**2.2.5 The Spiral Model**

The spiral model is an evolutionary software process model that couples the iterative nature of prototyping with the controlled and systematic aspects of the linear sequential model. It provides the potential for rapid development of incremental versions of the software. Using the spiral model software is developed in a series of incremental releases. During early iterations, the incremental release might be a paper model or prototype. During later iterations, increasingly more complete versions of the engineered system are produced. ‘A spiral model is divided into a number of framework activities, also called task regions. Typically, there are between three and six task regions. Figure 2.5 depicts a spiral model that contains six task regions:

1. **Customer communication**: tasks required to establish effective communication between developer and customer.
2. **Planning**: tasks required to define resources, timelines, and other project related information.
3. **Risk analysis:** tasks required to assess both technical and management risks.
4. **Engineering**: tasks required to build one or more representations of the application.
5. **Construction and release**: tasks required to construct, test, install, and provide user support (e.g., documentation and training).
6. **Customer evaluation**: tasks required to obtain customer feedback based on evaluation of the software representations created during the engineering stage and implemented during the installation stage.

****

**Figure 2.5 the Spiral Model**

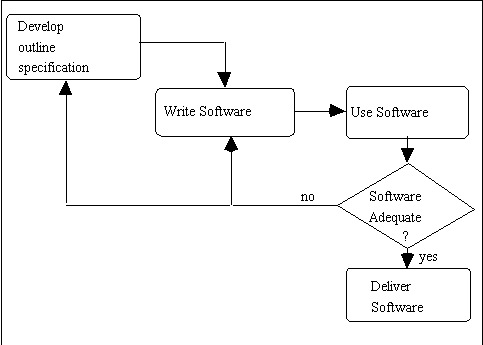
Each of the regions is populated by a set of work tasks, called a task set, that are adapted to the characteristics of the project to be undertaken. For small projects, the number of work tasks and their formality is low. For larger, more critical projects, each task region contains more work tasks that are defined to achieve a higher level of formality.

**2.2.6 The Exploratory Model**

The Exploratory- model, shown in figure 2.6, is extremely simple in its construction; it is composed of the following steps:

1. **Initial Specification Development:** using whatever information is immediately available, a brief system specification is created to provide a rudimentary starting point.
2. **System Construction / Modification (System Build):** a system is created and / or modified according to whatever information is available.
3. **System Testing:** the system is tested to see what it does, what can be learned from it, and how it may be improved.
4. **System Implementation (Delivery):** after many iterations of the previous two steps produce satisfactory results, the system is dubbed as "finished" and implemented.

This model is used in some situation in which it is very difficult, if not impossible, to identify any of the requirements for a system at the beginning of the project. Theoretical areas such as Artificial Intelligence are candidate for using the Exploratory Model, because much of the research in these areas is based on guess-work, estimation and hypothesis. In These cases, an assumption is made as to how the system might work and then rapid iterations are used to quickly incorporate suggested changes and build a usable system. A distinguishing characteristic of the Exploratory Model is the absence of precise specifications. Finishing project is based on the end result and not on its requirements.



**Figure 2.6 Exploratory Model**

**3. Software Requirements:**

The problems that software engineers have to solve are often very complex. Understanding the nature of the problems can be very difficult, especially if the system is new.

The description of the services and constraints are the requirements for the system and the process of finding out, analysis, documenting and checking these services and constraint is called requirements engineering. Requirements engineering are the services that customer requires from a system and the constraints under which it operates and is developed. These requirements should be flexible according to the business needs.

However there are three types of requirements according to the levels of description. They are:

1. **User requirements:** are what services the system is expected to provide - and the constraints under which it must operate.
2. **System requirements:** set out the system services and constraints in detail. The system requirements document, which is sometimes called a functional specification, should be precise.
3. **Software design specification:** is an abstract description of the software design which is a basis for more detailed design and implementation.

**Software system requirement**

Software system requirements are often classified as functional or non-functional requirements or as domain requirements:

**3.1 Functional Requirements:**

* Statements of services the system should provide; how the system should react to particular inputs and how the system should behave in particular situations.
* Describe functionality or system services.
* Depend on the type of software, expected users and the type of system where the software is used.
* Functional user requirements may be high-level statements of what the system should do but functional system requirements should describe the system services in detail.

**Examples of Functional requirements**

* The user shall be able to search either all of the initial set of databases or select a subset from it.
* The system shall provide appropriate viewers for the user to read documents in the document store.
* Every order shall be allocated a unique identifier (ORDER\_ID) which the user shall be able to copy to the account's permanent storage area.

**3. 2 Non-Functional Requirements:**

* Define system properties and constraints e.g. reliability, response time and storage requirements. Constraints are I/O device capability, system representations, etc.
* Process requirements may also be specified mandating a particular case system, programming language or development method
* Non-functional requirements may be more critical than functional requirements. If these are not met, the system is useless.

**Non-Functional Requirements Examples**

* The system should be easy to use by experienced controllers and should be organized in such a way that user errors are minimized.
* Experienced controllers shall be able to use all the system functions after a total of two hours training. After this training, the average number of errors made by experienced users shall not exceed two per day.

**3.3 Domain requirements:**

* Derived from the application domain and describe system characteristics and features that reflect the domain.
* May be new functional requirements, constraints on existing requirements or define specific computations.
* If domain requirements are not satisfied, the system may be unworkable.

**Domain requirements problems**

* ***Understandability***
* Requirements are expressed in the language of the application domain.
* This is often not understood by software engineers developing the system.
* ***Implicitness***
* Domain specialists understand the area so well that they do not think of making the domain requirements explicit.