World Academy of Science, Engineering and Technology
International Journal of Economics and Management Engineering
Vol:1, No:9, 2007

Core Issues Affecting Software Architecture in Enterprise Projects

Halûk Gümüşkaya

***Abstract*—**In this paper we analyze the core issues affecting software architecture in enterprise projects where a large number of people at different backgrounds are involved and complex business, management and technical problems exist. We first give general features of typical enterprise projects and then present foundations of software architectures. The detailed analysis of core issues affecting software architecture in software development phases is given. We focus on three main areas in each development phase: people, process, and management related issues, structural (product) issues, and technology related issues. After we point out core issues and problems in these main areas, we give recommendations for designing good architecture. We observed these core issues and the importance of following the best software development practices and also developed some novel practices in many big enterprise commercial and military projects in about 10 years of experience.

***Keywords*—**Software architecture, enterprise projects.

I. Introduction

One of the major issues in software systems development today is quality. A quality attribute is a nonfunctional characteristic of a component or a system. ISO/IEC 9126-1 [1] defines a software quality model. According to this definition, there are six categories of characteristics (functionality, reliability, usability, efficiency, maintainability, and portability), which are divided into subcharacteristics. The idea of predicting the quality of a software product from a higher-level design description is not a new one. In 1972, Parnas [2] described the use of modularization and information hiding as a means of high level system decomposition to improve flexibility and comprehensibility. In 1974, Stevens et al. [3] introduced the notions of module cohesion and coupling to evaluate alternatives for program decomposition. A software module is stable if cohesion (intra-module communication) is strong and coupling (inter-module interaction) is low. Good software architecture tries to maximize cohesion and minimize coupling.

One of the major design tasks in building enterprise applications is to design good software architecture. During recent years, the notion of software architecture has emerged as the appropriate level for dealing with software quality. The software architecture of a system is defined as "the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them" [4]. This definition focuses only on the internal aspects of a system and most of the software analysis methods and tools are based on it. Another definition establishes software architecture as "the structure of components in a program or system, their interrelationships, and the principles and design guidelines that control the design and evolution in time". This process-centered definition takes into account the presence of principles and guidelines in the architecture description. We take this second definition as a our base definition and add other factors to this in section 3 and find a more comprehensive definition for software architecture especially in enterprise projects.

Although project management techniques, software development methodologies, design patterns, development, testing and architectural modeling techniques and tools have developed in the last decade; many software projects still fail and the percentage of successful projects completed on-time and on-budget is still very low. The Standish Group's "Chaos Report" in 1994 [5] reported that only 16.2% of software projects were completed on-time and on-budget. In 2004, 29% of projects completed on-time and on-budget, with required features and functions. Although the improvement is significant, it is dismal when compared with traditional engineering disciplines, such as architecture or electrical engineering. In the literature there are many excellent resources, surveys, and research papers showing the critical success and failure factors of software projects [6]–[11]. McConnell in his book [6] lists 36 classical software mistakes and divides these into four groups: People, process, product, and technology related mistakes. These and other classical mistakes in software development have small and large affects on software architecture. Bad software architecture is one of the reasons for software failures.

In this paper we analyze core issues affecting software architecture in enterprise projects where a large number of people at different backgrounds are involved and complex business, management and technical problems exist. We look at all social, organizational, managerial, and business implications and core aspects of development activities affecting software architecture.

This paper is structured as follows. In Section 2, we first give general features of typical enterprise projects. In Section 3, we present foundations of architectures. In Section 4, the

Manuscript received October 14, 2005.

Halûk Gümüşkaya is with the Department of Computer Engineering at Fatih University, Istanbul, Turkey. (phone: +90-212-8890810; fax: +90-212-8890906; e-mail: haluk@fatih.edu.tr).

detailed analysis of core issues affecting software architecture in software development phases is given. We focus on three main areas in each development phase: people, process, and management related issues, structural (product) issues, and technology related issues. After we point out core issues and problems in these main areas, we give recommendations for designing good architecture.

