Chapter 4: Methodology for Information Systems Project Management

CHAPTER OUTLINE

Introduction: The Methodology Choice Becomes the Process
A Taxonomy of Information Systems Project Types
A Generic IT Project Management Process
Software Development Projects
    Waterfall Model, Evolutionary Development Model, Transform Model, Spiral Model
Re-engineering Projects
System Integration Projects
System Maintenance Projects
Component Integration Projects
Rapid Application Development Projects
Agile Software Development
Project Management with Scrum
Selection Methodology: For Software, Processes, and Projects
Methodologies for System Conversion/Cut-Over Projects
Project Standards
    Summary
    Exercises
    References
    Supplement 4.1
    Supplement 4.2

LEARNING OBJECTIVES

After reading this chapter you will be able to:
1. Discuss why process selection is important.
2. Describe what the major IT project types are.
3. Identify what steps are commonplace to all projects.
4. Understand why there is such a strong focus on shortening the duration of projects.
5. Learn how to get projects done quicker and with less cost by modification of the project process or methodology.

INTRODUCTION: THE METHODLOGY CHOICE BECOMES THE PROCESS

Box 4.1: What's all this Process Stuff?

With the advent of the early 1990s, corporate analysts began to investigate processes—the mechanisms by which stuff gets done, product gets delivered in response to a customer order, or a new product gets developed, for example. Prior to that time, analysts focused on resources, on organizational structures, but never processes. Organizations were the focus in the 1970's while resources were concentrated on in the decade of the 80's. But processes have been the focus of the 1990's and into the 21st century. Processes are nothing more than the steps by which a particular objective gets accomplished, on a repetitive basis.

As microcomputers, local area networks and such became increasingly commonplace in corporate America, organizations began assessing the productivity gain that had accrued from them. America's Fortune 500 firms had spent literally billions of dollars, yet as late as 1994, there was still little discernible productivity gain. For the most part, these IT investments were used to automate existing processes. The end result was a huge increase in capacity but very little productivity benefit. Then analysts began suggesting that, by using the same technology, the processes could be completely re-engineered, re-designed so that the work could be accomplished much quicker, at lower cost and with better quality. This was the turnaround that totally changed the productivity measurements. By mid 1995, the government and industry analysts were seeing substantial increases in productivity. What are the implications for all of us—quite simply a better quality of life, a better standard of living and more wealth to be shared by all.

So why a chapter on methodology? The methodology is the process by which the work is accomplished. If better methodologies can be found, it follows that IT projects can be completed more quickly, cheaper and with better quality. Particularly in today's fast-paced world, speed is an issue. Project managers are continually being asked to complete their projects more quickly in order to meet management-established deadlines. In this chapter, we discuss the various IT project types and present their rudimentary methodologies.

Indigenous to every project is its processes—the methodological steps by which the project is brought to fruition and the deliverable is created and turned over to the client. In this chapter we briefly review the general methodology presented in Chapter 3. We follow that up with detailed methodologies specific to a variety of information technology project types. Software development projects, web-based projects, re-engineering projects, configuration projects, conversion projects, maintenance projects, component integration projects, system integration projects and rapid application development projects all possess methodologies that are specific to that type of project. In this chapter you will learn to identify the more prominent IT project types. It is important that readers familiarize themselves with these types so they can classify a particular project as belonging to a specific type and thereby select the appropriate methodology for its completion. Systems integration consulting firms pride themselves on their "methodology" which they believe gives them an advantage over their competition.

What if there is no methodology. Well, there has to be because no-methodology is itself a methodology, usually known as the "on-the-fly" methodology. According to Greenberg (1998), the "on-the-fly" methodology gets used more often than any of us would like to admit.

A TAXONOMY OF INFORMATION SYSTEMS PROJECT TYPES

Most information systems students are familiar with software development projects in which software is created from source code. While this is an important type of IT project, it is not the most important, not even the most prevalent. For example, it is estimated that maintenance projects are several times more prevalent than development projects for one simple reason—for

every dollar spent on development, firms are inadvertently setting themselves up for expenditure of three-to-five dollars on maintenance down the road. What does this mean? Suppose a firm spends $1,000,000 developing a modest application. Down the road (and probably within the next ten years), that same firm will ultimately spend another $3,000,000 to $5,000,000 just maintaining that same application over its useful lifetime. This is where the concept of **lifecycle costs** comes from. Lifecycle costs are all the costs incurred in the creation and use of a product from inception and development to maintenance over many years of use and final removal. For twenty years software engineers have been concerned with ways to reduce lifecycle costs, suggesting that by spending more on development up-front, maintenance costs will be lower as will lifecycle costs in general.

Another project in vogue, enterprise resource planning, has become very significant in terms of resources committed to it. Perhaps you've heard of SAP[1]. SAP is a German software product collection that is used to replace old mainframe financial legacy codes, specifically accounts receivable, accounts payable, payroll, inventory, general ledger. During the 1990's, firms were driven to replace their old legacy codes because these had Y2K problems. Such replacement led to the acquisition of COTSS—commercial-off-the-shelf-software such as that available from Oracle, IBM, Microsoft and SAP. All of these software firms are significant players in the enterprise software market, which grew to a market of 100 billion (US) annually. This project belongs to the installation/conversion /cut-over category of IT project and is very different from the traditional software development type of project.

So far, we've discussed two commonplace project types—maintenance projects and installation/conversion/cut-over projects. In total, what project types are there? A list of the main ones (that will be discussed here) appears below.

Table 4.1: A List of Major IT Project Types

Analysis Projects
Architecture Projects
Component Integration Projects
Development Projects:
Internet Development Projects
Component-based Development Projects
Decision Support System Development Projects
Data Warehousing and Data Mining Projects
Component Development Projects
Rapid Application Development Projects
Installation/Conversion/Cut-over Projects
Maintenance Projects
Re-engineering Projects
Visioning and Storyboarding Projects

A GENERIC IT PROJECT MANAGEMENT PROCESS

We have been characterizing project lifecycles as having four major stages, a definition stage, a planning stage, an execution stage and a termination stage. In this section, we provide additional detail regarding these stages in terms of steps that we believe most all projects must complete. See Figure 4.1.

In the **conceptualization and definition** stage, the following steps are appropriate:

1. Verify that a detailed study is worth doing

[1] SAP is an acronym for Systenabakyse und Programmentwicklung (systems analysis and program development) a German software firm started in 1972 by five renegade IBM engineers in Mannheim, Germany. It is the world's fourth largest software firm trailing only Microsoft, Oracle, and IBM.

2. Determine owners and stakeholders
3. Obtain user requirements
4. Define scope, size, and resource requirements
5. Ensure fit with business strategy and priorities
6. Assess technology consistency
7. Identify dependencies with other projects
8. Assess overall risk
9. Test alignment with/impact on strategies and plans
10. Test resource availability
11. *Make go/no-go decision*
12. Endorse/obtain funding
13. Review alternative approaches
14. Commit resources
15. Assign project management

The second stage, **planning and budgeting**, may involve the following steps, generally.

1. Develop impact statement
2. Define control requirements
3. Determine probable architecture
4. Identify solution centers and service providers that might be involved
5. Determine possible alternative solutions
6. Initiate controls analysis
7. Develop project organization proposal
8. Recommend steering team
9. Perform risk contingency planning
10. Determine resource availability and identify project team
11. Prepare project plan (work breakdown structure, project schedule, project budget, project organization and personnel)
12. Prepare presentation
13. Obtain signature approvals

The third stage, **execution and control**, may entail some or all of the following steps, generally:

1. Develop a detailed technical design
2. Design/re-engineer infrastructure processes
3. Conclude all contract negotiations
4. Design start-up and steady state organization, roles and responsibilities
5. Obtain or build testing tools
6. Obtain, install and test hardware components
7. Obtain and install software packages
8. Perform programming, parameter setting and other software construction tasks and perform testing
9. Manage changes to specifications
10. Develop disposal plans for obsolete hardware and software
11. Prepare detailed start-up plan
12. Develop fallback plan to respond to start-up problems
13. Perform reliability and scalability testing
14. Complete detailed documentation
15. Pilot solution where appropriate
16. Conduct acceptance test

The fourth and final stage, **termination and closure**, might involve the following steps:

1. Deliver the completed software product
2. Start-up the completed software product
3. Monitor and repair as necessary
4. Acceptance by customer

5. Prepare project close-out report
6. Conduct investment reappraisal
7. Document lessons learned
8. Release team, resources
9. Get paid

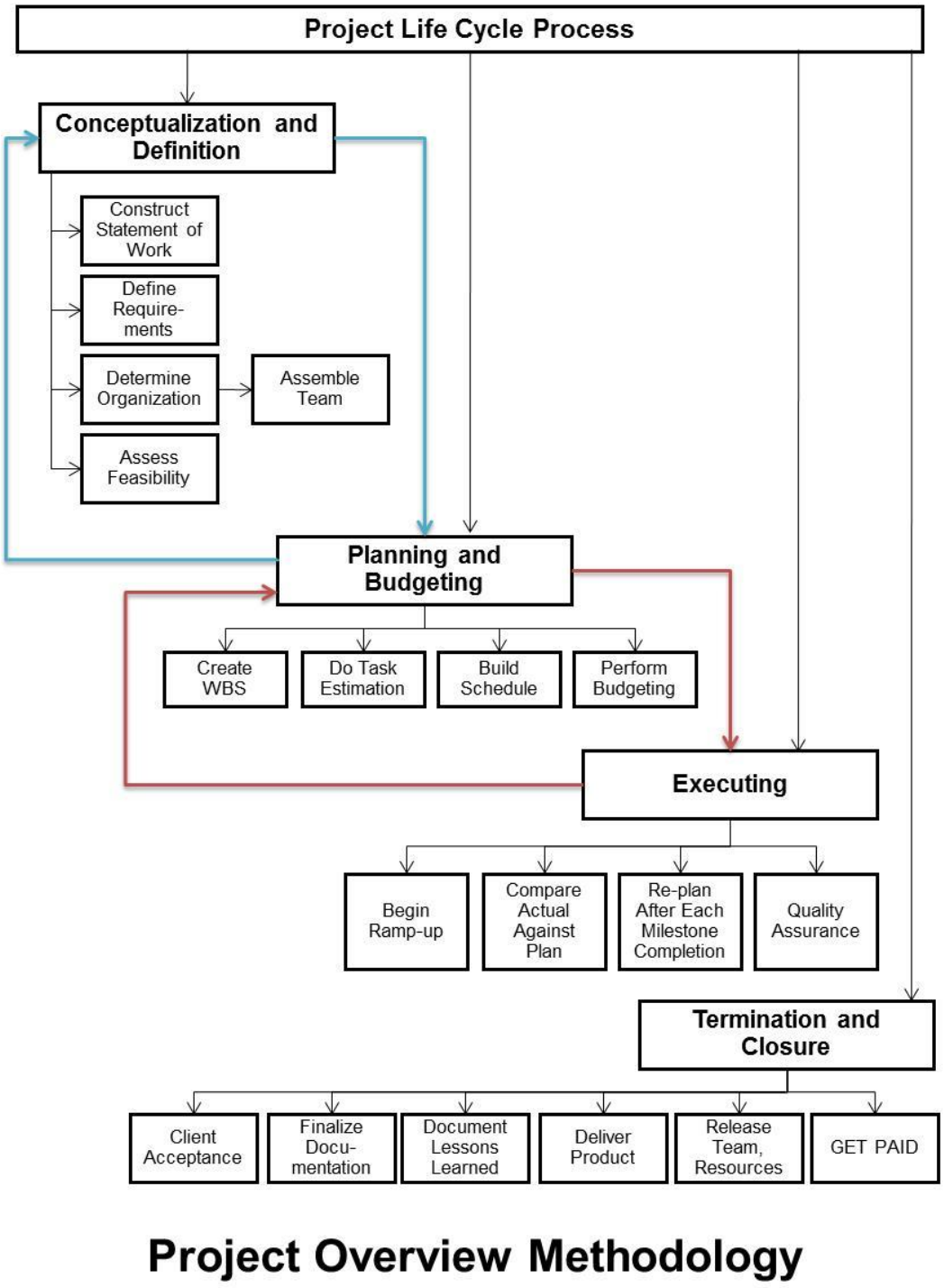

Figure 4.1: Project Overview Methodology

SOFTWARE DEVELOPMENT PROJECT

Perhaps the most commonplace of all software projects is the development project. In the development project, a new software product is created, at least some of it from source code—Java, C++, XML, Smalltalk, COBOL, Visual Basic, etc. The commonest software development process model[2] is the waterfall model. It is still used, especially in software projects for the federal government, where it is a requirement in most projects. It is depicted below in Figure 4.2.

