Chapter V

The Software Engineering Discipline

He who hurries cannot walk with dignity.

(Ancient Chinese saying)

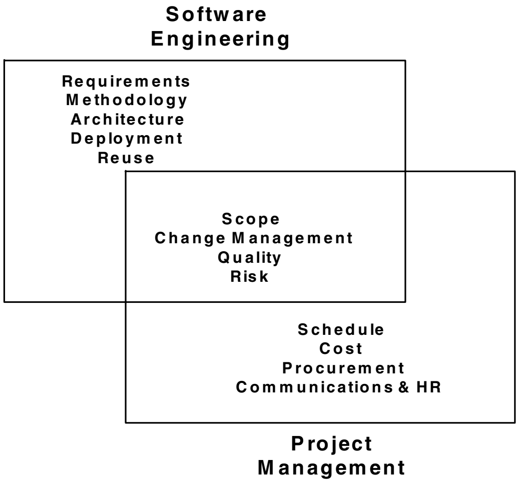
Software engineering is vital for the proper planning of IT projects, although it is not a formal part of project management. The software engineering embedded in the acquired products will significantly affect long-term project success factors, even for IT projects that primarily involve software acquisition and integration instead of software development,. In this chapter I review software engineering and its relation to IT project management.

Software Engineering vs. Project Management

The project management and software engineering disciplines overlap considerably, as is illustrated in Figure 5.1. The Institute of Electrical and Electronics Engineers (IEEE) software standard 1490-2003 provides for the adoption of PMI Standard (PMBOK).

The IT industry has no one methodology, architecture, or set of standards; however, in other industries, there are typically established codes, frameworks, patterns, methods, and tools that are almost always used. For example, the home building industry has county building codes, frameworks for house patterns (ranch, colonial, Tudor, contemporary, etc.), subdivision guidelines and limitations, standard methods, and tools of the trades involved. The IT industry has a number of rapidly changing and evolving

Figure 5.1. Software engineering vs. project management



standards, frameworks, architectures, tools, and methodologies from which to choose. Therefore, before the project is planned in terms of breaking down and assigning to resources the scope/requirements, these other issues need to be addressed. Many of the problems in project management can be traced back to problems in methodology, architecture, reuse (lack of), and standards.

The term *software engineering* was coined by Bauer (1972) who was a principal organizer of the 1968 NATO conference on that subject. His definition of *software engineering* was "the establishment and use of sound engineering principles in order to economically obtain software that is reliable and works on real machines." The IEEE definition is "the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software (IEEE Std 610-1990). The modern Webopaedia definition follows:

> *Software engineering is the computer science discipline concerned with developing large computer applications. Software engineering covers not only the technical aspects of building software systems, but also management issues, such as directing programming teams, scheduling, and budgeting.*

Software Development Life Cycle Methodology

According to Webster's dictionary, *methodology* is "a system of methods." My definition for *methodology* is "organized know-how." The most common and established methodology used in building and/or integrating IT systems has been informally called the waterfall method and formally called the software development life cycle methodology (SDLC). This notion and term was first applied to IT systems by Royce (1970). The steps (illustrated in Figure 5.2) in this classical methodology are:

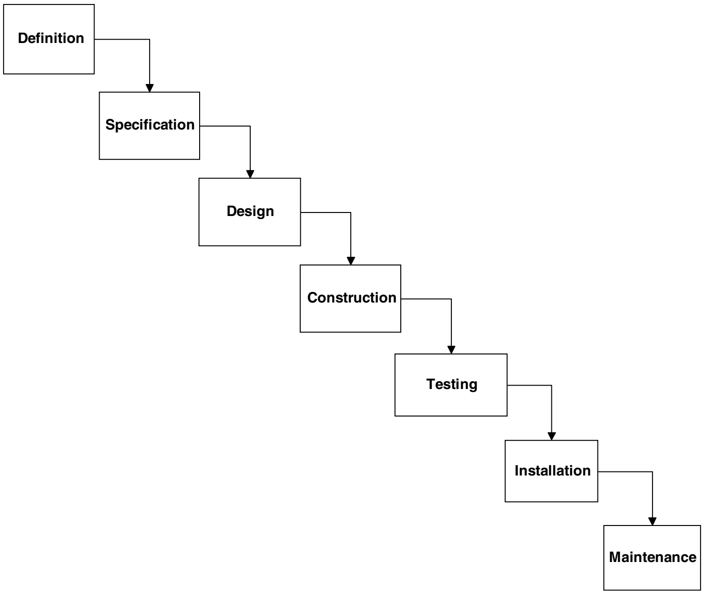
- Definition
- Specification (requirements)
- Design
- Construction (programming and unit testing)
- Testing (system and integration)
- Installation
- Operation and Maintenance

In theory these steps are not supposed to overlap or iterate. Some of the newer software methodologies are variations of or alternatives to the basic waterfall approach.

One hears many comments about the classical waterfall software development life cycle:

- The software development life cycle is the cornerstone of development!
- The life cycle is out of date!

Figure 5.2. SDLC waterfall



- Get good people and the life cycle will manage itself!
- There must be better and faster ways to build IT systems!

SDLC goals are to

- Do it right the first time!
- Meet customers' stated requirements!
- Have a timely completion!
- Complete within cost constraints!
- Build system with the necessary quality!

The *definition step* involves making a clear statement of goals, identifying why and how the proposed system will be better–cheaper–faster than the system it is replacing, and usually a overall/rough cost-benefit analysis. This phase is typified by frequent customer interaction, elimination of arbitrary constraints, negotiation and compromise on scope (features) versus time and cost, statement of assumptions, rough time and cost estimates, a rough project plan, and a signed go-ahead (i.e., the project charter).

The *specification step* involves a complete statement of scope (requirements), use case scenarios, preparation of preliminary user manual (external design specifications), detailed project plan (including work breakdown structure [WBS]), specification of needed resources, refined estimate of time and cost, refined cost-benefit analysis, and signed approval of requirements and user manual by stakeholders. However in practice the user's manual is rarely written at this stage. The reason the user's manual (or at least a draft) should be done at this step is so that some other dependent activities can begin, such as test planning and test scripts, making training plans and materials, and other dependent tasks, like internal or external marketing.

The *design step* involves resolution of critical technical issues, selection of architecture and platform(s), adoption of standards, assignment of staff, completion of external design (user interface design), design of critical data structure and database, internal design of algorithms and processes, Requirements Traceability Matrix, preliminary test script, final time and cost estimate, and a final cost-benefit analysis. Often the design step is divided into two steps; analysis (or overall design) and design (or detailed design).

The *construction step* involves the implementation of the design (i.e., via coding), unit testing, systems integration, draft internal documentation, and the completion of test scripts.

The *testing step* involves full scale integration and system testing, completion of user documentation, completion of training material, adoption of formal change control procedures, completion of the internal documentation, completion of installation manual and roll-out or phase-in plan. Testing is further discussed in Chapter X.

The *installation step* involves product roll-out, end-user training, producing lessons learned documentation, and defining procedures for handling operations, user support, and configuration management.

The *maintenance step* involves following and revising procedures for problem resolution and problem escalation, operations, backup and security, configuration control, and quality/performance monitoring.

At the end of each step there is usually a formal meeting in which a document is produced for the culmination of effort in that step. This document is reviewed by project management, the performing organization line management, and the benefiting organization (customer). If any of these stakeholders are not satisfied with the results of that step, the project can be terminated or the step repeated; the project will not proceed unless the stakeholders have given approval to move forward at each step. In theory this should result in a product that satisfies the initial requirements and the stakeholders. The following is what can, and often does, go wrong:

- User requirements are misunderstood, incomplete, not fully documented, or not fully implemented
- Requirements have changed
- Documentation is "unusable"
- System is difficult to use
- Training is ineffective
- Capacity or performance problems are present
- Audit and integrity problems are present
- "Bugs" and other quality issues are present
- Standards are not followed
- Estimation of workload is poor
- Project is managed poorly
- Budget is exceeded
- Not completed on time

These issues and others are discussed in detail in later chapters, along with the examination of root causes and remedies. Many in the field feel that the classical waterfall approach is too slow in today's fast-paced and rapidly changing world. Remember that only a small number of all IT projects result in fully working systems. Other projects are sent back for reconstruction, abandoned after delivery, or never completed. Therefore, how, *in general*, does one keep things from going wrong, even when a sound methodology is employed? These questions have to be answered:

- Are you committed to the methodology, or is it just words in a book gathering dust on the shelf?
- Do you have the ability to follow the methodology organizationally and with respect to resources?

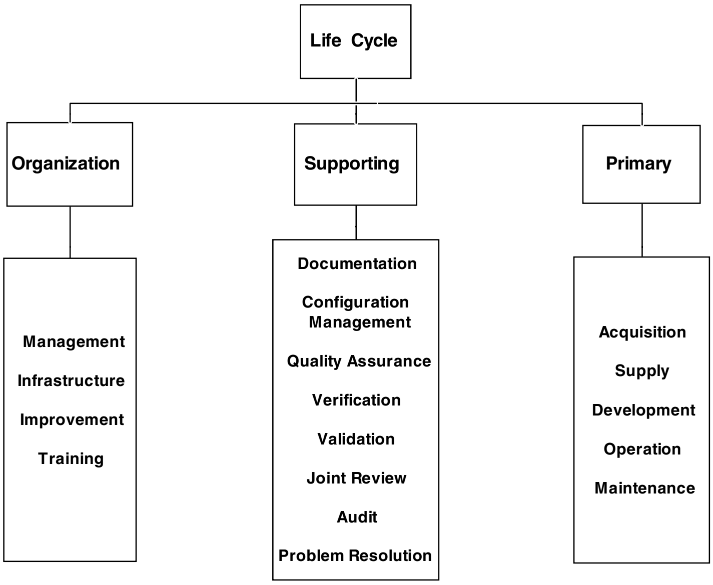
- Does the methodology specify the things that need to be done?
- Do you measure via metrics or benchmarks things that have been done and done properly?

Under the Software Engineering Institute's (SEI) capability maturity model (CMM), which is detailed later in this chapter, these questions correspond to the "common features" of the maturity models:

- Commitment to Perform
- Ability to Perform
- Activities Performed
- Measurements and Analysis
- Verifying Implementation

Later chapters provide practical answers for the last three questions as they relate to specific project management and/or software engineering methods and tools. Figure 5.3 shows the IEEE/EIA Life Cycle Process definition as well as the definitions for support and organizational processes. (These processes are also covered in later chapters.)

