Validation of Software Migration

Model-Driven Co-Migration of Test Cases

Ivan Jovanovikj

Faculty of Computer Science, Electrical Engineering, and Mathematics
Paderborn University

Dissertation submitted in partial fulfillment
of the requirements for the degree of
Doktor der Naturwissenschaften (Dr. rer. nat.)

December 2020

I would like to dedicate this thesis to my loving parents, Violeta and Gradimir,
to my sister Aleksandra, and my aunt Snežana.

Acknowledgements

Writing this thesis would never have been possible without being accompanied and supported by many people. Hence, I would like to thank the following people for sharing their experience, time, and patience.

First and foremost, I would like to thank my advisor and mentor, Prof. Dr. Gregor Engels, for his trust and for giving me the opportunity to pursue my degree in his research group. Gregor, thanks a lot for all the encouragement, advice, and guidance. I am especially grateful for your continuous support during the final phase of writing up the thesis and also in phases of intensive project work. Talking about project work and industry, I would also like to thank Dr. Stefan Sauer who allowed me with a lot of confidence to be part of s-lab – Software Quality Lab or Software Innovation Lab nowadays. I also thank the external co-reviewer of my thesis Prof. Dr. Jürgen Ebert. Thank you Prof. Ebert for your review of this thesis. In addition, I would like to express my gratitude to Prof. Dr. Eric Bodden and Prof. Dr. Olaf Zimmermann for being members of my doctoral committee.

During my research, I was blessed to have great colleagues at work and I would like to thank all of them for the joint work on publications or their critical reviews and feedback. First, I would like to thank my office mate, Dr. Enes Yiğitbaş. Enes, thank you very much for the productive collaboration on numerous papers, and thank you for introducing me to the domain of your research. Also, thanks a lot for your encouragement and support especially during the project-intensive phases. Then, I would like to thank Dr. Marvin Grieger, whose advice and critical feedback helped me a lot in the crucial phase of my PhD, the topic finding phase. I also thank Dr. Barış Güldali for his advice at the early beginning and his unconditional support regarding project work. Thank you Barış and thank you Marvin for also being co-authors on my very first scientific paper. Then, I would like to thank Dr. Anthony Anjorin, whose critical feedback and advice helped me a lot regarding the final part of my work. Thank you Tony for the long and very productive discussions on mutation analysis. Further, I thank Simon Schwichtenberg, for sharing his project experience and data, that was crucial for the conceptualization and evaluation of my work. Then, I also thank Mirko Rose, my seminar thesis mentor, whose mentoring and feedback helped me a lot when I was entering the scientific and research world. I would also like to thank Dr. Masud

Fazal-Baqie for his worthful advice on doing a PhD in general and his encouragement at the very beginning. I also thank Dennis Wolters for the fruitful discussions in the basketball domain. Last, but not least, I would like to thank Prof. Dr. Christian Gerth, my master's thesis mentor, whose mentoring and critical feedback inspired me to do a PhD.

Finally, I want to thank my parents, Violeta and Gradimir, my sister Aleksandra, and my aunt Snežana, for being my greatest supporters. Thank you for loving, understanding, and supporting me with great patience through my PhD journey.

Abstract

Software testing plays an important role in the context of software migration as it is used to validate and ensure functional equivalence as a key requirement. As the creation of test cases is an expensive and time-consuming activity, whenever test cases are existing, their reuse should be considered, thus implying their co-migration. However, reusing test cases is beneficial, only when the test cases have some value. Migrating test cases that cover parts of the system that are not used anymore or test cases with low quality can result in migrated test cases which cannot properly validate the migrated system. Therefore, before doing anything with the test cases, their quality needs to be evaluated. During the actual co-migration of test cases, two main challenges have to be addressed: situativity and co-evolution. The first one suggests that when a test migration method is developed, the situational context has to be considered as it influences the quality and the effort regarding the test case migration. The latter suggests that the changes that happen to the system have to be considered and eventually reflected to the test cases. Lastly, as the migrated test cases are used as safeguards for the system migration, their correct migration is crucial. Once migrated and executed, the test report may also contain some false positives and false negatives which could potentially lead to false conclusions about the correctness of the migrated system. Therefore, the validation of the test case migration itself is also necessary.

We address the aforementioned challenges by proposing a framework which provides an end-to-end solution by addressing the three general migration phases: pre-migration, migration, and post-migration. Firstly, in the pre-migration phase, a test case quality evaluation is performed to evaluate the quality of the existing test cases. This phase mainly relies on our approach for development of quality plans for test case quality evaluation. Then, during the migration phase, by employing situational method engineering, a situational method for the test cases is developed and enacted. This development of situation-specific test migration methods is centered around the idea of double horseshoe model, which incorporates two horseshoe models, one for the system and another for the test case migration. We extend the basic method development process with a co-evolution analysis in order to detect and reflect the changes from the system migration to the test cases. Finally, during the post-migration

phase, we validate the test case migration by applying our novel approach for test case migration which is based on mutation analysis.

In order to demonstrate the applicability of the developed framework in practice, we performed two feasibility studies in which parts of the well-known Eclipse Modeling Framework (EMF) along with the Object Constraint Language (OCL) were migrated to the multi-platform enabled modelling framework CrossEcore. The feasibility studies addressed two different languages in the target environment, namely C# and TypeScript, and consequently two different target testing frameworks, MSUnit and Jasmine.

Zusammenfassung

Das Testen von Software spielt im Kontext der Softwaremigration eine wichtige Rolle, da es zur Validierung und Sicherstellung der Funktionsäquivalenz als Schlüsselanforderung verwendet wird. Da die Erstellung von Testfällen eine teure und zeitaufwändige Aktivität ist, sollte ihre Wiederverwendung in Betracht gezogen werden, wenn Testfälle vorhanden sind, was ihre Co-Migration impliziert. Die Wiederverwendung von Testfällen ist jedoch nur dann von Vorteil, wenn die Testfälle einen gewissen Wert haben. Das Migrieren von Testfällen, die Teile des Systems abdecken, die nicht mehr verwendet werden, oder von Testfällen mit geringer Qualität kann zu migrierten Testfällen führen, die das migrierte System nicht ordnungsgemäß validieren können. Bevor etwas mit den Testfällen gemacht wird, muss daher deren Qualität bewertet werden. Während der eigentlichen Co-Migration von Testfällen müssen zwei Hauptherausforderungen angegangen werden: Situativität und Co-Evolution. Die Testmigrationsmethode sollte den Situationskontext berücksichtigt, da er die Qualität und den Aufwand für die Testfallmigration beeinflusst. Letzteres legt nahe, dass die Änderungen, die am System auftreten, berücksichtigt und schließlich in den Testfällen abgespiegelt werden müssen. Da die migrierten Testfälle als Schutzmaßnahmen für die Systemmigration verwendet werden, ist ihre korrekte Migration von entscheidender Bedeutung. Nach der Migration und Ausführung kann der Testbericht auch einige falsch positive und falsch negative Ergebnisse enthalten, die möglicherweise zu falschen Schlussfolgerungen über die Richtigkeit des migrierten Systems führen können. Daher ist auch die Validierung der Testfallmigration selbst erforderlich.

Wir begegnen den oben genannten Herausforderungen, indem wir ein Framework vorschlagen, das eine End-zu-End-Lösung bietet, indem wir die drei allgemeinen Migrationsphasen behandeln: Vormigration, Migration und Nachmigration. Zunächst wird in der Vormigrationsphase eine Bewertung der Testfallqualität durchgeführt, um die Qualität der vorhandenen Testfälle zu bewerten. Diese Phase basiert hauptsächlich auf unserem Ansatz zur Entwicklung von Qualitätsplänen für die Bewertung der Testfallqualität.

Während der Migrationsphase wird dann unter Verwendung von Situationsmethoden-Engineering eine Situationsmethode für die Testfälle entwickelt und implementiert. Diese Entwicklung situationsspezifischer Testmigrationsmethoden konzentriert sich auf die Idee

des Doppelhufeisenmodells, das Hufeisenmodelle sowohl für das System als auch für die Testfallmigration umfasst. Wir erweitern den grundlegenden Methodenentwicklungsprozess um eine Co-Evolutions-Analyse, um die Änderungen von der Systemmigration zu den Testfällen zu erkennen und widerzuspiegeln. Schließlich validieren wir während der Nachmigrationsphase die Testfallmigration, indem wir unseren neuartigen Ansatz für die Testfallmigration anwenden, der auf Mutationsanalyse basiert.

Um die Anwendbarkeit des entwickelten Frameworks in der Praxis zu demonstrieren, haben wir zwei Machbarkeitsstudien durchgeführt, in denen Teile des bekannten Eclipse Modeling Framework (EMF) zusammen mit der Object Constraint Language (OCL) auf die plattformübergreifende Modellierungsframework CrossEcore migriert wurden. Die Machbarkeitsstudien befassten sich mit zwei verschiedenen Sprachen in der Zielumgebung, nämlich C# und TypeScript, und folglich mit zwei verschiedenen Ziel-Testing-Frameworks, MSUnit und Jasmine.

Contents

List of Figures **xiii**

List of Tables **xviii**

I Foundations and Related Work 1

1 Introduction **2**

1.1 Motivation . . . 2
1.2 Problem Statement . . . 7
1.3 Solution Overview . . . 10
1.4 Publication Overview . . . 13
1.5 Structure of the Thesis . . . 14

2 Foundations **17**

2.1 Software Reengineering . . . 17
2.1.1 Software Migration . . . 18
2.1.2 Concept Modeling . . . 19
2.2 Software Co-Evolution . . . 21
2.2.1 Change Detection . . . 22
2.2.2 Impact Analysis . . . 22
2.2.3 Change Propagation . . . 23
2.2.4 Validation . . . 23
2.3 Method Engineering . . . 24
2.3.1 Situational Method Engineering . . . 24
2.3.2 Project-Specific Software Engineering Methods . . . 25
2.3.3 Concept-Based Engineering of Situation-Specific Methods . . . 27
2.4 Software Testing . . . 28
2.4.1 Model-based Testing . . . 28

2.4.2 Testing Languages and Testing Frameworks . . . 29
2.4.3 Mutation Testing . . . 30
2.4.4 Test Case Quality . . . 32
2.5 Technologies . . . 35
2.5.1 Eclipse Modeling Framework . . . 35
2.5.2 Xtend . . . 36
2.5.3 Object Constraint Language (OCL) . . . 38

3 Requirements and Related Work 41
3.1 Test Case Co-Migration Scenario . . . 41
3.1.1 System Migration . . . 43
3.1.2 Test Case Migration . . . 44
3.2 Solution Requirements . . . 46
3.2.1 Pre-Migration Phase: Test Case Quality Evaluation . . . 46
3.2.2 Migration Phase: Co-Evolution and Migration . . . 47
3.2.3 Post-Migration Phase: Migration Validation . . . 49
3.3 Related Work . . . 49
3.3.1 Quality Evaluation . . . 50
3.3.2 Evolution, Reengineering, and Migration . . . 51
3.3.3 Migration Validation . . . 60
3.4 Summary . . . 62

II Solution Concept 65

4 Solution Overview 66
4.1 Overview of the TeCoMi Framework . . . 66
4.2 Pre-Migration Phase: Test Case Quality Evaluation . . . 68
4.2.1 Roles . . . 70
4.3 Migration Phase: Co-Evolution Analysis and Method Engineering . . . 71
4.3.1 Method Base . . . 71
4.3.2 Method Engineering Process . . . 72
4.3.3 Roles . . . 74
4.4 Post-Migration: Migration Validation . . . 75
4.4.1 Roles . . . 77
4.5 Summary . . . 77

5 Pre-Migration Phase:
Test Case Quality Evaluation 79
5.1 The Test Case Quality Plan Approach 79
5.1.1 Characterization of Context 81
5.1.2 Identification of Information Needs 85
5.1.3 Definition of a Common Quality Understanding 88
5.1.4 Definition of Measurements 90
5.1.5 Tool Support 94
5.2 The Quality Evaluation Process 95
5.2.1 Context Characterization 96
5.2.2 Test Case Quality Plan Creation 96
5.2.3 Measurement Tool Implementation 97
5.2.4 Execution and Decision-Making 99
5.3 Summary and Discussion 99

6 Migration Phase: Method Engineering considering Co-Evolution Analysis 102
6.1 Overview of the Migration Phase 102
6.2 Method Base 106
6.2.1 Transformation Phase Fragments 106
6.2.2 Tool Implementation Phase Fragments 113
6.2.3 Test Method Patterns 116
6.2.4 Co-Migration Method Patterns 121
6.2.5 Formalization 127
6.3 Method Engineering Process 128
6.3.1 Situational Context Identification 129
6.3.2 Transformation Method Construction 138
6.3.3 Tool Implementation 143
6.3.4 Transformation 145
6.4 Summary and Discussion 146

7 Post-Migration Phase: Migration Validation 148
7.1 Overview of the Post-Migration Phase 148
7.2 Mutation Analysis Repository 151
7.2.1 Mutation Analysis Scenarios 152
7.2.2 Mutation Operators 166
7.2.3 Mutation Method Patterns 169
7.3 Test Case Migration Validation Process 171

7.3.1 Context Characterization . . . 172
7.3.2 Mutation Method Construction . . . 173
7.3.3 Mutation Tool Implementation . . . 177
7.3.4 Mutation Execution and Analysis . . . 182
7.4 Summary and Discussion . . . 184

III Evaluation and Conclusion 187

8 Evaluation 188
8.1 Evaluation Questions . . . 188
8.2 Feasibility Study 1: Junit OCL Test Cases to MSUnit OCL Test Cases . . . 189
8.2.1 Pre-Migration Phase . . . 190
8.2.2 Migration Phase . . . 194
8.2.3 Post-Migration Phase . . . 202
8.3 Feasibility Study 2: JUnit OCL Test Cases to Jasmine OCL Test Cases . . . 207
8.3.1 Migration Phase . . . 208
8.4 Discussion . . . 215
8.5 Summary . . . 218

9 Conclusion and Future Work 219
9.1 Summary of Contributions . . . 219
9.2 Requirements Revisited . . . 222
9.3 Future Work . . . 225

Bibliography 230

A Extended Quality Model for Test Cases 242
A.1 Extended Quality Model for Test Cases . . . 242

List of Figures

1.1 The co-migration problem: test case migration and system migration 4
1.2 The reengineering horseshoe model [KWC98] 6
1.3 The double horseshoe reengineering model for co-migration of test cases . . 10
1.4 The general phases with the corresponding activities of our approach 11
1.5 Overview of the publications related to this thesis 13
1.6 Overview of the thesis structure 15

2.1 Research areas relevant to this thesis 17
2.2 Representation of a software system as a set of concepts (based on [Gri16]) 20
2.3 The staged process model for co-evolution [MD08] 22
2.4 Overview of the MESP framework [FB16] 26
2.5 Overview of the MEFiSTo framework [Gri16] 27
2.6 The general MBT approach [PP04] 29
2.7 The basic mutation testing process 31
2.8 The Test Specification Quality Model [ZVS+07] based on the ISO/IEC 9126-1 Quality Model for External and Internal Quality [ISO01] 33
2.9 The MQP Process [VE08] 34
2.10 An excerpt of the MQP metamodel [VE08] 35

3.1 The co-migration problem: Migration of a part of the EMF framework to provide cross-platform availability 42
3.2 OCL implementation in EMF in JIT-fashion (left), CrossEcore's AOT implementation of OCL (right) [SJGE18] 43
3.3 Implementation of an OCL Test Case in EMF (Just-In-Time (JIT) Compilation) 45
3.4 Implementation of an OCL Test Case in CrossEcore (Ahead-Of-Time (AOT) Compilation) 45
3.5 Evaluation of selected test case quality evaluation approaches against requirements 51

3.6 Evaluation of selected test case evolution approaches against requirements . 54
3.7 Evaluation of selected test case reengineering approaches against requirements 56
3.8 Evaluation of selected software migration and modernization projects against requirements 60
3.9 Evaluation of selected test case validation approaches against requirements 61

4.1 Overview of the different levels of the solution approach 67
4.2 Overview of the pre-migration phase, i.e., the test case quality evaluation . . 69
4.3 Method base: test method fragments and test method patterns & co-migration patterns 72
4.4 Overview of the method engineering process 73
4.5 Overview of the post-migration phase, i.e., the migration validation 76

5.1 Test Case Quality Plan Process. 80
5.2 The context description metamodel 82
5.3 Context factors for test cases 83
5.4 Information needs metamodel 86
5.5 The ISO/IEC 25000 Standard-based Test Case Quality Model 89
5.6 Quality Metamodel based on [Voi09] ant its relation the Information Need Metamodel and the Measurement Metamodel 90
5.7 Overview of the Measurement Metamodel as defined in [Voi09] 91
5.8 A screenshot of the initial dashboard of TCQEval 94
5.9 Core activities of the test case quality evaluation process 96
5.10 An excerpt of a context model created in the TCQEval tool 97
5.11 An excerpt of a quality plan 98

6.1 Core activities of the method engineering process of TeCoMi 103
6.2 Overview of the structure of the method base in TeCoMi 104
6.3 Test method fragments of the transformation phase on the system layer . . . 107
6.4 Test method fragments of the transformation phase on the platform-specific layer 109
6.5 Test method fragments of the transformation phase on the platform-independent layer 111
6.6 Roles and Tools of the transformation phase 113
6.7 Test method fragments of the tool implementation phase 114
6.8 Basic test transformation method patterns. 116
6.9 Schema to characterize test method patterns 117
6.10 Characterization of the Language-based Test Transformation Pattern 118

6.11 Characterization of the Test Language-based Test Transformation Pattern . 119
6.12 Characterization of the Conceptual Test Transformation Pattern 120
6.13 Characterization of the Test Reimplementation Pattern 120
6.14 Characterization of the Test Code Removal Pattern 121
6.15 Co-Migration Patterns containing Test Reimplementation 122
6.16 Co-Migration Patterns containing Language-based Test Transformation . . 123
6.17 Co-Migration Patterns containing Test Language-based Test Transformation 125
6.18 Co-Migration Patterns containing Conceptual Test Transformation 126
6.19 An excerpt of the TeCoMi Intermediate Modelling Language (TIML) . . . 128
6.20 Situational context identification process 129
6.21 The concept metamodel to formalize the concept models in co-migration context 130
6.22 Concept Identification Process 131
6.23 Concept model of the example scenario 132
6.24 The impact metamodel to formalize the relation between the test and system concepts in the co-migration context 133
6.25 Co-Evolution Analysis Process 134
6.26 Impact model of the example scenario. 135
6.27 The influence factor metamodel for the co-migration context 136
6.28 Influence Factor Identification Process 137
6.29 Excerpt of the situational context model showing the evaluated suitability of the test method patterns 138
6.30 Transformation method construction process 139
6.31 Instantiating a fragmented test transformation method specification from a situational context model 139
6.32 Method pattern selection and configuration process 141
6.33 Integrating a fragmented test transformation method specification 141
6.34 Instantiation of tool implementation phase fragments 143

7.1 Core activities of the migration validation process of TeCoMi 149
7.2 Core Components of the Mutation Analysis Repository 150
7.3 Overview of mutation analysis usage scenarios 153
7.4 System and test case migration in the context of our running example . . . 154
7.5 Mutation of Original System 156
7.6 Mutation of Migrated System 159
7.7 Mutation of System Migration 161
7.8 Mutation of Original Test Cases 162

7.9 Mutation of Migrated Test Cases 164
7.10 Mutation for Test Case Migration 165
7.11 Overview of the mutation operator types 167
7.12 Overview of the language mutation operators 167
7.13 Example set of test mutation operators for the JUnit framework 168
7.14 Example set of domain-specific mutation operators for the OCL Language . 169
7.15 Overview of the mutation method patterns 169
7.16 An example of a situational context model 173
7.17 An example of a test transformation method specification 173
7.18 Mutation method construction process 174
7.19 Mutation method specification 176
7.20 The architecture of the mutation framework 177
7.21 The general process of the mutation framework 179
7.22 Implemented mutation tool chain 181
7.23 OCL-specific mutation operators specified as transformation rules 181
7.24 Execution of *Collection Type Replacement* mutation operator specified as a transformation rule 182
7.25 Execution of *Collection Method Replacement* mutation operator specified as a transformation rule 183
7.26 Scenario 4: Original and mutated old test case (Bad Smell 1) 183
7.27 Scenario 4: Mutated migrated test case (bad smell) 184
7.28 Scenario 4: Incorrect implementation of the assertion function 184

8.1 Context Model 191
8.2 Context-specific quality plan 192
8.3 Excerpt of the defined base measures 193
8.4 TCQEval dashboard showing the results of the performed quality evaluation 194
8.5 Configured concept model with the dependencies between the test and system concepts 196
8.6 Horseshoe model for the OCL test case concept 198
8.7 Generic Tool Infrastructure 199
8.8 Enacting a transformation method to transform the test concepts 201
8.9 Evaluation Results 203
8.10 Scenario 4: Original and mutated old test case (Bad Smell 1) 204
8.11 Scenario 4: Mutated migrated test case (Bad Smell 1) 205
8.12 Scenario 4: Incorrect implementation of the assertion function 205
8.13 Scenario 4: Original and mutated old test case (Bad Smell 2) 205

8.14 Scenario 4: Mutated migrated test case (Bad Smell 2) 205
8.15 Scenario 6: Original test cases asserting results of the *greater than* operator 206
8.16 Scenario 6: Original and mutated test case transformation 206
8.17 Scenario 6: Generated migrated test case mutants showing missing migrated test case mutants 206
8.18 Configured concept model with the dependencies between the test and system concepts 209
8.19 Horseshoe Model 211
8.20 Enacting a transformation method to transform the test concepts 213

A.1 The ISO/IEC 25010 Standard-based Quality Model for Test Cases 242

List of Tables

5.1 A base measure counting all test cases 93
5.2 A base measure counting test cases with a specified expected result 93
5.3 An Example of defining an Indicator 93

A.1 Sub-characteristics of Test Effectivity 243
A.2 Sub-characteristics of Reliability 243
A.3 Sub-characteristics of Usability 244
A.4 Sub-characteristics of Performance Efficiency 244
A.5 Sub-characteristics of Security 245
A.6 Sub-characteristics of Maintainability 245
A.7 Sub-characteristics of Portability 246
A.8 Sub-characteristics of Reusability 246

Part I

Foundations and Related Work

Chapter 1

Introduction

In this chapter, we describe the motivation and problem statement of this thesis. Then, we present our solution approach and main contributions. In the end, we present an overview of publications that are related to the solution concept defined in this thesis and provide a structural overview of the thesis.

1.1 Motivation

Software systems are nowadays critical assets for many companies, as their everyday business highly relies on them. But, as business or legal requirements evolve, an existing system has to evolve as well, so that it still fits the changed requirements. Moreover, improving non-functional characteristics or removing bugs also changes an existing system [SHT05]. Over time, an existing system, as anything else, ages and it becomes more difficult for further maintenance, i.e., its evolvability decreases. But, if such a system still has some value for the ongoing business, it is called a legacy software system.

Different solutions exist in practice when it comes to the legacy system problem [SWH10]. Software migration is a well-established method for transferring software systems in new environments while keeping the data and the functionality of the system [BLWG99]. By systematically transforming the system, it minimizes the risk of losing business-critical requirements.

When migrating an existing system into a new environment, a crucial requirement that must be fulfilled, as already mentioned, is that the original system and the migrated system are equal in terms of functionality. In other words, behavioral equivalence must be satisfied after the migration is performed. We define the behavioral equivalence based on the term of exchangeability, i.e., as long as two systems are exchangeable and provide the same behavior to an observer, they are said to be behaviorally equivalent. Hence, migration validation is

a very important step that comes directly after the migration is performed. It assesses the migration's success by checking the behavioral equivalence between the original and the migrated system.

A widely used validation technique in software migration projects is software testing [MA14, Moh10]. Software testing is a method for asserting, among others, whether a software system provides a required functionality [SL09]. A central artifact in software testing is a test case which, in general, is a construct that consists of input data, actions, i.e., test steps, and expected result. In a process called test case design, based on previously defined functional requirements, test cases are created and then executed against the system. When executed, a test case produces an actual result which is then compared to the specified expected result. In case they match, the functionality being tested is said to be validated. Considering a migration context, when all defined test cases pass, it implies that the migrated system behaves as intended and the system migration is considered to be successful. The success of this strategy depends on the test coverage regarding the functional requirements.

Depending on the test case design, which has been seen as an expensive and time-consuming activity [Sne99], the software testing-based approaches used in software migration projects can be grouped into several groups. In a worst-case scenario, the original system is without any documentation, meaning neither a system functional description nor a system design is present. In that case, test cases can be created by observing the system's behavior or system code of the original system. However, this requires a good knowledge about the existing system and a lot of manual work thus resulting in higher costs in terms of time and money. Another scenario is when some kind of system documentation like functional requirements or system design are already given. If that is the case, then these artifacts can be used as a starting point for the design of test cases. However, the lack of structure and some level of formality in these artifacts, especially in the case of legacy software migration, makes the test cases test design a pretty tedious activity. Due to this fact and the fact that the test cases are manually created, this scenario also tends to be risky [Sne99].

As a result, whenever existing test cases for the original system are available, reusing them to test the migrated system is certainly worth considering. Reusing test cases can not only substantially reduce the cost of testing the migrated system but can also help retain valuable information about the expected functionality of the original system and thus the desired functionality of the migrated system. Furthermore, the existing test cases contain very useful test data in terms of test inputs and expected results.

The existing test cases, like any other software artifact, after their initial creation, are often being changed, e.g., due to system changes [FB99] or due to test case refactoring [Mes07]. Existing test cases are being updated or deleted or new test cases are being created. These

actions influence the quality of the test cases which could be expressed by characteristics like effectiveness, understandability, or structuredness. Moreover, some of the existing test cases might no longer test the updated behavior, i.e., they might become obsolete or even erroneous. This basically means that such test cases are useless for the existing system and, thus, for the migrated system as well. In such a scenario, according to the phrase "garbage in, garbage out", migrating useless test cases, would result in useless test cases while still spending resources on migrating them. Therefore, before performing any further migration activities, the value, i.e., the quality of the test cases needs to be considered, to decide whether to migrate the test cases or not.

The migration of the test cases comes down to the problem of co-migration, i.e., the test cases have to be migrated along with the system as their migration is dependent on the migration of the system they are testing (Figure 1.1). The co-migration is practically defined by the co-evolution of the test cases and the corresponding system. In general, co-evolution refers to two or more objects evolving alongside each other, such that there is a relationship between the two that must be maintained [MD08]. In our case, this refers to the test cases evolving alongside the migrating code, such that the test cases remain correct for testing the migrated code. According to [MD08], the co-evolution process consists of the following consecutive activities: *Change Detection*, *Impact Analysis*, *Change Propagation*, and *Validation*. Firstly, in *Change Detection*, all changed parts of the system being migrated are identified. Having these changes identified, in the next step called *Impact Analysis*, all affected parts of the test cases are identified and an estimate of the efforts required to accomplish the change together with involved risks is determined. Then, based on the results

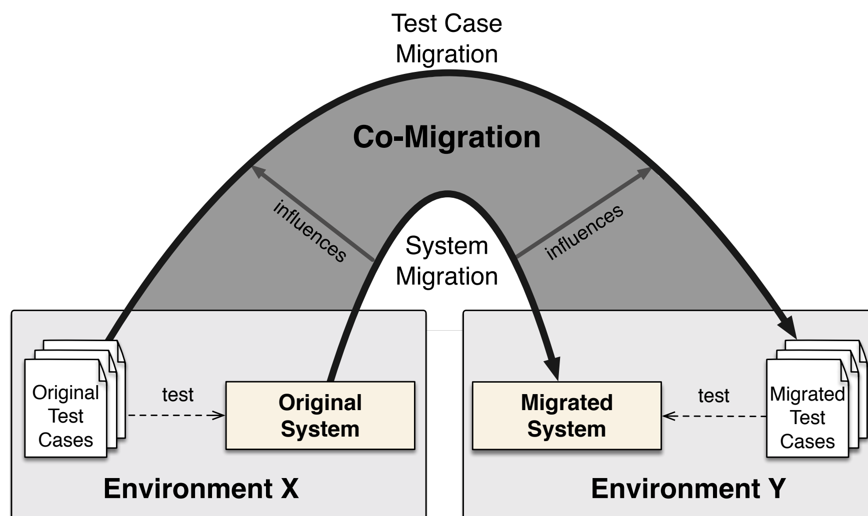


Figure 1.1 The co-migration problem: test case migration and system migration

of the impact analysis, as part of the *Change Propagation* step, the actual propagation of the changes to transform the test cases is performed. In the end, *Validation* is carried out to ensure that all test cases subjected to change have evolved consistently. These steps should be incorporated in the test case migration to provide a proper evolution of the test cases and thus, their proper migration to the target platform.

When performing a software migration, a transformation method is being enacted. Such a transformation method serves as a technical guideline and describes the activities necessary to perform, tools to be used, and roles to be involved to migrate a given system. In the case of software migration, the transformation methods are instances of the *reengineering horseshoe model* [KWC98] and it comes from the fact that software migration is a kind of reengineering. The reengineering methods, as shown in Figure 1.2, consist in general of the following consecutive phases *Reverse Engineering*, *Restructuring*, and *Forward Engineering* [CC90]. When performing the transformation, a given original system is being represented on different levels of abstraction in terms of different artifacts. When applying reverse engineering, representations on a higher level of abstraction are represented by applying parsing for example. These representations are then restructured so that they are suitable for the new environment. In the end, by applying forward engineering techniques, the high-level representations are concretized in the target environment by eventually generating executable source code. In general, the whole concept of reengineering is applicable to the domain of test case migration, thus enabling a basis to define test case transformation methods. However, so far, there is still no established reengineering horseshoe model for test cases which could be directly applied in test case migration.

During a software migration project, a transformation method firstly needs to be developed. This is a very important task as it influences the overall success of the migration project in terms of effectiveness and efficiency. As defined in [GFEK16], effectiveness relates to properties of the resulting migrated system, e.g., non-functional properties. Efficiency relates to the properties of the actual process applied to realize the transformation, e.g., the time required or the budget. Being efficient and effective at the same time means having a transformation method that minimizes the effort required while maximizing certain non-functional properties. To achieve this, the situational context of the migration project should be taken into consideration when a transformation method is being developed. The developed transformation method is then considered to be situation-specific. The situational context comprises different influence factors like characteristics of the original system, characteristics of the target environment, the goals of the stakeholders, etc. Concerning test case migration, the situational context gets even more complex as beside the influence factors of the system

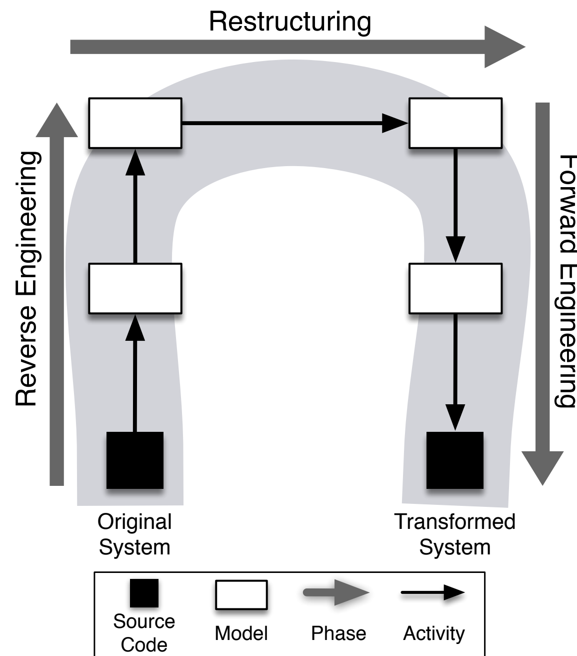


Figure 1.2 The reengineering horseshoe model [KWC98]

migration, influence factors from the test migration like characteristics of the original test cases or characteristics of the test target environment have to be considered as well.

Developing a situation-specific transformation method is an important and challenging task, as a transformation method not suitable for a particular situation may result in a migration project that is inefficient and ineffective thus raising the risk of failure. To minimize this risk, a common practice is to support and/or guide the development of transformation methods by a specific method engineering approach. Ideally, such an approach should provide a transformation method suitable, i.e., applicable in a given situation, by either providing successfully applied methods before or guiding the development of new ones. The applicability of a given method in different contexts mainly depends on the *degree of controlled flexibility*. *Flexibility* in this context means, to what extent a transformation method may be adapted to a specific situation, whereas *control* refers to the degree of guidance provided during the development of the method. As shown in [GFEK16], the modular construction of transformation methods provides the highest degree of controlled flexibility by assembling predefined method parts. Moreover, this approach also provides construction guidelines on how to perform the assembly itself. Providing an effective and efficient test case transformation method is also the main goal in test case migration. So, this implies the same requirement as in the software system migration, namely the consideration of the situational context. This is, however, a bit more complex task as the previously

discussed co-evolution aspect should be incorporated when identifying the situational context of both the system and test case migration. Only then, a developed test case migration method could migrate the test cases efficiently and effectively.

As we already mentioned about system migration, a crucial requirement that must be fulfilled, is that the original system and the migrated system are functionally equivalent. Similarly, test case migration is the process of transferring test cases to a new environment without changing their "functionality", i.e., without changing the expected behavior of the migrated system the test cases assert. As migrated test cases are used to validate system migration, validating test case migration is clearly very important. However, test case migration is in general far from trivial as several challenges need to be addressed [JGY16] and, due to the tight coupling of system migration and test migration, validating test case migration is especially challenging.

1.2 Problem Statement

Based on the discussion in the previous section, providing an end-to-end solution for test case migration is quite challenging. Starting even before the actual migration, a decision is necessary whether it is beneficial to perform a migration of the existing test cases. If yes, a proper test case migration method should be developed. As previously discussed, the development of the migration method is important and it should be guided. Furthermore, the method development should incorporate the co-evolution analysis to enable a proper migration of the test cases. Last but not least, it should be assured that one can have confidence in the migrated test cases when executing them against the migrated system. Based on this and the discussion in the previous section, in the following, we discuss the general challenges related to test case migration.

C1: How to make a systematic quality evaluation of test cases?

Reusing test cases is beneficial, only when the test cases have some value. Migrating test cases that are redundant or cover unused parts of the system or if they have low quality, e.g., if their structuredness or effectiveness is low, can result in migrated test cases that cannot properly validate the migrated system. The execution of the migrated test cases would not be efficient and in a worse case, it could lead to a false conclusion about the migrated system. Therefore, to counteract this problem, before doing anything with the test cases, their quality needs to be evaluated. For this purpose, relevant quality characteristics, e.g., effectiveness or structuredness need to be assessed. Such a quality assessment should provide clear indications on the quality aspects and on this basis, one can decide whether it is worth

investing time and effort in migrating the existing test cases. However, checking thousands of test cases against given quality criteria is quite challenging if no systematic approach is available.

C2: How to incorporate the co-evolution analysis in test case migration?

The migration of the test cases is coupled with the migration of the system they are testing. In other words, reusing test cases implies a co-migration together with the system. Therefore, the system changes need to be detected, and then their impact on the test cases analyzed, and if necessary propagated. This implies that the co-evolution process [MD08] has to be considered in the test case migration. However, this is far from trivial, because, first of all, the changes in the system have to be detected, analyzed, and understood. Then, their impact on the test cases has to be identified, so that the relevant changes could be propagated to the test cases when the test case transformation method is being enacted. As the system changes could be diverse, as different types of migration exist, e.g., language migration or architectural migration, providing a generic solution for the co-evolution could be quite challenging.

C3: How to transform the test cases in an automated way?

When migrating a large number of test cases, the test case migration should be ideally completely automated. The existing test cases may be structurally complex and as a result, a direct transformation would be hardly possible. Furthermore, the system migration may be performed on a higher level of abstraction thus requiring reverse engineering techniques for test cases. Also, the migrated test cases should be appropriate for the target environment, i.e., the test cases have to be adapted for the target environment, this requiring appropriate restructuring activities and tools that would ease and automate the transformation of the test cases. Last but not least, the generation of the test code for the migrated test cases is also a step that should be automated. All in all, the solution approach should support the typical reengineering activities like reverse engineering, restructuring, and forward engineering for test cases.

C4: How to provide a situation-specific test case transformation method?

As each migration project is performed in a specific migration scenario, a "one-size-fits-all" test case migration method is not a viable solution. That means, in different migration contexts, e.g., a language migration versus architectural migration, a different set of changes should be applied to the test cases. Whereas in the language migration, the test cases have to be transformed in the new language, in the architectural migration the test cases have to be modified structurally due to the structural changes of the system. Furthermore, the language migration imposes also a change of the testing framework, as for different languages different

test frameworks exist. This means the test cases have to be also transformed in that way so they can be executed in a specific target testing framework. Often, a language migration is coupled with an architectural migration, thus making the development of an appropriate transformation method for the test cases even more challenging. In general, the test case changes depend on the system changes as well as on the requirements imposed by the new test environment. Not considering the complete situational context may result in test case transformation methods that are inefficient and/or ineffective.

C5: How to validate a test case migration?

The main requirement in test case migration, similarly to system migration, is to transfer the test cases to a new environment without changing their "functionality". Regarding test case migration, this means without changing the expected behavior asserted by the test cases. As the migrated test cases are used as safeguards for the system migration, their correct migration is crucial. But, as the test cases change along with the system, it is challenging to validate their migration. Once migrated and executed, the test report may also contain some false positives and false negatives which could potentially lead to false conclusions about the correctness of the migrated system. Therefore, validating the migrated test cases is a crucial activity for establishing adequate trust in the test case migration and consequently in the migrated system as well.

C6: How to provide the different phases of the test case migration with appropriate tooling?

If each solution of the previously introduced challenges requires manual work, it would result in a lot of time and effort. Firstly, if the quality evaluation of the test cases is not supported by proper tooling which provides guidance up to some extent, it would require a lot of time only to create a quality plan. Then, the development and enactment of test transformation methods without proper tool support and guidance could lead to a quite complex manual work and consequently would require a lot of time. Also, the validation of the test cases, when not supported by tooling, would require a lot of manual work to ensure the success of the test case migration. All in all, to achieve a more effective and efficient test case transformation as well as better acceptance of the approach in practice, the different phases should be tool-supported.

In summary, all the challenges previously stated form the research question of this thesis:

How to enable an end-to-end migration approach for test cases that supports the evaluation of the test case quality, the construction of situation-specific automated co-migration methods, and the migration validation?

1.3 Solution Overview

In order to address the aforementioned challenges that led to the research question, we propose a Test Co-Migration (TeCoMi) framework which provides an end-to-end solution for test case co-migration. The basic idea, as shown in Figure 1.3, evolves around the double horseshoe reengineering model [KWC98] which we propose as a solution to the co-migration problem.

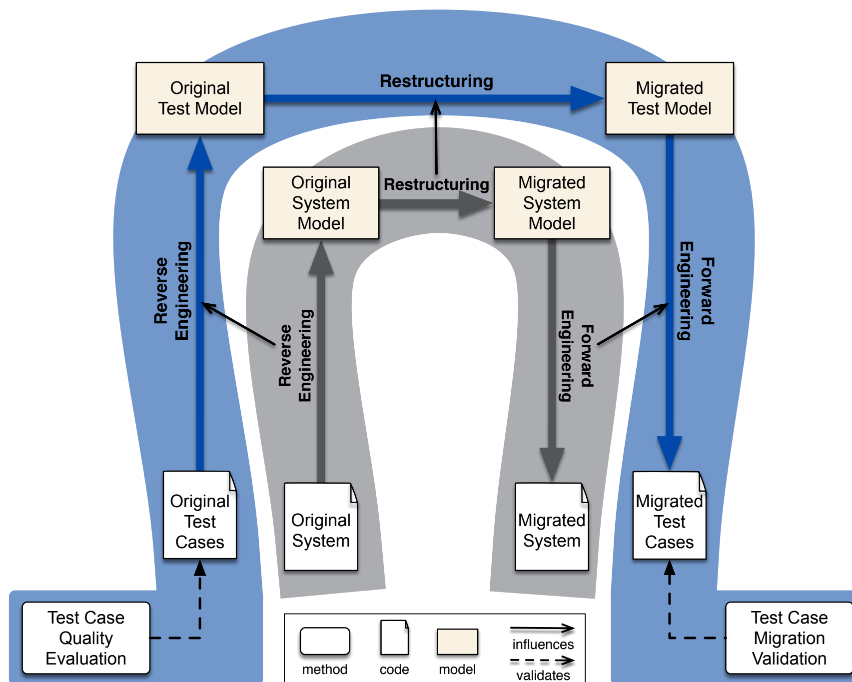


Figure 1.3 The double horseshoe reengineering model for co-migration of test cases

Motivated by the idea of model-driven software migration [FWE+12], the solution approach applies the Model-Driven Engineering (MDE) principles to the test case migration domain. Furthermore, it combines techniques from *Software Evolution* and *Situational Method Engineering (SME)* to address the co-evolution and situativity challenges, respectively. Thus, the resulting model-driven migration methods enable automated co-evolution of test cases for a specific migration context. Additionally, as shown in the bottom left-hand corner of Figure 1.3, *Test Case Quality Evaluation* provides a means to check the quality of the existing test cases. Similarly, in the bottom right-side corner of Figure 1.3, *Test Case Migration Validation* provides a means to validate the migrated test cases.

Having the basic solution idea introduced, Figure 1.4 shows how the test case migration framework supports the three general migration phases: *Pre-Migration Phase*, *Migration*

Phase, and *Post-Migration Phase*. The labels from **C1** to **C5** represent the previously introduced challenges and show which part of the solution addresses which challenge. Tooling was also developed to support the three phases, but it is not explicitly shown in Figure 1.4 and for this reason, **C6** is not depicted in the figure.

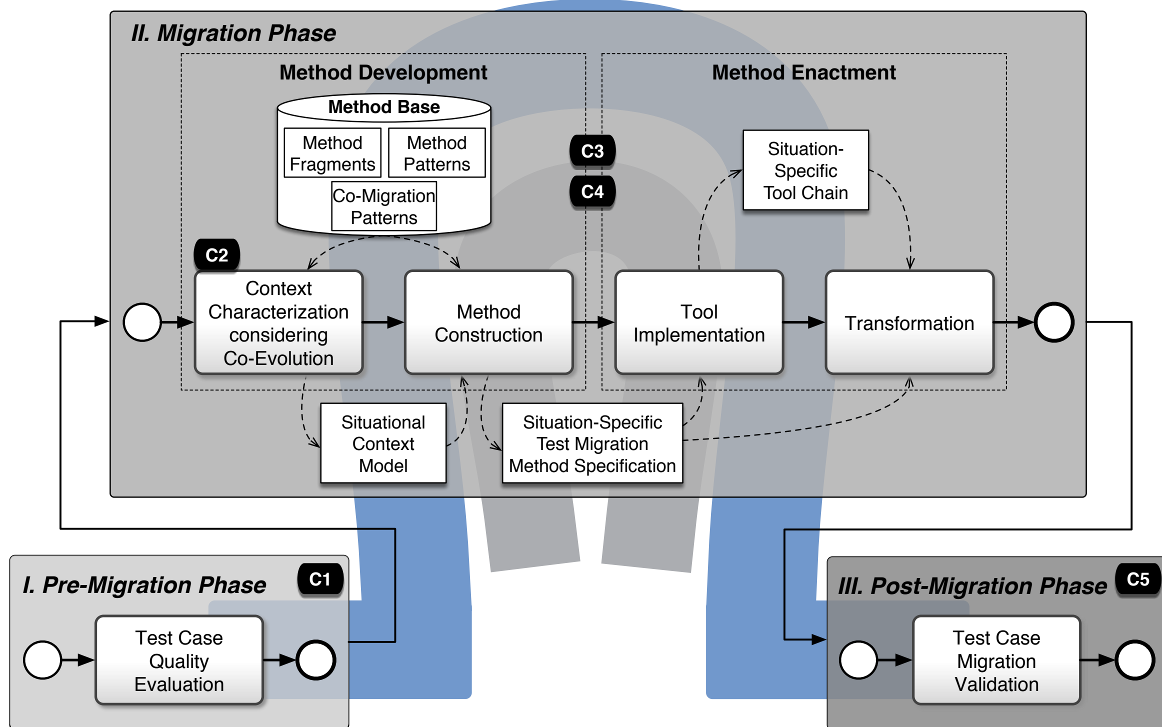


Figure 1.4 The general phases with the corresponding activities of our approach

During the *Pre-Migration Phase*, *Test Case Quality Evaluation* is performed to evaluate the quality of the existing test cases. Our approach enables a systematic tool-supported development of test case quality plans [JNES18] which consider the context information and integrate a standardized quality model, namely the ISO/IEC 25010 [ISO11a]. A quality plan serves as a guideline for the quality evaluation of test cases and emphasizes the context of use of test cases as a major factor of influence for the whole quality evaluation. After a quality evaluation is performed, an indication of the quality of the test cases is obtained. Based on this quality results a decision is made whether the existing test cases should be migrated and if yes to what extent, or not.

The *Migration Phase*, which is the main phase, has activities that are split into the two main disciplines: *Method Development* and *Method Enactment*. It represents the part of the framework which is actually the implementation of the idea of the double horseshoe model.

Following the basic idea of the existing *Method Engineering Framework for Situation-Specific Software Transformation Methods (MEFiSTo)* [GFBEK16], which is further based on *Situational Method Engineering (SME)* [HSRÅR14], during *Method Development* and *Method Enactment* respectively, a situation-specific test case migration method is developed and enacted.

The general idea is to use predefined method blocks, called *Method Fragments* which are stored in a *Method Base*. A method fragment is an atomic building block of a migration method, i.e., an activity, artifact, or tool [HSRÅR14]. As we follow the idea of model-driven software migration [FWE+12], our method fragments belong to one of the following reengineering processes: reverse engineering, restructuring, or forward engineering [CC90]. *Method Patterns*, on the other hand, represent a proven migration strategy and show how different migration fragments could be combined to realize this strategy. Each pattern has a set of characteristics that express its suitability to a certain situation. Besides the *Method Patterns*, the repository also contains a set of co-migration method patterns. They express the dependency between the system and the test case migration patterns, thus supporting the co-evolution analysis.

Having the method fragments and method patterns, guidance is needed on how to create a test migration method for a specific context. This is done by a method engineering process, which guides the development and the enactment of the context-specific test migration method. The method engineering process begins with *Context Characterization considering Co-Evolution* which takes the original test code as the main input. During this activity, both the system migration and testing contexts are being characterized, and additionally, co-evolution analysis is performed. Namely, the two previously introduced co-evolution activities, change detection, and impact analysis are applied. The overall outcome of the analysis defines what kinds of modifications of the test cases are necessary in terms of test case changes.

Then, based on the previously collected context information in terms of *Situational Context Model*, an appropriate method pattern is selected and configured. The result of this activity, *Situation-Specific Test Migration Method Specification*, is used as a base input for the *Tool Implementation*, where for every specified activity that shall be performed (semi-)automatically, an appropriate tool is implemented. The migration phase ends with the *Transformation* step, where based on of the *Situation-Specific Test Migration Method Specification* and the *Situation-Specific Tool Chain* the actual transformation of the test cases is performed.

Last but not least, in the *Post-Migration Phase*, the migrated test cases are validated. With the help of a novel validation method, it is checked whether the test cases are migrated

without changing their behavior, i.e., without changing what they actually test. As the main goal of the migration validation is to identify false positives and false negatives among the migrated test cases, the validation method relies on mutation analysis.

1.4 Publication Overview

In the course of this PhD thesis, a number of publications were created and published on different workshops and conferences. As shown in Figure 1.5, the classification is made according to the phase of the solution approach a given publication is related to.

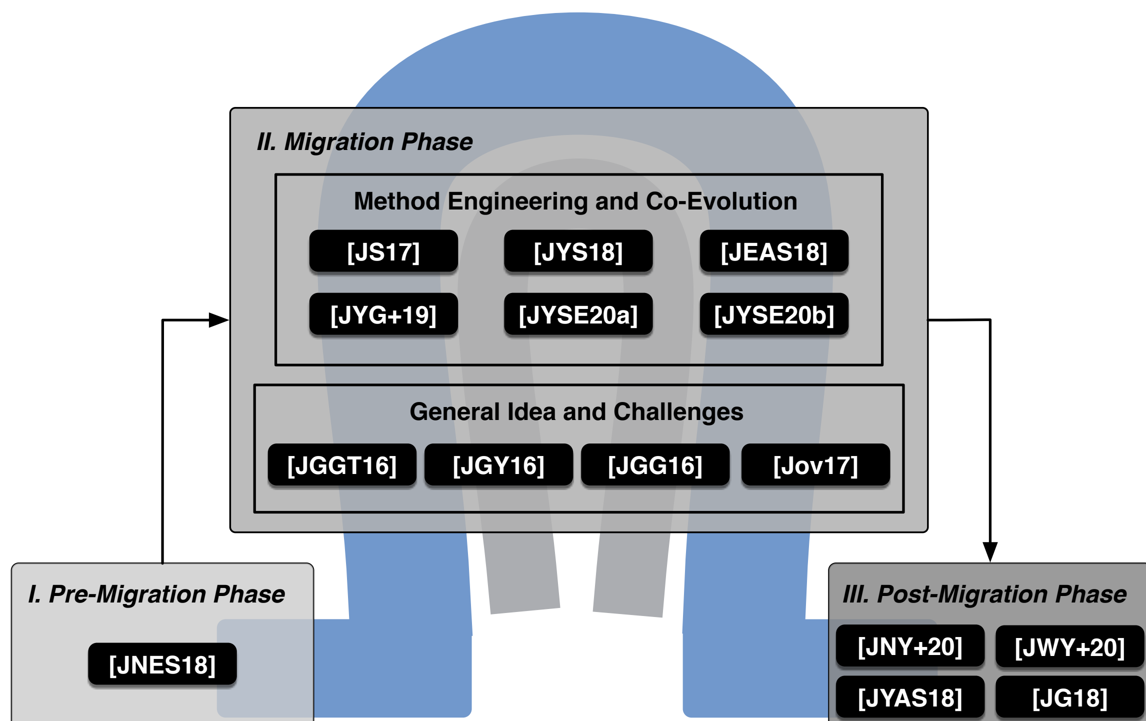


Figure 1.5 Overview of the publications related to this thesis

Regarding the *Pre-Migration Phase*, the test case quality evaluation is addressed by [JNES18], which presents a method for test case quality assessment called Test Case Quality Plans (TCQP). The method enables the creation of appropriate test case quality plans which can be executed against a particular test case suite and provide insight on the quality of the test cases.

The main idea about applying the model-driven engineering principles in the field of test case migration was presented in [JGGT16]. More specifically, in this paper, the problem domain and the challenges in test case reengineering as part of software migration are discussed.

Providing a model-driven method for reusing test cases in software migration projects was firstly presented in [JGY16]. Based on a set of challenges, a migration method represented as a test case reengineering horseshoe model was introduced. This reengineering horseshoe model follows the already existing system reengineering horseshoe model thus representing the co-evolution of the test cases.

As our method is model-driven, in [JGG16] we discussed its close relation to model-based testing [PP04]. In general, there are several model-based testing scenarios, depending on the source of the test model. Accordingly, our method can be characterized as a combination of the existing scenarios.

Providing a one-size-fits-all migration method is not possible, and for this reason, in [JS17, Jov17], we discussed the initial idea about providing a framework for the construction of situation-specific test case migration methods which support co-evolution. In [JYS18], we provided a reference migration method in terms of a test case reengineering model as a constituting part of the framework. A detailed description of the test horseshoe model, covering all basic artifacts, activities, and tools needed for test method construction is presented in [JEAS18]. The co-evolution analysis which is combined with the concept modeling is presented in [JYSE20a]. Furthermore, the co-migration patterns are presented in [JYSE20b]. The complete solution on the modular construction of context-specific test case migration methods is presented in [JYG+19].

A basic idea about test case migration validation applied in the *Post-Migration Phase* was initially presented in [JYAS18, JG18]. The method, based on mutation analysis, provides different scenarios and guidelines to identify false positives and false negatives among the migrated test cases. The complete validation method based on mutation analysis was presented in [JNY+20]. The supporting model-driven mutation framework was presented in [JWY+20].

1.5 Structure of the Thesis

An overview of the structure of this thesis is shown in Figure 1.6.

In Chapter 2, we introduce the needed foundation for different research areas related to our work, particularly Model-Driven Engineering, Software Reengineering, Software Evolution, and Method Engineering.

In Chapter 3, we present a general overview of the related work. Firstly, we describe in detail the requirements which should be fulfilled by a solution concept. Then, we identify and classify the existing approaches in four different areas: Model-Based/Model-Driven Testing, Test Case Reengineering, Test Case Evolution, and Software Migration.

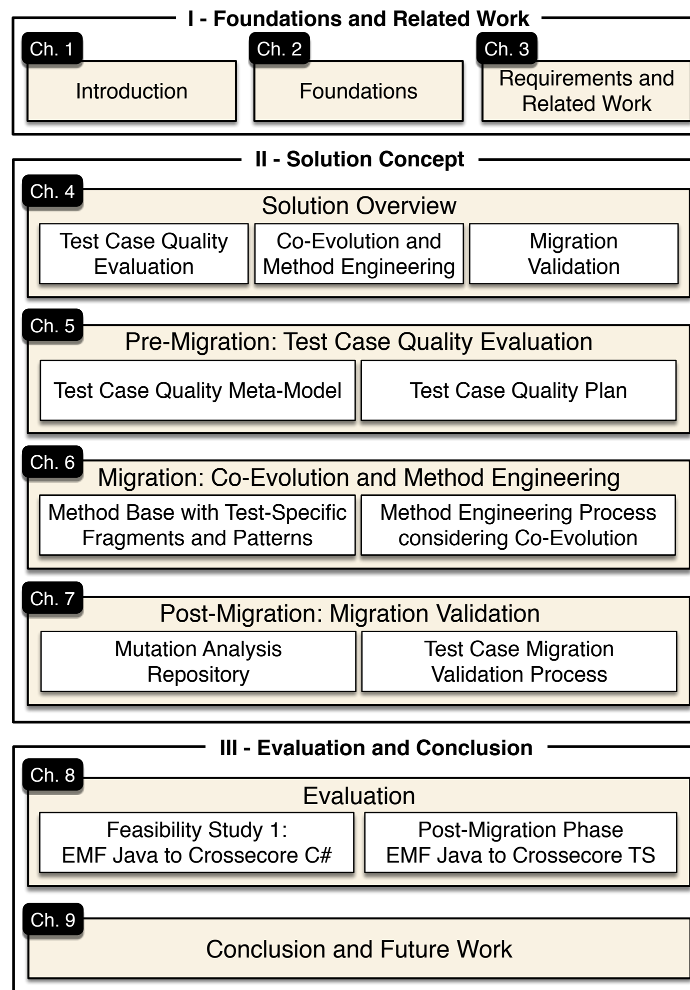


Figure 1.6 Overview of the thesis structure

In Chapter 4, we present our general solution idea by introducing the three main phases supported by our framework: *Pre-Migration*, *Migration*, and *Post-Migration Phase*. We briefly describe first how a quality assessment is performed. Then, how by considering co-evolution analysis, a situation-specific test migration method can be constructed. Finally, we present a validation method for test case migration. We also define a set of evaluation criteria which fulfillment is discussed by two feasibility studies.

In Chapter 5, we present the first phase of our solution approach, the Pre-Migration Phase, which addresses the quality evaluation of test cases. Therefore, we present our method for quality evaluation of test cases called *Test Case Quality Plans (TCQP)* and we discuss its main constituting parts, process, and the metamodel. The process specifies the steps needed to create a quality plan and the metamodel how such a test case plan could look like.

In Chapter 6, we present the main phase of the solution approach, namely the Migration Phase. We firstly introduce the method base by introducing its constituents, namely the test-specific method fragments and the method patterns as well as the co-migration patterns. After this, we introduce the method engineering process for the development and enactment of test transformation methods. The process also considers, the test case co-evolution analysis, with the two main activities: change detection and change impact analysis.

In Chapter 7, we present the Post-Migration Phase, i.e., the migration validation. We firstly introduce the mutation analysis repository with its main constituents, namely mutation analysis scenarios, mutation patterns, and mutation operators. Thereafter, we introduce the test case migration validation process that relies on the mutation analysis repository.

In Chapter 8, we firstly state a set of evaluation questions whose fulfillment we aim to discuss by feasibility studies. Thereafter, we present the feasibility studies that we have performed in practice to demonstrate the applicability of the TeCoMi framework. We performed two feasibility studies in which we transformed real-world test cases to different target platforms. Based on the findings, we discuss the previously introduced evaluation questions.

In Chapter 9, we conclude the thesis with a summary of our main contributions and we give an outlook on possible future work.

Chapter 2

Foundations

In this chapter, an overview of foundations is given that are relevant for this thesis. As shown in Figure 2.1, we grouped the foundations into five main areas that form the basis for the development of the solution concept. Firstly, we introduce in Section 2.1 the general concepts of *Software Reengineering*. In Section 2.2, we introduce *Software Co-Evolution*. Further, in Section 2.3, we introduce the essential foundation of *Method Engineering*. In Section 2.4, we introduce the fundamentals of *Software Testing*. Finally, in Section 2.5, we give an overview of *Used Technologies*.

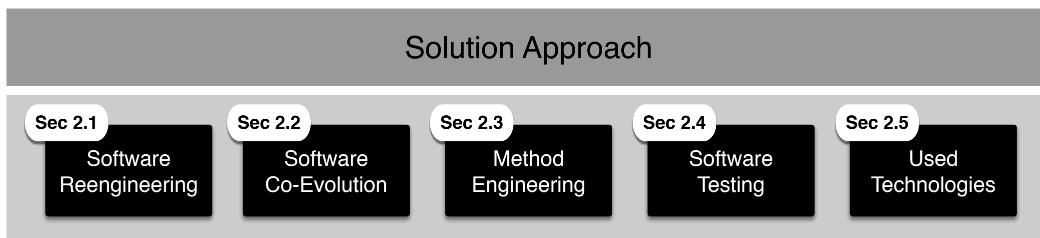


Figure 2.1 Research areas relevant to this thesis

2.1 Software Reengineering

In this section, we introduce the area of software reengineering as our approach employs different reengineering techniques. According to [CC90], software reengineering is defined as "examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form". We firstly describe software migration, as a manifestation of software reengineering and we also discuss the transformation methods in the software migration context. Thereafter, we introduce concept modeling, a technique that

can be used to represent the functionality of a software system as well as the functionality of test cases.

2.1.1 Software Migration

Software migration is a well-established method for transferring software systems into new environments while keeping the data and the functionality of the system [BLWG99]. As this definition suggests, software migration is related to two main characteristics: *environment change* and *functionality and data preservation*. An *environment change* can be a change regarding the architecture and/or the programming language. Functionality and data preservation suggest that the functionality of the system as well as the data should stay unchanged after the migration is performed.

Model-Driven Engineering (MDE) has been established to deal with the increasing complexity of development, maintenance, and evolution of nowadays software systems. In order to achieve this, it relies on models and model transformations [BCW12]. Model-Driven Architecture (MDA), proposed by the Object Management Group (OMG), defines several software models on different levels of abstraction, thus clearly separating the business complexity from the implementation details [OMG14]. MDA defines three levels of abstraction: *Computational-Independent Model (CIM)*, *Platform-Independent Model (PIM)*, and *Platform-Specific Model (PSM)*. The *Computational-Independent Model (CIM)* focuses on the context and the requirements of the system. The main focus of the *Platform-Independent Model (PIM)* is on the operational capabilities of the system without consideration of a specific technology. Finally, the *Platform-Specific Model (PSM)* focuses on a specific platform.

In case when MDA principles are followed, migration approaches are known as model-driven software migration (MDSD) [FWE$^+$12]. In general, software reengineering consists of three consecutive phases: *Reverse Engineering*, *Restructuring*, and *Forward Engineering* [KWC98]. *Reverse Engineering*, according to [CC90], is the process of analyzing a subject system to create representations of the system in another form or on a higher level of abstraction. *Restructuring*, as defined by [CC90], is "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)". *Forward Engineering* is defined as "the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system" [CC90]. As already mentioned, we define software migration as a manifestation of software reengineering:

Notation 1 (Software Migration)**.** *Software migration is a kind of software reengineering dealing with the transition of an existing system to a new environment while retaining the functionality and the data of the system [BLWG99].*

Migration of existing systems, i.e., transferring them into a new environment is a complex endeavor. For this reason, commonly, a migration project is established and a software migration method gets enacted. Such a method guides the complete endeavor of software migration. We define a software migration method as follows:

Notation 2 (Software Migration Method)**.** *A software migration method is a method that is used to guide a software migration endeavor.*

During a software migration, a transformation of the existing system on a technical level is performed. How to perform this technical alteration is defined by the software transformation method which is actually an instance of the horseshoe model shown in Figure 1.2. We define a software transformation method as follows:

Notation 3 (Software Transformation Method)**.** *A software transformation method is an instance of the horseshoe model and used to guide the technical transition of an existing system into a new environment during a software migration endeavor.*

2.1.2 Concept Modeling

Concept modeling is a technique to represent a software system by a set of concepts [KNE92]. Thereby, each concept belongs to a particular level of abstraction and refers to a specific part of the software system‘s source code. The general idea of concept modeling is illustrated in Figure 2.2. The concepts are split into two different groups, namely language concepts and abstract concepts. Language concepts directly correspond to syntactic entities of the programming language, like variables, declarations, statements, etc. [KNE92]. The abstract concepts, on the contrary, represent language-independent ideas of computation and problem-solving methods [KNE92]. Abstract concepts are further classified into architectural and programming concepts.

Notation 4 (Language Concept)**.** *A language concept is a syntactic entity of a programming language.*

Abstract concepts reside on higher levels of abstraction. They represent a general idea of computation or problem-solving principle [KNE92]. Thereby, they are not associated with a specific programming language but represent language-independent principles. In this thesis, we define an abstract concept as follows:

Notation 5 (Abstract Concept)**.** *An abstract concept represents a language-independent idea of computation or problem-solving principle.*

Abstract concepts can be further classified into programming concepts and architectural concepts. A programming concept represents general programming strategies, data structures or algorithms [KNE92]. In this thesis, we define a programming concept as follows:

Notation 6 (Programming Concept)**.** *A programming concept is an abstract concept and represents general programming strategies, data structures or algorithms.*

An architectural concept represents components or interfaces that reside within a software system [KNE92]. In contrast to programming concepts, they do not represent some functionality of the system but focus on describing its overall structure. In this thesis, we define an architectural concept as follows:

Notation 7 (Architectural Concept)**.** *A programming concept is an abstract concept and represents general programming strategies, data structures or algorithms.*

The architectural concepts are associated with interfaces or components whereas the programming concepts represent a general coding strategy, data structure or algorithm. Concepts can be related to each other by *is-a* relation, to express a hierarchy between different concepts, and *consists-of* relation to express dependencies between concepts. In [Gri16], when applying the idea of concept modeling to software modernization, three classes of concepts are distinguished, source concepts, target system concepts, and shared system concepts. Regarding the original system, the language concepts are determined by the language elements that are already used, whereas target system concepts are those that will be used after the transformation. Finally, a shared concept is an abstract concept of the original system that can be realized in the target environment. All in all, the concept model

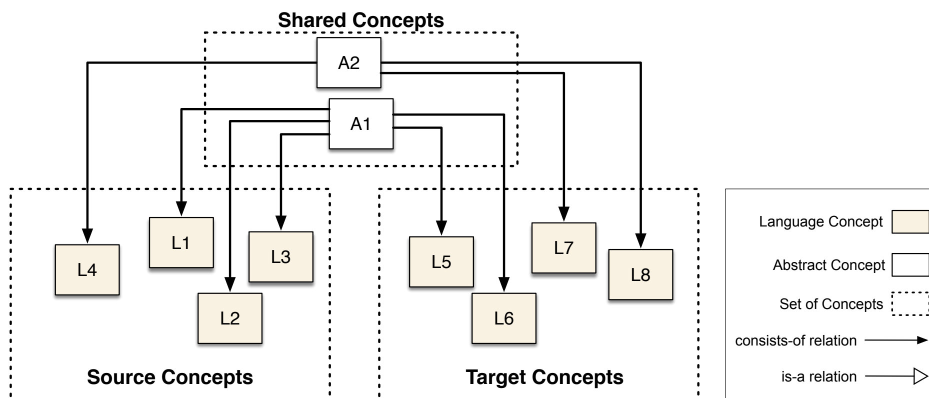


Figure 2.2 Representation of a software system as a set of concepts (based on [Gri16])

is defined as a directed, acyclic and connected graph. The nodes represent the concepts, whereas the edges between them represent *is-a* or *consists-of* relations.

We have seen that the software reengineering techniques like software migration and concept modeling deal with the transformation of a given software artifact. Very often, when some artifact changes, the depending artifacts have to be also correspondingly changed. Therefore, the following section deals with the topic of *Software Co-Evolution* to describe the process of analyzing changes, identify their impact, and propagating them if needed.

2.2 Software Co-Evolution

Co-migration of test cases includes their co-evolution. Therefore, in this section, we introduce the basic co-evolution process and we subsequently explain each step of the process. Modern software systems are rapidly evolving due to continuously changing technological and business requirements [MD08]. These systems must often undergo changes to fulfill these changed requirements. The number of new requirements and maintenance requests often grows faster than software developer's abilities to implement them [MWD$^+$05]. Changing some requirements affects other related and dependent artifacts such as design specifications, source code, test cases, and documentation. Therefore, changes to one artifact may not be reflected immediately to all other interdependent artifacts, which results in inconsistency among dependent artifacts. A key process of managing consistency between software artifacts is co-evolution and it is defined as follows:

Notation 8 (Software Co-Evolution)**.** *Software co-evolution refers to two or more artifacts evolving alongside each other, such that there is a relationship between the two that must be maintained [MD08].*

We distinguish between two types of artifacts: evolving artifact and depending artifact. We define them as follows:

Notation 9 (Evolving Artifact)**.** *The artifact that undergoes changes is called evolving artifact.*

Notation 10 (Depending Artifact)**.** *On the other hand, those artifacts which are affected by changes to the evolving artifact are known as depending artifacts.*

For example, when a change (e.g., adding a feature) is applied to a system, not only its code should evolve (evolving artifact), but also its design models (depending artifact) should evolve. The common process proposed by [MD08], as shown in Figure 2.3, consists

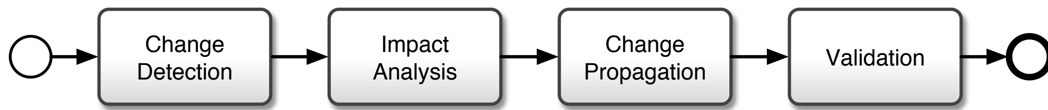


Figure 2.3 The staged process model for co-evolution [MD08]

of the following activities: *Change Detection*, *Impact Analysis*, *Change Propagation*, and *Validation*.

Firstly, in *Change Detection*, all changed parts of the system are identified. Having these changes identified, in the next step called *Impact Analysis*, all affected artifacts are identified and an estimation of the effort required to accomplish the change together with involved risks is determined. Then, based on the results of the impact analysis, as part of the *Change Propagation* step, the actual propagation of the changes to transform the dependent artifact is performed. In the end, *Validation* is carried out to ensure that all dependent artifacts subjected to change have evolved consistently.

2.2.1 Change Detection

In the first step of the co-evolution process, it should be determined how and which changes can be detected. According to the kind of detection, there are three main approach types: *state-based*, *operation-based*, *manual*, and *hybrid*. *State-based* approaches match the original version and the evolved version of the artifact in order to derive the applied changes. On the contrary, the *operation-based* approaches rely on the recorded or logged applied changes. In the manual approaches, one needs to define the changes by hand. As a last option *hybrid* approaches which combine the previously mentioned types can be applied. The granularity of change is another characteristic of a change detection approach and according to it two different types of approaches: those that detect changes as atomic units, i.e., not interconnected single changes, or as composite units, i.e., a sequence of changes which are semantically connected.

2.2.2 Impact Analysis

The identified changes may have an impact on the depending artifacts and this should be determined as part of the impact analysis step. Firstly, a detailed analysis is necessary on the depending artifacts, to reveal the potential effects on the artifacts. For example, when analyzing a relation between source code and tests, the relationship "is-tested-by" can be utilized to identify the test cases that could be possibly affected. The outcome of the analysis should be usable for user intervention, i.e., it should clearly indicate the affected artifacts and

the type of impact on them. The impacts may be detected on a different level of granularity, ranging from fine-grained to coarse-grained level. In the case of models, on a fine-grained level is for example when the features are impacted, whereas on a coarse-grained level is for example when a whole package is impacted.

2.2.3 Change Propagation

Based on the outcome of the impact analysis, it should be determined how the identified changes should be propagated to the depending artifacts and how the consistency between the artifacts may be ensured. There are different strategies when it comes to the actual propagation of the changes, the different approaches may offer existing predefined strategies which could be further customized, e.g., parameterized, adapted or completely overwritten. The propagation can be either done automatically, without any intervention of the user, or semi-automatic, i.e., with user intervention. As the last point comes the consistency between the artifacts, i.e., the intra-artifact consistency. Namely, in the case of code and test case co-evolution, when a set of test cases are impacted by a given code change, then this change has to be propagated to all of the impacted test cases.

2.2.4 Validation

In the final step, also an optional step, the consistency between the artifacts has to be checked. Namely, after the propagation of the changes, it must be ensured that artifacts have co-evolved in a consistent way. Two types of validation are possible: a syntactical and a semantical validation. A syntactical validation may be performed by using consistency rules and a semantical validation by applying regression testing. The validation may be performed on a given instance of an artifact, or all instances of a given kind or all instances of all kinds.

Software co-evolution defines the exact steps on how to keep the artifacts synchronized when changes happen. Combined with the software reengineering techniques, this enables the transformation of multiple artifacts, e.g., a system and the corresponding test cases, in an automated way. How to exactly perform a transformation is described by the means of a transformation method. Therefore, in the following section, we introduce *Method Engineering* which deals with the topic of development and adaptation of transformation methods.

2.3 Method Engineering

Migrating test cases means the enactment of a transformation method which would transform the test cases for the target environment. Therefore, in this section, we introduce foundations in the area of method engineering which is the discipline to systematically develop or adapt methods [Bri96]. A method guides a complex software engineering endeavor, like the development of a software system or its transformation. This endeavor is guided by the method by specifying the activities to enact, artifacts to generate, tools to use or roles to involve [ES10]. According to [ES10, Gri16], a method is defined as follows:

Notation 11 (Method)**.** *A method is a description of how to systematically perform an endeavor. This comprises a process and its contained activities, artifacts, roles, tools, and relationships between these elements on varying levels of granularity.*

In the following, we describe Situational Method Engineering (SME) which is a specific manifestation of method engineering. Subsequently, we introduce two concrete SME approaches that form the base for our approach.

2.3.1 Situational Method Engineering

Situational Method Engineering (SME) is a kind of method engineering which encompasses all aspects of creating a method for a specific situation [HSRÅR14]. During the development of the method, the SME approaches take into consideration the situational context in which a method will be applied. As a result, the created method is situation-specific as it can be adapted to the context. An SME approach can be realized in various ways depending on the degree of *controlled flexibility* which is the degree of freedom given during the development of a method to adapt it to the given situation. *Flexibility* refers to the degree of freedom given during the method's development to adapt the method to the given situation. In order to sustain the quality of the resulting method, the development additionally needs to be controlled. Therefrom, *control* refers to the degree of guidance given during the method's development. With such guidance, one can ensure the result of the development, e.g., the correctness or quality of the method.

In this thesis, we focus on the class of approaches with the highest level of controlled flexibility by enabling a modular construction of situation-specific methods. Those approaches usually define two main constituents, namely the method base and the method engineering process. A method base constitutes a repository that contains reusable building blocks of methods whereas a method engineering process is defined to systematically construct a method. According to [HSRÅR14], the building blocks of methods are called method parts

and common examples are method fragments, method chunks or method components. On the one hand, a method fragment can be seen as an atomic building block of a method. On the other hand chunks as well as components aggregate multiple fragments. The focus in this thesis is on the method fragments which we define as follows:

Notation 12 (Method Fragment)**.** *A method fragment is a reusable, atomic building block of a method, i.e., a single activity, artifact, role or tool [HSRÅR14].*

However, developing a complete method by solely using method fragments is a cumbersome task, as methods can become large in practice. One way to address this problem is to use larger method parts than method fragments. This increases the efficiency of the method development as fewer elements of the method base need to be considered [HSGPR08]. The solution concept of this thesis follows another way by using method patterns. A pattern, in general, is associated with a reoccurring problem in a certain context [AIS77]. For the associated problem, it describes the core of a solution. A method pattern transfers this idea to the field of method engineering [FBLE13]. Each pattern is associated with a problem that shall be addressed by enacting a method. The solution to the problem is encoded by the pattern in the form of construction guidelines for the method [RP96]. As method patterns are an essential part of this thesis, we define them as follows:

Notation 13 (Method Pattern)**.** *A method pattern is associated with a problem that shall be addressed by enacting a method. It encodes the solution in the form of construction guidelines for a method, i.e., it specifies which method fragments to use and how to assemble them [Gri16].*

2.3.2 Project-Specific Software Engineering Methods

The MESP framework (Method Engineering with Method Services and Method Patterns) is a solution for software engineering method management.

It consists of two main types of method building blocks: method services and method patterns. A method service is a reusable, compositional, interoperable, and executable unit of a method based on the service-oriented paradigm. A method pattern provides a means to capture abstract orderings of activities as guidance for the project method engineer. As shown in Figure 2.4, the method engineering process comprises three main layers: *Method Content Definition*, *Method Tailoring*, and *Method Enactment*. For the tasks on each layer, the framework defines a responsible role based on the software engineering method management hierarchy as each layer requires a different level of knowledge and experience.

