Component-Based Development in Geographically Dispersed Teams

Melvin Pérez
MSE 17-652
May 6, 2002

Abstract

This paper discusses how a Component-Based Methodology (CBD) can influence Software Development when it is being done by geographically dispersed development teams. It presents pro and cons specific to CBD, and provide some implementation strategies.

Table of Contents

Table of Figures

Introduction

The purpose of this paper is to discuss how component technology can help or hurt software engineering on dispersed development teams. This paper starts discussing why of this phenomenon, and analyzes the advantages and disadvantages of a Component-Based method for addressing it. Finally, several implementation strategies are presented, and one of them is discussed briefly.

This paper is being presented to **Dr. Sam Harbaugh** as an assignment of *Methods of Software Development* at Carnegie Mellon University (summer 2002).

Dispersed Software Engineering

Why is this happening?

Somebody could ask why we should do Software Engineering with dispersed teams, instead of one team. There are several reasons driving to this software industry trend. The following sections present the most predominant ones.

Space Limitations

Simply, sometimes, due space limitations, we cannot put the entire team within the same room or building. This implies that we have to structure and distribute several teams according to a functional criterion that doesn't hurt the project.

Lack of Expertise

Similarly, sometimes our team doesn't have all the necessary skills for complete a development project, so we have to subcontract a third party for filling those gaps.

Globalization

This long-time buzzword has had influences in almost every business activity around the world, and Software Development is not an exception. By 1995, according with Farshad Rafii and Sam Perkins[1]

[Increasingly, software companies are dispersing their development organizations in an effort to understand local market and customer requirements and to capitalize on technical expertise and cost efficiencies around the globe. Oracle, a database-software vendor, recently announced plans to open an R&D center in Bangalore, India, to create "greater synergy with the Asian region" and to take advantage of "a combination of rich talent, government incentives, and cost effectiveness." It will be the company's fourth international development facility. Lotus has also been expanding its global development presence with efforts on four continents and recently began collaborating in China with the Chinese Academy of Science to translate current Lotus software and to develop new products.]1

Today that is a reality: almost all the major software companies are taking advantage of disparate economies for making theirs better.

Major obstacles

There are several issues associated with doing software engineering with geographically dispersed teams, most of them related to communication and project coordination.

Project Coordination

Working with geographically separated groups for the same final goal is not an easy task. Necessarily, additional roles must be defined, in order to keep each subproject under control and according with the entire project goal. Each team could require a team leader, and some other roles, such as Technical Writers and Test Engineers, could be replicated in each team, instead of having a centralized unit for its respective tasks.

Another issue associated with project coordination is that is not always easy separate a software application that maps to team's structures. Such the work can't be distributed evenly, and at some time one or several teams will be able and hunger for work , while others are having hard time finishing their part.

Communication

Since development is being done concurrently and separately, communication paths increases notably, besides the planning ahead required. This is even worse when locations have different time zones, language and cultures.

[..Physical distance and time-zone and work-calendar differences create complex communication patterns and interaction networks that require significantly greater coordination. Moreover, when team composition crosses national boundaries, social and cultural differences create a greater potential for conflict and misunderstanding, adding another layer of complexity. For example, views toward schedule commitments, the role of the project leader, or the nature of monitoring and control mechanisms differ widely and must be managed more intensively]1

Policies Compliance

To achieve that every team follows the Methodology, Standards and any other Policies, is a tough labor. It's a major risk, since integration and quality could be affected severely.

Costs

Paradoxically, project costs could increase significantly, due the activities and operative platform required to avoid or mitigate the obstacles presented here so far. Hardware and network infrastructure must be arranged, in order to guarantee development times (i.e. if there isn't enough bandwidth for communicating teams efficiently, local servers will be required in every location, instead of one if the entire team were in a single location). Additional costs for traveling and communication should be allocated also.

Component-Based Development

One the common software construction metaphor is *Building Software*[2], and component-based software is simply its major exponent. That's what component software all about is: build software from functional blocks, while hiding the details. Obviously, as McConnell in his book *Code Complete*[2] exemplifies, building a simple structure like a doghouse, require a few simple components (some wood and nails), and is not too hard to put all of them together to obtain the final product (a doghouse). However, building a more complex structure, a house for example will require more planning, more components, and more expertise, due to its complexity.

Component Goals

Components systems manage complexity by using the principle *divide and conquer* (solve a problem by breaking down in to smaller pieces, and build up the solution by using the solutions implemented for those smaller pieces). However, components are primarily implemented with Object-Oriented technology, which implies the use of fundamental principles as *encapsulation, identity,* and *unification of data and function.*

Component's Structure

In order to guarantee that a software application can be built from software components, should comply with certain **standard** that makes its integration easy. This paradigm tries to emulate other more mature domains, like computer hardware, automobiles, and electrics, in which a product can be built from available parts, even from different vendors, complying with a common standard. Another property, a component must have, is its **specification**, which explains how can be used and the other components it can live with. This specification should contain an **interface**, acting as the assembly point with other components. With this structure is valid to say that a component could be **replaceable** by any other component supporting the same standard, and offering the same interface.

