**CHAPTER 6 – ARCHITECTURE DESIGN**

Section 6.1 and 6.2 are in notebook

**6.3 - ARCHITECTURAL PATTERNS**

The idea of patterns as a way of presenting, sharing, and reusing knowledge about software systems is now widely used.

**WHAT IS AN ARCHITECTURAL PATTERN?**

An architectural pattern should describe a system organization that has been successful in previous systems. It should include information of when it is and is not appropriate to use that pattern, and the pattern’s strengths and weaknesses.

----- The layered architecture and MVC patterns are examples of patterns where the view presented is the conceptual organization of a system.

1. **Model View Controller (MVC)**

Figure 6.3 - The organization of the MVC (pg. 156).

Figure 6.4 **-** Web application architecture using the MVC pattern (pg. 157).

This pattern is the basis of interaction management in many web-based systems. The stylized pattern description includes the pattern name, a brief description with an associated graphical model, and an example of the type of system where the pattern is used. You should also include information about when the pattern should be used and its advantages and disadvantages.

* MVC separates elements of a system, allowing them to change independently.

The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction.

Model - Encapsulates Application State, Notifies View of State Changes.

View - Renders Model, Requests Model Updates, Sends User Events to Controller.

Controller **-** Maps User Actions to Model Updates Selects View.

**6.3.1 - Layered Architecture**

The layered architecture pattern is another way of achieving separation and independence. The system functionality is organized into separate layers, and each layer only relies on the facilities and services offered by the layer immediately beneath it. This layered approach supports the incremental development of systems. As a layer is developed, some of the services provided by that layer may be made available to users.

The architecture is also changeable and portable. When layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected. As layered systems localize machine dependencies in inner layers, this makes it easier to provide multi-platform implementations of an application system.

Layered architecture with four layers:

- The lowest layer includes system support software — typically database and operating system support. -- The next layer is the application layer that includes the components concerned with the application functionality and utility components that are used by other application components.

-The third layer is concerned with user interface management and providing user authentication authorization,

- The top layer providing user interface facilities.

----- The number of layers is arbitrary. Any of the layers could be split into many layers.

Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system.

Figure 6.5 - The layered architecture pattern (pg. 158)

Figure 6.6 - A generic layered architecture (pg. 158)

Figure 6.7 - The architecture of the LIBSYS system (pg. 159)

**6.3.2 – Repository Architecture**

Describes how a set of interacting components can share data. The majority of systems that use large amounts of data are organized around a shared database or repository. This model is therefore suited to applications in which data is generated by one component and used by another.

---- Examples: Command and control systems & management information systems.

All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.

Figure 6.8 - The repository pattern (pg. 159)

Figure 6.9 - A repository architecture for an IDE - Shows an IDE that includes different tools to support model-driven development (pg. 160)

--------- The repository pattern is concerned with the static structure of a system and does not show its run-time organization.

**6.3.3 - Client–server architecture**

Figure 6.10 - The client–server pattern (pg. 161)

Figure 6.11 - A client—server architecture for a film library (pg. 162)

The major components of this model are:

1. A set of servers that offer services to other components. Examples: Print servers, file servers and compile server.

2. A set of clients that call on the services offered by servers. There will normally be several instances of a client program executing concurrently on different computers.

3. A network that allows the clients to access these services. Most client–server systems are implemented as distributed systems, connected using Internet protocols.

------ In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.

Client–server architectures are usually thought of as distributed systems architectures, but the logical model of independent services running on separate servers can be implemented on a single computer. Again, an important benefit is separation and independence. Services and servers can be changed without affecting other parts of the system.

Clients may have to know the names of the available servers and the services that they provide. However, servers do not need to know the identity of clients or how many clients are accessing their services. Clients access the services provided by a server through remote procedure calls using a request-reply protocol such as, http.

The most important advantage of the client–server model is that it is a distributed architecture. Effective use can be made of networked systems with many distributed processors. It is easy to add a new server and integrate it with the rest of the system.

------ In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.

**6.3.4 - Pipe and Filter Architecture**

------ The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.

This is a model of the run-time organization of a system where functional transformations process their inputs and produce outputs. Data flows from one to another and is transformed as it moves through the sequence.

Each processing step is implemented as a transform. Input data flows through these transforms until converted to output. The transformations may execute sequentially or in parallel. The data can be processed by each transform item by item or in a single batch.

The name ‘pipe and filter’ comes from the original UNIX system where it was possible to link processes using ‘pipes’. These passed a text stream from one process to another.

The term ‘filter’ is used because a transformation ‘filters out’ the data it can process from its input data stream.

When transformations are sequential with data processed in batches, this pipe and filter architectural model becomes a batch sequential model, a common architecture for data processing systems (e.g., a billing system).

The architecture of an embedded system may also be organized as a process pipeline, with each process executing concurrently.

Interactive systems are difficult to write using the pipe and filter model because of the need for a stream of data to be processed. Although simple textual input and output can be modeled in this way, graphical user interfaces have more complex I/O formats and a control strategy that is based on events such as mouse clicks or menu selections.

It is difficult to translate this into a form compatible with the pipelining model.

Figure 6.12 - The pipe and filter pattern (pg. 162)

Figure 6.13 – An example of the pipe and filter architecture (pg. 163)

**NOTE:** While Design Patterns impact a specific section of the code base, Architectural Patterns are high-level strategies that concern large-scale components, the global properties and mechanisms of a system.

**6.4 - APPLICATIONS ARCHITECTURE**

Application systems are intended to meet a business or organizational need.

Application architectures encapsulate the principal characteristics of a class of systems.

The application architecture may be re-implemented when developing new systems but, for many business systems, application reuse is possible without reimplementation.

A software designer can use models of application architectures in a number of ways:

1. As a starting point for the architectural design process, if you are unfamiliar with the type of application that you are developing, you can base your initial design on a generic application architecture. Of course, this will have to be specialized for the specific system being developed, but it is a good starting point for design.

2. As a design checklist, if you have developed an architectural design for an application system, you can compare this with the generic application architecture. You can check that your design is consistent with the generic architecture.

3. As a way of organizing the work of the development team, the application architectures identify stable structural features of the system architectures and in many cases, it is possible to develop these in parallel. You can assign work to group members to implement different components within the architecture.

4. As a means of assessing components for reuse, if you have components you might be able to reuse, you can compare these with the generic structures to see whether there are comparable components in the application architecture.

5. As a vocabulary for talking about types of applications, if you are discussing a specific application or trying to compare applications of the same types, then you can use the concepts identified in the generic architecture to talk about the applications.

Describing the following architectures of two types of application:

1. **Transaction processing applications** Transaction processing applications are database-centered applications that process user requests for information and update the information in a database. These are the most common type of interactive business systems. They are organized in such a way that user actions can’t interfere with each other and the integrity of the database is maintained. This class of system includes interactive banking systems, e-commerce systems, information systems, and booking systems.

2. **Language processing systems** Language processing systems are systems in which the user’s intentions are expressed in a formal language (such as Java). The language processing system processes this language into an internal format and then interprets this internal representation. The best-known language processing systems are compilers.

**6.4.1 – Transaction Processing System**

Transaction processing (TP) systems are designed to process user requests for information from a database, or requests to update a database.

Technically, a database transaction is sequence of operations that is treated as a single unit (an atomic unit). All of the operations in a transaction have to be completed before the database changes are made permanent. This ensures that failure of operations within the transaction does not lead to inconsistencies in the database.

From a user perspective, a transaction is any coherent sequence of operations that satisfies a goal.

----- Example given on pg. 166

----- Transaction processing systems are usually interactive systems in which users make asynchronous requests for service.

Figure 6.14 - The Database structure of transaction processing applications (pg. 166)

Figure 6.15 - The software architecture of an ATM system (pg. 167)

------ Transaction processing systems may be organized as a ‘pipe and filter’ architecture with system components responsible for input, processing, and output.

**6.4.2 – Information System**

An information system allows controlled access to a large base of information, such as a library catalog, a flight timetable, or the records of patients in a hospital.

The system is modeled using a layered approach.

Figure 6.16 - Layered information system architecture (pg. 168)

Figure 6.17 – The architecture of the MHC-PMS (pg. 168)

These systems are often implemented as multi-tier client server/architectures:

1. The web server is responsible for all user communications, with the user interface implemented using a web browser.

2. The application server is responsible for implementing application-specific logic as well as information storage and retrieval requests.

3. The database server moves information to and from the database and handles transaction management.

**6.4.3 – Language processing Systems**

Language processing systems translate a natural or artificial language into another representation of that language and, for programming languages, may also execute the resulting code.

Figure 6.18 - The architecture of a language processing System (pg. 170)

A possible architecture for a language processing system for a programming language is illustrated in Figure 6.18. The source language instructions define the program to be executed and a translator converts these into instructions for an abstract machine. These instructions are then interpreted by another component that fetches the instructions for execution and executes them using (if necessary) data from the environment. The output of the process is the result of interpreting the instructions on the input data.

Programming language compilers that are part of a more general programming environment have a generic architecture (Figure 6.19) that includes the following components:

1. A lexical analyzer, which takes input language tokens and converts them to an internal form.

2. A symbol table, which holds information about the names of entities (variables, class names, object names, etc.) used in the text that is being translated.

3. A syntax analyzer, which checks the syntax of the language being translated. It uses a defined grammar of the language and builds a syntax tree.

4. A syntax tree, which is an internal structure representing the program being compiled.

Figure 6.19 - A pipe and filter compiler architecture (pg. 170)

Figure 6.20 - A repository architecture for a language processing system (pg. 172)

**CHAPTER 8 – SOFTWARE TESTING (Somerville)**

Testing is intended to show that a program does what it is intended to do and to discover program defects before it is put into use.