II. General Features of Typical Enterprise Projects

The typical enterprise applications are internet and intranet sites, enterprise resource planning applications, inventory management systems, payroll applications, management information systems. Many enterprise projects use different agile development models (extreme programming, feature driven development, and so on) and evolutionary prototyping. In planning and management, project planning is done incrementally, test and quality assurance planning are performed as needed, mostly they have informal change control. Many typical projects have informal requirements specification, and design and coding are combined. In construction, many project have individual coding, some have pair programming. Most of them have informal check-in procedure or no check-in procedure. In testing and quality assurance, developers test their own code, some projects use test-first development. Generally there is little or no testing by a separate test group. They have informal deployment procedure. Enterprise projects tend to benefit from highly iterative approaches, in which planning, requirements, and architecture are interleaved with construction, system testing and quality assurance activities. These general features have very important affects on the definition, design, construction, testing, and deployment of an enterprise architecture.

III. Foundations of Software Architectures and General Issues

Software architecture is more than just a technical blueprint of a complex enterprise project. In addition to its technical functions, software architecture has important social, organizational, managerial, and business implications [4]. We can't simply regard it as a result of technical work and ignore other implications. There are many people affecting the construction of a software system. Customer, end users, analysts, product managers, architects, developers, project manager, business architects, sales people are few examples. These are called stakeholders. Stakeholders have different concerns and goals, some of which may be contradictory. Architectures are also influenced by the structure and the nature of the development organization. There are three classes of influence that come from the developing organization: immediate business, long-term business, and organizational structure. The other important influence is the background, experience and the education of the architects. The technical environment will also influence the architecture.

Influences on architecture come from a wide variety of sources, and some of are only implied, while others are explicitly in conflict. Architects must understand the business and technical requirements, effectively communicate with all stakeholders, and resolve conflicting problems. For an effective architect, communication, diplomacy and negotiation skills are very important.

IV. Issues in Software Development Phases

Software projects are divided into 3 conceptual stages, as shown in Fig 1 [12]. At the beginning of the project, the focus is on "discovery"— especially discovery of the user's main requirements. This first phase is characterized by some technical investigation work such as interviewing users, building user interface prototypes and developing and working on a requirements document. The project manager prepares a management document like a software project management plan [13] in this phase. In the middle of the project, the focus shifts to invention. At the macro level, architects invent a software architecture and design. At the micro level, each package or class may require small inventions. During the final part of the project, the focus shifts again, this time to implementation. As Fig 1 illustrates, these three phases occur to some degree throughout a software project.

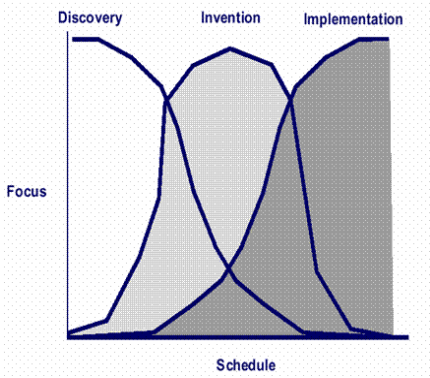

Fig. 1 Conceptual phases of a software project

A simplified diagram of software development phases of a typical process based software project is shown in Fig 2. The project is first carefully defined and designed and then functionality is delivered in successive iterations.