Waterfall Model. The waterfall model was developed to overcome the difficulties of the older **code-and-fix models**, the earliest of software development process models. In the code-and-fix model, the developer wrote some code and then fixed the problems in the code. Then, the developer thought about the requirements, design, test, and maintenance later. This model was found to be replete with a number of problems. Among these, after a number of fixes, the code became so poorly structured that subsequent fixes were very expensive. In addition, even well-designed software was such a poor match to the user's needs that it was either rejected outright or expensively redeveloped. Finally, the code was expensive to fix because of the absence of structure or consistency to the code. The code and fix model was used long before structured programming was ever thought of. The stages (steps) of the waterfall model are indicated below in Table 4.2.

Box 4.2: The Real Software Development Process

Out of jest and with tongue firmly in cheek, someone suggested the following software development process:

1. Order the T-shirts for the development team
2. Announce availability of the product (This helps to motivate the team.)
3. Write the code (Let's get with it!)
4. Write the manual
5. Hire the project manager
6. Spec the software (Writing the specs after the code helps to ensure that the software meets the specifications.)
7. Ship
8. Test (The customers are a big help with this step.)
9. Identify bugs as potential enhancements
10. Announce the upgrade program

Table 4.2: Steps of the Standard Waterfall Model

NUM	NAME
1	Requirements Determination
1.1	Interviews
1.2	**Prepare Requirements Document**
1.3	**Prepare Project Plan**
1.4	**Prepare Proposal**
2	Analysis
2.1	Interviews
2.1.1	Management
2.1.2	Supervisor
2.1.3	Technical
2.1.4	Clerical
2.2	Analyze Existing Documents
2.3	Synthesis of Interviews with Existing Documents
2.4	Delineate Architectural (Top-level) Design

[2] The word "model" is used in the literature to connote "methodology." Its use here is synonymous with "methodology."

2.5	Delineate Data Flow and Entity-relation Diagrams
2.6	Prepare **Functional Specification**
2.7	Re-estimate Cost and Duration, Adjust Schedule as Necessary
2.8	Write **Development Proposal** (optional)
2.9	Presentation to Client
3	Design
3.1	System Design (medium level design)
3.2	Walkthrough
3.3	File Design
3.4	Walkthrough
3.5	Write **Acceptance Test Plan**
3.6	Write **Design Specification**
4	Construction
4.1	Plan the Integration
4.2	Module Design
4.3	Walkthrough
4.4	Plan Module Testing
4.5	Coding
4.6	Module Test
4.7	Write **User Documentation**
5	System Integration Testing
5.1	Assemble **Test Documentation**
6	Acceptance Test
7	Installation/Conversion/Cut-over (Implementation)
8	Operation
9	Maintenance

Except possibly the first, all of these steps would fall within the control and execution stage of the development project.

The numbers shown to the left in Table 4.2 above are there to indicate subordination and precedence. The waterfall model is the commonest depiction of the steps in structured software development. One problem with the waterfall model is its emphasis on fully elaborated documentation. The documents are shown in bold in the methodology given above. In a sense, the waterfall model is a document-driven methodology. All of the required documentation makes the waterfall model both time-consuming and expensive. This is particularly true when it becomes necessary after several steps to go back to an earlier phase, say from construction back to analysis. When this happens, all the intervening documents have to be rewritten.

As mentioned, almost all software development performed for the federal government is required to use the waterfall model as the contractor must conform the specifications to government-defined standards that embody the waterfall model.

Another way to view the waterfall model is as a sequence of waterfall stair-steps each interacting with adjacent steps through cycles, as shown in Figure 4.2.

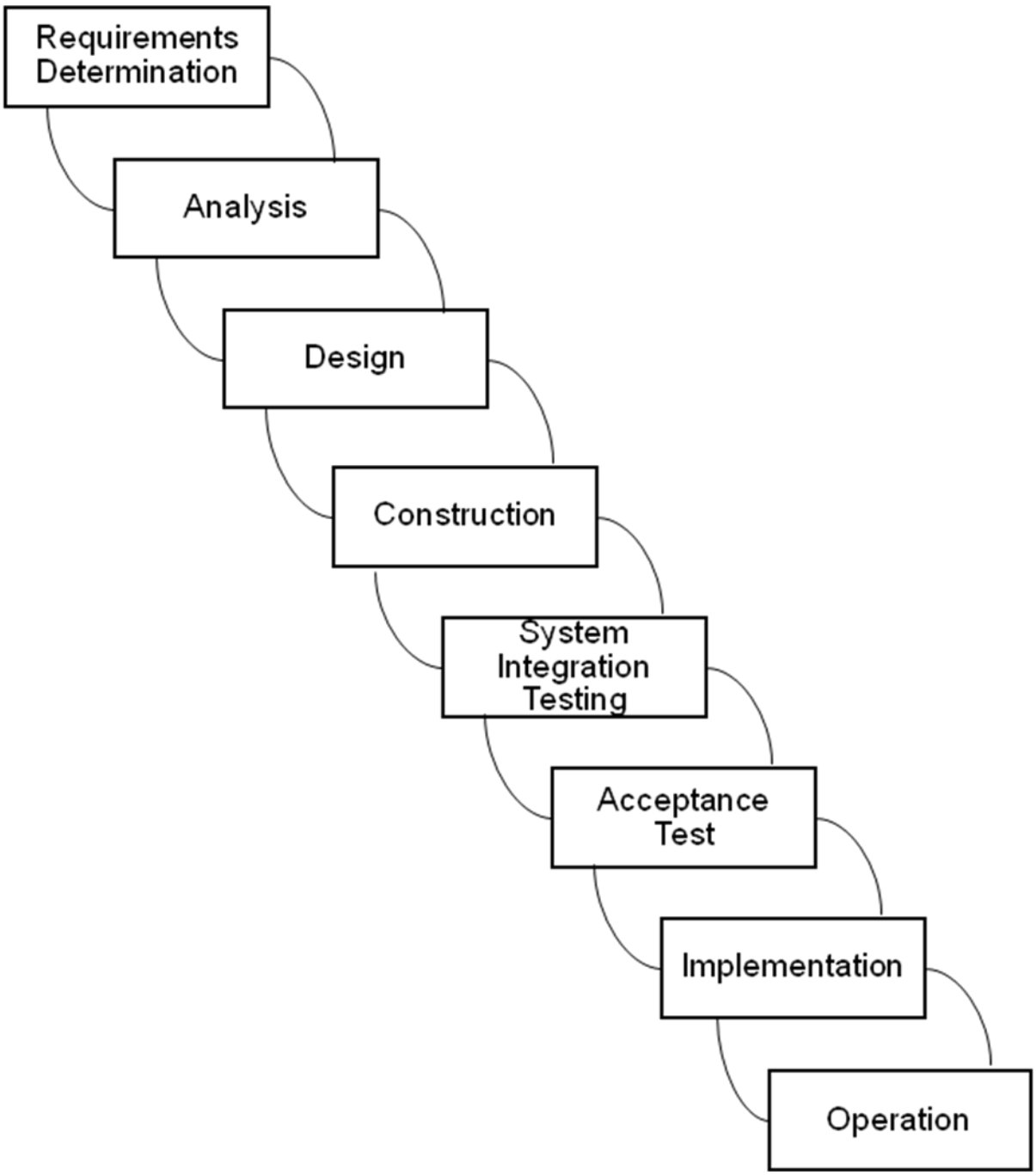

Figure 4.2: The Standard Waterfall Software Development Model

For some classes of software, the waterfall methodology is appropriate and useful, as is the case when developing an operating system or a compiler or an artificial intelligence engine or database engine. However, it is well-known that the waterfall model does not work well when developing interactive end-user applications. It is simply too expensive and too slow.

The Evolutionary Development Model. According to Boehm (1988), a better choice for interactive end-user software development would be the evolutionary development model (EDM). The phases of this model consist of expanding increments of an operational software product,

with the directions of evolution being determined by operational experience. The EDM is ideally matched to fourth-generation language applications like Visual Basic, PowerBuilder, Delphi, etc. The EDM, according to Boehm, also works well when users say, “I can’t tell you what I want, but I’ll know it when I see it.” This development methodology gives users a rapid initial operational prototype that they can play with and react to, even improve upon.

Nevertheless, the EDM has its difficulties. First, it is not much different from the old code-and-fix model. Thus, the evolutionary model shares many of the same shortcomings of the old code-and-fix model, including an inability to accommodate many and varied modifications. Second, it doesn’t have the flexibility to accommodate future unplanned evolutionary paths.

The Transform Model. The difficulties associated with the code-and-fix, waterfall and evolutionary models were addressed by the transform model, but not very successfully. The transform model assumes the existence of a software module that can “parse, interpret and compile” written specifications automatically into machine code. When changes occur to the specifications, these are simply rewritten into the old specifications and “recompiled” to machine code. With the transform model there is no intermediary code. Hence, the transform model bypasses the difficulty of having to modify code that has become poorly structured through repeated re-optimizations, since the modifications are made to the specification.

Even so, the task of writing machine-comprehensible specifications can be arduous even for the best of transform modules. And, automatic transformation capabilities are available only for small software products derived from spreadsheets, and small language applications. Large-scale development could not use the methodology. Further, the transform model, like the evolutionary model does not accommodate unplanned evolutionary paths well.

The Spiral Model. The spiral model was developed by Barry Boehm (1988), one of the most esteemed and respected of software engineering academics. According to its Dr. Boehm, the spiral model can accommodate most previous development methodologies as special cases. Further, the spiral model provides guidance as to which combination of previous models best fits a given software situation.

Unlike the document-driven waterfall model, the spiral model is a risk-driven methodology. As exhibited in Figure 4.3, it endeavors to depict both the passage of time and the accumulation of expenditure. The angular dimension represents the progress made in completing each cycle of the spiral, while the radial dimension represents the cumulative cost incurred in accomplishing the steps.

