Available online at www.sciencedirect.com
SCIENCE DIRECT®
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 148 (2006) 89–111
www.elsevier.com/locate/entcs

An Introduction to Software Testing

Luciano Baresi[1]

Dipartimento di Elettronica e Informazione
Politecnico di Milano
Milano, Italy

Mauro Pezzè[2]

Dipartimento di Informatica, Sistemistica e Comunicazione
Universitá degli Studi di Milano Bicocca
Milano, Italy

Abstract

The development of large software systems is a complex and error prone process. Faults might occur at any development stage and they must be identified and removed as early as possible to stop their propagation and reduce verification costs. Quality engineers must be involved in the development process since the very early phases to identify required qualities and estimate their impact on the development process. Their tasks span over the whole development cycle and go beyond the product deployment through maintenance and post mortem analysis. Developing and enacting an effective quality process is not a simple task, but it requires that we integrate many quality-related activities with product characteristics, process organization, available resources and skills, and budget constraints.
This paper discusses the main characteristics of a good quality process, then surveys the key testing phases and presents modern functional and model-based testing approaches.

Keywords: Software Quality, Software Testing, Integration Testing, System and Acceptance Testing, Functional Testing, Model-based Testing.

1 Introduction

The development of large software products involves many activities that need to be suitably coordinated to meet the desired requirements. Among these

[1] Email:baresi@elet.polimi.it
[2] Email: pezze@disco.unimib.it

doi:10.1016/j.entcs.2005.12.014

tasks, we can distinguish activities that contribute mainly to the construction of the product, and activities that aim at checking the quality of the development process and of produced artifacts. This classification is not sharp, since most activities contribute to some extent to both advancing the development and checking the quality. In some cases, this characterization of activities is not precise enough, but helps identify an important thread of the development process that includes all quality-related activities and is often referred to as the *quality process*.

The quality process is not a phase, but it spans through the whole development cycle: it starts with the feasibility study, and goes beyond product deployment through maintenance and post mortem analysis. The qualities relevant to the product must be defined in the feasibility study; requirements and design specifications must be inspected and analyzed as early as possible to identify and remove design errors otherwise hard to reveal and expensive to remove; tests must be designed and planned in the early design phases to improve the specifications and reduce the chances of delivering badly tested and low quality products, and must be executed many times through different builds and releases to avoid regression of product functionality.

The quality process includes many complementary activities that must be suitably blended to fit the specific development process, and meet cost and quality requirements. The quality engineer must sometimes face contradicting requirements, like keeping costs low, guaranteeing high quality, avoiding interference with the development process, and meeting stringent deadlines. The selection of a suitable set of quality activities is hard and requires deep experience of software validation, strong knowledge of design and development, good background of management and planning, and excellent abilities to mediate among different aspects and needs.

Quality activities address two different classes of problems: revealing faults and assessing the readiness of the product. Quality cannot be added at the end of the process, but it must be enforced throughout the whole development cycle. Important classes of faults are difficult to reveal and hard to remove from the final product. Many analysis and testing techniques aim at revealing faults, which can then be eliminated. Identifying and removing faults through the development process certainly help increase the quality of the final product, but cannot assure the absence of faults that can persist after product delivery. Continuing to search for faults until all faults are revealed and removed would lead to executing quality activities forever without any rationale. Users are interested in solving problems, and measuring the quality of the software products in terms of dependability, usability, costs, and ultimately ability of meeting users' expectations, and not in terms of avoided

or removed faults. Users tolerate few annoying failures in products that address their problems cost-effectively, but do not accept critical failures —even if they are very rare—, or too frequent annoying failures. Thus, it is important to pair quality activities that aim at revealing faults, with activities that estimate the readiness of a product in terms of dependability and usability.

Most quality activities are carried on independently of the development, but their effectiveness strongly depends on the quality of the development process. For example, reviews and inspections of requirements specifications are much more effective when specifications are well-structured than when they are badly written, and integration testing is more effective when software is implemented according to good design principles than when it is developed without a coherent approach. Quality activities can help increase the quality of the development practice by providing feedback on the causes of faults and common errors.

The rest of the paper is organized as follows. Section 2 discusses the main characteristics of the quality process and emphasizes the design of a good quality plan. Section 3 introduces the differences between verification and validation activities. Section 4 addresses the problems of functional and model-based testing by sampling some popular modern approaches, and highlighting their complementarities. Section 5 describes the different levels of testing activities and presents the main approaches to module, integration, system and acceptance testing. Section 6 concludes the paper and identifies relevant open research problems.

2 Quality process

The quality process involves many activities that can be grouped in five main classes: planning and monitoring, verification of specifications, test case generation, test case execution and software validation, and process improvement.

Planning and monitoring activities aim at steering the quality activities towards a product that satisfies the initial quality requirements. Planning activities start in the early phases of development with the identification of the required qualities and the definition of an early analysis and test plan, and continue through the whole development by monitoring the quality process and by refining and adjusting it to take care of new problems and avoid that deviations from the original plan lead to project failures.

Specifications can be verified for both internal consistency and consistency with respect to the corresponding specifications. Verification of intra-specification consistency aims at revealing and eliminating inconsistency or incompleteness of specifications; verification of inter-specification consistency

aims at revealing deviations in the development process that manifest themselves as either differences in respect to the corresponding specifications or missing elements in detailed specifications. Specifications can be verified by means of many technologies that span from simple syntactic checks and low-technology inspection to model checking and formal verification.

Test cases are usually generated from specifications, and are integrated with information from the application environment, development technology, and code coverage. The application environment and development technology can provide information on common faults and risks. Many organizations collect general test suites that derive from legacy systems and characterize specific application environments, either in the form of regression test suites for specific applications, or in the form of general suites for well known features. Programming languages and development technologies present different weaknesses that can lead to specific classes of faults. For example, the C++ freedom in memory management comes with well-known risks that can lead to dangerous memory leaks, which can be limited —but not avoided— with disciplined design and programming practice, suitably paired with analysis and tests. Code coverage indicates regions of code that have not been adequately tested, and may suggest additional test cases to complete functional test suites or indicate ill-designed code.