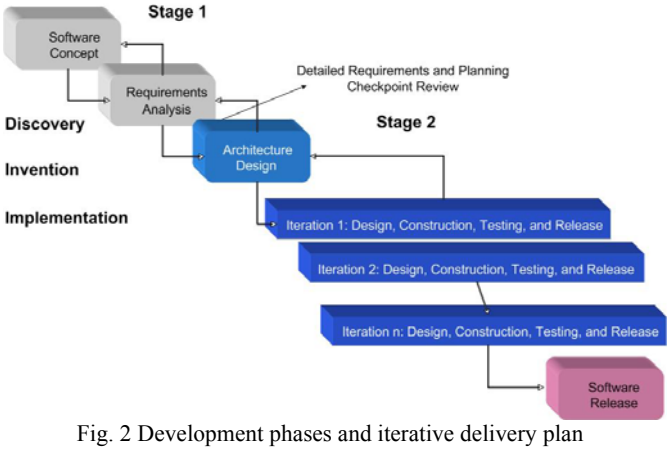

Fig. 2 Development phases and iterative delivery plan

The detailed phases of a well defined software development in many current popular development processes are shown in Fig 3. Many classical software engineering sources and processes divide requirements and architecture design phases into two sub phases [7], [14], [15] and there are totally eight phases [14] in a software lifecycle.

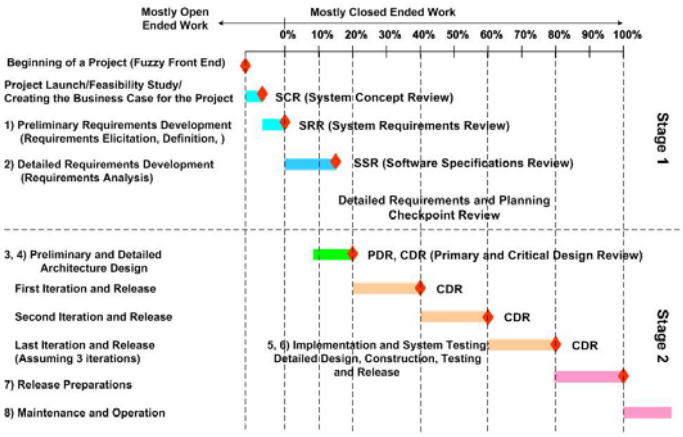

Fig. 3 Detailed software development phases and iterative delivery plan

In the following subsections, the detailed analysis of the core issues affecting software architecture in these software development phases is given. We focus on three main areas in each development phase: people, process, and management related issues; structural (product) issues; and technology related issues.

Many projects have generally a fuzzy front end which is the time before project officially starts—in approval, budgeting, and feasibility-investigation phases. Projects often spend weeks or months in the fuzzy front end, and then try to finish rapidly. People must be put in charge as early as possible, goals and objectives for front-end activities must be set and actively managed by a risk management plan.

A. Creating the Business Case for the Product

This is the first important step in creating and constraining any future requirements. The questions such as cost, business goals, target market, the product's targeted time to market, should be defined at this phase. Business people, product managers, system architects must be in charge in defining business case for the product.

B. Requirements Elicitation and Analysis Phases

The requirements process consists of two activities: Requirements elicitation and requirements analysis [15]. Requirements elicitation is the definition of the system in terms understood by the customer ("Problem Description"). Requirements analysis is the technical specification of the system in terms understood by the developer ("Problem Specification").

In the requirements elicitation and analysis phases, system analysts, business and product managers, system architects start to define an enterprise architecture in an enterprise project. There are many different approaches to describing the elements of an enterprise architecture. The most popular approach to describing an enterprise architecture that has grown in popularity in the past few years is based on a framework developed by John Zachman. Zachman originally proposed his framework in 1987 by his paper [16]. Zachman puts an emphasis on describing what exists on each level of an enterprise. In the simplest version of the framework, Zachman proposes to describe within each level: what things are involved (data); how things are done (function), where things are done (network). This framework is used by IT managers, developers, and business managers to define an enterprise architecture for a large organization.

The classical mistakes which many projects do at this requirements elicitation and analysis phase: insufficient senior staff on the requirements team, developing incomplete, unstable written requirements and design documents, insufficient user input, and setting an optimistic (or frequently changing) schedule. Incomplete and changing requirements are the first cause of software project failures [5]. Comprehensive, 100% stable requirements are usually not possible, but most requirements changes arise from requirements that were incompletely defined in the first place, not "changing markets" or other similar reasons. Involving users throughout the project is a critical software project skill.