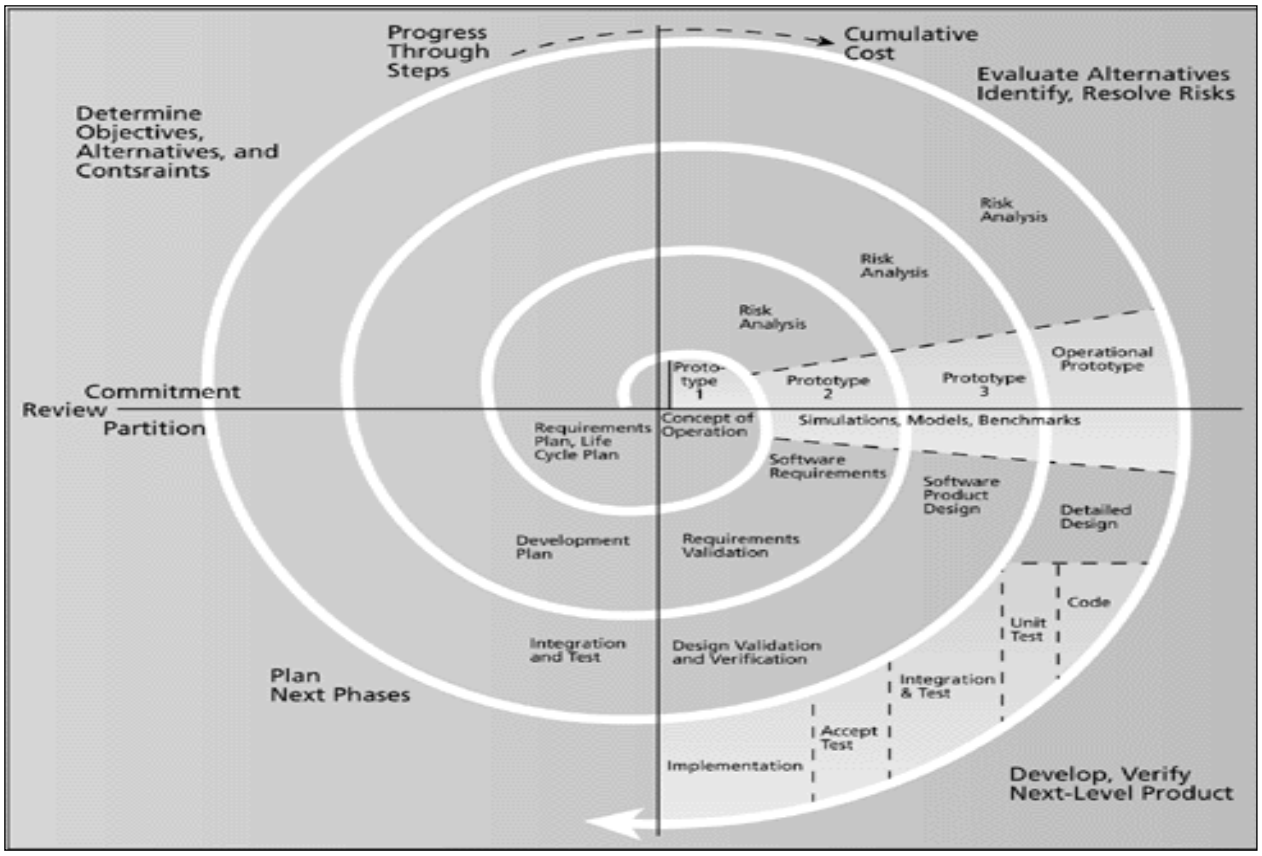

Figure 4.3: The Spiral Development Model

Each loop of the spiral begins with the following: 1) identify the objectives of the product being elaborated; 2) identify the alternatives to implementation of the product, and 3) determine the constraints imposed on the alternatives. The next step is to evaluate the alternatives relative to the objectives and the constraints. This step will surface significant sources of project risk. These sources should then drive the choice of methodological detail that alleviate or mitigate these perceived sources of risk. For example, if there is risk associated with the ultimate design of the product, then prototyping, simulation or benchmarking should be utilized to mitigate this risk. If there is risk associated with the specification of the product, then reference checking, administering user questionnaires, analytic modeling, or combinations of these and other risk-resolution techniques might be appropriate. In the spiral model, each step is determined by the relative remaining risks.

Thus the methodology is dynamic, and dependent upon the relative risks remaining. If user-interface risks strongly dominate product considerations, or there are internal interface control risks, then the next step may be to use evolutionary development. Risk considerations can lead to a project implementing only a subset of all the potential steps in the model.

The risk-driven nature of the spiral model allows the model to accommodate any conceivable combination of the other methodologies as sub-methodologies. Each loop iteration of the spiral model is completed only after a formal review committee has approved it.

If a project has a low risk in such areas as getting the wrong user interface or not meeting stringent performance requirements and if it has a high risk of not meeting cost or schedule commitments, then these considerations drive the spiral model into an equivalent of the waterfall model.

On the other hand, a project may have a low risk in such areas as staying within budget and on schedule, or of encountering large-system integration problems, or coping with information sclerosis. That same project may have a high risk of a wrong user interface or of incorrect user decision support requirements, then these risk considerations drive the spiral model into an equivalent evolutionary development model.

Advantages of the spiral model include the following:

1. The spiral model applies equally well to software maintenance projects as it does to development projects; thus the need for separate methodologies are eliminated.
2. It incorporates prototyping as a risk-reduction option at any stage of development.
3. It accommodates reworks or go-backs to earlier stages as more attractive alternatives are identified.
4. It focuses early attention on options involving the reuse of existing software.
5. It accommodates preparation for life-cycle evolution, growth, and changes of the software product.
6. It provides a mechanism for incorporating software quality objectives into software product development.
7. It focuses on eliminating errors and unattractive alternatives early.
8. For each of the sources of project activity and resource expenditure, it answers the key question, “how much is enough?”
9. It provides a viable framework for integrating hardware-software system development.
10. If CASE or other types of transform tools are available and useful, then the spiral model accommodates them either as options for rapid prototyping or for application of the transform model.

Some possible difficulties with the spiral model include its use by outside software contractors and system integrators. These outside developers find its use difficult because of possible inflexible contractual mechanisms, such as would be associated with fixed-price contracts[3]. Recently, progress has been made in establishing more adaptable contract provisions such as the use of competitive front-end contracts for concept definition or prototype studies, the use of level-of-effort and award-fee contracts for evolutionary development, and the use of design-to-cost contracts.

The spiral model relies on project professionals with risk-assessment expertise. Teams of inexperienced professionals will waste time on low-risk elements of the projects so as to give the allusion of progress, but make no real progress at all on the difficult, high-risk elements.

In general, the steps of the spiral model need further elaboration. For example, there is a need for more detailed definitions of the nature of spiral model specifications and milestones, definitions of the nature and objectives of spiral model reviews, and a need for techniques for estimating and scheduling. Moreover, there is a need for definition of the nature of spiral model status indicators and tracking procedures. Highly experienced professionals can successfully use the spiral approach without these elaborations, however. When there are greatly differing experience bases that are brought to the project, then the added level of elaboration, such as has been accumulated over the years for document-driven approaches, are important in ensuring consistent interpretation and use of the spiral approach across the project.

The spiral model is more adaptable to the full range of software project situations than the document-driven waterfall methodology or the code-driven evolutionary methodology. It is exceptionally applicable to very large, complex, ambitious software systems.

[3] Fixed-price contracts are ones in which the contractor is obligated to deliver a specific product at a fixed price.

RE-ENGINEERING PROJECTS

One of the driving forces for change today is the prevalent managerial belief that requires processes for product development, marketing, production, distribution and field service to be continually innovated and improved in order to maintain competitive parity with other players in the market place. Today, the focus is on processes, particularly those that embrace core competencies. Doing so has enabled firms to improve quality, lower cost and decrease time-to-market. Of these three dimensions of competition there has been an emphasis on the latter, time-to-market.

For more than ten years, such consultants as Michael Hammer, James Champy, David Taylor, Thomas Davenport and James Harrington have been advocating a focus upon processes, rather than resources or organizations per se. The major themes have been to 1) eliminate non-value adding steps, 2) simplify processes, 3) aggregate steps and 4) eliminate or reduce handoffs. The processes are made simple by making the work performed within them more complex. Thus, the work has been enlarged vertically and horizontally as workers have been empowered to make decisions for themselves. IT has played a key role in facilitating these re-engineered projects by permitting information to be transmitted electronically and hence instantaneously to as many different locations as it is needed. The end-result is there is rampant change in a firm's processes and in the applications that support these processes. The steps actually involved in any re-engineering project are indicated below.

1. Determine measures of performance
2. Install measures of performance
3. Delineate entire existing process in all its gory detail
4. Perform process value analysis and activity-based costing
5. Benchmark processes by comparison with other processes
6. Design re-invented process
7. Simulate re-invented process
8. Prepare report with recommendations
9. Upgrade the software applications that support the re-engineered process
10. Install re-invented process
11. Measure improvements

Whenever a process is re-engineered, its supporting software application must also be either reconfigured or replaced. This is typically the bottleneck (most time consuming initiative) in any re-engineering implementation.

SYSTEM INTEGRATION PROJECTS

This is the project category into which the enterprise resource planning (ERP) initiative largely belongs. Using vendors like SAP, Oracle, IBM and Microsoft, firms have been replacing their old financial accounting software with COTSS from these vendors. Moreover, the old software suffered from a myriad of problems that were largely due to the nature of the architecture inherent within it. These problems included the absence of data integration, the lack of a windows GUI (Graphical User Interface), and the mainframe-intensive usage. But the most pressing problem with the old software was the requirement to route modifications through the centralized MIS shop, where lead times were out to 35 or more months. In that lengthy period, requirements can change several times over. By replacing the old software with new software that has a distributed computing architecture, several problems get solved at once. Because of the distributed computing architecture, data integration is fully supported. Also because of the distributed computing architecture, ownership of the applications can be transferred from the centralized MIS shop to the user groups, the process owners whose activities are supported by the application. Now when the process changes, when the requirements change, the supporting

application can be changed immediately by MIS professionals that sit "shoulder-to-shoulder" with the end-users. Let's examine what steps are involved in the execution of these ERP projects.

Before anything else happens, a project manager should be chosen by upper-level management. The PM (Project Manager) should convene a group of interested and affected professionals—end-users, MIS, finance, accounting, etc. A Joint Requirements Definition (JRD) session should be held off-site to conceptualize the project and define requirements. The PM should create a project plan that specifies duration, cost and schedule. Among the activities to be included in the project would be the following:

1. Convene JRD session
2. Define requirements
3. Conceptualize the project
4. Put out RFP to vendors
5. Grade proposals and select vendors
6. Purchase the software
7. Install the COTSS[4]
8. Perform integration testing
9. Fine-tune the software to the user's specific needs
10. Allow users to conduct acceptance testing
11. Conduct a final overall system-wide test
12. Commence pilot conversion
13. Cut-over to new system

SYSTEM MAINTENANCE PROJECTS

The use of the term "maintenance" is a bit of a misnomer in MIS circles. When used in conjunction with hardware, maintenance connotes the need to grease certain parts or replace parts that wear out with time. Such is never the case with software. However, there are occasions in which the software requires enhancement, adaptation or revision to accommodate the needs of the process that it supports, such process having just undergone a change. There are also occasions in which a new "bug" appears as a result of something different that is being done with the software. Further, adaptations to the software are necessary to insure compatible interfaces with other software systems that have also undergone revision. When used in conjunction with software, maintenance connotes either enhancement or debugging.