Figure 5.3. IEEE/EIA life cycle process



Management Stage Gates

The notion of stage gates can be traced back to phased project planning used decades ago by NASA for handling very large aerospace projects. These gates are attractive to management because they restrict investment in later stages until the anticipated return on investment is clarified and made more certain by earlier phases. Under this methodology, stages and gates divide the total effort into a series of consecutive stages, whereby gating criteria must be met before the project can move from one stage to the next. This is illustrated in Figure 5.4. The gating criteria involve the review of the defined outputs from the previous stage as well as metrics to justify that the project scope can still be completed within the time and budget constraints.

For IT projects, a traditional stage gate technique can be superimposed upon the chosen methodology. For example, combining the classical waterfall methodology with stage gates may result in the stage deliverables/outputs as shown in Figure 5.5.

At each stage gate, management would traditionally review the following:

- Defined output (stage deliverables)
- Completion status of activities
- Actual costs to date

Figure 5.4. Stage gates

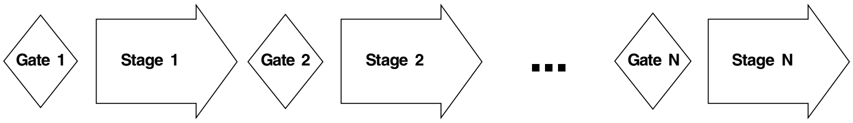
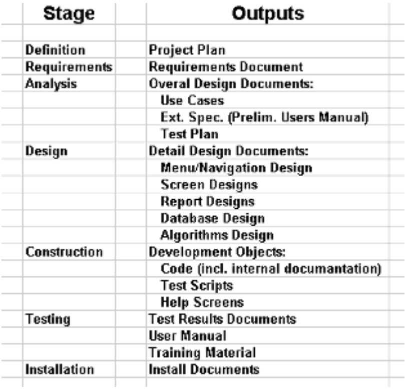


Figure 5.5. Stage gate outputs



Stage	Outputs
Definition	Project Plan
Requirements	Requirements Document
Analysis	Overal Design Documents:
	Use Cases
	Ext. Spec. (Prelim. Users Manual)
	Test Plan
Design	Detail Design Documents:
	Menu/Navigation Design
	Screen Designs
	Report Designs
	Database Design
	Algorithms Design
Construction	Development Objects:
	Code (incl. internal documantation)
	Test Scripts
	Help Screens
Testing	Test Results Documents
	User Manual
	Training Material
Installation	Install Documents

- Estimated cost at completion
- Estimated time to complete
- Updated risk analysis (i.e., the need for more or less reserves)

The stage gate technique can be used with any of the variations or alternatives to the classical waterfall discussed later in this chapter. For example, the stage gates can be set for each iteration if an iterative methodology is used. Another way to implement this approach is to set the gates at specific time periods, such as each month or each quarter.

This traditional stage gate process can slow down a project, however, due to the amount of review time at each gate. The events that comprise review time involves producing reports, sending reports to management, having management privately review reports, and then scheduling a public review meeting. Project cost is also higher due to the high cost of the people involved with a stage gate review. To overcome these disadvantages, a number of alternatives to the basic stage gate process have been proposed. These variations are called a number of names, such as fuzzy gates or exception gates, but the common goal is to have a process that is not slowed down by the gates, unless there is a significant problem.

As was discussed in earlier chapters, I suggest using a dual gating approach with a management stage gate at specific time periods (i.e., monthly or quarterly) wherein the earned value critical ratio is the dominant metric used for the go/no-go decision. This management gating technique is combined by a quality gating process, and there may be multiple quality gates within each management gate (or vice versa). The management gate focuses on the completion criteria and the quality gates address the satisfaction criteria. The quality gates review specific preliminary product manifestations in regard to the satisfaction factors of operation, utility, maintainability. A revised cost benefit analysis (based on latest earned value estimate at completion and revised benefit numbers) can also be included. *This combined gating process effectively and efficiently addresses all the project success factors discussed previously in this book.* Later chapters elaborate on earned value, quality management, metrics for all the success factors, and this type of gating process. Figure 5.6 illustrates this combined gating process and its relation to the project success criteria.

SDLC Variations and Alternatives

Because of the extensive and formal stakeholder review at the end of each step and the lack of overlap, the classical waterfall methodology can be slow in getting a software product to market. Also, the waterfall method becomes unstable if the initial requirements are significantly in error or change much. New technologies and global competition are quickly changing the business landscape, creating another problem. From the time a business problem is analyzed and a solution built, the “shape” of the original problem has changed significantly; thus, the developed solution no longer matches the original problem. This is illustrated in Figure 5.7. Project success rates show that large IT projects

Figure 5.6. Dual stage gates

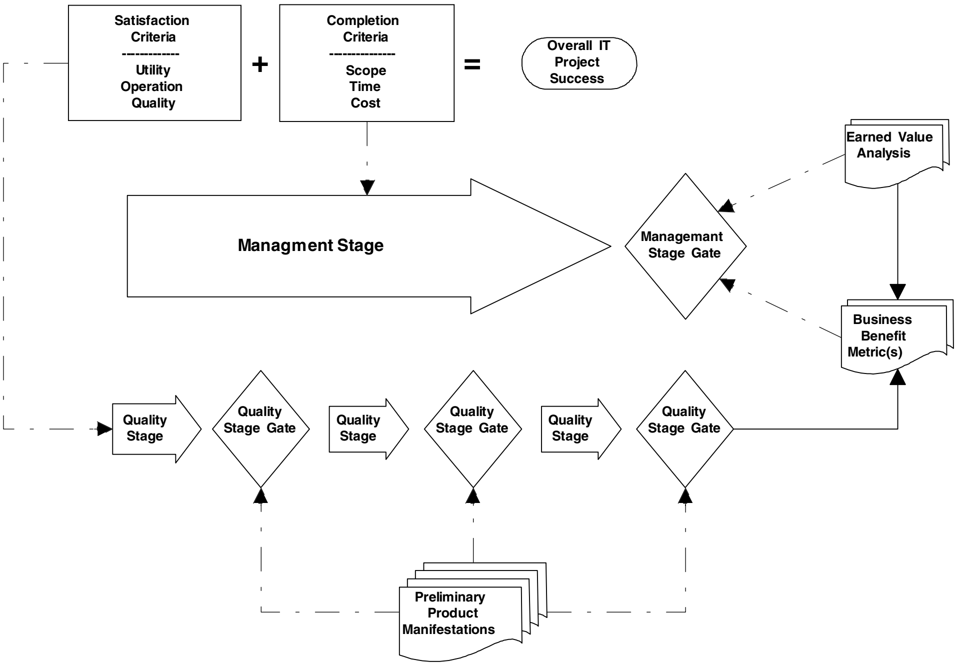
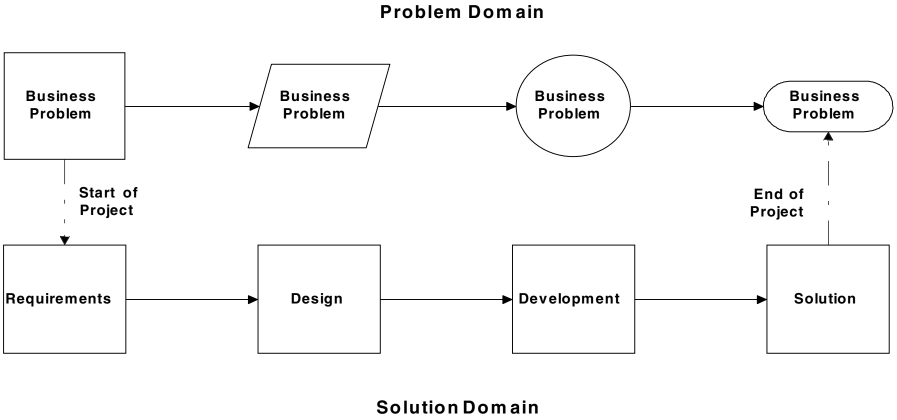


Figure 5.7. Changing shape of IT problems



are getting harder to complete successfully (Standish Group, 2004). Projects over $10 million have success rates of only 2%, projects between $3 and $10 million have success rates from 23% to 11%, and projects under $3 million have success rates from 33% to 46%.

Because of these low success rates, a number of variations and alternatives have been suggested and tried, with varying degrees of success. Many of these approaches take large IT projects and break them down into smaller, more manageable pieces. However, there is no single silver bullet approach (Jones, 1994). Some of the software development life cycle methodology (SDLC) alternatives, to list a few, include Yourdon Structured Design, Ward/Mellor, Stradis, Spectrum, SDM/70, LBMS, Information Engineering, IBM AD/Cycle, Gane & Sarson Structured Analysis, DeMarco Structured Analysis, Anderson Method/1, Bachman, Agile and XP, and Clean Room. In this chapter, we will first discuss SDLC variations generically, and then discuss modern specific implementations.

The Overlap or Free-Flow Method allows any task to proceed as long as its dependent tasks are completed. Here the basic waterfall steps may have considerable overlap (this is illustrated in Figure 5.8). For example, even though the total-system design is not completed (or documented and approved), the implementation of those components whose design is completed may begin. This overlap is built into the dependency relationships in the work breakdown structure (WBS), which is discussed later in this book. One is betting that the total design (such as may be manifested in UML design drawings) will be approved. The concept is similar to optimistic record locking in an interactive database application. This technique works very well with the use of the dual stage gate approach of this book. This is also a good technique on contracts where incentives are available for early completion. Obviously risks are greater with larger projects and for projects where requirements can change significantly.

