the project they are working on than they would have if it happened at the same site.
- The culture of the business unit or department they are transferred to can differ greatly from the environment they are used to.

These reasons make it more appealing to move work packages between sites instead of moving personnel to another location. To accomplish this, the work packages should not be set up and distributed in a monolithic fashion, but divided up into smaller units. There is an additional benefit to this, as it allows an incremental approach of handing out tasks, which can be done according to the progress the teams are making or other circumstances within the project.

While it seems to be very appealing at first to divide the system in a great number of small work packages to gain maximum flexibility, there is a tradeoff. While a good modular design adheres to the principle of loose coupling not only between the top-level subsystems, but also at every level of granularity, the amount of dependencies – and therefore the need for additional communication and the probability of mismatches – becomes larger in the low-level stages of a system.

Therefore, the architect has to find a compromise between flexibility and loose coupling. A number of factors influence this decision:

- **The complexity of the software**. If the software is more complicated, it gets even more important to isolate its functionality into parts that are clearly distinct.

- **The architectural style.** If dealing with a pipe-and-filter style, the single parts are already split into very distinctive modules; it is not hard to find a boundary between them. This makes it easier to create a loosely coupled system.
- **The dynamics of the project.** When it is obvious from the start of development that there will be major changes down the road (or an initial risk assessment resulted in a high probability for this), the flexibility aspect gains a lot of importance.
- **The margin for error.** If developing e.g. a mission-critical system for a bank, the flexibility aspect plays a minor role compared to the advantages a clear distribution of functionality. Additionally, a loose coupling provides for a better accountability (cf. section 6.3).

Question: How good is the relationship between flexibility in the work package breakdown and a loosely coupled system?
Metric: This question cannot be answered by finding a "best" way, as there will always be a tradeoff between flexibility and coupling. The architects will have to make a decision based on the advantages and disadvantages that they discover for a distribution of these properties. When comparing two architectural prototypes, where one is providing more flexibility and the other one is more loosely coupled, they create lists of the negative implication of both designs. Then, lists with all advantages are put down; both comparisons are judged against each other.

6.6 Flexibility of Level and Completeness of Design

General Question: Is the architecture appropriate for outlining some parts in more detail than others, at every stage of the project?

In large projects with many software engineers, the architects will have to deal with people of varying background and expertise. While some developers are very familiar with the software engineering methods employed by these architects – perhaps due to an existing working relationship from a previous project, or experience from a similar project,– others break new ground. This issue has two consequences:

- General guidelines, conventions and rules have to be explained in more detail for the second group of people. This, however, does not cause any disadvantages for the experienced developers, as they benefit from an exhaustive specification as well.
- The part of the system assigned to the less experienced group has to be specified in more detail. As they are not as familiar with the way development in the environment is conducted, an architectural description that is sufficient for the first group is not sufficient for them to realize their part of the project consistently, leading to a possible architectural mismatch.

The second point brings up an interesting issue: it would not make sense to specify every subsystem in detail, as this would present unnecessary work for the architects. Instead, they should make sure that the granularity of the description is tailored to the team that is going to work on it.

An additional layer of complexity is introduced by the fact that the teams themselves are not homogenous; even in groups working together for years major differences between team members are possible. Paulish (Paulish, 2001) stated explicitly that there is a strong correlation between the success of a software development project and whether the architecture is well understood by all team members. If the individuals work with the same clear vision of the system that the architect has, architectural mismatch becomes much less of an issue. This leads to two important points: Firstly, the overall architecture has to be understood by everyone, and only specifications for subsystems can neglect that (according to the principle of information hiding, cf. section 6.4.4). Secondly, each specification of a subsystem has to be detailed enough that all team members understand its implications completely – and to minimize future risks, the probability that developers with a different skill set are added to the team should be taken into account.

Even in the face of these problems, the time the architects can save for designing a specification that is tailored to the knowledge of the teams makes up for them. The tradeoff of this approach is that the flexibility of distributing the work is severely limited. If a work package has been specified in less detail than others have because it was intended to hand it over to an experienced team, and this work distribution is about to be changed, the level of detail would have to be adjusted as well.

Question: Is the specification comprehensive enough, but not overly detailed?
Metric: The result of this metric is hard to quantify, as there will always be a tradeoff between the cost of specifying the architecture as detailed as possible and the cost of having to clarify ambiguities later on. It is up to the discretion of the architect to find an acceptable solution for every part of the architecture. Similar to the metric in section 6.6, the stakeholders decide between two proposed alternatives, which is the better one for all parts of the architectural specification.

6.7 Communicability of the Architecture

General Question: How complicated is it to communicate the architecture to the remote teams without visiting them? How difficult is it to communicate changes? How well can questions about the architecture be resolved via email, phone and videoconferences?

The criterion of the communicability of software architecture describes the effort that is needed to explain and show it to the remote teams. Consequently, it is derived from the clarity and comprehensibility of the documentation of the architecture, the abilities of the architects to explain it, and evidently a number of issues concerning the architecture itself, such as dependencies and complexity of call structures.
The question whether an architecture can be communicated properly depends on the architectural description, the complexity of the design and the abilities and the experience of the personnel involved – on the side of the architect as well as concerning the remote teams. It is possible to extract a set of concerns that provide an information basis for deciding whether some key issues have been addressed.

- Does a reference architecture exist and does it have been documented properly? Are the team members familiar with this architecture? If a reference architecture exists which is well understood by the developers, it becomes easier to communicate key issues, as the general concepts are already known. The overhead of explaining fundamental design decisions taken by the architect of the system does not apply under these circumstances.

- Does the architecture make exhaustive use of standard architectural styles/patterns? These standard ways of structuring the design are known by almost all software engineers; they minimize the amount of inquiries with whom the architects have to deal with. As all software engineers should be familiar with these patterns, it is easier for them to understand the system.
- Does the architecture incorporate some kind of contract or other method to make interface definitions highly visible to everybody involved? While every interface has to be specified, there is a difference between a verbal description in a document and a contract-based approach. These issues are discusses in more detail in section 6.3.1.
- What kind of medium is used to publish the architectural description? What approach has been chosen to address different architectural descriptions? It is possible to create lists of Frequently Asked Questions (FAQ), set up a Wiki to keep the specifications up-to-date and consistent, to use a plain Word file or a project management system.
- Has the number of interfaces between components been minimized? This obviously minimizes the effort that participants have to invest in communicating about it, if it has been specified ambiguously. Additionally, it enables the developers to focus on implementation details of the subsystems assigned to them instead of working about the interaction with other parts.
- The complexity of the system gives a good indicator about how easy it is for the remote teams to understand the design and the rationale of the architects. A large body of research has already been conducted investigating how one can measure the complexity of software. An overview about the practical approaches can be found in (Weyuker, 1988), examples include Lines of Code (LoC), Cyclomatic Complexity (McCabe, 1976), and the Halstead Complexity Measures (Halstead, 1977), as well as object oriented measurements described in (Lorenz and Kidd, 1994) and (Chidamber and Kemerer, 1994).
- Reuse of components already developed by the central and the remote teams.
- Similarity to projects already developed by the teams.

