Test Automation Design Patterns for Reactive Software Systems

A.-G. Vouffo Feudjio, I. Schieferdecker
Fraunhofer Institute for Open Communication Systems (FOKUS)
Alain.Vouffo, ina.schieferdecker@fokus.fraunhofer.de

Abstract

Patterns have been successfully applied in software development to improve the development process, by facilitating reuse, communication and documentation of sound solutions. However, the testing domain is yet to benefit from a similar approach. This although, with the growing complexity of test automation solutions, identifying and instrumenting patterns in test design to facilitate reuse appears to be a promising approach for shortening the development cycle and save costs. This paper presents a collection of patterns for designing test automation solutions for reactive software systems and reports on first experiences of applying those patterns in a case study.

1 Introduction

It is now widely acknowledged that testing is no longer just an art, but an engineering discipline in its own, with test development following a similar process as generic software development. Patterns are a canonical documentation of the essential concepts underlying successful solutions to recurrent engineering and design problems. They are used to capture experiences, expertise, and facts to improve system quality and facilitate the production of new solutions. In previous publications, we proposed a similar approach for the design and implementation of tests automation systems [23]. This is particularly interesting with the growing popularity of Model-Driven Testing (MDT), which raises the level of abstraction for test design, to a degree that it allows reuse of concepts for new solutions.

In this paper, we present a selection of test patterns we have collected so far by performing pattern mining on existing test automation solutions designed using different test design and test scripting notations e.g. the Testing and Test Control Notation (TTCN-3) [12] the UML Testing Profile(UTP) [21] or JUnit [15]. Each test pattern is defined along a template we introduced in previous work [23], but which was refined to align with generic pattern methodologies. This work is organised as follows: Section 2 presents a selection of the test patterns we have identified, then Section 3 reports on a case study in which a prototype tool implementing some of those patterns was used to design a conformance test suite for the IP Multimedia Subsystem communication protocol (IMS). Section 3.2 discusses some related work in this area, before Section5 concludes the paper and draws an outlook for further research.

Readers familiar with the pattern concept can skip parts of section 3 and go directly to section 2.1, where the description of patterns starts.

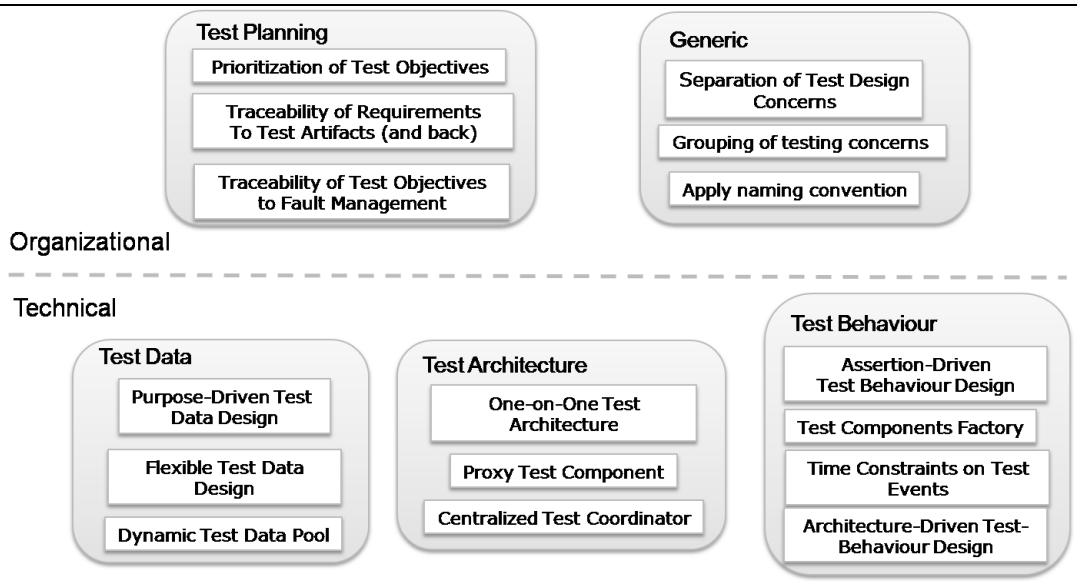

Figure 1. Overview of Identified Test Patterns

2 A Collection of Test Patterns for Black-box test design

We consider test engineering to be a process that starts from the definition of test objectives via abstract test models through to executable test cases. We classify test patterns along the various activities of that process into the following categories:

