PREVIOUS < Free Open Study > NEXT

Software Development Life Cycle Models

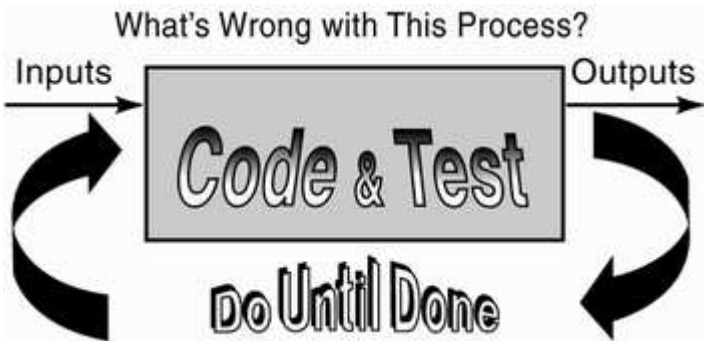
The most well-known and widely used software development life cycles include: the waterfall, V-shaped, evolutionary rapid prototyping, rapid application development, incremental, and spiral models. In the following sections, each model will be described; strengths and weaknesses of each will be discussed, and examples of modified models and guidance on tailoring will be provided. Tables 4-1 through 4-4 will offer criteria to assist the project manager in selecting an appropriate life cycle model for a given project.

The Waterfall Software Development Life Cycle Model

The "classic" waterfall model, despite recent bad press, has served the software engineering community well for many years. Understanding its strengths and flaws improves the ability to assess other, often more effective life cycle models that are based on the original.

In the earliest days of software development, code was written and then debugged. It was common to forego planning altogether and, starting with a general idea of the product, informally design, code, debug, and test until the software was ready for release. The process looked something like the one in Figure 4-8. There are several things "wrong" with such a process (or lack thereof). Primarily, because there was no formal design or analysis, it is impossible to know when you are done. There is no way to assess whether the requirements or the quality criteria have been satisfied.

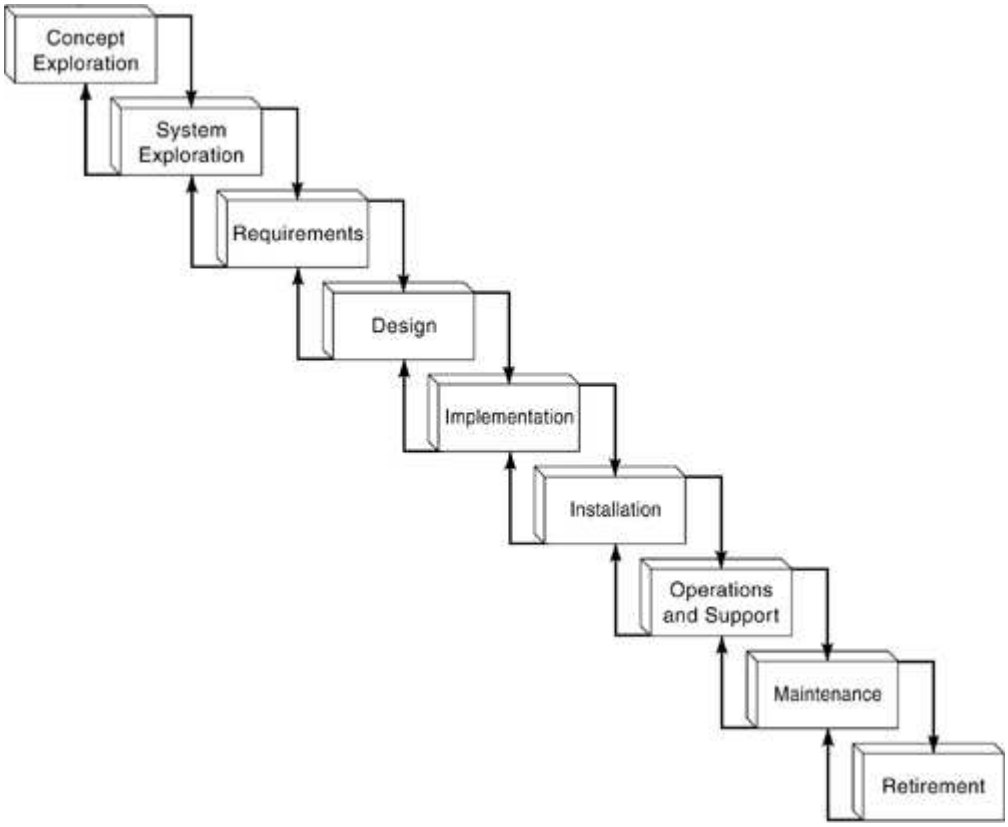
Figure 4-8. "Do Until Done" Process Model



The waterfall model was first identified in 1970 as a formal alternative to the *code-and-fix* software development method prevalent at the time. It was the first to formalize a framework for software development phases, placing emphasis on up-front requirements and design activities and on producing documentation during early phases.

Execution of the waterfall model begins at the upper left of Figure 4-9 and progresses through the orderly sequence of steps. It assumes that each subsequent phase will begin when activities in the current phase have been completed. Each phase has defined entry and exit criteria: inputs and outputs. Internal or external project deliverables are output from each phase, including documentation and software. Requirements analysis documents are passed to system engineers, who hand off high-level design documents to software architects, who hand off detailed specifications to coders, who hand off code to testers.

Figure 4-9. Classic Waterfall Model with Feedback



Transition from one phase to the next is accomplished by passing a formal review, a way to provide customers insight into the development process and to check on product quality. Typically, passing the review indicates an agreement among the project team members and the customer that the phase has ended and the next one can begin. The end of a phase is a convenient place for a project milestone.

In conjunction with certain phase completions, a baseline is established that "freezes" the products of the development at that point. If a need is identified to change these products, a formal change process is followed to make the change.

At critical points on the waterfall model, baselines are established, the last of which is the product baseline. This final baseline is accompanied by an acceptance review.

Other software development life cycles have evolved from attempts to optimize the waterfall model. Software prototyping helps provide the complete understanding of the requirements; the incremental and spiral models allow the phases identified in the classic waterfall to be revisited repeatedly before declaring a product to be final.

The salient attributes of the waterfall model are that it is a formal method, a type of top-down development, composed of independent phases executed sequentially and subject to frequent review.

A Brief Description of the Phases in the Waterfall Model

The following is a brief description of each phase of the waterfall model, from concept exploration through retirement, including the integration phases:

- **Concept exploration**— Examines requirements at the system level, to determine feasibility.

- **System allocation process**— May be skipped for software-only systems. For systems that require the development of both hardware and software, the required functions are mapped to software or hardware based on the overall system architecture.

- **Requirements process**— Defines software requirements for the system's information domain, function, behavior, performance, and interfaces. (Where appropriate, this includes the functional allocation of system requirements to hardware and software.)

- **Design process**— Develops and represents a coherent, technical specification of the software system, including data structures, software architecture, interface representations, and procedural (algorithmic) detail.

- **Implementation process**— Results in the transformation of the software design description to a software product. Produces the source code, database, and documentation constituting the physical transformation of the design. If the software product is a purchased application package, the major activities of implementation are the installation and testing of the software package. If the software product is custom developed, the major activities are programming and testing code.

- **Installation process**— Involves software being installed, checked out, and formally accepted by the customer, for the operational environment.

- **Operation and support process**— Involves user operation of the system and ongoing support, including providing technical assistance, consulting with the user, recording user requests for enhancements and changes, and handling corrections or errors.

- **Maintenance process**— Concerned with the resolution of software errors, faults, failures, enhancements, and changes generated by the support process. Consists of iterations of development, and supports feedback of anomaly information.

- **Retirement process**— Removing an existing system from its active use, by either ceasing its operation or replacing it with a new system or an upgraded version of the existing system.

- **Integral tasks**— Involves project initiation, project monitoring and control, quality management, verification and validation, configuration management, document development, and training throughout the entire life cycle.

Strengths of the Waterfall Model

We can see that the waterfall model has many strengths when applied to a project for which it is well suited. Some of these strengths are:

- The model is well known by nonsoftware customers and end-users (it is often used by other organizations to track nonsoftware projects).

- It tackles complexity in an orderly way, working well for projects that are well understood but still complex.

- It is easy to understand, with a simple goal—to complete required activities.

- It is easy to use as development proceeds one phase after another.

- It provides structure to a technically weak or inexperienced staff.
- It provides requirements stability.
- It provides a template into which methods for analysis, design, code, test, and support can be placed.
- It works well when quality requirements dominate cost and schedule requirements.
- It allows for tight control by project management.
- When correctly applied, defects may be found early, when they are relatively inexpensive to fix.
- It is easy for the project manager to plan and staff.
- It allows staff who have completed their phase activities to be freed up for other projects.
- It defines quality control procedures. Each deliverable is reviewed as it is completed. The team uses procedure to determine the quality of the system.
- Its milestones are well understood.
- It is easy to track the progress of the project using a timeline or Gantt chart—the completion of each phase is used as a milestone.

