**Module-4**

**Test Execution, Process framework and planning**

1. **Overview of Test Execution:**

* Testing is the process of executing a program with the intent of finding faults. This definition of testing even though denotes a mechanical activity, testing involves many more than execution.
* These various approaches that we discussed in our last 3 modules gave an idea about what are the requirements to design a test case. Thus, test design requires creativity, and this ideally requires manual assistance and this entire process cannot be automated.
* Whereas test execution is a mechanical activity that requires repeating the same process with different values. Thus, this activity can be automated without human intervention.
* This topic aims to provide run time support for generating and managing test data, creating scaffolding for test execution, and automatically distinguishing between correct and incorrect test case executions.

1. **From Test case specifications to Test Cases:**

* Test Case specifications denotes the sequential steps in words describing how the testing must be performed whereas test case signifies associating a value to the specifications and producing results.

Eg: one that calls for “a sorted sequence, length greater than 2, with items in ascending order with no duplicates may designate many possible test cases.

* If the test design possesses concrete values and expected results, then producing a complete test case may be as simple as filling a template with those values.
* There is no sharp line between test case design and generation. A rule of thumb is that while test case design involves judgment and creativity, test case generation should be a mechanical step.
* Generally, the changes made to test case specification are localized and smaller, hence automatic generation of test cases from abstract test case specification will have little impact in the course of development.
* In the process of automating the test cases, test cases with independent constraints can looks simpler for automation than the ones with highly correlated constraints.

Eg: Identifying the even number or positive integer possess independent constraint whereas constraint on several input parameter values is tedious to automate.

* Test cases that involve highly intensive computations are difficult to automate.

Eg: If a test case calls for program execution corresponding to a certain traversal of transitions in a finite state machine model, the test data must trigger that traversal, which may be quite complex.

1. **Scaffolding:**
   1. **Need for Scaffolding:**

* In this era of agile development, only a portion of the full system is available for testing.
* Most of the partially developed system consists of one or more runnable programs serving as prototype of the actual system being built.
* Performing testing starts at the prototype level, eventually testing as and when newer features are added to the prototype, but the external interface of the evolving system may not be ideal for testing; often additional code must be added.
* Eg: In order management system, even if the actual subsystem for placing an order with a supplier is available and fully operational, it is probably not desirable to place a thousand supply orders each night as part of an automatic test run.
  1. **What is Scaffolding?**
* Code developed to facilitate testing is called scaffolding, by analogy to the temporary structures erected around a building during construction or maintenance.

Scaffoldings may include

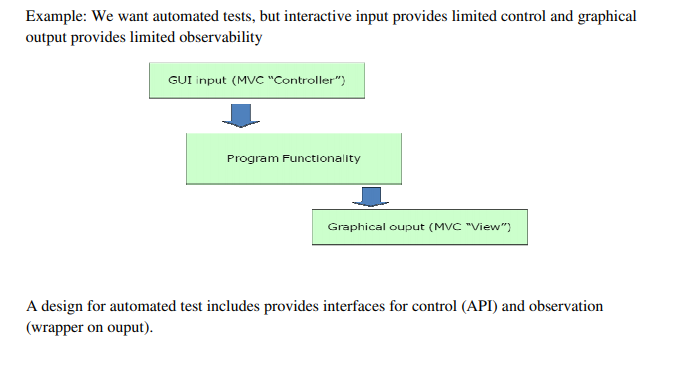
1)Test drivers (substituting for a main or calling population)

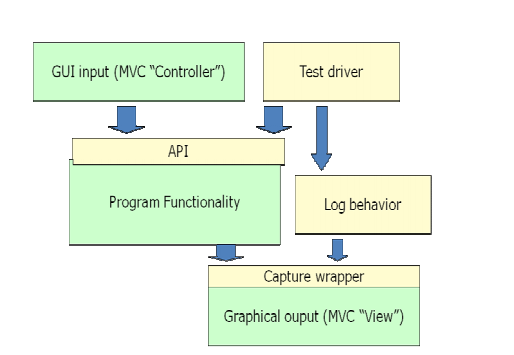
2)Test harness (substituting for parts of the deployment environment)

3)Stubs (substituting for functionally called or used by the software under test.

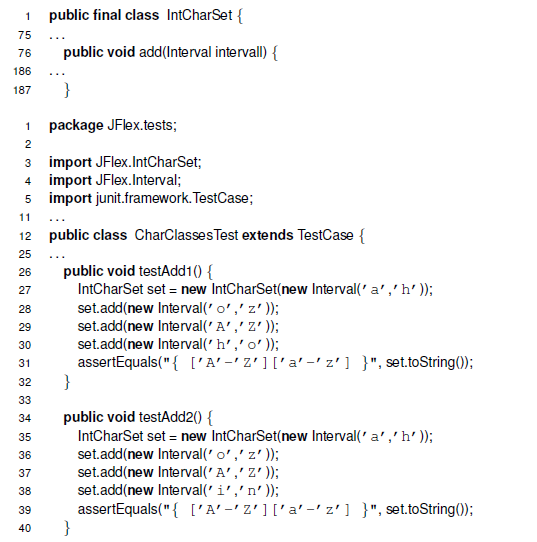
* A common estimate is that half of the code developed in a software project is scaffolding of some kind, but the amount of scaffolding that must be constructed with a software project can vary widely, and depends both on the application domain and the architectural design and build plan, which can reduce cost by exposing appropriate interfaces and providing necessary functionality in a rational order.
  1. **Purpose of Scaffolding:**
  + Purpose is to provide controllability to execute test cases and observability to judge the outcome.
  + Make the module executable
  + Check the interfaces

Consider, for example, an interactive program that is normally driven through a graphical user interface. Assume that each night the program goes through a fully automated and unattended cycle of integration, compilation, and test execution. It is necessary to perform some testing through the interactive interface, but it is neither necessary nor efficient to execute all test cases that way. Small driver programs, independent of the graphical user interface, can drive each module through large test suites in a short time.





* 1. **Types of Scaffolding -Generic Vs Specific:**
* Simplest Scaffolding generally includes at least a driver code for execution.
* Writing thousands of test specific drivers is generally not a good practice if the driver program written takes up a more general form. Eg: a test case specification calls for executing method calls in a sequence, this is easy to accomplish by writing the code to make the method calls in that sequence.
* The above said condition requires factor out some common driver code into reusable components. Aiming for creation of generic test drivers.
* At least some level of generic scaffolding support can be used across a wide class of applications. Such support typically includes, in addition to a standard interface for executing a set of test cases, basic support for logging test execution and results.
* Program below illustrates use of generic test scaffolding in the JFlex lexical analyser generator.