- *Generic test design patterns* are those applicable to all activities of the test engineering process.
- *Test objectives design patterns*: According to IEEE 829 [16], a test objective[1] is a brief and precise description of the special focus or objective for a test case or a series of test cases. Test objectives can be viewed as the equivalent to system requirements in product development and will certainly benefit from the application of patterns in a similar way as with requirements engineering (RE) patterns [13]. Accordingly, *test objective design patterns* are those addressing that activity in the test engineering process for a given system under test (SUT).
- *Test procedure design patterns*: A test procedure is defined as a prose description of a sequence of actions and events to follow or to observe for executing of a test case. A test procedure describes how a test objective will be assessed. Therefore, Test procedure design patterns are those that are applicable when designing the test procedures for a given SUT.
- *Test architecture design patterns* define good practices and established recommendations in selecting and designing appropriate test architectures. The test architectures describe the topology of a test system, i.e. its composition as a collection of (parallel) test components, interconnected among each other and with the SUT and communicating through Points of Control and Observation (PCOs). Depending on the overall goal of a test e.g. conformance, performance, functionality, robustness, etc., different test architectures are suitable.

[1]Test objectives are sometimes also referred to as *test purposes*, *test requirements* or *test directives* in the literature

- *Test data design patterns* describe approaches for designing the data used in test scenarios as stimuli for the SUT or to express assertions, based on which the SUT's response will be evaluated to assess if they meet their requirements or not.

- *Test behaviour design patterns* document approaches and principles for designing the behaviour of test systems, i.e. the patterns of interactions between entities in a test architecture.

The ultimate goal of test patterns is to increase the quality of tests. Which leads us to the issue of defining the characteristics of test quality. The ISO/IEC 9126 standard [17] defines a model for internal and external quality of software, including quality characteristics and associated metrics.

Zeiss et al [26] demonstrated the applicability of that model to tests and came up with a model combining test-specific quality characteristics such as test effectiveness and test efficiency with more generic ones such as (re-)usability, maintainability, portability etc. However, the value and applicability of those characteristics will mainly depend on the chosen test design strategy. Binder [2] classifies test design strategies in four main categories: Responsibility-based , implementation-based, hybrid and fault-based test design. Responsibility-based test design uses specified or expected responsibilities of an SUT to design tests and is synonymous to "black-box", "specification-based", "behavioural", "functional" testing. Implementation-based test design relies on internal knowledge (e.g. source code, internal design) of the SUT for test design and is also labelled "structural", "white-box", "glass box" or "clear box" testing. Hybrid test design combine responsibility and implementation-based test design. Whereas fault-based testing purposely injects faults in the SUT to check whether those faults are discovered by a test suite.

Although some of the patterns discussed in this work may be applicable to other test design strategies, the main concern is on responsibility-based test design. As expected, this has had repercussions on our methodology for pattern mining. This is illustrated for example by the fact that test effectiveness - i.e. the capability of tests to reveal faults on the SUT - only plays a marginal role for test quality in responsibility-based test, although it's a key factor in implementation-base test design.

Basically, pattern mining for test design patterns can be driven by one or both of the following questions:

- Question 1: What is the best way for designing tests, so that they would help uncovering as many errors of the SUT as possible, before it is delivered to end customers [14]?

- Question 2: What is the best way for designing and modelling tests so that the resulting test specification and/or solution matches best main quality criteria such as reusability, maintainability, understandability etc.?

A pattern template is a list of subjects (sections) that comprise a pattern [2]. The content of the test pattern template depends on which of question 1 or question 2 above is the main driving force for pattern mining. In [2], Binder proposes a test pattern template, which is driven by question 1. Our pattern mining activities are mainly driven by question 2, although question 1 is considered as well. Therefore, we took the test pattern template provided by Binder [2] as the base for our own template, but modified it to reflect the fact that our focus is more on reuse towards more automation than on effectiveness of the tests.

As a consequence subjects such as the *subjects fault model*, the *entry criteria* and the *exit criteria* proposed by [2] were removed from the template. Instead, we added the *applicable test scope* subject to capture the preconditions for applying test patterns. Our test modelling pattern template consists of the following subjects:

- *Pattern name*: A meaningful name for the test pattern.

- *Context*: To which specific context does it apply? This includes the kind of test pattern (organisational vs. design, generic, architectural, behavioural or test data etc.) as well as the test scope for [2] .

[2]The test scope describes the granularity of the item under test [19], which may vary from a low-level entity such as class (for unit testing) to a whole software system (for system testing).

- *Problem*: What is the problem, this pattern addresses and which are the forces that come into play for that problem?
- *Solution*: A full description of the test pattern.
- *Known Uses*: Known applications of the test pattern in existing test solutions (e.g. test specifications, test models, test suites, or test systems) or by test modelling approaches.
- *Resulting context*: What impact does this pattern have on test design in general and on other patterns applicable to that same context in particular?
- *Related patterns*(optional): Test design pattern related to this one or system design patterns in which faults addressed by this test pattern might occur. This section is optional and will be omitted, if no related pattern can be named.
- *References*(optional): Bibliographic references to the pattern. This section is also optional and will be omitted, if no reference can be provided.