Weaknesses of the Waterfall Model

We can also note weakness of the model when it is applied to a project for which it is not well suited:

- It has an inherently linear sequential nature—any attempt to go back two or more phases to correct a problem or deficiency results in major increases in cost and schedule.
- It does not handle the reality of iterations among phases that are so common in software development because it is modeled after a conventional hardware engineering cycle.
- It doesn't reflect the problem-solving nature of software development. Phases are tied rigidly to activities, not how people or teams really work.
- It can present a false impression of status and progress—"35 percent done" is a meaningless metric for the project manager.
- Integration happens in one big bang at the end. With a single pass through the process, integration problems usually surface too late. Previously undetected errors or design deficiencies will emerge, adding risk with little time to recover.
- There is insufficient opportunity for a customer to preview the system until very late in the life cycle. There are no tangible interim deliverables for the customer; user responses cannot be fed back to developers. Because a completed product is not available until the end of the

process, the user is involved only in the beginning, while gathering requirements, and at the end, during acceptance testing.

- Users can't see quality until the end. They can't appreciate quality if the finished product can't be seen.
- It isn't possible for the user to get used to the system gradually. All training must occur at the end of the life cycle, when the software is running.
- It is possible for a project to go through the disciplined waterfall process, meet written requirements, but still not be operational.
- Each phase is a prerequisite for succeeding activities, making this method a risky choice for unprecedented systems because it inhibits flexibility.
- Deliverables are created for each phase and are considered frozen—that is, they should not be changed later in the life cycle of the product. If the deliverable of a phase changes, which often happens, the project will suffer schedule problems because the model did not accommodate, nor was the plan based on managing a change later in the cycle.
- All requirements must be known at the beginning of the life cycle, yet customers can rarely state all explicit requirements at that time. The model is not equipped to handle dynamic changes in requirements over the life cycle, as deliverables are "frozen." The model can be very costly to use if requirements are not well known or are dynamically changing during the course of the life cycle.
- Tight management and control is needed because there is no provision for revising the requirements.
- It is document-driven, and the amount of documentation can be excessive.
- The entire software product is being worked on at one time. There is no way to partition the system for delivery of pieces of the system. Budget problems can occur because of commitments to develop an entire system at one time. Large sums of money are allocated, with little flexibility to reallocate the funds without destroying the project in the process.
- There is no way to account for behind-the-scenes rework and iterations.

When to Use the Waterfall Model

Because of its weaknesses, application of the waterfall model should be limited to situations in which the requirements and the implementation of those requirements are very well understood.

The waterfall model performs well for product cycles with a stable product definition and well-understood technical methodologies.

If a company has experience in building a certain genre of system—accounting, payroll, controllers, compilers, manufacturing—then a project to build another of the same type of product, perhaps even based on existing designs, could make efficient use of the waterfall model. Another example of appropriate use is the creation and release of a new version of an existing product, if the changes are well defined and controlled. Porting an existing product to a new platform is often cited as an ideal project for use of the waterfall.

In all fairness, critics of this model must admit that the modified version of the waterfall is far less rigid than the original, including iterations of phases, concurrent phases, and change management. Reverse arrows allow for iterations of activities within phases. To reflect concurrence among phases, the rectangles are often stacked or the activities within the phases are listed beneath the rectangles showing the concurrence. Although the modified waterfall is much more flexible than the classic, it is still not the best choice for rapid development projects.

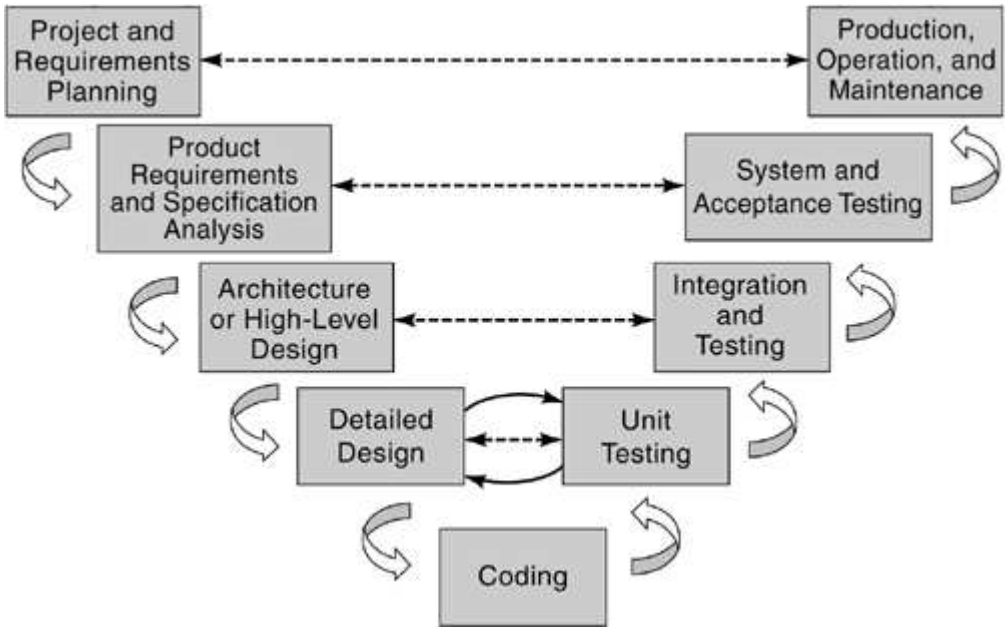
Waterfall models have historically been used on large projects with multiple teams and team members.

The V-Shaped Software Development Life Cycle Model

The V-shaped model was developed to assist the project team in planning and designing for the testability of a system. The model places a strong emphasis on the verification and validation activities of the product. It illustrates that the testing of the product is discussed, designed, and planned in the early phases of the development life cycle. The customer acceptance test plan is developed during the planning phase, the system integration test plan is developed during the analysis and design phases, and so on. This test plan development activity is represented by the dotted lines between the rectangles of the V.

The V-shaped model, shown in Figure 4-10, was designed as a variation of the waterfall model; therefore, it has inherited the same sequence structure. Each subsequent phase is begun at the completion of the deliverables of the current phase. It is representative of a comprehensive approach to defining the phases of the software development process. It emphasizes the relationship between the analytical and design phases that precede coding with the testing phases that follow coding. The dotted lines indicate that these phases should be considered in parallel.

Figure 4-10. The V-Shaped Software Development Life Cycle Model



Phases in the V-Shaped Model

The following list contains a brief description of each phase of the V-shaped model, from project and requirements planning through acceptance testing:

- **Project and requirements planning**— Determines the system requirements and how the resources of the organization will be allocated to meet them. (Where appropriate, this phase allocates functions to hardware and software.)

- **Product requirements and specification analysis**— Includes analysis of the software problem at hand and concludes with a complete specification of the expected external behavior of the software system to be built.

- **Architecture or high-level design**— Defines how the software functions are to implement the design.

- **Detailed design**— Defines and documents algorithms for each component that was defined during the architecture phase. These algorithms will be translated into code.

- **Coding**— Transforms the algorithms defined during the detailed design phase into software.

- **Unit testing**— Checks each coded module for errors.

- **Integration and testing**— Interconnects the sets of previously unit-tested modules to ensure that the sets behave as well as the independently tested modules did during the unit-testing phase.

- **System and acceptance testing**— Checks whether the entire software system (fully integrated) embedded in its actual hardware environment behaves according to the software requirements specification.

- **Production, operation, and maintenance**— Puts software into production and provides for enhancements and corrections.

- **Acceptance testing** (not shown)— Allows the user to test the functionality of the system against the original requirements. After final testing, the software and its surrounding hardware become operational. Maintenance of the system follows.

Strengths of the V-Shaped Model

When applied to a project for which it is well suited, the V-shaped model offers several strengths:

- The model emphasizes planning for verification and validation of the product in the early stages of product development. Emphasis is placed on testing by matching the test phase or process with the development process. The unit testing phase validates detailed design. The integration and testing phases validate architectural or high-level design. The system testing phase validates the product requirements and specification phase.

- The model encourages verification and validation of all internal and external deliverables, not just the software product.

- The V-shaped model encourages definition of the requirements before designing the system, and it encourages designing the software before building the components.

- It defines the products that the development process should generate; each deliverable must be testable.

- It enables project management to track progress accurately; the progress of the project follows a timeline, and the completion of each phase is a milestone.
- It is easy to use (when applied to a project for which it is suited).

Weaknesses of the V-Shaped Model

When applied to a project for which it is *not* well suited, the weaknesses of the V-shaped model are evident:

- It does not easily handle concurrent events.
- It does not handle iterations of phases.
- The model is not equipped to handle dynamic changes in requirements throughout the life cycle.
- The requirements are tested too late in the cycle to make changes without affecting the schedule for the project.
- The model does not contain risk analysis activities.

It is often graphically shown (as in Figure 4-10) without the integral tasks. This is an easily remedied issue, mentioned here only to remind the reader that integral tasks are present with the use of all life cycle models.

