Chapter 2

Critical Success Factors for Global Software Development

John works for BAS Corporation, which grew over years through mergers and acquisitions of companies around the world. BAS Corporation wants to consolidate the disparate products from its different companies into a single product line to simplify new product development and achieve economies of scale. John is asked to spearhead this effort. Although he has managed many challenging projects successfully, John has never worked on one that involved coordinating the efforts of development teams from multiple sites around the world. He begins to wonder how his approach to this project would be different from managing a single-site collocated project.

In the age of global economy, many of us are finding ourselves in a similar situation for perhaps a different reason. Labor rates are currently low in Eastern European nations, Brazil, India, and China, and there could be cost savings if development is outsourced to these regions. A company may have expertise in a set of technologies or a domain, thus making it attractive for partnership. Shortage of staff and time-to-market pressures may force an organization to carry out parallel development using a workforce from around the world.

This chapter looks at issues of concern in global software development (GSD) projects and provides factors that are critical to the success of such projects. In addition, it begins to describe a process framework for how

one might realize these critical success factors. The overview in this chapter serves as a road map for the details that follow in subsequent chapters.

2.1 Issues

When organizations initially get involved with GSD, they often drastically underestimate the impact of using geographically distributed teams. One reason for this is that the extent to which people in collocated environments rely on ad hoc and informal communications to develop software is under-recognized. The impact of not having this informal communication and not accounting for its absence is often quite disastrous. The authors have seen this play out time and time again in failed projects from many different organizations.

Based on these experiences, a number of factors critical to the success of a GSD project have been identified. These factors are admittedly high-level and do not in and of themselves adequately convey the intuition born of years of experience. To help explain how these critical success factors can be operationalized, we have developed a process framework to illustrate a process that embodies these factors. This is not meant to suggest that this process framework is a solution for everybody. A one-size-fits-all solution does not exist. Rather, it is meant to further explain and demonstrate how these critical success factors can be realized in a GSD project.

2.2 Critical Success Factors

This section discusses the critical success factors for GSD projects. While these factors are useful for all projects, the business case and necessity is much greater when projects are geographically distributed across several time zones with many remote teams having no history of ever working together, compounded by language and culture differences. These factors are meant to embody experience from many such projects (and thus very abstract). How these factors are realized in any given project depends on many things, such as the organizational processes, the characteristics of the teams, the organizational structures, and the system to be built. Implementing these success factors must be seen as an investment that pays off in risk mitigation and quality. In the remainder of this book we give examples and guidance for how to account for these factors in your projects.

2.2.1 Reduce Ambiguity

While it is desirable to reduce ambiguity for all aspects of any project, a lot of uncertainty and ambiguity gets addressed in collocated projects

through informal communication. GSD projects do not have the same luxury. Ambiguity leads to assumptions. These assumptions are not readily apparent to the powers that be, and therefore conflicting assumptions can exist for quite some time before they manifest themselves as problems. These problems lead to re-planning, redesign, and rework. All of these are difficult to execute in a distributed environment. Coordination can be slow, arduous, and ineffective. The impact of having to do these coordination-intensive activities can cause long delays, leave teams idle and frustrated, cause quality problems, and have (on more than one occasion) caused a project to be stopped.

These ambiguities can be with respect to the organizational processes, the management practices, the requirements, or the design. Combined with a lack of experience and domain knowledge in the teams, the situation can get even more complex. Therefore, much thought and work should go into establishing conventions for how different teams work together to ensure that they extract the intended meaning from project artifacts. Processes should have clearly defined rules of engagement, requirements should be modeled so they are easily understood, architecture should be elaborated with dependencies among modules identified and components specified with well-defined interfaces, work packages should be created for individual teams with clearly stipulated task assignments, and, above all when communication occurs between teams to seek some clarification, one must ensure that answers to questions are correctly understood.

What is a clear articulation to one team with a particular background and culture may very well not be clear to another. Mechanisms for verifying that particular aspects of the project are understood, training programs, or personnel swapping may be required. Staffing should be done carefully to balance technology experience and domain knowledge. The particular characteristics of a project are going to govern the amount of ambiguity that the project can deal with as well as the appropriate mechanism for addressing this ambiguity. For example, if a sponsoring organization has been involved with a long-term relationship with remote teams, likely there is an established working culture that makes it easier to communicate particular aspects of the project, versus two teams that have never before met or worked together.