Figure 1 displays an overview of those patterns we have identified so far for each of the categories mentioned above. In the following sections, we present a selected subset of those patterns.

2.1 Pattern: Separation of Test Design Concerns

2.1.1 Context

This pattern is a generic organisational test design pattern and is applicable at any test scope for large size test projects. It is assumed that test development is process running in parallel to the development of the SUT or integrated to it, with both of them having the requirements as a common starting point.

2.1.2 Problem

How to organise the file structure of test artifacts. Test artifacts are resources used for storing the design and implementation of a test automation solution. They include high level design models, documentation artifacts through to source code of executable test scripts. The size and the complexity of those test artifacts can grow considerably, raising questions as to how to organise properly to keep a good overview and facilitate collaborative work.

Forces

- To avoid test design activity becoming a bottleneck to the development process, having different teams working in collaboration on the will speed up that process.
- Synchronisation and version control conflicts between the actors involved in test design may cause resources being wasted to address them.
- Large compilation units increase the risk of potential version control conflicts among parallel developers/designers.

2.1.3 Solution

Divide the various tasks over several test designers, by organising modules accordingly. Each task is addressed separately to allow parallel processing. Applying this pattern requires that the technologies involved (e.g. the notation used for designing the tests) provide such mechanisms. Modules may be organised based on the aspect they cover(e.g. Test data, test architecture) or based on the SUT feature they target.

2.1.4 Known Uses

Instantiations of this test pattern can be observed in numerous test automation solutions. The code snippet below from the IPv6 conformance test suite [22] displays an example in TTCN-3 of a test script importing elements of other test modules to design test behaviour.

```
1  module AtsIpv6_Common_Functions {
2   // Importing Generic Libraries
3   //LibCommon
4   import from LibCommon_BasicTypesAndValues all;
5   import from LibCommon_DataStrings all;
6     . . .
7   // Importing test data modules
8   //LibIpv6
9   import from LibIpv6_Interface_Templates all;
10  import from LibIpv6_CommonRfcs_TypesAndValues all ;
11    . . .
12  // Importing test architecture modules
13  //AtsIpv6
14  import from AtsIpv6_TestSystem all;
```

```
15  import from AtsIpv6_TestConfiguration_TypesAndValues all ;
16    . . .
17 } //end module AtsIpv6_Common_Functions
```

2.1.5 Discussion

A difficulty in applying this pattern consists in ensuring that the number of separate modules remains within sensible limits. Otherwise, the effort of managing all parallel activities can reduce the positive impact of the pattern and even lead to less productivity. However a small number of modules will inevitably lead to more version controlling conflicts, with several people potentially working in parallel on the same modules. In such cases the usage of an appropriate version controlling system, along with clearly defined policies is highly recommended.

2.1.6 Related Patterns

This pattern is an application of the Separation of Concern, a.k.a *Divide and Conquer* design pattern known both in generic software engineering, as well as in test design [7].

2.2 Pattern: Prioritization of test objectives

2.2.1 Context

This pattern is an organizational pattern that addresses test objectives design

2.2.2 Problem

Due to resource limitations, often not all test cases can be implemented and/or executed at a within the decided deadline. Some key decisions need to be taken confidently for planning the testing activities and to be able to react to changes in a proper way. Example of key decisions include:

- Which test cases need to be implemented and executed first and which ones can be left aside for later stage in the testing process?
- When can test activities be considered sufficient to provide a level of confidence in the SUT, that is high enough to allow its release?

2.2.3 Solution

As recommended by IEEE 829[16], introduce a prioritization scheme for test objectives in the test model. Prioritization should be provided for a test objective taken individually or for a group of test objectives. Prioritization of test objectives can be based on factors such as:

- Priority level of the feature or requirement(s) covered by the test objective.
- Level of criticality of the errors targetted by the test objective.

Testing activities (e.g. design, implementation, execution) can then be planned based on the priority level of the test objectives and taking the time and resources constraints into account, to ensure that test cases with highest priority are available on time before product delivery.

2.2.4 Discussion

The size of the testing project and the time constraints it faces will be taken into account, whenever the application of this pattern is considered. Obviously, applying the pattern for large scale projects yields more benefits than doing so for smaller ones.

2.2.5 Known Uses

Prioritization of test cases is used implicitly in several instances, though it is not always supported by a specific test notation. Generally a separate tool is used to manage that aspect of the test process. However, it would be highly beneficial to integrate it into the test design process, so that relating it to other tasks in the product development process would be more straightforward.