The testing process has two goals:

1. To demonstrate to the developer and the customer that the software meets its requirements.

- For custom software, this means that there should be at least one test for every requirement in the requirements document.

- For generic software products, it means that there should be tests for all of the system features, plus combinations of these features, that will be incorporated in the product release.

1. To discover situations in which the behavior of the software is incorrect, undesirable, or does not conform to its specification. These are a consequence of software defects.

Defect testing is concerned with rooting out undesirable system behavior such as system crashes, unwanted interactions with other systems, incorrect computations, and data corruption.

The first goal leads to validation testing, where you expect the system to perform correctly using a given set of test cases that reflect systems’ expected use.

The second goal leads to defect testing, where the test cases are designed to expose defects.

-- The test cases in defect testing can be deliberately obscure and need not reflect how the system is normally used.

-- During validation testing, you will find defects in the system; during defect testing, some of the tests will show that the program meets its requirements.

-- Testing cannot demonstrate that the software is free of defects or that it will behave as specified in every circumstance. It is always possible that a test that you have overlooked could discover further problems with the system.

* **“Testing can only show the presence of errors, not their absence.”**

Testing is part of a broader process of software verification and validation (V & V).

* Validation: Are we building the right product?
* Verification: Are we building the product right?

Verification and validation processes are concerned with checking that software being developed meets its specification and delivers the functionality expected by the people paying for the software. These checking processes start as soon as requirements become available and continue through all stages of the development process.

The aim of verification is to check that the software meets its stated functional and non-functional requirements. Validation, however, is a more general process.

The aim of validation is to ensure that the software meets the customer’s expectations. It goes beyond simply checking conformance with the specification to demonstrating that the software does what the customer expects it to do. Validation is essential because requirements specifications do not always reflect the real wishes or needs of system customers and users.

The ultimate goal of verification and validation processes is to establish confidence that the software system is ‘fit for purpose’.

The level of required confidence depends on the system’s purpose, the expectations of the system users, and the current marketing environment for the system:

**1.** **Software purpose -** The more critical the software, the more important that it is reliable.

For example, the level of confidence required for software used to control a safety-critical system is much higher than that required for a prototype that has been developed to demonstrate new product ideas.

**2. User expectations** - Because of their experiences with buggy, unreliable software, many users have low expectations of software quality. They are not surprised when their software fails.

In these situations, you may not need to devote as much time to testing the software. However, as software matures, users expect it to become more reliable so more thorough testing of later versions may be required.

**3. Marketing environment** - When a system is marketed, the sellers of the system must take into account competing products, the price that customers are willing to pay for a system, and the required schedule for delivering that system. In a competitive environment, a software company may decide to release a program before it has been fully tested and debugged because they want to be the first into the market.

As well as software testing, the verification and validation process may involve software inspections and reviews. Inspections and reviews analyze and check the system requirements, design models, the program source code, and even proposed system tests. These are so-called ‘static’ V & V techniques in which you don’t need to execute the software to verify it.

Inspections mostly focus on the source code of a system but any readable representation of the software, such as its requirements or a design model, can be inspected. When you inspect a system, you use knowledge of the system, its application domain, and the programming or modeling language to discover errors.

There are three advantages of software inspection over testing:

1. During testing, errors can mask (hide) other errors. When an error leads to unexpected outputs, you can never be sure if later output anomalies are due to a new error or are side effects of the original error. Because inspection is a static process, you don’t have to be concerned with interactions between errors. Consequently, a single inspection session can discover many errors in a system.

2. Incomplete versions of a system can be inspected without additional costs. If a program is incomplete, then you need to develop specialized test harnesses to test the parts that are available. This obviously adds to the system development costs.

3. As well as searching for program defects, an inspection can also consider broader quality attributes of a program, such as compliance with standards, portability, and maintainability. You can look for inefficiencies, inappropriate algorithms, and poor programming style that could make the system difficult to maintain and update.

It is claimed that more than 90% of defects can be discovered in program inspections. However, inspections cannot replace software testing. Inspections are not good for discovering defects that arise because of unexpected interactions between different parts of a program, timing problems, or problems with system performance. Furthermore, especially in small companies or development groups, it can be difficult and expensive to put together a separate inspection team.

Figure 8.3 is an abstract model of the ‘traditional’ testing process, as used in plan-driven development. Test cases are specifications of the inputs to the test and the expected output from the system (the test results), plus a statement of what is being tested. Test data are the inputs that have been devised to test a system. Test execution can be automated. The expected results are automatically compared with the predicted results so there is no need for a person to look for errors and anomalies in the test run.

Typically, a commercial software system has to go through three stages of testing:

**1.** **Development testing**, where the system is tested during development to discover bugs and defects. System designers and programmers are likely to be involved in the testing process.

**2.** **Release testing,** where a separate testing team tests a complete version of the system before it is released to users. The aim of release testing is to check that the system meets the requirements of system stakeholders.

**3. User testing**, where users or potential users of a system test the system in their own environment. For software products, the ‘user’ may be an internal marketing group who decide if the software can be marketed, released, and sold. **Acceptance testing** is one type of user testing where the customer formally tests a system to decide if it should be accepted from the system supplier or if further development is required.

In practice, the testing process usually involves a mixture of manual and automated testing. In manual testing, a tester runs the program with some test data and compares the results to their expectations. They note and report discrepancies to the program developers. In automated testing, the tests are encoded in a program that is run each time the system under development is to be tested. This is usually faster than manual testing.

**8.1 – DEVELOPMENT TESTING**

----- Development testing includes all testing activities that are carried out by the team developing the system.

During development, testing may be carried out at three levels of granularity:

1. **Unit testing**, where individual program units or object classes are tested. Unit testing should focus on testing the functionality of objects or methods.

2. **Component testing**, where several individual units are integrated to create composite components. Component testing should focus on testing component interfaces.

3. **System testing**, where some or all of the components in a system are integrated and the system is tested as a whole. System testing should focus on testing component interactions.

Development testing is primarily a defect testing process, where the aim of testing is to discover bugs in the software. It is therefore usually interleaved with debugging — the process of locating problems with the code and changing the program to fix these problems.

**8.1.1 – Unit Testing**

Unit testing is the process of testing program components, such as methods or object classes. Individual functions or methods are the simplest type of component. Your tests should be calls to these routines with different input parameters.

When you are testing object classes, you should design your tests to provide coverage of all of the features of the object. This means that you should:

• Test all operations associated with the object.

• Set and check the value of all attributes associated with the object.

• Put the object into all possible states. This means that you should simulate all events that cause a state change.

Figure 8.4 - The weather station object interface (pg. 212)

The interface of this object is shown in Figure 8.4 – see how to make an interface.

--- Unit testing frameworks provide generic test classes that you extend to create specific test cases. They can then run all of the tests that you have implemented and report, often through some GUI, on the success or failure of the tests.

An automated test has 3 parts:

1. A setup part, where you initialize the system with the test case, namely the inputs and expected outputs.

2. A call part, where you call the object or method to be tested.

3. An assertion part where you compare the result of the call with the expected result. If the assertion evaluates to true, the test has been successful; if false, then it has failed.

--- Sometimes the object that you are testing has dependencies on other objects that may not have been written or which slow down the testing process if they are used.

--- In these cases, you may decide to use mock objects. Mock objects are objects with the same interface as the external objects being used that simulate its functionality. Therefore, a mock object simulating a database may have only a few data items that are organized in an array. They can therefore be accessed quickly, without the overheads of calling a database and accessing disks. Similarly, mock objects can be used to simulate abnormal operation or rare events.

**8.1.2 – Choosing Unit Test Cases**

Testing is expensive and time consuming, so it is important that you choose effective unit test cases.

Effectiveness, in this case, means two things:

1. The test cases should show that, when used as expected, the component that you are testing does what it is supposed to do.

2. If there are defects in the component, these should be revealed by test cases. You should therefore write two kinds of test case. The first of these should reflect normal operation of a program and should show that the component works. The other kind of test case should be based on testing experience of where common problems arise. It should use abnormal inputs to check that these are properly processed and do not crash the component.

Two possible strategies here that can be effective in helping you choose test cases. These are:

1. Partition testing, where you identify groups of inputs that have common characteristics and should be processed in the same way. You should choose tests from within each of these groups.

2. Guideline-based testing, where you use testing guidelines to choose test cases. These guidelines reflect previous experience of the kinds of errors that programmers often make when developing components.

The input data and output results of a program often fall into a number of different classes with common characteristics. Examples of these classes are positive numbers, negative numbers, and menu selections. Programs normally behave in a comparable way for all members of a class. That is, if you test a program that does a computation and requires two positive numbers, then you would expect the program to behave in the same way for all positive numbers. Because of this equivalent behavior, these classes are sometimes called equivalence partitions or domains.

Figure 8.5 - Equivalence partitioning (pg. 214)

Figure 8.6 - Equivalence partitions (pg. 215)

Once you have identified a set of partitions, you choose test cases from each of these partitions. A good rule of thumb for test case selection is to choose test cases on the boundaries of the partitions, plus cases close to the midpoint of the partition. The reason for this is that designers and programmers tend to consider typical values of inputs when developing a system.

When you use the specification of a system to identify equivalence partitions, this is called ‘black-box testing’. Here, you don’t need any knowledge of how the system works.

Equivalence partitioning is an effective approach to testing because it helps account for errors that programmers often make when processing inputs at the edges of partitions. You can also use testing guidelines to help choose test cases. Guidelines encapsulate knowledge of what kinds of test cases are effective for discovering errors.

Some of the most general guidelines are:

\_ Choose inputs that force the system to generate all error messages.

\_ Design inputs that cause input buffers to overflow.

\_ Repeat the same input or series of inputs numerous times.

\_ Force invalid outputs to be generated.

