FAKULTÄT FÜR INFORMATIK

Faculty of Informatics

Isolated Testing of Software Components in Distributed Software Systems

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur/in

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

François Thillen
Matrikelnummer 1029188

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung
Betreuer: Ao. Univ.-Prof. Dr. Stefan Biffl
Mitwirkung: Univ.-Ass. Dr. Richard Mordinyi

Wien, 12.10.2012

(Unterschrift Verfasser/in) (Unterschrift Betreuer/in)

Erklärung zur Verfassung der Arbeit

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

________________________	________________________
(Ort, Datum)	(Unterschrift Verfasser/in)

Abstract

In modern software developments, different software architectures and concepts of software reusability have been introduced. An example of these paradigms is component based systems. Component based software engineering emphasizes the composition of software systems through loosely coupled independent components.
Testing is an important part in the software's lifecycle because it allows conclusions on the quality of software made. Furthermore tests help to verify code changes and increase the stability of the software.
Software testers use several testing approaches like unit tests or integration tests to find defects in the code and to inspect the correct behaviour of software components.
In the field of component testing, approaches like Jata or JRT exist, which allows the testing of remote components. Remote components are units of code that are communicating over the network connection. However, these testing approaches rely on the definition that a component is independent to other components. Although software components are binary units of independent software artefacts, they typically interact with other components as they form a functioning system and thus implicitly define dependency relations to other components. Since current test strategies reflect the total independence of components by their technical aspects (like its interface), they do not completely take into account the dependency structure to other "components defined though system functionality requirements". Therefore those test strategies are limited in their effectiveness of detecting defects in the overall system. In other words, software testers can create test cases which allow the testing of dependent components but the entire system has to be available and running.
In this work the "effective tester in the middle" (ETM) approach is presented which takes into account the implicitly defined dependencies between distributed software components. The ETM is a dependent component testing approach that enables isolated testing of individual software components of a distributed software system by simulating relations to dependants through network communication models representing established network communication protocols. While the ETM does not alter software components, it pretends existing network communication, and filters and analyses exchanged network data segments for test scenario-specific communication behaviour. Based on the test scenarios' configuration, the ETM responds with an appropriate response rather than the originally targeted software component. This enables component testing independent of the system which results in earlier defect detection.
The ETM approach improves the efficiency of software testers in applying test scenarios since they only need to focus on a single component and on configuring the relations to components it directly depends on. This allows software testers to create test scenarios without taking into consideration the entire software system. Additionally, the ETM approach improves effectiveness of testing since it enables the detection of dependency defects between components.
Based on a real world use case scenario, the ETM approach has been evaluated by measuring its configuration complexity, the time of configuration, and the effectiveness of the ETM framework in comparison to traditional approaches. In addition, the performance of the interaction between the ETM framework and the component under tests is measured. The evaluation results show that test cases can be written in an efficient, simple way, and facilitate systems with high test coverage.

Kurzfassung

In der modernen Software-Entwicklung wurden verschiedene Software-Architekturen und Konzepte der Wiederverwendbarkeit von Software eingeführt. Ein Beispiel für ein solches Paradigma sind Komponentenbasierte Systemen. Komponentenbasierte Software Entwicklung betont die Zusammensetzung von Softwaresystemen durch lose gekoppelte unabhängige Komponenten.

Testen ist ein wichtiger Bestandteil im Software Lebenszyklus, da Rückschlüsse auf die Qualität der Software vorgenommen werden können. Darüber hinaus helfen Tests Änderungen im Code zu überprüfen und erhöhen die Stabilität der Software.

Software-Tester verwenden eine Vielzahl von Tests Ansätze wie Unit Tests oder Integrationstests um Mängel im Code zu finden und zusätzlich das richtige Verhalten der Software-Komponenten zu untersuchen.

Im Bereich von Komponenten testen existieren Ansätze wie Jata oder JRT, die das Testen von Remote-Komponenten ermöglicht. Verteilte Komponenten sind Codeeinheiten, die über die Netzwerkverbindung kommunizieren. Die vorgestellten Testansätze basieren auf der originalen Definition, bei der eine Komponente unabhängig von anderen Komponenten ist. Obwohl Software-Komponenten binäre Einheiten von unabhängigen Software-Artefakten sind, interagieren sie in der Regel mit anderen Komponenten, um ein funktionierendes System zu bilden. Dadurch wird eine Abhängigkeit zu anderen Komponenten impliziert. Da aktuelle Teststrategien die völlige Unabhängigkeit von Komponenten durch ihre technischen Aspekte (wie Schnittstellen) voraussetzen, berücksichtigt sie nicht vollständig die Abhängigkeitsstruktur zu anderen Komponenten über die System Anforderungen. Daher sind jene Teststrategien in ihrer Wirksamkeit, Erkennung von Fehlern im gesamten System begrenzt. In anderen Worten, Software-Tester erstellen Testfälle, die das Testen von abhängigen Komponenten erlauben, jedoch muss das gesamte System vorhanden sein und ausgeführt werden.

In dieser Arbeit wird der „Effektive Tester in der Mitte“ (ETM) Ansatz vorgestellt, welcher die implizit definierten Abhängigkeiten zwischen verteilten Software-Komponenten berücksichtigt. Der ETM ist ein Ansatz zum Testen von abhängigen Komponenten, welcher das isolierte Testen von einzelnen Komponenten in einem verteilten Softwaresystem durch Simulation von Beziehungen zu Abhängigkeiten erlaubt. Konkret wird dies über Netzwerkkommunikationsmodelle realisiert, welche Kommunikationsprotokolle darstellen. Der ETM simuliert die vorhandene Netzwerk-Kommunikation, filtert und analysiert ausgetauscht Netzwerk Datensegmente und verändert dadurch keine Software-Komponente. Basierend auf einer testspezifischen Konfiguration antwortet der ETM mit einer passenden Nachricht, anstelle der ursprünglich angestrebten Software-Komponente. Dies ermöglicht Komponenten unabhängig von dem System zu testen, das zu einer frühen Fehlererkennung führt.

Der ETM Ansatz verbessert die Effizienz der Software-Tester bei der Implementierung von Testszenarien, da sie sich nur noch auf eine einzige Komponente und die Konfiguration der Beziehungen zu den abhängigen Komponenten konzentrieren müssen. Dies ermöglicht Software-Testern Testszenarien zu erstellen ohne das gesamten Software-System zu berücksichtigen. Darüber hinaus verbessert der ETM Ansatz die Wirksamkeit von Tests, da Fehler erkennt werden können, welche durch die Abhängigkeit zwischen Komponenten entstanden sind.

Der ETM Ansatz wird anhand eines realen Anwendungsfall ausgewertet. Dies geschieht durch Messen der Komplexität und Zeitaufwand der Konfiguration, sowie die Wirksamkeit des ETM im Vergleich zu herkömmlichen Ansätzen. Des Weiteren wird die Performanz zwischen dem ETM und der Komponente unter Tests gemessen. Die Auswertungsergebnisse zeigen, dass Testfälle in effizienter und einfacher Weise geschrieben werden können. Zusätzlich ermöglicht der ETM Systeme mit hoher Testabdeckung.

Contents

CHAPTER 1

Introduction

Nowadays software testing has become a very important part in the software engineering process. There are several paradigms in modern Software engineering like Web services and component based systems.

“Component-based Software Developments focused on assembling previously existing components (or other non-developmental items) into large software systems, and migrating existing applications towards component-based systems”[1]. The advantage of component based systems is that components can be reused, implemented by third parties, and modified without changing the complete system. Furthermore, systems can be composed to a software system in a loosely coupled manner.

Since Component based systems getting more attention, several definitions has been created, as: „software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system”[2]. This definition shows that a component has to be independent to other components. Furthermore, components are offering their services over interfaces, which can be used from different components to invoke the presented methods. The following picture should illustrate the structure of component based systems.

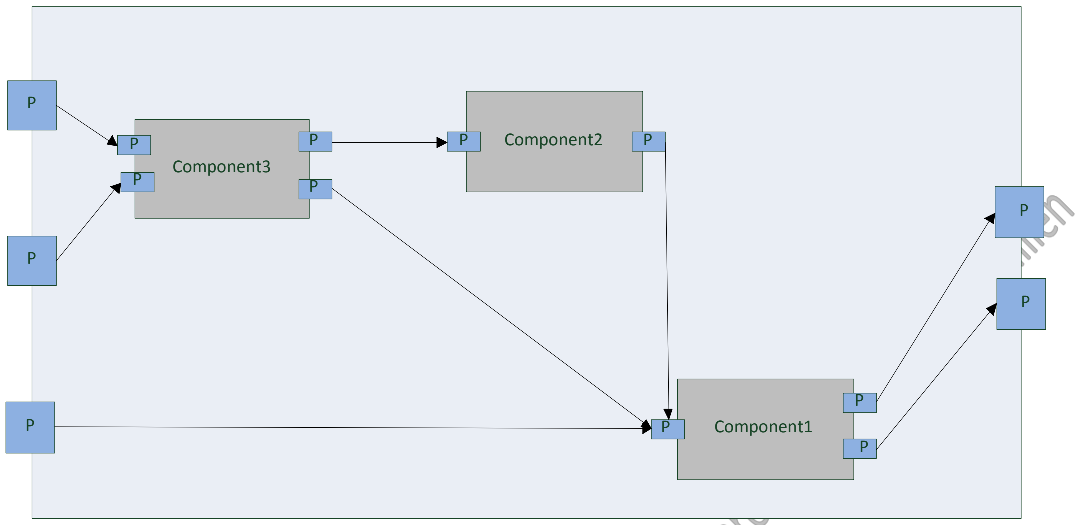


Figure 1.1: Example of Component based Systems

A component can have different ports *P*, on which other components can invoke methods. The components them self can invoke or forward a request to other components. Like presented in Figure 1.1 a component can consist of different components. These inside components can be self-running units but can just be indirectly invoked from outside.

Component based systems concentrate on assembling prefabricated parts which can be an organization's own implementation or some parts can be from professional component vendors [3]. Furthermore, the concept or reusing pre-existing components has the aims to reduce development time, costs and risks, while developing larger and more complex systems quickly and with high product quality[4]. With this nouveau paradigm, new challenging problems in component based systems, like avoiding of dependencies between components has to be faced [5].

From the definition, components should be independent, have a clearly defined interface and interact with a function system [2]. Components should not be standalone but work altogether in a system. To be able to communicate with each other, components provide interface, which allows a communication between components. Next, definitions of a components implies that a component should live independently but it does not imply that a component does not need further components to able to complete the work [6][5]. However, components typically interact with other components as they from a functioning system and thus implicitly define dependency relations [7], which are called dependent components.

There two different kinds of components. Once are for local use and others are for remote usage. A remote components can use, for example the remote method invocation (RMI), the remote procedure call (RPC), or the message orientated middleware (MOM), to communicate. An example for remote components in distributed systems is presented in Figure 1.2.

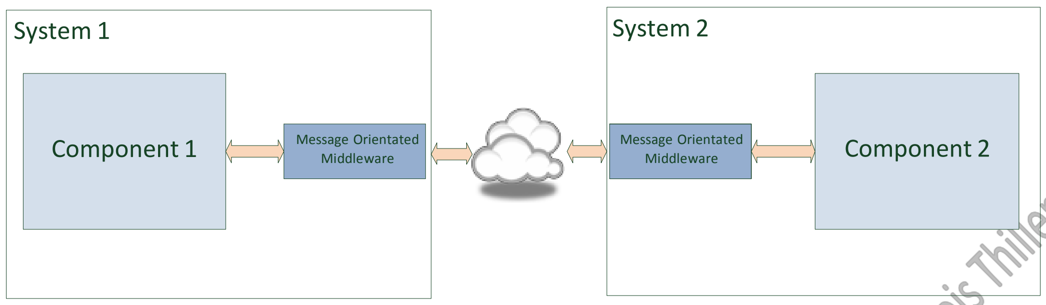


Figure 1.2: Remote component example

Components can have dependencies to other external components or internal component. From the testing point of view, higher dependencies leads to a more complex system [7], which makes it challenging to modify, verify and understand the system. It follows that testing get more complex with increasing dependencies between components. Altogether, dependencies between components are common and should be handled very important in the field of software testing.

There are several testing approaches and frameworks at software testers' disposal to find defect in the code and analyse the correct behaviour of a software component in distributed software systems [8]. Some established test strategies for independent components are integration tests [9] or web service tests [10]. However, those testing approaches rely on the definition of an independent component.

If software tester want to ensure the quality of dependent components by testing the components' interface, they have the option of make use of mock-up frameworks [11]. However, these do not support testing of the complete component, but only a subset of it because parts of the components are mocked and thus does not take into account any component dependencies. Other test strategies for dependent components are integration tests which can be performed in the complete running system. In this case the entire component is tested and its dependencies considered. However, this is a rather uncomplicated task for non-distributed environments, it might be challenging in distributed one. Therefore, current test strategies are limited in their effectiveness of detecting defects in the overall system with minimal effort. Nevertheless, as with any software engineering approaches the goal of quality and stability code rely on thoroughly tested components.

To test a component, depending to other components and/or the network communication in architectures, the dependent components have to be simulated. One way to achieve this aim, is mock-up the other components. This approached depends on the complexity of the component and on number of interacting complex components. Furthermore, the source code for the components has to be open. This thesis presents the so called "Effective Tester in the Middle" (ETM) approach which improves the testing of distributed components depending on other components in the system. With the ETM concept, interaction models, and network communication models are introduced, which facilitate isolated testing of the complete component without the need to start the entire system.

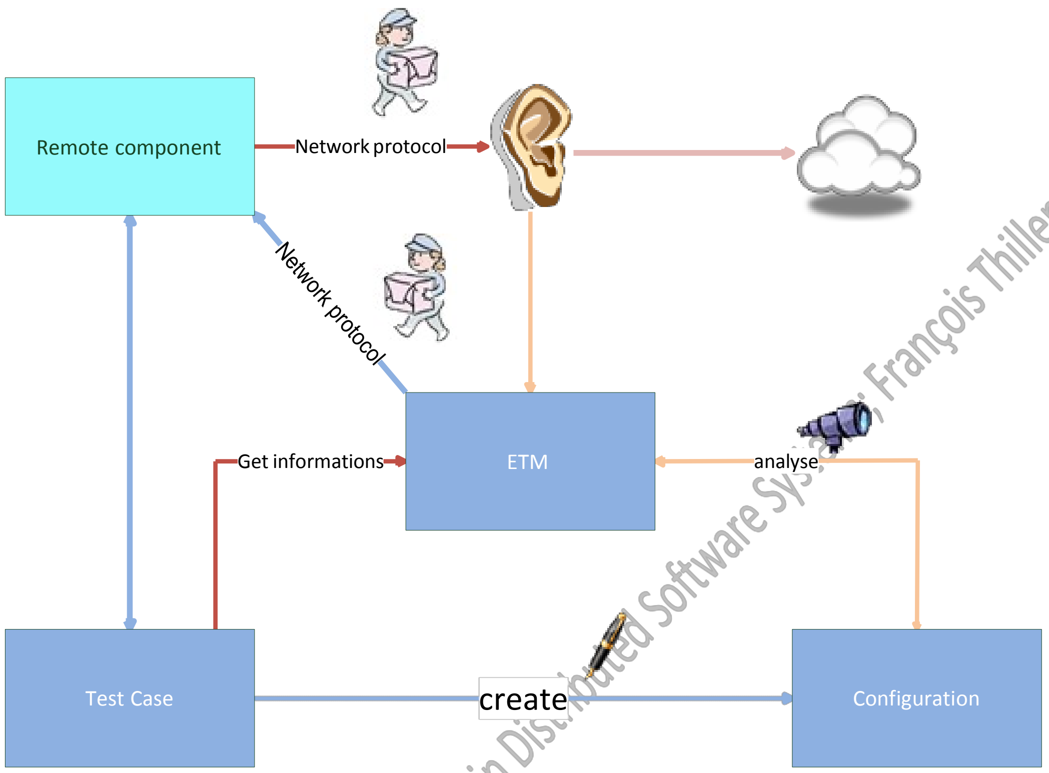


Figure 1.3: Testing dependent components

The proposed approach makes use of the established network protocols to filter for requests sent by tested components as in Figure 1.3. The request is analysed and based on a given test scenario and its specific interaction model an appropriate response is returned. The software testers implement test scenarios in unit test manner and configure request-response interaction models, which are parsed by the ETM while monitoring the network communication. A main advantage of the ETM is that the components can be tested isolated to the system, which increases the correctness of the entire system.

An example for this structure is the Simple Object Access Protocol (SOAP[1]). The client invokes a method which forwards a method call to Web service. The request has an html part and a SAOP envelope which contains all the method information and the parameters. This request is recognised by the framework, the (ETM) and then proceeds. In other words, the framework listens on a specific port and receives the packages over a socket. These packages are dissembled to a message and first the protocol information is separated from the data. In the next step, the received data are analysed and then a corresponding response is chosen. The response is defined in a

[1] http://www.w3.org/TR/soap/

specific protocol which gets assembled to byte and forwarded to the client. The client does not recognise the ETM and think that the Web service is answering.

The ETM is evaluated by implementing test scenarios for a system integration platform that is based on the Open Engineering Service Bus (OpenEngSB). The evaluation results show that although software testers are able to create unit test like integration tests with minimal effort, a high one-off effort has be invested into the implementation of network communication models to filtering purposes. Furthermore, results indicate increased effectiveness in error detection, since it may detect errors which cannot be easily detected with current testing frameworks and by enabling the injection of faulty request or response messages.

The remainder of the paper is structured as the following: Chapter 2 reports on background and related work regarding component-based systems and established concepts. Chapter 3 describes the research issues concerning effective error detection. Chapter 4 describes use case scenarios to clarify the problem statement. Chapter 5 describes the concept and the architecture of the ETM while Chapter 6 presents a prototypic implementation for evaluation purposes which are presented in Chapter 7. Chapter 8 discusses the approach with respect to related work. In the last chapter, Chapter 9 presents the conclusion followed by the future work, which broaden the field of the ETM is presented

CHAPTER 2

Related work

The following section describes the related work. In the first part, process models are described, which gives a global overview over the project, followed by general testing approaches, which have as main porpoise to cover up bugs. Next, the scope of component based system is presented, followed by dependencies between components. After, new test approaches on component based software will be described and at the end different remote concept are introduced, which are the basics used in the use case respectively in the implementation.

2.1 Process models in Software Development

In software engineering systems development models are very important because they structure the project, give a global picture about the future, and implemented parts[12]. There exists a lot of different systems development models like the Waterfall model or the spiral model. In the next section, only a briefly introduction of the V-model and Scrum process model will be presented. Further process models are presented in the corresponding book [12].

