SPRINGER BRIEFS IN INFORMATION SYSTEMS

Tim A. Majchrzak

Improving Software Testing

Technical and Organizational Developments

ERCIS

Springer

Tim A. Majchrzak

Improving Software Testing

Technical and Organizational Developments

Springer

Tim A. Majchrzak
Institut für Wirtschaftsinformatik
Westfälische Wilhelms-Universität
Leonardo Campus 3
48149 Münster
Germany

ISSN 2192-4929 e-ISSN 2192-4937
ISBN 978-3-642-27463-3 e-ISBN 978-3-642-27464-0
DOI 10.1007/978-3-642-27464-0
Springer Heidelberg New York Dordrecht London

Library of Congress Control Number: 2011946088

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

SpringerBriefs in Information Systems

Series Editor

Jörg Becker

For further volumes:
http://www.springer.com/series/10189

For Imke

Foreword

We are delighted to announce a new series of the SpringerBriefs: SpringerBriefs in Information Systems. SpringerBriefs is a well-established series that contains concise summaries of up-to-date research with a practical relevance across a wide spectrum of topics with a fast turnaround time to publication. The series publishes small but impactful volumes with a clearly defined focus. Therefore, readers of the new series are able to get a quick overview on a specific topic in the field. This might cover case studies, contextual literature reviews, state-of-the art modeling techniques, news in software development or a snapshot on an emerging topic in the field.

SpringerBriefs in Information Systems present research from a global author community and allow authors to present their ideas in a compact way and readers to gain insights into a specific topic with minimal time investment. With succinct volumes of 50–125 pages, SpringerBriefs are shorter than a conventional book but longer than an average journal article. Thus, publishing research in the SpringerBriefs series provides the opportunity to bridge the gap between research results already published in journal articles or presented on conferences and brand new research results.

Information Systems research is concerned with the design, development and implementation of information and communication systems in various domains. Thus, SpringerBriefs in Information Systems give attention to the publication of core concepts as well as concrete implementations in different application contexts. Furthermore, as the boundary between fundamental research and applied research is more and more dissolving, this series is particularly open to interdisciplinary topics.

SpringerBriefs are an integral part of the Springer Complexitiy publishing program and we are looking forward to establishing the SpringerBriefs in Information Systems series as a reference guide for Information Systems professionals in academia and practice as well as students interested in latest research results.

The volume at hand is the first SpringerBriefs in Information Systems. It presents the work of Tim A. Majchrzak on software testing. The outline is exemplary for the intended scope: Not only a literature-driven introduction to software testing is given but also results from three current research topics are summarized and their future implications sketched.

Münster, January 2012 Jörg Becker

Preface

In spite of the increasing complexity of software, clients expect improvements in its quality. In order to enable high-quality software in this context, the tools and approaches for testing have to be reconsidered. This Brief summarizes technical as well as organizational aspects of testing. For the latter, an empirical study of best practices for testing has been conducted. As a result, a set of recommendations has been formulated, which help to organize the testing process in a company while taking into account parameters such as its size. It turns out that companies test very differently. While some (typically larger) companies have installed well established testing procedures and use state of the art tools e.g. in order to automate testing as far as possible, other (often smaller) companies test on an ad-hoc basis without predefined processes and with little tool support. In particular, the latter can benefit from the best practices formulated in this book. This part is particularly interesting for practitioners.

From a technical perspective, this Brief presents a new way for generating so-called glass-box test cases automatically. The approach is based on a symbolic execution of Java bytecode by a symbolic Java Virtual Machine (JVM). The latter is an extension of the usual JVM by components known from abstract machines for logic programming languages such as the Warren Abstract Machine. In particular, it contains a trail, choice points, and logic variables and it uses a system of constraint solvers. The symbolic JVM systematically follows all relevant computation paths and generates a set of test cases ensuring a predefined coverage of the control and data flow of the tested software. This part of the Brief is particularly interesting for researchers working on testing.

A third part shows how the mentioned test-case generator can be profitably used in an E-assessment system which automatically corrects Java classes uploaded by students. The test cases are generated from an example solution and serve as a measure for evaluating the correctness of the uploaded solutions to programming exercises. This approach has been successfully applied in a practical programming course. Corresponding experimental results are given.

