3 Testing in the Software Life Cycle

This chapter explains the role of testing in the entire life cycle of a software system, using the general V-model as a reference. Furthermore, we look at test levels and the test types that are used during development.

Each project in software development should be planned and executed using a life cycle model chosen in advance. Some important models were presented and explained in section 2.2. Each of these models implies certain views on software testing. From the viewpoint of testing, the general V-model according to [Boehm 79] plays an especially important role.

The role of testing within life cycle models

The V-model shows that testing activities are as valuable as development and programming. This has had a lasting influence on the appreciation of software testing. Not only every tester but every developer as well should know this general V-model and the views on testing it implies. Even if a different development model is used on a project, the principles presented in the following sections can be transferred and applied.

3.1 The General V-Model

The main idea behind the general V-model is that development and testing tasks are corresponding activities of equal importance. The two branches of the *V* symbolize this.

The left branch represents the development process. During development, the system is gradually being designed and finally programmed. The right branch represents the integration and testing process; the program elements are successively being assembled to form larger subsystems (integration), and their functionality is tested. →Integration and testing end when the acceptance test of the entire system has been completed. Figure 3-1 shows such a V-model.[1]

1. The V-model is used in many different versions. The names and the number of levels vary in literature and the enterprises using it.

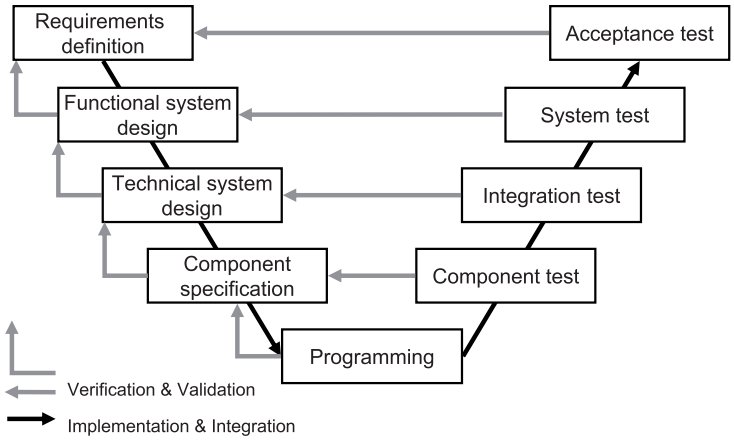


Figure 3–1
The general V-model

The constructive activities of the left branch are the activities known from the waterfall model:

- **→Requirements definition**
  The needs and requirements of the customer or the future system user are gathered, specified, and approved. Thus, the purpose of the system and the desired characteristics are defined.
- **Functional system design**
  This step maps requirements onto functions and dialogues of the new system.
- **Technical system design**
  This step designs the implementation of the system. This includes the definition of interfaces to the system environment and decomposing the system into smaller, understandable subsystems (system architecture). Each subsystem can then be developed as independently as possible.
- **Component specification**
  This step defines each subsystem, including its task, behavior, inner structure, and interfaces to other subsystems.
- **Programming**
  Each specified component (module, unit, class) is coded in a programming language.

Through these construction levels, the software system is described in more and more detail. Mistakes can most easily be found at the abstraction level where they occurred.

Thus, for each specification and construction level, the right branch of the V-model defines a corresponding test level:

- **Component test**
  (see section 3.2) verifies whether each software →component correctly fulfills its specification.
- **→Integration test**
  (see section 3.3) checks if groups of components interact in the way that is specified by the technical system design.
- **System test**
  (see section 3.4) verifies whether the system as a whole meets the specified requirements.
- **→Acceptance test**
  (see section 3.5) checks if the system meets the customer requirements, as specified in the contract and/or if the system meets user needs and expectations.

Within each test level, the tester must make sure the outcomes of development meet the requirements that are relevant or specified on this specific level of abstraction. This process of checking the development results according to their original requirements is called →validation.

Does a product solve the intended task?

When validating,[2] the tester judges whether a (partial) product really solves the specified task and whether it is fit or suitable for its intended use.

Is it the right system?

The tester investigates to see if the system makes sense in the context of intended product use.

Does a product fulfill its specification?

In addition to validation testing, the V-model requires verification[3] testing. Unlike validation, →verification refers to only one single phase of the development process. Verification shall assure that the outcome of a particular development level has been achieved correctly and completely, according to its specification (the input documents for that development level).

Is the system correctly built?

Verification activities examine whether specifications are correctly implemented and whether the product meets its specification, but not whether the resulting product is suitable for its intended use.

2. To validate: to affirm, to declare as valid, to check if something is valid.
3. To verify: to prove, to inspect.

In practice, every test contains both aspects. On higher test levels the validation part increases. To summarize, we again list the most important characteristics and ideas behind the general V-model:

Characteristics of the general V-model

- Implementation and testing activities are separated but are equally important (left side / right side).
- The *V* illustrates the testing aspects of verification and validation.
- We distinguish between different test levels, where each test level is testing "against" its corresponding development level.

The V-model may give the impression that testing starts relatively late, after system implementation, but this is not the case. The test levels on the right branch of the model should be interpreted as levels of test execution. Test preparation (test planning, test analysis and design) starts earlier and is performed in parallel to the development phases on the left branch[4] (not explicitly shown in the V-model).

The differentiation of test levels in the V-model is more than a temporal subdivision of testing activities. It is instead defining technically very different test levels; they have different objectives and thus need different methods and tools and require personnel with different knowledge and skills. The exact contents and the process for each test level are explained in the following sections.

3.2 Component Test

3.2.1 Explanation of Terms

Within the first test level (component testing), the software units are tested systematically for the first time. The units have been implemented in the programming phase just before component testing in the V-model.

Depending on the programming language the developers used, these software units may be called by different names, such as, for example, modules and units. In object-oriented programming, they are called classes. The respective tests, therefore, are called →module tests, →unit tests (see [IEEE 1008]), and →class tests.

Component and component test

Generally, we speak of software units or components. Testing of a single software component is therefore called component testing.

4. The so-called W-model (see [Spillner 00]) is a more detailed model that explicitly shows this parallelism of development and testing.

Test basis

Component testing is based on component requirements, and the component design (or detailed design). If white box test cases will be developed or white box →test coverage will be measured, the source code can also be analyzed. However, the component behavior must be compared with the component specification.

3.2.2 Test objects

Typical test objects are program modules/units or classes, (database) scripts, and other software components. The main characteristic of component testing is that the software components are tested individually and isolated from all other software components of the system. The isolation is necessary to prevent external influences on components. If testing detects a problem, it is definitely a problem originating from the component under test itself.

Component test examines component internal aspects

The component under test may also be a unit composed of several other components. But remember that aspects internal to the components are examined, not the components' interaction with neighboring components. The latter is a task for integration tests.

Component tests may also comprise data conversion and migration components. Test objects may even be configuration data and database components.

3.2.3 Test Environment

Component testing as the lowest test level deals with test objects coming "right from the developer's desk." It is obvious that in this test level there is close cooperation with development.

Example: Testing of a class method

In the VSR subsystem *DreamCar*, the specification for calculating the price of the car states the following:

- The starting point is `baseprice` minus `discount`, where `baseprice` is the general basic price of the vehicle and `discount` is the discount to this price granted by the dealer.
- A price (`specialprice`) for a special model and the price for extra equipment items (`extraprice`) shall be added.
- If three or more extra equipment items (which are not part of the special model chosen) are chosen (`extras`), there is a discount of 10 percent on these particular items. If five or more special equipment items are chosen, this discount is increased to 15 percent.

- The discount that is granted by the dealer applies only to the baseprice, whereas the discount on special items applies to the special items only. These discounts cannot be combined for everything.