A research area that has been the focus of a lot of attention recently has been the use of architectural styles and patterns in designing an architecture; standard works about this topic include (Buschmann et al., 1996), and (Fowler, 2002). Patterns are used to solve common problems with a general solution; they can be applied to a variety of different situations. An additional benefit of using patterns is the fact that commonly used design patterns are recognized by experienced software engineers quickly. Patterns help to comprehend the general structure instead of gathering it from the composition of classes and other available documentation.

For this reason, extensive use of patterns reduces the need to communicate, as the rationale for a standard pattern can be more easily deducted than the solution for the same problem that has been created without using patterns. Architectural patterns thus constitute a kind of language that should be understood by all software engineers, and implicitly contain a lot of information.

Question: Are architectural styles/patterns employed to solve standard problems?
Metric: A prerequisite of measuring the application of patterns is their exhaustive documentation. If they were not documented in the specification, an additional step would be required where the measuring party has to extract the patterns from the design documents. While this is possible, and some tools exist to support this procedure (Antoniol et al., 1998; Gustafsson et al., 2002; Kramer and Prechelt, 1996), the results cannot be guaranteed to be comprehensive.

One metric is to calculate the ratio of classes that are part of a standard design pattern to the total number of classes in the project. This gives a rough estimate, but has its shortcomings. The number of employable patterns can vary a lot, as factors such as the complexity of the problem play a large role. Therefore, it is not possible to compare projects with each other, but only two designs of the same projects. Additionally, not all patterns can be applied on a class level, but involve superstructures such as packages. While one could count all classes involved, it creates the impression of an extensive use of patterns, even if this is obviously not the case. Thus, the metric above can be optimized by not counting the involved classed, but the involved entities.

Reference models and reference architectures are used if a standard approach of creating a complete architecture for a certain kind of projects exists. While a reference model *"is a division of functionality together with data flow between the pieces"*, a reference architecture is a *"reference model mapped onto software elements […] and the data flows between them"* (Bass et al., 2003).

The advantage of using these concepts is that – similar to design patterns – they are known to software engineers. The basic parts of a compiler or web application are common knowledge. By employing such a reference, it is much easier to communicate the architecture to the remote teams.

Question: Is the architecture based on a reference architecture or similar project?
Metric: This is a simple nominal metric where architectures can be divided into categories: those based on a reference model, those on a reference architecture, and the set of architectures that does not belong to one of these. The problem with this approach is to judge whether an architecture can be considered to be based on a reference model in the context of the knowledge of the involved developers. A simple example would be the reference architecture for a compiler. This would be no problem for an engineer with a theoretical background, while a developer without computer science knowledge would be familiar with the architecture of three-tier web container frameworks. Additionally, when a complex product is developed, different subsystems might employ different reference architectures.

As a second step, the impact the reference has on the system under development is assessed using an ordinal scale as *high*, *medium*, and *low*. The architect can use these results to investigate whether the system can be communicated clearly by using the reference model or architecture as a concept or whether additional steps have to be taken to explain it.

Question: How good is the overall communicability of the architecture?
Metric: This metric cannot be measured by a formal approach. The question whether a specification can be understood by the implementers cannot be answered by referring to a technical method. Instead, the architects would have to conduct a review of the architecture, or at least of the parts they consider particularly important or error-prone. One possibility for such a review is an *Architecture Tradeoff Analysis Method (ATAM)*. This method, developed at the Software Engineering Institute at Carnegie Mellon University, *"reveals how well an architecture satisfies particular quality goals […]"* (Bass

et al., 2003). Participants include the evaluation team, project decision makers, and architecture stakeholders. In this instance, we are concerned with the quality goal of the communicability and conduct the ATAM with a focus on this subject. Some of the output of the procedure is mentioned in the ATAM description already. Additionally, for our purpose, we let all assessors in the review give out a grade for the communicability: *good*, *sufficient*, or *insufficient*, and provide a rationale for their verdict. This ordinal scale can then be used by the architecture team to decide whether steps have to be taken to improve the specifications. If this is the case, they are already provided with some pointers.

When conducting an ATAM for projects developed in a distributed fashion, the effort that has to be invested to organize it is even greater than in co-located projects. One of the most important points of such a review is the factor that all stakeholders are put into the same room to discuss important architectural decisions. The scheduling issues when the people have to be brought in from different site can be complicated, but it is required for a satisfying outcome.

Furthermore, it is important to schedule the ATAM at the right point in time. If it takes place to early, the participants will not find a sufficient amount of artifacts and information to work with, and the outcome will not be rewarding. If on the other hand the ATAM is conducted to late, a lot of work has been done already and key decisions have been made. Therefore, it will be costly and meet resistance from many project members if major changes are suggested as the outcome.

Chapter 7
Scorecard/Overall Assessment

7.1 Purpose

This chapter deals with the consolidation of the metrics that have been collected by using the methods described in the last chapter, and using them to assess an architecture specification as a whole with respect to distributed development. While the process of evaluating the quality of an architecture will always be dependent on subjective factors (as outlined in the next section), it is important that we create a scorecard that can be used for various purposes:

- To give the stakeholders a first overview about the feasibility of the proposed architecture and potential reasons for problems.
- To compare their advantages and disadvantages and support decision-making processes if two or more architectural alternatives are proposed.
- To track improvements of the overall architecture and plan the next steps, if the evaluation is done at regular intervals.

7.2 Limitations of Measurement

One point is important: the metrics introduced in the last chapter are a subjective way to put a number to a system from the viewpoint of an independent observer. They are, however, a good basis for discussing certain aspects of the system under development. To gain objective results, the use of e.g. modifiers that are established by the evaluators (cf. section 6.3.3) would not be possible. These influences may or may not have been intended by the evaluators, and may lead to a wrong impression about the qualities of this architectural variant when compared to another one.

The modifiers and the use of soft measuring criteria such as taking the outcome of a modified ATAM into account give the architects tools that can be used to evaluate alternatives. When a metric has been applied to two prototypes of the architecture (or a part thereof) and there is a major difference between these ratings, the evaluators are not supposed to give the one variant preference over the other immediately, but consider the following issues:

- Could this outcome be related to the modifiers, i.e., do we have the wrong priorities?
- Was there a bias for or against an alternative, which led to a skewed outcome in the soft factors?
- Is there enough information the measurements could be based upon? An example would be the abilities of the teams. How much do the evaluators really know about them?
- If these factors are considered, one can judge that one alternative is really better than the other one.