\_ Force computation results to be too large or too small.

**8.1.3 – Component Testing**

Software components are often composite components that are made up of several interacting objects. Testing composite components should therefore focus on showing that the component interface behaves according to its specification. You can assume that unit tests on the individual objects within the component have been completed.

There are different types of interface between program components and, consequently, different types of interface error that can occur:

**1. Parameter interfaces -** These are interfaces in which data or sometimes function references are passed from one component to another. Methods in an object have a parameter interface.

**2. Shared memory interfaces** - These are interfaces in which a block of memory is shared between components. Data is placed in the memory by one subsystem and retrieved from there by other sub-systems. This type of interface is often used in embedded systems, where sensors create data that is retrieved and processed by other system components.

**3. Procedural interfaces** - These are interfaces in which one component encapsulates a set of procedures that can be called by other components. Objects and reusable components have this form of interface.

**4. Message passing interfaces** - These are interfaces in which one component requests a service from another component by passing a message to it. A return message includes the results of executing the service. Some object-oriented systems have this form of interface, as do client–server systems.

--- Interface errors are one of the most common forms of error in complex systems. These errors fall into 3 classes:

**1**- **Interface misuse -** A calling component calls some other component and makes an error in the use of its interface. This type of error is common with parameter interfaces, where parameters may be of the wrong type or be passed in the wrong order, or the wrong number of parameters may be passed.

**2**- **Interface misunderstanding** - A calling component misunderstands the specification of the interface of the called component and makes assumptions about its behavior. The called component does not behave as expected which then causes unexpected behavior in the calling component. For example, a binary search method may be called with a parameter that is an unordered array. The search would then fail.

**3- Timing errors** - These occur in real-time systems that use a shared memory or a message-passing interface. The producer of data and the consumer of data may operate at different speeds. Unless particular care is taken in the interface design, the consumer can access out-of-date information because the producer of the information has not updated the shared interface information.

--- Testing for interface defects is difficult because some interface faults may only manifest themselves under unusual conditions.

Some general guidelines for interface testing are:

**1.** Examine the code to be tested and explicitly list each call to an external component.

Design a set of tests in which the values of the parameters to the external components are at the extreme en5.ds of their ranges. These extreme values are most likely to reveal interface inconsistencies.

**2.** Where pointers are passed across an interface, always test the interface with null pointer parameters.

**3.** Where a component is called through a procedural interface, design tests that deliberately cause the component to fail. Differing failure assumptions are one of the most common specification misunderstandings.

**4.** Use stress testing in message passing systems. This means that you should design tests that generate many more messages than are likely to occur in practice. This is an effective way of revealing timing problems

--- Inspections and reviews can sometimes be more cost effective than testing for discovering interface errors. Inspections can concentrate on component interfaces and questions about the assumed interface behavior asked during the inspection process.

**8.1.4 – System Testing**

System testing during development involves integrating components to create a version of the system and then testing the integrated system. System testing checks that components are compatible, interact correctly and transfer the right data at the right time across their interfaces.

-- It obviously overlaps with component testing but there are two important differences:

**1.** During system testing, reusable components that have been separately developed and off-the-shelf systems may be integrated with newly developed components. The complete system is then tested.

**2.** Components developed by different team members or groups may be integrated at this stage. System testing is a collective rather than an individual process. In some companies, system testing may involve a separate testing team with no involvement from designers and programmers.

When you integrate components to create a system, you get emergent behavior. This means that some elements of system functionality only become obvious when you put the components together. This may be planned emergent behavior, which has to be tested.

Sometimes, however, the emergent behavior is unplanned and unwanted. You have to develop tests that check that the system is only doing what it is supposed to do.

Therefore system testing should focus on testing the interactions between the components and objects that make up a system. You may also test reusable components or systems to check that they work as expected when they are integrated with new components. This interaction testing should discover those component bugs that are only revealed when a component is used by other components in the system. Interaction testing also helps find misunderstandings, made by component developers, about other components in the system.

**8.2 – TEST-DRIVEN DEVELOPMENT**

Test-driven development (TDD) is an approach to program development in which you interleave testing and code development.

--- Essentially, you develop the code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test. Test-driven development was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

Figure 8.9 - Test-driven development

The fundamental TDD process is shown in Figure 8.9. The steps in the process are as follows:

1. You start by identifying the increment of functionality that is required. This should normally be small and implementable in a few lines of code.

2. You write a test for this functionality and implement this as an automated test. This means that the test can be executed and will report whether or not it has passed or failed.

3. You then run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail. This is deliberate as it shows that the test adds something to the test set.

4. You then implement the functionality and re-run the test. This may involve refactoring existing code to improve it and add new code to what’s already there.

5. Once all tests run successfully, you move on to implementing the next chunk of functionality.

--- An automated testing environment, such as the JUnit environment that supports Java program testing, is essential for TDD. As the code is developed in very small increments, you have to be able to run every test each time that you add functionality or refactor the program. Therefore, the tests are embedded in a separate program that runs the tests and invokes the system that is being tested.

--- A strong argument for test-driven development is that it helps programmers clarify their ideas of what a code segment is actually supposed to do.

Few benefits of test-driven development are:

**1. Code coverage -** In principle, every code segment that you write should have at least one associated test. Therefore, you can be confident that all of the code in the system has actually been executed. Code is tested as it is written so defects are discovered early in the development process.

**2. Regression testing** - A test suite is developed incrementally as a program is developed.

You can always run regression tests to check that changes to the program have not introduced new bugs.

**3. Simplified debugging -** When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified. You do not need to use debugging tools to locate the problem. Reports of the use of test-driven development suggest that it is hardly ever necessary to use an automated debugger in test-driven development.

**4. System documentation -** The tests themselves act as a form of documentation that describe what the code should be doing. Reading the tests can make it easier to understand the code.

--- One of the most important benefits of test-driven development is that it reduces the costs of regression testing.

Regression testing involves running test sets that have successfully executed after changes have been made to a system. Regression testing is very expensive and often impractical when a system is manually tested, as the costs in time and effort are very high. However, automated testing, which is fundamental to test-first development, dramatically reduces the costs of regression testing. After making a change to a system in test-first development, all existing tests must run successfully before any further functionality is added.

Test-driven development is of most use in new software development where the functionality is either implemented in new code or by using well-tested standard libraries. Test-driven development may also be ineffective with multi-threaded systems. The different threads may be interleaved at different times in different test runs, and so may produce different results.

If you use test-driven development, you still need a system testing process to validate the system; that is, to check that it meets the requirements of all of the system stakeholders. System testing also tests performance, reliability, and checks that the system does not do things that it shouldn’t do.

Test-driven development has proved to be a successful approach for small and medium-sized projects. Programmers are happy and find this approach more productive. It leads to improved code quality, while other disagree. The results have been inconclusive. However, there is no evidence that TDD leads to poorer quality code.

**8.3 – RELEASE TESTING**

Release testing is the process of testing a particular release of a system that is intended for use outside of the development team.

Normally, the system release is for customers and users. It can be for other teams that are developing related systems or for product management who then prepare it for sale.

Two important distinctions between release testing and system testing during the development process:

**1.** A separate team that has not been involved in the system development should be responsible for release testing.

**2.** System testing by the development team should focus on discovering bugs in the system (defect testing). The objective of release testing is to check that the system meets its requirements and is good enough for external use (validation testing).

The primary goal of the release testing process is to convince the supplier of the system that it is good enough for use so that it can be released as a product.

Release testing, therefore, has to show that the system delivers its specified functionality, performance, and dependability, and that it does not fail during normal use. It should take into account all of the system requirements, not just the requirements of the end-users of the system.

Release testing is usually a black-box testing process where tests are derived from the system specification. The system is treated as a black box whose behavior can only be determined by studying its inputs and the related outputs. Another name for this is ‘functional testing’, so-called because the tester is only concerned with functionality and not the implementation of the software.

**8.3.1 – Requirement-based Testing**

It is a good practice that requirements should be testable.

Requirements-based testing is a systematic approach to test case design where you consider each requirement and derive a set of tests for it. Requirements-based testing is validation rather than defect testing—you are trying to demonstrate that the system has properly implemented its requirements.

**(Example on pg. 225)**

Testing a requirement does not mean just writing a single test. You normally have to write several tests to ensure that you have coverage of the requirement. You should also maintain traceability records of your requirements-based testing, which link the tests to the specific requirements that are being tested.

**8.3.2 – Scenario Testing**

Scenario testing is an approach to release testing where you devise typical scenarios of use and use these to develop test cases for the system.

A scenario is a story that describes one way in which the system might be used. Scenarios should be realistic and real system users should be able to relate to them.

--A scenario test should be a narrative story that is credible and fairly complex. It should motivate stakeholders i.e. they believe that it is important that the system passes the test. It should be easy to evaluate. If there are problems with the system, then the release testing team should recognize them.

**An example of scenario for MHC-PMS on pg. 226 is given.**

When you use a scenario-based approach, you are normally testing several requirements within the same scenario. Therefore, as well as checking individual requirements, you are also checking that combinations of requirements do not cause problems.

**8.3.3 – Performance Testing**

Performance tests have to be designed to ensure that the system can process its intended load. This usually involves running a series of tests where you increase the load until the system performance becomes unacceptable.

Performance testing is concerned both with demonstrating that the system meets its requirements and discovering problems and defects in the system.

To test whether performance requirements are being achieved, you may have to construct an operational profile. An operational profile is a set of tests that reflect the actual mix of work that will be handled by the system. Otherwise, you will not get an accurate test of the operational performance of the system.

This approach is not necessarily the best approach for defect testing. In performance testing, this means stressing the system by making demands that are outside the design limits of the software. This is known as ‘stress testing’.