The following C++-function calculates the total price:[5]

```
double calculate_price
   (double baseprice, double specialprice,
   double extraprice, int extras, double discount)
{
   double addon_discount;
   double result;

   if (extras >= 3) addon_discount = 10;
   else if (extras >= 5) addon_discount = 15;
   else addon_discount = 0;
   if (discount > addon_discount)
      addon_discount = discount;

   result = baseprice/100.0*(100-discount)
   + specialprice
   + extraprice/100.0*(100-addon_discount);
   return result;
}
```

In order to test the price calculation, the tester uses the corresponding class interface calling the function calculate_price() with appropriate parameters and data. Then the tester records the function's reaction to the function call. That means reading and recording the return value of the previous function call. For that, a →test driver is necessary. A test driver is a program that calls the component under test and then receives the test object's reaction.

For the test object calculate_price(), a very simple test driver could look like this:

```
bool test_calculate_price() {

double price;
bool test_ok = TRUE;

// testcase 01
price = calculate_price(10000.00,2000.00,1000.00,3,0);
test_ok = test_ok && (abs (price-12900.00) < 0.01);[6]
```

5. Actually, there is a defect in this program: Discount calculation for >= 5 is not reachable. The defect is used when explaining the use of white box analysis in chapter 5.
6. Floating point numbers should not be directly compared, as there may be imprecise rounding. As the result for price can be less than 12900.00, the absolute value of the difference of "price" and 12900.00 must be evaluated.

```
// testcase 02
price = calculate_price(25500.00,3450.00,6000.00,6,0);
test_ok = test_ok && (abs (price-34050.00) < 0.01);

// testcase ...

// test result
return test_ok;
}
```

The preceding test driver is programmed in a very simple way. Some useful extensions could be, for example, a facility to record the test data and the results, including date and time of the test, or a function that reads test cases from a table, file, or database.

To write test drivers, programming skills and knowledge of the component under test are necessary. The component's program code must be available. The tester must understand the test object (in the example, a class function) so that the call of the test object can be correctly programmed in the test driver. To write a suitable test driver, the tester must know the programming language and suitable programming tools must be available.

This is why the developers themselves usually perform the component testing. Although this is truly a component test, it may also be called developer test. The disadvantages of a programmer testing his own program were discussed in section 2.3.

Often, component testing is also confused with debugging. But debugging is not testing. Debugging is finding the cause of failures and removing them, while testing is the systematic approach for finding failures.

Hint

- Use of component testing frameworks (see [URL: xunit]) reduces the effort involved in programming test drivers and helps to standardize a project's component testing architecture. [Vigenschow 2010] demonstrates the use of these frameworks using examples of Junit for Java as well as nUnit and CppUnit for C++. Generic test drivers make it easier to use colleagues[7] who are not familiar with all details of the particular component and the programming environment for testing. Such test drivers can, for example, be used through a command interface and provide comfortable mechanisms for managing the test data and for recording and analyzing the tests. As all test data and test protocols are structured in a very similar way, this enables analysis of the tests across several components.

7. Sometimes, two programmers work together, each of them testing the components that their colleague has developed. This is called *buddy testing* or *code swaps*.

3.2.4 Test objectives

The test level called component test is not only characterized by the kind of test objects and the testing environment, the tester also pursues test objectives that are specific for this phase.

Testing the functionality

The most important task of component testing is to check that the entire functionality of the test object works correctly and completely as required by its specification (see →functional testing). Here, *functionality* means the input/output behavior of the test object. To check the correctness and completeness of the implementation, the component is tested with a series of test cases, where each test case covers a particular input/output combination (partial functionality).

Example: Test of the VSR price calculation

The test cases for the price calculation of *DreamCar* in the previous example very clearly show how the examination of the input/output behavior works. Each test case calls the test object with a particular combination of data; in this example, the price for the vehicle in combination with a different set of extra equipment items. It is then examined to see whether the test object, given this input data, calculates the correct price. For example, test case 2 checks the partial functionality of "discount with five or more special equipment items." If test case 2 is executed, we can see that the test object calculates the wrong total price. Test case 2 produces a failure. The test object does not completely meet the functional requirements.

Typical software defects found during functional component testing are incorrect calculations or missing or wrongly chosen program paths (e.g., special cases that were forgotten or misinterpreted).

Later, when the whole system is integrated, each software component must be able to cooperate with many neighboring components and exchange data with them. A component may then possibly be called or used in a wrong way, i.e., not in accordance with its specification. In such cases, the wrongly used component should not just suspend its service or cause the whole system to crash. Rather, it should be able to handle the situation in a reasonable and robust way.

Testing robustness

This is why testing for →robustness is another very important aspect of component testing. The way to do this is the same as in functional testing. However, the test focuses on items either not allowed or forgotten in the specification. The tests are function calls, test data, and special cases. Such test cases are also called →negative tests. The component's reaction should be an appropriate exception handling. If there is no such exception

handling, wrong inputs can trigger domain faults like division by zero or access to a null pointer. Such faults could lead to a program crash.

Example: Negative test

In the price calculation example, such negative tests are function calls with negative values, values that are far too large, or wrong data types (for example, `char` instead of `int`):[8]

```
// testcase 20
price = calculate_price(-1000.00,0.00,0.00,0,0);
test_ok = test_ok && (ERR_CODE == INVALID_PRICE);
…
// testcase 30
price = calculate_price("abc",0.00,0.00,0,0);
test_ok = test_ok && (ERR_CODE == INVALID_ARGUMENT);
```

Excursion

Some interesting aspects become clear:

- There are at least as many reasonable negative tests as positive ones.
- The test driver must be extended in order to be able to evaluate the test object's exception handling.
- The test object's exception handling (the analysis of `ERR_CODE` in the previous example) requires additional functionality. Often more than 50% of the program code deals with exception handling. Robustness has its cost.

Component testing should not only check functionality and robustness.

All the component's characteristics that have a crucial influence on its quality and that cannot be tested in higher test levels (or only with a much higher cost) should be checked during component testing. This may be nonfunctional characteristics like efficiency[9] and maintainability.

Efficiency test

Efficiency refers to how efficiently the component uses computer resources. Here we have various aspects such as use of memory, computing time, disk or network access time, and the time required to execute the component's functions and algorithms. In contrast to most other nonfunctional tests, a test object's efficiency can be measured during the test. Suitable criteria are measured exactly (e.g., memory usage in kilobytes, response times in milliseconds). Efficiency tests are seldom performed for all the components of a system. Efficiency is usually only verified in effi-

8. Depending on the compiler, data type errors can be detected during the compiling process.
9. The opportunity to use these types of checks on a component level instead of during a system test is not often exploited. This leads to efficiency problems only becoming visible shortly before the planned release date. Such problems can then only be corrected or attenuated at significant cost.

ciency-critical parts of the system or if efficiency requirements are explicitly stated by specifications. This happens, for example, in testing embedded software, where only limited hardware resources are available. Another example is testing real-time systems, where it must be guaranteed that the system follows given timing constraints.

Maintainability test

A maintainability test includes all the characteristics of a program that have an influence on how easy or how difficult it is to change the program or to continue developing it. Here, it is crucial that the developer fully understands the program and its context. This includes the developer of the original program who is asked to continue development after months or years as well as the programmer who takes over responsibility for a colleague's code. The following aspects are most important for testing maintainability: code structure, modularity, quality of the comments in the code, adherence to standards, understandability, and currency of the documentation.

Example: Code that is difficult to maintain

The code in the example `calculate_price()` is not good enough. There are no comments, and numeric constants are not declared but are just written into the code. If such a value must be changed later, it is not clear whether and where this value occurs in other parts of the system, nor is it clear how to find and change it.

Of course, such characteristics cannot be tested by →dynamic tests (see chapter 5). Analysis of the program text and the specifications is necessary. →Static testing, and especially reviews (see section 4.1) are the correct means for that purpose. However, it is best to include such analyses in the component test because the characteristics of a single component are examined.

3.2.5 Test Strategy

As we explained earlier, component testing is very closely related to development. The tester usually has access to the source code, which makes component testing the domain of white box testing (see section 5.2).

White box test

The tester can design test cases using her knowledge about the component's program structures, functions, and variables. Access to the program code can also be helpful for executing the tests. With the help of special tools (→debugger, see section 7.1.4), it is possible to observe program variables during test execution. This helps in checking for correct or incorrect behavior of the component. The internal state of a component

cannot only be observed; it can even be manipulated with the debugger. This is especially useful for robustness tests because the tester is able to trigger special exceptional situations.

