CHAPTER

1

Basic Aspects of Software Testing

Contents

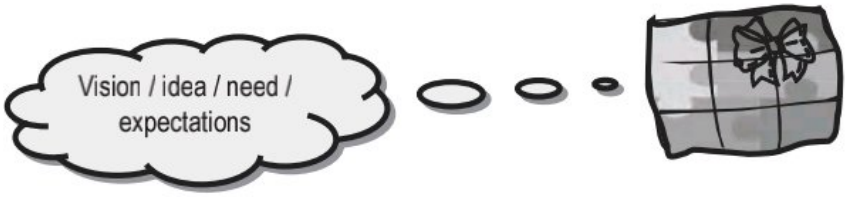
Testing is not an isolated activity, nor is it a development activity. Testing is a support activity: meaningless without the development processes and not producing anything in its own right: nothing developed entails nothing to test.

Testing is, however, a very important part of the life cycle of any product from the initial idea, during development, and in deployment until the product is taken out of deployment and disposed of.

Testing has it place intertwined with all these activities. Testing must find its place and fill it as well as possible.

1.1 Testing in the Software Life Cycle

The intention of product development is to somehow go from the vision of a product to the final product.



To do this a development project is usually established and carried out. The time from the initial idea for a product untill it is delivered is the development life cycle.

When the product is delivered, its real life begins. The product is in use or deployed until it is disposed of. The time from the initial idea for a product until it is disposed of is called the product life cycle, or software life cycle, if we focus on software products.

Testing is a necessary process in the development project, and testing is also necessary during deployment, both as an ongoing monitoring of how the product is behaving and in the case of maintenance (defect correction and possibly evolution of the product).

Testing fits into any development model and interfaces with all the other development processes, such as requirements definition and coding. Testing also interfaces with the processes we call supporting processes, such as, for example, project management.

Testing in a development life cycle is broken down into a number of test levels—for example component testing and system testing. Each test level has it own characteristics.

1.1.1 Development Models

Everything we do in life seems to follow a few common steps, namely: conceive, design, implement, and test (and possibly subsequent correction and retest).

The same activities are recognized in software development, though there are normally called:

- Requirements engineering;
- Design;
- Coding;
- Testing (possibly with retesting, and regression testing).

Building blocks

In software development we call the building blocks "stages," "steps," "phases," "levels," or "processes."

The way the development processes are structured is the development life cycle or the development model. A life cycle model is a specification of the order of the processes and the transition criteria for progressing from one process to the next, that is, completion criteria for the current process and entry criteria for the next.

Software development models provide guidance on the order in which the major processes in a project should be carried out, and define the conditions for progressing to the next process. Many software projects have experienced problems because they pursued their development without proper regard for the process and transition criteria.

The reason for using a software development model is to produce better quality software faster. That goal is equal for all models. *Using any model is better than not using a model.*

A number of software development models have been deployed throughout the industry over the years. They are usually grouped according to one of the following concepts:

- Sequential;
- Iterative;
- Incremental.

The building blocks—the processes—are the same; it is only a matter of their length and the frequency with which they are repeated.

The sequential model is characterized by including no repetition other than perhaps feedback to the preceding phase. This makeup is used in order to avoid expensive rework.

1.1.1.1 Sequential Models

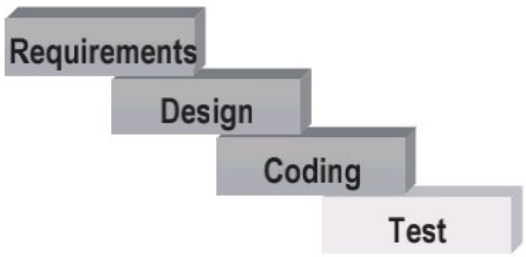
The *assumptions* for sequential models are:

- The customer knows what he or she wants.
- The requirements are frozen (changes are exceptions).
- Phase reviews are used as control and feedback points.

The characteristics of a successful sequential development project are:

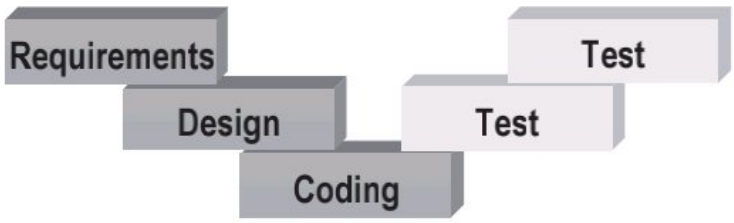
- Stable requirements;
- Stable environments;
- Focus on the big picture;
- One, monolithic delivery.

Historically the first type of sequential model was *the waterfall model*. A pure waterfall model consists of the building blocks ordered in one sequence with testing in the end.



The goals of the waterfall model are achieved by enforcing fully elaborated documents as phase completion criteria and formal approval of these (signatures) as entry criteria for the next.

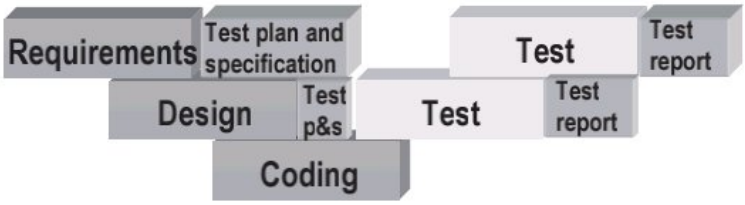
The V-model is an expansion of the pure waterfall model introducing more test levels, and the concept of testing not only being performed at the end of the development life cycle, even though it looks like it.



The V-model describes a course where the left side of the V reflects the processes to be performed in order to produce the pieces that make up the physical product, for example, the software code. The processes on the right side of the V are test levels to ensure that we get what we have specified as the product is assembled.

The pure V-model may lead you to believe that you develop first (the left side) and then test (the right side), but that is not how it is supposed to work.

A *W-model* has been developed to show that the test work, that is, the production of testing work products, starts as soon as the basis for the testing has been produced. Testing includes early planning and specification and test execution when the objects to test are ready. The idea in the V-model and the W-model is the same; they are just drawn differently.



When working like this, we describe what the product must do and how (in the requirements and the design), and at the same time we describe how we are going to test it (the test plan and the specification). This means that we are starting our testing at the earliest possible time.

The planning and specification of the test against the requirements should, for example, start as soon as the requirements have reached a reasonable state.

A W-model-like development model provides a number of advantages:

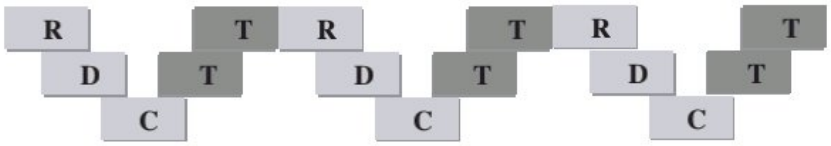
- More time to plan and specify the test
- Extra test-related review of documents and code
- More time to set up the test environment(s)
- Better chance of being ready for test execution as soon as something is ready to test

For some classes of software (e.g., safety critical systems, or fixed-price contracts), a W-model is the most appropriate.