2.2.6 References

[8, 9, 6]

2.3 Pattern: Traceability of Test Objectives to Requirements

2.3.1 Context

This pattern is an organizational pattern that addresses the management of large test suites under restrictive time and resource constraints.

2.3.2 Problem

To keep control of your development process, you want to be able at any point in time to evaluate the progress of the test project to gain objective criteria for making decisions on the project.

Forces:

- 100% code coverage is an illusion
- 100% requirements coverage is achievable, but needs to supported with clear and sensible metrics.
- Being able at any time to give an estimation of the current coverage of requirements by the specified test objectives will facilitate decision making for releasing the product.

How to achieve traceability between tests and system requirements to enable automatic coverage analysis?

2.3.3 Solution

Provide a mean for linking each test objective to a (set of) requirements or features of the SUT that it addresses. Those include functional as well as non-functional requirements. The test objectives will be designed based on potential risks for the SUT related to a particular feature or as a mean for verifying that the SUT meets the requirements

2.3.4 Known Uses

Please refer to [24] for an overview of requirements traceability that includes numerous examples of traceability to test artifacts as proposed in this pattern.

2.3.5 Discussion

Benefits of this pattern include the fact that the selection of test cases to address specific products or features is facilitated based on the requirements they support. Furthermore, automated requirements coverage analysis of the test cases can be achieved at any time in the lifecycle.

One key difficulty in applying this pattern is to ensure that changes to the test model are propagated in both directions of the link to avoid dead links and keep the test model consistent. The test design tool should take care of that and update a test objective element accordingly, if one of the covered system requirements is altered (e.g. deleted, moved to another location, renamed etc.). Such a propagation of changes could be facilitated by the usage of the same notation or of the same modeling technology (e.g. EMF, MOF) for those aspects being linked with each other. Otherwise, some serious maintainability issues might emerge.

2.4 Pattern: Traceability of Test Objectives to Fault Management

2.4.1 Context

This pattern is an organizational pattern that addresses the management of large test suites under restrictive time and resource constraints.

2.4.2 Problem

In spite of all testing efforts, errors in software are inevitable and will eventually occur. We want to avoid experiencing and fixing the same errors many times. How can it be ensured, that the information gathered in analyzing and fixing errors identified at the user end or through testing can be exploited for the benefit of future testing activities and for improving the overall quality of the software product under test?

Forces

- Fixing errors is generally granted higher priority than documenting them.
- Besides, who cares about fixed issues?
- Developers lack of time to do such additional presumably activities. So they tend to postpone them until they pop-up again as higher priorities.

2.4.3 Solution

Provide a mechanism for ensuring traceability between entries in the fault management system and elements of the testing process. The mechanism should fullfil the following requirements:

- The mechanism should be integrated in the test development/management tool to ensure that it can the process does not cost too much additional effort.
- Every time a failure is (inadvertently or deliberately) discovered on a version of the SUT, make sure that while creating a new entry for that failure in the fault management system, that it is associated with a test objective addressing the root cause of the bug and that test cases are implemented to cover that test objective.
- Provide technical means for enforcing that policy automatically online (i.e. in the process of creating the entries in the model repository) or offline (after the elements have been created)
- Automatically integrating the newly added tests in subsequent regression tests would yield additional benefits.

2.4.4 Known Uses

Agile methods apply this pattern by making test development an integrated part of the development lifecycle (e.g. Test-driven development in XP).

2.4.5 Discussion

The same type of potential issues identified for the *traceability of test objectives to system requirements* pattern (section 2.3) also apply for this pattern.

2.5 Pattern: One-on-One Test Architecture

2.5.1 Context

This pattern addresses test architecture design for an SUT that can be viewed as one entity providing a well-known set of entry points and interacting with its environment following a sequential non-concurrent behaviour. Functional testing at unit or system level is the goal.

2.5.2 Problem

How to design a static test architecture for achieving testing the SUT with highest possible efficiency.

Forces

- Resources planned for testing are generally and straightforward solutions are always welcome.
- The level of complexity of the test system should be kept as low as possible, to keep maintainance and associated efforts as low as possible.
- Usage of concurrency in the test system increases the risk of introducing erroneous test behaviour and the cost of the test system, because a coordination mechanism is required to control the choreography of parallel test components.

2.5.3 Solution

Design the test architecture consisting of one single test component connected to the SUT in a way that it can stimulate the SUT and verify its response to those stimuli. One possible way of achieving that is by making the test component a mirrored image of the SUT, e.g. by providing interfaces required by the SUT and using interfaces the SUT provides.