Example: Code as test basis

Analyzing the code of `calculate_price()`, the following command can be recognized as a line that is relevant for testing:

```
if (discount > addon_discount)
      addon_discount = discount;
```

Additional test cases that lead to fulfilling the condition (`discount > addon_discount`) can easily be derived from the code. The specification of the price calculation contains no information about this situation; the implemented functionality is extra: it is not supposed to be there.

In reality, however, component testing is often done as a pure black box testing, which means that the code structure is not used to design test cases.[10] On the one hand, real software systems consist of countless elementary components; therefore, code analysis for designing test cases is probably only feasible with very few selected components.

On the other hand, the elementary components will later be integrated into larger units. Often, the tester only recognizes these larger units as units that can be tested, even in component testing. Then again, these units are already too large to make observations and interventions on the code level with reasonable effort. Therefore, integration and testing planning must answer the question of whether to test elementary parts or only larger units during component testing.

"Test first" development

Test first programming is a modern approach in component testing. The idea is to design and automate the tests first and program the desired component afterwards.

This approach is very iterative. The program code is tested with the available test cases. The code is improved until it passes the tests. This is also called test-driven development (see [Link 03]).

10. This is a serious flaw because 60 to 80% of the code often is never executed—a perfect hideout for bugs.

3.3 Integration Test

3.3.1 Explanation of Terms

After the component test, the second test level in the V-model is integration testing. A precondition for integration testing is that the test objects subjected to it (i.e., components) have already been tested. Defects should, if possible, already have been corrected.

Integration

Developers, testers, or special integration teams then compose groups of these components to form larger structural units and subsystems. This connecting of components is called integration.

Integration test

Then the structural units and subsystems must be tested to make sure all components collaborate correctly. Thus, the goal of the integration test is to expose faults in the interfaces and in the interaction between integrated components.

Test basis

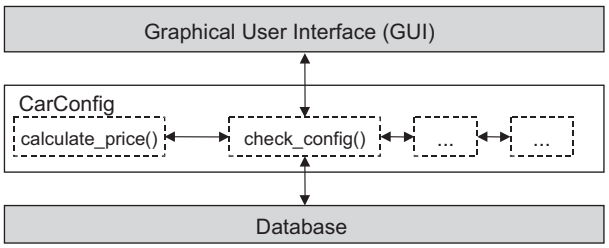
The test basis may be the software and system design or system architecture, or workflows through several interfaces and use cases.

Why is integration testing necessary if each individual component has already been tested? The following example illustrates the problem.

Example: Integration test VSR-DreamCar

The VSR subsystem *DreamCar* (see figure 2-1) consists of several elementary components.

Figure 3–2
Structure of the subsystem VSR-DreamCar



One element is the class `CarConfig` with the methods `calculate_price()`, `check_config()`, and other methods. `check_config()` retrieves all the vehicle data from a database and presents them to the user through a graphical user interface (GUI). From the user's point of view, this looks like figure 3-3.

When the user has chosen the configuration of a car, `check_config()` executes a plausibility check of the configuration (base model of the vehicle, special equipment, list of further extra items) and then calculates the price. In this example (see figure 3-3), the total resulting price from the base model of the chosen vehicle, the special model, and the extra equipment should be $29,000 + $1,413 + $900 = $31,313.

However, the price indicated is only $30,413. Obviously, in the current program version, accessories (e.g., alloy rims) can be selected without paying for them. Somewhere between the GUI and `calculate_price()`, the fact that alloy rims were chosen gets lost.

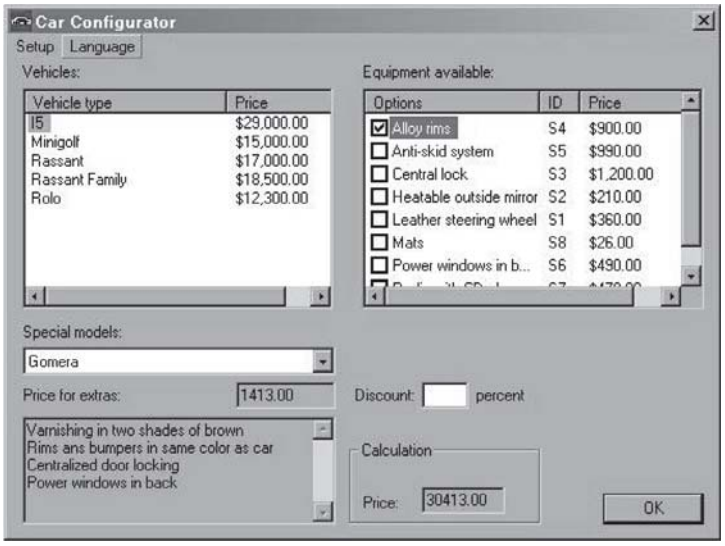


Figure 3–3
User interface for the VSR subsystem DreamCar

If the test protocols of the previous component tests show that the fault is neither in the function `calculate_price()` nor in `check_config()`, the cause of the problem could be a faulty data transmission between the GUI and `check_config()` or between `check_config()` and `calculate_price()`.

Even if a complete component test had been executed earlier, such interface problems can still occur. Because of this, integration testing is necessary as a further test level. Its task is to find collaboration and interoperability problems and isolate their causes.

Example:
VSR integration test

Integration of the single components to the subsystem *DreamCar* is just the beginning of the integration test in the project VSR. The other subsystems of the VSR (see chapter 2, figure 2-1) must also be integrated. Then, the subsystems must be connected to each other. *DreamCar* has to be connected to the subsystem *ContractBase*, which is connected to the subsystems *JustInTime* (order management), *NoRisk* (vehicle insurance), and *EasyFinance* (financing). In one of the last steps of integration, VSR is connected to the external mainframe in the IT center of the enterprise.

Integration testing in the large

As the example shows, interfaces to the system environment (i.e., external systems) are also subject to integration and integration testing. When interfaces to external software systems are examined, we sometimes speak of →system integration testing, higher-level integration testing, or integration testing in the large (integration of components is then integration test in the small, sometimes called →component integration testing). System integration testing can be executed only after system testing. The development team has only one-half of such an external interface under its control. This constitutes a special risk. The other half of the interface is determined by an external system. It must be taken as it is, but it is subject to unexpected change. Passing a system integration test is no guarantee that the system will function flawlessly in the future.

Integration levels

Thus, there may be several integration levels for test objects of different sizes. Component integration tests will test the interfaces between internal components or between internal subsystems. System integration tests focus on testing interfaces between different systems and between hardware and software. For example, if business processes are implemented as a workflow through several interfacing systems and problems occur, it may be very expensive and challenging to find the defect in a special component or interface.

3.3.2 Test objects

Assembled components

Step-by-step, during integration, the different components are combined to form larger units (see section 3.3.5). Ideally, there should be an integration test after each of these steps. Each subsystem may then be the basis for integrating further larger units. Such units (subsystems) may be test objects for the integration test later.

External systems or acquired components

In reality, a software system is seldom developed from scratch. Usually, an existing system is changed, extended, or linked to other systems (for example database systems, networks, new hardware). Furthermore, many system components are →commercial off-the-shelf (COTS) software products (for example, the database in *DreamCar*). In component testing, such existing or standard components are probably not tested. In the integration test, however, these system components must be taken into account and their collaboration with other components must be examined.

The most important test objects of integration testing are internal interfaces between components. Integration testing may also comprise configuration programs and configuration data. Finally, integration or

system integration testing examines subsystems for correct database access and correct use of other infrastructure components.

3.3.3 The Test Environment

As with component testing, test drivers are needed in the integration test. They send test data to the test objects, and they receive and log the results. Because the test objects are assembled components that have no interfaces to the "outside" other than their constituting components, it is obvious and sensible to reuse the available test drivers for component testing.