* Fully generic scaffolding may suffice for small numbers of hand-written test cases.
* We could build a driver and stubs for each test case or at least factor out some common code of the driver and test management (e.g., JUnit)
* ... or further factor out some common support code, to drive many test cases from data (as in DDSteps)
* – ... or further, generate the data automatically from a more abstract model (e.g., network traffic model).
* Mock is the simplest form of stub that can be generated automatically by analysis of the source code. A mock is limited to checking expected invocations and producing precomputed results that are part of the test case specification or were recorded in a prior execution.
* Depending on system build order and the relation of unit testing to integration in a particular process, isolating the module under test is sometimes considered an advantage of creating mocks, as compared to depending on other parts of the system that have already been constructed.
* The balance of quality, scope, and cost for a substantial piece of scaffolding software— say, a network traffic generator for a distributed system or a test harness for a compiler — is essentially similar to the development of any other substantial piece of software, including similar considerations regarding specialization to a single project or investing more effort to construct a component that can be used in several projects.
* The balance is altered in favor of simplicity and quick construction for the many small pieces of scaffolding that are typically produced during development to support unit and small-scale integration testing. For example, a database query may be replaced by a stub that provides only a fixed set of responses to query strings.

1. **Test Oracles:**
   1. **Why Oracles?**

* Relying on human intervention to judge test outcomes is not merely expensive, but also unreliable.
* Even the most conscientious and hard-working person cannot maintain the level of attention required to identify one failure in a hundred program executions, little more one or ten thousand.
* This must be automated and that is what we are trying to do with Oracles.
  1. **What is Test Oracle?**
* Software that applies a pass/fail criterion to a program execution is called a test oracle/ oracle
* A test oracle may apply a pass/fail criterion that reflects only a part of the actual program specification, or is an approximation, and therefore passes some program executions it ought to fail
* The best oracle we can obtain is an oracle that detects deviations from expectation that may or may not be actual failure.
* Several partial test oracles may be more cost-effective than one that is more comprehensive
* A test oracle may also give false alarms, failing an execution that is ought to pass.
* False alarms in test execution are highly undesirable.
* The best oracle we can obtain is an oracle that detects deviations from expectation that may or may not be actual failure.
* Not only oracles helps with execution, but it can also classify behaviours that exceed human capacity.
* Checking real time response against latency requirements or dealing with voluminous output data in a machine readable rather than human readable form.
  1. **Types of Oracles:**
     1. **Comparison Based Oracle:**
* With a comparison based oracle , we need predicted output for each input
* Oracle compares actual to predicted output, and reports failure if they differ.
* It is best suited for small number of hand generated test cases example: for handwritten Junit test cases.
* They are used mainly for small, simple test cases
* Expected outputs can also be produced for complex test cases and large test suites
* apture-replay testing, a special case in which the predicted output or behavior is preserved from an earlier execution
* Often possible to judge output or behavior without predicting it



**Fig: a test harness with a comparison based test oracle processes test cases consisting of (program input, predicted output) pairs.**

* + 1. **Partial Oracles:**
* Oracles that check results without references to predicted output are often partial, in the sense that they can detect some violations of the actual specification but not others.
* They check necessary but not enough conditions for correctness.
* A cheap partial oracle that can be used for many test cases is often combined with a more expensive comparison-based oracle that can be used with a smaller set of test cases for which predicted output has been obtained
* Specifications are often incomplete
* Automatic derivations of test oracles are impossible
  + 1. **Self Check as Oracles:**
* An oracle can also be written as self checks
* Often possible to judge correctness without predicting results.
* Typically these self checks are in the form of assertions, but designed to be checked during execution.
* It is generally considered good design practice to make assertions and self checks to be free of side effects on program state.
* Self checks in the form of assertions embedded in program code are useful primarily for checking module and subsystem-level specification rather than all program behaviour.
* Devising the program assertions that correspond in a natural way to specifications poses two main challenges:
* Bridging the gap between concrete execution values and abstractions used in specification
* Dealing in a reasonable way with quantification over collection of values



* Structural invariants are good candidates for self checks implemented as assertions
* They pertain directly to the concrete data structure implementation
* It is sometimes straight-forward to translate quantification in a specification statement into iteration in a program assertion
* A run time assertion system must manage ghost variables
* They must retain “before” values
* They must ensure that they have no side effects outside assertion checking

Advantages:

* -Usable with large, automatically generated test suites.

Limits:

* -often it is only a partial check. -recognizes many or most failures, but not all.

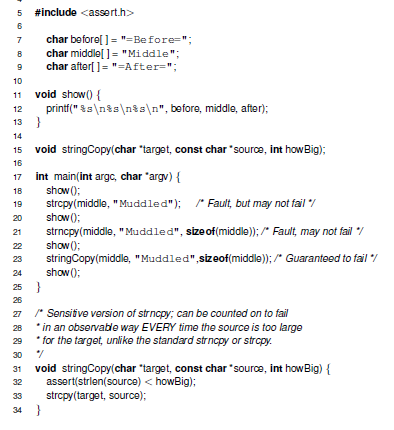
1. **Capture and Replay:**

* Sometimes it is difficult to either devise a precise description of expected behaviour or adequately characterize correct behaviour for effective self checks.
* Example: even if we separate testing program functionally from GUI, some testing of the GUI is required.
* If one cannot completely avoid human involvement test case execution, one can at least avoid unnecessary repetition of this cost and opportunity for error.
* The principle is simple:
* The first time such a test case is executed, the oracle function is carried out by a human, and the interaction sequence is captured. Provided the execution was judged (by human tester) to be correct, the captured log now forms an (input, predicted output) pair for subsequent automated testing.
* The savings from automated retesting with a captured log depends on how many build-and-test cycles we can continue to use it, before it is invalidated by some change to the program.
* Mapping from concrete state to an abstract model of interacting sequences is some time possible but is generally quite limited.

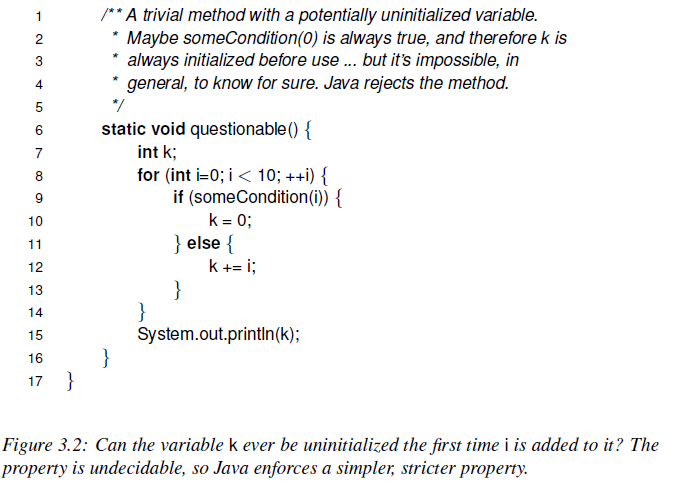
**Process Framework**

1. **Basic Principles:**

* Mature engineering disciplines are characterized by the basic principles.
* Principles don’t get restricted by technology and these provide a rationale for defining, selecting and applying techniques and methods.
* Along with the general engineering principles like partition, visibility and feedback principles specific to analysis and testing like sensitivity, redundancy and restriction are discussed in this topic.
  1. **Sensitivity:**
* Principle states that “better fail every time than sometimes”.
* Example: This principle can be justified by using the below example.