1.1.1.2 Iterative and Incremental Models

In iterative and incremental models the strategy is that frequent changes should and will happen during development. To cater for this the basic processes are repeated in shorter circles, iterations. These models can be seen as a number of mini W-models; testing is and must be incorporated in every iteration within the development life cycle.

This is how we could illustrate an iterative or incremental development model.



The goals of an iterative model are achieved through various prototypes or subproducts. These are developed and validated in the iterations. At the end of each iteration an operational (sub)product is produced, and hence the product is expanding in each iteration. The direction of the evolution of the product is determined by the experiences with each (sub)product.

Note that the difference between the two model types discussed here is:

- In *iterative development* the product is not released to the customer until all the planned iterations have been completed.
- In *incremental development* a (sub)product is released to the customer after each iteration.

The *assumptions* for an iterative and incremental model are:

- The customer cannot express exactly what he or she wants.
- The requirements will change.
- Reviews are done continuously for control and feedback.

The characteristics of a successful project following such a model are:

- Fast and continuous customer feedback;
- Floating targets for the product;
- Focus on the most important features;
- Frequent releases.

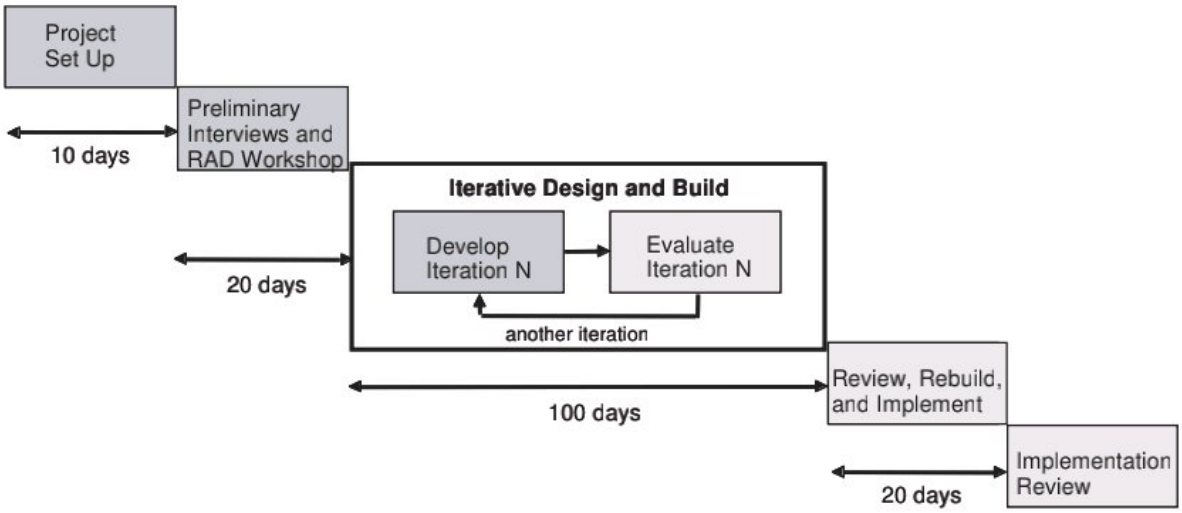
The iterative/incremental model matches situations in which the customers say: "I can't tell you what I want, but I'll know it when I see it"—the last part of the sentence often expressed as "IKIWISI."

These models are suited for a class of applications where there is a close and direct contact with the end user, and where requirements can only be established through actual operational experience.

A number of more specific iterative models are defined. Among these the most commonly used are the RAD model and the Spiral model.

The *RAD model* (Rapid Application Development) is named so because it is driven by the need for rapid reactions to changes in the market. James Martin, consultant and author, called the "guru of the information age", was the first to define this model. Since then the term RAD has more or less become a generic term for many different types of iterative models.

The original RAD model is based on development in timeboxes in few—usually three—iterations on the basis of fundamental understanding of the goal achieved before the iterations start. Each iteration basically follows a waterfall model.



When the last iteration is finished, the product is finalized and implemented as a proper working product to be delivered to the customer.

Barry Boehm, TRW Professor of Software Engineering at University of Southern California, has defined a so-called *Spiral Model*. This model aims at accommodating both the waterfall and the iterative model. The model consists of a set of full cycles of development, which successively refines the knowledge about the future product. Each cycle is risk driven and uses prototypes and simulations to evaluate alternatives and resolve risks while producing work products. Each cycle concludes with reviews and approvals of fully elaborated documents before the next cycle is initiated.

The last cycle, when all risks have been uncovered and the requirements, product design, and detailed design approved, consists of a conventional waterfall development of the product.

In recent years a number of incremental models, called *evolutionary* or *agile development models*, have appeared. In these models the emphasis is placed on values and principles, as described in the "Manifesto of Software Development." These are:

- Individuals and interactions are valued over processes and tools
- Working software is valued over comprehensive documentation
- Customer collaboration is valued over contract negotiation
- Responding to change is valued over following a plan

One popular example of these models is the eXtreme Programming model, (XP). In XP one of the principles is that the tests for the product are developed first; the development is test-driven.

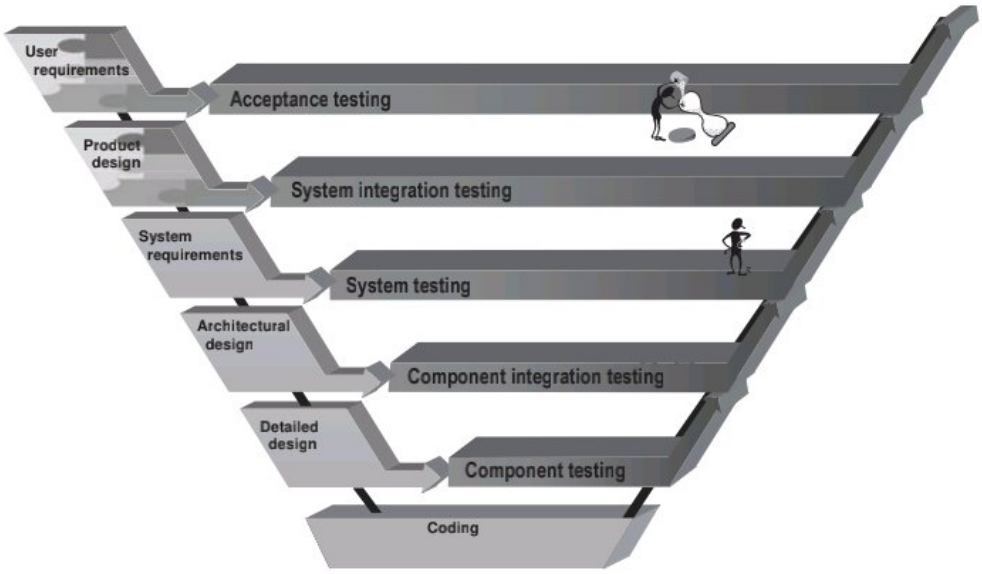
The development is carried out in a loosely structured small-team style. The objective is to get small teams (3–8 persons) to work together to build products quickly while still allowing individual programmers and teams freedom to evolve their designs and operate nearly autonomously.