Architectural Structure

Any method used for building a software application from components will lay on an architectural approach. However, a question still unanswered: how to break software architecturally? We can conceive a vertical modularization by application functionality and horizontal layering by logic distribution (*see Figure 1 – Layers and Blades of a Software Application.*)

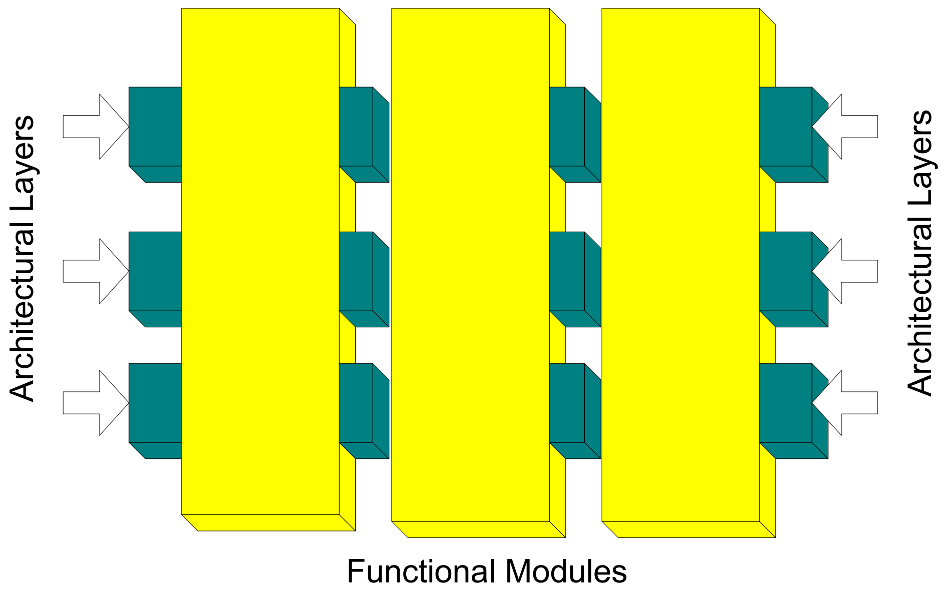

Figure 1 – Layers and Blades of a Software Application

This distribution will be the major strategic step toward to achieve the desired results and make the *divide and conquer* principle works. Component-based software development is a matter of dependency management[3]. So the better we divide the problem, according with dependency, better we can arrange distributed development and get the expected results.

Once we have established the different chunks of our application we can start allocating them to the different development groups. Actually, this is one of the praises of CBD.

Pros and Cons of CBD as Dispersed Software Development Method

The efficiency of any process is determined by the satisfaction of the consumer of its results. So, in terms of software development, I define success as producing a software product on time, on budget, and, the most important one, the continuous customer satisfaction. Even though, we know that continuity cannot be given for sure, we shall guarantee a good level of extensibility. In the following sections, we define the following advantages and disadvantages in terms of such attributes.

Pros

Advantage	**Explanation**
1. Eases work distribution	The entire software application is divided according with functionality and dependency, which makes easier to identify and allocate the appropriate personnel for each component.
2. Eases maintainability	Systems are built by major components with clear interfaces between them. This isolates problems easier than in systems built monolithically. Since such components could be developed by different people in different locations, the only guaranty that every team has is the *contract* or interface between those components, what is verified on integration
3. Parallel Development	Once the dependency has been defined, separate teams can work simultaneously in different part of the system
4. Extensibility	Components can be replaced by better components offering the same interface without affecting the rest of the system.

Cons

Most of these disadvantages are more associated with dispersed software engineering, no matter the method being used.

Disadvantage	Explanation
1. Communication Overhead	Ongoing and informal communication is propagated when working on different components simultaneously. Even though it could happens within a single location it's hampered by the communication mechanism required for dispersed groups
2. Timing	Groups depending on components being developed have to wait until it is delivered. Separate development groups enforce a no-publishing-while-not-tested policy, due the costs of continuous integration. Collocated teams can have informal communications easier and are able to take some assumptions necessary for its activities.
3. Difficult Integration	Since components are being developed separately, components could fail at integration stage due several issues as interface changes, environment changes, etc.

Implementing the Method

Implementing a CBD method with geographically separated groups is a matter of structuring the architecture of both: the software and the team.

Setting up the Team

Is time to analyze how to conform the development team to address the component-based software development process with separate development groups. *Figure 2 - Work Environment Structure* isolated the major parts involved in software development. As such figure illustrates, project coordination is related to every area, which is the major risk area as we have mentioned earlier.

