Chapter 18: Software Requirements

Software requirements analysis is a process that technical practitioners use to understand software requirements. Many methods are currently available to analyze customers software requirements, and many more are emerging. The IT project manager may have to select a method for requirements analysis. This chapter presents the software development plan, various modeling techniques, and software requirements analysis specification.

Software Requirements Process

The software requirements process is the starting point for logically understanding and analyzing the customers requirements (Figure 18−1). The software requirements analysis results in the definition of a complete set of functional, performance, interface, and qualification requirements for the computer software configuration item (CSCI) and subcomponents. Graphic models are used by the software developers to analyze the requirements. The goal is to present the customer with the requirements analysis in such a graphic way that less ambiguity exists than there would be in an English−language narrative. The ultimate goals are to reduce cost and increase efficiency. This process provides the following benefits:

- It minimizes risk by encouraging systematic software development and progressive definition of detailed requirements.
- It allows frequent reviews or decision checkpoints before starting each phase.
- It uses resources efficiently because parallel software development tasks can be carried out concurrently.
- It makes management planning and control easier.
- Structured analysis or object−oriented analysis is practically language and machine independent. The main concern in this phase is to understand the customers requirements clearly and assure the customer that the requirements are understood mutually and accepted by the software developers.
- It avoids excessive maintenance because all of the necessary documentation is developed as the software evolves.
- It shortens decision time for management.
- It improves communication.
- Transition from software requirements analysis to software design and software implementation phases is smooth.
- Many commercial item (CI) tools are available to assist in the software requirements analysis, design, and implementation.

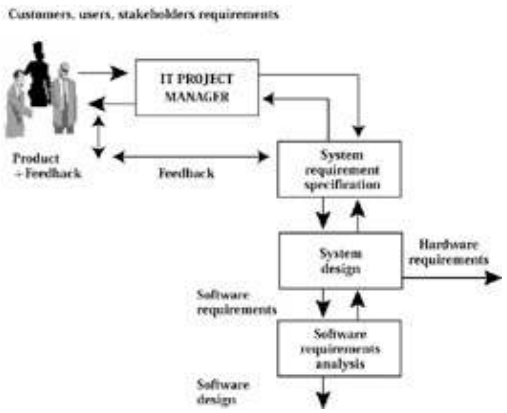

Figure 18−1: IT system design process

Software Development Plan

The software development plan describes the complete plan for development of software and how the software developers will transform the requirements for the software item into architecture. The software architecture describes its top−level structure and identifies the software components. The software developers ensure that all of the requirements for the software item are allocated to its software components and further refined to facilitate detailed design. This plan assists the customer in monitoring the procedures, management, and contract work of the organization that is performing the software development within the given time and budget.

The software developers describe their complete plan and intention for conducting software development. This includes detail of their organization, equipment, documents, facility, and training program. The plan starts with a scope, which details the systems overview and capabilities. The plan also includes the software development management plan.

The software development management plan presents the IT project organization, resources, schedule, and milestones for each activity. This includes risk management, security, safety, external interfaces, reviews (formal and informal), and all related items. The plan includes a software development relationship between the organization and resources, personnel, and software−engineering environment. Included in the plan is the use of standards and procedures while various activities are conducted. The plan explains formal and informal testing schema. It also includes software reuse, software product evaluation procedures, and tools. The plan covers configuration management (CM) and quality management factors in detail and covers the software product acceptance plan.

The plan includes the process of developing satisfactory software that may sometimes require more than one method. The IT project manager may have to select a method to be used for requirements analysis and another method for software design. The plan refers the computer language for coding logical instructions that will be used in software implementation. The plan explains the selection of a software life−cycle model.

Software Life−Cycle Model

The software life−cycle model encompasses many concepts throughout the software development. These concept phases are software requirements analysis, software design, software coding, software testing, and software maintenance. Software requirements may change with time.

Thus the life cycle provides a way of looking at various phases of software development for a system. The phases assist in going through various aspects of software development into a common language, which may be graphic, so that the effects of changes to the development process can be determined. A software life−cycle model is either a prescriptive or descriptive characterization of software evolution.