Figure 2 displays two examples resulting from applying that pattern. The upper part of the figure shows a test architecture consisting of a single test component that uses one port both for sending impulses to and receiving responses from the SUT to verify its correct behaviour. On the other hand, the lower part of the figure illustrates a test architecture for an SUT providing three different entry points for stimuli and responses. Benefits: Having a

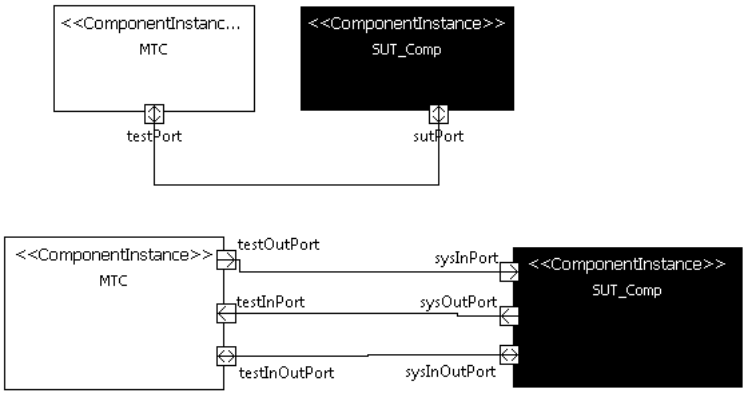

Figure 2. Test architecture Diagram for One-on-One Pattern

single test component implies that synchronization mechanisms based on message exchange or other Remote Procedure Control(RPC) or similar mechanisms do not have to be implemented at the testing side. Variables defined

in the test component can be used to describe states based on which decisions can be made on the test verdict. Shortcomings: The test component has to emulate the complete behaviour of system component it replaces. Depending on the level of complexity of that behaviour, this might be more or less difficult to achieve. Furthermore, having a single component makes it difficult to deal with concurrency at the testing side, if required.

2.5.4 Known Uses

This pattern is applied in numerous conformance test suites, e.g. the collection of IPv6 test suites [22] used e.g. for the IPv6 logo brand ,the IMS benchmark test suite [5] used for performance testing IMS server equipment or the CORBA component test suite [1] used for integration testing of CORBA components

2.5.5 Discussion

Potential difficulties in handling concurrent behaviour from the SUT and to emulate similar behaviour to stimulate the SUT.

2.5.6 Related Patterns

This pattern is the logical opposite to the *Centralized Test Coordinator* test pattern described in section 2.6. It is also referred to as the *Centralized tester* test pattern [10].

2.5.7 References

[10]

2.6 Pattern: Centralized Test Coordinator for Concurrent Test Components

2.6.1 Context

- This pattern addresses test architecture design.
- This pattern is more applicable to integration and system testing. It is less the case for unit testing at the class level. However, it can be applied for system testing, whereby a unit testing framework is instrumented for that purpose.

2.6.2 Problem

How to model a test architecture, that is suitable for load- , performance- or conformance testing on an SUT requiring parallel and possibly distributed processing.

Forces The motivations for this pattern are:

- An SUT featuring concurrent behaviour cannot be verified through a test system supporting only sequential behaviour.
- Certain requirements of software (e.g. robustness, load, performance) can hardly be addressed using test architectures that allow only sequential behaviour.
- Simple test architectures (e.g. the *One-on-One test architecture pattern*) restrict the level of flexibility for the test system with regard to deployment. The fact that a distributed testing setup would not be possible is an example of those restrictions. A conscequence of those restrictions is that certain test scenarios would not be possible.

However, this pattern comes with its liabilities that should be considered as well:

- It must be ensured that, despite the introduction of concurrency in the test system, the tests remain reproducible and deterministic.
- The introduction of concurrency will require some form of coordination between the entities involved. The effort for providing that coordination scheme should be taken into account as well.

2.6.3 Solution

As depicted on figure 3, this pattern features a test component acting as test coordinator and thus controlling the life cycle other components it controls. Each of the controlled test components is connected to the controlling component via a connection through which coordination messages can be exchanged to control the components' behaviour. To keep the overhead of processing those coordination messages as low as possible, to not affect the proper test behaviour, coordination messages should be kept as simple as possible in their structure. The real testing activities are performed by the controlled test components, which are directly connected to the SUT.

2.6.4 Known Uses

Several TTCN-3 projects such as [20] involving UTML protocol testing (Siemens) and [4] involving BCMP protocol performance testing.