V-Modell

The first version of the V-Modell has been presented in the year 1979, which is the basic for the most of the nowadays software models. The V-Modell has two main characteristics, the correlate phase and the validation and verification, which are clearly shown in the Figure 2.1.[13]

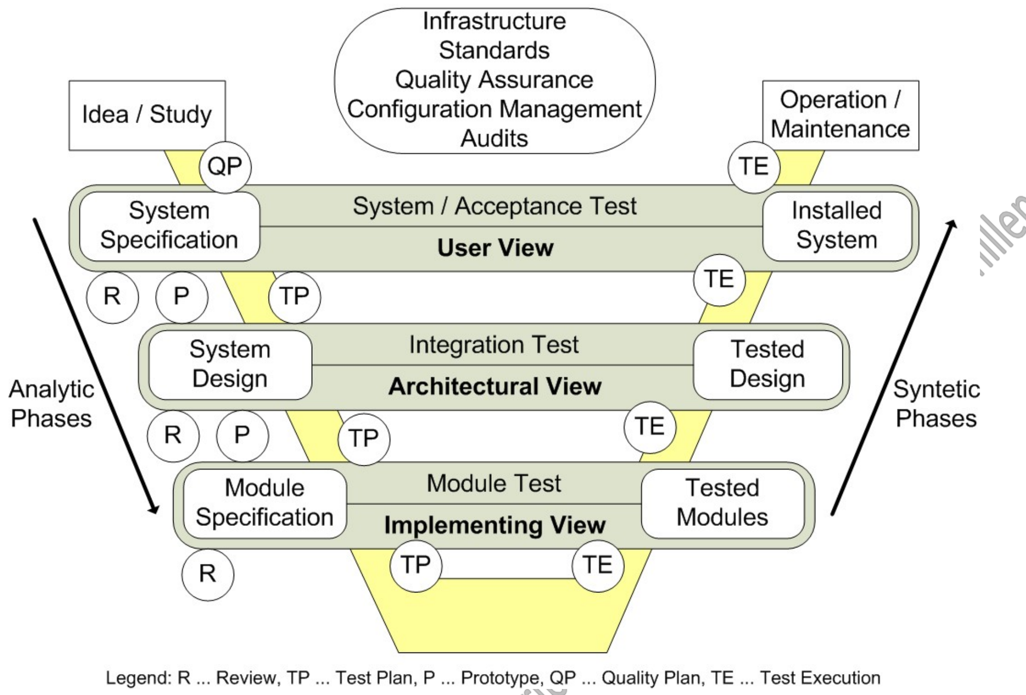


Figure 2.1: V-Modell[14]

- The V-Modell has a similar sequential structure as the waterfall[13] model. For each phase on the left side exists a correlating testing phase on the right side. This model has been the first that included test steps. This is the main different to the waterfall model [13].
- The V-Modell separates between validations (Are we building the correct product?) and verification (Is the project correct in the corresponding phase?) [13].

The V-Modell has the following advantages:

- Separation of process phase and logic to process these phases [13]
- It defines a logical relationship between the phases, which leads to an easy following road for the software development [13]
- For each phase a test document has to be written [13]

The V-Modell has the following disadvantages:

- The documentation effort is very high. Since all the requirements have to be known first, changes in the requirement are very challenging to handle. [13]

- The separation of testing and integration parts leads to problems. [13]
- No adaption for project or organisation is allowed. [13]

In a nutshell, the V-model offers a very good structured and documented project but it should not be used in flexible and fast changing project requirements. [13]

Scrum Process Modell

Scum is mainly a Change-Management approach and it gives a way for the different management layers. Scrum defines how to interact with human, colleges, customers and managers. Furthermore, it gives a direction, how to discipline and responsibility should be handled. From the human point of view, Scum has the use to give space and to unfold the talent of the staff. Scum is not a self-running system, thus need people which controls the process. These people are the so called ScrumMasters. The ScrumMaster needs leadership talent, inspire, passion and creativity[15].

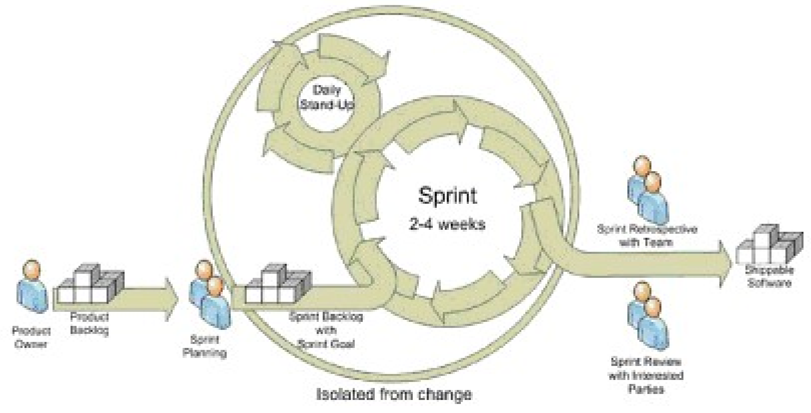


Figure 2.2: Scrum Modell [16]

Scrum makes uses of a series of time blocks, which are called sprint that focus on delivering working software. The time interval for a sprint is typically two to four weeks and is defined by a goal or theme. To allow the team to concentrate on providing working software, springs are isolated from the change [16].

"The consistent sprint duration combined with the team being time boxed to work on features that cannot be changed in that time frame, as well as short meetings and regular retrospective, improve development practice by generating a development rhythm”[16]. The presented rhythm gives the team the possibility to concentrate on designing, developing and implementing high-quality software [16].

2.2 Software testing

Testing is a very important part in the field of software engineering but unluckily it is very rare done. Very often, developer uses the sentence “I don’t have time for testing” but the fact that testing makes software more stable, should change the point of view. Furthermore, when testing is not done, the code has more errors so more time for debugging is needed and so last but not least testing reduces debugging time [17]. Mainly, testing grouped into manual and automated testing strategies. Manual testing strategies are executed by users and have as main feather that unexpected behaviour is tested.

For automated testing strategies several testing approaches exists, as performance, load, integration, and unit tests. The term unit test comes from the pre-object-orientated systems, where tests were for units and not for the total system. Nowadays, unit tests can also be called as component tests [17]. Different definition for Unit tests has been introduced like, “In object-oriented systems, this “unit to be tested” can take different shapes. The span reaches from a single method over a class and subsystem to the entire system”[17]. A second very important test strategy is the integration test or interaction test. The automated testing strategies can be splits in black box testing and white box testing.

First the different between manual and automated testing strategies is presented, followed by the automated testing strategies, white box, black box, grey box, and integration tests.

Manuel tests strategy

Manuel tests are done by a user. It is not possible to make all test cases automatically. One example could the analysing of report. Furthermore, some manual tests are from big advantage. These can be “destructive Tests”[18]. These test aims to force the application to fail. Easy Destructive tests are the following example:

- Put the whole hand on the keyboard
- Shutdown the computer while a process is not finished [18]
- Test the reaction, when the printer has a paper jam [18]

The main problem from manual test strategies is the time for every test execution. To achieve a complete test coverage, probably a lot of testers are needed [18]. From the view of time and benefits, automated tests have clearly more advantages. Anyway, to have good test result a mix between automated and manual tests should be used [18].

Automated tests strategy

Mostly automated tests are used in the software development process. The automated test show there power when the tests are executed repeatedly. The benefits are even more visible in the field of integration tests [9]. With the help of automated test tools incremental software builds are very fast verified. The fact that software is changing and extending very fast automated tests is helping to verify the correctness and stability of software[9]. The here presented solution is manly based on integration tests. More details about automated tests are presented in the section 2.2.

Black box

Black box testing is based on functional request to a program. The quantity of the test cases is based on the input, output and there functional connection[19] as shown in Figure 2.3.

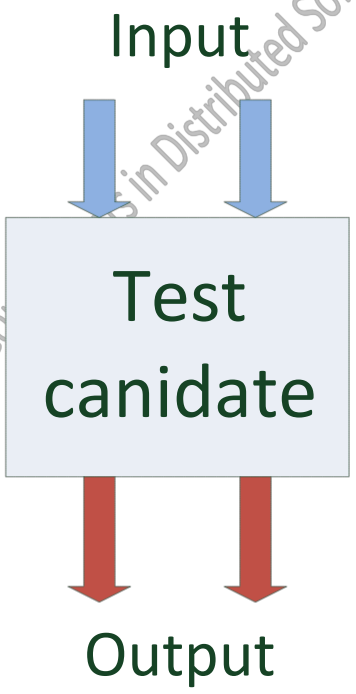


Figure 2.3: Black-Box Testing

For the creation of the test cases, the following parameters are recommended:

- Functional overlapping

  Every function (from the specification) is executed minimal once [19]

- Input overlapping

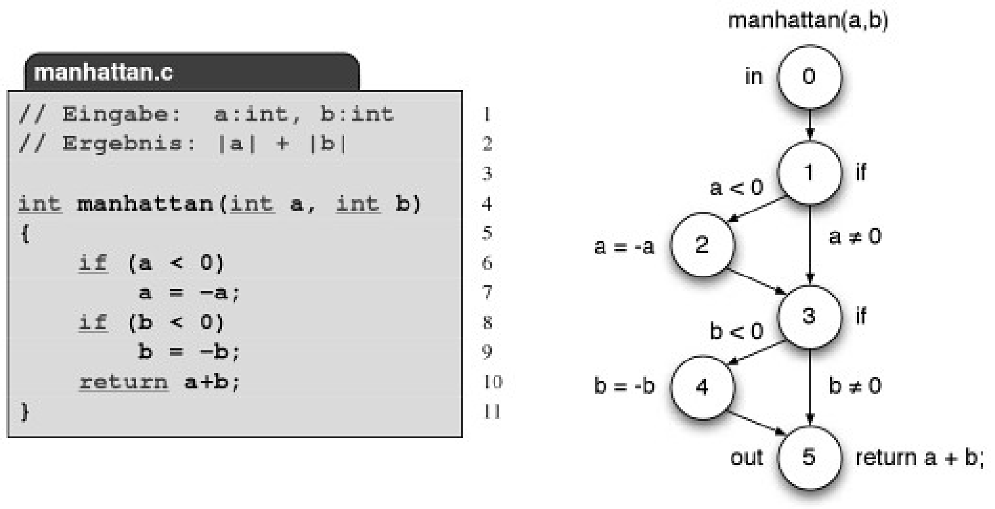
  Every possible input is used in minimal one test case (this does not mean that every possible value should be chosen) [19]

- Output overlapping

  Every possible output, which is defined by the specification, should be tested once [19]

White box testing

In contrast to the black box testing, white box testing is based on the inner structure of the program. Dependent on the aspects of the program, the code is analysed and is used to create test cases. The test cases can be deduced from a data control flow. [20][20][20][20][20][20][20][20][20][20][20][20][20][20][20][20][20]An example for the control flow is the following, which calculates |a|+|b| [20].



manhattan.c

```
// Eingabe:  a:int, b:int
// Ergebnis: |a| + |b|

int manhattan(int a, int b)
{
    if (a < 0)
        a = -a;
    if (b < 0)
        b = -b;
    return a+b;
}
```

Figure 2.4: White box testing example [20]

Grey box testing

The test cases have the same principle as the black box testing. In addition to the black box testing, the software developer takes the code into account to construct the test cases. It follows that all tests, which has been created from the developer fall under the term of grey box testing. Furthermore, in contrast to the white box testing, the test cases are not formally created.[20]

Integration tests

Integration test are testing the architecture of a system. The execution of the integration tests requires knowledge of the component interfaces, which is the reason why integrations are executed by developers. The aim of the integration tests is it, to test the interfaces between the internal components. The arguments of the method calls contains the arguments, which are described by the interface[21].

Integration tests should be build Bottom-up because of the initialisation of the different states of the components. First, components are tested, which have no dependencies, i.e. method calls, which do not implies an invocation of methods from other classes. Secondly, the classes are tested, which invokes operation on already tested classes etc., until the top of the basic classes are tested[21]. An example structure is presented in Figure 2.5.

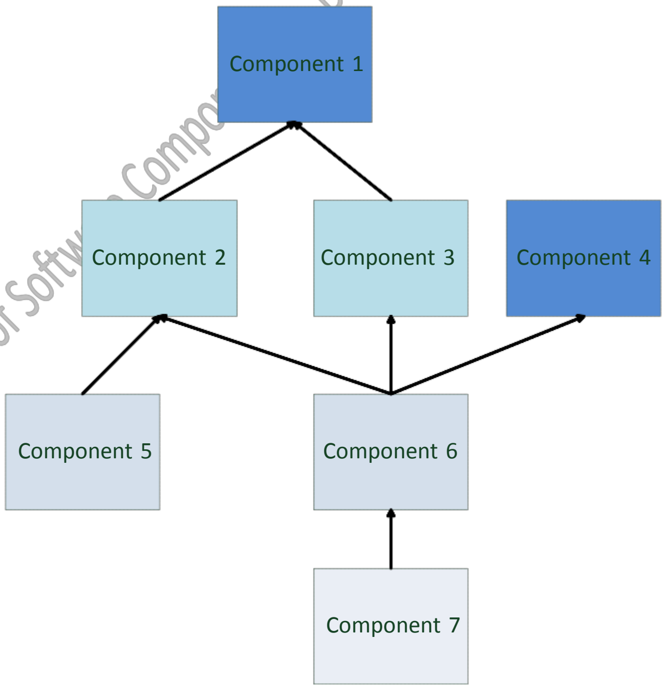


Figure 2.5: Integration testing example

First, the Component 1 or Component 4 is tested because it has no dependencies to other components. Otherwise, when there is a defect in Component 1, it is challenging to find defects when the test case for Component 6 fails. The test order for this example is the follows: Component1, Component 4 → Component 2, Component 3 → Component 5, Component 6 → Component 7.

The integration tests are created from the architecture and so the corresponding return values are tested [21].

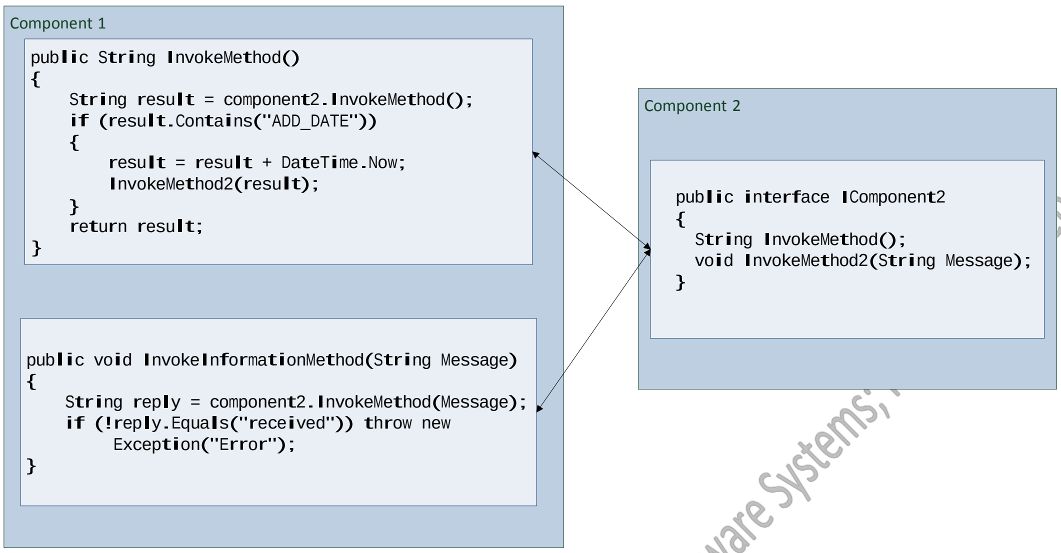
2.3 Component based systems

The following section describes the key concepts of component based systems and their advantages. In the early starts of the software engineering, the object orientate programming has be introduced. With this new concept the applications could be split into objects, which lead to a systematic structure. Furthermore, the object structure gives a better systematically structure but it was challenging in terms of reusability because the knowledge of the internal architecture is required. Moreover, objects are mostly dependent to the development context that leads to a very low reusability. From these problems, the term software component has been introduced[22]. Several definition for a software component has been introduced but they all have in common that functionalities are grouped and capsuled together[22]. One definition is the following: „Software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system"[2]. The system itself should then be built together with several interacting components, which is presented in the Figure 1.1.

In the view of testing components based system, components might be implemented by third parties, which make it challenging to create test cases. The main reason is that they do not always provide total access to source code, which allows only black box testing.

2.4 Dependencies in component based systems

In software engineering, component based development has become an important part[6]. One of the challenging problems in component based systems (CBS) development is the avoiding of dependencies between components[5]. A component should be independent, have a clearly defined interface and interact with a function system[2]. To form a running system, generally the components provide interface, which other components can use. These interaction leads to the described dependencies[7]. Furthermore, from the definition of a components follows that a component should live independently but it does not implies that a component does not need other/external components to fulfil the work[6][5]. The following figure should illustrate dependencies between components.



```
public String InvokeMethod()
{
    String result = component2.InvokeMethod();
    if (result.Contains("ADD_DATE"))
    {
        result = result + DateTime.Now;
        InvokeMethod2(result);
    }
    return result;
}
```

```
public void InvokeInformationMethod(String Message)
{
    String reply = component2.InvokeMethod(Message);
    if (!reply.Equals("received")) throw new
        Exception("Error");
}
```

```
public interface IComponent2
{
  String InvokeMethod();
  void InvokeMethod2(String Message);
}
```

Figure 2.6: Dependencies between Components

The component 1 invokes methods on the component 2 that are defined by the interface description. The components are independent but the component 1 needs the component 2 to fulfil the work. If the component 2 is not present, different test approaches has to be chosen to test the component 1.

Like mentioned before, components often have dependencies to other external components, to form a running system. However, components can also have internal dependencies, which means that they are not only depend on self-generated elements but also have relations between the input and output[6]. Higher dependencies leads to a more complex system[7], which makes it harder to modify, verify and understand the system. It is not only important to know these dependencies, to verify and modify components but also to define the influence of external dependencies in the component behaviour[6]. Dependencies should be handled very important in the field of software testing.

2.5 Testing of components based systems

In the following section new testing approaches for component based system are presented. Several definition of software components exists, like the follows: „software components are binary units of independent production, acquisition, and deployment that interact to form a functioning system”[2]. The important fact is the independents of components, for which several testing tools for different scenarios exist. An example is Jata [23], an language for testing distributed components which mainly integrates the benefits of JUnit[2] and TTCB-3[24]. This

[2] http://www.junit.org/

approach defines the function of a component that allows it to simulate components and supports MOMs and web services.

Another approach is the decomposition and Hybrid approach, which introduces a new formal way, to find the correct behaviour of a component. Furthermore, it combines a model-checking techniques and black box testing[25].

A further tool to test independent components is JRT[4], which wraps java components (JRT-Server), automatically executing the test case (JRTClient) and offers a graphical representation (JRTClientGUI). JRT is implemented for CORBA and the behaviour is illustrated with banking system example[4].

A further way of testing component based system is an approach, which is based on automated statistical tests[26]. This approach is based on state-based component models, which are used to create compact interaction test models. The goal of this approach is to test system interactions and functionalities that are split in several systems [26]. Several approaches and tools are already presented to test component based systems. A general guideline of improving testing in component based systems are presented[27]. The goal is to enhance testing methods and the practicality, especially from the integrator view[27].

Mock-up in component based systems

In software development mostly the systems are complex, which can produce dependencies to external system like JMS middleware or application servers.[11] “All these moving parts can be difficult to manage and provide interacts that are outside the scope of a unit test”[11]. However, to test the business logic or the interaction in the system, the dependencies to other components are not needed and so can be mocked[11] (Figure 2.6). There are several frameworks like Mockito[3], jMock[4], EasyMock[5] or NMock6, which provide ways to mock-up method calls. These frameworks create mock-ups by implementing an own class (mocked class) from the selected class. The behaviour is modelled over keywords, which allows testing and verifying if the interactions in the system are working correctly. The following example, which is based on Mockito, presents the described behaviour.