This type of testing has two functions:

**1.** **It tests the failure behavior of the system.**

Circumstances may arise through an unexpected combination of events where the load placed on the system exceeds the maximum anticipated load. In these circumstances, it is important that system failure should not cause data corruption or unexpected loss of user services. Stress testing checks that overloading the system causes it to ‘fail-soft’ rather than collapse under its load.

**2.** **It stresses the system and may cause defects to come to light that would not normally be discovered.** Although it can be argued that these defects are unlikely to cause system failures in normal usage, there may be unusual combinations of normal circumstances that the stress testing replicates. Stress testing is particularly relevant to distributed systems based on a network of processors. These systems often exhibit severe degradation when they are heavily loaded. The network becomes swamped with coordination data that the different processes must exchange. The processes become slower and slower as they wait for the required data from other processes. Stress testing helps you discover when the degradation begins so that you can add checks to the system to reject transactions beyond this point.

**8.4 – USER TESTING**

User or customer testing is a stage in the testing process in which users or customers provide input and advice on system testing. This may be formally testing a system that has been commissioned from an external supplier, or could be an informal process where users experiment with a new software product to see if they like it and that it does what they need.

User testing is essential, even when comprehensive system and release testing have been carried out. The reason for this is that influences from the user’s working environment have a major effect on the reliability, performance, usability, and robustness of a system. It is practically impossible for a system developer to replicate the system’s working environment, as tests in the developer’s environment are inevitably artificial.

In practice, there are three different types of user testing:

**1. Alpha testing**, where users of the software work with the development team to test the software at the developer’s site.

**2. Beta testing**, where a release of the software is made available to users to allow them to experiment and to raise problems that they discover with the system developers.

**3. Acceptance testing**, where customers test a system to decide whether or not it is ready to be accepted from the system developers and deployed in the customer environment.

In **alpha testing**, users and developers work together to test a system as it is being developed. This means that the users can identify problems and issues that are not readily apparent to the development testing team.

Alpha testing is often used when developing software products that are sold as shrink-wrapped systems. Users of these products may be willing to get involved in the alpha testing process because this gives them early information about new system features that they can exploit. It also reduces the risk that unanticipated changes to the software will have disruptive effects on their business. However, alpha testing may also be used when custom software is being developed e.g. agile methods, such as XP.

**Beta testing** takes place when an early, sometimes unfinished, release of a software system is made available to customers and users for evaluation. Beta testers may be a selected group of customers who are early adopters of the system.

Alternatively, the software may be made publicly available for use by anyone who is interested in it. Beta testing is essential to discover interaction problems between the software and features of the environment where it is used. Beta testing is also a form of marketing— customers learn about their system and what it can do for them.

**Acceptance testing** is an inherent part of custom systems development. It takes place after release testing. It involves a customer formally testing a system to decide whether or not it should be accepted from the system developer. Acceptance implies that payment should be made for the system.

6 stages in acceptance testing process, are shown in Figure 8.11 – Acceptance testing process (pg. 229)

**1. Define acceptance criteria** - This stage should, ideally, take place early in the process before the contract for the system is signed. The acceptance criteria should be part of the system contract and be agreed between the customer and the developer. In practice, however, it can be difficult to define criteria so early in the process. Detailed requirements may not be available and there may be significant requirements change during the development process.

**2. Plan acceptance testing** - This involves deciding on the resources, time, and budget for acceptance testing and establishing a testing schedule. The acceptance test plan should also discuss the required coverage of the requirements and the order in which system features are tested. It should define risks to the testing process, such as system crashes and inadequate performance, and discuss how these risks can be mitigated.

**3. Derive acceptance tests** - Once acceptance criteria have been established, tests have to be designed to check whether or not a system is acceptable. Acceptance tests should aim to test both the functional and non-functional characteristics (e.g., performance) of the system. They should, ideally, provide complete coverage of the system requirements.

**4. Run acceptance tests** - The agreed acceptance tests are executed on the system. Ideally, this should take place in the actual environment where the system will be used, but this may be disruptive and impractical. Therefore, a user testing environment may have to be set up to run these tests. It is difficult to automate this process as part of the acceptance tests may involve testing the interactions between end-users and the system. Some training of end-users may be required.

**5. Negotiate test results** - It is very unlikely that all of the defined acceptance tests will pass and that there will be no problems with the system. If this is the case, then acceptance testing is complete and the system can be handed over. More commonly, some problems will be discovered. In such cases, the developer and the customer have to negotiate to decide if the system is good enough to be put into use.

**6. Reject/accept system** - This stage involves a meeting between the developers and the customer to decide on whether or not the system should be accepted. If the system is not good enough for use, then further development is required to fix the identified problems. Once complete, the acceptance testing phase is repeated.

**(Extra about testing in agile - XP)**

--- In agile methods, such as XP, acceptance testing has a rather different meaning. In principle, it shares the notion that users should decide whether or not the system is acceptable. However, in XP, the user is part of the development team (i.e., he or she is an alpha tester) and provides the system requirements in terms of user stories. He or she is also responsible for defining the tests, which decide whether or not the developed software supports the user story. The tests are automated and development does not proceed until the story acceptance tests have passed. There is, therefore, no separate acceptance testing activity.

One problem with user involvement is ensuring that the user who is embedded in the development team is a ‘typical’ user with general knowledge of how the system will be used. It can be difficult to find such a user, and so the acceptance tests may actually not be a true reflection of practice.

Furthermore, the requirement for automated testing severely limits the flexibility of testing interactive systems. For such systems, acceptance testing may require groups of end-users to use the system as if it was part of their everyday work.

--- If a system fails acceptance test, then it is not necessary that the client will reject the system.

They may be willing to accept the software, irrespective of problems, because the costs of not using the software are greater than the costs of working around the problems. Therefore, the outcome of negotiations may be conditional acceptance of the system. The customer may accept the system so that deployment can begin. The system provider agrees to repair urgent problems and deliver a new version to the customer as quickly as possible.

**CHAPTER 11 – UI DESIGN (Pressman)**

**11.1 – THE GOLDEN RULES**

Theo Mandel coins three *golden rules that* form the basis for a set of user interface design principles are:

**1.** Place the user in control.

**2.** Reduce the user’s memory load.

**3.** Make the interface consistent.

**11.1.1 - Place the user in control.**

A number of design principles that allow the user to maintain control:

1. **Define interaction modes in a way that does not force a user into unnecessary or undesired actions**

An interaction mode is the current state of the interface.For example, if spell checkis selected in a word-processor menu, the software moves to a spell-checking mode. There is no reason to force the user to remain in spell-checking mode. The user should be able to enter and exit the mode with little or no effort.

1. **Provide for flexible interaction**

Because different users have different interaction preferences, choices should be provided. For example, software might allow a user to interact via keyboard commands, mouse movement, a digitizer pen, a multi-touch screen, or voice recognition commands. But every action is not amenable to every interaction mechanism.

1. **Allow user interaction to be interruptible and undoable**

Even when involved in a sequence of actions, the user should be able to interrupt the sequence to do something else without losing the work that had been done. The user should also be able to “undo” any action.

1. **Streamline interaction as skill levels advance and allow the interaction to be customized**

Users often find that they perform the same sequence of interactions repeatedly. It is worthwhile to design a “macro” mechanism that enables an advanced user to customize the interface to facilitate interaction.

1. **Hide technical internals from the casual user**

The user interface should move the user into the virtual world of the application. The user should not be aware of the operating system, file management functions, or other arcane computing technology. In essence, the interface should never require that the user interact at a level that is “inside” the machine.

1. **Design for direct interaction with objects that appear on the screen**

The user feels a sense of control when able to manipulate the objects that are necessary to perform a task in a manner similar to what would occur if the object were a physical thing.

**11.1.2 - Reduce the user’s memory load**

The more a user has to remember, the more error-prone the interaction with the system will be. It is for this reason that a well-designed user interface does not tax the user’s memory. Whenever possible, the system should remember information and assist the user with an interaction scenario that assists recall. Mandel defines design principles that enable an interface to reduce the user’s memory load:

1. **Reduce demand on short-term memory**

When users are involved in complex tasks, the demand on short-term memory can be significant. The interface should be designed to reduce the requirement to remember past actions, inputs, and results. This can be accomplished by providing visual cues that enable a user to recognize past actions, rather than having to recall them.

1. **Establish meaningful defaults**

The initial set of defaults should make sense for the average user, but a user should be able to specify individual preferences. However, a “reset” option should be available, enabling the redefinition of original default values.

1. **Define shortcuts that are intuitive**

When mnemonics are used to accomplish a system function (e.g., alt-P to invoke the print function), the mnemonic should be tied to the action in a way that is easy to remember.

1. **The visual layout of the interface should be based on a real-world metaphor**

For example, a bill payment system should use a checkbook and check register metaphor to guide the user through the bill paying process. This enables the user to rely on well-understood visual cues, rather than memorizing an arcane interaction sequence.

1. **Disclose information in a progressive fashion**

The interface should be organized hierarchically. That is, information about a task, or some behavior should be presented first at a high level of abstraction. More detail should be presented after the user indicates interest with a mouse pick.

An example, common to many word-processing applications, is the underlining function. Every underlining capability is not listed. The user must pick underlining; then all underlining options (e.g., single underline, double underline, dash underline) are presented.

**11.1.3 - Make the interface consistent**

The interface should present and acquire information in a consistent fashion. This implies that:

(1) All visual information is organized according to design rules that are maintained throughout all screen displays

(2) Input mechanisms are constrained to a limited set that is used consistently throughout the application

(3) Mechanisms for navigating from task to task are consistently defined and implemented.