The level of detail that is supplied by the method is prescriptive in that a direction is provided on how the various activities will be accomplished. A prescriptive life−cycle model is developed more easily than a descriptive model. Many software development details can be ignored, glossed over, or generalized. This should raise concern for the relative validity and robustness of such life−cycle models when different kinds of application systems are being developed in different kinds of development settings.

A descriptive life−cycle model characterizes how software systems are actually developed. Articulation is less common and more difficult because one must observe or collect data throughout the development of a software system. This development period is an elapsed time that is usually measured in years. Also, descriptive models are specific to the systems that are observed and only generalized through systematic

analysis.

The following are types of software life−cycle models:

- Grand design
- Incremental
- Evolutionary
- Waterfall
- Prototyping and simulation
- Assembling reusable components
- Spiral
- Operational
- Transformational
- Star process
- Domain prototype
- Domain functional

Grand Design Model

The grand design model is essentially a once−through model used to determine user needs, define requirements, design the system, develop the system, test, fix, and deliver. All of the requirements are defined first.

Incremental Model

The incremental model determines the customers needs, defines the software requirements, and performs the rest of the development in a sequence of builds. The first build incorporates part of the planned capabilities, the next build adds more capabilities, and so on until the system is complete. Multiple developments are permissible in this model. All of the requirements can be defined first. Field interim products may be possible in this model.

The incremental model follows the pattern of increments of an initial subset of the system and is fully developed at that level. Then, in successive steps, more elaborate versions are built upon the previous ones. The architectural design of the total system is envisioned in the first step, but the system is implemented by this successive elaboration. The software is developed in increments (Figure 18−2), which represent degrees of functional capability. Software is built in small manageable increments. Each increment adds a new function to the system.

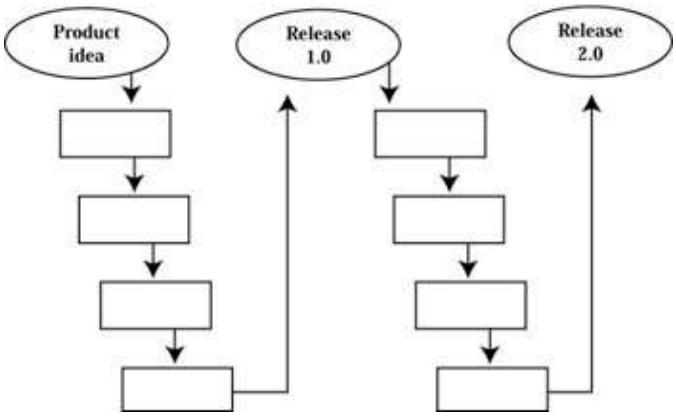

Figure 18−2: A sample of the incremental model

The strength of incremental development is that the increments of functional capability are much more easily understood and tested at a level. Use of successive increments provides a way that incorporates the customers experience into a refined product in a much less expensive way than doing all at one time.

This model combines the classic software life cycle with iterative enhancement at the level of software development. It also provides a way to periodically distribute software maintenance updates and services to scattered users. This is a popular model of software in the computer industry.

Evolutionary Model

The evolutionary model also develops a system in builds, but it differs from the incremental model in that it acknowledges that the user need is not fully understood and all requirements cannot be defined up front. User needs and system requirements are partially defined up front, and then they are defined in each succeeding build. Multiple development cycles are permissible, and field interim products are possible.

Waterfall Model

W. Royce first introduced the classic waterfall model in 1970. The waterfall model follows the pattern of waterfall for software development. Various phases of software development, such as software requirements analysis, software design, and software implementation, precede one after another in that sequence. This software evolution proceeds through an orderly sequence of transition from one phase to the next in linear order. The model divides the software process into the following phases (Figure 18−3):

- Analysis
- Design
- Implementation
- Testing

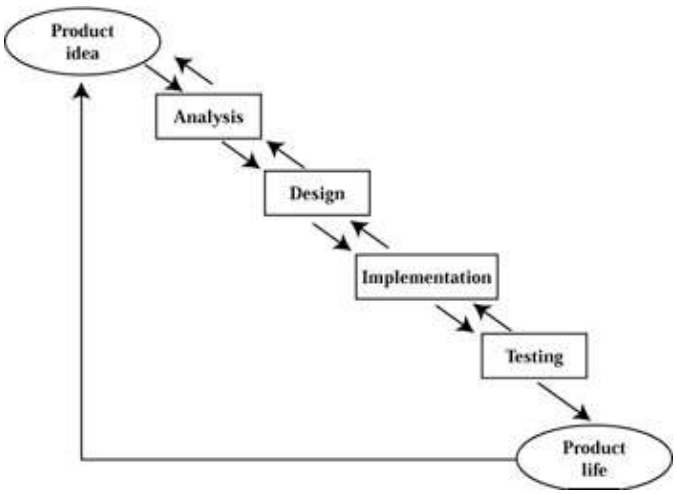

Figure 18−3: A sample of the waterfall model

Each phase is conceived in terms of input, process, and output. The software process proceeds through a sequence of steps with iterative interaction between phases, which are confined to successive steps. Deliverable software is usually produced and then used as input into the next phase.

In reality a system or software development is never so clean. As shown in Figure 18−3, feedback loops exist between phases. Because of these feedback loops, the deliverable products at each phase need reviews and changes before their acceptance. The design phase may be subdivided into many phases with an increase in

detail.

The waterfall model derives its strength from its classic steps, which are taken successively. It provides a structured template for software development. This model is not popular among object−oriented software developers because more interactions are needed between nonsuccessive steps. Identification of all of the requirements for object−oriented software is not an easy job.

The waterfall model has been useful and has helped structure and manage software development projects that are large, medium, and small in organizational settings.

The waterfall model is the phased model that partitions the software process into a sequence of distinct phases that follow each other. Each of these phases is well defined for a specific project in terms of its inputs, outputs, activities, resources, and criteria before transit to the next phase. The waterfall model expects that each phase be completed before the next one is started, as in the top−down concept (Figure 18−3). This model is well suited for well−defined and understood requirements. In practice, the model is not so clean and has many iterations of feedback in each phase from other phases. The drawback is that the distinct phases are unrealistic in practice.

Prototype and Simulation Model

The prototype and simulation model advocates the early development of components that represent the eventual system. Often these components represent the users interface to the system. A skeletal implementation of this interface is developed with the intent that an opportunity would provide feedback from the software user before the final system is specified and designed. Clarification of the user interface is one goal, but prototyping may also be employed as a concept within the context of another model. In this case, the second model of the software process may regard prototyping as but one component of the process that will be used for clarification of the behavior of the system at an early point in the development.

Prototyping technologies usually accept some form of software functional specifications as input, which in turn are simulated, analyzed, or directly executed. As such, these technologies let the designer initially skip or gloss over software design activities. In turn, these technologies can allow rapid construction of primitive versions of software systems that users can evaluate. These user evaluations can then be incorporated as feedback, which refines the emerging system specifications and designs. Depending on the prototyping technology, the complete working system can be developed through a continual process of revising and refining of the input specifications.

This model has an advantage in that it always provides a working version of the developing system while software design and testing activities are redefined for input of specification refinement and execution.

Prototyping is one of the variations on the waterfall model (Figure 18−4). This model is useful for quick results and is without proper documents to show the concept of proof for the system requirements.

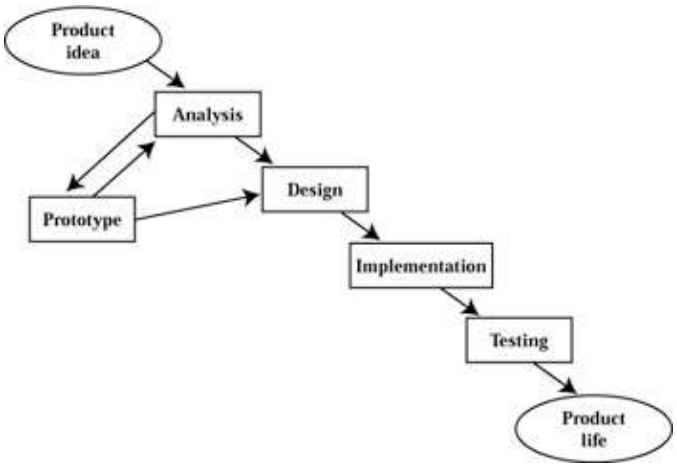

Figure 18−4: A sample of the rapid prototyping model

Assembling Reusable Components Model

The basic approach of reusability configures preexisting software components into viable application systems. However, the characteristics of the components depend on their size, complexity, and functional capability. Most approaches attempt use of components similar to the common data structure with algorithms as their manipulation. Other approaches attempt use of components resembling functionally complete systems or subsystems that are user interface management systems. Many ways probably exist for use of reusable software components in evolving software systems. However, cited studies suggest that their initial use during architectural or component design specification is a way that implementation can be made quicker. They may also be used for prototyping purposes if a suitable software prototyping technology is available.

Spiral Model

The spiral model involves multiple iteration through cycles with the intent of analyzing the results of prior phases by determining risk estimates for future phases (Figure 18−5). B. Boehm developed the model at TRW in 1988. At each phase, developers evaluate alternatives with respect to the objectives and constraints that form the basis for the next cycle of the spiral. The developer completes each cycle with a review that involves parties with a vested interest. Boehm states, The model reflects the underlying concept that each cycle involves a progression that addresses the same sequence of steps for each portion of the product and for each of its levels of elaboration from an overall concept−of−operation document down to the coding of each individual program.

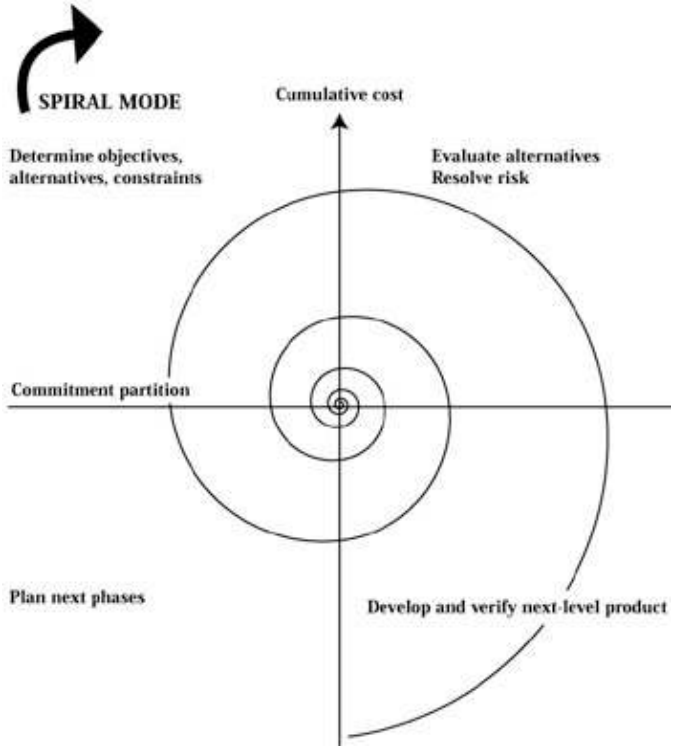

Figure 18−5: A sample of the spiral life−cycle model

The spiral life−cycle model is also a variation of the classic waterfall model. Each of the development phases is carried out in one or more cycles. Each cycle begins with a risk assessment and includes a prototype to provide insight into the risk issues.

The radial dimension represents the added incremental cost incurred in completing the developmental steps. The angular dimension represents the progress made in completing each cycle of the spiral. The basic premise of the model is that a certain sequence of steps is repeated while developing or maintaining software. The steps are first done at a high level of abstraction, and then each loop of the spiral represents a repeat of the steps at successively lower levels of abstraction.

The strength of the model lies in its flexibility for managing a system software life cycle. The developer can plan an examination of risk at each major abstraction. The model accommodates a mixture of specification−oriented, process−oriented, object−oriented, or other approaches to software development. This model is favorable by object−oriented software developers because of its strength as defined previously. A weakness of the model is a lack of matching with any existing standards. The model depends on identification and management of sources at project risk. The model needs more uniformity and consistency. This model assists in the systematic reuse process.

Operational Model

Behaviors particular to the problem domain are modeled and simulated in the beginning stages of the operational model. This is done so that the software customer can explore the way and order in which events happen. Exploration is made possible with the construction of an operational specification of the system. Concern at the specification level with how the system behaves is in contrast to other models whose specifications define the system in terms of a black box that maps the inputs to outputs.

The behavior of the problem domain is emulated in the specification, and the software structures will eventually be used in the actual system, which produce this behavior and are determined later in the

development process.

Transformational Model

The transformational model starts with a program specification and ends with a program. The transformational models progress between the two points is made through an automated series of transformations. H. Partsch and R. Steinbrueggen stated that transformation rules are partial mappings from one program scheme to another so that an element of the domain and its image under the mapping constitute a correct transformation. Transformational programming is a software process of program construction in which successive applications of transformation rule. Usually this process starts with a formal specification of a formal statement with a problem or its solution and ends with an executable program.

The benefit that is associated with the transformational model is the reduction of labor intensity of software production through the automated transformation. This model assists in preserving correctness through the application of formal transformations and replaces the final product testing by verification of the program specifications. One of this models abilities produces a desired transformation through a combination of small units from specialized programming knowledge.

Star Process Model

The star process model is a nonphased model that is useful for user interface activities. User interface development occurs in bottom−up and top−down activities. The star process model is based on empirical evidence from observations of software development. The star process model allows small localized minicycles that are important to user interface development.

Domain Prototype Model

The domain prototype model (DPM) captures in detail the domain−specific knowledge of the application in a form that lends itself to careful point−by−point verification by the domain engineer. The DPM provides, through formal models of the problem domain, a detailed and documented foundation on which requirement decisions are made. The DPM transfers the domain knowledge accurately to the software engineers and communicates the requirements analysis in a form that is easily understood and mapped into design.

DPM concepts do the following:

- Understand the customers requirements
- Show the logic to the customers graphically
- Understand the stated requirements within the domain boundary and environment
- Determine objects
- Establish relationships between objects within the domain infrastructure
- Determine instantiation criteria for objects and their relationships
- Develop functional processes (Figure 18−6)

The domain engineer receives requirements, analyzes, and looks around by accessing domains to see what assets he or she can find to reuse them within the framework.

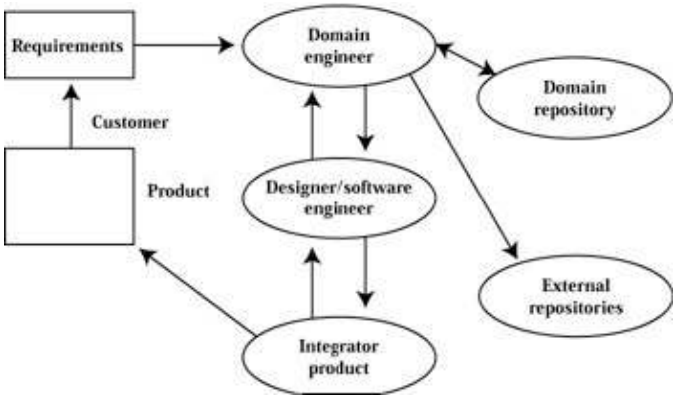

Figure 18−6: A sample of the domain prototype model

Domain Functional Model

The domain functional model (DFM) makes use of data flow diagrams and develops the required processes that drive the objects through their event chains. The following is a suggestive list of processes for the development of a data flow diagram of a single object:

- Develop an object behavior model that formalizes the behavior of an object over time.
- Analyze the action that is performed when the state is entered.
- Break the action down into a sequence of processes.
- Depict each process as a data flow diagram.
- Place each action data flow diagram on a separate page.

Object data are represented as a data store in the data flow diagram. The data store is recorded in the domain data dictionary (DDD). The data store for the object contains all of the attributes of the object. The instances of the object are created and stored in the data store. The data store holds all data and all attributes of all instances of the object. The data store is shared by all action data flow diagrams. The data flow diagram involves data stores that are made up of attributes of that data. These processes are discrete. Data flow diagrams are the offspring of a context diagram. The context diagram establishes the external interfaces of the system (Figure 18−7), and the data flow diagrams identify the internal interfaces of the system (Figure 18−8).

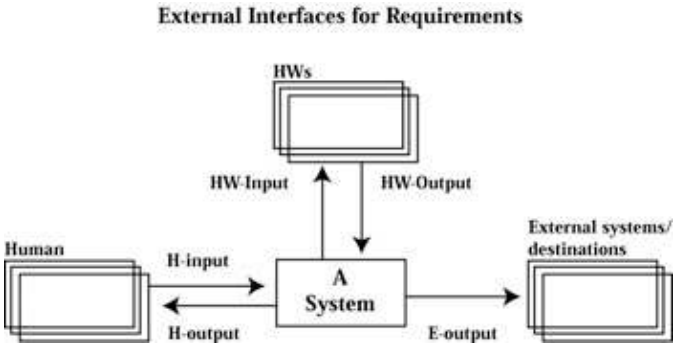

Figure 18−7: A sample context diagram

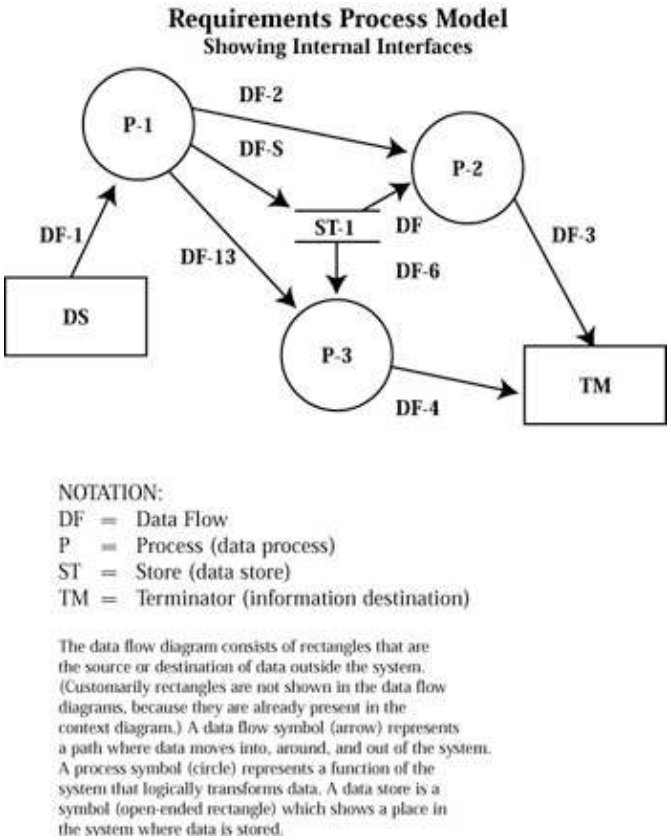

Figure 18−8: A sample data flow diagram. That data flow diagram consists of rectangles that are the source or destination of data outside the system. (Customarily, rectangles are not shown in the data flow diagrams because they are already present in the context diagram.) A data flow symbol *(arrow)* represents a path where data moves into, around, and out of the system. A process symbol *(circle)* represents a function of the system that logically transforms data. A data store is a symbol *(open−ended rectangle)* that shows a place in the system where data is stored.
DF, Data flow; *P*, process (data process); *ST*, store (data store); *DS*, data source; *TM*, terminator (information destination).

Case Study 18−1

You are proud to have a checking account in The Universe Bank. You are free to write as many checks as you want, as long as you have enough money to cover your checks written in the account. The bank has a good reputation, and there is a $25 charge for every check that is insufficient. They do not want to keep customers whose checks bounce regularly.

Once a month, the banks bookkeeper sends your canceled checks, a computerized financial statement, a balance sheet, and other documents, which they process on the bank computer Data Universe. You reconcile your checkbook for errors made by the bank or by you (e.g., while entering the transaction in your checkbook). You also record annual tax−deductible expenses in your yearly journal that will be recorded on a 1040 form. You automate your system that balances your checkbook in the most economical way.