First of all, based on lessons learned or a description of methods on literature, a senior method engineer extracts reusable method content. On this basis, the basic method and

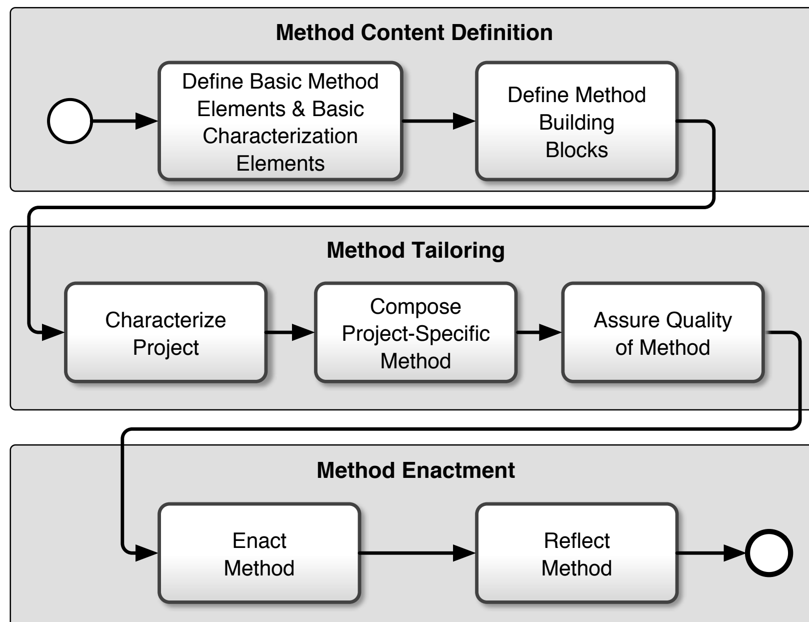


Figure 2.4 Overview of the MESP framework [FB16]

characterization elements are modeled (*Define Basic Method Elements & Basic Characterization Elements*). Consequently, as part of the *Define Method Building Blocks*, the actual method building blocks are defined, and they can be used to create situation-specific software engineering method models.

At the *Method Tailoring* layer, the project method engineers firstly characterize the project in order to ease the finding of suitable building blocks for the project. Having the suitable method building blocks for their project chosen, the building blocks are composed to a tailored method model. As last task of the tailoring layer, quality assurance of the tailored method is performed in order to check for some quality issues like missing building blocks, contradictions in control-flow and data-flow, etc.

As part of the *Method Enactment* layer, the project team firstly creates the software for their project by following the previously tailored method. The composed method model is then executed with a process engine whose role is to coordinate the activities of the project team members. Furthermore, it should provide guidance on the pending tasks so that the method is enacted as prescribed. Finally, with the *Reflect Method* task, the project team collects feedback about the method enactment. This knowledge can be used by the senior method engineer in improving method building blocks.

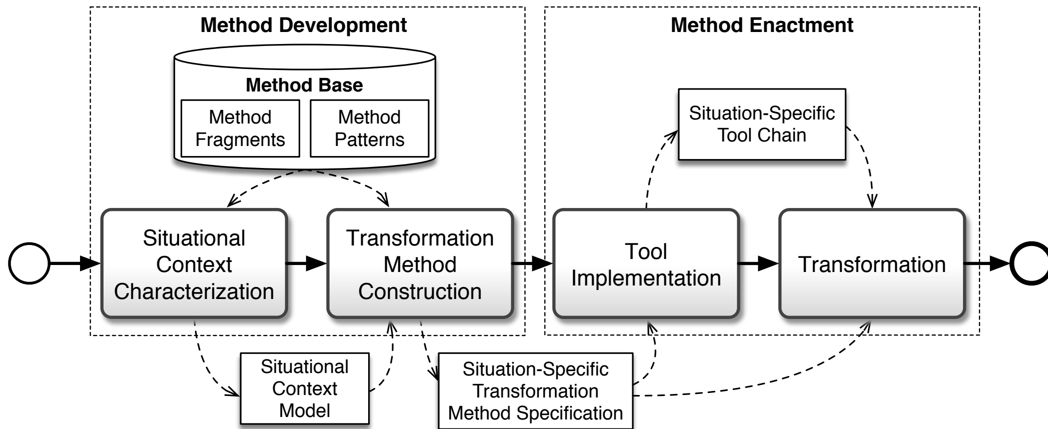


Figure 2.5 Overview of the MEFiSTo framework [Gri16]

2.3.3 Concept-Based Engineering of Situation-Specific Methods

The MEFiSTo framework (Method Engineering Framework for Situation-Specific Software Transformation Methods) [Gri16] is a method engineering framework that enables the modular construction of situation-specific software transformation methods.

As can be seen in Figure 2.5, the framework consists of a method base and a method engineering process. The method base provides the basic building blocks for assembling transformation methods and it consists of *Method Fragments* and *Method Patterns*. The method engineering process provides guidance on the development and enactment of situation-specific methods. Consequently, as shown in [Gri16], the activities of the method engineering process are separated into the two main disciplines: *Method Development* and *Method Enactment*. To develop a situation-specific transformation method, activities of the former discipline are performed. By performing activities of the latter discipline, the developed method actually transforms the original system.

The purpose of the *Situational Context Characterization* activity is to systematically identify the situational context, e.g., characteristics of the original and the migrated system. The outcome of this activity is obtained in terms of a *Situational Context Model*. The identified situational context serves as input for the next activity, namely the *Transformation Method Construction*, which guides the endeavor of constructing the transformation method. This includes selecting suitable method patterns and customizing method fragments. For those activities that shall either be performed automatically or semi-automatically, during *Tool Implementation*, a corresponding tool as part of an integrated toolchain is implemented.

Once the transformation method and the required tools have been developed, the actual transformation of the original system needs to take place as part of the *Transformation* activity.

So far, we have seen different methods and techniques on how to develop methods (*Method Engineering*) that can transform (*Software Reengineering*) multiple artifacts while keeping them consistent (*Software Co-Evolution*). As we intend to do this in the domain of test cases, in the following section, we give an overview of the area of software testing.

2.4 Software Testing

In this section, we introduce foundations in the area of software testing as we deal with the migration of the central artifact in software testing, namely the test cases. Software testing is a well-known method for asserting, among others, whether a software system provides a required functionality [SL09]. Different testing techniques on different levels of testing like unit, integration or system testing can be applied to assert a system's functionality. A central artifact in software testing is a test case which is a construct that consists of input data, actions, i.e., test steps, and expected result. Therefore, in a process called test case design, based on previously defined functional requirements, test cases are created and then executed against the migrated system. When executed, a test case produces an actual result which is then compared to the specified expected result. In case they match, i.e., a test case results in a positive overall outcome, the functionality being tested is said to be validated.

In the following, we introduce Model-Based Testing, a software testing methodology which aims at the automation of creation and execution of test cases. Then, we introduce Mutation Testing which is used as a technique to check the quality of the test cases. Lastly, we introduce quality models in general as well as test case quality models.

2.4.1 Model-based Testing

Model-based Testing (MBT) [PP04] is a software testing methodology that relies on using (semi-)formal models that encode the expected behavior of the system under test to derive test artifacts like test cases, test data or test configuration. A testing process that is MBT-based addresses not only the creation of test cases, rather it involves another starting with the definition of test models and ending up with the analysis of the test coverage after the tests are executed. According to [PP04] (shown in Figure 2.6), a typical MBT-based testing process involves the following activities:

1. Defining a test model - a test model gets defined for example by a tester based on the requirements or the system specification

2. Defining a test case specification addressing selection criteria

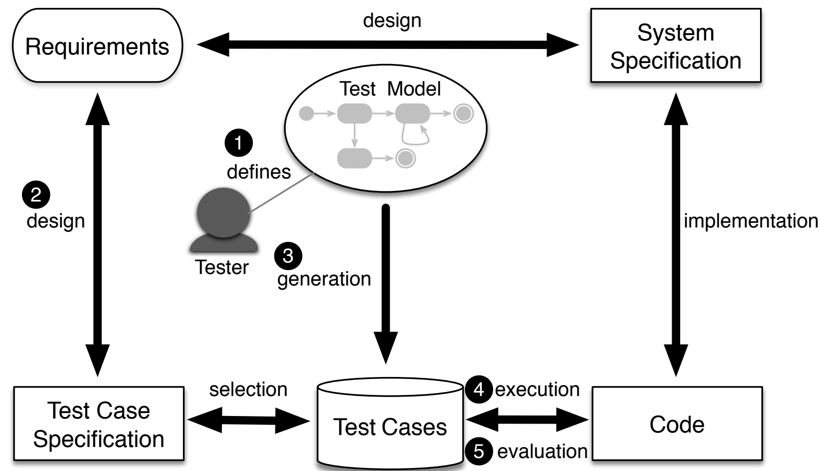


Figure 2.6 The general MBT approach [PP04]

3. Generation of test cases from a test model with respect to test selection criteria
4. Execution of test cases - the generated test cases are then executed against the system
5. Evaluation of test results - the results of the execution like the success or the test coverage are analyzed

Regarding the origin of the models, six different scenarios have been identified so far [PP04, GMS10]. In the context of test case migration, the most interesting scenario is *Models from Test Cases* where the test models are reverse engineered from the existing test cases.

This scenario assumes that an existing set of test cases contains relevant information about the system under test as well as about the test inputs and expected results. The existing test cases are basically reused and by using reverse engineering techniques, the test models are automatically derived from these test cases. This scenario is described as useful in the case of migration from classical to model-based testing [GMS10]. After applying reverse engineering, the obtained test models could be used for the generation of new test cases.

Model-Driven Testing (MDT) [HL03, EGL06] is a type of model-based testing that follows the Model-driven Engineering principles, i.e., the test cases are automatically generated from a test model using model transformations.

2.4.2 Testing Languages and Testing Frameworks

UML Testing Profile [OMG13b] is a language standardized by OMG which supports testing on model level. It can be divided into four main parts: *Test Architecture*, *Test Behavior*, *Test*

Data, and *Test Management*. *Test Architecture* is used for the specification of the structural aspects of the test environment and the corresponding test configuration. *Test Behavior* specifies the stimulation and observation of the system under test (SUT) by using any kind of UML behavior diagram. Using *Test Data* one can specify the pre- and post-conditions as well as the input and expected output of the SUT. Last but not least, using *Test Management* one can manage, for example, the test planning, scheduling or execution of test specifications.

Test Description Language (TDL) [ETS16] is a testing language standardized by the European Telecommunications Standards Institute (ETSI)[1] bridges the gap between high-level test purpose specifications and executable test cases [Ulr14]. The language is scenario-based and it can be used for design, documentation, and representation of formalized test descriptions. The main ingredients of the language are *Test Data*, *Test Configuration*, *Test Behavior*, *Test Objectives*, and *Time*. *Test Data* specifies the elements needed to express data sets and data instances that are used in test descriptions. Then, *Test Configuration* typed components and gates and the connections among the gates. *Test Behavior* defines expected behavior. Using *Test Objectives* one can define the objectives of the testing by attaching them to behavior or to a whole test description.

Testing and Test Control Notation version 3 (TTCN-3) [ETS05] is a testing language for test specification and implementation standardized by the European Telecommunications Standards Institute (ETSI). It supports the creation of abstract test specifications which can be executed by providing additional implementation components, such as a SUT adapter.

xUnit [Mes07] is a family of unit-testing frameworks that share common characteristics. Probably the most popular frameworks of this family are JUnit [JUn] and nUnit [NUn] used for testing Java for C# software systems receptively. MSUnit [Mic] is a Microsoft's testing framework for unit testing integrated into Visual Studio which has a similar, but still a little bit different structure compared to the xUnit-family frameworks.

2.4.3 Mutation Testing

Mutation testing or mutation analysis is a technique used to create new test cases as well as examine the quality of available test cases [LS78, HG77]. The standard process of mutation testing is depicted in Figure 2.7. In general, applying mutation analysis on a *System S* starts with the *System Mutation* activity, where *System S* is modified to create *System Mutant S'*. The mutant is obtained by making atomic syntactic changes by using so-called mutation operators.

[1] http://www.etsi.org/

Notation 14 (Mutation Operator)**.** *Let T be a transformation T: S ! S' that happens during the System Mutation activity and creates a mutant by modifying an existing System S in System S'. T is called mutation operator (also known as the mutant operator, mutation rule) [Won01].*

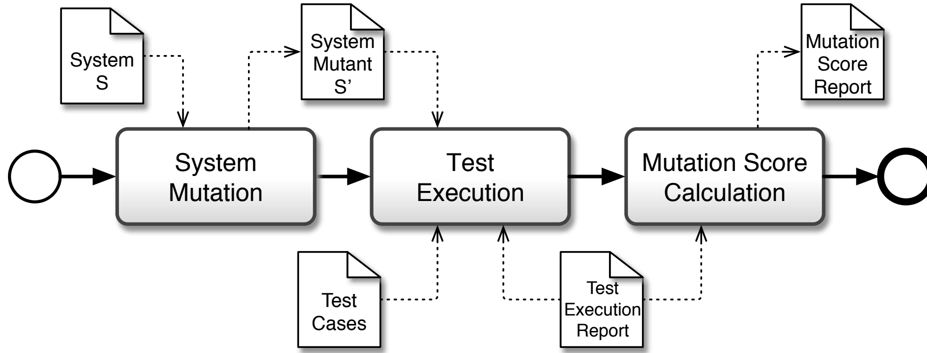


Figure 2.7 The basic mutation testing process

As part of the next activity, namely *Test Execution*, *Test Cases* are executed against the mutated system, which results in two possible outcomes [Won01]. The first possible outcome is the deterministic case, where the mutant is detected, i.e., killed. In case that more mutants are generated at a time, then it is checked that all generated mutants are killed in order to ensure that all system mutants, i.e., incorrect systems, are rejected by the existing test cases. The other possible outcome is a bit more complex as it deals with the equivalent mutants.

Notation 15 (Equivalent Mutant)**.** *An equivalent system mutant is a system mutant whose behavior is the same as that of the existing system but, syntactically different.*

The detection of an equivalent mutant is, in general, very difficult [BA82, JEF14]. After obtaining the execution results in terms of *Test Execution Report*, the mutation score is calculated as part of the *Mutation Score Calculation* activity. Equation 2.1 defines the mutation score as the ratio between the total number of killed mutants (*KilledMutants*) and the total number of non-equivalent mutants (*TotalNumberOfMutants - NumberOfEquivalent-Mutants*) [WAS14]. The final result is provided in terms of *Mutation Score Report*.

$$MutationScore = \frac{NumberOfKilledMutants}{TotalNumberOfMutants - NumberOfEquivalentMutants} \tag{2.1}$$

This *Mutation Score* is an indicator for the quality of the test cases and it value can range from 0 to 1, meaning the closer to 1, the better. In case the mutation score is 1, it means that the test suite was able to detect all seeded faults [WAS14]. In case it is less than 1, it suggests

that the test suite cannot detect all some of the faults in the source code and additional test cases should be created in order to detect those faults [HG77].

2.4.4 Test Case Quality

The work in test case quality area is currently split into two main areas: approaches that identify and provide a set or a catalog of metrics and approaches that provide a general and a standardized quality model that is applicable in any setting. In general, quality models divide the term quality into its essential quality characteristics. Each of these characteristics can be subdivided into more detailed quality sub-characteristics and finally into quality attributes.

Quality Model for Test Specification

The international ISO/IEC standard 9126 defines a general quality model for software products and it contains a quality model for the external and internal quality [ISO01]. It categories software quality attributes into six characteristics, which are further subdivided into sub-characteristics. For each characteristic and sub-characteristic, the capability of the software is determined by a set of internal attributes that can be measured. The characteristics and sub-characteristics can be measured externally by the extent to which the capability is provided by the system containing the software. As this general quality model can be instantiated for a particular domain, in [ZVS+07], an adaptation of the ISO/IEC 9126 quality model to test specification is presented (shown in Figure 2.8). This quality model is an instantiation for test specifications written in the Testing and Test Control Notation (TTCN-3). The discussed general quality model contains a two-part quality model. One part of the quality model is applicable for modeling the internal and external quality of a software product, whereas the other part is intended to model the quality in use of a software product [ISO01]. These product qualities are used during different stages of development and are not completely independent but influence each other. Thus, internal metrics may be used to predict the quality of the final product, also in early development stages [ZVS+07]. For this reason, the quality model for test specification addresses only the internal quality characteristics.

The model is divided into seven main characteristics: *Test Effectivity*, *Reliability*, *Usability*, *Efficiency*, *Maintainability*, *Portability*, and *Reusability*. Each main quality characteristic is structured into several sub-characteristics. The definition for most of the characteristics are generously re-interpreted from the ISO/IEC 9126 and applied for test specification. However, some of the characteristics are renamed in the context of testing, e.g., *Functionality* to *Test Effectivity*. The relationship of the quality model to ISO/IEC 9126 is indicated by

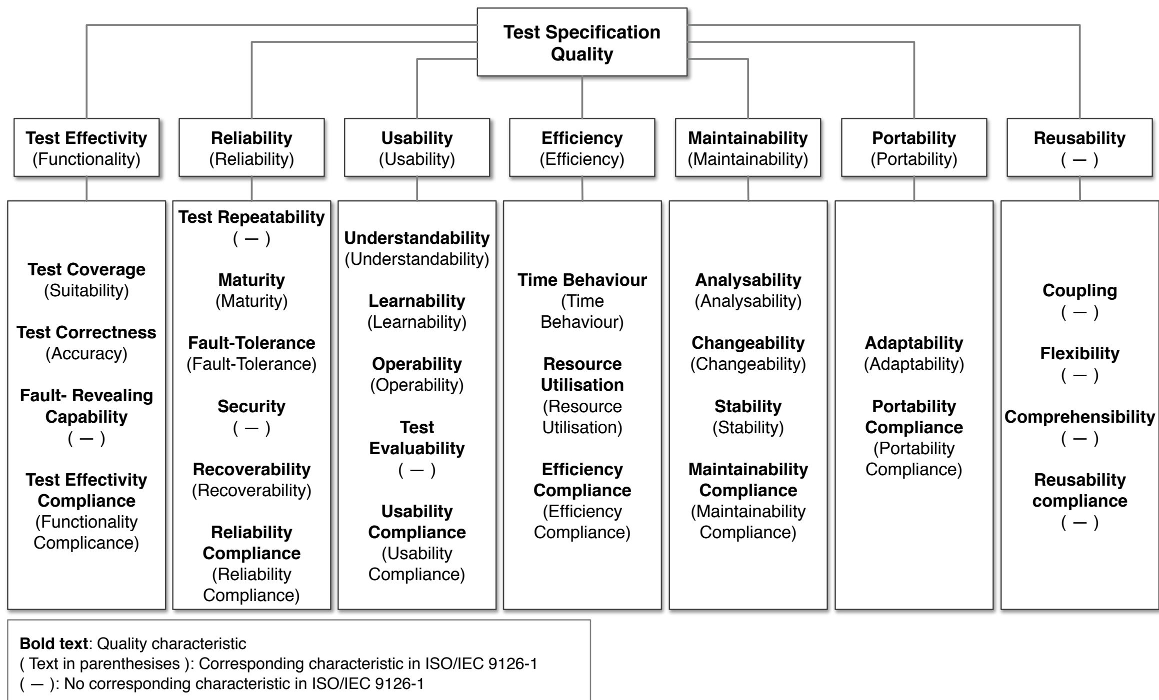


Figure 2.8 The Test Specification Quality Model [ZVS+07] based on the ISO/IEC 9126-1 Quality Model for External and Internal Quality [ISO01]

providing the corresponding name of the ISO/IEC 9126 characteristics in parenthesis. The quality characteristics which have no corresponding aspect in the general ISO/IEC 9126 quality model, are denoted by the sign (*). The quality model for test specification added the quality characteristic *Reusability* which is not explicitly covered in ISO/IEC 9126. This characteristic is added for a reason because, test specifications and parts of them are often reused for different kinds of testing, e.g. test cases, and test data for system level testing may be reused for regression testing, performance testing, or testing different versions of the System Under Test (SUT). Thus, design for reusability is an important quality criterion for test specifications [ZVS+07]. The quality model for test specification presents a set of metrics for each sub-characteristic, gathered through the Goal-Question-Metric (GQM) approach [BCR94].

Model Quality Plan (MQP) approach

The Model Quality Plan (MQP) approach [VE08] defines a procedure for building a quality plan for the quality assessment of software models. This approach combines the advantages of both Goal Question Metric (GQM) and quality models [VGE08]. The quality plan generated from the MQP approach forms the basis for carrying out the quality assessment.

The MQP approach is based on a top-down process and a related metamodel. The incremental and iterative process, illustrated in Figure 2.9, serves as a guideline for defining a quality plan. All the relevant information contained in a model quality plan is formalized by the metamodel shown in Figure 2.10. Hence, the metamodel defines how a model quality plan may look like, and the process guides one how to build it up [VGE08].

At first, during the *Characterization of Context* step, the context factors of a software model are documented to find out what is accurate for the considered software model (e.g. used modeling language, diagram types, development phase). Then, as part of the *Identification of Information Needs* step, the context factors are used for identifying information needs specified by goals and questions. Goals and questions are described based on quality characteristics and quality attributes respectively. These derived characteristics and attributes are the first set-up for the quality model. During the *Definition of Quality Understanding* step, the quality model is extended (e.g. sub-characteristics are introduced) and all characteristics and attributes are interrelated and defined. In the last step, namely the *Definition of Measurements*, the measurement of the bottom level of the quality model (the quality attributes) is documented. The measurements are classified into base measures, derived measures, and indicators. The documentation of a base measure includes, for example, a name, acronym, type, and unit of measurement, informal or formal definition of the measurement method, scale, and scale type. The MQP metamodel groups classes according to the main steps of the MQP process by several packages shown in Figure 2.10.

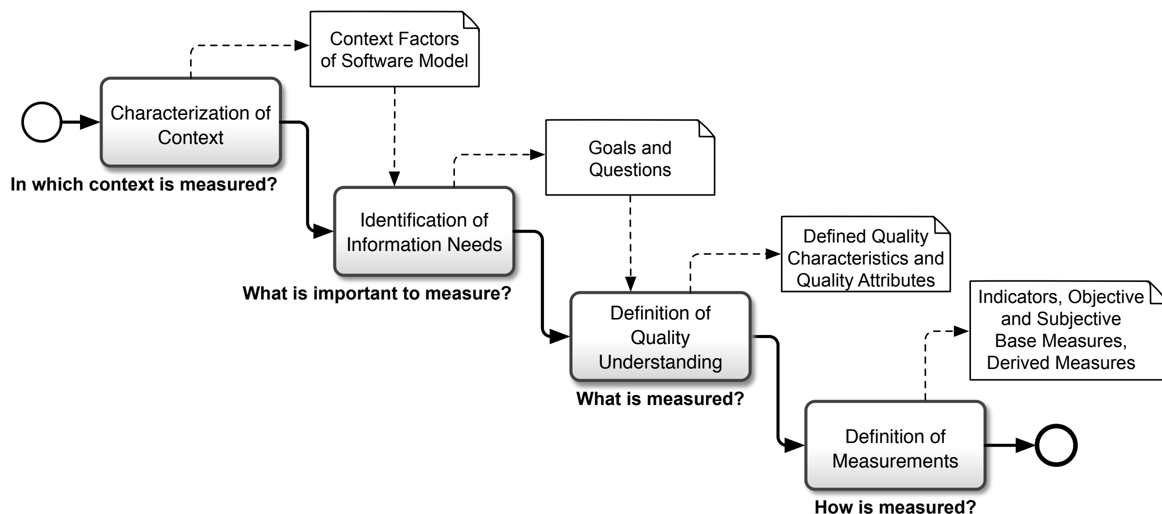


Figure 2.9 The MQP Process [VE08]

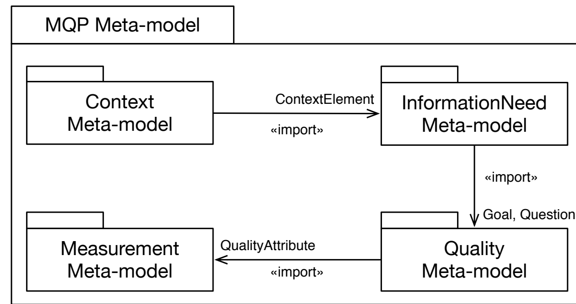


Figure 2.10 An excerpt of the MQP metamodel [VE08]

The packaged classes possess a strong interrelated structure which can be noticed by the imported classes (e.g. ContextElement) point out the seamless transitions between two given steps.

2.5 Technologies

In this section, we give a brief overview of the technologies used to implement the solution approach of this thesis. Subsequently, we introduce the Eclipse Modeling Framework (EMF), Xtend, and Object Constraint Language (OCL).

2.5.1 Eclipse Modeling Framework

The Eclipse Modeling Framework [SBPM09] is a modeling framework for the Integrated Development Environment Eclipse. The central part of the framework is the Ecore modeling language which is used to describe domain models. The models can be created by using different tools provided by EMF, e.g., a graphical editor. Subsequently, based on the domain model, a code generator can be used to create Java code. EMF consists of three main parts: *EMF*, *EMF.Edit*, and *EMF.Codegen*. *EMF* or the core part of the EMF framework includes a metamodel (Ecore) which is used to describe models. It also provides runtime support for the models including, persistence support, change notification, and a reflective API to manipulate EMF objects in a generic way. *EMF.Edit* provides generic reusable classes for building editors for EMF models. It is a command framework, that includes a set of generic command implementation classes so that one can build editors with the full support of automatic undo and redo functions. The *EMF.Codegen* part provides a code generation facility that is capable of generating the necessary artifacts in order to build an editor for an EMF model. The generation facility relies on the JDT (Java Development Tools) component

of Eclipse. As JDT is of high importance for our work, as it was used in the case study we present in Chapter 8, we describe it in more detail.

The JDT component provides APIs to access and manipulate Java source code. Generally, it provides a set of plug-ins, one very important for our work, namely the JDT Core. JDT Core deals with the non-UI infrastructure and it includes Java Model that provides API for navigating as well as the Java Abstract Syntax Tree (AST). By using the Java Model each Java project is internally represented as a model which is a lightweight and fault-tolerant representation of the project. Compared to the AST representation, it contains less information but is fast to create. The Java model is represented as a tree structure and it is defined in the `org.eclipse.jdt.core` plug-in. For example, a *Java project* is represented as *IJavaProject* java model element, each package is represented as the *IPackageFragment* java model element.

The AST on the other hand, is a detailed tree representation of the Java source code and it provides an API to manipulate the source code (modify, create, read, and delete). The AST is located in the `org.eclipse.jdt.core` plug-in, or more precisely in the `org.eclipse.jdt.core.dom` package. Each Java source element is represented as a subclass of the *ASTNode* class and it provides specific information about the object it represents. For example, *MethodDeclaration* is used for methods, *VariableDeclarationFragment*, for variable declarations, *SimpleName* for any string which is not a Java keyword, a Boolean literal (true or false) or the null literal.

The process of working with AST is typically the following: Firstly, by using the *ASTParser* located in `org.eclipse.jdt.core.dom.ASTParser`, a java source code is parsed and its AST is obtained. Then, if modifications are necessary then the AST of the source code is manipulated. An AST Node could be found by checking the different levels in the AST or by using the visitor pattern via the *ASTVisitor* class. In the end, the changes are written back to the source code from the AST.

2.5.2 Xtend

Xtend[2] is a statically-typed programming language which translates to comprehensible Java source code. Syntactically and semantically Xtend has its roots in the Java programming language and as it uses the Java type system, it is fully interoperable with Java. Hence, any existing Java library could be used seamlessly. Xtend also supports specification of multi-line code templates, thus making it suitable for implementation of code generators, i.e., Model-to-Text (M2T) transformations. The compiled output is readable and pretty-printed

[2]https://www.eclipse.org/xtend/

and tends to run as fast as the equivalent handwritten Java code. Xtend improves Java in many aspects, but we focus mainly on those that were relevant for our work, namely extension methods, switch expressions, and template expressions.

Extension methods are a useful feature of Xtend which allows adding new methods to existing types without modifying them. As shown in Code Excerpt 2.1, one option would be to pass the first argument of an extension method inside the parentheses of a method invocation. By using extension methods, the method can be called with the first argument as its receiver, i.e., it can be called as if the method was one of the argument type's members. Doing so, the code becomes more readable which is especially advantageous when method calls are chained.

```
class MyClass {
      def static void main ( String [] args ){
        Object obj = new Object();

        doSomething(obj); // passing as argument to the method
        obj.doSomething(); // calling the extension method as

      }
}
```

Code Excerpt 2.1 Extension method example

Xtend provides a powerful switch expression which is type-based with implicit casts. Compared to Java's switch expressions, it is very different as it is not limited to certain values. The switch expression can be used for any object reference as Object.equals(Object) is used to compare the value in the case with the value one is switching over, as shown in Code Excerpt 2.2.

```
    switch exampleString {
        case exampleString.length > 10 : "a long string."
        case 'simple' : "simple string."
        default : "another simple string."
    }
```

Code Excerpt 2.2 Switch expression example

Xtend also provides specification of templates with intelligent white space handling thus enabling a readable string concatenation. A template starts and ends with a triple single quote (''') and can span multiple lines. Furthermore, a template expression can be nested which are evaluated and the resulting string representation is inserted at the exact position of the nested expression. To interpolate an expression, guillemet terminals are used, namely «expression».

Code Excerpt 2.3 shows a specification of a template and Code Excerpt 2.4 shows its return value.

```
def generateClass(List<Entity> entities) '''
    «FOR e : entities»
      public class «e.name» {
        «FOR f : e.features»
          «IF f instanceOf Attribute»
            «generateField(f)»
          «ENDIF»
        «ENDFOR»
      }
    «ENDFOR»
'''

def generateField(Attribute a) '''
    private «f.type.toFirstUpper» «f.name.toFirstLower»;
'''
```

Code Excerpt 2.3 Template expression example

```
public class Car {
    private Integer numberOfDoors;
    private Integer numberOfSeats;
    private String description;
}
```

Code Excerpt 2.4 Return value of a template expression

2.5.3 Object Constraint Language (OCL)

Object Constraint Language (OCL) [OCL] is a formal language, originally introduced to describe expressions over UML models as UML alone is not sufficient. For example, by solely using the features of the UML class diagram, there is no way to specify constraints on model elements. One alternative would be to describe the constraints in natural language, but they are commonly ambiguous. Another alternative would be to use formal languages like predicate logic, which is unambiguous, but is cumbersome to use. Furthermore, domain experts commonly do not have the expertise to use a formal language. Therefrom, OCL is a compromise between formality on the one side and usability on the other side. It was designed to be read and written more easily than traditional languages but still formal enough. Using OCL, values of instances can be accessed, connected instances can be navigated or collections of instances can be iterated.

OCL is a typed, side-effect free, and declarative specification language. Firstly, each OCL expression evaluates to a type (either a predefined OCL type or a type in the model being evaluated). Furthermore, it must conform to the rules and operations of that type. Side-effect free means that OCL expressions can query or constrain the state of the system but not modify the system itself. Declarative means that OCL specifies what shout be done but not how. Finally, specification refers to the fact that the language definition does not provide any implementation details nor implementation guidelines. In the following, we discuss the most common applications of OCL.

Invariants

In order to specify integrity constraints in OCL, invariants are defined in the context of a specific type also known as the context type of the constraint. The boolean condition, which forms the body of the invariant, is checked. An invariant holds when it is satisfied by all instances of the context type. As shown in the example in Code Excerpt 2.5, all *Students* must have at least one passed exam.

```
1 context Student
2 inv PassedExamsOverZero:self.numberOfPassedExams>0
```

Code Excerpt 2.5 OCL invariant example

Certainly, invariants are the most commonly used OCL expression as they provide the designers with an easy and intuitive way to specify all kinds of conditions that the system must conform to.

Initialization Expressions

OCL can be also used to specify the initial value of a property upon the object's creation. Consequently, the type of expression and type of the initialized property must conform. As shown in the example in Code Excerpt 2.6, every *Student* has not passed exams at its very beginning, i.e., once it is created in the system.

```
1 context Student::hasPassedExams: boolean init: false
```

Code Excerpt 2.6 OCL initialization example

Derived Element

By using derived elements one can define elements whose value is inferred from the value of other model elements. The logic of the derivation is defined in the element's derivation rule. As shown in the example in Code Excerpt 2.7, the element *qualifiedForProjectGroup* is of boolean type and it is defined as positive if the student has at least four passed exams.

```
1 context Student::qualifiedForProjectGroup:boolean derive:
2 if self.exams->select(e|e.status='passed')->size()>=4
3 then true else false endif
```

Code Excerpt 2.7 OCL derivation element example

Query operations

In order to query the system data and return some useful information, a query operation, which is a wrapped OCL expression, can be used. As shown in the example in Code Excerpt 2.8, the query returns all the publications of the given status.

```
context Student::examsWithStatus(s:ExamStatus):Set(Exam)
body: self.exams->select(e|e.status=s);
```

Code Excerpt 2.8 OCL query operation example

Chapter 3

Requirements and Related Work

In this chapter, we give an overview of the related work of this thesis. For this purpose, we first describe a real-world co-migration scenario in Section 3.1 that resulted in the problem statement addressed by this thesis. Based on this scenario, we derive in Section 3.2 a set of requirements that a solution concept should fulfill. We identify and classify related work in Section 3.3 and evaluate it against the specified requirements. The findings of this chapter are summarized in Section 3.4.

3.1 Test Case Co-Migration Scenario

As a running example, we use a real-world migration project (shown in Figure 3.1) where a co-migration was observed [SJGE18] eventually resulting in the problem statement of this thesis (cf. Section 1.2). More specifically, in this context, the problem of enabling cross-platform availability of the well-known Eclipse Modeling Framework (EMF) [SBPM09] was addressed. EMF is highly adopted in practice and generates Java code from platform independent models with embedded Object Constraint Language (OCL) [OCL] expressions. However, applications that target multiple platforms like Windows, macOS, Android, iOS or web browsers usually need to be implemented in different programming languages. As feature-complete Ecore and OCL runtime APIs are not available for all these platforms, their functionality has to be re-implemented. Hence, the reuse of the created models across different technologies is quite difficult and requires time and costs for re-development.

To address this drawback of EMF, a migration to CrossEcore [SJGE18], a multi-platform enabled modeling framework, was performed. By following a generic migration method also presented in [SJGE18], language and architectural migrations were performed so that CrossEcore can generate C#, Swift, TypeScript, and JavaScript code from Ecore models with embedded OCL. The OCL expressions are translated into expressions of the target language

by an OCL compiler. As a result, the CrossEcore's Ecore and OCL API can be consistently used across platforms, thus facilitating application portability.

In order to ensure that the migration of the EMF framework, i.e., the migration of the OCL implementation particularly, was correct, a validation method was needed. As the EMF's OCL implementation is well-tested, with the test cases available in the EMF public code repositories, their reuse was a very intuitive solution to be selected. However, their reuse was not that straightforward as the CrossEcore's OCL implementation was completely different in comparison to the EMF's OCL implementation. But, before even starting with the migration, the large set of existing test cases had to be analyzed whether it is beneficial to migrate them at all. It may be that some parts of the OCL were omitted purposely, thus making the corresponding OCL test cases not needed anymore. Then, no transformation method was existent to guide the migration endeavor and which would address the fundamentally different implementations of OCL. Further, as the migration was performed to different targeting platforms, i.e., programming languages, a "one-size-fits-all" approach is not a perfect solution. This implies usage of a situation-specific transformation method, suitable for the situation for example regarding the target language or the target testing platform. Finally, after the migration of the tests is performed, a validation of the migrated test cases is necessary to avoid possible false conclusions about the success of the migration.

The solution concept provided by this thesis shall address the aforementioned problems, i.e., to provide an end-to-end approach that guides the evaluation of the quality of the test cases, the development of a situation specific transformation method to migrate to the different target platforms and the validation of the test case migration. In the following, we describe both the system and test case migration.

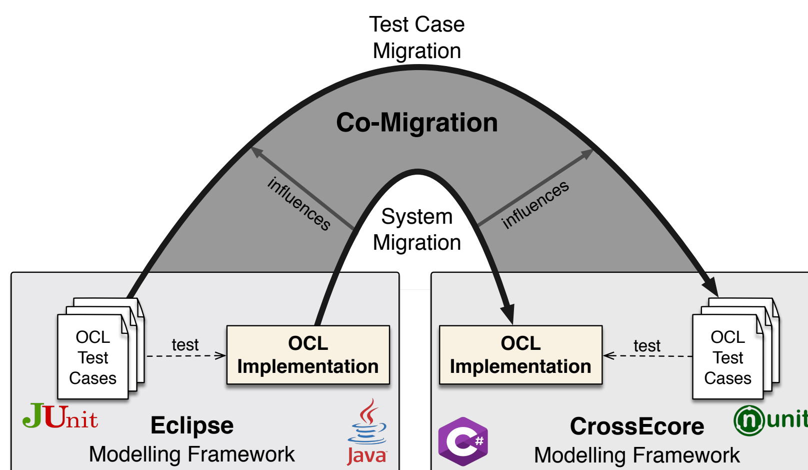


Figure 3.1 The co-migration problem: Migration of a part of the EMF framework to provide cross-platform availability

3.1.1 System Migration

The Eclipse Modeling Framework (EMF) is a modeling framework and a code generation facility for building tools and other applications based on a structured data model for the Integrated Development Environment Eclipse. Ecore, a concrete implementation of the Meta-Object-Facility (MOF) [OMG13a], is the core component of EMF and it is the metametamodel for describing metamodels.

OCL Implementation in EMF

EMF provides an implementation of the Object Constraint Language (OCL) which is an OMG standardized formal language used to describe expressions over UML models. More specifically, OCL expressions are embedded within Ecore models where it can be used to derive the values, execute parameterized model queries, define preconditions and post-conditions, and define class invariants. As shown on the left-hand side left in Figure 3.2), the *EMF Code Generator* component embeds native string-based OCL expressions directly in the emitted Java code with the help of *Java Emitter Templates*. The interpretation of the OCL expressions is delegated by *EMF Runtime API* to the *OCL Interpreter* which firstly parses the OCL expressions from strings and executes them at run-time, i.e., in *Just-In-Time (JIT) manner*. All in all, as EMF mostly focuses on Java, the reuse of the created models across different technologies is quite difficult and requires time and costs for re-development.

OCL Implementation in CrossEcore

CrossEcore [SJGE18], a cross-platform enabled modeling framework, addresses this problem by providing a code generation of platform-specific model code from platform-independent Ecore models with embedded OCL expressions. As shown on the right-hand side left in

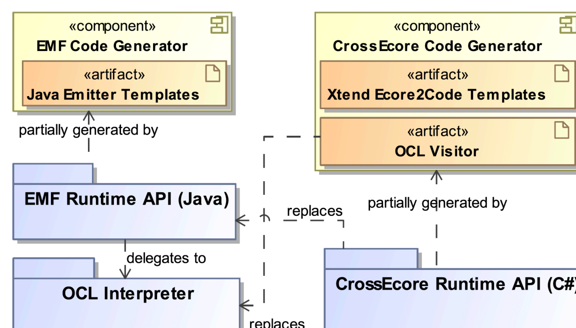


Figure 3.2 OCL implementation in EMF in JIT-fashion (left), CrossEcore's AOT implementation of OCL (right) [SJGE18]

Figure 3.2), the framework includes an Ecore and OCL API (*CrossEcore Runtime API*) that can be used consistently across the platform, as it has a nearly identical structure and behavior on every platform. The OCL compiler (*OCL Visitor*), as part of the *CrossEcore Code Generator*, transcompiles the string-based OCL expressions embedded in a platform-independent model into expressions of the target programming language. The operation bodies, invariants, and derived attributes are automatically translated into expressions of the target language. Hence, the OCL expressions are translated at design-time and ahead of compilation, i.e., in *Ahead-Of-Time (AOT)* manner. As a result, this increases the productivity of the software engineers as they do not have to implement the features and OCL expressions for each platform individually.

The Migration Approach

The system migration was performed by applying the generic migration method provided by Schwichtenberg et al. [SJGE18]. Firstly, the *Base Classes* of Ecore and OCL are implemented. Then, primitive type mappings are defined between platform-independent Ecore/OCL data types to platform-specific types. In the next step, the Ecore code templates are implemented in Xtend [Xte] which is a Java dialect with a simplified syntax. Most of the templates are easy to adjust, as they contain no or simple program logic. Then, the OCL compiler for the new programming language is adopted by providing an individual *OCL Visitor* method. In the next step, the previously defined templates are used for the generation of the Ecore classes. Last but not least, the migration to CrossEcore for the new target programming language is validated. The validation step checks whether the CrossEcore's code generator emits code that correctly implements the functionality of Ecore and OCL.

3.1.2 Test Case Migration

From a migration validation perspective, as already mentioned, we had to check whether the OCL implementation in the migrated CrossEcore framework is functionally equal to the OCL implementation in the EMF framework. A proven technique that validates whether a system provides the desired functionality is software testing.

OCL Test Cases in EMF

As EMF is a well-tested framework, with more than 4000 JUnit [JUn] test cases being available on public code repositories[1], a major goal was to reuse these test cases to validate

[1] http://git.eclipse.org/c/ocl/org.eclipse.ocl.git/tree/tests/

the migrated OCL implementation in CrossEcore. An OCL test case tests a specific part of the OCL implementation, by executing and asserting a particular OCL expression, e.g., filtering or casting of a given collection. The OCL implementation in EMF is in a *Just-In-Time (JIT)* manner, and therefore, the test cases testing the OCL implementation in EMF as well. This means that they also contain native string-based OCL expressions as shown in Figure 3.3 written in *JUnit*. The particular test case tests the casting functionality, i.e., it checks whether the execution of the `asBag()` function on the `Sequence{1, 2.0, "3"}` collection, results in `Bag{1, 2.0, "3"}`, which is the expected result. The assert function `assertQueryResults()`, compares the values of the expected result and the particular OCL functionality which are specified as strings.

```
assertQueryResults("Bag{1, 2.0, '3'}",
                   "Sequence{1, 2.0, '3'}->asBag()");
```

Figure 3.3 Implementation of an OCL Test Case in EMF (Just-In-Time (JIT) Compilation)

OCL Test Cases in CrossEcore

As we have already seen in Section 3.1.1, the OCL implementation in CrossEcore differs fundamentally compared to the OCL implementation in EMF. In contrast to the JIT-implementation of OCL in EMF, *Ahead-Of-Time (AOT) compilation* is used for the OCL implementation in CrossEcore. Consequently, in the migrated OCL test cases, the OCL expressions have to be translated into equivalent expressions of the target programming language which suggests that they are compiled before the test code is being executed. Figure 3.4 shows the should-be state or the projected transformation of the test case already shown in Figure 3.3. The test case addresses the C# API realization of the CrossEcore and is written in *MSUnit* [Mic]. As can be seen, the expected result as well as the particular OCL functionality being tested are defined as C# code, i.e., in accordance with the changes previously defined and performed in the system migration. Similarly to the previous example,

```
assertQueryResults(new Bag<object>{1, 2.0, "3"},
                   new Sequence<object>{1, 2.0, "3"}.asBag());
```

Figure 3.4 Implementation of an OCL Test Case in CrossEcore (Ahead-Of-Time (AOT) Compilation)

this test tests the casting functionality, i.e., it checks whether the execution of the `asBag()` function on the `Sequence{1, 2.0, "3"}` collection defined as a C# object, results in the object `Bag{1, 2.0, "3"}`, which is the basically the expected result. The assert function `assertQueryResults()`, compares the values of the expected result and the casting OCL function.

3.2 Solution Requirements

In this section, based on the co-migration scenario described in the previous section, as well as on the challenges defined in Section 1.2, we derive a set of requirements that a general solution to the test case migration problem should address. We classify the requirements into three main categories according to the three main phases: *Pre-Migration Phase: Test Case Quality Evaluation*, *Migration Phase: Co-Evolution and Migration*, and *Post-Migration Phase: Migration Validation*.

3.2.1 Pre-Migration Phase: Test Case Quality Evaluation

R1. Quality Assessment of Test Cases. This requirement states that the solution approach should provide means to perform a quality evaluation of test cases, which in turn means a feasibility analysis of the test case migration. To address this requirement, the solution approach needs to provide a test case quality assessment method for the existing test cases for the original system.

- **R1.1 - Common Quality Understanding**: *The solution should use definitions for qualities of test cases for a consistent and common quality understanding.* Among the stakeholders, every team member should have the same understanding of quality related to the test case domain. False interpretations can lead to misunderstandings and incorrect results.

- **R1.2 - Context Characterization**: *The solution should be able to provide a minimum set of context factors.* The quality of test cases strongly depends on the context of use, in which they are created, managed, and applied. This means that different factors like the available artifacts, the environment of the test, test case type (code-based or natural language-based), etc. should be considered.

- **R1.3 - Definition of Measurements**: *The solution should distinguish between objective and subjective measurements.* The evaluation of test cases requires measurements

to ensure the attainment of numerical quality goals. For this reason, metrics are introduced to quantify different quality aspects of test cases or software artifacts in general.

- **R1.4 - Systematic Approach**: *The quality evaluation of test cases should be a systematic process that guides the stakeholders based on the context factors.* Existing evaluation approaches define a set of measurements that applies to test cases. But, according to the context and information needs, this set of measurements might be reduced or extended. Hence, a systematic process is required that guides the stakeholders based on the test case's context factors.

3.2.2 Migration Phase: Co-Evolution and Migration

R2. Co-evolution of Test Cases. *This requirement states that the solution approach should provide means to perform a co-evolution analysis between the system and its test cases and incorporate the results in the method engineering of the transformation method.* To address this requirement, the solution approach shall address the three main activities from the co-evolution process, namely the change detection, the impact analysis, and the change propagation and incorporate them in the method engineering process.

- **R2.1 - Change Detection**: *The change detection regarding the system migration should be performed on a conceptual level.* To address this requirement, the solution approach needs to provide a change detection method for the original and migrated system as well as a description of the changes in a structured way.
- **R2.2 - Impact Analysis** *The impact analysis should be supported by a method that performs the analysis on a conceptual level.* To address this requirement, the solution approach needs to provide an impact analysis method that provides the impact of the system migration to the test case migration in terms of an impact mode that describes the necessary changes to the existing test cases of the original system.
- **R2.3 - Change Propagation** *The change propagation of the detected system changes to the test cases should be on a conceptual level.* To address this requirement, the solution approach needs to provide a change propagation method that defines the necessary changes relevant in the test case migration.

R3. Automated Transformation of Test Cases. *This requirement states that the solution approach should enable automated transformation, i.e., migration of the test cases whenever*

possible. To address this requirement, the solution approach shall provide a set of methods to apply reengineering activities.

- **R3.1 - Reverse Engineering**: *The reverse engineering activity should be supported by a (semi-)automated extraction of meaningful standardized test models on a different level of abstraction.* Initial extraction of a test model out of the existing test code, with the help of text-to-model transformation, should be supported. Furthermore, model-to-model transformations from lower to a higher level of abstraction should be also supported.

- **R3.2 - Restructuring** *The restructuring activity should be supported by a (semi-) automated transformation and adaptation of test models at a different level of abstraction.* Restructuring test cases means dealing directly with their structural complexity in order to perform, if necessary, a set of relevant changes.

- **R3.3 - Forward Engineering** *The forward engineering activity should be supported by an automated transformation of test models from higher to lower abstraction level, i.e., to perform concretization.* Concretization activities, i.e., model-to-model transformations from higher to lower abstraction level, should be supported. The last step of forward engineering should enable test case code generation out of the previously concretized test models.

R4. Situativity of Test Case Migration Method. *This requirement states that the solution approach should enable high flexibility to different migration scenarios.* To address this requirement, the solution approach shall provide a meta-method which allows easy construction of situation-specific transformation methods for test cases.

- **R4.1 - Test-specific Method Base**: *This method base should provide test-specific method fragments and method patterns.* The method fragments and the method patterns, as constituting part of a method base, should be test-specific which means that they have to properly reflect the different test concepts like the test artifacts, test code, and test model and appropriate activities like test case extraction, test case understanding, etc.

- **R4.2 - Test-specific Method Engineering Process**: *The method engineering process should support method development and method enactment of test transformation methods for test cases.* The method engineering process should rely on the situational context model which contains also test case characteristics. Furthermore, it should rely on the test-specific migration patterns, which provide common test case transformation strategies.

- **R4.3 - Co-evolution inclusion**: *The method engineering approach should use the obtained results from the co-evolution activities (change detection, impact analysis, and change propagation).* The method engineering process, consisting of method development and method enactment should use the results obtained from the co-evolution analysis step.

3.2.3 Post-Migration Phase: Migration Validation

R5. Validation of Migrated Test Cases. *This requirement states that the solution approach should indicate the success of the test case migration.* To address this requirement, the solution approach shall provide a validation method that could identify bad smells for the test case migration, i.e., identify false positives and false negatives among the migrated test cases.

- **R5.1 - Systematic process** *The solution approach must be a systematic process that guides the stakeholders in a co-migration setting to perform a validation process.* The result of such a process should ideally indicate that problematic, i.e., erroneous migrated test cases exist.

- **R5.2 - Automation** *The validation of the test cases shall be automated in order to deal with a high number of test cases.* To address this requirement, the solution approach should provide a light-weight framework for (semi-)automated test case migration validation.

- **R5.3 - Generality** *The applicability of the validation approach shall not be limited to a single migration context.* To address this requirement, the solution approach cannot assume that the original system and test cases and the migrated system and test cases were developed in a specific technology or employ a specific software architecture.

3.3 Related Work

In this section, we give an overview of the related work of this thesis. Thereby, the related work falls into three categories: test case quality evaluation, then evolution, reengineering and migration, and finally, test case migration validation.

3.3.1 Quality Evaluation

The work in the test case quality assessment area is currently split into two main areas: approaches that identify and provide a set or a catalog of metrics and approaches that provide a general and a standardized quality model that is applicable in any setting. The existing work which is done regarding the metrics is predominantly considering test effectiveness [SWH11, MNDT09, GJG14, EBI06, Che01a]. As there are other quality aspects for test cases besides effectiveness, all these approaches fulfill partially only the requirement **R1.3**.

The introduction of metrics to quantify different quality aspects of test cases is considered in [Sne03]. The intention of Sneed [Sne03] is to provide a set of internationally standardized and recognized test metrics from which the software development teams can select to plan their test projects and evaluate their test operation. This addresses fully the requirement **R1.3**. However, these metrics do not consider the context of use of test cases. Moreover, a common definition for the quality characteristics is also missing.

Bowes et al. [BHP+17], provide a list of 15 testing principles that represent the basis of testing goals and best practices and how they can be quantified as indicators for test case quality, thus addressing partially requirements **R1.2** and **R1.3**. However, their main focus is not on relation to an existing quality standard and consideration of the context of use.

Kaner argues in [Kan03] that it depends on the purpose of how good given test cases are. According to Kaner, test cases can be "good" in different ways, but it is impossible that is good at all of them. As test cases are created according to different styles in different domains, a good test case in one domain is different from a good test in another domain. Thereby, Kaner clearly emphasizes the importance of the context of use of test cases, thus addressing the requirement **R1.2**. However, neither relation to a quality standard nor a systematic quality assessment approach is presented.

In general, quality models divide the term quality into its essential quality characteristics. Each of these characteristics can be subdivided into more detailed quality sub-characteristics and finally into quality attributes. In [ZVS+07], an adaptation of the ISO/IEC 9126 quality model [ISO01] to test specifications is presented. The definition for most of the characteristics are generously re-interpreted from the ISO/IEC 9126 and applied for test specification. Moreover, this approach introduces sub-characteristics, quality attributes, and even measurements for some of the quality attributes. So, this approach addresses fully the requirements **R1.2** and **R1.3**. However, quality models do not document their assumptions about the context and it remains unclear to which degree a quality model is applicable to a given set of test cases. Moreover, this approach does not provide a systematic process on how to perform a quality evaluation. Lastly, the ISO/IEC 9126 was replaced by the ISO/IEC 25010 [ISO11c] standard in 2011.

A well-known methodology to find appropriate metrics for an explicitly stated purpose is the GQM approach [BCR94]. GQM considers the characterization of context factors for the organization and development projects. The Model Quality Plan (MQP) approach [VE08] specializes GQM to describe a procedure for building a quality plan for quality assessment of both static and dynamic software models. It combines the advantages of both GQM and quality models [VGE08] and thus basically addresses up to some extent all of the previously defined requirements. However, MQP does not explicitly focus on test cases. This means, that the common quality understanding is not based on a common, standardized quality model for test cases, meaning that the requirement **R1.1** is partially fulfilled. Then it does not provide all required context factors from a testing perspective implying that the requirement **R1.2** is also partially fulfilled. The measures defined in the method are specialized for the software models and therefore is the requirement **R1.3** also partially fulfilled. Finally, the requirement **R1.4** is fully addressed as it defines a process that is generic and can be reused in the domain of test cases.

● fulfilled ◑ partly fulfilled ○ not fulfilled	R 1.1	R 1.2	R 1.3	R 1.4
Sneed [Sne03]	○	○	●	○
Bowes et al. [BHP+17]	○	◑	◑	○
Kaner [Kan03]	○	◑	○	○
Zeiss et al. [ZVS+07]	○	●	●	○
Voigt and Engels [VE08]	◑	◑	◑	●

Figure 3.5 Evaluation of selected test case quality evaluation approaches against requirements

In summary, as it could be seen in the evaluation matrix shown in Figure 3.5, the existing approaches are not fully addressing the previously identified requirements on test case quality evaluation (**R1.1 - R1.4**). However, the evaluation matrix suggests a possible complete solution, namely a combination of the last two approaches, i.e., the adaptation of the ISO/IEC 9126 quality model for test specifications [ZVS^{+}07] and the Model Quality Plan (MQP) approach [VE08]. By combining the benefits of these two approaches, we present in Chapter 5 a novel approach for test quality evaluation.

3.3.2 Evolution, Reengineering, and Migration

The related work regarding the migration phase is split into three main areas: Test Case Evolution, Test Case Reengineering, and Software & Test Case Migration.

Test Case Evolution

In the area of test case evolution, there is already a lot of work, which is predominantly oriented on the continuous evolution of test cases with the system. Compared to the evolution of test cases in a migration setting, it is much more fine granular. This means that small changes in the system are detected, analyzed, and propagated to the test cases.

Zaidman et al. [ZRDvD08] study the co-evolution by mining software repositories. The overall goal of the approach is to investigate whether the production code and the accompanying tests co-evolve by exploring a project's versioning system, code coverage reports, and size metrics. The approach deals with unit and integration tests. The automated approach provides a tool called *TeMo*(Test Monitor) three different views on the co-evolution: change history view which visualizes the commit-behavior of the developers, growth history view which shows the relative growth of the production code and test code over time, and finally test quality evolution view which plots the test coverage of a system against the fraction of test code at discrete times. As this approach focuses on continuous co-evolution, it partially the three defined evolution requirements.

Farooq et al. [FIMR10] propose a model-based regression testing approach for evolving systems with flexible tool support. Based on the analysis of the relationship between system class diagram and system state-machines the changes are firstly detected and then propagated to the corresponding test suites. The semi-automatic approach takes as input the system models and the test cases in XML format. Then, an XMI parser firstly parses the input models and then, a state machine comparator computes the delta between the models. A test suite classifier classifies the baseline test suite into obsolete, reusable, and re-testable test cases. As this approach is model-driven, it is close to fulfilling the specified requirements. However, as the goal is to understand the conceptual changes in the system migration as well as their impact on the test cases, this method only partially addresses evolution requirements.

In order to understand the myths and the realities in test suite evolution, Pinto et al. [PSO12] study the test suite evolution in a systematic and comprehensive manner. The outcome of the approach provides an understanding of how and why unit tests evolve (i.e., how and why tests are modified, added, and deleted). The semi-automatic approach combines various static and dynamic analysis techniques that compute the difference between test suites associated with two versions of a program and categorization of such changes along two dimensions: the static differences between tests in the two test suites and the behavioral differences between such tests. The tool called TestEvol takes as input a Java program and JUnit test cases and automatically computes differences in the behavior of the test suites on the two program versions, classifies the actual repairs performed between the versions, and computes the coverage attained by the tests on the two program versions. This work mainly

focuses on the evolution of test cases and does not analyze the co-evolution, i.e., the impact a system changes could have. Hence, only the **R2.1** requirement is partially addressed.

Mirzaaghaei et. al [MPP12] provide a semi-automatic approach that supports test suite evolution through test case adaptations by automatically repairing and generating test cases during software evolution. In their work, they identify frequent actions for adapting test cases that software developers commonly adopt to repair and generate test cases. Furthermore, they define five algorithms for evolving test cases as a solution to support software developers. The solution approach, firstly, analyzes the software changes by diffing the original and modified version of a software. Then, by using one of the five test evolution algorithms, on the base of the previously identified common test case adaptations, the test cases are properly adapted. The authors also provide a prototypical framework implementation called TestCareAssistant (TCA) which takes the original and the modified version of the program and applies the previously described steps. As this approach deals with the problem of repairing existing test cases and generating new ones to react to incremental changes in software systems, it only partially addresses the requirements.

Lastly, Rapos [Rap15] analyzes the co-evolution of model-based tests for industrial automotive software. He proposes a method which should improve the model-based test efficiency by co-evolving test models alongside system models. To enable this, he studies software model evolution patterns and their effects on test models in order to apply updates directly to the tests. The solution approach is automated and is used for system testing. The tool, an extension of SimuLink, takes two consecutive versions of a model or set of models along with the first version's test models. As output, a set of relevant changes that need to be made to the test model is provided. These changes should ensure that the updated test model is a correct test for the newly updated model. Again, this approach analyzes the incremental changes in software systems, thus only partially addresses the three requirements.

In summary, as shown in Figure 3.6, the existing evolution or co-evolution approaches address just partly the three defined requirements. The main drawback of the existing approaches is that they are meant to deal with incremental changes and not coarse-grained changes, i.e., conceptual changes. Most of them describe change detection and impact analysis on a very fine granular level, i.e., how to deal with relatively simple, incremental code or model changes. As a consequence, none of the methods defines means to detect conceptual changes and analyze their impact on the test cases. Nevertheless, the analysis of the existing methods helped us to come to a novel approach for applying co-evolution analysis on a conceptual level by leveraging the concept modeling technique.

● fulfilled ◑ partly fulfilled ○ not fulfilled	R 2.1	R 2.2	R 2.3
Zaidman et al. [ZRDvD08]	◑	◑	◑
Farooq et al. [FIMR10]	◑	◑	◑
Pinto et al. [PSO12]	◑	○	○
Mirzaaghaei et al. [MPP12]	◑	◑	◑
Rapos [Rap15]	◑	◑	◑

Figure 3.6 Evaluation of selected test case evolution approaches against requirements

Test Case Reengineering

In enabling automated test case migration, similarly to system migration, an important role plays the reengineering. For this purpose, we analyze different existing reengineering approaches for test cases.

Smartesting [BL14] is an approach that enables the transition from classical to model-based testing. That means, in a semi-automated way, from a set of existing test cases a test model is extracted thus enabling easier refactoring and maintenance of the test cases. The approach focuses on system testing and starts with the refactoring of the test legacy. Then, using model inference, a test model in the form of BPMN is obtained. Using the test model, test cases are generated that further could be imported into a specific test management tool. The whole process is supported by the Impulse tool. The Smartesting approach fulfills partially the **R3.1** requirement, as only a single level of abstraction is supported by the reverse engineering method. It also fulfills partially the **R3.3** requirement, as only the test code generation is supported.

FormTester [DLW15] is an approach that enables the effective integration of model-based and manually specified test cases. The overall goal is to convert existing test projects to model-based testing by extracting test models from existing test cases in a semi-automated way. The approach focuses on system testing and takes as input test cases specified in Gherkin format and applies reverse engineering techniques that produce a test model represented as a finite state machine. In the next step, the test models are used as the basis for the test case generation. In the end, the test cases are executed in online-testing fashion in order to identify untestable parts of a web application. This approach partially fulfills the **R3.1** requirement, as it supports reverse engineering with a single level of abstraction. Besides, it also partially fulfills the **R3.3** requirement, as it only supports the test code generation.

In [JKK+09], a similar approach for synthesizing of test models from test cases is presented. The overall goal of the method is to improve the defect detection capability in

system testing. The approach relies on the method of parallel composition and allows the creation of a single test model from a number of test cases. Firstly, it takes the existing test cases as input and relevant actions are listed parameterized. In order to hold some of the state information of the SUT, variables are created. Then, the SUT is initialized and recurring states within test cases are marked and labeled. Finally, the test cases merged with the variables and the initialization to form a new test model in terms of a labeled state transition system (LSTS). This approach only addresses the **R3.1** requirement as it only provides a reverse engineering method. However, as it only provides a single level of abstraction, we consider that it partially fulfills this requirement.

Milani Fard et al. [FMM14] present an approach that leverages existing tests in automated test generation for web applications. The basic idea comes from the fact that a human-written test suite is a valuable source of domain knowledge, which can be used to enable automated test generation for web applications. The approach mines the human knowledge present in the form of input values, event sequences, and assertions. It takes Selenium test cases as input and outputs state-flow graph as test models. It combines the inferred knowledge with the power of automated crawling and extends the test suite for uncovered/unchecked portions of the web application under test. The approach is supported by a tool called Testilizer. This approach supports reverse engineering with a single level of abstraction, and therefore, it only partially fulfills the **R3.1** requirement. Besides, as it only supports the test code generation, it also partially fulfills the **R3.3** requirement.

Werner et al. [WG11] propose a state-merging approach termed semantic state-merging for model reconstruction by mining test cases. The reconstructed model should serve for online monitoring. For a given set of smaller test suites, a behavioral specification is learned. The approach relies on the learning algorithm L* to mine the existing test case. Firstly, the semantic properties of test cases are exploited in order to detect implicitly defined behavior. The available test cases are stored in a data structure, a trace graph. Then, based on defined merging rules for cyclic test cases and for test cases with default branches the trace graph is completed. As this approach only provides a reverse engineering method with a single level of abstraction, we conclude that it partially fulfills only the **R3.1** requirement.

Xu et al. [XXBW12] present an approach for mining executable specifications of web applications from Selenium IDE tests thus enabling model-based testing. They mine a behavior specification that captures the behavior of the tests at a high level of abstraction. The extracted specification can be used to simulate the behavior of the system. As shown, all the tests previously used to mine the specification are completely reproducible. Before starting with the actual mining of the test, pre-synthesis-activities are performed in order to prepare the test for the mining activity. Firstly, clustering of similar test actions is performed

by identifying similar test actions context-sensitive clustering to normalize the given Selenium IDE tests. Then, pattern mining of test actions that represent meaningful functions. Finally, a transformation of Selenium IDE tests into abstract tests is performed, which are similar to the tests used in the existing model-mining techniques. Thereafter, comes the actual synthesis of a high-level Petri-net from the abstract tests. The semi-automatic approach works for system tests and relies on a process mining tool called ProM, and Selenium IDE. This approach provides a reverse engineering method with a single level of abstraction, and therefore, we conclude that it only partially fulfills the **R3.1** requirement. Besides, it also partially fulfills the **R3.3** requirement, as it only supports the test code generation.

Hungar et al. [HMS03] propose a method for constructing models utilizing automata learning. The goal is to enhance error detection and diagnosis, to support regression testing, to enable coverage analysis, and to support online testing. Firstly, the existing test cases are observed and abstraction is performed. Then, by applying L* learning algorithm, models in terms of finite automata are constructed. The method is semi-automated and deals with system testing. This approach also addresses only the **R3.1** requirement as it only provides a reverse engineering method. As it only provides a single level of abstraction, we consider that it partially fulfills this requirement.

● fulfilled ◑ partly fulfilled ○ not fulfilled

	R 3.1	R 3.2	R 3.3
Bouzy and Legeard [BL14]	◑	○	◑
Dixit et al. [DLW15]	◑	○	◑
Jääskeläinen et al. [JKK+09]	◑	○	○
Milani Fard et al. [MFMM14]	◑	○	◑
Werner and Grabowski [WG11]	◑	○	○
Xu et al. [XXBW12]	◑	○	◑
Hungar et al. [HHMS03]	◑	○	○

Figure 3.7 Evaluation of selected test case reengineering approaches against requirements

In summary, as shown in Figure 3.7, the existing reengineering approaches address just partly the three defined requirements. The main drawback of the existing approaches, as Figure 3.7 clearly shows, is that they do not support the restructuring as an integral part of the reengineering process. However, this does not mean that no changes are applied to the extracted models, but rather an implicit restructuring either done as part of the reverse engineering or forward engineering. Furthermore, all of them only support reverse engineering to a single level of abstraction with pretty much different modeling notations

and languages. Similarly, regarding forward engineering only on the basis of model-to-text transformation, test code is being generated from the previously extracted test model. Even though the evaluation matrix shown in Figure 3.7 does not suggest a clear combination of some of the existing works, it helped us regarding the definition of the method base, where we had to define the relevant method fragments in terms of artifacts, (code and models), activities (extraction and generation), tools, etc.

Software & Test Case Migration

In the area of software and test case migration, we analyze different software migration projects and frameworks that also cover up to some extent the migration of test cases. Each project was dominantly analyzed from a testing perspective and against the following set of features: migration type, overall testing goal, test level, testing strategy and testing method, degree of automation, and tooling.

In the *Remics* (REuse and Migration of legacy applications to Interoperable Cloud Services [Moh10]) project, an architectural migration to the cloud was performed. Seen from a testing point of view, the overall goal was to safeguard the functional compliance between the modernized and the legacy system as well as to validate some non-functional requirements (performance, reliability, security). To fulfill these goals the following testing strategies were applied: integration testing, regression testing, functional testing, online testing, and performance testing (load testing). As no test cases were reused from the legacy system, a model-based approach was applied to derive the test cases for the migrated system. For that purpose, the previously reengineered functional requirements were compiled into software/systems requirements specification and a test model was manually derived. Then, out of the test model, functional system tests were automatically generated and executed. The test modeling was performed by using the UML Testing Profile (UTP) in the *TestFokus!MBT* tool. The UTP tests were then concretized in terms of TTCN-3 test cases and such were executed in the TTCN-3 test execution system *TTworkbench*. This approach partially fulfills the requirement **R4.2**, as it only defines model-based testing based on the extracted system models.

In the SOAMIG [ZWH+11] project, two different types of migration to the cloud were performed, a language migration and an architectural migration. In the language migration from COBOL to Java, the overall testing goal was to ensure the syntactical correctness and semantical equivalence. Regarding the syntactical check, the testing was performed by manual creation of test cases for each construct in COBOL thus covering all syntactic and semantic variations. On the other hand, the transformed Java code was verified to be syntactically correct by compiling it. The semantical check was performed by comparing the

results of the legacy COBOL constructs with those of the migrated Java constructs. In the case of the architectural cloud migration, the overall test goal was to ensure the equivalent behavior of legacy and migrated system in the cloud. As the testing discipline was outside of the project scope only manually testing was performed. The correctness of each service was verified by unit tests (JUnit) and the interaction of services, the Enterprise Service Bus, the business process layer and the view layer were checked by manual integration and system tests. Additionally, manual regression and acceptance tests were performed by the developers of the legacy system ensured that the service-oriented system did behave like the legacy system. As the approach only defines model-based testing based on the extracted system models., it partially fulfills the requirement **R4.2**.

The DynaMod [vHFG^{+}11] project is dealing with model-driven modernization and architectural migration. The overall testing goal is to perform performance testing based on workload generation, allowing to compare quality properties, such as performance and reliability, among the modernized and the outdated system. To create the performance tests, a model-based testing approach was followed. Namely, the test cases were derived from the usage models extracted from the legacy system under the production workload. The approach is semi-automatic as the test models are created manually whereas the generation and execution of test cases are automatic. The load testing and performance testing was performed by using Apache JMeter and Markov4JMeter. This approach addresses partially the requirement **R4.2**, as it provides only a rudimentary approach to generate test cases out of the system models.

The Artist [MKA^{+}14] migration project aims at supporting the migration and modernization of legacy software assets and businesses to the cloud. Therefore, it provides a generic customizable model-based methodology and corresponding open source tooling for migrating such applications to the cloud. It covers the traditional reverse engineering and forward engineering phases, i.e., the actual migration. Furthermore, it also addresses the pre-migration by providing feasibility analysis from both technical and business perspectives as well as post-migration by providing verification and certification of the migrated system. From testing point of view, it aims at ensuring behavioral equivalence (functional goals) and fulfillment of non-behavioral characteristics like performance efficiency and reliability (non-functional goals). Regarding the testing methods used, it supports model-based testing, end-user functional (typical user scenarios executed on both the legacy and the migrated system) and non-functional testing. In the model-based testing approach, the test cases are migrated in a model-driven way, by applying the same reengineering tool as for the system code. During the transformation of the test cases, relevant changes are mirrored on the test

cases, i.e., the changes of the system that also influence the test cases, thus fully addressing the requirement **R4.2** and partially the requirement**R4.3**.

The MEFiSTo (Method Engineering Framework for Situation-Specific Software Transformation Methods) framework [Gri16] is a Situational Method Engineering (SME) framework. SME is an established engineering discipline for developing situation-specific methods by considering the situational context in which the method will be applied. In the domain of software modernization, the context comprises the characteristics of the legacy system, the intended target design, and the characteristics of the modernization project. The MEFiSTo framework is an approach that enables the modular construction of transformation methods thus enabling a high degree of controlled flexibility. The transformation methods are developed by assembling reusable building blocks of methods which are stored in a repository called method base [Bri96]. MEFiSTo, uses two different types of building blocks, namely method fragments and method patterns. The method fragments are the atomic building blocks of migration methods, whereas the method patterns are representing strategies and indicate which fragments should be used. Furthermore, it defines a method engineering process which describes the main activities to be followed in order to create a situation-specific migration method as well as their relation to the method base. However, this approach addresses solely the system artifacts, as it deals with software modernization. This implies that it partially addresses the requirements **R4.1** and **R4.2**. Consequently, the co-evolution of test cases is not considered when the situational context is characterized.

Compared to the previous migration projects, MoDisco [BCJM10a] is a generic and extensible framework that relies on MDE principles and techniques that can be efficiently applied in the domain of reverse engineering by addressing both the model discovery and model understanding activities. In a case study where architectural migration from VB6 to JEE was performed, the aim was to verify that the transformation has not caused unintended effects and that the new system still behaves like the initial one. The testing strategy was to combine reference testing and regression testing. Firstly, before the migration is even performed, based on scenarios described as Unit Tests the original system is tested. Then, the migration is performed and by applying regression testing and reusing the same test scenarios, the migrated system is tested. In the end, the results of both testing activities are compared. When all equal, the migration is considered to be successful. The implementation of MoDisco comes as an Eclipse open-source project which provides an extensible and customizable MDRE framework. Using this framework, model-driven tools can be developed that support different reengineering scenarios such as legacy migration or modernization. This approach is quite generic, thus making it applicable in almost any context. But, on the other hand, there is no support on migrating specific artifacts like test cases, which means it fulfills only

partially the requirements **R4.1** and **R4.2**. Consequently, the co-evolution of test cases is also not considered.

● fulfilled ◑ partly fulfilled ○ not fulfilled	R 4.1	R 4.2	R 4.3
Mohagheghi [Moh10]	○	◑	○
Zillmann et al. [ZWH+11]	○	◑	○
van Hoorn et al. [vHFG+11]	○	◑	○
Menychtas et al. [MKA+14]	○	●	◑
Grieger [Gri16]	◑	◑	○
Bruneliere [BCJM10b]	◑	◑	○

Figure 3.8 Evaluation of selected software migration and modernization projects against requirements

In summary, as shown in Figure 3.8, the existing reengineering approaches address just partly the three defined requirements. Most of them provide some kind of process to apply some kind of testing, mostly model-based testing based on the extracted system models. Most of the approaches (except [Gri16]), do not support the specification of reusable blocks like method fragments or method patterns. However, as the main drawback of most of the existing approaches (except [MKA^{+}14]) is the missing support for the co-evolution of test cases. Exactly by combining the last two mentioned approaches, namely the MEFiSTo approach [Gri16] and the ARTIST framework [MKA^{+}14], lead us to a solution approach that combines the benefits of both.

3.3.3 Migration Validation

Code and test case refactoring are similar to system and test case migration as observable behavior is to be preserved in both cases while improving the “internal” structure of what is to be refactored/migrated. As we also advocate for test case migration, the first step in (code) refactoring is to provide a solid set of test cases that should ensure that the observable behavior is preserved after refactoring is done [FB99]. The refactored code is considered to be correct as long as all test cases pass. In this manner, the test cases can serve as a safeguard for code refactoring.

Concerning test case refactoring, Deursen et al. [DMBK01] describe a set of *test smells* that indicate problems in test code together with a set of test refactorings, which describe how to overcome some of these problems by applying small modifications. They do not, however, describe how to validate the correctness of the refactoring process.

The refactoring patterns proposed by Meszaros [Mes07] are presented as "safe refactorings" that should minimize the risk of introducing a change in the behavior of test cases. To additionally minimize this risk, Meszaros also suggests that major refactorings should be avoided. Similarly to Deursen et al. [DMBK01], however, the question of validation is not handled.

Guerra et al. [GF07] also propose a set of test case refactoring patterns and do propose a validation approach that can be applied after a particular refactoring pattern is applied. The general idea is to identify whether behavior, in their case defined on a level of test suites, is unchanged after the test refactoring is applied. To do this, the behavior is characterized as a set of *verifications*, i.e., a combination of one assertion and all the actions that appear before the assertion that is related to the same assertion's test target [GF07]. As long as two test suites perform the same verifications, they are considered to be equivalent.

Refactored/migrated test cases can be faulty in two distinct ways: either as false positives (rejecting correct system behavior) or as false negatives (accepting incorrect system behavior). Avoiding the latter appears to be particularly challenging and has been addressed via mutation testing [LS78] in the context of test case refactoring.

All mentioned approaches have served as inspiration but cannot be directly applied to the task of validating the coupled system and test case migration; the additional challenge being that *both* system and test cases are typically changed in the migration process.