Industry observers Fried (1995), McClure (1992) and others are quick to point out that at least 80% of MIS budgets get consumed with maintenance projects. Yet most MIS curricula have little or nothing to say about this most important type of IT project.

McClure (1992) has created a most thorough discussion of software maintenance. She identified three sources of maintenance—software failures due to bugs, environmental changes due to changes in software or hardware with which the software interacts, and user enhancement requests. These three sources of maintenance give rise to three types of maintenance—corrective maintenance in which work is performed to overcome processing, performance or implementation failures; adaptive maintenance in which work is performed to bring the software current with its environment; and perfective maintenance in which work is performed to enhance performance, add functionality or improve maintainability.

Because the old software landscapes were not very maintainable, maintenance traditionally has been a high-risk venture due to the difficulty of understanding the implications of a change. Because of poor structure, convoluted logic, meaningless program names, absence of standards, absence of definitions, absence of documentation, maintaining legacy software code is

[4] COTSS—commercial off-the-shelf software

extremely difficult and risky business. Weinberg (1983) reported that the top ten most expensive programming errors were all maintenance errors. The top three, which involved changing only a single line of code, cost their companies $1.6 billion, $900 million, and $245 million. The cavalier manner in which these "projects" were managed was the problem in each case. Assignments were given verbally and the assignee was not required to develop a formal plan for effecting the change. In particular, no formal testing of the change was instituted prior to operationalizing the change. In lieu of this anomaly, the following methodology for maintenance is proposed:

1. Write down the purpose of effecting the change.
2. Document how the change will be accomplished.
3. Take the software system off line.
4. Make the change.
5. Test ***ALL*** paths affected by the change.
6. Submit the modified software system to a beta testing group for analysis.
7. Once all parties are satisfied that the modified system works within all of its contexts, change-over to the modified system.

COMPONENT INTEGRATION PROJECTS

Component integration projects begin with prefabricated modules and assemble the final product from there through a comprehensive program of integration and testing. The idea of building significant computer programs quickly from existing software modules or **components**, has attracted widespread attention among researchers and developers. Notwithstanding their potential for streamlining the entire software development process, a relative new approach to software development has evolved, called **component-based software development** or CBSD. CBSD assembles small or large components together to produce a software system.

Component integration projects are based on the principle of reuse. Reuse, as a software engineering concept has been around for more than ten years. Basically, **reuse** is the re-utilization of already-developed software. This happens almost every time a new distributed computing application is developed because to do so is to reuse a pre-existent data management component. That database engine is just as much a part of the application as if it were being created from scratch. However, to re-create it from scratch would take hundreds (perhaps thousands) of man-months.

Today, there are literally thousands of commercial off-the-shelf software components already available for reuse. Young IT project professionals wishing to accelerate their careers should become familiar with these components. Component libraries in OCX, Active-X, JAVA beans, and other formats are compatible with almost any GUI 4GL development environment, including Visual BasicTM, and many others.

Any CBSD methodology must include the following coarse-grained steps:

1. Define requirements
2. Search for re-usable components
3. Evaluate candidate components
4. Make component selection decisions
5. Integrate components, possibly using regression testing
6. Perform system testing
7. Perform conversion and cut/over

The search for re-usable components takes place both within and without the organization. Mature organizations will maintain a repository of re-usable components that are catalogued and documented. Searches without the organization will take place among the vendors' commercially-available software components. In the ERP space, for example, upwards

to a dozen different vendors of a financial module or a marketing module are readily available. The ability of firms to mix and match their component choices from a variety of different vendors is becoming increasingly commonplace. See Supplement 4.2 for additional detail.

RAPID APPLICATION DEVELOPMENT PROJECTS

Assuming the requirements are right, the coding is correct, the testing is thorough, and the documentation is accurate and fully implemented, faster is almost always better than slower; sooner is almost always better than later. Companies are forever looking for ways to get things done quicker, faster. Product lifetimes are now much shorter; new product development cycles must be shorter as well. Process re-inventions happen more frequently and should therefore take less time to get implemented. The bottleneck in these process re-inventions is always the supporting software. Enter **RAD—Rapid Application Development**—a methodology for developing software systems rapidly. The defining principal of RAD is the following: 80% of the functionality can be obtained in about 20% of the time. It may be pointless to wait until the product is 100% completed to release the product. Instead, quickly build 80% of the solution and iteratively improve the system until it meets all of the users needs. It has been said that there are just two basic categories of methodologies for software development—RAD and SAD (Slow Application Development). RAD is better because if the product development takes too long, the requirements will change. RAD, to be really successful, should, like component-based development and distributed computing development, extensively utilize reuse.

A methodology for RAD is discussed in this section. Using the newer development tools, the methodology presented here is moderately structured, without going all the way to very complex waterfall or object-oriented development methods. If time is not a concern, the standard waterfall or spiral methodology is the right process. But often, both are simply too formal, too complex, too expensive, or too slow.

It is possible to fully satisfy business requirements while some operational requirements are not satisfied. Moreover, the acceptability of a system can be assessed using the minimum, rather than total, useful set of requirements. What this all leads to is producing a not-yet perfect, but working product in a short amount of time and then building onto that project until all requirements are met.

In general, RAD can be broken down into four stages: requirements definition, user design, construction, and implementation. Each stage involves a high amount of user involvement to guarantee satisfaction. It is important to note that because of the lack of an official RAD standard each of these stages are often modified according to different company styles.

The requirements definition phase is an intensive study of the current system and objectives of the proposed system. First, it is necessary to understand the current system, including problems, strong points, and user opinions. This is necessary to locate reusable pieces, avoid recreating problems, and to get a feel of what the system needs to do. Next, using a Joint Requirements Definition (JRD) session involving users, executives and IS professionals, create an outline of the system area and definition of scope. This is also the beginning of creating requirements specifications. Finally, the requirements are finalized, a tentative schedule and budget are created.

The user design phase should produce a systems area model, a systems design outline and an implementation plan. Here, Joint Application Design (JAD) sessions bring the users into decisions such as design outline, user screens, and implementation planning. The group meets to create and refine a system design outline. Next, screens are designed to user specifications. All reusable pieces are discovered. Finally, an implementation plan is produced that is approved for the project.

The construction phase involves finalizing the design outline and actually creating the design. Here, the user is involved in every step. After the design is fully finalized, the group moves into Rapid Iterative Prototyping (RIP). The user has the opportunity within RIP to review and criticize iterative prototypes until one is developed that can meet the requirements defined in the first stage. This is also where planning for the implementation stage takes place. Ideas such as training, contingency and transition need to be addressed at this stage.

The final phase of RAD is implementation. This involves the transition from the old to the new system. User training, data conversion, installation and responding to emergencies are very important to this phase. Once again, users are present throughout each phase. This phase, along with the entire development, does not end until user acceptance is granted.

Several years ago, a group of experts from various companies came together to discuss the development of a public domain RAD method. Over the course of several more meetings, this Consortium released the first version setting a standard RAD methodology known as Dynamic Systems Development Method (DSDM). The objectives of this group included the creation, promotion and training of DSDM and the encouragement of membership into the DSDM Consortium. This methodology is the most widely recognized method in the United Kingdom and is quickly being accepted on a worldwide basis.

The Consortium published nine principles they felt were the foundation of DSDM:

1. Active user involvement is imperative.
2. DSDM teams must be empowered to make decisions.
3. The focus is on frequent delivery of products.
4. Fitness for business purpose is the essential criterion for acceptance of deliverables.
5. Iterative and incremental development is necessary to converge on an accurate business solution.
6. All changes during development are reversible.
7. Requirements are baselined at a high level.
8. Testing is integrated throughout the life cycle.
9. A collaborative and co-operative approach between all stakeholders is essential.

Box 4.3: DuPont and RAD

DuPont, a chemical company, produces thousands of products for such industries as agriculture, plastics, manufacturing and transportation. DuPont's experience was the business units were taking about six months to develop new products, while the information systems units were taking about two years to support the new products with software applications. What DuPont did was come up with a rapid development methodology that speeded up the development process by a factor of four. This methodology was based on calendar-driven techniques, inflexible timeboxes, fixed periods of time, typically three months, in which all of the details of the application had to be completed. What DuPont found was that, by using automated tools and calendar-driven development, they could get the applications out the door much faster and exactly within the time period specified. They are trading time for functionality, but they are getting the applications out. The development team is authorized to drop low priority functionality if they cannot meet the timebox due date for delivery. They have done hundreds of such projects successfully, getting the applications out the door even though not all the functionality was there.

This requires intensive project management. Structure and discipline is imposed on the development process. It responds to the business imperative to get applications out quickly. It does entail making sure the applications are maintainable. But the benefit is that the application is in the end-user's hands at an early stage, where it can be evaluated for effectiveness and fit. This approach is about twice as fast as using the same tool in non calendar-driven methodologies. The application has to meet the users' specification at the time of cut-over.

DuPont was one of the first firms to use timeboxes to deliver functionality and the results were impressive. Initially, timeboxes were three months in duration, with a finalized application being delivered in six months. This gave the project team three months to deliver the initial functionality and three more months to test and refine that functionality.

DSDM is an iterative and incremental development method. In classic development, requirements are fixed and resources and time vary. DSDM stresses the exact opposite—time and resources are fixed while requirements are allowed to change as the project progresses.

In order to handle flexible requirements within fixed times, DSDM uses a mechanism known as timeboxing. For the overall project, there is a fixed due date that creates a timebox for the entire project. DSDM then breaks that into smaller timeboxes, each with a prioritized set of requirements to be fulfilled. Because the timebox has an immovable deadline, prioritization of requirements is essential to a successful product. Each timebox contains all necessary reviews and tests and will end with the production of something visible, that allows for progress and quality to be assessed.

DSDM created a standard that can and has been used successfully by many companies in both public and private sectors. The Consortium constantly publishes experiments and tests on the application of new ideas, concepts and techniques. Through their work and research, they have created a standard that can be taught, tested and certified to ensure quality RAD projects.

A switch to RAD methodologies brings many obvious advantages. Because of the reuse, the timeboxing, and the flexible implementation of required functionality, a substantial savings of time and money can be realized. RAD allows for early visibility, allowing the user to better explain what is needed. Users are more involved because they are now represented on the team at all times. And finally, manual coding is dramatically decreased through reuse and utilization of CASE (Computer-Aided Software Engineering) tools.

As with all methodologies, these advantages must be evaluated in relation to a list of disadvantages. While acceptance of DSDM is growing, no universal standard of RAD has been accepted. Because of this and the youth of the method, progress and scientific precision are more difficult to gauge. Reuse and automatic coding software can lower efficiency of the code, increase costs, decrease creativity in look and feel, and include unwanted features. Because time does not permit a thorough analysis of the requirements, there is a likelihood that errors in the requirements will be made.

With the older development processes, the business users needs would change in the development interim. Traditionally, MIS took great pains to thoroughly understand the business user's needs and spent lots of time up front analyzing those needs before ever doing any design, much less writing any code. As a result, perhaps as many as half of traditionally developed applications never go into production. The older development procedures simply don't work in time-compressed environments. Most end users don't know what they want until they see it anyway. This is really true in data warehousing; in data warehousing, if users can specify even half of the requirements up front, they are doing extremely well. The other 50% have to come from discovery. What we are looking for is a process that is so flexible that the requirements can change all during the development process, yet we can still meet the needs of the end-users at the time of cut-over.

