A TESTING ENVIRONMENT FOR DISTRIBUTED SYSTEMS

Marcin Sztandarski, Grzegorz Sowa,
Piotr Goetzen, Alina Marchlewska

IT Institute, Academy of Management, Lodz, Poland
marcin.sztandarski@gmail.com, (gsowa, goetzen, amarchlewska)@swspiz.pl

Abstract

The article presents the basics of modern software testing theory. Testing automation and the integration of testing into code writing will be examined in detail, and concept of a testing environment for distributed systems will be introduced.

Key words: distributed systems, software testing, testing automation, test-driven development

1 Introduction

The architecture of modern software systems is complex as most systems are distributed systems. Testing this type of system is a fairly complicated process, due to the various system platforms on which the software is based, the lack of the specification of the interfaces between system modules, and the difficulty in preparing the whole environment of the distributed system.

Over time, a variety of tests and testing methods have appeared. Along with the development of agile methodologies, testing has gone hand in hand with software creation from its earliest stages.

2 Software testing theory

2.1 Quality management and software system testing

Institutions that choose distributed systems tend to be, for example, financial organisations or large logistics companies. These systems are expected to function reliably and above all in line with their specifications. Quality management is essential, and one part of this is testing. In quality engineering

interdisciplinary methods are used. Practical skills in the business processes which the system is designed to serve are needed [2]. The testing is aimed at component systems of varying granularity– individual classes, components and the whole system are tested. [3]

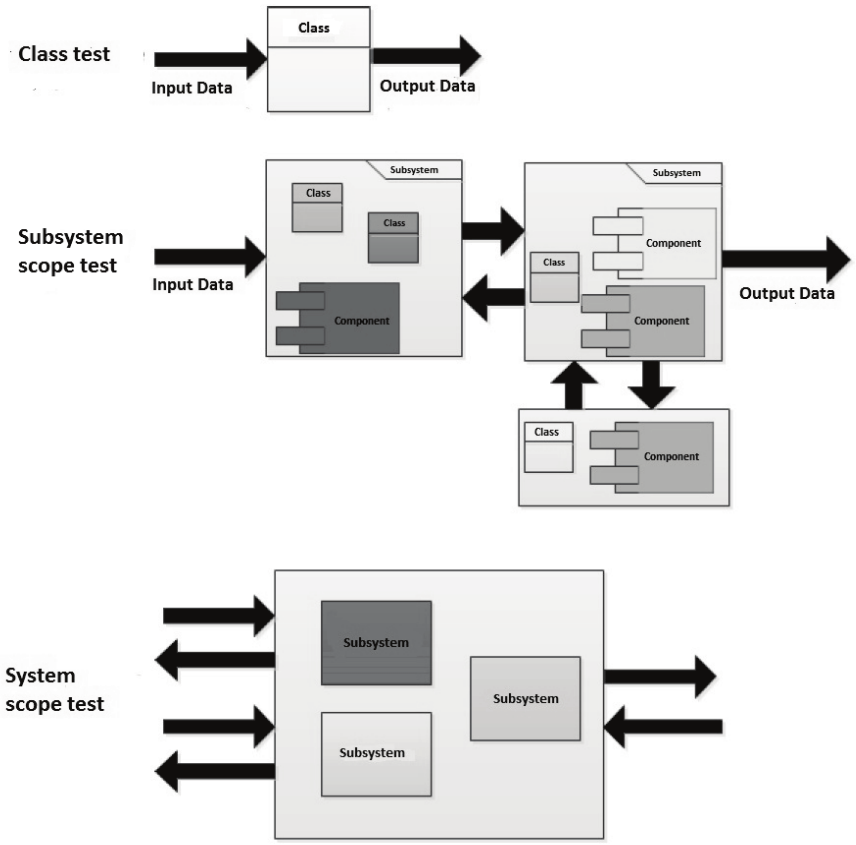

Figure 1. Typical test range (on a basis of [3])

Test design generally includes the following steps:
- analysing and modelling the expected system behaviour
- designing test variants (entry and exit)
- designing test variants arising from structural analysis and other error detection methods (e.g. heuristics)
- stating the expected results for each test variant.