Concerning behavioral equivalence as a notion of correctness, Park [Par81] suggests using bisimulation to define the behavioral equivalence of test cases. The work presented by Makedonski et al. [MGN09] addresses the issue of validation of behavioral equivalence in the domain of TTCN-3 [GHR^{+}03] by showing that the observable behavior of test cases before and after refactoring remains unchanged. To do this they apply weak bi-simulation, which compared to strong bi-simulation, ignores the internal actions that occur between observable events [MGN09]. It is questionable, however, if these approaches can be directly applied to test case migration, which is always coupled with a corresponding system migration.

● fulfilled ◑ partly fulfilled ○ not fulfilled	R 5.1	R 5.2	R 5.3
Deursen [DMBK01]	○	◑	○
Meszaros [Mes07]	○	◑	○
Guerra and Fernandes [GF07]	◑	●	◑
Makedonski et al. [MGN09]	◑	●	◑

Figure 3.9 Evaluation of selected test case validation approaches against requirements

In summary, as shown in Figure 3.9, the existing validation approaches address just partly the three defined requirements. The first two approaches deal with the problem of refactoring and provide some tooling, thus addressing the requirement **R5.2** up to some extent. The last two approaches fulfill all the requirements, the requirement regarding automation (**R5.2**) fully, and the other ones partially. However, due to the highly complex domain of test case co-migration, none of the existing approaches was directly applicable. However, they served as a basis to define the basic concepts for migration validation and as inspiration to look for alternative techniques. In the end, we came up with a novel approach for validating a test case migration by applying mutation analysis as the main goal was to identify possible false positives and false negatives among the migrated test cases.

3.4 Summary

In this chapter, we introduced the test case co-migration scenario wherefrom the problem statement in this thesis originated. Then, on this basis, we derived corresponding requirements that a solution concept should address. Lastly, we introduced related work, i.e. we introduced existing approaches regarding the test case co-migration scenario and evaluated them against the specified requirements. As the evaluation of related work showed, that existing approaches have various shortcomings and an integrated solution for test case migration is still missing.

Firstly, we introduced the co-migration scenario of this thesis in Section 3.1. We described how the system migration and the test case migration are coupled when one migrates the test cases along with the system. We described both the system and test environments on an architectural level and described their differences. In Section 3.2, we discussed a set of requirements that arise due to the considered co-migration scenario. These requirements should be fulfilled by a solution concept in order to be applicable in the co-migration scenario. In Section 3.3, we introduced the related work of this thesis. More specifically, we introduced and classified existing test quality evaluation approaches, evolution approaches, migration and reengineering frameworks, and migration validation approaches that could have been applied in the co-migration scenario. In the end, we evaluated them against the previously identified requirements and various shortcomings were identified. Additionally, we also discussed how the solution concept of this thesis addresses these shortcomings, i.e., how it addresses the requirements.

All in all, the analysis of the related work regarding the different migration phases has shown that an integrated, end-to-end solution is not available. Moreover, zooming into each of the phases, one can see that even there no established approaches, methodologies or

frameworks could be found. However, the existing work served as a solid starting point in coping with the complex problem of test case co-migration.

Starting with the first phase, the pre-migration phase, where a test case quality evaluation has to be performed, we have identified two approaches, namely, the adaptation of the ISO/IEC 9126 quality model for test specifications [ZVS+07] and the Model Quality Plan (MQP) approach [VE08], which lead us to a novel approach for test case quality evaluation presented in Chapter 5.

Then, regarding the migration phase, which is actually the main phase, three main related domains were analyzed, test case co-evolution, test case reengineering, and migration.

The main drawback of the existing co-evolution approaches is that they analyze incremental changes and not conceptual changes. Nevertheless, these approaches served as a good basis for understanding the problem better and eventually come to a novel approach for applying co-evolution analysis on a conceptual level by leveraging the concept modeling technique.

Two main problems regarding the test case reengineering approaches were identified: no support for explicit restructuring and no support for different abstraction levels. Namely, most of the identified approaches support only implicitly restructuring either as part of reverse engineering or forward engineering. Regarding the second problem, the missing support for different abstraction levels, the existing approaches only support a single level of abstraction regarding reverse engineering as well as forward engineering with different modeling notations and languages. However, the identified approaches still provided useful information regarding the definition of the method base, regarding the definition of relevant method fragments artifacts, activities, tools, etc.

Most of the analyzed existing migration projects and frameworks rely on software testing as a validation technique. The ARTIST framework [MKA+14] is the only approach that defines a method for the migration of test cases. Additionally, this method also provides initial steps towards supporting co-evolution as the relevant system changes are propagated to the test cases. The only drawback of this approach is that it provides a static method created for the particular migration context making it not applicable to a broader scope of migration contexts. On the other hand, the MEFiSTo approach [Gri16]) supports the specification of reusable blocks like method fragments and method patterns. So, the combination of these two approaches, lead us to a solution approach that combines the benefits of both.

Finally, regarding the post-migration phase, we analyzed validation approaches dealing with the behavioral equivalence of test cases. Even though not directly applicable to the complex domain of test case co-migration, the identified approaches served as a basis to define the basic concepts for migration validation, among which is the definition of the main

validation goal: detection of possible false positives and false negatives among the migrated test cases. This served as an inspiration to search for alternative techniques, eventually ending with a novel approach for validation of test case migration based on mutation analysis.

Part II

Solution Concept

Chapter 4

Solution Overview

In the previous chapter, we have seen that the existing approaches address certain aspects of the test case migration domain, but no solution addresses all the requirements we have previously introduced. In this chapter, we give an overview of the solution concept which addresses all these requirements. In the beginning, in Section 4.1, we explain the general idea of the solution and provide an overview of the main constituents. Then, we firstly introduce the pre-migration phase, i.e., the test case quality evaluation in Section 4.2. Thereafter, in Section 4.3, we introduce the migration phase, i.e., the phase that addresses co-evolution analysis and method engineering. Lastly, we introduce the post-migration phase, i.e., the test case migration validation in Section 4.4. These phases are then revisited and discussed in more detail in the subsequent chapters. Finally, in Section 4.5, the findings of this chapter are summarized and discussed.

4.1 Overview of the TeCoMi Framework

To enable a situation-specific co-migration of test cases, we propose a test case migration framework called TeCoMi. It as an end-to-end solution for the test case migration as it addresses the three main phases of test case migration: pre-migration, migration, and post-migration phase, as shown in Figure 4.1. Besides the migration phase, it also addresses the pre-migration phase to ensure migration of test cases with good quality and the post-migration phase in order to ensure that the migration was properly done. Figure 4.1 depicts the three different levels of the solution approach, namely the *Requirements Level*, the *Method Development Level*, and finally, the *Method Enactment Level*. As the name suggests, on the *Requirements Level*, we have the requirements which were previously stated in Section 3.2. On the *Method Development Level*, we present the approaches that we have developed in order to address those requirements. Finally, on the *Method Enactment Level*, we have the

actual steps of the end-to-end migration process that are actually supported by the approaches on the level above.

Starting with the *Pre-Migration Phase*, we have defined the first requirement *R1* regarding the quality assessment of the test cases and it was addressed by the Test Case Quality Plan approach. This approach supports the *Test Case Quality Evaluation* step of the end-to-end migration process. We describe the *Pre-Migration Phase* in Section 4.2.

Then, regarding the *Migration Phase*, we have defined three requirements in total, namely *R2* regarding the co-evolution of the test cases, then, *R2* regarding the automated transformation of the test cases, and finally, *R4* regarding the situativity of the test case migration methods. These requirements were addressed by the Situational Method Engineering approach which also incorporates co-evolution analysis. This approach supports the *Test Case Migration* step of the end-to-end migration process. We describe the *Migration Phase* in Section 4.3.

Finally, regarding the *Post-Migration Phase*, we have defined the *R5* requirement which relates to the validation of the test case migration. This requirement was addressed by the Migration Validation approach, which relies on Mutation Analysis. This approach supports the *Test Case Migration Validation* step of the end-to-end migration process. We describe the *Post-Migration Phase* in Section 4.4.

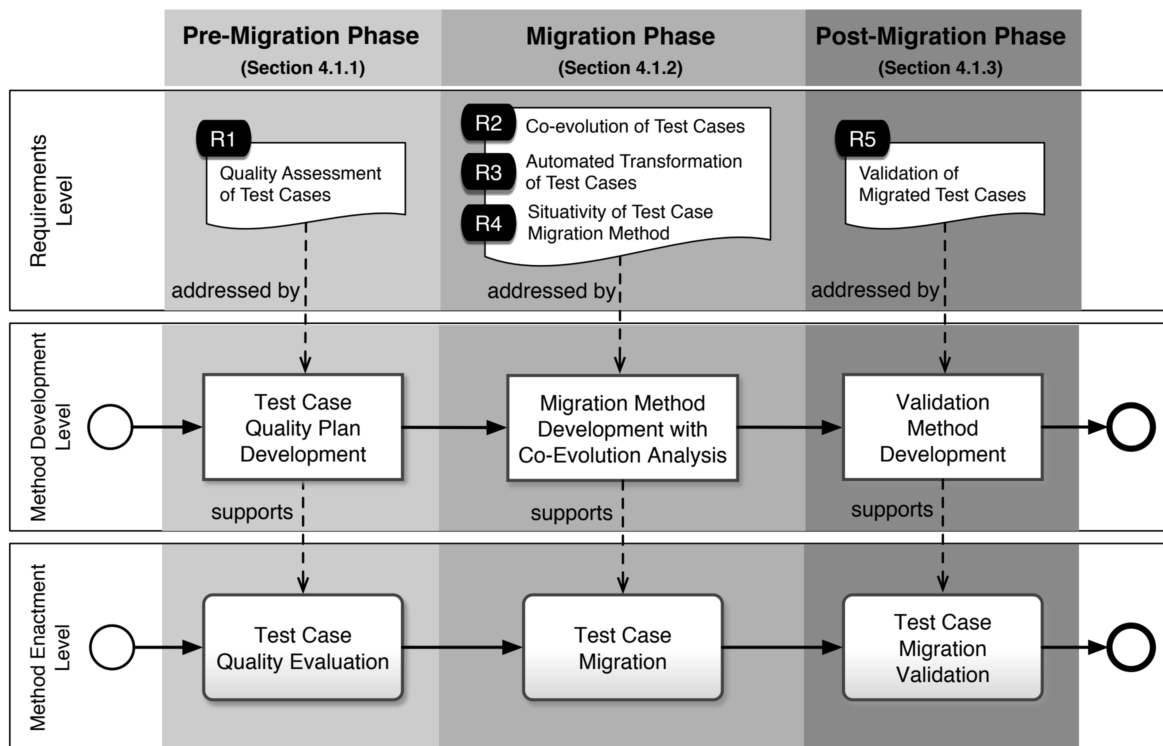


Figure 4.1 Overview of the different levels of the solution approach

4.2 Pre-Migration Phase: Test Case Quality Evaluation

In this section, we introduce the first phase of TeCoMi, namely the quality evaluation of test cases. Before performing any activity towards the migration of the test cases, their quality needs to be evaluated. As shown in Figure 4.2, the activities are split into two main disciplines, namely *Test Case Quality Plan Development* and *Test Case Quality Evaluation*.

The activities of the first discipline rely on our approach called *Test Case Quality Plan (TCQP)* [JNES18] that has been developed as part of this thesis. The TCQP approach provides a systematic process that considers the context information and integrates a standardized quality model. It consists of a top-down process, called *TCQP Process*, and a related metamodel, called *TCQP Metamodel*. The *TCQP Process* serves as a guideline for establishing a quality plan for the quality evaluation of test cases, whereas the *TCQP Metamodel* contains all relevant information concerning the quality plan. So, first of all, a suitable quality plan is being developed for a given migration context by performing the activities *Context Characterization* and *Test Case Quality Plan Creation*. These two activities basically execute the activities defined in the *TCQP Process*, which comprises four main activities, as shown in the upper part of Figure 4.2.

As part of the *Context Characterization* activity, firstly the context information specific to a given set of test cases is identified. This corresponds to the TCQP's *Characterization of Context* activity. Context factors are essential elements that may affect the outcome of the evaluation. Therefore, identifying the relevant test case context factors which include the environment, domain, and associated artifacts, assists in selecting suitable measures for evaluating the test cases [PWGW08]. The test case-relevant context factors are defined by the *Context Metamodel*.

As part of the *Test Case Quality Plan Creation* activity, the next three activities of the TCQP process are being executed, namely *Identification of Information Needs*, *Definition of Common Understanding*, and *Definition of Measurement*.

During the *Identification of Information Needs* activity, the quality goals for the evaluation of test cases are documented. By documenting the *information needs*, an insight necessary to manage the objectives, goals, and risks related to a specific quality goal of the test cases is identified and documented. Further, each identified goal is refined into a question. Both the goals and questions are specified through interviews and structured brainstorming sessions with the stakeholders. The corresponding metamodel for this activity is the *Information Need Metamodel*.

The quality goals and their related quality focus had to be described in common terms so that everyone who is involved in the evaluation has the same perception of the term quality. In the *Definition of a Common Understanding* activity, we utilize the Quality Model for Test

Specification [ZVS+07] for establishing a common quality understanding. With this general quality understanding the stakeholders would not understand quality characteristics like *Test Effectivity* or *Usability* differently. Further, the goals defined with a certain quality focus are mapped to quality (sub-)characteristics of the standardized model. For the identified questions, corresponding quality attributes are identified and documented.

Lastly, in the *Definition of Measurement* activity, suitable *measures* are documented for the quality attributes identified in the previous phase. The measures are specified according to the ISO/IEC 15939 standard [ISO02]. This standard provides the Measurement Information Model (MIM) that defines what has to be defined during measurement planning and evaluation. The standard defines base measures, derived measures, and indicators. The corresponding metamodel for this activity is the *Measurement Metamodel*. As we have

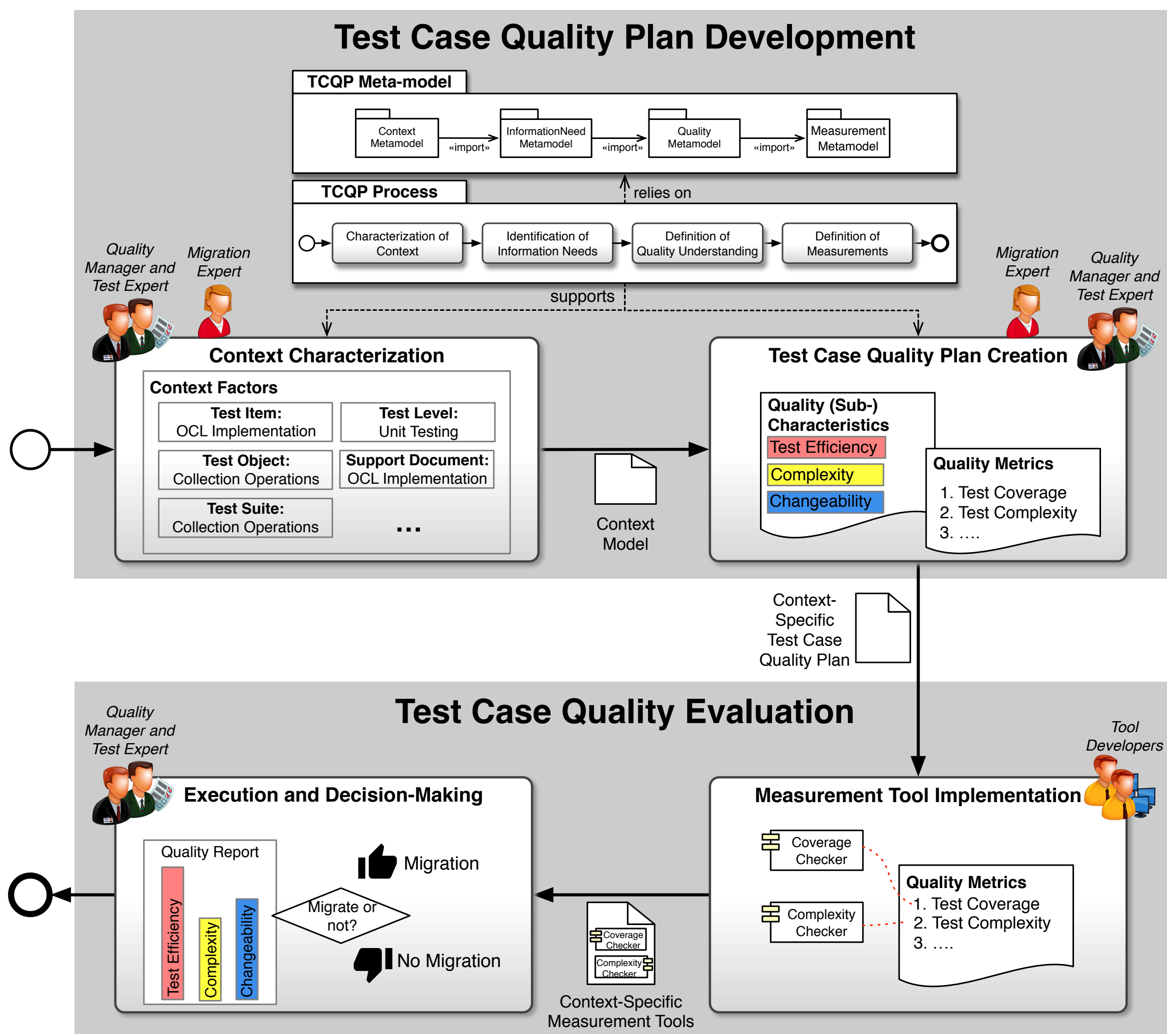


Figure 4.2 Overview of the pre-migration phase, i.e., the test case quality evaluation

seen, each TCQP process activity has a corresponding metamodel package in the TCQP metamodel. The first two activities of the process regarding the context characterization and test case quality plan creation are supported by *TCQEval (Test Case Quality Evaluator)*, an Angular[1] application that has been implemented as part of this thesis.

Once the quality plan is prepared, the necessary tools have to be implemented so that the quality evaluation of the test cases is performed automatically. For this reason, the activities of the *Test Case Quality Evaluation* discipline are performed. Based on the developed *Context-Specific Test Case Quality Plan*, as part of the *Measurement Tool Implementation* activity, the required tools for the specified measures are developed. At this step, either existing tools are identified and reused, or new tools are developed. We assume that associated tool developers use the specification of measures and indicators as some kind of guidance. Having the tools implemented, the test case quality plan can be executed as part of the last activity of the whole process, namely, *Execution and Decision-Making*. The outcome of the execution, i.e., the quality report is then analyzed, and the decision is made whether to migrate the test cases and which of them or not. This last activity is also supported by the *TCQEval* tool which supports the import of the evaluation results and visualization in terms of dashboards regarding the different quality aspects.

The TCQP approach as well as its usage in TeCoMi is discussed in detail in Chapter 5.

4.2.1 Roles

The test case quality evaluation process consists of four core activities and such segmentation provides a clear separation of concerns regarding the required expertise. For instance, for the development of a test case quality plan knowledge of software testing and software migration is required, whereas for the actual execution of the developed quality plan tool development skills are required. As shown in Figure 4.2, the core activities are associated with their respective roles.

The first two activities deal with the development of the test quality plan and both activities are performed by persons in the role of *Test Expert* and *Migration Expert*. These experts need to have knowledge of the source and the target system and test environment as well as software migration so that the context could be characterized properly. Firstly, the knowledge of the involved test environments is required to systematically identify the context factors, e.g., the identification of the characteristics of the original test cases. Then, through a series of interviews, the actual quality goals are identified and expressed in terms of quality attributes. Finally, for each quality attributes quality metrics are defined. To perform

[1]https://angular.io/

the construction of the quality plan, knowledge of quality assessment is required, which are provided by a person with the role of *Quality Manager*.

The third activity, namely the *Measurement Tool Implementation*, is performed to develop the tool to automate (part of) the quality assessment. This activity is performed by one or multiple persons in the role of a *Tool Developer* who needs to know about developing a tool for static code analysis. This knowledge is necessary in order to implement a tool addressing the metrics previously defined in the quality plan.

The fourth and last activity, namely the *Execution and Decision-Making*, is performed to actually perform the quality assessment by enacting the developed test quality plan supported by the developed tool. A person in the role of a *Software Developer* executes the quality plan and analysis of the obtained quality report. Additionally, a person in the role of a *Test Expert* is involved in order to support the decision making on the migration of the test cases.

4.3 Migration Phase: Co-Evolution Analysis and Method Engineering

The migration of the test cases comes down to the problem of co-migration, i.e., the test cases have to be migrated along with the system as their migration is dependent on the system migration. In order to address the previously mentioned challenges, based on the *Method Engineering Framework for Situation-Specific Software Transformation Methods (MEFiSTo)* [Gri16], we provide a solution that combines techniques from *Situational Method Engineering (SME)* [HSRÅR14] and *Software Evolution* [MD08].

Figure 4.4 depicts an overview of our method engineering process whose activities are split into two main disciplines: *Method Development* and *Method Enactment*. Besides the *Method Engineering Process*, another integral part of the solution approach is the *Method Base*. In the following, we describe both the *Method Base* as well as the *Method Engineering Process*.

4.3.1 Method Base

The *Method Base* contains the building blocks, *Method Fragments*, *Method Patterns*, needed for assembling the test migration method. *Method Fragments* are atomic building blocks of a test migration method, e.g., an artifact or an activity. *Method Patterns* represent a proven migration strategy and define which fragments are necessary and how to assemble them. The suitability of each pattern to a certain situation is expressed by a set of characteristics. To additionally support the co-evolution analysis and to express the relation between the system

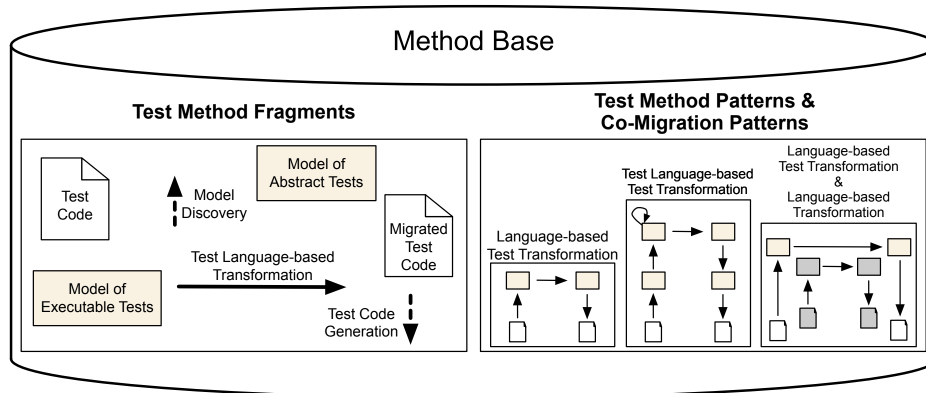


Figure 4.3 Method base: test method fragments and test method patterns & co-migration patterns

and the test case migration directly, we extend the method base with *Co-Migration Patterns*. Technically, a co-migration method pattern is a combination of a test method pattern and a system method pattern, visually represented as a double horseshoe model.

4.3.2 Method Engineering Process

Using the *Method Base*, the *Method Engineering Process* guides the development and the enactment of the situation-specific test migration method. By performing activities of the *Method Development* discipline, a situation-specific test method gets developed. It comprises the following two activities: *Situational Context Identification* and *Transformation Method Construction*. During this activity, the situational context is analyzed and characterized from both system migration and testing perspective. Firstly, in the *Concept Identification* activity, both the source and the target tests and system are represented as a set of concepts by applying concept modeling. Then, based on this concept representation in terms of a *Concept Model* the impact of the system changes on the test cases is identified and captured in terms of an *Impact Model* in the *Co-Evolution Analysis* activity. Lastly, as part of the *Influence Factor Identification* activity, the influence factors are identified. Having the context information collected in terms of a *Situational Context Model*, the *Method Construction* activity can be initiated and a situation-specific test migration method gets constructed.

The overall outcome of *Method Development* is a *Situation-Specific Test Migration Method Specification* which defines how to do the migration by defining the activities to be performed and the artifacts that should be generated. The two activities belonging to the

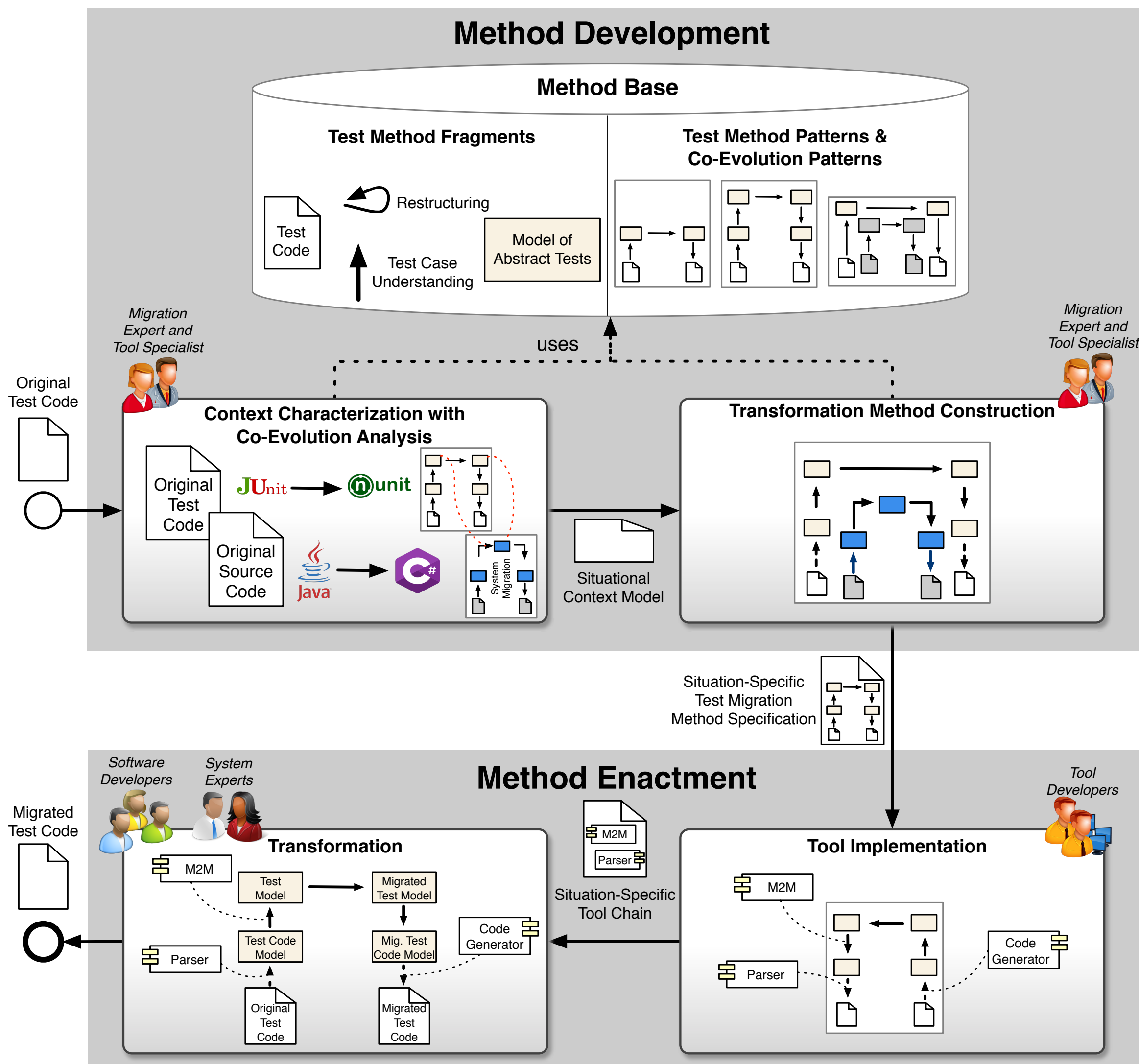


Figure 4.4 Overview of the method engineering process

Method Development discipline are supported by a graphical editor that has been implemented as part of this thesis in Eclipse Sirius[2].

During the first activity of *Method Enactment*, namely the *Tool Implementation*, a *Situation-Specific Toolchain* is developed that is required for the automation of the migration method, e.g., a parser or a code generator. Thereafter, during the *Transformation* activity, the test migration method is enacted as defined in the test migration method specification.

The method base and the method engineering process are discussed in detail in Chapter 6.

[2]https://www.eclipse.org/sirius/

4.3.3 Roles

The method engineering process consists of four core activities which are common for SME approaches including the MEFiSTo framework [Gri16]. Also, such segmentation provides a clear separation of concerns regarding the required expertise. For instance, for the development of the transformation method, a knowledge of software migration is required, whereas for the enactment of the developed method tool development skills are required. As we clearly distinguish the core activities, we enable the inclusion of external experts. As shown in Figure 4.4, the core activities are associated with the respective roles.

The first two activities deal with the development of the transformation method and both activities are performed by a person in the role of a *Migration Expert*. The expert needs to have knowledge of the source and the target system and test environment. Moreover, knowledge of software migration and co-evolution as well as method engineering is necessary. Firstly, the knowledge of the involved system and test environments is required to systematically identify the situational context, e.g., the identification of the characteristics of the original test cases or the characteristics of the migrated system. Then, the knowledge of migration enables relating the identified test context to the characteristics of different transformation strategies, namely test method patterns stored in the method base. Furthermore, the knowledge of co-evolution enables relating the identified test context to the identified system concepts.

To perform the construction of the test transformation method, e.g., selection and configuration of the transformation method, method engineering skills are required. Additionally, a person with the role of a *Tool Specialist* is also involved. So, the *Migration Expert* needs to have knowledge of both reengineering tools and method engineering. Knowledge about reengineering tools is important for the effort assessment of adapting or developing tools. More precisely, knowledge of model-driven reengineering tools is necessary as the solution framework is focused on the construction of model-driven toolchains. To specify the use of the tools or their adaptation, method engineering skills are required.

The third activity, namely the *Tool Implementation*, is performed to develop the toolchain to automate (part of) the transformation. This activity is performed by one or multiple persons in the role of a *Tool Developer* who needs to have knowledge of model-driven engineering developing reengineering tools. This knowledge is necessary in order to implement a toolchain as previously defined by the transformation method specification.

The fourth and last activity, namely the *Transformation*, is performed to actually transform the original test cases by enacting the developed test transformation method supported by the developed toolchain. General development skills are required for this activity, provided by one or multiple persons in the role of a *Software Developer*. It can be required to additionally

include *Test Experts* with specific knowledge, e.g., regarding decisions about the resulting test architecture.

4.4 Post-Migration: Migration Validation

Test case migration is the process of transferring test cases to a new environment without changing the expected system behavior they assert. As migrated test cases are used to validate system migration, validating test case migration is clearly important. The main goal in the validation phase is to identify false positives and false negatives among the migrated test cases. A migrated test case is considered to be a false positive if it fails and the system is correct. Similarly, a migrated test case is considered to be a false negative if it passes but the system is incorrect.

Motivated by the idea of the application of mutation analysis in test code refactoring, we propose a novel method for validation of software migration. The central part of our approach is the mutation analysis repository which contains mutation analysis scenarios, predefined mutation operators, and mutation patterns. Based on what is being mutated, we have defined six mutation analysis scenarios in total. Each scenario is defined by a set of assumptions and indications which describe how to perform mutation analysis and how to interpret the results. Besides the scenarios, we provide predefined mutation operators, which are split into three main groups, namely test mutation operators, language mutation operators, and domain specific mutation operators. Lastly, the repository contains a set of mutation patterns, which represent construction guidelines for mutation methods and follows a certain strategy

As shown in Figure 4.5, the activities of the migration phase are split into two main disciplines, namely *Validation Method Development* and *Validation Method Enactment*. As part of the *Validation Method Development* discipline, firstly the *Context Characterization* activity is performed, as the validation method being developed should be suitable to the context. For this purpose, the artifacts created in the previous phase, namely, e.g., *Situational Context Model*, can be reused. Based on this *Context Information*, a suitable mutation method is being constructed as part of the *Mutation Method Construction* activity. Firstly, a mutation analysis scenario is selected as well as mutation operators. Based on these selections and also on the previously identified context information, a mutation method pattern is selected and configured. The outcome of this activity is a *Context-Specific Mutation Method Specification* As part of *Validation Method Enactment*, firstly the *Mutation Tool Implementation* activity is performed in order to develop the necessary tools that should automate the mutation process. For example, depending on the functionality being asserted in the test cases, a set of mutation

operators in terms of model transformations has to be implemented. To improve the process of automated mutation and to increase the reuse of existing components, we introduce a flexible and extensible model-driven mutation framework. The mutation framework should serve as a project-independent tool infrastructure and it is based on the *eMoflon*[3] transformation framework, which is a tool suite for applying Model-Driven Engineering (MDE) and provides visual and formal languages for (meta)modeling and model management.

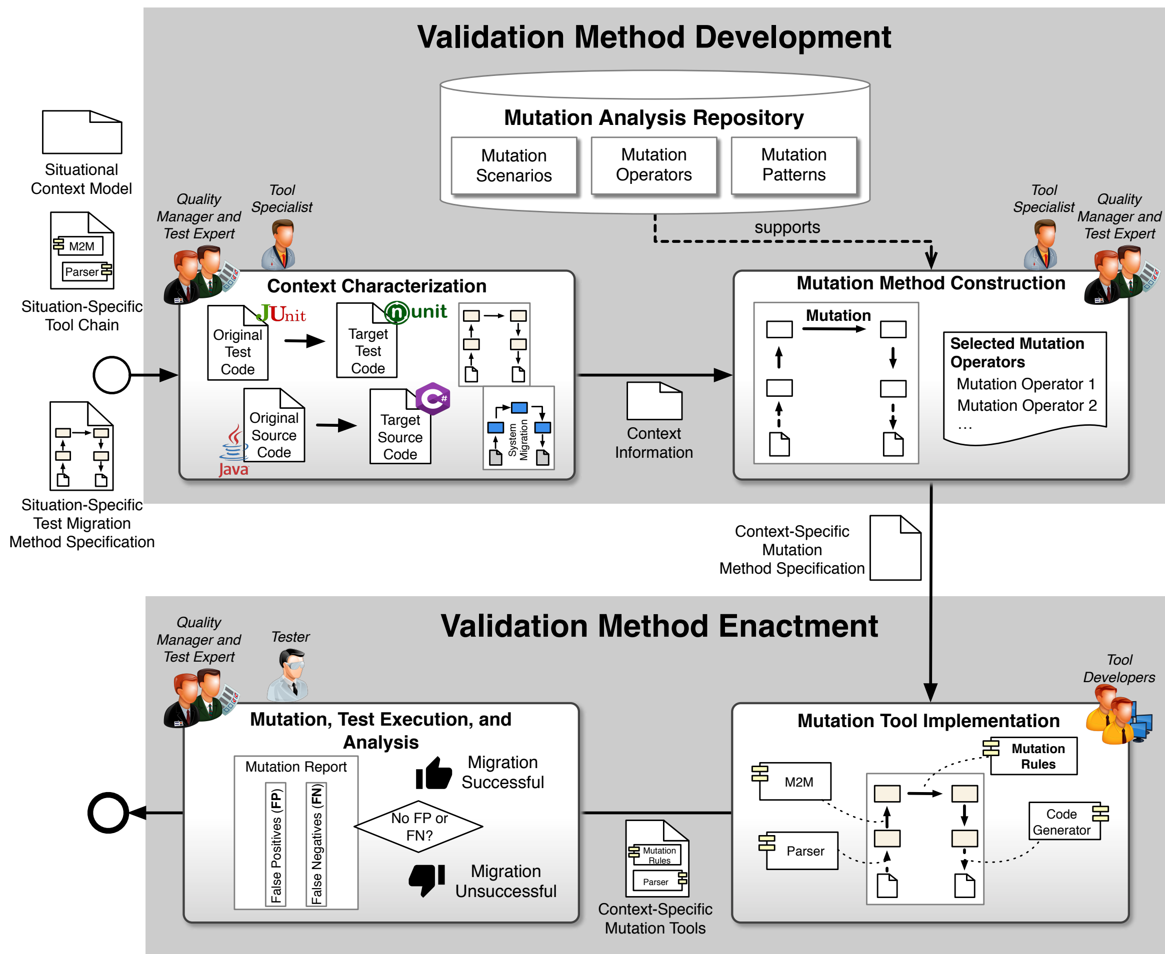


Figure 4.5 Overview of the post-migration phase, i.e., the migration validation

Lastly, the *Mutation, Test Execution, and Analysis* activity is performed. The created mutation method together with the developed tools are enacted, resulting in mutated test cases. Thereafter, the mutated test cases are then executed against the migrated system. The outcome of the test case execution of the mutated test cases is then analyzed. If no false positives and false negatives are identified, the migration of the test cases is considered to be

[3]https://emoflon.org/

successful. Otherwise, the identified false positives and false negatives have to be analyzed in order to fix them. The migration validation approach is discussed in detail in Chapter 7.

4.4.1 Roles

The migration validation process consists of four core activities thus providing a clear separation of concerns regarding the required expertise. For instance, for the development of a validation method knowledge of software testing, more specifically mutation testing is required, whereas for the actual execution of the developed migration method tool development skills are required. As shown in Figure 4.5, the core activities of the first phase are associated with their respective roles.

The first two activities deal with the development of the mutation method and both activities are performed by persons in the role of *Quality Manager* and *Test Expert*. These experts need to have knowledge of the source and the target system and test environment as well as software migration so that the context could be characterized properly. Firstly, the knowledge of the involved test environments is required to systematically identify the context, e.g., the identification of the characteristics of the original test cases. Then, based on this knowledge, a mutation method is getting constructed specifically for the identified context. To perform the construction of the mutation method, knowledge of the quality assessment is required, which are provided by a person with the role of *Quality Manager*.

The third activity, namely the *Mutation Tool Implementation*, is performed to develop the tool to automate (part of) the migration validation. This activity is performed by one or multiple persons in the role of a *Tool Developer* who needs to have a knowledge of developing tools for mutation analysis. This knowledge is necessary in order to implement a tool addressing the mutation operators previously defined by the method.

The fourth and last activity, namely the *Mutation, Test Execution, and Analysis*, is performed to actually perform the mutation by enacting the developed method, thus creating mutants and then executing the tests. A person in the role of a *Tester* executes the mutation method and then the obtained test cases. Persons in the role of a *Test Expert* and *Quality Manager* are involved in order to support the decision on the success of test migration.

4.5 Summary

Currently, there is no end-to-end test case migration approach that supports all three migration phases. In this chapter, we introduced the solution concept which addresses this problem. In Section 4.1, we introduced the solution concept which is represented in terms of a

framework that addresses the three main phases in test case migration. Firstly, it provides a method for test case quality evaluation based on which a decision is made whether to migrate the test cases or not (Section 4.2). Then, by using the method engineering process, which considers co-evolution analysis and the method base, a situation-specific test case migration method is developed and enacted (Section 4.3). Finally, a validation method is developed and enacted to assess the success of the test case migration (Section 4.4).

In the next three chapters, we will go into detail on the three main phases of the framework. In Chapter 5, details on the pre-migration phase, i.e., test case quality evaluation are given. Then, details of the migration phase, i.e., the method engineering process and the method base are described in Chapter 6. Finally, details on the post-migration phase, i.e., the migration validation are provided in Chapter 7.

Chapter 5

Pre-Migration Phase: Test Case Quality Evaluation

In the previous chapter, an overview of the TeCoMi framework for the construction of situation-specific migration methods was given. In this chapter, we introduce the first phase of the solution approach, namely the *Test Case Quality Evaluation*. In Section 5.1, we introduce our approach for test case quality evaluation, the Test Case Quality Plan (TCQP) defined by its process and the corresponding metamodel. In Section 5.2, we introduce the actual process for the quality evaluation of test cases which relies on the TCQP approach. As already introduced in the previous chapter, it consists of four core activities: *Context Characterization*, *Test Case Quality Plan Creation*, *Measurement Tool Implementation*, and *Execution and Decision-Making*. Finally, the findings of this chapter are summarized in Section 5.3.

5.1 The Test Case Quality Plan Approach

In this section, we introduce our approach for quality evaluation of test cases, called *Test Case Quality Plan (TCQP)* [JNES18]. This approach provides the basis for the activities in the pre-migration phase. TCQP builds upon the *Model Quality Plan (MQP)* approach [VE08] which is relevant in the domain of software models. To apply conceptually the MQP approach in the domain of test cases, we provided a new metamodel relevant to the domain of test cases. The TCQP approach consists of a top-down process, called *TCQP Process* and a related metamodel, called *TCQP metamodel* (Figure 5.1). The *TCQP process* serves as a guideline for establishing a quality plan for the quality evaluation of test cases. The *TCQP metamodel* contains all relevant information concerning the quality plan and it is structured

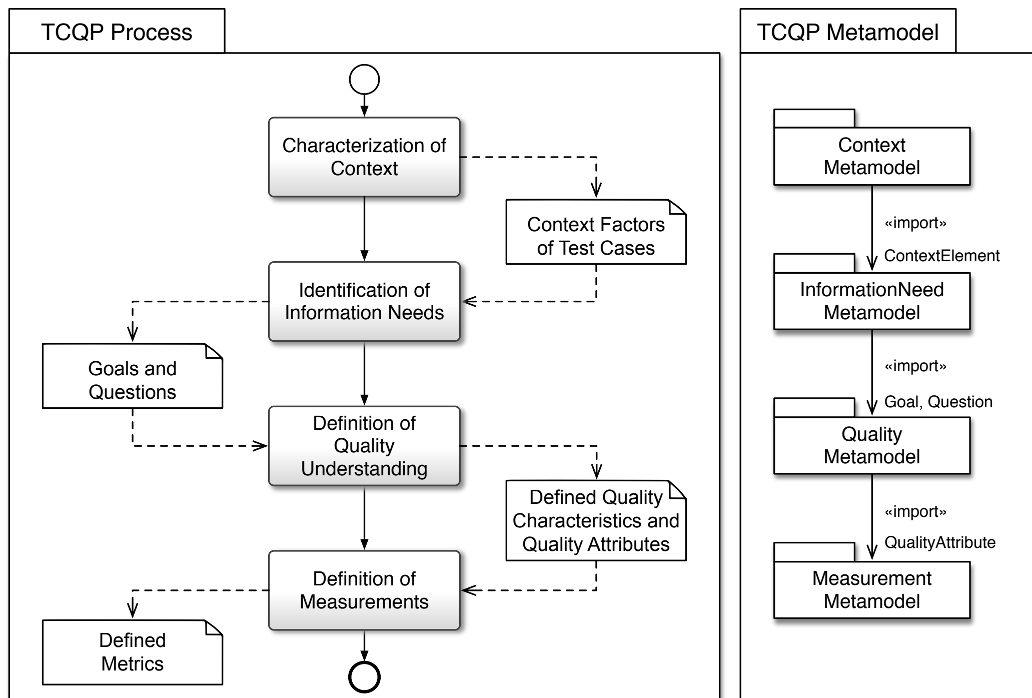


Figure 5.1 Test Case Quality Plan Process.

into packages which are linked to their respective activities in the *TCQP process*. In the following, we give an overview of the process steps and the corresponding metamodels.

The TCQP process, shown on left in Figure 5.1, consists of four activities, similar to the MQP process [VE08]. As can be seen on the right side of Figure 5.1, each of the four activities is supported by a suitable metamodel.

Firstly, the *Characterization of Context* activity involves identifying the context information specific to a given set of test cases. Context factors are essential elements that may affect the outcome of the evaluation. Test cases are usually derived from requirement specifications, directly from the structure of a component or system or they can be also based on tester's experience and intuition. Identifying the relevant test case context factors which include the environment, domain, and associated artifacts, assists in selecting suitable measures for evaluating the test cases [PWGW08]. The test case relevant context factors are defined by the *Context Description Metamodel* shown in Figure 5.2.

Secondly, any successful evaluation is performed towards an explicitly stated purpose. In the *Identification of Information Needs* activity, the quality goals for the evaluation of test cases are documented. By documenting the *information needs*, an insight necessary to manage the objectives, goals, risks, and problems related to a specific quality goal of the test cases is identified and documented. As identifying the *information needs* is a creative

process and requires a significant human resource, the context factors determined in the previous activity are used as an additional input. The GQM (Goal Question Metric) [BCR94] approach is used to select the insights mentioned above targeting a specific quality goal. The corresponding metamodel for this step is the *Information Need Metamodel* and an excerpt of it is shown in Figure 5.4.

Third, the quality goals and their related *quality focus* had to be described in common terms, so that everyone who is involved in the evaluation has the same perception of the term quality. In the *Definition of a Common Understanding* activity, we utilize the Quality Model for Test Specification [ZVS+07] for establishing a common quality understanding. As the quality model for test specifications presented in [ZVS+07] relies on the ISO/IEC 9126 [ISO01] quality standard, which was replaced in 2011 by the new ISO/IEC 25010 [ISO11b], we have compared the differences and extended the quality model for test specifications.

Fourth, in the *Definition of Measurement* activity, suitable *measures* are documented for the quality attributes identified in the previous activity. The *measures* are specified according to the ISO/IEC 15939 standard [ISO02]. This standard provides the Measurement Information Model (MIM) that helps in determining what has to be defined during measurement planning and evaluation [ISO02].

Having the brief overview of the process and the corresponding metamodels, in the following sections, we explain each process activity in detail. Additionally, each part of the metamodel is explained in detail when the corresponding process step is discussed.

5.1.1 Characterization of Context

In this section, we present the first activity of the TCQP process for evaluating the quality test cases which involves the identification of the context information specific to a given set of test cases. Firstly, we present the context metamodel which includes a set of context factors along with their description as well as the relation between them. Thereafter, we discuss the objectives, input documents, process description, and the results of this activity.

Context Metamodel

The overview of the *Context Metamodel* in Figure 5.2 shows the identified relevant concepts and the relation between them. Each of the identified concepts represents a context factor as shown in Figure 5.3. The central class in the metamodel is, intuitively, the *Test Case* and therefore, it is important to determine the deriving sources of these test cases. Although we consider the complete test suite for measurement, the context description is about one

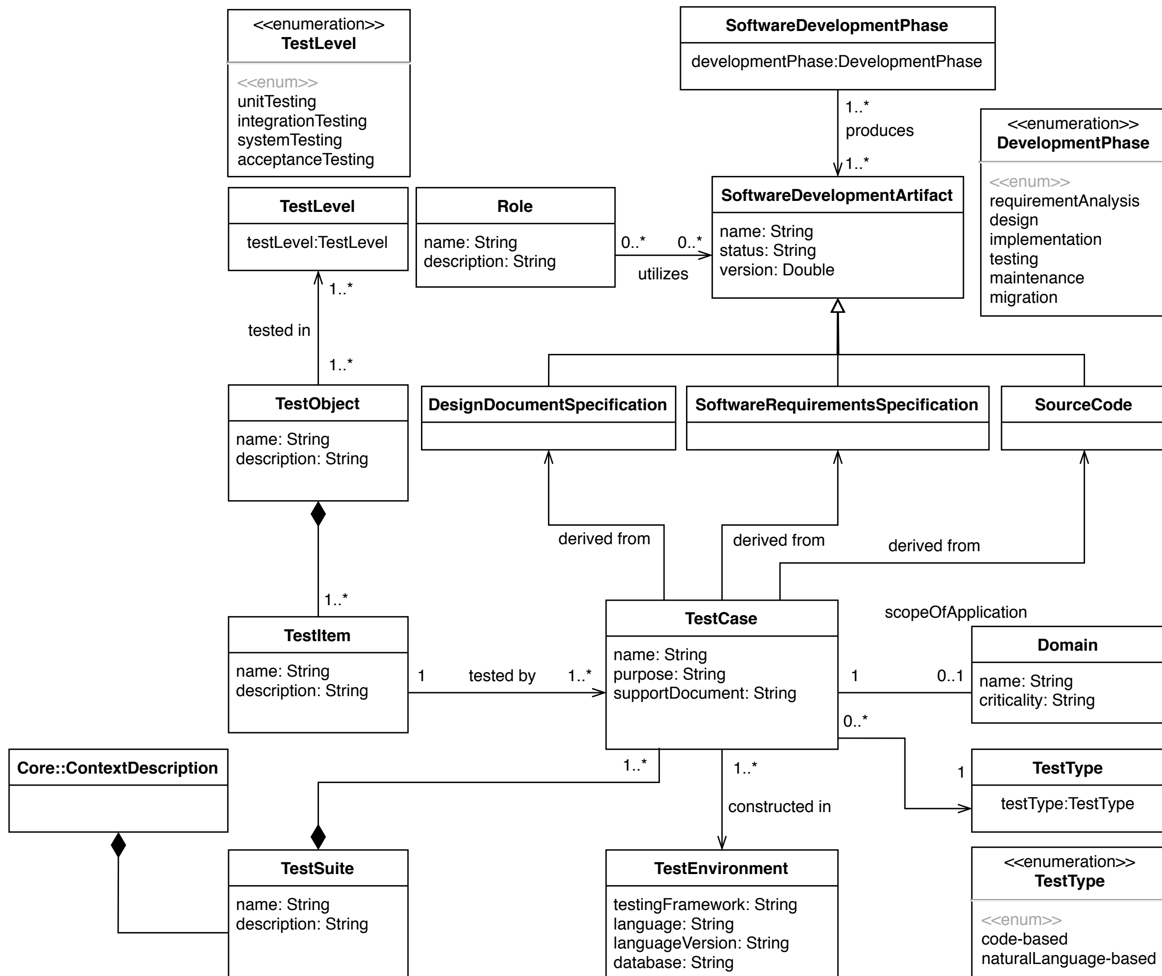


Figure 5.2 The context description metamodel

test case belonging to a particular context. In the following, we provide definitions for each context factor:

Definition 1. ***Software Development Artifact:*** *A software development artifact is a tangible by-product or a product produced during a software development phase [PP16]*

Definition 2. ***Software Development Phase:*** *Software development is the set of phases that results in software products. In each software development phase, a variety of activities take place [PP16]*

Definition 3. ***Software Requirements Specification:*** *A software requirements specification (SRS) is a complete description of the requirements that a software system to be developed must or should fulfill [ISO09].*

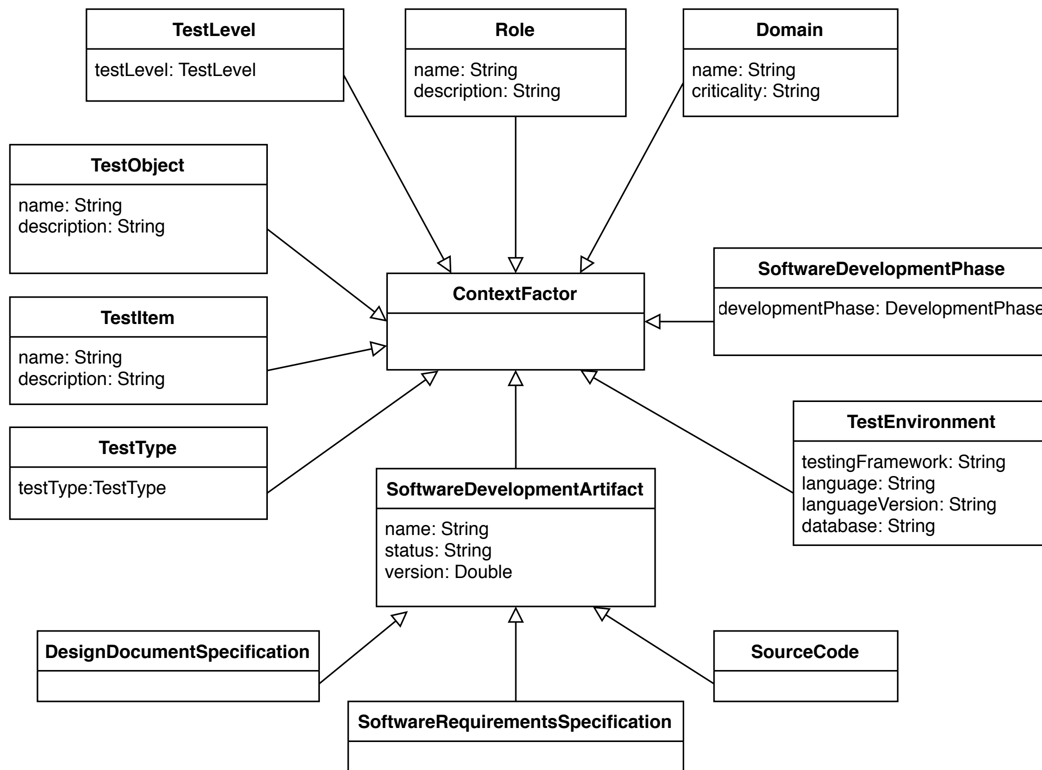


Figure 5.3 Context factors for test cases

Definition 4. ***Source Code:*** *Computer instructions and data definitions expressed in a form suitable for input to an assembler, compiler, or other translator. Note: A source program is made up of source code [IEE91].*

Definition 5. ***Role:*** *A Role depicts a set of related skills, competencies, and responsibilities in a software development process [OMG08].*

Definition 6. ***Test Case:*** *A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [IEE91].*

Definition 7. ***Test Suite:*** *A set of several test cases for a component or system under test, where the postcondition of one test is often used as the precondition for the next one [IST].*

Definition 8. ***Test Level:*** *A group of test activities that are organized and managed together. A test level is linked to the responsibilities in a project. Examples of test levels are component test, integration test, system test and acceptance test [IEE91].*

Definition 9. ***Test Object:*** *The component or system to be tested [IEE91].*

Definition 10. ***Test Item:*** *A software item which is an object of testing [IST].*

Definition 11. ***Test Environment:*** *An environment containing hardware, instrumentation, simulators, software tools, and other support elements needed to conduct a test [IST].*

Definition 12. ***Software Requirement Specification:*** *Documentation of the essential requirements (functions, performance, design constraints, and attributes) of the software and its external interfaces [IEE91].*

Definition 13. ***Software Design Specification:*** *A representation of software created to facilitate analysis, planning, implementation, and decision making. The software design specification is used as a medium for communicating software design information and may be though as a blueprint or a model of the system [IEE91].*

The Process Step

The intention behind describing the context is to enable the deriving of information needs in terms of questions and quality attributes and then to find suitable measures. The *Characterization of Context* activity involves identifying the context factors specific to considered test cases. Context factors are essential elements that may affect the outcome of the evaluation. Test cases are usually derived from requirement specifications, directly from the structure of a component or system or can also be based on tester's experience and intuition. Identifying these relevant test case context factors which include the environment, domain, and associated artifacts, assists in selecting suitable measures for evaluating the test cases.

Objective: The objective of this activity is to document the context description of the selected test suite. This activity explicitly describes *in which context the test suites are measured.*

Input Document: Documentation of the project, Project Plan, Test plan

Result: TCQP *ContextDescription*

Participants: Quality Manager and Tester

Process Description: For example, the following questions can be asked:

- What is the test level of the test cases?
- What is the criticality of the domain the test cases are used in?
- From which artifact are derived the test cases?

Characterizing the context involves the stakeholders who are interested in evaluating the quality of a set of test cases. The stakeholders are usually the quality managers and testers who analyze and describe the context according to the context factors presented in Section 5.1.1. Characterizing the context serves as an additional input in defining the information needs, e.g., the goals of the stakeholders. As the identification of information needs is a creative process, the document context information would assist in deriving the quality goals specific to the context.

The *Context Metamodel*, which is depicted in Figure 5.2, shows the relations between the context factors. The object of study in our approach is the test suite as a quality statement about single test cases is hardly needed. As an initial step, it is important to determine the deriving sources of these test cases. Test cases are usually derived from artifacts such as the software requirement specifications, user manual, source codes, test results or failure reports. It is also important to identify the environment of the test which would influence later on the specification of the measures. Besides the environment, it is also important to determine what type of test cases taken are into account. Test case type defines whether the test cases are code-based or natural language-based.

5.1.2 Identification of Information Needs

In this section, we present the *Identification of Information Needs* activity where based on the context factors collected in the previous activity goals are identified. Firstly, we introduce the *Information Needs Metamodel*. Then, the goal definition template is introduced. Finally, questions and attributes related to the described context are defined.

Information Needs Metamodel

Having the context identified, the information needs are discussed and described. The identification of information needs involves various stakeholders working together to set the goals for the quality evaluation. The main concepts of this activity are shown in the *Information needs metamodel* in Figure 5.4.

This metamodel consists of a set of goals, which are refined by a question or set of questions. By using the goal template, each goal is described by the object of study, the purpose of the evaluation, the quality characteristics that are in focus, the viewpoint, and the context which is being analyzed. Each identified goal is related to a set of questions which are then related to corresponding quality attributes.

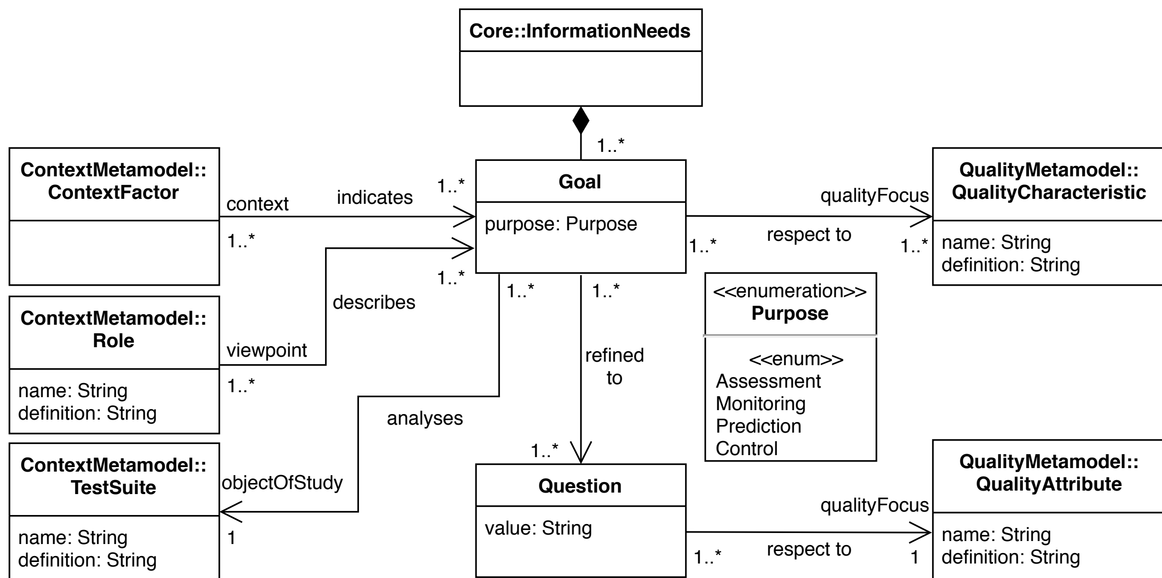


Figure 5.4 Information needs metamodel

Goal Template for the Quality Evaluation of Test Cases

In our approach, we apply GQM (Goal Question Metric) [BCR94] to define relevant goals for the quality evaluation test cases. GQM provides a systematic approach for integrating goals to models based on specific needs. Firstly, measurement goals are defined which are further refined to operational level questions and then relevant metrics are selected. The metrics provide all the necessary information for answering those questions. But before defining metrics, the corresponding quality attributes are identified. Quality attributes are atomic properties of the test cases that can be measured directly.

Goal specify *"What is to be achieved?"* and they are systematically documented by using the structured goal template. A goal template includes aspects such as object of study, purpose, quality focus, viewpoint, and context. In the following, we provide a short description of each of this aspect:

- *Object of study:* Identifies the object to be evaluated
- *Purpose:* Indicates the purpose of the quality evaluation
- *Quality Focus:* Identifies the quality characteristic to be evaluated
- *Viewpoint:* Defines the persons who are interested in the measurement results
- *Context:* Refers to the context characterized by the context factors

The Process Step

The context factors, identified in the previous activity, are used in *Identification of Information Needs* activity. Using the GQM (Goal Question Metric) approach, the insights like goals based on the quality focus are selected.

Objective: The objective of this activity is to document the information needs which describe the quality goals for the evaluation. The selected goals state *what is important to measure*. The evaluation of test cases does not depend on a fixed quality goal, but always relative to the given *QualityFocus*. The *QualityFocus*, in turn, is dependent upon the *ContextDescription*.

Input Document: Context Description from the previous activity; Existing TCQP or TCQP rules that are relevant to the context

Result: TCQP *Information Needs*

Participants: Quality Managers; Developers; Testers

Process Description: For processing this activity, the source from which the information needs could be identified is given in the following:

- The interviews among the stakeholders (managers, developers and testers)
- More refined information need is gathered through the additional input *ContextDescription*, the information need could also be independent of the *ContextDescription*
- If there are existing TCQPs or TCQP rules for similar contexts, we can reduce the effort of human resources by reusing this experience from the existing plans and rules.

The measurement goals are now determined from the goal dimension template. The next step is to refine the goals into questions and then into quality attributes. The questions focus on the object of study and characterize the evaluation of a specific goal. This characterization would help in finding a suitable quality attribute for the given goal. Figure 5.4 also shows the relation between the *Information Needs Metamodel* and the *Context Description Metamodel* and the *Quality Metamodel*. A goal is specified for the context and is then refined by questions. Each question is described concerning a quality attribute. The process of identifying the goals, questions, and quality attributes is performed through conducting interviews with the stakeholders. The goals are further refined into questions by structured brainstorming sessions and other similar methods.

5.1.3 Definition of a Common Quality Understanding

In this section, we present how a quality model for test specifications is used to define a general interpretation of the term quality related to test cases. Firstly, we present the extended quality model for test specification based on the ISO/IEC 25010 quality model. Thereafter, we illustrate the process of defining a quality model with respect to the information needs defined in the previous step.

Extended Quality Model for Test Specification

Quality models are well-accepted means to evaluate and set goals for the quality of a software product [DJLW09]. The term quality is decomposed by these models into quality characteristics and can be further subdivided into more specific quality sub-characteristics and finally into quality attributes. In contrast to the quality characteristics and sub-characteristics, attributes are atomic properties and can be measured directly.

The quality model for test specification presented in [ZVS$^+$07] is a specialization of the ISO/IEC standard 9126 quality model. This quality model focuses on the different quality aspects of test specifications. As the ISO/IEC 9126 quality model has been replaced in 2011 by the ISO/IEC 25010 quality model, we analyzed the differences and we extended the test case quality model presented in [ZVS$^+$07]. Figure 5.5 depicts the extended quality model for test cases and as can be seen it comprises eight main characteristics: *Test Effectivity*, *Reliability*, *Usability*, *Performance Efficiency*, *Security*, *Maintainability*, *Portability*, and *Reusability*. As can be seen, each main characteristic is further refined by multiple sub-characteristics. The definitions for the unchanged characteristics from ISO/IEC 9126 to ISO/IEC 25010 standard are directly reused from [ZVS$^+$07]. Those quality characteristics from the ISO/IEC 25010 quality model, which have no corresponding quality characteristics in the quality model for test specifications [ZVS$^+$07] were adapted to the test domain. For example, the *Security* characteristic is not any longer a sub-characteristic of *Reliability* characteristic, but a separate characteristic and it contains three sub-characteristics, namely *Confidentiality*, *Accountability*, and *Security Compliance*.

The quality model discussed so far is kept abstract to support the application to different types of test cases. Each main characteristic contains a compliance sub-characteristic which denotes the capability of the test cases to adhere to standards, conventions or regulations relating to the main characteristic. This quality model can be adapted to various contexts, and the characteristics can be defined with a fitting description with respect to the context. Depending on the context, the relevant quality attributes are identified with respect to these characteristics.

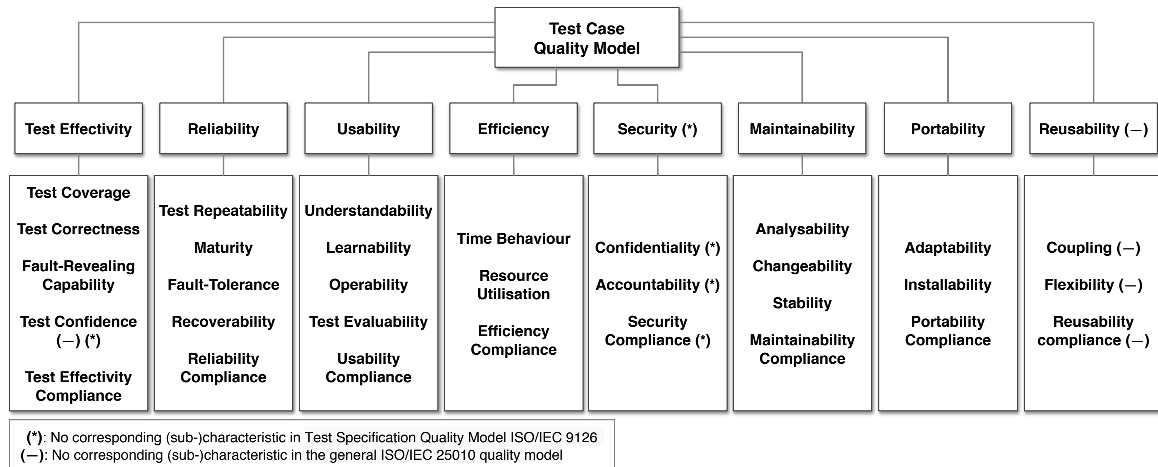


Figure 5.5 The ISO/IEC 25000 Standard-based Test Case Quality Model

Process Step

The *Definition of a Common Understanding* activity relies on the quality model previously introduced in Section 5.1.3. Defining such a quality understanding should enable the project team to have the same perception of quality related to their context. As part of this activity, we utilize the extended quality model for test cases introduced previously. Further, the goals defined with a quality focus mapped with the quality characteristics and the quality attributes are mapped with the sub-characteristics as shown in Figure 5.6.

Objective: The objective of this activity is to provide a common quality understanding of the quality goals. The quality model for test specification contains the relevant quality characteristics, sub-characteristics, and quality attributes. This activity describes *what is measured* by providing a common description that is interpretable for all the stakeholders

Input Document: *InformationNeeds* from the previous phase; Existing TCQPs or TCQP rules for a similar context; Existing quality models for test specification

Result: TCQP *QualityModel*

Participants: Quality Managers; Project Team

Process Description: As part of the *Definition of Common Quality Understanding* activity, the quality models are fully integrated into the GQM approach [Voi09]. Quality models provide a definition for the goals and attributes and acts as a link between questions and measurements. A quality model is used to specify quality understanding and the attributes

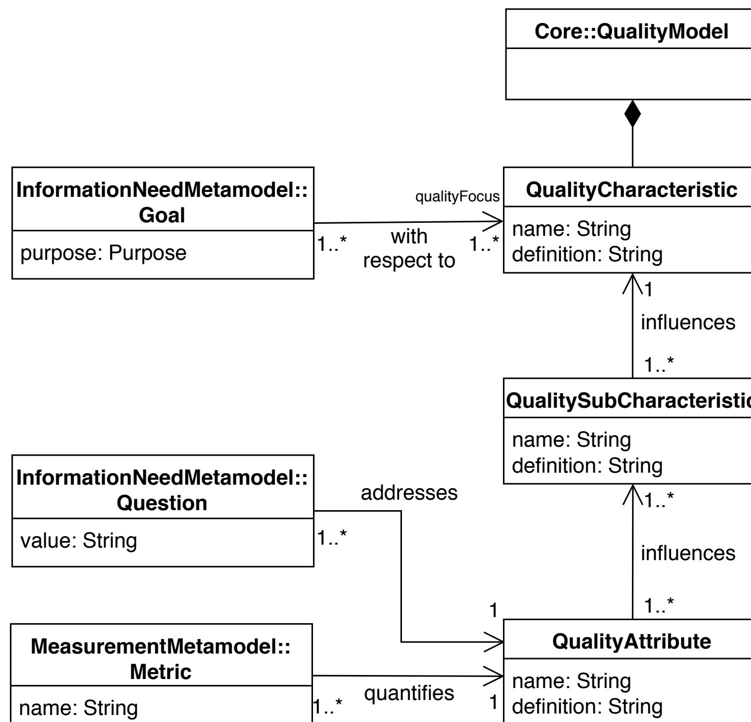


Figure 5.6 Quality Metamodel based on [Voi09] ant its relation the Information Need Metamodel and the Measurement Metamodel

of test cases that are measured as shown in Figure 5.6. Based on the quality focus defined in the goal dimension templates, the characteristics are mapped to the existing quality model by the quality manager. At the same time, the quality sub-characteristics and quality attributes are also identified.

5.1.4 Definition of Measurements

This section describes how measurements are defined for the identified quality attributes. The definition of the measurement plan is based on the Measurement Information Model (MIM) [ISO02]. Firstly, we discuss the measurement metamodel, and thereafter, we illustrate the process of defining the measures.

Measurement Metamodel

In the previous activity, for each question, a quality attribute was identified. As the quality attributes are the atomic properties of test cases, they can be quantified using measurements. hence, a measurement provides a quantitative insight into the quality of a set of test cases. By using measurements, problems can be identified as well as improvement

opportunities [ISO02]. Therefore, in this section, we address the approach for describing the measurements.

The ISO/IEC 15939 standard provides the “activities and tasks that are necessary to identify, define, select, apply, and improve measurement within an overall project or organizational measurement structure” [ISO02]. This standard defines a measurement process applicable to system and software engineering and management disciplines.

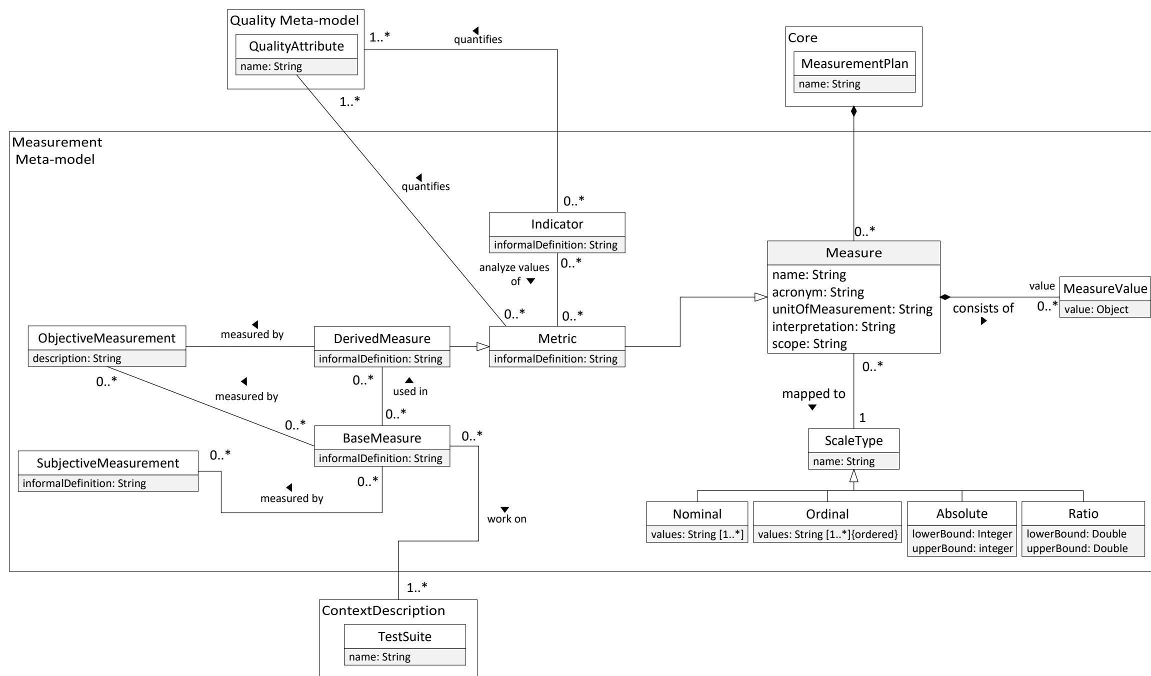


Figure 5.7 Overview of the Measurement Metamodel as defined in [Voi09]

The specification of a measurement plan in the TCQP approach is based on the ISO/IEC 15939 standard. It relies on the measurement metamodel which is shown in Figure 5.7. Mainly, a measure is described as follows: To refer to a given measure either a *Name* or an *Acronym* is used. The *Scale Type* of a measure depends on the nature of the relationship between values on the scale. According to the standard [ISO02], the commonly defined scales are *Nominal, Ordinal, Absolute, and Ratio*. *Metrics* enable the quantification of quality attributes. They are classified into three measures *base measures*, *derived measures*, and *indicators*.

A *base measure* is functionally independent of other measures. For a particular quality attribute, a *measurement method* is used to obtain a *base measure*. These methods usually work on the *object of study*, which is identified previously, e.g., a test suite. *Derived measures*, on the other hand, are defined as functions of two or more *base measures*. The *derived measures* are applied in the *analysis model* along with the associated decision criteria. *Indicators*

provide a qualifying statement and related analysis about the quality attributes. They provide an estimation or the evaluation of the measured attribute. Additionally, the measurements are differentiated into *subjective* and *objective* measurements. *Subjective measurements* depend on a human judgment. As a consequence, they cannot be fully automated for computation and are more expensive than the *objective measurements*. The measurement method for subjective measurements is specified by an informal definition which should be understandable for all stakeholders. On the other hand, *objective measurements* are more efficient as they can be completely automated. For *objective measurements*, besides the informal definition, a formal definition of the measurement method should be specified to enable automatic computing.

The Process Step

As part of this activity, the measurements that quantify the quality attributes are documented. We distinguish between three types of metrics: base measures, derived measures, and indicators. In general, some metrics would be used as a denominator in derived measures. In the following, the process of defining measurements is explained:

Objective: The objective of this activity is to document the measurements that would complete the quality plan. This activity describes *how the quality attributes are measured* by providing an explicit description of the measures. From a conceptual point of view, this measurement plan is used for carrying out the quality evaluation of test cases [Voi09].

Input Document: Quality model for test specification; Existing TCQPs or TCQP rules for a similar context; Existing collection of measurements

Result: TCQP *MeasurementPlan*

Participants: Quality Managers; Project Team

Process Description: *Definition of Measurement* activity involves tailoring the ISO/IEC standard 15939 for defining the measurements for the quality attributes of test cases.