This Brief is an extract of the PhD thesis of the author. It gives a concise overview of the mentioned aspects with many references to the relevant literature. It is interesting for practitioners as well as researchers. For studying the most interesting aspects in depth, a look into the mentioned literature is recommended.

Münster, December 2011 Herbert Kuchen

Acknowledgments

This book is based on my PhD thesis. In fact, it contains the revised first part of the thesis, which introduces the research topics that I dealt with, sketches its background, and thereby summarizes the body of knowledge of software testing. Therefore, I would like to thank again those people that supported me in the preparation of the thesis.

In particular, I am grateful for support by Prof. Dr. Herbert Kuchen, Prof. Dr. Jörg Becker, and Prof. Dr. Ulrich Müller-Funk who were the members of my doctoral committee. As my advisor, Herbert Kuchen also provided invaluable support that contributes to the quality of the content provided in this book. Moreover, he and Jörg Becker kindly wrote the foreword and the preface.

Feedback on the thesis manuscript was provided by Dr. Philipp Ciechanowicz, Prof. Dr.-Ing. Marco K. Koch, Thomas Majchrzak, Martin Matzner, Claus Alexander Usener, and Imke Wasner. Imke, my wife, also soothed me when ludicrous LaTeX package incompatibilities decelerated my progress. In the end, feedback and support are also reflected in this book. Thank you!

With regard to this Briefs volume I would like to thank Christian Rauscher from Springer-Verlag who comprehensively answered all my questions concerning the manuscript preparation and the publication process.

Münster, December 2011 Tim A. Majchrzak

Abbreviations

.NET	Microsoft .NET framework
ABS	Anti-lock braking system
ACM	Association for Computing Machinery
AI	Artificial intelligence
Ajax	Asynchronous JavaScript and XML
APA	American Psychological Association
API	Application programming interface
B2B	Business-to-business
B2G	Business-to-government
CASE	Computer aided software engineering
CAST	Computer aided software testing
CF	Control flow
CFG	Control flow graph
CLI	Common Language Infrastructure
CMMI	Capability Maturity Model Integration
CobiT	Control Objectives for Information and Related Technology
CPU	Central processing unit
CS	Computer science
CSTE	Certified software tester
DU	Def-use
EASy	E-Assessment System
ERCIS	European Research Center for Information Systems
ESC	Electronic stability control
FMEA	Failure mode and effects analysis
FMECA	Failure mode and effects and criticality analysis
FPP	Fredge Program Prover
GTB	German Testing Board
GUI	Graphical user interface
HCI	Human–computer interaction
I/O	Input/output
IAI	Institut für Angewandte Informatik

IDE	Integrated development environment
IEEE	Institute of Electrical and Electronics Engineers
IHK	Industrie- und Handelskammer
IoC	Inversion of control
IS	Information system
ISO	International Organization for Standardization
ISTQB	International Software Testing Qualifications Board
IT	Information technology
ITIL	IT infrastructure library
JIST	Journal of Information Science and Technology
JJ-path	Jump-to-jump path
JNI	Java native interface
JRE	Java Runtime Environment
JSR	Java Specification Request
JVM	Java virtual machine
LCSAJ	Linear code sequence and jump
LOC	Lines of code
MiB	Mebibyte, i.e. 2^{20} bytes
Muggl	Muenster generator of glass-box test cases
MVC	Model-view-controller
RE	Requirements engineering
ROI	Return on investment
RUP	Rational Unified Process
SE	Software engineering
SI	International system of units (Système international d'unités)
SJVM	Symbolic Java virtual machine
SMT	Satisfiability modulo theory
STQC	Standardisation Testing and Quality Certification
SWEBOK	Software Engineering Body of Knowledge
TCG	Test case generation
TDD	Test driven development
TMap	Test Management Approach
TPI	Test Process Improvement
TQM	Total Quality Management
UML	Unified Modeling Language
VM	Virtual machine
vs.	Versus
w.r.t.	With respect to
W3C	World Wide Web Consortium
XML	Extensible Markup Language
XP	Extreme Programming

Contents

Chapter 1
Introduction

In the following sections, this book's topic is introduced, objectives and research questions are discussed, and the course of action is sketched. First of all, the setting for software testing is described.