The above list shows that even though an evaluation might not result in an immediate, obvious outcome, the scrutinizing of the results can benefit the whole project. This is similar to an ATAM (cf. section 3.4.2), which has the stated purpose of discovering risks, sensitivity points and tradeoffs. It gives the stakeholders a better understanding of the system and provides the architect with hints of possible complications and critical issues.

7.3 Aggregation of Results

During the last chapter, we introduced twelve questions and their corresponding metrics that are listed in the following.

- How does the architecture provide for ways to check against the interfaces?
- How well does the architecture ensure that for every aspect of the system, one team takes responsibility, and can be held accountable?
- How well does the architecture ensure that the implementation can be checked against the architectural specification
- How does the architecture specification minimize the level of skills and knowledge required for implementing the subsystems?

- How good do the interfaces match the team distribution?
- How many dependencies does the system have?
- How good is the relationship between flexibility in the work package breakdown and a loosely coupled system?
- How can one minimize the communication between the subsystem developers?
- Are architectural styles/patterns employed to solve standard problems?
- Is the architecture based on a reference architecture or a similar project?
- How good is the overall communicability of the architecture?
- Is the specification comprehensive enough, but not overly detailed?

The scorecard is not going to be used to give a single number for the whole system, but to create table containing the metrics and a set of attributes for every one of them, describing the impact and other issues related to it. The cells for these attributed are either filled with values of a scale (such as a percentage), or by a textual representation. Usually, the table will be created by the stakeholders involved in the specification of the architecture, similar to the personnel involved in an ATAM review.

Table 10 specifies the name of the metric in the first, and the value or outcome (in case the metric cannot be quantified) in the second column. The first attribute we assign to the metric is shown in the fourth column. It describes the risk associated with an unsatisfying or not accepted outcome (that fails the acceptance test by the customer). We consider the metric "How well does the architecture ensure the conformance of the implementation to the architectural specification?" here, which resulted in a value of 0.6 points out of 7.2 for the MSLite project. While we have the hindsight of knowing about the outcome of the project now, we still can extrapolate from the assumptions we derived from a risk analysis during the early stages of the development process. Back then, the lack of a set of tools and methods for the enforcement and the verification of the architecture would have been considered a large risk for the success of the project. We estimate that we would have estimated the risk of a failure related to this problem very high, resulting in an outcome that did not meet the acceptance criteria set by the central team.

If a substantial risk is for failure has been observed, the next column should give an indication whether a mitigation strategy exists and if so, what it would be. In the case of this metric, we would have pointed out that in our case, it would have been possible to change the scope of the project and even the expected outcome substantially, as the academic nature of the project allowed for these kinds of modifications. While not a very subtle or "real-life" approach, this would have led to a better compliance of the implementation to the specifications. Additionally, we studied the use of code reviews and similar schemes.

The cost for improvements column specifies what kind of effort – usually in person days – would be required to increase the score in the second column. In our case, a major improvement could have been achieved by assigning more people to outlining the specifications in detail, but those were not available. Estimating how the costs would have been calculated back then is not going to result in a sensible output now.

Metric	**Value/Outcome**	**Risk of unsatisfying/ not accepted outcome**	**Mitigation strategy**	**Cost for improvements**	**Subjective satisfaction with result**
How well does the architecture ensure the conformance of the implementation to the architectural specification?	0.6 points out of 7.2	High.	Change the scope and/or expected outcome. Use code reviews and walkthroughs.	Medium/High.	Low.
How does the architecture provide for ways to check against the interfaces?	Almost no measures.	High.	E. g. by specifying interfaces up front.	High.	Low.
How does the architecture specification minimize the level of skills and knowledge required for implementing the subsystems?	Medium – Domain knowledge required by four teams. GUI skills needed by three teams.	Low/Medium, complexity of the task not very high.	Reassign subsystems. Extract functionality such as Remoting.	Medium/High.	Medium/High.
Is the architecture based on a reference architecture or a similar project?	No.	Medium.	None.	N/A.	Medium.

Table 10. Overall Scorecard Examples

The last column is an indicator of the stakeholders' opinion of the situation. It reflects whether the stakeholders see room for improvement, and whether they think this improvement can be reached with an acceptable level of effort or whether the status quo should suffice for the success of the project. For the MSLite project, the satisfaction with this topic was low among the central team members.

The other rows of the table are filled with examples from other metrics to provide further insights into how to scorecard is created.

The result that was achieved using the table described above gives a result concerning the suitability of the architecture for a distributed development effort. The feasibility of the architecture itself – e.g. the criteria against which a co-located project would be measured – is not examined by this procedure and has to be evaluated using another set of criteria.

Chapter 8
Conclusion and Further Work

8.1 Summary

In this thesis, we described the architecture-related issues that arise in a distributed development project. By analyzing empirical data, we have shown that the architectural design decisions concerning distributed development have a large impact on the project success. We examined the context and the architecture of the MSLite project and depicted the problems and challenges we encountered. We then formulated basic hypotheses about the problems of developing software in such a distributed fashion, namely

- The match between the work package structure and the subsystem design has to be examined.
- Interfaces and dependencies of the system have to be related to the communication channels within the project
- The enforcement of architectural specifications and rules play a larger role in distributed development.
- The architecture influences the way communication between the remote teams and the central team is conducted.

Starting from these hypotheses, we established a number of goals that have to be fulfilled to increase the chances that the development project is going to have a successful outcome. We summarized these goals into three major problem fields:

- The problem of feedback.
- Subsystem decomposition and work distribution.
- Better communication of the architecture to the remote teams.

To be able to judge whether an architecture satisfies these goals, we created a criteria catalogue containing a number of metrics. To derive the questions and metrics from the goals, we employed the Goal Question Metric approach, and grouped them according to the problem fields. Finally, we provided a scorecard as a unified way to provide an overview about the suitability of an architecture for distributed development. The architect can use these metrics as tools to assess the quality of the architecture that is about to be implemented. Risks can be evaluated, and architectural variants can be compared to each other.

The feedback-related problems are measured by assessing how the architecture provides for ways to check progress and to enforce rules. To analyze the match between subsystems and teams distribution, metrics for the relationship between the interfaces and dependencies in a system on the one hand and the properties of the remote teams on the other hand are established. The last problem field contains measurements whether the communication is minimized and clearly structured and whether the architecture is easy to communicate in the first place.