The V-shaped model may be modified to overcome these weaknesses by including iteration loops to handle the changing of requirements beyond the analysis phase.

When to Use the V-Shaped Model

Like its predecessor, the waterfall model, the V-shaped model works best when all knowledge of requirements is available up-front. A common modification to the V-shaped model, to overcome weaknesses, includes the addition of iteration loops to handle the changing of requirements beyond the analysis phase.

It works well when knowledge of how to implement the solution is available, technology is available, and staff have proficiency and experience with the technology.

The V-shaped model is an excellent choice for systems that require high reliability, such as hospital patient control applications and embedded software for air-bag chip controllers in automobiles.

The Prototype Software Development Life Cycle Model

Fred Brooks's classic, *The Mythical Man-Month*, is as fresh today as it was in 1975. Technology has changed the world in drastic ways, but many foibles of software project management remain the same. Decades ago, Brooks said:

> In most projects, the first system built is barely usable. It may be too slow, too big, awkward to use, or all three. There is no alternative but to start again, smarting but smarter, and build a redesigned version in which these problems are solved….

> When a new system concept or new technology is used, one has to build a system to throw away, for even the best planning is not so omniscient as to get it right the first time.
>
> The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. The only question is whether to plan in advance to build a throwaway, or to promise to deliver the throwaway to customers….[9]

It is this concept of building a pilot, or prototype system that led to the "structured," "evolutionary" rapid prototyping model, the RAD model, and the spiral model. In his later, equally wise work, "No Silver Bullet, the Essence and Accidents of Programming," Brooks believes that most software development errors still have to do with getting the system concept wrong, not the syntax or the logic. Software development will always be difficult, and there will never be a magic panacea or silver bullet. He offers a positive note in the application of rapid prototyping techniques:

> The hardest single part of building a software system is deciding precisely what to build. No other part of the conceptual work is as difficult as establishing the detailed technical requirements, including all the interfaces to people, to machines, and to other software systems. No other part of the work so cripples the resulting system if done wrong. No other part is more difficult to rectify later.
>
> Therefore, the most important function that the software builder performs for the client is the iterative extraction and refinement of the product requirements. For the truth is, the client does not know what he wants.
>
> One of the most promising of the current technological efforts, and one which attacks the essence, not the accidents, of the software problem, is the development of approaches and tools for rapid prototyping of systems as part of the iterative specification of requirements.[10]

Watts Humphrey, best known as the inspiration for the SEI CMM, supports Brooks in the importance of requirements and evolution:

> There is a basic principle of most systems that involve more than minor evolutionary change: The system will change the operational environment. Since the users can only think in terms of the environment they know, the requirements for such systems are always stated in the current environment's terms. These requirements are thus necessarily incomplete, inaccurate, and misleading. The challenge for the system developer is to devise a development process that will discover, define, and develop to real requirements. This can only be done with intimate user involvement, and often with periodic prototype or early version field tests. Such processes always appear to take longer but invariably end up with a better system much sooner than with any other strategy.[11]

Definitions of Prototyping

According to Connell and Shafer, an evolutionary rapid prototype is:

> An easily modifiable and extensible working model of a proposed system, not necessarily representative of a complete system, which provides users of the application with a physical representation of key parts of the system before implementation. An easily built, readily modifiable, ultimately extensible, partially specified, working model of the primary aspects of a proposed system.[12]

And Bernard Boar defined a prototype as "a strategy for performing requirements determination wherein user needs are extracted, presented, and developed by building a working model of the ultimate system—quickly and in context."[13]

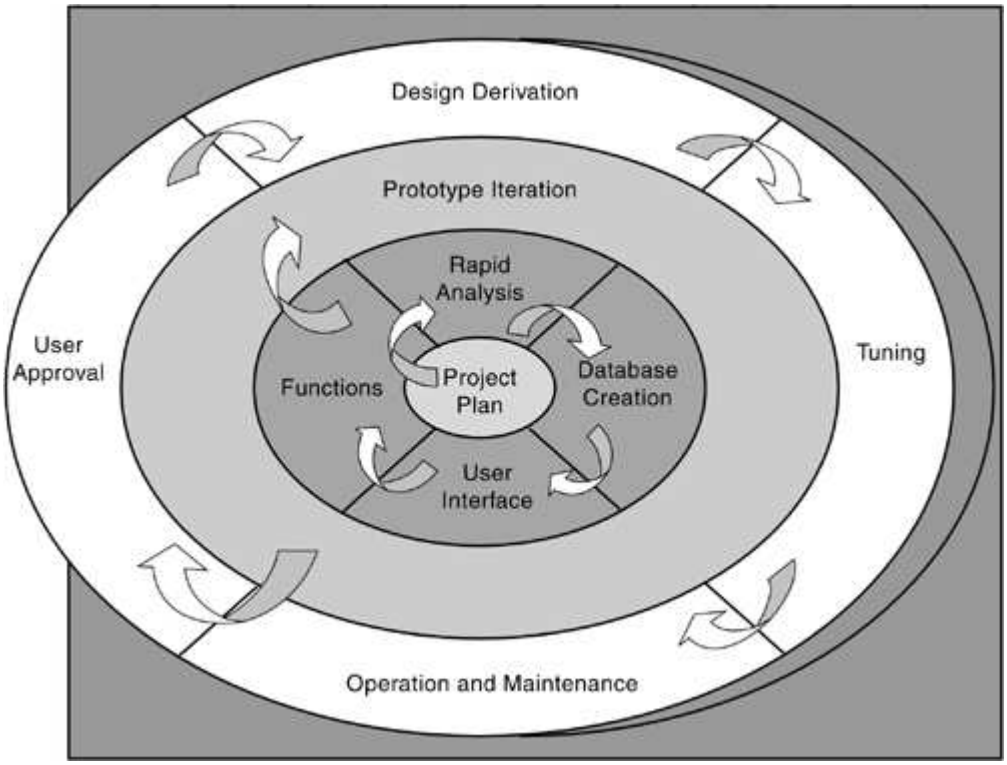
Description of the Structured Evolutionary Prototyping Model

Prototyping is the process of building a working replica of a system. The prototype is the equivalent of a mock-up, or "breadboard," in the hardware world.

Evolutionary programs are conducted within the context of a plan for progression toward an ultimate capability. This strategy also requires the development of increments of software demonstrable to the user, who is involved throughout the entire development process.

A "quick" partial implementation of the system is created before or during the requirements definition phase. The end users of the system use the rapid prototype and then supply feedback to the project team for further refinement of the requirements of the system. This refinement process continues until the user is satisfied. When the requirements definition process has been completed, through the development of rapid prototypes, the detailed design is derived and the rapid prototype is fine-tuned using code or external utilities to create a final working product. Ideally, the prototyping model is evolvable—nothing wasted—and of high quality—no skimping on documentation, analysis, design, testing, and so on. Hence, the name is "structured rapid prototyping model," as shown in Figure 4-11.

Figure 4-11. The Structured Evolutionary Rapid Prototyping Model



The life cycle begins in the center of the ellipse. User and designer develop a preliminary project plan from preliminary requirements. Using rapid analysis techniques, user and designer work together to define the requirements and specifications for the critical parts of the envisioned system. Project planning is the first activity of the rapid analysis phase, producing a document outlining rough schedules and deliverables.

The project plan is created, and then a rapid analysis is performed, followed by database, user interface, and function creation. The second activity is rapid analysis, during which preliminary user interviews are used to develop an intentionally incomplete high-level paper model of the system. A document containing the partial requirements specification is output from this task and used to build the initial prototype that is created in the next three phases. The designer constructs a model (using tools), a partial representation of the system that includes only those basic attributes necessary for meeting the customer's requirements. Next the rapid prototype iteration loop begins. The designer demonstrates the prototype; the user evaluates its performance. Problems are identified then, and the user and designer work together to eliminate them. The process continues until the user is satisfied that the system represents the requirements. The project team will remain in this loop until the user has agreed that the rapid prototype accurately reflects the system requirements. Creation of the database is the first of these phases. After the initial database is set up, menu development may begin, followed by function development—a working model. The model is then demonstrated to the user for suggestions for improvements, which are incorporated into successive iterations until the working model proves satisfactory. Then formal user approval of the prototype's functionality is obtained. Next, a preliminary system design document is produced. The prototype iteration phase is the heart—through scenarios provided by the working model, the user may role-play and request successive refinements to the model until all functional requirements are met. When the user approval has been obtained, the detail design is derived from the rapid prototype and the system is tuned for production use. It is within this tuning phase that the rapid prototype becomes a fully operational system instead of a partial system during the prototyping iteration loop.

A detailed design may be derived from the prototypes. The prototype then is tuned using code or external utilities, as required. The designer uses validated requirements as a basis for designing the production software.

Additional work may be needed to construct the production version: more functionality, different system resources to meet full workload, or timing constraints. Stress testing, benchmarking, and tuning follow, and then comes operation maintenance as usual.