Several models are used to cope with the complexity of the system. Each model describes a particular type of test and has a particular aim.

2.2 Test automation

Testing a large amount of software creates the need for test automation. An automatic test systems allows entry data to be applied and for test results to be verified. Such systems must be compatible with the interfaces and infrastructure of the system being tested. Test automation systems ensure that tests can be carried out repeatedly. This facilitates regressive tests, for example, which are generally carried out after the introduction of a repair or new function. Although it is estimated that testers find only 15% of errors through automated tests [6], the most interesting attribute of this kind of test is its repeatability, which allows the test procedure to be generated in a different hardware platform, for example, or in another configuration.

2.3 Black and white box testing

There are two ways to design tests. If the designer only takes the entry and exit specifications of the system into account, then the test is a black box test. For example, testing a log-in box with two fields: "user" and "password" and the "log in" button. The tester enters data according to the specifications of the test into the appropriate fields, and then checks the effect of pressing the button. In the case of this test, the internal data processing is not relevant – only the exit data obtained for specific entry data are checked [10].

The opposite approach in known as white-box testing. Here, the internal structure of the system is taken into account. The steering path routes in the system being tested are analysed, and also solution implementation methods. In the case of the log-in window, the internal components and the interaction between them will be checked.

It is increasingly common for a distributed system to be rolled out by many teams simultaneously, with solutions being delivered at different points in time. Part of the system maybe created outside the main organisation. White-box tests are designed only for the parts of the system, which are implemented by the designer, whereas complex black-box test are the best means of quality control in cases when we ourselves have not created the code.

2.4 Typical software development processes and testing

Generally speaking, the process of creating software is based on the translation of information, such as information about a business process, into source code. This information is transferred during the following stages of software creation [8]:

- Needs establishment: the user (the sponsor of the project), working with an analyst, defines what is expected of the system that is to be created. These expectations are written down as the formal aims of the project.
- Analysis and description of goals: the details are agreed upon and the relationship between each goal is specified, taking into account priorities, affordability, and any compromises.
- Creation of external specifications: system elements are described as "black boxes", in other words only interfaces and the interaction with the user (and in the case of batch systems, entry and exit specifications) are specified.
- Creation of the project structure system structure: the system is divided into a series of elements of decreasing granularity. In other words, it is divided into programs, components, etc, and interfaces are also defined.
- Functional specification of modules and their interfaces: the function of each module, the relationships between modules and working guidelines are established.
- Exact specification of each module, defining the interface and functional elements.
- Creation of source code.

Incorrect transfer of information can occur at any of the above stages. In order to eliminate mistakes, testing processes are used simultaneously with each stage. Each testing stage is responsible for eliminating a particular kind of mistake. The relationship between software creation processes and testing processes is often represented as a "V" shape, where the series of stages associated with creating the system are on the left side, and the corresponding tests on the other. The "V" model, which is an extension of the typical waterfall model, can be adapted to methods, where an iterant approach is used, as well as to agile methodologies. In this case, the route through each stage is completed for each iteration. This approach improves the reliability of each stage of system creation, as each particular type of mistake is eliminated as soon as it might appear. Thus, when the external specification of the system is being tested, functional testing is carried out rather than broad system testing. The tester focuses on mistakes in functionality implementation, and not on, for example, data processing efficiency.

The test structure along with the corresponding stages is as follows:

- Acceptance testing at the needs establishment stage. This type of test assesses to what extent the system being tested corresponds to the expectations set out in the specification. Often this stage of testing is carried out by the client's own team of testers. Functional as well as non-functional expectations are tested (e.g. efficiency).

– Systems testing at the goals description stage. This kind of test is essential due to the face that certain characteristics and functions of the system are only visible when the software is treated as a whole. The difficulty of designing this type of test is due to the fact that the document which describes the project aims is a general one, and so cannot provide specific systems tests. Therefore, user documentation is also used when designing systems tests.