Test cases might and should be generated as soon as the corresponding specifications are available. Early test case generation has the obvious advantage of alleviating scheduling problems: tests can be generated in parallel with development activities, and thus be excluded from critical paths. Moreover, test execution can start as soon as the corresponding code is ready, thus reducing the testing time after coding. Early generation of test cases has also the important side effect of helping validate specifications. Experience shows that many specification errors are easier to detect early than during design or validation. Waiting till when specifications are used for coding may be too late and may lead to high recovery costs and expensive project delays. Many modern methodologies suggest early generation of test cases up to the case of *extreme programming* that substitutes module specification with test case generation.

Test cases may need to be executed in absence of the complete system. This can be due either to the decision of incrementally testing modules without waiting for the development of the whole system, or the need of isolating the modules-under-test to focus on particular features and facilitate the localization of faults. Executing test cases without the whole system requires the development of adequate scaffolding, i.e., a structure that substitutes the missing parts of the systems. Adequate scaffolding includes drivers, stubs, and

oracles. Drivers and stubs surrogate the execution environment by preparing the activation environment for the module-under-test (drivers) and by simulating the behavior of missing components that may be required for the execution of the module-under-test (stubs). Oracles evaluate the results of executed test cases and signal anomalous behaviors. To build adequate scaffolding, software engineers must find a good compromise between costs and benefits. Accurate scaffolding can be very useful for fast test execution, but it can be extremely expensive and dangerously faulty. Cheap scaffolding can reduce costs, but may be useless for executing tests.

Testing can reveal many kinds of failures, but may not be adequate for others, and thus it should be complemented with alternative validation activities. For example the Cleanroom approach [7] suggests that testing be complemented with code inspection to reveal faults at the unit level.

Process improvement focuses mostly on clusters of projects that share similar processes, engineering teams, and development environments. Quality experts collect and analyze data on current projects to identify frequent problems and reduce their effects. Problems can be either eliminated by changing development activities, or alleviated by introducing specific quality activities for removing problems as early as possible. Although some corrective actions can be already taken in the current project, in many cases, corrective actions are usually applied to future projects.

2.1 Quality Plan

The quality engineer should design the quality plan in the very early phases of the design cycle. The initial plan is defined according to the test strategy of the company and the experience of the quality engineer. The test strategy describes the quality infrastructure of the company that includes process issues, e.g., the adoption of a specific development process, organizational issues, e.g., the choice of outsourcing specific testing activities, tool and development elements, e.g., the need for using a particular toolsuite, and any other element that characterizes and influences the quality plan.

The initial plan is then refined to take into account the incremental knowledge about the ongoing project, e.g., by detailing module testing activities once design specifications are available. When the plan cannot be detailed and adapted to the project needs, the current plan must be substituted with an emergency plan, to cope with new unforeseen situations.

A quality plan should include all information needed to control and monitor the quality process, from general information, like the items to be tested, to very detailed information, like a scheduling of the single quality activities and the resources allocated to conduct them. The main elements of a good quality plan are:

Test items characterize the items that have to be tested, indicating for example the versions or the configurations of the system that will be tested

Features to be tested indicate the features that have to be tested, among all the features offered by the items to be tested.

Features not to be tested indicate the features that are not considered in the plan. This helps check for completeness of the plan, since we can check if we explicitly considered all features before selecting the ones to be tested.

Approach describe the approach to be chosen. We can for example require that all modules be inspected, and we may prescribe specific testing techniques for subsystem and system testing, according to the company standards.

Pass/Fail criteria define the acceptance criteria, i.e., the criteria used for deciding the readiness of the software-under-test. We may for example tolerate some moderate impact faults, but ask for the absence of severe and critical faults before delivering a module.

Suspension and resumption criteria describe the conditions under which testing activities cannot be profitably executed. We may for example decide to suspend testing activities when the failure rate prevents a reasonable execution of the system-under-test, and resume testing only after the success of a "sanity test" that checks for a minimum level of executability of the unit-under-test.

Risks and contingencies identify risks and define suitable contingency plans. Testing may face both risks common to many other development activities: e.g., personnel (loss of technical staff), technology (low familiarity with the technology) and planning (delay in some testing tasks) risks, and risks specific to testing, e.g., development (delivery of poor quality components to the testing group), execution (unexpected delays in executing test cases) and requirements (critical requirements) risks.

Deliverables list all required deliverables.

Task and Schedule articulate the overall quality process in tasks scheduled according to development, timing and resources constraints to meet the deadlines. The early plan is detailed while progressing with the development, to adjust to the project structure. The early plan for example indicates generic module testing and inspection tasks that will be detailed when the design identifies the specific modules of the system.

Staff and responsibilities identify the quality staff and allocate responsibilities.

Environmental needs indicate any requirements that may derive from the environment, e.g., specific equipment that is required to run the test cases.

2.2 Monitoring the Quality Process

The quality process must be monitored to reveal deviations and contingencies, and to adapt the quality activities to the new scenarios as early as possible. Monitoring can take two forms: the classic supervision of the progress of activities and the evaluation of quantitative parameters about testing results.

The supervision of the progress consists in periodically collecting information about the amount of work done and compare it with the plan: the quality engineer records start and end dates, consumed resources, and advances of each activity, and reacts to deviations to the current plan by either adapting it, when deviations are tolerable, or adopting a contingency plan, when deviations are critical. The evaluation of quantitative progress is difficult and has been exploited only recently. It consists in gathering information about the

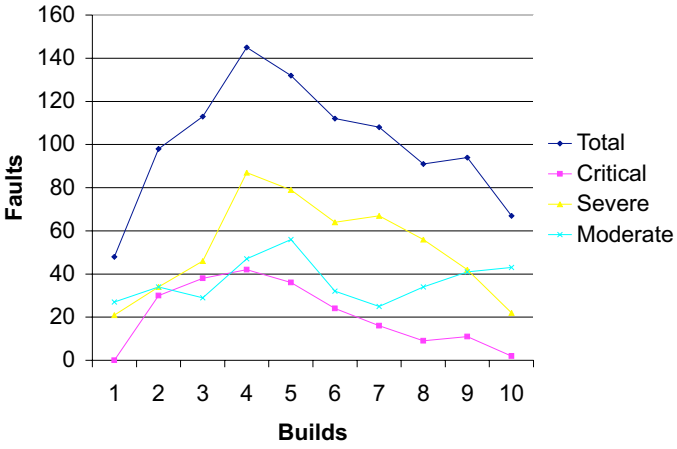

Fig. 1. A typical distribution of faults for system builds over time.

distribution of faults and comparing it with historical data. Fault exposure follows similar patterns across similar projects both in terms of frequency of faults and in terms of distribution across different fault categories.