Quality attributes are the atomic quality properties that can be measured directly. For each identified quality attribute, the quality manager selects relevant metrics. The metrics are either selected from the existing collection of measurements or new measurements are defined. A *base measure* can be either *objective* or *subjective* measurement. The more fine-grained the description of the measure is, the more comprehensible and reusable it is. It is important to mention that, when a *decision criteria* for *indicators* is defined, in particular for *subjective* measurements, the *decision criteria* must be checked thoroughly. Otherwise, incorrect *decision criteria* could lead to serious consequences due to wrong interpretation.

In the following, we present examples for the definition of measurement. Firstly, in Table 5.1, we show a base measure which deals with counting the number of all test cases in a test suite. This base measure can be used in other derived measures.

Base Measure	
Name (Acronym)	Number of Test Cases in a Test Suite (NTCTS)
Informal Definition	Count the number of test cases in a test suite
Type of Measurement	Objective
Scale (Type of scale)	Integer from Zero to Infinity (Absolute)

Table 5.1 A base measure counting all test cases

The example depicted in Table 5.2 shows a measure which deals with counting of test cases with the expected result specified.

Base Measure	
Name (Acronym)	Number of Test Cases with Expected Result in a Test Suite (NTCWERTS)
Informal Definition	Count the number of test cases having expected result specified
Type of Measurement	Subjective
Scale (Type of scale)	Integer from Zero to Infinity (Absolute)

Table 5.2 A base measure counting test cases with a specified expected result

Indicator	
Name (Acronym)	Ratio of Test Cases with Expected Result (RTCWER)
Informal Definition	Divide the number of test cases with a specified Expected Result (NTCWERTS) by the number of test cases (NTCTS)
Analysis Model	RTCWER <0.05 - "Acceptable" RTCWER <0.1 - "Moderate" RTCWER >0.2 - "Not Acceptable"
Type of Measurement	Objective
Scale (Type of scale)	Real from Zero to One (Ratio)

Table 5.3 An Example of defining an Indicator

The example shown in Table 5.3 shows a definition of an indicator for the measure that computes the ratio of test cases with a specified expected result. The analysis model shows the decision criteria which helps in interpreting the results and assists in the decision making.

5.1.5 Tool Support

In order to support the TCQP process, i.e., the creation as well as the visualization and interpretation of the evaluation results, we developed the *Test Case Quality Evaluator* (*TCQEval*).

The tool, shown in Figure 5.8, consists of the following modules: module for quality plans, module for quality assessments i.e., executed quality plans, module for management of measurement tools, and documentation module.

Figure 5.8 A screenshot of the initial dashboard of TCQEval

The module for quality plans includes two features, the creation of new and overview of existing quality plans. The feature regarding the creation of a quality plan supports actually the TCQP process, i.e., the four activities of the quality evaluation process, by providing a wizard-like user interface that guides the user through the creation process. The second feature provides an overview of the already created quality plans with details about the context, the defined goals and questions, as well as defined measures.

The module for quality assessments, i.e., executed quality plans, includes also two features, import of quality assessment results and overview of the already executed quality

plans, i.e., it provides an overview of the respective quality reports. It provides a dashboard that gives an insight into the quality of the test cases on three different levels of granularity: general quality of all test cases, quality of a specific test suite, and quality of a single test case. This enables the user to easily locate a test case or a group of test cases that need quality improvement, i.e., to locate those test cases with a lower score regarding a specific characteristic or sub-characteristic. This module provides multiple views regarding the quality of the assessed test cases: an overview area, which as its name suggests, gives a general overview of the test case quality regarding all characteristics. Using charts, the quality level of each quality characteristics and sub-characteristics is displayed. The graphical representation of the test case quality eases the interpretation of evaluation results. A detailed quality section of this module gives more precise information by providing the measured values of quality attributes for each respective quality characteristic and sub-characteristic. This module also supports the import of results generated by a measurement tool in a *JSON* format which should follow a format defined by the TCQP metamodel.

The module for management of measurement tools supports the import of existing measurement tools concerning specific quality attributes. Lastly, there is a module that provides documentation for the complete approach.

Seen from a technological perspective, the tool is an Angular[1] web application with a node.js[2] backend and a MongoDB[3] database. In the next section, as part of the *Context Characterization* activity (Section 5.2.1), a screenshot of *TCQEval* is shown in Figure 5.10.

5.2 The Quality Evaluation Process

In this section, we give an overview of the process which is part of the first phase of our approach, namely the *Test Case Quality Evaluation*. The purpose of this process is to guide the process of systematic quality evaluation of test cases which, at the end, should support the decision-making whether to migrate the test cases or not. In Section 4.2, we briefly introduced the core activities of the test quality evaluation process. These activities are also shown in Figure 5.9. It is important to mention that the first two steps rely on the previously introduced TCQP approach. Namely, the first activity *Context Characterization* corresponds to the TCQP's *Characterization of Context* activity, whereas the second activity, i.e., *Test Case Quality Plan Creation*, makes use of the last three activities of the TCQP approach. In the following, we refine each activity by describing its emphasis.

[1]https://angular.io/
[2]https://nodejs.org/
[3]https://www.mongodb.com/

5.2.1 Context Characterization

The purpose of the *Context Characterization* activity is to systematically discover the context of a given set of test cases. As already mentioned, this activity fully corresponds to the TCQP's activity *Characterization of Context* introduced in Section 5.1.1. An excerpt from a context model created in the TCQEval tool is presented in Figure 5.10.

5.2.2 Test Case Quality Plan Creation

The purpose of the *Test Case Quality Plan Creation* activity is the creation of a context-specific test case quality plan on the basis of the previously identified context information. This activity encompasses the last three activities of the TCQP approach. Namely, first, as part of the *Identification of Information Needs* activity, through interviews and structured brainstorming sessions with the stakeholders, the goals and questions are specified. Then, as part of the *Definition of a Common Understanding* activity, the identified goals are mapped to quality (sub-)characteristics, and for the corresponding questions, quality attributes are identified and documented. Lastly, as part of the *Definition of Measurement* activity, suitable measures are documented for the identified quality attributes are defined. As we have already introduced the three steps in Section 5.1, we do not go into detail. Figure 5.11 shows an excerpt of a quality plan. Based on the previously introduced context model shown in Figure 5.10, we have identified two goals, focusing on *Test Effectivity* and *Usability*. Each goal was refined by defining questions and for each question, a quality attribute was selected. For the *Line Coverage* quality attribute, a measurement named *Code coverage* was defined (shown in the bottom part of Figure 5.11).

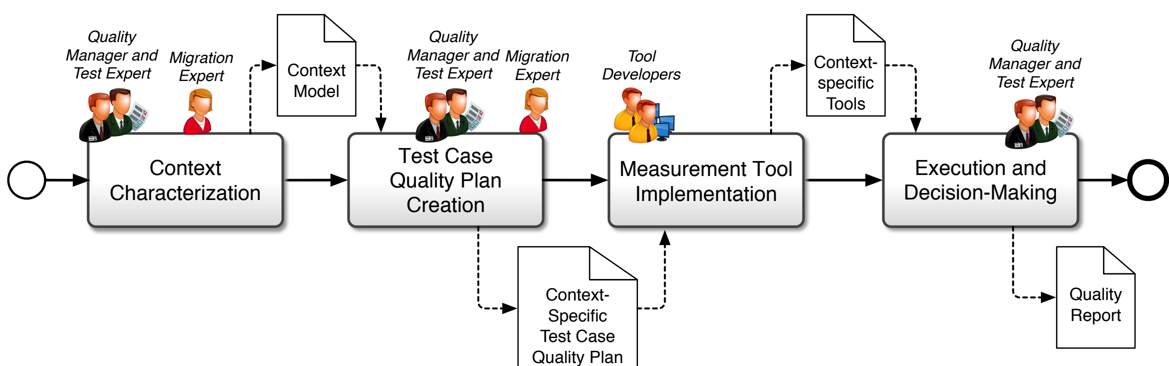


Figure 5.9 Core activities of the test case quality evaluation process

Figure 5.10 An excerpt of a context model created in the TCQEval tool

5.2.3 Measurement Tool Implementation

So far, a context-specific test case quality plan has been developed. The purpose of the *Measurement Tool Implementation* is to develop the required tools for the specified measures. We assume that associated tool developers use the specification of measures and indicators as some kind of guidance.

The developers can use the measurement model, i.e., the measurement specification to get an overview of a given measure or indicator. By doing so, they can get an understanding of what should be exactly measured, i.e., they can understand what the developed measurement tool is supposed to measure. Furthermore, the part that specifies the measurement method, i.e., the formal or the informal definition, can be used to understand how it needs to be measured. Here we envision two possible outcomes, either development of a measurement tool from scratch or reuse of existing tools that evaluate test code quality. For example, there are a lot of existing tools that already provide code coverage support, e.g., Eclipse for JUnit [JUn]. If a quality measure needs this information, then such an existing tool could

Goal Dimension	Quality Characteristic	Quality Sub-Characteristic	Question	Quality Attribute
Analyze the **PivotTestSuite** for the purpose of **Quality Assessment** with respect to **Test Effectivity** from the viewpoint of **Quality Managers and Testers**, in the previously defined context	**Test Effectivity**	**Test Coverage**	What is the code coverage (line coverage)?	Line Coverage
		Test Correctness	Do all the test cases have expected result specified?	Specified expected result
			Do all the test cases have actual result specified?	Specified actual result
Analyze the **PivotTestSuite** for the purpose of **Quality Assessment** with respect to **Usability** from the viewpoint of **Testers**, in the previously defined context	**Usability**	**Operability**	Are the test cases compilable?	Test case compilation
			Are the test cases executable?	Test case execution
		Understandability	Are the test cases properly commented?	Proper documentation

Metric Definition	
Name	Code coverage
Informal Definition	Divide the number of source code lines covered by the test cases by the total number of lines in the source code.
Type of Measurement	Objective
Measurement Method	$\frac{\text{\# of source code lines covered by the test cases}}{\text{\# of lines in the source code}} \times 100$
Scale (Type)	0 to 100 (ratio)
Interpretation	closer to 100 is better
Tool Dependency	Yes (Eclipse JUnit coverage plugin)

Figure 5.11 An excerpt of a quality plan

be reused. However, if a base measure needs a property which is not covered by an existing test quality tool, then it needs to be developed. In that case, the developers can analyze the complete measurement specification and identify derived measures that share base measures and implement the tooling in a more modular thus providing a more flexible and extensible architecture of the tool.

The result of the measurement should be as correct as possible so that the right decision regarding the migration of the test cases can be made. Therefore, during and after the implementation of the required measurement tool, its quality needs to be validated. For sure, one could think of checking a characteristic like performance, but we focus here more on the functional correctness of the measurement tool, i.e., whether the implemented measurement function is according to the specification of the measure, i.e., the measurement method with the respective formal and informal definition. For this purpose, we assume that the developers use a small amount of the original test cases to test the developed measurement tool.

The use of the measurement tool, i.e., the usage of the different measurement functions should be documented so that it can be used by non-experts as well. In order to be able to use the *TCQEval* tool for the visualization of the results, the measurement tool should provide

an export functionality. The exported results should follow a defined format by the TCQP metamodel and should be exportable in *JSON* format.

5.2.4 Execution and Decision-Making

So far, a context-specific quality plan has been developed and specified according to the TCQP metamodel. Also, a tool required for the measurement has been implemented. Here, we discuss the *Execution and Decision-Making* activity which, as its name suggests, comprises two sub-activities, namely the execution of the test case quality and the decision-making. The purpose of the former sub-activity is to perform the actual quality assessment of the test cases whereas the purpose of the latter sub-activity is the decision-making regarding the migration of the test cases based on the obtained evaluation results.

We assume that associated test developers as well as quality managers and system experts use the test case quality plan as some kind of guidance. The people involved can browse the quality plan to get an overview of the actual measurement that shall take place. When executing the different measurement functions, the people can read the provided definitions of the measures so that they can easily interpret the score at the end.

We also foresee the use of separate guidance documents in terms of a tooling manual. In case such documents are used, then they should be read instead. As most of the activities belonging to the first sub-activity, i.e., the execution of the quality plan, are performed automatically, the interaction between people and tools during the measurement needs to be addressed. More specifically, people must get notified whenever a tool has created an output, based on which they need to perform the export of the results. Having the results obtained, they should be exported and imported in *TCQEval*, the tool that we introduced in Section 5.1.5.

The TCQEval tool, as already described, provides an insight of the overall quality of the test cases according to the created test case quality plan as well as detailed insight regarding each quality characteristic, sub-characteristic or quality attribute. Based on this information, the *Quality Manager* and the *Test Expert* can decide whether to migrate the test cases or not. Potentially, a specific test suite or test cases could be selected for migration or improvement before being migrated.

5.3 Summary and Discussion

In this chapter, we have introduced the first phase of our approach, namely the *Test Case Quality Evaluation*. Firstly, we introduced a novel approach for quality evaluation of test

cases called Test Case Quality Plan (TCQP) which enables a systematic development of context-specific quality plans for test cases. Then, we introduced the process for quality evaluation which makes use of the TCQP approach. Some of the findings presented in this chapter are based on master theses [Nar17, Cha20].

In Section 5.1, we gave an overview of the TCQP approach which consists of a process and a corresponding metamodel. The process has the role of guiding in developing a test case quality plan, whereas the metamodel defines how a quality plan looks like. First, we introduced and discussed the various context factors which are relevant for the test case domain. Identification of context factor is important because they impact the outcome of the quality evaluation. Then, we presented the goal definition template and we define questions and attributes related to the test case context. Information needs provide an insight to manage objectives goals, risks, and problems. Subsequently, we presented the extended quality model for test specifications based on the ISO/IEC 25010 standard. We also discussed in detail the quality characteristics, sub-characteristics, and quality attributes and we illustrated the process of defining a common quality understanding using the quality model. Afterward, we presented the measurement model and we also illustrated the different types of measures with examples.

In the second part of this section, i.e., in Section 5.2, we introduced the main four activities of the first phase: *Context Characterization*, *Test Case Quality Plan Creation*, *Measurement Tool Implementation*, and *Execution and Decision Making*. Firstly, as part of the *Context Characterization* activity, the context of the test cases has to be characterized so that in the second step a suitable quality plan for the test cases can be created. Having the context characterized, as part of the second activity, namely the *Test Case Quality Plan Creation*, the main goals and questions of the stakeholders are identified. Based on them, appropriate quality attributes are identified as well as the corresponding quality characteristics and sub-characteristics leading to establishing a common quality understanding. Finally, for each of the identified quality characteristics, measures are defined. As part of the third activity, namely the *Measurement Tool Implementation*, tools are developed to automate the measurement process. In the last activity, i.e., *Execution and Decision Making*, the developed quality plan is being executed. The outcome of the execution in terms of a quality report containing the different scores is used to make a decision regarding the eventual migration of evaluated the test cases.

The context characterization and the creation of the quality plans are supported by the *TCQEval (Test Case Quality Evaluator)* tool. Such support is still missing for the next two activities, namely measurement tool implementation and execution and decision-making. A flexible component-based measurement tool infrastructure would definitely the

tool implementation activity. For example, by providing a flexible plug-and-play mechanism for existing quality tools as well as those that are specifically developed for a given project. Furthermore, an automated or semi-automated import of the evaluation results would largely improve the analysis process. The quality evaluation approach that we have presented focuses on assessing the quality of the test cases, but these results are not used for the eventual improvement of the test cases. As a quality report provides an insight into different quality aspects, the test cases with lower quality could be identified and improved. For example, test cases with low fault-revealing capability can be improved by applying mutation testing.

In the next chapter, we will go into detail on the main phase of our approach. i.e., the migration phase. If the quality evaluation results coming from the execution of the quality plan developed according to the TCQP approach suggest reuse of the existing test cases, a co-evolution analysis as well as a systematic development of test case migration methods are performed.

Chapter 6

Migration Phase: Method Engineering considering Co-Evolution Analysis

In the previous chapter, the pre-migration phase was introduced where the quality evaluation of the original test cases was performed. In this chapter, we introduce the main phase, i.e., the migration phase, by introducing the content of the method base as well as the method engineering process. First, we give an overview of the migration phase in Section 6.1 by giving an overview of the basic structure of the method base and the core activities of the process. In Section 6.2, we refine the structure of the method base and describe its content. Basically, the method base consists of two constituents, namely test method fragments and test method patterns. In Section 6.3, we introduce the method engineering process. Essentially, the activities of the engineering process are split into two disciplines: method development and method enactment. The findings of this chapter are summarized in Section 6.4.

6.1 Overview of the Migration Phase

In this section, we give an overview of the migration phase of the **Te**st **C**ase C**o**-**Mi**gration (**TeCoMi**) framework where method construction considering co-evolution analysis as well method enactment are performed. As shown in Figure 6.1, we support this phase by providing an extended method engineering process that relies on a method base that contains test-specific fragments and patterns. In the following, we briefly give an overview of the content of the method base and the method engineering process.

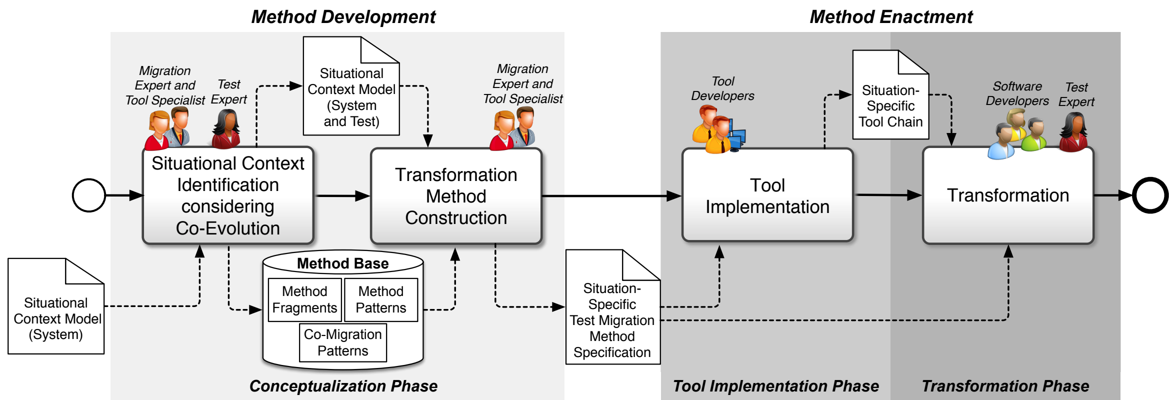


Figure 6.1 Core activities of the method engineering process of TeCoMi

Method Base

The purpose of the *Method Base* is to provide reusable building blocks for transformation method specifications. As already briefly introduced in Section 4.3.1, the method base contains two types of building blocks: *Method Fragments* and *Method Patterns*. The method fragments are atomic building blocks that can be combined to compose a transformation method.

We classify the method fragments at the highest level, based on the phase they primarily belong to, which is either the tool implementation or transformation phase. The fragments that are related to the tool implementation phase can be used to specify tool development activities, whereas the fragments related to the transformation phase can be used to describe the transformation. As shown in Figure 6.2, we further classify the fragments of both phases based on their type, namely, artifacts, activities, tools, and roles. The method fragments provided in this thesis are based upon the method fragments provided by the MEFiSTo framework. However, they have been either adopted or new test-specific fragments were created as the method fragments in MEFiSTo are defined specifically for the system migration domain.

How to combine the method fragments is encoded in the method patterns which provide construction guidelines for a method. The proposed patterns in this thesis are based upon the method patterns from the MEFiSTo framework. However, these patterns served only as an initial inspiration. The newly created test method patterns are completely reworked and adapted to the test domain because the method patterns in MEFiSTo are defined according to the system artifacts and the abstraction levels they are belonging. The test method patterns we present are functionality preserving patterns as they deal with the transformation of the test cases which preserves their functionality, by following a transformation strategy, e.g., conversion or reimplementation. As the test method patterns do not express the dependency

between the system and the test case migration, we additionally propose a set of co-migration method patterns. Technically, a co-migration method pattern is a combination of a system method pattern and a test method pattern visually resembling a double horseshoe model. A co-migration pattern encodes the relation between the applied system migration pattern and the selected test method pattern. The details about the method base are presented in Section 6.2.

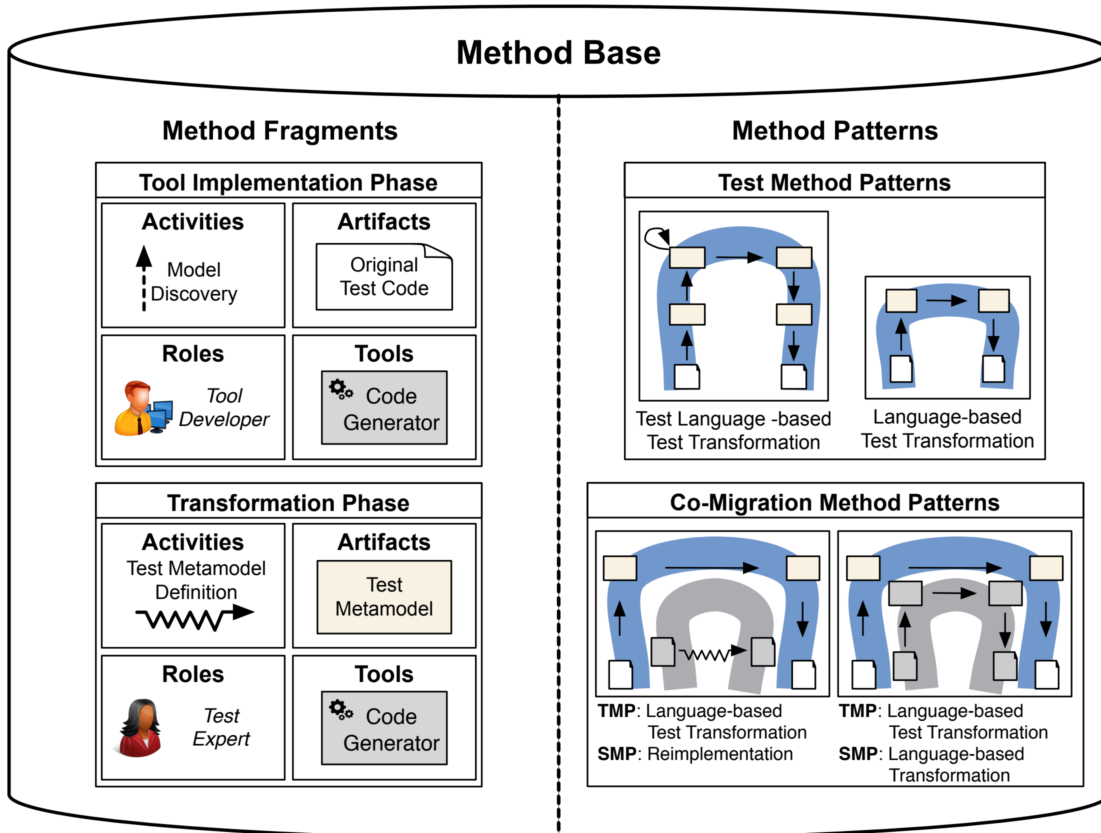


Figure 6.2 Overview of the structure of the method base in TeCoMi

Method Engineering Process

Relying on the method base, the method engineering process, shown in Figure 6.1, guides the development and the enactment of the situation-specific test migration method. In Section 4.3.2, we briefly introduced the core activities of the method engineering process. Each of the four core activities is assigned to one of the two main disciplines, namely *Method Development* and *Method Enactment*. By performing activities of the *Method Development* discipline, a situation-specific test method gets developed. It comprises the following two activities: *Situational Context Identification considering Co-Evolution* and *Transformation*

Method Construction. During the *Situational Context Identification considering Co-Evolution* activity, the situational context is systematically discovered by analyzing and characterizing it from both the system migration and as well as testing perspective. Firstly, both the source and the target tests and systems are represented as a set of concepts by applying concept modeling. This basically decomposes both the system and the tests into distinct parts so that for different parts different patterns could be applied. Then, based on this decomposed representation, the impact of the system changes on the test cases is identified and captured in terms of an impact model. Lastly, the influence factors are systematically identified. These influence factors are used by the next activity when selecting a suitable method pattern. Having the context information collected in terms of a *Situational Context Model*, the *Transformation Method Construction* activity guides the pattern-based development of a test transformation method.

Firstly, the method fragments are systematically customized as the fragments in the method base are not associated with a specific technology, but rather generically described. We aim to provide systematic guidance on the customization of the fragments. Secondly, the selected migration patterns have to be integrated as simply applying different patterns on different tests or different parts of the tests would result in an unconnected method. Therefore, to avoid this problem, we also provide guidance on the systematic integration of the patterns. The overall outcome of *Method Development* is a *Situation-Specific Test Migration Method Specification* which specifies how to do the migration by defining the activities to be performed and the artifacts that should be generated. In contrast to the first two core activities, which are concerned with the development of the transformation method specification, the last two core activities are concerned with its enactment. During the first activity of *Method Enactment*, namely the *Tool Implementation*, a *Situation-Specific Tool Chain* is developed that is required for the automation of the migration method. Namely, it enacts those parts of the method specification that describe the development of the tools which are required to automate (parts of) the transformation. We assume that a generic tool infrastructure is available when using the TeCoMi framework so that a foundation for the development of the situation-specific tools is provided. Thereafter, during the last core activity, namely the *Transformation* activity, the test migration method is enacted by enacting those parts of the test migration method specification that describe the actual transformation of the original test cases, using the tools developed.

6.2 Method Base

The *Method Base* contains the knowledge of the available migration strategies. It is actually a repository containing reusable building blocks of methods and includes *test method fragments* and *test method patterns*. *Test method fragments* are the atomic building blocks of test migration methods, whereas the *test method patterns* are representing strategies and indicate which fragments should be used. We classify the fragments based on the phase they belong to, namely the transformation phase (Section 6.2.1) or the tool implementation phase (Section 6.2.2). Besides the *test method patterns*, the repository contains a set of co-migration method patterns which express the dependency between the system and the test case migration patterns. In the following, we present the main constituents of the method base, namely the test method fragments, test method patterns, and the co-migration patterns.

6.2.1 Transformation Phase Fragments

Here, we introduce the method fragments used to specify the actual transformation also known as transformation phase fragments. A method fragment is an atomic building block of a migration method, i.e., an *artifact* or an *activity*. *Artifacts* are constituting parts of each migration method and are distinguished by the level of abstraction they are belonging to. More precisely each artifact belongs to a different level of abstraction, namely, the *System Layer*, *Platform-Specific Layer*, or *Platform-Independent Layer*. *Activities*, on the other hand, produce or consume appropriate artifacts. As we follow the idea of model-driven software migration [FWE+12], our method fragments belong to one of the following reengineering processes: *Reverse Engineering*, *Restructuring*, and *Forward Engineering* [CC90].

Therefore, the activities as well as the artifacts are represented in Figure 6.3 to Figure 6.5 as horseshoe models [KWC98]. More particularly, we present a series of double horseshoe models, i.e., a combination of a test case horseshoe and a system horseshoe model, according to the above-mentioned abstraction layers.

Activities and Artifacts

Firstly, we introduce the artifacts and the activities and we start with the lowest level of abstraction, namely the *System Layer*. Seen from a testing perspective, on the *System Layer*, textual artifacts representing test code and models of the test code are placed (Figure 6.3). Regarding the textual artifacts, this is either the *Original Test Code* or the *Migrated Test Code*. The test code can be either the test cases, implemented in a specific testing framework,

e.g., JUnit[1] or MSUnit[2], test configuration scripts or a manually implemented additional test framework. Similarly, regarding the models of the code, it is either the *Model of Original Test Code* or the *Model of Migrated Test Code*. These models represent the test code in a form of Abstract Syntax Tree [OMG11a] of the appropriate language of the original or the target environment.

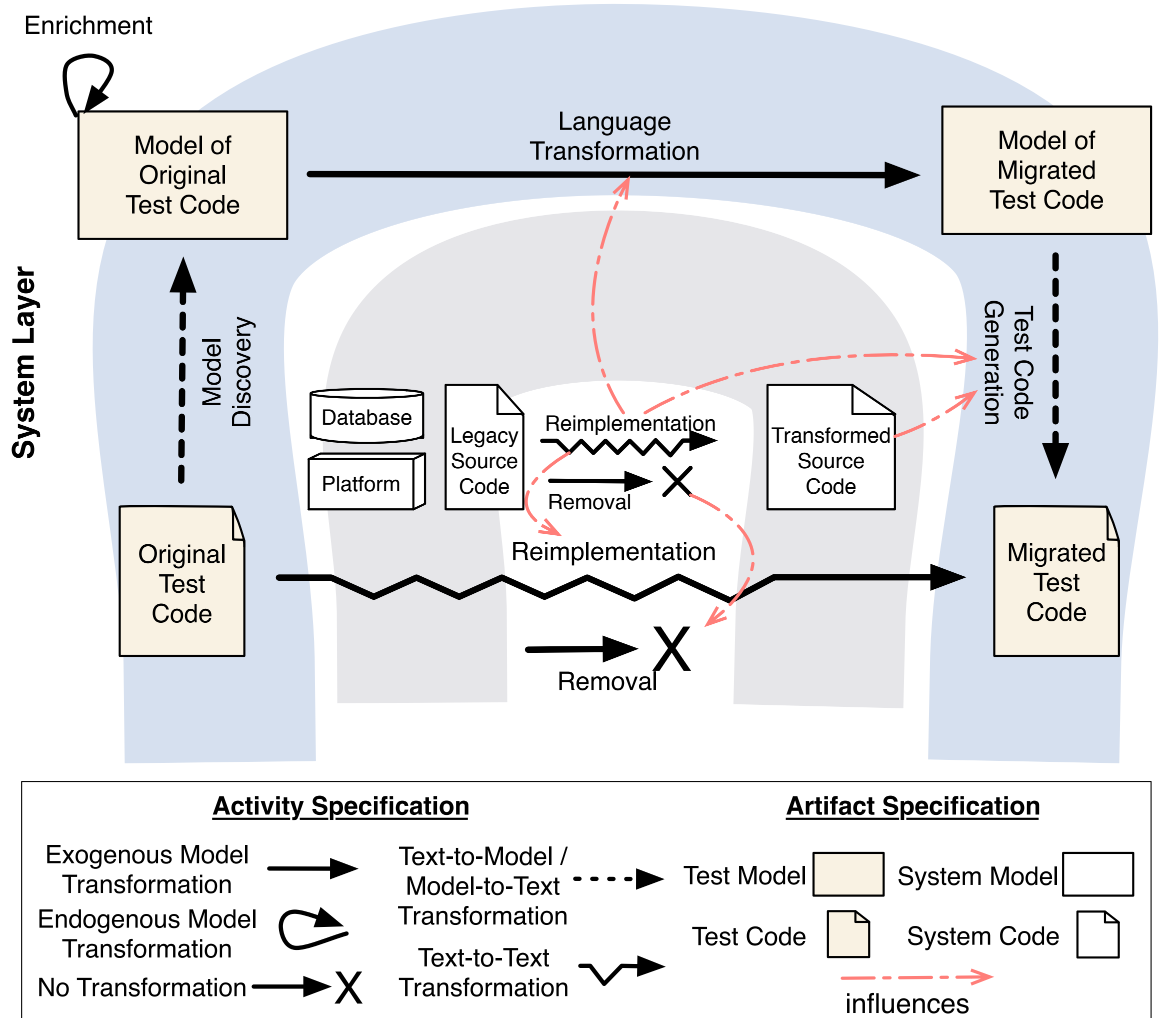


Figure 6.3 Test method fragments of the transformation phase on the system layer

Regarding the activities, as can be seen in Figure 6.3, we distinguish between five types of activities: *Endogenous Model Transformation*, *Exogenous Model Transformation*, *Model-to-Text Transformation*, *Text-to-Model Transformation*, and *No Transformation* (removal/deletion of particular parts). These activities can be further distinguished by the

[1]http://junit.org/junit4/
[2]https://msdn.microsoft.com/en-us/library/ms243147.aspx

reengineering process they belong to, namely *Reverse Engineering*, *Restructuring* or *Forward Engineering* [BCJM10b]. On the system layer, we have the first reverse engineering activity namely the *Model Discovery*. The *Model Discovery* activity relies on syntactical analysis and by using a parser it allows automatic text-to-model transformation to create a model of the test case source code represented as an Abstract Syntax Tree (AST) [OMG11a]. The obtained model, the *Model of Original Test Code*, is known as the initial model because it has a direct correspondence to the test cases. Optionally, the *Enrichment* activity can be applied to the *Model of Original Test Code*. By applying this activity, e.g., by using annotation, additional information could be inserted into the tests. The *Language Transformation* also known as AST-based transformation [KWC98], defines a direct mapping between the *Model of Original Test Code* and the *Model of Migrated Test Code*. The *Model of Migrated Test Code* is already specific for a particular testing framework and can be used for the generation of the *Test Code* by executing the *Test Code Generation* activity, which is a model-to-text transformation. As test cases are generated from the test model, this can be seen as e typical model-based testing activity [JGG16]. The tools which support forward engineering are known as generators and for each specific target platform, an appropriate generator is needed. Custom code generators can be built by using Xtend[3] which is a statically typed programming language which offers features like template expressions and intelligent white-space management. As part of this double horseshoe model, we also introduce *Reimplementation* which is a text-to-text transformation. This activity is performed by developers or testers manually by observing the original test cases and then implementing them for the target testing environment. Finally, the *Test Code Removal* activity is used to specify on which part of the test case code a transformation should not be performed. Intuitively, this activity compared to all other is the only one that does not produce output.

The system method fragments, provided by the MEFiSTo framework, which belong to the system layer, are only code artifacts and reimplementation and removal activities. Nevertheless, any of these activities can potentially have an influence on the activities defined by the test horseshoe model, namely the language transformation, reimplementation or removal. For example, a reimplementation guideline can be used when performing manual transformation of the test cases, i.e., when performing the *Reimplementation* activity on the test cases. It can also influence the definition of the model transformation rules if *Language Transformation* of the test cases is performed. Or, if a part of the system is removed by applying the *Removal* activity, then, consequently, this has to be done for those test cases that test that part of the system. Also, the *Test Code Generation* activity is influenced by the

[3]http://www.eclipse.org/xtend/

Reimplementation activity of the system or from the *Transformed Source Code* as they are directly defining the system in the target environment.

The *Platform-Specific Layer* is a higher level of abstraction compared to the system layer. As shown in Figure 6.4, technology-specific test concepts are used to represent the test cases for both the source and the target environment. The *Model of Original Executable Tests* and *Model of Migrated Executable Tests* are considered as platform-specific as they represent executable test cases by using testing concepts which are specific for a particular testing framework, e.g., JUnit or MSUnit.

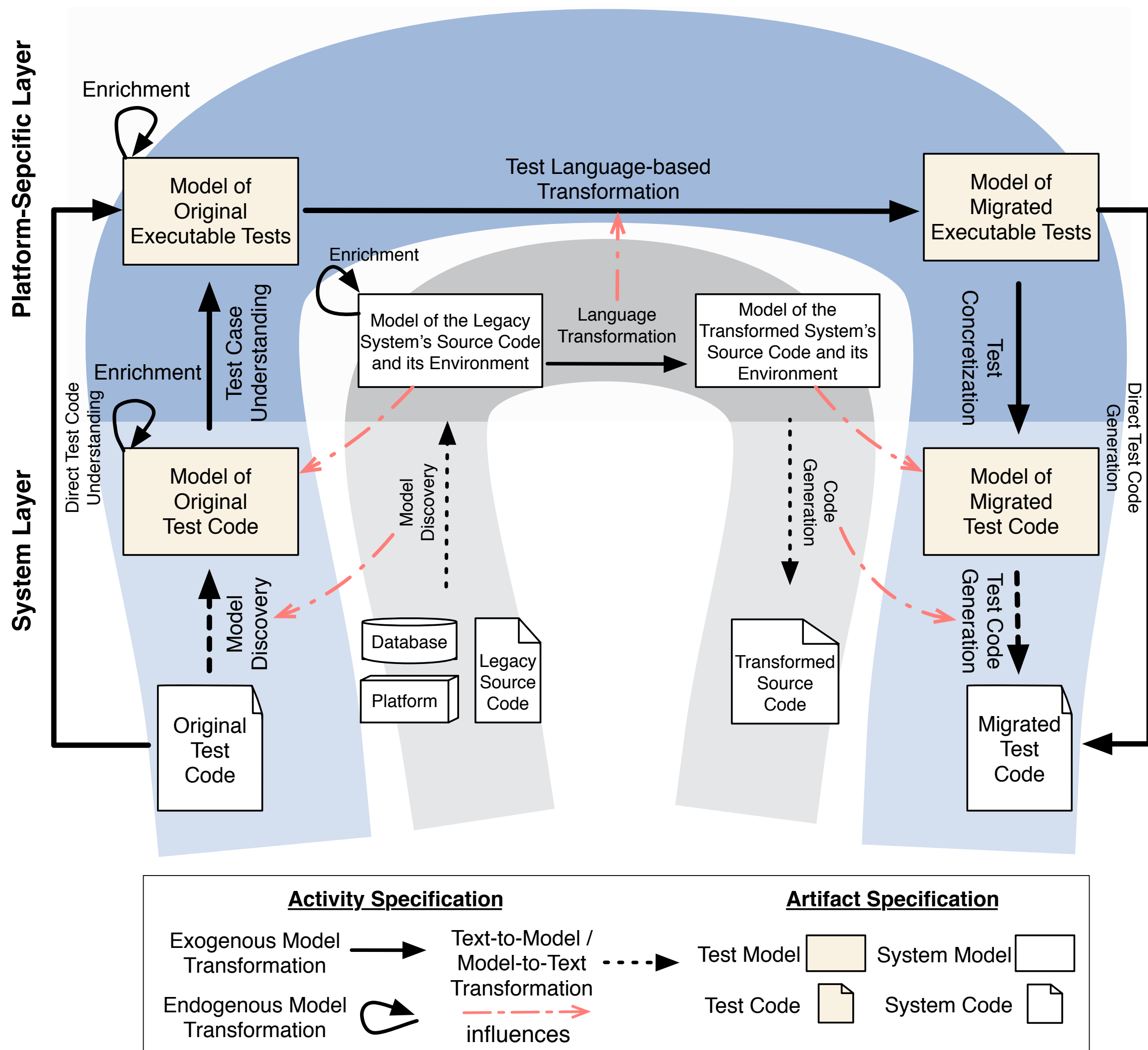


Figure 6.4 Test method fragments of the transformation phase on the platform-specific layer

In order to obtain the *Model of Original Executable Tests*, firstly the initial model, i.e., the *Model of Original Test Code* is explored by navigating through its structure in an activity called *Test Case Understanding*. Model elements which represent test relevant concepts

like test suite, test case or assertion are identified and then, by applying a model-to-model transformation, the *Model of Original Executable Tests* is obtained. This model is platform-specific and it is an instance of a metamodel of a particular testing framework, e.g., JUnit, MSUnit or TTCN-3. The *Model of Original Executable Tests* can be directly obtained from the *Original Test Code* by performing *Direct Test Case Understanding*. The *Test Language-based Transformation* activity defines a mapping directly between two testing frameworks, i.e., it defines a mapping between the testing concepts inside the original and the target testing framework. By applying this transformation on the *Model of Original Executable Tests*, the *Model of Migrated Executable Tests* is obtained. Having the *Model of Migrated Executable Tests*, by applying the forward engineering activity called *Test Concretization* a *Model of Migrated Test Code* is obtained. The *Model of Migrated Test Code* can be directly obtained from the *Model of Migrated Executable Tests* by performing *Direct Test Code Generation*.

Seen from a system perspective, according to MEFiSTo, we have the *Model Discovery*, *Language Transformation*, and *Code Generation* activities as well the *Model of the Legacy System's Source Code and its Environment* and *Model of the Transformed System's Source Code and its Environment*. In the following, we discuss the dependencies between the activities and the artifacts from the double horseshoe model shown in Figure 6.4. Firstly, the *Model Discovery* activity for both the test and the system horseshoe model is the same. This suggests that it could be completely reused, if available, to obtain the *Model of Original Test Code*. Analogously, the same applies to the *Code Generation* and *Test Code Generation* activities, with eventual minor adjustments of the code generation templates. The same applies to the metamodels on this level the reverse engineered *Model of Original Test Code* has to be conform to. This comes from the fact that both *Model of Original Test Code* and *Model of the Legacy System's Source Code and its Environment* are actually abstract syntax trees of the same language. The *Test Language-based Transformation* is directly influenced by the *Language Transformation* applied to the system. More details on the dependencies are presented in Section 6.2.4.

On the *Platform-Independent Layer*, the models representing the test cases are independent of any particular testing framework or testing technology. On this level of abstraction, as shown in Figure 6.4, the *Model of Abstract Tests* is placed. The *Model of Abstract Tests* is considered to be platform-independent as it is independent of any concrete testing framework. Standardized languages like UML Testing Profile (UTP) [OMG13b] and Test Description Language (TDL) [ETS16] are used for modeling the abstract tests.

By applying the *Test Abstraction* activity, which is a model-to-model transformation as well, one can obtain a model of the abstract test cases which is platform-independent. Regarding the technical side of the model-to-model transformations, different transformation

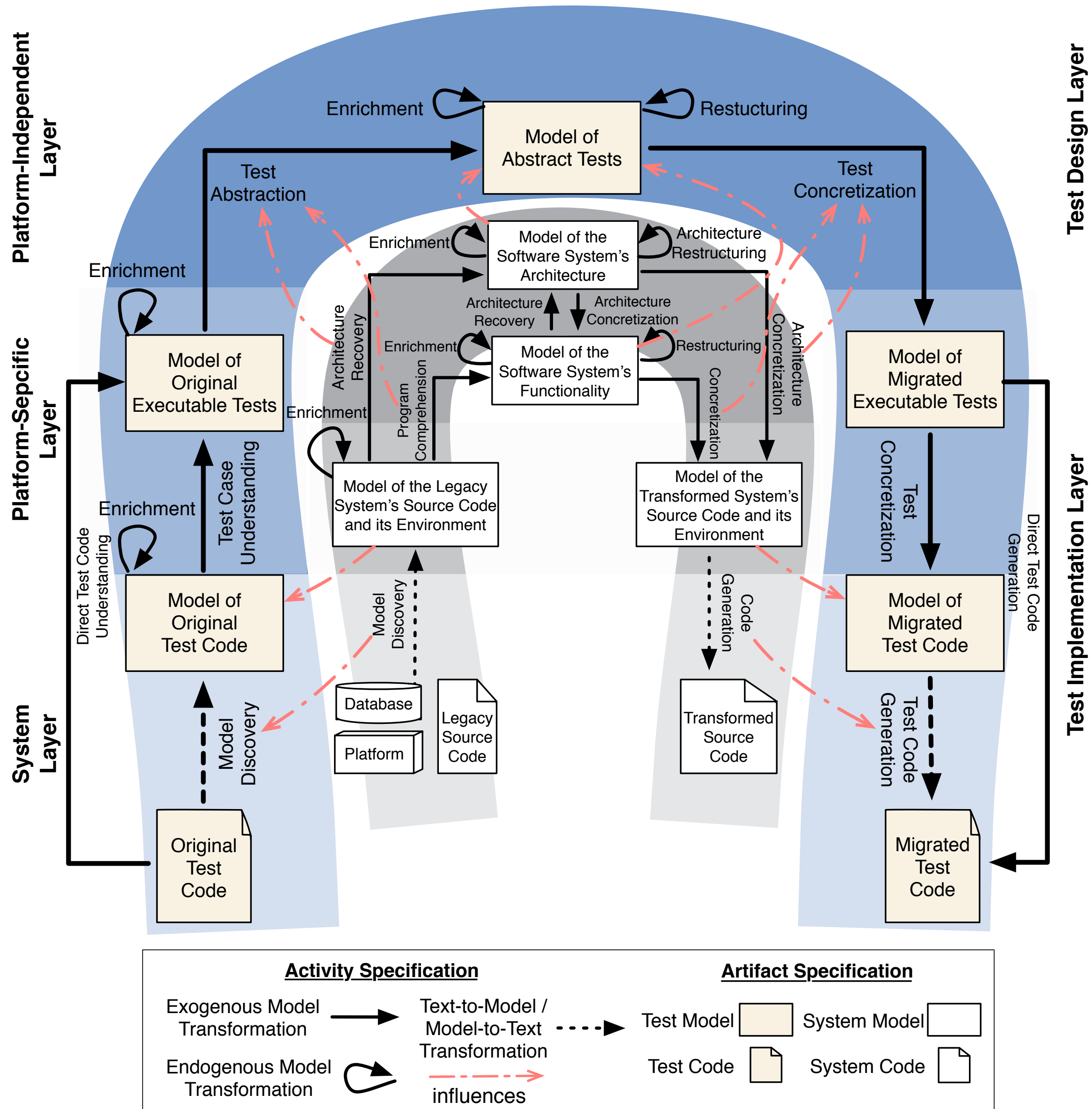


Figure 6.5 Test method fragments of the transformation phase on the platform-independent layer

languages can be used, e.g., QVT [OMG11b], JDT[4] or ATL[JABK08]. At this point, a *Restructuring* activity has been foreseen on the *Model of Abstract Tests*. According to [CC90], *Restructuring* is "the transformation from one representation form to another at the same relative abstraction level, while preserving the subject system's external behavior (functionality and semantics)". In the testing domain, we define test restructuring as the transformation from one test representation to another at the same relative abstraction level, while preserving

[4] https://www.eclipse.org/jdt/

the "semantics" of the tests. Here, with "semantics" we mean the functionality that is being checked by a particular test. The *Restructuring* activity is of course influenced by the target testing environment, testing tool, or by requirements on improving the quality of the test cases (e.g., maintainability). However, it could also be influenced by the changes that happen in system migration. As these changes may be relevant for the test models, they have to be reflected in the tests as well. The *Enrichment* activity can be applied to various models, e.g., *Model of Executable Tests*, *Model of Abstract Tests*. By using annotations, one can insert additional information to the tests.

As defined by [CC90], *Forward Engineering* is "the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system". In the field of software testing, this can be paraphrased as a process of moving of high-level test abstractions and logical implementation-independent design to the physical implementation of the test cases. The complete *Forward Engineering* side can be seen as Model-Based Testing, or more specifically Model-Driven Testing [HL03, EGL06]. The test models are used as input for a chain of model-to-model transformations, ending with a model-to-text transformation, which provides the test code as output. By applying *Test Concretization* a *Model of Migrated Executable Tests* is obtained.

Seen from a system perspective, according to MEFiSTo, we have to consider two different paths in the system horseshoe model. Firstly, we consider the path starting with *Program Comprehension*, *Restructuring*, *Enrichment*, and *Concretization*. The artifact obtained after applying *Program Comprehension* is the *Model of the Software System's Functionality*. The *Restructuring* and eventually *Enrichment* which are performed on this model can influence the *Restructuring* and the *Enrichment* of the *Model of Abstract Tests* as this is the point where the actual changes in the system can happen. The same applies when *Architectural Recovery* is performed, i.e., when *Model of the Software System's Architecture* is obtained and *Architecture Recovery* or *Enrichment* are performed. Section 6.2.4 provides further insight into these dependencies.

The three abstraction layers analyzed so far, can be mapped to two testing abstraction layers defined by the Test Description Language standard [ETS16]. The *Platform-Independent Layer* corresponds to the *Test Design Layer*, where the initial test model is designed, namely the *Model of Abstract Tests*. Then, on the *Test Implementation Layer*, *Model of Original Executable Tests* and *Model of Migrated Executable Tests* are placed. These two models represent a test implementation with concrete test data. Finally, at the lowest level of abstraction, we have *Model of Original Test Code* and *Model of Migrated Test Code* as well as *Original Test Code* and *Migrated Test Code*.

Tools

In order to support the previously introduced activities, thus enabling a (semi-) automatic transformation, we foresee in total three types of tools that are needed (Figure 6.6). Firstly, a *Parser* is needed to obtain the initial models out of the textual artifacts, i.e., to perform the *Model Discovery* activity. Then, in a series of model-to-model transformations, which are specified by transformation rules, initial models are obtained. The specified rules are then executed by a *Model Transformation Engine*. Finally, a *Code Generator* is needed to perform the model-to-text transformation by executing the test code generation rules previously specified, thus generating the test code for the migrated test cases.

Roles

Regarding the transformation phase, two roles are associated, as already discussed in Section 4.3, namely, *Test Expert* and *Software Developer*. Consequently, we provide corresponding method fragments for both roles, shown in Figure 6.6.

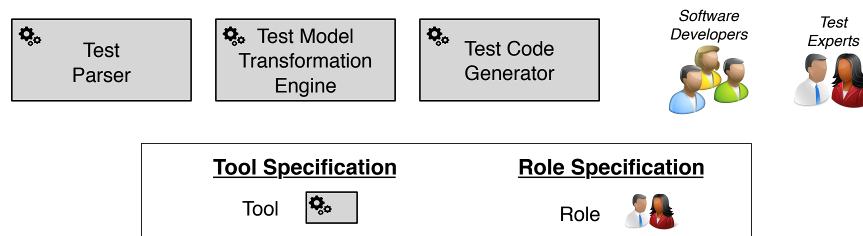


Figure 6.6 Roles and Tools of the transformation phase

A *Test Expert*, on the other hand, can be involved in tasks which require knowledge of certain aspects. For example, a test expert could be a tester of the original system. During the transformation of the test cases, her knowledge can be used to perform some enrichment activities on the test cases. Another example for a system expert is a test engineer who could be involved in an architectural restructuring of the test cases. For example, when performing such a restructuring, decisions between different alternatives should be made regarding the structure of the test suites and test cases in the target testing environment. A person in the role of *Software Developer* is responsible for performing reimplementation activities.

6.2.2 Tool Implementation Phase Fragments

In this section, method fragments for the tool implementation phase are introduced. These fragments are used to specify the development of the required tools and can be derived from the previously introduced method fragments that specify the actual transformation. For

example, the method fragments of the transformation phase are used to specify that a *Test Language-based Transformation* activity shall be performed. This kind of transformation means that an original platform-specific test model is transformed into another target test model. In order to perform the transformation activity, an artifact should be available, namely the *Model Transformation Rules*. Hence, developing the model transformation rules is a necessary preparation. Considering this relation between the method fragments of both phases, the method fragments of the tool implementation phase were derived from the ones of the transformation phase. Figure 6.7 depicts the resulting method fragments in terms of activities, artifacts, and tools.

Artifacts & Activities

We have identified six method fragments including both activities and their corresponding artifacts, similar to the MEFiSTo framework, which served as a starting point. Automated transformations of test cases are based on the use of test models, thus implying corresponding test metamodels. Therefore, we introduce the *Test Metamodel Definition* activity which represents the definition of a required test metamodel. The corresponding artifact is specified by the *Test Metamodel* fragment. Regarding the actual definition metamodel, we foresee three different ways to realize it. For sure, as a first option, a metamodel can be developed from scratch. But, in the case of transformation methods, metamodels either for programming languages or test languages used in the source or target environment are required. In general, these metamodels are well-defined and stable. This gives the second option, namely the reuse of those metamodels which are available. Sometime, however, the available metamodels should be adapted to specific needs. Hence, a third option would be to use a profiling mechanism to adapt an existing metamodel.

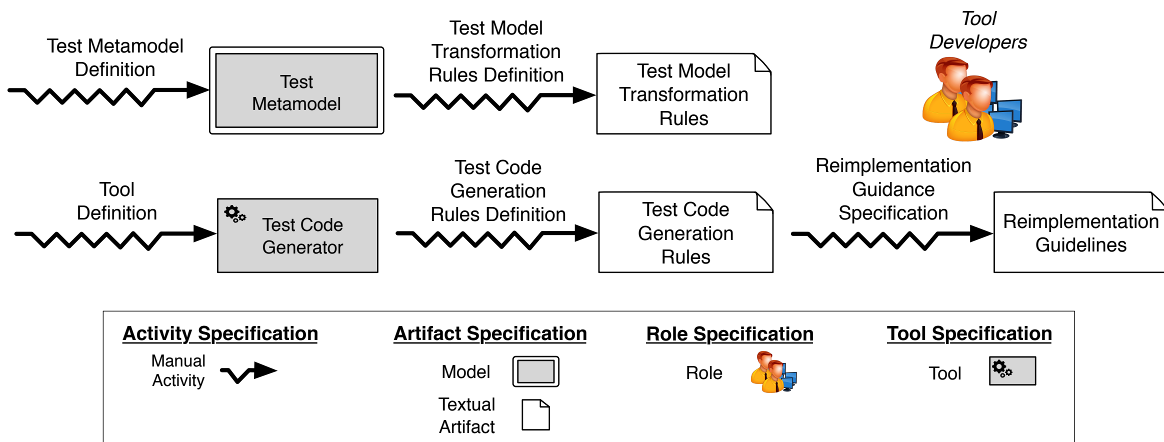


Figure 6.7 Test method fragments of the tool implementation phase

Some activities may require the use of tools. For example, performing *Model Discovery* requires using parsers or *Test Understanding* requires reverse engineering tools. The *Tool Definition* activity represents the development of such tools. Similarly to metamodels, one can either develop a required tool from scratch or reuse an existing one. As can be seen in Figure 6.7, the output of this activity is a tool and not an artifact, compared to the other activities.

Those activities that specify a transformation between test models rely on model transformations which are executed automatically. The definition of the required transformation rules is represented by the *Test Model Transformation Rules Definition* activity. The resulting artifact of this activity is represented by the *Test Model Transformation Rules* fragment. Similarly, the activity *Test Code Generation Rules Definition* is used to represent the definition of the generation rules. The resulting artifact in this case is represented in terms of the *Test Code Generation Rules* fragment. The *Reimplementation* activity is performed manually by software developers and therefore it needs to be guided. The definition of such guidance is represented by the activity *Reimplementation Guidance Specification*. The corresponding resulting artifact is specified in terms of the *Reimplementation Guidance* fragment. The *Reimplementation Guidance* is used by the developers when the actual manual transformation of the test cases takes place.

Roles

There is only one role associated to the implementation phase and that is a *Tool Developer*. Accordingly, we provide a corresponding method fragment. Intuitively, the *Tool Developer* is responsible for the previously introduced activities, e.g., reimplementation guidance, the definition of test metamodels or definition of any tool which is required. Consequently, we expect that a person in this role has knowledge of model-driven engineering and developing reengineering tools for test cases.

Tools

Regarding tools, two types of tools are distinguished: kind of general tools which are required by any transformation method and method-specific tools, i.e., tools which are specific to a certain method. The general tools, on the one hand, are required by any method that contains automated parts of the transformation. A good example of such a tool is a model transformation engine which is required for the execution of test model transformation rules. We have already introduced the method fragments which specify these kinds of tools as part of the transformation phase fragments, like *Parser* and *Test Code Generator*. Method-specific

tools on the other hand are those tools which are specifically developed for a certain method. For example, reverse engineering tools or clustering tools are developed for each method. For the specification of this kind of tool, we provide a generic fragment.

6.2.3 Test Method Patterns

Having only method fragments like artifacts, activities or tools is not sufficient as no guidance is provided on how to assemble them and thus create a test case migration method. For this purpose, we additionally provide method patterns.

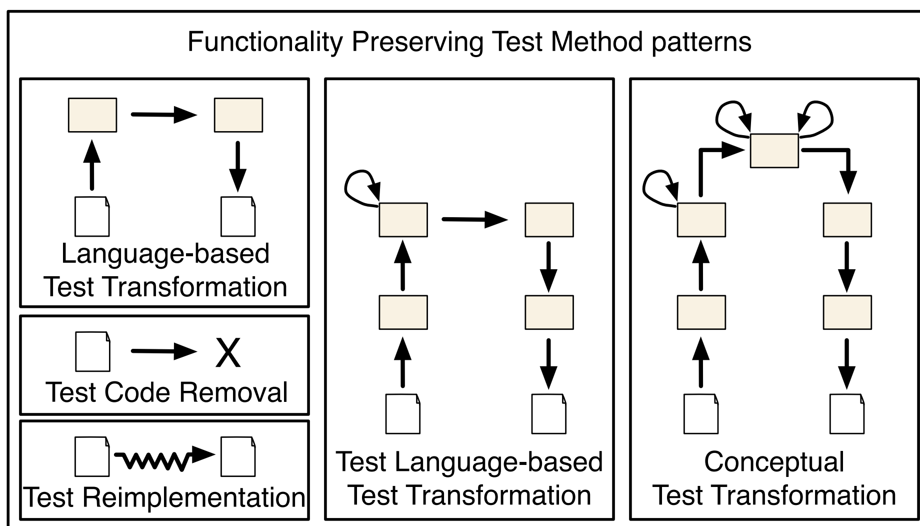


Figure 6.8 Basic test transformation method patterns.

A method pattern, intuitively, represents construction guidelines for migration methods and follows a certain strategy. It contains the methodological knowledge with the purpose to address the problem associated with it. Technically seen, a method pattern defines which method fragments should be customized and how to put them together. Functionality preservation is the main requirement in any migration project, and therefore, in test case migration too [BLWG99]. To preserve this functionality, a consistent path in the horseshoe model has to be realized from the *Original Test Code* to the *Migrated Test Code*. In the following, as shown in Figure 6.8, we present the basic test method patterns that preserve functionality. We describe each pattern according to the schema depicted in Figure 6.9, which is also used in [Gri16]. The tabular description of the schema summarizes the most important characteristics of each test method pattern. Additionally, we provide a description to better understand these characteristics in detail.

Intent	What problem does the pattern address?
Strategy	Which methodological solution does the pattern provide?
Structure	The structure of the pattern, depicted as a path in the horseshoe model
Applicability	In which situations is the pattern suitable? What are the most important influence factors on its efficiency or effectiveness?
Preparation	Which artifacts or tools have to be developed in advance of the transformation when applying the pattern?

Figure 6.9 Schema to characterize test method patterns

Language-based Test Transformation

The *Language-based Test Transformation* (Figure 6.10) pattern defines the migration of the functionality of the test cases by defining a mapping between the language constructs in both the original and target environment. The mapping is applied by a direct transformation between the *Model of Original Test Code* and the *Model of Migrated Test Code*. Theoretically seen, this pattern could be applied actually in any migration scenario, but its suitability mainly depends on the complexity of the model transformations between both models.

Firstly, the extracted *Model of Original Test Code* needs to be interpreted to identify the test concepts to be transformed. Then, it could be necessary to restructure the explicit test concepts, like test behavior or test assertions. Once prepared, the test concepts on the original environment have to be mapped to the test concepts of the target environment. Thus, this pattern could be considered suitable if the effort of test interpretation and restructuring is relatively low. However, seen from a test perspective, it basically means, the transformation of the test concepts has to be done implicitly.

Test Language-based Test Transformation

The *Test Language-based Test Transformation* (Figure 6.11) patterns define to migrate the functionality of the test cases by using an intermediate test representation on platform-specific layer.

The testing concepts together with the test data are explicitly represented by a *Model of Original Executable Tests*. By applying the separation of concerns principle, this pattern makes the transformation step less complex compared to the *Language-based Test Transformation*. Namely, the first concern of interpretation is explicitly addressed by the *Test Case Understanding* activity. Then, the *Restructuring* could be applied to the *Model of Original*

Intent	Perform an automated transformation of the original test cases into a new environment, following a conversion-based transformation strategy
Strategy	Definition of a direct mapping between the programming languages of the source and target environment. This is realized by representing the original test cases as a model of the AST of the programming language on a platform-specific layer. A model transformation that transforms this model into an AST of the target environment is a realization of the mapping between the programming languages involved.
Structure	
Applicability	Use when the test logic that should be transformed is realized comparably in the source and target test environment and a high number of test cases is considered. The difference in the realization determines the complexity of the mapping between the languages involved, influencing the efficiency and effectiveness of the pattern
Preparation	Applying this pattern essentially requires realizing a test code parser, model-to-model transformations and test code generation rules.

Figure 6.10 Characterization of the Language-based Test Transformation Pattern

Executable Tests, which enables direct manipulation of the test concepts, e.g. test assertions. After *Restructuring*, the mapping to the target test framework, i.e., mapping to the *Model of Migrated Executable Tests* is performed. Finally, *Direct Test Code Generation* is applied either directly or via *Test Concretization* and *Model of Migrated Test Code*, the *Migrated Test Code* is generated. This pattern could be considered suitable when the difference of the test case implementation is significantly different in the original and the target testing frameworks. Compared to the *Language-based Test Transformation* pattern, this pattern enables direct, i.e., explicit representation and manipulation of test constructs.

Conceptual Test Transformation

The *Conceptual Test Transformation* (Figure 6.12) pattern defines to migrate the test functionality by using an intermediate representation in terms of a *Model of Abstract Tests* on a platform-independent layer. This improves the dependent framework transformation on the platform-specific layer by explicitly representing some test concepts on a higher level of abstraction as part of the *Model of Abstract Tests*.

For example, seen from the test design perspective, the test architecture or test behavior could be explicitly represented with this model. This pattern could be considered suitable

Intent	Perform an automated transformation of the original test cases into a new environment, following a conversion-based transformation strategy
Strategy	Definition of a direct mapping between the test languages of the test environments involved. This is realized by representing the original executable test cases as a model on a platform-specific layer. A model transformation that transforms this test model of executable tests into a model of executable tests of the target environment is a realization of the mapping between the test languages involved.
Structure	
Applicability	Use when the test logic to transform is realized comparably in the original and target test environment and a high number of test cases is considered. The complexity of the mapping between the test languages involved is determined by the difference in the realization, which further influences the efficiency and effectiveness of the pattern
Preparation	Applying this pattern essentially requires realizing a test code parser, model-to-model transformations and test code generation rules.

Figure 6.11 Characterization of the Test Language-based Test Transformation Pattern

when some test concepts are realized completely different in both environments or when a restructuring of the test architecture or test data is necessary.

Test Reimplementation

The *Reimplementation* (Figure 6.13) pattern defines to migrate the test functionality by having it manually transformed by software developers. Firstly, they explore the existing test cases and try to identify the functionality being tested. Then, they try to implement the same test cases, testing the same functionality in the target environment. This pattern could be suitable in cases when an automatic migration is difficult to be implemented, i.e., when the number of the test cases is not that high or when they are of no such high complexity.

Test Code Removal

The *Test Code Removal* (Figure 6.14) pattern defines not to migrate certain parts of the test code, i.e., no transformation should be performed on it. Due to the evolutionary development of the system as well as of the test cases, inconsistencies between the test code and the system code may exist, i.e., it may happen that no longer supported features or non-existing parts

Intent	Perform an automated transformation of the original test cases into a new environment, following a conversion-based transformation strategy
Strategy	Use a model of abstract cases to transform on a platform-independent layer. The representation is reverse engineered from model of original executable test on a platform-specific layer. After a potential restructuring, it is transformed into a model of executable test of the target environment, before test code gets generated
Structure	
Applicability	Use when the test cases to transform is realized significantly different in the source and target test environment and a high number of test cases is considered. The use of a model of abstract tests can reduce the complexity of the transformation by separating the concerns of reverse engineering, restructuring, and mapping the functionality. The efficiency and effectiveness of this pattern is essentially influenced by the complexity of these concerns
Preparation	Applying this pattern essentially requires realizing a test code parser, model-to-model transformations and test code generation rules.

Figure 6.12 Characterization of the Conceptual Test Transformation Pattern

Intent	Perform a manual transformation of the original test cases into a new test environment, following a reimplementation-based transformation strategy
Strategy	Provide guidance for software developers / testers who manually reimplement the test cases in the target test environment
Structure	
Applicability	Use when automatic approaches are either inefficient or ineffective. The amount of available developers / testers and their experience has an essential influence on the efficiency and effectiveness of the pattern
Preparation	Applying this pattern essentially requires defining guidance documents to systematize the reimplementation.

Figure 6.13 Characterization of the Test Reimplementation Pattern

of the system are still being tested. Furthermore, it could be that some parts of the original system are now being implicitly supported in the new environment, e.g., by a library or a

framework, so it no longer needs to be tested. Consequently, on those parts of the test code with obsolete test cases, a transformation is not performed.

Intent	Perform no transformation of some test cases
Strategy	Ignore corresponding test code
Structure	□ → X
Applicability	Use when the functionality under test is not present any longer in the target environment (dead code) or if it is implicitly provided by the target system environment
Preparation	/

Figure 6.14 Characterization of the Test Code Removal Pattern

6.2.4 Co-Migration Method Patterns

As the test method patterns do not express the relation between the system and the test case migration, we propose a set of co-migration method patterns. We have already introduced in Section 6.2.1 the test method fragments in the form of test horseshoe model paired with the system horseshoe model at three different levels of abstraction. This is actually the starting point for defining the co-migration patterns. Technically, a co-migration method pattern is a combination of a system method pattern and a test method pattern, visually resembling a double horseshoe model. More precisely, we define a co-migration method pattern as follows:

Notation 16 (Co-migration method pattern)**.** *A co-migration method pattern is a method pattern which relates a test method pattern and a system method pattern by explicitly establishing the relation between the corresponding method fragments.*

By explicitly establishing the relation between test and system method patterns, we aim to ease the process of the selection and configuration of a test method pattern. An already configured system method pattern, with selected and concertized method fragments, i.e., artifacts and activities, suggests in what way the test method fragments should be selected and configured. Consequently, it suggests in what way the tools supporting the different method fragments should be developed.

The co-migration patterns also facilitate the reuse of existing artifacts and activities from the system migration method. As an explicit relation between the system and test method

patterns exists, it facilitates the reuse of the already existing artifacts and activities defined for the system transformation method. Furthermore, the developed and used tooling for the system migration, e.g., a language parser or a language meta-model, which corresponds to an activity or an artifact, could be reused.

Our test method patterns were mainly inspired by the method patterns presented in [GFEK16], where four different functionality preserving method patterns were defined, namely: *Reimplementation*, *Language-based Transformation*, *Conceptual Transformation*, and *Code Removal*. As we already said at the beginning of this section, a co-migration comprises a test method pattern and a system method pattern. We created the co-migration patterns by combining each of the test method patterns with each of the system method patterns, excluding the *Code Removal* pattern. The *Code Removal* was not taken into consideration as it does not influence the reuse of method fragments. In the following, we analyze each co-migration method pattern regarding two aspects, namely reusable method fragments and impacted method fragments. Reusable method fragments are those method fragments from the system transformation method which could be directly reused in the test transformation method. Impacted method fragments are those test method fragments which are impacted by the system method fragments.

Co-Migration Patterns containing Test Reimplementation

Figure 6.15 depicts the co-migration patterns that combine the *Test Reimplementation* method pattern (*CMP1* to *CPM3*) with the three possible system migration patterns *Reimplementation*, *Language-based Transformation*, and *Conceptual Transformation*.

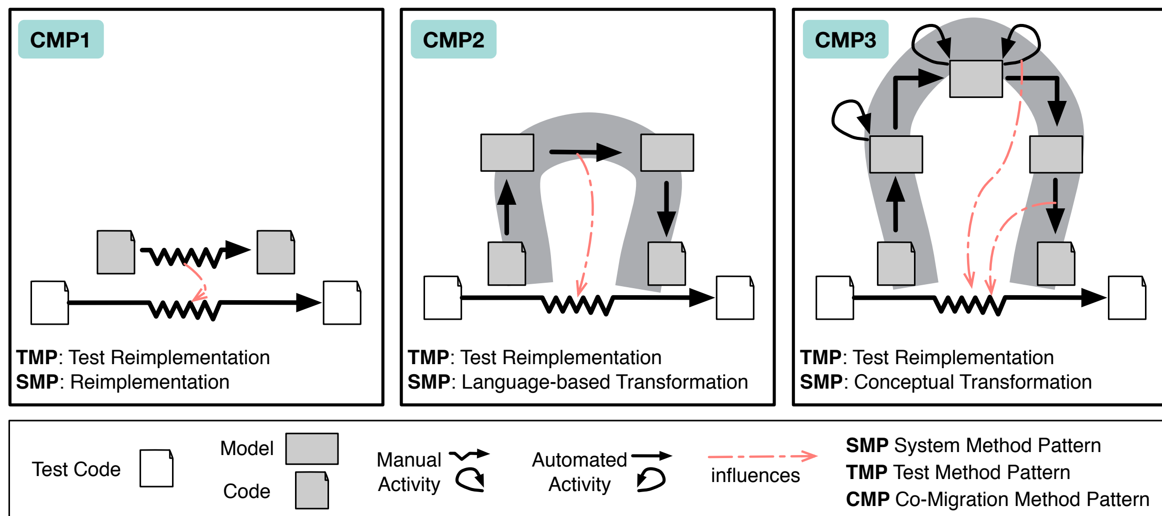


Figure 6.15 Co-Migration Patterns containing Test Reimplementation

The pattern *CMP1* is a combination of two reimplementation method patterns and it is a very simple pattern which suggests a manual migration of the test cases. The ease of reimplementation of the test cases depends on the documentation of the system transformation method, the more structured the better. In the case of *CMP2* and *CMP3* patterns, *Test Reimplementation* is combined with *Language-based Transformation* and *Conceptual Transformation*, respectively. In this constellation, the reimplementation of the test cases should be easier as the transformation of the system is specified explicitly in terms of transformation rules. However, no system method fragments could be directly reused as part of the reimplementation of the test cases.

Co-Migration Patterns containing Language-based Test Transformation

Figure 6.16 depicts the co-migration patterns that combine the *Language-based Test Transformation* method pattern with the three possible system migration patterns *Reimplementation* (*CMP4*), *Language-based Transformation* (*CMP5*), and *Conceptual Transformation* (*CMP6*).

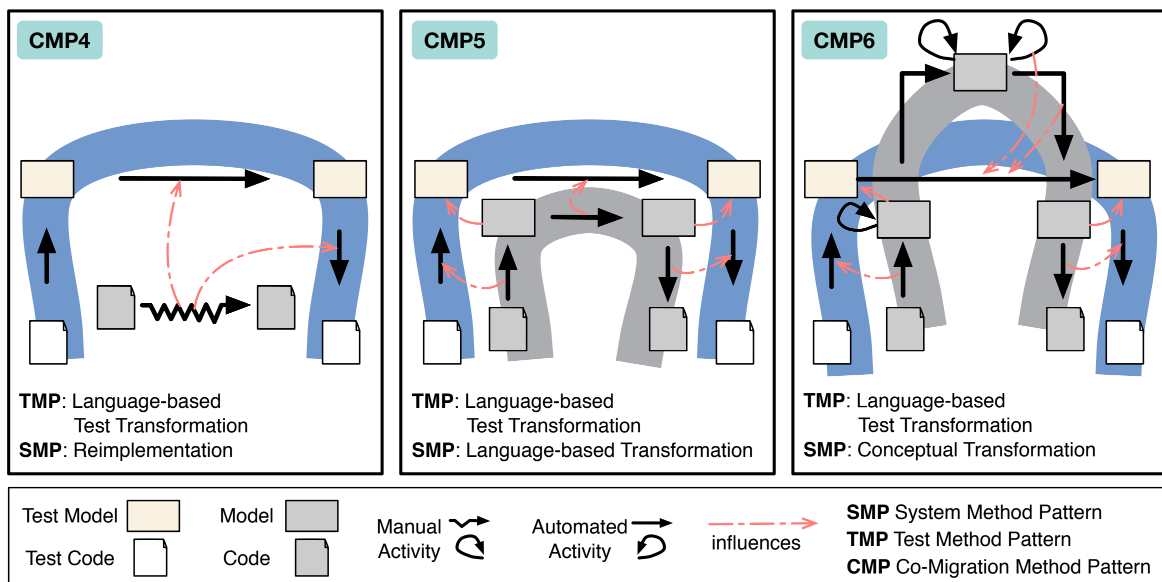


Figure 6.16 Co-Migration Patterns containing Language-based Test Transformation

The pattern *CMP4* is a combination of a *Language-based Test Transformation* and a *Reimplementation*. This pattern is suitable if the system reimplementation was well documented so that some transformation or code generation rules can be derived in order to automate the transformation of the test cases. However, it suggests the implementation of a parser for the source language as well as a code generator for the target language. Similar to the previous co-migration patterns, no system method fragments could be directly reused.

The pattern *CMP5* is a combination of *Language-based Test Transformation* and *Language-based Transformation*. This pattern has symmetric constellation, as two transformations on the same abstraction level are combined. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be completely reused. The reuse of existing method fragments is also possible in the scope of the transformation step (e.g., metamodels or transformation rules). However, the complexity of the transformation could be higher if the source and the target frameworks differentiate a lot, meaning that the transformation of the test relevant concepts should be done implicitly. Regarding the impacted test method fragments, the *Language Transformation* activity is impacted by the corresponding method fragment from the system transformation method.

The pattern *CMP6* is a combination of *Language-based Test Transformation* and *Conceptual Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be completely reused. Due to the difference in the abstraction levels, the reuse of existing method fragments in the scope of the transformation step is only possible in an indirect way. Namely, the transformation on the conceptual level could be used as an input when the language transformation of the test cases is performed, i.e., the conceptual transformation parameterizes the language-based test transformation. Similarly as with *CMP5*, the complexity of the transformation could be higher if the source and the target frameworks differentiate a lot due to the implicit transformation of the test concepts.

Co-Migration Patterns containing Test Language-based Test Transformation

Figure 6.17 depicts the co-migration patterns that combine the *Test Language-based Test Transformation* method pattern with the three possible system migration patterns *Reimplementation* (*CMP7*), *Language-based Transformation* (*CMP8*), and *Conceptual Transformation* (*CMP9*).

The pattern *CMP7* is a combination of a *Test Language-based Test Transformation* and *Reimplementation*. This pattern is suitable if the system reimplementation was well documented so that some transformation rules can be derived in order to automate the transformation of the test cases. However, it suggests the implementation of a parser for the source language as well as a code generator for the target language. Furthermore, a test case understanding fragment and test case concretization fragment should be configured and implemented in terms of model-to-model transformations.

