IN THE COMPUTER WORLD, HARDWARE IS ANYTHING YOU CAN HIT WITH A HAMMER, SOFTWARE IS WHAT YOU CAN ONLY CURSE AT.

SOURCE UNKNOWN

Introduction

Systems and Software Engineering Processes are presented in the following topic areas:

- **Introduction**
- **Life Cycles and Process Models**
- **Systems Architecture**
- **Requirements Engineering**
- **Requirements Management**
- **Software Analysis, Design, and Development**
- **Maintenance Management**

Life Cycles and Process Models

The software process is a set of tools, methods, and practices used to produce a software product. (Humphrey, 1989)[27]

In general, a software engineering process can be described using three generic phases: definition, development, and maintenance. These phases are encountered in all software work, regardless of application area, project size, or complexity. (Pressman, 1993)[53]

The project manager's plan for ordering tasks is a development model. Each model offers a different balance among the development tradeoffs. The project manager is best served by using a development model that allows him/her flexibility in the areas that are most likely to change. (Kaner, 1993)[39]

Definition Phase

The definition phase focuses on "what?" During the definition phase, the software developer and customer attempt to identify the following "what" questions:

- **What information is to be processed?**
- **What functions and performances are desired?**
- **What interfaces are to be established?**
- **What design constraints exist?**
- **What validation criteria are required to define a successful system?**

Life Cycles and Process Models (Continued)

Definition Phase (Continued)

By answering these what questions, the key requirements of the software and the system are defined. Although the methods applied during the definition phase will vary, depending on the software engineering paradigm that is applied, three specific steps will occur in some form:

1. **Customer contact - This is a research and/or consultation activity. Through customer contact (in research and development projects) or through industry or regulatory specifications, each element in a computer based system is defined, and focus is established on the role that software will play in project and product implementation.**

2. **Project planning - Once the scope of the software has been established, risks are analyzed, resources are allocated, costs are estimated, and work assignments and schedules are defined.**

3. **Requirements analysis - The scope defined for the software provides direction, but a more detailed definition of the information, behavior, and function of the software is necessary before work can begin. This technical detail comes from a formal analysis of the requirements.**

Development Phase

The development phase focuses on the "how": That is, during development, the software developer attempts to define "how":

- **Data structures are to be designed**
- **Software architectures are to be designed**
- **Procedural details are to be implemented**
- **The design will be translated into a programming language**
- **Testing will be performed**

Life Cycles and Process Models (Continued)

Development Phase (Continued)

The methods applied during the development phase will vary, but three specific steps will occur in some form:

Design - Design translates the software requirements into a set of representations (graphical, tabular, or language-based) that describe the data structure, architecture, and algorithmic procedures, as well as the nature of the human-computer interface.

Coding - Design representations must be translated into a programming language. The language may be a conventional programming language or a nonprocedural language (e. g., a language generated by a case tool) that results in instructions that can be executed by a computer.

Testing - The machine executable form must then be tested to uncover errors in function, logic, or implementation.

Maintenance Phase

The maintenance phase focuses on change that is associated with:

- **Error correction**
- **Required adaptations as the software environment evolves**
- **Enhancements brought about by changing customer requirements**

The maintenance phase reapplies the steps of the definition and development phases, but does so in the context of the existing software.

Four types of changes are encountered during the maintenance phase:

1. **Corrective - Even with the best software quality assurance activities, it is possible that the customer will uncover defects in the software. Using corrective maintenance techniques, changes are made to the software to correct defects.**

2. **Adaptive - Over time, the original environment (e.g., CPU, operating system, peripherals) for which the software was developed, is likely to change. Adaptive maintenance results in modifications to the software to accommodate changes to the external operation environment.**

Life Cycles and Process Models (Continued)

Maintenance Phase (Continued)

3. Perfective - Enhancement can be beneficial. As software is used, the customer/user will recognize additional functions. Perfective maintenance extends the software beyond its original functional requirements.

4. Reengineering - Old software is reengineered (manually or using case tools) so that its internal workings can be better understood to improve performance.