The diagram of Figure 1 is taken from [9] and illustrates the distribution of faults over releases, considering three levels of severity. The diagram clearly indicates that fault occurrence grows for the first builds before decreasing. The number of faults decreases with different speed: critical and severe faults decrease faster than moderate ones, and we can even expect a slight growth of moderate faults. Different distributions of faults signal possible anomalies in the quality process: a non-increasing fault occurrence in the early builds may indicate poor tests, while a non-decreasing fault occurrence in the latter releases may indicate bad fault diagnosis and removal.

The Orthogonal Defect Classification (ODC) introduced by IBM in the early nineties proposes a detailed classification of faults and suggests monitoring different distributions to reveal possible problems in the quality process. Details about ODC can be found in [3,1].

Despite available technologies and tools, the quality process heavily depends on people. The allocation of responsibilities is a key element of quality strategies and plans and can determine the success of a project. As with many other aspects, there is no unique solution, but approaches depend on organizations, processes and projects. Large organizations usually prefer to separate development and quality teams, while small organizations and agile processes (e.g., extreme programming) tend to assign development and quality responsibilities to the same team. Separating development and quality teams encourages objective judgment of quality and prevents it from being

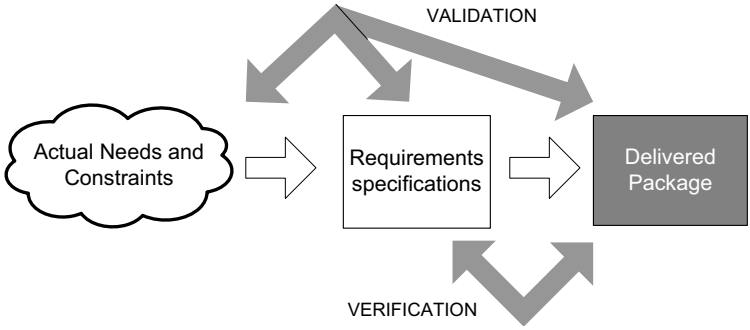

Fig. 2. The different perspectives of validation versus verification.

subverted by scheduling pressure, but restrict scheduling freedom and requires good communication mechanisms between the two teams.

3 Verification and validation

Activities that aim at checking the correspondence of an implementation with its specification are called *verification activities*, while activities that aim at checking the correspondence between a system and users' expectations are called *validation* activities. The distinction between validation and verification is informally illustrated by the diagram of Figure 2 and has been well-framed by Barry Boehm [2], who memorably described validation as *"building the right system"* and verification as *"building the system right"*.

Validation and verification activities complement each other. Verification can involve all development stages with users' reviews of requirements and design specifications, but the extent of users' reviews is limited by the ability of users to understand design and development details. Thus, the main validation activities concentrate on the final product that can be extensively tested by the users during acceptance testing. Users' needs and the late execution of validation activities lead to high costs and risks, thus validation must be paired with verification activities, which can be extensively executed at early stages and do not involve expensive and not-always available users' availability.

Software is characterized by many properties that include dependability, usability, security, interoperability, etc. Some properties can be naturally verified. For example, the ability of a Web application to serve up to a given number N of users with response time below a given threshold τ can be verified by simulating the presence of N users and measuring the response time. Other properties can be difficult to verify and are a natural target for validation. For example, the ability of users to easily obtain the required information from a Web application is hard to verify, but can be validated, e.g., by monitoring a

sample population of users.

Ultimately, the verifiability of properties depends on the way properties are expressed. For example, a property of a Web application expressed as: *users must be able to easily add an item to the shopping cart without experiencing annoying delays* cannot be intuitively verified, since it refers to subjective feelings ("easily" and "annoying"). However, the same property expressed as: *users must be able to add a given item to the shopping cart with no more than 4 mouse clicks starting from the home page, and the delay of the application must not exceed a second after the click when the application is serving up to ten thousand users working concurrently* can be verified, since subjective feelings are rendered as quantifiable entities ("number of mouse clicks" and "delays in seconds"). Thus, system testing starts early in the design when writing requirements specifications. Mature development processes schedule inspection activities to assess their testability and thus maximize the verifiability of the software product.

Verification and validation of different properties require specific approaches. Dependability properties can be verified using functional and model-based testing techniques discussed in the next section, but usability properties for example require special-purpose techniques for their validation. In this case, a standard process includes the following main steps:

- Inspecting specifications with classic inspection techniques and ad-hoc checklists.
- Testing early prototypes produced by statically simulating the user interfaces.
- Testing incremental releases with both usability experts and end users.
- Considering final system and acceptance testing that includes expert-based inspection and testing, user-based testing, comparison testing, and automatic analysis and checks.

We can notice that usability testing heavily relies on users, unlike functional and model-based testing that do not require user intervention. User based testing is carefully planned and evaluated: the usability team identifies classes of users, selects suitable samples of the population according to the identified classes, defines sets of interactions that well represent common and critical usages of the system, carefully monitors the interactions of the selected users with the system, and finally evaluates the results. More details on usability testing can be found in [13].

4 Functional and model-based testing

Moving to verification, functional testing is the base-line technique for designing test cases. It applies to all levels of the requirements specification and design process, from system to module testing, it is effective in finding some classes of faults that typically elude code- and fault-based testing, it can be applied to any description of program behavior, and finally, functional test cases are typically less expensive to design and execute than tests designed by means of other techniques.