Reuse of the test environment

If the component test was well organized, then some test drivers should be available. It could be one generic test driver for all components or at least test drivers that were designed with a common architecture and are compatible with each other. In this case, the testers can reuse these test drivers without much effort.

If a component test is poorly organized, there may be usable test drivers for only a few of the components. Their user interface may also be completely different, which will create trouble. During integration testing in a much later stage of the project, the tester will need to put a lot of effort into the creation, change, or repair of the test environment. This means that valuable time needed for test execution is lost.

Monitors are necessary

During integration testing, additional tools, called monitors, are required. →Monitors are programs that read and log data traffic between components. Monitors for standard protocols (e.g., network protocols) are commercially available. Special monitors must be developed for the observation of project-specific component interfaces.

3.3.4 Test objectives

Wrong interface formats

The test objectives of the test level integration test are clear: to reveal interface problems as well as conflicts between integrated parts.

Problems can arise when attempting to integrate two single components. For example, their interface formats may not be compatible with each other because some files are missing or because the developers have split the system into completely different components than specified (chapter 4 covers static testing, which may help finding such issues).

Typical faults in data exchange

The harder-to-find problems, however, are due to the execution of the connected program parts. These kinds of problems can only be found by dynamic testing. They are faults in the data exchange or in the communication between the components, as in the following examples:

- A component transmits syntactically incorrect or no data. The receiving component cannot operate or crashes (functional fault in a component, incompatible interface formats, protocol faults).
- The communication works but the involved components interpret the received data differently (functional fault of a component, contradicting or misinterpreted specifications).
- Data is transmitted correctly but at the wrong time, or it is late (timing problem), or the intervals between the transmissions are too short (throughput, load, or capacity problem).

Example: Integration problems in VSR

The following interface failures could occur during the VSR integration test. These can be attributed to the previously mentioned failure types:

- In the GUI of the *DreamCar* subsystem, selected extra equipment items are not passed on to `check_config()`. Therefore, the price and the order data would be wrong.
- In *DreamCar,* a certain code number (e.g., 442 for metallic blue) represents the color of the car. In the order management system running on the external mainframe, however, some code numbers are interpreted differently (there, for example, 442 may represent red). An order from the VSR, seen there as correct, would lead to delivery of the wrong product.
- The mainframe computer confirms an order after checking whether delivery would be possible. In some cases, this examination takes so long that the VSR assumes a transmission failure and aborts the order. A customer who has carefully chosen her car would not be able to order it.

None of these failures can be found in the component test because the resulting failures occur only in the interaction between two software components.

Nonfunctional tests may also be executed during integration testing, if attributes mentioned below are important or are considered at risk. These attributes may include reliability, performance, and capacity.

Can the component test be omitted?

Is it possible to do without the component test and execute all the test cases after integration is finished? Of course, this is possible, and in practice it is regretfully often done, but only at the risk of great disadvantages:

- Most of the failures that will occur in a test designed like this are caused by functional faults within the individual components. An implicit component test is therefore carried out, but in an environment that is not suitable and that makes it harder to access the individual components.

- Because there is no suitable access to the individual component, some failures cannot be provoked and many faults, therefore, cannot be found.
- If a failure occurs in the test, it can be difficult or impossible to locate its origin and to isolate its cause.

The cost of trying to save effort by cutting the component test is finding fewer of the existing faults and experiencing more difficulty in diagnosis. Combining a component test with a subsequent integration test is more effective and efficient.

3.3.5 Integration Strategies

In which order should the components be integrated in order to execute the necessary test work as efficiently—that is, as quickly and easily—as possible? Efficiency is the relation between the cost of testing (the cost of test personnel and tools, etc.) and the benefit of testing (number and severity of the problems found) in a certain test level.

The test manager has to decide this and choose and implement an optimal integration strategy for the project.

Components are completed at different times

In practice, different software components are completed at different times, weeks or even months apart. No project manager or test manager can allow testers to sit around and do nothing while waiting until all the components are developed and they are ready to be integrated.

An obvious ad hoc strategy to quickly solve this problem is to integrate the components in the order in which they are ready. This means that as soon as a component has passed the component test, it is checked to see if it fits with another already tested component or if it fits into a partially integrated subsystem. If so, both parts are integrated and the integration test between both of them is executed.

Example: Integration Strategy in the VSR project

In the VSR project, the central subsystem *ContractBase* turns out to be more complex than expected. Its completion is delayed for several weeks because the work on it costs much more than originally expected. To avoid losing even more time, the project manager decides to start the tests with the available components *DreamCar* and *NoRisk*. These do not have a common interface, but they exchange data through *ContractBase*. To calculate the price of the insurance, *NoRisk* needs to know which type of vehicle was chosen because this determines the price and other parameters of the insurance. As a temporary replacement for *ContractBase*, a →stub is programmed. The stub receives simple car configuration data from *DreamCar*, then determines the vehicle type code from this data and passes it on

to NoRisk. Furthermore, the stub makes it possible to put in different relevant data about the customer. *NoRisk* calculates the insurance price from the data and shows it in a window so it can be checked. The price is also saved in a test log. The stub serves as a temporary replacement for the still missing subsystem *ContractBase*.

This example makes it clear that the earlier the integration test is started (in order to save time), the more effort it will take to program the stubs. The test manager has to choose an integration strategy in order to optimize both factors (time savings vs. cost for the testing environment).

Constraints for integration

Which strategy is optimal (the most timesaving and least costly strategy) depends on the individual circumstances in each project. The following items must be analyzed:

- The **system architecture** determines how many and which components the entire system consists of and in which way they depend on each other.
- The **project plan** determines at what time during the course of the project the parts of the system are developed and when they should be ready for testing. The test manager should be consulted when determining the order of implementation.
- The **test plan** determines which aspects of the system shall be tested, how intensely, and on which test level this has to happen.

Discuss the integration strategy

The test manager, taking into account these general constraints, has to design an optimal integration strategy for the project. Because the integration strategy depends on delivery dates, the test manager should consult the project manager during project planning. The order of component implementation should be suitable for integration testing.

Generic strategies

When making plans, the test manager can follow these generic integration strategies:

- **Top-down integration**
  The test starts with the top-level component of the system that calls other components but is not called itself (except for a call from the operating system). Stubs replace all subordinate components. Successively, integration proceeds with lower-level components. The higher level that has already been tested serves as test driver.
  - Advantage: Test drivers are not needed, or only simple ones are required, because the higher-level components that have already been tested serve as the main part of the test environment.

- Disadvantage: Stubs must replace lower-level components not yet integrated. This can be very costly.

- **Bottom-up integration**
  The test starts with the elementary system components that do not call further components, except for functions of the operating system. Larger subsystems are assembled from the tested components and then tested.
  - Advantage: No stubs are needed.
  - Disadvantage: Test drivers must simulate higher-level components.
- **Ad hoc integration**
  The components are being integrated in the (casual) order in which they are finished.
  - Advantage: This saves time because every component is integrated as early as possible into its environment.
  - Disadvantage: Stubs as well as test drivers are required.
- **Backbone integration**
  A skeleton or backbone is built and components are gradually integrated into it [Beizer 90].
  - Advantage: Components can be integrated in any order.
  - Disadvantage: A possibly labor-intensive skeleton or backbone is required.

Top-down and Bottom-up integration in their pure form can be applied only to program systems that are structured in a strictly hierarchical way; in reality, this rarely occurs. This is the reason a more or less individualized mix of the previously mentioned integration strategies[11] might be chosen.

Avoid the big bang!

Any nonincremental integration—also called →big bang integration—should be avoided. Big bang integration means waiting until all software elements are developed and then throwing everything together in one step. This typically happens due to the lack of an integration strategy. In the worst cases, even component testing is skipped. There are obvious disadvantages of this approach:

- The time leading up to the big bang is lost time that could have been spent testing. As testing always suffers from lack of time, no time that could be used for testing should be wasted.

11. Special integration strategies can be followed for object-oriented, distributed, and real-time systems (see [Winter 98], [Bashir 99], [Binder 99]).