The result of employing metrics is used to find mitigation strategies, i.e. making changes to the architecture to improve it. Therefore, the investigation is an ongoing process. By using such an iterative approach, it is possible to refine the architecture continuously, and to match the project plan and the architectural specification.

8.2 Transferability of Results

During the course of the MSLite project, we were able to gather a large body of empirical data. The hypotheses we formulated represent the conclusions we drew from it. It has to be noted, however, that while the amount of information that was available was large compared to many projects conducted in an academic setting, only one project was investigated. Therefore, the value of the conclusions is dependent on the quality of the project setup. One has to be careful to generalize upon these experiences. Further research has to be conducted, gaining data from projects varying with respect to the application domain, the number and location of the development sites, and the experience of the involved software engineers.

Another concern is the question of how well the experiences gathered from a project conducted by students can be transferred to a corporate environment. While it was tried to establish a "real-life" atmosphere, the psychological impact of knowing that the project's outcome is not going to influence one's career, bonus etc. should not be underestimated. Additionally, students typically have a very homogenous background concerning their experiences, particularly as only members of one team had already gathered exhaustive work experience outside of the university. In a company, there are usually people with different amounts of experience who can help each other out.

In general, however, the way the project was conducted could be compared to a typical undertaking as it is executed at companies such as SCR, which was confirmed by Siemens employees who had insights into the MSLite project as well as other ventures taking place at Siemens. Therefore, we gather that the general problem set is transferable, but further empirical studies have to be conducted to confirm this theory.

This thesis is focused on a single aspect of distributed development, the architecture of a software system. While in some parts, connections with other problem areas have been mentioned, the general approach has been to isolate the architectural issues. It is possible that researching concerns such as the development process, the management, and the communication structures could have a larger impact on the software architecture than it was assumed.

At Siemens Corporate Research, however, research is being conducted to address these issues as well. The development process is covered in detail in (Gersmann, 2005). An experimental process for distributed development called MD.Rad (Sangwan and Masticola, 2005) was employed partially, and a thesis about the management of distributed projects has been written (Reuvers, 2004). Some of this work has found its way into this thesis. The usage of tools and methods during the implementation phase and their relation to the architecture were investigated by us e.g. in section 6.3.3.

We consider it as an advantage, though, to have written exclusively about architectural aspects, as we were able to abstract the problems more easily. That allows us to focus on some core points; we did not have to consider too many variables. For a complete documentation of the distributed development process and where the architecture comes into play, more research that is ongoing has to be taken into account.

8.3 Further work

The complexity of the topic addressed in this thesis does not allow for an examination of every aspect of it. Therefore, there had to be a focus on some core topics. It would be an interesting project to combine the different papers and theses that have been written about the MSLite project, and extract the connections and dependencies between the diverse problem areas. Furthermore, this thesis did not discuss the following issues:

- **Management concerns**. When dealing with a large number of software engineers who are distributed across sites, the management of these people becomes a major concern. For the software architect, this leads to some complicated problems, such as scheduling meeting at the right intervals, creating milestones for architectural decisions and so on. While this is process-related to some degree, the architecture-specific part is a research subject on its own.
- **Internal politics**. In these kinds of projects, the number of different organizational units taking part is high. Additionally, the responsibilities can become somewhat confusing for everybody involved. A prime example is the fact that while the staff at one location has to report to the head architect, they usually still have a superior at their own site. While these issues should be made clarified in the beginning of a project, it often becomes apparent that these authority structures can become muddy very fast, particularly if the personnel are involved in more than one project.
- **The project schedule** and its influences on the architecture were not described in detail; neither did we establish a metric to measure the quality of the schedule. We feel, however, that this concern is more closely related to the whole process, of which the architecture design phase is part of, instead of being closely tied to the architecture itself. Work could be done where techniques such as *timeboxing* are investigated.
- **Requirements engineering** for distributed software development projects should be examined comprehensively. To our knowledge, no major work has been done pointing in this direction. Additionally, the transition from the requirements to architecture can be explored. It would be interesting to observe how a distributed environment influences the reasoning behind building an

architecture out of certain set of requirements, particularly because the nature of the distribution can be seen as a requirement or quality attribute itself.

- **The architectural design process** was not discussed in detail. Based on this thesis, one could define an approach that uses the metrics to establish a method for creating an architecture. Instead of considering distributability as a quality attribute, it is possible to incorporate it into the design process itself.

Bibliography

Antoniol, G., Fiutem, R. and Cristoforetti, L. (1998) Design pattern recovery in object-oriented software. In *Program Comprehension, 1998. IWPC '98. Proceedings., 6th International Workshop on*, pp. 153-160.

Avritzer, A. and Weyuker, E. (1998) Investigating Metrics for Architectural Assessment. In *METRICS '98: Proceedings of the 5th International Symposium on Software Metrics*.

Basili, V. (1992) *Software modeling and measurement: the Goal/Question/Metric paradigm.*, Univ. of Maryland, Institute for Advanced Computer Studies, College Park, MD, USA.

Bass, L., Clements, P. and Kazman, R. (2003) *Software Architecture in Practice, Second Edition*. Addison-Wesley Professional.

Bauer, A. and Pizka, M. (2003) The contribution of free software to software evolution. In *Proceedings of the Sixth International Workshop on Principles of Software Evolution*, pp. 179.

Best Current Practices: Software Architecture Validation. (1991), AT&T Bell Laboratories.

Boehm, B. (2002) Software engineering economics. Software pioneers: contributions to software engineering, 641-686.

Booch, G. (1991) *Object oriented design with applications*. Benjamin/Cummings Pub. Co., Redwood City, Calif., Benjamin/Cummings series in Ada and software engineering., pp. xix, 580 p.

Bratthall, L., Geest, R., Hofmann, H., Jellum, E., Korendo, Z., Martinez, R., Orkisz, M., Zeidler, C. and Andersson, J. (2002) Integrating hundred's of products through one architecture: the industrial IT architecture. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pp. 614.

Brooks, F. (1975) *The mythical man-month and other essays on software engineering*. Dept. of Computer Science, University of North Carolina at Chapel Hill.

Bruegge, B. and Dutoit, A. (2003) *Object-Oriented Software Engineering: Using UML, Patterns and Java, Second Edition*. Prentice Hall.

BSD License (2005), BSD License, Open Source Initiative, available at http://www.opensource.org/licenses/bsd-license.php, accessed on November 6, 2005.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., Sommerlad, P. and Stal, M. (1996) *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons.

Chidamber, S. and Kemerer, C. (1994) A metrics suite for object oriented design. IEEE Transactions on Software Engineering 20, 493.