The final phase is operation and maintenance, which reflects the activities to move the system into a production state.

There is no "right" way to use the prototype approach. The result may be thrown away, used as a foundation for enhancement, or repackaged as a product, depending on the original objectives, the process used, and the desired quality. Clearly, cost and schedules benefit from *evolutionary* prototyping because no components are "thrown away."

Strengths of the Structured Evolutionary Rapid Prototyping Model

When ap-plied to a suitable project, the strengths of the structured evolutionary rapid prototyping model can be seen in the following ways:

- The end user can "see" the system requirements as they are being gathered by the project team—customers get early interaction with system.
- Developers learn from customers' reactions to demonstrations of one or more facets of system behavior, thereby reducing requirements uncertainties.
- There is less room for confusion, miscommunication, or misunderstanding in the definition of the system requirements, leading to a more accurate end product—customer and developer have a baseline to work against.

- New or unexpected user requirements can be accommodated, which is necessary because reality can be different from conceptualization.
- It provides a formal specification embodied in an operating replica.
- The model allows for flexible design and development, including multiple iterations through life cycle phases.
- Steady, visible signs of progress are produced, making customers secure.
- Communications issues between customers and developers are minimized.
- Quality is built in with early user involvement.
- The opportunity to view a function in operation stimulates a perceived need for additional functionality.
- Development costs are saved through less rework.
- Costs are limited by understanding the problem before committing more resources.
- Risk control is provided.
- Documentation focuses on the end product, not the evolution of the product.
- Users tend to be more satisfied when involved throughout the life cycle.

Weaknesses of the Structured Evolutionary Rapid Prototyping Model

When applied to a project for which it is *not* suited, the weaknesses of this model can be seen in the following ways:

- The model may not be accepted due to a reputation among conservatives as a "quick-and-dirty" method.
- Quick-and-dirty prototypes, in contrast to evolutionary rapid prototypes, suffer from inadequate or missing documentation.
- If the prototype objectives are not agreed upon in advance, the process can turn into an exercise in hacking code.
- In the rush to create a working prototype, overall software quality or long-term maintainability may be overlooked.
- Sometimes a system with poor performance is produced, especially if the tuning stage is skipped.
- There may be a tendency for difficult problems to be pushed to the future, causing the initial promise of the prototype to not be met by subsequent products.

- If the users cannot be involved during the rapid prototype iteration phase of the life cycle, the final product may suffer adverse effects, including quality issues.
- The rapid prototype is a partial system during the prototype iteration phase. If the project is cancelled, the end user will be left with only a partial system.
- The customer may expect the exact "look and feel" of the prototype. In fact, it may have to be ported to a different platform, with different tools to accommodate size or performance issues, resulting in a different user interface.
- The customer may want to have the prototype delivered rather than waiting for full, well-engineered version.
- If the prototyping language or environment is not consistent with the production language or environment, there can be delays in full implementation of the production system.
- Prototyping is habit-forming and may go on too long. Undisciplined developers may fall into a code-and-fix cycle, leading to expensive, unplanned prototype iterations.
- Developers and users don't always understand that when a prototype is evolved into a final product, traditional documentation is still necessary. If it is not present, a later retrofit can be more expensive than throwing away the prototype.
- When customers, satisfied with a prototype, demand immediate delivery, it is tempting for the software development project manager to relent.
- Customers may have a difficult time knowing the difference between a prototype and a fully developed system that is ready for implementation.
- Customers may become frustrated without the knowledge of the exact number of iterations that will be necessary.
- A system may become overevolved; the iterative process of prototype demonstration and revision can continue forever without proper management. As users see success in requirements being met, they may have a tendency to add to the list of items to be prototyped until the scope of the project far exceeds the feasibility study.
- Developers may make less-than-ideal choices in prototyping tools (operating systems, languages, and inefficient algorithms) just to demonstrate capability.
- Structured techniques are abandoned in the name of analysis paralysis avoidance. The same "real" requirements analysis, design, and attention to quality for maintainable code is necessary with prototyping, as with any other life cycle model (although they may be produced in smaller increments).

When to Use the Structured Evolutionary Rapid Prototyping Model

A pro-ject manager may feel confident that the structured evolutionary rapid prototyping model is appropriate when several of the following conditions are met:

- When requirements are not known up-front;

- When requirements are unstable or may be misunderstood or poorly communicated;
- For requirements clarification;
- When developing user interfaces;
- For proof-of-concept;
- For short-lived demonstrations;
- When structured, evolutionary rapid prototyping may be used successfully on large systems where some modules are prototyped and some are developed in a more traditional fashion;
- On new, original development (as opposed to maintenance on an existing system);
- When there is a need to reduce requirements uncertainty—reduces risk of producing a system that has no value to the customer;
- When requirements are changing rapidly, when the customer is reluctant to commit to a set of requirements, or when application not well understood;
- When developers are unsure of the optimal architecture or algorithms to use;
- When algorithms or system interfaces are complex;
- To demonstrate technical feasibility when the technical risk is high;
- On high-technology software-intensive systems where requirements beyond the core capability can be generally but not specifically identified;
- During software acquisition, especially on medium- to high-risk programs;
- In combination with the waterfall model—the front end of the project uses prototyping, and the back end uses waterfall phases for system operational efficiency and quality;
- Prototyping should always be used with the analysis and design portions of object-oriented development.

Rapid prototyping is well suited for user-interface intensive systems, such as display panels for control devices, interactive online systems, first-of-a-kind products, and decision support systems such as command and control or medical diagnosis, among others.

The Rapid Application Development (RAD) Software Development Life Cycle Model

In the 1980s, IBM responded to the constricting nature of formal methods, such as the waterfall model, with the use of a rapid application development (RAD) approach. James Martin's book *Rapid Application Development* introduced this approach to the software community. With RAD, the user is involved in *all* phases of the life cycle—not only requirements definition, but design,

development, test, and final delivery as well. User involvement is increased from the norm by the use of a development tool or environment that allows product evaluation in all stages of its development. The availability of graphical user interface development tools and code generators made it possible. Tools such as Oracle Designer/2000, Java Jbuilder 3, Linux, Visual C++, Visual Basic 6, SAS, and other applications have entire books dedicated to using them as rapid application tools.

RAD is characterized by the quick turnaround time from requirements definition to completed system. It follows a sequence of evolutionary system integrations or prototypes that are reviewed with the customer, discovering requirements along the way. The development of each integrated delivery is restricted to a well-defined period of time, usually about 60 days, called a time-box.

Factors that allow a system to be created in the 60 days of the time-box, without sacrificing quality, include the use of high-powered development tools, a high reuse factor, and knowledgeable and dedicated resources.

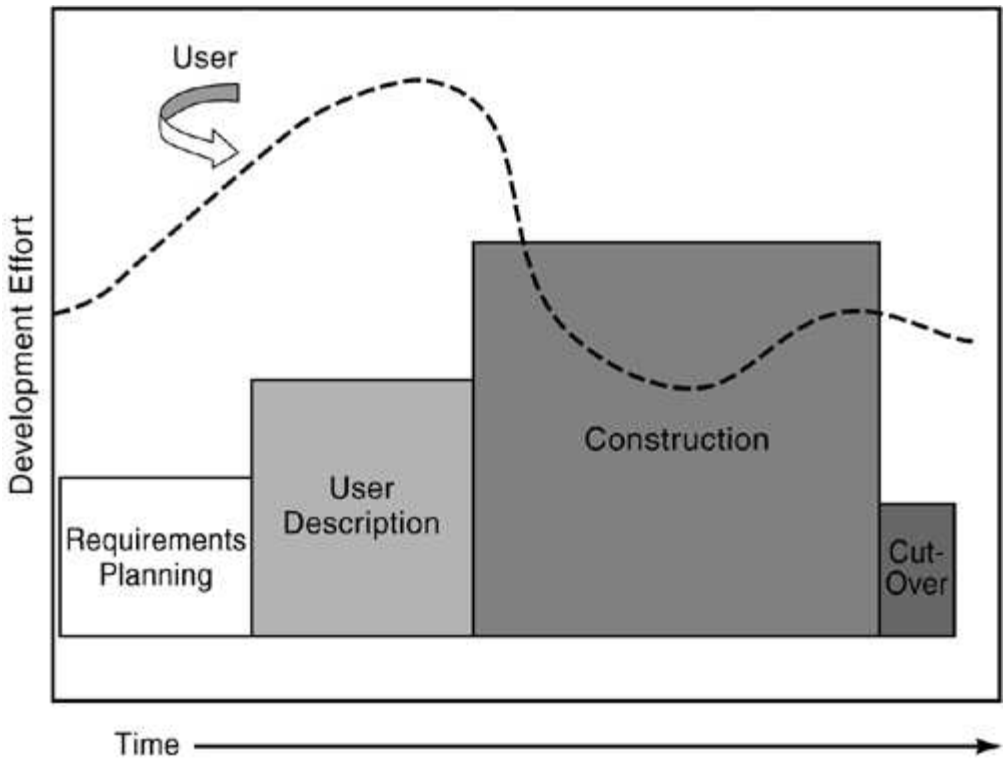
The critical end-user roles shift work from programming and testing to planning and design. More work is created for users at the front of the life cycle, but they are rewarded with a system that is built more quickly.