These small teams evolve features and whole products incrementally while introducing new concepts and technologies along the way. However, because developers are free to innovate as they go along, they must synchronize frequently so product components all work together.

Testing is perhaps even more important in iterative and incremental development than in sequential development. The product is constantly evolved and extensive regression testing of what has previously been agreed and accepted is imperative in every iteration.

1.1.2 Dynamic Test Levels

In the V-model, and hence in basically all development, each development process has a corresponding dynamic test level as shown here.



The V-model used here includes the following dynamic test levels:

- Acceptance testing—based on and testing the fulfillment of the user requirements;
- System testing—based on and testing the fulfillment of the (software) system requirements;
- Component integration testing—based on and testing the implementation of the architectural design;
- Component testing—based on and testing the implementation of the detailed design.

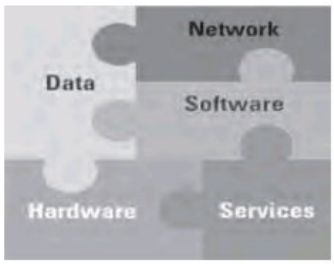
Note that coding does not have a corresponding test level; it is not a specification phase, where expectations are expressed, but actual manufacturing! The code becomes the test object in the dynamic test levels.

There is no standard V-model. The V-model is a principle, not a fixed model. Each organization will have to define its own so that it fits with the nature of the products and the organization. The models can have different make-ups, that is there may be more or less specification phases on the left side and hence testing levels on the right side, and/or the phases and levels may be named differently in different organizations.

In cases where the final product is part of or in itself a complex product it is necessary to consider more integration test levels.

In the case of a system of systems development project, described in Section 1.2.1, we will need a system integration test level.

Sometimes the product we are developing consists of a number of different types of systems, like for example:



- Software
- Hardware
- Network
- Data
- Services

In such cases there will be product design specification phases to distribute the requirements on the different systems in the beginning of the development life cycle and we will therefore need more or more integration test levels, such as, for example, hardware-software system integration and software-data system integration.

Note: the puzzle does NOT indicate possible interfaces between systems, only the fact that a product may be made up of different types of systems.

We could also be producing a product that is going to interface with system(s) the customer already has running. This will require a customer product integration test level.

No matter how many test levels we have, each test level is different from the others, especially in terms of goals and scope.

The organizational management must provide test strategies specific to each of the levels for the project types in the organization in which the testing is anchored. The contents of a test level strategy are discussed in Section 3.2.2.

Based on this the test responsible must produce test plans specific for each test level for a specific project. The contents of a test plan are discussed in Section 3.2.3.

The specific test plans for the test levels for a specific project should outline the differences between the test levels based on the goals and scope for each.

The fundamental test process is applicable for all the test levels. The test process is described in detail in Chapter 2.

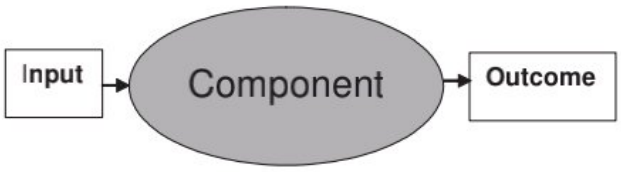
The dynamic test levels in the V-model used here are discussed next.

1.1.2.1 Component Testing

Component testing is the last test level where work, that is, planning, can start, the first where test execution can start, and therefore also the first to be finished.

The *goal* is to find defects in the implementation of each component according to the detailed design of the component.

The test object is hence individual components in isolation, and the basis documentation is the detailed design, and sometimes also other documentation like the requirements specification.



It is not always easy to agree on what a component is. A component could be what is contained in a compilable file, a subroutine, a class, or … the possibilities are legion. The important thing in an organization is to define "a component"—it is less important what a component is defined as.

The scope for one component test is the individual component and the full scope of the component testing could be all components specified in the design, though sometimes only the most critical components may be selected for component testing.

An overall *component test plan* should be produced specifying the order in which the testing of the components is to take place. If this is done sufficiently early in the development, we as testers may be able to influence the development order to get the most critical components ready to test first. We also need to consider the subsequent component integration testing, and plan for components with critical interfaces to be tested first. For each component a very short plan (who, when, where, and completion criteria) and a test specification should be produced.

The assignment of the responsibility for the component testing depends on the level of independence we need. The lowest level of independence is where the manufacturer—here the developer—tests his or her own product. This often happens in component testing. The next level of independence is where a colleague tests his or her colleague's product. This is advisable for component testing. The level of independence to use is guided by the risk related to the product. Risk management is discussed in Section 3.5.

The *techniques* to use in component testing are functional (black-box) techniques and structural (white-box) techniques. Most often tests are first designed using functional techniques. The coverage is measured and more tests can be designed using structural techniques if the coverage is not sufficient to meet the completion criteria.

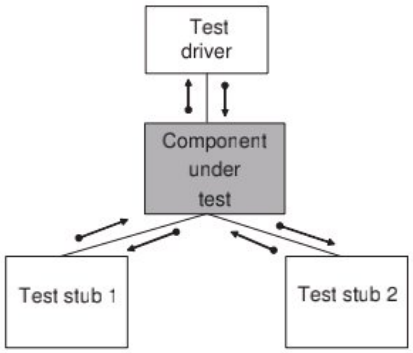
The code must never be used as the basis documentation from which to derive the expected results.

Nonfunctional requirements or characteristics, such as memory usage,

defect handling, and maintainability may also be tested at the component testing level.

To isolate a component, it is necessary to have a driver to be able to execute the component. It is also usually necessary to have a stub or a simulator to mimic or simulate other components that interface with the component under test. Test stubs are sometimes referred to as test harness.

The needs for test drivers and stubs must be specified as part of the specification of the test environment. Any needed driver and stubs must, of course, be ready before each individual component testing can start.



Many tools support component testing. Some are language-specific and can act as drivers and stubs to facilitate the isolation of the component under test. Tools are discussed in Chapter 9.

Component *test execution* should start when the component has been deemed ready by the developer, that is when it fulfilles the entry criteria. The least we require before the test execution can start is that the component can compile. It could be a very good idea to require that a static test and/or static analysis has been performed and approved on the code as entry criteria for the component test execution.