Clements, P., Bachmann, F., Bass, L., Garlan, D., Ivers, J., Little, R., Nord, R. and Stafford, J. (2002a) *Documenting Software Architectures: Views and Beyond*. Addison-Wesley Professional.

Clements, P., Bass, L., Kazman, R. and Abowd, G. (1995) Predicting software quality by architecture-level evaluation. In *Proceedings of the Fifth International Conference on Software Quality*.

Clements, P., Kazman, R. and Klein, M. (2002b) *Evaluating Software Architectures: Methods and Case Studies*. Addison-Wesley Professional.

Clements, P. and Northrop, L.M. (1996) Software Architecture: An Executive Overview. Software Engineering Institute, Carnegie Mellon University, Pittsburgh.

Conway, M.E. (1968) How Do Committees Invent? Datamation 14, 28-31.

CORBA (2005), CORBA 3.0, Object Management Group, Inc., available at http://www.omg.org/technology/documents/formal/corba_2.htm, accessed on November 6, 2005.

Curtis, B., Krasner, H. and Iscoe, N. (1988) A field study of the software design process for large systems. Communications of the ACM 31, 1287.

de Bruijn, H. and van Vliet, J. (2001) Scenario-Based Generation and Evaluation of Software Architectures. In *GCSE '01: Proceedings of the Third International Conference on Generative and Component-Based Software Engineering*, Springer-Verlag, pp. 139.

DeMarco, T. (1986) *Controlling Software Projects: Management, Measurement, and Estimates*. Prentice Hall PTR.

Duenas, J.C., de Oliveira, W.L. and de la Puente, J.A. (1998) A Software Architecture Evaluation Model. In *Proceedings of the Second International ESPRIT ARES Workshop on Development and Evolution of Software Architectures for Product Families*, Springer-Verlag, pp. 148-157.

Espinosa, J.A., Cummings, J.N., Pearce, B.M. and Wilson, J.M. (2002) Research on teams with multiple boundaries. In *Proceedings of the 35th Annual Hawaii International Conference on System Sciences*, pp. 3438.

Fowler, M. (2002) *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional.

Garlan, D., Allen, R. and Ockerbloom, J. (1995) Architectural mismatch or why it's hard to build systems out of existing parts. In *ICSE '95: Proceedings of the 17th international conference on Software engineering*, pp. 185.

Garlan, D. and Perry, D. (1994) Software architecture: practice, potential, and pitfalls. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pp. 364.

Garlan, D. and Shaw, M. (1994) An Introduction to Software Architecture. Carnegie Mellon University, Pittsburgh.

Gersmann, S. (2005) *Not named.* Diploma Thesis, Technische Universität München, Munich, Germany.

Gibson, V. and Senn, J. (1989) System structure and software maintenance performance. Communication of the ACM 32, 358.

GPL (2005), GNU General Public License, Free Software Foundation, available at http://www.fsf.org/licensing/licenses/gpl.html, accessed on November 6, 2005.

Goldenson, D.R., Gopal, A. and Mukhopadhyay, T. (1999) Determinants of Success in Software Measurement Programs: Initial Results. In *Proceedings of IEEE Metrics*,Boca Raton, FL, USA.

Grady, R.B. and Caswell, D.L. (1987) *Software metrics : establishing a company-wide program*. Prentice-Hall, Englewood Cliffs, N.J., pp. xv, 288 p.

Grudin, J. (1994) Computer-supported cooperative work: history and focus. Computer 27, 19-26.

Gustafsson, J., Paakki, J., Nenonen, L. and Verkamo, A. (2002) Architecture-centric software evolution by software metrics and design patterns. In *Software Maintenance and Reengineering, 2002. Proceedings. Sixth European Conference on*, pp. 115.

Halstead, M. (1977) *Elements of software science (Operating and programming systems series)*. Elsevier.

Herbsleb, J. and Grinter, R. (1999a) Architectures, Coordination, and Distance: Conway's Law and Beyond. IEEE Softw. 16, 70.

Herbsleb, J. and Grinter, R. (1999b) Splitting the organization and integrating the code: Conway's law revisited. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pp. 95.

Herbsleb, J.D. and Moitra, D. (2001) Global software development. IEEE Software 18, 20.

Hilliard, R.F., Kurland, M.J. and Litvintchouk, S.D. (1997) MITRE’s Architecture Quality Assessment. In *Software Engineering & Economics Conference*,McLean, VA, USA.

Hofmeister, C., Nord, R. and Soni, D. (1999) *Applied Software Architecture*. Addison-Wesley Professional.

How Define (2005), How Do You Define Software Architecture?, Software Engineering Institute, Carnegie Mellon University, available at http://www.sei.cmu.edu/architecture/definitions.html, accessed on November 6, 2005.

Hsi, I. (2004) *Analyzing the Conceptual Coherence of Computing Applications Through Ontological Excavation*. Ph.D. Thesis, Georgia Institute of Technology, Atlanta, GA, USA.

Hunt, A. and Thomas, D. (1999) *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley Professional.

IEEE Recommended practice for architectural description of software-intensive systems. (2000) IEEE Std 1471-2000, i-23.

ISO (1991) *ISO. Information Technology – Software Product evaluation – Quality Characteristics and Guidelines for their Use. Int. Standard ISO/IEC 9126*. ISO.

Jazayeri, M., Ran, A.C.M. and Linden, F.v.d. (2000) *Software architecture for product families : principles and practice*. Addison-Wesley, Boston, pp. xxvi, 257 p.

Kan, S. (2002) *Metrics and Models in Software Quality Engineering (2nd Edition)*. Addison-Wesley Professional.

Karlsson, E.-A., Andersson, L.-G. and Leion, P. (2000) Daily build and feature development in large distributed projects. In *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, pp. 658.

Kazman, R., Bass, L., Webb, M. and Abowd, G. (1994) SAAM: a method for analyzing the properties of software architectures. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, IEEE, pp. 81-90.

Kiczales, G. (1996) Aspect-oriented programming. ACM Computing Surveys 28.

Kramer, C. and Prechelt, L. (1996) Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software. In *WCRE '96: Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE '96)*, IEEE.

Kruchten, P. (1995) The 4+1 View Model of Architecture. IEEE Software 12, 50.

Kurpjuweit, S. (2001) Software Architecture Analysis and Evaluation (Lecture Notes). In.

Lindvall, M., Tesoriero, R. and Costa, P. (2002) Avoiding architectural degeneration: an evaluation process for software architecture. In *Proceedings of the Eighth IEEE Symposium on Software Metrics*, pp. 86.

Lorenz, M. and Kidd, J. (1994) *Object-Oriented Software Metrics*. Prentice Hall.