A Brief Description of the Phases in the RAD Model

The RAD model, shown in Figure 4-12, represents the phases of its life cycle development process and the user involvement throughout the phases (the curved line).

- **Requirements planning phase**— Requirements are gathered using a workshop technique called joint requirements planning (JRP), a structured discussion of the business problems at hand.
- **User description**— Joint application design (JAD) is used to harness user involvement; the project team often uses automated tools to capture information from the users during this nontechnical design of the system.
- **Construction phase ("do until done")**— This phase combines detailed design, the build (coding and testing), and the release to the customer inside a time-box. It is heavily dependent on the use of code generators, screen generators, and other types of productivity tools.
- **Cutover**— This phase includes acceptance testing by the users, installation of the system, and user training.

Figure 4-12. The Rapid Application Development Model



Strengths of the RAD Model

When applied to a project for which it is well suited, strengths of the RAD model include the following:

- Cycle time for the full product can be reduced due to the use of powerful development tools.
- Fewer developers are required because the system is developed by a project team familiar with the problem domain.
- Quick initial views of the product are possible.
- Reduced cycle time and improved productivity with fewer people spell lower costs.
- The time-box approach mitigates cost and schedule risk.
- It makes effective use of off-the-shelf tools and frameworks.
- Ongoing customer involvement minimizes the risk of not achieving customer satisfaction and ensures that the system meets the business needs and that the operational utility of the product is sound.
- Each time-box includes analysis, design, and implementation (phases are separated from activities).
- Constant integrations isolate problems and encourage customer feedback.

- The focus moves from documentation to code—what you see is what you get (WYSIWYG).
- It uses modeling approaches and tools—business modeling (how information flows, where it is generated, by whom, where it goes, how it is processed); data modeling (data objects and attributes and relationships identified); process modeling (data objects transformed); application generation (fourth-generation techniques).
- It reuses existing program components.

Weaknesses of the RAD Model

Weaknesses of this model when applied to a project for which it is *not* well suited include the following:

- If the users cannot be involved consistently throughout the life cycle, the final product will be adversely affected.
- This model requires highly skilled and well-trained developers in the use of the chosen development tools to achieve the rapid turnaround time.
- It requires a system that can be properly modularized.
- It can fail if reusable components are not available.
- It may be hard to use with legacy systems and many interfaces.
- It requires developers and customers who are committed to rapid-fire activities in an abbreviated time frame.
- Blindly applied, no bounds are placed on the cost or completion date of the project.
- Teams developing commercial projects with RAD can overevolve the product and never ship it.
- There is a risk of never achieving closure—the project manager must work closely with both the development team and the customer to avoid an infinite loop.
- An efficient, accelerated development process must be in place for quick response to user feedback.

When to Use the RAD Model

A project manager may feel confident that the RAD model is appropriate when several of the following conditions are met:

- On systems that may be modularized (component-based construction) and that are scalable;
- On systems with reasonably well-known requirements;
- When the end user can be involved throughout the life cycle;

- When users are willing to become heavily involved in the use of automated tools;
- On projects requiring short development times, usually about 60 days;
- On systems that can be time-boxed to deliver functionality in increments;
- When reusable parts are available through automated software repositories;
- On systems that are proof-of-concept, noncritical, or small;
- When cost and schedule are not a critical concern (such as internal tool development);
- On systems that do not require high performance, especially through tuning interfaces;
- When technical risks are low;
- On information systems;
- When the project team is familiar with the problem domain, skilled in the use of the development tools, and highly motivated.

The Incremental Software Development Life Cycle Model

Incremental development is the process of constructing a partial implementation of a total system and slowly adding increased functionality or performance. This approach reduces the cost incurred before an initial capability is achieved. It also produces an operational system more quickly by emphasizing a building-block approach that helps control the impact of changing requirements.

The incremental model performs the waterfall in overlapping sections, producing usable functionality earlier. This may involve a complete up-front set of requirements that are implemented in a series of small projects, or a project may start with general objectives that are refined and implemented in groups.

Boehm describes the incremental approach as combining elements of the linear sequential model and prototyping, and advocates developing the software in increments of functional capability. He reports that his experience is that this refinement of the waterfall model works equally well on extremely large as well as on small projects. In a short example of a product delivered in three increments, increment 1 would provide basic algorithms and basic output of results, increment 2 would add some valuable production-mode capabilities such as the ability to file and retrieve previous runs, and increment 3 would add various nice-to-have features for the user interface and added computational features.

The incremental model describes the process of prioritizing requirements of the system and then implementing them in groups. Generally, increments become smaller, implementing fewer requirements each time. Each subsequent release of the system adds function to the previous release, until all designed functionality has been implemented. This approach reduces costs, controls the impact of changing requirements, and produces an operational system more quickly by developing the system in a building-block fashion.

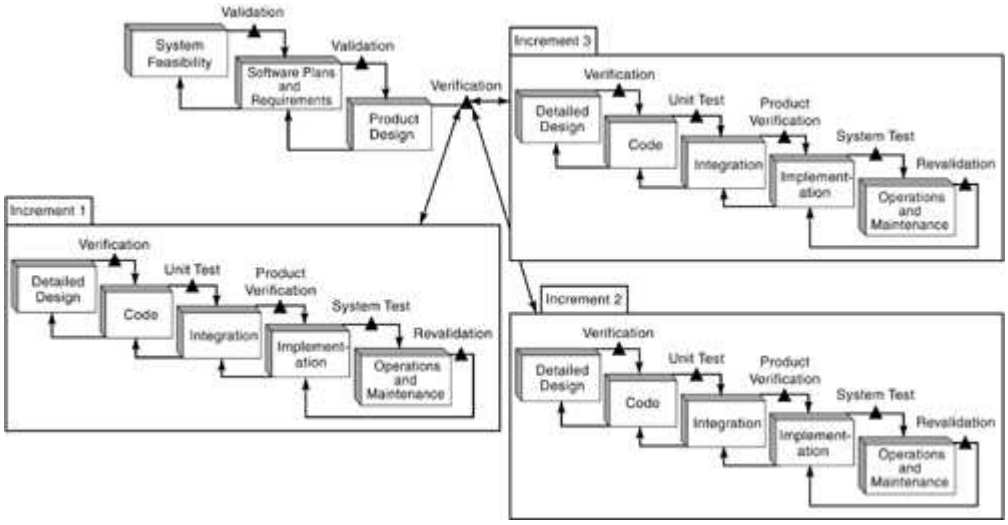
The early phases of the life cycle (planning, analysis, and design) consider the entire system to be developed. During these phases, the increments and the functions within them are defined for

development. *Each increment* then proceeds through the remaining phases of the life cycle: code, test, and delivery.

A set of functions that are the core, or highest priority requirements critical to the success of the project, or that will reduce risk is constructed, tested, and implemented first. Subsequent iterations expand on the core, gradually adding increased functionality or performance. Functions are added in significant increments to meet user needs in a cohesive fashion. Each additional function is validated against the entire set of requirements.

The linear sequences may be staggered over calendar time, with each one producing a deliverable increment of software, as shown in Figure 4-13.

Figure 4-13. The Incremental Model



This type of development can be combined with other models. It is often integrated with the spiral model, the V-shaped model, and the waterfall model to reduce costs and risks in the development of a system.

Strengths of the Incremental Model

When applied to a project for which it is well suited, strengths of the incremental model include the following:

- Funds for a total product development need not be expended up-front because a major function or high-risk function is developed and delivered first.
- An operational product is delivered with each increment.
- Lessons learned at the end of each incremental delivery can result in positive revisions for the next; the customer has an opportunity to respond to each build.
- The use of the successive increments provides a way to incorporate user experience into a refined product in a much less expensive way than total redevelopment.
- The "divide and conquer" rule allows the problem to be broken down into manageable pieces, preventing the development team from becoming overwhelmed with lengthy requirements.

- Limited staff can be used, with the same team working sequentially to deliver each increment, keeping all teams working (labor distribution curve may be leveled out through the time-phasing of project effort).
- Starting the next build during the transition phase of the last smoothes staffing exchanges.
- Project momentum can be maintained.
- Costs and schedule risks may be revisited at the end of each incremental delivery.
- Initial delivery cost is lowered.
- Initial delivery schedule is faster, perhaps allowing response to a market window.
- Risk of failure and changing requirements is reduced.
- User needs are more controllable because the development time for each increment is so small.
- Because the leap from present to future does not occur in one move, the customer can adjust to new technology in incremental steps.
- Customers can see the most important, most useful functionality early.
- Tangible signs of progress keeps schedule pressure to a manageable level.
- Risk is spread across several smaller increments instead of concentrating in one large development.
- Requirements are stabilized (through user buy-in) during the production of a given increment by deferring nonessential changes until later increments.
- Understanding of the requirements for later increments becomes clearer based on the user's ability to gain a working knowledge of earlier increments.
- The increments of functional capability are much more helpful and easy to test than the intermediate level products in level-by-level top-down development.