Evolutionary Development begins with only the user requirements that are very well understood and builds a first version. Often that first version is just a prototype. Analysis, design, implementation, and testing are done in a free flow overlapping manner without any formal review of documents. This first version is then taken back to the customer for review and definition of further requirements. Next the second version is built, and then taken back to the customer. This process continues until the customer is satisfied with a version and no further extensions are required. (This is illustrated in Figure 5.9.) Documentation, training, acceptance testing and other project completion activities are done at that point at which all (or most) of the customer's requirements have

Figure 5.8. Overlap method

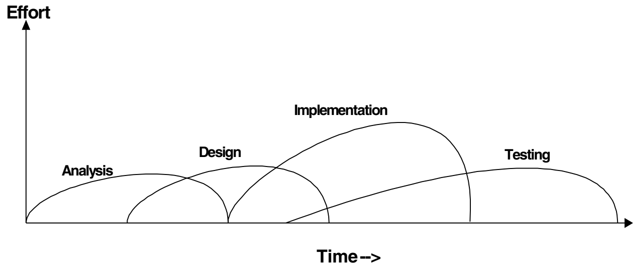
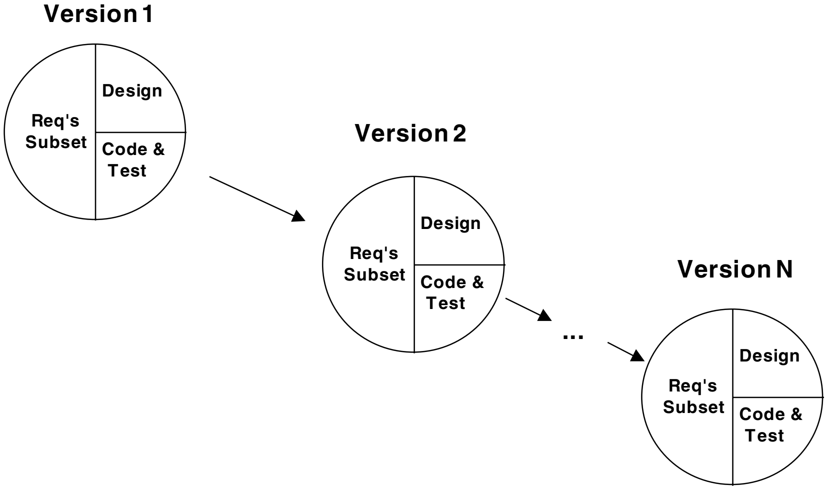


Figure 5.9. Evolutionary development

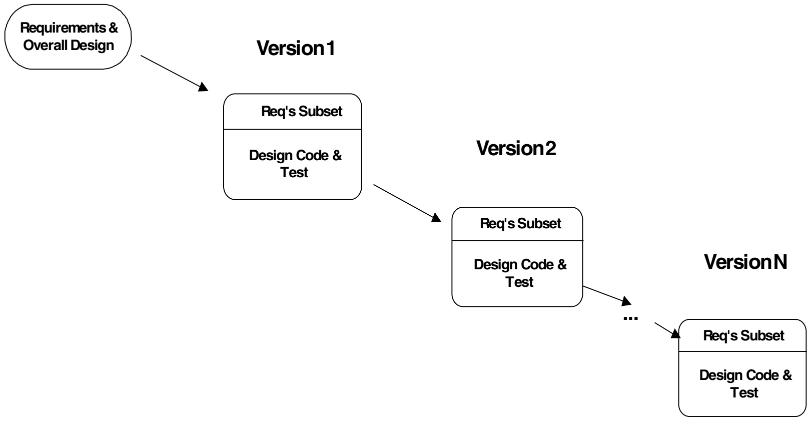


finally been included. This methodology is much faster than a waterfall approach and also somewhat quicker than the free-flow method. However, management visibility may be limited because little intermediate documentation is produced. Also, in a contracted environment (external or internal), a fixed price contract could not be used since the overall scope is not initially determined and priced. For a contracting environment, either a cost plus or time and material contract would have to be used. Contracting and procurement are discussed in a later chapter of this book. Also, internal design is often poor for evolutionary development because the entire scope is not visible from the beginning, and continually changing a system leads to a design that is less adaptable and harder to maintain. If the system is designed and built in a fully object-oriented manner, this problem may be minimized; object-oriented design is discussed later in this chapter.

Incremental Development begins with a determination of all the requirements, but only in a rough outline form. Next, those requirements are prioritized normally based upon those features that are most important from a business perspective. Because time is spent up front looking at all requirements a more appropriate overall platform, architecture, and design can be selected. This is particularly important for security requirements, because security cannot be an afterthought. Good security has to be built into the total product (and the methodology of constructing it), not bolted on afterwards.

After the initial requirements phase, development proceeds as in the evolutionary method. (This is illustrated in Figure 5.10.) Each increment typically represents a product portion that can be placed into service. Incremental development is not as quick as evolutionary development, but attempts to avoid the design problems caused by not knowing all the major requirements initially. However it suffers from the same contract type issues as the evolutionary method. Another potential problem is that the increments are based on the priorities of the requirements, and sometimes priorities may significantly

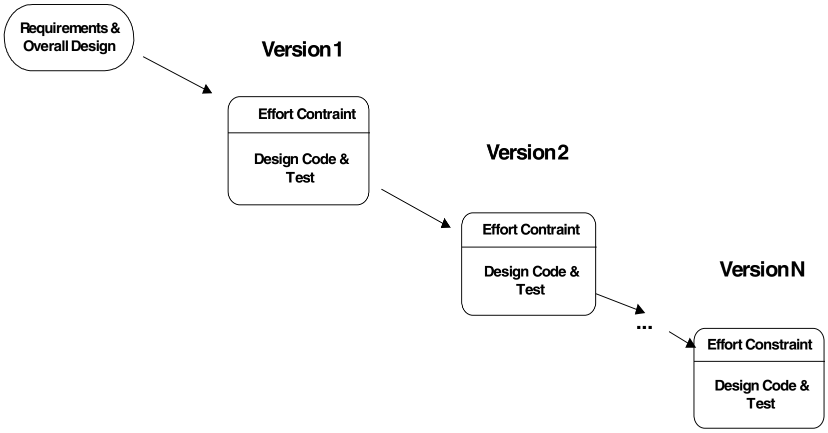
Figure 5.10. Incremental development



change during the time of developing the increments. Both the Rational Unified Process and Extreme Programming as well as several other new methodologies use this technique and these are discussed later in this chapter.

Bounding Box Development is similar to incremental development, except that each increment is not based on a certain scope (requirements subset) but is instead based on a measure of effort. If the effort put into an increment is constrained by calendar time then the term *timebox* is commonly used. (This is illustrated in Figure 5.11.) For contracted development (external or internal), the increments are usually based on a dollar (budget) amount. Thus this method does not have the contracting disadvantage that evolutionary or iterative methods have, but customers must be willing to contract for portions of the total system. However, because the amount of scope that will actually be completed in each increment is not known, each increment may not represent a product portion that can be placed into service.

Figure 5.11. Bounding box development



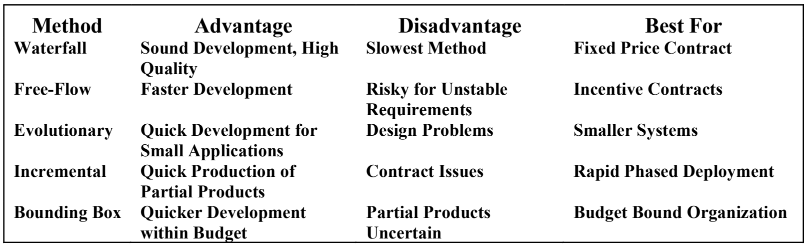
There is no one best methodology. An organization must select the methodology that is most appropriate for the type and size of the IT project at hand, the nature of the customer and stakeholders, the contracting environment, and the resources involved, both people and financial. Combinations of these methodologies can also be used such as in the Rational Unified Process which combines incremental and free-flow techniques. Figure 5.12 shows the primary advantage and disadvantage of each of the methodologies.

Development Acceleration

In addition to modifications of the basic SDLC, a number of other philosophies, methods, and tools have been proposed and tried, to compress all or certain steps in the SDLC.

Rapid Application Development (RAD) is a generic term for software development methods and tools that speed up the development process. It was commonly applied to products that automatically generated code to create user interface screens both of the character type and graphical type (GUI) starting in the 1990s. Products such as Clarion, FoxPro, Visual Basic, and PowerBuilder are examples of this era. Often these products also featured automatic report generation capabilities, and some products were devoted to this aspect such as Crystal Reports. So-called fourth-generation languages, such as Natural, were also part of this RAD landscape. Many RAD products were combined with databases such as Microsoft Access, and other RAD products could interface to stand-alone database products. Earlier RAD products built stand-alone PC applications or multiuser PC applications via LAN (local area network) file redirection (mapped drives). Later RAD products used SQL (structured querylanguage) to create true client-server applications; however, almost all of these still required client machine configuration such as mapping drives, client installation of drivers, or other middleware. These RAD products did speed up considerably the implementation step for smaller applications, but they did not accelerate other development steps: requirements, analysis, design, and testing. For larger or more complex systems, automatic code generation is not powerful or flexible enough to meet application needs and programmers usually have to resort to traditional hand coding. Today, modern IDEs (integrated development environments), such as Dreamweaver, NetBeans, Eclipse, and Visual Studio, contain features including WYSIWYG (what you see is what you get), drop and drag screen generators, and

Figure 5.12. Methodology comparison



Method	Advantage	Disadvantage	Best For
Waterfall	Sound Development, High Quality	Slowest Method	Fixed Price Contract
Free-Flow	Faster Development	Risky for Unstable Requirements	Incentive Contracts
Evolutionary	Quick Development for Small Applications	Design Problems	Smaller Systems
Incremental	Quick Production of Partial Products	Contract Issues	Rapid Phased Deployment
Bounding Box	Quicker Development within Budget	Partial Products Uncertain	Budget Bound Organization

including, possibly producing, thin client Web applications, which do not require any client–machine configuration changes. *It is often said that RAD products speed up the easy part of the development; however, RAD products are very useful in creating prototypes and are particularly useful in incremental types of development methods*. RAD tools are constantly evolving with the fast-changing IT landscape, and RAD tool vendors are constantly being acquired by other IT vendors. Therefore, if one develops a product today, using a particular RAD tool, that tool may be obsolete or be in the hands of a different vendor tomorrow (with little or no remaining support).

CASE (computer aided software automation) refers to computer software systems that generally support several steps in the development process, usually including requirements, analysis, design, and possibly other steps such as coding, testing, documentation, and version control. Most CASE products are primarily geared towards the development of business software systems and central themes are a data dictionary and various design drawing and associated repositories. There are different types of CASE products, and these products can be classified in a number of ways. Some products support one particular methodology and others can support several methodologies; some support one particular computer language and some may support several languages. Some CASE products are called horizontal tools or workbenches or environments, and they provide integrated tools for the support of multiple development steps. A modern CASE environment consists of a number of tools operating on a common hardware and software platform; these tools may be from one or multiple vendors. This CASE environment also supports a number of different types of users: project managers, designers, programmers, database administrators, testers, technical writers, and so forth. The CASE environment is distinguished from a random tool set in that the environment facilitates the integration of those tools so that they can work together coherently without duplicating effort or information.

Today there are hundreds of CASE tools available, and perhaps many in each of the aforementioned categories. Commercial products include such names as Amadeus, Continuous, InConcert, Life*Flow, MATE, Process Continuum, Process Engineer, ProcessWeaver, ProcessWise, ProSLCSE Vision, and SynerVision. Although these products have provided significant development accelerations (up to 50% in some cases), *one must be careful not to let CASE replace a sound methodology, or all you will accomplish is to build the wrong system even faster***.**

Prototyping involves creating a scaled-down model of the product to be built. The scaled-down model is usually live, in that it is implemented in software and has some degree of functionality; however, it may be only on paper (paper prototype). A prototype is not fully tested, does not implement all features/requirements, and typically does not address tasks/issues as database interaction, concurrency, transaction boundaries, scalability, security, recoverability, maintainability, reusability, and so forth. Speed and capacity are not issues here unless the prototype is specifically built to test those characteristics. Prototypes should be easily modifiable to explore different layouts, navigations, and behaviors; thus, today's prototypes are usually built with RAD/IDE type products. They can often serve as an external design for portions of the final system. The purposes of prototypes are to demonstrate feasibility, evaluate alternatives, clarify and flush out user requirements, and evaluate "user friendliness" and other qualitative aspects.