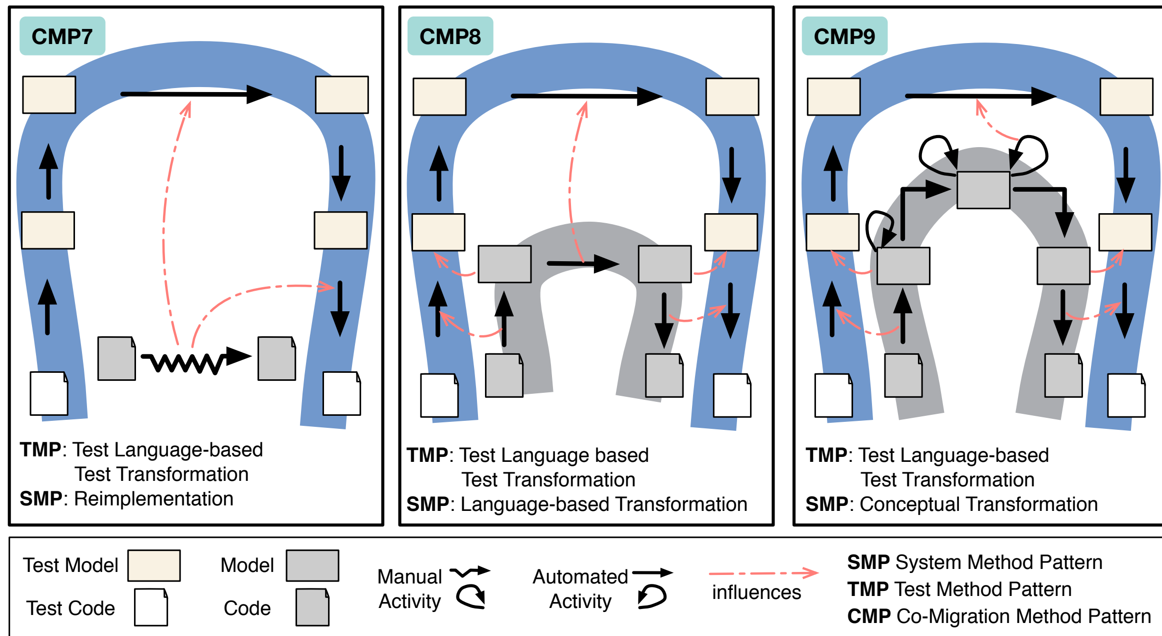


Figure 6.17 Co-Migration Patterns containing Test Language-based Test Transformation

The pattern *CMP8* is a combination of *Test Language-based Test Transformation* and *Language-based Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be reused. Reuse of existing method fragments is also possible in the scope of the transformation step. But a test case understanding fragment and test case concretization fragment should be still selected and implemented in terms of model-to-model transformations. However, the complexity of the transformation is lower compared to *CMP7*, as the transformation activity from the system method pattern could be reused to a higher extent as it is specified explicitly through a model-to-model transformation. On the other side, the complexity of the transformation is lowered as an explicit mapping between the testing languages is defined.

The pattern *CMP9* is a combination of *Test Language-based Test Transformation* and *Conceptual Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be completely reused. Due to the difference in the abstraction levels, the reuse of existing method fragments in the scope of the transformation step is only possible in an indirect way. Namely, the transformation on a conceptual level could be used as an input when the *Test Language-based Test Transformation* is configured performed, i.e., the *Conceptual Transformation* parameterizes the Test *Language-based Test Transformation*. Similarly to

CMP8, the complexity of the transformation is lowered as an implicit mapping between the testing languages is defined.

Co-Migration Patterns containing Conceptual Test Transformation

Figure 6.18 depicts the co-migration patterns that combine the *Conceptual Test Transformation* method pattern with the three possible system migration patterns *Reimplementation* (*CMP10*), *Language-based Transformation* (*CMP11*), and *Conceptual Transformation* (*CMP12*).

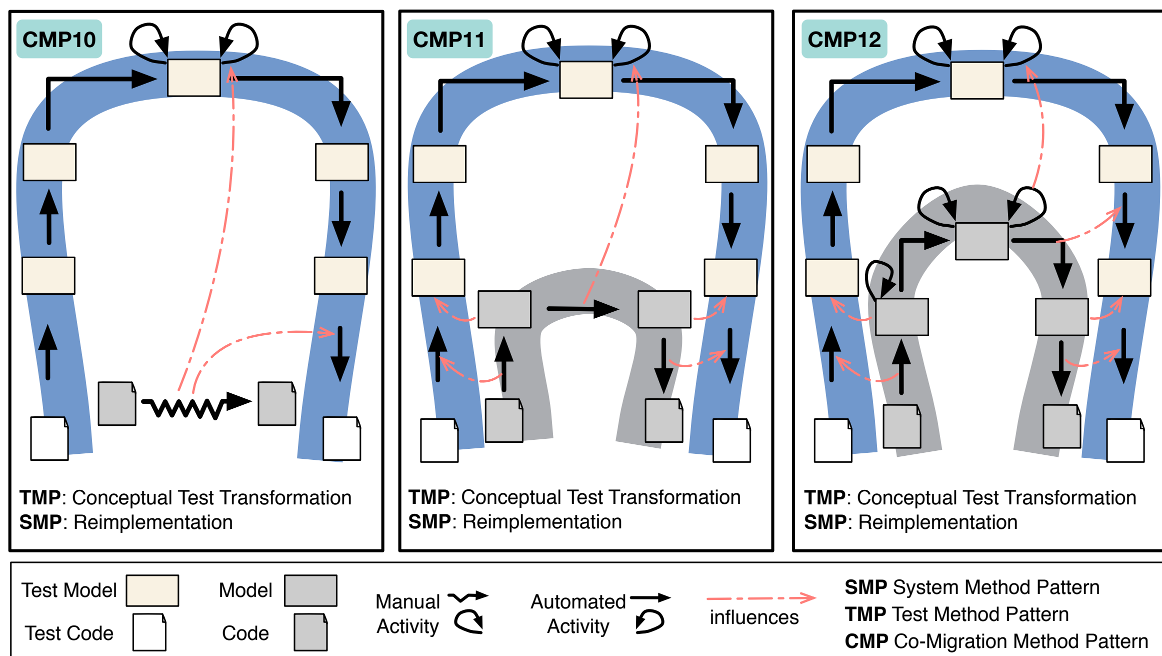


Figure 6.18 Co-Migration Patterns containing Conceptual Test Transformation

The pattern *CMP10* is a combination of *Conceptual Test Transformation* and *Reimplementation*. The suitability of this pattern depends on the system reimplementation, whether it was well documented so that some transformation rules can be derived in order to automate the transformation of the test cases. However, it suggests the implementation of a parser for the source language as well as a code generator for the target language. Furthermore, a test case understanding fragment and test case concretization fragment should be configured and implemented in terms of model-to-model transformations.

The pattern *CMP11* is a combination of *Conceptual Test Transformation* and *Language-based Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be

reused. The reuse of existing method fragments is also possible in the scope of the transformation step. But a test case understanding fragment and test case concretization fragment should be still selected and implemented in terms of model-to-model transformations. However, the complexity of the transformation is lower compared to *CMP10*, as the transformation activity from the system method pattern could be reused to a higher extent as it is specified explicitly through a model-to-model transformation. On the other side, the complexity of the transformation is lowered as an explicit mapping between the testing languages is defined.

The pattern *CMP12* is a combination of *Conceptual Test Transformation* and *Conceptual Transformation*. In such a constellation, both the reverse engineering and forward engineering fragments, *Model Discovery* and *Test Code Generation* respectively, can be completely reused. Due to the difference in the abstraction levels, the reuse of existing method fragments in the scope of the transformation step is only possible in an indirect way. Namely, the transformation on a conceptual level could be used as an input when a *Test Language-based Test Transformation* method pattern is configured, i.e., a *Conceptual Transformation* pattern parameterizes the *Conceptual Test Transformation*. Similarly to *CMP11*, the complexity of the transformation is lowered as an implicit mapping between the testing languages is defined.

6.2.5 Formalization

In the previous sections, we have introduced the test method fragments and test method patterns. We formalize them by introducing an intermediate representation called TeCoMi Intermediate Modelling Language (TIML). It extends the MEFiSTo Intermediate Modeling Language (MIML) [Gri16] by providing additional classes in the metamodel in order to address the test specific fragments and patterns.

Figure 6.19 shows an excerpt of the TIML metamodel with the two main packages, namely the `Fragment` and the `Pattern` package. The `Fragment` package contains the elements which are essential for specifying test methods, like artifacts, activities, tools, and roles. These rather generic method fragments are further refined to explicit language elements, e.g., `ModelDiscovery` activity or `TestParser` as a tool.

The `Pattern` package, on the other hand, enables the specification of `Test Method Patterns` and `Co-Migration Patterns`. A `Method Pattern`, in general, consists of fragments, namely, `TransformationPhaseFragments` and `ToolImplementationFragments`. Consequently, the same applies to the `Test Method Pattern`. As can be seen in Figure 6.19, `Co-Migration Pattern` consists of a `Test Method Pattern` and a `System`

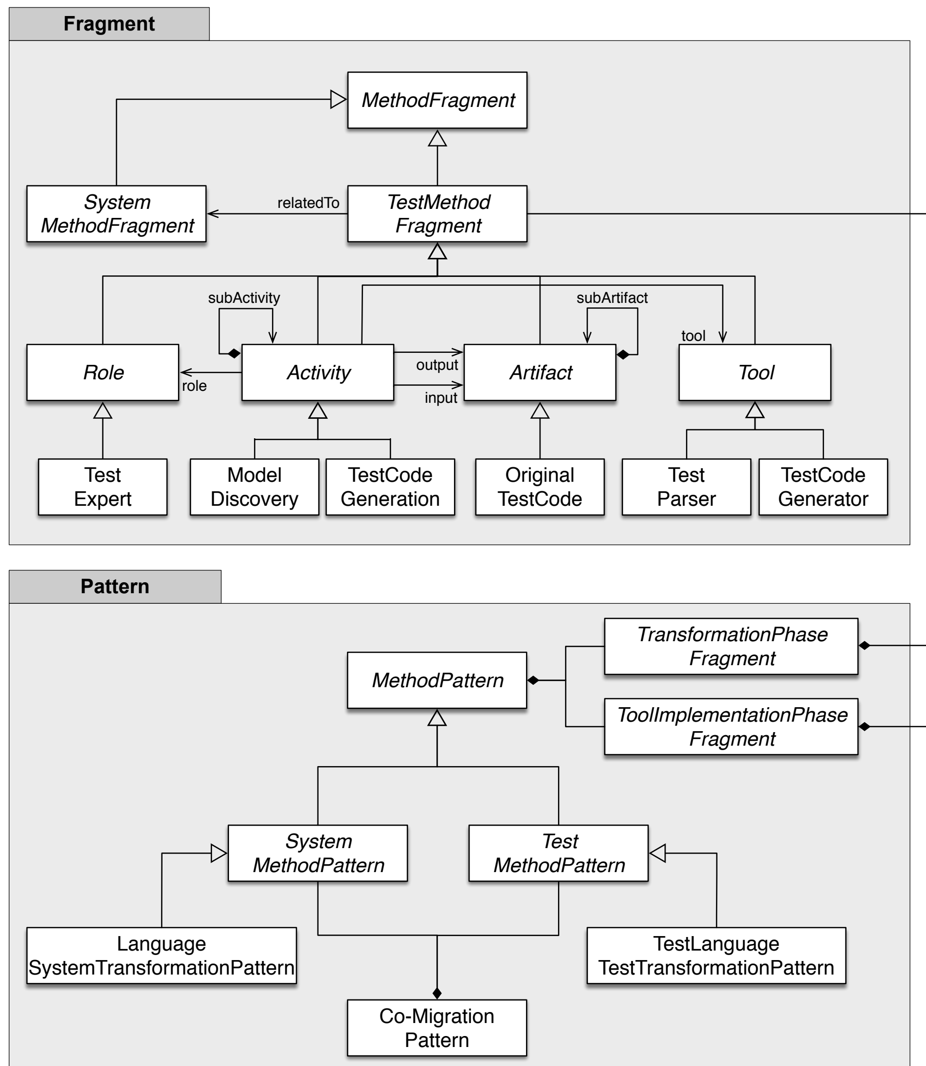


Figure 6.19 An excerpt of the TeCoMi Intermediate Modelling Language (TIML)

`Method Pattern`. Similarly to the `Fragment` package, for the generic classes, e.g., `TestMethod Pattern`, we provide explicit language elements, i.e., `TestLanguageTestTransformation`. Additionally, constraints can be used to specify, for example, which fragments can only be part of a given `Test Method Pattern` or `Co-Migration Pattern`.

6.3 Method Engineering Process

The process describes the main activities to be followed in order to create a context-specific test case migration method as well as their relation to the method base. As already shown in Figure 6.1, the activities are split into two main disciplines: *Method Development* and *Method*

Enactment. The essential process input is the *Original Test Code*, which gets transformed to *Migrated Test Code* once the process is being enacted. The *Migrated Test Code* represents the test cases which can be run in the target environment and validate the system migration. The activities of both disciplines are associated with a flow with attached *Context-Specific Migration Method Specification* which describes the actual migration method.

By performing activities of the *Method Development* discipline, a situation-specific method gets developed. The main activities are *Situational Context Identification* and *Transformation Method Construction*. During *Method Enactment* the situation-specific tools are developed that are required for the automation of the migration method or part of it. Thereafter, the migration method is performed as defined in the migration method specification. The main activities are *Tool Implementation* and *Transformation*.

6.3.1 Situational Context Identification

The first activity of the method engineering process is *Situational Context Identification*, in which the migration context is analyzed and characterized, from both system migration and testing perspective. Furthermore, co-evolution analysis is also performed to identify the impact that the system changes have on the test cases. The gathered knowledge about the migration context is required in order to develop a suitable test case migration method.

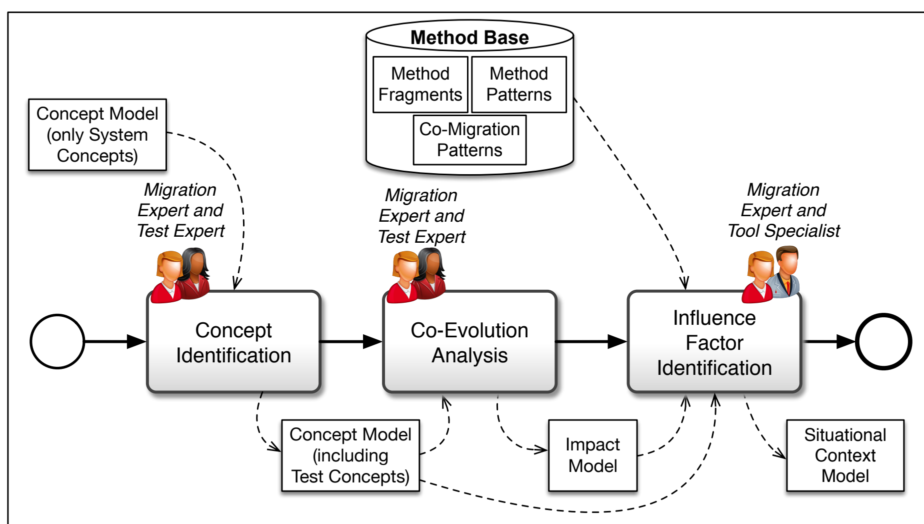


Figure 6.20 Situational context identification process

Concept Identification

The purpose of the *Concept Identification* activity is to model a decomposition of the system and the testing artifacts into distinct parts for both the source and the target environment. We use the principles of Concept Modeling [KNE92] to represent the functionality of the system as a set of concepts. The concepts are split into two different groups, namely language concepts and abstract concepts. Language concepts directly correspond to syntactic entities of the programming language, like variables, declarations, statements, etc. [KNE92]. The abstract concepts, on the contrary, represent language-independent ideas of computation and problem-solving methods [KNE92]. Abstract concepts are further classified into architectural and programming concepts.

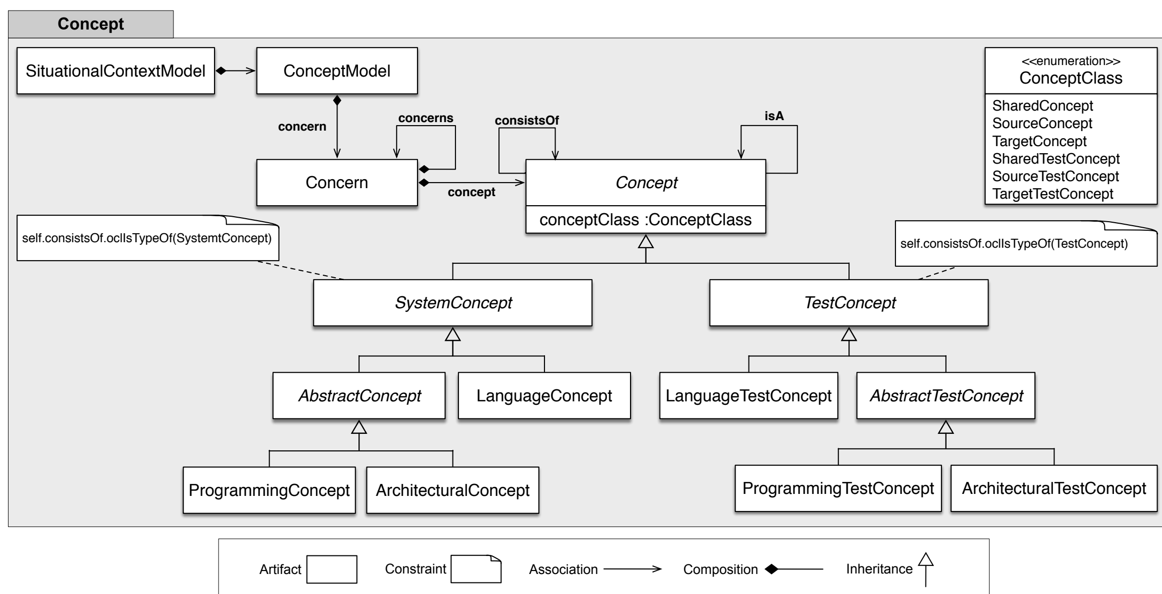


Figure 6.21 The concept metamodel to formalize the concept models in co-migration context

The architectural concepts are associated with interfaces or components whereas the programming concepts represent a general coding strategy, data structure or algorithm. Concepts can be related to each by *isA* relation, to express a hierarchy between concepts and *consistsOf* relation to express dependencies between concepts. In [Gri16], when applying the idea of Concept Modeling to software modernization, three classes of concepts are distinguished: original system concepts, target system concepts, and shared system concepts. Regarding the original system, the language concepts are determined by the language elements that are already used, whereas language concepts regarding the target system concepts are those language concepts that will be used after the transformation. Finally, a shared concept is an abstract concept of the original system that can be realized in the target

environment. All in all, the concept model is defined as a directed, acyclic, and connected graph. The nodes represent the concepts, whereas the edges between them represent *isA* or *consistsOf* relations. In the following, we present the concept model as well as the concept identification process.

As shown in Figure 6.21, our *Concept Model* is a part of the *Situational Context Model*. The *Concept Model*, which extends the concept model introduced in [Gri16], can contain a set of *Concerns* which can further contain sub-*Concerns* and a set of *Concepts*. Besides the *SystemConcept* subclass which expresses system related concepts, the *ConceptModel* contains an additional subclass for expressing test-related concepts, namely the *TestConcept* class. This class has additional subclasses like the *AbstractTestConcept* which is further specialized into *ProgrammingTestConcept* and *ArchitecturalTestConcept*, and the *LanguageTestConcept* for expressing concrete syntactic entities related to testing. As defined by the *conceptClass* attribute in the *Concept* class, each *Concept* belongs to one out of six classes as defined by the enumeration type *ConceptClass*. Furthermore, a *Concept* can be related to other concepts by the *consists-of* and *is-a* relations. The invariants of the concept model instances are ensured by the OCL constraints. For example, the two OCL constraints shown in Figure 6.21 define that the target of a *consists-of* relation for a *SystemConcept* or a *TestConcept* can only be another *SystemConcept* or *ProgrammingConcept*, respectively.

The concept identification process defines the necessary steps to be performed in order to capture the test and system concepts for both source and target environment.

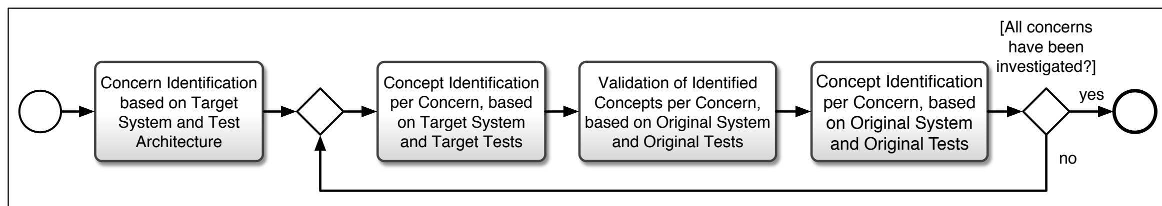


Figure 6.22 Concept Identification Process

The process is target-driven as the desired transformation outcome that is desired delivers the concepts to be identified. Firstly, a set of concerns is identified based on the system and test target architectures, which in turn provides a coarse-grained structure of the concept model. The next three activities are performed repeatedly for each concern. Firstly, concepts for the current concern are identified based on the target system and the target tests. This activity is based solely on the experience of the expert or by evaluating supporting materials like development or test tutorials. Figure 6.23 shows the concept model of the example introduced in Chapter 3. From a system perspective, we have identified the abstract architectural concept

OCL (AOT) which represents the *Ahead-of-Time (AOT)* realization of the shared abstract concept *Object Constraint Language (OCL)* in *CrossEcore*.

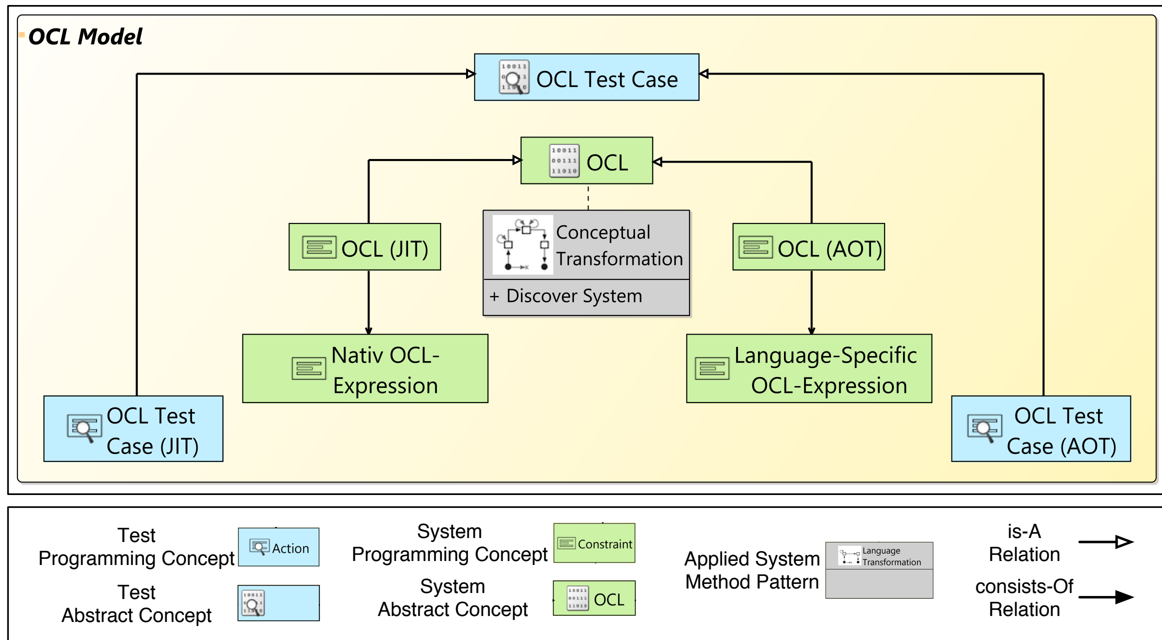


Figure 6.23 Concept model of the example scenario

It further consists of a concrete programming concept named *Language-Specific OCL-Expression*, which represents the OCL expression that is defined in *CrossEcore* by using language constructs. From a testing perspective, we have identified the *OCL Test Case* as a shared abstract test concept. As a target test concept, we have identified the language test concept *OCL Test Case (AOT)*. Secondly, by performing a detailed analysis of the original system and original tests the identified concepts are validated. This is necessary because the concepts have so far been identified without considering the original system and the original tests. Thirdly, a concluding source-driven identification takes place as the original system and original tests are analyzed to eventually identify additional concepts that are specific to the original system and the original test cases. From a system perspective, we have identified the abstract architectural concept *OCL (JIT)* which represents the *Just-in-Time (JIT)* realization of the shared abstract concept *Object Constraint Language (OCL)* in EMF. It consists of a concrete programming concept named *Native OCL-Expression* which represents the OCL expression in EMF. From a testing perspective, we have identified the source test concept *OCL Test Case (AOT)*. The obtained model which shown in Figure 6.23, contains system and test concepts. A more detailed version of this model is presented in the first feasibility study in Section 8.2.

Co-Evolution Analysis

Having identified the concepts, from both system and test perspective, in terms of a concept model, in this step a co-evolution analysis is performed. According to [MD08], the co-evolution process consists of several activities from which *Change Detection* and *Impact Analysis* are relevant during the situational context identification. During *Change Detection*, all changed parts of the system being migrated are identified. Having identified these changes, all affected parts of the test cases are identified in the next step called *Impact Analysis*. Finally, an estimate of the effort required to accomplish the changes together with involved risks is made. Therefrom, the main idea is to describe the changes that happened to the system by identifying and relating corresponding source and target system concepts to each other.

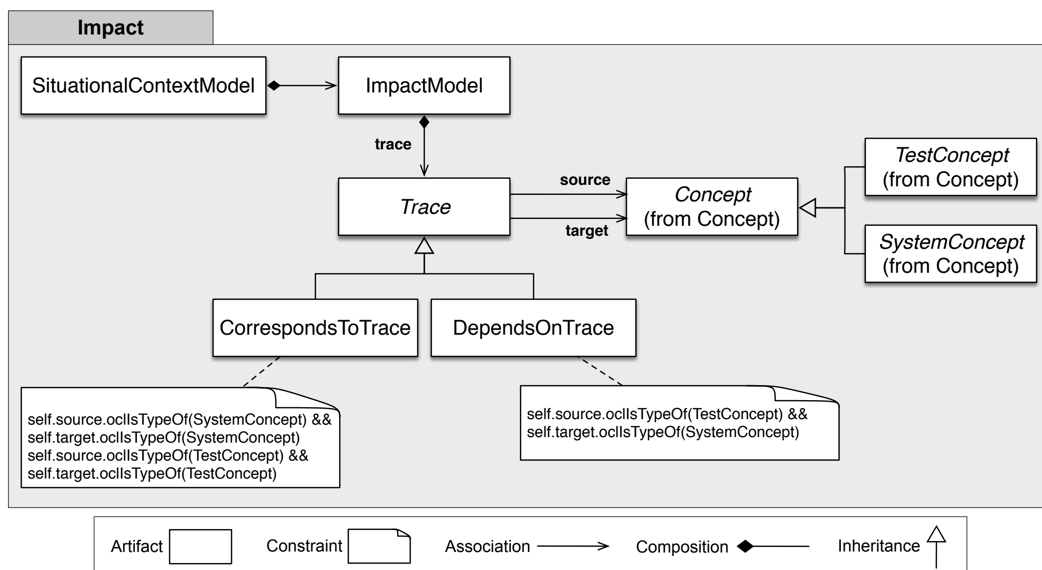


Figure 6.24 The impact metamodel to formalize the relation between the test and system concepts in the co-migration context

Then, the relation between the source and target test concepts to their corresponding system concepts is also established. Having these relations, the impact of the system changes on the test cases can be derived. In order to enable these activities, we provide a metamodel (shown in Figure 6.24) that formalizes the relations between the system and test concepts in terms of traces. Furthermore, we also introduce the impact analysis process which defines the necessary actions to perform the impact analysis.

As can be seen in the upper left of Figure 6.24, we consider the *ImpactModel* to be a part of the *SituationalContextModel*. The *ImpactModel* can contain a set of *Traces* which can be either *CorrespondsToTrace* or a *DependsOnTrace*. Each *Trace* has a *source* and a *target Concept*. These traces are needed in order to express the different types of relations between

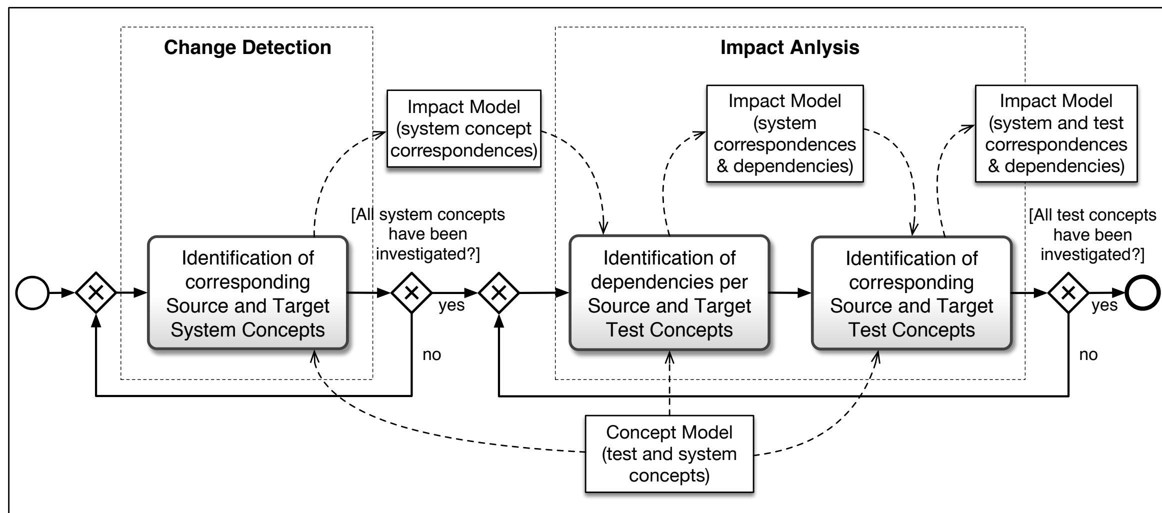


Figure 6.25 Co-Evolution Analysis Process

the concepts. In the case of a *CorrespondsToTrace*, as defined by the related OCL constraint, both *target* and *source* are of the type *SystemConcept*. This type of trace is used to express the relation between source system concepts and target system concepts, i.e., to relate the corresponding concepts from the two environments. On the other hand, a *DependsOnTrace*, as defined by the related OCL constraint, has as *target* a *TestConcept* and a *SystemConcept* as *source*. This type of trace is used to express the relation between test and system concepts in both the source and target environment. More precisely, it expresses the dependency the test concepts have on the system concepts.

As previously introduced, the co-evolution analysis has two main concerns, namely, to identify changes in how system concepts are realized system changes and to identify their impact on the test concepts (Figure 6.25). The co-evolution analysis process starts with the activity that addresses *Change Detection*. On the basis of the previously created *Concept Model*, for each source system concept, a corresponding target system concept is identified and a *CorrespondsToTrace* is created. The main idea behind the activity is to identify the differences, i.e., to detect the changes that happen to the realization of the system concepts. Then, as part of *Impact Analysis*, two activities are performed. Firstly, each test concept, both source and target, is related to the system concept that it tests by establishing a *DependsOnTrace*. Doing so, it is explicitly modeled on which system concepts a given test concept depends on. Finally, to complete the *Impact Model*, the correspondences between the source and the target test concepts are also established.

After the second activity is done, a complete traceability model is produced in terms of an *ImpactModel*. The traces in the *ImpactModel* express the actual impact the system changes

have on the test cases. Related to the example shown in Figure 6.26, the source system concept *Native OCL-Expression* corresponds to the target system concept *Language-specific OCL-Expression*.

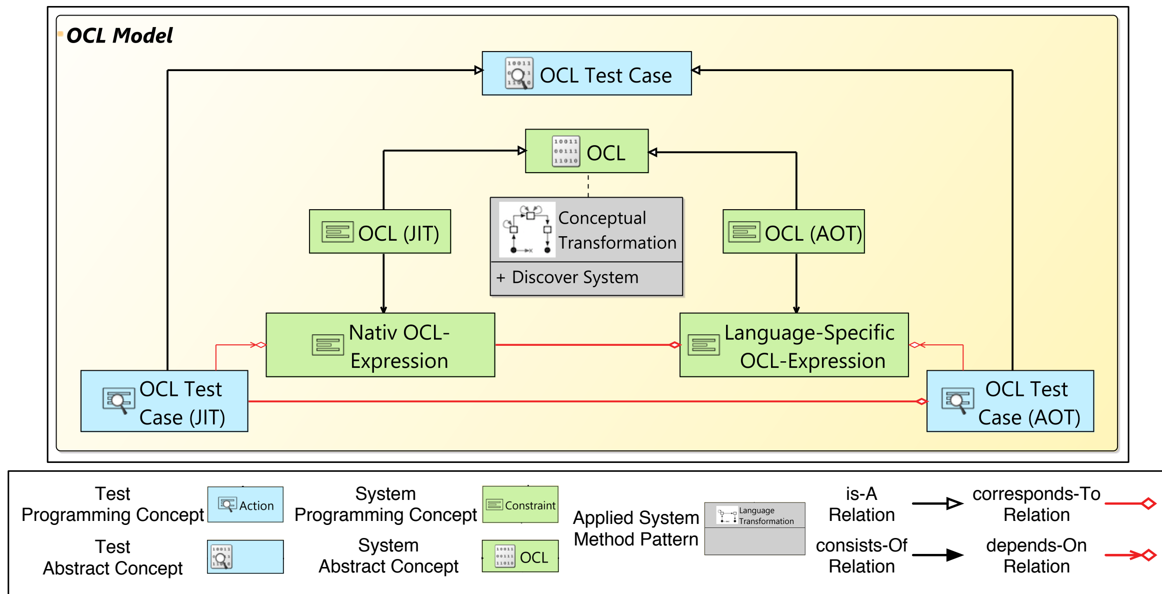


Figure 6.26 Impact model of the example scenario.

Regarding the test concepts, on the one hand, the source test concept *OCL Test Case (JIT)* depends on the source system concept *Native OCL-Expression*. On the other hand, the target test concept *OCL Test Case (AOT)* depends on the corresponding system target concept *Language-specific OCL-Expression*. These relations suggest indirectly in which way the tests are influenced by the system changes. Namely, the test cases have to be changed in a way that the contained part of the system should be changed in accordance with the correspondence relation between the system concepts *Native OCL-Expression* and *Language-specific OCL-Expression*.

Influence Factor Identification

In this activity, similarly to [Gri16], for each identified concept, a test method pattern is chosen. In order to decide which test method pattern to use, one needs to systematically search for characteristics that influence the pattern's efficiency or effectiveness. Each pattern has a set of characteristics which actually express its suitability to a certain situation. To support the influence factor identification, we provide an influence factor metamodel and a general guideline for identifying influence factors. The *InfluenceFactorModel* is a part of the *SituationalContextModel* (Figure 6.27).

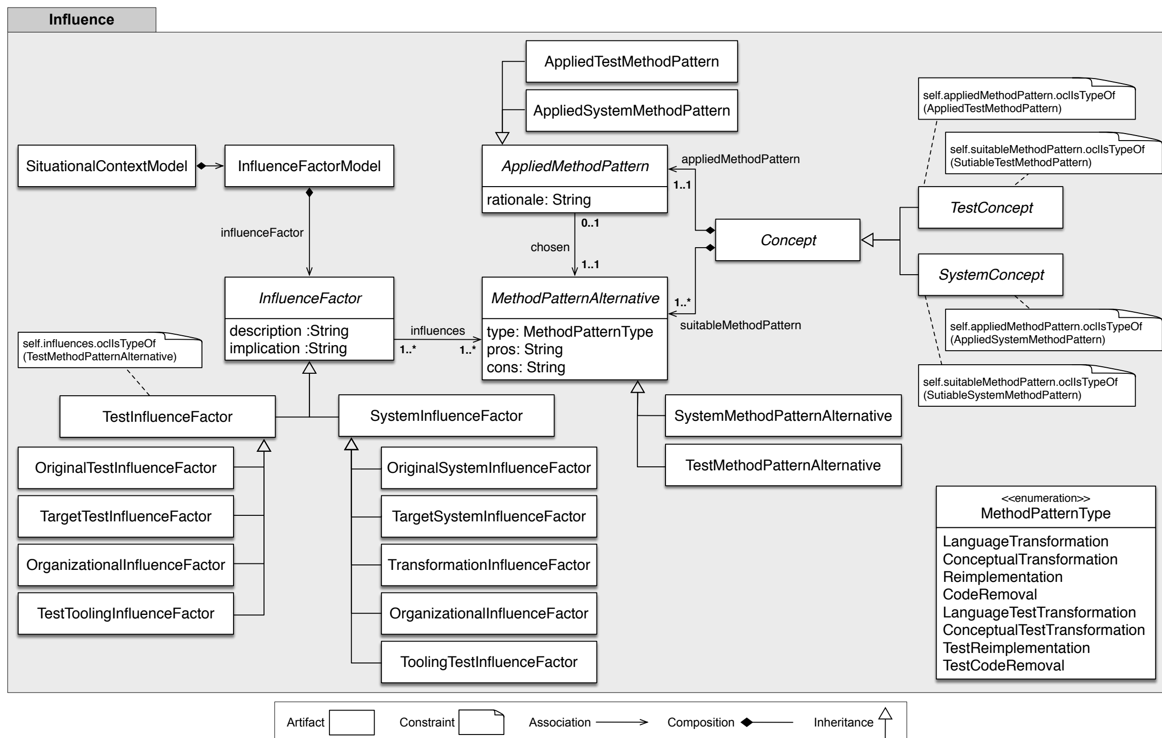


Figure 6.27 The influence factor metamodel for the co-migration context

It can contain a set of *InfluenceFactors*, further split into two subclasses: *TestInfluence-Factor* and *SystemInfluenceFactor*. An influence factor is defined as a characteristic of a co-migration project that has some impact on the efficiency or effectiveness of a transformation method. The *TestInfluenceFactor* is used to describe the test-related influence factors for both source and target environments as well as organizational and test tooling-related influence factors. Similarly, one can use the *SystemInfluenceFactor* to specify influence factors from a system perspective. Note that an additional class appears, namely the *TransformationInfluenceFactor*, which has the role to specify the influence that the system migration could eventually have on the test case migration. A given *Concept* is associated with a set of suitable *MethodPatternAlternatives*, showing that each *MethodPatternAlternative* can be influenced by *InfluenceFactors*.

Then, one pattern can be chosen to be applied, which is expressed by the *Applied-MethodPattern* class. For both *MethodPatternAlternative AppliedMethodPattern* classes, we distinguish between test and system migration patterns, indicated by the subclasses *SystemMethodPatternAlternative* and *TestMethodPatternAlternative*, and *AppliedSystem-MethodPattern* and *AppliedTestMethodPattern*, respectively. For example, the realization of a concept in the original system will influence all suitable method patterns. However, the *InfluenceFactor* only needs to be specified once and can be linked to all affected *Method-*

PatternAlternatives. In order to ensure invariants in the model, OCL constraints are used. For example, the OCL constraint related to the class *TestInfluenceFactor* defines that each *TestInfluenceFactor* can influence only a *TestMethodPatternAlternative*.

In the following, we describe the process necessary for the instantiation of an influence factor model (Figure 6.28).

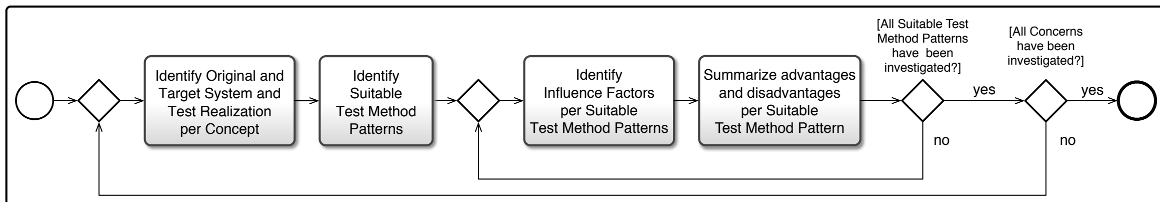


Figure 6.28 Influence Factor Identification Process

Firstly, a fine-grained analysis of the test and system realization, both source and target, for each concept is performed. In contrast to the coarse-grained superficial analysis from context identification, here we identify influence factors that may allow the exclusion of some of the possible test method patterns, thus reducing the overall analysis effort. For example, if the realization of the given concept in the original and the target environment is significantly different, the patterns which do not include conceptual transformation are not suitable. Once a set of suitable test method patterns is obtained, influence factors for each pattern are identified and described.

In order to systematize the process of identifying influence factors, each test method pattern is analyzed from a perspective of the comprising method fragments. For example, as illustrated in the third pattern (from the left to the right) in Figure 6.29, a method fragment defines that a parser is needed.

In this case, the availability of such a parser is an influence factor. The situational context model contains the identified abstract test concept, namely the *OCL Test Case*, its realization in the source test environment as well as the planned realization in the target test environment. Additionally, it also contains the assessed suitability of the possible test method patterns, in terms of effectiveness and efficiency. For example, regarding the leftmost method pattern, the testers would be challenged with re-implementing the test cases as they are not experienced with *CrossEcore*. The second pattern has been identified as not suitable as it does not provide a way to interact directly with the test concepts like the expected result or the test action. Having the influence factors identified, all suitable test method patterns are analyzed and assessed.

6.3.2 Transformation Method Construction

Having the context information collected, the *Transformation Method Construction*, as shown in Figure 6.30 activity can be initiated. In this activity, on the basis of the previously identified context information, a situation-specific migration method gets constructed.

First of all, a suitable pattern is selected, which in turn means a decision on how to transform the test cases. Once a pattern is selected, then it has to be configured, i.e., a set of method fragments has to be instantiated. Then, the instantiated fragments are customized regarding the functionality they are transforming. Customization of fragments means specifying them at a level of detail so that it provides guidance during the enactment. The outcome of this step and thus from the *Method Development* part is the *Context-Specific Migration Method Specification*. It describes how to perform the migration by defining the activities to be performed and the artifacts that should be generated and it serves as guidance during the enactment of the migration method.

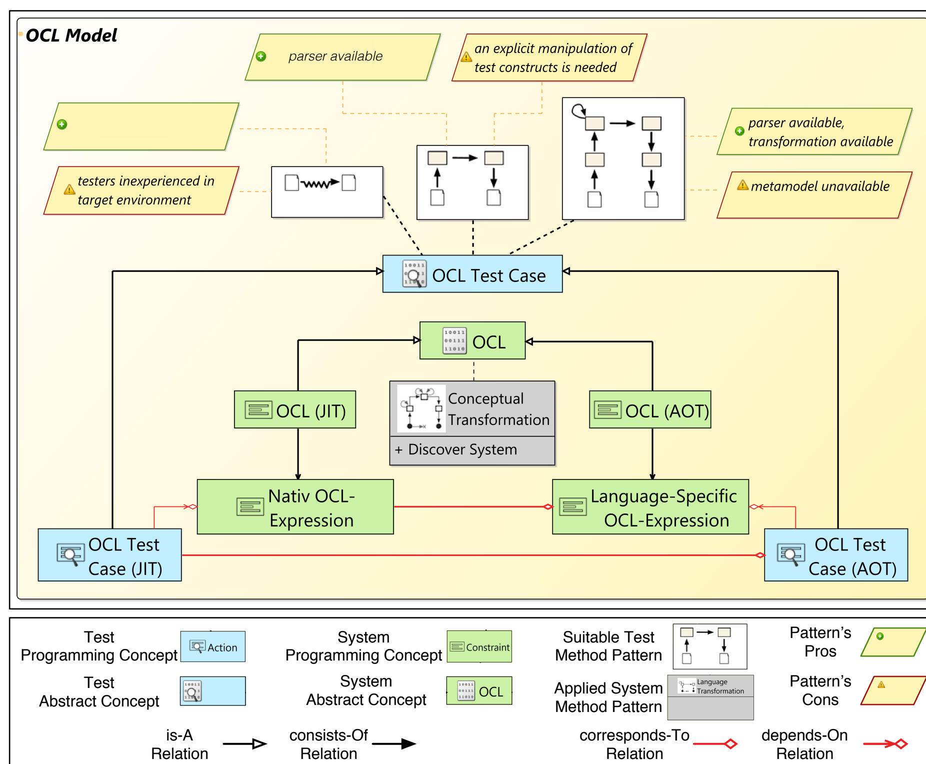


Figure 6.29 Excerpt of the situational context model showing the evaluated suitability of the test method patterns

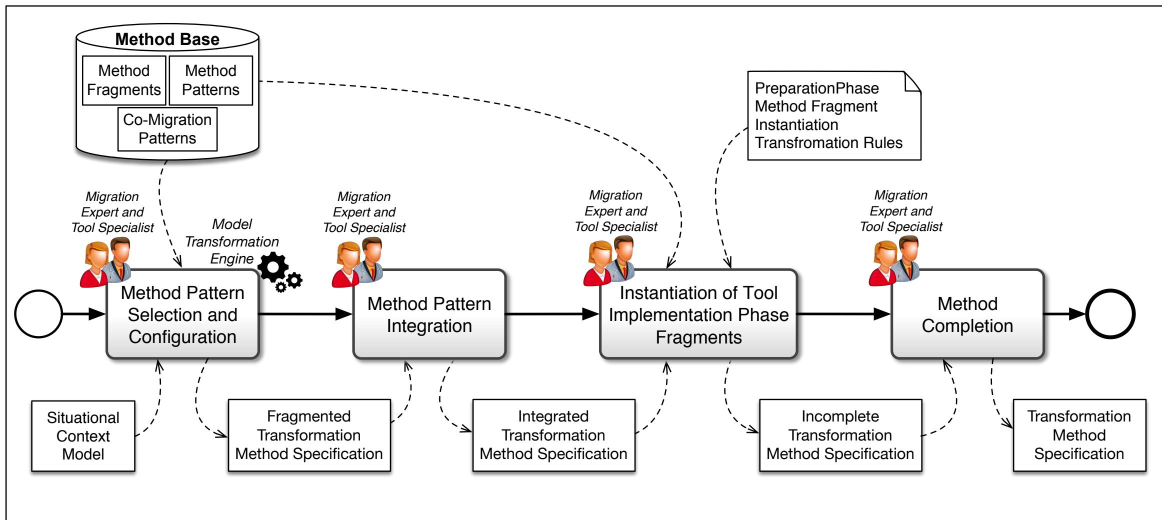


Figure 6.30 Transformation method construction process

Method Pattern Selection and Configuration

Until now, we have collected knowledge about the migration project in terms of a situational context model. This model is used to systematically derive a situation-specific test transformation method. Firstly, for each identified test concept, we select and configure a test method pattern. The core of the idea of this activity, namely *Method Pattern Selection and Configuration*, is shown in Figure 6.31.

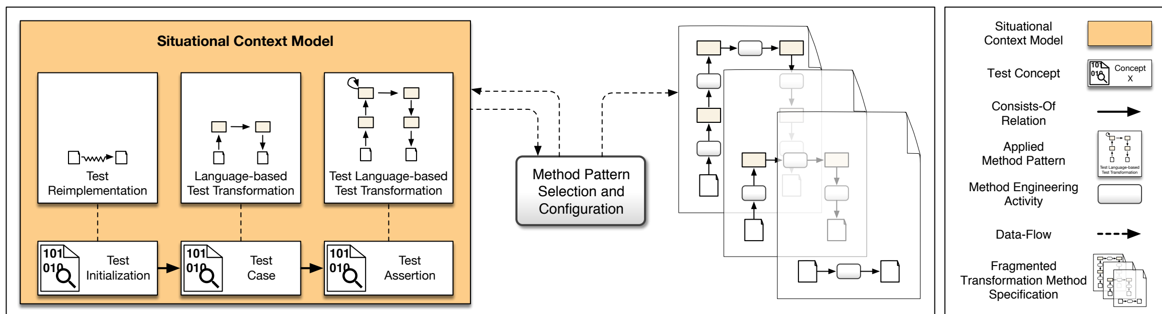


Figure 6.31 Instantiating a fragmented test transformation method specification from a situational context model

This activity starts with the situational context model, which, as already introduced in Section 6.3.1, contains a concept, an impact, and an influence factor model. The test concepts in the concept model decompose a test suite or the whole testware into distinct parts so that each part can be transformed by using a test method pattern. Each different identified concept can be, if necessary, transformed by applying a different pattern. Firstly, as we have already presented in Section 6.3.1, for each identified test concept a set of suitable

test method patterns has been identified. Furthermore, for each identified pattern a set of influence factors is associated which can be interpreted in order to determine how effective and efficient is the analyzed pattern.

The activity *Method Pattern Selection and Configuration* is an activity which consists of several sub-activities as shown in Figure 6.32. Firstly, a test method pattern has to be selected for each test concept, i.e., a decision on how a test concept should be transformed needs to be performed. This decision will essentially influence the overall efficiency of the transformation method and, therefore, it should be based on the identified influence factors. We assume, however, that in the situational context model, method patterns are already selected for each system concept. After performing the selection, as can be seen in Figure 6.31, each test concept contained in the situational context model is associated with an *Applied Method Pattern*. Then, a test transformation method specification gets automatically instantiated, on the basis of the obtained situational context model. However, the specification, at this point, contains only instances from elements of the `Pattern` package in TIML, e.g., Method Pattern and Method Pattern Configuration instances. Therefore, in order to obtain an initial test transformation method specification, as second the part of the process, a configuration of the instantiated transformation method needs to be performed. The configuration of the test transformation method is two-step. Firstly, a coarse-granular configuration of the applied method patterns is performed which comprises a selection of optional parts, e.g., enrichment activities, and the provisioning of meaningful names. This configuration is essential, as on its basis, a set of customized test method fragments gets automatically instantiated as prescribed by the applied test method pattern. The instantiated test method fragments are part of the `Fragments` package in TIML, e.g., `Model Discovery` or `Model of Original Executed Tests`. Once the customized test method fragments for one test concept according to the test method pattern applied are instantiated, the resulting set of fragments forms a horseshoe model. Similarly to the definition in [Gri16], we define a horseshoe model as follows:

Notation 17 (Horseshoe Model)**.** *A horseshoe model is a model which consists of a set of customized method fragments and conforms to a method pattern. The fragments specify a method to transform a concept.*

Finally, as a last activity, a fine-granular configuration takes place and the resulting set of test method fragments can be additionally configured. For example, the generated fragments can be renamed or removed, or additional ones can be added. As shown in Figure 6.31, the overall output of the *Method Pattern Selection and Configuration* activity is a fragmented test transformation method specification, i.e., a test method specification which comprises distinct horseshoe models. Similarly to the definition in [Gri16], we define a fragmented transformation method specification as follows:

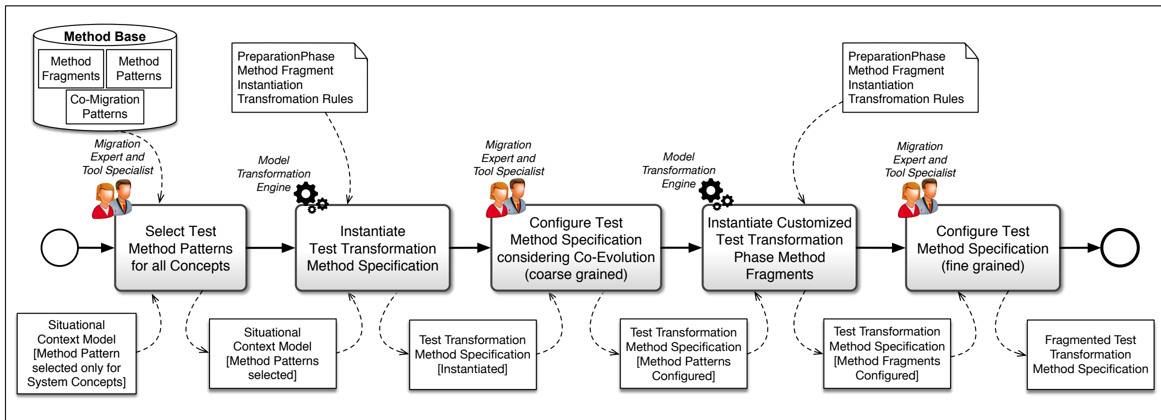


Figure 6.32 Method pattern selection and configuration process

Notation 18 (Fragmented Transformation Method Specification)**.** *A fragmented transformation method specification is a specification that consists of a set of horseshoe models that have not been integrated.*

Having the fragmented transformation specification, the next step is to integrate the different test method patterns and is addressed by the next activity of the method engineering process called *Method Pattern Integration*.

Method Pattern Integration

Until now, we have selected and configured test method patterns for all identified test concepts and based on this we have derived a fragmented test transformation method specification, as shown in Figure 6.31. In order to obtain a coherent specification, the distinct parts need to be systematically integrated. This is the purpose of the activity called *Method Pattern Integration* and its core idea is shown in Figure 6.33.

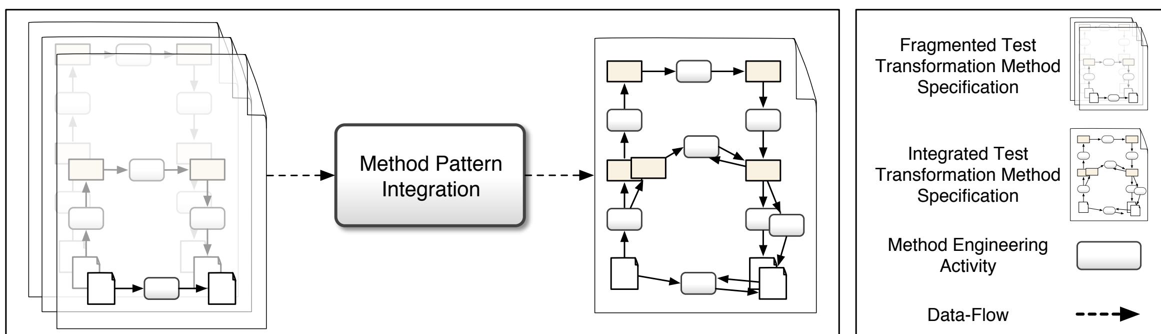


Figure 6.33 Integrating a fragmented test transformation method specification

So, the starting point of this activity is a fragmented test transformation specification. As we have already seen, this specification consists of customized test method fragments for each applied pattern. The customized test fragments inside one pattern form a horseshoe model and there is a relation between the fragments in terms of control-flow or data-flow relations. However, there is no relation between the different applied test method patterns, i.e., their horseshoe model manifestation. So, what we basically have, is a specification consisting of a set of separated specifications, as depicted on the left side of Figure 6.33. Therefore, we aim to integrate the separated specifications as part of this activity. The main goal is to establish relations between fragments from the different horseshoe models. In the end, this activity shall result in an integrated specification, as depicted on the right side of Figure 6.33. Similarly to the definition in [Gri16], we define an integrated transformation method specification as follows:

Notation 19 (Integrated Test Transformation Method Specification)**.** *An integrated test transformation method specification is a specification that consists of a set of horseshoe models whose method fragments have been integrated.*

The process of integrating the fragments needs to be systematized and according to [Gri16], there are different types of operations that can be executed. In general, there are two forms of integration of activities and artifacts. Interleaving is the most basic form of integration. When this form of integration is performed, then either control-flow or data-flow relations between fragments from the different applied patterns are introduced. This basically means, that the interleaving-based operations only introduce additional relations between method fragments. Merging-based operations, on the contrary, modify the method fragments, i.e., the application of the merging-based operations results in merging of two or more fragments. Hence, the merging fragments should originate from different applied patterns and should have the same type.

Instantiation of Tool Implementation Phase Fragments

So far, we have developed an integrated test transformation method specification. As we have already described, this specification consists of customized method fragments that describe the transformation phase. Within this activity, namely *Instantiation of Tool Implementation Phase Fragments*, the tool implementation phase is covered, and therefore the related customized fragments are added. Figure 6.34 visualizes the core idea of this activity.

As shown in the left side of Figure 6.34, the current state of the test transformation method specification contains only customized method fragments. These fragments, however, are

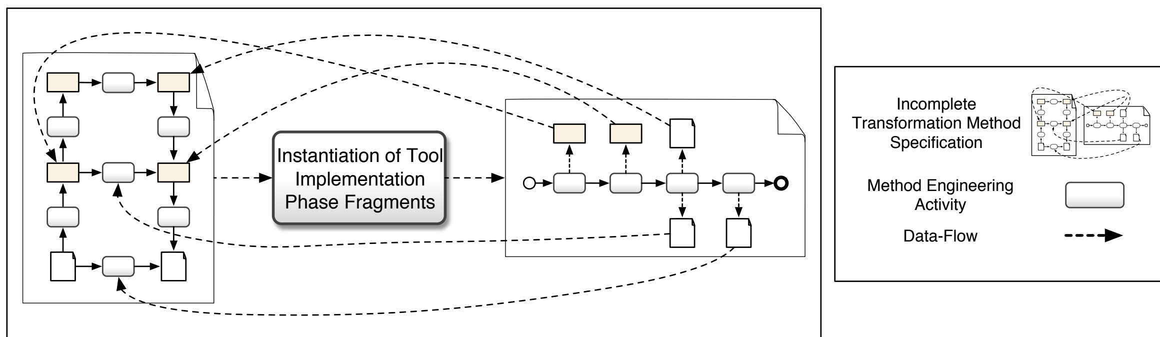


Figure 6.34 Instantiation of tool implementation phase fragments

not enough for the enactment of the transformation as various tools are needed. For example, in order to instantiate a test model, a test metamodel is required or manually performed reimplementation activities require guidance documents. So, the purpose of the *Instantiation of Tool Implementation Phase Fragments* activity is to enrich the existing specification by fragments to guide the development of the tools which are required, and it is shown in the right side of Figure 6.34. The tool implementation fragments are derived from the test transformation specification. Generally, the specification comprises two specifications, one specification that contains the fragments that specify the actual transformation, and another one that contains fragments that specify the development of the tools which are required.

6.3.3 Tool Implementation

Having the situation-specific method specification developed, we proceed to the enactment of the method. The first activity is the *Tool Implementation* activity and during this activity, we enact only those parts of the transformation method specification that specify the development of tools. We assume that the specification is used by the associated tool developers as some kind of guidance. In this case, we foresee at least two kinds of uses of the specification. The developers can use the specification to get an overview of the test migration method. By checking the transformation phase, they can get an understanding of how the actual transformation of the test cases should be performed. On the contrary, the part regarding the tool implementation phase specifies what needs to be developed. Furthermore, by reading the description of the tool implementation activities, the tool developers can stepwise enact the specification. These descriptions define what to do as part of each activity in order to develop the necessary tools. Currently, we assume this kind of more flexible kind of guidance.

Here we envision two possible outcomes, either development of a tool from scratch or reuse of existing tools. For example, if parser for parsing the test cases is needed, the one used for the system migration can be reused. Analogously, the metamodels can be reused as

well. However, if a tool deals with the transformation of test specific model elements, new transformation rules have to be created.

The result of the transformation of the test cases should be as correct as possible so that they can be used to validate the migrated system. Therefore, during and after the implementation of the required tools, their quality needs to be validated. Here, we focus more on the functional correctness of the tools, i.e., whether the implemented tool is functionally correct with respect to the tool specification. Therefore, we assume that the tool developers use a small amount of the original test cases to test the developed tools. Detailed examples about the development and the use of real tools are provided in Chapter 8 as part of the case study. In the following, we rather focus on the capabilities of a generic, project-independent tool infrastructure that should ease the implementation of the necessary tools. Similarly to the MEFiSTo framework, we expect that such tools need to be developed always when applying our framework.

The set of the capabilities described in the following is based on the method fragments that we have proposed in Section 6.2.1 and Section 6.2.2. In the following, we introduce the capabilities that support the automated, i.e., conversion-based, transformation strategies.

Model Repository. Test models representing a high number of test cases could be large. Furthermore, test models on different levels of abstraction, related to each other, may arise. A tool infrastructure shall, therefore, support their integrated management by providing a model repository for persisting or versioning of test models. In our case, we used *Neo4J*[5], a graph database management system, which provides high scalability for model management tasks.

Model Transformation. Transforming test models is essential when defining an automated transformation method in the TeCoMi framework. Therefore, this feature should be supported by the tool infrastructure. In the case study, we used the Java Development Tools (JDT)[6] for specifying model transformations.

Code Generation. Generating test code out of test models is essential when defining an automated transformation method in the TeCoMi framework. Therefore, this feature should be supported by the tool infrastructure. In TeCoMi, we rely upon Xtend[7], a statically typed programming language, which enables the specification of custom test code generators.

Parser Generation. The first activity when defining an automated transformation method using the TeCoMi framework is the parsing of test source code. The results of this activity are represented in the form of an Abstract Syntax Tree (AST) of executable test cases. Parsers for various programming languages already exist and can be used directly. However, if a

[5]https://neo4j.com/
[6]https://www.eclipse.org/jdt/
[7]http://www.eclipse.org/xtend/

suitable parser does not exist, it should be developed. The development of a parser can be supported by a parser generator, which in turn is a tool that generates a parser on the basis of proper grammar [ALSU06]. Therefore, the tool infrastructure should support the generation of a parser. Regarding the case study, as we had to parse JUnit test cases, we used an existing Java parser, namely the JDT parser, provided by the above-mentioned Java Development Tools.

6.3.4 Transformation

Up to this point, a situation-specific transformation method for the test cases has been developed. Additionally, tools which are required for the transformation have been developed. In this section, we present the last activity, namely the *Transformation* activity, of the method engineering process which deals with the enactment of the method as specified. The purpose of *Transformation* activity is to perform the actual transformation of original test cases as specified. Therefore, when performing this activity, only the specification parts that specify the transformation itself are enacted. We assume that associated software developers as well as test experts use the method specification as some kind of guidance. Similarly to the tool implementation activity, we foresee the same kinds of use. The people involved can browse the method to get an overview of the actual transformation of the test cases that shall happen. Moreover, they can enact the specification in a stepwise manner, as it specifies which activities and in what order they need to perform. When an activity gets performed, the provided descriptions can be used. Furthermore, separate guidance documents in terms of tooling manuals are also foreseen and if used, they have to be read. Compared to the activities belonging to the *Tool Implementation* activity, where all activities are performed manually, some of the activities belonging to the *Transformation* activity are performed automatically. This implies that the interaction between people using the tools and tools during the transformation needs to be addressed. More specifically, whenever a tool creates an output, the people using them must be notified on which basis they need to perform an activity. Furthermore, whenever an activity is performed, and a tool needs that output, the person who performed the activity must know which tool shall be invoked and how. Detailed examples of the transformation of real-world test cases are provided in Chapter 8 as part of the case study. Similarly to MEFiSTo [Gri16], we apply an incremental enactment of the test transformation method. The incremental enactment shall enable the incremental transformation of the test cases and not transforming all the test cases at once. Doing so, we reduce the for the test case migration project to fail.

6.4 Summary and Discussion

The TeCoMi framework enables the modular construction of situation-specific test transformation methods based on the reuse of methodological knowledge stored in a method base. In this chapter, we introduced the main phase of the TeCoMi framework, namely the migration phase, with its two main constituents, the method base and the method engineering process. In the first part of the chapter, we introduced the content of the method base, namely a set of test method fragments, test method patterns, and co-migration patterns. Thereafter, we introduced a process to develop and enact situation-specific test transformation methods which considers co-evolution analysis. Some of the findings presented in this chapter are based on master theses [Tho20, Vin20].

First, in Section 6.1, we gave an overview of the migration phase, by giving an overview of the structure of the method base as well as the structure of the method engineering process. Then, in Section 6.2, we proposed a set of test method fragments that are stored in the method base of TeCoMi. In the context of the TeCoMi framework, a method fragment is an atomic part of a method, namely an artifact, activity, role or tool. The fragments are classified based on the phase they belong to, namely the tool implementation or transformation phase. Thereafter, we proposed a set of test method patterns that are also stored in the method base of TeCoMi. In the context of the TeCoMi framework, a method pattern encodes methodological knowledge of transformation methods. Basically, each test method pattern defines a specific transformation strategy by indicating which test method fragments to customize and also how to assemble them. Besides the test method patterns, we introduced the co-migration patterns which are a combination of a system method pattern and a test method pattern and encode the relation between those two patterns. Both test method patterns and co-migration patterns were discussed in terms of their characteristics and also examples were provided.

In Section 6.3, we introduced the method engineering process, by discussing in detail each of the core activities. Firstly, we introduced a process to systematically discover and model the situational context from both system and test perspective, and also to identify the impact the system changes may have on the test cases. We firstly develop the concept model that describes the test cases through a set of distinct concepts. Based on this model and the concept model containing the system concepts, we analyze the dependencies between the system and test concepts and model those relations by an impact model. Lastly, for each identified test concept a set of suitable test method patterns is identified. Subsequently, for each test method pattern, a set of related influence factors that affect their efficiency or effectiveness is also identified and modeled as an influence factor model. Based on the discovered situational context, as part of the next activity of the method engineering process, we described the process to systematically construct a test transformation method. Firstly,

for each concept in the concept model is decided how to be transformed by selecting which test method pattern should be applied, thus configuring the concept model. Then, a set of horseshoe models is derived and each of them specifies a test transformation method. In order to form a coherent method, the horseshoe models are integrated. Additionally, the specification is extended by a part which guides the development of the project-specific tools. Next, the capabilities of a generic tool infrastructure are discussed so that support for the development of project-specific tools is provided. Finally, we discuss the last two activities of the method engineering process which are concerned with the enactment of the developed transformation method specification. The more detailed description of the method enactment can be found in Chapter 8 as part of the feasibility study.

The test method fragments we have presented are placed on different levels of abstraction, from system layer to platform-independent layer. However, as was out of the scope of this work, we did not focus on the behavioral model of the system. This model is actually the basis for the creation of the abstract model of the test cases. For example, a standardized language like the OMG's UML Testing Profile (UTP) [OMG13b] can be used for the representation of behavioral models. Providing this artifact as well as the corresponding activities for abstraction and concretization would enable migration towards model-based or even model-driven testing. As we have seen, the concept modeling is supported by A Sirius modeling editor but it is a manual activity, performed by the migration and test experts. Once test and system concepts are modeled, as part of the impact analysis, the correspondences and dependencies are also manually identified and modeled. This manually performed step requires a lot of knowledge and could be error-prone. Incorporating analysis of the code dependencies between test code and system code into the impact-analysis part could assist the experts when performing this activity. Lastly, the co-migration patterns are used to support the creation of test method patterns by explicitly showing the relation between the system and the test case method fragments. Still, at the current version of the Sirius modeling editor, the configuration of the test method fragments should be done manually. As the co-migration patterns provide the relations between the system and test fragments, an already configured system transformation method could be used as a starting point to configure a test transformation method. Doing so, it can save time and can reduce the chance for error when performing the method configuration.

In the next chapter, we introduce the third and final phase of the TeCoMi framework, namely the Post-Migration Phase, which deals with the validation of test migration.

Chapter 7

Post-Migration Phase: Migration Validation

In the previous two chapters, the pre-migration and migration phases of the TeCoMi framework were introduced. In this chapter, we introduce the third and the last phase, namely migration validation. First, in Section 7.1, we give an overview of the mutation analysis repository and the migration validation process. Subsequently, in Section 7.2, we introduce the mutation analysis repository which contains mutation analysis strategies to identify false positives and false negatives among the migrated test cases. More precisely, we present a set of different mutation scenarios, mutation operators, and mutation method patterns. Then, in Section 7.3, we introduce the migration validation process which guides the process of development and enactment of validation methods. Finally, in Section 7.4 the findings of this chapter are summarized.

7.1 Overview of the Post-Migration Phase

In this section, we firstly give a brief introduction to the problem of validation of test case migration and, then, we introduce our solution approach. Test case migration is the process of transferring test cases to a new environment without changing their ”functionality”, i.e., without changing the expected system behavior the test cases assert. As migrated test cases are used to validate system migration, validating test case migration is clearly crucial. Test case migration is in general far from trivial as several challenges need to be addressed [JGY16] and, due to the tight coupling of system migration and test migration, validating test case migration is especially challenging.

According to Fowler [FB99], refactoring is defined as the process of improving the internal structure of a system without changing its observable behavior. Code refactoring is similar to system migration but is simpler as correctness can be clearly equated to all test cases passing. Test cases with an adequate coverage of the system are thus a safeguard for code refactoring. Test case refactoring is more challenging than code refactoring as it is no longer trivial to ensure correctness. Clearly, a refactored test case must still pass when executed against the current system (which is correct with respect to the original test cases). This is not sufficient, however, to guarantee that the refactored test cases reject all incorrect systems. To tackle this challenge, mutation analysis [LS78] can be used to examine the quality of the refactored test cases; mutants of the system are created and these should be detected (killed) by the refactored test cases.

While this has inspired our application of mutation analysis to test case migration, it is also clear that there is a substantial difference: refactored test cases can be executed on the same system as the original tests while this is typically *not* the case for test case migration. Test case migration is typically coupled with corresponding system migration and migrated test cases can only be executed on the migrated system. As a consequence, applying mutation analysis to validate test case migration is a challenging task.

In the following, we give an overview of the post-migration phase of TeCoMi, where a context-specific validation method gets developed and enacted in order to validate the migrated test cases. As shown in Figure 7.1, we support this phase by providing a migration validation process which relies on mutation analysis repository that provides mutation analysis scenarios, patterns, and mutation operators.

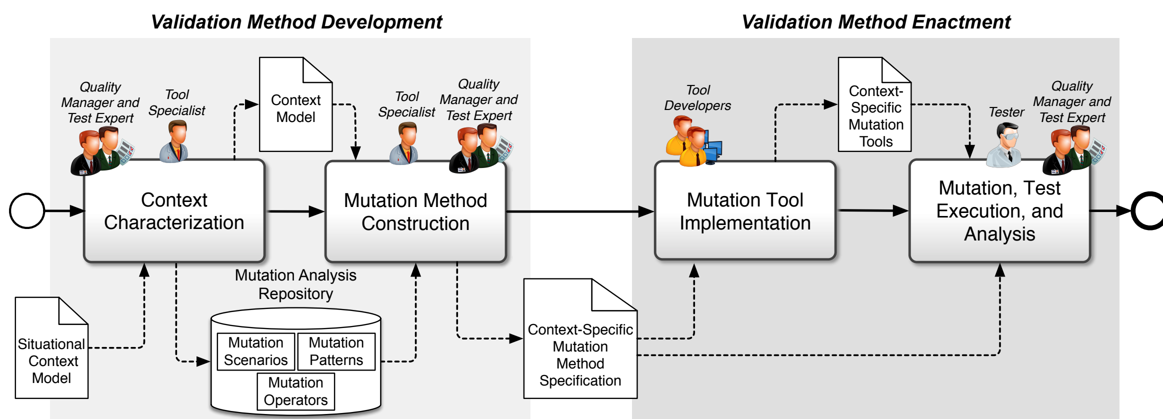


Figure 7.1 Core activities of the migration validation process of TeCoMi

In the following, we briefly give an overview of the content of the mutation analysis repository and the migration validation process.

Mutation Analysis Repository

The purpose of the *Mutation Analysis Repository* is to provide guidelines in terms of scenarios of mutation analysis and patterns for the specification of test case validation methods. As shown in Figure 7.2, the *Mutation Analysis Repository* contains *Mutation Analysis Scenarios*, *Mutation Method Patterns*, and *Mutation Operators*.

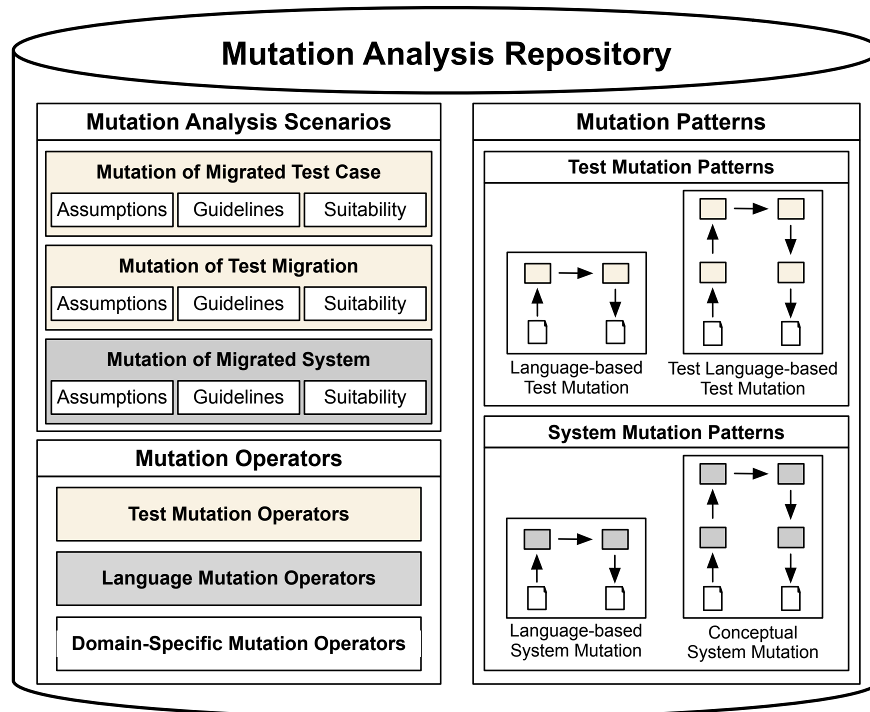


Figure 7.2 Core Components of the Mutation Analysis Repository

In total, we have defined six usage scenarios depending on what is mutated: the original or the migrated system, the original or the migrated test cases, or the system or test cases migration. For each usage scenario, we define a set of assumptions which must hold for a particular scenario to be feasible, e.g., that an appropriate mutation framework exists or that existing test cases can actually be executed. Based on these assumptions, we provide an in-depth analysis of indications that can be obtained. Indications are presented as “bad smells” for the test case migration, i.e., problematic test cases. Finally, we provide a discussion of the suitability of each usage scenario, taking implied assumptions into account.