2.2.2 Maximize Stability

Instability has an influence on many aspects of a project. Agile processes, for example, are a response to unclear and unstable requirements. Some of the major tenets of agility strive to create an environment that optimizes the amount of ad hoc and informal communication (e.g., pair programming,

working in a collocated space, on-site customer representatives, short iterations, daily stand-up meetings, etc.). It makes sense then that in distributed projects, where these types of communications are difficult, stability is a factor. The impact of having unstable aspects of the project is similar to that of having ambiguous areas in your project.

What does this mean, however? Well, again, it largely depends on the particulars of the project, but it may mean that one must delay initiation of the development phase beyond that of typical collocated projects to allow the specification of requirements and design to further stabilize. Change requests are a major risk to GSD projects as they need 2.4 times longer to resolve in distributed environments as compared to collocated projects (Herbsleb and Mockus, 2003). Therefore, the need for excellent requirements engineering and architecture cannot be underemphasized. It may mean that additional prototypes are developed, UI mock-ups are designed, frameworks are developed, or development environments are customized. There are many ways in which the stability of various aspects of the project could be increased. We will give examples and talk more about how this can be done later in the book.

2.2.3 Understand Dependencies

The interdependencies between tasks imply the volume, frequency, and type of coordination needed among the participants in a GSD project. It is critical for planning and execution of the project to have a handle on what coordination needs are likely. Typically, the dependencies are thought of as a "calls relationship" among the subsystems that are to be distributed. While such a relationship does imply a dependency, there are many other aspects to consider as well. From a technical perspective there are many factors that may imply coordination. For example, if there is a hard latency requirement, that could imply some coordination between all the teams involved in developing code that could execute concurrently; or, if one team is responsible for developing a subsystem that requires a complex image processing algorithm, there may be an increased need for coordination.

The technical aspects of the project are not the only sources of dependencies, however. There are many temporal dependencies that stem from the planning or staging of the implementation. For example, implementation might be distributed by development phases or by complete modules or subsystems allocated to different development sites. The latter can be much more advantageous than the former. We go into some detail about how to identify these dependencies and how to account for them in your project.

2.2.4 Facilitate Coordination

The complement to understanding the interdependent nature of the tasks is to facilitate the corresponding coordination. It must be the case that the teams that need to coordinate are able to do so in a way that is commensurate with the need. There are many different ways in which teams can coordinate. While this is usually thought of strictly in terms of communication, it is much broader than that. Communication is one way in which teams can coordinate, but they can also coordinate via processes, management practices, and product line architectures, to name a few. These choices can be seen as a trade-off between overhead and risk. For example, if an organization were to design and develop a framework that would constrain the design and implementation choices of a remote team, while enforcing the remote team's compliance with the original intent of the team, the cost of building such a framework, however, is quite high. The framework would also eliminate the need to communicate the embodied design decisions in other ways.

In our experience, it is often the case that GSD projects put too much emphasis on cost reduction by moving some of their development to low-cost sites. Little attention, if any, is given to investing in improving the process and development environment that could automate to some degree and guide to a great extent the coordination efforts of the collaborating teams.

It is not always clear up-front, however, what the need is or how well a given solution would work in a given environment, so it is often wise to have back-up strategies in place if things go awry.

2.2.5 Balance Flexibility and Rigidity

In many ways the practices of GSD projects should be both more flexible and more rigid than their collocated counterparts. Remote teams often have different backgrounds, different development processes, different experience levels, different domain knowledge, different cultures, and in some cases different organizational practices. The overall development process should be flexible enough to accommodate these differences. This may mean giving the remote teams some freedom in adopting a more agile internal development process. The process, however, also should be more structured and rigid in a particular way. It must be rigid enough to ensure that particular aspects of the project are well defined and the processes followed as the normal safety nets are in place, for things such as instructions are understood, architectures are complied with, requirements are achieved, configuration management (CM) processes are adequately defined, integration and test procedures are appropriate, etc. This is necessary to monitor progress, ensure that deadlines are met, and guarantee quality.

2.3 A Process Framework

Figure 2.1 shows a process framework for GSD. It is not the intent here to describe a rigorous process that every project should follow. Rather, to more easily introduce the concepts found within this book and demonstrate how the critical success factors can be realized, a high-level process framework with steps that project teams are likely to follow is described. This is not intended to be a one-size-fits-all process. While we have used this process in practice, it usually requires tailoring and adoption for the given context. The activities within this framework are shown as sequential steps covering the areas of requirements, architecture, project planning, and product development, but in practice these steps will be highly iterative.