Markus, L. and Connolly, T. (1990) Why CSCW applications fail: problems in the adoption of interdependent work tools. In *CSCW '90: Proceedings of the 1990 ACM conference on Computer-supported cooperative work*, ACM, pp. 371-380.

Martin, R. (1999) Iterative and Incremental Development (IID). C++ Report.

McAndrews, D.R. (1993) Establishing a Software Measurement Process. Carnegie Mellon University, Software Engineering Institute.

McCabe, T. (1976) A complexity measure. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*.

Medvidovic, N. and Taylor, R. (2000) A Classification and Comparison Framework for Software Architecture Description Languages. IEEE Transactions on Software Engineering 26, 70-93.

Meyer, B. (1992) Applying 'design by contract'. IEEE Computer 25, 40-51.

O'Hara-Devereaux, M. and Johansen, R. (1994) *GlobalWork: Bridging Distance, Culture and Time (The Jossey-Bass Management Series)*. Jossey-Bass.

Palmer, S. and Felsing, J. (2002) *A Practical Guide to Feature-Driven Development (The Coad Series)*. Prentice Hall PTR.

Papazoglou, M.P. (2003) Service-oriented computing: concepts, characteristics and directions. In *Web Information Systems Engineering, 2003. WISE 2003. Proceedings of the Fourth International Conference on*, pp. 12.

Parnas, D. (1972) On the Criteria To Be Used in Decomposing Systems into Modules. Communication of the ACM 15, 1058.

Parnas, D. and Weiss, D. (1985) Active design reviews: principles and practices. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, IEEE, pp. 132-136.

Paulish, D. (2001) *Architecture-Centric Software Project Management: A Practical Guide*. Addison-Wesley Professional.

Perry, D., Staudenmayer, N. and Votta, L. (1994) People, Organizations, and Process Improvement. IEEE Softw. 11, 45.

Perry, D. and Wolf, A. (1992) Foundations for the study of software architecture. ACM SIGSOFT Software Engineering Notes 17, 52.

Raymond, E. (2001) *The Cathedral & the Bazaar*. O'Reilly.

Reuvers, S. (2004) *Management of Distributed Projects*. Final Thesis, Delft University of Technology, Delft, The Netherlands.

Sangwan, R.S. and Masticola, S.P. (2005) Model-Driven Rapid Application Development: A Framework for Agile Development in Outsourced Environments. Siemens Corporate Research.

Shaw, M. and Garlan, D. (1996) *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall.

Weinberg, G. (1998) *The Psychology of Computer Programming: Silver Anniversary Edition*. Dorset House Publishing Company, Incorporated.

Weyuker, E.J. (1988) Evaluating software complexity measures. IEEE Transactions on Software Engineering 14.

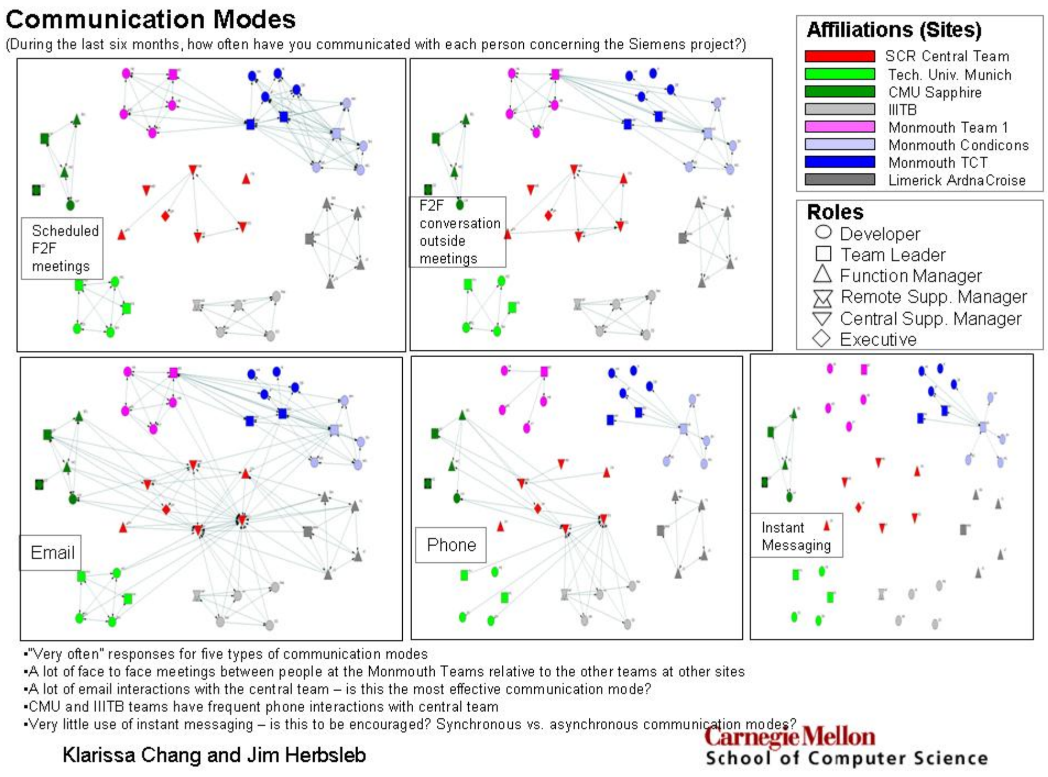
Yourdon, E. and Constantine, L.L. (1975) *Structured Design*. Yourdon.

List of Abbreviations

ADL	Architecture Description Language
AOP	Aspect Oriented Programming
ATAM	Architecture Tradeoff Analysis Method
BACNet	Data communication protocol for building automation and control networks
C&C	Component & Connector viewtype
CMU	Carnegie Mellon University, Pittsburgh, PA, USA
CSCW	Computer Supported Collaborative Work
DbC	Design by Contract
GQM	Goal Question Metric approach
GSP	Global Studio Project
GUI	Graphical User Interface
FSS	Field System Simulator
IDL	Interface Definition Language
IIITB	International Institute of Information Technology, Bangalore, India
L&R	Logic & Reaction subsystem
LoC	Lines of Code
MSLite	Management Station Lite
MU1	Monmouth University, NJ, USA – Team #1
MU2	Monmouth University, NJ, USA – Team #2
QA	Quality Attribute
SCR	Siemens Corporate Research
SDD	System Design Document
SOM	System Object Model subsystem
SOW	Statement of Work
SRS	Software Requirements Specification
TUM	Technische Universität München, Germany
UL	University of Limerick, Ireland

Appendix A
Interaction Diagrams

This section contains material that Klarissa Chang and Jim Herbsleb from Carnegie Mellon University gathered by evaluating interviews with the project participants. It provides insights into the communication structures between the different teams and individual team members.