[3] http://code.google.com/p/mockito/
[4] http://www.jmock.org/
[5] http://www.easymock.org/
[6] http://www.nmock.org/

```
1.  import static org.junit.Assert.*;
2.  import static org.mockito.Mockito.stub;
3.  import static org.mockito.Mockito.verify;
4.  import static org.mockito.Mockito.mock;
5.  import org.junit.Test;
6.
7.  public class GreetingTest {
8.    @Test
9.    public void shouldTestGreetingInItalian(){
10.     //setup
11.     ITranslator mockTranslator = mock(ITranslator.class);
12.     Greeting greeting = new Greeting(mockTranslator);
13.     stub(mockTranslator.translate("English", "Italian",
14.         "Hello")).toReturn("Ciau");
15.     //execute
16.     assertEquals("Ciau Paulo", greeting.sayHello("Italian", "Paulo"));
17.     //verify
18.     verify(mockTranslator).translate("English", "Italian", "Hello");
19.   }
20. }
```

Figure 2.7: Mockito example[28]

In the eleventh row, the class is mocked up with Mockito. The thirteenth row indicates the behaviour of the mock-up. In other words, when the method translate is invoked with the parameters “English”, “Italian” and “Hello” then the return value should be “Ciau”. Next, the behaviour is tested.

The presented mock-up approach implies that the behaviour of the component has to be known and that the code is accessible. However, the method, which is mocked, has to be tested in another test scenario because just the behaviour of it is simulated but the method itself is never executed, which implies that bugs can be hidden behind the mock-up. Nether less, this approach is based on testing the interactions of the system and the correct behaviour of method. This implies that all components, which dependencies to other (external) components are mocked-up in the test scenario. In a nutshell, to test the integration of the system, the methods/components have to be independent to other components or network access.

The mock-up approach can be used in combination with integration tests. Figure 2.8 is related to Figure 2.5.

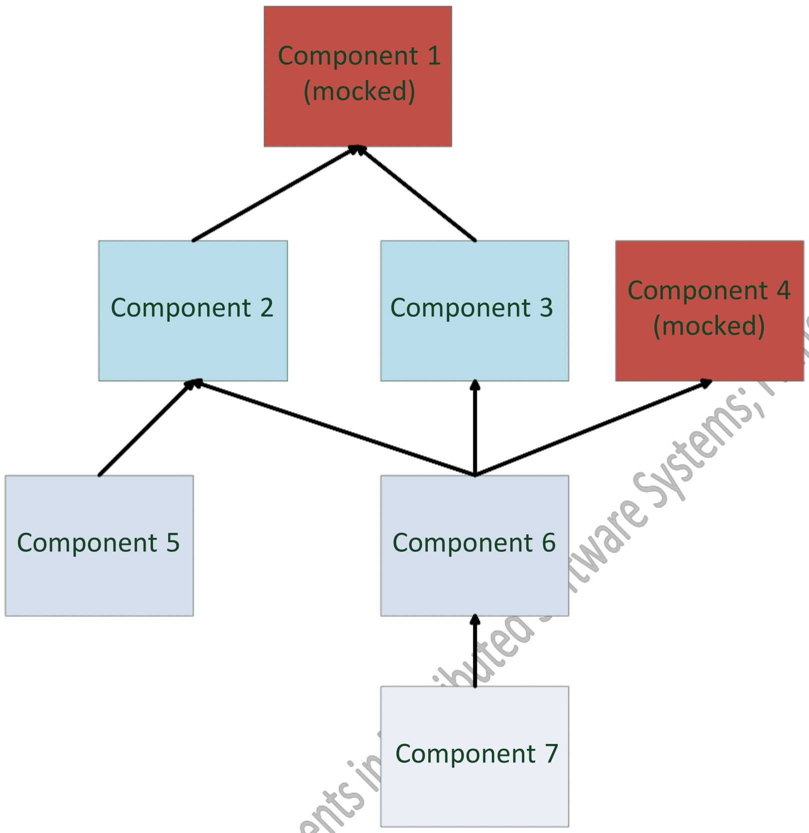


Figure 2.8: Integration tests and mock-up

The component 1 and Component 4 are mocked and other components rely on them. It follows that it is guaranteed that the behaviour of the mocked components is knows and does not have any errors, which leads to the conclusion that components have a high validly. Anyway, testing depended component is still challenging. The question whether component based systems require new testing approaches and introduces a new integrated testing technique has been explored[29] and helps to illustrate the problem of testing component based systems.

2.6 Remote concepts

In the following section, an introduction to different remote concept is presented. First, sniffing and "Man in the Middle" are presented, which are used to record the communication. Next, Web services are described, followed by the Java Message Service.

Recording network communication

This section will describe the basic idea for listening and simulating remote connections. The indicated reference described two ideas, to record message that has been send over Secured Socket Layer (SSL) or Transport Layer Security (TLS) Protocol[30]. The first methods use sniffers to record the encoded message. The Figure 2.9 gives a briefly graphical visualisation of the sniffer concept.

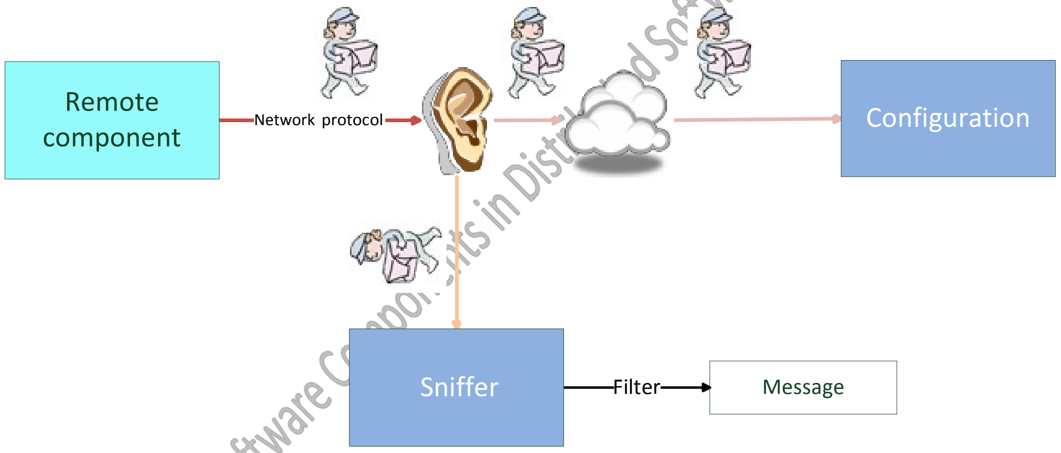


Figure 2.9: Sniffer architecture

The sniffer listens on the network connection and records the received packages. These packages are grouped together and the desired messages are filtered.

These messages are then forwarded to a SSL/TLS decoding component. The approach is a passive approach, which only records the connection and no changes on the components or infrastructure has to be realised. In the second approach some proxies are used. These proxies are logging the connection, the so called Man-in-the-Middle proxies. The concept is an active approach, which means, that some active components have to be included in the system. When these components have a wrong configuration then the whole system breaks down[30].

In contrast to the Man-in-the-Middle, the here presented approach also sends an answer as reply. More detailed, the messages are captured, processed, and corresponding answer is send back. The picture Figure 1.3 from the introduction represents this behaviour.

Web services

"Web services are a logical evolution of software components"[31]. They are integration technologies, which enable a dynamic correlation between components in networks under the use of open standardise internet technologies. They are web applications which offer endpoints on which web services can be published, localised and invoked [32].

In addition, Web services offers interface descriptions that allow the client to invoke methods. The description is providing to the client over "Web Services Description Language" (WSDL). The communication is based on the "Simple Object Access Protocol" (SAOP), the "Hypertext Transfer Protocol" (HTTP), and the "Internet Protocol" (IP) [32]. A more detailed description is presented in the Chapter 3.1 Dependency models .

In combination with the "Universal Description, Discovery and Integration" (UDDI) a publication and search of web services is possible.

Several testing approaches for web services have been introduced. A case study encourages the Test Driven Development (TDD) that presents the applications GRIDL[10] and TxFlow[10], which are distributed and multi-language tools. These tools are using Web services as interfaces between services and client components and offer an easy way to test Web services[10]. Moreover, a possibility is offered to perform the tests in different programming languages.

These approaches are based on the original definition of web service, i.e. Web services are independent. It follows, that these test strategies can just be used in their specific field and cannot be transformed to dependent components.

Java Message Service Application Programing Interface (JMS API)

In the following, the idea of JMS API will be introduced, which received a standard in Java applications. Over the years, systems become better in reliability, increased scalability, and more flexibility. In response of the demand of better and fast systems, developers uses messaging as an answer of the several problems [33].

Messaging is a concept to allow the communication between software components. The messages can be send or receive from any client and a client can send message to any client. Each client connects itself to a messaging agent who provides methods to send, receive messages. Furthermore, messaging allows a loosely coupled communication. More detailed, a component sends a message to a destination and a receiver can retrieve the message from a destination. To communicate with each other, the sender and the receiver do not need to be available in the same time. Even further, the sender and receiver do not need to know anything from the other one. They only need to know the message format and the destination to send/receive messages[34]. Messages differ from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know application methods.[34]

The java API is designed by Sun and several partner companies. They define a common set of interface and associated semantics which allows applications to create, send, and read messages. This Java API, the Java Message Service (JMS) allows the programs, written in the Java[7] programming language to communicate with other application. These application have to implement the API [34]. "The JMS API minimizes the set of concepts a programmer must learn to use messaging products but provides enough features to support sophisticated messaging applications"[34]. The JMS API not only has the feature of loosely coupled communications but also provide the possibility that a provider can deliver messages to a client asynchronous. In other words, a client does not have to request messages in order to receive them. Furthermore, the JMS API can ensure that a message offers the possibility of reliability. This means that a message is delivered once and only once [34].

The main purpose of JMS API, which has been introduced in 1998, was to allow Java application the access to message-orientated middleware (MOM) systems. Since the introduction, a lot of different implementation of the JMS API has been created so that a JMS product can now provide a complete messaging capability for an enterprise [34]. With the release of the version 1.3 of the J2EE platform, a service provider on J2EE[8] technology, the JMS API is an integral part of the J2EE platform, and developer can use the messaging between components using the J2EE API [34].

[7] http://www.java.com/
[8] http://java.sun.com/j2ee/overview.html

CHAPTER 3

Research Issues

Test driven development is a big part in the software engineering process. Test allows to measuring the software quality and to verify software. A lot of testing approaches has been introduced and presented, which helps to find software defects. Modern System are based on component based system, which increase the reusability, reduce defects code and saves costs. A lot of systems are using remote services, which leads to dependencies between components.

From the original definition component has to be independent to other components. To form a running system, components have to interact with each other and often lose their independents to other components. Integration tests can be used when the components are on a single platform and does not make use of the network connection. However, modern software systems are distributed system that implies that the components have to send message over the network interface. These components are the so called dependent remote components. Several test concepts are based on the original definition of a component and thus cannot be used to test dependent remote components. Mock-up frameworks (Section 2.5) overs the possibility to mock-up components but the code has to be open to create expressive test cases. According to these problems it is challenging to test dependent remote components isolated to the entire system. The "Effective Tester in the Middle" (ETM) will present a way to test dependent remote components with common test strategies. The ETM listens on the network interface for messages and send corresponding messages back.

In order to investigate the feasibility and applicability of the proposed approach, we derive the following research issues:

- How can dependencies between software components be modelled, so that system properties are reflected?
- To what extent does the proposed approach improve efficiency of component testing?

- How much does the proposed approach contribute to improving effectiveness of component testing?

The following sections will describe the three research issues in more detail.

3.1 Dependency models reflecting system properties

Dependency models are interaction messages that representing the messages, which are send respectively received from one endpoint of a component .The system properties are the dependent components, which are simulated from the ETM, i.e. the ETM replaces the dependent remote components and simulates the entire system to the component under test. To interact with the components it is imported to know the possible interaction of the component under test, otherwise it is now possible to generate meaningful tests.

Remote dependent components need data from other components to fulfil their work. The interaction between remote components happens over network connections and/or the Internet, which implies that a network protocol has to be used, to indicate the sender and receiver network endpoints. The following figure illustrates the communication between components:

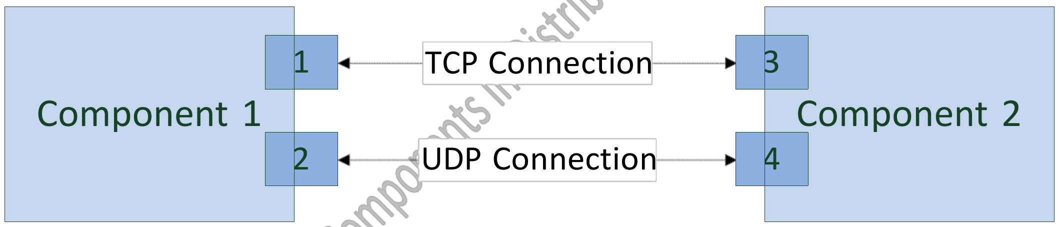


Figure 3.1: Communication between components

Every component contains at least one Endpoint. These endpoints are defined by a port and an IP-address. In the Figure 3.1 the components have each two Endpoints, which are using different technologies to communicate with each other.

These different technologies are the so called Transport protocols, which provide a way, to send and receive data. In most of the cases, the Transmission Control Protocol (TCP) [35] or User Datagram Protocol (UDP) [35] is used.

After the transport layer very often an application protocol is used that allows the application to filter the information in an efficient way. The application protocol can be based on other application protocols. The technical dependency structure is illustrated in the following picture.

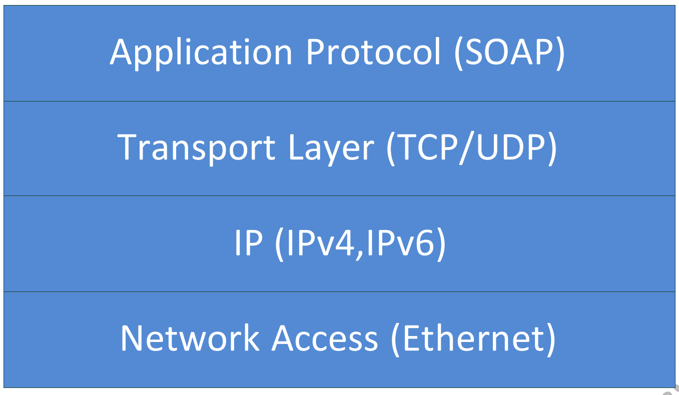

Application Protocol (SOAP)
Transport Layer (TCP/UDP)
IP (IPv4,IPv6)
Network Access (Ethernet)

Figure 3.2: Technical dependency

An example for this structure is SOAP. SOAP is an application protocol, which uses the HTTP[9] or HTTPS[10]. HTTP and HTTPS are both using TCP as Transport Protocol that again uses the IP (IPv4 or IPv6) Protocol. Last but not least, IP uses the network access, which can be for example Ethernet or Token Bus. This example is represented in Figure 3.3.

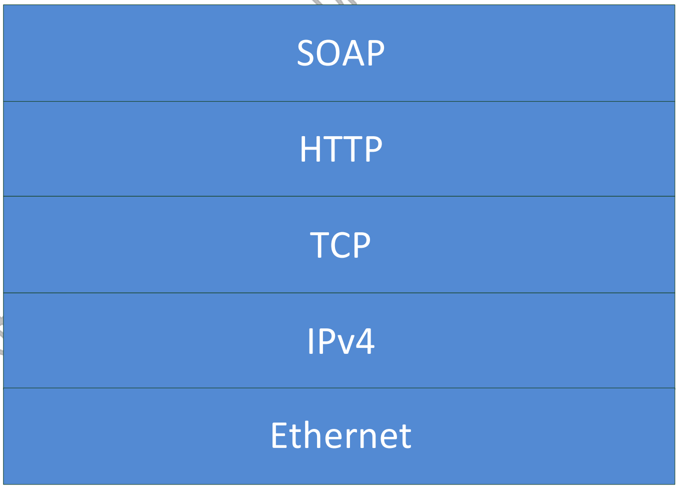


Figure 3.3: SOAP example

[9] http://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol
[10] http://de.wikipedia.org/wiki/Hypertext_Transfer_Protocol_Secure

All the layers under the transport layer are standardized and handled by the specified programming language and the operation system. Application protocol does not need to be standardized. Like mentioned, the programming language allows to receive data from the network connection over the transport layer. At this time, the received data cannot be directly used because the data is in the byte format and the application protocol information is not split from the original message. By filtering the application protocol data from the received data, the message that has been really sent, is retrieved. The SOAP example of Figure 3.3 shows the received message (Transport Layer) from a request to execute a method *HelloWorld* with *no parameter*. The data from the Transport Layer is coded in binary and could be encrypted. For readability, the data is shown to a plain text String. Like described, the components are sending the message with an application protocol to the dependent components. On order to analysed and send corresponding message can back, an interaction messages is needed that include the protocol information and the corresponding message, i.e. the framework can receive the data but it does not know how to handle the application protocol, the message and if a response is requested, how this should look like (Figure 4.4, 2, 3, 4, 5). There has to a configuration, which explicitly describes, how the framework has to behave. The evaluation is done by analysing the model. The model should represent the dependencies and should reflect the system properties.

3.2 Effectiveness of the proposed approach

The effectiveness of the ETM approach is interpreted by the amount and diversity of bugs that can be found in addition to traditional approaches. With the traditional approach the complete system has to be started and all the components have to be already implemented. The start-up time for the complete system takes time, human and financial resources. Furthermore, in the software development system not all the components are implemented at the same time or are third part resources, which makes it difficult to start the complete system and create correct test cases.

Components communicate with dependent components by sending messages over the transport layer. These messages cannot (or only by high effort) be manipulated so that invalid massages cannot be generated, which implies that the reaction of components to incorrect messages cannot be tested. After a release, components can be updated, modified or replaced with other components introducing additional hidden bugs into the system. The ETM can generate user defined messages, which helps to generate invalid messages. The question is, in which way the ETM can generate messages to find bugs and to be more effective than the traditional approach. It is important to know if the approach is effective to allow a comparison to other approaches.

The evolution is done by analysing the expressiveness and power of the test cases, i.e. is it possible to create test cases with the ETM in a way that it is more effective to use the ETM instead of other approaches.

3.3 Efficiency of the proposed approach

Software testers have difficulties, to test remote components, since they have to start the entire system to execute test cases, which leads to losing time and money. The proposed ETM seeks to model dependencies between the components and thus testers do not have to start the entire system. The ETM needs an interaction model and has to know the used application protocol. The setup for the application protocol and the interaction model need time which leads to the question, if it takes more time to create the configuration for the ETM than starting whole system. Furthermore, is there a size at which it is recommended using the ETM approach instead of the tradition approach because of the start-up time? An example of an application protocol is presented in the following figure, which is based on the SOAP protocol.

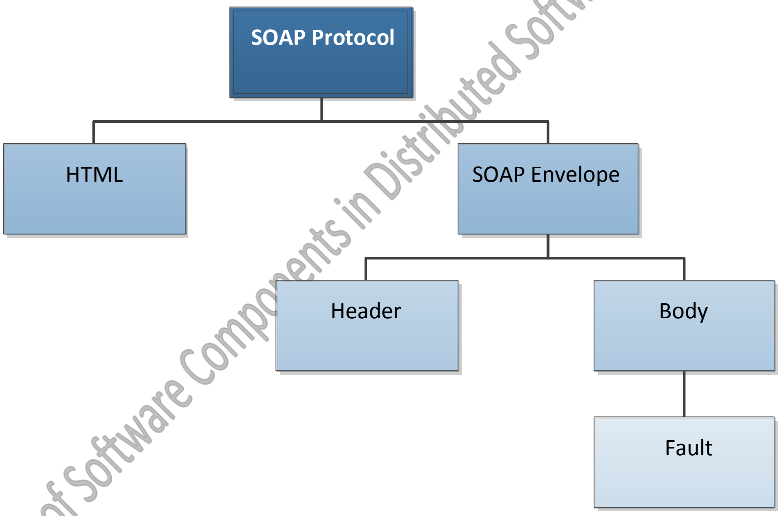


Figure 3.4: SOAP Protocol

Like described in the previous section, the SOAP protocol consists of a HTML, Header, Body and Fault elements. Before, this protocol can be implemented, the specification has to be analysed. The time to analyse the protocol is related to the complexity of the protocol.