A functional specification is a description of the expected behavior of the program. Specifications can be given in many ways. From the testing viewpoint, we distinguish between specifications expressed in natural language and specifications expressed in some (semi-)formal languages, e.g., finite state machines, decision tables, control flow diagrams, UML diagrams. When specifications are expressed in natural language. Functional testing techniques define a set of steps that help analyze specifications to obtain a satisfactory set of test cases. When specifications are given as (semi-)formal models, functional testing consists in applying suitable criteria to extract a set of test cases. We also refer to this activity as *model-based* testing.

4.1 Functional testing

Modern general-purpose functional testing approaches include category-partition [14], combinatorial [4], and catalog-based [12] testing.

Category partition testing applies to specifications expressed in natural language and consists of three main steps. We start *decomposing the specification into independently testable features:* the test designer identifies specification items that can be tested independently, and identifies parameters and environment elements that determine the behavior of the considered feature (*categories*). For example, if we test a Web application, we may identify the catalog handler functionality as an independently testable feature. The following excerpt of a specification determines the production and the sale of items on the basis of the sales and the stock in the last sale period:

> *The production of an item is suspended if in the last sales period the number of orders falls below a given threshold* t_1 *or if it falls below a threshold* $t_2 > t_1$ *and the amount in stock is above a threshold* s_2*. An item is removed from the catalog if it is not in production, the amount of orders in the previous period remains below* t_1 *and the amount in stock falls below a threshold* $s_1 < s_2$*. The production of an item in the catalog is resumed if the amount of orders in the former period is higher than* t_2 *and the amount in stock is less than*

s_1.

Items that are also sold in combination with other items are handled jointly with the assembled items, i.e., the production is not suspended if one of the assembled items is still in production, despite the sales and the stock of the considered item, and similarly it is kept in the catalog even if eligible for withdraw, if the assembled items are kept in the catalog.

The amount in stock cannot exceed the maximum capacity for each item.

¿From the informal specification, we can deduce the following categories that influence the behavior of the functionality: number of orders in the last period, amount in stock, type and status of the item, status of assembled items.

Then we *identify relevant values:* the test designer selects a set of representative classes of values (*choices*) for each parameter characteristic. Values are selected in isolation, independently of other parameter characteristics. For example, Figure 3 shows a possible set of choices for the categories extracted from the specification of the catalog handler (readers should not consider the keywords in square parenthesis yet). We notice that choices are not always individual values, but in general indicate classes of homogeneous values.

When selecting choices, test designers have to refer to normal values as well as boundary and error values, i.e., values on the borderline between different classes (e.g., zero, or the values of the thresholds for the number of orders or the amount in stock that are relevant for the considered case.)

In the end, we *generate test case specifications:* test case specifications can straightforwardly be generated as combinations of the choices identified in the above steps.

Unfortunately, the mere combination of all possible choices produces extremely large test suites. For example, the simple set of choices of Figure 3 produces more than 1,000 combinations. Moreover, many combinations may make little or no sense. For example, combining individual items with different status of assembled items makes no sense, or test designers may decide to test only once the boundary cases.

Test designers can eliminate erroneous combinations and limit singletons by imposing simple constraints: *[error]* marks erroneous values and requires at most one test case for that value, *[single]* marks singleton values and requires at most one test cases for them as well, pairs *["label"]*, *[if-"label"]* constrain one value to occur in combination with a value indicated by the label. The constraints given in Figure 3 reduce the number of generated test cases from more than 1,000 to 82 that represent an adequate set of test cases.

The category partition approach helps when natural constraints between choices reduce the number of combinations, as in the simple catalog handler

orders in the period	**amount in stock**	**status of the item**
0 *[single]*	0 *[single]*	in production
$< t_1$	$< s_1$	in catalog
t_1 *[single]*	s_1 *[single]*	not available *[error]*
$< t_2$	$< s_2$	**status of assembled item**
t_2 *[single]*	s_2 *[single]*	in production *[if assmbl]*
$> t_2$	$< s_{max}$	in catalog *[if assmbl]*
$>> t_2$	s_{max} *[single]*	not available *[if assmbl][error]*
type of item	$> s_{max}$ *[error]*	
individual		
assembled *[assmbl]*		

Fig. 3. Category partition testing: A simple example of categories, choices and constraints for the example catalog handler. The choices for each category are listed in the corresponding columns. Constraints are shown in square brackets.

example. However, in other cases, choices are not naturally constrained, and the amount of test cases generated by considering all possible combinations of choices may exceed the budget allocated for testing.

Let us consider for example the informal specification of a discount policy. The following excerpt of a specification of a discount policy handler determines the applicable discount on the basis of the status of the customer and the amount of purchases.

> *Discount is applied to both individuals and educational institutions. Additional discounts are applied if the amount of orders in the considered sale period exceeds a threshold o_1 or a threshold $o_2 > o_1$, respectively. Discounts are applied also if the amount of the current invoice or the total amount of invoices in the current sales period exceeds given thresholds (i_1 and $i_2 > i_1$ for the current invoice, and t_1 and $t_2 > t_1$ for the total invoices in the period). Discounts can be cumulated without limits. Customers with a risky credit history do not qualify for any discount.*

None of the choices of Figure 4, which derive from the above specification, are naturally constrained, and thus we obtain a combinatorial amount of test cases. In the current example, it goes up to 182 test cases and grows exponentially when the number of categories or choices increases as in most real cases.

Forcing artificial constraints does not help, since it reduces the number of generated test cases by eliminating combinations of values that could reveal important faults. A different approach consists in limiting the combinations

customer type	# of orders in the period	amount in the invoice	total invoices in the period	credit situation
individual	$\leq o_1$	$\leq i_1$	$\leq t_1$	ok
business	$\leq o_2$	$\leq i_2$	$\leq t_2$	risky
educational	$> o_2$	$> i_2$	$> t_2$	

Fig. 4. A set of categories and choices that do not have natural constraints

by covering only pairwise combinations of choices. For example, all pairwise combinations of Figure 4 can be covered with less than 18 test cases. Generating test cases that cover only pairwise combinations is known as *combinatorial testing* and is based on experimental evidence that most failures depend on single choices or pairwise combinations of choices, and rarely depend on specific combinations of many different choices. Thus covering all pairwise combinations of choices can reveal most potential failures with test suites of limited size.