Mandel defines a set of design principles that help make the interface consistent:

1. **Allow the user to put the current task into a meaningful context**

Many interfaces implement complex layers of interactions with dozens of screen images. It is important to provide indicators (e.g., window titles, graphical icons, consistent color coding) that enable the user to know the context of the work at hand and should be able to determine where he has come from.

1. **Maintain consistency across a family of applications**

A set of applications (or products) should all implement the same design rules so that consistency is maintained for all interaction.

1. **If past interactive models have created user expectations, do not make changes unless there is a compelling reason to do so**

Once a particular interactive sequence has become a de facto standard (e.g., the use of alt-S to save a file), the user expects this in every application he encounters. A change will cause confusion.

**11.2 – USER INTERFACE ANALYSIS AND DESIGN**

The overall process for analyzing and designing a user interface begins with the creation of different models of system function. You begin by delineating the human and computer-oriented tasks that are required to achieve system function and then considering the design issues that apply to all interface designs.

Tools are used to prototype and ultimately implement the design model, and the result is evaluated by end users for quality.

**11.2.1 – Interface Analysis and Design Model**

Four different models come into play when a user interface is to be analyzed and designed.

A software engineer establishes a **user model**, the software engineer creates a **design model**, and the end user develops a mental image that is often called the **user’s mental model** or the system perception, and the implementers of the system create an **implementation model**. These models may differ significantly, so an interface designer reconcile these differences and derive a consistent representation of the interface.

The user model establishes the profile of end users of the system.

To build an effective user interface, all design should begin with an understanding of the intended users, including profiles of their age, gender, education and etc. The users can be categorized as:

**Novices -** No syntactic knowledge of the system and little semantic knowledge of the application or computer usage in general.

**Knowledgeable, intermittent users -** Reasonable semantic knowledge of the application but relatively low recall of syntactic information necessary to use the interface.

**Knowledgeable, frequent users -** Good semantic and syntactic knowledge that often leads to the power-user syndrome (individuals who look for shortcuts and abbreviated modes of interaction).

The **user’s mental model** (system perception) is the image of the system that end users carry in their heads.

The **implementation model** combines the outward manifestation of the computer-based system (the look and feel of the interface), coupled with all supporting information (books, manuals, videotapes) that describes interface syntax and semantics. When the implementation model and the user’s mental model are coincident, users generally feel comfortable with the software and use it effectively. So, the design model must have been developed to accommodate the information contained in the user model, and the implementation model must accurately reflect syntactic and semantic information about the interface.

The models described in this section are ‘abstractions of what the user is doing or thinks he is doing when he uses an interactive system.’

**11.2.2 – The process**

The analysis and design process for user interfaces is iterative & can be represented using a spiral model.

Referring to Figure 11.1, the user interface analysis and design process begins at the interior of the spiral and encompasses four distinct framework activities:

(1) Interface analysis and modeling

(2) Interface design

(3) Interface construction

(4) Interface validation.

The spiral shown in Figure 11.1 implies that each of these tasks will occur more than once, with each pass around the spiral representing additional elaboration of requirements and the resultant design.

**Interface analysis** focuses on the profile of the users who will interact with the system. Skill level, business understanding, and general receptiveness to the new system are recorded; and different user categories are defined. For each user category, requirements are elicited. You work to understand the system perception for each class of users.

After general requirements, a more detailed **task analysis** is conducted. Those tasks that the user performs to accomplish the goals of the system are identified, described, and elaborated. Finally, **analysis of the user environment** focuses on the physical work environment. Few questions asked are:

• Where will the interface be located physically?

• Will the user be sitting, standing, or performing other tasks unrelated to the interface?

The information gathered as part of the analysis action is used to create an analysis model for the interface. Using this model as a basis, the design action commences.

The **goal of interface design** is to define a set of interface objects and actions that enable a user to perform all defined tasks in a manner that meets every usability goal defined for the system.

**Interface construction** normally begins with the creation of a prototype that enables usage scenarios to be evaluated. As the iterative design process continues, a user interface tool kit may be used to complete the construction of the interface.

**Interface validation** focuses on:

(1) The ability of the interface to implement every user task correctly, to accommodate all task variations, and to achieve all general user requirements.

(2) The degree to which the interface is easy to use and easy to learn.

(3) The users’ acceptance of the interface as a useful tool in their work.

**11.3 – INTERFACE ANALYSIS**

A key tenet of all software engineering process models is tounderstand the problem before you attempt to design a solution.

Understanding the problem means:

(1) The end users who will interact with the system through the interface

(2) The tasks that end users must perform to do their work.

(3) The content that is presented as part of the interface.

(4) The environment in which these tasks will be conducted.

**11.3.1 User Analysis**

To get the mental image and the design model to converge is to work to understand the users themselves and how they will use the system. This gives a clearer idea of how the user interface must be characterized to meet the users’ needs. Information from a broad array of sources can be used to accomplish this:

**User Interviews -** The most direct approach, members of the software team meet with end users to better understand their needs, motivations, work culture, and a myriad of other issues. Either one-on-one meetings or through focus groups.

**Sales input -** Sales people meet with users on a regular basis and can gather information that will help the software team to categorize users and better understand their requirements.

**Marketing input -** Market analysis can be invaluable in the definition of market segments and an understanding of how each segment might use the software in subtly different ways.

**Support input -** Support staff talks with users on a daily basis. They are the most likely source of information on what works or doesn’t, what users like or dislike, what features generate questions and what features are easy to use.

**11.3.2 – Task Analysis and Modeling**

The goal of task analysis is to answer the following questions:

• What work will the user perform in specific circumstances?

• What tasks and subtasks will be performed as the user does the work?

These techniques are applied to the user interface.

**Use cases -** When used as part of task analysis, the use case is developed to show how an end user performs some specific work-related task. In most instances, the use case is written in an informal style (a simple paragraph) in the first-person. From it, you can extract tasks, objects, and the overall flow of the interaction.

**Task elaboration -** Stepwise elaboration (also called functional decomposition or stepwise refinement) is a mechanism for refining the processing tasks that are required for software to accomplish some desired function. Task analysis for interface design uses an elaborative approach to assist in understanding the human activities the user interface must accommodate.

Task analysis can be applied in two ways. To understand the tasks that must be performed to accomplish the goal of the activity, you must understand the tasks that people currently perform and then map these into a similar set of tasks that are implemented in the context of the user interface. Alternatively, you can study an existing specification for a computer-based solution and derive a set of user tasks that will accommodate the user model, the design model, and the system perception.

Regardless of the overall approach to task analysis, you must first define and classify tasks

**Object elaboration -** Rather than focusing on the tasks that a user must perform, you can examine the use case and other information obtained from the user and extract the physical objects that are used by the interior designer. These objects can be categorized into classes. Attributes of each class are defined, and an evaluation of the actions applied to each object provide a list of operations.

**Workflow analysis -** When different users of various roles makes use of a user interface, it is necessary to apply workflow analysis. This technique allows you to understand how a work process is completed when several people and roles are involved.

**Hierarchical representation -** A process of elaboration occurs as you begin to analyze the interface. Once workflow has been established, a task hierarchy can be defined for each user type. The hierarchy is derived by a stepwise elaboration of each task identified for the user.

**11.3.3 Analysis of Display Content**

The user tasks identified lead to variety of different types of content. For modern applications, display content can range from character-based reports (e.g., a spreadsheet), graphical displays (e.g., a histogram, a 3-D model, a picture of a person), or specialized information (e.g., audio or video files). The analysis modeling techniques identify the output data objects that are produced by an application. These data objects may be:

(1) Generated by components (unrelated to the interface) in other parts of an application.

(2) Acquired from data stored in a database that is accessible from the application.

(3) Transmitted from systems external to the application in question.

**11.3.4 Analysis of the Work Environment**

People do not perform their work in isolation. They are influenced by the activity around them, the physical characteristics of the workplace and the work relationships they have with other people.

In some applications the user interface for a computer-based system is placed in a user-friendly location, but in others lighting may be suboptimal, noise may be a factor, a keyboard or mouse may not be an option, display placement may be less than ideal. The interface designer may be constrained by factors that mitigate against ease of use.

In addition to physical environmental factors, the workplace culture also comes into play. These and many related questions should be considered before the interface design commences.

**11.4 – INTERFACE DESIGN STEPS**

Once interface analysis has been completed, all tasks required by the end user have been identified in detail and the interface design activity commences. Interface design is an iterative process. Each user interface design step occurs a number of times, elaborating and refining information developed in the preceding step.

Different user interface design models suggest some combination of the following steps:

**1.** Using information developed during interface analysis define interface objects and actions (operations).

**2.** Define events (user actions) that will cause the state of the user interface to change. Model this behavior.

**3.** Depict each interface state as it will actually look to the end user.

**4.** Indicate how the user interprets the state of the system from information provided through the interface.

Regardless of the sequence of design tasks, you should always follow the golden rules, model how the interface will be implemented, and consider the environment (e.g., display technology, OS, development tools) that will be used.

**11.4.1 - Applying Interface Design Steps**

The definition of interface objects and the actions that are applied to them is an important step in interface design. The user scenarios are parsed; Nouns (objects) and verbs (actions) are isolated to create a list of objects and actions.

Once the objects and actions have been defined and elaborated iteratively, they are categorized by type. Target, source, and application objects are identified.

A **source object** (e.g., a report icon) is dragged and dropped onto a **target object** (e.g., a printer icon). The implication of this action is to create a hard-copy report.

An **application object** represents application-specific data that are not directly manipulated as part of screen interaction.

When you are satisfied that all important objects and actions have been defined, screen layout is performed. Like other interface design activities, screen layout is an interactive process in which graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and definition of major and minor menu items are conducted.

Objects and actions are extracted from this list of homeowner tasks. The majority of objects noted are application objects.