The numbers of interaction models between the protocols increases with the numbers of messages send between the components. The complexity and the numbers of interaction increase the time to implement the interaction. The following figure shows that the time to implementing the interaction models is mainly based on the complexity.

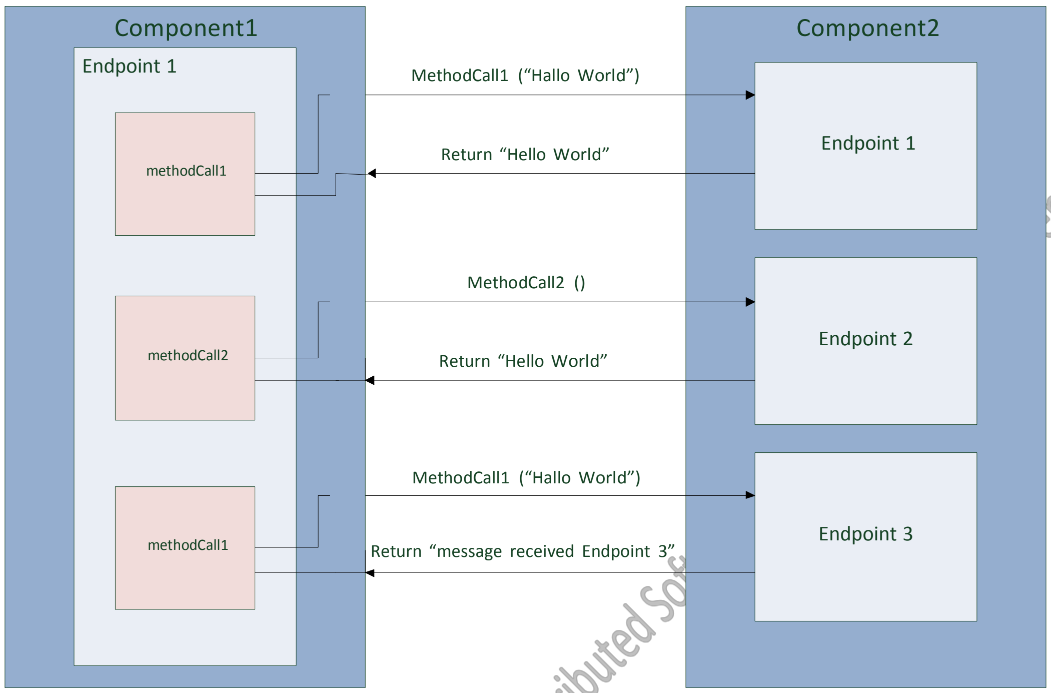


Figure 3.5: Complexity example

The Component1 is the component under test, the method calls are invoking methods on different sockets. The interaction configuration is getting more complex because the method can invoke methods on different endpoints or could switch endpoint. It is important to know if the approach is efficient to allow a comparison to other approaches.

The evolution is done by measured the implementing effort (Time and Lines of code) for test cases on a defined used case scenario. The implementation effort includes the implementation of the application protocol, the behaviour and the test cases itself.

CHAPTER 4

Use Case

In the following section a use case will be described that defined the requirements to develop the presented approach.

Remote component based systems aims reusing and assembling software systems of software parts, which could be created by third-parties with inaccessible source code. Introduced test methods (Chapter 2) helps to find software defects. However, testing in distributed environments is very challenging. An example is the network connection, which does not assure that all the messages are correctly transmitted or if the remote dependent component is available.

One example for these presented remote systems is the so called Engineering Service Bus (EngSB) [36] [37]. Current developers of software systems use a wide range of tools from software vendors, open source communities, and in-house developers. To get these tools working together to support a development process in an engineering environment is still challenging because there is a wide variety of standards, which these tools implement [38]. In comparison to the Enterprise Service Bus which integrates services [39], the EngSB integrates not only different tools and systems but also different steps in the software development lifecycle [36] [37] - the platform aims at integrating software engineering disciplines. First, the Open Engineering Service Bus is presented, an implementation of the EngSB concept, followed by the used case.

4.1 Open Engineering Service Bus

The Open Engineering Bus (OpenEngSB) allows different tools to communicate over a bus[11]. The core architecture is implemented in Java and is based on the OSGi[12] specification. The communication works over connectors that can be written in different programming languages.

[11] http://computer.yourdictionary.com/software-bus
[12] http://www.osgi.org/Specifications/HomePage

The OpenEngSB is a new implementation of the Engineering Service Bus concept, which is described in detail in [40] and [41]. The concept of the OpenEngSB involves the connector and domain concept, where a domain has basically the interface description and the connector is an implementation of this interface.

Domain and Connectors

In [37] he concepts of “Tool Domains” and “Tool Connectors” have been introduced. Tool domains are used as description for a group of tools. These descriptions can be implemented by Tool connectors and can take place, where the descriptions are defined. “Tool domains could be compared best to the concept of abstract classes in in [sic!] object orientated programming languages”[40].

The definition of a tool connector has been defined as follows: “connect external tools in a protocol and platform independent approach to the OpenEngSB”[40]. In the following, this document reference Tool connectors just as “Connector”.

The following picture should illustarte the interatcion between connectors and Domains.

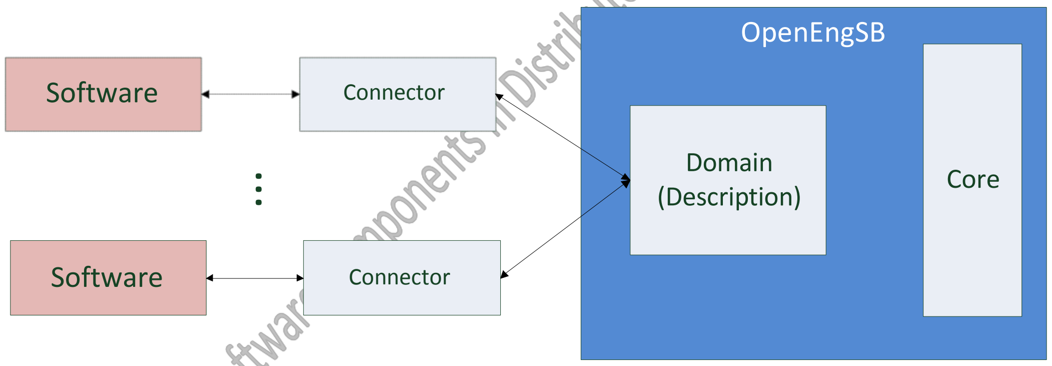


Figure 4.1: Interaction between Connectors and Domains

A domain can have zero or numerous connectors, which are communicating with the OpenEngSB. Behind the connector software, tools or other pieces of code can communicate with the connector.

The .Net Bridge is a connector but additional converts the received messages to a .Net code. This special kind of connector allows software, which is implemented in .Net to communicate with the OpenEngSB.

.Net Bridge

The .Net bridge is, beside the OpenEngSB a key component in this thesis. The concept, “Effective Tester in the Middle” (ETM) has been introduced from the testing problem with the .Net Bridge. This does not imply that the ETM can only be used for this specific component. Only the concepts and implementation are presented on the .Net Bridge, which is a dependent component.

The .Net Bridge offers the possibility to .Net developer to communicate with OpenEngSB. Furthermore, the .Net Bridge handles the creation, registrations, unregistration and deletion of a connection with a specific domain. Next, it provides a possibility to invoke methods on the chosen domain, by transforming a method call to a JSON message and forward it to the domain. This message contains all the information for a method call, like for example the name of the method and the parameters. Additional, the .Net Bridge converts messages from the domain that are in JSON format to method calls and invoke them. The whole messaging happen over two different types of ActiveMQ (6.1) queues. The sequence of interaction between the OpenEngSB and the .Net Bridge is presented in Figure 4.2.

In the first step, the factory initialises a creation of a connector, which implies that the event handler first opens a receive queue. This queue is listening permanently and waits for messages that have been sent from the OpenEngSB. Next, the connector is created by forwarding a creation method call to the OpenEngSB. The OpenEngSB answers with a void message that symbolise that the method call has no return value. When the creation of the connector is complete, the .Net Bridge registers itself with a register method call to the OpenEngSB. This behaviour is the same for unregister and delete the .Net Bridge.

As described, the .Net Bridge offers the possibility to invoke specific methods on the OpenEngSB. These methods are provided over Web Service Description Language (WSDL[13]) that stands for the interface of the tool domain. These WSDL are converted to an interface that is provided to the .Net Bridge over a dynamic-link library (dll[14]). By invoking a method call, the OpenEngSB answers with a call back method call to the receive queue. The message is converted and the corresponding method is invoked. The result is send back to the OpenEngSB (if the method is a void method, a void message is send back). The dll (from the WSDL) contains a second interface, which symbolise the methods that can be called from the OpenEngSB. In other words, the user implements the second interface and provides it to the .Net Bridge.

[13] http://www.w3.org/TR/wsdl
[14] http://msdn.microsoft.com/en-us/library/1ez7dh12(v=vs.80).aspx

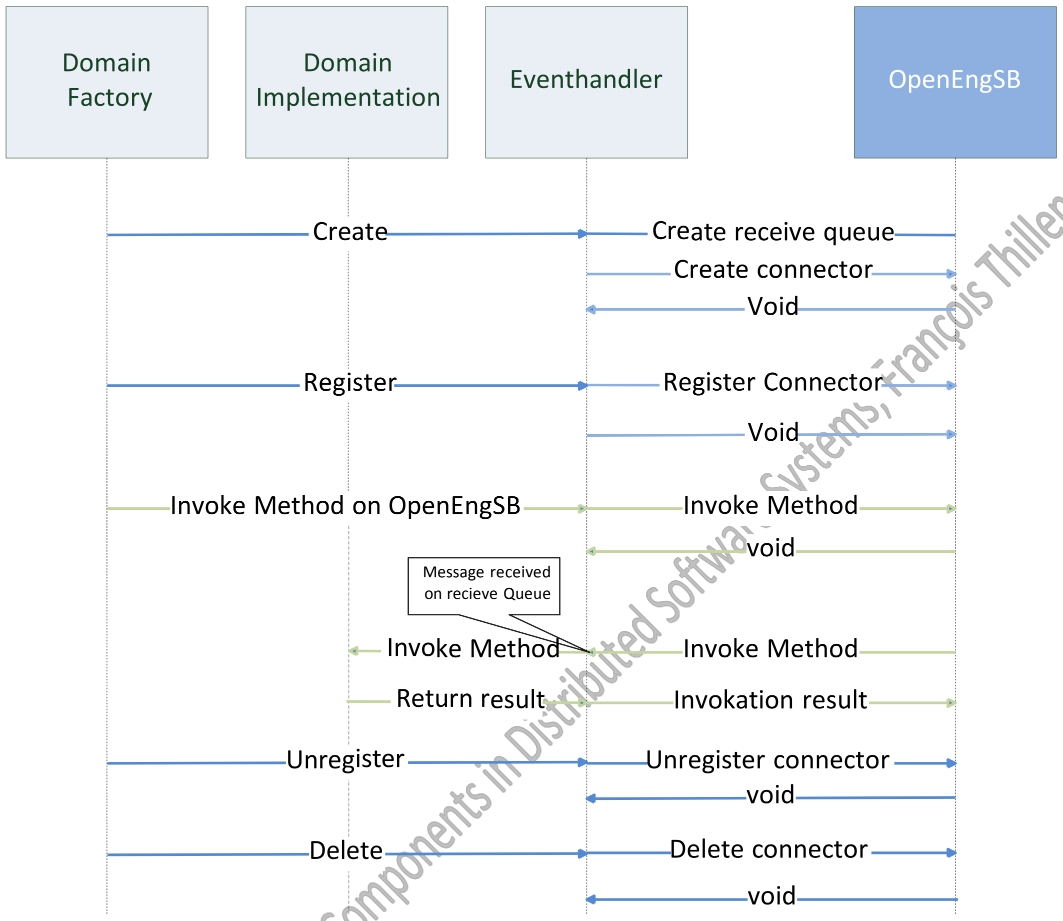


Figure 4.2: Sequence diagram of the .Net Bridge

Each method call to the .OpenEngSB, the .Net Bridge opens a temporary queue that is only valid until it receives the method call result. After, the queue gets closed. The receive queue is valid as long as the .Net Bridge is valid. Furthermore, this queue gives the possibility to the OpenEngSB to communicate with the .Net Bridge.

The .Net Bridge and the OpenEngSB are both components, while the .Net Bridge is dependent to the OpenEngSB

4.2 Concrete Use case

A concrete architecture of the testing purpose is the EngSB [42] (respectively OpenEngSB), the presented .Net Bridge (4.1) and the Engineering Object Editor (EOE). Like mentioned, the key concepts of the EngSB are Domains and Connectors, which are used to offer the possibility to communicate between engineering tools and the EngSB. Domains provide engineering tool instance independent interfaces, which have to be implemented by the connector and mapped onto the engineering tool instance specific behaviour. The .Net Bridge implements a chosen interface from a domain (e.g. EngSB) and provides a translation between java and .Net. A .Net implementation (e.g. EOE) uses this offer and is so able to communicate with the EngSB.

The communication between remote connectors and a domain works over JSON messages (send from ENGSB) to .Net Objects and vice versa. An example of a method call, which is converted transmitted over JSON, is the following.

```
{
  "methodCall":{
    "classes":[
      "org.openengsb.domain.example.event.LogEvent"
    ],
    "methodName":"doSomethingWithLogEvent",
    "args":[
      {
        "level":"12",
        "message":"TestCase1",
        "name":"Test",
        "origin":"123",
        "processId":123
      }
    ],
    "metaData":{
      "serviceId":"0e25f8d8-174b-470a-bc18-65c84c3df01a"
    }
  },
  "answer":true,
  "destination":"tcp://localhost.:6549?0e25f8d8-174b-470a-65c84c3df01a",
  "principal":null,
  "credentials":null,
  "timestamp":0,
  "callId":"646bf4e9-1077-4188-b58a-787d610a135a"
}
```

Figure 4.3: Method call example in JSON format

In the method call bracket, the field *classes* define the type of the arguments. Second, the method name is defined to indicate the corresponding method, which should be invoked. The parameters to invoke the method are defined in *args*. The order in *classe* and *args* is important because the first position in *classes* means that the first argument in *args* is of that type. The *metadata* tag is used for the EngSB to identify the connector. The *answer*, *destination*, *principal*, *credentials*, *timestamp* and the *callId* are used for the EngSB to defining the next step of the interaction with the connector.

These messages are used to invoke methods on the .Net Bridge and EngSB side. The EOE is a concrete implementation, which uses the .Net Bridge to checkout and commits information on the EngSB. The following figure illustrates a simplified structure of the system, the relations between the components, the flow between the components, and the problem space of that environment.

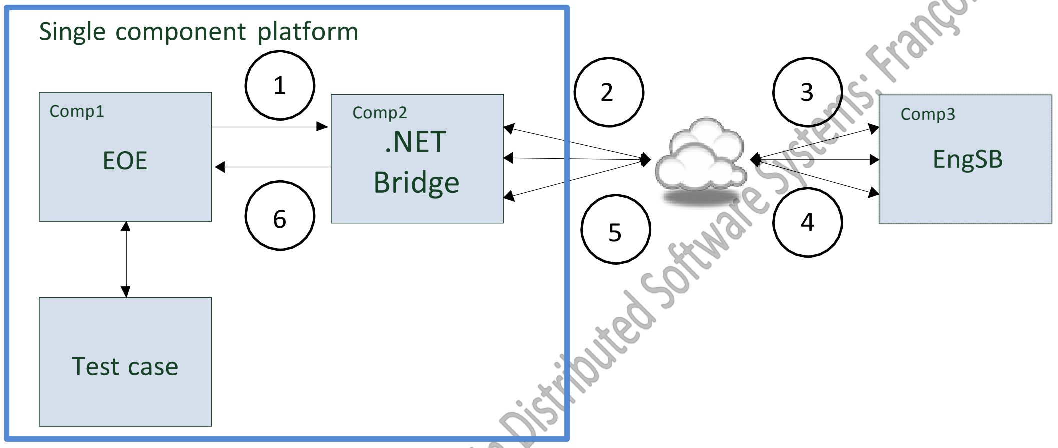


Figure 4.4: Communication between EOE, .Net Bridge and the EngSB

The EOE is an excel plugin, which does a visual representation for data stored in the EngSB. Therefore whenever the EOE checks out (Figure 4.4, 1) via a domain then the method call is forwarded to the .Net Bridge and blocks until the data arrives. The .Net Bridge receives the method call, marshals the method name and parameters to a JSON object (Figure 4.3) and forwards this request over the network connection to the EngSB (Figure 4.4, 2). The communication of the .Net Bridge with the EngSB works over a Message-oriented Middleware (MOM), like ActiveMQ (6.1), which supports the JSM API and several extensions [43]. The EngSB receives the JSON message (Figure 4.4, 3), unmarshal it to an object and invoke the corresponding method. The method result is marshalled (Figure 4.4, 4) and transmitted to the .Net Bridge over the MOM. The .Net Bridge does the same like the EngSB, e.g. receives the JSON message (Figure 4.4, 5), converts the message to a method call and invokes the corresponding method, with the corresponding parameters. In this case, the method call has as parameters the result from the EngSB method call. By a call back method, the data are forwarded to the EOE (Figure 4.4, 6), which then returns from the blocking state and represent the received data.

Test cases invoke methods on the EOE. However, to test the EOE, the complete system has to be started, which correspond to start the EngSB, .Net Bridge, and EOE. When these sub systems are running, then the test case can be executed. However, starting up the entire system for running defined testseenarios costs efforts, human resources and time.

CHAPTER 5

Proposed Solution Approach

The "Effective Tester in the Middle" (ETM) approach relies on test scenario specific interaction models and network communication. In the following the approach is described in an abstract way, which consists of three main parts: The ETM core takes care of the communication and handles messages. Like introduced in previous sections, the components communicate over network connections with each other. The ETM listens on the Transport Layer for messages. The first prototype of the ETM considers just TCP connections. In the next stages the ETM was able to consider SOAP and ActiveMQ commands as well. The ETM opens first a socket, which listen on the specified port and IP-address. When the socket receives messages, which are in bytes, it parses throw all the configured of the specified protocol types and asks if all data has been arrived. If there are still missing messages, the socket combines the old received data with the new ones and parse again throw all protocols. On the other hand, when the protocol replies with all data have been received, the ETM parses throw the configuration and forwards the data to each configuration. The configuration analyses the data and checks if the message is complete. When this is the case, the ETM uses the response interaction model and send it to a specific socket that is bound to the client. The application protocol model references the protocol, which the component uses. Examples are SOAP (Section 3.1) and ActiveMQ (Section 6.1). These protocol implementations serialize byte message to protocol messages and deserialize the message from the received protocol message to a desired type. Every protocol has characteristics, which shows that it is valid. SOAP for example needs an HTML part and en open/closing envelope tag. These characteristics of the protocols can be used to identify if messages in a byte format are valid. The interaction model represents the message transfer between the components, and needs the following parameters.

1. Port and IP-address from which a client sends messages

2. A message in a protocol format, which stand for the request message
3. Interaction models referencing a reply (Can be null)
4. Port and IP-address, on which the ETM itself should open to receive requests, form a client.