Category partition and combinatorial testing can be fruitfully combined by first constraining the combination of choices and then covering only pairwise combinations.

When selecting choices for the identified categories, we identify both normal values and boundary and error conditions. In general many faults hide in special cases that depend on the type of considered elements. For example, when dealing with a range *[low, high]* of values, test experts suggest considering at least a value within the boundaries, the bounds *low* and *high* themselves, the values before and after each bound, and at least another value outside the range. The knowledge of expert test designers can be captured in *catalogs* that list all possible cases that have to be considered for each type of specification. We may build both general purpose and specialized catalogs. The first can be used in most cases, while the second only apply to specific domains that are characterized by particular cases. Catalogs apply well to well-structured specifications. Thus in general, *catalog-based* testing first transforms specifications in the form of pre- and post-conditions, variables, definitions, and functions, and then applies catalogs to derive a complete set of test cases.

4.2 Model-based testing

When specifications are expressed in a (semi-)formal language, we can derive test cases by applying test generation criteria to these models. Test designers concentrate on the creative steps of testing (the derivation of the finite state model) and use automatic techniques for the repetitive tasks (the application of test case generation criteria). In this paper we focus on finite state

automata, but the same approach can be applied to several different models.

When specifications describe transitions among a finite set of states, it is often natural to derive a finite state model for generating test case specifications. For example, let us consider the following informal specifications of the functionality of a *shoppingCart* of a Web application:

A shopping cart can be manipulated with the following functions:
- ***createCart()*** *creates a new empty cart.*
- ***addItem(item, quantity)*** *adds the indicated quantity of items to the cart.*
- ***removeItem(item, quantity)*** *removes the indicated quantity of items from the cart.*
- ***clearCart()*** *empties the cart regardless of its content.*
- ***buy()*** *freezes the content of the cart and computes the price.*

Alternatively, the shopping cart can be modeled with the finite state machine of Figure 5. It does not substitute the informal specification, but captures state-related aspects. Test cases can be generated by deriving sequences of invocations that cover different elements of the finite state machine. A simple criterion requires that all transitions be covered (*transition coverage*). Sophisticated criteria require that different kinds of paths be covered (*single state path coverage*, *single transition path coverage*, *boundary interior loop coverage*).

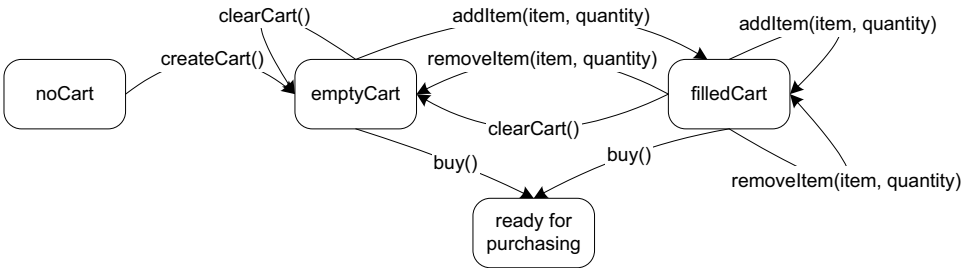

Fig. 5. A finite state machine model extracted from the informal specification of the shopping cart

Often finite state behaviors are rendered with models that provide features for simplifying the model. Statecharts are one of the most well-known examples: and- and or-decomposed states as well as histories and default states can greatly simplify a finite state machine. For example, in the Statecharts of Figure 6, we grouped the *empyCart* and *filledCart* states of the finite state machine of Figure 5, in a single or-decomposed state, thus merging the two *buy* edges into a single one. In most cases, we can easily extend the criteria available for finite state machines and add new ones that take advantage of the

specific structure of the new model. For example, with large Statecharts, we can reduce the size of generated test cases by covering only the transitions of the Statecharts and not all transitions of the equivalent finite state machine. In the example, we would cover transition *buy* only once and not twice as in the equivalent machine.

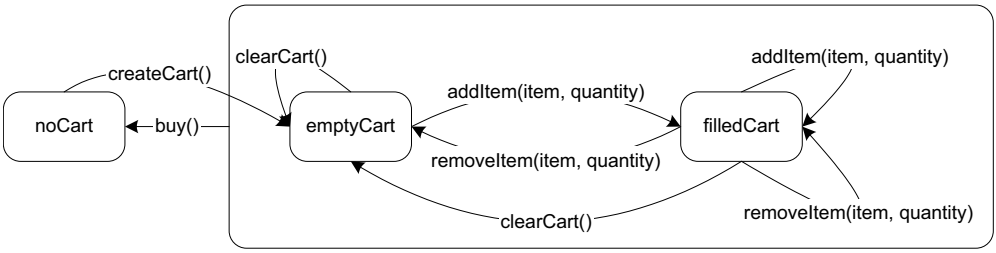

Fig. 6. A Statecharts specification of the shopping cart described above

5 Testing levels

Software development involves different abstraction levels: *modules* or *components*, which may be developed for the specific system or reused from previous systems or libraries, *subsystems*, which are obtained by integrating sets of related components, and the final *system*, which is obtained by assembling components and subsystems in an application that satisfies initial requirements. Testing is performed at each level, as illustrated by the classical V model of Figure 7. We often distinguish *module*, *integration*, *system* and *acceptance* testing.

5.1 Module Testing

Modules or components are first verified in isolation usually by the developers themselves who check that the single modules behave as expected (*module testing*). Thorough module testing is important to identify and remove faults that otherwise can be difficult to identify and expensive to remove in later development phases. Module testing includes both functional and structural testing. Functional test cases can be derived from module specifications using an appropriate functional or module-based testing technique, like the ones outlined in the previous section.