Measures of time spent on the testing activities, on defects found and corrected, and on obtained coverage should be collected. This is sometimes difficult because component testing is often performed as an integrated development/testing/debugging activity with no registration of defects and very little if any reporting. This is a shame because it deprives the organization of valuable information about which kinds of defects are found and hence the possibility for introducing relevant process improvement.

The component testing for each individual component must stop when the completion criteria specified in the plan have been met. For each component a very short summary report should be produced.

A summary *report* for the collection of components being tested should be produced when the testing has been completed for the last component.

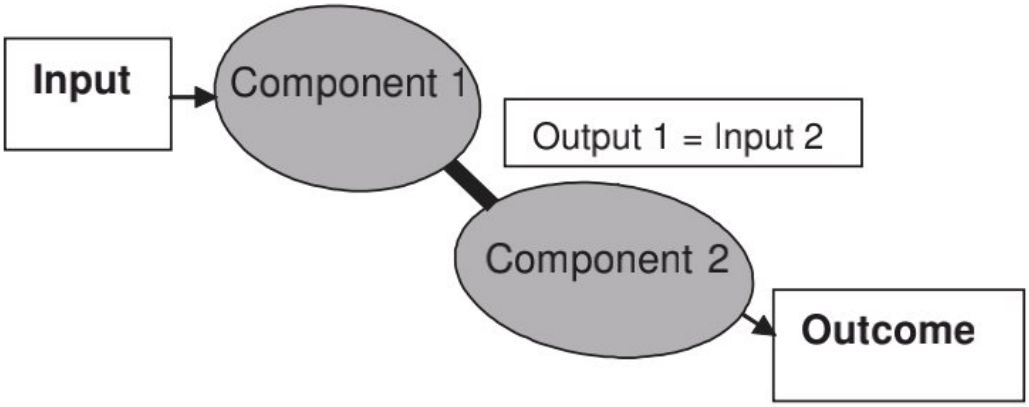
Any *test procedures should be kept*, because they can be very useful for later confirmation testing and regression testing. Drivers and stubs should be kept for the same reason, and because they can be useful during integration testing as well.

The goals of integration testing are to find defects in the interfaces and invariants between interacting entities that interact in a system or a product. Invariants are substates that should be unchanged by the interaction between two entities.

1.1.2.2 Integration Testing

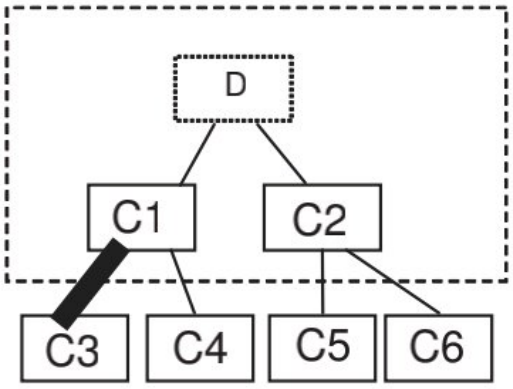
The objective is not to find defects inside the entities being integrated—the assumption being that these have already been found during previous testing.

The entities to integrate may be components as defined in the architectural design or different systems as defined in the product design. The principles for integration testing are the same no matter what we are integrating.



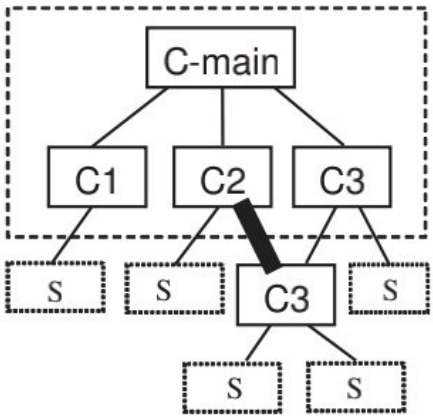
For the collection of interfaces to test an overall integration test plan should be produced specifying, among other things, the order in which this testing is to take place. There are four different strategies for the testing order in integration testing:

- Top down;
- Bottom up;
- Functional integration;
- Big-bang.



In top-down integration the interfaces in the top layer in the design hierarchy are tested first, followed by each layer going downwards. The main program serves as the driver.

This way we quickly get a "shell" created. The drawback is that we (often) need a large number of stubs.



In bottom-up integration the interfaces in the lowest level are tested first. Here higher components are replaced with drivers, so we may need many drivers. This integration strategy enables early integration with hardware, where this is relevant.

In functional integration we integrate by functionality area; this is a sort of vertically divided top-down strategy. We quickly get the possibility of having functional areas available.

In big-bang integration we integrate most or everything in one go. At first glance it seems like this strategy reduces the test effort, but it does not—on the contrary. It is impossible to get proper coverage when testing the interfaces in a big-bang integration, and it is very difficult to find any defects in the interfaces, like looking for a needle in a haystack. Both top-down and bottom-up integration often end up as big-bang, even if this was not the initial intention.

For each interface a *very short plan* (who, when, where, and completion criteria) and a test specification should be produced. Often one of the producers of the entities to integrate has the responsibility for that integration testing, though both should be present.

Both the formality and the level of independence is higher for system integration testing than for component integration, but these issues should not be ignored for component integration testing.

The *techniques* to use must primarily be selected among the structural techniques, depending on the completion criteria defined in the plan. Nonfunctional requirements or characteristics, such as performance, may also be tested at the integration testing level.

The necessary drivers or stubs must be specified as part of the environment and developed before the integration testing can start. Often stubs from a previous test level, for example, component testing, can be reused.

The *execution* of the integration testing follows the completion of the testing of the entities to integrate. As soon as two interacting entities are tested, their integration test can be executed. There is no need to wait for all entities to be tested individually before the integration test execution can begin.

Measures of time spent on the testing, on defects found and corrected, and on coverage should be collected.

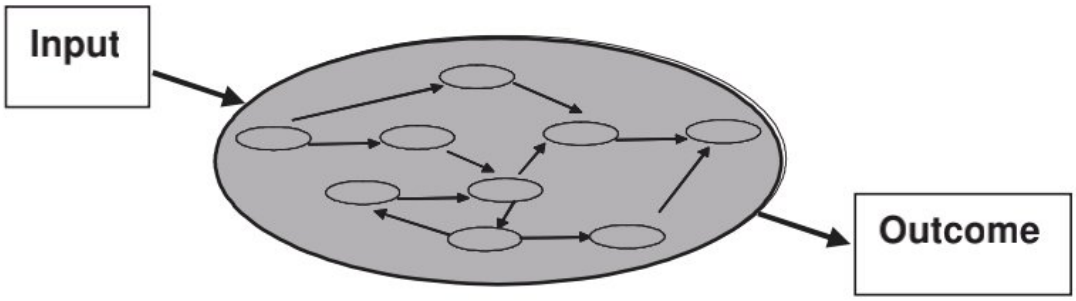
The integration testing for each individual interface must stop when the completion criteria specified in the plan have been met. A very short test report should be produced for each interface being tested. We must keep on integrating and testing the interfaces and the invariants until all the entities

have been integrated and the overall completion criteria defined in the integration test plan have been met.