1.1 The Software Crisis

Since the advent of computers, the available computational power and the capacity of memory and storage systems have grown exponentially [1, 2]. At the same time, computer systems decreased in size and their prices plummeted. But it was software that made the vast computational power operational. Software enables humans to take advantage of general purpose computers for distinct applications in business, science, communications, specialized fields such as healthcare, and entertainment.

In the 1960s, the costs of developing software for the first time became higher than the costs of developing the corresponding hardware. As a consequence, the discipline *software engineering* was installed [3]. In 1972, Dijkstra eventually coined the term *software crisis* [4] which summarizes the observed phenomena. Software grows in size and complexity. This development is exacerbated by the fact that the increase in computational power demands more complex software in order to be utilized. Furthermore, the introduction of the Internet added more complexity. While programmers formerly had to care for local resources and how a program ran on a single computer, nowadays its embedding into a network and possible connections to a myriad of services have to be kept in mind.

Unfortunately, it can be observed that the software crisis is lasting as impressively demonstrated by the shift to *multicore* (or even *manycore* [5]) computer architectures [6]. In theory, the computational power of a central processing unit (CPU) (almost) doubles when doubling the number of cores. While there are hardware limitations for this rate and some overhead has to be taken into account, most software today is written sequentially. Despite some attempts, such as the release

T. A. Majchrzak, *Improving Software Testing*, SpringerBriefs in Information Systems,
DOI: 10.1007/978-3-642-27464-0_1,

of conveniently usable libraries (e.g. [7]), programming for multicore systems is not common; means to convert existing software in order to allow effective utilization of multiple cores are not yet satisfying. As a consequence, the misalignment between computational power and easy means of using it deteriorates. Until better ways have been found, programming for multicore computers will be burdensome. Programs that entirely utilize more than one core remain to be costly. It is unlikely that many programmers will use adequate programming techniques; in fact, it will stay a domain for experts and researchers [8].[1]

The software crisis has two main consequences. Firstly, software development projects exceed the calculated costs, the developed software lacks specified functionality and has inferior quality, or projects fail completely [9]. Secondly, possible amenities are not realized. It is either not feasible to write software for a desired task due to the inherent complexity of it, or the effort of development and the possibility of failure is avoided since the risks are considered to be too high. Both consequences result in increased costs and less welfare in an economic sense. Therefore, finding ways of effectively using software is very important.

Whereas it is impossible to quantitatively assess the missed welfare, studies document project failure. These studies underline that there were not only some large projects that failed but that irregularities seem to be inherent to software development. Reports of problems of varying severity can be found for any kind of project, for any industrial sector it has been conducted in, and for any project size [10]. Stories of failure easily fill up full books [11–13]. Some (almost) failed projects have extensively been studied by the literature (cf. [14, Chap. 1] [15, Chap. 2]). A prominent example is the software for the modernization of US taxing authorities (Internal Revenue Service). With a delay of 8 years and a loss of (at least) 1.6 billion USD it can be seen as one of the worst failures in the history of software development [15, p. 140ff.] [16, p. 1].

There are many more examples of projects that failed. Unfortunately, some failures endangered the live of humans or even lead to fatalities. The introduction of a new system for registering and dispatching emergency calls in London has been thoroughly studied. When the system was activated in October 1992, it did not properly work. Emergency calls were delayed and ambulances reached patients too late. The problems with the new system have been attributed for up to 30 fatalities [17, 18]. Despite administrative failures that caused the problems [19], it also was a technical disaster. Whereas it is good to know that lessons have been learned from such incidents [20], it has to be asked why they cannot be avoided beforehand. In retrospective, it seems like "good engineering practice had been ignored, every guideline of software engineering disregarded" and "basic management principles neglected" [21]. With a closer look, however, it has to be said that it apparently was a systematic

[1] Of course, there is much progress and the number of programs that effectively and efficiently use multicores increases. However, research often is fundamental and there is no programming paradigm that would allow to program for multicore computers as convenient as it is to write sequential programs.