The goal of the requirements engineering step of the process is to gain a good understanding of the requirements so one knows what is it that one is trying to build, and to identify the driving requirements — those that are architecturally significant and have the most value to the business for which the system is being built. We use model-driven requirements engineering (Berenbach, 2003, 2004a, 2004b) that uses the Unified Modeling Language (UML) to specify the requirements. UML as a specification does not provide any guidance on how it must be used; we have, through our research, developed approaches that leverage UML's ability to add meaning to the links among the requirements, thus making them easier to analyze for completeness, consistency, and quality as well as allow for automated tracing to design models and tests.

While model-driven requirements engineering is a systematic development and maintenance of detailed product requirements that begins when

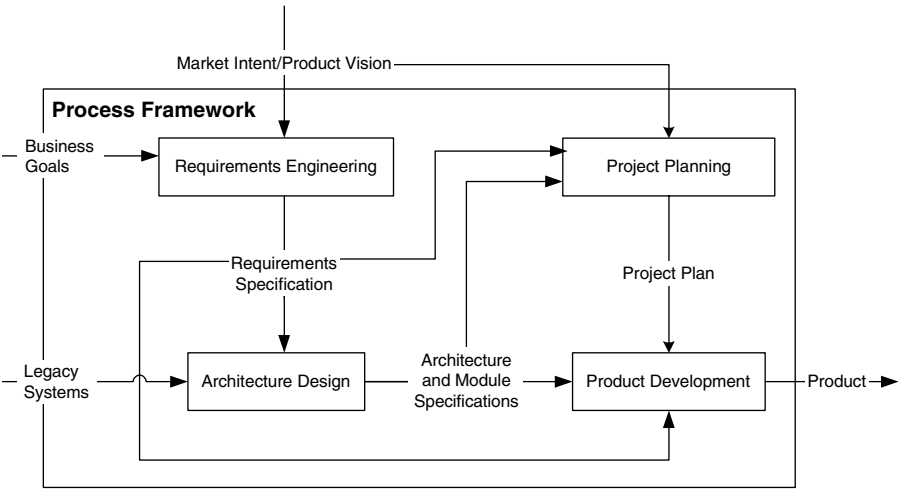

Figure 2.1 A process framework for GSD.

a project has gained enough momentum, there is a need early in the project to develop a business case, define a product, and gather customer requirements. This can be accomplished through prototyping and storyboarding (Song et al., 2005). To discover key architecture-relevant requirements, their interactions with other project requirements, and formulate strategies to address any resulting conflicts, we use a technique called Global Analysis (Hofmeister et al., 2000).

For globally distributed software development projects, model-driven requirements engineering offers several benefits that help overcome some of the issues related to communication, coordination, and control. Visual modeling aids comprehension and communication among distributed teams. Tool support for evaluating the models gives on demand a measure of size, progress, consistency, completeness, and quality. Most significantly, however, UML models can support automated traceability; modeled requirements can be traced forward to their corresponding design and tests, and backward to the market intent, stakeholder requests, and business goals. This can help with requirements coverage (i.e., what requirements have been implemented in a given release) and impact analysis (i.e., what is the impact of a requirement change).

The focus of the architecture design step is to get a detailed understanding of the architecturally significant requirements and create an executable architecture. Many software development methodologies treat architecture only indirectly or implicitly. The quality of systems developed using such methodologies thus depends largely on the skill level and experience of its architect. We use architecture-centric methods (Bass et al., 2003) such as the Quality Attribute Workshop (QAW), Attribute Driven Design (ADD), and the Architecture Tradeoff Analysis Method (ATAM), which provide explicit and methodical guidance to an architect in creating systems with desirable qualities. The QAW uses stakeholders early in the software development life cycle to discover the quality attribute requirements that drive the structure of a system; ADD uses these requirements to design the architecture; and the ATAM helps stakeholders understand the consequences once an architecture for a system has been determined.

The project manager uses the requirements specification and the architecture and module specifications to generate a project plan to implement the new product (Paulish, 2002). The project manager performs risk analyses, defines proposed project strategies, does incremental release planning, and generates a proposed development plan describing how and when the product should be developed. Estimation methods are used to determine the effort and schedule to implement the product.

In the product development step, the application modules are designed, implemented, tested, and integrated to produce intermediate product releases, some of which are beta tested at the customer site. This eventually

produces a product baseline mature enough to be deployed in the user community.