Software Engineering Paradigms

Like every other engineering discipline, software engineering encompasses a set of proven technical methods that are applied within the context of a process framework. This framework often refers to a software engineering paradigm, which is chosen based on:

- The nature of the project and application
- The methods and tools to be used
- The controls and deliverables that are required

Because software engineering is still maturing, many different paradigms have been proposed, each of which is designed to support a different perspective of software development. A software engineering paradigm provides a framework through which a software project can be effectively managed. A paradigm defines the specific technical steps that should be conducted, and by implication, the milestones that should be achieved, and the deliverables to be produced. In essence, a paradigm provides a basis for the creation of a work breakdown structure (WBS) for software engineering. (Pressman, 1992)[54]

Why be Familiar with Different Paradigms?

Over the past two decades, most organizations, that have used a disciplined approach for software development, have selected a single paradigm, and used it exclusively as a framework for software engineering development work. However, the trend today is a bit more eclectic. It is possible, and often desirable, to mix the best features of two or more paradigms or to shift the paradigms depending on the situation and development environment. It is for this reason that the software quality engineering function should be familiar with a number of the major paradigms.

Waterfall Model

The oldest paradigm for software engineering is the classic waterfall model. This paradigm takes a linear, sequential view of the software engineering process. The name comes from its appearance when viewed as drawn in Figure 4.1. Various texts illustrate this approach with 4 to 12 phase descriptions. Boehm (1983)[3] describes the process using 8 phases.

The process begins at the system level and continues through the maintenance level until product retirement. Each box encompasses a set of tasks that result in the production of one or more work products. Time is tracked from left to right and top to bottom. Each new phase begins when the work products from the previous phase are completed, frozen, and signed off. For example, the code phase will begin when the design phase is completed, and it will end when the coding is complete and before testing begins. The actual names and number of phases will often vary from project to project.

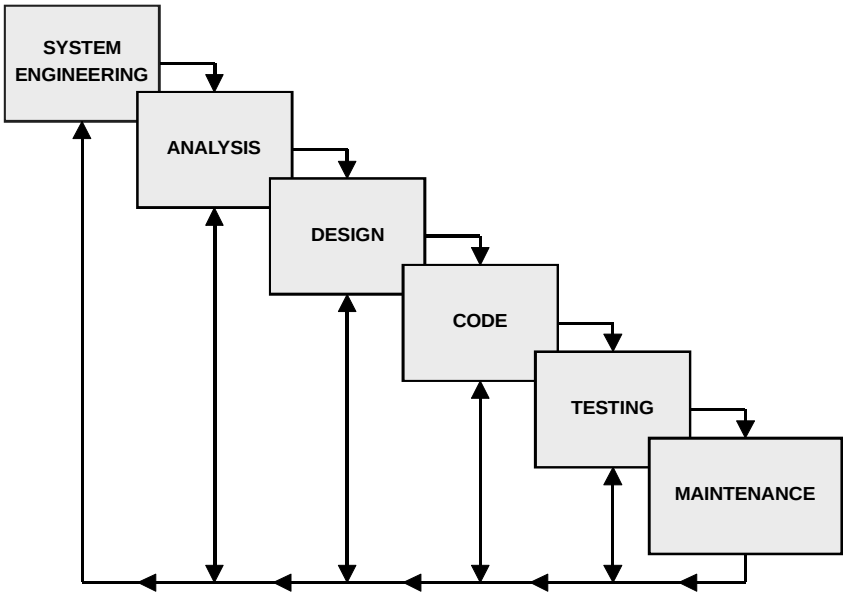


Figure 4.1 An Example Waterfall Model

Waterfall Model (Continued)

The major features of the waterfall model are the following:

System Engineering

Because software is usually part of a larger system, work begins by establishing requirements for system elements and then allocating some subset of these requirements to software. System engineering and analysis encompasses requirements gathering at a system level with a small amount of top-level design and analysis.

Analysis

In this phase, the requirements gathering process is intensified and focused specifically on software. To understand the nature of the programs to be built, the software engineer (or analyst) must understand this information domain for the software, as well as the required function, behavior, performance, and interfacing characteristics.

Design

Software design is actually a multi-step process that focuses on three distinct attributes of the program:

- **Data structures**
- **Software architecture**
- **Procedural details**

The design process translates requirements into a representation of the software. This representation can be assessed for quality, for example, through the inspection and/or review processes and placed under configuration control before coding begins.

Coding

The design must be translated into code readable form. The coding step performs this task. Code is verified, for example, through the inspection process and put under configuration management control prior to the start of the formal testing phase.