These models are used by the ETM to correctly reply to exchange messages. Figure 5.1 shows the integration of the ETM in the use case presented in section 4. In contrast to the traditional system, communications with the EngSB are simulated by the ETM. The EOE does not recognise that the EngSB is simulated. This means that check out and check in are still performed the same way (Figure 5.1, 1 and 6). Also the .Net bridge does not know that the ETM is the responder (Figure 5.1, 2 and 5). The ETM catches the communication messages (Figure 5.1, 4), parses the request, and replies with a corresponding message to the .Net Bridge (Figure 5.1 , 5), which then forwards it to the EOE. The complete single component platform does not need any changes or adaptation. The system does not know at any point that it is not communicating with the original component that is in this case the ETM.

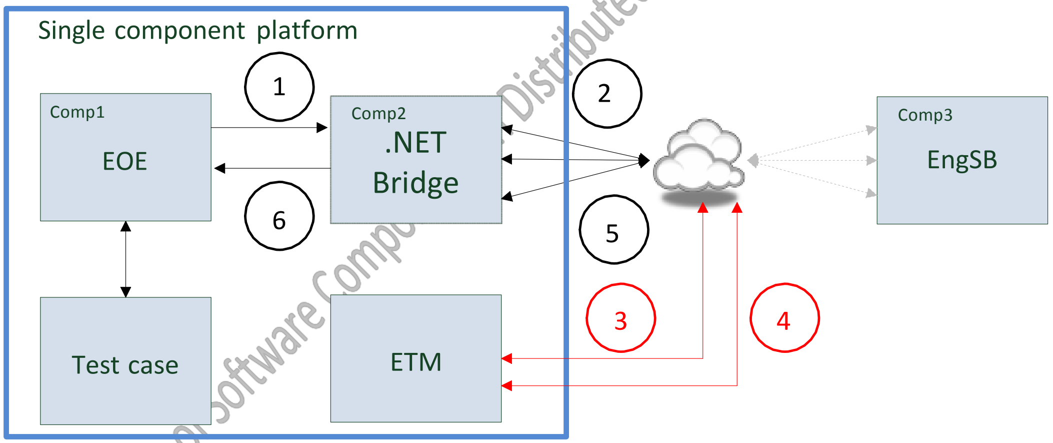


Figure 5.1: ETM in the presented environment

CHAPTER 6

Implementation

The "Effective Tester in the Middle" (ETM) is based on specific interaction models and network communication models.

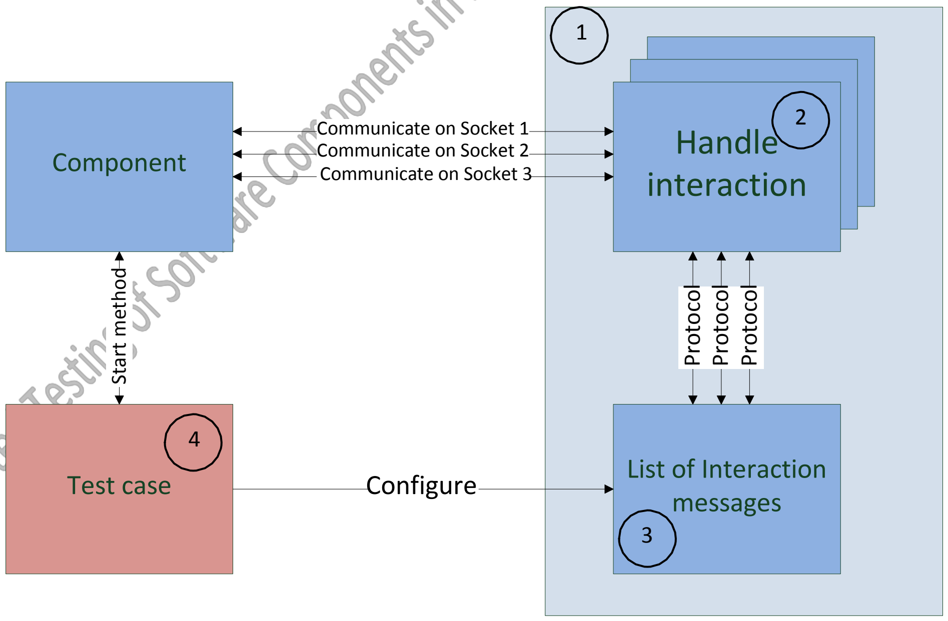


Figure 6.1: ETM technique concept

In the following a detailed description about the implementation is described and consists of the following main parts. First, an introduction to the ActiveMQ application protocol is described. Next, the ETM core (Figure 6.1, 1) is presented, which itself contains, the socket handling (Figure 6.1, 2), triggering a method, and a new list implementation (Figure 6.1, 3). The ETM core is followed by the interaction model and the application protocols. In the last section an example use case is presented, which describes the interaction between the components (Figure 6.1, 4). All the sub points are grouped together in Figure 6.1.

6.1 ActiveMQ

ActiveMQ is an open source message broker, which is generally stable and high-performing. It might be run standalone or inside another process, application server or java EE application. Furthermore it supports the JSM API (0) and several extensions [43]. The following section gives a short explanation of the ActiveMQ protocol for the ActiveMQ example in sub point 6.4.

The communication between ActiveMQ clients and an ActiveMQ server works over transport layers. On the view of the application protocol, ActiveMQ uses the OpenWire protocol which is a cross language Wire protocol [44].

Objects are marshalled as commands and forwarded to the corresponding endpoint. On the view of a network connection, commands are back to back on a stream. Furthermore, commands are not delimited in anyway and are variable in their size [45]. There are several types of commands [45] but for the simulation of the presented use case, not all of these are needed.

Before it is possible that ActiveMQ can send some message several commands have to be transmitted. A general overview is presented in Figure 6.2.

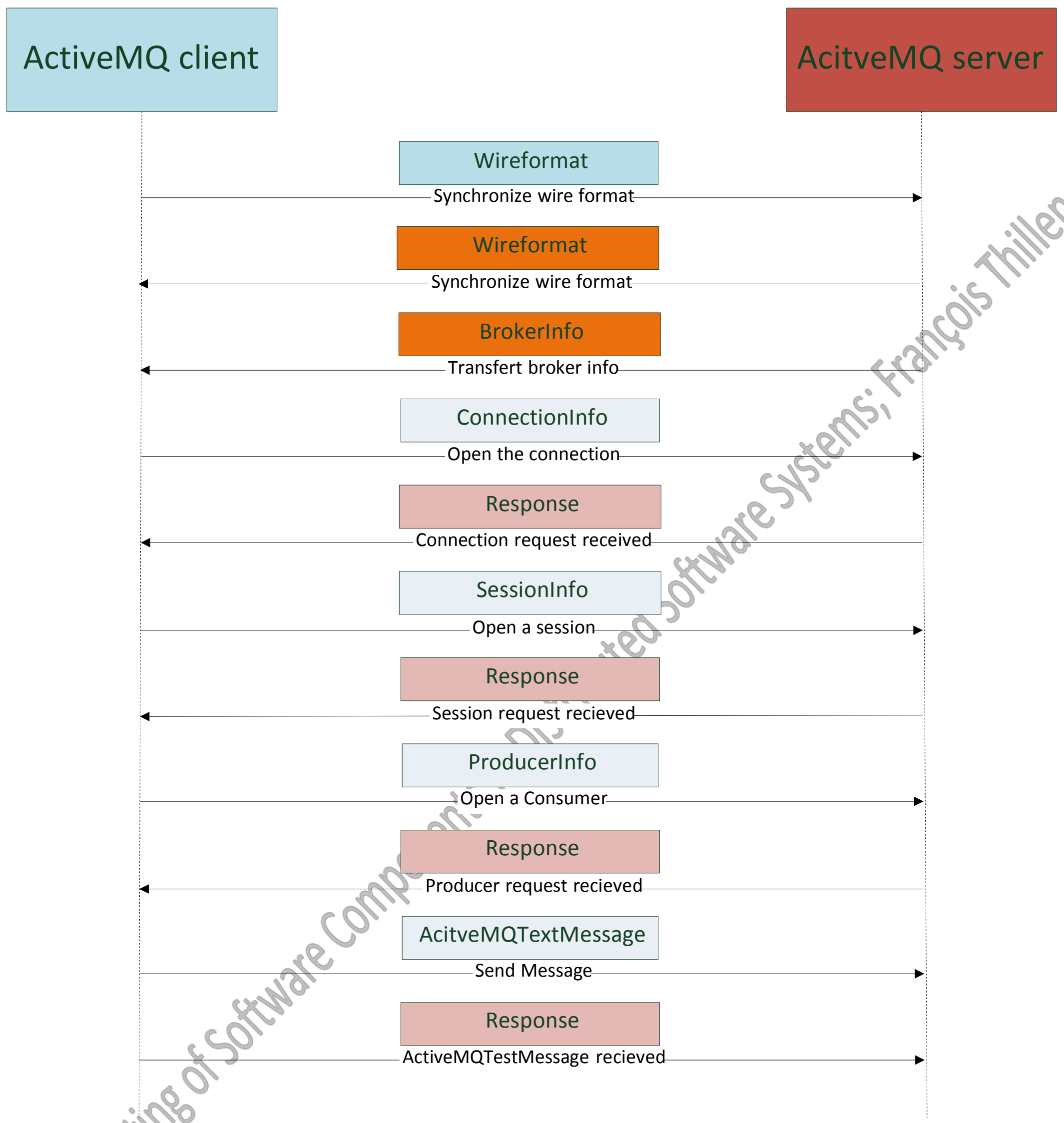


Figure 6.2: Send an object with ActiveMQ example

First, in opening a consumer or producer, a Wire format command is send, which serialise and deserialize messages. Both sides need a wire format, which have to be synchronised and valid. The client (starter of the synchronisation process) requires as response a Wireformat and a brokenInfo command. In the next step, the corresponding commands are sent (ConnectionInfo, SessionInfo, and ProducerInfo or ConsumerInfo) to open a consumer respectably a producer. For each of this commands (excepted Wireformat) a Response command is required to valid the corresponding command. This Response itself requires a correlated command id, which can be retrieved from the received commands. After the validation of these commands, it is possible to send ActiveMQ objects. These objects can be of different types as for example the here used ActiveMQTextMessage. To create a consumer, in contrast to the producer example (Figure 6.2), a ConsumerInfo command is send instead of a ProducerInfo. To send from the ActiveMQ server side an object to a consumer, a dispatcher command is needed. The structure is shown in the Figure 6.3.

```
public class MessageDispatch : BaseCommand
{
    public const byte ID_MESSAGEDISPATCH = 21;
    public MessageDispatch();
    public ConsumerId ConsumerId { get; set; }
    public ActiveMQDestination Destination { get; set; }
    public Message Message { get; set; }
    public int RedeliveryCounter { get; set; }
     ...
}
```

Figure 6.3: Message dispatcher structure

The dispatcher needs the ConsumerId and the destination of the consumer. This information can be retrieved from the received commands that have been recorded since the communication started. For each consumer or producer that has been created a different thread is opened, which communicate with the server over a new port. In a nutshell, to send a message to a consumer, not only the ConsumerId and Destination are needed but also the used endpoint.

6.2 The ETM core

Like introduced in previous sections, the components communicate over the network connection with each other. The ETM listens on the Transport Layer for messages. The first prototype considered only TCP connection. Next, web services have been included and last an integration of the ActiveMQ protocol has been realised.

The ETM opens first a socket, which listen on the specified port and IP-address and accept incoming sockets asynchrony. When the socket receives messages that are in the byte format, it parses throw all the configured and specified protocol types and verify if all data has been arrived. If this is not the case, the socket combines the old received data with the new ones and parse again throw all protocols. On the other hand, when the protocol is valid, the ETM parses throw the configuration and forward the data to each configuration. The configuration analyse the data and analyse if the message fulfil the requirements. If this second validity check is true then this message is picked and the corresponding message is send back to the client. This concept is explained in the following sub sections.

New List implementation

Typically, components are sending message with the same type. It follows that the answer for the first message should be the first interaction model and the second request should be the second. This behaviour is presented in the following picture.

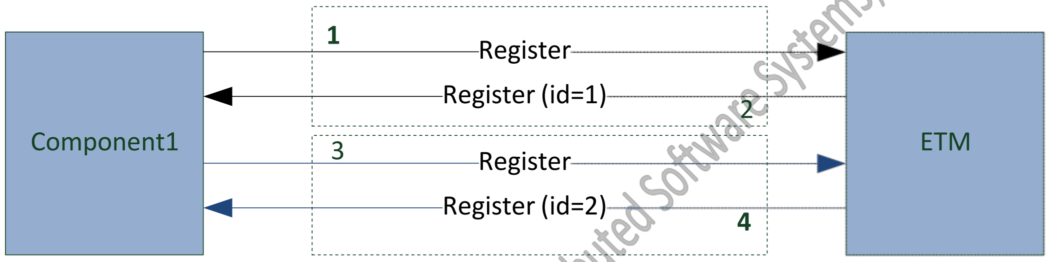


Figure 6.4: Message Request Order

The first message from the component1 is a registration message (Figure 6.4, 1), which is caught by the ETM, processed, and a message from the configuration is send back (Figure 6.4, 2). This could be for example an identifier (id). In the next step, the component1 send a second registration message (Figure 6.4, 3), which requires is of the same type as the first one. Corresponding to the example, the answer should contain a different identifier values as the previous (Figure 6.4, 4).

From the concrete implementation point of view, the new list has a counter for each configuration, which is initialised with zero. When an interaction message gets picked, as answer then the corresponding counter get increased by one (Figure 6.5, 2 and 4). This behaviour is realised by wrapping the interaction message with a counter in a new class. The search for the configuration is structured as shown in the following code part.

```
public InteractionMessage SearchElement(IProtocol item, int socketID)
{
    AddMessagesForAllSocketsToSpecificSocket(socketID);
    CountedInteractionMessage canidates = null;
    int min = int.MaxValue;
    foreach (CountedInteractionMessage message
            in interactionMessagesPerSocket[socketID])
    {
        if (CompaireProtocolAndInteractionmessage(item, socketID, message))
        {
            if (message.PickedNumber < min)
            {
                canidates = message;
                min = message.PickedNumber;
            }
        }
    }
    if (canidates != null)
    {
        canidates.PickedNumber++;
    }
    else
    {
        return null;
    }
    return canidates.InteractionMessage.Clone() as InteractionMessage;
}
```

Figure 6.5: Search algorithms with order

First the protocol, which is the converted from bytes is compared with the interaction models that are grouped by the socket number. If this fulfils the criteria, next the return number is compared. This number stands for the numbers returns, i.e. how often the interaction message has been picked and returned. Summarised, the interaction message that fulfils the condition and has the lowest number of returns is picked. This list enables an easy and thread safe way to retrieve interaction models.

Trigger Method

Like shown in the Figure 6.2, the ETM is reacting on received message. However, components can wait for method calls from outside, like shown in the following sequent diagram.

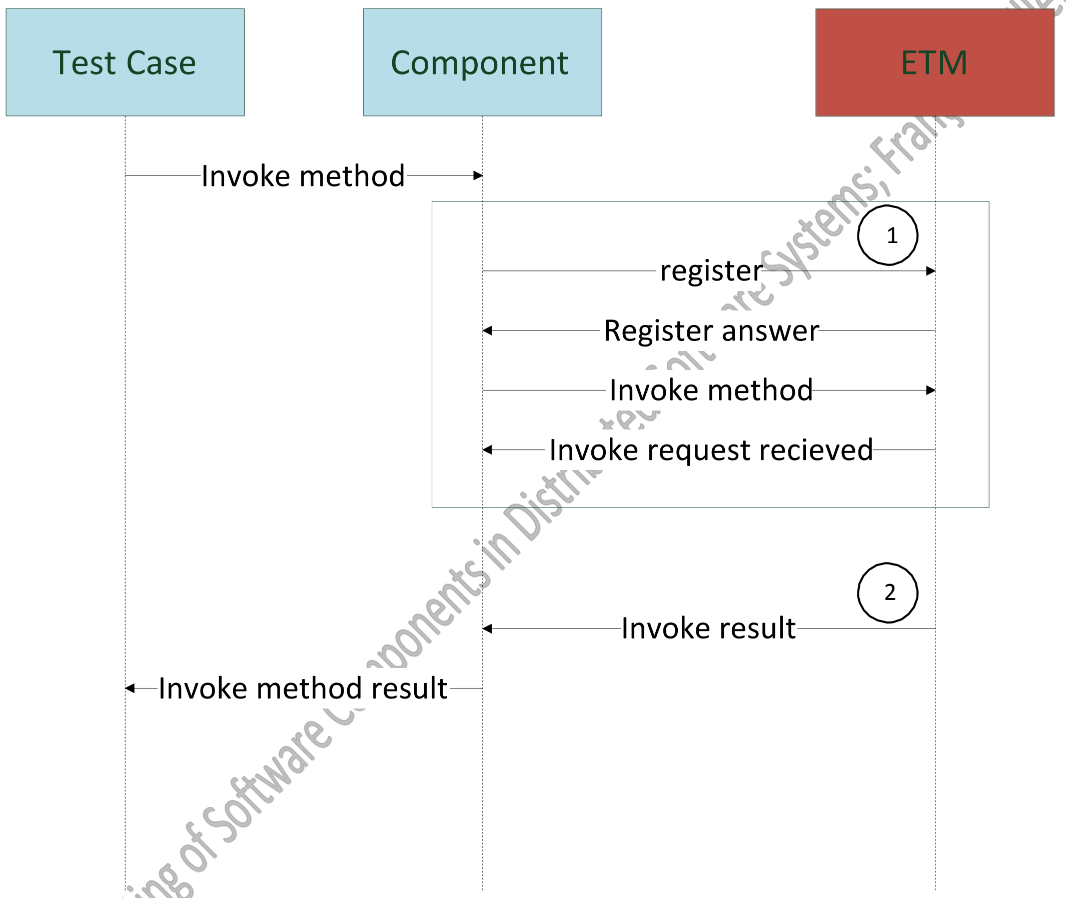


Figure 6.6: Trigger method example

The Test case invokes a method on the component, which does first a registration followed by the method call, and waits for the method call result (Figure 6.6, 1). Until this point, the behaviour is the same as described in the previous section, i.e. for each received message, an answer can be sent. However, components can block until a specific messages from the ETM has been received (Figure 6.6, 2). In other words, the ETM need an interface that offers the possibility to send messages to a component.

The ETM stores all the open socket and so just need to know on which one the message should be send. Like presented in the section 6.3 the interaction model consists these information, so the method structure is defined like in Figure 6.7.

```
void TriggerMessage(InteractionMessage messagetoSend);
```

Figure 6.7: Trigger message syntax

Socket Handling

Section 3.1 presented that components are communicating over the network connection with each other. Most programming languages are offering sockets, which are handling the sending and receiving of package i.e. the programming languages are handling all the layers up to the transport layer (Figure 3.2). The socket gets configured by an Endpoint that consists of an IP-address, a port, and the used transport layer like for example TCP.

The components can communicate over different ports with each other. These make it challenging to simulate a component. Besides, the connection of the sockets has to be open until the component closes the port. An example is shown in the following picture.