Some of the best practices and techniques for defining stable, complete, written requirements: requirements workshop, user interface prototyping, user interviews, use cases, preparing user manual as specification at the beginning of the project, usability studies, incremental delivery, requirements reviews/inspections [7].

We believe the importance of breaking software project funding into two major stages as shown in Fig 2 and Fig 3. The project manager first requests funding for the exploratory phase during which the first 10 to 20 percent of the project is completed. This gives the organization an opportunity to look at canceling a project as a positive decision. Deferring the bulk of the funding request until after the project is 10 to 20 percent complete provides for much more reliable and realistic funding requests for the bulk of the project. Requiring a project manager to complete 10 to 20 percent of a project before requesting funding for the rest of it forces the manager (and the team) to focus on the upstream activities that are critical to the project's success [7].

At the end of this phase, the project team holds a Requirements and Planning Checkpoint Review. In conjunction with that review, senior management or the customer makes a go/no go decision, and then the project manager requests funding for the remainder of the project. The following items can be checked in this review: Name of project's key decision maker(s), team, roles, vision statement, business case for the software, preliminary effort and schedule goals and estimates, top 10 risks list, user interface style guide, user interface prototype, user manual, requirements specification, software quality assurance plan.

Software industry data from the 1970s to the present day

clearly indicates that projects will run best if appropriate preparation activities are done before construction begins in earnest. If software quality is emphasized at the beginning of the project, the development team plans for, requires, and designs a high-quality architecture, and at the end a product. In general, the principle is to find an error as close as possible to the time at which it was introduced. The longer the defect stays in the software development phases, the more damage it causes further down the phase. Since requirements are done first, requirements defects have the potential to be in the system longer and to be more expensive. Defects inserted into the software upstream also tend to have broader effects than those inserted further downstream. That also makes early defects more expensive. Fig 4 shows the relative expense of fixing defects depending on when they're introduced and when they're found [17].

	Time Detected				
Time Introduced	Re-quire-ments	Archi-tecture	Con-struc-tion	System Test	Post-Re-lease
Requirements	1	3	5-10	10	10-100
Architecture	—	1	10	15	25-100
Construction	—	—	1	10	10-25

Fig. 4 Average cost of fixing defects based on when they're introduced and detected

C. Architecture Design Phase

The software architecture design process generally consists of two activities: High level design and low level design. High level design is also known as "preliminary design" or "system design", and low level design is known as "detailed design" or "object design" [14], [15].

The typical system design ("high level" or "top level" design) activities are as follows [15]: Determining design goals, subsystem decomposition, handling concurrency, hardware/software mapping, persistent data management, global resource handling and access control, software control, boundary conditions.

These are the typical object design ("low level design") activities [15]:

1. Reuse: Identification of existing solutions, use of inheritance, off-the-shelf components and additional solution objects, and design patterns [18]–[20]
2. Interface specification (describe precisely each class interface)
3. Object model restructuring (transform the object design model to improve its understandability and extensibility)
4. Object model optimization (transform the object design model to address performance criteria such as response time or memory utilization).

The quality of the architecture determines the conceptual integrity of the system. That in turn determines the ultimate quality of the system. A well thought-out architecture provides the structure needed to maintain a system's conceptual integrity from the top levels down the bottom. It provides guidance to programmers—at a level of detail appropriate to the skills of the programmers and to the job at hand. It partitions the work so that multiple developers or multiple development teams can work independently. Good architecture makes construction easy. Bad architecture makes construction almost impossible.

The architecture should define the major building blocks in a program. Depending on the size of the program, each building block might be a single class, or it might be a subsystem consisting of many classes. The architecture doesn't need to specify every class in the system; aim for the 80/20 rule: specify the 20 percent of the classes that make up 80 percent of the systems' behavior [21], [22].

We have observed the importance of that the architecture should be the product of a single or a small group of architects with an identified leader in a large enterprise project [4]. In this our project experience we saw that the company manager tried to please most of the developers by letting them to be in the architectural meetings. In these meetings there were a lot of useless technical discussions and long wasted hours.