A summary report for the collection of interfaces being tested should be produced when the testing has been completed for the last interface.

1.1.2.3 System Testing

The goal of system testing is to find defects in features of the system compared to the way it has been defined in the software system requirements. The test object is the fully integrated system.



The better the component testing and the component integration testing has been performed prior to the system testing, the more effective is the system testing. All too often system testing is impeded by poor or missing component and component integration testing.

A comprehensive system test plan and system test specification must be produced.

The system test specification is based on the system requirements specification. This is where all the expectations, both the functional and the non-functional should be expressed. The functional requirements express what the system shall be able to do—the functionality of the system. The non-functional requirements express how the functionality presents itself and behaves. In principle the system testing of the two types of requirements is identical. We test to get information about the fulfillment of the requirements.

The techniques to use will most often be selected among the functional techniques, possibly supplemented with experience-based techniques (exploratory testing, for example), depending on the completion criteria defined in the plan. *Experience-based test techniques should never be the only techniques used in the system testing.*

The *execution* of system test follows the completion of the entire component integration testing. It is a good idea to also require that a static test

has been performed on the requirements specification and on the system test specification before execution starts.

Many tools support system testing. Capture/replay tools and test management tools are especially useful to support the system testing.

Measures of time spent on the testing, on faults found and corrected, and on coverage should be collected. The system testing must stop when the completion criteria specified in the plan have been met.

A system test report should be produced when the system testing has been completed.

1.1.2.4 Acceptance Testing

The acceptance testing is the queen's inspection of the guard. The goal of this test level is not, like for all the other ones, to find defects by getting the product to fail. At the acceptance test level the product is expected to be working and it is presented for acceptance.

The customer and/or end users must be involved in the acceptance testing. In some cases they have the full responsibility for this testing; in other cases they just witness the performance.

In the acceptance testing the test object is the entire product. That could include:

- Business processes in connection with the new system;
- Manual operations;
- Forms, reports, and so forth;
- Document flow;
- Use cases and/or scenarios.

The techniques are usually mostly experience-based, where the future users apply their domain knowledge and (hopefully) testing skills to the validation of the product. Extracts of the system test specification are sometimes used as part of the acceptance test specification.

An extra benefit of having representatives of the users involved in the acceptance testing is that it gives these users a detailed understanding of the new system—it can help create ambassadors for the product when it is brought into production.

There may be a number of acceptance test types, namely:

- Contract acceptance test;
- Alpha test;
- Beta test.

The contract acceptance test may also be called factory acceptance test. This test must be completed before the product may leave the supplier; the product has to be accepted by the customer. It requires that clear acceptance criteria have been defined in the contract. A thorough registration of the results is necessary as evidence of what the customer acceptance is based on.

An alpha test is usage of the product by representative users at the development site, but reflecting what the real usage will be like. Developers must not be present, but extended support must be provided. The alpha test is not used particularly often since it can be very expensive to establish a "real" environment. The benefits rarely match the cost.

A beta test is usage of the product by selected (or voluntary) customers at the customer site. The product is used as it will be in production. The actual conditions determine the contents of the test. Also here extended support of the users is necessary. Beta tests preferably run over a longer period of time. Beta tests are much used for off-the-shelf products—the customers get the product early (and possibly cheaper) in return for accepting a certain amount of immaturity and the responsibility for reporting all incidents.

1.1.3 Supporting Processes

No matter how the development model is structured there will always be a number of supporting activities, or supporting processes, for the development.

The primary supporting processes are:

- Quality assurance;
- Project management;
- Configuration management.

These processes are performed during the entire course of the development and support the development from idea to product.

Other supporting processes may be:

- Technical writing (i.e., production of technical documentation);
- Technical support (i.e., support of environment including tools).

The supporting processes all interface with the test process.

Testing is a product quality assurance activity and hence actually part of the supporting processes. This is in line with the fact that testing is meaningless without the development processes and not producing anything in its own right: nothing developed entails nothing to test.

The test material, however, is itself subject to quality assurance or testing, so testing is recursive and interfaces with itself. Testing also interfaces with

project management and configuration management as discussed in detail in the following.

Testing also interfaces with technical writing. The documentation being written is an integrated part of the final product to be delivered and must therefore also be subject to quality assurance (i.e., to static testing).

When the product—or an increment—is deployed, it transfers to the maintenance phase. In this phase corrections and possibly new features will be delivered at defined intervals, and testing plays an important part here.

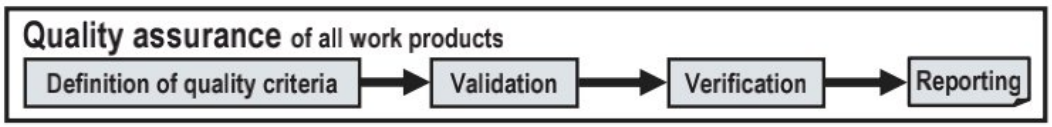
1.1.3.1 Product Quality Assurance

It is not possible to test quality into a product when the development is close to being finished. The quality assurance activities must start early and become an integrated part of the entire development project and the mindset of all stakeholders.

Quality assurance comprises four activities:

- Definition of quality criteria
- Validation
- Verification
- Quality reporting

Note that the validation is not necessarily performed before the verification; in many organizations it is the other way around, or in parallel.



First of all, the *Quality criteria* must be defined. These criteria are the expression of the quality level that must be reached or an expression of "what is sufficiently good." These criteria can be very different from product to product. They depend on the business needs and the product type. Different quality criteria will be set for a product that will just be thrown away when it is not working than for a product that is expected to work for many years with a great risk of serious consequences if it does not work.

There are two quality assurance activities for checking if the quality criteria have been met by the object under testing, namely:

- Validation;
- Verification.

They have different goals and different techniques. The object to test is delivered for validation and verification from the applicable development process.

Validation is the assessment of the correctness of the product (the object) in relation to the users' needs and requirements.

You could also say that validation answers the question: *"Are we building the correct product?"*

Validation must determine if the customer's needs and requirements are correctly captured and correctly expressed and understood. We must also determine if what is delivered reflects these needs and requirements.

When the requirements have been agreed upon and approved, we must ensure that during the entire development life cycle:

- Nothing has been forgotten.
- Nothing has been added.

It is obvious that if something is forgotten, the correct product has not been delivered. Is does, however, happen all too often, that requirements are overlooked somewhere in the development process. This costs money, time, and credibility.