A *Mutation Method Pattern* is the technical implementation of a given scenario. Depending on the test case and system migration context, an appropriate mutation method pattern regarding the abstraction level has to be selected. After a pattern is selected, a set of suitable *Mutation Operators* has to be defined. Depending on the abstraction level the mutation is

defined on, the mutation operators have to be correspondingly specified. Mutation operators can be either language-specific, test framework-specific or domain-specific.

Migration Validation Process

Relying on the mutation analysis repository, the migration validation process, shown in Figure 7.1, guides the development and the enactment of the situation-specific validation method. In Section 4.4, we briefly introduced the core activities of the migration validation process. Each of the four core activities is assigned to one of the two main disciplines, namely *Method Development* and *Method Enactment*. By performing activities of the *Method Development* discipline, a situation-specific validation method gets developed. It comprises the following two activities: *Context Identification* and *Mutation Method Construction*.

Firstly, during the *Context Identification* activity, information is collected about the context where the mutation should take place. This analysis relies on the *Situational Context Model*, the *Test Transformation Method Specification*, and the *Situation-Specific Tool Chain* from the migration phase of the TeCoMi framework. Based on this information, as part of the *Mutation Method Construction* activity, firstly, a suitable mutation analysis scenario is selected. Then, depending on the functionality being asserted in the test cases, a set of mutation operators is chosen. Finally, a suitable mutation method pattern for the selected scenario and mutation operators is selected. The outcome of this activity is specified in terms of a *Context-Specific Mutation Method Specification*.

Once this specification is obtained, during the *Mutation Tool Implementation* activity, a toolchain is implemented in order to automate the process of test case mutation. At this point, we rely on our generic and flexible model-driven tool infrastructure, which enables a common platform for performing mutation analysis in an automated way. Finally, as part of the *Mutation, Test Execution, and Analysis* activity, the method is enacted and the mutants are created. The mutated test cases are then executed against the migrated system and the outcome of test case execution is then analyzed. If no false positives and false negatives are identified, the migration of the test cases is considered to be successful. Otherwise, the identified false positives and false negatives have to be further analyzed.

7.2 Mutation Analysis Repository

In this section, we introduce the *Mutation Analysis Repository* which contains the knowledge of the available mutation validation strategies in terms of *Mutation Analysis Scenarios*, *Mutation Method Patterns*, and *Mutation Operators*. The central constituent of the repository are the *Mutation Analysis Scenarios* which describe how mutation analysis can be performed

and how the results are interpreted. A mutation cannot be performed without mutation operators and, therefore, we have *Mutation Operators* in the repository as well. As the mutation analysis scenarios describe a mutation analysis from a conceptual point of view, we present a set of *Mutation Method Patterns*, which are used for the technical realization of the scenarios. In the following, we describe each constituent of the *Mutation Analysis Repository*.

7.2.1 Mutation Analysis Scenarios

Mutation analysis or mutation testing is a technique used to create migrated test cases as well as to evaluate the quality of existing test cases [LS78, HG77]. It involves the modification or mutation of an existing system *S* by making small syntactic changes to create mutants *S'*, *S"*, etc. A transformation that creates a mutant from an existing program is called a mutation operator (also known as a mutant operator, mutagenic operator, mutation rule [Won01]). Ideally, there should be no *equivalent mutants* among generated mutants, i.e., mutants whose behavior is indistinguishable from that of the original system.

The general idea of mutation analysis is to seed small faults in the system and then execute an existing test suite *TS* against the mutants. There are in general two possible outcomes: (i) there is at least one test case that "detects" the change introduced in the mutant or (ii) no test case "detects" the change. In the first case, the mutant is considered to be killed, in the second case not. In general, there are two possible explanations why a mutant is not killed: (i) either the mutant is equivalent, or (ii) the test cases were not able to detect and reject the erroneous system behavior.

In order to explain the applicability of mutation analysis in the test case migration domain, we have analyzed the basic migration setting depicted in Figure 7.3. Our running example, which we already introduced in Section 3.1, is depicted again in Figure 7.4. It can be seen as a concrete instance of the abstract schema depicted in Figure 7.3.

To prepare for the presentation of the scenarios, in the following, we define the fundamental concepts to give, at least in the scope of this thesis, precise meaning to sentences such as "the system migration is correct" or "the test case migration is buggy":

Definition 14 (System)**.** *A collection of components organized to accomplish a specific function or set of functions [IEE90].*

Definition 15 (System Migration)**.** *System migration is a (manually performed or automated) transformation that takes an old (software) system as input and produces the migrated (software) system as output.*

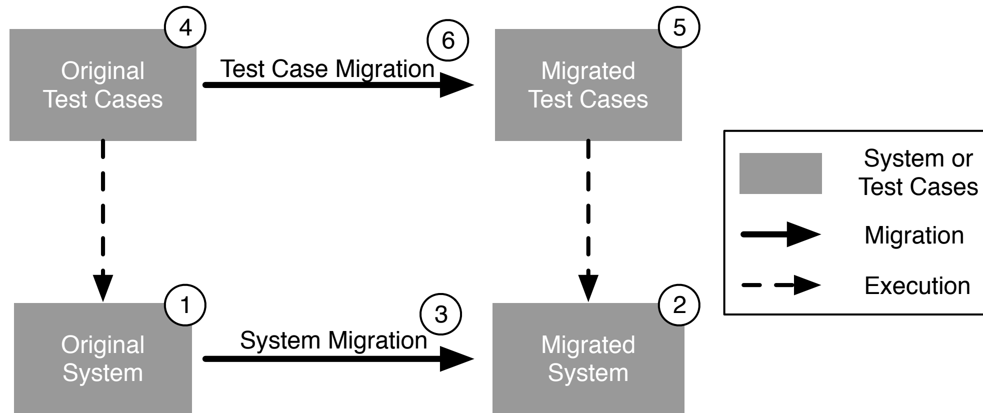


Figure 7.3 Overview of mutation analysis usage scenarios

System migration is considered correct if the functional equivalence of the original and migrated system is ensured. The notion of equivalence has different definitions in different areas of computer science like software design, program verification or database theory. Generally speaking, two things can be considered equivalent if they are "regarded as mutually compensating each other, or as exchangeable" [SW89]. Similarly to [MA14], we define the equivalence around the term exchangeability. Hence, two equivalent systems are indistinguishable to an external observer which means that the observable behavior won't be changed when the systems are exchanged.

Definition 16 (Correctness of System Migration)**.** *Given a system migration that transforms an original system S to a migrated system S', let sm denote the corresponding transformation of the original system S to the migrated system S'. The system migration is* correct *if S and S' are equivalent according to a particular equivalence relation.*

As our framework is meant to deal with real-world, very often also large, software systems, it is very difficult to formally verify full equivalence. Therefore, the notion of behavioral equivalence is, similarly to [MA14], relaxed to ad hoc equivalence which refers to equivalence relation that is satisfied for a particular purpose.
We now move on to test cases. On the same abstraction level as for the system, a test case is "executed" (see the arrows labeled *ex* in Figure 7.4) on the system by using it to stimulate and assert the beginning and end state of the corresponding system behavior.

Definition 17 (Test Case)**.** *A test case consists of a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement [IEE90].*

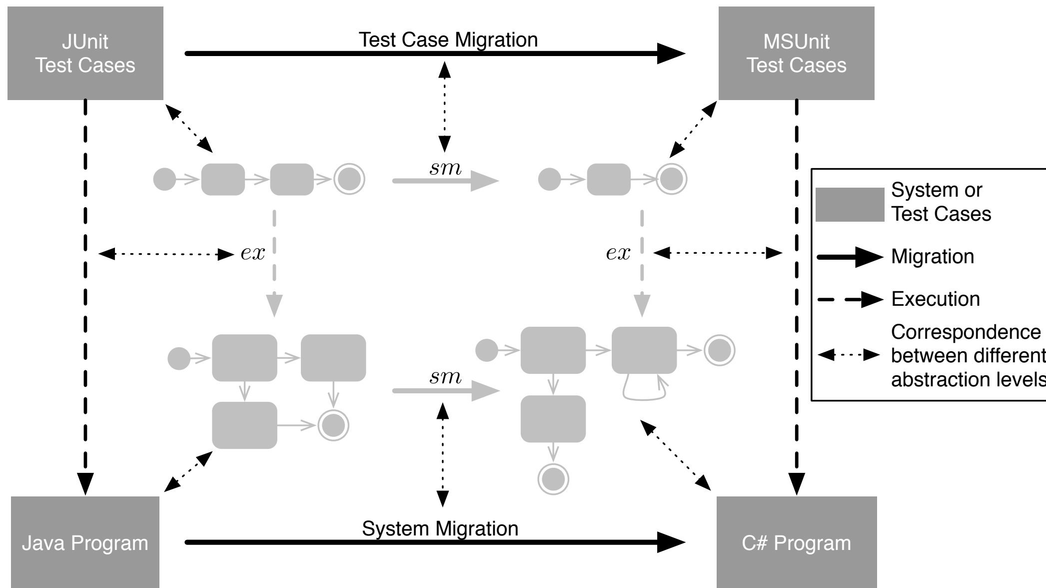


Figure 7.4 System and test case migration in the context of our running example

Definition 18 (Test Case Migration)**.** *Test case migration is a (manually performed or automated) transformation that takes a set of original test cases as input and produces a set of migrated test cases as output.*

Definition 19 (Correctness of Test Case Migration)**.** *Let assume that sm is a correct system migration. A test case migration tm is correct if the original system S passes the original test suite TS and the migrated system sm(S) passes the migrated test suite tm(TS) or the original system S fails to pass the original test suite TS and the migrated system sm(S) also fails to pass the migrated test suite tm(TS).*

Remark (Correctness of Test Case Migration)**.** *(1) This definition is sufficient but not necessary. So, if this holds for a test case migration tm then it can be considered correct. However, there might be a correct test case migration tm for which this does not hold. For example, when system migration sm fixes erroneous systems such as when replacing code with a call to a standard library. (2) In practice, a software migration sm is never perfect. So, you have to be able to work with buggy software migrations in practice too. For these reasons, the definition is nice on a conceptual level but otherwise idle.*

This leads us actually to the idea of applying mutation analysis and bad smell indications.

Based on our definition of correctness for system and test case migration, we can now classify old and migrated test cases:

Definition 20 (Classification of Migrated Test Cases)**.**
Let a system migration and a corresponding test case migration be given for an original

system and original test cases, respectively.
A migrated test case is a true/false positive *if it fails and the system migration is incorrect/-correct.*
A migrated test case is a false/true negative *if it passes and the system migration is incorrec-t/correct.*

Having covered the basics of mutation analysis, we now present the identified mutation analysis scenarios for validating test case migration. We started by analyzing a simple constellation of test case and system migration performed together (Figure 7.3).

Typical mutation analysis scenarios are when the system under test is mutated. In our case, there are two such scenarios as both the old and migrated system can be mutated. These scenarios correspond to *Scenario 1* and *Scenario 2* in Figure 7.3, respectively. Mutation analysis can, however, also be used to mutate test cases. As we have original test cases and migrated test cases, we have two such scenarios (*Scenario 5* and *Scenario 6* in Figure 7.3, respectively). The last two scenarios can be viewed as variants of *Scenario 2* and *Scenario 6*. By mutating the system migration or the test case migration, one indirectly mutates the result of the migration transformation, which is either the migrated system or the migrated test cases. Hence *Scenario 3* is comparable to *Scenario 2*, and *Scenario 4* to *Scenario 6*. Before each usage scenario is explained in-depth, we first introduce the following general assumptions for all usage scenarios:

(A1) All original test cases as well as migrated test cases can be executed and pass for the old and migrated system, respectively.

(A2) The Competent Programmer Hypothesis [ABD+79, LS78] is applied.

(A3) For simplicity and clarity, the problem of equivalent mutants is ignored for generated mutants.

These general assumptions are crucial for applying mutation analysis in the context of validation of test case migration. Regarding Assumption *(A1)*, we assume for simplicity that all test cases are executable. Although this might not be the case for legacy migration, handling the case where original test cases are *not* executable deserves a separate discussion and goes beyond the scope of work.

Analogously to the general case of mutation analysis [ABD+79, LS78], we assume with *(A2)* that the programmers (in this case, the migration specialists) are competent and that they tend to implement migration transformations that are already close to the correct migration transformations. The last assumption, Assumption *(A3)* concerns the well-known Equivalent Mutant problem [OP97, ABD+79, JH11] from mutation analysis. This problem

appears when a generated mutant is syntactically different but behaviorally equivalent to the original program. As shown in [OP97, ABD+79], it is impossible to automatically detect all equivalent mutants as program equivalence is undecidable [BA82] in general. There are, however, some approaches that deal with this problem quite successfully [AHH04, BS79, GSZ09]. To simplify our analysis, we ignore the problem of equivalent mutants for generated mutants, i.e., we consider generated mutants to be always non-equivalent.

In the following, we present all six usage mutation scenarios. Each scenario is presented following the same structure: a short description providing general information about the scenario, and then two parts: *Assumptions* presenting additional scenario-specific assumptions, and *Indications* discussing the results of applying mutation analysis for the specific scenario.

Scenario 1: Mutation of Original System

In the first scenario, the mutated object is the original system, i.e., the system before migration. As depicted in Figure 7.5, generated original system mutants can be either tested against the original test cases (sub-scenario that follows the sequence **1a, 2a**) or transformed with the system migration to produce migrated system mutants (sub-scenario that follows the sequence **1b, 2b, 3b**).

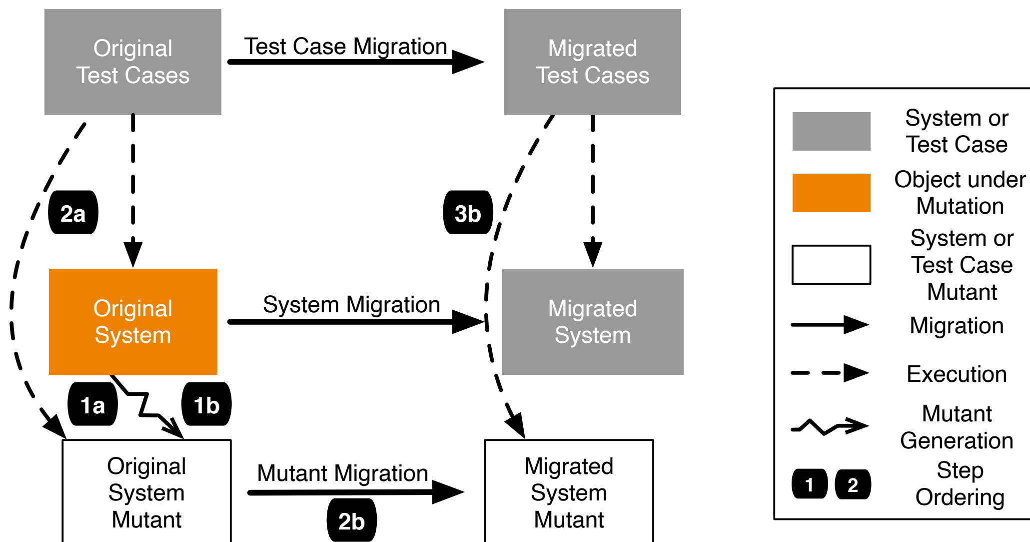


Figure 7.5 Mutation of Original System

Sub-Scenario 1a-2a: Mutation of Original System

The first sub-scenario (1a-2a in Figure 7.5), is a standard mutation analysis scenario that can be used to evaluate the quality of the original test cases.

Assumptions

(A1a.1) A suitable mutation framework for the original system exists.

Indications

The outcome of this analysis indicates how good the original test cases are. The higher the mutation coverage, the better the quality of the original test cases. For low mutation coverage, it makes sense to invest initial effort in improving the original test suite by creating additional test cases before starting with the task of validating the test case migration.

This scenario imposes just a single additional assumption, Assumption *A1a.1*, namely the existence of a suitable mutation framework for the original system. In our running example, for instance, the original system is a Java program for which there exist plenty of mutation techniques and frameworks [PIT, MOK05, KCM01, KKCM00, JJCM99, KCM99, CT03, Che01b].

Suitability. The first sub-scenario, namely the one following the steps 1a and 2a in Figure 7.5), is a standard mutation analysis scenario that can be used to evaluate the quality of the original test cases. It can be used as a quality evaluation strategy to improve the fault-revealing capability of the original test cases.

Sub-Scenario 1b-2b-3b: Mutation of Original System with Migration

The second sub-scenario (1b-2b-3b in Figure 7.5) analyzes original system mutants from a system migration perspective. The generated original system mutants are migrated using the system migration and then the obtained migrated system mutants are tested against the migrated test cases.

Assumptions

(A1.1) A suitable mutation framework for the original system exists.

(A1.2) System migration is (semi-)automated.

(A1.3) Original system mutants are killed by original test cases (if not, see 1a).

(A1.4) Original system mutants can be migrated.

Indications

```
if migrated system mutant is killed then
    if migrated system mutant is equivalent then
        At least one migrated test case is a false positive
    else
        Expected case
    end
else
    if migrated system mutant is equivalent then
        No indication
    else
        At least one migrated test case is a false negative
    end
end
```

The general idea in this scenario is to check for inconsistencies in the test or system migration by migrating the generated mutant of the original system. If the migrated system mutant is killed by the migrated test cases, but it is equivalent w.r.t. the migrated system, then at least one migrated test case is a false positive. Due to Assumption *(A1)*, this is, however, unlikely as the migrated test cases are assumed to accept a correct migrated system (an equivalent migrated system mutant would be such a correct migrated system). Note that even with Assumption *(A3)*, we still have to discuss equivalent mutants as these can be produced by the migration from non-equivalent generated mutants. For example, when the system migration replaces functionality with an off-the-shelf library, mutating these parts in the original system would lead to equivalent and correct migrated system mutants. The expected case, increasing trust in the test case migration, is when a non-equivalent migrated mutant is killed by the migrated test cases, as the mutant's behavior differs from the desired behavior.

The other half of the analysis in this scenario deals with the case when the migrated system mutant is not killed. When the migrated system mutant is equivalent this gives no indications, and it neither reduces nor increases trust in both migrations. When the migrated system mutant is non-equivalent, however, then it means that at least one test case is a false negative. A trivial example for this would be when all migrated test cases are erroneously set to always result in `true`, irrespective of the migrated system under test.

Suitability. The second sub-scenario (the one following the steps 1b, 2b, and 3b in Figure 7.5) focuses on the mutation of the original system and migrating the mutants in the target environment. The obtained migrated system mutants are tested against the migrated test cases. This scenario is suitable when the creation of the original system is easier compared to the mutation of the migrated system.

Scenario 2: Mutation of Migrated System

In this scenario, the mutated object is the migrated system. For this scenario, we have the following assumptions:

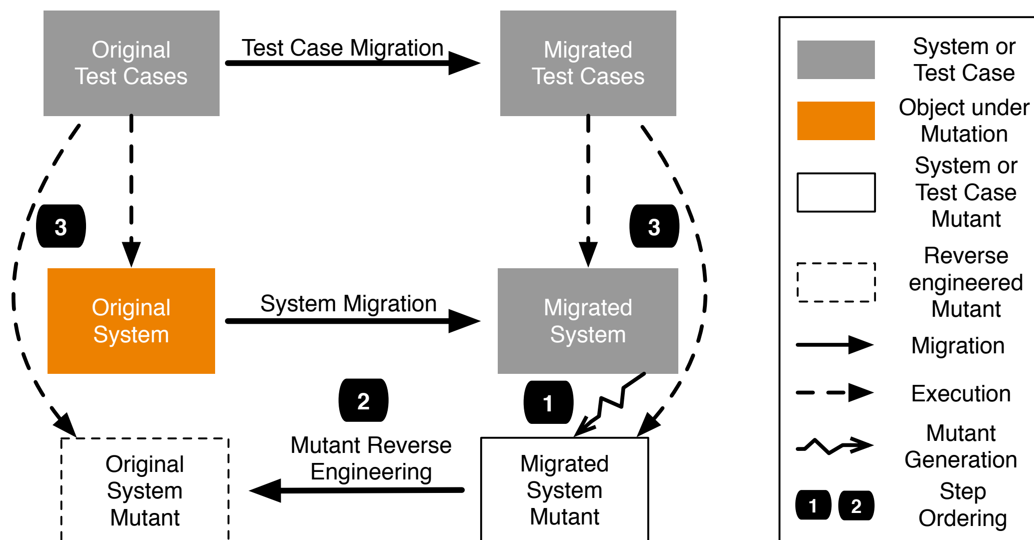


Figure 7.6 Mutation of Migrated System

Assumptions

(A2.1) System migration is (semi-)automated.

(A2.2) A suitable mutation framework for the migrated system exists.

(A2.3) It is possible to derive original system mutants from migrated system mutants via reverse engineering.

Indications

In this scenario, we start first with the case when the migrated system mutant is killed. A corresponding original system mutant is obtained via reverse engineering. If the obtained system mutant is killed as well, then this represents the expected case that increases the trust in the migration. If the obtained original system mutant was not killed, however, then it is either equivalent (no indications) or non-equivalent meaning that we have an erroneous original system that is not detected by the original test cases (see *Scenario 1a*).

The other half of the analysis deals with the case when the migrated system mutant is not killed. If the reverse engineered original system mutant is killed, however, this suggests that

```
if migrated system mutant is killed then
    if original system mutant is killed then
        Expected case
    else
        if original system mutant is equivalent then
            No indication
        else
            Scenario 1a should be revisited
        end
    end
else
    if original system mutant is killed then
        At least one migrated test case is a false negative
    else
        if original system mutant is equivalent then
            No indication
        else
            Scenario 1a should be revisited
        end
    end
end
```

at least one migrated test case is a false negative. If the reverse engineered original system mutant is not killed and it is equivalent, then no indications can be derived. If the reverse engineered original system mutant is not equivalent, then *Scenario 1a* should be revisited as an erroneous original system obtained via reverse engineering is not detected as such.

An important requirement that can be derived from Assumption *(A2.3)* concerns the choice of transformation language for the system migration. To simplify the reengineering of original system mutants, for example, a suitable bidirectional transformation language [CFH^{+}09] could be used.

Suitability. In general, this scenario is suitable when the mutation of the migrated system is easier compared to the mutation of the original system. According to Assumption *(A2.1)*, a suitable mutation framework for the migrated system is required. In our running example, this is a C# program for which there exist numerous techniques and frameworks [DS07, DS08, Der06].

Scenario 3: Mutation of System Migration

In this scenario, the mutated object is the system migration. This scenario can be seen as an indirect mutation of the migrated system, i.e., another way to apply *Scenario 2*.

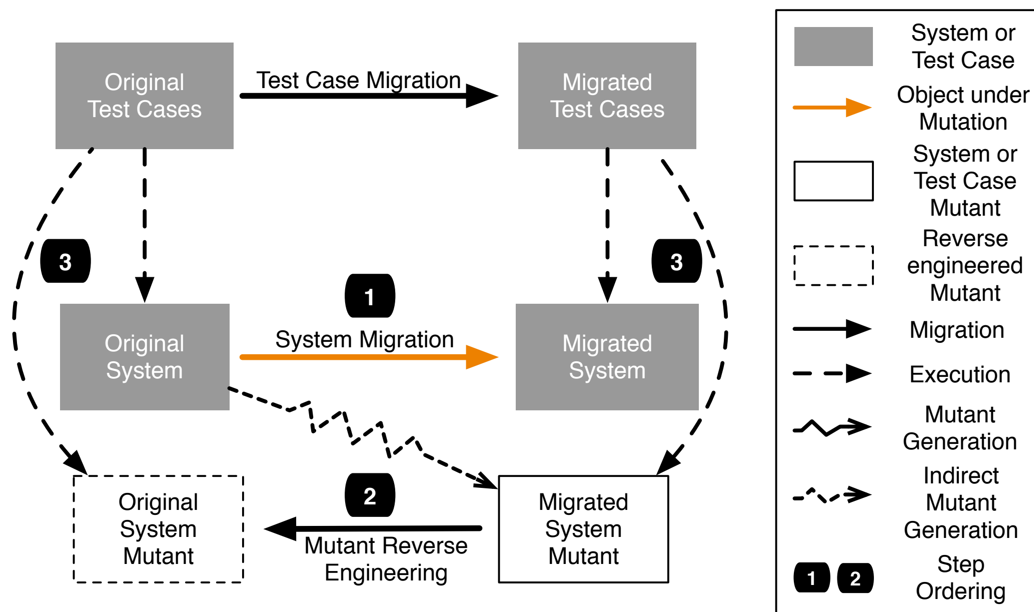


Figure 7.7 Mutation of System Migration

Assumptions

(A3.1) A suitable mutation framework for the system migration exists.

(A3.2) System migration is (semi-)automated.

(A3.3) Backward transformation is possible for migrated system mutants.

Indications

if *migrated system mutant is not equivalent* **then**
| Apply Scenario 2
else
| Apply Scenario 1b
end

The challenge in this scenario is to decide whether the system mutant is non-equivalent one by means of reverse transformation or with an explicit inspection. Otherwise, it does not make sense to analyze further a system equivalent.

Suitability. The advantage in applying this scenario and not directly *Scenario 2*, is that it might be easier to mutate the system migration depending on the context and the mutation framework that is used. The obtained migrated mutants via mutation of the system migration, according to *Scenario 2*. Assumption *A3.1* requires the existence of a suitable mutation framework of the system mutation. Due to the limitation of the classical mutation operators, Mottu et al. [MBLT06] propose a set of specific mutation operators for applying mutation analysis to model transformations.

Scenario 4: Mutation of Original Test Cases

In this scenario, the mutated objects are the original test cases, i.e., the test cases before migration. If the original test case mutants are executed against the original system, this scenario can be seen as an augmentation of the original test cases with negative test cases [SJ16]. Furthermore, the original test case mutants can be migrated and executed against the migrated system. In that case, they check the correctness of the migrated system and the actual indications are presented in the following.

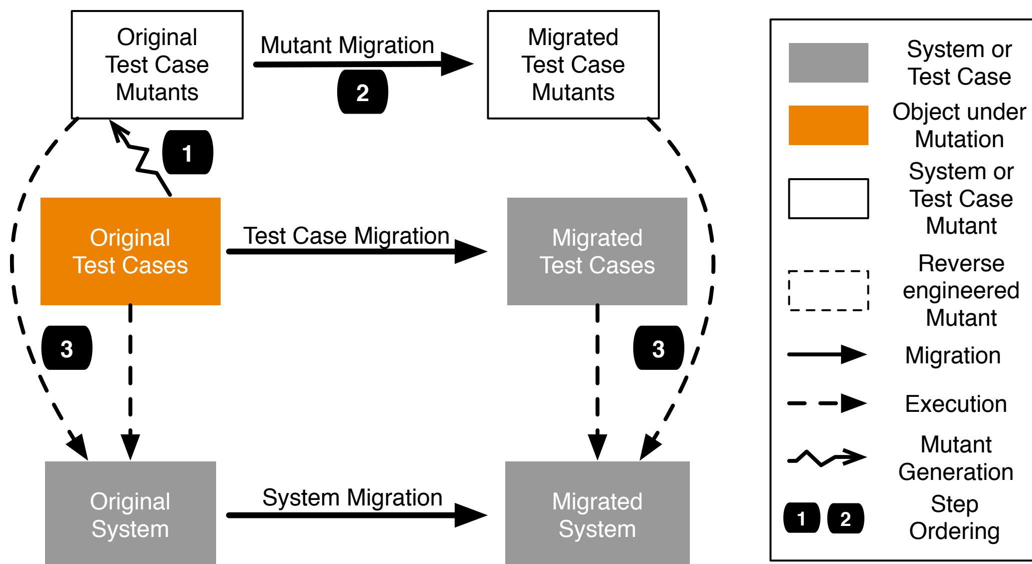


Figure 7.8 Mutation of Original Test Cases

Assumptions

(A4.1) A suitable mutation framework for original test cases exists.

Indications

```
if original test case mutant is killed then
    if original test case mutant is equivalent then
        Impossible (would contradict Assumption A1)
    else
        if migrated test case mutant is killed then
            Expected case
        else
            Migrated test case mutant is a false negative
        end
    end
else
    if original test case mutant is equivalent then
        if migrated test case mutant is not killed then
            Expected case
        else
            At least one migrated test case is a false positive
        end
    else
        Scenario 1a should be revisited
    end
end
```

First, we start with the case where the original test case mutant is killed. If the original test case mutant is equivalent, then it will contradict with *Assumption A1*. If the original test case mutant is killed, then we also expect the migrated test case mutant to get killed. Otherwise, this suggests that the migrated test case is a false negative.

The other half of the analysis deals with the case where the original test case mutant is not killed. If the original test case mutant is equivalent, then we also expect the migrated test case mutant to be equivalent as well giving out no indication. Otherwise, this suggests that at least one migrated test case is a false positive. However, if the original test case mutant is not killed and not equivalent, then it suggests that *Scenario 1a* should be revisited.

Suitability. This scenario is suitable for improving the overall quality of the original test cases prior to the migration if an indication for low quality exists. Furthermore, if the mutated original test cases are migrated, then, they can be used to check the migrated system. This strategy can be suitable if the mutation of the original tests is easier than the mutation of the migrated tests.

Scenario 5: Mutation of Migrated Test Cases

In this scenario, the mutated objects are the migrated test cases.

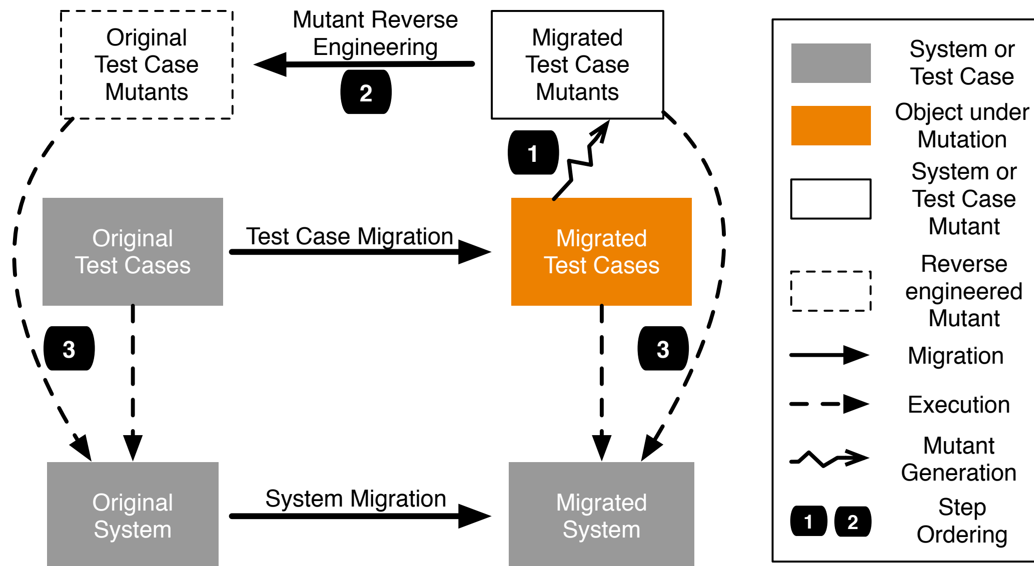


Figure 7.9 Mutation of Migrated Test Cases

Assumptions

(A5.1) A suitable mutation framework for the migrated test cases exists.

(A5.2) Backward transformation of migrated test case mutants is possible.

Indications

if *original test case mutant fails* **then**
  Expected case
else
  if *original test case mutant is equivalent* **then**
    No indication
  else
    Bad smell for test case migration
  end
end

This scenario requires that the backward transformation of the migrated test case mutant is possible. If the original test case mutant fails, then that is something that is expected. If it does not fail, then if the original test case is an equivalent test case mutant, no indication could be seen. This could happen when an API library is used in the migrated system and in the case of the original system it was manually implemented. However, if the original test case mutant is not equivalent, then it suggests that it is a false positive as it validates a correct system, which in turn means that the test case migration may be buggy and should be checked.

Suitability. This scenario is suitable when the mutation of the migrated test cases is easier compared to the mutation of the original test cases.

Scenario 6: Mutation for Test Case Migration

In this last scenario, the mutated object is the test case migration. This scenario is basically an indirect mutation of the migrated test cases.

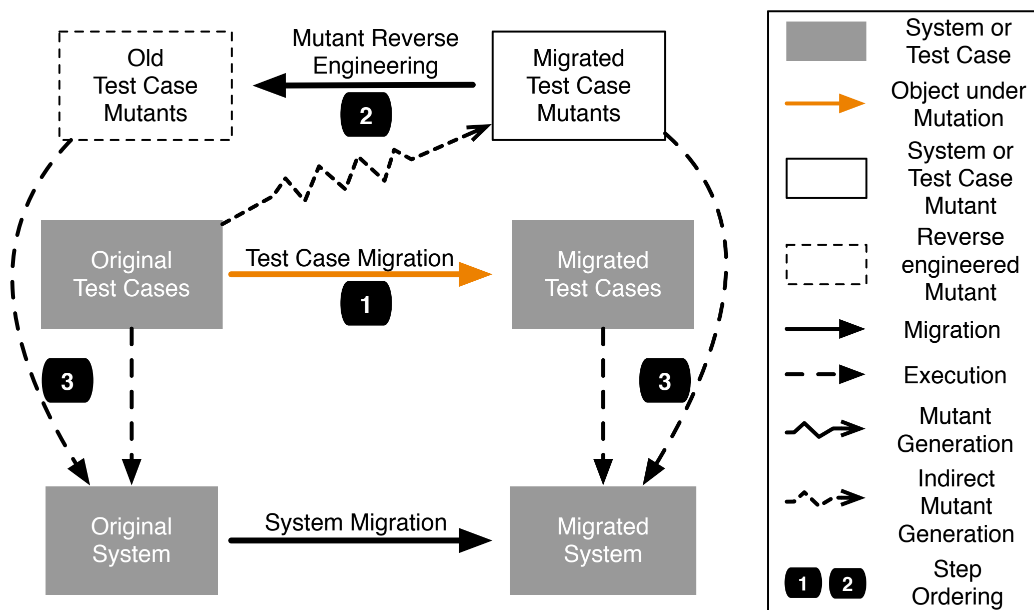


Figure 7.10 Mutation for Test Case Migration

Assumptions

(A6.1) A suitable mutation framework exists for the test case migration.

(A6.2) Test case migration is (semi-)automated.

(A6.3) Backward transformation of migrated test case mutants is possible.

Indications

```
if migrated test case mutant passes for migrated system then
    if migrated test case mutant is equivalent then
        No indication
    else
        Bad smell for system migration
    end
else
    if migrated test case mutant is equivalent then
        Impossible (would contradict Assumption A1)
    else
        Apply Scenario 5
    end
end
```

If the migrated test case mutant passes, then it is either equivalent (no indication can be obtained) or non-equivalent (indicating a bad smell for system migration as the migrated system exhibits non-expected behavior). The second half of the analysis is the case when the migrated test case mutant fails. Due to Assumption *(A1)*, the migrated test case mutant must be non-equivalent, meaning that *Scenario 5* can be applied.

Suitability. In general, this scenario is suitable when the mutation of the test case migration is easier compared to the mutation of the migrated test cases. Therefore, it can be seen as an alternative to *Scenario 5*. Instead of directly mutating the migrated test cases, the test case migration is mutated, indirectly producing migrated test case mutants.

7.2.2 Mutation Operators

A mutation operator, also known as a mutagenic operator or mutation rule, is a transformation that creates a mutant from an existing program or existing test case. In our case, as shown in Figure 7.11, we distinguish between three different types of mutants, *Language Mutation Operator*, *Test Mutation Operator*, and *Domain-specific Operator*.

Language Mutation Operators

Numerous mutation operators have been defined for languages such as Java [MOK05, PIT, KCM99] or C# [Vis, Str]. The process of designing mutation operators is usually based on

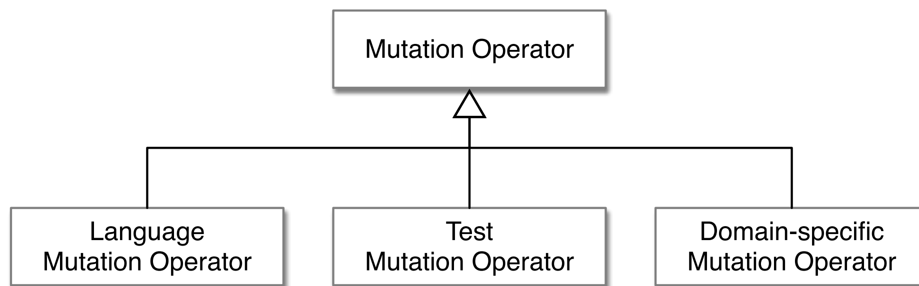


Figure 7.11 Overview of the mutation operator types

examining elements of the languages or motivated from other mutation operators designed for similar languages. By combining the existing language mutation operators, we defined our set of mutation operators as shown in Figure 7.12. As can be seen, we distinguish between two main types of language mutation operators according to the level of mutation namely *Method Level* and *Class Level* mutation operators.

Level of Mutation	Mutation Operator
Method Level	Conditionals Boundary Mutator
	Negate Conditionals Mutator
	Remove Conditionals Mutator
	Math Mutator
	String Mutator
	Return Values Mutator
	Empty Returns Mutator
	Null Returns Mutator
	Empty Mutation Operator
	Content Comparison Operator
	Statement Deletion Operator
	Exception Throw Mutator
	Method Mutator

Level of Mutation	Mutation Operator
Class Level	Access Modifier Change
	Hiding Variable Deletion
	Overriding Method Deletion
	Overriding Method Rename
	Parameter Change with Child Class Type
	Reference Assignment Mutator
	Type Cast Change Mutator

Figure 7.12 Overview of the language mutation operators

Test Mutation Operators

Compared to the language mutation operators, which are more general, the test mutation operators are more specific as their name already suggests, namely, they deal with the test constructs. As defined in the test horseshoe model, on the platform-specific and platform independent layer we have models that represent the test cases by using test-specific modeling

Level of Mutation	Mutation Operator
Test Case Level	Assertion Type Mutator
	Expected Result Mutator
	Actual Result Mutator
	Order of Execution Annotation Mutator
	Ignore Annotation Mutator
Test Class Level	Order of Execution Annotation Mutator
	Test Result Method Mutator
	Test Suite Method Mutator

Figure 7.13 Example set of test mutation operators for the JUnit framework

elements, e.g., assertion, expected result, or actual result. The role of a test mutation operator is to make atomic changes regarding these model elements, e.g., swap the expected result with the actual result or switch from one type of assertion to another. As shown in Figure 7.13, for the JUnit framework we distinguish between two different levels of mutation, namely *Test Case Level* and *Test Class Level*. On the *Test Case Level*, we have mutation operators like *Assertion Type Mutator* which would change the type of the assertion used in a given test case, e.g., `assertTrue` to `assertFalse`. Another example is the *Order of Execution Annotation Mutator* which switches `@After` annotation to `@Before` which means that a given the statement being affected will not be executed after each test case but before each test case. The same example applies to the *Order of Execution Annotation Mutator* defined on the *Test Class Level*.

Domain-specific Operators

The third type of mutation operators we have are the domain-specific operators. These types of operators as their name suggests are specific to the domain and should be accordingly defined. For example, in our migration scenario, we are dealing with the migration of OCL test cases. As we have seen in the migration phase, the OCL implementation had to be changed from *Just-in-Time* execution to *Ahead-of-Time* execution. This means, possibly in the OCL logic there could be the same unwanted changes that we want to detect. However, by none of the previous two types of mutation operators, this could be addressed. Hence, a mutation operator has to be defined for the OCL language. For example, *Collection Type Replacement* is a mutation operator that swaps the type of collection, e.g., from `BAG` to `SET`. Further domain-specific mutation operators for the OCL language are shown in Figure 7.14. This set of mutation operators is largely based on the OCL operators proposed by [Str16].

Mutation Operator
Collection Type Replacement
Collection Operation Replacement
Collection Operation Deletion
Collection Operation Insertion
Collection Element Replacement
Collection Element Deletion
Collection Element Insertion

Mutation Operator
Arithmetic Operator Replacement
Arithmetic Operator Deletion
Arithmetic Operator Insertion
Relational Operator Replacement
Conditional Operator Replacement
Conditional Operator Deletion
Conditional Operator Insertion

Figure 7.14 Example set of domain-specific mutation operators for the OCL Language

7.2.3 Mutation Method Patterns

Having only the different scenarios, on a rather conceptual level is not enough to perform the actual mutation analysis in practice. For this reason, we additionally provide a set of mutation method patterns, similar to the migration patterns introduced in Section 6.2.3, in order to support the mutation of test cases on different levels of abstraction.

Similarly to method patterns, a mutation method pattern represents construction guidelines for mutation methods and follows a certain strategy. In the following, as shown in Figure 7.15, we present the basic mutation method patterns that can be applied to mutate test cases.

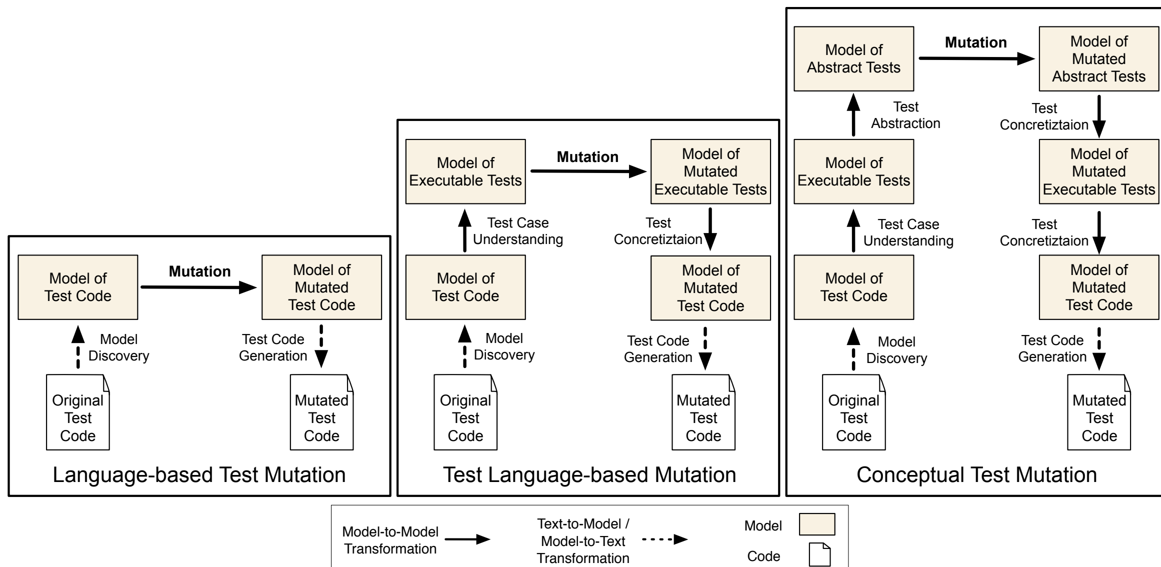


Figure 7.15 Overview of the mutation method patterns

Language-based Test Mutation

The *Language-based Test Transformation* pattern defines the mutation of test cases by defining a mutation of the language constructs contained by the *Model of Test Code*. So, as shown in Figure 7.15, the mutation is applied as a direct transformation of the *Model of the Test Code* and results in the *Model of Mutated Test Code*. This pattern could be applied actually in any migration context, but its suitability depends on the applied method pattern as well as the selected mutation operators that have to be applied. If the applied test method pattern is on a lower level of abstraction, in this case, it means a *Test Reimplementation* was applied, then, besides the *Mutation* activity, a *Model Discovery* activity has also to implemented and applied. In any other case, the *Model Discovery* would be already applied. The second aspect that defines the suitability of this pattern are the selected operators to be applied. If basic mutation operators are selected, e.g., swapping logical operators then, the implementation of the model transformation that supports the *Mutation* activity, then this will not result in a high effort. More particularly, the model transformations for the selected mutation operators will not be complex as they use directly the Abstract Syntax Tree elements. However, if test-specific mutation operators are selected to be applied, e.g., swapping expected value to actual value, then the complexity of the model transformation will be higher as these concepts are not existing explicitly.

Test Language-based Test Mutation

The *Test Language-based Test Transformation* pattern (Figure 7.15) defines the mutation of test cases by defining a mutation of the platform-specific test constructs contained by the *Model of Executable Tests*. In other words, the mutation is applied as a direct transformation of the *Model of Executable Tests* and results in the *Model of Mutated Executable Tests*. As for any mutation method pattern, the suitability of this mutation method pattern depends on the applied method pattern as well as the selected mutation operators that have to be applied. If the applied test method pattern is on a lower level of abstraction, in this case, it means wither *Test Reimplementation* or *Language-based Transformation* was applied, then, besides the *Mutation* activity, a *Model Discovery* as well as *Test Understanding* have also to be implemented and applied in order to obtain the *Model of Executable Tests*. In cases when *Test Language-based Test Transformation* or *Conceptual Test Transformation* is applied, *Model Discovery* and *Test Understanding* would be already applied. As this pattern enables direct, i.e., explicit representation and manipulation of test constructs, e.g., test assertion or expected result, it is suitable when test-specific mutation operators have to be applied, like swapping test assertion from `isTrue` to `isFalse`.

Conceptual Test Mutation

The *Conceptual Test Mutation* pattern (Figure 7.15) defines the mutation of test cases by defining a mutation of the platform-independent test constructs contained by the *Model of Executable Tests*. That means, the mutation is applied as a direct transformation of the *Model of Abstract Tests* and results in the *Model of Mutated Abstract Tests*. Regarding the first aspect that influences the suitability of this pattern, namely the applied test method pattern, this pattern is most suitable when *Conceptual Test Transformation* was applied and the *Model of Abstract Tests* is already given. Otherwise, all the reverse engineering activities leading to the *Model of Abstract Tests*, i.e., *Model Discovery*, *Test Understanding*, and *Test Abstraction* have also to implemented and applied. Similarly to the *Test Language-based Mutation*, this pattern enables direct, i.e., explicit representation and manipulation of test constructs, e.g., test assertion or expected result but rather on the platform-independent level. For example, from the test design perspective, the test architecture or test behavior could be explicitly represented with this model.

7.3 Test Case Migration Validation Process

In this section, we give an overview of the post-migration phase of TeCoMi where a context-specific validation method gets developed and enacted. As shown in Figure 7.1, we support this phase by providing an extended process which relies on the previously introduced mutation analysis-based validation approach. Firstly, a mutation method suitable for the context has to be developed. This means that firstly context information has to be analyzed. Based on this, a suitable mutation analysis scenario (Section 7.2.1) or multiple suitable scenarios are selected. Thereafter, depending on the functionality being asserted in the test cases, a set of suitable mutation operators (Section 7.2.2) is selected. Based on the selected mutation operators and selected mutation analysis scenarios, a suitable mutation method pattern (Section 7.2.3) is selected. Then, the selected set of mutation operators has to be implemented in terms of model transformations. Once the validation method is created, it can be executed and the mutated test cases will be created. The mutated test cases are then executed against the migrated system. The outcome of the test case execution of the mutated test cases is then analyzed. If no false positives and false negatives are identified, the migration of the test cases is considered to be successful. Otherwise, the identified false positives and false negatives have to be analyzed in order to fix them.

7.3.1 Context Characterization

The first activity of the validation process is *Context Characterization*, in which the migration context is analyzed and characterized, from both system migration and testing perspective. However, as this activity was performed already in the previous phase, the obtained *Situational Context Model* can be reused for the development of a suitable test case validation method. Moreover, the *Situation-Specific Test Migration Method Specification* as well as the *Situation-Specific Tool Chain* contain also important information about the context of the test migration. In the following, we discuss how each of the above-mentioned artifacts is utilized by this activity.

Firstly, we analyze the *Situational Context Model* which, as presented in Section 6.3.1, comprises *Concept Model*, *Impact Model*, and *Influence Factor Model*. The *Concept Model* represents the functionality of the system and the test cases in terms of concepts. These concepts are analyzed by a person in the role *Test Expert*, who can clearly see which concepts could be potentially selected for mutation. The *Impact Model* additionally provides information on the influence the system concepts have on the test concepts which can also point out at what place the mutation should take place. This comes from the fact that the relations provided by the *Impact Model* show exactly what is the relation between the system concepts being changed and the test concepts. Figure 7.16 shows the same situational context model we used in Section 6.3.1. Based on this example, the *Test Expert* can identify and select the artifacts for mutation. In this concrete example, the source test concept *OCL Test Case (JIT)* depends on the *Native OCL-Expression*. This has changed in the target environment, and the target test concept *OCL Test Case (AOT)* depends on the *Language-Specific OCL-Expression*. So, this suggests where actual the focus of the mutation should be, i.e., the part of the test cases that has changed, and that is the part containing the OCL expressions. This further suggests that *Domain-Specific Mutation Operators* are needed for the mutation of the OCL expressions.

Then, the *Situation-Specific Test Migration Method Specification* provides information on the applied test transformation patterns, which is important for the next activity when the actual mutation method gets constructed. The information about the different test transformation patterns selected for the test cases influences the selection of the migration scenario and the mutation method pattern. Furthermore, the *Situation-Specific Tool Chain* also influences the selection of the migration scenario and the mutation method pattern from the implementation perspective. Related to the previously introduced situational context model, the test transformation method shown in Figure 7.17 was performed. It also shows the tool specification, i.e., the tools used for the method enactment. The test method pattern

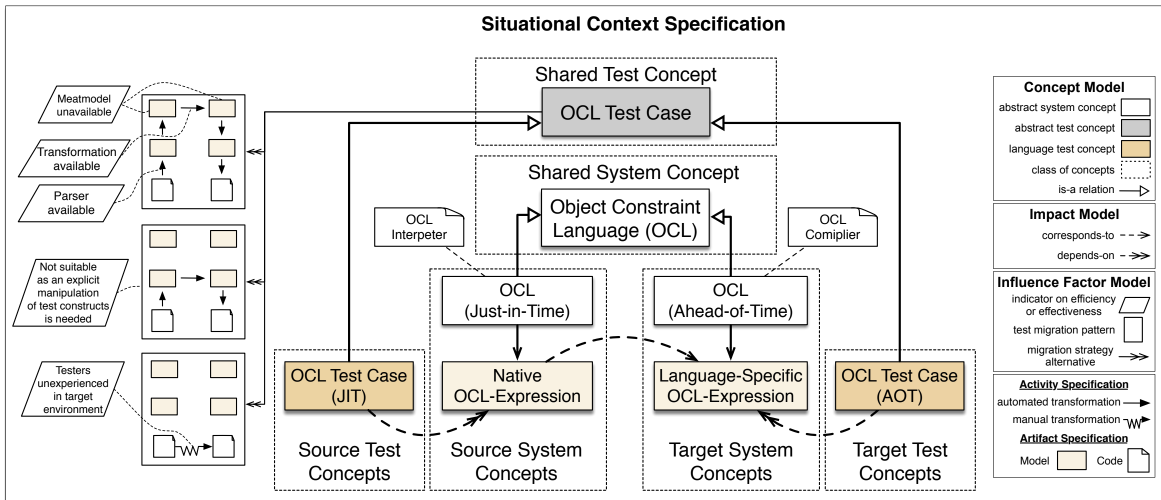


Figure 7.16 An example of a situational context model

used here, namely the *Test Language-based Transformation Pattern* suggests using the *Test Language-based Mutation Pattern* as most of the developed tools can be reused.

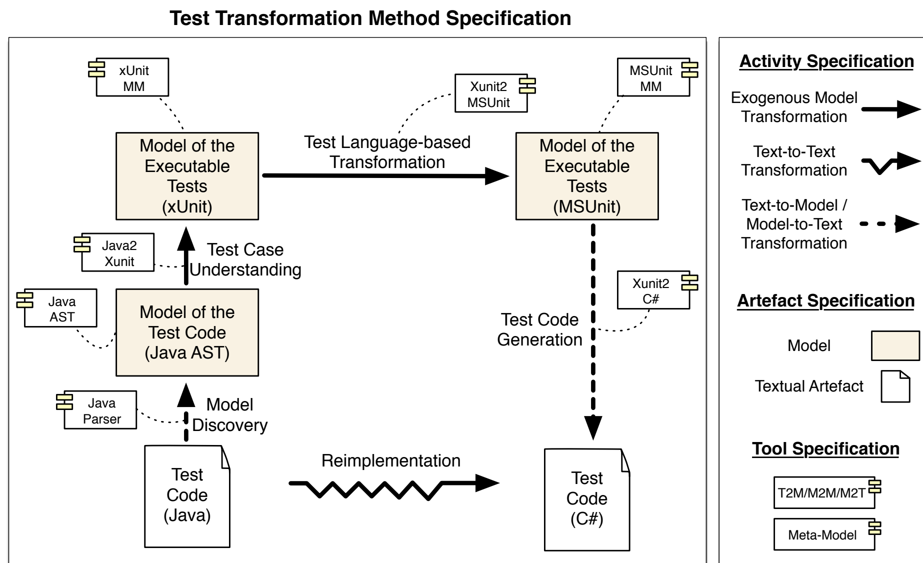


Figure 7.17 An example of a test transformation method specification

7.3.2 Mutation Method Construction

The purpose of this activity is the construction of a suitable mutation method. As shown in Figure 7.18, the process comprises three activities, namely *Selection of Mutation Analysis Scenario*, *Selection of Mutation Operators*, and *Selection and Configuration of Mutation*

Method Pattern. Before proceeding with the discussion on each of these activities, in the following, we refer to the three artifacts from the migration phase as *context information* (the artifact at the bottom in Figure 7.18).

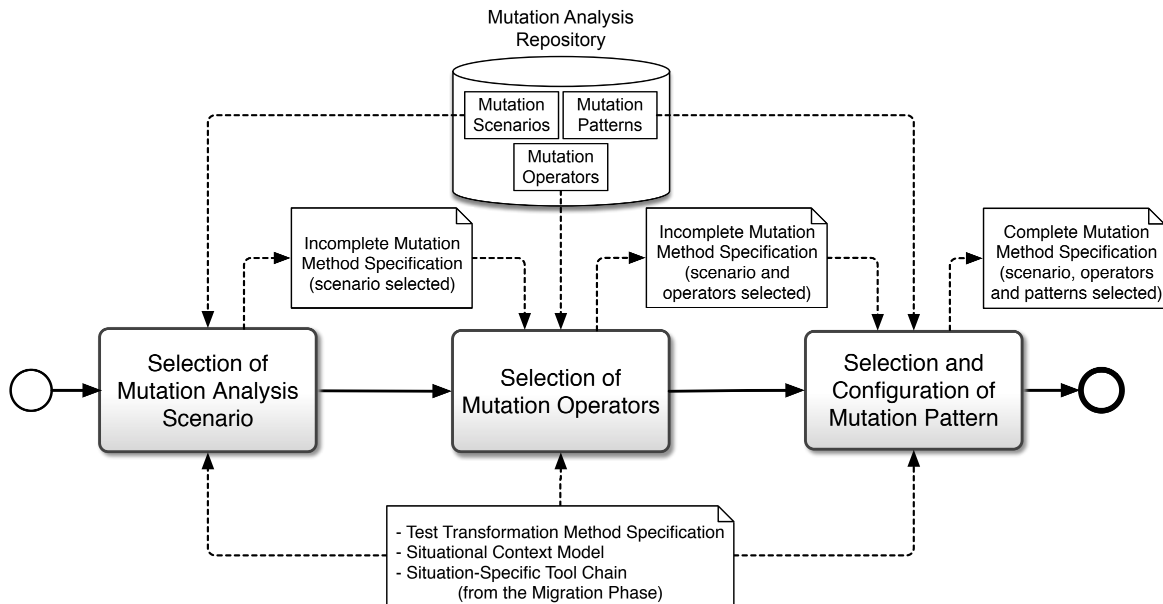


Figure 7.18 Mutation method construction process

Firstly, as part of the *Selection of Mutation Analysis Scenario*, a suitable mutation scenario is being selected from the *Mutation Analysis Repository*. As we already described in Section 7.2.1, each mutation scenario contains a discussion on its suitability. So, upon these characteristics and the *context information*, a suitable scenario is being selected.

Related to the example we introduced in the previous section (Figure 7.16 and Figure 7.17), the *Mutation of Original Test Cases* scenario was selected, as it is easier to mutate native OCL expressions than language-specific expressions defined in the target environment. The mutation of the system code was not considered as the migration was performed manually.

The outcome of this step is an *Incomplete Mutation Method Specification* as it contains only the selected mutation analysis scenario.

Secondly, based on the *context information*, and the previously selected mutation analysis scenario, as part of the *Selection of Mutation Operators*, a set of mutation operators is selected from the *Mutation Analysis Repository*. As already discussed in Section 7.2.2, three types of changes are provided in the repository, namely language mutation operators, test mutation operators, and domain-specific operators. Depending on the *context information*, more precisely on the test concepts being changed as well as the corresponding applied transformation pattern, mutation operators are being selected For example, if a test language-

based transformation was applied, where a test model elements are being transformed from source testing framework to a target testing framework, then applying test mutation operator are reasonable choice as they address exactly the test elements. Also, related to the running example (Figure 7.16 and Figure 7.17), applying domain-specific mutation operators is reasonable as the test cases contain OCL expressions that have been transformed. The outcome of this step is still an *Incomplete Mutation Method Specification* as it still misses a mutation method pattern which is selected in the next step.

Finally, as part of the last activity, namely *Selection of Mutation Method Pattern*, a mutation method pattern is being selected for the selected mutation analysis scenario and the selected mutation operators. As already discussed in Section 7.2.3, three mutation method patterns are defined in the repository, a language-based mutation, a test language-based mutation, and a conceptual mutation.

Depending on the *context information*, and on the selected mutation analysis scenarios and mutation operators, one of the three mutation patterns is selected. The *context information*, more specifically the applied test transformation method influences strongly the selection of the mutation method pattern. For example, if a language-based transformation was applied for the migration of the test cases, then a mutation on higher level of abstraction requires additional model transformations and metamodels on a higher level of abstraction. Furthermore, the selected mutation operators can be also seen as an influence factor when selecting a mutation method pattern. For example, if test mutation operators were selected, a test language-based mutation method pattern is more suitable than a language-based mutation method pattern as the test concepts like assertion or expected and an actual result that should be mutated are explicitly represented and thus easier to be implemented. Related to the running example (Figure 7.16 and Figure 7.17), the *Test Language-based Mutation Pattern* would be the pattern to implement as the *Test Language-based Test Transformation* was performed.

The configuration of the selected pattern is dependent on the applied test transformation method. Namely, the same method fragments are reused for the mutation method. For example, regarding the artifacts, the same *Model of the Test Code* or the same *Model of the Executable Tests* are used. The same applies to the activities, e.g., *Model Discovery* or *Test Case Understanding*. According to the example we have shown in Figure 7.17, the same fragments regarding reverse engineering and forward engineering activities and artifacts can be reused. The only difference is the transformation activity, that should be changed to a mutation activity.

The outcome of this step and at the same time of the overall *Mutation Method Construction* activity is a *Mutation Method Specification* as shown in Figure 7.19. On the left-hand side

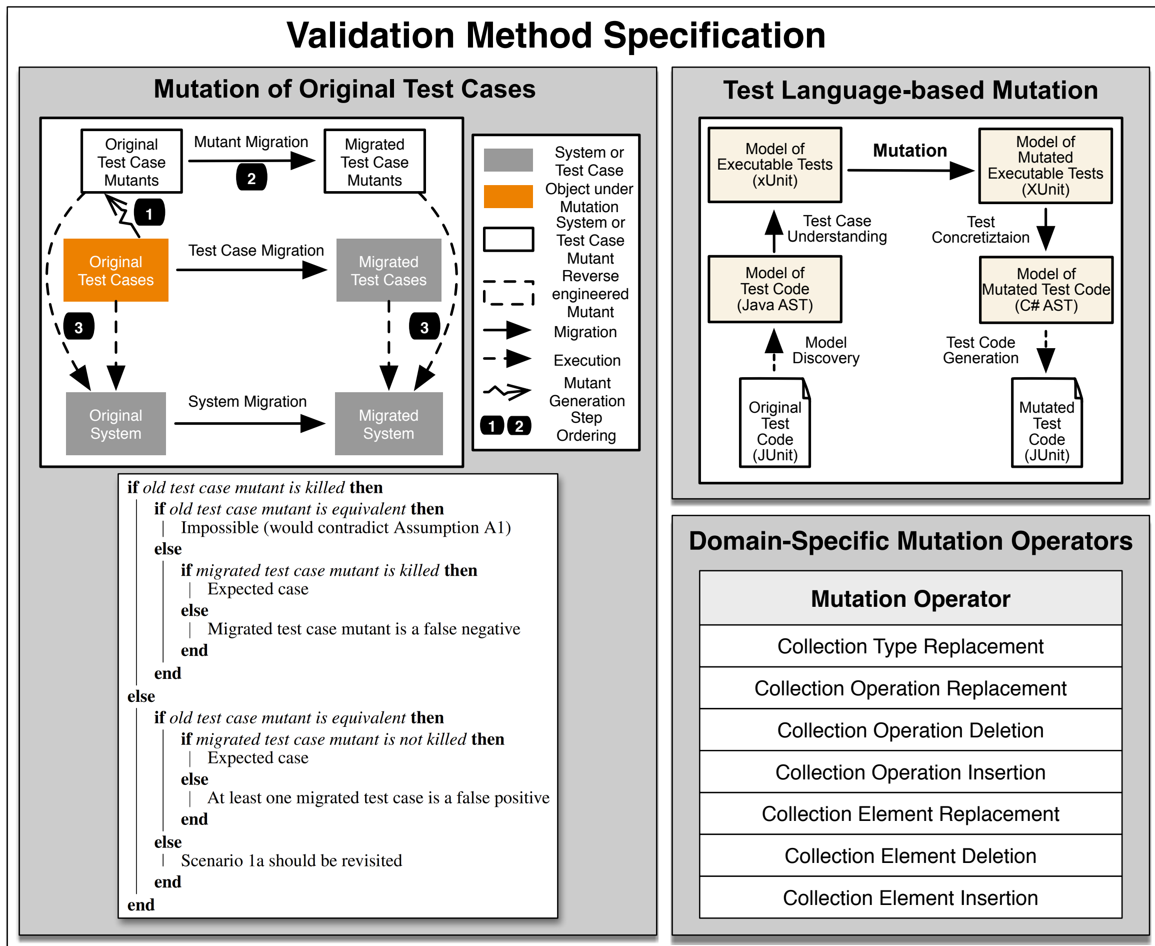


Figure 7.19 Mutation method specification

of the specification, the selected mutation analysis scenario is shown, namely the *Mutation of Original Test Cases* scenario. The graphical representation at the top describes which steps have to be performed and in what order. Below the graphical representation is the algorithm which provides knowledge on the interpretation of the execution results. This information is important for the last activity of the migration validation process, namely for the *Mutation, Test Execution, Analysis* activity. Furthermore, on the right-hand side of Figure 7.19, the selected test mutation operators are shown, in this case, the *Domain-Specific Mutation Operators*. Finally, the selected mutation method pattern, namely the *Test Language-based Mutation* pattern shows how the technical realization of the mutation scenario should be done, showing the artifacts and the activities to be used. The configuration of the mutation pattern was largely influenced by the configured test transformation method shown in Figure 7.17. The selected pattern as well as selected mutation operators are relevant to the tool developers in the next activity, namely the *Mutation Tool Implementation*.

7.3.3 Mutation Tool Implementation

Having the mutation method specification developed, we proceed to the development of the tools to automate the mutation. We assume that the specification is used by the associated tool developers as some kind of guidance. The developers can use the specification to get an overview of the validation method. By checking the mutation method pattern, they can get an understanding of how the actual mutation of the test cases should be performed and what needs to be developed. Currently, we assume this kind of more flexible kind of guidance.

To improve the process of automated mutation and to increase the reuse of existing components, we introduce a flexible and extensible model-driven mutation framework which should serve as a project-independent tool infrastructure. It is a component-based framework, consisting of three main components, *Input Module*, *Mutation Module*, and an *Output Module*, as shown in Figure 7.20. The *Input Module* deals with providing the proper data for the main component, the *Mutation Module*, which provides means to specify mutation operators and apply mutation on the input data. The *Output Module* deals with the generation of the mutated code. The component-based architecture guarantees an easy adaptation of the framework to any situation.

Mutation Tool Infrastructure

The solution architecture of the mutation framework is depicted in Figure 7.20 and shows the three main components of the framework, *Input Module*, *Mutation Module*, and *Output Module*.

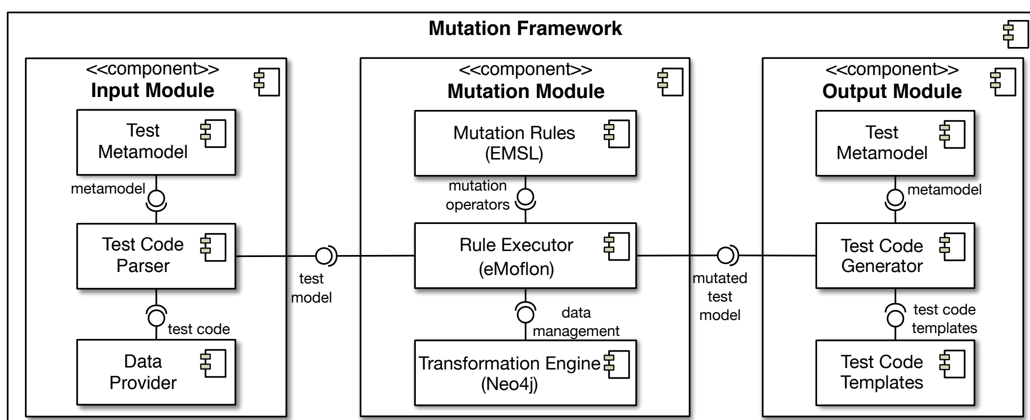


Figure 7.20 The architecture of the mutation framework

The leftmost component, namely the *Input Module*, as the name suggests, handles the input data, i.e., the test cases that have to be mutated. It consists of three components, namely

Test Metamodel, *Test Code Parser*, and *Data Provider*. The *Test Code Parser* requires the metamodel of the test cases that have to be parsed. The *Test Metamodel* component provides the metamodel to the parser, as shown in Figure 7.20. The test cases are provided by the *Data Provider* to the *Test Code Parser*. The *Input Module* component provides the parsed data in terms of a test model to the main component of the mutation framework, namely the *Mutation Module*.

The *Mutation Module* component is based on the *eMoflon* transformation framework, which is a tool suite for applying Model-Driven Engineering (MDE) and provides visual and formal languages for (meta)modeling and model management[1]. It consists of three components, *Mutation Rules*, *Rule Executor*, and *Transformation Engine*. The *Mutation Rules* component is responsible for the specification of the mutation operators in terms of mutation rules which are basically model transformation rules. The specified mutation rules can be executed by the *Rule Executor* and completely relies on *eMoflon*. This requires the mutation operators as well as the outcome of the *Test Code Parser* component, namely the test model. Having this input, the *Rule Executor* applies the rules and mutates the test model, i.e., the test cases which are part of the test model. Regarding the execution, the application of the mutation rules can be configured. By default, each mutation rule is applied to the first match in a test case. For efficient execution, the transformation rules are executed to the *Transformation Engine* which relies on *Neo4J*, which is a graph database that exchanges data via Cypher queries. Cypher[2] is Neo4j's graph query language that allows users to store and retrieve data from the graph database. Cypher's syntax provides a visual and logical way to match patterns of nodes and relationships in the graph. Once the mutation rules are executed, i.e., the mutation of the test model is performed, the mutated test model is used by the *Output Module*.

The *Output Module* component consists of three main components, *Test Metamodel*, *Test Code Generator*, and *Test Code Templates*. The central component is the *Test Code Generator* component, which is concerned with the generation of the executable mutated test cases. It requires the test metamodel as well as the test code templates so that the test code generation can be performed.

Figure 7.21 shows the general mutation process of the mutation framework. The horseshoe model as a common representation in the area of reengineering is used to visualize the model-driven approach. In general, the process consists of three main activities, *Test Model Discovery*, *Mutation*, and *Test Code Generation*. Besides the activities, we distinguish between two types of artifacts, models and textual artifacts. Lastly, we have also visualized

[1]https://emoflon.org/
[2]https://neo4j.com/developer/cypher-query-language/

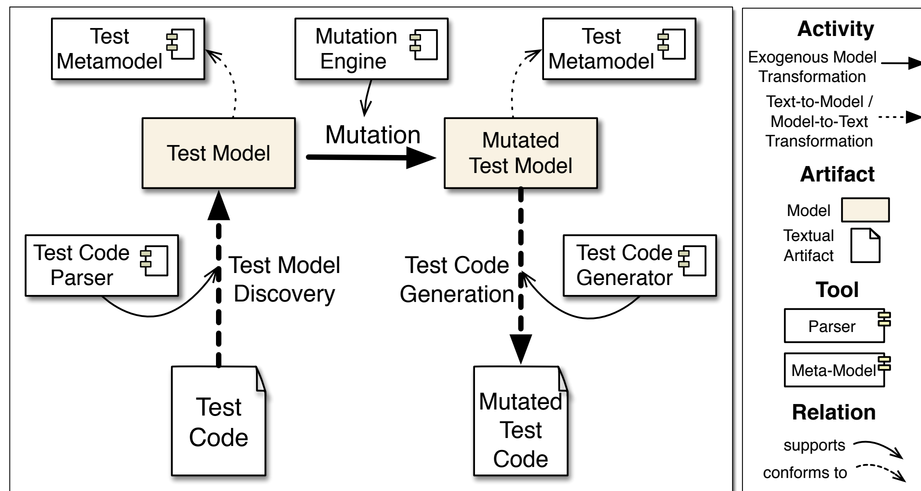


Figure 7.21 The general process of the mutation framework

the tools, i.e., the framework components introduced in the previous section, necessary for the automation of the whole process.

In the first activity, i.e., *Test Model Discovery* (shown on the left-hand side of Figure 7.21), *Test Model* out of the *Test Code* is extracted by using the previously introduced *Test Code Parser*. Having the outcome of this activity, i.e., the *Test Model* which conforms to *Test Metamodel*, the actual model mutation can be performed. For the *Mutation* activity, first of all, a set of relevant *Mutation Rules* has to be specified in terms of model transformation rules specified in *EMSL*[3], a uniform language for model management. Then, the *Mutation Module*, takes the *Mutation Rules* and executes them against the previously obtained *Test Model*. The outcome of the *Mutation* activity is a *Mutated Test Model*, which also conforms to the *Test Metamodel*.

In the third and the last activity of the mutation process, namely the *Test Code Generation* activity, based on the *Mutated Test Model*, and the *Test Code Templates*, the *Test Code Generator* generates executable mutated test cases, i.e., *Mutated Test Code*.

The mutation framework was implemented as a composition of three different *Eclipse* plugins which correspond to the three main components shown in Figure 7.20. Such an architecture enables flexibility and extensibility towards supporting mutation for different types of testing frameworks as well as mutation of different types of systems.

Firstly, the plugin corresponding to the *Input Module* component, was implemented to support the two main activities of the *Test Model Discovery* process, namely the parsing and the understanding activities. The parsing activity firstly parses the code and the test code

[3]https://emoflon.org/

and extracts an abstract syntax tree. In the case of *JUnit*[4] test cases, this means that firstly, a suitable Java parser is necessary (e.g., *JDT* Parser). Having the abstract syntax tree, a more concrete model can be obtained by applying model-to-model transformation as part of the understanding activity. When considering, *JUnit*, a suitable metamodel is necessary to enable this activity. As an outcome of the understanding activity, a test model is obtained in terms of a *JUnit* test model or if necessary, a more general representation is possible in terms of *xUnit* metamodel.

The main component, namely the *Mutation Module*, was realized also as an Eclipse plugin, which highly relies on the *eMoflon* framework. Among others, *eMoflon* has own language for specifying model transformation rules, namely the *EMSL* (*eMoflon* Specification Language). Consequently, the rules in the *Mutation Rules* component are specified in *EMSL*. *Rule Executor* further provides means to execute the specified rules. The management of the data is realized by the *Transformation Engine*, which is *Neo4J*[5], a graph database management system, which provides high scalability for model management tasks [WAF^{+}19]. The outcome of this plugin is the mutated test model, e.g., a *JUnit* test model.

The last Eclipse plugin realizes the *Output Module* component. The central component, i.e., the *Test Code Generator* is realized with the help of *Xtend*[6], a statically typed programming language being placed on top of Java known for developing code generators. The generator needs the *Test Code Templates*, which are defined as *Xtend* templates and are dependent on the target framework, i.e., the testing framework for which the mutated executable test cases have to be generated. Furthermore, the generator needs the *Test Metamodel* which in the concrete implementation is a *xUnit* metamodel. The outcome of the *Test Case Generation* activity and at the same time of the whole mutation process is the *Mutated Test Code*, which in the concrete example are *JUnit* test cases.

So, for the particular mutation method specification in Figure 7.19, we created the toolchain shown in Figure 7.22.

Namely, first of all, we developed the *Test Case Parser* and the *Test Metamodel*. In this particular case, we used the JDT Parser[7] to parse the test code and then we developed model transformations to obtain the test model conform to the *xUnit* metamodel. Having the test model, we had to develop meaningful mutation operators. As the original test cases were testing the OCL functionality, we had to define mutation operators that will change the semantics of the OCL expressions, contained in the test cases, either as the action to be performed or the expected result to be checked.