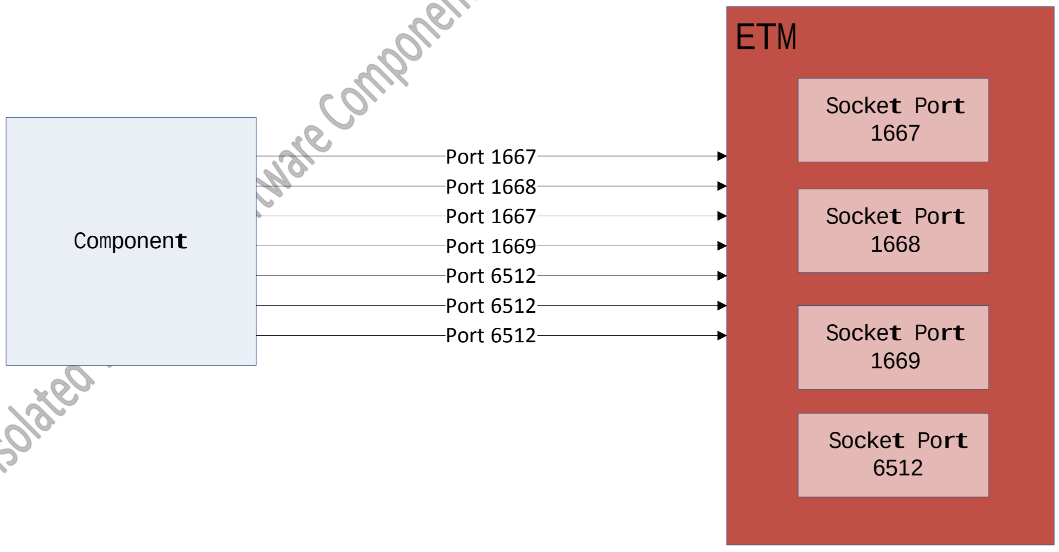


Figure 6.8: Communication between several components

The component communicates over several ports with the ETM. The ETM opens for each port an own thread that handles the communication with the component on that specific port (Figure 6.8). Because of the thread behaviour the list (Section 6.2) that has been presented in the previous section has to be thread saves. Every thread itself searches for the corresponding message an answer.

6.3 Interaction Model

The behaviour of the ETM has to be defined in order to react in a correct way. This behaviour is defined as interaction models, which consists of all the information. This helps that the ETM can send the correct responses to a request. An interaction model has the code structure as shown in Figure 6.9.

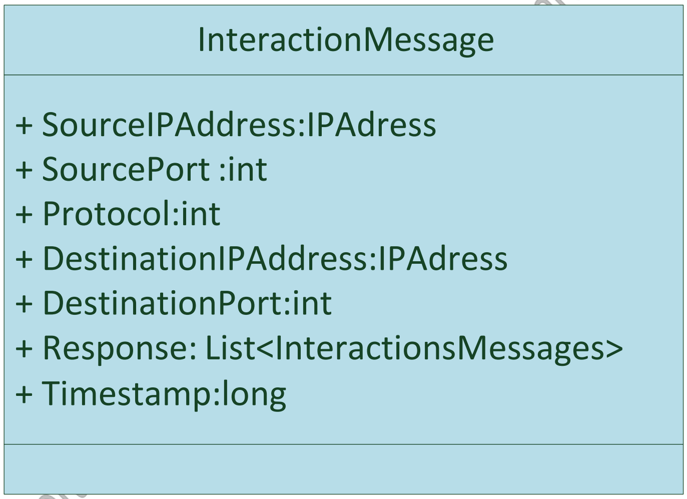

InteractionMessage
+ SourceIPAddress:IPAdress
+ SourcePort :int
+ Protocol:int
+ DestinationIPAddress:IPAdress
+ DestinationPort:int
+ Response: List<InteractionsMessages>
+ Timestamp:long

Figure 6.9: Interaction model structure

The source IP address and the source port define the endpoint on which the ETM should listens. The protocol contains the definition for the data, which the component under test sends. The destination IP-address and the destination port define the endpoint on which the ETM should send the response. The response contains a list of interaction messages. This interaction messages could have a different endpoints then the components under test. The timestamp indicates the time at which the interaction message has been created.

6.4 Application Protocol

The implementation of the application protocol allows it to convert bytes into the desired protocol. The desired protocol can be analysed and a search, with criteria can be performed. Furthermore, the protocol offers the possibility to serialized messages to bytes and sends these to the component. Every protocol has some characteristics, which makes it possible to check the validity. The validity is used to check if the protocol has all the necessary bytes or if addition bytes are required. Every protocol has to implement the interface that is shown in Figure 6.10.

```
public interface IProtocol : ICloneable, IComparable<IProtocol>
{
    int SocketNumber { get; set; }
    IProtocol ConvertToProtocol(Byte[] message);
    void RetrieveInfoFromReceivedMessage(IProtocol receivedMessage);
    Byte[] Valid();
    Boolean GetMoreBytes();
}
```

Figure 6.10: Protocol interface

The socket number defines, on which socket the protocol should be valid. In other words, it allows grouping the messages to sockets and just using interaction messages for a specific socket. The ETM defines a socket with an increasing number. A socket is bound to a number, when it gets open. For example, the ports from the Figure 6.8 have the following numbers:

- 1:Socket on port 1667
- 2:Socket on port 1668
- 3: Socket on port 1669
- 4: Socket on port 6512

When a socket number is defined with -1 then this protocol is valid for all sockets. The ConvertToProtocol method (with the byte array as input) has main purpose of converting received bytes from the socket to the protocol. With the help of this method, it is possible to filter information from the received bytes. The GetMessage method converts the protocol message to a byte array, which is send to a specific component. Last but not least the valid method is used to ask if the protocol is valid. The main porpoise of this method is to check if the protocol, which has been created from the received bytes, is valid or if more bytes are required.

The clone interface ICloneable[15] is used to offer the possibility to clone a protocol, which guaranties that the variables of the origin cannot be modified from a different reference. The comparable interface ICompairable[16] provides the possibility to compare two objects.

The next section will show concrete implementation of the presented interface. This are the TCP protocol that is basically just handles bytes, the SOAP protocol that is used by Web services and the ActiveMQ protocol that is used to simulate the presented use case.

TCP protocol

The TCP protocol is a byte array that symbolizes the message. The implementation of the interface protocol is shown in Figure 6.11. When the ETM receives bytes, it forwards it to the protocol, which directly stores it. There is no need to check the validity because the TCP data are always valid. Furthermore, when the ETM wants to get the byte data from the protocol then only the message itself will be returned because the message is in byte format and does not need any further conversion.

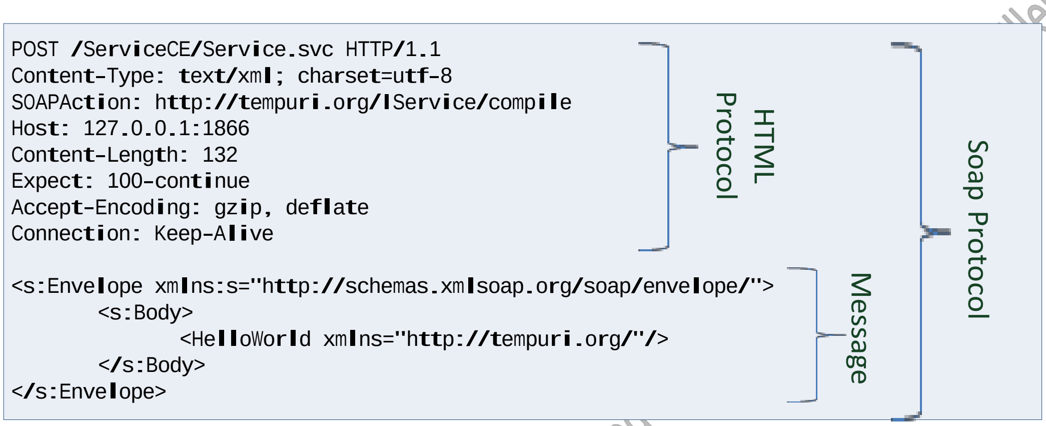
```
public class TCPProtocol : IProtocol{
    private byte[] message = null;
    public int SocketNumber { get; set; }

    public TCPProtocol(byte[] message, int SocketID){
        this.message = message;
        this.SocketNumber = SocketID;
    }
    public IProtocol ConvertToProtocol(byte[] message){
        return new TCPProtocol(message, -1);
    }
    public byte[] GetMessage() {
        return message;
    }
    public int CompareTo(IProtocol protocol){
        if (CompaireByteArraysWithoutLength(protocol.GetMessage(), message))
            return 1;
        return 0;
    }
    public bool Valid() {
        return true;
    }
    public void RetrieveInfoFromReceivedMessage(IProtocol receivedMessage){}
    public object Clone() {
        return new TCPProtocol(message, -1);
    }
}
```

Figure 6.11: TCP protocol implementation

[15] http://msdn.microsoft.com/en-us/library/system.icloneable.aspx
[16] http://msdn.microsoft.com/en-us/library/system.icomparable.aspx

SOAP protocol

A SOAP protocol consists of two parts, an HTML and XML part, which is shown in Figure 6.12.



```
POST /ServiceCE/Service.svc HTTP/1.1
Content-Type: text/xml; charset=utf-8
SOAPAction: http://tempuri.org/IService/compile
Host: 127.0.0.1:1866
Content-Length: 132
Expect: 100-continue
Accept-Encoding: gzip, deflate
Connection: Keep-Alive

<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
       <s:Body>
              <HelloWorld xmlns="http://tempuri.org/"/>
       </s:Body>
</s:Envelope>
```

Figure 6.12: General structure for a SOAP message.

A SOAP message includes a HTML part that consists of information, as for example the complete length of the send message. Next the original message is encoded in a SOAP envelope that contains a Header and body tag. In the body a fault part can be present, which symbolise an exception type. Further information to SOAP can be retrieved in the indicated book [46].

When the ETM receives bytes, these are forwarded to the SOAP protocol implementation. Next they are converted into a String and the corresponding parts as for example the header, are stored in the corresponding variables (Figure 6.13, 1). To control the validity, all the required parts should be present (Figure 6.13, 2).

An answer to a request is produced by adding the HTML Protocol to the envelope. This String is then converted to a byte array and forwarded to the client (Figure 6.13, 3).

```
public class SoapEnvelope : IProtocol
{
    private String[] htmlInfo;
    public int SocketNumber { get; set; }
    private SoapHeader header;
    private SoapBody body;
1   public SoapEnvelope(String message):this(message,-1) {}
    public SoapEnvelope(String message, int SocketNumber) {
        this.SocketNumber = SocketNumber;
        if (!message.Contains("<")) htmlInfo = filterHTMLInfo(message);
        String result = filterSOAPEnvelobe(message);
        header = new SoapHeader(splittext("header", result));
        Body = new SoapBody(splittext("body", result));
    }
    public override String ToString() {
        List<Object> tmp = new List<Object>(){ Header, body }
        return htmlresult + ConvertToString(tmp);
    }
    public IProtocol ConvertToProtocol(byte[] message){
        ASCIIEncoding asci = new ASCIIEncoding();
        return new SoapEnvelope(asci.GetString(message),-1);
    }
    public int CompaireTo(IProtocol protocol){
        Boolean result = true;
        SoapEnvelope envelopr = (SoapEnvelope)protocol;
        if (compaireHtml(envelopeHtml,htmlInfo) && body.Compaire(envelopr.body))
           return 1;
        return 0;
    }
    public byte[] GetMessage(){
3       ASCIIEncoding encoder = new ASCIIEncoding();
        return encoder.GetBytes(this.ToString());
    }
    public bool Valid(){
2       return htmlInfo != null && header!=null && body != null
    }
    public void RetrieveInfoFromReceivedMessage(IProtocol receivedMessage){
        return;
    }
    public object Clone(){
        return new SoapEnvelope(this.ToString(),-1);
    }
}
```

Figure 6.13: SOAP Protocol implementation

ActiveMQ Protocol

Like presented in the previous chapter 4.2, the ActiveMQ protocol is used from the Engsb and the .Net Bridge to communicate with each other. The ActiveMQ protocol is based on commands, which are serialized to bytes, forwarded to the transport layer, on the other side deserialized, and the information is retrieved. Like presented in section 6.1, ActiveMQ uses the OpenWire format to communicate with each other. The ETM needs a protocol implementation to be able to communicate with a component that uses ActiveMQ. The Figure 6.14 illustrates the ActiveMQ protocol implementation. ActiveMQ provides a class, OpenWireFormat that offers the possibility to serialize and deserialze objects to/from a byte array (Figure 6.14, 1). These values have to be same on the client and the server side because the formation have a specific variable, which are explained in detail in the indicated reference[45]. When a Wireformat is transmitted then all the ActiveMQ protocols should adapt to this format. It implies that the formation has to be static (Figure 6.14, 1).

The conversion from byte to an object works only over the OpenWireFormat (Figure 6.14, 1). In this case the Memory stream is an Endian binary reader, provided by ActiveMQ that stores the byte in endian format (Figure 6.14, 2).

Like introduced in the section 6.1, every Response needs a corresponding CommandId and every MessageDispatcher needs a ConsumerId and a Destination. This information can be retrieved from a received protocol. In the method RetieveInfoFromReceivedMessage, the command type is recognized and the information is added to the Message (Figure 6.14, 3).

```
public class ActiveMQProtocol : IProtocol {
    public Command Message { get; set; }
    public int SocketNumber { get; set; }
1   public static OpenWireFormat format;

    public ActiveMQProtocol(Object obj, int socketNumber) {
        this.SocketNumber = socketNumber;
        this.Message = (Command)obj;
        if (obj is WireFormatInfo) { format.RenegotiateWireFormat((WireFormatInfo)obj); }
    }
    public IProtocol ConvertToProtocol(Object message) {
        if (message is String) { Message = new ActiveMQTextMessage((String)message); }
        else { Message = (Command)message; }
        return new ActiveMQProtocol(Message, -1);
    }
    public IProtocol ConvertToProtocol(byte[] Message){
        try {
2           return new ActiveMQProtocol(format.Unmarshal(
                 new EndianBinaryReader(new MemoryStream((byte[])Message.Clone()))), -1);
        }
        catch { return null; }
    }
    public int CompareTo(IProtocol protocol) {
         if (protocol is ActiveMQProtocol) {
            ActiveMQProtocol pr = (ActiveMQProtocol)protocol;
            if (pr.Message.GetType().IsInstanceOfType(Message) &&
              Message.GetType().IsInstanceOfType(pr.Message)) { return 1; }
        }
        return 0;
    }
    public byte[] GetMessage() {
       byte[] buffer = new byte[8192];
       MemoryStream mem = new MemoryStream(buffer);
       format.Marshal(Message, new EndianBinaryWriter(mem));
       Byte[] result = new Byte[mem.Position];
       Array.Copy(buffer, result, mem.Position);
       return result;
    }
    public bool Valid() {
        return Message != null;
    }
    public void RetrieveInfoFromReceivedMessage(IProtocol receivedMessage) {
        if (receivedMessage is ActiveMQProtocol){
          if (Message is Response) { ((Response)Message).CorrelationId
               =
               ((ActiveMQProtocol)receivedMessage).Message.CommandId;
3         }
          if (Message is MessageDispatch){
             MessageDispatch dispatcher = Message as MessageDispatch;
             ConsumerInfo consumerinfo = ((ActiveMQProtocol)receivedMessage).Message as ConsumerInfo;
             if (dispatcher.ConsumerId == null) { dispatcher.ConsumerId = consumerinfo.ConsumerId; }
             if (dispatcher.Destination == null){ dispatcher.Destination = consumerinfo.Destination;}
             Message = dispatcher;
          }
        }
    }
    public object Clone() {
        return new ActiveMQProtocol(Message, SocketNumber);
    }
  }
```

Figure 6.14: ActiveMQ protocol implementation

6.5 Examples of test cases

In the following section, a test case example for TCP, SOAP and ActiveMQ will be presented. The TCP example is sending bytes from one socket to the other one. The SOAP example is based on the explanation on the previous section (SOAP protocol). The ActiveMQ example is based on the presented use case, on the Figure 4.4 and simulates the ActiveMQ message handling as presented in Figure 6.2.

TCP test case example

The test case scenario consists of three main parts. First the configuration of the ETM will be created, which simulates the TCP component. In the next line, the ETM is started with the specified configuration followed by starting the TCP component. The received bytes are than compared with the expected result. All these parts are shown in Figure 6.15.

```
[TestMethod]
public void TCPConnectionTest()
{
  asci = new ASCIIEncoding();
  CreateTCPWorkflowReaction();
  ETM.start();
  StartTCPComponentForTestingTCPConnection();
  byte[] response=asci.GetBytes("Hello World too"))
  Assert.AreEqual(recievedmessage.Length, response.Length);
  Assert.AreEqual(recievedmessage, respone);
  }
}
```

Figure 6.15: Test case example with TCP

In the CreateTCPWorkflowReaction method, the interaction messages are defined. This consist of the message that the ETM receives (*"Hello World"*) and the response itself (*"Hello World too"*) on the correspond endpoints (*localhost* and a *random port*). A more detailed view is shown in Figure 6.16.

```
private void CreateTCPWorkflowReaction()
{
  List<InteractionMessage> receiveMessage = new
  List<InteractionMessage>() { new InteractionMessage(localSocketPort,
        remoteSocketPort, new TCPProtocol(asci.GetBytes("Hello World"),-1), null,
              IPAddress.Loopback, IPAddress.Loopback) };

  InteractionMessage SendMessage = new InteractionMessage(remoteSocketPort,
        localSocketPort, new TCPProtocol(asci.GetBytes("Hello World too");,-1),
              receiveMessage, IPAddress.Loopback, IPAddress.Loopback);
  ETM.Add(SendMessage);
}
```

Figure 6.16: TCP example configuration

The TCP component is a socket, which opens a TCP connection and sends the message "Hello World". The response is stored in the variable received message, followed by closing the socket again. This behaviour is shown in Figure 6.17.

```
private void StartTCPComponentForTestingTCPConnection()
{
  socket = new Socket(AddressFamily.InterNetwork, SocketType.Stream,
       ProtocolType.Tcp);
  socket.Bind(new IPEndPoint(IPAddress.Loopback, remoteSocketPort));
  socket.Connect(new IPEndPoint(IPAddress.Loopback, localSocketPort));
  ASCIIEncoding encoder = new ASCIIEncoding();
  socket.Send(asci.GetBytes("Hello World"));
  receiveMessage = socket.Receive(message, SocketFlags.None);
  socket.Close()
}
```

Figure 6.17: Start the TCP component

SOAP test case example

The test case scenario consists of three different parts. These three parts are shown in the code snippet in Figure 6.18. First the configuration of the ETM will be created, which simulates the

Web service. In the next line, the Web service client is initialised while in the third line the method getData of the Web service is invoked. The result of this method call is compared with 412571.

```
[TestMethod]
public void TestSoapConnection(){
       ConfigureStartETMWithSOAPProtocol();
       IService test = new ServiceClient();
       Assert.AreEqual("412571", test.getData());
}
```

Figure 6.18: Test case example with SOAP

```
1. private void ConfigureStartETMWithSOAPProtocol()
2. {
3.     String request = GetRequestTestCase1();
4.     String reply = GetReplyTestCase1();
5.
6.     IProtocol requestMessage = new SOAPProtocol(request);
7.     IProtocol answerMessage = new SOAPProtocol(reply);
8.
9.     TranscationMessage replyTransaction = new TranscationMessage(1866, 0,
10.        answerMessage, null, IPAddress.Loopback, IPAddress.Loopback);
11.    TranscationMessage requestTransaction = new TranscationMessage(51446,1866,
12.        requestMessage, replyTransaction, IPAddress.Loopback, IPAddress.Loopback);
13.
14.    ETM = new ETMTCP(new List<TranscationMessage>() { requestTransaction });
15.    ETM.Start();
16. }
```