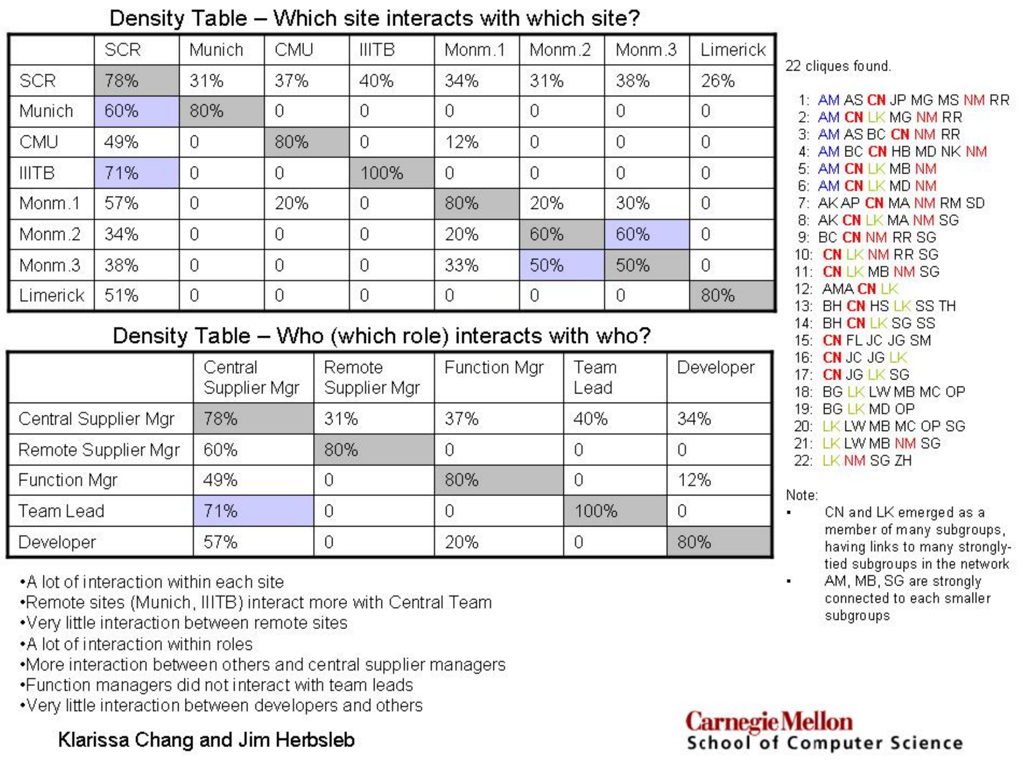

Communication Modes

(During the last six months, how often have you communicated with each person concerning the Siemens project?)

•"Very often" responses for five types of communication modes
•A lot of face to face meetings between people at the Monmouth Teams relative to the other teams at other sites
•A lot of email interactions with the central team – is this the most effective communication mode?
•CMU and IIITB teams have frequent phone interactions with central team
•Very little use of instant messaging – is this to be encouraged? Synchronous vs. asynchronous communication modes?

Klarissa Chang and Jim Herbsleb

Figure 12. Communication modes



Density Table – Which site interacts with which site?

	SCR	Munich	CMU	IIITB	Monm.1	Monm.2	Monm.3	Limerick
SCR	78%	31%	37%	40%	34%	31%	38%	26%
Munich	60%	80%	0	0	0	0	0	0
CMU	49%	0	80%	0	12%	0	0	0
IIITB	71%	0	0	100%	0	0	0	0
Monm.1	57%	0	20%	0	80%	20%	30%	0
Monm.2	34%	0	0	0	20%	60%	60%	0
Monm.3	38%	0	0	0	33%	50%	50%	0
Limerick	51%	0	0	0	0	0	0	80%

Density Table – Who (which role) interacts with who?

	Central Supplier Mgr	Remote Supplier Mgr	Function Mgr	Team Lead	Developer
Central Supplier Mgr	78%	31%	37%	40%	34%
Remote Supplier Mgr	60%	80%	0	0	0
Function Mgr	49%	0	80%	0	12%
Team Lead	71%	0	0	100%	0
Developer	57%	0	20%	0	80%

- A lot of interaction within each site
- Remote sites (Munich, IIITB) interact more with Central Team
- Very little interaction between remote sites
- A lot of interaction within roles
- More interaction between others and central supplier managers
- Function managers did not interact with team leads
- Very little interaction between developers and others

22 cliques found.

1: AM AS **CN** JP MG MS NM RR
2: AM **CN** LK MG NM RR
3: AM AS BC **CN** NM RR
4: AM BC **CN** HB MD NK NM
5: AM **CN** LK MB NM
6: AM **CN** LK MD NM
7: AK AP **CN** MA NM RM SD
8: AK **CN** LK MA NM SG
9: BC **CN** NM RR SG
10: **CN** LK NM RR SG
11: **CN** LK MB NM SG
12: AMA **CN** LK
13: BH **CN** HS LK SS TH
14: BH **CN** LK SG SS
15: **CN** FL JC JG SM
16: **CN** JC JG LK
17: **CN** JG LK SG
18: BG LK LW MB MC OP
19: BG LK MD OP
20: LK LW MB MC OP SG
21: LK LW MB NM SG
22: LK NM SG ZH

Note:
- CN and LK emerged as a member of many subgroups, having links to many strongly-tied subgroups in the network
- AM, MB, SG are strongly connected to each smaller subgroups

