Robert Glück
Michael Lowry (Eds.)

LNCS 3676

Generative Programming and Component Engineering

4th International Conference, GPCE 2005
Tallinn, Estonia, September/October 2005
Proceedings

Springer

TP311-53
G725
2005

Robert Glück Michael Lowry (Eds.)

Generative Programming and Component Engineering

4th International Conference, GPCE 2005
Tallinn, Estonia, September 29 – October 1, 2005
Proceedings

Springer

Volume Editors

Robert Glück
University of Copenhagen, DIKU, Department of Computer Science
Universitetsparken 1, 2100 Copenhagen, Denmark
E-mail: glueck@acm.org

Michael Lowry
NASA Ames Research Center, M/S 269-2
Moffett Field, CA 94035-1000, USA
E-mail: lowry@email.arc.nasa.gov

Library of Congress Control Number: 2005932566

CR Subject Classification (1998): D.2, D.1, D.3, K.6

ISSN	0302-9743
ISBN-10	3-540-29138-5 Springer Berlin Heidelberg New York
ISBN-13	978-3-540-29138-1 Springer Berlin Heidelberg New York

Typesetting: Camera-ready by author, data conversion by Scientific Publishing Services, Chennai, India
Printed on acid-free paper SPIN: 11561347 06/3142 5 4 3 2 1 0

Preface

Generative Programming and Component Engineering (GPCE) is a leading research conference on automatic programming and component engineering. These approaches to software engineering have the potential to revolutionize software development as automation and components revolutionized manufacturing. The conference brings together researchers and practitioners interested in advancing automation for software development. It is also a premier forum for cross-fertilization between the programming language and software engineering research communities.

GPCE arose as a joint conference, merging the prior conference on Generative and Component-Based Software Engineering (GCSE) and the Workshop on Semantics, Applications, and Implementation of Program Generation (SAIG). The proceedings of the previous GPCE conferences were published in the LNCS series of Springer as volumes 2487, 2830, and 3286. In 2005 GPCE was co-located with the International Conference on Functional Programming (ICFP) and the symposium on Trends in Functional Programming (TFP), reflecting the vigorous interaction between the functional programming and generative programming research communities. GPCE and ICFP are both sponsored by the Association for Computing Machinery.

The quality and breadth of the papers submitted to GPCE 2005 was impressive. All 86 papers, including 5 papers for tool demonstrations, were rigorously reviewed by 17 highly qualified Program Committee members. The members of the Program Committee first provided in-depth individual reviews of the submitted papers, and then debated the merits of the papers through an extended electronic Program Committee meeting. After much (friendly) argument, 25 regular papers and 2 tool demonstration papers were selected for publication. The Program Committee provided extensive technical feedback to the authors of the submitted papers. The conference program was complemented with three invited talks, three extended tutorials, and three all-day workshops.

The accepted papers are grouped into eight topic areas: aspect-oriented programming, component engineering and templates, demonstrations, domain-specific languages, generative techniques, generic programming, meta-programming and transformation, and multi-stage programming. The invited talks were from leading innovators in the field: Oscar Nierstrasz on object-oriented reengineering patterns, Oege de Moor on the AspectBench compiler for AspectJ, and Bernd Fischer on certifiable program generation.

The program chairs would like to thank foremost the authors of the submitted papers: their research is the justification for this conference. Both program chairs were impressed by the expertise and diligence of the Program Committee members and their co-reviewers. Their technical dedication, as reflected in the quality of their reviews, was the foundation of the strength of these proceedings.

The general chair, Eugenio Moggi, was tireless in steering the program chairs towards a technically superb program. The publicity chair, Eelco Visser, went beyond the call of duty in raising awareness of the conference in the software engineering and programming languages research communities. Andrew Malton and Jeff Gray solicited and organized a workshop and tutorial program of interest to researchers and practitioners alike. Tarmo Uustalu graciously served as local arrangements chair, providing a hospitable atmosphere in the beautiful venue of Tallin, Estonia. The paper submissions and the reviewing process were ably supported by the Web-based EasyChair system (http://www.easychair.org/). The program chairs would like to extend our appreciation to Andrei Voronkov, who developed EasyChair and is the leading force behind its continued development. His personal attention to our conference greatly facilitated managing the volume of reviews and discussions amongst the Program Committee. Finally, we would like to recognize the importance of the gentle guidance of the GPCE Steering Committee. Their long-term dedication is the core that binds together this research community.