On the surface it is perhaps not so bad if something has been added. But it does cost money and affect the project plan, when a developer—probably in all goodwill—adds some functionality, which he or she imagines would be a benefit for the end user.

What is worse is that *the extra functionality will probably never be tested* in the system and acceptance test, simply because the testers don't know anything about its existence. This means that the product is sent out to the customers with some untested functionality and this will lie as a mine under the surface of the product. Maybe it will never be hit, or maybe it will be hit, and in that case the consequences are unforeseeable.

The possibility that the extra functionality will never be hit is, however, rather high, since the end user will probably not know about it anyway.

Validation during the development process is performed by analysis of trace information. If requirements are traced to design and code it is an easy task to find out if some requirements are not fulfilled, or if some design or code is not based on requirements.

The ultimate validation is the user acceptance test, where the users test that the original requirements are implemented and that the product fulfills its purpose.

Verification, the other quality assurance activity, is the assessment of whether the object fulfills the specified requirements.

Verification answers the question: *"Are we building the product correctly?"*

The difference between validation and verification can be illustrated like this:

Validation confirms that a required calculation of discount has been designed and coded in the product.
Verification confirms that the implemented algorithm calculates the discount as it is supposed to in all details.

A number of techniques exist for verification. The ones to choose depend on the test object.

In the early phases the test object is usually a document, for example in the form of:

- Plans;
- Requirements specification;
- Design;
- Test specifications;
- Code.

The verification techniques for these are the static test techniques discussed in Chapter 6:

- Inspection;
- Review (informal, peer, technical, management);
- Walk-through.

Once some code has been produced, we can use static analysis on the code as a verification technique. This is not executing the code, but verifying that it is written according to coding standards and that is does not have obvious data flow faults. Finally, dynamic testing where the test object is executable software can be used.

We can also use dynamic analysis, especially during component testing. This technique reveals faults that are otherwise very difficult to identify. Dynamic analysis is described in Section 4.6.

A little memory hint:

Validation

Correct

a comes before e
t comes before y

Verification

Correctly

Quality assurance reports on the findings and results should be produced.

If the test object is not found to live up to the quality criteria, the object is returned to development for correction. At the same time incident reports should be filled in and given to the right authority.

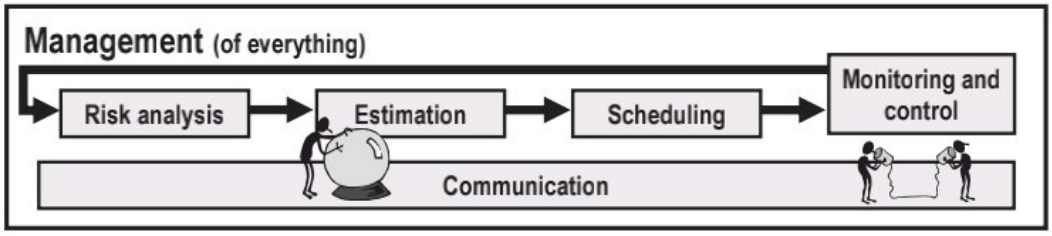
Once the test object has passed the validation and verification, it should be placed under configuration management.

1.1.3.2 Project Management

It is obviously important that the development process and the supporting processes are managed and controlled during the entire project. Project management is the supporting process that takes care of this, from the first idea to the release.

The most important activities in project management are:

- Risk analysis;
- Estimation;
- Scheduling;
- Monitoring and control;
- Communication.



Test management is subordinated to project management.

The estimation, risk analysis, and scheduling of the test activities will either have to be done in cooperation with the project management or by the test manager and consolidated with the overall project planning. The results of the monitoring and control of the test activities will also have to be coordinated with the project management activities.

The project management activities will not be discussed further here.

The corresponding test management activities are described in detail in Chapter 3.

1.1.3.3 Configuration Management

Configuration management is another supporting process with which testing interacts. The purpose of configuration management is to establish and maintain integrity of work products and product.

Configuration management can be defined as:

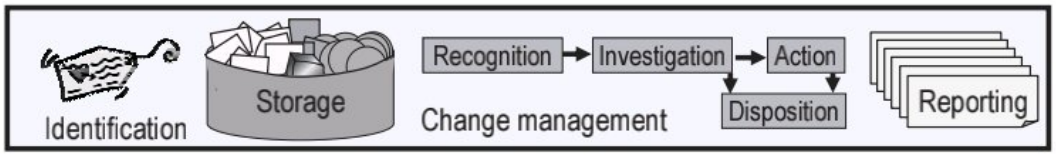
- Unique identification;
- Controlled storage;
- Change management (recognition, investigation, action, and disposition);
- Status reporting.

for selected

- Work products;
- Product components;
- Products.

during the entire life time of the product.

An object under configuration management is called a configuration item.



The purpose of *identification* is to uniquely identify each configuration item and to specify its relations to the outside world and to other configuration items. Identification is one of the cornerstones of configuration management, as it is impossible to control something for which you don't know the identity.

Each organization must define the conventions for unique identification of the configuration items.

Test cases
10.3.1.6 (80) Test for correct bank identity number 1.A
The identification encompasses:
Current section number in document: 10.3.1.6
Running unique number: 80
Version of test case: 1.A

The purpose of *storage* is to ensure that configuration items don't disappear or are damaged. It must also be possible to find the items at any time and have them delivered in the condition in which we expect to find them.

Storage is something physical. Items that are stored are physically present at a specific place. This place is often called the library, or the controlled library.

Configuration items are released from storage to be used as the basis for further work. *Usage is all imaginable deployment of configuration items without these being changed*, not just usage of the final product by the final users.

Usage may for instance be a review, if a document is placed under configuration management in the form of a draft and subsequently has to be reviewed.

It may be testing of larger or minor parts of the system, integration of a subcomponent into a larger component, or proper operation or sale of a finished product.

Configuration items released from storage must not be changed, ever! But new versions may be issued as the result of change control.

The purpose of *change management or change control* is to be fully in control of all change requests for a product and of all implemented changes. Any change should be traced to the configuration item where the change has been implemented.

The initiation of change control is the occurrence of an incident. Incident management is discussed in Chapter 7.

The purpose of *status reporting* is to make available the information necessary for effective management of the development, testing, and maintenance of a product, in a useful and readable way.

Configuration management can be a well of information.

A few words about the concept of a *configuration item* are needed here. In principle everything may be placed under configuration management. The following list shows what objects may become configuration items, with the emphasis on the test ware.

- Test material: Test specifications, test data(base), drivers, and stubs
- Environments: Operating systems, tools, compilers, and linkers

- Technical documentation: Requirements, design, and technical notes
- Code: Source code, header files, include files and system libraries
- Project documentation: User manuals, build scripts, data, event registrations, installation procedures, and plans
- Administrative documents: Letters, contracts, process description, sales material, templates, and standards
- Hardware: Cables, mainframe, PC, workstation, network, storage, and peripherals

1.1.3.4 Technical Writing

Technical writing is a support process much used in the United Kingdom. Other European countries do not use technical writers that much—here the developers, testers, and the rest of the project team are left to their own devices.