The organization of product solution development is optimized using small, distributed module development teams synchronized by a central organization. The development teams worldwide receive module specifications from the central team. Each team, led by a supplier manager, is responsible for the design, implementation, and testing of the assigned module. In our experience, the role of the supplier manager is often not sufficiently defined and implemented but is a key role in our approach (see Chapter 10).

In keeping with the best practices of agile methodologies, these teams carry out iterative implementation of their modules in small increments using test-first development. Tests to exercise module interfaces are provided by the architecture team because these tests not only help verify the modules, but also help developers understand the semantics of the interfaces much more clearly (Simons, 2002). The modules are designed using known design patterns and refactored on a continuous basis to maintain their quality (Fowler, 2004).

2.4 Development Phases and Decision Points

We use a phased product planning process. Like the standard Rational Unified Process (RUP), it has four significant stages in the software development life cycle (Arlow and Neustadt, 2002; Larman, 2005). The *inception phase* is the visionary milestone phase wherein the problem to be solved and its potential solutions are investigated to determine the feasibility of the project. Once considered feasible, the project enters into the *elaboration phase*. This is the core architecture milestone phase wherein requirements are prioritized and those deemed architecturally significant are implemented first. At the end of the elaboration phase, a more reliable software development plan is put into place and the *construction phase* of the software begins. This phase is the operational capability milestone phase because at the end of this phase the software is deployed in the production environment of one or more beta customers to demonstrate its operational capability. Once operational, the software moves to the *transition phase*. This is the product release milestone phase, as the software product is now generally available in the market.

Tip: Make offshore members a part of the central team in the early phases of the project.

A number of requirements and architecture related activities require collaboration to impart domain and

architecture expertise to the offshore teams. It is advisable in the inception and elaboration phases to involve members of the offshore teams so they begin to gain an understanding of the domain and the architectural vision. These members can then be relocated to their home sites when the construction phase begins and act as local experts. Furthermore, we suggest that these key members of the offshore teams be relocated to the central site with their families. This minimizes home-sickness, and provides local support as "culture shock" issues arise.

These phases span over multiple iterations, each iteration leading to an executable release of a fraction of the complete product. The number of iterations in the phases and their duration will depend on the size of the project. We use Scrum techniques (Schwaber, 2004; Schwaber and Beedle, 2001), and therefore our iterations for large projects are further subdivided into monthly *sprints*. Teams are assigned work packages for an iteration that are completed incrementally on a monthly boundary. We call these increments *Engineering Releases*. At the end of an iteration, several engineering releases make up an executable release of a fraction of a product. For smaller projects, the duration for an iteration may well be the same as that for a sprint and, therefore, this type of planning may not be necessary.

These phases often have management decision points separating them. A decision point review is conducted to evaluate the results of the current phase, the proposed plans for the next phase, and authorize the resource investment required for the next phase.

A phased product planning process is used to achieve the following benefits:

- Product development is viewed as an innovation funnel of new ideas. Many ideas are researched, but a smaller number of the best ideas are invested in to become products.
- Early phases typically require fewer resources than later phases, where potentially large development teams are required to implement and test the new product.
- Communications from development to the field sales force are controlled during early phases. The sales force is discouraged from selling new ideas rather than existing or very soon-to-be-developed products. Product research ideas, if communicated too early, can increase customer expectations and sometimes negatively impact the sales of current products.

- Quality targets can be better identified, measured, and controlled. The decision point reviews can be used as gates by which a project must meet certain quality criteria before proceeding to the next phase of development.
- Decision point reviews allow management to view and control proposed budgets by project, by phase. If a particular potential product looks promising in the market, but its development requires an unanticipated investment, fiscal-year budgets can be modified or trade-offs can be made among projects.
- Less promising projects can be stopped. Phases and decision points help management decide how to best invest in future products and balance their portfolio of products.

New product development can consist of some incremental improvements to a current product as a new version or release, a new product within an existing market segment, or an entirely new product line. The focus of this book is on new initiatives where a new or modified software architecture is required to implement the product. Thus, situations where relatively minor new features are added to an existing product with an existing architecture are not addressed. In this situation, organizations usually define an incremental release plan, and enhancements are made until the new version is released to the field. Phased approaches can also be used for major functionality additions, where investments are being made in parallel for requirements definition for the next or future releases of the product currently being developed.

This book puts special emphasis on the early phases of software product development. If requirements analysis, architecture design, and project planning are done well during the early phases of development, the product is much more likely to be developed per scope, schedule, and budget than if these activities are skipped or done poorly. On the other hand, these activities are never fully completed, and issues are often left open until the very end of development. Thus, a set of guidelines, rules of thumb, and tips are provided for effectively managing the early phases of software product development.