You want to identify reusable components in the requirement. You look around and explore the organization domain repository for reusable assets with similar requirements. The name looks familiar. You access other repositories, including the common object request broker architecture (CORBA), to see if you can reuse available assets.

Almost all people maintain a checkbook, so a reusable asset must be available somewhere in a repository. The following figures provide a sample solution:

- Figure 18−9 provides a context diagram and identifies external interfaces.

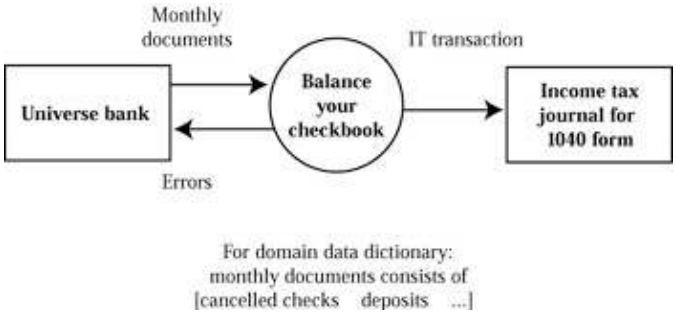

Figure 18−9: A context diagram

- Figure 18−10 partitions the requirements and identifies internal interfaces.

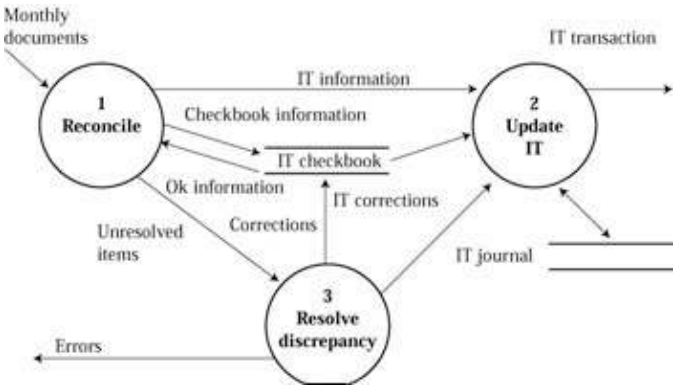

Figure 18−10: A sample of level 1 partitioning

- Figure 18−11 shows that the model can be easily partitioned to create reusable assets in future systems.

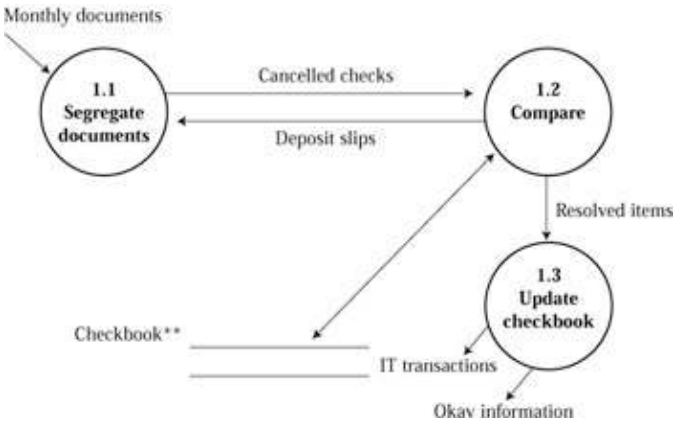

Figure 18−11: A sample of level 2 partitioning

Software Requirement Specification

The software requirement specification contains various activities and tasks for each software item. The IT project manager generates this document to pass on to the software designer, implementers, testers, and

maintainers. The software requirement specification document includes the following:

- Functional and capability specifications, including performance, physical characteristics, and environmental conditions under which the software item is to perform
- Interfaces that are external and internal to software items
- Verification and validation of software requirements
- Safety specifications
- Security specifications
- Human factor engineering
- Data definition
- Database requirements
- Installation and acceptance requirements for the delivered software
- Quality characteristics
- Configuration management criteria
- User documentation
- User operation and execution requirements
- Training outlines
- User maintenance requirements
- Requirement traceability to system requirements and system design
- Software testing
- Acceptance tests