* The program calls three different string copy functions with the value “Muddled” that is too long to fit in the array “Middle”
* The vulnerability of strcpy is well known however this fault may or mayn’t propagate as a failure as this depends clearly on the arrangement of the memory.
* Common solution to strcpy is to use strncpy that it truncates the input without warning, and sometimes without properly null-terminating the output.
* The replacement function stringCopy, on the other hand uses an assertion to ensure that, if the target string is too long, the program always fails in an observable manner.
* Replacing strcpy and strncpy with stringCopy in the program is a simple example of application of the sensitivity principle in design.
* This sensitivity principle can be applied in three main ways: at the design level, changing the way in which the program fails; at the analysis and testing level, choosing a technique more reliable with respect to the property of interest; and at the environment level, choosing a technique that reduces the impact of external factors on the results.
* Sensitivity principle at the language level are used to check runtime array bound check, memory allocation and reference faults and a fail-fast iterator has the property that an immediate and observable failure (throwing ConcurrentModificationException) occurs when the illegal modification occurs.
* This principle greatly helps in the reduction of cost associated with correcting failures. Cost of correcting faults at unit level is low compared to integration and system testing. By making use of the sensitivity principle one can identify faults at the unit level resulting in less cost and effort.
* Sensitivity application is far beyond the code and design that can go upto testing and analysis as well in the form of removing deadlocks and race conditions.
* Code inspection can reveal many subtle faults. However, inspection teams mayproduce completely different results depending on the cohesion of the team, the discipline of the inspectors, and their knowledge of the application domain and the design technique. The use of detailed checklists and a disciplined review process may reduce the influence of external factors, such as teamwork attitude, inspectors’ discipline, and domain knowledge, thus increasing the predictability of the results of inspection. In this case, sensitivity is applied to reduce the influence of external factors.
  1. **Redundancy:**
* Redundancy means making intension explicit i.e one part of a software artifact (program, design document, etc.) constrains the content of another, then they are not entirely independent, and it is possible to check them for consistency.
* In **software test and analysis**, we wish to *detect faults that could lead to differences between intended behavior and actual behavior,* so *the most valuable form of redundancy is in the form of an explicit, redundant statement of intent.*
* **Static type checking** is a classic application of this principle: The **type declaration** is a statement of intent that is at least partly redundant with the use of a variable in the source code.
* The type declaration constrains other parts of the code, so a consistency check (type check) can be applied.
* An important trend in the evolution of programming languages is **introduction of additional ways to declare intent and automatically check for consistency**.
* For example, Java enforces rules about explicitly declaring each exception that can be thrown by a method.
* **Checkable redundancy** is not limited to program source code, nor is it something that can be introduced only by programming language designers.
* For example, software design tools typically provide ways to check consistency between different design views or artifacts.
* When redundancy is already present - as between a software specification document and source code - then the remaining challenge is to *make sure the information is represented in a way that facilitates cheap, thorough consistency checks.*
* Checks that can be implemented by ***automatic tools*** are usually preferable, but there is value even in organizing information to make inconsistency easier to spot in manual inspection.
* **Defensive programming**, explicit run-time checks for conditions that should always be true if the program is executing correctly, is another application of redundancy in programming.
  1. **Restriction:**
* Restriction means “making the problem easier”.
* When there are no acceptably cheap and effective ways to check a property, sometimes one can change the problem by checking a different, more restrictive property or by limiting the check to a smaller, more restrictive class of programs.
* **Consider the problem of ensuring that each variable is initialized before it is used, on every execution.**
* Although the property is Simple, it is not possible for a compiler or analysis tool to precisely determine whether it holds.

****

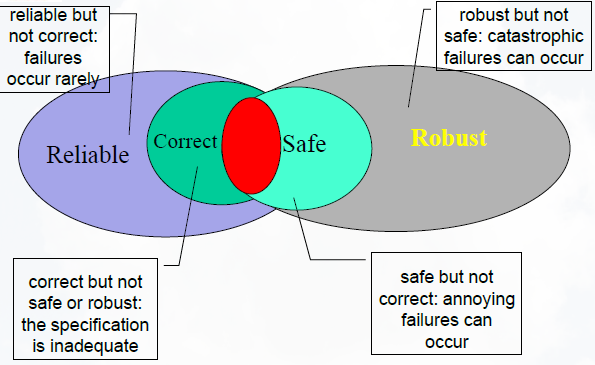
* ***A program is not permitted to have any syntactic control paths on which an uninitialized reference could occur, regardless of whether those paths could actually be executed*.**
* **initialization before use is a program source code restriction that enables precise, efficient checking of a simple but important property by the compiler.**
* Restriction is mainly a principle to be applied in design for test in the form of detecting potential access of uninitialized variables, or non serializable execution of transactions.
* **Stateless component interfaces** are an example of **restriction applied at the architectural level**. An interface is stateless if each service request (method call, remote procedure call, message send and reply) is independent of all others; that is, the service does not "remember" anything about previous requests.
  1. **Partition:**
* Partition, often also known as "divide and conquer“, is a general engineering principle.
* Dividing a complex problem into sub problems to be attacked and solved independently is probably the most common human problem-solving strategy.
* Software engineering in particular applies this principle in many different forms and at almost all development levels, from early requirements specifications to code and maintenance.
* Analysis and testing are no exception: the partition principle is widely used and exploited.
* Partitioning can be applied both at **process** and **technique** levels.
* At the **process level**, we divide complex activities into sets of simple activities that can be attacked independently.
* For example, testing is usually divided into unit, integration, subsystem, and system testing.
* In this way, we can focus on different sources of faults at different steps, and at each step, we can take advantage of the results of the former steps.
* Many static analysis techniques first construct a model of a system and then analyze the model.
* In this way they divide the overall analysis into two subtasks: first simplify the system to make the proof of the desired properties feasible and then prove the property with respect to the simplified model.
* The question "Does this program have the desired property?" is decomposed into two questions, "Does this model have the desired property?" and "Is this an accurate model of the program?"
* Systematic test strategies must identify a finite number of classes of test cases to execute.
* Verification techniques fold the input space according to specific characteristics, grouping homogeneous data together and determining partitions.
  1. **Visibility:**
* Visibility means making information accessible.
* Visibility is closely related to **observability**, the ability to extract useful information from a software artifact.
* A variety of simple techniques can be used to improve observability.
* Architectural design and build plan of system determines what we can observe at each stage of development which determines the visibility of progress against goals at that stage.
* Protocols like HTTP and SMTP are based on exchange of simple text commands, we can choose simple human-readable text rather than a more compact binary format at application layer Which in turn help in observability , helps in construction of test drivers and oracles.
* During design and coding visibility property is considered and all the effort is made to make things accessible and observable to different programmer and tester.
  1. **Feedback:**
* Feedback is another classic engineering principle that applies to analysis and testing.
* Feedback applies both to the process itself (process improvement) and to individual techniques (e.g., using test histories to prioritize regression testing).
* *Systematic inspection and walkthrough derive part of their success from feedback.*
* Participants in inspection are guided by checklists, and checklists are revised and refined based on experience.
* New checklist items may be derived from root cause analysis, analyzing previously observed failures to identify the initial errors that lead to them.