- All the failures will occur at the same time. It will be difficult or impossible to get the system to run at all. It will be very difficult and time-consuming to localize and correct defects.

3.4 System Test

3.4.1 Explanation of Terms

After the integration test is completed, the third and next test level is the system test. System testing checks if the integrated product meets the specified requirements. Why is this still necessary after executing component and integration tests? The reasons for this are as follows:

Reasons for system test

- In the lower test levels, the testing was done against technical specifications, i.e., from the technical perspective of the software producer. The system test, though, looks at the system from the perspective of the customer and the future user.[12] The testers validate whether the requirements are completely and appropriately implemented.
- Many functions and system characteristics result from the interaction of all system components; consequently, they are visible only when the entire system is present and can be observed and tested only there.

Example: VSR-System tests

The main purpose of the VSR-System is to make ordering a car as easy as possible.

While ordering a car, the user uses all the components of the VSR-System: the car is configured (*DreamCar*), financing and insurance are calculated (*EasyFinance*, *NoRisk*), the order is transmitted to production (*JustInTime*), and the contracts are archived (*ContractBase*). The system fulfills its purpose only when all these system functions and all the components collaborate correctly. The system test determines whether this is the case.

The test basis includes all documents or information describing the test object on a system level. This may be system requirements, specifications, risk analyses if present, user manuals, etc.

12. The customer (who has ordered and paid for the system) and the user (who uses the system) can be different groups of people or organizations with their own specific interests and requirements for the system.

3.4.2 Test Objects and Test Environment

After the completion of the integration test, the software system is complete. The system test tests the system as a whole in an environment as similar as possible to the intended →production environment.

Instead of test drivers and stubs, the hardware and software products that will be used later should be installed on the test platform (hardware, system software, device driver software, networks, external systems, etc.). Figure 3-4 shows an example of the VSR-System test environment.

The system test not only tests the system itself, it also checks system and user documentation, like system manuals, user manuals, training material, and so on. Testing configuration settings as well as optimizing the system configuration during load and performance testing (see section 3.7.2) must often be covered.

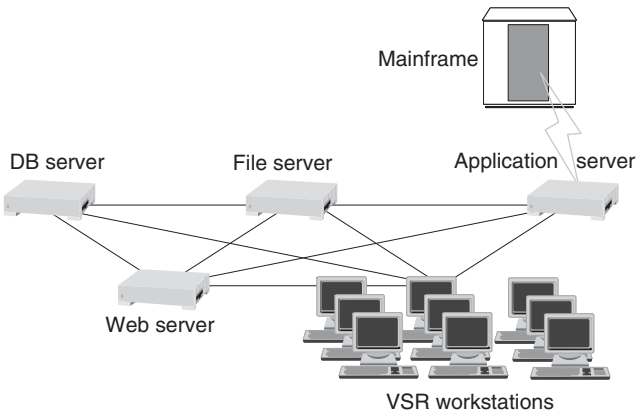


Figure 3–4
Example of a system test environment

It is getting more and more important to check the quality of data in systems that use a database or large amounts of data. This should be included in the system test. The data itself will then be new test objects. It must be assured that it is consistent, complete, and up-to-date. For example, if a system finds and displays bus connections, the station list and schedule data must be correct.

→*data quality*

One mistake is commonly made to save costs and effort: instead of the system being tested in a separate environment, the system test is executed in the customer's operational environment. This is detrimental for a couple of reasons:

System test requires a separate test environment

- During system testing, it is likely that failures will occur, resulting in damage to the customer's operational environment. This may lead to expensive system crashes and data loss in the production system.
- The testers have only limited or no control over parameter settings and the configuration of the operational environment. The test conditions may change over time because the other systems in the customer's environment are running simultaneously with the test. The system tests that have been executed cannot be reproduced or can only be reproduced with difficulty (see section 3.7.4 on regression testing).

System test effort is often underestimated

The effort of an adequate system test must not be underestimated, especially because of the complex test environment. [Bourne 97] states the experience that at the beginning of the system test, only half of the testing and quality control work has been done (especially when a client/server system is developed, as in the VSR-example).

3.4.3 Test Objectives

It is the goal of the system test to validate whether the complete system meets the specified functional and nonfunctional requirements (see sections 3.7.1 and 3.7.2) and how well it does that. Failures from incorrect, incomplete, or inconsistent implementation of requirements should be detected. Even undocumented or forgotten requirements should be identified.

3.4.4 Problems in System Test Practice

Excursion

In (too) many projects, the requirements are incompletely or not at all written down. The problem this poses for testers is that it's unclear how the system is supposed to behave. This makes it hard to find defects.

Unclear system requirements

If there are no requirements, then all behaviors of a system would be valid and assessment would be impossible. Of course, the users or the customers have a certain perception of what they expect of "their" software system. Thus, there *must be* requirements. Yet sometimes these requirements are not written down anywhere; they exist only in the minds of a few people who are involved in the project. The testers then have the undesirable role of gathering information about the required behavior after the fact. One possible technique to cope with such a situation is exploratory testing (see section 5.3, and for more detailed discussion, [Black 02]).

Missed decisions

While the testers identify the original requirements, they will discover that different people may have completely different views and ideas on the same subject. This is not surprising if the requirements have never been documented, reviewed, or released during the project. The consequences for those responsible for system testing are less desirable: They must collect information on the requirements; they also have to make decisions that should have been made many months earlier. This collection of information may be very costly and time consuming. Test completion and release of the completed system will surely be delayed.

Project fail

If the requirements are not specified, of course the developers do not have clear objectives either. Thus, it is very unlikely that the developed system will meet the implicit requirements of the customer. Nobody can seriously expect that it is possible to develop a usable system given these conditions. In such projects, execution of the system test can probably only announce the collapse of the project.

3.5 Acceptance Test

All the test levels described thus far represent testing activities that are under the producer's responsibility. They are executed before the software is presented to the customer or the user.

Before installing and using the software in real life (especially for software developed individually for a customer), another last test level must be executed: the acceptance test. Here, the focus is on the customer's and user's perspective. The acceptance test may be the only test that the customers are actually involved in or that they can understand. The customer may even be responsible for this test!

→Acceptance tests may also be executed as a part of lower test levels or be distributed over several test levels:

- A commercial-off-the-shelf product (COTS) can be checked for acceptance during its integration or installation.
- Usability of a component can be acceptance tested during its component test.
- Acceptance of new functionality can be checked on prototypes before system testing.

There are four typical forms of acceptance testing:

- Contract acceptance testing
- User acceptance testing
- Operational acceptance testing
- Field testing (alpha and beta testing)

How much acceptance testing?

How much acceptance testing should be done is dependent on the product risk. This may be very different. For customer-specific systems, the risk is high and a comprehensive acceptance test is necessary. At the other extreme, if a piece of standard software is introduced, it may be sufficient to install the package and test a few representative usage scenarios. If the system interfaces with other systems, collaboration of the systems through these interfaces must be tested.

Test basis

The test basis for acceptance testing can be any document describing the system from the user or customer viewpoint, such as, for example, user or system requirements, use cases, business processes, risk analyses, user process descriptions, forms, reports, and laws and regulations as well as descriptions of maintenance and system administration rules and processes.

3.5.1 Contract Acceptance Testing

If customer-specific software was developed, the customer will perform contract acceptance testing (in cooperation with the vendor). Based on the results, the customer considers whether the software system is free of (major) deficiencies and whether the service defined by the development contract has been accomplished and is acceptable. In case of internal software development, this can be a more or less formal contract between the user department and the IT department of the same enterprise.

Acceptance criteria

The test criteria are the acceptance criteria determined in the development contract. Therefore, these criteria must be stated as unambiguously as possible. Additionally, conformance to any governmental, legal, or safety regulations must be addressed here.

In practice, the software producer will have checked these criteria within his own system test. For the acceptance test, it is then enough to rerun the test cases that the contract requires as relevant for acceptance, demonstrating to the customer that the acceptance criteria of the contract have been met.