**11.4.2 - User Interface Design Patterns**

Graphical user interfaces have become so common that a wide variety of user interface design patterns has emerged. As I noted earlier in this book, a design pattern is an abstraction that prescribes a design solution to a specific, well-bounded design problem.

A vast array of interface design patterns has been proposed over the past decade.

**11.4.3 - Design Issues**

As the design of a user interface evolves, four common design issues almost always surface: system response time, user help facilities, error information handling, and command labeling. Unfortunately, many designers do not address these issues until relatively late in the design process. Unnecessary iteration, project delays, and end-user frustration often result. It is far better to establish each as a design issue to be considered at the beginning of software design, when changes are easy and costs are low.

**Response time -** System response time is the primary complaint for many interactive applications. In general, system response time is measured from the point at which the user performs some control action until the software responds with desired output or action. System response time has two important characteristics: length and variability.

System response shouldn’t be too long. **Variability** refers to the deviation from average response time. Low variability enables the user to establish an interaction rhythm, even if response time is relatively long.

**Help facilities -** Almost every user of an interactive, computer-based system requires help now and then. Manuals and modern software provides online help facilities that enable a user to get a question answered or resolve a problem without leaving the interface.

**Error handling -** Error messages and warnings are bad news delivered to users of interactive systems when something has gone awry. At their worst, error messages and warnings impart useless or misleading information and serve only to increase user frustration. An explanation for error must exist. The error message provides no real indication of what went wrong or where to look to get additional information.

In general, every error message or warning produced by an interactive system should have the following characteristics:

• The message should describe the problem in jargon that the user can understand.

• The message should provide constructive advice for recovering from the error.

• The message should indicate any negative consequences of the error (e.g., potentially corrupted data files) so that the user can check to ensure that they have not occurred (or correct them if they have).

• The message should be accompanied by an audible or visual cue. That is, a beep might be generated to accompany the display of the message, or the message might flash momentarily or be displayed in a color that is easily recognizable as the “error color.”

• The message should be “nonjudgmental.” That is, the wording should never place blame on the user.

**Menu and command labeling -** The typed command was once the most common mode of interaction between user and system software and was commonly used for applications of every type. Today, the use of window-oriented, point-and pick interfaces has reduced reliance on typed commands, but some power-users continue to prefer a command-oriented mode of interaction.

**Application accessibility -** Software engineers must ensure that interface design encompasses mechanisms that enable easy access for those with special needs. *Accessibility* for users who may be physically challenged is an imperative for ethical, legal, and business reasons.

**Internationalization -** Too often, interfaces are designed for one locale and language and then jury-rigged to work in other countries. The challenge for interface designers is to create “globalized” software. That is, user interfaces should be designed to accommodate a generic core of functionality that can be delivered to all who use the software. **Localization** features enable the interface to be customized for a specific market.

**11.5 - WEBAPP INTERFACE DESIGN**

Every user interface should exhibit the usability characteristics that were discussed earlier. A Web-App interface should answers three primary questions for the end user:

1. **Where am I?** - The interface should provide an indication of the Web-App that has been accessed and inform the user of her location in the content hierarchy.
2. **What can I do now? -** The interface should always help the user understand his current options—what functions are available, what links are live and what content is relevant.
3. **Where have I been, where am I going? -** The interface must facilitate navigation. Hence, it must provide a map of where the user has been and what paths may be taken to move elsewhere within the Web-App.

**11.5.1 - Interface Design Principles and Guidelines**

The user interface of a Web-App is its first impression.

A set of fundamental characteristics that all interfaces should exhibit and should be followed by every Web-App interface designer:

Effective interfaces are visually apparent and forgiving, instilling in their users a sense of control. Users quickly see the breadth of their options, grasp how to achieve their goals, and do their work.

Work is carefully and continuously saved, with full option for the user to undo any activity at any time.

Effective applications and services perform a maximum of work, while requiring a minimum of information from users.

In order to design Web-App interfaces that exhibit these characteristics, **a set of overriding design principles:**

**Anticipation -** A Web-App should be designed so that it anticipates the user’s next move.

For example, consider a customer support Web-App developed by a manufacturer of computer printers. A user has requested a content object that presents information about a printer driver for a newly released operating system. The designer of the Web-App should anticipate that the user might request a download of the driver and should provide navigation facilities that allow this to happen without requiring the user to search for this capability.

**Communication -** The interface should communicate the status of any activity initiated by the user. Communication can be obvious (e.g., a text message) or subtle (e.g., an image of a sheet of paper moving through a printer to indicate that printing is under way

**Consistency -** The use of navigation controls, menus, icons, and aesthetics (e.g., color, shape, and layout) should be consistent throughout the Web-App. For example, if underlined blue text implies a navigation link, content should never incorporate blue underlined text that does not imply a link. Moreover, every feature of the interface should respond in a manner that is consistent with user expectations.

**Controlled autonomy -** The interface should facilitate user movement throughout the Web-App, but it should do so in a manner that enforces navigation conventions that have been established for the application.

**Efficiency -** The design of the Web-App and its interface should optimize the user’s work efficiency, not the efficiency of the developer who designs and builds it or the client-server environment that executes it.

**Flexibility -** The interface should be flexible enough to enable some users to accomplish tasks directly and others to explore the Web-App in a somewhat random fashion. In every case, it should enable the user to understand where he is and provide the user with functionality that can undo mistakes and retrace poorly chosen navigation paths.

**Focus -** The Web-App interface (and the content it presents) should stay focused on the user task(s) at hand. In all hypermedia there is a tendency to route the user to loosely related content. Why? Because it’s very easy to do! The problem is that the user can rapidly become lost in many layers of supporting information and lose sight of the original content that she wanted in the first place.

**Fitt’s law -** “The time to acquire a target is a function of the distance to and size of the target.” Fitt’s law is an effective method of modeling rapid, aimed movements, where one appendage (like a hand) starts at rest at a specific start position, and moves to rest within a target area.

If a sequence of selections or standardized inputs (with many different options within the sequence) is defined by a user task, the first selection (e.g., mouse pick) should be physically close to the next selection.

**Human interface objects -** A vast library of reusable human interface objects has been developed for Web-Apps. Use them. Any interface object that can be ‘seen, heard, touched or otherwise perceived’ by an end user can be acquired from any one of a number of object libraries.

**Latency reduction -** Rather than making the user wait for some internal operation to complete (e.g., downloading a complex graphical image), the Web-App should use multitasking in a way that lets the user proceed with work as if the operation has been completed.

In addition to reducing latency, delays must be acknowledged so that the user understands what is happening.

**Learnability.** A Web-App interface should be designed to minimize learning time, and once learned, to minimize relearning required when the Web-App is revisited. In general the interface should emphasize a simple, intuitive design that organizes content and functionality into categories that are obvious to the user.

**Metaphors -** An interface that uses an interaction metaphor is easier to learn and easier to use, as long as the metaphor is appropriate for the application and the user. A metaphor should call on images and concepts from the user’s experience, but it does not need to be an exact reproduction of a real-world experience

**Maintain work product integrity -** A work product (e.g., a form completed by the user, a user-specified list) must be automatically saved so that it will not be lost if an error occurs.

A Web-App should be designed to auto save all user-specified data. The interface should support this function and provide the user with an easy mechanism for recovering lost information.

**Readability -** All information presented through the interface should be readable by young and old. The interface designer should emphasize readable type styles, font sizes, and color background choices that enhance contrast.

**Track state -** When appropriate, the state of the user interaction should be tracked and stored so that a user can logoff and return later to pick up where she left off.In general, cookies can be designed to store state information.

**Visible navigation -** A well-designed Web-App interface provides ‘the illusion that users are in the same place, with the work brought to them.’ When this approach is used, navigation is not a user concern. Rather, the user retrieves content objects and selects functions that are displayed and executed through the interface.

A well-designed interface improves the user’s perception of the content or services provided by the site. It need not necessarily be flashy, but it should always be well structured and ergonomically sound.

**11.5.2 Interface Design Workflow for Web-Apps**

Information contained within the requirements model forms the basis for the creation of a screen layout that depicts graphical design and placement of icons, definition of descriptive screen text, specification and titling for windows, and specification of major and minor menu items.

Tools are then used to prototype and ultimately implement the interface design model. The following tasks represent a Rudimentary workflow for Web-App interface design:

**1. Review information contained in the requirements model and refine as required.**

**2. Develop a rough sketch of the Web-App interface layout -** An interface prototype may have been developed as part of the requirements modeling activity. If the layout already exists, it should be reviewed and refined as required. If the interface layout has not been developed, you should work with stakeholders to develop it at this time.

**3. Map user objectives into specific interface actions -** For the vast majority of Web-Apps, the user will have a relatively small set of primary objectives. These should be mapped into specific interface actions.

**4. Define a set of user tasks that are associated with each action -** Each interface action (e.g., “buy a product”) is associated with a set of user tasks. These tasks have been identified during requirements modeling. During design, they must be mapped into specific interactions that encompass navigation issues, content objects, and Web-App functions.

**5. Storyboard screen images for each interface action -** As each action is considered, a sequence of storyboard images (screen images) should be created to depict how the interface responds to user interaction. Content objects should be identified, Web-App functionality should be shown, and navigation links should be indicated.

**6. Refine interface layout and storyboards using input from aesthetic design -** In most cases, you’ll be responsible for rough layout and storyboarding, but the aesthetic look and feel for a major commercial site is often developed by artistic, rather than technical, professionals. Aesthetic design is integrated with the work performed by the interface designer.

**7. Identify user interface objects that are required to implement the interface -** This task may require a search through an existing object library to find those reusable objects (classes) that are appropriate for the Web-App interface.

**8. Develop a procedural representation of the user’s interaction with the interface -** This optional task uses UML sequence diagrams and/or activity diagrams to depict the flow of activities and decisions that occur as the user interacts with the Web-App.