Klarissa Chang and Jim Herbsleb

Carnegie Mellon School of Computer Science

Figure 13. Interactions between sites

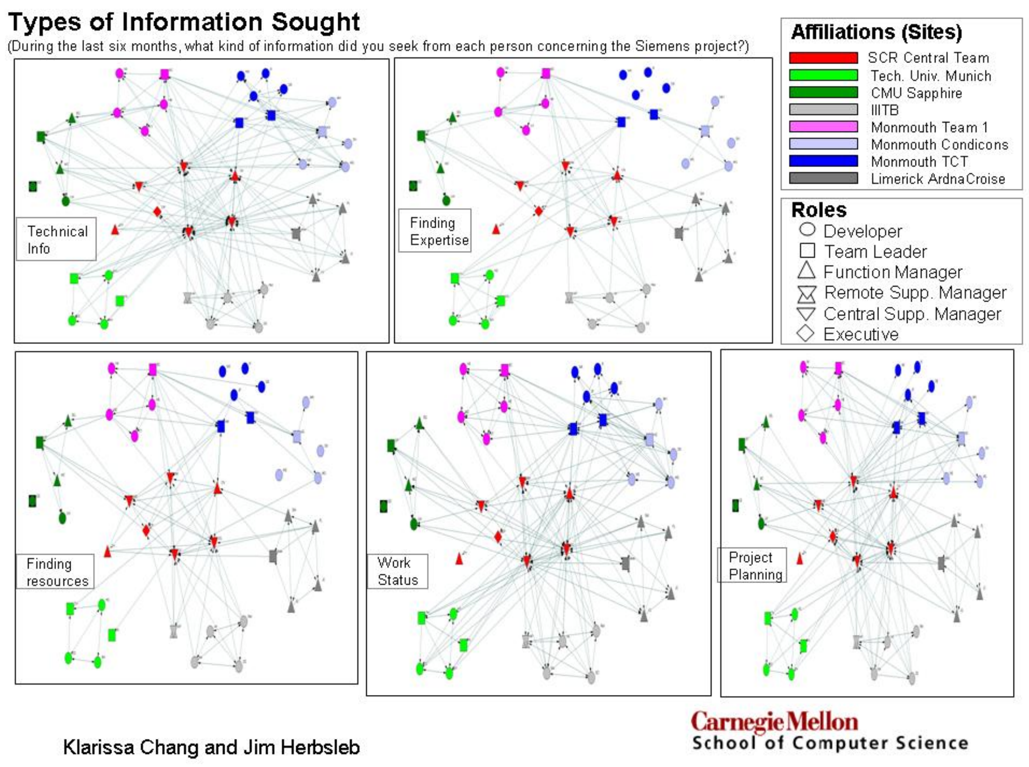
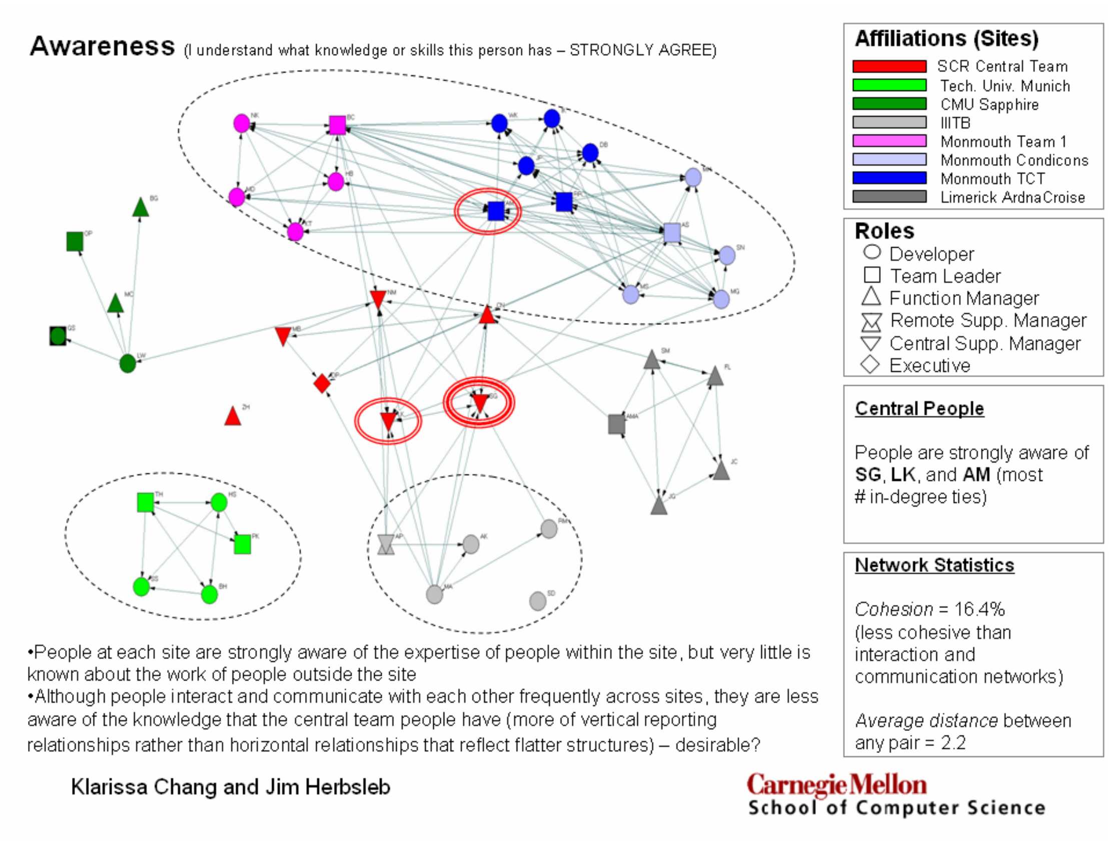


Figure 14. Types of Information Sought



Awareness (I understand what knowledge or skills this person has – STRONGLY AGREE)

•People at each site are strongly aware of the expertise of people within the site, but very little is known about the work of people outside the site
•Although people interact and communicate with each other frequently across sites, they are less aware of the knowledge that the central team people have (more of vertical reporting relationships rather than horizontal relationships that reflect flatter structures) – desirable?

Central People

People are strongly aware of **SG**, **LK**, and **AM** (most # in-degree ties)

Network Statistics

Cohesion = 16.4% (less cohesive than interaction and communication networks)

Average distance between any pair = 2.2

Klarissa Chang and Jim Herbsleb

Carnegie Mellon
School of Computer Science

Figure 15. Awareness of projects participants in the MSLite project

Appendix B
Research Goals

1) What is the optimal frequency and types of communications and interactions between the distributed component teams and the central team?
2) How do Siemens tools (Design Advisor and Code Inspector) and vendor tools (e.g., Rational Rose, DOORS) affect the global development process, the productivity of the development teams, and the quality of the resulting product?
3) What MD.RAD reference process tasks are appropriate for distribution, and the central team in one location must do what tasks? How useful is MD.RAD as a reference process for global development?
4) What are efficient ways to validate components?
5) What is an appropriate level of specificity for acceptance and regression tests?
6) What types of integration strategies work well for integrating distributed components?
7) What is the appropriate level of detail for specifying the requirements for the component development teams?
8) How do you specify and communicate the “non-functional” or “quality” requirements to the component development teams?
9) What amount of subject matter or domain expertise is necessary for the distributed teams?
10) What types of quality assurance, review, or oversight methods are optimal to ensure technical and management coordination of the distributed teams (e.g., avoiding architectural drift)?
11) How agile could the distributed teams’ processes be considering that they must work within a more formal, constrained global development process?