However, there are dangers in using prototypes. Often, a prototype can assume a life of its own. Sometimes when managers or customers see the prototype, they think that the product is close to being ready; they are not aware that the prototype is grossly scaled down and that some of the most difficult and time-consuming parts of the product are not in the prototype. *Despite these dangers, this author thinks that prototyping is a vital part of modern application development independent of whatever overall methodology is utilized.*

Paper prototypes, or *storyboards*, have many of the same advantages of live prototypes, but they do not have the aforementioned disadvantages of live prototypes, and they can be produced and modified quicker and cheaper. Paper prototypes can be constructed with paper and pencil, simple drawing programs, presentation software, or RAD products. The television and motion picture industry have used story boards for decades, but use in IT was minimal until the advent of Web applications. Storyboards essentially walk the user through the interaction between the end users and the system, and visually show that interaction through screen mock-ups of both input and output screens. After basic requirements are gathered and perhaps represented with use cases, storyboarding is the next logical step in clarifying or detailing those requirements and flushing out added requirements. Each interaction in a use case diagram could be simulated via a storyboard.

For example in the case of building modern Web-based applications, storyboards and simple user interface prototypes can be easily constructed with RAD products, like Dreamweaver. Even if the final Web product is going to use a Web server API (application programming interface), the tags and scripts (JSP, PHP, Cold Fusion, or ASP) can be added directly to the prototype HTML/XML/JavaScript code.

Joint application design (JAD) was formulated by IBM personnel in the late 1970s. In 1980, IBM Canada held several workshops to demonstrate the concept, and later in the 1980s, JAD was utilized in a number of companies. IBM defined JAD as an "interactive systems design concept involving discussion groups in a workshop setting." The purpose of JAD was to bring together developers and users in a structured environment for the purpose of obtaining quality user requirements. They believed this structured approach provided a better alternative to traditional serial interviews by analysts of the performing organization. The prime advantage of JAD is a reduction in the time it takes to complete a project. It improves the quality of the final product by focusing on the initial portion of the SDLC thus reducing the likelihood of errors that are expensive to find and correct later on.

JAD takes place in a structured workshop session. Representatives from the performing and benefiting organization meet in a room and discuss preset issues. Everyone gets a chance to speak, and questions are answered immediately—there is no telephone tag or waiting for memos or e-mails to recycle. JAD also seeks to eliminate the problems with traditional meetings by turning them into organized workshops, with facilitators, visual aids, agendas, deliverables, and feedback.

As JAD was used more in the 1990s, the term was broadened to include more collaborative efforts between the benefiting and performing organizations, including conflict management, brainstorming sessions, and motivational meetings. Sometimes the sessions were more like technical workshops, where participants focused on needs analysis and applied software tools in the process of gathering business requirements. Sometimes these sessions used RAD and CASE tools, and JAD usage expanded to functions other than the requirement gathering step in the software development life cycle.

Today, JAD is used in many steps of SDLC and is often defined as a system development methodology. In 20th century, JAD brought users and developers together in the same physical location, but today JAD is often held via virtual meetings. JAD meetings must be structured, and some guidelines include the following (Dennis, 1999):

- Have management support
- Use experienced facilitators
- Get the right people to participate and set their roles
- Set clear session objectives and deliverables
- Have a detailed agenda and stick with it
- Produce deliverables shortly after the session

I feel that clear and complete communication between the performing and benefiting organization is vital to the success of IT projects. JAD offers a valuable technique for effective communication particularly in requirements gathering. One must be careful to follow the above guidelines or these sessions may just waste more of everyone's time in meetings. There are also certain types of customers where access to customer personnel is very limited. *JAD sessions combined with review of prototypes are excellent way to solidify requirements early in a project.*

Modern SDLC Implementations

The Rational unified process (RUP) is based upon both the incremental methodology and the free-flow methodology discussed earlier. Instead of attempting to address all of a project's requirements, RUP produces software iteratively that addresses a compromised but known feature set and evolves the project over time (Jacobson, 1999; Kruchten, 1998). The difference between evolutionary methods and RUP, though, is that one identifies the requirements for the entire system, but only details the top 20% or so of architecturally significant cases during a single increment. This enables the determination of an appropriate architecture and design which will accommodate the remaining 80% of the requirements without compromising the integrity and quality of the total system. This is particularly important for security requirements, and a plug-in to the standard RUP, called CLASP (Comprehensive Lightweight Application Security Process), is available which, provides a structured way to address security issues in the software development process.

RUP specifies different roles for project participants. Before an architect ever gets involved, an analyst is building use cases and evaluating and prioritizing them with the customer. Before the coders begin implementation, architects work with analysts to identify the architecture which best satisfies requirements and constraints. UML tools are used to build a consistent model from requirements to detail design. RUP uses the free-flow methodology also in that there is considerable overlap in activities of different roles.

RUP is a phased approach that defines four distinct phases:

- *Inception:* Understanding the need, understanding the proposed system to address the need, making the business case for the proposed solution
- *Elaboration:* Selecting the architecture and developing the project plan
- *Construction:* Design, coding, integrating, and testing
- *Transition:* Installing the product on the target platform(s) and getting the user take ownership of the system

The key to RUP is iteration, both within each of the aforementioned phases and within the incremental production of version. Each iteration within a phase ends in a deliverable, and each increment results in a working product version. RUP defines static workflows, core workflows (business models, requirements, analysis/design, testing, deployment) and support workflows (change management, project management, environment and tools). However each of these static workflows is not associated with any one phase, and some degree of each type of workflow goes on within each phase.

The transition from phase to phase is not separated by a stage gate, and management control is not done by placing dates upon the phase boundaries. Management control is only done upon iterations. The project plan contains a list of proposed iterations (which is likely to change), and each iteration has an estimate (which is also likely to change). The proposed iterations are not assigned due dates, but decision points are set up in time (usually based upon weeks). At each decision point, a decision is made in regard to adding/removing resources, adding/removing iterations form the next release (version) of the product, or killing/holding the project. These decisions are based upon progress, cost, and/or earned value metrics. Thus a key part of the project plan is how risks will be managed; it is a plan of contingencies, as opposed to just a plan of activities. *RUP is not suitable for all IT projects. It is complex and difficult to quantify in a contracting arrangement. However for internal projects that are large and risky, and where quick deployment of partial products is necessary, it may be an appropriate choice.*

Agile programming (AP) is a name given to a growing number of lightweight methodologies with names like Crystal (Cockburn, 2001), Scrum (Schwaber, 2001), Adaptive (Highsmith, 2000), Feature-Driven Development (Palmer, 2002), Dynamic Systems Development Method (DSDM; Stapleton, 1997), and Extreme Programming (Beck, 1999). During the 1990s, there was such a need to quickly build new IT systems to take advantage of new technologies like Web applications and e-commerce, as well as the need to address the Y2K problem, that IT organizations began exploring these lightweight techniques. Lightweight methodologies do away with much of the SDLC process overhead that slow down developers, such as detailed formal requirements definitions and extensive documentation. Some feel that these new development approaches are based on the premise that if you hire competent programmers who always know what they are doing, then any problems they encounter are organizational and communications ones; and those are what the agile approach focuses on.

Although the various agile methods differ, they have some things in common. Most use an incremental free-flow approach, as does RUP. The common intent is to be agile, so one should embrace change and produce software that is adaptable; thus, most of these methods call for the use of object-oriented languages. Another common feature is a lot of contact time with users. Still another key focus is a focus on people, not processes, thus emphasizing team morale building. Most AP methods have some core principles, including the following:

- Use a simple design (the old military KISS principle)
- Design as you go, and keep refactoring the code
- Take small, incremental steps (when changing or adding code, take the smallest step possible, then test again)
- Stick to one task at a time (do not add code to accomplish two things at the same time)
- Use IDE and RAD tools
- Use only the techniques that really work for you

These methods may seem basic and obvious to many developers, but I know many programmers who never followed any of these principles. Some programmers, instead of modifying a module or class for a small change or addition, will spend a great deal of time writing the entire module from scratch. The advantages of these AP principles include the following:

- Faster reaction to changes in requirements
- Overall simplicity of the design
- Earlier coding is possible
- By refactoring the code, the most important parts get the most attention (no time invested in changing what does not need to be changed)
- Code in progress is always stable

Refactoring, one of the core development concepts, is a new word for cleaning up the code. More formally, refactoring improves the design and maintainability of code in small incremental steps confined to areas of current interest. One problem with refactoring, however, is that, when a programmer comes under pressure to finish quickly, he or she may not complete the refactoring work.

AP is relatively new, so the success and applicability of these methods is unclear. It is felt that AP is suitable for small projects and small teams; whether it has practical application for larger environments is still in question.

Extreme programming (XP) is a software development approach initially created for small teams on risk-prone projects with unstable requirements (Beck, 1999). Kent Beck, a programmer and project leader, developed XP while working on a long-term project to rewrite Chrysler's payroll application. XP is a form of AP based on a lightweight methodology. XP however, differs from most other agile approaches by being much more prescriptive. Like AP, XP is an incremental method with free-flow. XP advocates say the methodology (creating user scenarios and performing upfront feature testing) allows them to develop and deliver code more quickly with fewer bugs. XP is built around rapid iterations, an emphasis on code writing, and working closely with end users. The 12 basic practices of XP are:

1. Customers define requirements via use case scenarios ("stories")
2. Early on, teams release small increments into production
3. Teams use standard names and descriptions
4. Simple object-oriented coding is used
5. Designers write automated unit test scripts before coding
6. Refactoring is used extensively
7. Programmers work in pairs
8. Programmers have collective ownership of all code
9. Teams integrate and check code back into repositories frequently (no longer than 1 day)
10. Developers work only 40-hour weeks
11. User representative(s) remain on site
12. Programmers follow strict coding standards

Although XP in different forms has been used for a few years, many IT organizations have been reluctant to try it. A major issue is that some XP principles contradict longstanding IT policies. For example, XP specifies pair programming, in which two programmers sit side by side, working at a single workstation. Pair programming seems inefficient, but studies have shown that it is no less efficient that traditional programming and usually results in fewer code defects (Williams, 2000). Fewer defects eventually means quicker delivery. However, not all programmers want or are suited for pair programming. Very good programmers should not be encumbered with a sidekick. Many programmers like solitude—that is one of the reasons they choose to work as programmers. Often, programmers consider themselves masters of the trade, and two masters often cause conflicts.