Functional tests may fail in revealing some classes of problems, since developers may implement the same feature in different ways and the test cases derived from a given feature may not cover all possible implementations. An intuitive example is a well-designed sorting routine that implements different algorithms to cope with very small sets, reasonably large sets, and sets larger

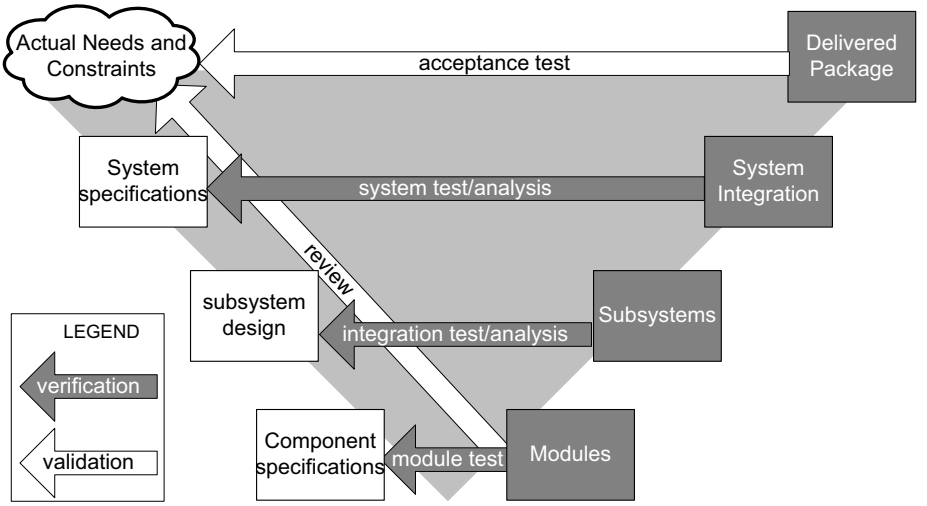

Fig. 7. Development and testing phases in the V model

than available memory. We all know that a simple quadratic algorithm, like *bubblesort*, is adequate for sorting small sets, while *quicksort* may be preferred for sorting large sets that fits in memory, but sets larger than available memory are sorted better with an algorithm like *treesort*. If the choice of the algorithms is left to the programmer and not included in the specifications, functional test cases that are derived without knowing this choice might not cover the whole implementation even if well designed.

Structural testing complements functional testing by covering the structure of the code and thus coping with cases not included in functional testing. Structural testing is often applied at two stages: first programmers measure the code coverage with simple coverage tools that indicate the amount of covered code and highlight the uncovered elements of the code, and then generate test cases that traverse uncovered elements. Coverage may be measured by taking into account different elements of the code. The simplest coverage criterion focuses on statements, and measures the percentage of statements executed by the test cases. Branch and condition coverage criteria measure the amount of branches or conditions exercised by the tests. Path coverage criteria measure the amount of paths covered by the tests. Different path coverage criteria can be identified based on the way paths are selected. Additional criteria refer to flow of data. Readers may find additional information on code coverage in [5].

Coverage criteria refer to all the considered elements, e.g., statements, as statically identified, and thus includes also non executable elements. Unfortunately the problem of identifying executable elements is undecidable, and thus we cannot automatically select only the right subset of executable ele-

ments. In most cases, code coverage is not used as an absolute indicator, but it is rather employed as an approximate measure to monitor module testing activities. For example, if we refer to statement coverage, the presence of up to 10-15% of non-executable statements may be acceptable, but a higher percentage of non-executable statements, i.e., a statement coverage below 85-90% may indicate either a bad design, which leads to far too many non-executable statements, a bad specification, which does not allow to derive a suitable set of functional test cases, or a shallow module testing with an insufficient set of test cases. When the coverage falls below an acceptable threshold, test designers and developers examine the module closer to identify and fix the problem.

In some critical cases, all elements that are not covered by the tests are examined to understand if they can be covered or motivate the presence of non-executable statements. For example, the RTCA/DO-178B "Software Considerations in Airborne Systems and Equipment Certification," and its European equivalent EUROCAE ED-12B, which are quality standards commonly used in the avionic industry, require MCDC coverage for on-board software and require manual inspection of elements that are not covered by tests.

The modified condition adequacy (MCDC) criterion applies to basic conditions, i.e., elementary components of predicates that select the branch of the program to be executed, and requires that each basic condition be shown to independently affect the outcome of each decision. That is, for each basic condition C, there are two test cases in which the truth values of all conditions except C are the same, and the compound condition as a whole evaluates to *true* for one of those test cases and *false* for the other.

5.2 Integration and component-based testing

The quality of single modules is necessary but not sufficient enough to guarantee the quality of the final system. The failure of low quality modules fatally leads to system failures that are often difficult to diagnose, and hard and expensive to remove. Unfortunately, many subtle failures are caused by unexpected interactions among well-designed modules. Well-known examples of unexpected interactions among well-designed software modules are described in the investigation reports of the Ariane 5 accident that caused the loss of the rocket on July 4th, 1996 [11], and of the Mars Climate Orbiter failure to achieve the Mars orbit on September 23rd, 1999 [8].

In the Ariane accident, a module that was adequately tested and successfully used in previous Ariane 4 missions failed in the first Ariane 5 mission causing the chain of events that led to the loss of the rocket. The module was in charge of computing the horizontal bias, and it failed because of an

overflow caused by the higher horizontal velocity of the Ariane 5 rocket than that of the Ariane 4. The Mars Climate Orbiter failure was caused by the unexpected interactions between software developed by the JPL laboratories and software developed by the prime contractor Lockheed Martin. The software developed by Lockheed Martin produced data in English units, while the Spacecraft operating data needed for navigation were expected in metric units. Both modules worked well when integrated in systems referring to homogenous measure systems, but failed when incorrectly integrated.

Integration faults are ultimately caused by incomplete specifications or faulty implementations of interfaces, resource usage, or required properties. Unfortunately, it may be difficult or cost-ineffective to specify all module interactions completely. For example, it may be very difficult to predict interactions between remote and apparently unrelated modules that share a temporary hidden file that happens to be given the same name by two modules, particularly if the name clash appears rarely and only in some particular configurations. Integration faults can come from many causes:

- Inconsistent interpretation of parameters or values, as in the case of the Mars Climate that interpreted English and metric units;
- Violations of value domains or of capacity/size, as it happened in some versions of the Apache 2 Web server that could overflow a buffer while expanding environment variables while parsing configuration files;
- Side effects on parameters or resources, as can happen when modules use resources not explicitly mentioned in their interfaces;
- Missing or misunderstood functionality, as can happen when incomplete specifications are badly interpreted;
- Non-functional problems, which derive from under specified non-functional properties, like performances;
- Dynamic mismatches, which derive from unexpected dynamic bindings.