1. **Test and Analysis Activities within a software Process:**
   1. **The Quality Process:**

* The quality process provides a framework for selecting and arranging activities aimed at a goal, while also considering interactions and trade-offs with other important goals.
* The goal is to deliver a high-quality product within the stipulated schedule/ highly user-friendly product.
* If this objective must be met, the process that we are adopting should be competent enough and all such activities falls into the rubric quality assurance activities.
* Dependability property is an unavoidable term while talking about adopting a good quality process.
* high dependability is usually in tension with time to market, and in most cases it is better to achieve a reasonably high degree of dependability on a tight schedule than to achieve ultra-high dependability on a much longer schedule, although the opposite is true in some domains (e.g., certain medical devices).
* The quality process should check completeness, cost effectiveness and timeliness.
* **Completeness means that appropriate activities are planned to detect each important class of faults.** What the important classes of faults are depends on the application domain, the organization, and the technologies employed (e.g., memory leaks are an important class of faults for C++ programs, but seldom for Java programs).
* **Timeliness means that faults are detected at a point of high leverage, which in practice almost always means that they are detected as early as possible.**
* **Cost-effectiveness** means that, subject to the constraints of completeness and timeliness, one chooses activities depending on their cost as well as their effectiveness.
* Cost must be considered over the whole development cycle and product life, so the dominant factor is likely to be the cost of repeating an activity through many change cycles.
  1. **Planning and Monitoring:**
* Delivering a good product at right time is the most fundamental factor to achieve quality.
* Timely delivery is possible only if there exists process visibility.
* Process visibility is a key factor in software process in general, and software quality processes.
* Visibility of a process should be in a such that everyone under process can answer to questions relative to schedule.
* How much is the progress against the plan is the question that must be answered?
* One cannot gain confidence in the quality of the software system long before it reaches final testing, the quality process has not achieved adequate visibility.
* A well-designed quality process balances several activities across the whole development process, selecting and arranging them to be as cost-effective as possible, and to improve early visibility.
* Visibility is particularly challenging and is one reason (among several) that quality activities are usually placed as early in a software process as possible.
* For example, one designs test cases at the earliest opportunity (not “just in time”) and uses both automated and manual static analysis techniques on software artefacts that are produced before actual code.
* Quality goals can be achieved only by careful planning of activities and these activities match to identified project objectives.
* Planning is part of quality process and it is elaborated and revised throughout the project.
* Analysis and testing identify company or project wide standards that must be satisfied:
  + Procedures for obtaining quality certificates required for certain class of products, techniques, and tools.
  + Some companies follow ISO 9000 or Capability Maturity Model. All these need detailed documentation and management of analysis and test activities and well-defined phases.
  1. **Quality Goals:**
* *Process visibility requires a clear specification of goals*, and in the case of quality process visibility this includes a *careful distinction among dependability qualities.*
* A team that does not have a clear idea of what factors are important and what factors are not will succeed in product development.
* Goals are needed to be refined into a *clear and reasonable set of objectives.*
* Quality goals of one project varies from another project.
* For some time to market is important and for some safety is important.

Eg: Life critical machines such as the ones used in hospitals requires safety than time to market whereas the web applications just requires timely delivery.

* Product qualities are the goals of software quality engineering, and process qualities *are means to achieve those goals*.
* Software product qualities can be divided into those that are directly visible to a client (external) and those that primarily affect the software development organization. (internal)
* **Reliability**, for example, is directly visible to the client.
* Maintainability primarily affects the development organization, although its consequences may indirectly affect the client as well, for example, increasing the time between product releases.
* Properties that are directly visible to users of a software product, such as dependability, latency, usability, and throughput, are called external properties.
* Properties that are not directly visible to end users, such as maintainability, reusability, and traceability, are called internal properties.
* Most important task in software quality analysis is to make desired properties explicit, as properties that are unspecified may eventually surface unpleasantly when we discover that they are not met.
* Most of the cases it is better to specify implicit properties also Cleary.
* It is better to mention usability property explicitly than assuming developer or tester will consider.
  1. **Dependability Properties:**
     1. **Correctness:**
  + The simplest of the dependability properties is correctness.
  + A program or system is correct if it is consistent with its specification.
  + By definition, a specification divides all possible system behaviours into two classes, successes (or correct executions) and failures.
  + All the possible behaviours of a correct system are successes.
  + A program cannot be mostly correct or somewhat correct or 30% correct. It is correct on all possible behaviours, or else it is not correct.
    1. **Reliability:**
* Reliability is a statistical approximation to correctness, in the sense that 100% reliability is indistinguishable from correctness.
* Roughly speaking, reliability is a measure of the likelihood of correct function for some "unit" of behaviour, which could be a single use or program execution or a period.
* *Like correctness*, reliability is relative to a specification (which determines whether a unit of behaviour is counted as a success or failure).
* *Unlike correctness*, reliability is also relative to a usage profile. The same program can be reliable depending on how it is used.
* **Availability** is an appropriate measure of time for which the system is available to user and providing desired functionality.
* For example, a failure of a network router may make it impossible to use some functions of a local area network until the service is restored; between initial failure and restoration we say the router is "down" or "unavailable." The availability of the router is the time in which the system is "up" (providing normal service) as a fraction of total time. Thus, a network router that averages 1 hour of down time in each 24-hour period would have an availability of 23/24, or 95.8%.
* **Mean time between failures (MTBF)** is yet another measure of reliability, also using time as the unit of execution.
* The hypothetical network switch that typically fails once in a 24hour period and takes about an hour to recover has a mean time between failures of 23 hours
  + 1. **Safety:**
* Safety is concerned with preventing certain undesirable behaviors, called hazards.
* It is quite explicitly not concerned with achieving any useful behavior apart from whatever functionality is needed to prevent hazards.
* Software safety is typically a concern in "critical" systems such as avionics and medical systems, but the basic principles apply to any system in which particularly undesirable behaviors can be distinguished from failure.
* For example, while it is annoying when a word processor crashes, it is much more annoying if it irrecoverably corrupts document files.
* Just as correctness is meaningless without a specification of allowed behaviors, safety is meaningless without a specification of hazards to be prevented, and in practice the *first step of safety analysis is always finding and classifying hazards.*
* Typically, hazards are associated with some system in which the software is embedded (e.g., the medical device), rather than the software alone.
* The distinguishing feature of safety is that it is concerned only with these hazards, and not with other aspects of correct functioning.
  + 1. **Robustness:**
* *Correctness and reliability depend upon normal operating conditions.*
* It is not reasonable to expect a word processing program to save changes normally when the file does not fit in storage, or to expect a database to continue to operate normally when the computer loses power, or to expect a Web site to provide completely satisfactory service to all visitors when the load is 100 times greater than the maximum for which it was designed.
* Software that fails under these conditions, which violate the premises of its design, may still be "correct" but we need to verify how system behaves when these conditions occur.
* It is acceptable that the word processor fails to write the new file that does not fit on disk, but it is unacceptable if it corrupts the previous version of the file in attempt.
* It is acceptable for the database system to cease to function when the power is cut, but unacceptable for it to leave the database in a corrupt state.
* Consider a website handling 1000 users and when users increase then it is preferable for the Web system to turn away some arriving users rather than becoming too slow for all or crashing.
* *Software that gracefully degrades or fails "softly" outside its normal operating parameters is robust.* 
  + 1. **Relation among dependability properties:**