There are several categories of system tests which focus only on specific aspects of the system. They are not used in every system, however.

– Facility testing – which tests their compatibility with the established aims.
– Volume testing – which checks how the system copes with a large amount of entry data.
– Stress testing - which assesses the system's ability to process a large amount of data in a short time.
– Usability testing – which checks the how user-friendly the interface is.
– Safety testing (data protection) checks, among other things, whether the system is vulnerable to data leakage.
– Effectiveness testing assesses how the system copes with varying demands, i.e.: whether the time it takes the system to produce an answer in a given configuration matches the estimated times.
– Configuration testing checks the system in terms of its ability to cope with different equipment configurations and environments (for example different browsers in the case of internet applications.)
– Compatibility and conversion testing checks whether data can be converted from one version of the system to another (for example, whether data can migrate from previous versions) or whether the system can work in so-called compatibility mode.
– Installation procedure testing aims to identify mistakes in the software installation process.
– Reliability testing establishes whether the system can carry out given functions in particular conditions.
– Emergency function testing checks how resilient the system is in case of breakdown. Most frequently, the average time it will take for the system to recover after a breakdown is estimated (Mean Time To Recovery).
– Service testing involves checking to what extent service and conservation of the system are possible. For example, the test establishes whether a status report can be produced.
– Documentation testing aims to eliminate ambiguity, and the documentation is assessed for completeness and detail, among other aspects.

– External specification functional testing identifies any discrepancies between the expected behaviour of the system from the perspective of the user and its actual behaviour.

The testing processes listed above are very much horizontal tests. In subsequent development stages, tests at the level of individual units are used, as follows:
– Integration testing aims to identify defects in interfaces and in the interaction between units.
– Unit (module) testing.

2.5 Units testing

Program units are part of the program code, in the form of sub-programs, classes or methods. Units consist of the smallest element of the system which it is worthwhile testing. Initiating testing while writing a unit, for example a class, has a host of benefits for the programmer. The specification of how the code behaves in tests will significantly facilitate the analysis of the code by other programmers. Refactorisation is safe – the programmer does not worry that he or she will change the way the code works when changing the code, creating a mistake [9]. It is important that the programmer is expected to take care over the project: dividing the code into parts, according to which particular unit they are intended for. Unit tests which have had their connection with other parts of the system removed can be carried out simultaneously.

Michael Feathers [4] outlines certain desirable characteristics of unit tests:
– Unit tests should work fast,
– Unit tests should not communicate with the database,
– Unit tests should not used network communication,
– Unit tests should not use the file system,
– The programmer cannot carry out additional preparation procedures in order to carry out a unit test. Programmers sometimes introduce additional connections into unit tests, for example a connection with the data base, which turns the unit test into an integrative test.

Creating unit tests requires a well designed system, in which there are not many connections between modules. This enables classes and methods to be tested independently of each other. One method of isolating classes is to use mock objects, which does, however, increase the complexity of the system. Another method is appropriate system design and application, for example, dependency injection [11].

2.6 An outline of test-driven development

Test-driven development (TDD) is a software development system which consists of three steps: creating a test, writing an appropriate code, refactorisation. This cycle is often known as the "red-green-refactor mantra" among programmers who use TDD, which is due to the behaviour of TDD support tools, in which red means that the test produced a negative result, and green that it produced a positive result. The most important, seemingly simple rule of TDD is the golden rule: never write a new functionality before you have written a test, which produces a different result than expected (red).

Code writing using TDD consists of the following steps [4]:

1. Choosing the task and creating the test – the programmer starts writing a testing code which specifies the desired behaviour of, for example, a certain method. Of course, the code is not compiled, as there is no implementation code.
2. In the next step (represented as red), the minimum implementation code necessary for the particular class/method being tested is created, with the aim of enabling compilation. The testing tools are marked as red.
3. Writing the correct code (represented as green) means implementing the method in such a way, as to fulfil the requirements of the test. This step lasts up until the point when the colour green appears.
4. Refactorisation, which means modifying the structure of the code that has been tested without changing its functionality. These changes generally aim to improve the code's readability.