Technical writers are people with special writing skills and education. They assist other staff members when difficult issues need to be made clear to the intended audience in writing.

We as testers interface with technical writers in two ways:

- We subject their work to static tests.
- We use their work in our testing.

We can of course also use their skills as writers, but that does not happen very often. Testers usually write for other testers and for a technical audience.

1.2 Product Paradigms

The use of computers to assist people in performing tasks has developed dramatically since the first huge (in physical size) computers were invented around the middle of the last century. The first computers were about the size of a family home and you could only interact with them via punch cards or tape and printed output. Those were the days.

Today we as testers may have to cope with a number of different product types or product paradigms, and with different development paradigms and coding languages. Not all of us encounter all of them, but it is worth knowing a little bit about the challenges they each pose for us.

We always need to be aware of the product and development paradigm used for the (testing) projects we are involved in. We must tailor our test approach and detailed test processes to the circumstances and be prepared to tackle any specific obstacles caused by these as early as possible.

A few significant product paradigms are discussed here.

1.2.1 Systems of Systems

A system of systems is the concept of integrating existing systems into a single information system with only limited new development. The concept was first formulated by the American Admiral Owens in his book *Lifting the Fog of War*. The concept was primarily used for military systems but is spreading more and more to civilian systems as well.

The idea is to use modern network technologies to exploit the information spread out in a number of systems by combining and analyzing it and using the results in the same or other systems to make these even more powerful.

A tiny example of a system of systems is a sprinkling system at a golf course. The gardener can set the sprinkling rate for a week at the time. Using a network connection this system is linked to a system at the meteorological institute where hours of sunshine, average temperatures, and rainfall are collected. This information is sent to a small new system, which calculates the needed sprinkling rate on a daily basis and feeds this into the sprinkling system automatically. The gardener's time, water, the occasional flooding, and the occasional drying out of the green is saved.

Systems of systems are complicated in nature. The final system is usually large and complex as each of the individual systems may be. Each of the individual systems may in itself consist of a number of different subsystems, such as software, hardware, network, documentation, data, data repository systems, license agreements, services (e.g., courses and upgrades), and descriptions of manual processes. Few modern systems are pure software products, though they do exist.

Even if the individual systems are not developed from scratch these systems pose high demands on supporting the supporting processes, especially project management, but also configuration management and product quality assurance. In the cases where some or all of the individual systems are being developed as part of the construction of a system of systems this poses even higher demands in terms of communication and coordination.

From a testing point of view, there are at least three important aspects to take into account when working with systems of systems:

- System testing of the individual systems
- Integration testing of systems
- Regression testing of systems and integration

A system of systems is only as strong as the weakest link, and the completion criteria for the system testing of each individual system must reflect the quality expectations toward the complete system of systems. The system testing of each of the individual systems is either performed as part of the project, or assurance of its performance must be produced, for example in the form of test reports from the producer.

Systems of systems vary significantly in complexity and may be designed in hierarchies of different depths, ranging from a two-layer system where the final system of systems is composed of a number of systems of the same "rank" to many-layered (system of (systems of (systems of systems))). Integration of the systems must be planned and executed according to the overall architecture, observing the integration testing aspects discussed in Section 1.1.2.

It is inevitable that defects will be found during system and integration testing of systems of systems, and significant iterations of defect correction, confirmation testing, and not least regression testing must be anticipated and planned for. *Strict defect handling is necessary to keep this from getting out of control,* resulting, for example, in endless correction and recorrection circles.

Systems of systems may well contain systems of types where special care and considerations need to be made for testing. Examples may be:

- Safety-critical systems
- Large mainframe systems
- Client-server systems
- Web-based systems
- PC-based systems
- Web-based systems
- Embedded systems
- Real-time systems
- Object-oriented development

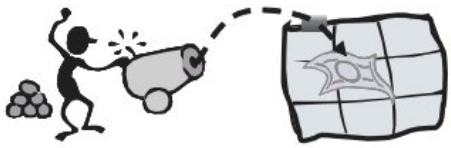
1.2.2 Safety-Critical Systems

Any system presents some risk to its owners, users, and environment. Some present more than others, and those that present the most risk are what we call safety-critical systems.

The risk is a threat to something valuable. All systems either have something of value, which may be jeopardized, inside them, or their usage may jeopardize some value outside them. A system should be built to protect the values both from the result of ordinary use of the system and from the result of malicious attacks of various kinds.

A typical categorization of values looks at values concerning:

- Safety
- Economy
- Security
- Environment



Many regulatory standards address how to determine the safety criticality of systems and provide guidelines for the corresponding testing. Some of them (but probably not all) are:

- CEI/IEC 61508—Functional safety of electrical/electronic/programmable safety-related systems
- DO-178-B—Software considerations in airborne systems and equipment certification
- pr EN 50128—Software for railway control and protection systems
- Def Stan 00-55—Requirements for safety-related software in defense equipment
- IEC 880—Software for computers in the safety systems of nuclear power stations
- MISRA—Development guidelines for vehicle-based software
- FDA—American Food and Drug Association (pharmaceutical standards)

The standards are application-specific, and that can make it difficult to determine what to do if we have to do with multidisciplinary products. Nonetheless, standards do provide useful guidance. The most generic of the standards listed above is IEC 61508; this may always be used if a system does not fit into any of the other types.

All the standards operate with so-called software integrity levels (SILs).

This table shows an example of a classification.

Ex.

SIL / Value	A (100.000.000)	B (100.000)	C (100)	D (1)
Safety	Many people killed	Human lives in danger	Damage to physical objects; risk of personal injury	Insignificant damage to things; no risk to people
Economy	Financial catastrophe (the company must close)	Great financial loss (the company is threatened)	Significant financial loss (the company is affected)	Insignificant financial loss
Security	Destruction/disclosure of strategic data and services	Destruction/ disclosure of critical data and services	Faults in data	No risk for data
Environment	Comprehensive and irreparable damage to the environment	Reparable, but comprehensive damage to the environment	Local damage to the environment	No environmental risk

The concept of SILs allows a standard to define a hierarchy of levels of testing (and development). A SIL is normally applied to a subsystem; that is, we can operate with various degrees of SILs within a single system, or within a system of systems. The determination of the SIL for a system under testing is based on a risk analysis.

The standards concerning safety critical systems deal with both development processes and supporting processes, that is, project management, configuration management, and product quality assurance.

Ex.

We take as an example the CEI/IEC 61508 recommends the usage of test case design techniques depending on the SIL of a system. This standard defines four integrity levels: SIL4, SIL3, SIL 2, and SIL1, where SIL4 is the most critical.