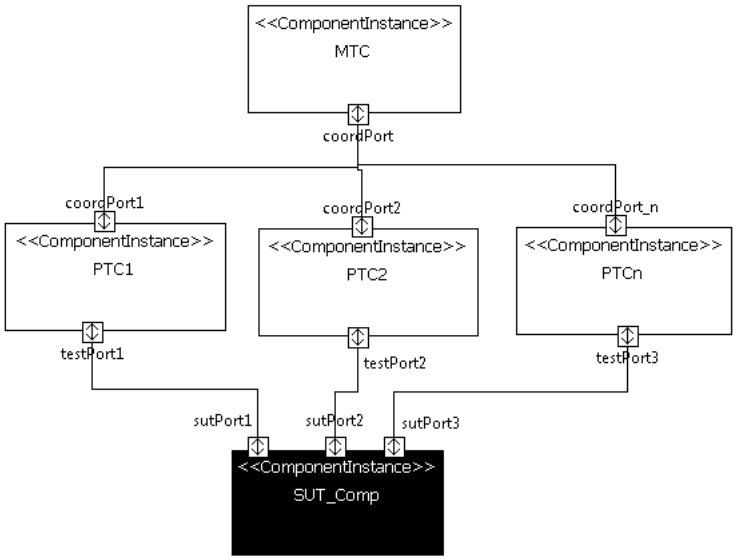

Figure 3. Test architecture Diagram for Centralized Test Coordinator Pattern

2.6.5 Discussion

A coordination scheme is required between the main test component and the parallel test components to control the latters behaviour according to the overall test choreography. The additional load and delays created by that communication should be taken into account while evaluating the SUT component's test results.

2.6.6 Related Patterns

This pattern is the opposite of the *One on One test architecture* pattern defined in section 2.5

2.6.7 References

[10]

3 Evaluation of the Approach: IMS Case Study

3.1 MDTester: A Pattern-Oriented Test Design Tool

Integrating patterns in a process requires a suitable concept, that would allow the creation of new artifacts in a flexible and efficient way, while at the same time ensuring that the rules defined by the patterns are followed in the creation process or can be verified afterwards. Domain Specific Modeling Languages (DSML) provide a good mean for integrating patterns to a given process. Firstly, because they operate at a level of expression, that is abstract enough to express concepts in a solution-independent, but yet formal manner. Secondly, because they can be tailored precisely to define model templates and associated rules, that are specific to the targeted process' domain. Therefore, to evaluate the impact of the patterns we presented in section 3, we defined a UML MOF Meta-model for a DSML dedicated to black box test engineering. The particularity of this DSML is that, tests are modelled based on meta-elements representing the patterns we mentioned earlier.

Test Pattern	Implementation Status	Application to Case Study
Separation of test design concerns	Yes	Yes
Grouping of concerns	Yes	Yes
Naming convention	Yes	Yes
Prioritization of test objectives	Yes	No
Traceability of test objectives to requirements	Yes	Yes
Traceability of test objectives to fault management	Yes	No
One on One test architecture	Yes	No
Proxy test component	Yes	Yes
Centralized test coordinator	Yes	Yes
Purpose-driven test data design	Yes	Yes
Flexible test data definition	Yes	Yes
Dynamic test data pool	No	No
Focus on expected test behaviour	Yes	Yes
Test component factory	No	No
Time constraints	Yes	Yes

Table 1. Summary of Test Patterns and Status

3.2 The IMS testing case study

Following an MDE process, we developed MDTester, a tool to support pattern-oriented test design with automated generation of test designs according to selected patterns and automated model transformations of the high-level test design into test scripting or test specification notation for specific target test environments (e.g. TTCN-3, JUnit). The MDTester tool was used to design functional tests for the IP Multimedia Subsystem (IMS) architecture. Table 1 lists all test patterns and their implementation status in the prototype tool, as well as their application to the IMS case study test model.

The impact of model-driven and pattern-oriented test development can be analyzed from a quantitative and a qualitative view point. The purpose of quantitative analysis is to evaluate how productivity is affected by the introduction of the methodology. On the other hand, qualitative analysis aims at measuring the effect on quality factors, both of the process itself and of its output, i.e. the generated test scripts. The goal of the case study was to analyse both the qualitative and the quantitative aspects of that impact and at the same time, to compare the results with those obtained with a “traditional” test development approach.

3.2.1 Quantitative Analysis

A key metric for quantitative analysis of any development process is productivity. Evaluating the productivity of pattern oriented test development is a relatively straightforward task. For that purpose, we simply have to correlate the output (e.g. number of implemented test cases) to the invested effort (e.g. number of person-days/person-months involved) for a project or a series of projects. However, to measure the impact of introducing a new approach on that productivity is a less trivial task, because productivity data before and after the introduction of the new approach need to be compared with each other. Ideally, to ensure a fair comparison, at least the following conditions need to be fulfilled:

- Both methodologies should be applied on the same case study: The starting point for both test development approaches should be the same system specification or test plan, targeting the same SUT
- Separate teams should apply the methodology, each on its side in a separate project.
- The same time frame will apply to both projects and results will be collected at the end for evaluation.
- Both teams should have comparable level of expertise in their respective field.