Figure 6.19: ETM configuration with SOAP

The ConfigureStartETMWithSOAPProtocol method configures and starts the ETM and is described in the code snippet in Figure 6.19. First, the request and reply method has to be defined. These two messages define the interaction between the Web service and the ETM. Furthermore, these two messages are used to generate the SOAP messages. This is done by forwarding the request and reply to the specified protocol that is in this case the SOAP protocol (Figure 6.19, 2&3). Next the answer respectively the reply has to be embedded into an interaction model, which represents the configuration for the ETM (Figure 6.19, 9-12). The interaction model has the properties as presented in the section 6.3. In this concrete case, the IP address is for both, the answer and reply, local host and the ports are defined by the web service configuration that is in this case 1866 (ETM listens) and 51446 (ETM sends the answer). The request and reply messages generated by the two methods look as in Figure 6.20 and Figure 6.21 presented. The GetRequestTestCase1 method returns the message, which will be invoked by the Web service client. With IGNOREFIELD the SOAP protocol can ignored, which means that only the Envelope will be analysed. The envelope part of the message stands for the method call or method call result. In the GetReplyTestCase1 method the reply message is generated which defines the response message and should be transmitted to the client (Figure 6.21). To complete the answer, the HTML header information is added to the envelope part and the complete message has the structure as shown in Figure 6.12.

```
IGNOREFIELD

<s:Envelope xmlns:s=\"http://schemas.xmlsoap.org/soap/envelope/\">
      <s:Body>
            <getData xmlns=\"http://tempuri.org/\"/>
      </s:Body>
</s:Envelope>
```

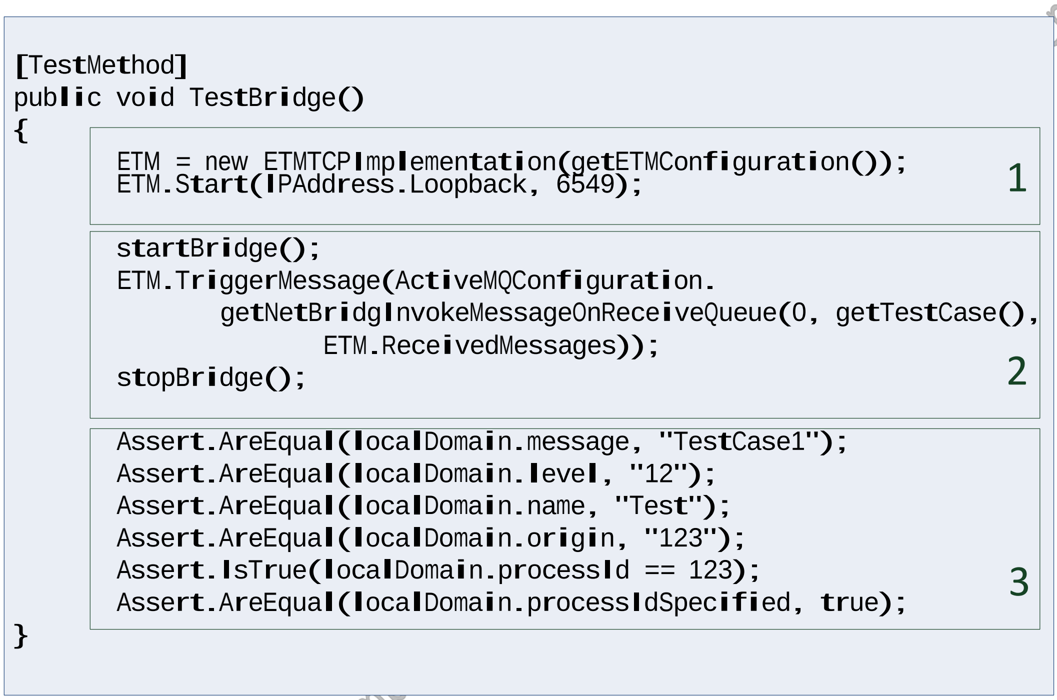
Figure 6.20: SOAP request message

```
<s:Envelope xmlns:s=\"http://schemas.xmlsoap.org/soap/envelope/\">
      <s:Body>
            <getDataResponse xmlns=\"http://tempuri.org/\">
                  <getDataResult>412571</getDataResult>
            </getDataResponse>
      </s:Body>
</s:Envelope>
```

Figure 6.21: SOAP response message

ActiveMQ test case example

The test case architecture is equivalent to the previous example and has tree part. The code snippet is presented in Figure 6.22. In the first step (Figure 6.22, 1) the ETM gets configured and started. The configuration creates the interaction models that are used to interact with the ActiveMQ client. In this case the client is the .Net Bridge (4.1 .Net Bridge). The configuration models are presented in Figure 6.25, which can be explained with the Figure 6.23. Every red box stands for a communication that is done by ActiveMQ. Furthermore, the red boxes are groupings, like for example producer is an abstraction of the Figure 6.2. Like shown in Figure 6.2 a consumer and producer is created several times. This means that the commands are the same and can be used in for all these sockets. In other words, the commands can be configured with the socket number -1 (6.4).



```
[TestMethod]
public void TestBridge()
{
        ETM = new ETMTCPImplementation(getETMConfiguration());
        ETM.Start(IPAddress.Loopback, 6549);

        startBridge();
        ETM.TriggerMessage(ActiveMQConfiguration.
               getNetBridgInvokeMessageOnReceiveQueue(0, getTestCase(),
                      ETM.ReceivedMessages));
        stopBridge();

        Assert.AreEqual(localDomain.message, "TestCase1");
        Assert.AreEqual(localDomain.level, "12");
        Assert.AreEqual(localDomain.name, "Test");
        Assert.AreEqual(localDomain.origin, "123");
        Assert.IsTrue(localDomain.processId == 123);
        Assert.AreEqual(localDomain.processIdSpecified, true);
}
```

Figure 6.22. Test case example with ActiveMQ

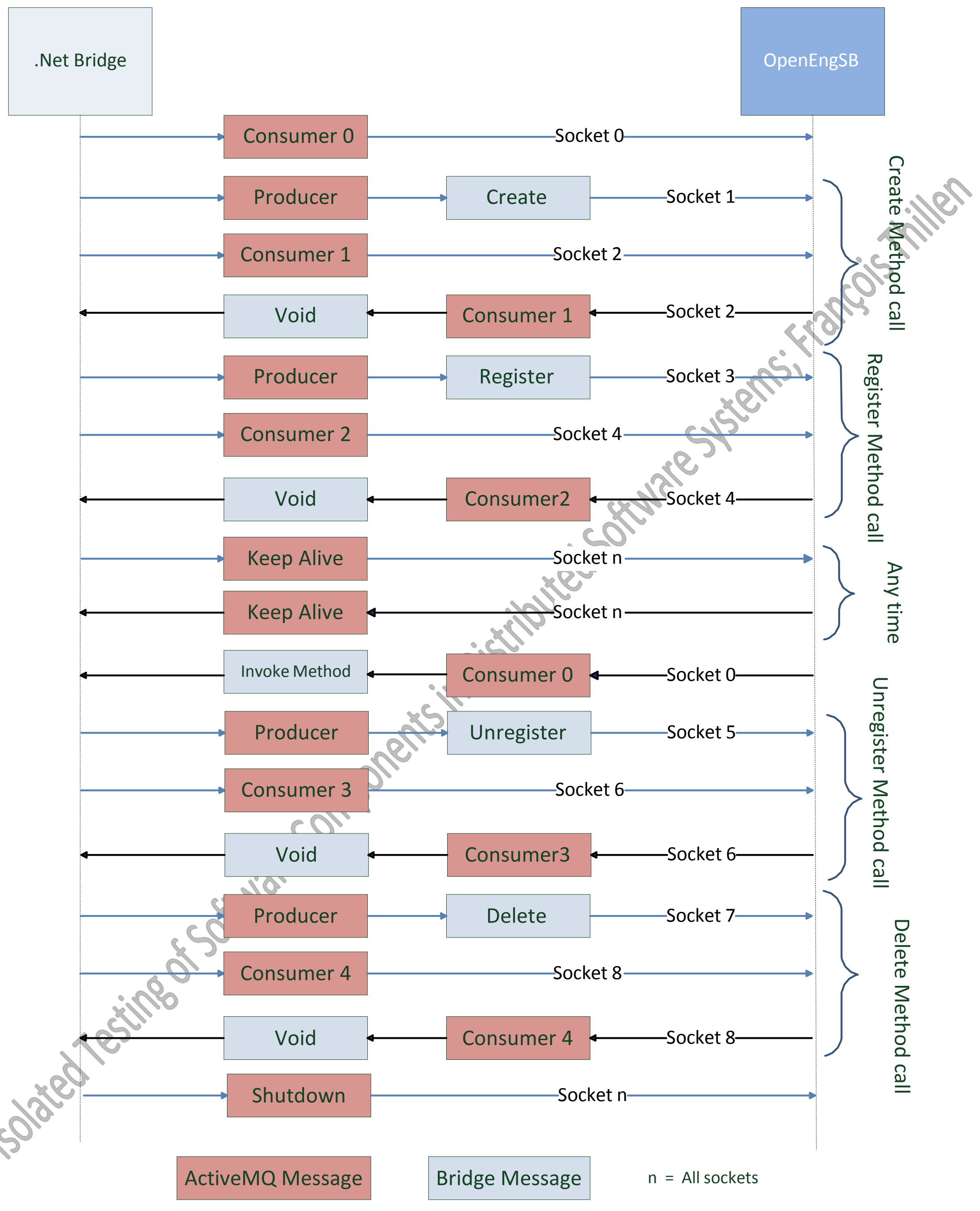


Figure 6.23: ActiveMQ and .Net Bridge communication

```
[TestMethod]
 private List<InteractionMessage> getETMConfiguration(){
      List<InteractionMessage> result = ActiveMQConfiguration.getConfiguration();
      result.Add(ActiveMQConfiguration.getSendToConsumerVoidMessage(2, getBrideVoidAnswer()));
      result.Add(ActiveMQConfiguration.getSendToConsumerVoidMessage(4, getBrideVoidAnswer()));
      result.Add(ActiveMQConfiguration.getSendToConsumerVoidMessage(6, getBrideVoidAnswer()));
      result.Add(ActiveMQConfiguration.getSendToConsumerVoidMessage(7, getBrideVoidAnswer()));
      result.Add(ActiveMQConfiguration.getSendToConsumerVoidMessage(9, getBrideVoidAnswer()));
      return result;
}
```

Figure 6.24: ETM configuration method

```
public static List<InteractionMessage> getConfiguration(WireFormatInfo wire)
{
     List<InteractionMessage> result = new List<InteractionMessage>();
     result.Add(ActiveMQConfiguration.getRemoveInfoAnswer(-1));
     result.Add(ActiveMQConfiguration.getShutdownInfoAnswer(-1));
     result.Add(ActiveMQConfiguration.getKeepAliveAnswer(-1));
     result.Add(ActiveMQConfiguration.getWireFormatAnswer(wire, -1));
     result.Add(ActiveMQConfiguration.getNetBridgeTextMessageAnswer(-1));
     result.Add(ActiveMQConfiguration.getAskedAnswer(-1));
     result.Add(ActiveMQConfiguration.getSessionInfoAnswer(-1));
     result.Add(ActiveMQConfiguration.getProducerInfoAnswer(-1));
     result.Add(ActiveMQConfiguration.getConnectionInfoAnswer(-1));
     result.Add(ActiveMQConfiguration.getConsumerInfoAnswer(-1));
     return result;
}
```

Figure 6.25: ActiveMQ configuration

In the next step of the configuration, the answers for the .Net Bridge have to be configured (Figure 6.24), which are shown as blue boxes in Figure 6.23.

The communication messages from the .Net Bridge is shown in abstract way in Figure 4.2 and with the ActiveMQ communication in Figure 6.23. The first message from the .Net Bridge is send on the Socket 2, which is a create message. According to the definition of a connector (4.1), a void message is needed, i.e. the ETM has to send that void message back. This is done by wrapping the void message in an ActiveMQTextMessage command, which again is wrapped in a MessageDispatcher (6.4 ActiveMQ). This behaviour is the same for the register, unregister, and delete message.

After the configuration, the .Net Bridge has to be started (Figure 6.23, 2) that is done by invoking the creation and registration method of the .Net Bridge. When the registration was successfully, some method calls on the bridge can be triggered to test the correct behaviour. This is done by forwarding the ETM a configuration, which finds the correct socket and forwards the request to the .Net Bridge. This request has to be send to the receive queue that is from the definition of the .Net Bridge always on socket 0. To be able to send a valid request to the correct consumer (receive queue), the ConsumerId and Destination of the receive queue is required. It follows that we have to search in the received messages for this specific message. This behaviour is done in the corresponding method (getNetBridgInvokeMessageOnReceiveQueue).

Next we have to close the .Net Bridge in a correct way that implies to send unregister and delete the .Net Bridge. The behaviour of the ETM is already defined in the previous configuration (Figure 6.24).

In the last step (Figure 6.23, 3) the correct behaviour is tested with normal unit tests. The method call, which has been triggered from the ETM, some variables had been set. These values get test on their correctness with the normal test strategies.

CHAPTER 7

Evaluation

The advantage of the ETM is that the test cases can be created at any time and are not dependent to a finished implementation of a component. Furthermore, with the ETM specified components can be test and so not the start up from the complete system is required. This means that the test cases with the ETM have to be created once and then be executed at any time without any requirements to the system.

7.1 Effectiveness

The effectiveness is measured, by the different kinds of bugs the ETM can help to find. The ETM has the advantage that it can generate user defined messages, which are inconsistent to the definition of the component. It follows that testing components with fault messages is possible and offers the possibility to test components in an adequate way.

The ETM does not execute test cases itself but only simulates other components. It follows that the effectiveness is dependent on the created test cases that has the same effectiveness as component tests. An indirect feature of the ETM is that it can simulate components, which does not exist yet. It the views of the Software development process, developer are independent and can work independently to the system, which improve the effectiveness of the complete team.

7.2 Costs of effort

In the following section the efforts are presented to implement the test cases. The measurement is illustrated by the time and the required Lines of Code (LOC) needed to implement the test cases. Three different protocol types are compared which illustrate low effort costs of the ETM.

TCP protocol:

The implementation of the TCP protocol contains 68 Lines of code (LOC) and needed 10 min implementation time. The complete test case consists of three parts. The interaction model for the ETM that has three lines of code, starting the TCP component requires 14 LOC, starting/configuring the TCP component has 53 LOC, and the correctness verification has 4 LOC. The implementation for the complete test case took 30 min and has 71 LOC. Protocol and test cases together have 179 lines of code, which needed approx. 40 minutes implementation time (section 6.5).

SOAP protocol:

The SOAP protocol is used to offer the possibility to simulate Web services. The Soap protocol had to be implemented from the scratch and so have more line of code as the other protocols. By using existing libraries (like commercial), the implementation and efforts can be reduced. However, this protocol implementation needed about 368 LOC and needed around one hour. The SOAP protocol does not require additional simulation of the behavior (i.e. like for example in ActiveMQ open consumer or producer (Figure 6.23)). The test case itself is very short, which is mainly because the component is started in one line. The complete test case has 64 LOC and 20 min of implementation time (section 6.5).

ActiveMQ protocol:

ActiveMQ is a complex application protocol because it interacts over several sockets and has an own behaviour, which has to be implemented as well. The implementation of the protocol is simple because the ActiveMQ libraries can be reused. The protocol code has 82 line of code and took twenty minutes to implement. However, the behaviour of ActiveMQ required more time to implement. The behaviour consists of 151 LOC and needed about 40 minutes to be implemented. The test case for the Use case contains three parts. The first part contains the .Net Bridge specific interaction message. The second part starts and stops the .Net Bridge, i.e. creating, registering, unregister and delete the connector. Furthermore, a method is triggered to have an expressive test case. The last part contains the verification of the test execution. Altogether, the test case needs 20 LOC and around 15 minutes to implement (section 6.5).

Protocol	Total		Protocol implementation		Simulate Behaviour		Test case	
	Time (min)	LOC	Time (min)	LOC	Time (min)	LOC	Time (min)	LOC
TCP	40	179	10	68	0	0	30	71
SOAP	80	512	60	368	0	0	20	64
ActiveMQ	75	308	20	82	40	151	15	20

Figure 7.1: Implementation time

Summary

The implementation and creation of the protocol and/or the behaviour of the component, costs a lot of time. Like Figure 7.1 shows, this is just a onetime effort. The ActiveMQ implementation for example needs in total 75 min of implementing the protocol, behaviour and the test case. In contrast, the test case just needs only 15 min and 20 LOC. It follows that all other test cases can use the configuration and the protocol. This improves the effectiveness extremely and it follows that implementing further test cases are very easy. However, the developer of the protocol has to understand the behaviour of the protocol. This requires also time that depends on the complexity of the used application protocol. In the case of SOAP, the developer does not need that much time to understand the protocol. In contrast, the ActiveMQ requires more time because of the more complex handling of the messages. This learning time is not included in the Figure 7.1.

7.3 Performance

The ETM is simulating a component and should be as fast as the component itself. Otherwise, the test case could fail and no adequate result can be found. The main point of the system is, to find the correct answer in the list (Section 6.2). The runtime complexity, for finding an element in the list is indicated in the following; ***x*** stands for the number of used protocol types and ***n*** for the number of interaction message.

- Worst case: $\mathrm{O}(x \bullet n) = O(n \bullet n) = O(n^2)$
- Average case & Best case: $\mathrm{O}(x \bullet n) = O(1 \bullet n) = O(n)$

Normally, a test scenario uses just one protocol type for a component. It follows, that there is one protocol type to that the received bytes have to be converted. This implies that x is a constant and so the runtime is linear. In the very unlikely case, where every protocol from the interaction messages is of a different type as the other, then the runtime complexity is squared.

The complete runtime complexity of receiving a message to sending a response depends on the runtime of finding the corresponding interaction message in the list. The other parts are all linear in there complexity. It follows that the runtime complexity is $O(n^2)$ in the worst case and $O(n)$ in the average case.

Start-up time

The ETM needs for every connecting socket time to find the correct behaviour but does it has no start-up and clean-up time, which implies that the ETM is directly present and can interact with the component. The traditional approach needs start-up and clean-up time before the test cases can be executed. In contrast the traditional approach is faster in answering a single message because no search for a corresponding answer has to be searched. The interaction times are expressed Figure 7.2

	Without startup (ms)	With startup and cleanup (ms)
ETM approach	7017	7051
Tradition approach	1516	45613

Figure 7.2: Interaction Time, ETM and tradition approach

7.4 Summary

The implementation of the application protocol is very fast and is mostly has already been implemented by other software developer. Furthermore, the application protocol has to be created only once and all other test cases can use that application protocol. In other words, the first test case needs time while following test cases can be implemented faster.

The ETM's limitation is implementing interaction messages. Interaction messages require knowledge about the dependent component. Furthermore, interaction messages can have high complexity, which increases the creation and complexity of the interaction message. Nevertheless, interaction messages have to be implemented once and can be reused or a template can be created, which again reduces complexity and time.

A further advantage of the ETM is that it increases the test coverage, since it can generate fault messages, which can cover up hidden errors. The ETM can also simulate components, which are not implemented yet, which leads to reduce of the development time of the complete process because components can be implemented independently to the system.

An additional feature of the ETM is that already existing tests with other testing concepts does not have to be adapted to the ETM. Only the protocol and the behaviour have to be provided to the ETM.