An example phased product planning process is given in Figure 2.2. The types of management decisions (*D*) that are made for our example phased product planning process are:

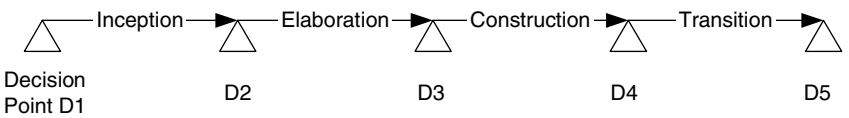

Figure 2.2 Example phased product planning process.

- *D1*: decision to initiate research (product, technology, or market) for a new product or products.
- *D2*: decision to initiate requirements definition and analysis and high-level architecture design.
- *D3*: decision to develop the product. A proposed project plan is reviewed at this meeting identifying the scope, schedule, and investment required to develop the product per the requirements and architecture defined within the elaboration phase.
- *D4*: decision to release the product to the market for customer use. Some organizations can also add an intermediate decision point where the new product is announced to the field prior to the completion of its development.
- *D5*: decision to sunset or remove the product from the market. Intermediate decision points can also be added to distinguish when the product will no longer be sold to new customers versus when the product will no longer be maintained with current customers.

2.5 Summary and Conclusions

This chapter identified issues of concern in GSD projects and provided factors that are critical to the success of such projects. A process framework for software development with geographically distributed teams that leverages the critical success factors in addressing some of the issues in GSD projects was described. A management framework for evaluating current phase and initiating the next phase was discussed.

2.6 Discussion Questions

1. List some of the significant differences between single-site collocated and multi-site distributed software development projects. What are some strategies to manage these differences?
2. Justify why model-driven requirements engineering is superior to text-based requirements engineering. Why do you think it is particularly important to global software development projects?
3. What is the significance of architecture-centric methods to the architecture of a system and to project planning?
4. The various decision points at the beginning and end of the software development phases act as quality gates. What do you understand by this statement?

References

Arlow, J. and Neustadt, I., *UML and the Unified Process: Practical Object-Oriented Analysis and Design*, Addison-Wesley, Boston, MA, 2002.

Bass, L., Clements, P., and Kazman, R., *Software Architecture in Practice, second edition*, Addison-Wesley, Boston, MA, 2003.

Berenbach, B., "The automated extraction of requirements from UML models," *Proceedings of the 11th Annual IEEE International Requirements Engineering Conference (RE'03)*, Monterey Bay, CA, September 8–12, 2003, pp 287–288.

Berenbach, B., "Towards a Unified Model for Requirements Engineering," *Fourth International Workshop on Adoption-Centric Software Engineering (ACSE 2004)*, Edinburgh, Scotland, U.K., May 23–28, 2004a, pp. 26–29.

Berenbach, B., "The evaluation of large, complex UML analysis and design models," *Proceedings of the 26th International Conference on Software Engineering (ICSE 2004)*, Edinburgh, Scotland, U.K., May 23–28, 2004b, pp. 232–241.

Fowler, M., "Using an Agile Software Process with Offshore Development," April 2004, retrieved on November 6, 2005, from Martin Fowler Web site: http://www.martinfowler.com/articles/agileOffshore.html

Herbsleb, D., Mockus, A., "An empirical study of speed and communication in globally distributed software development," *IEEE Transactions on Software Engineering*, 29(6), 481–494, June 2003.

Hofmeister, C., Nord, R., and Soni, D., *Applied Software Architecture*, Addison-Wesley, Boston, MA, 2000.

Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, third edition*, Prentice Hall, Upper Saddle River, NJ, 2005.

Paulish, D., *Architecture-Centric Software Project Management*, Addison-Wesley, Boston, MA, 2002.

Schwaber, K., *Agile Project Management with Scrum*, Microsoft Press, Redmond, WA, 2004.

Schwaber, K. and Beedle, M., *Agile Software Development with Scrum, 1st edition*, Prentice Hall PTR, Upper Saddle River, NJ, 2001.

Simons, M., "Internationally Agile," *InformIT*, March 15, 2002, retrieved on November 6, 2005, from InformIT Web site: http://www.informit.com/articles/article.asp?p=25929

Song, X., Rudorfer, A., Hwong, B., Matos, G., and Nelson, C., "S-RaP: A Concurrent, Evolutionary Software Prototyping Process," *Proceedings of the Software Process Workshop*, May 25–27, 2005, Beijing, China.