Waterfall Model (Continued)

Testing

Once code has been generated, program testing begins. Testing focuses on the logical internals of the software, ensuring that all statements have been tested, and on the functional externals; that is, conducting tests to uncover errors to ensure that the defined input will produce actual results that agree with required results.

Maintenance

Software will undoubtedly undergo change after it has been delivered to the customer (a possible exception is embedded software). Change will occur because errors have been encountered and new functionality is required. Software maintenance reapplies each of the preceding life cycle steps to an existing program, rather than a new one.

Waterfall Model Advantages/Disadvantages

The classic waterfall model has an important place in software engineering. Although it is rarely practiced in pure form, it provides a template into which methods, for analysis, design, coding, testing, and maintenance can be placed. Other life cycle models refer to the phases and tasks of the waterfall model for definitions and reference. Disadvantages of the sequential waterfall approach are:

- **Real projects rarely follow this sequential flow. In real life, iterations occur which create problems in application of the paradigm. For example, a design problem discovered during testing would require moving the entire project back into the design phase until the design is corrected and signed off. At that time, the project would reenter the coding phase until all the program changes implied by the design changes were completed and signed off. Then the project could begin a new testing phase.**

- **It is often difficult for a customer to explicitly state the requirements. In commercial software development, it may be impossible to know the requirements until the customer has had a chance to actually use the products (e.g., the first HP 35 calculator, the first Palm Pilot, or the first Apple iPhone).**

- **A working version of the software is not available until late in the project. A major defect, if undetected until system test, will cause losses and delays to the project.**

V-Model

The software engineering process may be viewed as a waterfall series of phases drawn in the form of a "V," as illustrated in Figure 4.2. This model is called the V-Model or Decomposition/Integration Model. Time progression is from left to right, and the project components become more and more specific and granular as a project moves down the left side of the "V." The components are integrated to form increasingly larger components moving up on the right side of the "V." The usefulness of the model comes from the observation that the information needed for testing components on the right is primarily from the corresponding phase on the left. For example, the requirements are the basis for identifying and designing the validation tests.

Initially, system engineering defines the role of the software which leads to software requirements analysis, where the information domain, function, behavior, performance, constraints, and validation criteria for the software are established. Moving inward along the left side of the "V," the next levels reached are design and then coding. To develop computer software, the process moves down the "V" slope that decreases the level of abstraction. (Pressman, 1992)[54]

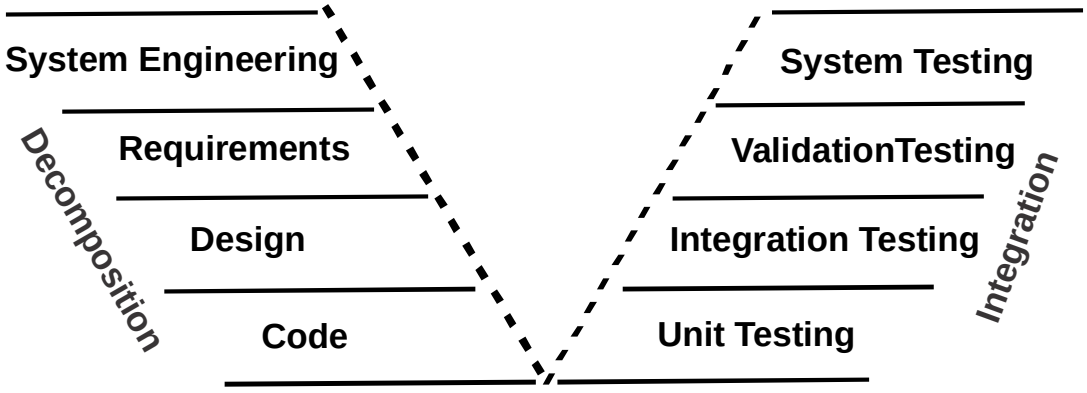


Figure 4.2 V-Model of Software Engineering Process

Adapted from (Pressman, 1992)[54]

V-Model (Continued)

The software testing element may be envisioned by moving up the right hand side of the V-Model. The sequence is described below:

Unit testing: **Concentration is on testing each unit of the software as to source code implementation. Unit testing ensures each module will function properly as a unit.**

Integration testing: **Here, the focus is on the design of the software architecture and module assembly. This testing ensures that the complete software package has been produced.**

Validation testing: **In this phase, the integrated, functional, behavioral, and performance features are validated against the software analysis requirements.**

System testing: **This is the highest level of test in which the software and other system elements are tested as a whole.**

The primary drawback to the V-Model is that better test planning and test generation can occur if one is not limited to documents and knowledge from only one particular phase. Integration and validation testing can usually be substantially improved by considering the code characteristics, and unit testing can usually be much more effective by considering requirements and design.