CHAPTER 8

Discussion

In this section a discussion about the advantages and limitations of the ETM compared to existing concepts and defined research issues in the context of the gained evaluation results is given. The existing concepts are applied to the use case (Chapter 4) and compared with the ETM. The existing concepts are all using the network connection to send test dependent message to components The ETM approach is language independent because it is listening on the transport layer. The transport layer is standardised and is handled by almost every programming language. Furthermore every operation system supports this layer, which implies that components can be tested in different environments, operation systems and languages. Like introduced in the related work, testing is a part of the software engineering process. For every development state a test scenario has to be created. The ETM offers a concrete implantation to full fill the testing desire.

Like presented in the related work, already concepts and approaches exists to test component based system. Every concept has been developed for specific needs and will be compared with the ETM on their effectiveness, efficiency and performance.

Mock-up frameworks are simulating components by implementing interfaces and the ETM in contrast is working on the network connection and does not do any changes on the components. Mock-up frameworks allow it to perform tests with the normal test strategies (for example unit tests). This approach is very applicable to every component until the source code is open. In the other case it is not possible to use mock-up frameworks because the inner structure has to be known (like white box testing). It is challenging, to use the mock-up framework with the presented approach because the ActiveMQ has to be mocked as well. Furthermore, every send and receive method has to be mocked in a way that every message is correctly represented. The test case for the ETM and the Mock-up are the same and both have to simulate the behavior of the component. It follows that the time to implement the behavior is very similar.

The approach of Bauer and Eschbach are using state-based components. However, the here presented approach is dealing with components, which are not state based, have dependencies to other components, and can have no clearly defined interface. The concept of the ETM is simulating the dependent components and allows testing a single component independent to the system, no matter if the component is state based or not.

The hybrid approach [25] uses black box testing to test the component. The concept of ETM facilitates testing of components, which design details does not have to be given. In the context of isolated testing of components the ETM approach is more efficient as details of dependencies (e.g., internal component behavior) do not need to be known. The hybrid approach does not take into account dependencies to other components. It follows that it is very challenging to use this approach with the presented use case.

The JRT framework, enables testing of remote server components. The concept is using the black box testing strategy and compares the received result with the expectations. The approach is applicable to server components but is not usable for the presented Use case. This is mainly because JRT is designed to test the dependent components. It creates messages that are forwarded to the dependent component and the result is analyzed. The component under test in contrast does not wait for message and send a response. It generates messages and waits for a response, which makes it challenging to use the JRT approach.

The Jata framework is a powerful framework to test components. It supports Remote Procedure Calls (RPC) and message-based communication. The main difference to the ETM is that ETM simulates the dependent component and Jata in contrast simulates the component that is dependent to other components. It is challenging for the Jata approach to simulate the Use case because the Jata should be used to test the OpenEngSB and not the .Net Bridge. Furthermore, in some use cases a message has to be triggered and send to the component because the component needs some data to full fill the work. In a nutshell, Jata is mainly used to test the here presented dependent components and the ETM can be used for all test cases.

From all these concepts, the traditional approach (i.e. starting all depending components) is the only one, which can be used for a comparison with the ETM.

Dependency models reflecting the system properties

The ETM uses an interaction message to describe the communication between the components. Implementing these models requires time. The traditional approach instead does not need any dependency models because the dependent components are present and fulfill their work when a request is detected. The advantage of the interaction models is that it allows it to send a test case specific message to the components under test. The disadvantage is that it requires time to implementing them. In contrast the tradition approach does not need any implementation time of a model but has no possibility to send test specific messages.

Effectiveness

Fault messages are messages, which deviate from the definition of the components respectively are invalid for the component. It follows that bugs can still be hidden, like presented in Figure 8.1.

```
public String Component1HidenError()
{
    String receivedMessage = receiveData();
    if (receivedMessage.Contains("MethodCall"))
    {
        return invokeMethod(receivedMessage);
    }
    else{
        return receivedMessage.Substring(0,3);
    }
}
```

Figure 8.1: Example hidden error

```
public String Component2Send()
{
    String messageToSend = getMethodName();
    String result;
    If (messageToSend != null){
        result = "MethodCall:" + messageToSend;
    }
    else{
        result = "Void"
    }
    sendMessage(result);
}
```

Figure 8.2: Send message from component 2

By analysing the workflow of component2Send it follows that a message with min length of four (Figure 8.2) is send (“*Void*” or “*MehtodCall:…*”). Component 1 receives these messages and parses it. The hidden bug is that when the received Message is smaller than four because the method *“Substring”* tries to copy the four letters from the message. When the message is smaller than four, the “*Substring*” method throws an exception. With the traditional approach (starting component2 and execute tests on componet1), the component always sends messages with the length greaten then four (from the workflow of component2 (Figure 8.2)). It follows that the bug could not be found. ETM and traditional approach

With the traditional approach, some test cases can be created at any time of the process but tests can only fail once all the required components are completely created, which leads to a very late time of error detection. The ETM is simulating the components and so can also simulate things, which aren't implemented yet.

Performance

From the view of performance, the tradition approach implies a start-up time. In the case where all the dependent components are present, these have to be started, configured and ready before the test case starts. In contrast, the ETM have to be started and next the test cases can be executed. Because the tradition approach does not need an implementation of the behaviour and the protocol, there is no search for a corresponding message (Section 6.2). This implies that the communication between the components is faster. However, the ETM needs some time for converting the message to the chosen protocol and searching the corresponding message. The execution time between the ETM and the traditional approach are shown in Figure 7.2. The ETM and the traditional approach are executed with the presented Use case (Chapter 4).

Efficiency

The start-up for the traditional approach is the following: Starting the OpenEngSB, provide the corresponding domain and execute the test case. This follows that the traditional approach needs a very long time to execute a single test, which are for the use case around 46 seconds. In contrast, the ETM needs for executing a test case 7 seconds, which is very fast compared to the start-up and execution of the tradition approach. A disadvantage is that the ETM needs still 7 seconds without a configuration. The traditional approach is five times faster than the ETM approach.

By including the time to implement the protocol and behaviour of a used protocol the ETM needs 93 executions of test cases (Figure 8.3) to be competitive to the traditional approach.

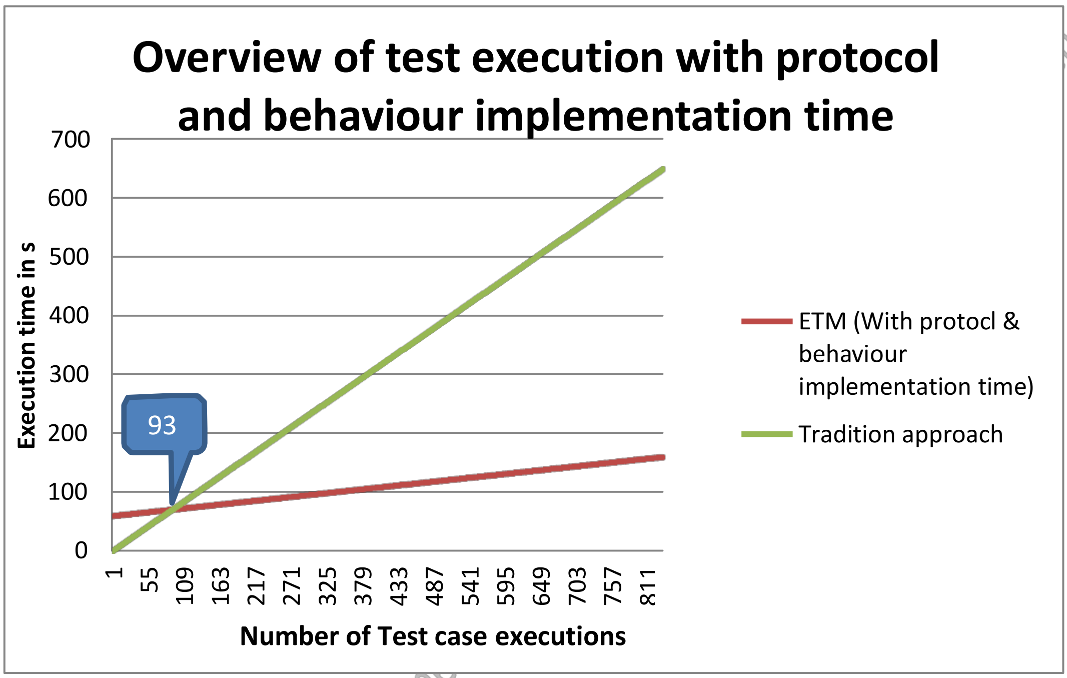


Figure 8.3: Execution time dependent to the Numbers of execution

Figure 8.3 is based on the execution time for the test cases for the presented use case. Generally, several test cases for a component are created and executed at the same time. This implies that the peak of 93 is very was reached and so the ETM justify the implementation of the protocol and the behaviour. The presented data are illustrated on the use case. In other use cases the answering time of a dependent component (traditional approach i.e. starting all the dependent components) time is maybe slower, it follows that the value for the peak is lower.

With the traditional approach, the components can’t be isolated tested, i.e. the dependencies have to be present and running witch implies that the dependent components have to be error free. This is not guaranteed and so defects can be in the component under test and in the dependent components. It follows that locating the source of error is challenging. The feature of the tradition approach is that the tests are always communicating with the newest versions of the dependent component. Furthermore, communication between the components exists and so no simulation has to be created. The generation of fault messages is challenging because the communication messages are defined and so the manipulation is very challenging. In contrast the ETM allows it to send user defined message as answer and so can generate fault messages.

The here presented approaches are focusing on the problem of testing distributed components. All of them have different working fields and helped to find errors. The ETM compared to the traditional approach is efficient, when several tests have to be executed several times. This should be the case in every software development process. It is challenging to compare the ETM with new testing concepts because they all have different requirements but does not provide the possibility to test the presented Use case. This is mainly because most of the approaches are testing component 3 from Figure 4.4 and not like the use case show component 2.

CHAPTER 9

Conclusion and Future work

Test driven development is a big part in the software engineering process. Test allows to measuring the software quality and to verify software. Modern Software Systems are based on software components, which intend to increase reusability, reduce defects in source code and save development costs. Distributed systems become more and more influence and so more systems are using remote services, which leads to dependencies between components. The original definition of a software component defines that components have to be independent to other components and have a clearly interface description. However, to form a running system, components have to interact with each other and therefore become dependent on other components. Several test strategies can be used when the components are on a single platform and does not make use of the network connection. However, remote communicating components have to send message over the network interface. These components are the so called dependent remote components. Several test concepts are based on the original definition of a component and thus make it challenging to be applied on dependent remote components. Mock-up frameworks offer the possibility to mock-up components by implementing the Interfaces with test case specific data but mock-up frameworks just are just testing a subset of the components. Furthermore, to create expressive test cases the code has to be open. According to these problems it is challenging to test dependent remote components isolated to the entire system.

In this work the "Effective Tester in the Middle" (ETM) was presented which describes a way to test dependent remote components with common test strategies. The ETM simulates the application protocol. Furthermore, the ETM allows it to forward test case dependent message to the component under test. This makes it possible to generate fault message, which checks the behaviours of a component under invalid messages. Furthermore, it isolate the components from the system and test there functionality. This gives very high test coverage of a single component, and on the total system. The advantages of the ETM are that the components does not have to be changed in any way and that components can be implemented independent to other component (for example, components are not implemented yet).

The ETM needs an application protocol implementation that allows it to simulate the connection. The implementation time is dependent on the complexity of the used application protocol. Altogether, it is recommended to use the ETM instead of starting the complete system because after several executions of the test cases the ETM more efficient. The evaluation is done by measuring the implementation time of test cases on a given use case scenario.

The presented prototype implementation of the ETM is capable of handling TCP, SOAP and ActiveMQ messages. For future work, the range of possible testing scenarios may be improved in the context of ActiveMQ. It can send messages over UDP, multicast and other transport layers. For further improvements of the ETM timeouts or a waiting time may be included. This will help to test, how components react if the messages are delivered with a delay or if they get lost.

Bibliography

[1] C. Alejandra, P. Mario, and V. Antonio, *Component-Based Software Quality: Methods and Techniques*. Springer, 2003, p. 403.

[2] C. Szyperski, “Component Software: Beyond Object-Oriented Programming,” in *Component Software: Beyond Object-Oriented Programming*, 2nd ed., Addison-Wesley Longman Publishing Co., Inc., 2002.

[3] H.-G. Gross, *Component-Based Software Testing with UML*. Springer-Verlag Berlin Heidelberg, 2005, p. 316.

[4] Y. Yao, “A framework for testing distributed software components,” *Electrical and Computer Engineering, 2005.*, no. May, pp. 1566–1569, 2005.

[5] M. Vieira and D. Richardson, “The role of dependencies in component-based systems evolution,” *Proceedings of the international workshop on Principles of software evolution - IWPSE ’02*, p. 62, 2002.

[6] M. Vieira and D. Richardson, “Analyzing Dependencies in Large Component-Based Systems,” *Automated Software Engineering, 2002. Proceedings. ASE 2002. 17th IEEE International Conference*, p. 241, Sep. 2002.

[7] A. Sharma, P. S. Grover, and R. Kumar, “Dependency analysis for component-based software systems,” *ACM SIGSOFT Software Engineering Notes*, vol. 34, no. 4, p. 1, Jul. 2009.

[8] D. Winkler, R. Hametner, T. Östreicher, and S. Bill, “A Framework for Automated Testing of Automation Systems,” *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference*, p. 4, 2010.

[9] E. Dustin, J. Rashka, and J. Paul, *Software Automatisch Testen: Verfahren, Handhabung Und Leistung*. Gabler Wissenschaftsverlage, 2001, p. 672.

[10] P. Hamill, D. Alexander, and S. Shasharina, "Web Service Validation Enabling Test-Driven Development of Service-Oriented Applications," *2009 Congress on Services - I*, pp. 467–470, Jul. 2009.

[11] M. T. Minella, *Pro Spring Batch*. Apress, 2011, p. 504.

[12] A. Schatten, M. Demolsky, D. Winkler, S. Biffl, E. Gostischa-Franta, and T. Östreicher, *Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen*. Spektrum Akademischer Verlag, 2010, p. 440.

[13] T. Grechenig, M. Bernhart, R. Breiteneder, and K. Kappel, *Softwaretechnik: Mit Fallbeispielen aus realen Entwicklungsprojekten*. Pearson Studium, 2009, p. 687.

[14] A.-P. Bröhl, *Das V-Modell: Der Standard in der Softwareentwicklung mit Praxisleitfaden*. Oldenbourg, 1993.

[15] B. Gloger, *Scrum: Produkte zuverlässig und schnell entwickeln*. Hanser Verlag, 2009, p. 318.

[16] S. Millett, *Pro Agile .NET Development with SCRUM*. 2011, p. 360.

[17] J. Link and P. Fröhlich, *Unit Testing in Java: How Tests Drive the Code*. Morgan Kaufmann Publishers Inc., 2003.

[18] A. Versteegen, Gerhard Chughtai, H. Dörnemann, R. Heinold, and R. Hubert, *Software-Management: Beherrschung Des Lifecycles*. Springer DE, 2002, p. 328.

[19] H. Frühauf, Karol Ludewig, Jochen Sandmayr, *Software-Prüfung: Eine Anleitung zum Test und zur Inspektion*. vdf Hochschulverlag AG, 2007.

[20] D. W. Hoffmann, *Software-Qualität*. Springer DE, 2008, p. 568.

[21] G. Thome and S. Wolfgang, *Grundlagen und Modelle Des Information Lifecycle Management*. Springer DE, 2007, p. 395.

[22] J. Matevska, *Rekonfiguration komponentenbasierter Softwaresysteme zur Laufzeit*. Wiesbaden: Vieweg+Teubner, 2010.

[23] J. Wu, L. Yang, and X. Luo, “Jata: A Language for Distributed Component Testing,” *2008 15th Asia-Pacific Software Engineering Conference*, no. 60603039, pp. 145–152, 2008.

[24] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe, “On the Design of the New Testing Language TTCN-3,” pp. 161–176, Aug. 2000.

[25] G. Xie, “Decompositional veri .cation of component-based systems - a hybrid approach,” in *Proceedings. 19th International Conference on Automated Software Engineering, 2004.*, pp. 414–417.

[26] T. Bauer and R. Eschbach, “Enabling statistical testing for component-based systems,” *subs.emis.de*, pp. 357–362, 2010.

[27] T. Toroi, “Testing Component-Based Systems Towards Conformance Testing,” University of Kuopio, PHD Thesis, 2009.

[28] AjaxProjects.com, “Mockito Tutorial.” [Online]. Available: http://www.ajaxprojects.com/ajax/tutorialdetails.php?itemid=957. [Accessed: 08-Aug-2012].

[29] B. Sami and G. Volker, *Testing Commercial-off-the-Shelf Components and Systems*. Springer, 2005, p. 409.

[30] T. Artner and T. Grechenig, “Konzeption und Entwurf eines Frameworks zur Unterstützung der Testdurchführung in sicherheitskritischen serviceorientierten Umgebungen,” TU Wien, Master Thesis, 2011.

[31] D. Karastoyanova and A. Buchmann, “Components, middleware and Web services,” *Proceedings of the IADIS International Conference on WWW/Interne*, p. 4, 2003.

[32] S. Auth, *Web Services*, 7th ed. GRIN Verlag, 2006, p. 27.

[33] M. Richards, R. Monson-Haefel, and D. A. Chappell, *Java Message Service*. O’Reilly Media, Inc., 2009, p. 336.

[34] M. Hapner, *Java Message Service Api Tutorial and Reference: Messaging for the J2Ee Platform*. 2002, p. 510.

[35] C. Leiden and M. Wilensky, *TCP/IP For Dummies*. John Wiley & Sons, 2009, p. 456.

[36] S. Biffl and A. Schatten, “A Platform for Service-Oriented Integration of Software Engineering Environments,” in *Proceeding of the 2009 conference on New Trends in Software Methodologies Tools and Techniques Proceedings of the Eighth SoMeT*, 2009, pp. 75–92.

[37] S. Biffl, A. Schatten, and A. Zoitl, “Integration of heterogeneous engineering environments for the automation systems lifecycle,” in *2009 7th IEEE International Conference on Industrial Informatics*, 2009, pp. 576–581.

[38] S. Engineering, S. Committee, and I. Computer, *IEEE Recommended Practice for CASE Tool Interconnection — Characterization of Interconnections*, no. January. 2007.

[39] D. A. Chappell, *Enterprise Service Bus*, vol. 4. O’Reilly Media, Inc., 2004, pp. 16–18.

[40] A. Pieber, S. Biffl, and A. Schatten, “Flexible Engineering Environment Integration for (Software +) Development Teams,” *Design*, vol. 2010, no. 0, 2010.

[41] C. Gritschenberger, “Security for the Integration of Software Tools in Multidisciplinary Engineering Processes,” *openengsb.org*, vol. 2011, 2011.

[42] R. Mordinyi, A. Pieber, and A. Schatten, “Technical Report M1-TR1.3 on Integration Concepts and Architecture of the Open Engineering Service Bus (OpenEngSB),” Vienna, 2011.

[43] SpringSource, “Introduction to JMS & Apache ActiveMQ,” 2008. [Online]. Available: http://www.springsource.com/files/uploads/all/pdf_files/news_event/Introduction_to_Apache_ActiveMQ_Webinar_Slides.pdf. [Accessed: 08-Aug-2012].

[44] The Apache Software Foundation, “OpenWire.” [Online]. Available: http://activemq.apache.org/openwire.html. [Accessed: 25-Aug-2012].

[45] The Apache Software Foundation, “OpenWire specification.” [Online]. Available: http://activemq.apache.org/openwire-version-2-specification.html.

[46] J. Snell, D. Tidwell, and P. Kulchenko, *Webservice-Programmierung mit SOAP*. O’Reilly Germany, 2002, p. 280.

List of Figures