Because the supplier may have misunderstood the acceptance criteria, it is very important that the acceptance test cases are designed by or at least thoroughly reviewed by the customer.

Customer (site) acceptance test

In contrast to system testing, which takes place in the producer environment, acceptance testing is run in the customer's actual operational environment.[13] Due to these different testing environments, a test case that worked correctly during the system test may now suddenly fail. The acceptance test also checks the delivery and installation procedures. The acceptance environment should be as similar as possible to the later operational environment. A test in the operational environment itself should be avoided to minimize the risk of damage to other software systems used in production.

The same techniques used for test case design in system testing can be used to develop acceptance test cases. For administrative IT systems, business transactions for typical business periods (like a billing period) should be considered.

3.5.2 Testing for User Acceptance

Another aspect concerning acceptance as the last phase of validation is the test for user acceptance. Such a test is especially recommended if the customer and the user are different.

Example: Different user groups

In the VSR example, the responsible customer is a car manufacturer. But the car manufacturer's shops will use the system. Employees and customers who want to purchase cars will be the system's end users. In addition, some clerks in the company's headquarter will work with the system, e.g., to update price lists in the system.

Get acceptance of every user group

Different user groups usually have completely different expectations of a new system. Users may reject a system because they find it "awkward" to use, which can have a negative impact on the introduction of the system. This may happen even if the system is completely OK from a functional point of view. Thus, it is necessary to organize a user acceptance test for each user group. The customer usually organizes these tests, selecting test cases based on business processes and typical usage scenarios.

13. Sometimes the acceptance test consists of two cycles: the first in the system test environment, the second in the customer's environment.

Present prototypes to the users early

If major user acceptance problems are detected during acceptance testing, it is often too late to implement more than cosmetic countermeasures. To prevent such disasters, it is advisable to let a number of representatives from the group of future users examine prototypes of the system early.

3.5.3 Operational (Acceptance) Testing

Operational (acceptance) testing assures the acceptance of the system by the system administrators.[14] It may include testing of backup/restore cycles (including restoration of copied data), disaster recovery, user management, and checks of security vulnerabilities.

3.5.4 Field Testing

If the software is supposed to run in many different operational environments, it is very expensive or even impossible for the software producer to create a test environment for each of them during system testing. In such cases, the software producer may choose to execute a →field test after the system test. The objective of the field test is to identify influences from users' environments that are not entirely known or specified and to eliminate them if necessary. If the system is intended for the general market (a COTS system), this test is especially recommended.

Testing done by representative customers

For this purpose, the producer delivers stable prerelease versions of the software to preselected customers who adequately represent the market for this software or whose operational environments are appropriately similar to possible environments for the software.

These customers then either run test scenarios prescribed by the producer or run the product on a trial basis under realistic conditions. They give feedback to the producer about the problems they encountered along with general comments and impressions about the new product. The producer can then make the specific adjustments.

Alpha and beta testing

Such testing of preliminary versions by representative customers is also called →alpha testing or →beta testing. Alpha tests are carried out at the producer's location, while beta tests are carried out at the customer's site.

A field test should not replace an internal system test run by the producer (even if some producers do exactly this). Only when the system test

14. This verifies that the system complies with the needs of the system administrators.

has proven that the software is stable enough should the new product be given to potential customers for a field test.

Dogfood test

A new term in software testing is *dogfood test*. It refers to a kind of internal field testing where the product is distributed to and used by internal users in the company that developed the software. The idea is that "if you make dogfood, try it yourself first." Large suppliers of software like Microsoft and Google advocate this approach before beta testing.

3.6 Testing New Product Versions

Until now, it was assumed that a software development project is finished when the software passes the acceptance test and is deployed. But that's not the reality. The first deployment marks only the beginning of the software life cycle. Once it is installed, it will often be used for years or decades and is changed, updated, and extended many times. Each time that happens, a new →version of the original product is created. The following sections explain what must be considered when testing such new product versions.

3.6.1 Software Maintenance

Software does not wear out. Unlike with physical industry products, the purpose of software maintenance is not to maintain the ability to operate or to repair damages caused by use. Defects do not originate from wear and tear. They are design faults that already exist in the original version. We speak of software maintenance when a product is adapted to new operational conditions (adaptive maintenance, updates of operating systems, databases, middleware) or when defects that have been in the product before are corrected (corrective maintenance). Testing changes made during maintenance can be difficult because the system's specifications are often out of date or missing, especially in the case of legacy systems.

Example: Analysis of VSR hotline requests

The VSR-System has been distributed and installed after intense testing. In order to find areas with weaknesses that had not been found previously, the central hotline generates an analysis of all requests that have come in from the field. Here are some examples:

1. A few dealers use the system on an unsupported platform with an old version of the operating system. In such environments, sometimes the host access causes system crashes.
2. Many customers consider the selection of extra equipment to be awkward, especially when they want to compare prices between different packages of

extra equipment. Many users would therefore like to save equipment configurations and to be able to retrieve them after a change.
3. Some of the seldom-occurring insurance prices cannot be calculated at all because the corresponding calculation wasn't implemented in the insurance component.
4. Sometimes, even after more than 15 minutes, a car order is not yet confirmed by the server. The system cuts the connection after 15 minutes to avoid having unused connections remain open. The customers are angry with this because they waste a lot of time waiting in vain for confirmation of the purchase order. The dealer then has to repeat inputting the order and then has to mail the confirmation to the customer.

Problem 1 is the responsibility of the dealer because he runs the system on a platform for which it was not intended. Still, the software producer might change the program to allow it to be run on this platform to, for example, save the dealer from the cost of a hardware upgrade.

Problems like number 2 will always arise, regardless of how well and completely the requirements were originally analyzed. The new system will generate many new experiences and therefore new requirements will naturally arise.

Improve the test plan

Problem 3 could have been detected during system testing. But testing cannot guarantee that a system is completely fault free. It can only provide a sample with a certain probability to reveal failures. A good test manager will analyze which kind of testing would have detected this problem and will adequately improve or adapt the test plan.

Problem 4 had been detected in the integration test and had been solved. The VSR-System waits for a confirmation from the server for more than 15 minutes without cutting the connection. The long waiting time happens in special cases, when certain batch processes are run in the host computer. The fact that the customer does not want to wait in the shop for such a long time is another subject.

These four examples represent typical problems that will be found in even the most mature software system:

1. The system is run under new operating conditions that were not predictable and not planned.
2. The customers express new wishes.
3. Functions are necessary for rarely occurring special cases that were not anticipated.
4. Crashes that happen rarely or only after a very long run time are reported. These are often caused by external influences.

Therefore, after its deployment, every software system requires certain corrections and improvements. In this context, we speak of software mainte-

nance. But the fact that maintenance is necessary in any case must not be used as a pretext for cutting down on component, integration, or system testing. We sometime hear, "We must continuously publish updates anyway, so we don't need to take testing so seriously, even if we miss defects." Managers behaving this way do not understand the true costs of failures.

Testing after maintenance work

If the production environment has been changed or the system is ported to a new environment (for example, by migration to a new platform), a new acceptance test should be run by the organization responsible for operations. If data has to be migrated or converted, even this aspect must be tested for correctness and completeness.

Otherwise, the test strategy for testing a changed system is the same as for testing every new product version: Every new or changed part of the code must be tested. Additionally, in order to avoid side effects, the remainder of the system should be regression tested (see section 3.7.4) as comprehensibly as possible. The test will be easier and more successful if even maintenance releases are planned in advance and considered in the test plans.

There should be two strategies: one for emergency fixes (or "hot fixes") and one for planned releases. For an ordinary release, a test approach should be planned early, comprising thorough testing of anything new or changed as well as regression testing. For an emergency fix, a minimal test should be executed to minimize the time to release. Then the normal comprehensive test should be executed as soon as possible afterwards.

Testing before retirement

If a system is scheduled for retirement, then some testing is also useful.

Testing for the retirement of a system should include the testing of data archiving or data migration into the future system.

3.6.2 Testing after Further Development

Apart from maintenance work necessary because of failures, there will be changes and extensions to the product that project management has intended from the beginning.