It is very important to note that steps 1-4 are carried out cyclically (even multiple times an hour), whereas completing one cycle gives the programmer immediate feedback. Another advantage of the TDD technique is that the programmer focuses on one task – either code writing or refactorisation. Moreover, applying this procedure fully in a project gives the programmer a sense of security when introducing changes in the code later on, since the base code is covered by tests. This technique is also suitable for legacy application. Both bug fixes in existing code and changes introduced to existing functionalities should begin with test writing [1].

2.7 The effectiveness of test-driven development

The principal benefit of TDD is the assurance that any mistakes made during the implementation of corrections or new functionalities will not introduce hidden errors. The "golden rule of TDD" ensures that every functionality has its own test. There is also a collection of regressive tests which allow the program to be retested in order to identify newly introduced mistakes. Following

the TDD rules generally leads to hundreds of tests a month and thousands of tests a year being carried out, which in practice cover more than 90% of production code [7]. Tests from which dependencies have been removed should only take a few minutes, even if there are a few thousand of them. This means that the programmer can check the effects of introducing a correction on the rest of the code within this short time. Similarly with refactorisation, the programmer is not afraid of making changes even in "messy" code (for example code in which abstractions are mixed, such as business rules and limited access to data) since it is practically impossible to "break" the code. Unit tests are also the most readily comprehensible documentation for programmers, since they are written in the same language as the system is created in.

In 2007, Ron Jeffries and Grigori Melnik [5] presented the results of research into Test-Driven Development techniques in the IT industry. Regardless of the benchmarks used, all the research results indicated that product quality increased significantly.

2.8 The problem of units integration

During unit testing, units are tested in isolation from other elements, which means that their code does not establish a connection with „the outside world". The units being tested do not communicate through the network, do not save files, do not go beyond the boundaries of the process. Unit testing should then be expanded into integrative testing. Various strategies are used to integrate modules: growth integration, increasing and decreasing integration [11]. The purpose of integrative testing is to identify defects in the interface and in interactions between units. Units in complex systems (and also distributed systems) give access to the interface or carry out calls for methods made accessible by other units and cannot be tested individually. It can be difficult to test interaction, because certain parts of the system may not yet be accessible.

In order to resolve this, environmental elements which replace the surrounding modules are used:

- The driver unit – calls to the tested unit are carried out form the level of the driver.
- The stub unit – creates access to the interface of the unit, whose methods are called up.

Figure 2 shows models of the configuration of test units for two examples of distributed systems:

System I illustrates an imaginary configuration for complex tests where interaction is taking place between three units (unit 3 has been replaced by an element which gives access to the interface of unit 2.)

System II is an integrative test which checks the interaction of module A with the outside world. The driver of module A enables calling up the methods of interface A, whilst the element of module B enables carrying out calls in the range of module B's interface.

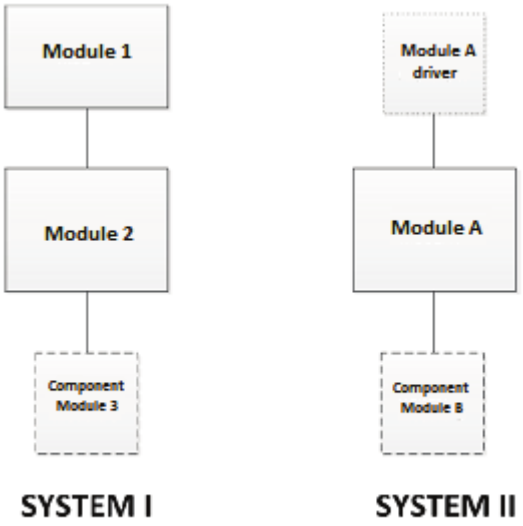

Figure 2. Model of configuration of units in integrative tests (complex texts) for examples of distributed systems. Source: own design.

3 System concept for testing distributed systems

3.1 High-level design architecture

High-level design describes components and their functions in the process of testing distributed systems. Only significant aspects will be discussed here. Figure 3 illustrates the main subsystems and their communications.

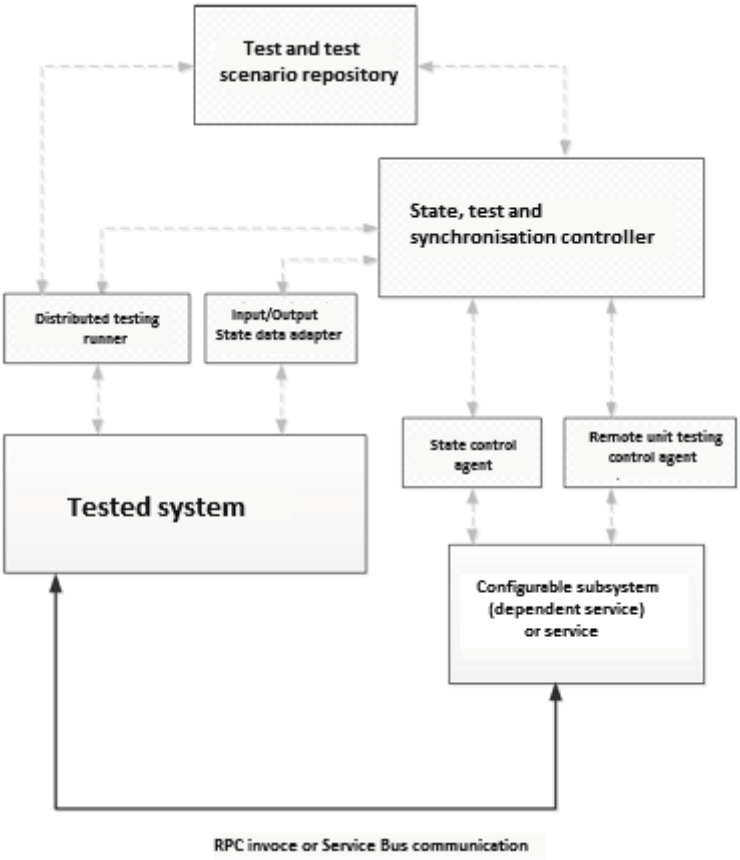

Figure 3. A diagram of the testing environment architecture of distributed systems. Source: own design.

3.2 The concept of test fixture

A test fixture is an agreed system status which ensures that test conditions are repeatable. The test fixture is created by the test environment. In practice, this is a service steered from the level of the status control agent. The test fixture is created by the system being tested along with the adapter.

3.3 What is required of the system being tested

In order for the system to be testable, two conditions must be met:
- The communication interface must be separated from the outer components which the tested system communicates with
- The data adapter must be completed.

3.4 The role of components (of subsystems)

The runner subsystem of distributed testing is a component, which has the function of communicating with the system being tested in order to initiate the given command, forcing shut down of the testing procedure, or dealing with a system failure.

The data adapter is a component which enables the entry data transferred to the system to be read, the exit data to be stored and the system status to be dumped.

The status, test and synchronisation controller is a subsystem responsible for the co-ordination of the whole environment and synchronizing tests. Status co-ordination involves sending commands to the status control agent. The commands are sent according to the content of the test fixtures. The controller also sends test signals to the runner controlling the system being tested.

The status control agent carries out commands to set up a service or an imitation service in a particular way.

The remote unit test control agent.

System imitation is an external service, which the system being tested depends on.

The repository of tests and test scenarios is an application which allows test scenarios and reports of tests that have been carried out to be collected.

4 Case study

4.1 The concept of the Long Running Process Server framework

A high-level design test for a system which supports key business processes is described below. Testing this system depends on the availability of many components and external systems.