However, we could not provide such an ideal setup for our IMS case study. Therefore we had to base our quantitative comparison on assumptions resulting from statistical analysis of past TTCN-3 test development projects.

Taking into account that the project duration was set to 5 person-days and that a total result of 19 test cases were implemented at its end, productivity factor is 19/5 = 3.8 test cases/day. It should be pointed that, this result was obtained with team of designers with a rather low level of testing and modelling expertise. Therefore, it can be assumed that slightly higher results would be obtained with experienced test designers.

To measure the productivity gain generated by our approach, we compare our results with those generally obtained through "traditional" test development approaches. Generally, for TTCN-3 test development, realistic estimations of productivity range between 2 and 5 test cases/day. The obtained results indicate that, if the existing process allows a production rate of more than 4 test cases/day (including test objectives definition, test procedure design and documentation), then applying our methodology would rather cause a productivity loss. On the other hand, the productivity could be significantly improved (30 to 90%), when the production rate of the existing methodology is between 2 and 4 test cases/day.

Moreover, if we estimate that, the specification of a test plan (test objectives) and of test procedures consumes 20% of the effort in pattern-oriented test development and are generally not taken into account, when estimating the productivity of the test development process, then the productivity gain is even higher.

3.2.2 Qualitative Analysis

Using model-driven approach to test development offers a wide range of qualitative benefits, compared to traditional development approach. Test models offer a higher level of readability, maintainability, documentation and flexibility that plain test scripts and non-formal notations. Furthermore, existing MDE frameworks (e.g. Eclipse EMF, TOPCASED) provide a wide range of functionalities for creating, managing, validating and transforming models, that can be used to provide powerful tool chains to support the process. However, a source of general concern is the quality of the test scripts generated automatically from the process. For our case study, we used the TRex [25] tool to measure the quality of the generated TTCN-3 test scripts. The authors of TRex define a metric called *Template coupling* (ranging between 1 and 3) to measure the maintainability of TTCN-3 scrips. The automatically generated IMS test scripts scored 1.015 on that metrics, indicating the high level of maintainability of those scripts (1.0 is best).

4 Related Works

The potential benefits of cataloguing best practices and patterns in test design has been advocated by several authors before. Binder [2] discusses a test pattern template, based on a pattern language of object oriented testing (PLOOT) proposed by Firesmith [11] and introduces a collection of test patterns from the object-oriented software design domain. Meszaros [18] presents a collection of test patterns for unit testing. Howden [14] presents a collection of patterns in selecting tests for maximum error detection. It appears that existing work on test patterns tend to focus on interactions at the object level and are hardly applicable for higher level (i.e. integration, system, and acceptance-level) testing whereby the applied programming paradigm are less relevant. Delano et al [3] present a collection of patterns focussing more on the organisational aspects of test development as a process, rather

than on test design itself. On the other hand, Dustin [7] covers all aspects of test development, with one chapter dedicated to test design and documentation. in 2005, the European Telecommunications Standards Institute (ETSI) started an initiative on patterns in test development (PTD) in which some of the patterns defined in this work were introduced and discussed. However, to the best of our knowledge, none of the existing work attempts to formalise test patterns, so that they could be instrumented to support the test development process in an automated way.

5 Conclusion and Outlooks

This paper has presented first ideas on a collection of patterns of test design based on a template defined for that purpose. First experiences with that prototype tool chain have shown some promising results. However, model-driven test development has not reached a high level of popularity yet. Therefore, some of the patterns described here can only be considered as mere candidates and will require further analysis with regard to their usability and their consequences. Also, we have presented a case study in which those patterns have been applied to develop tests for IMS. An analysis of the approach through that case study indicates that it can significantly improve the test process, both quantitatively and qualitatively. In the future, we intend to conduct further case studies to analyze the impact of the approach, when developing tests for other domains.

6 Acknowledgements

We would like to thank our shepherds Uwe Zdun and especially Christian Kohls (on-site shepherd) who were both very helpful in the process of improving this paper through their challenging comments, their patience as well as their interesting ideas. Also, we would like to thank all participants to writer's workshop E at the EuroPLoP 2009 conference for their contributions to bring this paper into shape. A special thank you to Dietmar (Didi) Schtz, Michael Kircher, Heiko Hashizume, klaus Marquardt and last but not least, Markus Voelter!