Another problem with XP (like all AP) is its application in contract environments, and still another problematic issue for XP is that all code is generally open for programming pairs to review and alter. This can open up the team to integrity and security issues. As was mentioned previously in this book, internal security is becoming a prime concern for IT organizations, for internally developed code and, particularly, for outsourced programming. Further XP does not address downstream SDLC issues such as training and user documentation.

XP requires the benefiting organization to take a very active role in the development process, even to the extent that users are asked to write tests that will prove that requested functions work properly before they are coded (e.g., customers may write needed scenarios or features—one scenario per card—on index cards. Using index cards is far cheaper and faster than writing, editing, and reviewing a large formal requirements document.). Then, the developers estimate the time needed to build that feature, and, based on the estimates, the customer prioritizes the features. Next, the customer writes the test, and the developers write code that will successfully pass the test. Testing is normally automated, and test harnesses organize test scripts that related to particular functional areas. However, because testing is limited to "acceptance" type testing, full multilevel testing is seldom performed. This may lead to problems with unanticipated inputs, scalability problems, and security problems. This is discussed further in Chapter X.

Because XP requires constant communication between the benefiting and performing organizations (as well as among the developers), and because communication time and traffic increases in proportion to the square of the number of communicating parties, XP is not suited to large teams (Beck, 1999, advises limiting project teams to no more than 12 developers, working in pairs). As with JAD and AP methods in general, a customer may not be able to commit his or her resources to that much involvement.

Thus, XP has a number of specific advantages and a number of specific disadvantages. This is a hot debate topic in the IT world. On the one hand it is thought of as a great breakthrough, and on the other it is akin to "letting the inmates run the institution." XP is not for all IT organizations.

Cleanroom software development (CSD) as a process was developed by Harlan Mills (Mills, 1996) and involves the application of formal specification to software design. CSD can use any of the methodologies previously discussed, but the incremental approach is most often used. Requirements are turned into formal (sometimes mathematical) specifications (Prowell, 1999). These formal specifications are then turned into the final code through a series of correctness–preserving transformations. "Cleanroom treats software as a set of communicating state machines that separate behavior and implementation concerns" (Garbett, 2003).The code is statically checked via rigorous inspections, then system testing is done using statistical techniques. The testing team must be in a separate organization from the developing team. The Cleanroom Reference Model may be obtained online at the Software Engineering Institute (www.sei.cmu.edu/publications/documents/). Research results indicated that CSD produces code with fewer defects (less than 1 bug per KLOC) at no greater cost than traditional methods (4 to 50 bugs per KLOC; Linger, 1994). *However, CSD requires high-level experienced analysts and programmers. For that reason, CSD is best applied to mission-critical and/or life-critical types of systems that are built internally by expert programmers.*

Component-based software engineering (CBSE) is a development philosophy that utilizes existing software modules to build application systems. Any of the aforementioned methodologies may be utilized, but the requirements specification stage here may be longer due to the preparation of procurement documentation. CBSE is a formalized system of reuse at a high level, formalized in the sense of a business approach rather than at the software architecture level. Later in this chapter, reuse is discussed in more detail in regard to software architecture. CBSE is based on having a cadre of reusable modules or programs and some framework for integrating these modules. “IS shops that institute component-based software development reduce failure, embrace efficiency and augment the bottom line” (Williamson, 1997). CBSE can be applied at several levels of granularity. At the highest level is the COTS (Commercial Off-the-Shelf Software) approach, whereby commercial programs are purchased and integrated through a data exchange mechanism. Software acquisition is discussed in Chapter XII.

Web services is a recently evolving approach to CBSE using the Internet. Using this approach, different services are provided by different vendors in real time on their servers, generally at a per usage price. This new computing architecture is formally called SOA, or service oriented Architectures (Hall, 2003). Web services are based upon modern open standards; unfortunately some of these standards (SOAP, WSDL, UDDI, etc.) do not have adequate security built into them yet. Web services architecture uses SOAP (simple object access protocol) as a lightweight remote method invocation process. Older. more complex protocols for distributed object services are Microsoft’s DCOM (distributed component object model), Java’s RMI (Remote Method Invocation), and OMG’s CORBA (Common Object Request Broker Architecture); RMI and CORBA are more secure than SOAP. Central repositories (registries) catalog which services are available and where. using the UDDI (Universal Description Discovery and Integration) protocol. Providers list the usage specifications of their services via the WSDL (Web Services Description Language) protocol. Web service applications can be created in a number of languages. with most written in Java, PHP, or Visual Basic-Net. Some think that the “service-oriented architecture” may become the core paradigm for software applications and integration (Eisenberg, 2004).

The advantages of the CBSE approach are speed of assembly and short-term cost. The disadvantages are that no strategic advantage is derived from the resulting product (nothing is proprietary, anyone can do it), compromise of requirements to meet capabilities of available components, vendor dependencies, possible performance issues, and security problems.

Object-Oriented Software

Object-oriented (OO) programming is a key part of many of the methodologies previously discussed, such as RUP, AP, XP, and CSD. Charles Darwin (1859) postulated that it was not the biggest, smartest, or fastest species that would survive, but the most adaptable. The same is true for application software. Applications must evolve, even before they are completely developed, because the environment under which they operate (business, regulatory, social, political, technical, etc.) changes during the time the software is designed and implemented (see Figure 5.7). In addition, there is the ever-present requirements creep, and even after the application is successfully deployed, there is a constant need for change. Conceptually, the concept of adaptable software is illustrated in Figure 5.13 (as compared to Figure 5.7).

Object-oriented software systems are inherently more adaptable and maintainable than traditional procedural software, and OO systems foster software reuse. "Object technology promises a way to deliver cost-effective, high quality and flexible systems on time to the customer" (McClure, 1996). In fact, out of all the aforementioned methodology variations and development acceleration techniques, OO is the only technique proven to be almost always effective in reducing *long-term* software development costs.

Modern OO design methods and programming languages are based on the concept of a class, which is a type of thing. Once a class is defined, it serves as a mold for making specific objects (instances) of that type. We could, for example, design a class for a student which would indicate the data properties (attributes) and functionality (methods) that is attributable to each specific student object. This is illustrated in Figure 5.14. Here we have defined a Rectangle class that has two properties: length and width, and

Figure 5.13. Software adapting to problem changes

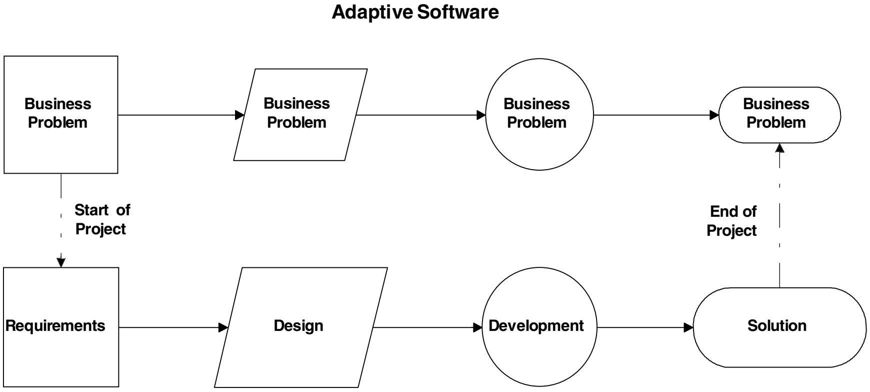
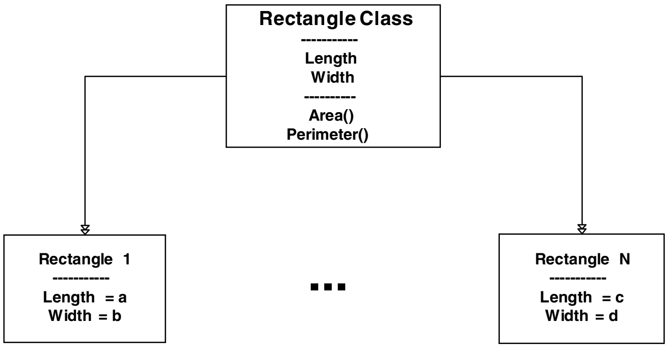


Figure 5.14. Class instances (objects)



also two methods: Area and Rectangle. Every instance of a Rectangle (each object) will have specific values for length and width.

OO classes have a property called encapsulation, which is a way to protect the properties from unauthorized access from other code (functions) in a software system. This is illustrated in Figure 5.15, in which the width and length data is protected. To find the value for one of an object's properties, a "get" method must be used, and to change the value of an object's properties, a set method must be used. These methods can be coded with whatever access protection is necessary.

Figure 5.16 uses an UML (unified modeling language) diagram to show the main relationships involved in OO analysis, design, and programming. As well as the classification relationship (an object is classified as being an instance of a class), there are composition and inheritance relationships. An object can be composed (physically or logically) of other objects, and this composition is also a part of the class definition.

Figure 5.17 illustrates the inheritance relationship. Here we have introduced another class, Positioned Rectangle, which inherits from (is derived from) the Rectangle class. The Rectangle is called the base or super class, and the Positioned Rectangle is called the derived or subclass. A derived class has the same properties and methods as were defined in the base class, but more properties and methods can be included. In this example, the Positioned Rectangle adds two more properties and one more method.