Creating, Buying or Selecting the Components and the Software Platform of the Architecture: The most radical solution to building software is not to build it at all—to buy it instead. You can buy GUI controls, database managers, image processors, graphics and charting components, Internet communications components, security and encryption components, spreadsheet tools, text processing tools—the list is nearly endless. One of the greatest advantages of programming in modern GUI environments is the amount of functionality you get automatically: graphics classes, dialog box managers, keyboard and mouse handlers, code that works automatically with any printer or monitor, and so on. If the architecture isn't using off-the-shelf components, it should explain the ways in which it expects custom-built components to surpass ready-made libraries and components.

Selecting a reference platform, say the J2EE or .NET, as the starting point for a product or product line has strategic implications. The selection of one community over another has major cost implications and future development investments.

If the plan calls for using pre-existing software, the architecture should explain how the reused software will be made to conform to the other architectural goals—if it will be made to conform. The architecture should clearly describe a strategy for handling changes. The architecture should show that possible enhancements have been considered and that the enhancements most likely are also the easiest to implement.

Architecture design is sloppy because it's hard to know when your design is "good enough." How much detail is enough? How much design should be done with a formal design notation, and how much should be left to be done at the keyboard? When are you done? Since design is open-ended, the most common answer to that question is "When you're out of time."

Communicating the Architecture: The architecture should be well documented with static and dynamic views,

using an agreed-on notation like UML (Unified Modeling Language) and design patterns that all stakeholders can understand. It should be reviewed by the project's stakeholders [23].

Analyzing and Evaluating the Architecture: The architecture should be analyzed for applicable quantitative measures and formally evaluated for quality attributes [24].

D. Iterative Implementation Phase

The architecture should lend itself to incremental implementation via the creation of a "skeletal" system in which the communication paths are exercised but which at first has minimal functionality.

Ensuring Conformance to the Architecture: The implementation may or may not conform to the desired architectural design. The purpose is to show in numbers how much the implemented system is worse than a desired architecture or another dependency-minimizing architecture. Refactoring techniques [25] are applied to improve the existing code after the conformance analysis and testing activities.

E. Testing Phase

The well architected system can be used to "grow" the system incrementally, easing the integration and testing efforts. There are generally four different testing activities in an enterprise project [15]: Unit testing, integration testing, system testing, and acceptance testing. Unit testing is carried out by developers, and it confirms that subsystems are correctly coded and carry out the intended functionality. Groups of subsystems (collection of classes) and eventually the entire system are tested by developers in integration testing. The main goal is to test the interfaces among the subsystems. In system testing the entire system is tested by developers to determine if the system meets the global functional and nonfunctional requirements. The client carries out the acceptance tests to evaluate the system delivered by developers. The main goal is to demonstrate that the system meets customer requirements and is ready to use. The different components and views of the architecture are tested during these different activities. Testing process and testers have also a very important affect on software architecture.

V. Conclusion

The author of this paper has observed these core issues affecting software architecture and the importance of following the best software development practices and also developed some novel practices in many big enterprise commercial and military projects in his about 10 years of project experience. He worked 5 years as Senior and Chief Researcher (in his last year) at the Scientific and Technical Research Council of Turkey-National Research Institute of Electronics and Cryptology (TÜBİTAK–UEAKE) between 5/1997 and 7/2002. During this period, he contributed to two large enterprise military projects. He worked in various institute project management process improvement teams as project manager.

He was a member of the project team in his first project at TÜBİTAK–UEAKE between 1997 and 1999. That project was the first large project in the institute, and we were using some new technologies, like Java, at that time. We had many problems related with people, development processes, management related issues, structural (product) issues, and technology. This first project lasted about 7 years, and produced a large software system of questionable quality, stress, burnt out developers, higher turnover, reduced esteem and loyalty, weakened capacity for the next project, strained relations among project stakeholders, more experience with an unrepeatable process.