The Long Running Process Server framework is a collection of components and interfaces installed in business applications which demand service for processes that are not synchronized. Processes that are not synchronized allow the application to send a command for a long-term task to another machine, and then to continue functioning without waiting for the result. It is also possible to finish the command process and memorize the task handler,

which would make it possible to refer to the result after the next command process has been initiated (or alternatively to delegate the reading of results to another process).

4.2 Architecture of the Long Running Process Server framework

A description of the high-level components needs to be added to the framework concept described below in order for the testing scenario to be clear.

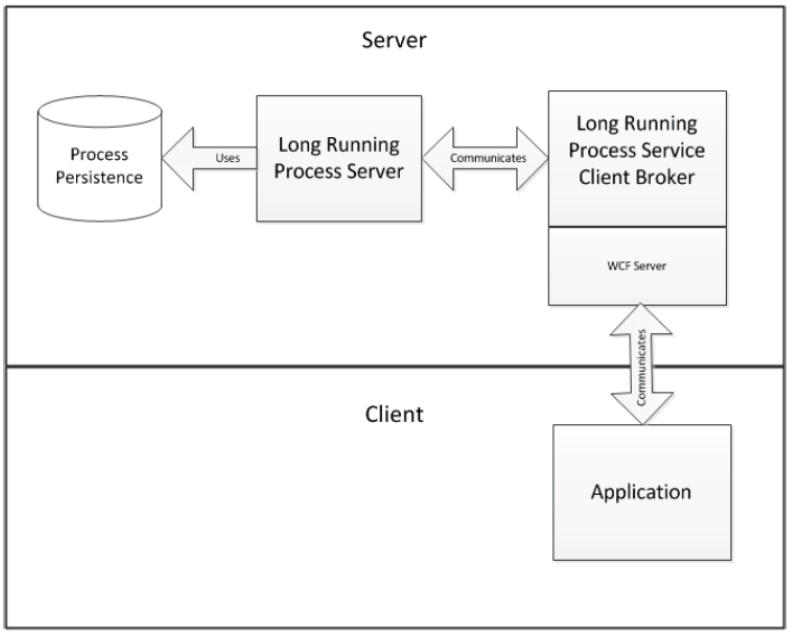

Figure 4. Architecture of the Long Running Process Server framework. Source: own design.

The structure and organization of the server are as follows. The Long Running Process Server (executive server) is a service for the Windows operating system. When the service is switched on/started, a configuration which identifies plugins is loaded, with a task executive element. After the plugins are loaded, the service process checks whether there are any tasks to be performed in the queue, and if so begins to process them according to the allotted task ordering algorithm. Both the executive parameters and the task process exits are consolidated in line with process persistence; in this implementation these are XML communications saved in the MS SQL Server base. The ser-

vice also opens up the interface for communication with the broker process. The executive server deals with all connections with the outside world for the tasks requested.

The Long Running Process Service Client Broker is a service for the Windows operating system which is a bridge between the Long Running Process Server and client applications. In practice, the broker is dedicated to one particular application and only serves specific tasks. The component is made up of the self-hosted Windows Communication Foundation Server, which serves different kinds of communications with clients, and a functional part which supports the loading of plugins for particular tasks. The broker is responsible for passing on instructions, carrying out tasks with entrance instructions and for answering questions about their status (and enabling results to be recorded.) The service makes the service accessible for client applications in the form of Remote Procedure Call requests. The Long Running Process Server can serve requests from many brokers and many applications, which means that the brokers can be specialized in terms of function.

Client applications are processes which have a shorter life cycle than the tasks requested by them. They are generally made up of a presentation layer and a layer which is responsible for communication with the broker. The presentation layer is built from component provided by the framework library.

4.3 Examples of business system testing scenarios, based on the Long Running Process Server.

The scenarios described below are examples of integrated systems testing which supports business processes in financial institutions offering clients credit, credit cards and medical care/insurance. There is often a considerable delay in processing tasks which require communication with outside systems, due to business working conditions such as the need for a form to be approved, or the availability of outside services, and so these tasks are transferred to the Long Running Process Server by the Window application operator. The correct implementation of the business process completion can be found in the BusinessTaskDisposal component.