**9. Develop a behavioral representation of the interface -**This optional task makes use of UML state diagrams to represent state transitions and the events that cause them. Control mechanisms are defined.

**10. Describe the interface layout for each state -** Using design information developed in Tasks 2 and 5, associate a specific layout or screen image with each Web-App state.

**11. Refine and review the interface design model -** Review of the interface should focus on usability.

It is important to note that the final task set you choose should be adapted to the special requirements of the application that is to be built.

**11.6 - DESIGN EVALUATION**

Once you create an operational user interface prototype, it must be evaluated to determine whether it meets the needs of the user.

After the design model has been completed, a first-level prototype is created. The prototype is evaluated by the user, who provides you with direct comments about the efficacy of the interface. In addition, if formal evaluation techniques are used e.g., questionnaires, you can extract information from these data. Design modifications are made based on user input, and the next level prototype is created. The evaluation cycle continues until no further modifications to the interface design are necessary.

The prototyping approach is effective, but if you identify and correct potential problems early, the number of loops through the evaluation cycle will be reduced and development time will shorten. If a design model of the interface has been created, a number of evaluation criteria can be applied during early design reviews:

**1.** The length and complexity of the requirements model or written specification of the system and its interface provide an indication of the amount of learning required by users of the system.

**2.** The number of user tasks specified and the average number of actions per task provide an indication of interaction time and the overall efficiency of the system.

**3.** The number of actions, tasks, and system states indicated by the design model imply the memory load on users of the system.

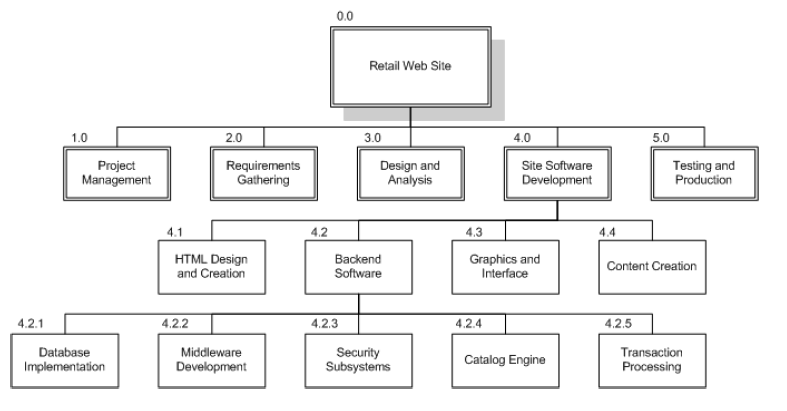
**4.** Interface style, help facilities, and error handling protocol provide a general indication of the complexity of the interface and the degree to which it will be accepted by the user.

Once the first prototype is built, you can collect a variety of qualitative and quantitative data that will assist in evaluating the interface. If quantitative data are desired, a form of time-study analysis can be conducted. Users are observed during interaction, and data are collected and used as a guide for interface modification.

**Work breakdown Structure**

Dividing complex projects to simpler and manageable tasks is the process identified as Work Breakdown Structure (WBS).

The WBS is a deliverable-oriented hierarchy of decomposed project components that organises and defines the total scope of the project. The WBS is a representation of the detailed project scope statement that specifies the work to be accomplished by the project.



**Pitfalls**

* Level of Work Package Detail
* Deliverables Not Activities or Tasks
* WBS is not a Plan or Schedule
* WBS Updates Require Change Control
* WBS is not an Organizational Hierarchy

**Example**

* Redecorate Room
  + Prepare materials
    - Buy paint
    - Buy a ladder
    - Buy brushes/rollers
    - Buy wallpaper remover
  + Prepare room
    - Remove old wallpaper
    - Remove detachable decorations
    - Cover floor with old newspapers
    - Cover electrical outlets/switches with tape
    - Cover furniture with sheets
  + Paint the room
  + Clean up the room
    - Dispose or store left over paint
    - Clean brushes/rollers
    - Dispose of old newspapers
    - Remove covers

**Directly from slides:**

* Gantt PERT (network diagram)
* Wideband Delphi – Estimation
* LOC
* FP

**CHAPTER 28 - RISK MANAGEMENT (PRESSMAN)**

**28.1 - REACTIVE VERSUS PROACTIVE RISK STRATEGIES**

A **reactive strategy** monitors the project for likely risks. Resources are set aside to deal with them, should they become actual problems. More commonly, the software team does nothing about risks until something goes wrong. Then, the team flies into action in an attempt to correct the problem rapidly. This is often called a fire-fighting mode.

An intelligent strategy for risk management is to be proactive. A **proactive strategy** begins long before technical work is initiated. Potential risks are identified, their probability and impact are assessed, and they are ranked by importance. Then, the software team establishes a plan for managing risk. The primary objective is to avoid risk, but because not all risks can be avoided, the team works to develop a contingency plan that will enable it to respond in a controlled and effective manner.

**28.2 - SOFTWARE RISK**

Risk always involves two characteristics which should be analyzed:

**Uncertainty** - The risk may or may not happen (no 100% probable).

**Loss** - If the risk becomes a reality, unwanted consequences or losses will occur.

To accomplish this, different categories of risks are considered.

**Project risks** threaten the project plan. That is, if project risks become real, it is likely that the project schedule will slip and that costs will increase. Project risks identify potential budgetary, schedule, personnel (staffing and organization), resource, stakeholder, and requirements problems and their impact on a software project.

Project complexity, size, and the degree of structural uncertainty can also be included in PR.

**Technical risks** threaten the quality and timeliness of the software to be produced.

If a technical risk becomes a reality, implementation may become difficult or impossible.

Technical risks identify potential design, implementation, interface, verification, and maintenance problems. In addition, specification ambiguity, technical uncertainty, technical obsolescence, and ‘leading-edge’ technology are also risk factors.

**Business risks** threaten the viability of the software to be built and often jeopardize the project or the product.

Candidates for the top five business risks are: (1) building an excellent product or system that no one really wants (market risk), (2) building a product that no longer fits into the overall business strategy for the company (strategic risk), (3) building a product that the sales force doesn’t understand how to sell (sales risk), (4) losing the support of senior management due to a change in focus or a change in people (management risk), and (5) losing budgetary or personnel commitment (budget risks).

A more general categorization of risks is:

**Known risks** are those that can be uncovered after careful evaluation of the project plan, the business and technical environment in which the project is being developed, and other reliable information sources (e.g., unrealistic delivery date, lack of documented requirements or software scope, poor development environment).

**Predictable risks** are extrapolated from past project experience (e.g., staff turnover, poor communication with the customer, dilution of staff effort as ongoing maintenance requests are serviced).

**Unpredictable risks** are the joker in the deck. They can and do occur, but they are extremely difficult to identify in advance.

**28.3 – RISK IDENTIFICATION (What can go wrong?)**

Risk identification is a systematic attempt to specify threats to the project plan (estimates, schedule, resource loading, etc.). By identifying known and predictable risks, the project manager takes a first step toward avoiding them when possible and controlling them when necessary.

There are two distinct types of risks for each of the categories:

* Generic risks
* Product-specific risks.

**Generic risks** are a potential threat to every software project.

**Product-specific** risks can be identified only by those with a clear understanding of the technology, the people, and the environment that is specific to the software that is to be built.

The following checklist can be used for risk identification. It focuses on some subset of known and predictable risks in the following generic subcategories:

• **Product size** - risks associated with the overall size of the software to be built or modified.

• **Business impact** - risks associated with constraints imposed by management or the marketplace.

• **Stakeholder characteristics** - risks associated with the sophistication of the stakeholders and the developer’s ability to communicate with stakeholders in a timely manner.

• **Process definition** - risks associated with the degree to which the software process has been defined and is followed by the development organization.

• **Development environment -** risks associated with the availability and quality of the tools to be used to build the product.

• **Technology to be built** - risks associated with the complexity of the system to be built and the “newness” of the technology that is packaged by the system.

• **Staff size and experience** - risks associated with the overall technical and project experience of the software engineers who will do the work.

The answers to these questions allow you to estimate the impact of risk. A different risk item checklist format simply lists characteristics that are relevant to each generic subcategory. Finally, a set of ‘risk components and drivers’ are listed along with their probability of occurrence.

**28.3.1 - Assessing Overall Project Risk**

The following questions have been derived from risk data obtained by surveying experienced software project managers.

**1.** Have top software and customer managers formally committed to support the project?

**2.** Are end users enthusiastically committed to the project and the system/ product to be built?

**3.** Are requirements fully understood by the software engineering team and its customers?

**4.** Have customers been involved fully in the definition of requirements?

**5.** Do end users have realistic expectations?

**6.** Is the project scope stable?

**7.** Does the software engineering team have the right mix of skills?

**8.** Are project requirements stable?

**9.** Does the project team have experience with the technology to be implemented?

**10.** Is the number of people on the project team adequate to do the job?

**11.** Do all customer/user constituencies agree on the importance of the project and on the requirements for the system/product to be built?

If any one of these questions is answered negatively, mitigation, monitoring, and management steps should be instituted without fail. The degree to which the project is at risk is directly proportional to the number of negative responses to these questions.

**28.3.2 - Risk Components and Drivers**

The risk components are defined in the following manner:

• **Performance risk**—the degree of uncertainty that the product will meet its requirements and be fit for its intended use.

• **Cost risk**—the degree of uncertainty that the project budget will be maintained.

• **Support risk**—the degree of uncertainty that the resultant software will be easy to correct, adapt, and enhance.

• **Schedule risk**—the degree of uncertainty that the project schedule will be maintained and that the product will be delivered on time.

The impact of each risk driver on the risk component is divided into one of four impact categories—negligible, marginal, critical, or catastrophic.