The author left that project and started to form a new enterprise project team as project manager in 1999. He applied some new project management techniques and the level 2 project management processes of CMM (Capability Maturity Model) to his development team. We started to use UML and Design Patterns. These techniques assist the project team in visualizing a system as it is or as it is intended to be, help in specifying the system's structure and behavior, provide a template that guides in constructing the system, document the decisions that the project development team has made. This second enterprise project was completed on-time and on-budget, so in one respect it was a big success. But heavy processes of CMM caused some people related problems. During at about this time, light weight processes like XP (Extreme Programming) started affecting other heavy development processes.

In 8/2002 the author joined the Fatih University as an Associate Professor of the Computer Engineering Department. He continues to work on the problems of enterprise projects as a consultant and a researcher on software engineering.

References

[1] *ISO/IEC 9126-1, Software Engineering - Product Quality - Part 1: Quality Model*, 2001.
[2] D. Parnas, "On the Criteria to be Used in Decomposing Systems into Modules", *Communications of ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.
[3] W.P. Stevens, G.J. Myers, L.L. Constantine, "Structured Design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115-139, 1974.
[4] L. Bass, P. Clements, R. Kazman, *Software Architecture in Practice*, 2nd Edition, Addison-Wesley, 2003.
[5] The CHAOS Reports, http://www.standishgroup.com .
[6] S. McConnell, *Rapid Development*, Microsoft Press, 1996.
[7] S. McConnell, *Software Project Survival Guide*, Microsoft Press, 1997.
[8] J. S. Reel, "Critical Success Factors In Software Projects", *IEEE Software*, May/ June 1999, pp. 18-23.
[9] A. Senyard, M. Michlmayr, "How to Have a Successful Free Software Project", *Proceedings of the IEEE 11th Asia-Pacific Software Engineering Conference (APSEC'04)*, November 2004, pp. 84-91.
[10] J. M. Verner, N. Cerpa, "Australian Software Development: What Software Project Management Practices Lead to Success?", *IEEE Australian Software Engineering Conference (ASWEC'05)*, March 2005, pp. 70-77.
[11] J. M. Verner, W. M. Evanco, "In-House Software Development: What Project Management Practices Lead to Success?", *IEEE Software*, January 2005, pp. 86-93.
[12] G. Booch, *Object Solutions: Managing the Object-Oriented Project*, Addison Wesley, 1996.
[13] *IEEE Standard 1058.1-1987, Software Project Management Plan*.

[14] *Recommended Approach to Software Development*, NASA-Software Software Engineering Laboratory Series, Revision 3, June 1992.
[15] B. Bruegge, A. H. Dutoit, *Object-Oriented Software Engineering: Conquering Complex and Changing Systems, Using UML, Patterns, and Java*, Prentice-Hall, 2004.
[16] J. A. Zachman, "A Framework Systems Architecture", *IBM System Journal*, Vol. 26, No. 3, See also www.zifa.com.
[17] S. McConnel, *Code Complete: A Practical Handbook of Software Construction*, 2nd Edition, Microsoft Press, 2004.
[18] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
[19] M. Fowler, *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2003.
[20] D. Alur, J. Crupi, D. Malks, *Core J2EE Patterns*, 2nd Edition, Prentice Hall, 2003.
[21] I. Jacobson, G. Booch, J. Rumbaugh, *The Unified Software Development Process*, Addison Wesley, 1999.
[22] P. Kruchten, *The Rational Unified Process: An Introduction*, 2d Ed., Addison Wesley, 2000.
[23] P. Clements, ed., *Documenting Software Architectures: Views and Beyond*, Addison Wesley, 2003.
[24] P. Clements, R. Kazman and M. Klein, *Evaluating Software Architectures: Methods and Case Studies*, Addison Wesley, 2002.
[25] M. Fowler, *Refactoring, Improving the Design of Existing Code*, Addison-Wesley, 1999.