[4]https://junit.org/junit5/
[5]https://neo4j.com/
[6]https://www.eclipse.org/xtend/
[7]https://www.eclipse.org/jdt/

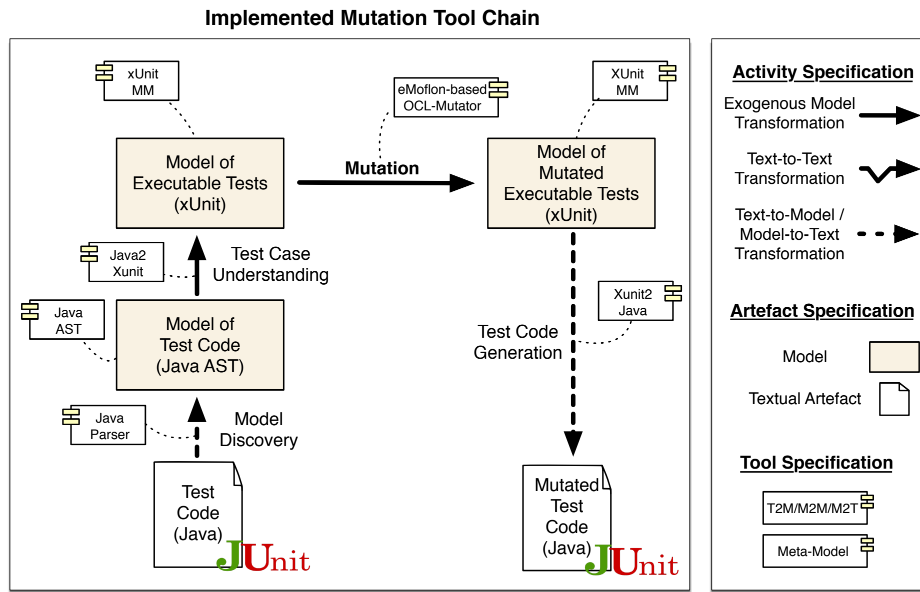
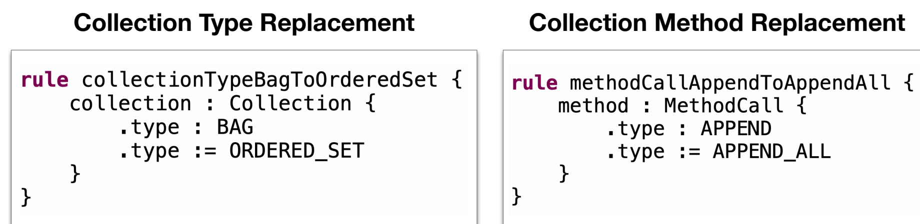


Figure 7.22 Implemented mutation tool chain

For this reason, an additional parser for the string-based OCL expressions was also developed. Having the OCL expressions interpreted, we then developed a set of OCL mutation operators in terms of model transformation rules. They were the practical implementation of the OCL-specific mutation operators shown in the bottom right corner in Figure 7.19. Figure 7.23 shows two mutation operators implemented in terms of transformation rules. The transformation rule on the left-hand side shows the implementation of the *Collection Type Replacement* mutation operator. As it can be seen, this rule, technically named as `collectionTypeBagToOrderedSet` rule, swaps a `Collection` of a type `BAG` to `ORDERED_SET`. The other transformation rule shows the implementation of the *Collection Method Replacement* mutation operator. Namely, this rule, technically named as `methodCallApendToApendAll` rule, swaps a `MethodCall` of a type `APPEND` to `APPEND_ALL`. Once the mutation operators were defined, they can be applied on the test cases and create their mutants.




```
rule collectionTypeBagToOrderedSet {
    collection : Collection {
        .type : BAG
        .type := ORDERED_SET
    }
}
```

```
rule methodCallAppendToAppendAll {
    method : MethodCall {
        .type : APPEND
        .type := APPEND_ALL
    }
}
```

Figure 7.23 OCL-specific mutation operators specified as transformation rules

7.3.4 Mutation Execution and Analysis

So far, the mutation method for the test cases has been developed. Also, the mutation tools which are required for the mutation have been developed. Here, we present the last activity of the *Migration Validation Phase*, namely the *Mutation Execution and Analysis* activity, that deals with the enactment of the mutation method and the analysis of the results as specified in Figure 7.19.

Mutation Execution

The purpose of *Mutation Execution* part is to perform the actual mutation of test cases as specified. We assume that the associated testers as well as test experts use the mutation method specification as some kind of guidance. The previously illustrated example of a mutation method specification in Figure 7.19, combined with the toolchain illustrated in Figure 7.22, shows how to perform the mutation of the test cases.

Figure 7.24 and Figure 7.25 show concrete executions of the two mutation operators we have previously defined in Figure 7.23. The upper part of the examples shows the original test case before the mutation, the middle part shows the mutation operators, i.e., the transformation rule, and the bottom part the mutated test case. The test case before the mutation step in the first example shown in Figure 7.24 tests the `append` functionality, i.e., it checks whether the execution of the `append('c')` function on the `Bag{'a','b'}` collection, results in `Bag{'a','b','c'}`, which is the expected result. The assert function `assertQueryResults()`, compares the values of the expected result and the particular OCL functionality which are specified as strings. When applied, the transformation rule looks for the first appearance of a collection of type `Bag` and swaps it to `OrderedSet`.

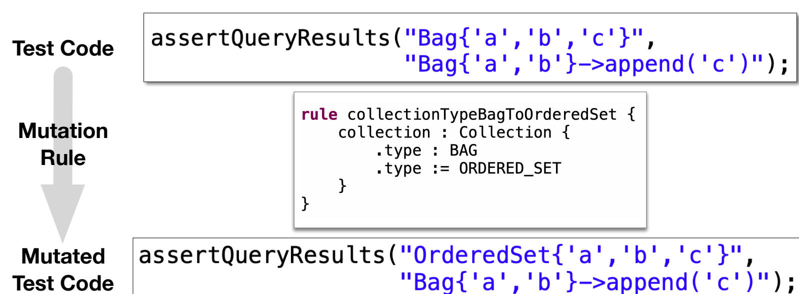


Figure 7.24 Execution of *Collection Type Replacement* mutation operator specified as a transformation rule

The test case before the mutation step in the second example Figure 7.25, is basically the same as the one in the previous example. However, the difference is that the transformation

rule looks for the first appearance of a method call of type `Append` and swaps it to `AppendAll`. It is important to mention, that we skip the activities and artifacts, dealing with reverse engineering and forward engineering parts (Figure 7.22), as we have explained this multiple times already, especially in the migration phase.

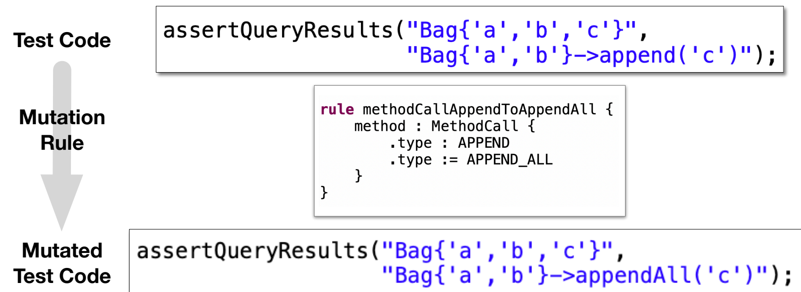


Figure 7.25 Execution of *Collection Method Replacement* mutation operator specified as a transformation rule

Analysis The purpose of *Analysis Execution* part is to analyze the test execution of the mutated test cases. After the mutation is performed and mutated test cases are executed, in the selected mutation analysis scenario there is an indication part which shows how to interpret the test execution results. Each mutation scenario defines when a problematic case can be identified by providing an algorithm defined in the pseudo code. The two examples we have previously introduced in Figure 7.24 and Figure 7.25, were executed against the original system, and as expected they failed. Then they were migrated and executed against the migrated system and also failed. So, according to *Scenario 4* defined in Section 7.2.1, this es the expected case, meaning no problems were discovered. In the following, we present an additional mutated test case which was identified as a bad smell according to *Scenario 4*.

The problem occurred when a collection entity such as `Bag`, `Set`, `OrderedSet`, and `Sequence` is converted to another type. In Figure 7.26, a `Sequence` is converted to `Bag` where we created a mutant converting a `Sequence` to `Sequence`.

```
@Test
public void testCollectionAsBag() {
  //Original
  //ocl.assertQueryResults("Bag{1,2.0,'3'}",
                         "Sequence{1,2.0,'3'}->asBag()");
  //Mutant - Applied mutation operator: Method mutator
  ocl.assertQueryResults("Bag{1,2.0,'3'}",
                        "Sequence{1,2.0,'3'}->asSequence()");
}
```

Figure 7.26 Scenario 4: Original and mutated old test case (Bad Smell 1)

The created original test case mutant was killed in the original system as the expected result is a Bag but not in the migrated system though the migration is similar (see Figure 7.27). This happened due to the incorrect implementation of *assert* method in the target testing environment (see Figure 7.28), which asserts only elements between two objects but not the type associated with the object. This suggests, the implementation of the assert function in the target environment is improper and it should be fixed. Thereafter, the test cases have to be executed again in the target environment.

```
[TestMethod]
  public void testCollectionAsBag() {
  var expectedResult = new Bag<object>{1,2.0,"3"};
  var actualResult = new Sequence<object>{1, 2.0,"3"}.asSequence();
  ocl.assertQueryResults(expectedResult,actualResult);
}
```

Figure 7.27 Scenario 4: Mutated migrated test case (bad smell)

```
public void assertQueryResults<T>(
AbstractCollection<T> expectedResult,
AbstractCollection<T> actualResult) {
  CollectionAssert.AreEqual(expectedResult,actualResult);
}
```

Figure 7.28 Scenario 4: Incorrect implementation of the assertion function

We kept this section rather short as a migration validation of developed mutation was performed as part of the feasibility studies presented in the next chapter.

7.4 Summary and Discussion

In this chapter, we introduced the third, and the last phase of the TeCoMi framework, namely the post-migration or test case migration validation phase, with its two main constituents, the mutation analysis repository and the validation engineering process. In the first part of the chapter, we introduced the content of the mutation analysis repository base, namely a set of mutation analysis scenarios, mutation operators, and mutation method patterns. Thereafter, we introduced a process to develop and enact validation methods, by using the mutation scenarios and mutation method patterns of the mutation analysis repository. Some of the findings presented in this chapter are based on a master's thesis [Bal19].

First, in Section 7.1, we gave an overview of the post-migration phase, by giving an overview of the structure of the mutation analysis repository as well as the structure of the migration validation process.

Then, in Section 7.2, we proposed a set of mutation analysis scenarios, mutation operators, and mutation method patterns stored in the mutation analysis repository. In the context of the TeCoMi framework, a mutation analysis scenario describes a particular strategy in terms of assumptions, guidelines, and suitability. In total, six different mutation analysis scenarios have been introduced depending on what is being mutated, either the original the target tests, original or the target systems or their transformations, i.e., test case or system migration. We further introduced three different categories of mutation operators that can be applied, namely, language mutation operators, test mutation operators, and domain-specific operators. Lastly, we introduced a set of three different mutation method patterns and depending on the abstraction level the mutation is performed, we distinguished between language-based test mutation, test language-based test mutation, and conceptual test mutation. A mutation method pattern can be seen as a technical implementation of the mutation analysis scenarios. In Section 7.3, we introduced the migration validation process, by discussing in detail each of the core activities. Firstly, we introduced a process to characterize the context from both the system and test perspective. As this was already addressed in the migration phase, we explained how the existing situational context model can be reused. However, seen from a validation perspective, we extended this with the created situation-specific test transformation specification and the developed situation-specific toolchain as part of the overall context information relevant for the migration validation process. Based on the discovered context information, as part of the next activity of the migration validation process, we described the process to systematically construct a validation method. Firstly, a proper mutation analysis scenario is selected. Based on this selection, a set of mutation operators is chosen and finally, a suitable mutation method pattern is selected and configured. Then, based on our mutation framework, tools are developed in order to automate the mutation of the test cases. Finally, the mutation is performed, the test cases are executed, and the execution results are analyzed based on the algorithms provided in the selected scenarios in order to eventually identify false positives or false negatives.

Currently, the specification of the mutation methods is mostly manual. However, the information from the previous phases can be used for automating certain parts of the specification of mutation methods. For example, suggestions for the optimal mutation scenario and mutation patterns can be made based on the situational context. We also introduced the model-driven mutation framework that addresses the automated generation of mutants. However, the part dealing with the test case execution and identification of bad smells is still

performed manually. By extending the mutation framework with automated execution and identification of problematic cases, the overall efficiency and effectiveness of the validation method can be improved.

In the next chapter, we present the two feasibility studies we have performed as part of the evaluation of the TeCoMi framework.

Part III

Evaluation and Conclusion

Chapter 8

Evaluation

In the previous chapters, the three main phases of the TeCoMi framework have been defined that enable the development and enactment of situation-specific test transformation methods. In this chapter, we describe two feasibility studies in which TeCoMi had been used to demonstrate the applicability of the framework. First, in Section 8.1, we present a set of evaluation questions related to the solution concept. The feasibility studies are described in Section 8.2 and Section 8.3. Thereafter, we discuss the evaluation questions based on the experiences made when performing the study in Section 8.4. The findings of this chapter are summarized in Section 8.5.

8.1 Evaluation Questions

In this section, we define a set of evaluation questions that address the main characteristics of the solution framework. The questions are then answered in Section 8.4 when discussing the outcome of the feasibility study that we have performed. We derived the evaluation questions from the requirements originally defined in Section 3.2. For each requirement, a corresponding evaluation question was derived which addresses two aspects, namely fulfillment and feasibility. The fulfillment aspect covers whether the solution approach addresses that requirement completely, whereas the feasibility aspect covers whether a certain task can be accomplished when using the solution approach. Therefrom, the corresponding evaluation questions are formulated as follows:

EQ1: Does the solution approach support the context-specific quality assessment of test cases in a systematic way?

EQ2: Does the solution support co-evolution analysis (i.e., change detection, impact analysis, and change propagation) between a system being migrated and its test cases?

EQ3: Does the solution approach enable automated transformation, i.e., migration of the test cases whenever possible?

EQ4: Does the solution support the construction of situation-specific transformation methods for test cases?

EQ5: Does the solution approach support the creation of a validation method for the validation of the test case migration?

In the following, we present the feasibility studies and discuss for each of them the previously described evaluation questions.

8.2 Feasibility Study 1: Junit OCL Test Cases to MSUnit OCL Test Cases

In this feasibility study, we migrated test cases of the well-known Eclipse Modeling Framework (EMF) [SBPM09] into a new environment by using the TeCoMi framework. EMF is highly adopted in practice which can generate source code from platform independent models with embedded Object Constraint Language (OCL) [OCL] expressions. Nowadays, more and more applications target multiple platforms like Windows, macOS, web browsers or mobile platforms like Android or iOS, which means that they need to be implemented in different programming languages. However, since its introduction in 2003, EMF is focused solely on Java as a target language. Hence, no feature-complete Ecore and OCL runtime APIs are available for all the platforms implying that their functionality has to be re-implemented.

CrossEcore [SJGE18], a multi-platform enabled modeling framework, addresses this drawback of EMF by supporting other target languages like C#, Swift, TypeScript, and JavaScript. Using CrossEcore, code from Ecore models with embedded OCL expressions can be generated. An OCL compiler translates OCL native expressions into expressions of the target language. Hence, CrossEcore's Ecore and OCL API can be consistently used across platforms. So, migration to CrossEcore was performed by applying the generic migration method that is used to adopt the CrossEcore's code generator for C# [SJGE18]. CrossEcore includes an OCL compiler that translates OCL expressions into respective expressions of the C# language. The migration strategy used was a semi-automated language-based transformation.

Once the system migration was completed, we used the TeCoMi framework to migrate the existing test cases addressing particularly the OCL implementation. The EMF's OCL

implementation is well-tested, with test cases available in the EMF public code repositories[1] (more than 4000 test cases). So, we decided to reuse. i.e., to co-migrate, these test cases by applying our framework. However, as we did not have any insight about the quality of the large set of test cases, before eventually starting with the actual migration, we had to analyze whether it is beneficial to migrate them all, part of them or none of them. It may be that some parts of the OCL were omitted purposely, thus making the corresponding OCL test cases not needed anymore.

Then, no transformation method was existent to guide the migration endeavor and which would address the fundamentally different implementations of OCL. Further, as the migration was performed to different targeting platforms, i.e., programming languages, a "one-size-fits-all" approach is not a perfect solution. This implies usage of a situation-specific transformation method, suitable for the situation for example regarding the target language or the target testing platform. The original test cases were written in JUnit which is the most commonly used Java Unit Testing framework. As the target environment, MSUnit, a C# unit testing framework, was selected, as we focused on the C# implementation in the CrossEcore framework. EMF and CrossEcore are significantly different in terms of the OCL implementation. The difference, namely the CrossEcore's "Ahead-of-Time" OCL implementation versus the EMF's "Just-in-Time" OCL implementation (cf. Section 3.1.1), was inherently relevant for the test cases. This comes from the dependency the test cases have to the system they test, i.e., the OCL implementation. So, those changes in the OCL implementation had to be reflected on the test cases in the target environment as well. Finally, once the test cases are migrated, a validation of the migrated test cases had to be performed in order to establish trust in the test results the migrated test cases would provide.

In the following, we describe the application of the TeCoMi framework to co-migrate the existing JUnit OCL test cases. We briefly describe the execution of all three phases of the approach, focusing mostly on the migration phase, i.e., on the enactment of all activities of the method engineering process. Additionally, the transformation of a selected test suite is described in detail.

8.2.1 Pre-Migration Phase

During this phase, the quality evaluation of test cases is performed, before performing any activity towards the migration of the test cases. For this purpose, we applied our approach called *Test Case Quality Plan (TCQP)*. By following its systematic process which considers the context information and integrates a standardized quality model, we created a quality

[1] http://git.eclipse.org/c/ocl/org.eclipse.ocl.git/tree/tests/

plan for our migration context. All activities were supported by our web-based tool *TCQEval* (Test Case Quality Evaluator), which was introduced in Section 5.1.5. The tool provides a user interface for the creation and edit of quality plans as well as for the analysis of the obtained evaluation results in terms of dashboards. In the following, we describe the activities of the pre-migration process we performed to evaluate the quality of the test cases.

Context Characterization

During this activity, the context information specific for the test cases to be migrated was identified (cf. Section 5.1.1). Context factors are essential elements that may affect the outcome of the evaluation and therefore we identified the test case context factors which include the environment, domain, and associated artifacts. During this activity, questions regarding the context, according to the context meta-model presented in Section 5.1.1, of the examined test cases were asked. For example, what is the framework the test cases are designed and executed in, what is the test level of the test cases, what is the criticality of the domain the test cases are used in, etc. The overall result is the context model shown in Figure 8.1.

Context Factor	Value
Test Object	OCL implementation in EMF
Test Suite	PivotTestSuite
Test Level	Unit testing
Development Phase	Migration
Test Case Type	Code-based test cases
Source Testing Framework	JUnit
Target Testing Framework	MSUnit
Test Item	Test Collection Operations
Test Tool	JUnit plugin for Eclipse

Figure 8.1 Context Model

Test Case Quality Plan Creation

Based on the identified context, the test case quality plan was constructed. Firstly, we identified and documented the quality goals for the evaluation (cf. Section 5.1.2). As shown in Figure 8.2, we have identified four main goals related to *Test Effectivity*, *Usability*, *Maintainability*, and *Reusability*. We further refined the identified goals into questions as

shown in the *Question* column in Figure 8.2. The goals and the questions were specified through interviews and structured brainstorming sessions with the stakeholders, namely the *Quality Manager* and the *Tester*. For establishing a common quality understanding (cf. Section 5.1.3), the goals defined were mapped to quality characteristics. Furthermore, for the corresponding questions, quality attributes were identified and documented, as shown in the *Quality Attribute* column in Figure 8.2.

Goal Dimension	Quality Characteristic	Quality Sub-Characteristic	Question	Quality Attribute
Analyze the **PivotTestSuite** for the purpose of **Quality Assessment** with respect to **Test Effectivity** from the viewpoint of **Quality Managers and Testers**, in the previously defined context	**Test Effectivity**	**Test Coverage**	What is the code coverage (line coverage)?	Line Coverage
		Test Correctness	Do all the test cases have expected result specified?	Specified expected result
			Do all the test cases have assertion specified?	Specified assertion
			Do all the test cases have actual result specified?	Specified actual result
Analyze the **PivotTestSuite** for the purpose of **Quality Assessment** with respect to **Usability** from the viewpoint of **Testers**, in the previously defined context	**Usability**	**Operability**	Are the test cases compilable?	Test case compilation
			Are the test cases executable?	Test case execution
		Understandability	Are the test cases properly commented?	Proper documentation
Analyze the **PivotTestSuite** for the purpose of **Quality Assessment** with respect to **Modifiability** from the viewpoint of **Quality Managers and Testers**, in the previously defined context	**Maintainability**	**Modifiability**	Are the test cases easily modifiable?	Test case modification
			Is there a test logic written in production code?	Test code dependency
			Is there redundant code in the test cases?	Test code redundancy
Analyze the **PivotTestSuite** for the purpose of **Quality Assessment** with respect to **Reusability** from the viewpoint of **Quality Managers and Testers**, in the previously defined context	**Reusability**	**Flexibility**	Are there hard-coded values in the test cases?	Hardcoded values
			Are there any test smells?	Test smells
		Coupling	Are there test cases with external data dependency?	External data dependency

Figure 8.2 Context-specific quality plan

Lastly, suitable *measures* were defined for all identified quality attributes (cf. Section 5.1.4), as illustrated in Figure 8.3.

Measurement Tool Implementation

Based on the defined measures shown in Figure 8.3, we approached to the implementation of the corresponding tools in order to execute the quality plan. The associated tool developers used the specification of measures and indicators as some kind of guidance. Here different strategies were applied, namely, development of a measurement tool from scratch and reuse of existing tools. For example, for the measure *Code Coverage* shown at the top-left corner of Figure 8.3, we decided to use the code coverage feature of the Eclipse Plugin for

Metric Definition	
Name	Code coverage
Informal Definition	Divide the number of source code lines covered by the test cases by the total number of lines in the source code.
Type of Measurement	Objective
Measurement Method	$\frac{\text{\# of source code lines covered by the test cases}}{\text{\# of lines in the source code}} \times 100$
Scale (Type)	0 to 100 (ratio)
Interpretation	closer to 100 is better
Tool Dependency	Yes (Eclipse JUnit coverage plugin)

Metric Definition	
Name	Ratio of Test Cases with Expected Results Specified
Informal Definition	Divide the number of test cases with expected results specified by total number of test cases
Type of Measurement	Objective
Measurement Method	$\frac{\text{\# of test cases with expected results}}{\text{\# of test cases}} \times 100$
Scale (Type)	0 to 100 (ratio)
Interpretation	closer to 100 is better
Tool Dependency	Yes (Test Case Parser)

Metric Definition	
Name	Ratio of Test Cases with Redundant Code
Informal Definition	Divide the number of test cases with redundant code by the total number of test cases
Type of Measurement	Objective
Measurement Method	$\frac{\text{\# of test cases with redundant code}}{\text{\# of test cases}} \times 100$
Scale (Type)	0 to 100 (ratio)
Interpretation	closer to 0 is better
Tool Dependency	Yes (CodePro / UCDetector)

Metric Definition	
Name	Ratio of Executable Test Cases
Informal Definition	Divide the number of executable test cases by the total number of test cases
Type of Measurement	Objective
Measurement Method	$\frac{\text{\# of executable test cases}}{\text{\# of test cases}} \times 100$
Scale (Type)	0 to 100 (ratio)
Interpretation	closer to 100 is better
Tool Dependency	No (simple execution is enough)

Figure 8.3 Excerpt of the defined base measures

JUnit [JUn]. The same applies to the measure dealing with redundant code (bottom-left corner of Figure 8.3). For some measures, no special tool was needed, as the condition in the measure was relatively trivial, namely, whether test cases are executable or not. In that case, a simple run of the test cases was sufficient. For other measures, like the one dealing with the presence of the expected result (top-right corner of Figure 8.3), the development of a tool, namely, a test case parser was necessary.

Execution and Decision-Making

Based on the developed quality plan and measurement tools, we performed the actual quality evaluation of the test cases. Figure 8.4 depicts the results obtained per each quality characteristic, quality sub-characteristic, and quality attributes in the TCQEval tool. The scores presented in this dashboard are calculated firstly for each quality attribute (bottom part in Figure 8.4) based on the measures previously defined in Figure 8.3. Each quality sub-characteristic's score is an average score of the corresponding quality attributes' scores. Similarly, each quality characteristic's score is an average score of the corresponding quality sub-characteristics' scores. For example, the *Test Effectivity* score is about 80The reason for this is the Test Coverage, which was about 60The Test Correctness was 100% as all of the test cases had specified an expected result, assertion, and actual result. To closely identify the problem with the Test Coverage, we checked the quality attribute Line Coverage, and the corresponding tool that measured it. The results were lower as expected as the tool was

measuring also the test code as part of the line coverage, which was strange. The actual system code coverage was about 99%, which was a clear suggestion that almost the complete code was covered with the existing tests. In a similar way, the other quality scores were analyzed with the test experts and migration experts and the decision was made that all of the test cases have to be migrated without any a priori changes of the test cases. The structural changes that were desired by the test expert were planned to be performed as part of the migration phase. Having a positive outcome of the decision-making process, we proceeded to the actual migration of the test cases.

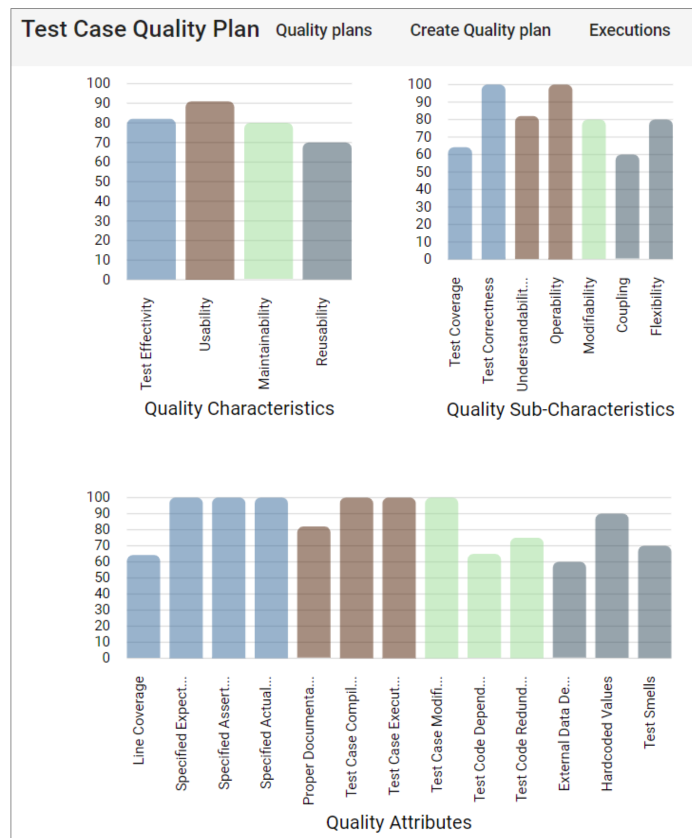


Figure 8.4 TCQEval dashboard showing the results of the performed quality evaluation

8.2.2 Migration Phase

During the migration phase, activities of the two main disciplines, namely, *Method Development* and *Method Enactment* were performed. As part of the first discipline, firstly the situational context was identified, which also included co-evolution analysis. Thereafter, we created the situation-specific test transformation method. As part of the latter discipline, a situation-specific toolchain was implemented and the test cases were transformed by enacting

the developed test transformation method. In the following, we present each activity that was performed in detail.

Situational Context Identification

During the first activity of the method engineering process, the migration context was analyzed and characterized, from both system migration and testing perspective. Thereafter, co-evolution analysis was performed to identify the impact that the system changes have on the test cases. Finally, we also analyzed the influence factors for each test concept. The resulting concept model is depicted in Figure 8.5. As can be seen, besides the concepts, this model contains also the dependencies between test and system concepts as well as the suitable patterns for the test concept.

Firstly, the concept model of the OCL implementation in both the source and target environment had to be identified and modeled. Regarding the identification of the system concepts, firstly the shared concept *OCL* was identified, which corresponds to the Object Constraint Language. Then, for both source and the target environment, this shared concept was refined to concepts showing the type of the OCL implementation, namely, *OCL (Just-in-Time)* as a source concept, and *OCL (Ahead-of-Time)* as a target concept. The source concept *OCL (Just-in-Time)* consists of *Native OCL-Expressions*, whereas the target concept *OCL (Ahead-of-Time)* consists of *Language-specific OCL-Expressions*. This basically shows the crucial differences in the implementation of the OCL language in the source and target environments. Further, both these concepts were further refined to *Derived Expression*, *Operation*, and *Constraint*, as core features of the OCL language. All of these three concepts are represented as *String Literal* concepts, as they are implemented in that in the source environment. In the target environment, however, a *Derived Expression* and an *Operation* consist of a *Function* and a *Constraint* consists of a *Logical Expression*. Furthermore, the transformation strategy used for the system has been addressed too. The applied system pattern *Language Transformation* represents the strategy used to perform the migration of the system. Having the system concepts identified and modeled, the test concepts had been addressed. As a shared test concept, the *OCL Test Case* was identified. This abstract test concept was refined into the concrete test concept *OCL Test Case (JIT)*, which represents a concrete test case implemented in a JIT manner. As a test case in the source environment contains an assertion function, an *Assertion* concept was added. Further, each *Assertion* consists of an *Action* and an *Expected Result*. Thereafter, the target test concepts, i.e., the test concepts in the target environment had been also modeled. Firstly, the shared test concept was refined into *OCL Test Case (AOT)*, which represents a concrete test case which should be implemented in AOT manner. This comes from the fact that this change can be seen

in the system concepts as well. Furthermore, there is a structural difference in the target realization of the OCL test cases, namely, the *Action*, the *Assertion*, and the *Expected Result* concepts are on the same level. Having the concepts identified, the impact analysis had been performed. Firstly, the corresponding source and target concepts had been identified, i.e., the *OCL (JIT)* and *OCL (AOT)* had been related as corresponding concepts. The same is done for the corresponding source and target test concepts, i.e., for *OCL Test Case (JIT)* and *OCL Test Case (AOT)*. Next, the dependencies between system and test concepts had been identified. Firstly, the dependency between the shared test concept *OCL Test Case* and the shared system concept *OCL* had been established. Then, on the basis of this dependency and by following the refined concepts, the dependencies between the concrete system and test concepts had been established. Hence, the source test concepts *Action* and *Expected Result* had been related to the source system concept *Native OCL-Expression* as they depend on this concept. The same thing had been performed for the target system and test concepts. Having the concepts as well as the correspondences and dependencies identified, the influence factors had been identified. Namely, for the shared test concept *OCL Test Case*, the suitable test patterns had been identified and analyzed.

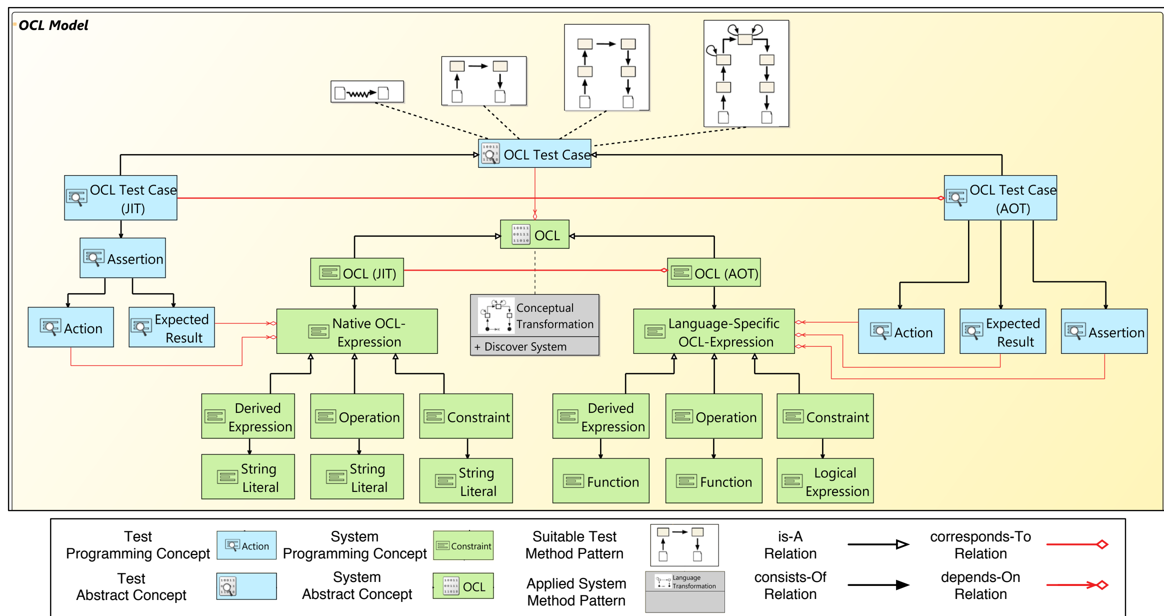


Figure 8.5 Configured concept model with the dependencies between the test and system concepts

On the right side of Figure 8.5, the envisioned realization of the test concepts in the target environment is shown. So, as the system concepts are realized in a different way in the target environment, namely, in an *Ahead-of-Time* manner, and due to the dependency, the test concepts have to the system concepts, we intended to do the same for the test concepts.

Furthermore, there were additional test requirements regarding the restructuring of the test cases in the target environment. As the system concepts would be realized differently in both environments, i.e., *Just-in-Time* and *Ahead-of-Time*, and therefore the test concepts as well, the experts assumed that the *Test Language-based Test Transformation Pattern* would be suitable. By further analyzing the method fragments of this pattern as well as the corresponding co-migration patterns, this hypothesis was further reinforced by the identified influence factors:

- For Java, an open-source parser was available, namely, the JDT Parser provided by the Java Development Tools (JDT)[2].
- Platform-specific test metamodels could be created based on [JSW07].
- The parsing of the OCL native strings contained by the original platform-specific model can be reused from the system migration (according to the Co-Migration Pattern CMP8 in Section 6.2.4).
- The transformation, i.e., the test concretization from the migrated platform-specific model to the model of the migrated test code can be reused from the system migration (according to the Co-Migration Pattern CMP8 Section 6.2.4).
- The test code generation templates could partly be derived from the code generation templates defined for the system code generation.

Transformation Method Construction

On the basis of the identified situational context, the test transformation method was constructed. Firstly, a method pattern is been selected, namely, the *Test Language-based Test Transformation Pattern* and has been coarse granularly configured for each test concept. Due to the good structure of the realization of the source test concepts, it was possible to transform all of the identified test concepts automatically. The completed horseshoe model is shown in Figure 8.6.

As shown in the concept model in Figure 8.5, the test concept *OCL Test Case (Just-in-Time)* consists of an *Assertion* which further consists of an *Action* and an *Expected Result*. Consequently, as part of the *Test Case Understanding* fragment, these concepts are extracted from the *Model of the Test Code (MOTC)*, or more specifically, the Java AST (Abstract Syntax Tree). For each of the three concepts, a corresponding action as part of the *Test Case Understanding* fragment is instantiated. Namely, *Extract Assertion*, *Extract Expected*

[2]https://www.eclipse.org/jdt/

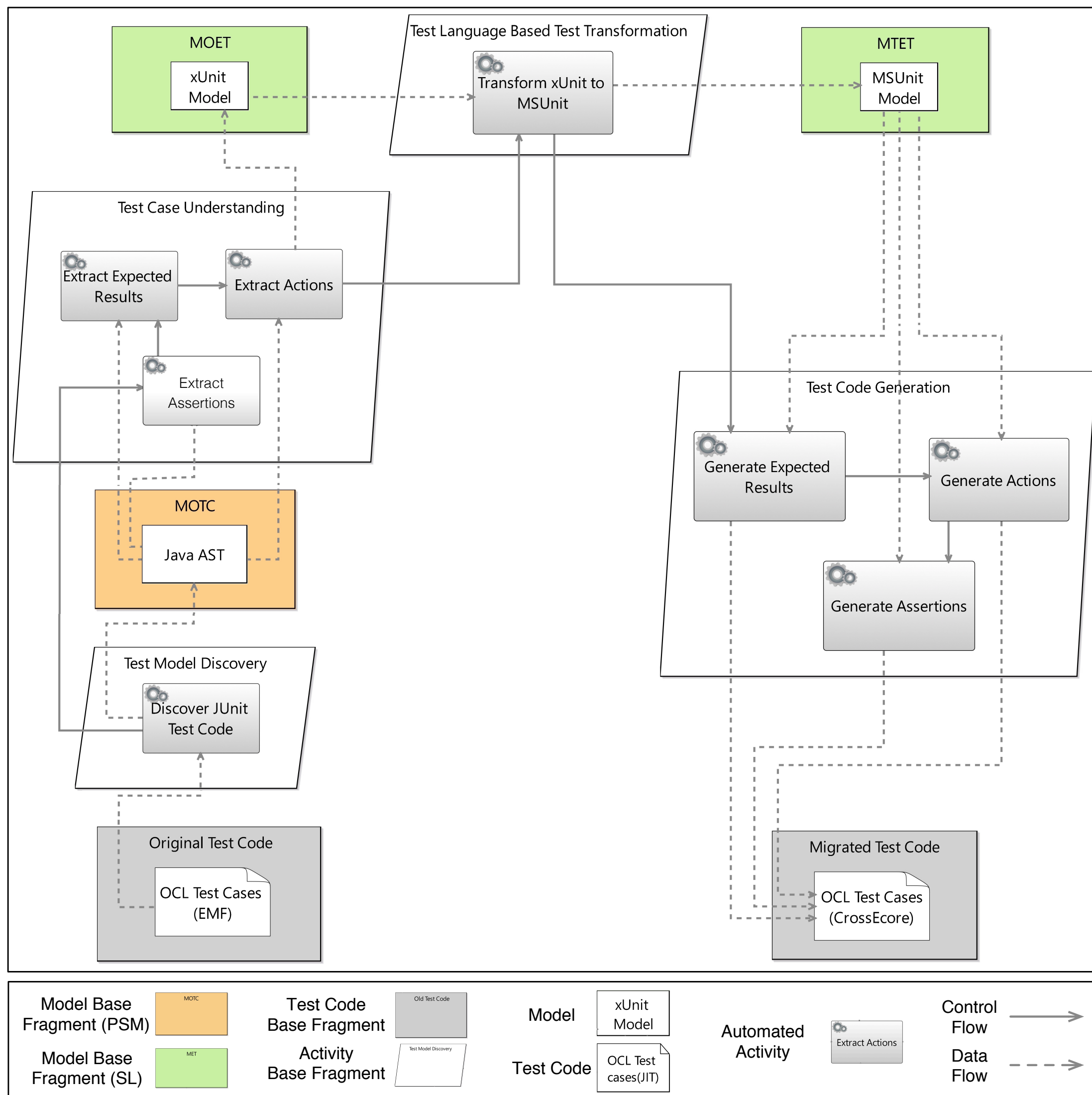


Figure 8.6 Horseshoe model for the OCL test case concept

Result, and *Extract Action* are the three automated activities that extract the concepts from the abstract syntax tree. The result of the *Test Case Understanding* is specified in terms of an *xUnit Model* which is a *Model of the Executable Tests*. So, due to the similarity of these concepts, namely, they all depend on the *Native OCL Expression* system concept, the same pattern was selected for all of them. Another reason for the selection of this pattern was that the target test framework namely the MSUnit testing framework is comparable to the JUnit framework regarding the architecture.

Also, at this level of abstraction was relatively easy to apply the same transformation rules for the OCL expressions to the target environment which were used in the system migration [SJGE18]. Finally, as part of the *Test Code Generation* fragment, three different

automated generation activities (*Generate Expected Result*, *Generate Action*, and *Generate Assertion*) are instantiated for each of the three test concepts. Namely, by combining the result of each of these activities, which rely also on the existing transformation rules for the OCL expressions from the system migration, the *Migrated Test Code*. More specifically, the OCL test cases for the CrossEcore implementation of OCL in C# are obtained.

Tool Implementation

Having the transformation method specification developed, the required tools were developed. As shown in Figure 8.7, the tool infrastructure has a component-based architecture, namely, there are three general components *Input Module*, *Transformation Module*, and *Output Model*. Some components within this architecture are project-independent, thus enabling further reuse in the next projects.

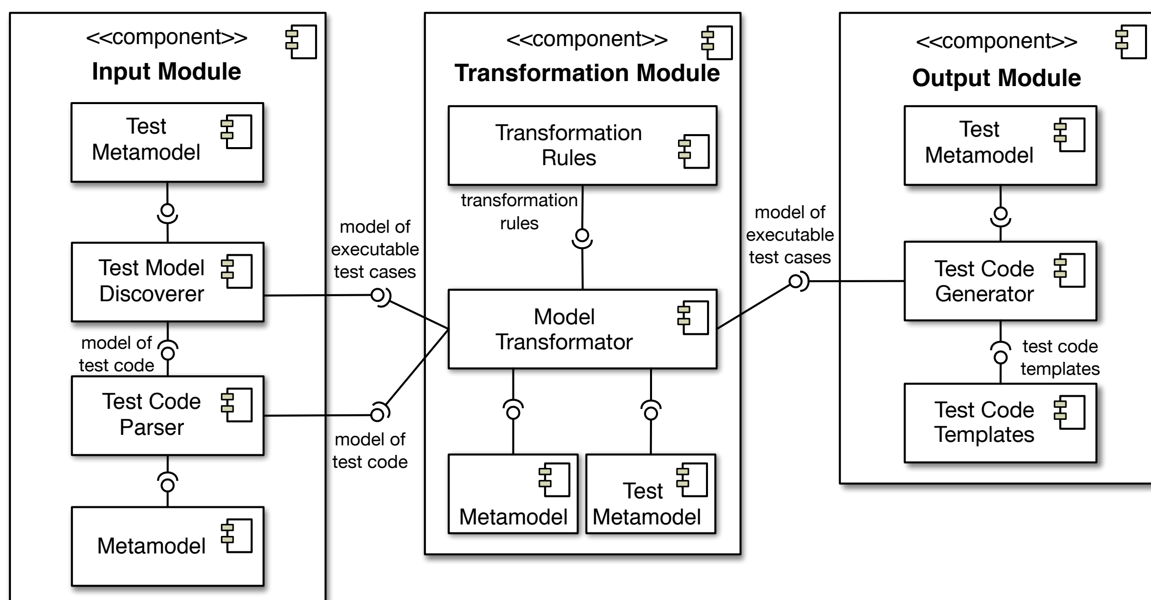


Figure 8.7 Generic Tool Infrastructure

The *Input Module* comprises the *Test Code Parser* and the *Test Model Discoverer*. The *Test Code Parser* component was responsible for the initial parsing of the original test cases written in JUnit. Technically, this component was based on the JDT Java Parser which delivers a Java Abstract Syntax Tree of the original test code. The *Test Model Discoverer* component takes the outcome of the *Test Code Parser* and extracts a model of the executable test cases conform to the xUnit metamodel [JSW07].

The models produced by the *Test Model Parser* as well as the *Test Model Discoverer* are provided to the *Transformation Module* or more specifically, to the *Model Transformator*

component. The *Model Transformator* relies on the *Transformation Rules* specified to perform the actual adaptation and restructuring of the test models. Technically, the *Model Transformator* was implemented in Java and the *Transformation Rules* rely on the Java xUnit API of the test models. The *Transformation Module*, i.e., the *Model Transformator* component is a project-independent component, hence it can be reused for subsequent projects.

The transformed model produced by the *Model Transformator* is input for the third module, namely the *Output Module*. More precisely, the *Test Code Generator* takes the model of executable code and on the base of *Test Code Templates*, generates the test cases for the MSUnit framework. The *Test Code Templates* were implemented in Xtend and rely on the xUnit metamodel.

Transformation

As the required tools were implemented, the actual test transformation of the original test cases into the target test environment was carried out. By using the developed tools on the original test cases, the automatic conversion was performed. Namely, around 4000 test cases of 12 different test suites were migrated. Each of the different test suites was targeting different parts of the OCL language, like *Collection Operations*, *Boolean Operations*, *Classifier Operations*, etc. In total, about 3700 test cases with more than 40.000 LOC were generated by the toolchain, thus achieving an automation rate of 92%.

Figure 8.8 shows an excerpt from the models and the entities which are arising when converting the test concepts *Assertion*, *Expected Result*, and *Action*. This figure shows actually the enactment of the transformation method specified in Figure 8.6. Firstly, the test code of the original test cases in JUnit was parsed in order to obtain the *Model of the Test Code* as shown in the lower left of Figure 8.8. In the concrete example, a test case named `testCollectionAsBag` was parsed. As can be seen, it contains a single assertion function, namely `assertQueryResults` which has three input parameters. The second parameter and the third parameter, the expected result and the action to be performed, respectively, show the Just-in-Time realization of the OCL as they are native OCL expressions specified as strings. The outcome of the parsing activity is a Java abstract syntax tree, which still does not contain any test-specific constructs like assertion or expected result. The obtained abstract syntax tree is the input for the next activity which extracts xUnit test-specific constructs like `TestCase`, `ExpectedValue` or `Assertion`. The model containing these constructs is the platform-specific *Model of the Executable Test Cases*, shown in the top left corner of Figure 8.8. By further applying transformations, the xUnit model is transformed to a MSUnit model of the executable tests, as MSUnit is the target framework. As one can

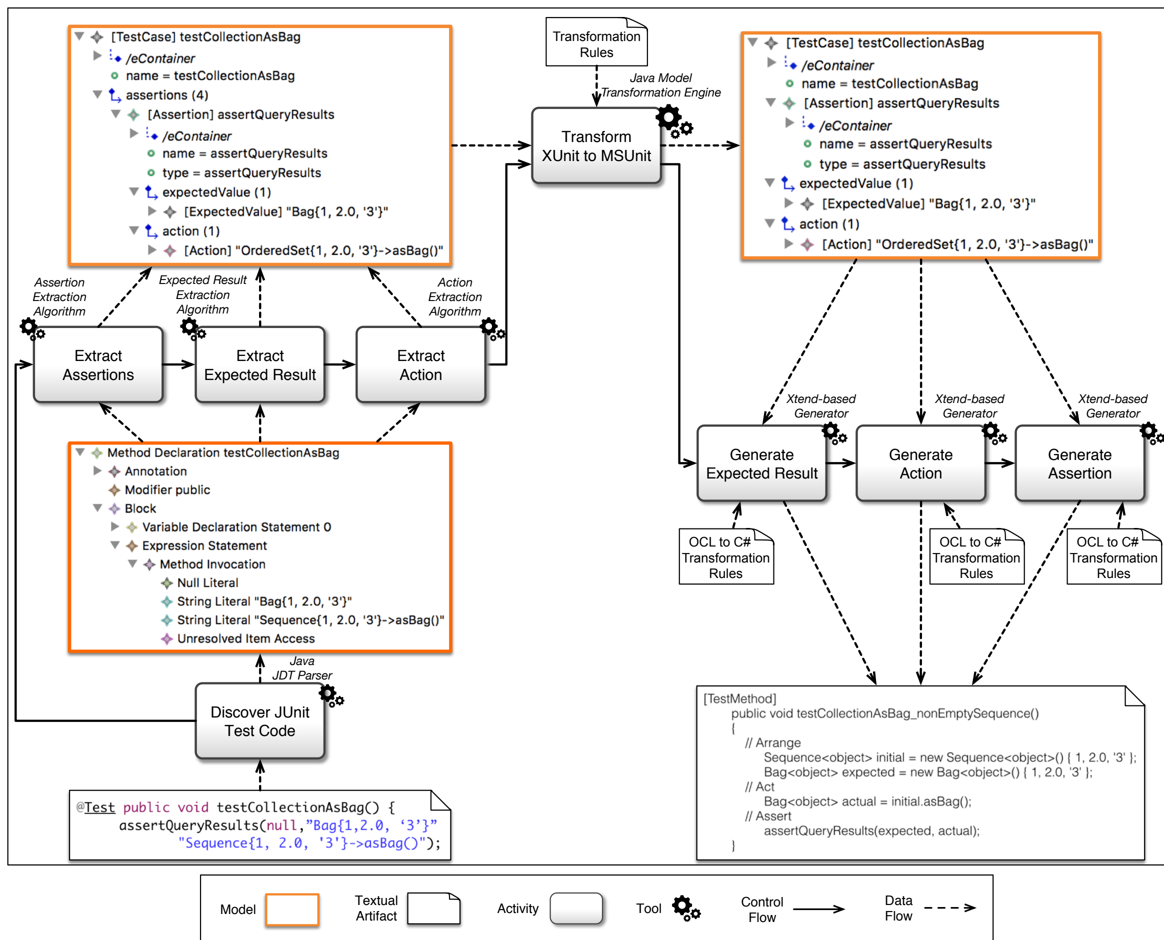


Figure 8.8 Enacting a transformation method to transform the test concepts

observe, `ExpectedValue` and the `Action` are no longer contained by the corresponding `Assertion`. The transformed model of the executable test cases was thereafter concretized in the target environment, i.e., the test code for the MSUnit test cases was generated. In order to address this task, based on test code generation rules provided for each of the three concepts, namely, *Generate Expected Result*, *Generate Action*, and *Generate Assertion*), the test code was generated. The native OCL expression strings contained by the model of the executable test cases were concretized to the target framework, i.e., CrossEcore, by applying the OCL to C# transformation used in the system migration. Note the difference of the test code in the bottom right corner of Figure 8.8, where corresponding API classes and functions were used instead of native OCL expressions.

8.2.3 Post-Migration Phase

After the test cases have been transformed, i.e., migrated, a migration validation has taken place with the main goal to identify false positives and false negatives among them when executed. In the following, we discuss the activities we performed to develop and execute the validation method.

Situational Context Identification

The obtained *Situational Context Model*, obtained in the previous phase, was here reused for the development of a suitable test case validation method. Therefore, we elaborate more on the usage of the context information in the subsequent activity, namely the *Mutation Method Construction*.

Mutation Method Construction

Firstly, according to the process we presented in Section 7.3, one mutation scenario had to be selected from the *Mutation Analysis Repository*. As we already described in Section 7.2.1, each mutation scenario contains a discussion on its suitability. However, at this step, we decided to apply multiple scenarios, and try to identify as many bad smells as possible. Namely, the target system was mutated (Scenario 2), the original and the migrated test cases were mutated (Scenario 5 and 6 respectively) and the migration of the test cases was mutated (Scenario 6). So, all scenarios except *Scenario 1* and *Scenario 3* were used. According to the description of *Scenario 1* in Section 7.2.1, it addresses the addition of adequate tests to increase the quality of an existing test suite. As OCL is a well-tested framework, this scenario was not carried out. Also, *Scenarios 3* was not performed, as the migration of the system was done manually. As the transformation performed on the test cases was test language-based, the mutation of the test cases was also test language-based. The mutations of the migrated system and the migration of the test cases were performed manually. The mutation operators were based on the structure of the test cases checking the collections and the corresponding methods like `Append`, `AsBag`, `AsSequence`, `AsSet`, `Excludes`, `ExcludesAll`, `Excluding`, `First`, `Includes`, etc. A complete list of these mutation operators was already presented in Figure 7.14. Also, in Section 7.3.2, a comprehensive discussion on the mutation method construction as well as on the decisions made was presented.

Mutation Tool Implementation

To improve the process of automated mutation and to increase the reuse of existing components, we have already introduced a flexible and extensible model-driven mutation framework

in Section 7.3.3. Similar to the tool architecture of the main phase, it is a component-based framework, consisting of three main components, *Input Module*, *Mutation Engine*, *Output Module*, as shown in Figure 7.20. The *Input Module* was completely reused from the migration tool infrastructure, namely, the parser of the test cases as well as the extractor of the xUnit test cases were reused. Then, for the *Mutation Module*, a set of *Mutation Operators* has been implemented in terms of transformation rules in *EMSL*[3], a uniform language for model management of eMoflon. The *Output Module* had to be implemented as the mutated test cases shad to be generated for the same language, namely Java, i.e., the JUnit testing framework. In the case of MSUnit, the test code generator from the main phase could be reused. The realization of the test code generator was done in Xtend.

Mutation, Test Execution, and Analysis

After the required tooling is developed, the validation approach was performed. The evaluation was carried out in a systematic way generating 5 mutants in a cycle for each scenario and then, repeating the same procedure. The non-uniform number of mutants created is due to the fact that importance was given to that scenario where we could not identify a bad smell. Figure 8.9 shows for each evaluated scenario the number of created mutants, killed mutants, bad smells and no indication cases, i.e., the equivalent mutants. In the following, we explain each of them in detail.

Scenario	Created Mutants	Killed Mutants	Bad Smell	No Indication
2	37	31	0	6
4	30	23	4	3
5	31	25	2	4
6	30	28	2	0

Figure 8.9 Evaluation Results

Scenario 2: Mutation of Migrated System. As we have not performed *Scenario 1*, part of *Scenario 2*, which involves generating old system mutants from migrated system mutants using the backward transformation is not carried out and hence, *Scenario 2* is only partially performed. Out of 37 created migrated system mutants, 31 mutants got killed and 6 mutants were found to be equivalent mutants.

Scenario 4: Mutation of Old Test Cases. Concerning *Scenario 4*, 30 old test case mutants were created to validate the implementation of the old system followed by transforming them into the target environment and validating the implementation of the migrated system. Out of

[3]https://emoflon.org/

30 created mutants, 23 mutants got killed in the old and migrated environment, 3 mutants were found to be equivalent mutants and 4 mutants were classified as a bad smell. In the following, we describe two of them.

Bad smell 1. This bad smell occurs when a collection entity such as Bag, Set, OrderedSet, and Sequence is converted to another type. In Figure 8.10, a Sequence is converted to Bag where we created a mutant by applying the *method mutator*[4] to convert a Sequence to Sequence.

```
@Test
public void testCollectionAsBag() {
  //Original
  //ocl.assertQueryResults("Bag{1,2.0,'3'}",
                           "Sequence{1,2.0,'3'}->asBag()");
  //Mutant – Applied mutation operator: Method mutator
  ocl.assertQueryResults("Bag{1,2.0,'3'}",
                         "Sequence{1,2.0,'3'}->asSequence()");
}
```

Figure 8.10 Scenario 4: Original and mutated old test case (Bad Smell 1)

The created old system mutant was killed in the old system as the expected result is a Bag but not in the migrated system though the migration is similar (see Figure 8.11). This happened due to the incorrect implementation of the assert method in the target (see Figure 8.12), which asserts only elements between two objects but not the type associated with the object.

Bad smell 2. This bad smell occurs when the type of expected result is modified. In Figure 8.13, the original test case expects an OrderedSet with a set of elements where we created a mutant by changing the type of the object expected to a *Set*, which is killed in the old system as the expected result is an OrderedSet but not in the migrated system (see Figure 8.14). The reason is again due to the incorrect implementation of assert method in the target testing environment.

Scenario 5: Mutation of Migrated Test Cases. In the case of this scenario, 31 migrated test case mutants were created. When migrated to the old environment, 25 mutants got killed in the old and migrated environment, 4 mutants were found to be equivalent mutants and 2 mutants were classified as bad smells, basically, the same bad smells as those in *Scenario 4*.

Scenario 6: Mutation of Test Case Migration. Concerning *Scenario 6*, 30 migrated test case mutants were created by mutating the transformation. Out of 30 created mutants, 28 mutants got killed in the old and migrated environment, no mutants were found to be equivalent mutants and 2 mutants were classified as bad smells. In the following, we explain one of them:

[4] https://github.com/stryker-mutator/stryker-handbook

```
[TestMethod]
  public void testCollectionAsBag() {
  var expectedResult = new Bag<object>{1,2.0,"3"};
  var actualResult = new Sequence<object>{1, 2.0,"3"}.asSequence();
  ocl.assertQueryResults(expectedResult,actualResult);
}
```

Figure 8.11 Scenario 4: Mutated migrated test case (Bad Smell 1)

```
public void assertQueryResults<T>(
AbstractCollection<T> expectedResult,
AbstractCollection<T> actualResult) {
  CollectionAssert.AreEqual(expectedResult,actualResult);
}
```

Figure 8.12 Scenario 4: Incorrect implementation of the assertion function

```
@Test
public void testCollectionAppend() {
  //Original
  //ocl.assertQueryResults("OrderedSet{1,3,4,2}",
                          "OrderedSet{1..4}->append(2)");
  //Mutant – Applied mutation operator: Method mutator
  ocl.assertQueryResults("Set{1,3,4,2}",
                        "OrderedSet{1..4}->append(2)");
}
```

Figure 8.13 Scenario 4: Original and mutated old test case (Bad Smell 2)

```
[TestMethod]
public void testCollectionAppend() {
  var expectedResult = new Set<int>{1,3,4,2};
  var actualResult = new OrderedSet<int>{1,2,3,4}.append(2);
  ocl.assertQueryResults(expectedResult,actualResult);
}
```

Figure 8.14 Scenario 4: Mutated migrated test case (Bad Smell 2)

Bad smell: The bad smell in this scenario is related to the transformation used in the migration of test cases which is found when conditional boundary mutator is applied. Figure 8.15 shows the original test cases that assert the *greater than* operator applied on numbers. Figure 8.16 shows the mutated test case transformation by swapping the *greater than* operator (“>") with *greater than or equal* operator (“≥"). Four mutants were expected, two being equivalent and two being killed. However, as shown in Figure 8.16, only two mutants got created, which resulted in an incorrect transformation. Also, only one mutant got killed, which indicates a bug in the test case transformation and requires further investigation.

```
@Test
public void testNumberGreaterThan() {
  ocl.assertQueryTrue("3 > 2");
  ocl.assertQueryTrue("3.0 > 2.0");
  ocl.assertQueryFalse("3.0 > 3");
  ocl.assertQueryFalse("3 > 3.0");
}
```

Figure 8.15 Scenario 6: Original test cases asserting results of the *greater than* operator

```
...
if(callExp.referredOperation.EContainingClass.name.equals("Integer_Class")) {
  if(callExp.referredOperation.name.equals(">")) {
    //Original
    //return '''«sourceResult» > «argumentResults.get(0)»''';
    //Mutant – Applied mutation operator: Conditionals boundary mutator
    return '''«sourceResult» >= «argumentResults.get(0)»''';
  }
}
...
```

Figure 8.16 Scenario 6: Original and mutated test case transformation

```
[TestMethod]
public void testNumberGreaterThan_1() {
  var actualResult = 3 >= 2;// Mutant
  ocl.assertQueryTrue(actualResult);
}
[TestMethod]
public void testNumberGreaterThan_2() {
  var actualResult = 3.0 > 2.0;// Not a mutant
  ocl.assertQueryTrue(actualResult);
}
```

```
[TestMethod]
public void testNumberGreaterThan_3() {
  var actualResult = 3.0 > 3;// Not a mutant
  ocl.assertQueryFalse(actualResult);
}
[TestMethod]
public void testNumberGreaterThan_4() {
  var actualResult = 3 >= 3.0;// Mutant
  ocl.assertQueryFalse(actualResult);
}
```

Figure 8.17 Scenario 6: Generated migrated test case mutants showing missing migrated test case mutants

In summary, this evaluation shows that the proposed scenarios can be used in real-world migration projects to identify false positives and false negatives among the migrated test cases. Even with a moderate number of mutants, we were able to identify problematic test cases. The question is, however, which scenarios should be used for which purposes. As each scenario has its own set of assumptions and indications, they define practically to which migration situation is suitable. For example, the evaluation with the running example, we have taken, gave us more bad smells concerning *Scenario 4*. However, *Scenario 5* gave us the already occurred bad smells of *Scenario 4* which makes it being redundant. Nevertheless, based on the investigation of *Scenario 4* bad smells, we can confirm that they are false negatives, which occur when the system migration is incorrect but when we take *Scenario 6*, the occurred bad smells are due to bugs in the test case transformation, which calls for an investigation in the transformation itself. As each basic mutation scenario, according to its

assumptions, suits a given situation, some situations may require a combination of multiple basic scenarios, i.e., a composed mutation scenario. For example, a part of the test cases may be migrated automatically, requiring usage of *Scenario 6*, and another part may be migrated manually, thus requiring usage of *Scenario 5*. In that case, a combination of both mutation scenarios would be the strategy to be followed.

So, all in all, this feasibility study demonstrated the benefit of our approach for supporting the co-migration of test cases. By using our method engineering process which includes co-evolution analysis and is supported by a modeling tool, we were able to develop a situation-specific test migration method and to migrate the original test cases to the target testing framework, namely MSUnit.

8.3 Feasibility Study 2: JUnit OCL Test Cases to Jasmine OCL Test Cases

As we already mentioned in the previous feasibility study, the CrossEcore [SJGE18] is a multi-platform enabled modeling framework which besides C# also addresses TypeScript as target language. In order to test the CrossEcore's OCL implementation in this language, we used once again the TeCoMi framework. In comparison with the first feasibility study, we focus here on the migration phase. So, we skip the pre-migration phase and the quality evaluation of test cases as the original test cases and the source environment, in general, are the same. Regarding the migration phase, as the target testing platform is different compared to the first feasibility study, we cannot reuse the same method. Seen from a testing perspective, as target testing framework Jasmine[5], a JavaScript behavior-driven testing framework that follows the BDD (Behavior-Driven Development [N$^+$06]) principle was selected. Seen from an architectural perspective, this testing framework is completely different compared to the JUnit testing framework. Namely, it follows the *Given-When-Then* structure for the specification of test cases. So, due to this fundamental difference between the source and the target testing framework, the migration needed to be performed on a higher level of abstraction. Seen from a system migration perspective, the EMF and CrossEcore are significantly different in terms of the OCL implementation, namely the CrossEcore's "Ahead-of-Time" versus the EMF's "Just-in-Time" OCL implementation. Those changes in the OCL implementation had to be reflected on the test cases in the JavaScript target environment as well. In the following, we describe the application of the TeCoMi framework to migrate the JUnit OCL test cases to the Jasmine testing framework. We mainly focus on

[5]https://jasmine.github.io/

the migration phase, as the architectural difference between the source and the target testing frameworks has the strongest manifestation there.

8.3.1 Migration Phase

During the migration phase, firstly, the situational context was identified, also including the co-evolution analysis. Thereafter, a situation-specific test transformation method was created and a situation-specific toolchain was implemented. Next, the test cases were transformed by enacting the developed test transformation method to the target testing framework namely, Jasmine. In the following, each activity that was performed is presented in detail.

Context Characterization

During this activity, the migration context was analyzed from system migration as well as testing perspective. Then, to identify the impact that the system changes have on the test cases, co-evolution analysis was performed. Finally, the influence factors were identified and analyzed for each test concept. Figure 8.18 depicts the resulting concept model. Besides the concepts, also the dependencies between test and system concepts as well as the suitable patterns for the test concepts are shown.

As the concept model of the OCL implementation in both source and target environment is similar to the concept model presented in the first feasibility study (Figure 8.5), we do not present the same details again. The same applies to the source test concepts, whose explanation can be found in Section 8.2.2. In the following, we only focus on the target test concepts as they are different due to the different target testing environment.

So, after having the system concepts identified and modeled, the test concepts had been addressed and as a shared test concept, the *OCL Test Case* was identified. This abstract test concept was refined into the concrete test concept *OCL Test Case (JIT)*, which represents a concrete test case implemented in a JIT manner. As a test case in the source environment contains an assertion function, an *Assertion* concept was added. Further, each *Assertion* consists of an *Action* and an *Expected Result*. Thereafter, the target test concepts, i.e., the test concepts in the target environment had been also modeled. Firstly, the shared test concept was refined into *OCL Test Case (AOT + BDD)*, which represents a concrete test case which should be realized in the target environment. The "*AOT + BDD*" references actually the two main characteristics of the test case. The AOT (Ahead-of-Time) part, similarly as with the first feasibility study, deals with the AOT implementation of OCL in the target environment. The BDD (Behavior Driven Development) part addresses the way the test cases are realized in the target environment, namely, they follow the BDD principle. Hence,

the target realization of the OCL test cases is structurally different, namely they follow the *Given-When-Then* pattern[6]. The *Given* part defines the initial state, the *When* part defines the action that transforms the state, and finally, the *Then* checks the state. Therefore, each of them is defined as a test concept and is contained by the *OCL Test Case (AOT + BDD)*.

Thereafter, based on the identified system and test concepts, the impact analysis had been performed. Again, we focus on the part which deals with the target test concepts. The source test concepts we have identified, namely the *Given*, *When* and *Then* test concepts had been related to the source system concept *Language-based OCL-Expression* as they depend on this concept. After the concepts and their correspondences and dependencies have been identified, the influence factors had been also identified. Namely, again, for the shared test concept *OCL Test Case* suitable test patterns had been identified and their advantages and disadvantages have been analyzed.

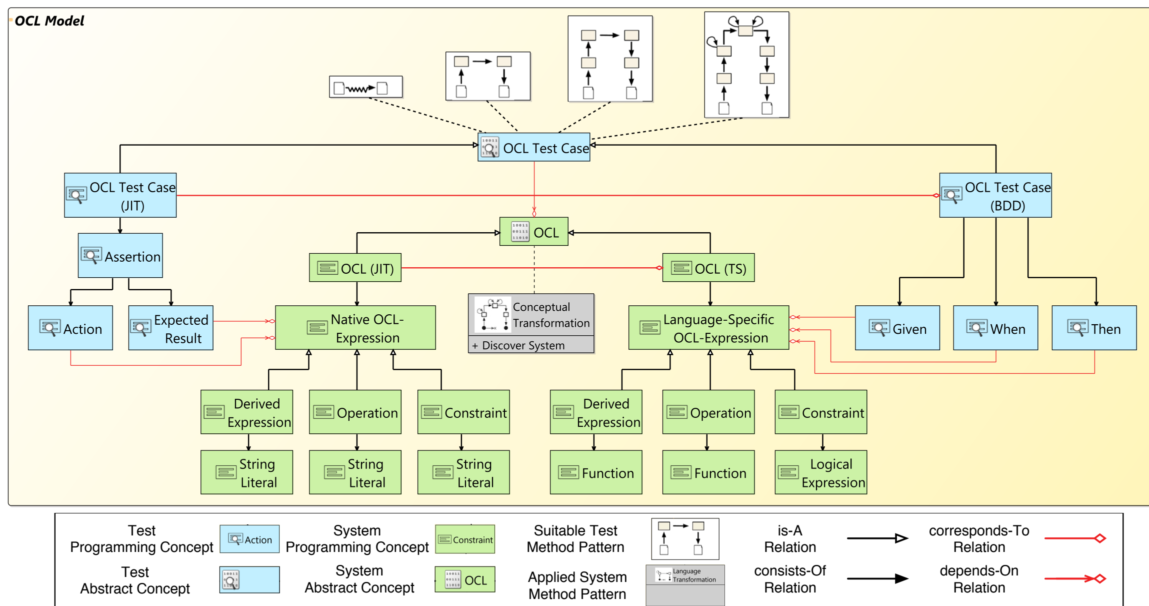


Figure 8.18 Configured concept model with the dependencies between the test and system concepts

As we have already presented, on the right-hand side of Figure 8.5, the envisioned realization of the test concepts in the target environment is shown. So, as have already concluded, the different realization of system concepts in the target environment, namely in an *Ahead-of-Time* manner, implies that the test concepts have to be realized in the same manner. However, the planned realization of the test cases, namely the Given-When-Then style in the Jasmine testing framework, is completely different compared to the xUnit style

[6]https://martinfowler.com/bliki/GivenWhenThen.html

of realization of the original test cases. Due to these differences that come from both system and testing side, the experts assumed that the *Conceptual Test Transformation Pattern* would be suitable. Therefore, the method fragments of this pattern as well as the corresponding co-migration patterns have been analyzed. The outcome of this analysis, namely the identified influence factors, have further reinforced this hypothesis:

- An open-source parser was available for Java, namely the JDT Parser provided by the Java Development Tools (JDT)[7].
- Platform-specific test metamodels could be based on the xUnit metamodel presented in [JSW07].
- The parsing of the OCL native strings contained by the original platform-specific model can be reused from the system migration (according to the Co-Migration Pattern CMP8 in Section 6.2.4).
- Due to the difference in the architecture of the testing frameworks, the conceptual transformation is a more suitable approach.
- UML Testing Profile [OMG13b] can be used for representing the Model of Abstract Tests.
- The transformation, i.e., the test concretization from the migrated platform-specific model to the model of the migrated test code can be reused from the system migration (according to the Co-Migration Pattern CMP8 Section 6.2.4).
- The test code generation templates could partly be derived from the code generation templates defined for the system code generation.

Transformation Method Construction

Based on the identified situational context, the test transformation method had been constructed. First, the *Conceptual Test Transformation* method pattern had been selected and a coarse granular configuration for each test concept has been done. As the realization source test concepts had a good structure, all of the identified test concepts could be transformed automatically. The completed horseshoe model is shown in Figure 8.19.

The test concept *OCL Test Case (Just-in-Time)*, according to the concept model shown in Figure 8.18, consists of an *Assertion* which further consists of an *Action* and an *Expected Result*. As all these test concepts are very similar as all of them depend on the *Native*

[7]https://www.eclipse.org/jdt/

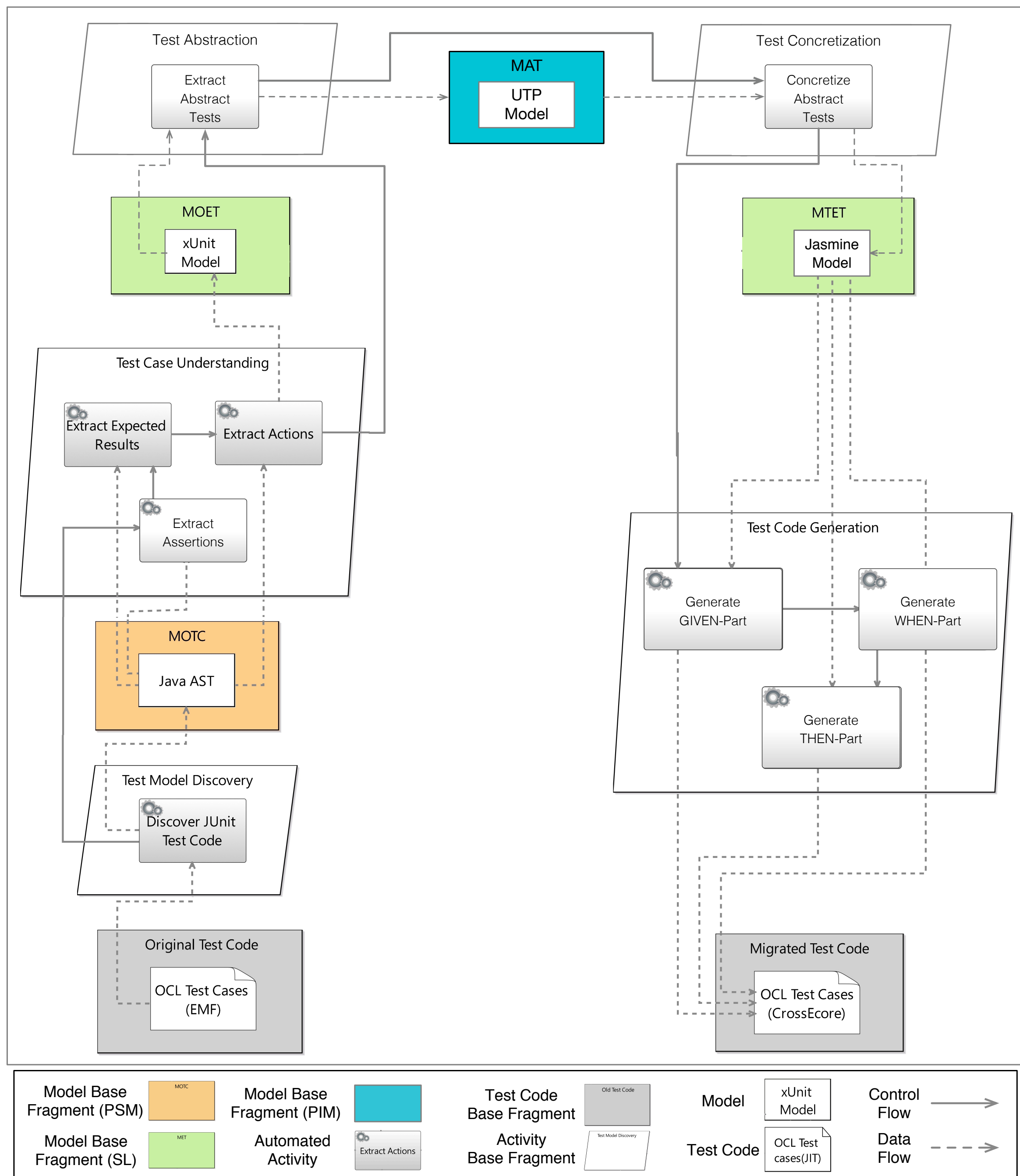


Figure 8.19 Horseshoe Model

OCL Expression system concept, the same transformation pattern was selected for all of them. Another reason for the selection of this pattern was that the target test framework namely the Jasmine testing framework is architecturally different compared to the JUnit framework. In case of a direct mapping between the testing frameworks, i.e., a *Test Language-based Transformation*, would mean that more logic had to be put in the transformation

rules. Therefore, after the *Model of the Executable Tests (MOET)*, an abstraction activity is performed, i.e., *Extraction of Abstract Tests*. The result of this activity is a *Model of Abstract Tests (MAT)* or more concretely a model conforms to the UML Testing Profile. This model is then concretized into *Model of Transformed Executable Tests (MTET)* which in this case is a model conform to the Jasmine metamodel which follows the BDD principle. Namely, this model contains the structural parts *Given*, *When* and *Then*. Finally, as part of the *Test Code Generation* fragment, three different automated generation activities (*Generate GIVEN-Part*, *Generate WHEN-Part*, and *Generate THEN-Part*) are instantiated for each of the three test concepts. Namely, by combining the result of each of these activities, which rely also on the existing transformation rules for the OCL expressions from the system migration, the *Migrated Test Code*. More specifically, the OCL test cases for the CrossEcore implementation of OCL in TypeScript are obtained.

Tool Implementation

Having the transformation method specification developed, the required tools were developed. For the second feasibility study, the same architecture as for the first one was used, which is shown in Figure 8.7 on page 199. The *Input Module* with the *Test Code Parser* and the *Test Model Discoverer* was completely reused.