Team sizes used in IT construction are typically between three and five individuals. The teams should be kept fairly small. Developers should not use a separate prototyping language; the same tool should be used for both prototyping and development. Time and cost estimates are estimated the first time around. Developers should count up the number of screens that have to be implemented. A good team can do three screens a day, whereas a low-experience team can do about one screen a day, with all the supporting logic.

The first couple of iterations (spirals) are really prototypes, but by the fifth iteration, developers have arrived at the final version of the program. There are some risks, however. Developers should not define all the business procedures up front. They should start and discover the business rules as they go along. Once the users start working with the prototype, they will get a better understanding of what the application should do. There should be very low cohesion between the various MDI frames that are being developed. Cross coupling and connectivity between MDI frames should be avoided, because it does entail some rework of the earlier prototypes. But it will happen. Functions are organized into three categories, with priorities of one, two or three. Priority three functions can be dropped if there is not enough time. Scope creep is a very real problem in this discovery mode approach, rework is a problem, changing requirements is a problem and so forth. The changes may be so bad that developers have to go all the way back and reconvene the JAD workshops to redo the requirements planning.

Teams are in competition with each other. The teams that do the best are the ones that reuse as many of the base classes as possible. Hundreds of classes of libraries come with Visual Studio and Java, but OCX's (Visual Basic) can be used as well, particularly in Internet applications. Developers will build their applications as OLE-enabled components. Partitioned applications—presentation, business logic, and data domains—is what gets distributed according to the architecture that is selected.

What are the steps involved in planning this? First, convene a JAD workshop in which all the important people are brought together and get people to agree, using the facilitated framework of a JAD workshop. Once you've made those decisions, you can procure the hardware, then provide training (5 days), then instruction in the base objects that come with the tool (two days), and finally learn the methodology. All of this might take two weeks.

Next comes the pilot project. The questions to be asked are: did the platform work, did the methodology work, etc. This takes six weeks to three months and brings five people up to speed on the project. Then bring in a consultant and provide JIT training on the use of that tool, to provide specific training on the tool (OLE automation, object classes, technology transfer). At the end of the pilot project there should be at least five trained people who will then train more people. Each of these five people will now be a project leader on a RAD project. Technology concepts are now transferring very rapidly. Some training in object-oriented analysis may also be desirable in which the team is trained in Rumbaugh and/or Jacobsen concepts. This is very different from traditional object-oriented or conventional software development and so the end-users must be a part of the team as the end-user contributions are absolutely vital to the success of the project. The advocate for this project on the end-user side must be a strong supporter. If pseudo-end-users can navigate the application, understand its functionality, then the form may be correct.

The project manager should be someone who is experienced with this type of development. The JAD facilitator uses applied psychology to get consensus. The data modeling person must listen to the end-users as they describe the functionality that they want and extract out of that the entities that are required. The entities must then be classified as new or existing, but creating new entities are rare. Again reusing existing entities or subclasses of them is what should happen. This is building the data model bottom up rather than taking an information engineering approach which builds applications top down and rarely works because it is too long and costly. The data modeler gradually accumulates data models until a data model for a division of the business and ultimately the entire enterprise, working bottom up is arrived at. Data models are built right up front during the JAD workshop, the JRP[5] session. When developers actually get into the application, the data model is usually pretty stable.

Team members must understand more than the stove pipe that they are working on so they won't end up building just a stove pipe. The business rules will be discovered as the project

[5] JRP—Joint Requirements Planning, a facilitated meeting often held offsite that lasts for up to three days

moves along. The IRS tried to do a top-down ER model and then a bottom-up ER model and both failed. Build a data model for a single data mart. Each time the data model is expanded, add 25 or 30 new entities that represent that part of the business.

The SWAD (SoftWare Application Development) team is then selected as a creative construct of psychology and technology. Team members should be enthusiastic, team players, self starters, problem solvers. They are extremely productive, they immensely enjoy the work environment. There must also be adequate reward for teams and compensation for them should be at a higher rate to get these teams to work at a higher level. If they are a great deal more productive, then they can be paid a great deal more.

The SWAD team must be trained to say no so that there isn't scope creep. Exactly how is the change process controlled? Developers (SWAD members) must negotiate with the end-users, so if they want more functionality, they must give up something. This is a fixed-bid process. The SWAD team knows what reusable components are, what OCX's they will use, and they bid the project based on what they know they will reuse. But if the end-users want to add a new function, another must be deleted.

There can be instances in which the timebox must be aborted. But the end-users will have to eat the cost, because they forced the timebox to be exceeded. The end-users are always asking for more. It is possible to be 90% done for months and months. If there is date inflexibility; then everybody, end-users, developers all are working to achieve completion by a specific date.

What is the purpose of the pilot project that is to be done? Who is the end-user? What are the requirements? At Texaco, an end-user group came to IS and said we want a training system and we need it now. IS said it would take 2 years using conventional methodology, but they also said there is new technology out there that is untested and how would you like to be a test case to use this new technology with the result that your application is built in six months? It happened.

The team specifies the priority of each business function. If the timebox cannot be made, the team is authorized to drop functionality. Time is more important than functionality. Basically, the goal is to get the application living quickly, learn from it and then enhance it. It is better to do it in two three-month timeboxes and deliver a significant amount of functionality every three months. When the second half of it is delivered at the end of the second three months, the users will have a much greater understanding of their requirements.

Developers must define standards for their specific site. As developers spiral their way down from analysis to the design phase, what happens in design? Mostly prototyping; high-level prototypes, user interfaces and GUI's further and further down. Developers should immediately test these on the end-user as quickly as possible. If the high-level interfaces are not acceptable to the user in terms of intuitive understanding, they must be redesigned. As developers work their way down, they discover the business rules and this is very different from the specification process that has been used for years and years. The approach described here is risky. The presumption is that, with close user involvement, developers can define each one of these detailed business rules on the fly as they work their way further and further and down. This may not work without some up-front detailed analysis. Close end-user participation is a must. Expect the end-users to get a flash of insight as to how the application could be made substantially better in terms of screen organization. If you were doing this in C or Cobol, this is the last thing you want. Complete the prototype, and continue to improve estimates as to how much time is going to be required to complete the work. Part of the planning for cut-over is to do all of the testing. The cut-over plan must also involve how the database gets reorganized for the new application. Test each screen using the user support team. Every day or two developers should deliver completed functionality and test it on the users. There should be version control, so all versions of the software can be tracked in terms of who made what changes and go back to an old version if necessary. Every few days, there should be significant functionality released for testing. Each

member of a three-member team can deliver from one to three screens a day. A significant release should be forthcoming from each developer each week. Each developer's assigned tasks should be broken down into chunks of functionality that must be delivered by a certain date.

A very important part of the construction phase is the development of documentation. Just-in-time training aids should be developed rather than formal training courses. The documentation people have to learn the same tool as the IS staff has learned. If IS is building the application with **Visual Basic**, then the documentation people have to learn **Visual Basic**. They also have to learn the OCX's that support multimedia so they can develop a JIT training module that can be activated by a mouse click. The user, if unsure as to how to proceed, can then click on the JIT training module icon and a virtual instructor pops up that will lead them step-by-step through the process. They can then interact with the module in an off-line basis with a test database. When they are convinced that they know that particular function, they can then return back to the module implementing that function. The application training and documentation people must create multimedia training aids that operate along with the application and they must do this four times faster than they are currently doing it. How?? What they need to do is to go through the same training in pilot project activities as the IS people and learn how to do JIT training. By eliminating formal training, project managers can cut their lifecycle costs in half. Allowing end-users to learn on their own in a JIT training environment will substantially cut lifecycle costs. The same thing applies to electronic documentation. Implement it as part of the Microsoft help facility. Both the training people and the documentation people must learn the new technological tool that IS is learning. Third party products to help with Microsoft documentation is available, and there are hundreds of multimedia OCX's that support the development of JIT training.

At the end of the construction phase, a GO/NO-GO decision is made as to whether to cut-over to production. If the decision is YES, then cut-over begins and it involves preparation. Prepare the team, deliver the final training, prepare for cultural change because some end-users may not have seen any part of this application, prepare the database, prepare for cut-over to the database. Some of this is quite fast. What then is the actual experience with the application. Continue to enhance the application by the SWAD team outside the timebox. During this enhancement phase, there is not as much time pressure. But, there must be executive sponsorship. Meet the strategic business need and create business value.

At Texaco, the Board had a very serious business problem, in which they needed to get applications out the door quicker. Time-to-market is very important, yet we must be realistic. The involvement of the end-users is one of the most important critical success factors. The JAD workshops are important and the timebox is important. Ways this can fail are when there is a lack of top management support, or there is not strong end-user involvement. If the end users don't show up for the JAD workshops, they must be canceled. At least two end-users should be available two days a week for each SWAD team. More than just one end-user is needed because of the requirement for diversity in terms of business experience. The JAD workshops that are part of the analysis phase are only two hours long. When convening the JRP, the project manager must have a model on the table that focuses the attention of the workshop participants.

AGILE SOFTWARE DEVELOPMENT

Agile Software Development is a conceptual framework for software development that promotes development iterations throughout the lifecycle of the project. Agile approaches evolved out of RAD concepts and are a response to the need to accommodate rapidly changing requirements.

There are many agile development methods; most minimize risk by developing software in short amounts of time. Software developed during one unit of time is referred to as an iteration, which typically lasts from one to four weeks. Each iteration passes through a full software development cycle: including planning, requirements analysis, design, coding, testing, and

documentation. An iteration may not add enough functionality to warrant releasing the product to market but the goal is to have an available release (without bugs) at the end of each iteration. At the end of each iteration, the team re-evaluates project priorities.

Agile methods emphasize face-to-face communication over written documents. Most agile teams are located in a single open office sometimes referred to as a scrum. At a minimum, this includes programmers and their "customers" (customers define the product; they may be product managers, a business analyst, or the clients). The office may include software testers, interaction designers, technical writers, and managers.

Agile methods also emphasize working software as the primary measure of progress. Combined with the preference for face-to-face communication, agile methods produce very little written documentation relative to other methods. While this has resulted in criticism of agile methods as being undisciplined, it has resulted in faster product releases.

The modern definition of agile software development evolved in the mid-1990s as part of a reaction against "heavyweight" methods, as typified by a heavily regulated, regimented, micro-managed use of the waterfall model of development. The processes originating from this use of the waterfall model were seen as bureaucratic, slow, demeaning, and inconsistent with the ways that software developers actually perform effective work. A case can be made that agile and iterative development methods are a return to development practice seen early in the history of software development. Initially, agile methods were called "lightweight methods." In 2001, prominent members of the community met at Snowbird, Utah, and adopted the name "agile methods." Later, some of these people formed The Agile Alliance, a non-profit organization that promotes agile development. They created the Agile Manifesto, widely regarded as the canonical definition of agile development and accompanying agile principles.

A number of methods similar to Agile were created prior to 2000. An adaptive software development process was introduced in a paper by Edmonds (1974). Notable earlier methods include Scrum (1986), Crystal Clear, Extreme Programming (1996), Adaptive Software Development, Feature Driven Development, and Dynamic Systems Development Method (DSDM) (1995).

Kent Beck created Extreme Programming (usually abbreviated as "XP") in 1996 as a way to rescue the struggling Chrysler Comprehensive Compensation (C3) project. While Chrysler eventually canceled that project, the method was refined by Ron Jeffries' full-time XP coaching, public discussion on Ward Cunningham's Portland Pattern Repository and further work by Beck, including a book in 1999. Elements of Extreme Programming appear to be based on Scrum and Ward Cunningham's Episodes pattern language.