Object-Oriented Development

The Object-Oriented (OO) approach can be used in the context of the preceding paradigms. However, a paradigm that explicitly addresses the OO method and incorporates the notion of software component reuse has been developed.

Like other process models, the OO paradigm begins with requirements gathering. Once the overall objectives for the software are defined, inclusive of functional, behavioral, and known data requirements, Object-Oriented Analysis (OOA) begins. OOA is an activity in which classes and objects are extracted from the initial statement of the requirements. Object-Oriented Design (OOD) is then used to create a model of each of the program components known as objects. The mechanism for communication between objects is also considered. OOA and OOD occur interactively until a reasonable design for the system has been derived.

Object-Oriented Development (Continued)

Once the design model has been created, the software engineer browses a library, or repository, that contains existing program components to determine if any of the components can be used in the design at hand. If reusable components are found, they are used as building blocks to construct a prototype of the software as illustrated in Figure 4.3. It is important to note that Object-Oriented (OO) methods result in the creation of program components that are inherently reusable. Therefore, the design and implementation steps result in the creation of other reusable components from the originals. Over time, the size of the component library grows, and the need to enter the design and implementation step is reduced.

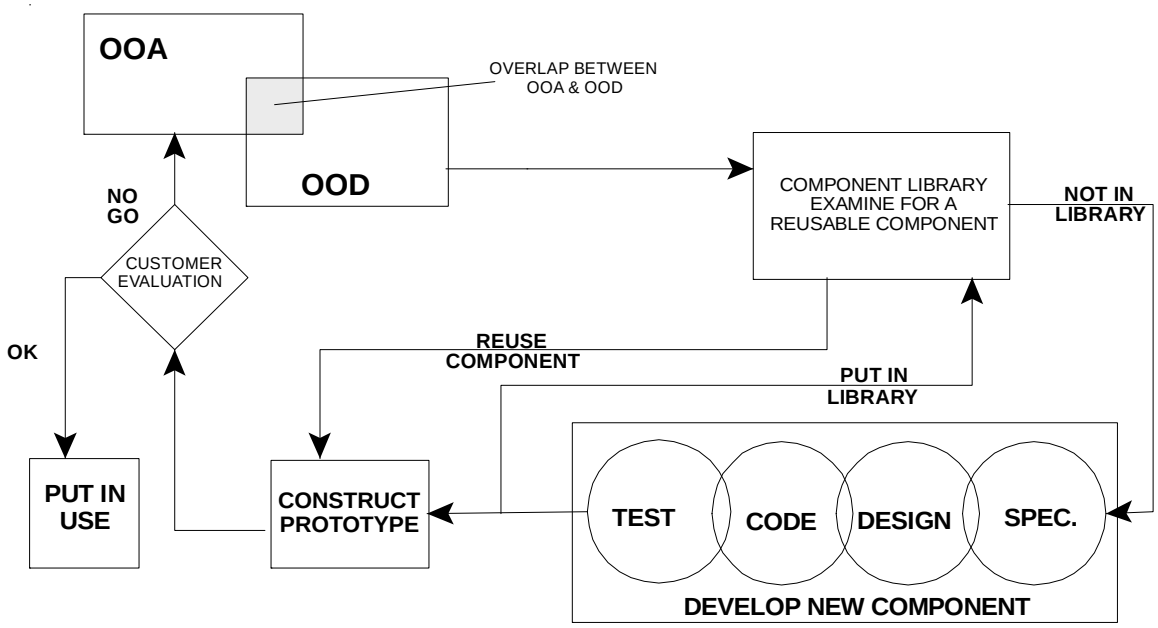


Figure 4.3 Object-Oriented Paradigm

Hybrid Development Models

Most commercial software projects use a combination of elements from more than one life cycle model. This project management approach allows experienced managers to tailor a model that best satisfies the project requirements.

Prototyping

Prototyping is a process that enables the developer to create a model of the software which is built in an evolutionary manner. A typical sequence of events for prototyping software is illustrated in Figure 4.4.