****

* 1. **Analysis:**
* Analysis does not involve execution of actual code.
* There are two types of analysis
  + Manual inspection
  + Automated analysis
* Analysis is well suited for early stages of specification and design where we do not have executable code.
* Inspection is applied to document and to code.
* Automated Static Analysis can be applied to documents like specification where we have UML diagrams not to documents with natural language.
* Static analysis is a method of computer program verifying code without executing the program. Automated tools will assist us in conducting static analysis.
  1. **Testing:**
* Despite the usefulness of manual inspections for a variety of documents including but not limited to program source code, dynamic testing remains a dominant technique.
* Tests are executed when the corresponding code is available, but testing activities start earlier, as soon as the artefacts required for designing test case specifications are available.
* Thus, acceptance and system test suites should be generated before integration and unit test suites, even if executed in the opposite order.
* Early design of test cases possesses numerous benefits ranging from early review and repair.
* programmers may use test cases to illustrate and clarify the software specifications, especially for errors and unexpected conditions
* “Earlier is better” rule dictates using inspection to reveal flaws in requirements and design before they are propagated to program code, the same rule dictates module testing to uncover as many program faults as possible before they are incorporated in larger subsystems of the product.
  1. **Improving the Process:**
* Goal of the quality process is to find the different ways in which we can find the defects.
* The quality process, as well as the software development process as a whole, can be improved by gathering, analyzing, and acting on data regarding faults and failures.
* One of the way through which the defect and faults are collected is through the subsequent occurrence of similar faults by developers.
* The goal of quality process improvement is to find cost-effective countermeasures for classes of faults that are expensive because they occur frequently, or because the failures they cause are expensive, or because, once detected, they are expensive to repair.
* Countermeasures may be either prevention or detection and may involve either quality assurance activities (e.g., improved checklists for design inspections) or other aspects of software development (e.g., improved requirements specification methods).
* We can improve the process used for s/w development and testing by feedback from the all the stages of process.
* In the process of feedback the most difficult process is collecting gathering sufficiently complete and accurate raw data about faults and failures.
* These type of data gathered from one project may not be beneficial to current project so it is better to integrate data collection as well as possible with other, normal development activities, such as version and configuration control, project management, and bug tracking.
* Raw data on faults and failures must be aggregated into categories and prioritized.
* Aiming for flawless categorization is impossible; however some form of categorization that is sufficiently fine-grained and tends to aggregate faults with similar causes and possible remedies, and that can be associated with at least rough estimates of relative frequency and cost.
* Quality process can be improved by classifying different defects and conducting root cause analysis for each defect.
  1. **Organizational Factors:**
* Quality process or designing a product with good quality not only depends on tools, process but they do depend on people building product
* There are many organizational factors affect quality process :

Poor Allocation of Responsibility:

* Poor allocation of responsibilities can lead to major problems in which pursuit of individual goals conflicts with overall project success.
* For example, splitting responsibilities of development and quality-control between a development and a quality team, and rewarding high productivity in terms of lines of code per person-month during development may produce undesired results.
* The development team, not rewarded to produce high-quality software, may attempt to maximize productivity to the detriment of quality.
* The resources initially planned for quality assurance may not suffice if the initial quality of code from the“very productive” development team is low.
* Conflict in team.
* Poor interaction among developer and tester.
* Management not willing to buy sophisticated tools for development and testing process.
* Cost factor.

**PLANNING AND MONITORING THE PROCESS**

1. **Quality Process:**

* Software development process is a complex activity and this can be successful only if the entire process is properly planned and monitored.
* Planning is necessary to order, provision, and coordinate all the activities that support a quality goal, and monitoring of actual status against a plan is required to steer and adjust the process.
* Planning involves scheduling activities, allocating resources, and devising observable, unambiguous milestones against which progress and performance can be monitored.
* Monitoring means answering the question, “How are we doing?”
* Quality planning is one aspect of project planning, and quality processes must be closely coordinated with other development processes.
* Coordination among quality and development tasks may constrain ordering (e.g., unit tests are executed after creation of program units).
* Quality planning begins at the inception of a project and is developed with the overall project plan, instantiating and building on a quality strategy that spans several projects.
* Like the overall project plan, the quality plan is developed incrementally, beginning with the feasibility study and continuing through development and delivery.
* Quality process is a set of activities and responsibilities focused primarily on ensuring adequate dependability concerned with project schedule or with product usability
* A framework for selecting and arranging activities considering interactions and trade-offs.
* Follows the overall software process in which it is embedded
* Example: waterfall software process ––> “V model”: unit testing starts with implementation and finishes before integration
* Example: XP and agile methods ––> emphasis on unit testing and rapid iteration for acceptance testing by customers

**Internal consistency checks**

* compliance with structuring rules that define “well-formed” artifacts of that type
* a point of leverage: define syntactic and semantic rules thoroughly and precisely enough that many common errors result in detectable violations

**External consistency checks**

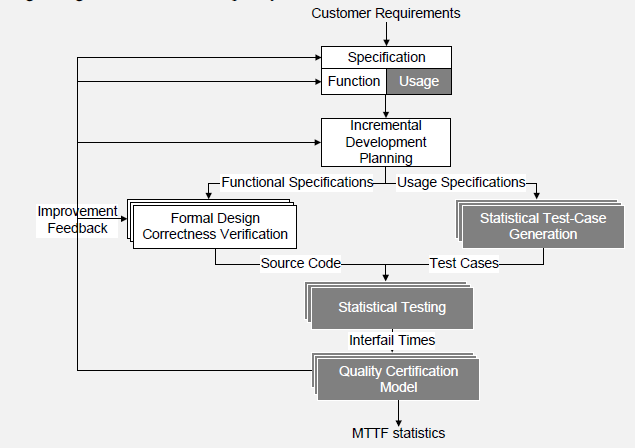
* consistency with related artifacts
* Often: conformance to a “prior” or “higher-level” specification

**Generation of correctness hypothesis**

* Correctness hypothesis: lay the groundwork for external consistency checks of other work products
* Often: motivate refinement of the current product.