The principles behind the Agile Manifesto include the following concepts. Customer satisfaction by rapid, continuous delivery of useful software. Working software is delivered frequently (weeks rather than months). Working software is the principal measure of progress. Even late changes in requirements are welcomed. Close, daily cooperation between business people and developers is strongly encouraged. Face-to-face conversation is the best form of communication (Co-location). Projects are built around motivated individuals, who should be trusted. Continuous attention to technical excellence and good design is required. Simplicity is a hallmark. Self organizing teams are always used. Regular adaptation to changing circumstances is accommodated. The manifesto spawned a movement in the software industry known as agile software development.

Agile methods are sometimes characterized as being at the opposite end of the spectrum from "plan-driven" or "disciplined" methods. This distinction is misleading, as it implies that agile methods are "unplanned" or "undisciplined." A more accurate distinction is that methods exist on a continuum from "adaptive" to "predictive." Agile methods lie on the "adaptive" side of this continuum.

Adaptive methods focus on adapting quickly to changing realities. When the needs of a project change, an adaptive team changes as well. An adaptive team will have difficulty describing exactly what will happen in the future. The further away a date is, the more vague an adaptive method will be about what will happen on that date. An adaptive team can report exactly what tasks are being done next week, but only which features are planned for next month. When asked about a release six months from now, an adaptive team may only be able to report the mission statement for the release, or a statement of expected value vs. cost.

Predictive methods, in contrast, focus on planning the future in detail. A predictive team can report exactly what features and tasks are planned for the entire length of the development process. Predictive teams have difficulty changing direction. The plan is typically optimized for the original destination and changing direction can cause completed work to be thrown away and done over differently. Predictive teams will often institute a change control board to ensure that only the most valuable changes are considered.

Agile methods have much in common with the "Rapid Application Development" techniques from the 1980/90s as espoused by James Martin and others.

Most agile methods share other iterative and incremental development methods' emphasis on building releasable software in short time periods. Agile development differs from other development models: in this model time periods are measured in weeks rather than months and work is performed in a highly collaborative manner. Most agile methods also differ by treating their time period as a strict timebox.

Agile development has little in common with the waterfall model. As of 2008, the waterfall model is still in common use. The waterfall model is the most predictive of the methods, stepping through requirements capture, analysis, design, coding, and testing in a strict, pre-planned sequence. Progress is generally measured in terms of deliverable artifacts: requirement specifications, design documents, test plans, code reviews and the like.

The main problem with the waterfall model is the inflexible division of a project into separate stages, so that commitments are made early on, and it is difficult to react to changes in requirements. Iterations are expensive. This means that the waterfall model is likely to be unsuitable if requirements are not well understood or are likely to change in the course of the project. Furthermore, testing is curtailed until the very end of the project.

Agile methods, in contrast, produce completely developed and tested features (but a very small subset of the whole) every few weeks or months. The emphasis is on obtaining the smallest workable piece of functionality to deliver business value early, and continually improving it/adding further functionality throughout the life of the project.

In this respect, agile critics incorrectly assert that these features are not placed in context of the overall project, concluding that, if the sponsors of the project are concerned about completing certain goals with a defined timeline or budget, agile may not be appropriate. Adaptations show how agile methods are augmented to produce and continuously improve a strategic plan.

Some agile teams use the waterfall model on a small scale, repeating the entire waterfall cycle in every iteration. Other teams, most notably Extreme Programming teams, work on activities simultaneously.

As with all development methods, the skill and experience of the users determine the degree of success and/or abuse of such activity. The more rigid controls systematically embedded within a process offer stronger levels of accountability of the users. The degradation of well-intended procedures can lead to activities often categorized as cowboy coding.

There is little if any consensus on what types of software projects are best suited for agile methodologies. Many large organizations have difficulty bridging the gap between a more traditional waterfall methodology and an agile one.

Large scale agile software development remains an active research area.

Some things that can negatively impact the success of an agile project are:

- Large scale development efforts (>20 developers), though scaling strategies and evidence to the contrary have been described.
- Distributed development efforts (non-co-located teams). Strategies have been described in Bridging the Distance and Using an Agile Software Process with Offshore Development.
- Command-and-control company cultures.
- Forcing an agile process on a development team.

Several successful large scale agile projects have been documented. BP has had several hundred developers situated in the UK, Ireland and India working collaboratively on projects and using Agile methods. While questions undoubtedly still arise about the suitability of some Agile methods to certain project types, it would appear that scale or geography, by themselves, are not necessarily barriers to success.

Barry Boehm and Richard Turner suggest that risk analysis be used to choose between adaptive ("agile") and predictive ("plan-driven") methods. The authors suggest that each side of the continuum has its own home ground. For agile software development, the home ground is a culture that thrives on chaos, low criticality, a small number of senior developers are used, and requirements change very often. For plan-driven methods, the home ground is high criticality, junior developers, requirements don't change too often, a large number of developers, and a culture that demands order.

PROJECT MANAGEMENT WITH SCRUM

Scrum is a type of agile development methodology. As such it is most appropriate for the home ground of agile development as discussed above. The Scrum approach to PM has been likened to a game of rugby rather than a relay race, which is what traditional PM is likened to. In rugby, the whole team tries to go the distance as a unit concurrently rather than just the last runner crossing the finish line with the baton in hand. DeGrace and Stahl, in their book *Wicked Problems, Righteous Solutions* (1991), referred to the approach as Scrum, a rugby term. In the early 1990s, Ken Schwaber, used an approach that led to Scrum at his company. Scrum has evolved out of the need to accommodate rapidly changing requirements.

Scrum is a process skeleton that includes a set of practices and predefined roles. There are two types of roles used in connection with Scrum, those who are committed are called ‘pigs’ and those who are involved who are called ‘chickens.’ Stakeholders are considered chickens whereas the project team and Scrum master (project manager) are called ‘pigs.’ Scrum consists of a series of sprints. Each sprint is a period of 15 to 30 days, during which the team creates a usable module of software. Scrum is considered ‘easy to learn’ and doesn’t require a lot of training to start using it. Sprint periods of 30 days are similar to the monthly timeboxes used in RAD. Each day during the sprint, a project status meeting occurs. This is called a scrum. The procedure for a scrum is the following:

1) the meeting starts precisely on time with team-decided punishments for tardiness
2) all are welcome, but only “pigs” may speak
3) the meeting is timeboxed at fifteen minutes regardless of the team’s size

4) all attendees should stand
5) the meeting should happen at the same location and same time every day

In summary, scrum is an agile process to manage and control development work. Scrum is a wrapper for existing software engineering practices. Scrum is a team-based approach to iteratively, incrementally develop systems and products when requirements are rapidly changing. Scrum is a process that controls the chaos of conflicting interests and needs. Scrum is a way to improve communications and maximize co-operation. Scrum is a way to detect and cause the removal of anything that gets in the way of developing and delivering products. Scrum is a way to maximize productivity. Scrum is scalable from single projects to entire organizations. Scrum has controlled and organized development and implementation for multiple interrelated products and projects with over a thousand developers and implementers. Scrum is a way for everyone to feel good about their job, their contributions, and that they have done the very best they possibly could.

SELECTION METHODLOGY: FOR SOFTWARE, PROCESSES AND PROJECTS

Selection projects are usually short-lived in terms of duration—six months or less. Even so, a specific methodology should be followed to insure success in the end. Information technologists are always involved in selection decisions with end-users and managers. Selection decisions must be made with respect to software, processes and projects, but not necessarily in that order. Process selection comes first. There are three types of decisions here—what processes to innovate, which ones to re-design, and which to merely improve. But relative to a specific process, a selection methodology can be applied to determine the exact steps to be applied. With respect to projects, a centralized project management department allocates a limited budget to a list of possible projects. Not all of the proposed projects can be resourced or funded. A project selection decision must be made. Once these decisions must be made, attention can then turn to software. Table 4.3 lists the basic steps for acquisition and installation of software.

Table 4.3: Ten Simple Steps to Software Acquisition/Installation

1. **Define the application to be computerized.** What are the inputs, the outputs? What makes your requirements different from those of other users, if the application is commonplace?
2. **Develop a list of what software is available to support the application.** To do this you have to know where to go to shop for software.
3. **Gather information about available packages.** One source of information about computer software is the trade literature which frequently review new software. Another is the Internet.
4. **Narrow the list of possible choices down to less than a half dozen.** For example, there are about 20 major ERP vendors. A thorough scrutiny of all of these would be impractical.
5. **Obtain hands-on demonstrations of the few remaining packages.** If the candidate programs are readily available at your local retailer, ask the dealer to demonstrate them to you and give you an opportunity to test them yourself.
6. **Of those that remain, perform a final detailed evaluation.** If possible read the operations manuals. Check the features to see if they align with your requirements. Observe the outputs and reports to ascertain if they will be functional in your application.
7. **Make a decision.** Decide which of the programs will provide all of the desired features at the lowest cost. Costs should subsume all of the potential costs of the product lifecycle. These would include purchase, modification, installation, training, updating and maintenance.
8. **Purchase the package.** The issue here is one of who to purchase the package from--local retailer or mail-order house. The tradeoffs are obviously cost versus service.
9. **Learn to use the package.** After following the vendor's installation instructions, you are ready to begin learning how to use the package. Many packages provide a tutorial to facilitate such training.
10. **Implement the package within the context of the intended application.** Use the package to create the records and files that you intended.

METHODOLOGIES FOR SYSTEM CONVERSION/CUT-OVER PROJECTS

Modern system conversion/cut-over projects can utilize many different methodologies. Methodologies for system conversion/cut-over projects are the different ways current systems are replaced with new systems. The main methodologies used today are Direct Installation, Parallel Installation, Single Location Installation, and Phased Installation. In this section, we explore each of these methodologies in more detail.

Steps to Choosing a Methodology

Organizations that implement a system conversion/cut-over project use a computer already throughout the organization to accomplish everyday tasks. The difficult challenge lies in converting from the existing system to the new one. Even if the proper system has been selected, the benefits that can be realized from the new system will be far outweighed by productivity losses of your users if the conversion is not handled properly. Not only do users have to learn how to use the new system, but the existing data must be electronically converted and/or manually re-entered into the new database.

The first thing you should do is set up a conversion team that will be responsible for all facets of implementing the new system. The project manager should have played an integral role in the system selection process, and should have a clear view of what the organization's overall goals and objectives were in choosing the new system. Other members of the team should consist of users of both the existing and new system, as well as individuals from the software development or implementation group. Implementing new a system may be a catalyst for thorough evaluation and redefinition of a firm's existing work methods. This will assure that maximum benefits will be realized from your investment in new technology.

Direct Installation

Direct Installation, also known as "Cold-turkey" or "Big Bang" installation, is an abrupt approach to installation in which the old system is turned off and the new system is turned on[6].