Project Management and Monitoring: Project Manager, Change Control Board, Stakeholders

Software Development: System Analysts, Software Engineers, Technical Writers, Test Engineers, Test Analysts, System Integrator

Software Engineering Process and Standards: Process Engineers, Quality Assurance, Auditors

Components Evaluation and Adoption: Software Architects

Technology Infrastructure Administration and Support: Configuration Manager, Tool Specialists, Database Administrator, Network and System Administrator

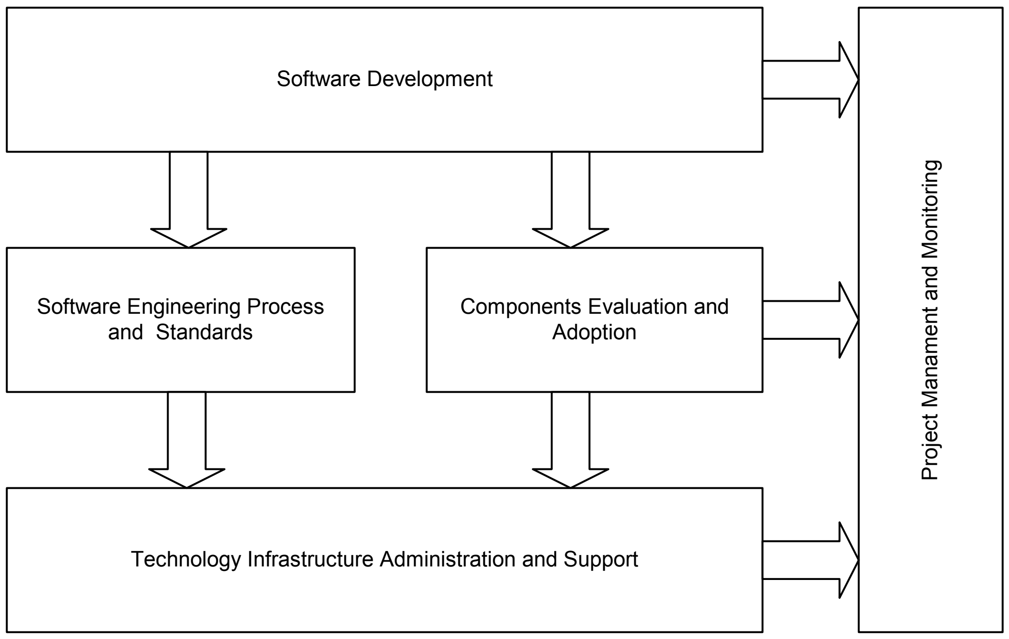
Software Development
Software Engineering Process and Standards
Components Evaluation and Adoption
Technology Infrastructure Administration and Support
Project Manament and Monitoring

Figure 2 - Work Environment Structure

The major challenge is how to structure the application dependency, and the structure of every development group. Which activities can be done centrally and which need to be propagated? Which activities can be done in a serial fashion and which in parallel? So, let's take a look on the different modes we could implement.

Development Mode[1]

The development mode will depend on components dependency, which will define the timing, and on the structure of the development groups. The different development modes presented here are driven by architecture specification and development, since it is the most significant activity for CBD, besides Requirements definition[2]. Additional activities, such as documentation, testing and construction are assumed to be carry out in every group.

The base architecture will consist of the architectural-significant components with its corresponding interfaces. Once specified, the entire team can reference such components, even though haven't been implemented completely, as far as *contracts* are maintained.

Mode 1: Central Sequential.

The entire architecture of the system is developed by a centralized group of Analysts and Architects and subsequently is adopted by the external groups.

[1] Adapted from Internationalizing Software with Concurrent Engineering (see References)

[2] Requirements definition are not covered in this paper, but we assume a Use-Case driven and Architecture-Centric approach is being used

Mode 2: Central Concurrent.

In this mode the base Architecture is developed centrally, but the rest of the architecture is implemented by separate teams. The conditions allow continuous and concurrent integration of code developed by the external groups.

Mode 3: Distributed Sequential

Distributed groups perform architecture development, after the base architecture has been developed. Integration occurs in a serial fashion.

Mode4: Distributed Concurrent

This is most complex strategy. Development activities occur in multiple locations in a parallel fashion. Code developed is integrated simultaneously from different locations.

The optimum development mode depends on the type kind of product being developed and the circumstances and conditions associated with it. However let's take a closer and brief look on *mode 2 (Central Concurrent)*, since it is balanced strategy.

Implementing the Central Concurrent Approach

1. Requirements
    a. Architects, Analysts, and Project Manager envision the system with the stakeholders.
    b. Architect start evaluating commercial components that can be used in the software being developed. If any is chosen it is adapted to any organization standard or policy, and then is adopted as a component of the project.
2. Architecture Definition
    a. Each part of the system is allocated to an Architect.
    b. The Architects of the different parts of the system define the base architecture according with the requirements specified.
    c. Architecture is defined in terms of components with its corresponding interfaces.
    d. Each Architect presents the particularities of its component and set the contracts with the interacting components.
    e. Interaction diagrams[3] are used to specify and document interaction between components
    f. Different scenarios are tested against the Architecture developed
3. Architecture specification is distributed among the different groups
    a. Each group develop the internals of its corresponding architecture
    b. Code is delivered to the central location for integration testing
    c. Any change required in the interface is notified timely for further discussion and adoption

[3] UML collaboration and sequence diagrams

References

[1] Farshad Rafii, Sam Perkins (1995). Internationalizing Software with Concurrent Engineering. IEEE Software Vol. 12, No. 5; September 1995, pp. 39-46

[2] Steve McConnell (1993). Code Complete

[3] John Cheesman, John Daniels (2001) UML Components