Example: Planning of the VSR development

In the development plan for VSR release 2, the following work is scheduled:

1. New communication software will be installed on the host in the car manufacturer's computing center; therefore, the VSR communication module must be adapted.

2. Certain system extensions that could not be finished in release 1 will now be delivered in release 2.
3. The installation base shall be extended to the EU dealer network. Therefore, specific adaptations necessary for each country must be integrated and all the manuals and the user interface must be translated.

These three tasks come neither from defects nor from unforeseen user requests. So they are not part of ordinary maintenance but instead normal further product development.

The first point results from a planned change of a neighbor system. Point 2 involves functionality that had been planned from the beginning but could not be implemented as early as intended. Point 3 represents extensions that become necessary in the course of a planned market expansion.

A software product is certainly not finished with the release of the first version. Additional development is continuously occurring. An improved product version will be delivered at certain intervals, such as, e.g., once a year. It is best to synchronize these →releases with the ongoing maintenance work. For example, every six months a new version is introduced: one maintenance update and one genuine functional update.

After each release, the project effectively starts over, running through all the project phases. This approach is called iterative software development. Nowadays this is the usual way of developing software.[15]

Testing new releases

How must testing respond to this? Do we have to completely rerun all the test levels for every release of the product? Yes, if possible! As with maintenance testing, anything new or changed should be tested, and the remainder of the system should be regression tested to find unexpected side effects (see section 3.7.4).

3.6.3 Testing in Incremental Development

Incremental development means that the project is not done in one (possibly large) piece but as a preplanned series of smaller developments and deliveries. System functionality and reliability will grow over time.

The objective of this is to make sure the system meets customer needs and expectations. The early releases allow customer feedback early and

15. This aspect is not shown in the general V-model. Only more modern life cycle models show iterations explicitly (see [Jacobson 99], [Beck 00], [Beedle 01]).

continuously. Examples of incremental models are Prototyping, Rapid Application Development (RAD) [Martin 91], Rational Unified Process (RUP), Evolutionary Development [Gilb 05], the Spiral Model [Boehm 86], and so-called agile development methods such as Extreme Programming (XP) [Beck 00], Dynamic Systems Development Method (DSDM) [Stapleton 02], and SCRUM [Beedle 01]. SCRUM has become more and more popular during recent years and is nowadays much used amongst agile approaches.

Testing must be adapted to such development models, and continuous integration testing and regression testing are necessary. There should be reusable test cases for every component and increment, and they should be reused and updated for every additional increment. If this is not the case, the product's reliability tends to decrease over time instead of increasing.

This danger can be reduced by running several V-models in sequence, one for each increment, where every next "V" reuses existing test material and adds the tests necessary for new development or for higher reliability requirements.

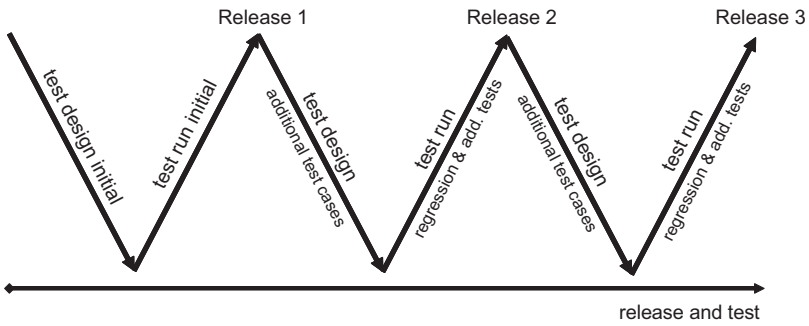


Figure 3–5
Testing in incremental development

3.7 Generic Types of Testing

The previous chapters gave a detailed view of testing in the software life cycle, distinguishing several test levels. Focus and objectives change when testing in these different levels. And different types of testing are relevant on each test level.

The following types of testing can be distinguished:

- Functional testing
- Nonfunctional testing
- Testing of software structure
- Testing related to changes

3.7.1 Functional Testing

Functional testing includes all kind of tests that verify a system's input/output behavior. To design functional test cases, the black box testing methods discussed in section 5.1 are used, and the test bases are the functional requirements.

Functional requirements

Functional requirements →specify the behavior of the system; they describe what the system must be able to do. Implementation of these requirements is a precondition for the system to be applicable at all. Characteristics of functionality, according to [ISO 9126], are suitability, accuracy, interoperability, and security.

Requirements definition

When a project is run using the V-model, the requirements are collected during the phase called "requirements definition" and documented in a requirements management system (see section 7.1.1). Text-based requirements specifications are still in use as well. Templates for this document are available in [IEEE 830].

The following text shows a part of the requirements paper concerning price calculation for the system VSR (see section 3.2.4).

Example: Requirements of the VSR-System

R 100: The user can choose a vehicle model from the current model list for configuration.

R 101: For a chosen model, the deliverable extra equipment items are indicated. The user can choose the desired individual equipment from this list.

R 102: The total price of the chosen configuration is continuously calculated from current price lists and displayed.

Requirements-based testing

Requirements-based testing uses the final requirements as the basis for testing. For each requirement, at least one test case is designed and documented in the test specification. The test specification is then reviewed. The testing of requirement 102 in the preceding example could look like the following example.

Example: Requirements-based testing

T 102.1: A vehicle model is chosen; its base price according to the sales manual is displayed.
T 102.2: A special equipment item is selected; the price of this accessory is added.
T 102.3: A special equipment item is deselected; the price falls accordingly.
T 102.4: Three special equipment items are selected; the discount comes into effect as defined in the specification.

Usually, more than one test case is needed to test a functional requirement.

Requirement 102 in the example contains several rules for different price calculations. These must be covered by a set of test cases (102.1–102.4 in the preceding example). Using black box test methods (e.g., →equivalence partitioning), these test cases can be further refined and extended if desired. The decisive fact is if the defined test cases (or a minimal subset of them) have run without failure, the appropriate functionality is considered validated.

Requirements-based functional testing as shown is mainly used in system testing and other higher levels of testing. If a software system's purpose is to automate or support a certain business process for the customer, business-process-based testing or use-case-based testing are other similar suitable testing methods (see section 5.1.5).

Example: Testing based on business process

From the dealer's point of view, VSR supports him in the sales process. The process can, for example, look like this:

- The customer selects a type of vehicle he is interested in from the available models.
- The customer gets the information about the type of extra equipment and prices and selects the desired car.
- The dealer suggests alternative ways of financing the car.
- The customer decides and signs the contract.

A business process analysis (which is usually elaborated as part of the requirements analysis) shows which business processes are relevant and how often and in which context they appear. It also shows which persons, enterprises, and external systems are involved. Test scenarios simulating typical business processes are constructed based on this analysis. The test

scenarios are prioritized using the frequency and the relevance of the particular business processes.

Requirements-based testing focuses on single system functions (e.g., the transmission of a purchase order). Business-process-based testing, however, focuses on the whole process consisting of many steps (e.g., the sales conversation, consisting of configuring a car, agreeing on the purchase contract, and the transmission of the purchase order). This means a sequence of several tests.

Of course, for the users of the *VirtualShowRoom* system, it is not enough to see if they can choose and then buy a car. More important for ultimate acceptance is often how easily they can use the system. This depends on how easy it is to work with the system, if it reacts quickly enough, and if it returns easily understood information. Therefore, along with the functional criteria, the nonfunctional criteria must also be checked and tested.

3.7.2 Nonfunctional Testing

→Nonfunctional requirements do not describe the functions; they describe the attributes of the functional behavior or the attributes of the system as a whole, i.e., "how well" or with what quality the (partial) system should work. Implementation of such requirements has a great influence on customer and user satisfaction and how much they enjoy using the product. Characteristics of these requirements are, according to [ISO 9126], reliability, usability, and efficiency. (For the new syllabus, which is effective from 2015, the basis is not ISO 9126 but ISO/IEC 25010:2011. Compatibility and security are added to the list of system characteristics.) Indirectly, the ability of the system to be changed and to be installed in new environments also has an influence on customer satisfaction. The faster and the easier a system can be adapted to changed requirements, the more satisfied the customer and the user will be. These two characteristics are also important for the supplier, because they help to reduce maintenance costs.