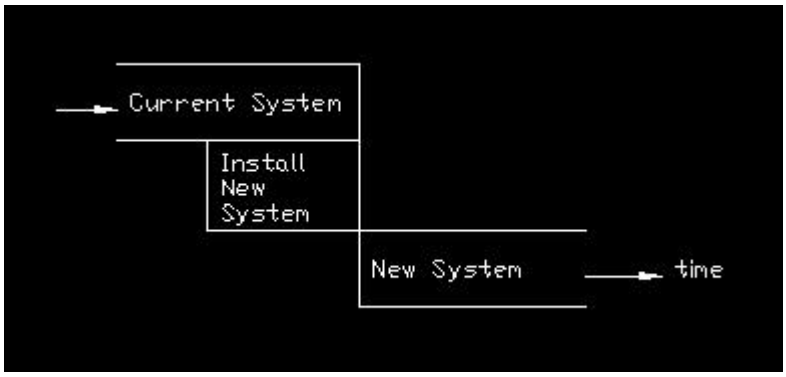

Figure 4.4: Direct, Abrupt Installation

This method entails an extended rehearsal for the actual conversion in which necessary procedures and tools for the conversion are proven, requisite changes are all identified, and skills relating to the new system are acquired. On the planned cut-over date, the new system is

[6] *Modern Systems Analysis & Design*, Second Edition, Jeffery Hoffer, Joey George, Joseph Valacich, page 776, Addison-Wesley, 1999.

installed in the target environment, with all necessary data migration and changes to procedures, organization, standards, etc., in place. If errors exist in the new system, considerable delay may occur until the old system is engaged and its database is brought up-to-date. This is a tremendous problem when dealing with a very large system. That is why Direct Installation has a reputation of being very risky. This method of conversion has its place in small systems in which it would be easier to make the old system operational if errors should occur. This method is the least expensive method due to the shortened conversion time. The conversion time is shortened by eliminating the redundancy and overhead associated with package-by-package implementation and by reducing planning requirements. Sometimes, direct Installation may be the only option for system conversion/cut-over projects because there is no way for both the current and new systems to operate at the same time. This methodology is fast and inexpensive, but has a great deal of risk.

Parallel Installation

Parallel Installation is a method for system conversion/cut-over projects that allows the old system to continue running alongside the new system until the organization is satisfied and moved over onto the new system completely. Once the new system is performing to the satisfaction of all users and management, the old system is turned off.

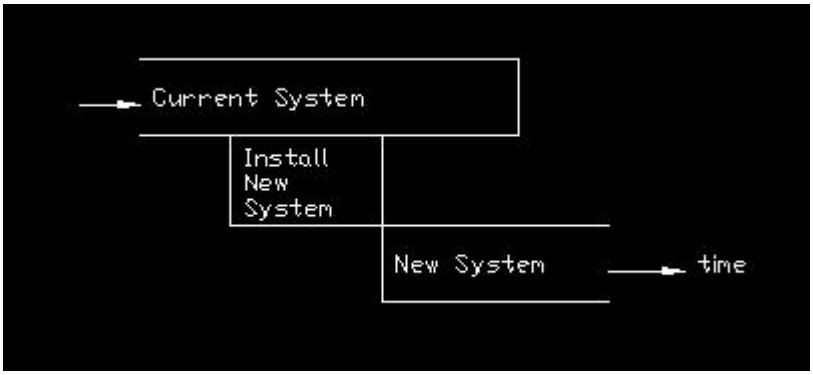

Figure 4.5: Parallel Installation

As the new system is implemented, there is little cost to the organization if errors and changes are made because the organization is still running the old system[7]. The cost is still high because both the old and new systems are running at the same time processing the same data concurrently and staff for both of the systems must be maintained. There is a great deal of redundancy in entering data into both of the systems. This method is slow and expensive, but has very little risk.

Single Location Installation

Single Location Installation, also known as “Location” or “Pilot” installation, is a method for system conversion/cut-over projects where the new system is installed in a single location first and the bugs are worked out at that location. Once the new system is determined to be satisfactory, the new system is installed throughout the organization.

[7] *Modern Systems Analysis & Design*, Second Edition, Jeffery Hoffer, Joey George, Joseph Valacich, page 776-8, Addison-Wesley, 1999.

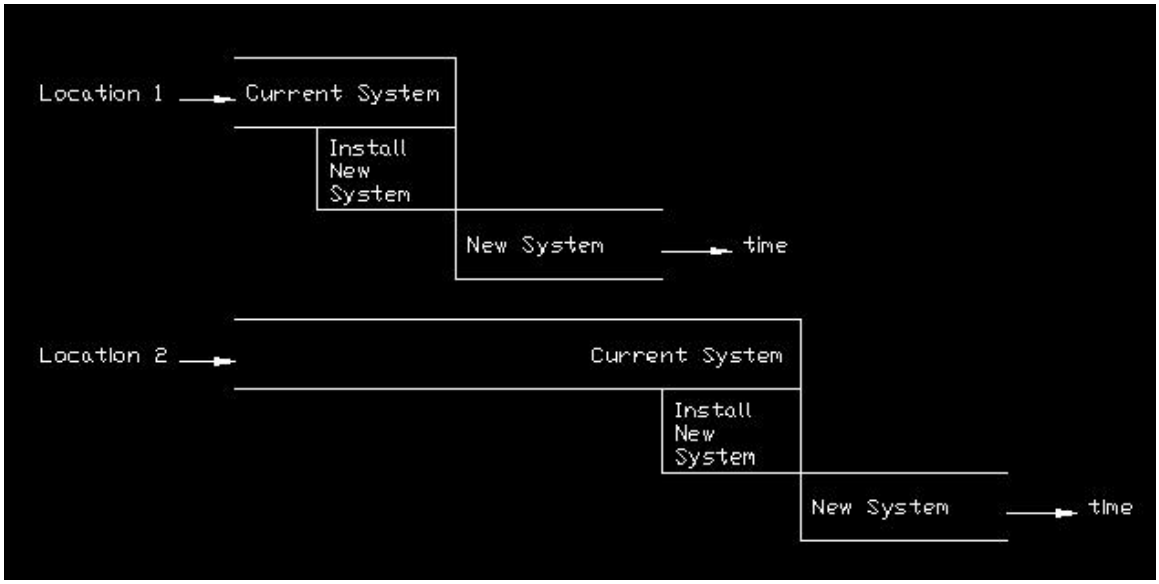

Figure 4.6: Single Location Installation

The advantage to this method is that it limits the possible damage and cost by limiting the effects to the single location[8]. The problem with this is that the single location may not find all bugs in the new system. Once the new system is deployed throughout the organization, newly discovered changes are very costly to correct. This methodology is a mixture of Direct Installation and Parallel Installation. It has moderate risk, expense, and speed to completion.

Phased Installation

Phased Installation, also known as "Staged" installation, is an incremental approach in which the new system is brought online in functional components.

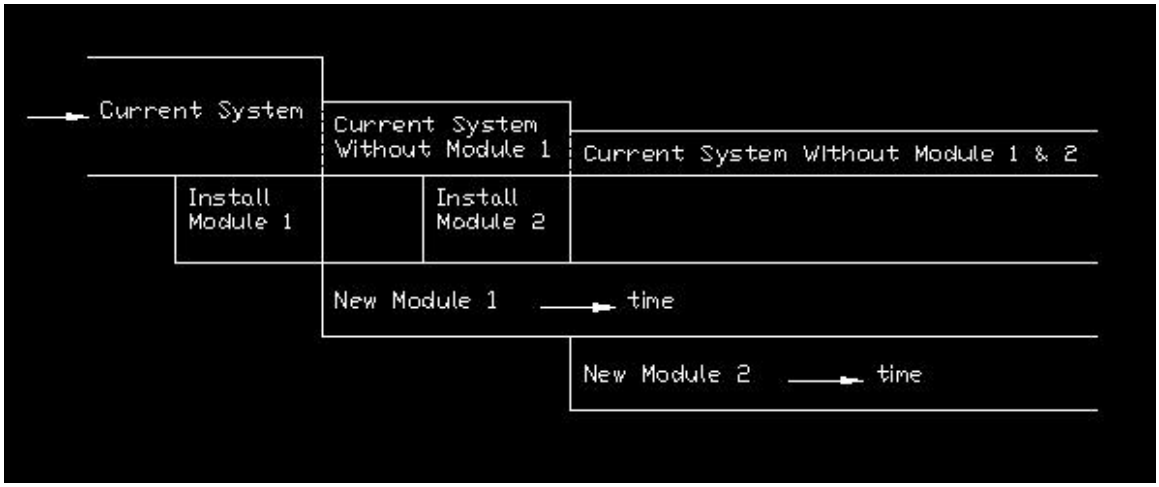

Figure 4.7: Phased Installation

[8] *Modern Systems Analysis & Design*, Second Edition, Jeffery Hoffer, Joey George, Joseph Valacich, page 776-8, Addison-Wesley, 1999.

The benefits of a phased approach include the ability to provide better and more accurate estimates of the amount of work to be undertaken, thereby avoiding large contingencies for risk management[9]. This leads to a higher quality solution more in tune with expectations and no reduction in quality due to the pressures of meeting tightening budgets and time scales[10].

Considerations to Choosing a Methodology

When the time comes for a system conversion/cut-over team to choose a method for a system conversion/cut-over project, there are three main variables to take into account: time, cost, and scope. Should the operational applications on the source system be gradually migrated to the target system, or should conversion activity take place behind the scenes and then bring forth a finished operational system in one implementation? Which of the three variables of time, cost, and scope present the most significant constraint on the system conversion/cut-over project? The system conversion/cut-over team will answer such questions during the process of planning installation. In practice the system conversion/cut-over team will rarely choose a single strategy to the exclusion of all others. Most installations will rely on the combination of two or more approaches. If the system conversion/cut-over team were to choose a Single Location Installation strategy, the team might thereafter deploy another strategy to proceed with installing the new system throughout the remaining organization. The system conversion/cut-over team could use Direct Installation, Parallel Installation, or Phased Installation, for example.

SUMMARY AND CONCLUSION

1. Discuss why process selection is important.
2. Describe what the major IT project types are.
3. Identify what steps are commonplace to all projects.
4. Understand why there is such a strong focus on shortening the duration of projects.
5. Learn how to get projects done quicker and with less cost by modification of the project process or methodology.

[9] *Modern Systems Analysis & Design*, Second Edition, Jeffery Hoffer, Joey George, Joseph Valacich, page 778-9, Addison-Wesley, 1999.
[10] Execom, http://www.execom.com.au/conversion_methodology.html.

EXERCISES

1. Define what is meant by:

   Direct installation
   Dynamic system development method
   Evolutionary development model
   Maintenance
   Parallel installation
   Phased installation
   RAD
   SAD
   Single location installation
   Spiral model
   Transform model
   Waterfall model

2. Why is the waterfall methodology an improvement over the old code-and-fix methods of software development?
3. What are some of the shortcomings of the waterfall methodology?
4. Under what circumstances would the evolutionary development methodology be preferred to other methods of software development.
5. What “tool” does the transform model assume is available?
6. Who invented the spiral model? Is the spiral model document-driven or risk-driven? Does the spiral model include any other models as special cases? What do the angular and amplitude dimensions represent? What is meant by a “spiral?”
7. When is RAD often a better approach to software development than SAD? What tools are often used for RAD development?
8. In a paragraph of 50 words, describe the Dupont model for RAD.
9. From the material in the chapter of RAD, create a list of steps required to complete a project using the RAD methodology.
10. Do you think RAD forces more parallelism in terms of product development, as compared to SAD approaches to software development? Why or why not?
11. Describe in a page or two the methodology used in a project you were involved in.
12. Discuss what is meant by Agile project management. To what kinds of projects is agile PM must appropriate?

REFERENCES

Boehm, Barry, "A Spiral Model of Software Development and Enhancement," *IEEE Computer*, Vol. 5, pp. 61-72, May 1988.

