Model-Driven Testing

Paul Baker · Zhen Ru Dai · Jens Grabowski ·
Øystein Haugen · Ina Schieferdecker ·
Clay Williams

# Model-Driven Testing

## Using the UML Testing Profile

With 94 Figures and 4 Tables

Springer

*Authors*

Paul Baker

Motorola Labs
Jays Close
Viables Industrial Estate
Basingstoke
Hampshire, RG22 4PD, UK
Paul.Baker@motorola.com

Zhen Ru Dai
Ina Schieferdecker

Fraunhofer Fokus
Kaiserin-Augusta-Allee 31
10589 Berlin, Germany
dai@fokus.fraunhofer.de
schieferdecker@fokus.fraunhofer.de

Jens Grabowski

Institute for Computer Science
University of Goettingen
Lotzestrasse 16-18
37083 Göttingen, Germany
grabowski@cs.uni-goettingen.de

Øystein Haugen

Department of Informatics
University of Oslo
P.O. Box 1080, Blindern
0316 Oslo, Norway
oysteinh@ifi.uio.no

Clay Williams

19 Skyline Drive
Hawthorne, NY 10532
USA
clayw@us.ibm.com

Library of Congress Control Number: 2007928307

ACM Computing Classification: D.2, K.6

ISBN 978-3-540-72562-6 Springer Berlin Heidelberg New York



Springer is a part of Springer Science+Business Media
springer.com


Cover Design: KünkelLopka, Heidelberg
Typesetting: by the authors
Production: Integra Software Services Pvt. Ltd., Puducherry, India

Printed on acid-free paper 45/3100/Integra 5 4 3 2 1 0

# Foreword

Psychologists have a phrase for the human ability to believe two completely contradictory things at the same time; it is called "cognitive dissonance." We do seem to be quite capable of believing that war is evil (though we should destroy our enemies), taxes are too high (though government spending should increase), and that politicians can keep a promise (perhaps enough said on that point). Psychologists in fact tell us that this ability to simultaneously face and ignore realities makes it possible to live in a complicated, constantly changing world. Engineers, however, are supposed to do better than live in hope. Engineers model systems, test their models against requirements, execute their designs, and then test their models against the post-development reality (e.g., system design versus as-built) to ensure the quality and sustainability of their product and in fact their production methods. Why, then, do we in the computer engineering field find ourselves with an enormous software quality problem, all over the world?

I've just finished reading an article in a leading software development magazine about the state of the software testing industry, which leads off with the unlikely sentence, "The ethos of quality is sweeping into the IT space..."[1]. I had to read the line several times before I realized that it did not read, "the ethos of quality has swept completely out to space," which I think is the more likely position! Software quality–measured either qualitatively or quantitatively–is atrocious, in nearly all settings, whether commercial, academic, or open-source. The level of concern for customer needs, even customer *safety,* is surprisingly small. Major software vendors beta-test (and even alpha-test) their software by placing it in the hands of millions of customers, many of whom are paying for the software they are testing. "Don't buy version 1.0," is the common credo of most Chief Information Officers that I know.

[1] "The Trouble with Software QC: Steps toward making better software better," by Colin Armitage, in Dr Dobb's Journal, March 2007.

Where might the solution to this problem lie? Certainly, there are several different foci needed to solve the problem, and the most important is a changing point of view (and therefore incentivization and management style) on the part of customers and development labs, a point of view that software quality is important and in fact might be worth paying for. These market issues, coupled with developer commitment to software quality expressed in process and methodology, will make a difference.

But technology plays a part also. In particular, the move to Model-Driven Architecture (MDA) as a basis for engineering software-based systems shows some promise to help support corporate commitments to quality. Although MDA originally focused primarily on software delivery, maintenance, and integration, with the intent of lowering the costs involved in those phases of the software lifecycle (especially maintenance and integration), the MDA approach has something to say about software testing as well.

Think back, for a moment, to what engineers did (and still do) when engineering meant purely nuts, bolts, steel, wood, and concrete. An architect or structural engineer would specify materials for, for example, bridge construction, including such things as steel tensile strength and admixture minima in the concrete mix. While these specifications—abstractions, or models—are used in various artifacts of the construction process (e.g., requests for quotation, requests for proposals, bills of materials, bills of lading), they are also used by responsible engineers in the test phase of construction. When the concrete appears on the job site, samples are tested to ensure that the admixture delivered meets the requirements specified in the construction plans. If it does not, a potential disaster awaits, as the entire structure may very well depend on the concrete block about to be poured. This lesson was learned in unfortunate detail throughout the 19th century, as the failings of cast iron as a bridge-building material (due to fracture) was slowly understood.

The key idea here, however, is not about bridge-building but about *tests based on designs.* Clearly, we should be doing the same thing in the software field if we are to be considered "engineers" in the same class as civil engineers and chemical engineers!

This concept of model-driven testing, of extending the concepts of tests based on designs (borrowed from older engineering fields) to *software* designs, has been a major focus area recently at OMG and is carefully expanded upon in the book you are holding. Once one has a system design—for a road bridge or for a bridge-playing program!—one should be able to specify in those requirements how to test the as-built artifact (the running code) to ensure that it meets the requirements of the intended (and often unintended) users.

Software modeling is not a trivial subject, and extending modeling to the automatic generation not only of code but of software unit and integration tests is not any simpler. This book, however, does an admirable job of integrating the concepts of modeling and testing and leverages the latest standards

and technologies to help you make your software testing regime and process better, faster, and cheaper. It is worth the read.

The problem with politicians, however, remains for a better day!