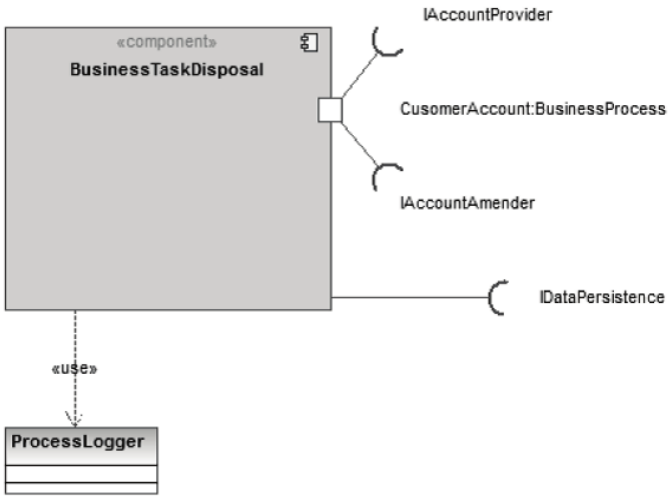

Figure 5. View of the BusinessTaskDisposal component. Source: own design.

The component that carries out tasks through the use of the Separated Interface model uses the potential of the external implementations of servicing client account services (IAccountProvider, IAccountAmender) and data consolidation (IDataPersistence). This approach also allows the stubs of any service to be used and focus tests to be carried out on only one service. The dependence on IDataPersistence is due to the fact that the Long Running Process Server consolidates entry data, exit data and the task status. Another important dependency is the ProcessLogger class, which consolidates data related to communication about the process status (this data is used to diagnose and display the status on the client side). The testing environment configuration in the case of each hub is made up of: the status control agent, the remote test control agent and the imitation of certain dependent services, which the task processes communicate with.

Smoke test scenario (preliminary test)

The preliminary test checks whether the system is ready for further, detailed tests to be carried out.

Test scenario 1 – read status of all system hubs for the configuration that has been loaded, expected result – status {OK} for each hub.

Test scenario 2 – load imitation of every dependency for BusinessTaskDisposal, and then carry out the given remote functional test.

Regressive test scenarios for given tasks

The aim of regressive tests is to ensure that no new mistakes have been introduced when making changes in the software. This test consists of repeating tests carried out before the changes were introduced.

Test scenario: load the regression test baseline set which contains the test examples that were saved along with the test fixtures, run the automated tests, check the results.

End-to-end test scenario

The aim of end-to-end tests is to test business transactions at the level of their components, i.e. to ensure that all components are working together correctly and processing data correcting (at the level of the business process).

Test scenario for a single end-to-end test:

1. Set the hubs to a status with no imitations.
2. Carry out a preliminary test for each hub which is part of the process
3. Transfer data to be processed on the client side
4. Run the test
5. Check the processing results for each hub.

Test scenario for functional tests for business processes which are carried out on the server side

During functional tests the implementation of the business function is tested (a black-box test).

Test scenario:

1. Set the dependent service hubs
2. Set the service statuses
3. Load the entry data
4. Run the test
5. Compare the actual and expected exit data

5 Conclusion

5.1 Complexity

Testing distributed systems is complicated. The behaviour of the network is to some extent unpredictable and preparing the testing environment is difficult. The test designer can only create an approximation of the conditions in which the system is going to function. Carrying out tests related to subsystem communication in distributed systems is often very expensive, because an organization might only have one production environment serving mass

communication with many clients at their disposal. Another problem is the question of how to simulate the behaviour of hundreds or thousands of clients. Moreover, network technologies are constantly being improved, while the life cycle of distributed systems is relatively long, which means that these systems have to function in a network environment which will be ten or more times faster in a few years.

5.2 Safety

The role of safety testing in distributed systems is increasingly important. The characteristics of distributed systems, their division into many subsystems, leads to an increased risk of data leakage. Testing the safety of the system as a whole at the end of the project, i.e. during the final integration, may not be sufficient, since repairing problems at this stage could be very expensive. Safety is a factor which should be clearly defined at the beginning of the project when the client's expectations of the system are set out. Safety testing begins with checking it at the level of components, and ends with testing at the level of the final integration and acceptance tests.

5.3 Distribution

The distribution of the project may extend beyond the boundaries of the organization. Many companies benefit from outsourcing and off-shoring. The quality of a system which has been prepared outside the organization must be measured. The test designer has to deal with the difficulties of constantly expanding acceptance tests which check both functional and non-functional characteristics. One method is the introduction of iterative methodologies which support such an approach. Creating distributed systems in a non-iterative way (for example the waterfall model) is too risky, as problems become apparent at the end – during the integration of the whole system.

5.4 Heterogeneity of environments

Designing tests for distributed systems requires a knowledge of the characteristics of many environments: equipment platforms, operating systems, debuggers. Often distributed systems are built with the aim of adapting old systems to new business conditions. One example would be any kind of banking system, which display part of their functionality in client systems like home banking. Therefore, an architect designing tests for this kind of system needs to know the characteristics of both the new and the old parts of the systems (e.g. a banking system such as core).

5.5 Automation

In order to automate tests for distributed systems, they need to be built in a particular way. In practice, this means, for example, adding a thin interface layer between the software containing the use interface and the layer below (for example, the business layer). An approach called hexagonal architecture is used, which means adding a range of adapters to the system which allow, for example, interaction with the user to be replaced with a range of automated API calls.

5.6 Synchronisation

In distributed systems, tasks are carried out in parallel and sometimes the processing cannot be done by a single machine. Tasks carried out on many different machines influence each other constantly, which means that they must be synchronized. This may cause errors that are difficult to diagnose, for example the appearance of blockages (in the file system as well as in the database).

Summary

The above attempt to design a testing environment for distributed systems aimed to solve a particular problem – testing an asynchronistic task processing server. The high-level design presented here was a very simplified one. A detailed design for the status controller component, tests and synchronization will be a considerable challenge. The flexibility of the system and ease of testing will depend on this implementation. The way that the system deals with unforeseen circumstances such as breakdowns, transaction cancellations or the peculiarities/characteristics of network communication will be significant.

References

1. Baley K., Belcham D., Manning 2010, *Brownfield Application Development in .NET*, , p.105
2. Bereza-Jarociński B., Szomański B., Helion 2009, Inżynieria Oprogramowania. Jak zapewnić jakość tworzonym aplikacjom, p. 69.
3. Binder V. R., WNT 2003, *Testowanie systemów obiektowych*, p. 48.
4. Feathers M., 2012, *Working Effectively with Legacy Code*, http://www.objectmentor.com/resources/articles/WorkingEffectivelyWithLegacyCode.pdf, downloaded Oct the 22

5. Jeffries R., Melnik G., 2007, IEEE SOFTWARE, *Professionalism and Test-Driven Development*, May/June 2007, IEEE Computer Society, p. 28
6. Kaner C., Bach J., Pettichord B., Wiley 2002, Lessons Learned in Software Testing: A Context-Driven Approach, p. 101.
7. Martin R., 2007, IEEE SOFTWARE, *Professionalism and Test-Driven Development*, May/June 2007, IEEE Computer Society, p. 33.
8. Myers G., Sandler C., Badgett T., Thomas T., Helion 2005, *Sztuka testowania oprogramowania*, p. 151.
9. Osherove R., 2009, *The Art of Unit Testing*, Manning, p. 55
10. Shaefer H., 2012, *What a Tester Should Know, even After Midnight*, http://www.sjsi.org/webgears/files/sjsi/File/tester/tester_5.pdf, downloaded Oct the 22, p. 40
11. Shore J., Warden S., Helion 2008, Agile Development. Filozofia programowania zwinnego, p. 354