Polymorphism is another property of OO systems, wherein a derived class can alter the behavior (code inside) of a method of the base class with the same name (signature: name and arguments). For some OO languages, the actual mechanisms involve more complex

Figure 5.15. Encapsulation

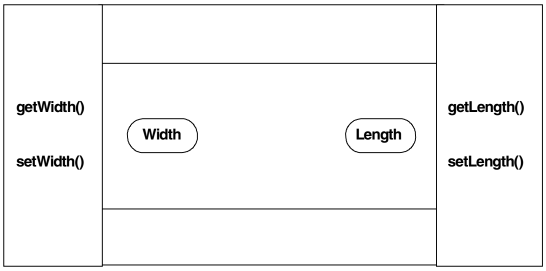


Figure 5.16. Object-oriented relationships

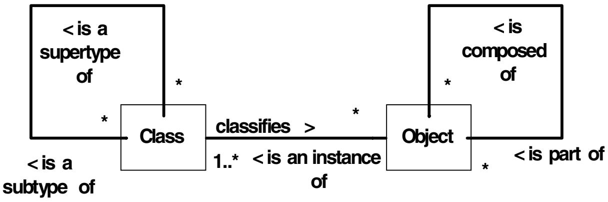


Figure 5.17. Class-based inheritance

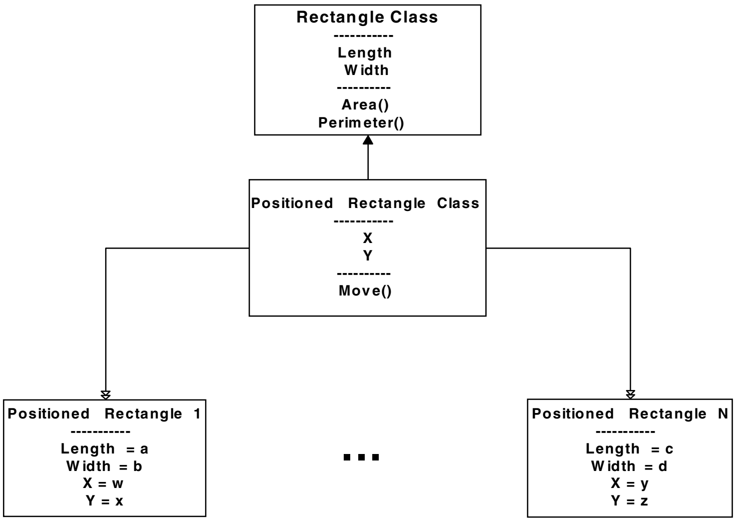
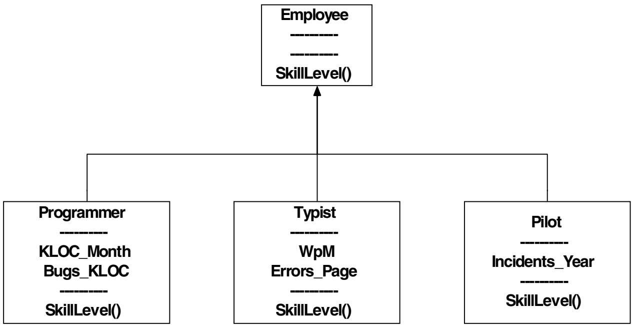


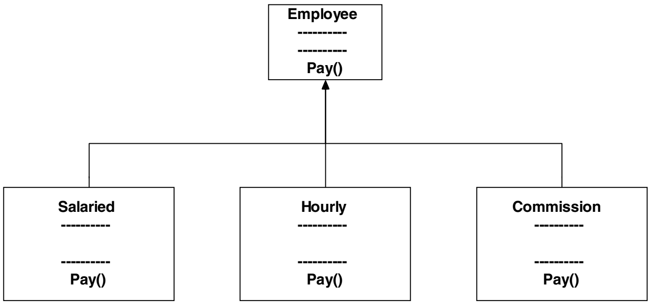
Figure 5.18. Polymorphism



notions, such as virtual functions and/or late/early binding. This is illustrated in Figure 5.18, in which the skill-level function (e.g., which might return the skill level as an integer) has a different meaning in each of the three subclasses of Employee. The code in the skill-level function would be different in each of the three sub classes and presumably use the properties which could also be different in each subclass.

As an example of the power of OO systems in contrast to non-OO systems, consider the code for a payroll system. In this system, there are three types of employees: salaried, hourly, and commission based. So we would have three subclasses of Employee, each with its own version of the Pay function. This is illustrated in Figure 5.19.

Figure 5.19. Subtypes of employees



To perform the payroll operation in an OO system, we would walk through our database or data structure of employees and invoke the Pay function in each instance. In a non-OO system, we cannot use the same function name with different code therein, so in such a traditional non-OO language, we would have to use conditional code such as if, go-to, or switch/case constructs. We would need to say something like, If the employee were salaried, then invoke the salary-pay function. If the employee were hourly, then invoke the hourly-pay function. If the employee were commissioned, then invoke the comm-pay function. This may not seem important, but consider the evolution and maintenance problems.

Payroll may be just one of the operations in a huge human resources system of hundreds of thousand of lines of code. Many of the operations therein have conditional code, because the nature of the operation is different depending upon the type of employee. That particular conditional code is implemented within the operations, and conditional code is spread throughout the system. To make a change to the logic, you must first find all the conditional constructs that need to be changed. In an OO system, we just change the method in question in each class.

Now suppose we need to add another type of employee, one that is a piece-work laborer. With a non-OO system, we would have to go to possibly every conditional construct and add another case. Of course, we first have to find each one. With an OO system, all we have to do is add another class. OO systems are also safely modifiable, because when we add that additional class we are not involved with the other code in the system. In the non-OO systems, we have to examine and possibly modify much of the code in the system, and chances are (40%) that we are going to create a problem elsewhere while trying to solve the original problem.

SIMULA was the first object-oriented language (SIMULA I was developed in 1962-1965, and SIMULA 67 was developed in 1967). It was based somewhat on the ALGOL 60 programming language, and it introduced most of the key modern concepts of object-oriented programming, including objects, classes, subclasses, and inheritance. SIMULA was developed at the Norwegian Computing Center by Ole-Johan Dahl and Kristen Nygaard, and it quickly got a reputation as a simulation programming language. Later, it was found to possess interesting properties as a general programming language when the inheritance mechanism was used. In 1968, Edsger Dijkstra wrote his famous letter titled GO TO Statement Considered Harmful, and this letter was perhaps the first stone cast in the battle against spaghetti code. SIMULA compilers were developed for UNIVAC, IBM, Control Data, Burroughs, DEC and other computers throughout the 1970s.

SIMULA is still used around the world, but its main impact was the introduction of the modern OO programming principles. Alan Kay in 1970 was the person to coin the terms *object oriented* and *object-oriented programming.* His group at Xerox PARC (Palo Alto Research Group) used SIMULA as a platform for their development of Smalltalk (initially developed around 1973), extending object-oriented programming by the integration of graphical user interfaces and interactive program execution. In 1975, Marvin Minsky introduced frames in artificial intelligence (AI) programming, which were an ancestor to modern objects implemented later in languages such as C++. Brian W. Kernighan and Dennis M. Ritchie published *The C Programming Language* in 1978, after years of development. Although not object-oriented itself, C heavily influenced and formed the basic syntax and structure of most modern OO programming languages, such as C++, Java, C#, and PHP.

In the 1980s, at Bell Labs, Bjarne Stroustrup started his development of C++ (originally called "C with classes") by bringing the key concepts of SIMULA into the C programming language. In the 1980s, considerable resources were invested in the ADA language by the U.S. Department of Defense and in PROLOG via the Japanese Fifth Generation Computer Project. ADA had some basic OO capabilities and extended the Pascal language with strong typing, which reduced the occurrence of many types of programming errors. Many computer experts initially believed that ADA, PROLOG, and Smalltalk (with its Xerox and IBM backing) would fight for dominance in the 1990s, but as object-oriented programming became the dominant style for implementing complex programs with large numbers of interacting components (such as in GUI class libraries), more capable and flexible languages as C++ raced to the forefront. In 1989 the Object Management Group (OMG) was founded. It is committed to developing vendor independent OO specifications for the software industry.

In 1991 Sun Microsystems developed the Java programming language as part of a research project to create software for consumer electronic devices like TVs and VCRs. It contains many object-oriented programming features similar to C++ and was extended to easily handle Web and multimedia type of applications. In 1994, Rasmus Lerdorf created the initial version of Hypertext Preprocessor (PHP), an open-source Web server scripting language with object-oriented capabilities similar to C++ and Java. A number of other OO languages have some adopters (i.e., Eiffel, CLOS, SELF), however, with the current push to rewrite and convert applications to a Web-centric environment, even C++ is being replaced by more web enabled and standards embracing languages such as Java and PHP.

Because one uses a modern object-oriented language (i.e., C++, Java, or PHP) does not necessarily mean that one has written an object-oriented program. One can still build poor, non–object-oriented and nonreusable software with a fully object-oriented language.

Software Reuse

Today's software development is characterized by many disturbing but well-documented facts:

- The supply of qualified IT professionals is less than the demand
- The complexity of software is constantly increasing
- IT needs better-cheaper-faster software development methods
- Most software development projects fail (Standish Group, 2004; Williamson, 1997)

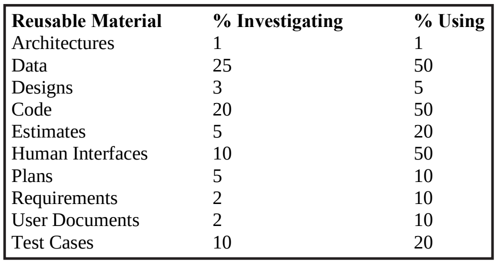
Granneman asked, "Why is this IT project so expensive?" (Granneman, 2004). His answer was that software is designed without much regard for future changes, even easily foreseeable changes. He suggested asking this question as part of the project design—Will this software need programming changes if:

- Business conditions change?
- Business partners change?
- Economic conditions change?
- Vendor relationships change?
- Banking relationships change?
- Products or services are added, changed, or discontinued?
- Business activity increases or decreases dramatically?
- The timing of how quickly information is needed changes?
- The source or destination of information changes?

"Reuse [software] engineering is a process where a technology asset is designed and developed following architectural principles, and with the intent of being reused in the future" (Bean, 1999). "If programming has a Holy Grail, widespread code reuse is it with a bullet. While IT has made and continues to make laudable progress in our reuse, we never seem to make great strides in this area" (Grinzo, 1998). "The quest for that Holy Grail has taken many developers over many years down unproductive paths" (Bowen, 1997).

Jones (1994) discussed the problem with the lack of reusability and itemized reusable material and the percentage of companies investigating and using the same. This is detailed in Figure 5.20. Although these numbers are somewhat outdated, the table illustrates the kinds of IT artifacts that can be reused and the relative percentage utilization of each. Jones also indicated that the amount of risk in a project is in inverse proportion to the amount of reuse. A root cause of the lack of reusability is that software development has evolved as a "craft" rather than as an engineering or manufacturing. The discipline of software engineering is an attempt to correct this historic evolution.

Figure 5.20. Reusable artifacts



Reusable Material	% Investigating	% Using
Architectures	1	1
Data	25	50
Designs	3	5
Code	20	50
Estimates	5	20
Human Interfaces	10	50
Plans	5	10
Requirements	2	10
User Documents	2	10
Test Cases	10	20

In his book, Jones (1994) itemized and discussed the risks from a lack of reusability in each of the above areas, and he indicated root causes, cost impact, and methods of prevention and control for each. Although outdated, many of the general principles still apply. “The bottom line is this: while it takes time for reuse to settle into an organization—and for an organization to settle on reuse—you can add increasing value throughout the process” (Barrett & Schmuller, 1999).