July 2005

Robert Glück
Michael Lowry

Organization

General Chair

Eugenio Moggi (Genoa University, Italy)

Program Committee Co-chairs

Robert Glück (University of Copenhagen, Denmark)
Michael Lowry (NASA Ames Research Center, USA)

Program Committee

Don Batory (University of Texas, USA)
Ira Baxter (Semantic Designs, USA)
Cristiano Calcagno (Imperial College London, UK)
Prem Devanbu (University of California at Davis, USA)
Ulrich Eisenecker (University of Leipzig, Germany)
Tom Ellman (Vassar College, USA)
Robert Filman (NASA Ames Research Center, USA)
Zhenjiang Hu (University of Tokyo, Japan)
Patricia Johann (Rutgers University, USA)
John Launchbury (Galois Connections Inc., USA)
Anne-Françoise Le Meur (University of Science and Technology Lille, France)
Hong Mei (Peking University, China)
Nicolas Rouquette (NASA Jet Propulsion Lab, USA)
William Scherlis (Carnegie Mellon University, USA)
Yannis Smaragdakis (Georgia Institute of Technology, USA)
Walid Taha (Rice University, USA)
Todd Veldhuizen (Chalmers University of Technology, Sweden)

Publicity Chair

Eelco Visser (Utrecht University, The Netherlands)

Workshop and Tutorial Chairs

Andrew Malton (University of Waterloo, Canada)
Jeff Gray (University of Alabama at Birmingham, USA)

Local Arrangements Chair

Tarmo Uustalu (Institute of Cybernetics, Estonia)

Additional Referees

Alexander Ahern
Robert L. Akers
Olivier Barais
Josh Berdine
Christie Bolton
John Tang Boyland
Dolores Diaz
Dan Dougherty
Laurence Duchien
Bernd Fischer
Aaron Greenhouse
Timothy J. Halloran
William Harrison
Jim Hook
Liwen Huang
Shan Shan Huang
Samuel Kamin
Paul Kelly
Oleg Kiselyov
Julia Lawall
Christopher League
Daan Leijen
Dongxi Liu
Jan-Willem Maessen
Michael Mehlich
Eugenio Moggi
Shin-Cheng Mu
Keisuke Nakano
Matthias Neubauer
Emir Pasalic
Renaud Pawlak
Nicolas Pessemier
Amr Sabry
Isao Sasano
Lionel Seinturier
Douglas Smith
Andreas Speck
Franklyn Turbak
Geoffrey Washburn
Tetsuo Yokoyama
Nobuko Yoshida
Hongjun Zheng
David Zook

Sponsors

GPCE 2005 was sponsored by ACM SIGPLAN, in cooperation with ACM SIGSOFT. We gratefully acknowledge the support of Utrecht University in hosting the conference Web site, and the Institute of Cybernetics, Tallinn, for handling the local arrangements and registration.

¥566.40元

Table of Contents

Object-Oriented Reengineering Patterns An Overview

Oscar Nierstrasz[1], Stéphane Ducasse[2], and Serge Demeyer[3]

[1] Software Composition Group, University of Bern, Switzerland
[2] Laboratoire d'Informatique, Systèmes, Traitement de l'Information, et de la Connaissance, Université de Savoie, France
[3] Lab On REengineering, University of Antwerp, Belgium

Abstract. Successful software systems must be prepared to evolve or they will die. Although object-oriented software systems are built to last, over time they degrade as much as any legacy software system. As a consequence, one must invest in reengineering efforts to keep further development costs down. Even though software systems and their business contexts may differ in countless ways, the techniques one uses to understand, analyze and transform these systems tend to be very similar. As a consequence, one may identify various *reengineering patterns* that capture best practice in reverse- and re-engineering object-oriented legacy systems. We present a brief outline of a large collection of these patterns that have been mined over several years of experience with object-oriented legacy systems, and we indicate how some of these patterns can be supported by appropriate tools.

1 Introduction

A *legacy software system* is a system that you have *inherited* and is *valuable* to you. Successful (*i.e.*, valuable) software systems typically evolve over a number of years as requirements evolve and business needs change. This leads to the well-documented phenomenon that such systems become more *complex* over time, and become progressively harder to maintain, unless special measures are taken to simplify their architecture and design [13].

Numerous problems manifest themselves as a legacy system begins to turn into a burden. First of all, *knowledge* about the system deteriorates. Documentation is often missing or obsolete. The original developers or users may have left the project. As a consequence, inside knowledge about the system may be missing. