Chapter 20: Software Implementation Process

Overview

This chapter discusses software implementation, software implementation with reuse, software testing, and configuration management (CM). The software implementation process includes the following:

- Writing source code, execute code, and test code for each identified item in the design phase
- Integrating software units and software components into software items
- Conducting software units and software components testing to ensure that they satisfy the requirements

Software Implementation

Software implementation covers coding of decomposed computer software configuration items (CSCIs) with the selected computer language. Each coded software units and components, as shown in Figure 20−1, are tested and documented in the software development files (SDF). Testing is no longer an activity that starts only after the coding phase is complete with the limited purpose of finding mistakes. Software testing is an activity that encompasses the whole development process and is an important part of the implementation.

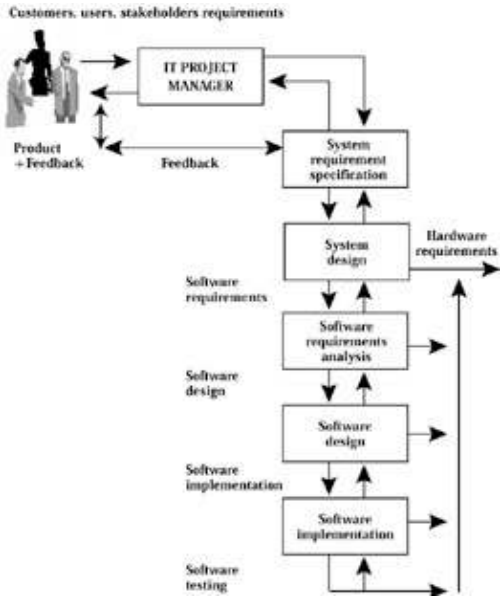

Figure 20−1: IT software implementation process

The IT manager begins to plan for testing at early stages of requirements analysis and design. He or she systematically and continuously refines the test plans and procedures as the development proceeds. These activities of planning and designing tests constitute a useful input to designers for highlighting potential weaknesses. Testing is seen as a means primarily for demonstrating that the prevention has been effective and for identifying anomalies in those cases in which it has not.

Software Implementation with Reuse

Software implementation with reuse is reality. The benefits of reusable software components are achieved if the reusable software is successfully used in another application development within the domain or in any

other domain. Reusability is one of the promises of the object−oriented approach, but the user must plan in advance to achieve the benefits of reuse. Many different versions of reuse exist in the industry, including code, inheritance, template, component, framework, pattern, artifact, and asset. The bottom line is not reusable; it cannot be used more than once.

Identification of software depends on whether it is a new development, reuse of the available existing software, reengineering of the software, or a reverse engineering process. Many commercial−off−the−shelf (COTS) software products are available that suit the requirements. Domain knowledge and experience of the system analyst or engineer play a major role in the selection decision. The selection must be based on development of the right software for the right requirements.

New Development

In new development, various software development phases are met. The major phases are domain analysis, requirements analysis, design, and implementation. Developers identify commonalties and differences in domain analysis. These can be further divided into subphases, each of which are reiterative. This is an example of a typical waterfall model. The models must be shared among the group of developers who seek to integrate software. The model must be well designed, consistent, and based on a proper design method. The model may be constantly evolving with new requirements or modifications.

The new development normally follows a set of prescribed standards, methods, and computer−aided software engineering (CASE) tools. Traditionally, the process moves from high−level abstractions and logical implementation−independent designs to the physical implementation of a system. It follows a sequence from requirements through the design of the implementation. It leads forward to a new development of software throughout the life−cycle phases. The process starts at the initial phase of analysis of the new requirements and progresses to the development of all phases of analysis until the project is completed.

Reusable software is developed in response to the requirements for one application and can be used in whole or in part for the satisfaction of requirements of another application. The domain analyst must find a way for the existing software that is pretested and cost and time efficient to be reused. The basic approach is configuration and specialization of preexisting software components into viable application systems. Cited studies suggest that initial use of reusable software during the architectural design specification is a way to speed up implementation.

Reverse Engineering

Reverse engineering extracts design artifacts and the building or synthesizing of abstractions that are less implementation dependent. This process implements changes that are made in later phases of the existing software and automatically brings back the early phases. It starts from any level of abstraction or at any stage of the life cycle. It covers a broad range that starts from the existing implementation, recaptures or recreates the design, and finally deciphers the requirements that the system implements.

Reengineering

Reengineering is the renovation, reclamation, examination, and alteration of the existing system software for changing requirements. This process reconstitutes the existing system software into a new form and the subsequent implementation of the new form. This dominates during the software maintenance life cycle. This helps the user identify and separate those systems that are worth maintaining from those that should be replaced.

Exploration of the Internet for Suitable Reusable Assets

Exploration of the Internet for suitable reusable assets is a mechanism to integrate reuse into software implementation. This is a process of building software systems by integration of reuse assets from the Internet repositories. Figure 20−2 shows a scheme for integration of software systems with reuse assets. For example, the software development team receives and analyzes new requirements for a system. The domain engineer also examines the requirements and confirms whether the requirement is already present in any domain repository. The software developers get the reusable well−tested assets and save time and money in the integration of a software system. Sometimes the developers may have to write and test software interfaces.

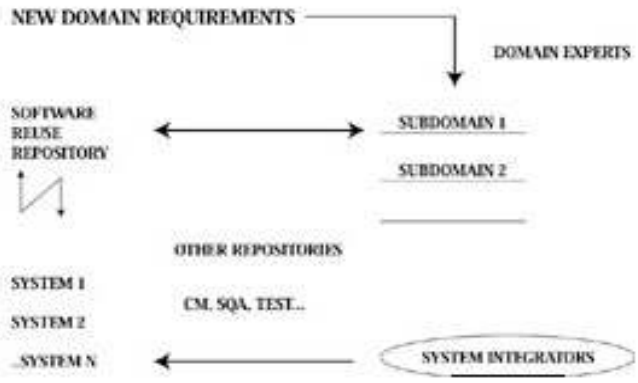

Figure 20−2: Integration of software systems with reuse assets

Case Study 20−1

The town department of Fort Cobb, Oklahoma, needs a traffic light (TL) on the main street. About 3000 residents live in this town, most of whom travel for jobs to nearby cities. Another road intersects the main street.

The object TL establishes a proper relationship with other objects, such as a clock, traffic sensor, traffic, intersection, and control panel. The TL attributes are manufacturer, model, serial number, and road intersection name. The clock attributes are manufacturer, model, serial number, TL manufacturer, TL model, and TL serial number. The traffic sensor attributes are manufacturer, model, serial number, TL manufacturer, TL model, TL serial number, and road name. The traffic attributes are road name, TL manufacturer, TL model, and TL serial number. The intersection attribute is road name. The control panel attributes are manufacturer, model, serial number, button selected, TL manufacturer, TL model, and TL serial number.

This case study and others discussed in this chapter are common scenarios in the industry. An important question is why these scenarios have to be designed from the beginning when enough information is available for reuse of assets and savings of time and money. Figure 20−3 shows a sample partial solution.

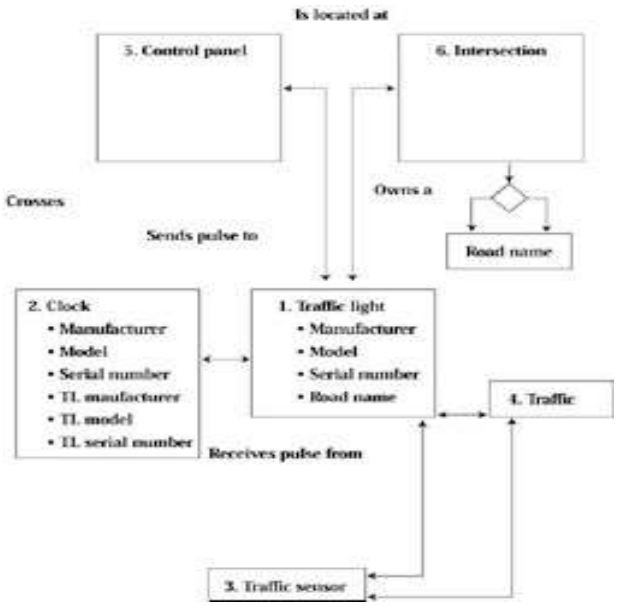

Figure 20−3: A sample solutions

The software implementers test each software unit and database, ensuring that it satisfies its requirements. They document the test procedures, data, and test results for each software unit and database in an SDF. They evaluate the software code and test results for the following criteria:

- Traceability to the requirements and design of the software item
- External consistency with the requirements and design of the software item
- Internal consistency between unit requirements
- Test coverage of units
- Appropriateness of coding methods and standards used
- Feasibility of software integration and testing
- Feasibility of operation and maintenance

The software implementers update the user documentation as necessary. They also update the test requirements and schedule for software integration.

Software Testing

Software testing is the verification and validation of the specific requirement. Verification means that software analysts check the observed behavior against functional specification. Validation means that analysts check the observed behavior against the customers expectations. The software testing consists of the dynamic verification of the behavior of a program on a finite set of test cases, suitably selected from the usually infinite execution domain, against the specified expected behavior. Dynamic testing means the code execution and implies execution of the program on valued inputs. Static analysis techniques involve peer review and inspection program execution on symbolic inputs or symbolic evaluation. Static testing indicates no code execution.

Many test cases are theoretically possible for even simple programs in which exhaustive testing could require years to execute. The number of executions that can realistically be observed in testing must be manageable. Testing always implies a trade−off between limited resources, schedules, and inherently unlimited test requirements. The IT manager decides whether the observed outcomes of program execution are acceptable; otherwise the testing effort would be useless. Software testing is a means for quality evaluation.

Software Testing Best Practices

Software testing best practices includes static and dynamic analysis techniques as discussed in the following sections.

Static Analysis Techniques

- Review, walk through, and inspection practices use visual review of a portion of technical documentation to detect errors. They are usually conducted by small peer groups of technical professionals who have vested interest, using software requirement documents, specification, design description, and source code listings to do a line−by−line inspection, walk−through, and reviews. These practices ensure that the differentiated software requirements are present and satisfied.
- The code auditor is a software programmer who examines the source code listing that he or she follows and complies with set standards and practices.
- The interface checker uses automated tools to analyze software requirements specification, design description, and source code listing to detect errors in information control passed between software units and components.
- Physical units. An automated tool to ensure consistency in software units that are involved in computations conducts testing. These practices ensure that the resultant units are rational in the real world.
- Data flow analysis practices ensure that sequential events occur as scheduled.
- Structural analysis detects violations of control flow standards, including improper calls to routines and subroutines, infinite loops, and recursive relationships in the source code listing.
- Cross−reference programs compile a list of where each data item is used in the source code listing.
- Path analysis generates test data sets to evaluate selected paths in the source code.
- Domain testing detects logical errors when input follows the wrong path.
- Partition analysis detects missing paths, incorrect operators, domain logical errors, and computation errors.
- Complexity analysis examines coded algorithms for simplicity.

Dynamic Analysis Techniques

- Cause−effect analysis uses input data systematically selected to have a high probability of detecting errors in output data.
- Performance−measuring techniques monitor software execution to detect code or execution inefficiencies for evaluation of central processing unit (CPU) cycles and waiting times.
- Path and structural analysis monitors the number of times that a specific line of source code is executed. The source code is classified as statements, branches, or a path.
- Interactive debugging allows the programmer to suspend execution at any point and examine the program status.
- Random testing uses random samples of input data and evaluates the resulting output. These practices generate unexpected conditions to evaluate program performance in those cases.
- Function testing is the most commonly practiced technique. This executes source code using specified controlled input to verify that the functions are performed as expected.
- Mutation analysis studies the behavior of a large number of different versions of the original program. Introducing a small number of errors in each version and studying how the software system responds can generate these practices.
- Error seeding is a statistical approach used to determine the number of errors remaining in large software systems by determining the effectiveness of testing. These practices determine the number of known errors that are detected, indicating the number of unknown errors remaining.

Case Study 20−2

A message processing system is to be developed to provide user communication with a central host computer. The user can compose a message, print a message on a local printer, send a message to a host, read a message from a host, and log a message sent to a host in a local floppy disk file.

A message is a free−format group of characters up to 1024 bytes long. A message buffer is circular, and if the message length exceeds 1024 bytes, then the first character will be overwritten. Commands from the user are single keystroke control codes:

- CRTL−P prints the message.
- CTRL−S sends the message.
- CTRL−R reads the message received from the host.
- CTRL−L invokes the logging control submenu.

As messages are received from the host, they are stored in a buffer. Each message received overwrites the message previously stored in the buffer so that the user can read only the latest message received at any particular time. The user can read the messages as many times as he or she wishes. If the user is reading a host message, then he or she must press the escape (ESC) key to resume message composition. He or she can log messages that are sent to a local floppy disk. The user controls this through a submenu activated by the CTRL−L command.

This case study is a typical example of a real−time system. Figs. 20−4 to 20−6 show a sample solution. In Figure 20−4, a dotted line indicates one bit of information and is represented by the label CTRL−L. In Figure 20−5, the storage buffer is shown a few times to avoid confusion. Each duplicate buffer is noted by *. The dotted circle number 5 will be further transformed into STD as shown in Figure 20−6.

Figure 20−4: Functional model (context diagram)

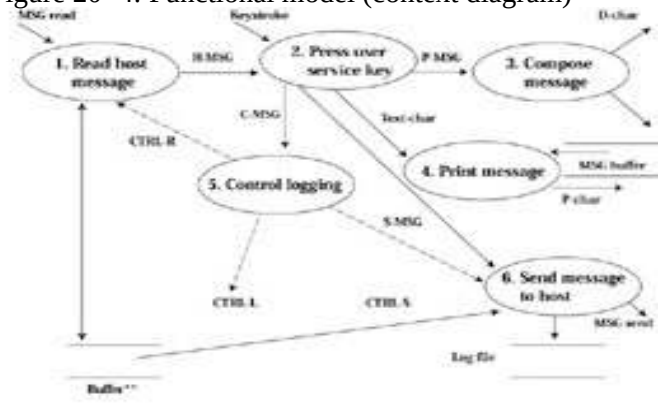

Figure 20−5: Functional model (data flow diagram)

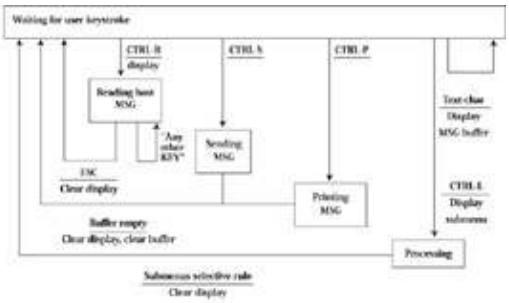

Figure 20−6: STD sample solution

As the software developer writes source codes for a requirement or modifies an existing software unit, he or she formulates and develops software test cases and procedures to verify that the completed software operates correctly. Once the developer tests and verifies the software unit, he or she integrates it with other software system units. Together they are logically cohesive to the modified software units, thus creating software components.

At the software component level, the software developer again conducts a series of tests to verify that the modified software operates correctly and interfaces with other software components that are associated with the modified software unit. The software developer determines how comprehensive the testing must be to verify that the modified software functions correctly and interacts properly with other software units that use the output from the modified software unit.

At a minimum, the software developer must conduct tests to verify the correct operation of all units that are affected by the modified software units. If a software unit is dependent on another software unit that has been modified, then the developer must also test the dependent software unit. The software developer also conducts tests on other integrated software units that are affected by the modified software unit. Once the developer verifies that the software components operate correctly, he or she integrates it in an iterative process with the remaining modified software components until the entire CSCI is built or rebuilt.

Software Integration Testing

Software integration testing is a plan to integrate software units and software components into the software item for testing. The IT project manager establishes the plan and includes test requirements, procedures, data, responsibilities, and schedule. The software implementers integrate the software units and software components and test as the aggregates are developed in accordance with the integration plan. The project manager ensures that each aggregate satisfies the requirements of the software item and that the software item is integrated at the conclusion of the integration activity.

For each qualification requirement of the software item, the software implementers develop and document a set of tests, test cases, use cases (inputs, outputs, and test criteria), and test procedures for conducting software qualification testing. The software implementers evaluate the integration plan, design, code, tests, test results, and user documentation considering the following criteria:

- Traceability to the software system requirements
- External consistency with the software system requirements
- Internal consistency
- Test coverage of the requirements of the software item
- Appropriateness of test standards and methods used

- Conformance to expected results
- Feasibility of software qualification testing
- Feasibility of operation and maintenance

The software implementers conduct the software qualification test (SQT). Upon successful completion of the SQT, the software developers establish a baseline for the design and code of the software item. They place the software unit components under CM control.

Configuration Management

CM is a discipline that applies technical and administrative direction and surveillance to identification and documentation of the functional and physical characteristics of a configuration item. The CM activities are grouped into four functions:

1. ***Configuration identification.*** Configuration identification activities identify, name, and describe the documented physical and functional characteristics of the code, specifications, design, and data elements to be controlled for the project. Controlled items are the intermediate and final outputs, such as executable code, source code, user documentation, program listings, databases, test cases, test plans, specifications, and management plans. Controlled items include the support environment, such as compilers, operating systems, programming tools, and test beds. The software implementers control changes to those characteristics and record and report changes to processing and implementation status. The configuration item (CI) is an aggregation of hardware and computer programs or any discrete portion thereof that satisfies an end−use function and is designated by the customer or software system developers for CM.
2. ***Configuration control.*** Configuration control activities request, evaluate, approve or disapprove, and implement changes to baseline CIs. Changes encompass error correction and enhancement.
3. ***Status accounting.*** Configuration status accounting activities record and report the status of the project CI.
4. ***Configuration audits and reviews.*** Configuration audits determine to what extent the CI reflects the required physical and functional characteristics. Configuration reviews are management tools for establishment of a baseline.

The IT manager establishes the CM section. The objective of CM is to assist the IT manager in achieving the following:

- Lowest life−cycle cost for software development and maintenance
- Good performance
- Realistic schedule
- Operational efficiency
- Logistic support
- CI readiness
- Audit trail

The IT manager introduces appropriate CM controls throughout the softwares development and maintenance phases. This policy achieves maximum efficiency in management of changes with respect to the cost, timing, and implementation, and it ensures uniformity in policy, procedures, data, forms, and reporting throughout the organization.

The IT manager should clearly define the requirements for a CM plan at the early stages of the software systems development and maintenance. The manager should employ baselines throughout the life cycle of a

CI to ensure an orderly transition from one major commitment point to the next in the software systems development, production, and logistic support processes. The manager should establish baselines at those points in a program where it is necessary to define a formal departure point for control of future changes in performance, design, production, and related technical requirements.

Software Change Process

A small change in software can start a chain reaction. The software change process helps in the management of such a chain reaction. The software developers should limit changes to those that are necessary or offer significant benefits. The degree of benefits is directly related to the promptness with which actions are processed. The software developers should make changes only for the following reasons:

- To correct deficiencies
- To satisfy changes in operational or logistic support requirements
- To effect substantial life−cycle cost savings
- To prevent or allow desired slippage in an approved schedule

Preparation of the Engineering Change Proposal

The engineering change proposal (ECP) requires a complete analysis of the influence of the implementation of the software change that it proposes. It requires that the proposal submitted with an ECP contain a description of all known interface effects and information concerning changes required in the functional/allocated/product baselines. The ECP further requires the submission of supporting data outlining the influence on integrated logistic support and overall estimated cost.

Configuration Control Board

The configuration control board (CCB) decides to take proper action concerning software change evaluations, discrepancy reports, processing, approval or disapproval, and implementation. The most important activity of the CCB is the evaluation of discrepancy reports and requests for software changes. The following are some of the factors that can guide decision making:

- Size of the change
- Time
- Complexity of the change
- Influence on the system architecture
- Cost
- Criticality of the area involved
- Influence on other software changes in progress
- Test requirements
- Available resources

The CCB is the official agency to act on all proposed software changes. The chairman of the CCB makes the final decision on all changes unless otherwise stated. The CCB critically evaluates every proposed configuration change, taking into consideration all aspects of the change on a CI and the associated CIs with which it interfaces. Such aspects include the following:

- System design
- Software design

- Reliability
- Performance reliability
- Maintainability
- Cost
- Schedule
- Operational effectiveness
- Safety
- Security
- Human factors
- Logistic support
- Transportability
- Training
- Reusability
- Modularity