According to [Myers 79], the following nonfunctional system characteristics should be considered in the tests (usually in system testing):

- →**Load test:** Measuring of the system behavior for increasing system loads (e.g., the number of users that work simultaneously, number of transactions)
- →**Performance test:** Measuring the processing speed and response time for particular use cases, usually dependent on increasing load
- →**Volume test:** Observation of the system behavior dependent on the amount of data (e.g., processing of very large files)
- →**Stress test:** Observation of the system behavior when the system is overloaded
- **Testing of security** against unauthorized access to the system or data, denial of service attacks, etc.
- **Stability** or **reliability test:** Performed during permanent operation (e.g., mean time between failures or failure rate with a given user profile)
- →**Robustness test:** Measuring the system's response to operating errors, bad programming, hardware failure, etc. as well as examination of exception handling and recovery
- **Testing of compatibility and data conversion:** Examination of compatibility with existing systems, import/export of data, etc.
- **Testing of different configurations of the system:** For example, different versions of the operating system, user interface language, hardware platform, etc. (→back-to-back testing)
- **Usability test:** Examination of the ease of learning the system, ease and efficiency of operation, understandability of the system outputs, etc., always with respect to the needs of a specific group of users ([ISO 9241], [ISO 9126])
- **Checking of the documentation:** For compliance with system behavior (e.g., user manual and GUI)
- **Checking maintainability:** Assessing the understandability of the system documentation and whether it is up-to-date; checking if the system has a modular structure; etc.

A major problem in testing nonfunctional requirements is the often imprecise and incomplete expression of these requirements. Expressions like "the system should be easy to operate" and "the system should be fast" are not testable in this form.

Hint

- Representatives of the (later) system test personnel should participate in early requirement reviews and make sure that every nonfunctional requirement (as well as each functional one) can be measured and is testable.

Furthermore, many nonfunctional requirements are so fundamental that nobody really thinks about mentioning them in the requirements paper (presumed matters of fact).[16] Even such implicit characteristics must be validated because they may be relevant.

Example: Presumed requirements

The VSR-System is designed for use on a market-leading operating system. It is obvious that the recommended or usual user interface conventions are followed for the "look and feel" of the VSR GUI. The DreamCar GUI (see figure 3-3) violates these conventions in several aspects. Even if no particular requirement is specified, such deviations from "matter of fact requirements" can and must be seen as faults or defects.

Excursion: Testing nonfunctional requirements

In order to test nonfunctional characteristics, it makes sense to reuse existing functional tests. The nonfunctional tests are somehow "piggybacking" on the functional tests. Most nonfunctional tests are black box tests. An elegant general testing approach could look like this:

Scenarios that represent a cross section of the functionality of the entire system are selected from the functional tests. The nonfunctional property must be observable in the corresponding test scenario. When the test scenario is executed, the nonfunctional characteristic is measured. If the resulting value is inside a given limit, the test is considered "passed." The functional test practically serves as a vehicle for determining the nonfunctional system characteristics.

3.7.3 Testing of Software Structure

Structural techniques (→structure-based testing, white box testing) use information about the test object's internal code structure or architecture. Typically, the control flow in a component, the call hierarchy of procedures, or the menu structure is analyzed. Abstract models of the software may also be used. The objective is to design and run enough test cases to, if possible, completely cover all structural items. In order to do this, useful (and enough) test cases must be developed.

Structural techniques are most used in component and integration testing, but they can also be applied at higher levels of testing, typically as

16. This is regrettably also true for functional requirements. The "of course the system has to do X" implicit requirement is one of the main problems for testing.

extra tests (for example, to cover menu structures). Structural techniques are covered in detail in sections 4.2 and 5.2.

3.7.4 Testing Related to Changes and Regression Testing

When changes are implemented, parts of the existing software are changed or new modules are added. This happens when correcting faults and performing other maintenance activities. Tests must show that earlier faults are really repaired (→retesting). Additionally, there is the risk of unwanted side effects. Repeating other tests in order to find them is called regression testing.

Regression test

A regression test is a new test of a previously tested program following modification to ensure that faults have not been introduced or uncovered as a result of the changes made (uncovering masked defects).

Thus, regression testing may be performed at all test levels and applies to functional, nonfunctional, and →structural test. Test cases to be used in regression testing must be well documented and reusable. Therefore, they are strong candidates for →test automation.

The question is how extensive a regression test has to be. There are the following possibilities:

How much retest and regression test

1. Rerunning of all the tests that have detected failures whose reasons (the defects) have been fixed in the new software release (defect retest, confirmation testing)
2. Testing of all program parts that were changed or corrected (testing of altered functionality)
3. Testing of all program parts or elements that were newly integrated (testing of new functionality)[17]
4. Testing of the whole system (complete regression test)

A bare retest (1) as well as tests that execute only the area of modifications (2 and 3) are not enough because in software systems, simple local code changes can create side effects in any other, arbitrarily distant, system parts.

Changes can have unexpected side effects

If the test covers only altered or new code parts, it neglects the consequences these alterations can have on unaltered parts. The trouble with software is its complexity. With reasonable cost, it can only be roughly estimated where such unwanted consequences can occur. This is particu-

17. This is a regression test in a broader sense, where *changes* also means new functionality (see the glossary].

larly difficult for changes in systems with insufficient documentation or missing requirements, which, unfortunately, is often the case in old systems.

Full regression test

In addition to retesting the corrected faults and testing changed functions, all existing test cases should be repeated. Only in this case would the test be as safe as the testing done with the original program version. Such a complete regression test would also be necessary if the system environment has been changed because this could have an effect on every part of the system.

In practice, a complete regression test is usually too time consuming and expensive. Therefore, we are looking for criteria that can help to choose which old test cases can be omitted without losing too much information. As always, in testing this means balancing risk and cost. The following test selection strategies are often used:

Selection of regression test cases

- Repeating only the high-priority tests according to the test plan
- In the functional test, omitting certain variations (special cases)
- Restricting the tests to certain configurations only (e.g., testing of the English product version only, testing of only one operating system version)
- Restricting the test to certain subsystems or test levels

Excursion

Generally, the rules listed here refer to the system test. On the lower test levels, regression test criteria can also be based on design or architecture documents (e.g., class hierarchy) or white box information. Further information can be found in [Kung 95], [Rothermel 94], [Winter 98], and [Binder 99]. There, the authors not only describe special problems in regression testing object-oriented programs, they also describe the general principles of regression testing in detail.

3.8 Summary

- The general V-model defines basic test levels: component test, integration test, system test, and acceptance test. It distinguishes between verification and validation. These general characteristics of good testing are applicable to any life cycle model:
  - For every development step there is a corresponding test level
  - The objectives of testing are specific for each test level
  - The design of tests for a given test level should begin as early as possible, i.e., during the corresponding development activity

  - Testers should be involved in reviewing development documents as early as possible
  - The number and intensity of the test levels must be tailored to the specific needs of the project
- The V-model uses the fact that it is cheaper to repair defects a short time after they have been introduced. Thus, the V-model requires verification measures (for example, reviews) after ending every development phase. This way, the "ripple effect" of defects (i.e., more defects) is minimized.
- Component testing examines single software components. Integration testing examines the collaboration of these components. Functional and nonfunctional system testing examine the entire system from the perspective of the future users. In acceptance testing, the customer checks the product for acceptance respective to the contract and acceptance by users and operations personnel. If the system will be installed in many operational environments, then field tests provide an additional opportunity to get experience with the system by running preliminary versions.
- Defect correction (maintenance) and further development (enhancement) or incremental development continuously alter and extend the software product throughout its life cycle. All these altered versions must be tested again. A specific risk analysis should determine the amount of the regression tests.
- There are several types of test with different objectives: functional testing, nonfunctional testing, structure-based testing, and change-related testing.