Weaknesses of the Incremental Model

When applied to a project for which it is *not* well suited, weaknesses of this model include the following:

- The model does not allow for iterations within each increment.
- The definition of a complete, fully functional system must be done early in the life cycle to allow for the definition of the increments.
- Because some modules will be completed long before others, well-defined interfaces are

required.

- Formal reviews and audits are more difficult to implement on increments than on a complete system.
- There can be a tendency to push difficult problems to the future to demonstrate early success to management.
- The customer must realize that the total cost will not be lower.
- Use of general objectives, rather than complete requirements, in the analysis phase can be uncomfortable for management.
- It requires good planning and design: Management must take care to distribute the work; the technical staff must watch dependencies.

When to Use the Incremental Model

A project manager may feel confident that the incremental model is appropriate when several of the following conditions are met:

- When most of the requirements are understood up-front but are expected to evolve over time;
- When there is a short market window and a need to get basic functionality to the market quickly;
- On projects that have lengthy development schedules, usually over one year;
- With an even distribution of different priority features;
- On low- to medium-risk programs;
- On a project with new technology, allowing the user to adjust to the system in smaller incremental steps rather than leaping to a major new product;
- When considerations of risk, funding, schedule, size of program, complexity of program, or need for early realization of benefits indicate that a phased approach is the most prudent;
- When it is too risky to develop the whole system at once;
- When deliveries occur at regular intervals.

The Spiral Software Development Life Cycle Model

The spiral model, introduced by Dr. Barry Boehm and published in *IEEE Computer*, 1988, addresses these concerns about the waterfall model: It does not adequately address changes, it assumes a relatively uniform and orderly sequence of development steps, and it does not provide for such methods as rapid prototyping or advanced languages.

The spiral model encompasses the strengths of the waterfall model while including risk analysis, risk management, and support and management processes. It also allows for the development of the product to be performed using a prototyping technique or rapid application development through the use of fourth-generation (and beyond) languages and development tools.

It reflects the underlying concept that each cycle involves a progression that addresses the same sequence of steps as the waterfall process model, for each portion of the product and for each of its levels of complexity, from an overall statement of need to the coding of each individual program.

As shown in Figure 4-14, each quadrant of the model has a purpose and supporting activities. The quadrants are listed here:

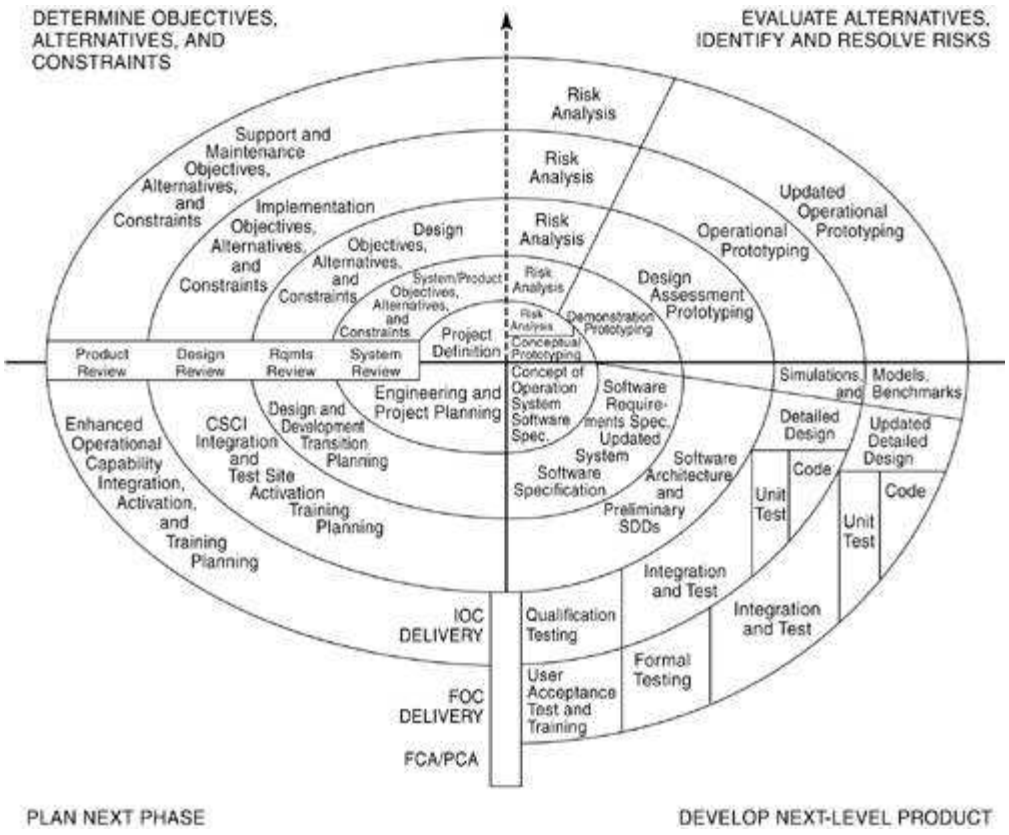
- **Determine objectives, alternatives, and constraints**— Objectives such as performance, functionality, ability to accommodate change, hardware/software interface, and critical success factors are identified. Alternative means of implementing this portion of the product (build, reuse, buy, subcontract, etc.) are determined; constraints imposed on the application of the alternatives (cost, schedule, interface, environmental limitations, etc.) are determined. Risks associated with lack of experience, new technology, tight schedules, poor processes, and so on are documented.

- **Evaluate alternatives, and identify and resolve risks**— Alternatives relative to the objectives and constraints are evaluated; the identification and resolution of risks (risk management, cost-effective strategy for resolving sources, evaluation of remaining risks where money could be lost by continuing system development [go/no-go decisions], etc.) occurs.

- **Develop next-level product**— Typical activities in this quadrant could be creation of a design, review of a design, development of code, inspection of code, testing, and packaging of the product. The first build is the customer's first look at the system. After this, a planning phase begins—the program is reset to respond to customer's reaction. With each subsequent build, a better idea of customer requirements is developed. The degree of change from one build to the next diminishes with each build, eventually resulting in an operational system.

- **Plan next phase**— Typical activities in this quadrant could be development of the project plan, development of the configuration management plan, development of the test plan, and development of the installation plan.

Figure 4-14. The Spiral Model



To read the spiral model shown in Figure 4-14, start in the center in Quadrant 1 (determine objectives, alternatives, and constraints), explore risks, make a plan to handle risks, commit to the approach for the next iteration, and move to the right.

For each iteration, determine objectives, alternatives, and constraints; identify and resolve risks; evaluate alternatives; develop the deliverables for that iteration and verify that they are correct; plan the next iteration; and commit to an approach for the next iteration, if you decide to have one.

There is no set number of loops through the four quadrants—as many should be taken as are appropriate, and iterations may be tailored for a specific project.

A salient factor is that coding is de-emphasized for a much longer period than with other models. The idea is to minimize risk through successive refinements of user requirements. Each "mini-project" (travel around the spiral) addresses one or more major risks, beginning with the highest. Typical risks include poorly understood requirements, poorly understood architecture, potential performance problems, problems in the underlying technology, and so on. Chapter 18, "Determining Project Risks," will describe the most common software project risks and techniques for their mitigation. The first build, the initial operational capability (IOC), is the customer's first chance to "test-drive" the system, after which another set of planning activities takes place to kick off the next iteration. It is also important to note that the model does not dispense with traditional structured methods—they appear at the end (outside loop) of the spiral. Design through user acceptance testing appears before final operational capability (FOC), just as they do in the waterfall model.

When using a prototyping approach, there may be a tendency for developers to dismiss good system development practices and misuse the model as an excuse for "quick-and-dirty" development. Proper use of the spiral model or one of its simpler variants will help prevent "hacking" and instill discipline. As seen in Figure 4-14, after much analysis and risk assessment,

the "tail" of the spiral shows a set of waterfall-like disciplined process phases.