The extraction of the abstract tests was done as part of the *Transformation Module*. Namely, the *Transformation Rules* component was extended with the extraction rules which were executed by the *Model Transformator*. The *Test Metamodel* used here was based on the UML Testing Profile. Furthermore, the rules for the concretization of the model of abstract tests were added to the *Transformation Rules* component. The components in the *Output Module* have been exchanged with the project-specific ones. Namely, we developed a *Test Metamodel* for the Jasmine test cases based on the BDD profile presented in [LMP10] and we also defined *Test Code Templates*. Based on these templates, the *Test Code Generator* generated the Jasmine test code, i.e., the Jasmine test cases.

Transformation

Using the implemented tools, the actual test transformation of the original test cases into the target test environment was performed. Similarly to the first feasibility study, 12 different test suites with around 4000 test cases have been migrated. In total, about 3700 test cases were generated by the toolchain, thus achieving an automation rate of over 90%.

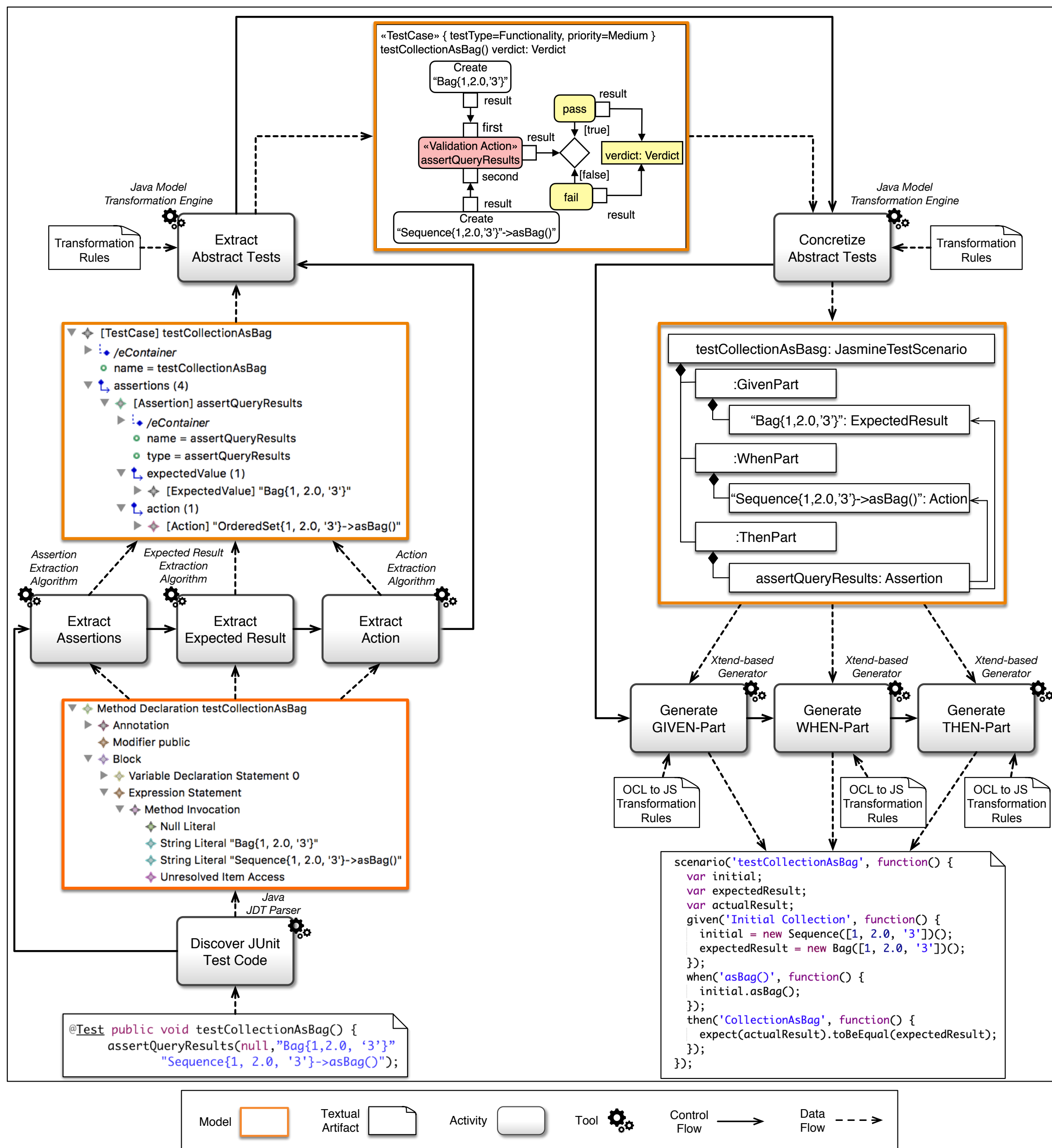


Figure 8.20 Enacting a transformation method to transform the test concepts

Figure 8.20 shows the enactment of the transformation method specified in Figure 8.19. More specifically, it shows an excerpt from the models and the entities which are arising when converting the test concepts *Assertion*, *Expected Result* and *Action*.

Firstly, as shown in the lower left of Figure 8.20, the original test cases written in JUnit were parsed in order to obtain the *Model of the Test Code*. The concrete example shows the JUnit test case named `testCollectionAsBag`. This test case contains a single

assertion function, namely `assertQueryResults` which has three input parameters. The first parameter is irrelevant for this consideration as it was used as a context variable in the source environment which is irrelevant in the target environment. The second and the third parameter, expected result and the action to be performed, respectively, are native OCL expressions specified as strings. As an outcome of the parsing activity, a Java abstract syntax tree is obtained. This abstract syntax tree is further the input for the next activity which extracts xUnit test-specific constructs like `TestCase`, `ExpectedValue` or `Assertion`. The outcome of this activity is the platform-specific *Model of the Executable Test Cases*, shown in the top left corner of Figure 8.20.

The next activity (*Extract Abstract Tests*) takes this model as input and transforms it into an abstract model of test cases which conforms to UTP (UML Testing Profile). This metamodel contains an abstract description of the test cases which is platform-independent and can be transformed into any concrete test model. Further, by applying the *Concretize Abstract Tests*, a platform-specific model is obtained, in this case, it is the model of the Jasmine test cases. This model contains the three structural elements specific for the testing framework following the BDD principle, namely, *Given*, *When*, and *Then*.

The next three activities (*Generate GIVEN-Part*, *Generate WHEN-Part*, and *Generate THEN-Part*), support the generation of test code out of the three main elements of Jasmine's *Model of Executable Tests*. It is important to mention, that as all of these elements contain OCL expressions which were concretized by using the CrossEcore's OCL to TypeScript transformation were used in the system migration.

This feasibility study demonstrated the benefit of our approach for supporting the co-migration of test cases. By using our method engineering process which includes co-evolution analysis and is supported by a modeling tool, we were able to develop a situation-specific test migration method and to migrate the original test cases to the target testing framework, namely Jasmine. By this feasibility study, we have shown that even in the cases of architecturally different testing frameworks, a suitable migration method can be developed and enacted with the help of our framework. So, we have basically addressed two important differences at once, one from a system perspective and one from a testing perspective. Namely, the conceptual transformation transforms the test cases regarding the difference of the OCL implementation, i.e., the EMF's "Ahead-of-Time" style versus the CrossEcore's "Just-in-Time" style. Furthermore, the architectural differences of the source and the target testing frameworks, JUnit and Jasmine, respectively, are also addressed by the test migration method.

8.4 Discussion

In this section, we discuss and answer the evaluation questions introduced in Section 8.1 based on our experiences made as part of the feasibility study.

EQ1: Does the solution approach support the context-specific quality assessment of test cases in a systematic way?

Our experiences gathered during the feasibility studies have shown that our solution approach regarding the quality assessment of test cases provides support in terms of the systematic approach called Test Case Quality Plan (TCQP) [JNES18]. The whole process is guided by the so-called TCQP Process with tool support in terms of the TCQEval web application. This approach considers the context of the test cases, which is important to get relevant insight into the quality of the test cases. Also, it takes into consideration the goals of all stakeholders, and refines them into questions, which are further mapped to quality attributes which can be measured. The tool also supports the analysis of the quality assessment results by providing a dashboard-like interface that visualizes the assessment results.

Nevertheless, a limitation of our quality assessment approach is that it does not provide an existing measurement tool infrastructure, but it rather relies on existing tools for test case quality. The results of these tools have to be gathered and manually inserted into the TCQEval tool for further analysis which means more effort and higher chance for error. However, this is an important aspect that should be addressed as part of feature work, as sometimes certain quality attributes selected by the stakeholders may be of great importance for the overall quality score. For example, if the fault-revealing capability of the test cases is a selected quality focus, a mutation testing is typically applied. For Java or C# there are well-known tools that could be used out of the box, but not all languages are supported. In that case, a tool needs to be implemented, otherwise, this quality aspect cannot be assessed.

EQ2: Does the solution support co-evolution analysis (i.e., change detection, impact analysis, and change propagation) between a system being migrated and its test cases?

Based on the experiences we got in the feasibility studies, we can say that our approach supports the co-evolution analysis. Namely, as part of the situational context identification activity of the migration phase, there is a dedicated activity for the co-evolution analysis which relies on concept-modeling [JYSE20a]. In that way, test cases can be modeled as a set of concepts representing their structure, for both the source and target environment as their realization may differ. Similarly, the functionality of the system, the original and migrated

one, is represented as a set of concepts. By using the impact model, one can express the dependencies between the system and the test concepts, thus encoding basically the impact the system changes can potentially have on the test cases. As the next steps are based on this model, this dependency is propagated until the end, i.e., it is part of the test transformation method that is enacted to migrate the test cases.

However, even though it works in this way, it is a manual activity performed by the migration and test experts and does not use the dependencies that exist between the system and test code. A future improvement would definitely be to automate this process or part of it. For example, by analyzing the dependencies that exist between test suite or test cases and the system under test, one can get recommendations on how to model the relations, i.e., the dependencies between the test and the system concepts.

EQ3: Does the solution approach enables automated transformation, i.e., migration of the test cases whenever possible?

Based on our experiences with migrating the test cases for both feasibility studies, we conclude that our solution approach enables the automated transformation whenever it is possible. This conclusion is based on two observations. First, our approach is based on the principles of model-driven architecture and model-driven software migration. This suggests that artifacts on different levels of abstraction are used as well as activities that enable the transitions between the different levels of abstraction in both directions, namely abstraction and concretization. These transitions are based on model transformation thus enabling automated transformation overall of the test cases. Second, the structure of our method base is built in that way that it supports the three basic activities of the reengineering process, namely reverse engineering, restructuring, and forward engineering [JEAS18]. The method base contains method fragments (artifacts and activities) that are on different levels of abstraction and also method patterns which are typical strategies to perform among the others also automated transformation of the test cases.

Nevertheless, not all abstraction levels are covered by the method fragments currently defined in the method base. For example, the behavioral model of the system which is actually the starting point for the creation of the abstract model of the test cases is not part of the method base. The same applies to the corresponding activities in both direction, abstraction and concretization.

EQ4: Does the solution support the construction of situation-specific transformation methods for test cases?

Based on the experience with the transformation of these test cases, we conclude that the TeCoMi Framework supports the construction of situation-specific transformation methods [JYG+19]. As we have already defined in Section 6.3.1, a method is said to be situation-specific if it is effective and efficient. Regarding the migration of test cases, effectiveness is related to the properties of the migrated test cases, whereas the efficiency related to the enacted migration process. To check the effectiveness of the applied method, we need to analyze whether we have migrated the test cases as intended. Our intention was in both of the feasibility studies to migrate the test cases to the same target framework but in different languages and to test the OCL implementation. Together with the migration expert, we executed the test cases in both source and target environment, test suite by test suite, and we could observe that they were checking the intended functionality. Namely, it could be observed that the test cases were transformed in the same way as the implementation of OCL was transformed into the target environment. Nevertheless, there were still some limitations of the approach when it comes to dealing with regression tests which is hard to be properly adapted to the target environment due to the lack of structure inside them. Regarding efficiency, we would need to compare all transformation methods in order to determine whether the selected transformation method was the most efficient one. A meaningful comparison to that extent was not performed as part of the thesis. However, in discussion with the migration and test experts, we conclude that the migration of the test cases was efficient. Even though the number of test cases was not that high (roughly 4000 test cases), in their opinion, reimplementing that amount of test cases, with embedded OCL expressions, would have resulted in a lot of time and effort. Moreover, debugging and error identification would have been more difficult.

EQ5: Does the solution approach support the creation of a validation method for the validation of the test case migration?

Based on the experiences we got in the feasibility studies, we can say that our approach supports the validation of the test case migration. It is a novel approach based on mutation analysis [JNY+20] that detects false positives and false negatives among the migrated test cases. The solution approach supports first of all the creation of a suitable mutation method. In order to ease the process of applying mutation analysis, which is powerful, but at the same time a very complex technique, we also provide a set of different mutation analysis scenarios that explain the interpretation of the results. Furthermore, a tool infrastructure is provided, which is project-independent when it comes to the specification of mutation operators and can be reused for different projects. However, a full study on this technique was still not performed as it is a topic on its own and we left it for future work. Nevertheless, even in a limited setting, it has shown quite a good potential for validating test case migration.

8.5 Summary

In this chapter, we described two feasibility studies for which we applied the TeCoMi framework. We migrated test cases from one source platform to two different platforms. Firstly, in Section 8.1, we introduced the evaluation questions upon which we discussed the feasibility studies.

Then, we described the feasibility studies in Sections 8.2 and 8.3. The first feasibility study was dealing with the co-migration of test cases from Java to C# whereas the second feasibility study was dealing with the co-migration of test cases from Java to TypeScript. We described for both studies the actual application of the TeCoMi framework.

We introduced and discussed the main artifacts and findings of each activity of all three phases. Additionally, we described in detail the transformation of selected test suites. Finally, in Section 8.4, based on the experiences made, the evaluation questions were discussed. In summary, we concluded that the TeCoMi framework enables an end-to-end co-migration of test cases and it is not limited to a specific context.

Chapter 9

Conclusion and Future Work

In the previous chapters, the TeCoMi framework has been defined and its application in practice has been demonstrated. In this chapter, we conclude the main findings of this thesis and discuss future work. In Section 9.1, we describe the main contributions of this thesis. In Section 9.2, we discuss in which way the TeCoMi framework fulfills the requirements that have been identified in Section 3.2. Finally, we give an overview of future work in Section 9.3.

9.1 Summary of Contributions

The development of situation-specific test co-migration methods is a challenging but very important part of a software migration project.

Firstly, even before the actual migration of the test cases, a quality assessment of the existing test cases is necessary to check whether it is beneficial to perform the migration. Then, a suitable test case migration method should be developed and it should be guided. As the test cases depend on the system that is being migrated, the method development should incorporate the co-evolution analysis in order to enable a proper migration of the test cases. Last but not least, the migrated test cases have to be validated whether they have been correctly migrated.

In this thesis, we addressed this problem by defining a framework called TeCoMi (Test Co-Migration) which supports the three phases. The framework supports the (i) pre-migration assessment of the test case quality, (ii) the development of model-driven situation-specific test case co-migration methods, and (iii) the post-migration validation of the migrated test cases. In the following, we discuss these three contributions of this thesis.

Pre-Migration Phase: Test Case Quality Evaluation

The first phase of the approach supports the quality evaluation of test cases. We introduced the *Test Case Quality Plan (TCQP)* approach which enables the creation of context-specific quality plans for the quality evaluation of test cases. A quality plan serves as a guideline for the quality evaluation of test cases and emphasizes the context of use of test cases as a major factor of influence for the whole quality evaluation. *TCQP* builds upon the *Model Quality Plan (MQP)* approach [VE08] which is relevant in the domain of software models. In this course, we developed a top-down process, called *TCQP Process* and a related metamodel, called *TCQP metamodel*. The *TCQP Process* guides the creation of quality plans by considering the context of use of the test cases. The *TCQP Metamodel* contains all relevant information for a quality plan and it is structured into packages that are linked to their respective activities in the *TCQP Process*. The creation of quality plans as well as the visualization of the evaluation results is supported by a web application called *TCQEval (Test Case Quality Evaluator)*.

Migration Phase: Method Construction considering Co-Evolution Analysis and Method Enactment

The second phase of the approach supports the method construction considering co-evolution analysis as well as method enactment of test case co-migration methods. Our approach builds upon the *MEFiSTo* approach [Gri16] which is relevant in the domain of software modernization.

We introduced a method base that contains test-specific fragments and patterns. The method base is a repository that contains reusable building blocks needed for assembling test migration methods. The two main constituents of the method base are test-specific method fragments and method patterns. Method fragments are atomic building blocks of a test migration method, i.e., activities, artifacts, tools or roles. The method fragments that we proposed, enable the specification of the actual transformation of test cases. They also enable expressing different test transformation strategies, e.g., an automated test conversion based on tools. Method patterns are proven migration strategies and indicate which fragments are necessary for a method that follows a particular strategy and how to assemble them. We enable for each pattern a fine-granular adaptation to the situation at hand. The suitability of each pattern to a certain situation is expressed by a set of characteristics. Besides the test method patterns, the method base also contains co-migration patterns. The co-migration patterns express the relation between the system and the test case migration on a level of fragments, thus supporting the co-evolution analysis.

We introduced a method engineering process that guides the modular construction of situation-specific test case co-migration methods. We described the purpose of each activity

within the process supported by detailed examples. Also, we provided an extension to the MEFiSTo intermediate language called xMIML (Extended MEFiSTo Intermediate Modeling Language) that also covers test relevant aspects. More specifically, all parts of the existing language were extended to support test concepts and we additionally provided means to express the dependencies between the system and test concepts. The method engineering process has been designed to support test case co-migration scenarios. Therefore, we focused the process around establishing the relation between identified system concepts and test concepts by using the technique of concept modeling. A concept represents either systems' or tests' functionality on a higher level of abstraction. Having the system and test concepts represented, the dependencies can be identified and modeled, thus enabling the following steps regarding the test transformation method construction. Based on an assessment of the situation, experts can decide which test method pattern to apply. To support the decision-making process of the experts, the process defines the identification of the situational context and documentation of the influence factors.

Post-Migration Phase: Migration Validation

The last phase supports the validation of the migrated test cases. We introduced a novel validation method for the test case migration which relies on mutation analysis. The validation method checks whether the test cases are migrated without changing their behavior, i.e., without changing what they test. In other words, the main goal of the migration validation is to identify false positives and false negatives among the migrated test cases. We introduced a mutation analysis repository to provide guidelines in terms of scenarios of mutation analysis and patterns for the specification of test case mutation methods. The repository contains mutation analysis scenarios, mutation method patterns, and mutation operators. Six mutation analysis scenarios are identified depending on what is mutated: the original or the migrated system, the original or the migrated test cases, or the system or test cases migration. We defined for each usage scenario, a set of assumptions which must hold for a particular scenario to be feasible, e.g., that an appropriate mutation framework exists. We provided also an in-depth analysis of indications that can be obtained in terms of "bad smells" for the test case migration, i.e., problematic test cases. Finally, we provided a discussion of the suitability of each usage scenario. We introduced mutation patterns which are the technical implementations of the given scenarios. The mutation patterns are defined on a different level of abstraction and their suitability depends on the test case and system migration context. Finally, we proposed a set of mutation operators that can be language-specific, test framework-specific or domain-specific.

Evaluation

To evaluate the feasibility in practice, we applied the TeCoMi framework in an industrial context. More specifically, we performed two feasibility studies and discussed the outcomes. The first feasibility study was dealing with the co-migration of OCL test cases from the EMF framework to the CrossEcore framework. Namely, the EMF's OCL test cases written in the JUnit framework and in *Just-in-Time* manner were transformed into the CrossEcore's OCL test cases written in the MSUnit framework and in *Ahead-of-Time* manner. By this study, we were able to show that the TeCoMi framework can be used to co-migrate test cases in a project context. In the second feasibility study, we co-migrated the same OCL test cases to a different platform, namely the TypeScript implementation of CrossEcore. The test target framework was Jasmine, which implements test cases in BDD style which is structurally different from the representational style of JUnit. In this study, we were able to show that the TeCoMi framework was able to address this changed context.

9.2 Requirements Revisited

In Section 3.2, we defined a set of requirements that an end-to-end solution approach needs to fulfill in order to enable the development of test transformation methods in the context of test case co-migration. Subsequently, we describe how our solution framework fulfills these requirements.

Pre-Migration Phase: Test Case Quality Evaluation

The general requirement for this phase (R1) states that the solution approach should provide a means to perform a quality evaluation of test cases. More specifically, the solution should use definitions for qualities of test cases for a consistent and common quality understanding (R1.1). Also, the solution should be able to provide a minimum set of context factors (R1.2) and it should distinguish between objective and subjective measurements (R1.3). Finally, the quality evaluation of test cases should be a systematic process that guides the stakeholders based on the context factors (R1.4).

The TeCoMi framework provides the Test Case Quality Plan (TCQP) approach for specifying test case quality plans for the evaluation of test case quality. The TCQP approach relies on a standardized quality model, namely ISO 25010 standard [ISO11b] to provide a common quality understanding (R1.1). The TCQP metamodel provides a context model that contains a set of context factors needed to describe the context of use of test cases (R1.2). Furthermore, the TCQP metamodel provides a measurement metamodel that relies on the ISO/IEC 15939 standard [ISO02] and the measurements are classified into subjective

and objective measurements (R1.3). The whole process is guided by the TCQP process of creating context-specific quality plans for the quality evaluation of test cases (R1.4).

Migration Phase: Method Construction considering Co-Evolution Analysis and Method Enactment

The general requirements for this phase (R2 - R4) state that the solution approach should provide means to perform a co-evolution analysis (R2), it should enable automated transformation when possible (R3), and finally, it should enable high flexibility to different migration scenarios (R4). The requirement regarding the co-evolution (R2), states the change detection (R2.1), impact analysis (R2.2), and change propagation (R2.3) regarding the system migration should be performed on a conceptual level. The requirement regarding the automated transformation (R3) states that the three main reengineering activities, namely reverse engineering (R3.1), restructuring (R3.2), and forward engineering (R3.3) should be supported. Finally, the requirement regarding situativity (R4) states that the method base should provide test-specific method fragments and method patterns (R4.1), the method engineering process should support method development and method enactment of test transformation methods (R4.2), and it should comprise co-evolution analysis (R4.3).

The TeCoMi framework provides a method engineering process that addresses the co-migration of test cases. Namely, as part of the process, the system and test concepts are modeled, by applying the concept modeling technique. Firstly, the different realizations of the source and target system concepts are explicitly related so that the change is obvious (R2.1). Then, the dependencies between the test and system concepts are identified, on which basis the impact of the system changes can be explored (R2.2). The identified dependencies are part of the situational model, which is input for the activity dealing the test transformation method construction which ensures that the identified impact will be propagated to the test cases in the end (R2.3).

The TeCoMi framework provides a method base that contains method fragments that we represented in terms of a test case reengineering horseshoe model. The method fragments belong to one of the reengineering activities *Reverse engineering*, *Restructuring*, and *Forward Engineering* and are placed on different levels of abstraction. Regarding reverse engineering, some activities support an initial extraction of a test model out of the existing test code, with the help of text-to-model transformation. Then, by applying model-to-model transformations, a model on a higher level of abstraction, i.e., a model of executable tests is obtained (R3.1). Furthermore, there are activities dealing with the architectural restructuring of the obtained model of abstract tests as well as diverse enrichment activities on different levels of abstraction (R3.2). The method base also contains concretization activities, i.e., model-to-model transformations to transform models from higher to lower abstraction level.

Lastly, as part of the forward engineering, there is an activity dealing with the test case code generation (R3.3).

The TeCoMi framework provides a method base that contains test-specific method fragments and patterns. Namely, as already explained when discussing the previous requirement, there are test method fragments that address test-specific artifacts (e.g., *Model of Executable Tests*) and reengineering activities (e.g., *Test Case Understanding*) represented in terms of test case reengineering horseshoe model. Furthermore, a set of test method patterns (e.g., *Test Language-based Test Transformation* or *Conceptual Test Transformation*) is provided, that is based on this horseshoe model (R4.1). The TeCoMi framework provides a method engineering process that comprises activities from the two main disciplines, method development and method enactment (R4.2). By performing activities of the method development discipline, a situation-specific method gets developed. During the first activity of method development, namely situational context identification, co-evolution analysis is performed to identify the impact that the system changes have on the test cases (R4.3). As part of the method enactment, the situation-specific tools are developed that are required for the automation of the migration method and in the end, the developed method is enacted.

Post-Migration Phase: Migration Validation

The general requirement for this phase (R5) states that the solution approach should indicate the success of the test case migration. More specifically, the solution approach must be a systematic process that guides the stakeholders in a co-migration setting to perform a validation process (R5.1). Then, the validation process shall be automated in order to deal with a high number of test cases (R5.2). Finally, the applicability of the validation approach shall not be limited to a single migration context (R5.3).

The TeCoMi framework provides a validation method for the test case migration. The framework provides a systematic process that guides the creation of a validation method which contains context identification, construction of the validation method, the implementation of the necessary tools and finally, the analysis of the validation results (R5.1). As we have seen, the process contains an activity which deals with the implementation of the tools that automate the validation method (R5.2). Furthermore, we provide a model-driven mutation framework which in turn can be seen as a project-independent tool infrastructure. Finally, as we mentioned, the process starts with the consideration of the context, making the creation of the validation method specific to the context at hand, which actually makes our approach general and applicable in any context (R5.3).

9.3 Future Work

In this thesis, we presented an end-to-end approach for the co-migration of test cases. Apart from its current features, certain aspects can be addressed in future work to further advance the solution approach. In this section, based on the phases of our approach, we discuss possible extensions and ideas for follow-up research.

Pre-Migration Phase: Test Case Quality Evaluation

Our approach aims to support the first phase of the quality evaluation of test cases. The context characterization and the creation of the quality plans are supported by the *TCQP (Test Case Quality Plan)* approach and by the *TCQEval (Test Case Quality Evaluator)* tool. However, such support is still missing for the measurement tool implementation and execution and decision-making activities. Namely, first of all, a flexible component-based measurement tool infrastructure would bring a lot of improvement to the overall process. It should enable a flexible plug-and-play mechanism for the different existing tools as well as the inclusion of newly developed project-specific tools. Currently, the results of different tools have to be gathered and manually inserted into the *TCQEval* tool for further analysis which can result in more effort and higher chance for error. Therefore, an automated or at least a semi-automated import of the evaluation results would improve the quality of the evaluation process. As any tool has its export format of the results, the *TCQEval* shall support the import of results by providing a dedicated flexible import module. This module shall enable the specification of mappings between the report structure of the external tool and the test case quality plan's structure.

Currently, the quality evaluation approach focuses on assessing the quality of the test cases, but not on the eventual improvement. However, the quality report provides a clear insight into the different quality aspects on different levels of granularity (quality characteristics, quality sub-characteristics, and quality attributes). Having this information, the test cases could be improved regarding the lower quality scores they have. For example, if the fault-revealing capability of the test cases is not high enough, a mutation testing can be applied. For Java or C# there are well-known tools that could be used out of the box, but not all languages are supported. However, we already provided a model-driven mutation framework in the post-migration phase, that can be applied in any context as it performs the mutation testing on a higher level of abstraction.

Migration Phase: Method Construction considering Co-Evolution Analysis and Method Enactment

Currently, the test case horseshoe reengineering model, which represents the method fragments in the method base, contains fragments until the model of abstract tests on the

platform-independent layer. However, not all artifacts and activities are covered by the method fragments currently defined in the method base as this was out of the scope of this work. For example, a behavioral model of the system which is actually the starting point for the creation of the abstract model of the test cases is not part of the method base. The same applies to the corresponding activities which involve this artifact, namely abstraction and concretization. Existing standardized languages like the OMG's UML Testing Profile (UTP) [OMG13b] or the ETSI's Test Description Language (TDL) [ETS16] can be used for the representation of behavioral models. The benefit of having the behavioral model is twofold, for the test migration as well as for the system migration. Seen from a testing perspective, it would enable migration to model-based or even model-driven testing in the target environment. The extracted behavioral model can be analyzed and when needed modified by the system and text experts. Seen from a system perspective, it can provide information on the actual system implementation in the source environment. This strategy is already used in some migration projects to get insight into the structure of legacy systems.

At the current state, the co-migration patterns are used to support the creation of test method patterns. They express explicitly the relation between the system and the test case migration on a level of fragments thus supporting the co-evolution analysis. However, currently, the configuration of the instantiated test migration methods should be done manually in the Sirius modeling editor we have provided. At this point, we envision supported configuration of those test fragments which have corresponding system fragments according to the co-migration patterns. Even just a fragment-completion (based on the well-established code-completion term), could speed-up the configuration of test transformation methods.

The concept modeling is a manual activity, performed by the migration and test experts. After the initial modeling of test and system concepts, during the impact analysis, correspondences and dependencies are identified and modeled, also manually. This requires a lot of knowledge from the experts and could be error-prone. On the other hand, the current process does not use the dependencies that exist between the system and the test code. Incorporation of the code-dependencies into the impact-analysis part would be a great improvement. For example, by analyzing the dependencies that exist between test suite or test cases and the system under test, the experts can get recommendations on how to model the relations, i.e., the dependencies between the test and the system concepts.

By the evaluation performed as part of this thesis, we were able to demonstrate that it is feasible to apply the TeCoMi framework in practice to co-migrate test cases (Section 8.4). However, it is still an open task to evaluate other characteristics (e.g., effectiveness, efficiency or usability) of the framework that are relevant for its use in practice. For example, it needs to be assessed whether the framework could be used to transform legacy test cases, for example,

written in COBOL, to a modern platform. Furthermore, it needs to be determined whether it could be used to transform an even larger number of test cases consisting of millions of lines of code.

Post-Migration Phase: Migration Validation

The application of mutation analysis as a technique for the migration validation has shown promising results (Section 8.2.3). However, the specification of the mutation methods requires still a lot of manual activities. This can be significantly improved if more knowledge from the previous phases is used for automation. For example, the situational context identified in the first two phases can be used to make suggestions for the optimal mutation scenario and mutation patterns. To address to some extent the automation problem, we introduced the model-driven mutation framework (Section 7.3.3). The framework addresses the most demanding part of the mutation analysis part, namely the automated generation of mutants. However, the test case execution and identification of bad smells are performed by the testers. By further extending the mutation framework with the automated execution and identification of problematic cases would largely improve the efficiency of the validation method.

The evaluation results presented in this work are obtained by applying the validation approach to a single migration project for two different scenarios as proof of concept. Therefore, more empirical analysis is needed, i.e., more different migration projects have to be validated by applying our validation approach. Due to the importance of validation of test case migration, we see further research of this topic as a follow-up work which is described in the following.

Follow-up Work: Optimal Selection of Mutation Operators in Test Case Migration Validation

Mutation analysis or mutation testing is a strategy that besides the basic usage of assessing and improving test effectivity, can be used to identify false positives and false negatives in test case migration. The main factor that defines the success of the mutation analysis is the selection of mutation operators. This depends on a lot of factors in a migration context, from both test and system migration perspectives. First of all, it depends on what is being migrated and what kind of transformation has been performed. Then, on what abstraction level the system was migrated, on what level the test cases have been migrated. In our work, we distinguish between mutation operators that can be either language-specific, test framework-specific or domain-specific. However, at this point, a big part of the decision regarding the selection and implementation of test mutation operators is done by the testing expert and the tester. Therefore, more intensive empirical research is needed on the optimal

selection of mutation operators in order to improve the mutation's efficiency. For example, machine learning algorithms can be used in order to come up with mutation operators an optimal set of mutation operators for a particular migration context.

Follow-up Work: Test Case Modernization

Proprietary testing frameworks are still common nowadays. The test cases written in such frameworks can be difficult to maintain or reuse. Therefore, a migration to a newer, standard-based framework, or other testing methodologies like BDD (Behavioral Driven Development) or MBT (Model-Based Testing), can be beneficial from both practical and economical reasons. With our framework, we established the basis for transformation and migration of test cases in any setting and it can serve as a good basis to address the problem of test case modernization. However, there are still some open questions that need to be additionally addressed. Namely, seen from a constructive perspective, the test cases have to be properly adapted to the targeting testing framework to fully benefit from the migration. By applying concept modeling and considering the target system and test environment, our solution establishes the basis for test case modernization. However, the development of the test transformation method should consider even more the target environment characteristics. Seen from the analytic perspective, after the migration some quality properties (e.g., performance or test effectivity) of the migrated test cases have to be evaluated. This evaluation should assess whether the adaptation to the target framework was properly done.

Follow-up Work: Benchmarking Environment for Testing Frameworks

Nowadays, there is a plethora of testing frameworks for any existing language especially for the most used ones like Java, C#, C, C++, etc. Each testing framework has different characteristics, described through a set of features like the used representational style, support for mocks, handling of exceptions, etc. When selecting the right testing framework for a given project setting, the testing frameworks can be compared against the aforementioned static features. However, when comparing multiple frameworks, the practitioners are interested beyond the static features. Namely, a comparison against the efficiency of the testing frameworks is also required. In that case, unless a set of test cases is executed in a given framework, no conclusions can be drawn, and therefore no comparison is possible. Furthermore, some frameworks support different representational styles like xUnit or BDD. A set of test cases, written in the same framework but in a different representational style, may impact the efficiency of the testing framework. So, multiple factors can influence the efficiency of a testing framework. Therefore, a flexible, model-based (or even model-driven) benchmarking framework is needed.

Follow-up Work: Co-Evolution of Framework-based Applications

The development of an information system is based on the usage of frameworks. When a given framework that is applied in some application evolves, i.e., then it should be replaced with the newer version, e.g., in order to use some new functionality. However, there may be incompatibility which would imply changes in the application. In order to check where exactly these discrepancies are, a compatibility analysis is needed. Based on the outcome of this analysis it can be decided whether to migrate the application or not. At this point, we envision the application of the double horseshoe model we have used in our approach. The difference is, however, that the inside horseshoe represents the evolution i.e., the migration of the framework whereas the outer horseshoe model represents the evolution of the applications. By executing the activities belonging to the reverse engineering phase of both horseshoe models, the framework and the application can be compared on a higher level of abstraction. If a decision to migrate the application is made, then, by subsequently applying the restructuring and forward engineering phases, the application can be migrated to the newer version. The actual migration method, implementing a suitable migration strategy, can be developed and enacted by following the principles of situational method engineering.

Bibliography

[ABD+79] Allen T Acree, Timothy A Budd, Richard A DeMillo, Richard J Lipton, and Frederick G Sayward. Mutation Analysis. Technical report, 09 1979.

[AHH04] Konstantinos Adamopoulos, Mark Harman, and Robert M. Hierons. How to Overcome the Equivalent Mutant Problem and Achieve Tailored Selective Mutation Using Co-evolution. pages 1338–1349. Springer, Berlin, Heidelberg, 2004.

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[BA82] Timothy A. Budd and Dana Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[Bal19] Achyuth Nagaraj Balasubramanian. Co-evolution and validation of test cases in software migration. Master's thesis, Paderborn University, Paderborn, Germany, 2019.

[BCJM10a] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: A generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 173, New York, New York, USA, 2010. ACM Press.

[BCJM10b] Hugo Bruneliere, Jordi Cabot, Frédéric Jouault, and Frédéric Madiot. Modisco: A generic and extensible framework for model driven reverse engineering. In *Proceedings of the IEEE/ACM international conference on Automated software engineering - ASE '10*, page 173, New York, New York, USA, 2010. ACM Press.

[BCR94] Victor R Basili, Gianluigi Caldiera, and Dieter H Rombach. {T}he {G}oal {Q}uestion {M}etric {A}pproach. In *Encyclopedia of Software Engineering*, volume I. John Wiley & Sons, 1994.

[BCW12] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering in Practice*. Morgan & Claypool Publishers, 1st edition, 2012.

[BHP+17] David Bowes, Tracy Hall, Jean Petrić, Thomas Shippey, and Burak Turhan. How Good Are My Tests? *International Workshop on Emerging Trends in Software Metrics, WETSoM*, pages 9–14, 2017.

[BL14] Arnaud Bouzy and Bruno Legeard. From test legacy to model-based testing. URL: https://ucaat.etsi.org/2014/presentations/T2_From_Test_Legacy_to_MBT.pdf, September 2014.

[BLWG99] Jesús Bisbal, Deirdre Lawless, Bing Wu, and Jane Grimson. Legacy information systems: issues and directions. *IEEE Software*, 16(5):103–111, 1999.

[Bri96] Sjaak Brinkkemper. Method Engineering: Engineering of Information Systems Development Methods and Tools. *Information and software technology*, 38(4):275–280, 1996.

[BS79] Douglas Baldwin and Frederick Sayward. Heuristics for Determining Equivalence of Program Mutations. Technical report, 1979.

[CC90] Elliot J Chikofsky and James H Cross. Reverse Engineering and Design Recovery: A Taxonomy. *IEEE Software*, 7(1):13–17, 1990.

[CFH+09] Krzysztof Czarnecki, John Nathan Foster, Zhenjiang Hu, Ralf Lämmel, Andy Schürr, and James Terwilliger. Bidirectional Transformations: A Cross-Discipline Perspective. In Richard F. Paige, editor, *ICMT 2009*, volume 5563 of *Lecture Notes in Computer Science (LNCS)*, page 260–283. Springer, 2009.

[Cha20] Abhishek Hassan Chandrashekar. Quality assessment of test cases in software migration projects. Master's thesis, Paderborn University, Paderborn, Germany, 2020. (work in progress).

[Che01a] Yuri Chernak. Validating and improving test-case effectiveness. *IEEE Softw.*, 18(1):81–86, 2001.

[Che01b] P. Chevalley. Applying mutation analysis for object-oriented programs using a reflective approach. In *Proceedings Eighth Asia-Pacific Software Engineering Conference*, pages 267–270. IEEE Comput. Soc, 2001.

[CT03] Philippe Chevalley and Pascale Thévenod-Fosse. A mutation analysis tool for Java programs. *International Journal on Software Tools for Technology Transfer*, 5(1):90–103, 11 2003.

[Der06] Anna Derezińska. Advanced mutation operators applicable in C# programs. In *Software Engineering Techniques: Design for Quality*, pages 283–288. Springer US, Boston, MA, 2006.

[DJLW09] Florian Deissenboeck, Elmar Juergens, Klaus Lochmann, and Stefan Wagner. Software quality models: Purposes, usage scenarios and requirements. In *2009 ICSE Workshop on Software Quality, WoSQ@ICSE 2009, Vancouver, BC, Canada, May 16, 2009*, pages 9–14. IEEE Computer Society, 2009.

[DLW15] Rahul Dixit, Christof Lutteroth, and Gerald Weber. Formtester: Effective integration of model-based and manually specified test cases. In Antonia Bertolino, Gerardo Canfora, and Sebastian G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 2*, pages 745–748. IEEE Computer Society, 2015.

[DMBK01] Arie Deursen, Leon M.F. Moonen, A. Bergh, and Gerard Kok. Refactoring test code, 2001.

[DS07] A. Derezińska and A. Szustek. Cream - a system for object-oriented mutation of C# programs. *Zeszyty Naukowe Wydziału ETI Politechniki Gdańskiej. Technologie Informacyjne*, T. 13:389–398, 2007.

[DS08] Anna Derezińska and Anna Szustek. Tool-Supported Advanced Mutation Approach for Verification of C# Programs. In *2008 Third International Conference on Dependability of Computer Systems DepCoS-RELCOMEX*, pages 261–268. IEEE, 2008.

[EBI06] Michael Ellims, James Bridges, and Darrel C. Ince. The Economics of Unit Testing. *Empirical Software Engineering*, 11(1):5–31, 3 2006.

[EGL06] Gregor Engels, Baris Güldali, and Marc Lohmann. Towards Model-Driven Unit Testing. In *Models in Software Engineering*. Springer Berlin Heidelberg, 2006.

[ES10] Gregor Engels and Stefan Sauer. Graph transformations and model-driven engineering. chapter A Meta-method for Defining Software Engineering Methods, pages 411–440. Springer-Verlag, Berlin, Heidelberg, 2010.

[ETS05] European Telecommunications Standards Institute ETSI. ETSI Standard ES 201 873-1: The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, V3.1.1, 2005.

[ETS16] European Telecommunications Standards Institute ETSI. ETSI ES 203 119-1: Methods for Testing and Specification (MTS); The Test Description Language (TDL); Part 1: Abstract Syntax and Associated Semantics, v1.3.1, 2016.

[FB99] Martin Fowler and Kent. Beck. *Refactoring : improving the design of existing code*. Addison-Wesley, 1999.

[FB16] Masud Fazal-Baqaie. *Project-specific software engineering methods : composition, enactment, and quality assurance*. PhD thesis, Paderborn University, Paderborn, Germany, 2016.

[FIMR10] Qurat-ul-ann Farooq, Muhammad Zohaib Z. Iqbal, Zafar I. Malik, and Matthias Riebisch. A model-based regression testing approach for evolving software systems with flexible tool support. In Roy Sterritt, Brandon Eames, and Jonathan Sprinkle, editors, *17th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, ECBS 2010, Oxford, England, UK, 22-26 March 2010*, pages 41–49. IEEE Computer Society, 2010.

[FMM14] Amin Milani Fard, Mehdi MirzaAghaei, and Ali Mesbah. Leveraging existing tests in automated test generation for web applications. In Ivica Crnkovic, Marsha Chechik, and Paul Grünbacher, editors, *ACM/IEEE International Conference on Automated Software Engineering, ASE '14, Vasteras, Sweden - September 15 - 19, 2014*, pages 67–78. ACM, 2014.

[FWE+12] Andreas Fuhr, Andreas Winter, Uwe Erdmenger, Tassilo Horn, Uwe Kaiser, Volker Riediger, and Werner Teppe. Model-Driven Software Migration - Process Model, Tool Support and Application. In *Migrating Legacy Applications: Challenges in Service Oriented Architecture and Cloud Computing Environments*, pages 153–184. IGI Global, 2012.

[GF07] Eduardo Martins Guerra and Clovis Torres Fernandes. Refactoring Test Code Safely. In *International Conference on Software Engineering Advances (ICSEA 2007)*, pages 44–44. IEEE, 8 2007.

[GFBEK16] Marvin Grieger, Masud Fazal-Baqaie, Gregor Engels, and Markus Klenke. Concept-based engineering of situation-specific migration methods. In Georgia M. Kapitsaki and Eduardo Santana de Almeida, editors, *Software Reuse: Bridging with Social-Awareness*, pages 199–214. Springer International Publishing, 2016.

[GFEK16] Marvin Grieger, Masud Fazal-Baqaie, Gregor Engels, and Markus Klenke. Concept-based engineering of situation-specific migration methods. In *Software Reuse: Bridging with Social-Awareness - 15th International Conference, ICSR 2016, Limassol, Cyprus, June 5-7, 2016, Proceedings*, pages 199–214, 2016.

[GHR+03] Jens Grabowski, Dieter Hogrefe, György Réthy, Ina Schieferdecker, Anthony Wiles, and Colin Willcock. An introduction to the testing and test control notation (TTCN-3). *Computer Networks*, 42(3):375–403, 6 2003.

[GJG14] Rahul Gopinath, Carlos Jensen, and Alex Groce. Code coverage for suite evaluation by developers. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 72–82, New York, New York, USA, 2014. ACM Press.

[GMS10] B. Güldali, M. Mlynarski, and Y. Sancar. Effort comparison for model-based testing scenarios. In *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 28–36, April 2010.

[Gri16] Marvin Grieger. *Model-Driven Software Modernization: Concept-Based Engineering of Situation-Specific Methods*. PhD thesis, Paderborn University, Paderborn, Germany, 2016.

[GSZ09] Bernhard J. M. Grün, David Schuler, and Andreas Zeller. The Impact of Equivalent Mutants. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 192–199. IEEE, 2009.

[HG77] R.G. Hamlet and R. G. Testing Programs with the Aid of a Compiler. *IEEE Transactions on Software Engineering*, SE-3(4):279–290, 7 1977.

[HL03] Reiko Heckel and Marc Lohmann. Towards model-driven testing. In *Electronic Notes in Theoretical Computer Science*. Elsevier, 2003.

[HMS03] Hardi Hungar, Tiziana Margaria, and Bernhard Steffen. Test-based model generation for legacy systems. pages 971–980, 2003.

[HSRÅR14] Brian Henderson-Sellers, Jolita Ralyté, Pär J. Ågerfalk, and Matti Rossi. Situational method engineering. In *Springer Berlin Heidelberg*, 2014.

[IEE90] IEEE. *IEEE Standard Glossary of Software Engineering Terminology.* 1990.

[IEE91] IEEE Standard Computer Dictionary A Compilation of IEEE Standard Computer Glossaries. 1991.

[ISO01] ISO/IEC. 9126:2001: Software engineering - Product quality - Part 1: Quality model. International Organization for Standardization (ISO) / International Electrotechnical Commision (IEC), 6 2001.

[ISO02] ISO/IEC. 15939:2002: Software engineering - Software measurement process. International Organization for Standardization (ISO) / International Electrotechnical Commision (IEC), 6 2002.

[ISO09] IEEE Recommended Practice for Software Requirements Specifications. 2009.

[ISO11a] ISO/IEC. 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. International Organization for Standardization (ISO) / International Electrotechnical Commision (IEC, 2011.

[ISO11b] ISO/IEC. 25010:2011 - Systems and software engineering – Systems and software Quality Requirements and Evaluation (SQuaRE) – System and software quality models. International Organization for Standardization (ISO) / International Electrotechnical Commision (IEC, 2011.

[ISO11c] International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 25010:2011: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - System and software quality models. 2011.

[ISO11d] International Organization for Standardization (ISO) and International Electrotechnical Commision (IEC) 25010:2011: Systems and software engineering - Systems and software Quality Requirements and Evaluation (SQuaRE) - data quality model. 2011.

[IST] ISTQB Glossary, https://www.astqb.org/glossary/, Accessed: 2020-29-04.

[JABK08] Frédéric Jouault, Freddy Allilaire, Jean Bézivin, and Ivan Kurtev. Atl: A model transformation tool. *Science of Computer Programming*, 72(1-2):31–39, June 2008.

[JEAS18] Ivan Jovanovikj, Gregor Engels, Anthony Anjorin, and Stefan Sauer. Model-driven test case migration: The test case reengineering horseshoe model. In *Information Systems in the Big Data Era - CAiSE Forum 2018, Tallinn, Estonia, June 11-15, 2018, Proceedings*, pages 133–147, 2018.

[JEF14] René Just, Michael D. Ernst, and Gordon Fraser. Efficient mutation analysis by propagating and partitioning infected execution states. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, ISSTA 2014, pages 315–326, New York, NY, USA, 2014. ACM.

[JG18] Ivan Jovanovikj and Baris Güldali. Presentation: Who guards the guards? on the validation of test case migration. User Conference on Advanced Automated Testing (UCAAT 2018), Paris, 2018.

[JGG16] Ivan Jovanovikj, Baris Güldali, and Marvin Grieger. Towards applying model-based testing in test case migration. *Softwaretechnik-Trends*, 36(3), 2016.

[JGGT16] Ivan Jovanovikj, Marvin Grieger, Baris Güldali, and Alexander Teetz. Reengineering of legacy test cases: Problem domain & scenarios. *Softwaretechnik-Trends*, 36(3), 2016.

[JGY16] Ivan Jovanovikj, Marvin Grieger, and Enes Yigitbas. Towards a model-driven method for reusing test cases in software migration projects. *Softwaretechnik-Trends*, 36(2), 2016.

[JH11] Yue Jia and Mark Harman. An Analysis and Survey of the Development of Mutation Testing. *IEEE Trans. Software Eng.*, 37(5):649–678, 2011.

[JJCM99] Sun-Woo Kim John, Sun-Woo Kim John, John A. Clark, and John A. Mcdermid. Assessing Test Set Adequacy for Object-Oriented Programs Using Class Mutation. *Symposium on Software Technology (SoST'99)*, pages 72–83, 1999.

[JKK+09] Antti Jääskeläinen, Antti Kervinen, Mika Katara, Antti Valmari, and Heikki Virtanen. Synthesizing Test Models from Test Cases. In *Hardware and Software: Verification and Testing: 4th International Haifa Verification Conference*, pages 179–193. Springer Berlin Heidelberg, 2009.

[JNES18] Ivan Jovanovikj, Vishwak Narasimhan, Gregor Engels, and Stefan Sauer. Context-specific quality evaluation of test cases. In *Proceedings of the 6th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2018, Funchal, Madeira - Portugal, January 22-24, 2018.*, pages 594–601, 2018.

[JNY+20] Ivan Jovanovikj, Achyuth Nagaraj, Enes Yigitbas, Anthony Anjorin, Stefan Sauer, and Gregor Engels. Validating test case migration via mutation analysis (to appear). In *Proceedings of the 2020 IEEE/ACM 1st International Conference on Automation of Software Test (AST 2020)*, 2020.

[Jov17] Ivan Jovanovikj. Presentation: Framework for constructing context-specific migration methods for test cases. User Conference on Advanced Automated Testing (UCAAT 2017), Paris, 2017.

[JS17] Ivan Jovanovikj and Stefan Sauer. Towards a framework for constructing context-specific migration methods for test cases. *Softwaretechnik-Trends*, 37(2), 2017.

[JSW07] Abu Zafer Javed, Paul A. Strooper, and Geoffrey Watson. Automated generation of test cases using model-driven architecture. In Hong Zhu, W. Eric Wong, and Amit M. Paradkar, editors, *Proceedings of the Second International Workshop on Automation of Software Test, AST 2007*, pages 3–9. IEEE Computer Society, 2007.

[JUn] JUnit. http://junit.org/junit4/.

[JWY+20] Ivan Jovanovikj, Nils Weidmann, Enes Yigitbas, Anthony Anjorin, Stefan Sauer, and Gregor Engels. Model-driven mutation framework for validation of test case migration (to appear). In *Proceedings of the 2020 International Conference on Systems Modelling and Management (ICSMM 2020)*, 2020.

[JYAS18] Ivan Jovanovikj, Enes Yigitbas, Anthony Anjorin, and Stefan Sauer. Who guards the guards? on the validation of test case migration. *Softwaretechnik-Trends, Proceedings of the 20th Workshop Software-Reengineering & Evolution (WSRE) & 9th Workshop Design for Future (DFF)*, 2018.

[JYG+19] Ivan Jovanovikj, Enes Yigitbas, Marvin Grieger, Stefan Sauer, and Gregor Engels. Modular construction of context-specific test case migration methods. In *Proceedings of the 7th International Conference on Model-Driven Engineering and Software Development, MODELSWARD 2019, Prague, Czech Republic, February 20-22, 2019.*, pages 534–541, 2019.

[JYS18] Ivan Jovanovikj, Enes Yigitbas, and Stefan Sauer. Test case migration: A reference process model and its instantiation in an industrial context. In *Joint Proceedings of the Workshops at Modellierung 2018 co-located with Modellierung 2018, Braunschweig, Germany, February 21, 2018.*, pages 153–162, 2018.

[JYSE20a] Ivan Jovanovikj, Enes Yigitbas, Stefan Sauer, and Gregor Engels. Concept-based co-migration of test cases (to appear). In *Proceedings of the 8th International Conference on Model-Driven Engineering and Software Development - Volume 1: MODELSWARD*, 2020.

[JYSE20b] Ivan Jovanovikj, Enes Yigitbas, Stefan Sauer, and Gregor Engels. Test case co-migration method patterns (to appear). *Software Engineering 2020 Workshopband*, 2020.

[Kan03] Cem Kaner. What Is a Good Test Case? *Software Testing Analysis & Review Conference (STAR East)*, 2003.

[KCM99] Sunwoo Kim, John A. Clark, and John A. McDermid. The rigorous generation of java mutation operators using hazop. 1999.

[KCM01] Sunwoo Kim, John A. Clark, and John A. McDermid. Investigating the Effectiveness of Object-Oriented Strategies with the Mutation Method. In *Mutation Testing for the New Century*, pages 4–4. Springer US, Boston, MA, 2001.

[KKCM00] Sunwoo Kim, Sunwoo Kim, John A. Clark, and John A. McDermid. Class Mutation: Mutation Testing for Object-Oriented Programs. *PROC. NET.OBJECTDAYS*, pages 9–12, 2000.

[KNE92] Wojtek Kozaczynski, Jim Ning, and Andre Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 1992.

[KWC98] Rick Kazman, Steven Woods, and Jeromy S. Carrière. Requirements for integrating software architecture and reengineering models: CORUM II. In *Proceedings Fifth Working Conference on Reverse Engineering*. IEEE Comput. Soc, 1998.

[LMP10] Ioan Lazăr, Simona Motogna, and Bazil Pârv. Behaviour-driven development of foundational uml components. *Electronic Notes in Theoretical Computer Science*, 264(1):91 – 105, 2010. Proceedings of the 7th International Workshop on Formal Engineering approaches to Software Components and Architectures (FESCA 2010).

[LS78] Richard J. Lipton and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[MA14] Andreas Menychtas and Et Al. Software modernization and cloudification using the ARTIST migration methodology and framework. *Scalable Computing: Practice and Experience*, 15(2):131–152, 7 2014.

[MBLT06] Jean-Marie Mottu, Benoit Baudry, and Yves Le Traon. Mutation Analysis Testing for Model Transformations. pages 376–390. Springer, Berlin, Heidelberg, 2006.

[MD08] Tom Mens and Serge Demeyer. *Software Evolution*. Springer Publishing Company, Incorporated, 1 edition, 2008.

[Mes07] Gerard. Meszaros. *XUnit test patterns : refactoring test code*. Addison-Wesley, 2007.

[MGN09] Philip Makedonski, Jens Grabowski, and Helmut Neukirchen. Validating the Behavioral Equivalence of TTCN-3 Test Cases. In *2009 First International Conference on Advances in System Testing and Validation Lifecycle*, pages 117–122. IEEE, 9 2009.

[Mic] Microsoft Unit Test Framework. https://msdn.microsoft.com/en-us/library/hh598960.aspx.

[MKA+14] Andreas Menychtas, Kleopatra Konstanteli, Juncal Alonso, Leire Orue-Echevarria, Jesus Gorronogoitia, George Kousiouris, Christina Santzaridou, Hugo Bruneliere, Bram Pellens, Peter Stuer, Oliver Strauss, Tatiana Senkova, and Theodora Varvarigou. Software modernization and cloudification using the ARTIST migration methodology and framework. *Scalable Computing: Practice and Experience*, 15(2):131–152, 7 2014.

[MNDT09] Audris Mockus, Nachiappan Nagappan, and Trung T. Dinh-Trong. Test coverage and post-verification defects: A multiple case study. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pages 291–301. IEEE, 10 2009.

[Moh10] Parastoo Mohagheghi. Reuse and Migration of Legacy Systems to Interoperable Cloud Services- The REMICS project. In *Mda4ServiceCloud 2010*, pages 1–13. Springer, Berlin, Heidelberg, 12 2010.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. MuJava: an automated class mutation system: Research Articles. *Software Testing, Verification & Reliability*, 15(2):97–133, 2005.

[MPP12] Mehdi MirzaAghaei, Fabrizio Pastore, and Mauro Pezzè. Supporting test suite evolution through test case adaptation. In Giuliano Antoniol, Antonia Bertolino, and Yvan Labiche, editors, *Fifth IEEE International Conference on Software Testing, Verification and Validation, ICST 2012, Montreal, QC, Canada, April 17-21, 2012*, pages 231–240. IEEE Computer Society, 2012.

[MWD+05] Tom Mens, Michel Wermelinger, Stéphane Ducasse, Serge Demeyer, Robert Hirschfeld, and Mehdi Jazayeri. Challenges in software evolution. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, IWPSE '05, pages 13–22, Washington, DC, USA, 2005. IEEE Computer Society.

[N+06] Dan North et al. Introducing BDD. *Better Software*, 12, March 2006.

[Nar17] Vishwak Narasimhan. Context-specific quality evaluation of test cases. Master's thesis, Paderborn University, Paderborn, Germany, 2017.

[NUn] NUnit. http://nunit.org/,.

[OCL] Object Constraint Language Specification Version 2.4.

[OMG08] OMG. Software and Systems Process Engineering Metamodel (SPEM), Version 2.0., 2008.

[OMG11a] OMG. *Architecture-driven Modernization: Abstract Syntax Tree Metamodel (ASTM)- Version 1.0*. Object Management Group, 2011.

[OMG11b] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Version 1.1, 1 2011.

[OMG13a] OMG. OMG Meta Object Facility (MOF) Core Specification, Version 2.4.1, 2013.

[OMG13b] OMG. UML Testing Profile (UTP), Version 1.2., 2013.

[OMG14] OMG. Model Driven Architecture (MDA): MDA Guide rev.2.0, 2014.

[OP97] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing, Verification and Reliability*, 7(3):165–192, 9 1997.

[Par81] David Park. Concurrency and automata on infinite sequences. In *Theoretical Computer Science*, pages 167–183. Springer-Verlag, Berlin/Heidelberg, 1981.

[PIT] PIT Mutation Testing. http://pitest.org/.

[PP04] Alexander Pretschner and Jan Philipps. Methodological issues in model-based testing. In *Model-Based Testing of Reactive Systems, Advanced Lectures [The volume is the outcome of a research seminar that was held in Schloss Dagstuhl in January 2004]*, pages 281–291, 2004.

[PP16] Kroll Per and Kruchten Philippe. *The Rational Unified Process Made Easy: A Practitioner's Guide to the RUP*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2016.

[PSO12] Leandro Sales Pinto, Saurabh Sinha, and Alessandro Orso. Understanding myths and realities of test-suite evolution. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 33:1–33:11, New York, NY, USA, 2012. ACM.

[PWGW08] Christian Pfaller, Stefan Wagner, Jörg Gericke, and Matthias Wiemann. Multi-Dimensional Measures for Test Case Quality. In *2008 IEEE International Conference on Software Testing Verification and Validation Workshop*, pages 364–368. IEEE, 2008.

[Rap15] Eric J. Rapos. Co-evolution of model-based tests for industrial automotive software. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*, pages 1–2. IEEE Computer Society, 2015.

[SBPM09] David Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework 2.0*. Addison-Wesley Professional, 2nd edition, 2009.

[SHT05] Harry M. Sneed, Martin Hasitschka, and Maria-Therese Teichmann. *Software-Produktmanagement: Wartung und Weiterentwicklung bestehender Anwendungssysteme*. dpunkt.verl., Heidelberg, 1. aufl. edition, 2005.

[SJ16] Joanna Strug and Joanna. Mutation Testing Approach to Negative Testing. *Journal of Engineering*, 2016:1–13, 7 2016.

[SJGE18] Simon Schwichtenberg, Ivan Jovanovikj, Christian Gerth, and Gregor Engels. Poster: Crossecore: An extendible framework to use ecore and ocl across platforms. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 292–293, May 2018.

[SL09] Andreas Spillner and Tilo Linz. *Basiswissen Softwaretest: Aus- und Weiterbildung zum Certified Tester – Foundation Level nach ISTQB-Standard*. dpunkt, Heidelberg, 3. edition, 2009.

[Sne99] Harry M. Sneed. Risks involved in reengineering projects. pages 204 – 211, 11 1999.

[Sne03] Harry M. Sneed. Software Testmetriken für die Kalkulation der Testkosten und die Bewertung der Testleistung. *GI Softwaretechnik Trends*, 4(23):11, 2003.

[Sne04] Harry M Sneed. Measuring the Effectiveness of Software Testing. In *Testing of Component-Based Systems and Software Quality, Proceedings of {SOQUA} 2004 (First International Workshop on Software Quality) and {TECOS} 2004 (Workshop Testing Component-Based Systems)*, page 109, 2004.

[Str] Stryker.NET. https://github.com/stryker-mutator/stryker-net.

[Str16] Joanna Strug. Applying mutation testing for assessing test suites quality at model level. In Maria Ganzha, Leszek A. Maciaszek, and Marcin Paprzycki, editors, *FedCSIS*, volume 8 of *Annals of Computer Science and Information Systems*, pages 1593–1596. IEEE, 2016.

[SW89] John Simpson and Edmund Weiner, editors. *The Oxford English Dictionary*. Oxford University Press, 1989.

[SWH10] Harry M. Sneed, Ellen Wolf, and Heidi Heilmann. *Software-Migration in der Praxis: Übertragung alter Softwaresysteme in eine moderne Umgebung*. 2010.

[SWH11] Matt Staats, Michael W. Whalen, and Mats P.E. Heimdahl. Programs, tests, and oracles. In *Proceeding of the 33rd international conference on Software engineering - ICSE '11*, page 391, New York, New York, USA, 2011. ACM Press.

[Tho20] Anu Tony Thottam. Situational context identification for test case migration. Master's thesis, Paderborn University, Paderborn, Germany, 2020. (work in progress).

[Ulr14] The ETSI Test Description Language TDL and its application. In Luís Ferreira Pires, Slimane Hammoudi, Joaquim Filipe, and Rui César das Neves, editors, *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pages 601–608. SciTePress, 2014.

[VE08] Hendrik Voigt and Gregor Engels. Kontextsensitive Qualitätsplanung für Software-Modelle. In Thomas Kühne, Wolfgang Reisig, and Friedrich Steimann, editors, *Modellierung 2008, 12.-14. März 2008, Berlin*, volume 127 of *LNI*, pages 165–180. GI, 2008.

[VGE08] Hendrik Voigt, Baris Güldali, and Gregor Engels. Quality Plans for Measuring the Testability of Models. In *Proceedings of the 11th International Conference on Quality Engineering in Software Technology (CONQUEST 2008), Potsdam (Germany)*, pages 353–370. dpunkt.verlag, 2008.

[vHFG+11] André van Hoorn, Sören Frey, Wolfgang Goerigk, Wilhelm Hasselbring, Holger Knoche, Sönke Köster, Harald Krause, Marcus Porembski, Thomas Stahl, Marcus Steinkamp, and Norman Wittmüss. Dynamod project: Dynamic analysis for model-driven software modernization. In *Joint Proceedings of the 1st International Workshop on Model-Driven Software Migration (MDSM 2011) and the 5th International Workshop on Software Quality and Maintainability (SQM 2011)*, volume 708 of *CEUR Workshop Proceedings*, pages 12–13, March 2011.

[Vin20] Vishal Joseph Vincent. Method patterns for construction of context-specific test case migration methods. Master's thesis, Paderborn University, Paderborn, Germany, 2020. (work in progress).

[Vis] VisualMutator - .NET mutation testing. https://visualmutator.github.io/web/.

[Voi09] Hendrik Voigt. Kontextsensitive Qualitätsplanung von Softwaremodellen, 2009. Paderborn, Univ., Diss., 2009.

[WAF+19] Nils Weidmann, Anthony Anjorin, Lars Fritsche, Gergely Varró, Andy Schürr, and Erhan Leblebici. Incremental bidirectional model transformation with emoflon: : Ibex. In *Proceedings of the 8th International Workshop on Bidirectional Transformations co-located with the Philadelphia Logic Week, Bx@PLW 2019*, pages 45–55, 2019.

[WAS14] Martin Wieber, Anthony Anjorin, and Andy Schürr. On the usage of tggs for automated model transformation testing. In Davide Di Ruscio and Dániel Varró, editors, *Theory and Practice of Model Transformations*, pages 1–16. Springer International Publishing, 2014.

[WG11] Edith Werner and Jens Grabowski. Model Reconstruction: Mining Test Cases. In *Third International Conference on Advances in System Testing and Validation Lifecycle*, VALID 2011, 2011.

[Won01] W. Eric Wong, editor. *Mutation Testing for the New Century*. Kluwer Academic Publishers, USA, 2001.

[Xte] Xtend - Modernized Java. http://www.eclipse.org/xtend/.

[XXBW12] Dianxiang Xu, Weifeng Xu, Bharath K. Bavikati, and W. Eric Wong. Mining executable specifications of web applications from selenium IDE tests. In *Sixth International Conference on Software Security and Reliability, SERE 2012, Gaithersburg, Maryland, USA, 20-22 June 2012*, pages 263–272. IEEE, 2012.

[ZRDvD08] Andy Zaidman, Bart Van Rompaey, Serge Demeyer, and Arie van Deursen. Mining software repositories to study co-evolution of production & test code. In *First International Conference on Software Testing, Verification, and Validation, ICST 2008, Lillehammer, Norway, April 9-11, 2008*, pages 220–229. IEEE Computer Society, 2008.

[ZVS+07] 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.

[ZWH+11] Christian Zillmann, Andreas Winter, Axel Herget, Werner Teppe, Marianne Theurer, Andreas Fuhr, Tassilo Horn, Volker Riediger, Uwe Erdmenger, Uwe Kaiser, and Denis Uhlig. The soamig process model in industrial applications. In Andreas Winter Tom Mens, Yiannis Kanellopoulos, editor, *Proceedings of the 15th European Conference on Software Maintenance and Reengineering IEEE Computer, Los Alamitos, 2011*, 03 2011.

Appendix A

Extended Quality Model for Test Cases

A.1 Extended Quality Model for Test Cases

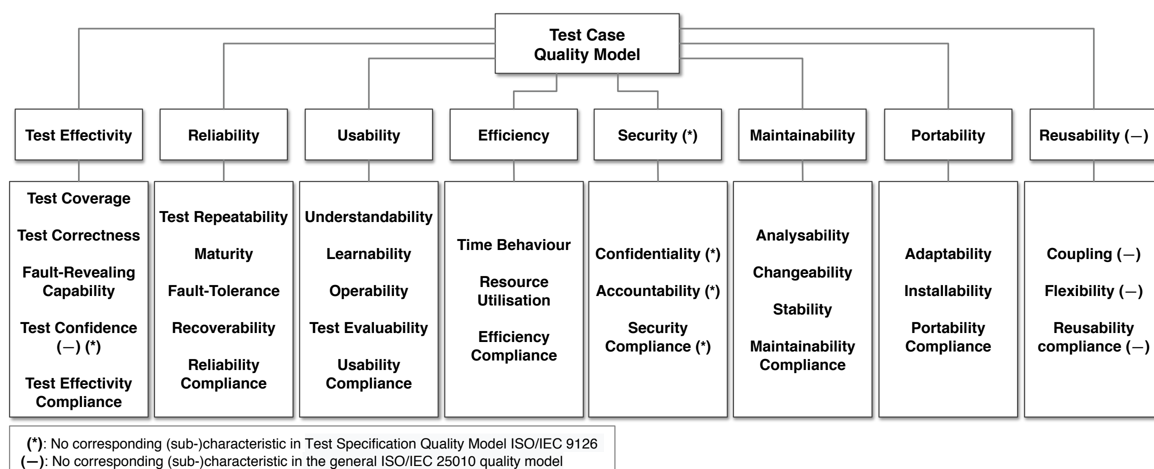


Figure A.1 The ISO/IEC 25010 Standard-based Quality Model for Test Cases

Test Effectivity The test effectivity characteristic represents the degree to which the specified test cases fulfil a given test purpose [ZVS$^+$07]. The test effectivity of the test cases could be expressed in terms of the sub-characteristics shown in Table A.1

Reliability The reliability characteristic represents the degree to which the specified test cases perform specified conditions for a specified period of time [ZVS$^+$07]. The test cases have to maintain a specific level of performance under different conditions. The sub-characteristics test repeatability and security was introduced in [ZVS$^+$07]. However, the current standard has a dedicated main characteristic for security. Hence, security is removed and added as a main characteristic. The reliability of the test cases could be expressed in terms of the sub-characteristics shown in Table A.2.

Sub-characteristics	Description
Test Coverage	The degree to which the test cases are complete for a particular coverage criterion [ZVS+07]. The completeness of test cases could be measured on different coverage criterion, e.g. test purpose coverage, system model coverage, requirements coverage, architectural coverage and code coverage [Sne04]
Test Correctness	The degree of correctness of the test cases with respect to the system specification or the test purposes[ZVS+07]
Fault-revealing Capability	The capability of the test cases to reveal faults
Test Confidence*	The degree of confidence of the test cases with respect to the errors reported by the tester and the user [Sne04]

Table A.1 Sub-characteristics of Test Effectivity

Sub-characteristics	Description
Test Repeatability	The degree to which the test cases reproduce the same test result in subsequent test runs. Otherwise, debugging the SUT to locate a defect becomes hard to impossible. Test repeatability includes the demand for deterministic test specifications [ZVS+07]
Maturity	The degree to which the test cases are reliable to be executed under normal operation [ZVS+07]
Fault-tolerance	The degree to which the test cases could be used despite the presence of hardware or software faults [ZVS+07]
Recoverability	The degree to which, in the event of an interruption or a failure, test cases can recover and re-establish the desired state of execution. The test cases should be able to log the reason for failure [ZVS+07]

Table A.2 Sub-characteristics of Reliability

Usability The usability characteristic represents the degree to which the specified test cases could be used with ease [ZVS+07]. This characteristic represents the degree to which the test cases could be used with at most satisfaction during a particular condition [ZVS+07]. The usability of test cases could be expressed in terms of the sub-characteristics shown in Table A.3.

Performance Efficiency The efficiency characteristic in the ISO/IEC standard 9126 has been renamed as performance efficiency in the ISO/IEC standard 25010. This characteristic represents the degree to which the specified test cases could provide an acceptable performance in terms of speed in executing the tests and resource usage [ISO11b]. The performance efficiency of test cases could be expressed in terms of the sub-characteristics shown in Table A.4.

Sub-characteristics	Description
Understandability	The degree to which the test cases are understandable for a particular need. Documentation and description of the overall purpose of the test specification are important factors and guides in finding the suitable test cases [ZVS+07]
Learnability	The degree to which the test cases contain documents describing how it is configured, what kinds of parameters are involved, and how they affect the test behavior [ZVS+07]. Proper documentation or style guides have positive influence on this quality as well
Operability	The degree to which the test cases are operable, i.e., measure to check if a test suite contains appropriate default values or a lot of non-automatable actions are required in the actual test execution. Such factors make it hard to set-up a test suite for execution or they make execution time-consuming due to a limited automation degree [ZVS+07]
Test Evaluability	The degree to which, in the event of an interruption or a failure, test cases can recover and re-establish the desired state of execution. The test cases should be able to log the reason for failure [ZVS+07]
Consistency*	The degree to which the test cases are free from contradiction and are coherent with other test cases in a specific context of use. Proper documentation or style guides have positive influence on this quality as well [ISO11b]
Completeness	The degree to which the test cases have information completely specified about their objective, inputs, outcomes, environmental needs. This information increases the understandability of test cases and could also be a factor in determining whether the test cases are created for the right objective [ISO11d]

Table A.3 Sub-characteristics of Usability

Sub-characteristics	Description
Time Behavior	The degree to which the execution times of the test cases, when performing its functions, meet requirements [ISO11b]
Resource Utilization	The amounts and types of resources used by the test cases, meet requirements [ISO11b]

Table A.4 Sub-characteristics of Performance Efficiency

Security The term security was a sub-characteristic in the ISO/IEC standard 9126, and it has been termed as the main characteristic in the ISO/IEC standard 25010. This characteristic represents the degree to which the specified test cases have information and data that are protected and are only available depending upon the levels of authorization [ISO11b]. This characteristic also covers issues such as included plain-text passwords that play a role when

test cases are made publicly available or are exchanged between development teams. The level of security of test cases could be expressed in terms of the sub-characteristics shown in Table A.5.

Sub-characteristics	Description
Confidentiality*	The degree to which the test cases has attributes that ensure that it is only accessible and interpretable by authorized users in a particular context of use [ISO11b]
Accountability*	The degree to which the test case actions could be traced, e.g., Details about the tester accountable for a particular test case has to be documented for further clarifications. In huge projects, such details are not maintained [ISO11b]

Table A.5 Sub-characteristics of Security

Maintainability The maintainability characteristic represents the degree to which the specified test cases could be modified with ease due to changes in the software. Maintainability of test specifications is critical when test developers are faced with changing or expanding a test specification. It characterizes the capability of a test specification to be modified for error correction, improvement, or adaptation to changes in the environment or requirements [ZVS+07]. The maintainability of test cases could be expressed in terms of the sub-characteristics shown in Table A.6.

Sub-characteristics	Description
Analyzability	The degree to which the test cases can be diagnosed for deficiencies. For example, test cases should be well-structured to allow code reviews. Test architecture, style guides, documentation, and well-structured code are elements that have influence in the quality of this property
Changeability	The degree to which the test cases could undergo modification when required. For example, poorly structured code, hard coded values or test cases that could not be expandable may have negative impact on quality
Stability	The degree to which the test case could be stable due to unexpected side effects. This depends on the test case language used, unexpected side effects due to a modification have an adverse impact on the stability aspect
Complexity*	The degree to which the test cases contain complex test data types and its instances. Test cases with long preconditions and procedure may also have an adverse impact on the complexity aspect

Table A.6 Sub-characteristics of Maintainability

Portability The portability characteristic represents the degree to which the specified test cases could be transferred from one operational or usage environment to another [ZVS+07]. The level of portability of test cases could be expressed in terms of the sub-characteristics shown in Table A.7.

Sub-characteristics	Description
Adaptability	The degree to which the test cases a capable to be adaptable to different system under tests or environments [ZVS+07]

Table A.7 Sub-characteristics of Portability

Reusability The reusability characteristic represents the degree to which the specified test cases could be reused for different test types [ZVS+07]. The level of reusability of test cases could be expressed in terms of the sub-characteristics shown in Table A.8.

Sub-characteristics	Description
Coupling	The degree to which the test cases have test data and other resources that are coupled with other sets of test cases [ZVS+07]. It is a measure to verify the test cases are loosely coupled and have a strong cohesion
Flexibility	The degree to which the test cases a capable to be adaptable to different system under tests or environments. For example, if fixed values appear in a part of a test specification, a parametrization likely increases its reusability [ZVS+07]
Comprehensibility	The degree to which the test cases are unambiguous and might not contain any conditional logic. Test cases with ambiguity or conditional logic can have an adverse impact on the comprehensibility aspect. Good documentation, comments, and style guides are necessary to achieve this goal [ISO11d]

Table A.8 Sub-characteristics of Reusability