There are two variations of a prototyping approach that differ primarily in the full scale development phase. Like all approaches to software development, both prototyping models begin with requirements gathering. The developer and customer meet and define the overall requirements. A quick design occurs and focuses on those aspects of the system that will be visible to the user (e.g., input approaches and output formats). A prototype (or mockup) is created based on the design to provide the external behaviors the user will see.

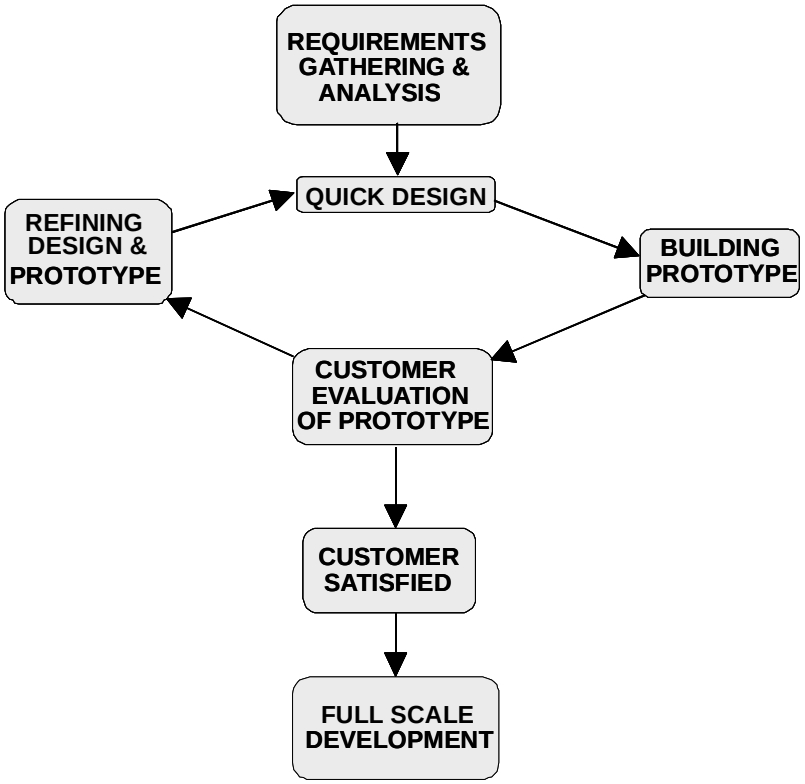


Figure 4.4 Prototyping Approach

Prototyping (Continued)

The prototype may either be built on the target environment or on a convenient platform. Either way, the prototype is evaluated by the customer/user and is used to refine the requirements. A process of tuning iterations occurs, allowing customer/user and the software developer to better understand and improve the requirements. The two prototyping approaches diverge substantially when the customer is satisfied.

If the prototype is built on the target platform, the prototyping naturally progresses until the prototype becomes complete enough that the user can begin using it. Additional work is usually required to stabilize and complete the product, but the prototyping approach continues until the product is complete.

If the prototype is built on a convenient platform, other than the one the system will be deployed upon, the accepted prototype becomes the specification of the system. Full scale development is launched (typically using a waterfall approach for design, implementation, and deployment) using the prototype as a detailed specification.