1. **Test and Analysis Strategies:**

* Lessons of past experience are an important asset of organizations that rely heavily on technical skills.
* A body of explicit knowledge, shared and refined by the group, is more valuable than islands of individual competence.
* Organizational knowledge in a shared and systematic form is more amenable to improvement and less vulnerable to organizational change, including the loss of key individuals.
* Capturing the lessons of experience in a consistent and repeatable form is essential for avoiding errors, maintaining consistency of the process, and increasing development efficiency.
  1. **Clean room process model:**
* The Cleanroom process model, introduced by IBM in the late 1980s, pairs development with V&V activities and stresses analysis over testing in the early phases.
* The Cleanroom process involves two cooperating teams, the development and the quality teams, and five major activities: specification, planning, design and verification, quality certification, and feedback.

****

* In the specification activity, the development team defines the required behavior of the system.
* The quality team defines usage scenarios that are later used for deriving system test suites.
* The planning activity identifies incremental development and certification phases.
* Based on specifications, the development and certification teams together define an initial plan for developing increments that will accumulate into the final system.
* For example, a 100 KLOC system might be developed in five increments averaging 20 KLOC each. The time it takes to design and verify increments varies with their size and complexity. Increments that require long lead times may call for parallel development.
* The development team then carries out a design and correctness verification cycle for each increment.
* The certification team proceeds in parallel, using the usage specification to generate test cases that reflect the expected use of the accumulating increments
* Design and code undergo formal inspection (“Correctness verification”) before release.
* One of the key premises underpinning the Cleanroom process model is that rigorous design and formal inspection produce “nearly fault-free software.”
* Usage profiles generated during specification are applied in the statistical testing activity to gauge quality of each release.
* Reliability is measured in terms of mean time between failures (MTBF) and is constantly controlled after each release. Failures are reported to the development team for correction, and if reliability falls below an acceptable range, failure data is used for process improvement before the next incremental release.
  1. **Strategy:**

The quality strategy is an intellectual asset of an individual organization prescribing

a set of solutions to problems specific to that organization.

Among the factors that particularize the strategy are:

**Structure and size:**

* Large organizations typically have sharper distinctions between development and quality groups.
* In smaller organizations, it is more common for a single person to serve multiple roles.
* In a smaller organization, or an organization that has devolved responsibility to small, semi-autonomous teams, there is typically less emphasis on formal communication and documents but a greater emphasis on managing and balancing the multiple roles played by each team member.

**Overall process:**

* We have already noted the intertwining of quality process with other aspects of an overall software process, and this is of course reflected in the quality strategy.
* For example, if an organization follows the Cleanroom methodology, then inspections will be required but unit testing forbidden.
* An organization that adopts the XP methodology is likely to follow the “test first” and pair programming elements of that approach, and in fact would find a more document-heavy approach a difficult fit.
* Notations, standard process steps, and even tools can be reflected in the quality strategy to the extent they are consistent from project to project.
* For example, if an organization consistently uses a particular combination of UML diagram notations to document subsystem interfaces, then the quality strategy might include derivation of test designs from those notations, as well as review and analysis steps tailored to detect the most common and important design flaws at that point. If a particular version and configuration control system is woven into process management, the quality strategy will likewise exploit it to support and enforce quality process steps.

**Application domain:**

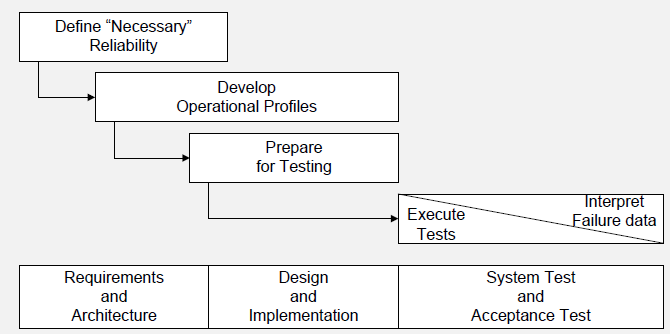
* The domain may impose both particular quality objectives (e.g., privacy and security in medical records processing), and in some cases particular steps and documentation required to obtain certification from an external authority.
* For example, the RTCA/DO-178B standard for avionics software requires testing to the modified condition/decision coverage (MC/DC) criterion.

**SRET (Software Reliability Engineering Testing):**

* The software reliability engineered testing (SRET) approach, developed at AT&T in the early 1990s, assumes a spiral development process.
* SRET identifies two main types of testing:

1. development testing, used to find and remove faults in software at least partially developed in-house
2. Certification testing used to either accept or reject outsourced software.

* The SRET approach includes seven main steps.
* Two initial, quick decision-making steps determine which systems require separate testing and which type of testing is needed for each system to be tested.
* The five core steps are executed in parallel with each coil of a spiral development process.

****

**Define “Necessary” Reliability:**

Determine operational models, that is, distinct patterns of system usage that require separate testing, classify failures according to their severity, and engineer the reliability strategy with fault prevention, fault removal, and fault tolerance activities.

**Develop Operational Profiles:**

Develop both overall profiles that span operational models and operational profiles within single operational models.

**Prepare for Testing:**

Specify test cases and procedures.

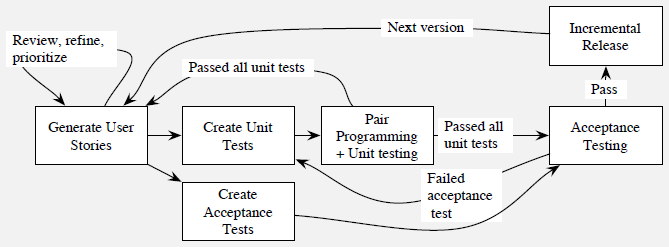
**Execute Tests**

**Interpret Failure Data:**

* Interpretation of failure data depends on the type of testing.
* In development testing, the goal is to track progress and compare present failure intensities with objectives. In certification testing, the goal is to determine if a software component or system should be accepted or rejected.

**Extreme Programming:**

* The extreme programming methodology (XP) emphasizes simplicity over generality, global vision and communication over structured organization, frequent changes over big releases, continuous testing and analysis over separation of roles and responsibilities, and continuous feedback over traditional planning.
* Customer involvement in an XP project includes requirements analysis (development, refinement, and prioritization of user stories) and acceptance testing of very frequent iterative releases. Planning is based on prioritization of user stories, which are implemented in short iterations. Test cases corresponding to scenarios in user stories serve as partial specifications.