**28.4 – RISK PROJECTION (What is the likelihood?)**

**Risk projection**, also called **risk estimation**, attempts to rate each risk in two ways:

(1) The likelihood or probability that the risk is real.

(2) The consequences of the problems associated with the risk, should it occur.

You work along with other managers and technical staff to perform four risk projection steps:

**1.** Establish a scale that reflects the perceived likelihood of a risk.

**2.** Delineate the consequences of the risk.

**3.** Estimate the impact of the risk on the project and the product.

**4.** Assess the overall accuracy of the risk projection so that there will be no misunderstandings.

The intent of these steps is to consider risks in a manner that leads to prioritization. By prioritizing risks, you can allocate resources where they will have the most impact.

**28.4.1 - Developing a Risk Table**

A risk table provides you with a simple technique for risk projection.

1. You begin by listing all risks in the first column of the table.
2. Each risk is categorized in the second column (e.g., PS implies a project size risk).
3. The probability of occurrence of each risk is entered in the next column of the table (The probability value for each risk can be estimated by team members individually. One way to accomplish this is to poll individual team members in round-robin fashion until their collective assessment of risk probability begins to converge)
4. Next, the impact of each risk is assessed and an impact category is determined. The categories for each of the four risk components—performance, support, cost, and schedule—are averaged to determine an overall impact value.

The table is sorted by probability and by impact. High-probability, high-impact risks percolate to the top of the table, and low-probability risks drop to the bottom. This accomplishes first-order risk prioritization.

You can study the resultant sorted table and define a cutoff line. The **cutoff line** implies that only risks that lie above the line will be given further attention. Risks that fall below the line are reevaluated to accomplish second-order prioritization.

Risk impact and probability have a distinct influence on management concern. A risk factor that has a high impact but a very low probability of occurrence should not absorb a significant amount of management time. However, high-impact risks with moderate to high probability and low-impact risks with high probability should be carried forward into the risk analysis steps that follow.

All risks that lie above the cutoff line should be managed. The column labeled **RMMM contains a pointer into a risk mitigation, monitoring, and management plan or, alternatively,** a collection of risk information sheets developed for all risks that lie above the cutoff. The RMMM plan and risk information sheets are discussed in Sections 28.5 and 28.6.

Risk probability can be determined by making individual estimates and then developing a single consensus value.

**28.4.2 - Assessing Risk Impact**

Three factors affect the consequences that are likely if a risk does occur: its nature, its scope, and its timing. The nature of the risk indicates the problems that are likely if it occurs.

The following steps can be applied to determine the overall consequences of a risk:

(1) Determine the average probability of occurrence value for each risk component

(2) Determine the impact for each component based on the criteria shown

(3) Complete the risk table and analyze the results as described in the preceding sections.

The overall **risk exposure** (RE) is determined using the following relationship:

RE = P x C, where P is the probability of occurrence for a risk, and C is the cost to the project should the risk occur.

It can also be used to predict the probable increase in staff resources required at various points during the project schedule.

The risk projection and analysis techniques are applied iteratively as the software project proceeds. The project team should revisit the risk table at regular intervals, reevaluating each risk to determine when new circumstances cause its probability and impact to change.

**28.5 – RISK REFINEMENT**

During early stages of project planning, a risk may be stated quite generally. As time passes and more is learned about the project and the risk, it may be possible to refine the risk into a set of more detailed risks, each somewhat easier to mitigate, monitor, and manage.

One way to do this is to represent the risk in **condition-transition-consequence (CTC) format.** That is, the risk is stated in the following form:

Given that <condition> then there is concern that (possibly) <consequence>.

Using the CTC format for the reuse risk noted in Section 28.4.2, you could write:

Given that all reusable software components must conform to specific design standards and that some do not conform, then there is concern that (possibly) only 70 percent of the planned reusable modules may actually be integrated into the as-built system, resulting in the need to custom engineer the remaining 30 percent of components.

This general condition can be refined in the following manner:

**Sub-condition 1.** Certain reusable components were developed by a third party with no knowledge of internal design standards.

**Sub-condition 2.** The design standard for component interfaces has not been solidified and may not conform to certain existing reusable components.

**Sub-condition 3.** Certain reusable components have been implemented in a language that is not supported on the target environment.

The consequences associated with these refined sub-conditions remain the same

(i.e., 30 percent of software components must be custom engineered), but the refinement helps to isolate the underlying risks and might lead to easier analysis and response.

**28.6 - RISK MITIGATION, MONITORING, AND MANAGEMENT**

All of the risk analysis activities presented to this point have a single goal—to assist the project team in developing a strategy for dealing with risk. An effective strategy must consider three issues: risk avoidance, risk monitoring, and risk management and contingency planning.

If a software team adopts a proactive approach to risk, avoidance is always the best strategy. This is achieved by developing a plan for **risk mitigation**.For example, assume that high staff turnover is noted as a project risk *r*1. Based on past history and management intuition, the likelihood *l*1 of high turnover is estimated to be 0.70 (70 percent, rather high) and the impact *x*1 is projected as critical. That is, high turnover will have a critical impact on project cost and schedule.

To mitigate this risk, you would develop a strategy for reducing turnover. Among the possible steps to be taken are:

• Meet with current staff to determine causes for turnover (e.g., poor working conditions, low pay, and competitive job market).

• Mitigate those causes that are under your control before the project starts.

• Once the project commences, assume turnover will occur and develop techniques to ensure continuity when people leave.

• Organize project teams so that information about each development activity is widely dispersed.

• Define work product standards and establish mechanisms to be sure that all models and documents are developed in a timely manner.

• Conduct peer reviews of all work (so that more than one person is “up to speed”).

• Assign a backup staff member for every critical technologist.

As the project proceeds, *risk-monitoring* activities commence. The project manager monitors factors that may provide an indication of whether the risk is becoming more or less likely. In the case of high staff turnover, the general attitude of team members based on project pressures, the degree to which the team has jelled, interpersonal relationships among team members, potential problems with compensation and benefits, and the availability of jobs within the company and outside it are all monitored.

In addition to monitoring these factors, a project manager should monitor the effectiveness of risk mitigation steps. For example, a risk mitigation step noted here called for the definition of work product standards and mechanisms to be sure that work products are developed in a timely manner. This is one mechanism for ensuring continuity, should a critical individual leave the project. The project manager should monitor work products carefully to ensure that each can stand on its own and that each imparts information that would be necessary if a newcomer were forced to join the software team somewhere in the middle of the project.

***Risk management and contingency planning*** assumes that mitigation efforts have failed and that the risk has become a reality. Continuing the example, the project is well under way and a number of people announce that they will be leaving. If the mitigation strategy has been followed, backup is available, information is documented, and knowledge has been dispersed across the team. In addition, you can temporarily refocus resources (and readjust the project schedule) to those functions that are fully staffed, enabling newcomers who must be added to the team to “get up to speed.” Those individuals who are leaving are asked to stop all work and spend their last weeks in “knowledge transfer mode.” This might include video-based knowledge capture, the development of “commentary documents or Wikis,” and/or meeting with other team members who will remain on the project.

It is important to note that risk mitigation, monitoring, and management (RMMM) steps incur additional project cost. For example, spending the time to back up every critical technologist costs money. Part of risk management, therefore, is to evaluate when the benefits accrued by the RMMM steps are outweighed by the costs associated with implementing them. In essence, you perform a classic cost-benefit analysis.

If risk aversion steps for high turnover will increase both project cost and duration by an estimated 15 percent, but the predominant cost factor is “backup,” management may decide not to implement this step. On the other hand, if the risk aversion steps are projected to increase costs by 5 percent and duration by only 3 percent, management will likely put all into place.

For a large project, 30 or 40 risks may be identified. If between three and seven risk management steps are identified for each, risk management may become a project in itself! For this reason, you should adapt the Pareto 80–20 rule to software risk.

Experience indicates that 80 percent of the overall project risk (i.e., 80 percent of the potential for project failure) can be accounted for by only 20 percent of the identified risks. The work performed during earlier risk analysis steps will help you to determine which of the risks reside in that 20 percent (e.g., risks that lead to the highest risk exposure). For this reason, some of the risks identified, assessed, and projected may not make it into the RMMM plan—they don’t fall into the critical 20 percent (the risks with highest project priority).

Risk is not limited to the software project itself. Risks can occur after the software has been successfully developed and delivered to the customer. These risks are typically associated with the consequences of software failure in the field.

**Software safety and hazard analysis** are software quality assurance activities that focus on the identification and assessment of potential hazards that may affect software negatively and cause an entire system to fail. If hazards can be identified early in the software engineering process, software design features can be specified that will either eliminate or control potential hazards.

**28.7 - THE RMMM PLAN**

A risk management strategy can be included in the software project plan, or the risk management steps can be organized into a separate ***risk mitigation, monitoring, and management plan* (RMMM)*.***The RMMM plan documents all work performed as part of risk analysis and is used by the project manager as part of the overall project plan.

Some software teams do not develop a formal RMMM document. Rather, each risk is documented individually using a **risk information sheet**. In most cases, the RIS is maintained using a database system so that creation and information entry, priority ordering, searches, and other analysis may be accomplished easily.

Once RMMM has been documented and the project has begun, risk mitigation and monitoring steps commence. Risk mitigation is a problem avoidance activity. Whereas, Risk monitoring is a project tracking activity with three primary objectives: (1) to assess whether predicted risks do, in fact, occur; (2) to ensure that risk aversion steps defined for the risk are being properly applied; and (3) to collect information that can be used for future risk analysis.

In many cases, the problems that occur during a project can be traced to more than one risk. Another job of risk monitoring is to attempt to allocate origin [what risk(s) caused which problems throughout the project].