Integration testing deals with many communicating modules. *Big bang testing*, which waits until all modules are integrated, is rarely effective, since integration faults may hide across different modules and remain uncaught, or may manifest in failures much later than when they occur, thus becoming difficult to localize and remove. Most integration testing strategies suggest testing integrated modules incrementally. Integration strategies can be classified as *structural* and *feature-driven*. Structural strategies define the order of integration according to the design structure and include *bottom-up* and *top-down* approaches, and their combination, which is sometimes referred to as *sandwich* or *backbone* strategy. They consist in integrating modules according

to the use/include relation, starting from the top, the bottom or both sides, respectively.

Feature-driven strategies define the order of integration according to the dynamic collaboration patterns among modules, and include *thread* and *critical module* strategies. Thread testing suggests integrating modules according to threads of execution that correspond to system features. Critical module testing integrates modules according to the associated risk factor that describes the criticality of modules.

Feature-driven test strategies better match development strategies that produce early executable systems, and may thus benefit from early user feedback, but they usually require more complex planning and management than structural strategies. Thus, they are preferable only for large systems, where the advantages overcome the extra costs.

The use of COTS components further complicates integration testing. Components differ from classical modules for being re-used in different contexts independently of their development. System designers, who reuse components, do not often have access to the source code or to the developers of such components, but can only rely on the specifications of the components' interfaces. Moreover, components are often reused in contexts that are not always foreseen at development time, and their behavior may not fully match the specific requirements. When testing components, designers should identify the different usage profiles of components and provide test suites for each of the identified profiles. System designers should match their requirements with provided profiles and re-execute the integration tests associated with the identified profiles before deriving test suites specific to the considered context.

5.3 System, acceptance, and regression testing

Module and integration testing can provide confidence on the quality of the single modules and on their interactions, but not about the behavior of the overall system. For example, knowing that a module correctly handles the product database and that the product database inter-operates correctly with the module that computes prices does not assure that the whole system implements the discount policies as specified in the system requirements. Moreover, knowing that the system matches the requirements specifications does not assure that it behaves as expected by the users, whose expectations may not fully match the results of the requirements analysis. We thus need to complete the verification and validation process by testing the overall system against its specifications and users' needs. System testing verifies the correspondence between the overall system and its specifications, while acceptance testing verifies the correspondence between the system and the users' expectations.

System and acceptance testing consider the behavior of the overall system in its functional and non-functional aspects. Module and integration testing are largely based on the internals of the software, which is hardly accessible to the user. Thus, they focus on verification activities that provide useful results without requiring the deployment of the whole system and the presence of the users. Most module and integration testing activities focus on functional properties, as discussed earlier in this section.

Some non-functional properties, like modularity, maintainability, and testability can be enforced through design rules and checked with simple static analysis tools and manual inspection during development, but do not involve extensive testing. Other important non-functional properties, like usability, and performance, can be partially enforced through good design practice, but their testing requires the presence of the whole system and the users. Performance and usability tests and analysis on early prototypes can assess important design decisions and prevent some errors in the development, but tests on the deployed system is essential to provide reliable data. Other non-functional properties, such as safety and security, do not involve the testing team, but are considered by special teams of experts that work in parallel with the testing team.

Structural testing is of little help in system testing, due to the size of the software. Measuring code coverage over millions of lines of code or identifying a path not yet exercised is of little help. On the other hand, identifying the set of offered features and covering all possible cases is very important for revealing system errors. Thus system testing is mostly based on functional and model-based testing. The most important problem in system testing is mastering the requirements specifications that can be very large and complex. Fortunately, specifications are written for software architects, who need to be able to understand the overall system and identify a coherent architecture of the software. Thus, specifications are often well structured and quality engineers can identify features that can be tested in isolation. Figure 8 outlines the main steps of system testing. Quality engineers start identifying individually testable features, i.e., features that can be handled coherently by a single test designer. For each of the identified features, test designers identify the best testing approach, which can be model-based (bottom path in the figure) or functional testing (top path in the figure).

Software is not produced linearly, but undergoes several builds and releases. Each new build or release may introduce or uncover faults and results in failures not experienced in previous versions. It is thus important to check that each new release does not regress with respect to the previous ones. Such testing is often called non-regression testing, or, for short, *regression testing*.

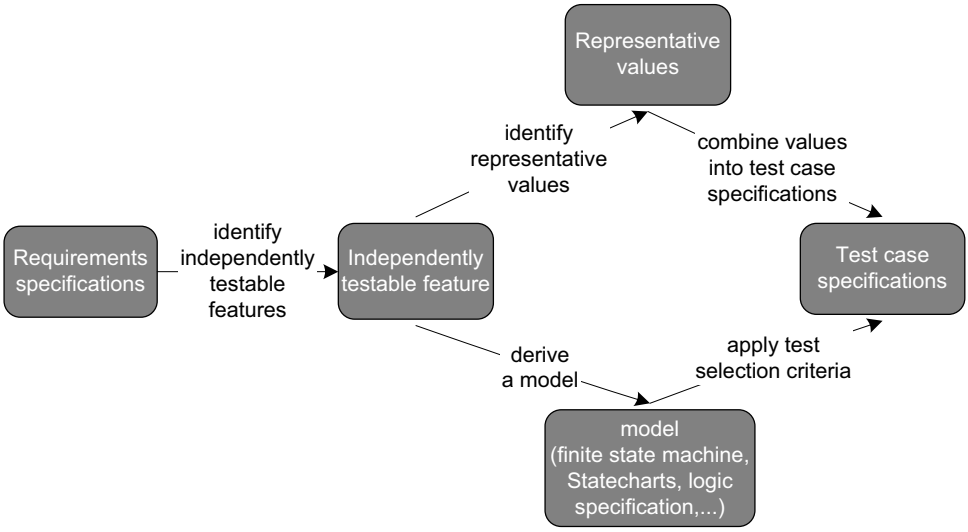

Fig. 8. A systematic approach to system testing

A simple approach to regression testing, known as the *retest all* approach, consists in re-running all the test cases designed for the previous versions, to check if the new version presents anomalous behaviors not experienced in the former versions. This simple approach may present non trivial problems and significant costs that derive from the need of adapting test cases not immediately re-executable on the new version. Moreover, the costs of re-running all test cases may be too expensive and not always useful.