Addressed more thoroughly than with other strategies, the spiral method emphasizes the evaluation of alternatives and risk assessment. A review at the end of each phase ensures commitment to the next phase, or, if necessary, identifies the need to rework a phase. The advantages of the spiral model are its emphasis on procedures, such as risk analysis, and its adaptability to different life cycle approaches. If the spiral method is employed with demonstrations, baselining, and configuration management, you can get continuous user buy-in and a disciplined process.[14]

Strengths of the Spiral Model

When applied to a project for which it is well suited, strengths of the spiral model include the following:

- The spiral model allows users to see the system early, through the use of rapid prototyping in the development life cycle.
- It provides early indications of insurmountable risks, without much cost.
- It allows users to be closely tied to all planning, risk analysis, development, and evaluation activities.
- It splits a potentially large development effort into small chunks in which critical, high-risk functions are implemented first, allowing the continuation of the project to be optional. In this way, expenses are lessened if the project must be abandoned.
- The model allows for flexible design by embracing the strengths of the waterfall model while allowing for iteration throughout the phases of that model.
- It takes advantage of the strengths of the incremental model, with incremental releases, schedule reduction through phased overlap of releases, and resources held constant as the system gradually grows.
- It does not rely on the impossible task of getting the design perfect.
- It provides early and frequent feedback from users to developers, ensuring a correct product with high quality.
- Management control of quality, correctness, cost, schedule, and staffing is improved though reviews at the conclusion of each iteration.
- It provides productivity improvement through reuse capabilities.
- It enhances predictability through clarification of objectives.
- All the money needed for the project need not be allocated up-front when the spiral model is adopted.
- Cumulative costs may be assessed frequently, and a decrease in risk is associated with the cost.

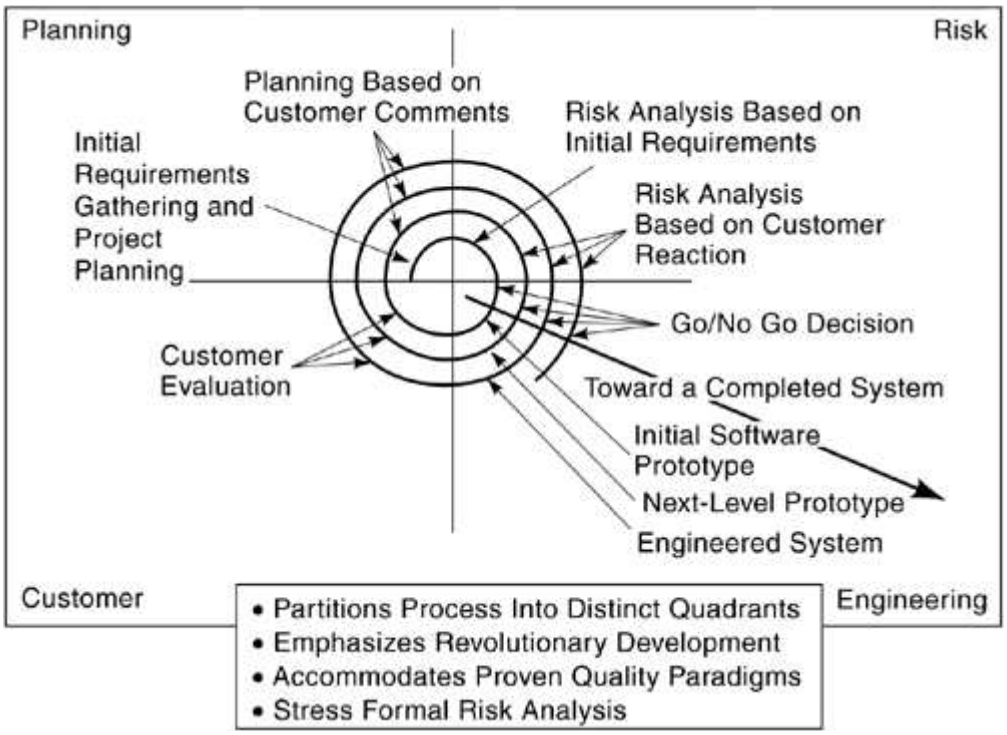
Weaknesses of the Spiral Model

When applied to a project for which it is *not* well suited, weaknesses of the spiral model include the following:

- If the project is low-risk or small, this model can be an expensive one. The time spent evaluating the risk after each spiral is costly.
- The model is complex, and developers, managers, and customers may find it too complicated to use.
- Considerable risk assessment expertise is required.
- The spiral may continue indefinitely, generated by each of the customer's responses to the build initiating a new cycle; closure (convergence on a solution) may be difficult to achieve.
- The large number of intermediate stages can create additional internal and external documentation to process.
- Use of the model may be expensive and even unaffordable—time spent planning, resetting objectives, doing risk analysis, and prototyping may be excessive.
- Developers must be reassigned during nondevelopment-phase activities.
- It can be hard to define objective, verifiable milestones that indicate readiness to proceed through the next iteration.
- The lack of a good prototyping tool or technique can make this model clumsy to use.
- The industry has not had as much experience with the spiral model as it has with others.

Some users of this technique have found the original spiral model to be complex and have created a simplified version, shown in Figure 4-15.

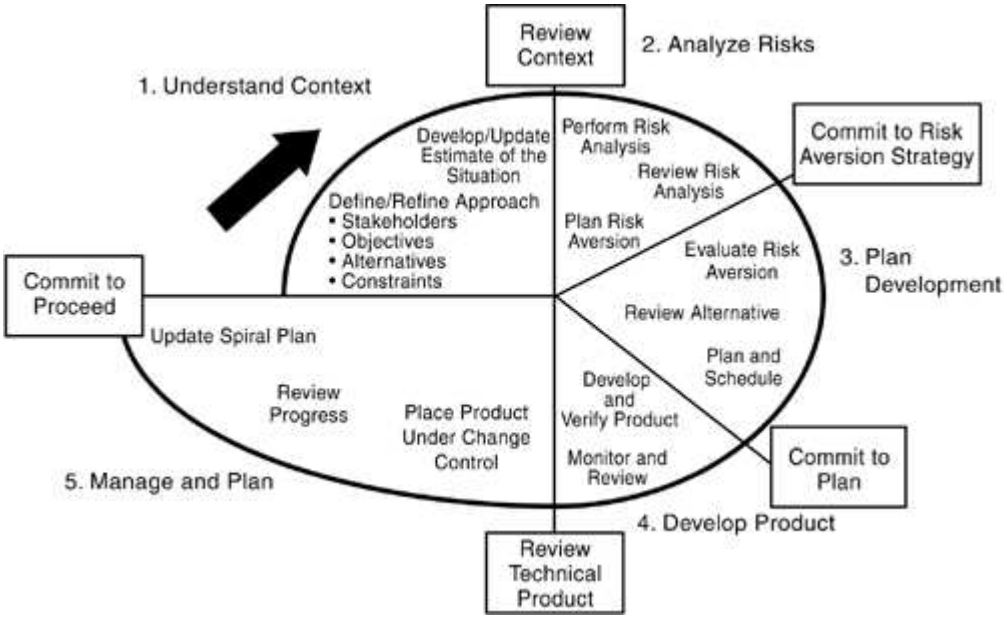
Figure 4-15. A Simplified View of the Spiral Model



Another version, a modified view of the spiral model from the Software Productivity Consortium, may be seen in Figure 4-16.

Figure 4-16. A Modified View of the Spiral Model

Source: Software Productivity Consortium.



When to Use the Spiral Model

A project manager may feel confident that the spiral model is appropriate when several of the following conditions are met:

- When the creation of a prototype is the appropriate type of product development;
- When it is important to communicate how costs will be increasing and to evaluate the project for costs during the risk quadrant activities;
- When organizations have the skills to tailor the model;
- For projects that represent a medium to high risk;
- When it is unwise to commit to a long-term project due to potential changes in economic priorities, and when these uncertainties may limit the available time frame;
- When the technology is new and tests of basic concepts are required;
- When users are unsure of their needs;
- When requirements are complex;
- For a new function or product line;
- When significant changes are expected, as with research or exploration;
- When it is important to focus on stable or known parts while gathering knowledge about changing parts;
- For large projects;
- For organizations that cannot afford to allocate all the necessary project money up-front, without getting some back along the way;
- On long projects that may make managers or customers nervous;
- When benefits are uncertain and success is not guaranteed;
- To demonstrate quality and attainment of objectives in short period of time;
- When new technologies are being employed, such as first-time object-oriented approaches;
- With computation-intensive systems, such as decision support systems;
- With business projects as well as aerospace, defense, and engineering projects, where the spiral model already enjoys popular use.

Tailored Software Development Life Cycle Models

Sometimes a project manager can pluck a life cycle model from a book and run with it. Other times, there seems to be nothing that quite fits the project needs, although one of the widely used and pretested models comes close. Need a life cycle that considers risk, but the spiral seems like overkill? Then start with the spiral and pare it down. Required to deliver functionality in increments

but must consider serious reliability issues? Then combine the incremental model with the V-shaped model. Several examples of tailored models follow.

Fast Track

A fast-track life cycle methodology speeds up, or bypasses, one or more of the life cycle phases or development processes. Many or most of the normal development steps are executed as usual, while the formality or scope of others may be reduced.

Tailoring of the life cycle is required for a fast-track approach best used on nonmajor software development and acquisition projects. Fast tracking may be needed to serve a time criticality such as being the first to market for a commercial product, or a national threat for a government agency. In addition to a shortened life cycle, one tailored for fast tracking is usually less formal. The overall life of the delivered product may be short, indicating a short maintenance phase.