Radding defines several different types of reusable components in IT business systems (Radding, 1998):

- *GUI Widgets:* Effective, but only provide modest payback
- *Server-Side Components:* Provide significant payback but require extensive up-front design and an *architectural foundation*
- *Infrastructure Components:* Generic services for transactions, messaging, and database … require extensive design and complex programming
- *High-Level Patterns:* Identify components with high reuse potential
- *Packaged Applications:* Only guaranteed reuse, however may not offer the exact functionality required

For all of the above types of reusable components, except packaged applications, OO programming is the most effective architectural and programming technique. Packaged applications and COTS software were identified and discussed earlier in this chapter. Reusing code has several key implementation areas: application evolution, multiple implementations, standards, and new applications. The reuse of code from prior applications in new applications has received the most attention. Just as important, however, is the reuse of code (and the technology embedded therein) within the same application. As stated earlier, applications must evolve even before they are completely developed, because the environment under which they operate (business, regulatory, social, political, technical, etc.) changes during the time the software is designed and implemented.

Another key need for reusability within the same application is for multiple implementations, the most common of which involves customizations, internationalization, and multiple platform support. For example, organizations whose software must be utilized globally might need to present an interface in customers' native languages and with a socially acceptable look and feel (localization). The multiple platform dimension of reuse today involves an architectural choice in delivery platforms (hardware and operating system) on both the client and server sides.

Corporate software development standards concern maintaining standards in all parts of an application and maintaining standards across all applications. For a computer system to have lasting value it must exist compatibly with users and other systems in an ever-changing information technology (IT) world (Brandon, 2002). Software reuse and OO programming as it relates to standards are covered in more detail in a later chapter of this book.

In most organizations, software reusability is a goal that is very elusive,"a most difficult promise to deliver on" (Bahrami, 1999). Radding stated, "Code reuse seems to make sense, but many companies find there is so much work involved, it's not worth the effort. … In reality, large scale software reuse is still more the exception than the rule" (Radding, 1998). Bean, in "Reuse 101," stated that the current decreased "hype" surrounding code reuse is likely due to three basic problems:

- Reuse is an easily misunderstood concept
- Identifying what can be reused is a confusing process
- Implementing reuse is seldom simple or easy to understand (Bean, 1999)

Grinzo (1998) also list several reasons and observations on the problem of reuse, other than for some "difficult to implement but easy to plug-in cases," such as GUI widgets; a "nightmare of limitations and bizarre incompatibilities"; performance problems; "thorny psychological issues" involving programmers' personalities; market components that are buggy and difficult to use; fear of entrapment; component size; absurd licensing restrictions; or lack of source code availability.

Some organizations try to promote software reusability by simply publishing specifications on class libraries that have been built for other in house applications or that are available via third parties, some dictate some type of reuse, and other organizations give away some type of "bonus" for reusing the class libraries of others (Bahrami, 1999).

But more often than not, these approaches do not result in much success.

"It's becoming clear to some who work in this field that large-scale reuse of code represents a major undertaking" (Radding, 1998). "An OO/reuse discipline entails more than creating and using class libraries. It requires *formalizing* the practice of reuse" (McClure, 1996).

There are two key components to *formalizing an effective software reuse practice* (Brandon, 2002):

1. Defining a specific information technology architecture within which applications would be developed and reuse would apply
2. Defining a very specific object-oriented “Reuse Foundation” that would be implemented within the chosen IT architecture. Such a foundation is typically the combination of an overall framework and specific reusable patterns.

“If you want reuse to succeed, you need to invest in the architecture first” (Radding, 1998). “Without an architecture, organizations will not be able to build or even to buy consistently reusable components.” The major architectures today are Java Two Enterprise edition (J2EE), Microsoft’s .Net, and open source LAMP (Linux, Apache, MySQL, PHP). Object-oriented frameworks are available from a number of vendors for each architecture, or an organization can create its own framework and patterns. Modern architectures may support one or more application servers, programming languages and IDEs (integrated development environments).

Application frameworks are a holistic set of specifications for the interaction and assembly of multiple reusable patterns. A pattern is the design of a core functional element such as the MVC (model-view-controller) pattern used for user interfaces. The boundary between architecture, framework, pattern, and programming language is blurry and not the same in different architectures. Examples of modern proprietary application frameworks include IBM’s Websphere, Macromedia’s ColdFusion and Flex, Sun’s I-Planet, and BEA’s Weblogic.

These reuse foundations (frameworks and patterns) are based on the key object-oriented principles of inheritance and composition. By establishing foundations like these, an organization can effectively begin to obtain significant reusability since programmers must inherit each of their classes from one of the established classes and they must only compose their classes of the established pre-built components. *As has been concluded by several authors, “A reuse effort demands a solid conceptual foundation” (Barrett, 1999).*

Software Engineering Institute

The Software Engineering Institute (SEI; www.sei.cmu.edu/cmm) is a research institute funded by the U.S. Department of Defense (DoD), contracted to Carnegie Mellon University, which was started in 1984. The SEI receives tens of millions of DoD dollars on an annual basis. Their overall goal is to advance the practice of software engineering, and they are perhaps best known for their formulation of software engineering “maturity models.” These models, called capability maturity models (CMM), define best practices—key practices (things to be done and ways of doing things) that organizations at different levels of software engineering “maturity” do.

A popular baseball analogy was first reportedly expressed by Watts Humphrey, known as the Father of CMM:

- *Immature Team:* When the ball is hit, some players run toward the ball and others stand around and watch, perhaps not even thinking about the game.
- *Mature Team:* When the ball is hit, every player reacts in a predefined disciplined manner. Depending upon the situation, the pitcher might cover home plate, infielders might set up a double play, and outfielders might back up their teammates.

The SEI has formulated a number of CMM over the years, including,

- SW-CMM: CMM for Software Development
- SA-CMM: Software Acquisition CMM
- P-CMM: People CMM
- SE-CMM: Software Engineering CMM
- IPD-CMM: Integrated Product Development CMM

The Software Capability Maturity Model (CMM for Software) was the first, best known, and probably used the most today. It defines five levels of software process maturity that determine effectiveness in delivering quality software:

- Initial
- Repeatable
- Defined
- Managed
- Optimized

It is primarily geared to large organizations and their contractors, however, many of the processes involved are appropriate to any organization, and if reasonably applied can be helpful. Organizations can receive CMM ratings by undergoing assessments by qualified auditors. *These ratings are useful to an organization for two reasons. First the assessment lets an organization know where it stands in terms of software engineering maturity as viewed by an independent source, and secondly the assessment (if very good) can be used by the organization in selling its services*. A description of each level follows:

- *Level 1 (Anarchy):* At this level, programmers generally do what they individually think best. Chaos, periodic panics, and heroic efforts are often required by individuals to successfully complete projects and successes is typically not repeatable. Cost, schedule, and quality are unpredictable. There is little formal planning or established programming practices. Overcommitment is common, and senior management does not understand application development/procurement.

- *Level 2 (Folklore):* Here programmers have experience developing certain kinds of applications. They have devised effective processes (policies and procedures in regard to project management, requirements, and configuration), and generally make time and cost estimates. Their strength depends upon doing the same kind of application, but they cannot adapt well to new applications, new methods, or new tools. Knowledge is only in heads of programmers.
- *Level 3 (Standards):* Here the corporate "mythology" is written down in a set of standards. These standard software development and maintenance processes are integrated throughout the organization and some sort of a software engineering process group is in place to oversee these processes. Groups may tailor the standards with approval, however the process has not been measured (by collecting data) or compared to other methods. Since it is not measured, programmers and managers debate the effectiveness of the metrics that are used to track productivity, processes, and products.
- *Level 4 (Managed):* Here project performance is predictable, and quality is consistently high. Metrics have been established, and hard data is collected to access process's effectiveness. Measurements are used to improve the product quality.
- *Level 5 (Optimized):* Here the focus is on continuous process improvement. The impact of new processes and technologies can be predicted and effectively implemented when required. Tools are available to automate collection of data, and measurements are used to improve the processes.

Each maturity level, except for the first, is broken down into key process areas. Each key process is described in terms of key practices. The key process areas at each level are:

Level 2 Key Process Areas [Basic Project Management]

Requirements Management
Software project planning
Software project tracking and oversight
Software subcontract management
Software quality assurance
Software configuration management

Level 3 Key Process Areas [Organizational Processes Standardization]

Organization process focus
Organization process definition
Training program
Integrated software management
Software product engineering
Intergroup coordination
Peer reviews

Level 4 Key Process Areas [Quantitative Process Analysis - Metrics]

Quantitative process management
Software quality management

Level 5 Key Process Areas [Continuous Improvement of Entire Process]

Defect prevention
Technology change management
Process change management

Note that the SEI CMM Level 2 key process areas encompass basic project management. We will look more into these level two processes in later chapters of this book. For convenience, the practices at all the levels are organized by common features, which address the level of implementation; these common features are:

- Commitment to perform
- Ability to perform
- Activities performed
- Measurements and analysis
- Verifying implementation

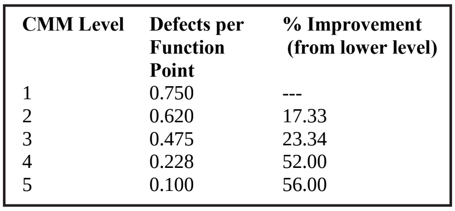
Note that these common features are embedded into our model for IT project critical success completion factors as described in Chapter II (in our model, verification assumes measurement and analysis).

The SEI is currently revising the software SW-CMM into a more comprehensive CMMI integrated model that will also encompass systems engineering and product development. Higher CMM levels have correlated with less software defects and higher cost savings (in terms of function points, which are discussed later in this book). Figure 5.21 shows these types of data, as published in *Computerworld (*King, 2003).

During the last 5 years, about 1,000 organizations have been assessed, and 27% were rated at Level 1, 39% at Level 2, 23% at Level 3, 6% at Level 4, and 5% at Level 5. Currently there are about 70 companies worldwide that are at Level 5 (King, 2003).

However, being at Level 5 does not guarantee that a company's internal implementation of these standards is best in class (King, 2003). The CMM standards describe what must be done, not how to do it. Remember also that "the CMM is a consensus among a particular group of software engineering theorists and practitioners concerning a collection of effective practices grouped according to a simple model of organizational evolution. As such, it is potentially valuable for those companies that completely lack software savvy, or for those who have a lot of it and want to avoid its pitfalls. At worse, the CMM is a whitewash that obscures the true dynamics of software engineering, suppressing alternative models" (Bach, 1994). A notable alternative model of software maturity is that of Jones: "Software Productivity Research" (Jones, 1994).

Figure 5.21. CMM levels and improvements



CMM Level	Defects per Function Point	% Improvement (from lower level)
1	0.750	---
2	0.620	17.33
3	0.475	23.34
4	0.228	52.00
5	0.100	56.00

Institute of Electrical and Electronics Engineers

The Institute of Electrical and Electronics Engineers (IEEE) is one of the world's largest professional organizations with over 350,000 members in over 150 countries. About one half of the IEEE members are outside of the United States, and that portion is growing more rapidly than the U.S. membership. IEEE publishes about one quarter of the world's literature within the technical fields it encompasses. The IEEE Computer Society is the largest of the 36 technical societies in IEEE, with over 100,000 members. The IEEE Computer Society is in the final stages of completing and approving a Software Engineering Body of Knowledge (SWEBOK). The knowledge areas to be covered include:

- Professional engineering economics
- Software requirements
- Software design
- Software construction and implementation
- Software testing
- Software maintenance
- Software configuration management
- Software engineering management
- Software engineering process
- Software engineering tools and methods
- Software quality

Most of the SWEBOK has already been documented within the various IEEE software standards, and many of these standards are discussed later in this book. These IEEE software standards include:

- 730-2002 IEEE Standard for Software Quality Assurance Plans
- 828-1998 IEEE Standard for Software Configuration Management Plans
- 829-1998 IEEE Standard for Software Test Documentation
- 830-1998 IEEE Recommended Practice for Software Requirements Specifications
- 982.1-1988 IEEE Standard Dictionary of Measures to Produce Reliable Software
- 1008-1987 IEEE Standard for Software Unit Testing
- 1012-1998 IEEE Standard for Software Verification and Validation
- 1016-1998 IEEE Recommended Practice for Software Design Descriptions
- 1028-1997 (R2002) IEEE Standard for Software Reviews
- 1044-1993 (R2002) IEEE Standard Classification for Anomalies
- 1045-1992, (R2002) IEEE Standard for Software Productivity Metrics
- 1058-1998 IEEE Standard for Software Project Management Plans
- 1058.1-1987 (R1993) IEEE Standard for Software Project Management Plans
- 1061-1998 (R2004) IEEE Standard for Software Quality Metrics Methodology
- 1062, 1998 Edition (R2002) IEEE Recommended Practice for Software Acquisition (includes IEEE 1062a)
- 1063-2001 IEEE Standard for Software User Documentation
- 1074-1997 IEEE Standard for Developing Software Life Cycle Processes
- 1175.1-2002 IEEE Guide for CASE Tool Interconnections-Classification and Description
- 1219-1998 IEEE Standard for Software Maintenance
- 1220-1998 IEEE Standard for the Application and Management of the Systems Engineering Process
- 1228-1994 (2002) IEEE Standard for Software Safety Plans
- 1233, 1998 Edition (R2002) IEEE Guide for Developing System Requirements Specifications (including IEEE 1233a)
- 1320.1-1998 (R2004) IEEE Standard for Functional Modeling Language - Syntax and Semantics for IDEF0
- 1320.2-1998 (R2004) IEEE Standard for Conceptual Modeling Language Syntax and Semantics for IDEF1X97 (IDEF object)
- 1420.1-1995 (R2002) IEEE Standard for Information Technology—Software Reuse—Data Model for Reuse Library Interoperability: Basic Interoperability Data Model (BIDM)
- 1420.1a-1996 (R2002) IEEE Supplement to Standard for Information Technology—Software Reuse—Data Model for Reuse Library Interoperability: Asset Certification Framework
- 1420.1b-1999 (R2002) IEEE Supplement to IEEE Standard for Information Technology—Software Reuse—Data Model for Reuse Library Interoperability: Intellectual Property Rights Framework

- 1462-1998 [Adoption of International Standard ISO/IEC 14102:1995(E)], Information technology — Guideline for the evaluation and selection of CASE tools
- 1465-1998 [Adoption of ISO/IEC 12119: 1994(E)], IEEE Standard Adoption of International Standard ISO/IEC 12119: 1994(E) Information Technology—Software Packages: Quality requirements and testing
- 1490-2003 IEEE Guide (©IEEE) — Adoption of PMI Standard—A Guide to the Project Management Body of Knowledge (©PMI)
- 1517-1999 (R2004) IEEE Standard for Information Technology—Software Life Cycle Processes—Reuse Processes
- 2001-2002 IEEE Recommended Practice for the Internet—Web Site Engineering, Web Site Management, and Web Site Life Cycle
- 1540-2001 IEEE Standard for Software Life Cycle Processes-Risk Management
- 12207.0-1996 IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology—Software Life Cycle Processes
- 12207.1-1997 IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology—Software Life Cycle Processes—Life Cycle Data
- 12207.2-1997 IEEE/EIA Standard: Industry Implementation of International Standard ISO/IEC 12207:1995 Standard for Information Technology— Software Life Cycle Processes—Implementation considerations

Other Software Standards Organizations

There are about 50 other organizations worldwide that produce software engineering standards. Some of these are the International Organization for Standardization (IOS), the American National Standards Institute (ANSI), World Wide Web Consortium (W3C) for Internet-related standards, NIST which is an agency of the U.S. Commerce Department's Technology Office, space agencies such as NASA and military organizations such as the U.S. Department of Defense (DoD). The American National Standards Institute (ANSI) is a private, nonprofit organization that is the focal point for the U.S. voluntary consensus standards system. ANSI consists of approximately 1,300 national and international companies as well as many government agencies, institutional members, professional, technical, and trade organizations. ANSI facilitates a consensus amongst its members to foster ANSI accredited standards. A key part of the approval process is the fact that all members have the opportunity to participate in the standards development process.

However, the most dominant of these other organizations worldwide is ISO/IEC, the software engineering subcommittee of the International Organization for Standardization. The International Organization for Standardization (ISO, from the Greek *isos*, meaning equal) has set international standards (about 14,000 of them) in many areas for many years, including software engineering standards. ISO has over 180 technical committees, covering many industry sectors and products. The American Society for Quality Control (ASQC) handles the U.S. Technical Advisory Group (TAG), which offers its opinions to the overall ISO technical committees. Many ISO standards are shared ("adopted") by IEEE, and are so noted in the above list of IEEE standards. The ISO/IEC 12207 standard provides a total framework for the acquisition, supply, development, operation, and maintenance of software. In addition, the standard provides a methodology for managing software life cycle activities and a reference point for new and emerging engineering standards. ISO/IEC 12207 has been adopted/adapted by ANSI, IEEE, EIA, and DoD in the United States.

The European Software Institute is a major industry initiative, founded by leading European companies, to improve the competitiveness of the European Software Industry. To this end ESI promotes good software engineering and management practice. Since 1993, the Software Process Improvement and Capability determination (SPICE) project, launched within ISO, has been developing a framework standard for software process assessment. ESI, is a key partner in SPICE, and is taking the leading role in the European adaptation of SPICE.

Chapter Summary

Earlier in this book, the three challenges to software engineering in the 21st century were outlined (Sommerville, 2003):

- *The Heterogeneity Challenge:* Flexibility to operate on and integrate with multiple hardware and software platforms from legacy mainframe environments to the landscape of the global web
- *The Delivery Challenge:* Ability to develop and integrate IT systems rapidly in response to rapidly changing and evolving global business needs
- *The Trust Challenge:* Being able to create vital (mission and/or life critical) software that is trustworthy in terms of both security and quality

These challenges can be met by a careful integration of modern project management and software engineering principles and practices. This was illustrated in Figure 1.4 in Chapter I, and this chapter has discussed software engineering maturity, methodologies, and OO architecture. In Chapter VI, project scope, phasing, and requirements are discussed; later chapters of this book will further detail these other principles and practices. Throughout the book, critical IT project success factors are used as the basis for key project management processes as performance, risk, and quality control.

References

Bach, J. (1994, September). The immaturity of CMM. *American Programmer*.

Bahrami, A. (1999). *Object oriented systems development*. New York: McGraw Hill-Irwin.

Barrett, K., & Schmuller, J. (1999, October). Building an infrastructure of real-world reuse. *Component Strategies*.

Bean, J. (1999, October). Reuse 101. *Enterprise Development*.

Beck, K. (1999). *Extreme programming explained: Embrace change.* Boston: Addison-Wesley Professional.

Bauer, F. (1972). Software engineering. *Information Processing, 71.*

Bowen, B. (n.d.). Software reuse with Java technology: Finding the Holy Grail. Retrieved from www.javasoft.com/features/1997/may/reuse.html

Brandon, D. (2002). Achieving effective software reuse for business systems. In *Successful software reengineering.* Hershey, PA: Idea Group Publishing.

Cockburn, A. (2001). *Agile software development*. Boston: Addison-Wesley.

Darwin, C. (1859). *The origin of species by means of natural selection (or the preservation of favoured races in the struggle for life)*.

Dennis, A. (1999, Spring). Business process modeling with group support systems. *Journal of Management Information Systems,* 115-142.

Eisenberg, R. (2004, April 17). Service-oriented architecture: The future is now. *Intelligent Enterprise*.

Garbett, S. (2003, August). Cleanroom software engineering. *Dr. Dobb's Journal.*

Granneman, M. (2004). Why is this IT Project so expensive? *Computerworld*, June 4.

Grinzo, L. (1998, September). The unbearable lightness of being reusable. *Dr. Dobbs Journal.*

Hall, M. (2003, May 19). The Web services tsunami. *Computerworld.*

Highsmith, J. (2000). *Adaptive software development*. New York: Dorset House.

Jacobson, I. (1999). *The unified software development process.* Boston, MA: Addison-Wesley Professional.

Jones, C. (1994). *Assessment and control of software risks*. Englewood Cliffs, NJ: Yourdon Press Computing Series.

King, J. (2003, December 8). The pros and cons of CMM. *Computerworld.*

Kruchten, P. (1998). *The rational unified process.* Boston: Addison-Wesley.

Linger, R. (1994). Cleanroom process model. *IEEE Software, 11*(2).

McClure, C. (1996). Experiences from the OO playing field. *Extended Intelligence.*

Mills, H. (1996). *Cleanroom software engineering*. Oxford, UK: Blackwell Publishers.

Palmer, S., & Felsing, J. (2002). *A practical guide to feature-driven development.* New York: Prentice Hall.

Prowell, S. (1999). *Cleanroom software engineering: Technology and process.* Boston: Addison-Wesley.

Radding, A. (1998, November 9). Hidden cost of code reuse. *Information Week*.

Royce, W. (1970). Managing the development of large software systems. *Proceedings IEEE WESTCON, IEEE Computer Society,* Los Angeles, CA.

Schwaber, K., & Beedle, M. (2001). *Agile software development with Schrum.* Upper Saddle River, NJ: Prentice Hall.

Sommerville, I. (2003). *Software engineering.* Boston: Pearson Addison Wesley.

Standish Group. (2004). *Chaos chronicles*. Retrieved from www.standisgroup.com

Stapleton, J. (1997). *DSDM dynamic systems development method*. Boston: Addison-Wesley.

Williams, L., & Kessler, R. (2000). Strengthening the case for pair programming. *IEEE Software, 17*(4), 19-25.

Williamson, M. (1997, May). Software reuse. *CIO Magazine.*