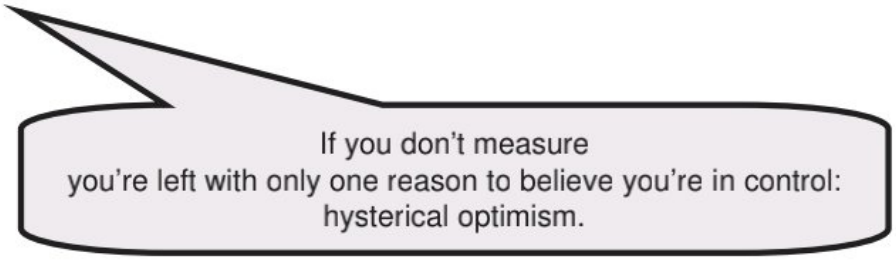
For a SIL4-classified system, the standard says that the use of equivalence partitioning is highly recommended as part of the functional testing. Furthermore the use of boundary value analysis is highly recommended, while the use of cause-effect graph and error guessing are only recommended. For white-box testing the level of coverage is highly recommended, though the standard does not say which level of which coverage.

The recommendations are less and less strict as we come down the SILs in the standard.

For highly safety-critical systems the testers may be required to deliver a compliance statement or matrix, explaining how the pertaining regulations have be follow and fulfilled.

1.3 Metrics and Measurement

Tom De Marco, one of the testing gurus, once said:



One of the principles of good planning, both of testing and anything else, is to define specific and measurable goals for the activities. But it is not enough for goals to be measurable; we must also collect facts that can tell us if we have indeed achieved the goals.

1.3.1 Measuring in General

For facts or data collection we operate with the following concepts:

- Metric—A definition of what to measure, including data type, scale, and unit
- Measuring method—The description of how we are going to get the data
- Measurements—The actual values collected for the metrics

An example could be that the metric for the size of a book is "number of pages"; the measuring method is to "look at the last page number"; and the measurement for *Alice in Wonderland,* ISBN 7409746, is "54."

It is a good idea to establish a measurement plan as part of the project plan or master test plan. This should specify the metrics we want to measure and the measuring methods, who is going to measure, and perhaps most importantly: how the measurements will be analyzed and used.

Our measurements are derived from raw data such as time sheets, incident reports, test logs, and work sheets. *Direct measurements* are measurements we get directly from the raw data, for example, by counting the number of log sheets for passed test procedures and counting the number of incident reports. *Indirect measurements* are measurements we can calculate from direct measurements.

Most direct measurements have no meaning unless they are placed in relation to something. Number of incidents as such—for example, 50—says nothing about the product or the processes. But if we calculate the defects found compared to the estimated amount of defects it gives a much better indication—either of our estimation or of the quality of the product!

It is a common mistake to think that only objective data should be used. Objective data is what you can measure independently of human opinions. But even though subjective data has an element of uncertainty about it, it can be very valuable. Often subjective data is even cheaper to collect than objective data.

A subjective metric could be:

The opinion of the participants in walk-throughs concerning the usefulness of the walk-through activity on a scale from 1 to 5, where 1 is lowest and 5 is highest.

This is easy to collect and handle, and it gives a good indication of the perception of the usefulness of walk-throughs.

The metrics should be specified based on the goals we have set and other questions we would like to get answers to, such as how far we are performing a specific task in relation to the plan and expectations.

1.3.2 Test-Related Metrics

Many, many measurements can be collected during the performance of the test procedures (and any other process for that matter). They can be divided into groups according to the possibilities for control they provide. The groups and a few examples of direct measurements are listed here for inspirational purposes; the lists are by no means exhaustive.

- **Measurements about progress**
  - Of test planning and monitoring:
    - Tasks commenced
    - Task completed
  - Of test development:
    - Number of specified and approved test procedures
    - Relevant coverages achieved in the specification, for example, for code structures, requirements, risks, business processes
    - Other tasks completed
  - Of test execution and reporting:
    - Number of executed test procedures (or initiated test procedures)
    - Number of passed test procedures

- Number of passed confirmation tests
- Number of test procedures run as regression testing
- Other tasks completed
- Of test closure:
  - Tasks completed

For each of these groups we can collect measurements for:

- Time spent on specific tasks both in actual working hours and elapsed time
- Cost both from time spent and from direct cost, such as license fees

- **Measurements about coverage:**
  - Number of coverage elements covered by the executed test procedures code structures covered by the test

- **Measurements about incidents:**
  - Number of reported incidents
  - Number of incidents of different classes, for example, faults, misunderstandings, and enhancement requests
  - Number of defects reported to have been corrected
  - Number of closed incident reports

- **Measurements about confidence:**
  - Subjective statements about confidence from different stakeholders

All these measurements should be taken at various points in time, and the time of the measuring should be noted to enable follow-up on the development of topics over time, for example the development in the number of open incident reports on a weekly basis.

It is equally important to prepare to be able to measure and report status and progress of tasks and other topics in relation to milestones defined in the development model we are following.

To be able to see the development of topics in relation to expectations, corresponding to factual and/or estimated total numbers are also needed. A few examples are:

- Total number of defined test procedures
- Total number of coverage elements
- Total number of failures and defects
- Actual test object attributes, for example, size and complexity
- Planned duration and effort for tasks
- Planned cost of performing tasks

1.3.3 Analysis and Presentation of Measurements

It is never enough to just collect measurements. They must be presented and analyzed to be of real value to us. The analysis and presentation of measurements are discussed in Section 3.4.2.

1.3.4 Planning Measuring

It is important that stakeholders agree to the definition of the metrics and measuring methods, before any measurements are collected. Unpopular or adverse measurements may cause friction, especially if these basic definitions are not clear and approved. You can obtain very weird behaviors by introducing measurements!

There is some advice you should keep in mind when you plan the data you are going to collect. You need to aim for:

- **Agreed metrics**—Definitions (for example, what is a line of code), scale (for example, is 1 highest or lowest), and units (for example, seconds or hours) must be agreed on and understood
- **Needed measures**—What is it you want to know, to monitor, and to control?
- **Repeatable measurements**—Same time of measure and same in strument must give the same measurement
- **Precise measurements**—Valid scale and known source must be used
- **Comparable measurements**—For example, over time or between sources
- **Economical measurements**—Practical to collect and analyse com pared to the value of the analysis results
- **Creating confidentiality**—Never use measurements to punish or award individuals
- **Using already existing measurements**—Maybe the measurements just need to be analyzed in a new way
- **Having a measurement plan**—The plan should outline what, by whom, when, how, why
- **Using the measurements**—Only measure what can be used immediately and give quick and precise feedback

Questions

1. What is the development life cycle and the product life cycle?
2. What are the building blocks in software development models?
3. What are the basic development model types?