Davenport, Thomas, *Process Innovation: Reengineering Work through Information Technology*, Cambridge, Mass: Harvard Business School Press, 1993.

Greenberg, Jerry, & J.R. Lakeland, *A Methodology for Developing and Deploying Internet & Intranet Solutions*, Upper Saddle River, NJ: Prentice Hall, 1998.

Hammer, Michael & James Champy, *Reengineering the Corporation: A Manifesto for Business Revolution*, New York: Harper Business, 1993.

Hammer, Michael, and Steven Stanton, *The Reengineering Revolution: A Handbook*, New York: Harper Business, 1995.

Harrington, H. James, *Business Process Improvement: The Breakthrough Strategy for Total Quality, Productivity, and Competitiveness*, New York: McGraw Hill, 1991.

Jacobson, Ivar and Maria Ericsson, Agneta Jacobson, *The Object Advantage*, Reading, MA: Addison-Wesley Publishing Company, 1995.

Mimno, Pieter, Seminar Presentation, 1997 Database Conference, 1997.

Parnas, David Lorge and Paul C. Clements, "A Rational Design Process: How and Why to Fake it," *IEEE Transactions on Software Engineering*, V. SE-12, no. 2, pp. 251-257, 1986.

Rumbaugh, James, and Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design*, Englewood Cliffs, N.J., Prentice Hall, 1991.

Taylor, David, *Object-Oriented Technology: A Manager's Guide*, Reading, Mass: Addison Wesley, 1990.

Taylor, David, *Business Engineering with Object Technology*, New York: John Wiley, 1995.

Champy, James, *Reengineering Management: The Mandate for New Leadership*, New York: Harper Business, 1996.

McClure, Carma, *The Three R's of Software Automation: Re-engineering, Repository*, Reusability, Englewood Cliffs, N.J.: Prentice Hall, 1992.

Fried, Louis, *Managing Information Technology in Turbulent Times*, New York: Wiley-QED, 1995.

Weinburg, G. "Kill that Code," *InfoSystems*, August 1983.

SUPPLEMENT 4.1: DATA WAREHOUSING PROJECTS AND PROJECTS INVOLVING MIDDLEWARE

Data warehousing is just a special database that fits the same multi-tier environment. It uses a single data model. The data is historical, its time-stamped, its typically read-only and intended mainly for data analysts, DSS[11] analysts. Generally, a copy of the transaction database is copied into the data warehouse once every time-bucket. The source database is the existing source files, then a cleanup layer cleans up the data. Some database vendors say they have a complete database solution for data warehousing, but then they just copy the data from the source files to the target files with no cleanup. You need software like Passport, Prism, Extract which take data from the source file, clean it up (resolving any naming redundancies in file formats, antonyms, etc.) and then store it in the target data warehouse, which could be a relational database or a multidimensional OLAP[12] database.

The fourth component is the data access tools, tools used to access and analyze the data. This database is a read/write database.

Some of the middleware is greatly neglected. It's always the last thing that's thought about. There is often a lack of an architectural plan and how you would move step-by-step from where you are to where you want to be. Oracle is a typical data management choice for operation on the server. With just this environment, firms were able to reduce their training and support costs dramatically. Development tools can be used to develop custom applications. OCX's and OLE components are reusable components and this is a very important new capability that can greatly reduce development time and costs. The middleware support security and transaction management.

Dynasty and Forte do much better. Forte has a runtime manager that looks at the loading of the servers and moves applications from one to another to create a more balanced work load. These work especially well with transaction managers. You can interface Forte. It does a much better job of supporting the repetitive cycle, than Dynasty because it generates high-level language and debugs it, whereas Dynasty generates a low-level language. Forte is quite expensive, whereas Dynasty is much less expensive. OLE COM, OLE automation, Sun DOE, CORBA ORB's (a universal standard), OSF DCE. **PowerBuilder** versions 5 and later support these as do **Visual Basic** versions 4 and later. This is not distributed OLE, but it is good enough. CORBA is very interesting--object-request brokers--DCE, MOM (Message-Oriented Middleware), or CORBA ORB. It is showing us what architectures will look like in the near future. You can buy the ORB's from any vendor and then plug them together, in spite of their supporting different standards. Ultimately, you would like to use any tool with any standard. You do this by use of the different standards, and you create ORBs that can be plugged into any standard and any tool. If the tools implement the Interface Definition Language, then this can happen; the tools can develop objects for any standard. The tools must implement an IDL for each one of the object structures that are of interest. This capability is just coming on the market; the tool vendors are just beginning to provide that kind of capability.

[11] Decision Support System
[12] On-line, Analytic Processing

Supplement 4.2

We provide here additional detail relative to the methodology for software selection. This detail can also support the selection of modules for CBSD, component-based software development.

Define the Application to be Completed

Suppose you, like so many other small computer users, are in the market for a word processing package. However, your application is different from theirs in that you want to be able to support scientific symbols—particularly those used in mathematics and statistics. Clearly, you have defined a nuance that will result in your making a choice that may be quite different from the word processing package choices of your acquaintances.

As a part of this step you will want to carefully delineate the format of the reports your application is to produce, especially if the application is accounting-related. And you will want to specify the capacity requirements of your application. For example, if you have a requirement for a word processor that will support the preparation of heavy, multiple-file documents, then a word processor designed primarily to support the production of letters and memos may not provide the features necessary to support the heavier word processing activity.

Develop a List of What Software is Available to Support the Application

There are at least eight different major sources of information about small computer software--the microcomputer trade literature, microcomputer software directories, software search bureaus, computer stores, user groups, turnkey system suppliers, mail-order discount houses, and the software manufacturers themselves.

The utility of the microcomputer trade literature and directories has been alluded to in Chapters 2 and 3. Certainly they are the best place for one to start becoming acquainted with the small computer software base. More will be said about this information resource later.

Computer stores will stock the trade literature and will provide access to the software itself. Ultimately, the user will want to see demonstrations of those packages of greatest interest, as discussed in what follows. Computer retailers are usually accommodating in this regard.

For detailed information about a software package that is not provided in any of the literature, contact the software vendor directly.

User groups exist for just about every category of small computer in major metropolitan areas. The groups will generally seek to educate their memberships regarding the software available and its usage on a small computer. To find out about user groups in your area, inquire at your local computer retailer.

Gather Information about Available Packages

Reviews of major applications software frequently appear in such periodicals as “Infoworld, PC: An Independent Guide to the IBM Personal Computer, PC World, Byte,” and many others. In addition to the advertising found in these periodicals, the reviews are a good source of information about software. Another source of important information about existing programs is Datapro's “Directory of Microcomputer Software.” Once again, evaluations of the software are published based upon user responses to questionnaires sent out by Datapro. Hence, in this material the software is reviewed and evaluated by users. The presentation is in the form of two-dimensional tables which enables a side-by-side comparison of user satisfaction with various aspects of all packages.

It is important to point out that reviews are often flawed, superficial, overly critical and inconsistent. Almost never is all the software available for a given application reviewed by the same reviewer or set of reviewers. As you read a review, ask yourself some introspective questions--does this review suggest that the reviewer is qualified to make the judgements about the package that is presented? Are the findings backed up with substantive tests and evaluations? Is the review superficial in the sense that the reviewer passes the package off with some laudatory remarks but provides no depth, no comparative evaluations, and little content? Is the review overly critical in the sense that the reviewer finds fault with almost every facet of the program and backs these harsh pronouncements up with little or no substance? To be on the safe side, consult the findings of several reviews for packages that are of particular interest to you.

Software manufacturers may be very helpful in providing technical information not available elsewhere. For example, you might have a requirement for a high-level language that will support large amounts of in-memory data. Such a requirement is typical in scientific computing. Many languages have a 64K byte data-segment limitation which makes them unsuitable for such applications. However, some do not. (For your information Microsoft FORTRAN and Microsoft Quick Basic 3.0 do not, but Microsoft's MS-DOS BASIC interpreter does.) This type of technical detail may not be supplied with any literature that you have seen, and the salespeople at the local computer emporium may not know the answer. In such cases you must call or write the software manufacturer.

Narrow the List of Possible Choices Down to Less Than a Half Dozen

One criteria that is available to owners of small computers who have not defined all of their applications yet is this: "Does the package run on my computer?" If it doesn't, then there is little use in giving the package any further consideration unless you are willing to exchange computers.

Obtain Hands-on Demonstrations of the Remaining Packages

A pleasure visit to the computer store is essential at this point in your evaluation. Before you buy any package, it is appropriate to first of all have that package demonstrated to you and, if possible, to actually test the package using some tests you might have developed beforehand. For example, if the package is a word processing package, bring along a page of text and try entering, editing, saving, and retrieving the textual material. The program should not contain any surprises for you. Its user interface should be acceptable in terms of your level of computer literacy. Are the basic features of the program seemingly clean, fast, and efficient when executed. In short, this is your opportunity to verify what you have no doubt read about in the reviews and other literature pertaining to the software being examined.

Glossbrenner [1] gives us the following list of items to consider during the demonstration. First, what hardware is required to support the package? Are there additional graphics boards, RAM cards, or peripherals that must be purchased in connection with the software package? These may not be visible so be sure to ask the salesperson. Does the software require expanded capacity floppies, a RAM disk, or a hard disk?

Second, can you actually begin using the program without reading the instructions? Most programs are not quite this obvious in terms of their use; however, some are. This is a benchmark of the program's user- friendliness. It has been said that a program should be designed so that you could use it even if you hadn't used it for six weeks and you lost the manual a month ago. To what extent is the program icon- or menu-driven? As we observed in Chapter 11, pop-up, pull-up, and pull-down menus are becoming increasing popular. These operate off of a menu bar at the top or bottom of the screen. Such programs are usually both menu and command driven. And the commands are obvious terse representations of the menus items. Ask yourself as you test the program, "Are the prompts and commands easy to understand?"

In [1] are suggested several tests to put a program through its paces, including deliberately trying to cause the program to fail. For example, the user might enter an invalid response to a prompt just to see how it responds. If a number is expected, enter a letter. If a letter is expected, enter a number. When the program prompts you for a filename, enter an invalid one just to see how it responds. Is the program "blown away" by these invalid responses, or does it "hang the machine" so that you have to reboot and start over? Or does the program trap your error and provide a graceful remedy, perhaps by suggesting why your response was invalid and thereafter letting you re-enter your response. For example, if the program requests a filename, and you commit an error in typing in the filename, it should inform you that the filename you entered was not found. It should give you an opportunity at this point to examine the directory of the diskette and to reenter the filename. Glossbrenner [1] suggests doing a disk access with no disk in the drive to see how the program reacts. This will cause many programs to revert to the operating system level.

Of Those that Remain, Perform a Final Evaluation

In a subsequent section a grading system based upon the attributes of the software, the weights you attribute to them, and the grade you assign to them is proposed for the evaluation of computer systems and hardware. That scheme is equally appropriate in the evaluation of software. To use it simply develop a list of attributes you consider desirable, assign weights to those attributes, and then grade each package you are considering in each of the attribute areas, as shown in Figure 4.8. The grade you actually assign to each package in each attribute area will be influenced by your impressions of the package derived from the demonstration and hands-on examination described above.

“Weight” Package 1 package 2 package 3•
Ease of learning
Editing features
Formating features
Printing features
File handling features
Ability to interface w/ other programs
Speed
Vendor Support
Figure 4.8: A Method for Grading Software.

What are the implications of methodology for the culture of an organization?