Fast-track projects should be attempted only in organizations accustomed to discipline. An institutionalized, defined development environment minimizes risk when employing this type of extreme measure. With a clearly defined, stable set of requirements and a method in place to accommodate changes, fast-track projects have an increased chance of success.

Concurrent Engineering

Concurrent engineering (CE) is concerned with making better products in less time. A basic tenet of the approach is that all aspects of the product's life cycle should be considered as early as possible in the design-to-manufacturing process. Early consideration of later phases of the life cycle brings to light problems that will occur downstream and, therefore, supports intelligent and informed decision-making throughout the process.[15]

Although borrowed from other engineering disciplines, concurrent engineering works for software as well. Especially on large projects, status tracking by major phases in a life cycle may be an oversimplified model. A snapshot in time would show that there are usually several activities (requirements gathering, design, coding, testing, etc.) going on simultaneously. In addition, any internal or external project deliverable may be in one of several states (being developed, being reviewed, being revised, waiting for the next step, etc.). Concurrent engineering considers all aspects of the life cycle as early as possible. Concurrent process models allow an accurate view of what state the project is in—what activities are being conducted and how the deliverables are progressing.

When using this approach, it is wise to assess technical risks involved to determine whether the technology being attempted is compatible with an accelerated strategy, leave some room in the schedule, assess the technological process periodically to see if it is still compatible with the plan, and, as with more traditional life cycles, ensure that there is a provision for testing and evaluation because there is extreme risk in skipping these activities.

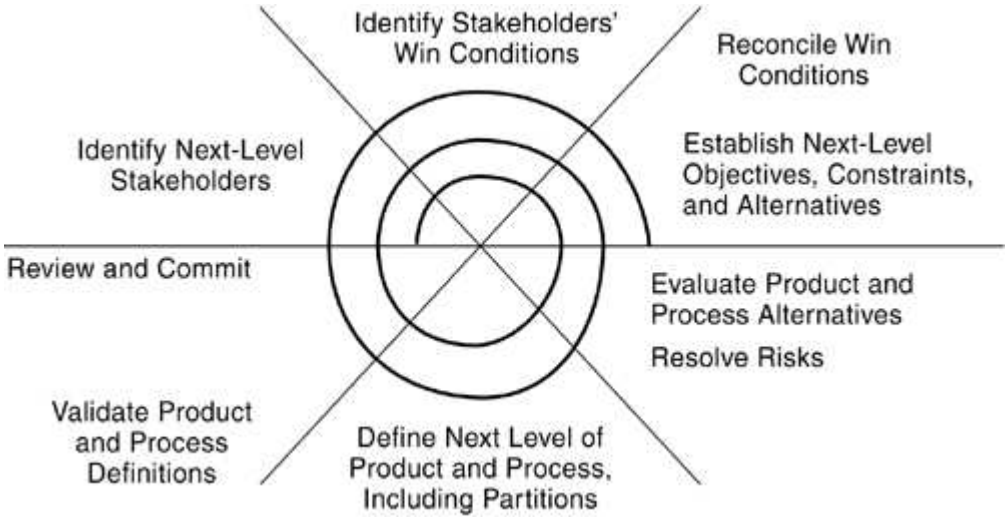
The spiral model, being risk-driven, is a good model to use in guiding multistakeholder concurrent engineering of software-intensive systems. It offers a cyclic approach for incrementally growing a system's definition and implementation, as well as providing anchor-point milestones for ensuring continued stakeholder commitment.

Win-Win Spiral Model

Boehm also offers a modified spiral model called the "win-win spiral model," shown in Figure 4-17. It contains more customer-focused phases by adding Theory W activities to the front of each cycle. *Theory W* is a management approach elevating the importance of the system's key stakeholders

(user, customer, developer, maintainer, interfacer, etc.), who must all be "winners" if the project is declared a success. In this negotiation-based approach, the cycles contain these phases or steps: Identify next-level stakeholders; identify stakeholders' win conditions; reconcile win conditions; establish next-level objectives, constraints, and alternatives; evaluate process and product alternatives and resolve risks; define the next level of the product and process, including partitions; validate the product and process definitions; and review and comment.

Figure 4-17. The Win-Win Spiral Model



Not shown in Figure 4-17, but an important step, is to then plan the next cycle and update the life-cycle plan, including partitioning the system into subsystems to be addressed in parallel cycles. This can include a plan to terminate the project if it is too risky or infeasible. Secure the management's commitment to proceed as planned.

Benefits of the win-win spiral model have been noted as: faster software via facilitated collaborative involvement of relevant stakeholders, cheaper software via rework and maintenance reduction, greater stakeholder satisfaction up-front, better software via use of architecture-level quality-attribute trade-off models, and early exploration of many architecture options.[16]

Evolutionary/Incremental

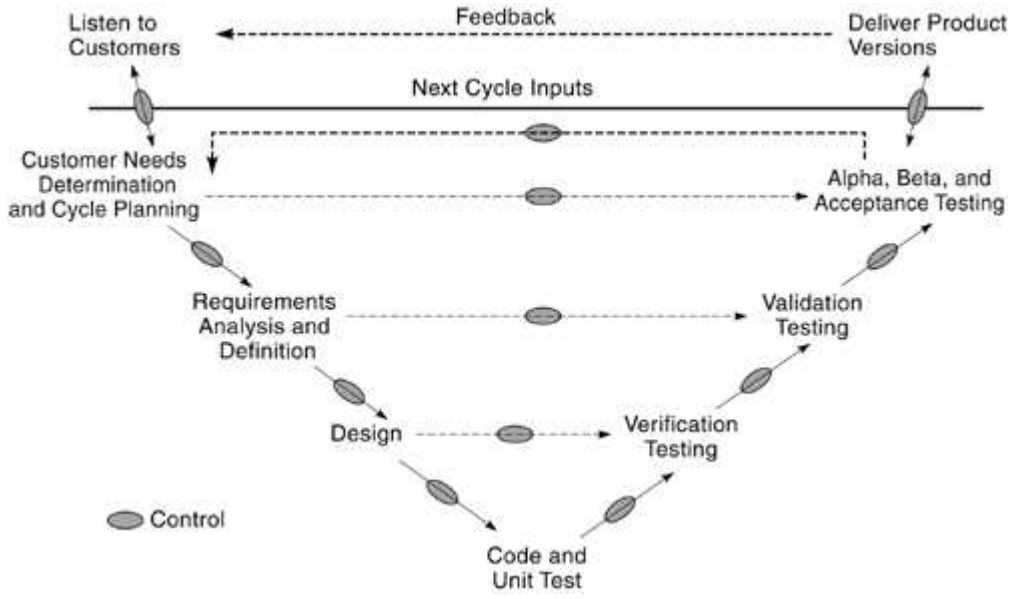
Due to their nature, the evolutionary/incremental acquisitions often encounter complications. Questions arise because each incremental build provides but a small part of the capability of the system to be acquired. In addition to normal development decision criteria, additional questions must be answered:

- Is the decision to develop this functionality for this amount of money a good idea?
- Is this the time to address the functionality question (user priorities, dictates of the evolution itself)?
- Is this a reasonable price for the functionality being added (or are we "gold-plating" one functional area before developing all required capabilities)?
- Will we run out of money before we complete the required system?

Incremental V

Figure 4-18 shows a combination model fashioned by Krasner. In *Constructing Superior Software*, he is approaching the software development life cycle model from teamwork considerations.

Figure 4-18. Incremental V Project Process Model



> Fashioning a good project life cycle model is a worthwhile up-front investment that puts all project staff on the same page … such as the traditional V model blended with the incremental, iterative development model. This model attempts to balance the need for management controls with the need for technical innovation and situation dynamics. The keys to success of the Incremental V model are what happen at the control … points. These are the formal mechanisms when management and development must jointly make explicit decisions to proceed to the next phase. Along with periodic management reviews and previews, these control points force the discussion of issues, risks, and alternatives. The meaning of each control point should be explicitly defined within the overall process. Behind such a high-level model are concrete plans based on rigorous estimates and well-defined milestones that lead down the path to success.[17]

Object-Oriented Rapid Prototyping

It is common to wonder why the subtle differences between the structured rapid prototyping, RAD, and spiral models matter—all three are user-involved, evolutionary approaches. The waterfall, V-shaped, and incremental models also have characteristics in common, such as the entry and exit phase criteria. The spiral can be thought of as an overlay of incremental, with the addition of risk management. All the models have some sort of scoping, requirements gathering, designing, developing, testing, and implementation activities.

In fact, most life cycles are adaptations and combinations, and all software evolves. Commercial products are always evolving, and government and IT shops call the later evolutionary loops "maintenance."

The project manager should feel free to select and customize a software development life cycle to suit the project needs, but he must also remember the importance of naming the phases and clearly demarking the transition from "developing" to "implementing." The importance of the

implementation or deployment line is all about control and management. There is a need for every software product to be considered "in production" at some point in time, or customers don't know when to pay the bill, users have no assurance of when functionality is stable, and the project team will not be able to baseline the product for configuration management purposes.

PREVIOUS < Free Open Study > NEXT