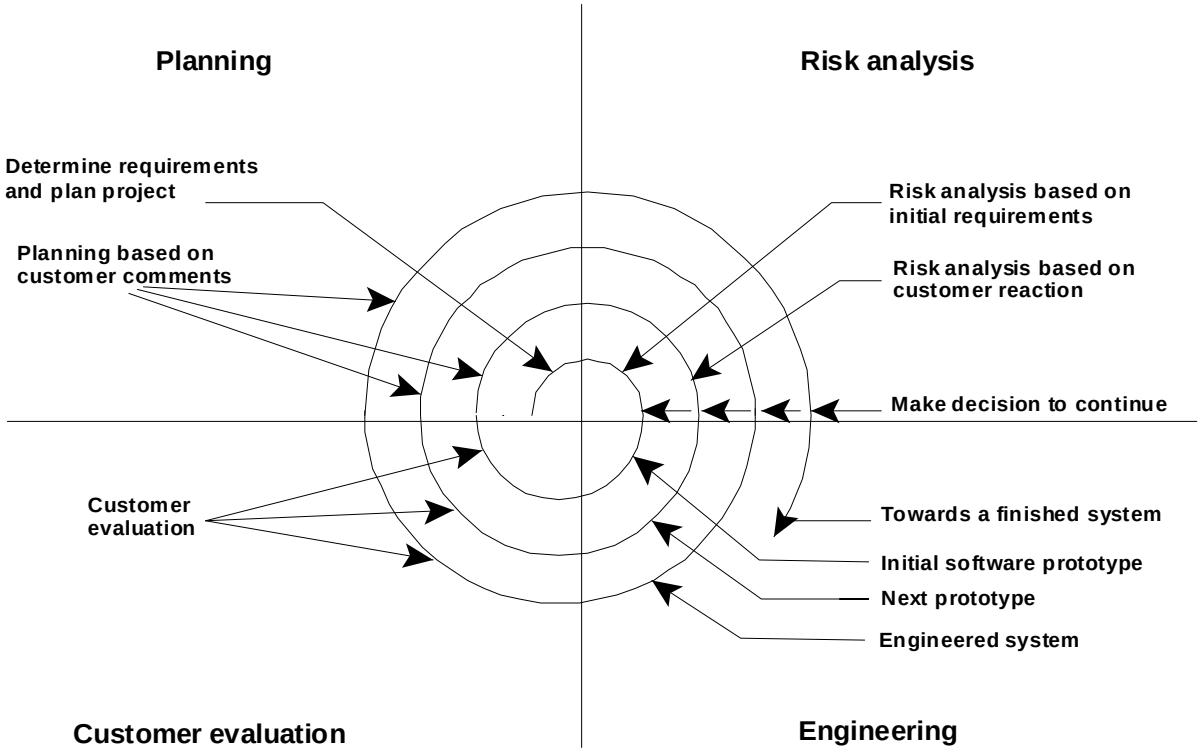
The advantages of the prototyping approach are that it provides a mechanism for identifying requirements and facilitating buy-in and agreement with the customer and/or end-user.

The disadvantages are that continued change tends to corrupt the software structure and that existence of a prototype may provide false impressions to the customer of the availability of functionality. Also, the developer may make implementation compromises, such as using an inappropriate operating system for the sake of expediency. There may be pressures to make a less than ideal design decision part of the final product. The customer and developer may spend too much time tuning and never get to the real development of the system.

Spiral Model

The spiral model is another evolutionary model. It has been developed to provide the best features of both the classic life cycle approach and prototyping, while at the same time adding the risk analysis element missing from each of these other paradigms. The model, represented by a spiral in Figure 4.5, defines four major activities represented by the four quadrants of the Figure:

1. **Planning - Determination of objectives, alternatives, and constraints**
2. **Risk analysis - Analysis of alternatives as well as an identification and/or resolution of risks**
3. **Engineering - Development of the next level of product**
4. **Customer evaluation - Assessment of the results of engineering**



(Boehm, 1983)[3]

Figure 4.5 The Spiral Model

Spiral Model (Continued)

With each iteration around the spiral, beginning at the center and working out, progressively more complete versions of the software are built. The following example illustrates one of the many approaches that may be used in implementing the spiral model.

During the first circuit around the spiral, if the results of a risk analysis indicate that there is uncertainty in the requirements, prototyping may be used in the engineering quadrant to assist both the developer and the customer. The customer then evaluates the engineering work (customer evaluation quadrant) and makes suggestions for modifications.

On the basis of the customer's input, the next phase of planning and risk analysis occurs. At each loop around the spiral, the culmination of the risk analysis results in a go/no-go decision. If risks are too great, the project can be terminated.

In most cases, however, flow around the spiral path continues, with each path moving the developer toward a more complete version of the system. Every circuit around the spiral requires engineering (lower right quadrant) that can be accomplished, using, for example, the waterfall or prototype approach. It should be noted, that the number of development activities occurring in the lower right quadrant increases as the activities move further from the center of the spiral.

The spiral model enables the developer, and the customer, to understand and react to risks at each evolutionary level. Prototyping may be used as a risk reduction mechanism at any stage in the evolution of the product. The major advantage of the spiral model is, that it maintains the systematic approach of the waterfall model but incorporates an iterative framework that more realistically reflects the real world of development. If properly applied, the spiral model should reduce design risks.

It may be difficult to convince a large customer that the evolutionary approach is controllable. Each loop around the spiral implies that project costs and schedules may be modified. This will create problems in fixed-price development. A sound understanding of risk assessment is required when using the spiral model.

(Pressman,1993)[53]