* Test cases suitable for batch execution are part of the system code base and are implemented prior to the implementation of features they check (“test-first”).
* Developers work in pairs, incrementally developing and testing a module.
* Pair programming effectively conflates a review activity with coding.
* Each release is checked by running all the tests devised up to that point of development, thus essentially merging unit testing with integration and system testing.
* A failed acceptance test is viewed as an indication that additional unit tests are needed.

**Analysis and Test Plan:**

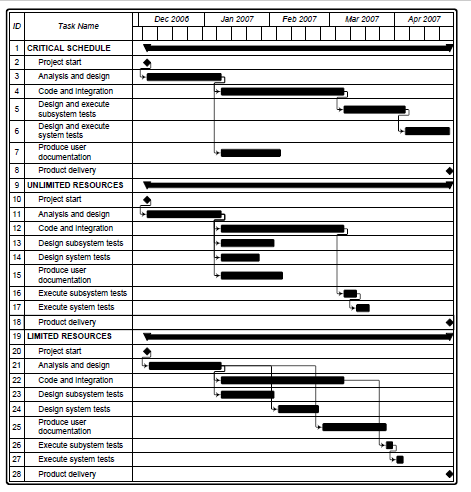
* An analysis and test plan consists of steps to be taken in a particular project.

A plan should answer the following questions:

* **What quality activities will be carried out?**
* **What are the dependencies among the quality activities and between quality and other development activities?**
* **What resources are needed and how will they be allocated?**
* **How will both the process and the product be monitored?**

**Note: In this topic, start explaining the quality manager responsibilities.**

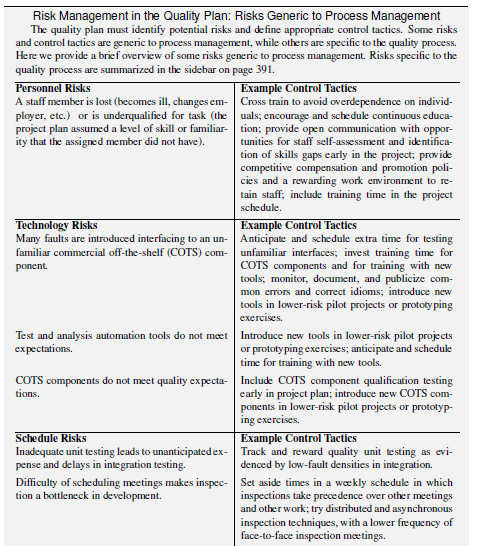
* The major objective of a quality manager is to prepare a quality plan that can answer the above mentioned questions.
* The quality manager must identify the items and features to be verified, the resources and activities that are required, the approaches that should be followed, and criteria for evaluating the results.
* The initial quality plan may be incomplete in nature as it purely depends on the past experiences and this can be incrementally refined.
* Quality goals must be expressed in terms of properties satisfied by the product and must be further elaborated with metrics that can be monitored during the project.
* *example:* before entering acceptance testing, the product must pass comprehensive system testing with no critical or severe failures
* After making clear quality objectives, the the next step in construction of a quality plan is to produce an overall rough list of tasks.
* The quality strategy and experience provide a basis for customizing the list to the current project and for scaling tasks appropriately.
* The quality manager must break large tasks into component subtasks to obtain better estimates, but it is inevitable that some task breakdown must await further elaboration of the overall project design and schedule.
* Another important responsibility of manager is to identify dependencies so that assessment data are provided continuously throughout the project, without unnecessary delay of other development activities.
* It is always better to do schedule quality activities as this can provide better visibility.
* If one has a choice between completing two tasks in four months, or completing the first task in two months and then the second in another two months, the schedule that brings one task to completion earlier is generally advantageous from the perspective of process visibility, as well as reduced coordination overhead.
* The quality plan, like the overall project plan, should include an explicit risk plan that lists major risks and contingencies.
* If a task in the overall process propagates throughout the process and causes delay in delivery must be identified and addressed at the earliest and this is performed as part of the scheduling activity.
* One first identifies the critical paths through the project schedule.
* Critical paths are chains of activities that must be completed in sequence and that have maximum overall duration.
* A critical dependence occurs when a task on a critical path is scheduled immediately after some other task on the critical path, particularly if the length of the critical path is close to the length of the project.
* The primary tactic available for reducing the schedule risk of a critical dependence is to decompose a task on the critical path, factoring out subtasks that can be performed earlier.
* The figure below shows sample schedule for the same project under three different conditions:

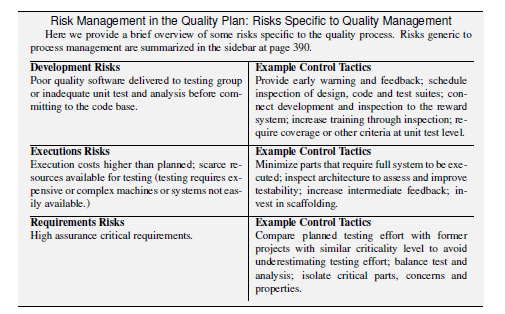


* The first schedule mentioned as critical schedule the tasks Analysis and design, Code and Integration, Design and execute subsystem tests, and Design and execute system tests form a critical path that spans the duration of the entire project. A delay in any of the activities will result in late delivery. In this schedule, only the Produce user documentation task does not belong to the critical path, and thus only delays of this task can be tolerated.
* In UNLIMITED RESOURCES, the test design and execution activities are separated into distinct tasks. Test design tasks are scheduled early, right after analysis and design, and only test execution is scheduled after Code and integration. In this way the tasks Design subsystem tests and Design system tests are removed from the critical path, which now spans 16 weeks with a tolerance of 5 weeks with respect to the expected termination of the project.
* This schedule assumes enough resources for running Code and integration, Production of user documentation, Design of subsystem tests, and Design of system tests.
* In limited resources, rearranges tasks to meet resource constraints. We can see that, despite the limited parallelism, decomposing testing activities and scheduling test design earlier results in a critical path of 17 weeks, 4 weeks earlier than the expected termination of the project.
* The GANTT diagram shows four main groups of analysis and test activities: design inspection, code inspection, test design, and test execution.
* The distribution of activities over time is constrained by resources and dependence among activities.
* The GANTT diagram does not highlight intermediate milestones, but we can easily identify two in April and July, thus dividing the development into three main phases.
* The first phase (January to April) corresponds to requirements analysis and architectural design activities and terminates with the architectural design baseline. In this phase, the quality team focuses on design inspection and on the design of acceptance and system tests.
* The second phase (May to July) corresponds to subsystem design and to the implementation of the first complete version of the system. It terminates with the first stabilization of the administrative business logic subsystem. In this phase, the quality team completes the design inspection and the design of test cases.
* In the final stage, the development team produces the final version, while the quality team focuses on code inspection and test execution.