References

[1] Harold J. Batteram, Wim Hellenthal, Willem A. Romijn, Andreas Hoffmann, Axel Rennoch, and Alain Vouffo. Implementation of an open source toolset for ccm components and systems testing. In Roland Groz and Robert M. Hierons, editors, *TestCom*, volume 2978 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004. [cited at p. 11]

[2] Robert V. Binder. *Testing Object Oriented Systems: Models, Patterns and Tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999. [cited at p. 3, 15]

[3] David E. Delano and Linda Rising. System test pattern language copyright 1996 ag communication systems corporation permission is granted to make copies for plop '96., 1996. [cited at p. 15]

[4] Sarolta Dibuz, Tibor Szabó, and Zsolt Torpis. Bcmp performance test with ttcn-3 mobile node emulator. In *TestCom*, pages 50–59, 2004. [cited at p. 12]

[5] George Din. An ims performance benchmark implementation based on the ttcn-3 language. *Int. J. Softw. Tools Technol. Transf.*, 10(4):359–370, 2008. [cited at p. 11]

[6] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Softw. Engg.*, 11(1):33–70, 2006. [cited at p. 7]

[7] E. Dustin. *Effective Software Testing. 50 Specific Way to Improve Your Testing*. Addison-Wesley, 2003. [cited at p. 6, 16]

[8] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002. [cited at p. 7]

[9] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12(3):185–210, 2004. [cited at p. 7]

[10] M. Frey et al. Etsi draft report: Methods for testing and specification (mts); patterns for test development (ptd). Technical report, European Telecommunications Standards Institute (ETSI), 2005. [cited at p. 11, 13]

[11] D.G. Firesmith. Pattern language for testing object-oriented software. *Object Magazin*, 1996. [cited at p. 15]

[12] Methods for Testing and Specification (MTS). The testing and test control notation version 3; part1: Ttcn-3 core language. Technical report, European Telecommunications Standards Institute (ETSI), 2003. [cited at p. 1]

[13] Lars Hagge and Kathrin Lappe. Sharing requirements engineering experience using patterns. *IEEE Software*, 22:24–31, 2005. [cited at p. 2]

[14] William E. Howden. Software test selection patterns and elusive bugs. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, pages 25–32, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 3, 15]

[15] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, September 2003. [cited at p. 1]

[16] IEEE. Draft ieee standard for software and system test documentation (revision of ieee 829-1998). Technical report, IEEE, 2008. [cited at p. 2, 7]

[17] ISO/IEC. Iso/iec standard no. 9126: Software engineering product quality; parts 14. Technical report, Organization for Standardization (ISO) / International Electrotechnical Commission (IEC), Geneva, Switzerland, 2001-2004. [cited at p. 3]

[18] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. [cited at p. 15]

[19] Helmut Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, Dissertation, Universität Göttingen, November 2004 (electronically published on http://webdoc.sub.gwdg.de/diss/2004/neukirchen/index.html and archived on http://deposit.ddb.de/cgi-bin/dokserv?idn=974026611 . Persistent Identifier: urn:nbn:de:gbv:7-webdoc-300-2), November 2004. [cited at p. 3]

[20] Andrej Pietschker. Automating test automation. *Int. J. Softw. Tools Technol. Transf.*, 10(4):291–295, 2008. [cited at p. 12]

[21] OMG ptc. Unified modeling language: Testing profile, finalized specification. Technical report, Object Management Group, 2004. [cited at p. 1]

[22] Stephan Schulz. Test suite development with ttcn-3 libraries. *Int. J. Softw. Tools Technol. Transf.*, 10(4):327–336, 2008. [cited at p. 5, 11]

[23] Alain Vouffo-Feudjio and Ina Schieferdecker. Test patterns with ttcn-3. In *FATES*, pages 170–179, 2004. [cited at p. 1]

[24] Stefan Winkler and Jens von Pilgrim. A survey of traceability in requirements engineering and model-driven development. *Software and Systems Modeling*, December 2009. [cited at p. 8]

[25] Benjamin Zeiß, Helmut Neukirchen, Jens Grabowski, Dominic Evans, and Paul Baker. TRex - An Open-Source Tool for Quality Assurance of TTCN-3 Test Suites. In *Proceedings of CONQUEST 2006 – 9th International Conference on Quality Engineering in Software Technology, September 27-29, Berlin, Germany*. dpunkt.Verlag, Heidelberg, September 2006. [cited at p. 15]

[26] Benjamin Zeiß, Diana Vega, Ina Schieferdecker, Helmut Neukirchen, and Jens Grabowski. Applying the ISO 9126 Quality Model to Test Specifications – Exemplified for TTCN-3 Test Specifications. In *Software Engineering 2007 (SE 2007). Lecture Notes in Informatics (LNI) 105. Copyright Gesellschaft für Informatik*, pages 231–242. Köllen Verlag, Bonn, March 2007. [cited at p. 3]