Richard Mark Soley, PhD
Chairman and CEO
Object Management Group, Inc.
April 25, 2007

# Preface

As software systems become more important in almost every facet of human life, the demand for software quality increases. As a result, several new technologies have emerged to help with the development of high-quality, systems. These include UML for system specification, JUnit, and TTCN-3 for test case design and execution, as well as new development paradigms such as Model-Driven Architecture (MDA).

Prior to the development of the UML Testing Profile (UTP), there was not a test specification language that aligned UML and other technologies such as those mentioned above. The consortium of institutions involved in the development of the UTP (Ericsson, Fraunhofer/FOKUS, IBM/Rational, Motorola, Telelogic, University of Lübeck) sought to address this absence by providing a UML profile for test specification. This book represents the first comprehensive introduction to the standard.

We would like to thank Telelogic AB for providing us with the Telelogic TAU tool for use in creating the diagrams in this book. We also want to thank Eric Samuelsson, our colleague at Telelogic, who was invaluable in developing the profile. He was unable to participate in the writing of this book, but his wise guidance during the development of the profile informs much of the material in this work. Finally, we want to thank our editor at Springer, Ralf Gerstner, for giving us a chance to publish the work and for providing us with careful guidance during the process.

*Paul Baker* Winchester, UK,
*Zhen Ru Dai* Berlin, Germany,
*Jens Grabowski* Göttingen, Germany,
*Øystein Haugen* Oslo, Norway,
*Ina Schieferdecker* Berlin, Germany,
*Clay Williams* New York, NY,
April 2007

# Contents

# Introduction

As software systems become increasingly complex, new paradigms are needed for their construction. One of these new paradigms is model-driven development, which already has a demonstrable impact in reducing time to market and improving product quality. This particular paradigm is concerned with the introduction of rigorous models throughout the development process, enabling abstraction and automation. To this end a standardized, graphical language called the *Unified Modeling Language* (UML)$^{TM}$ was developed by the IT and software industry for the construction of software systems. UML enables system requirement and design specifications to be created and visualized in a graphical manner, which supports improved communication and validation. It also enables the introduction of automated software production techniques. As a consequence, UML has become widely adopted as a development technology throughout the software industry.

However, the development of high-quality systems requires not only systematic development processes but also systematic test processes. This book is specifically about systematic, model-based test processes in the context of UML. As UML provides only limited means for the design and development of corresponding tests, a consortium was built by the *Object Management Group* (OMG) in order to develop a *Unified Modeling Language, version 2* (UML 2) profile to enable model-based testing with UML [24, 36]. We refer to this profile as the *UML Testing Profile* (UTP).

With the resulting UTP, the way models and their realizations can be tested and evaluated has been unified. Dedicated test concepts help to design and structure tests, to define test procedures and related test data precisely. The profile also allows users to derive and reason about the quality of tests and about the test coverage of a given test suite. The UTP closes a gap in the set of technologies around UML: It combines and integrates system and test development via UML and provides a common way for the different stakeholders to communicate and reason about the systems and their tests with which they are involved as developers, purchasers, customers, and users. The profile also allows system architects, designers, and developers to efficiently cooperate

with the testers throughout the system development process on the basis of a common model base.

Six organizations consisting of tool vendors, industrial users, and research institutes (Ericsson, Fraunhofer FOKUS, IBM/Rational, Motorola, Telelogic and University of Goettingen to Lubeck) decided to collaborate and produce the UTP specification jointly. While some of the consortium members come from the world of testing, others are experienced in system modeling with *Message Sequence Chart* (MSC), *Specification and Description Language* (SDL), and UML, respectively. These members agreed that by working in a larger team, the resultant profile would have a broader scope on testing, where the best ideas should be incorporated into the profile specification. After 2 years' work, the UTP specification has been adopted by the OMG [35]. Since summer 2005, the profile has become an official standard at the OMG [26].

In this book, we present how UML is used for the purposes of testing complex software systems. We introduce a systematic but simple methodology for addressing different types of testing within UML-based development. The book helps avoiding typical difficulties in designing, structuring, and developing tests systematically: UTP is introduced stepwise–driven by a case study that highlights the principles and concepts of the underlying methodology. The important issues of test completeness, correctness, and consistency are discussed. The book teaches the reader how to use the UTP for test modeling and test specification. It presents best practices for using UML for different aspects of testing and demonstrates the automated execution of UML-based tests with existing test frameworks such as the JUnit test framework for Java and *Testing and Test Control Notation* (TTCN-3).

To aid the reader, we introduce three types of highlighted text within the book:

1. **UTP Concepts.** These are presented to the reader as we introduce concepts from the UTP standard.
2. **UTP Methodology.** These denote key process aspects the reader should consider when using UTP.
3. **UTP Tips.** These are include to highlight pertinant or key points that are very useful or important when using or considering UTP.

Everyone involved in software quality assurance could benefit from the techniques introduced in this book. The book helps testers understand the concepts of UTP and applying it within a UML-based system development process. The book is

- for *testers and developers* who are looking to use model-based and automated testing or that are in organizations using UML for software system development;
- for *project managers, system architects, and designers* who want to address system quality and testability aspects throughout system development;

- for *students and academics* wishing to learn about testing with UML; and
- for *tool developers* wishing to get an overview of how tools can support UML-based testing.

Although UML is briefly introduced at the beginning of the book, it is advantageous for readers to be familiar with UML—for example, as described in [29].

We recommend reading this book sequentially from start to end. UML experts and those being less interested in the specific details of the case study may decide to start with Part II on *functional testing*. If you are interested in *advanced testing concepts*, for example, how to design test data efficiently or how to do performance, load, and scalability testing, you might start reading with Part III. *General aspects of applying UTP* are discussed in Part IV. Tools are covered in Part V and Section 11.1, respectively. Part VI provides an overview of a UTP reference guide, a list of acronyms, an index, and references.