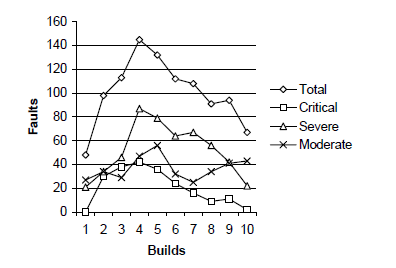
**Risk Planning:**

* **Identifying risks, addressing risks is an inevitable component in quality project plan.**
* **Risks cannot be eliminated, but they can be assessed, controlled, and monitored**

****

****

* In typical projects, the total number of open faults rises during early stages of construction and then falls as the product nears readiness.
* **Severe and critical faults** display this pattern more strongly than less critical faults.
* Comparing fault data on the current project to the history of similar projects (preferably for the same organization, and as similar as possible) provides a rough guide to progress. Large deviations from historical patterns should be explained.
* If we don’t see the predicted rise in the number of detected faults early in the project, we need to learn whether it is because of unusually high quality or unusually poor test and analysis.
* If we don’t see severe and criitical paths decreasing, is it because development is still unstable, introducing new faults with changes?
* Analysis by analogy (comparison to history) is far from perfect, but it is often the best tool available for recognizing problems before they become critical.



**Orthogonal Defect Classification:**

* The orthogonal defect classification (ODC) approach has two main steps:

(1) fault classification and

(2) fault analysis.

* ODC fault classification is done in two phases: when faults are detected and when they are fixed.
* At detection time, we record the activity executed when the fault is revealed, the trigger that exposed the fault, and the perceived or actual impact of the fault on the customer.
* The target indicates the entity that has been fixed to remove the fault, and can be requirements, design, code, build/package, or documentation/development.
* The type indicates the type of the fault.
* Taxonomies depend on the target.
* Fault types may be augmented with an indication of the nature of the fault, which can be: missing, that is, the fault is to due to an omission, as in a missing statement; incorrect, as in the use of a wrong parameter; or extraneous, that is, due to something not relevant or pertinent to the document or code, as in a section of the design document that is not pertinent to the current product and should be removed.

**1.Distribution of fault types versus activities**

Different quality activities target different classes of faults

* + example: algorithmic faults are targeted primarily by unit testing.

Activity: A high proportion of faults detected by unit testing should belong to this class

* + - If proportion of algorithmic faults found during unit testing
      * unusually small
      * larger than normal
      * unit tests may not have been well designed
    - proportion of algorithmic faults found during unit testing unusually large
    - integration testing may not focused strongly enough on interface faults

**Distribution of triggers over time during field test**

* + Faults corresponding to simple usage should arise early during field test, while faults corresponding to complex usage should arise late.
  + The rate of disclosure of new faults should asymptotically decrease
  + Unexpected distributions of triggers over time may indicate poor system or acceptance test
    - Triggers that correspond to simple usage reveal many faults late in acceptance testing
    - The sample may not be representative of the user population
    - Continuously growing faults during acceptance test
    - System testing may have failed

**Age distribution over target code**

* + Most faults should be located in new and rewritten code
  + The proportion of faults in new and rewritten code with respect to base and re-fixed code should gradually increase
  + Different patterns
  + may indicate holes in the fault tracking and removal process
  + may indicate inadequate test and analysis that failed in revealing faults early
  + Example
    - increase of faults located in base code after porting may indicate inadequate tests for portability

**Distribution of fault classes over time**

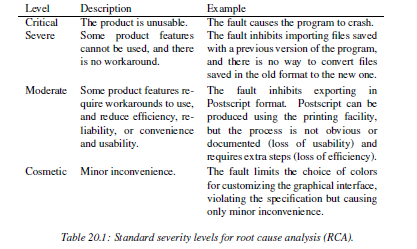
* + The proportion of missing code faults should gradually decrease
  + The percentage of extraneous faults may slowly increase, because missing functionality should be revealed with use
    - increasing number of missing faults
    - may be a symptom of instability of the product
    - sudden sharp increase in extraneous faults may indicate maintenance problems

**Improving the Process:**

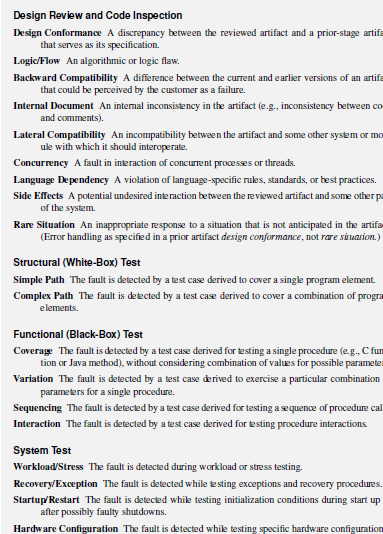
**Improving the Current and Next Processes:**

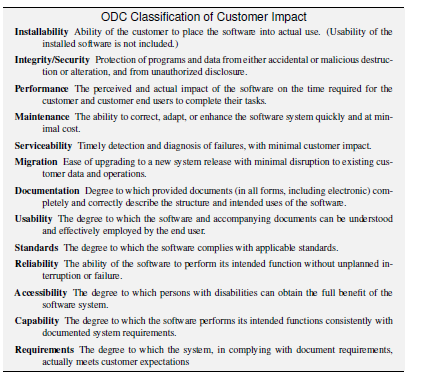
**Root Cause Analysis:**

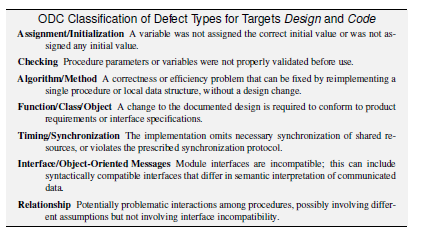
* severity = impact of the fault on the product
* Kind
  + No fixed set of categories; Categories evolve and adapt
  + Goal:
    - Identify the few most important classes of faults and remove their causes
    - Differs from ODC: Not trying to compare trends for different classes of faults, but rather *focusing* on a few important classes

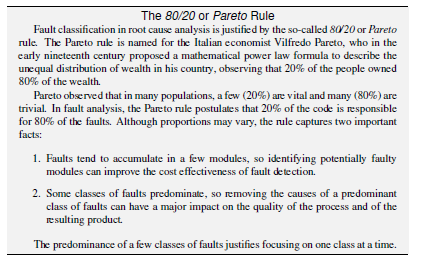


The faults occurs because of the above said reasons and this can be prevented by









**The Quality Team:**

* The quality plan must assign roles and responsibilities to people
* assignment of responsibility occurs at
  + strategic level
    - test and analysis strategy
    - structure of the organization
    - external requirements (e.g., certification agency)
  + tactical level

\* test and analysis plan

**Roles and Responsibilities   
at Tactical Level:**

* balance level of effort across time
* manage personal interactions
* ensure sufficient accountability that quality tasks are not easily overlooked
* encourage objective judgment of quality
* prevent it from being subverted by schedule pressure
* foster shared commitment to quality among all team members
* develop and communicate shared knowledge and values regarding quality

**Independent Testing Team:**