The number of test cases to be re-executed can be reduced with ad-hoc techniques tailored to the specific application or with general-purpose selection or prioritization techniques. Selection techniques work on code or specifications. The ones that work on code record the program elements exercised by the tests on previous releases, and select test cases that exercise elements changed in the current release. Various code-based selection techniques focus on different programming elements: control-flow, data-flow, program slices, etc... Code-based selection techniques find good tool support and work even when specifications are not properly maintained, but do not scale up easily: they apply well to simple local changes, but present difficulties when changes affect large portions of the software.

Specification-based selection techniques focus on changes in the specifications. They scale up much better than code-based techniques, since they are not bound to the amount of changed code, but require properly-maintained specifications. They work particularly well with model-based testing techniques, that can be augmented with change trackers to identify tests to be re-executed. For example, if the system is modeled with finite state machines,

we can easily extend classic test generation criteria to focus on elements of the finite state machines that are changed or added in the current build [6].

Test case prioritization techniques do not select a subset of test cases, but rather define priorities among tests and suggest different execution strategies. Priorities aim at maximizing the effectiveness of tests by postponing the execution of cases that are less likely to reveal faults. Popular priority schemas are based on execution histories, fault detection effectiveness, and code structure. History-based priority schemas assign low priority to recently executed test cases. In this way, we guarantee that all test cases will be re-executed in the long run. This technique works particularly well for frequent releases, e.g., overnight regression testing. Priority schemas that focus on fault-detection effectively raise the priority of tests that revealed faults in recent versions, and thus are likely to exercise unstable portions of the code and reveal new or persistent faults. Structural priority schemas give priority either to test cases that exercise elements not recently executed or to test cases that result in high coverage. In the first case, they try to minimize the chances that portions of code are not tested across many consecutive versions; in the second case, they try to minimize the set of tests to be re-executed to achieve high coverage [10].

All regression testing techniques rely on good maintenance of test cases and good test documentation. High-quality test suites can be maintained across versions by identifying and removing obsolete test cases, and revealing and suitably marking redundant test cases. Good test documentation includes planning, test specifications and reporting documents. Planning documents include test plans, which are briefly described in Section 2, and test strategies that summarize the quality strategies of the company or the group. Test specification documents focus on test suites and single test cases. Reporting documents summarize the results of the execution of single test cases and test suites. Regression testing requires the coordination of specification and reporting documents.

6 Conclusions

Software testing has been an active research area for many decades, and today quality engineers can benefit from many results, tools and techniques. However, challenges are far from being over: while many traditional research areas are still open, advances in design and applications open many new problems. So far, research on testing theory produced mostly negative results that indicate the limits of the discipline, but call for additional study. We still lack a convincing framework for comparing different criteria and approaches. Available testing techniques are certainly useful, but not completely satisfac-

tory yet. We need more techniques to address new programming paradigms and application domains, but more important, we need better support for test automation.

Component-based development, heterogeneous mobile applications, and complex computer systems raise new challenges. It is often impossible to predict all possible usages and execution frameworks of modern software systems, and thus we need to move from classic analysis and testing approaches that mostly work before deployment, to approaches that work after deployment, like dynamic analysis, self healing, self managing and self organizing software.

References

[1] I. Bhandari, M. J. Halliday, J. Chaar, K. Jones, J. Atkinson, C. Lepori-Costello, P. Y. Jasper, E. D. Tarver, C. C. Lewis, and M. Yonezawa. In-process improvement through defect data interpretation. *The IBM System Journal*, 33(1):182–214, 1994.

[2] B. W. Boehm. *Software Engineering Economics*. Prentice Hall, Englewood Cliffs, NJ, 1981.

[3] J.K. Chaar, M.J. Halliday, I.S. Bhandari, and R. Chillarege. In-process evaluation for software inspection and test. *IEEE Transactions on Software Engineering*, 19(11):1055–1070, November 1993.

[4] D. M. Cohen, S. R. Dalal, M. L. Fredman, and G. C. Patton. The AETG system: An approach to testing based on combinatiorial design. *IEEE Transactions on Software Engineering*, 23(7):437–444, July 1997.

[5] P. G. Frankl and O. Iakounenko. Further empirical studies of test effectiveness. In *Proceedings of the ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering (FSE-98)*, volume 23, 6 of *Software Engineering Notes*, pages 153–162, New York, Nov. 3–5 1998. ACM Press.

[6] T. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the 20th International Conference on Software Engineering*, pages 188–197. IEEE Computer Society Press, April 1998.

[7] P. A. Hausler, R. C. Linger, and C. J. Trammell. Adopting cleanroom software engineering with a phased approach. *IBM Systems Journal*, March 1994.

[8] Independent Assesment Team. Mars program independent assessment team summary report. Technical report, 2000.

[9] A. Jaaksi. Assessing software projects: Tools for business owners. In *Proceedings of the 9th European Software Engineering Conference held jointly with 10th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE 2003)*, pages 15–18. ACM Press, September 2003.

[10] J.-M. Kim and A. Porter. A history-based test prioritization technique for regression testing in resource constrained environments. In *Proceedings of the International Conference on Software Engineering (ICSE 2002)*, May 2002.

[11] J. L. Lions. Ariane 5, flight 501 failure, report by the inquiry board. Technical report, 1996.

[12] B. Marick. *The Craft of Software Testing: Subsystems Testing Including Object-Based and Object-Oriented Testing*. Prentice-Hall, 1997.

[13] J. Nielsen. *Designing Web Usability: The Practice of Simplicity*. New Riders Publishing, Indianapolis, 2000.

[14] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Commun. ACM*, 31(6):676–686, June 1988.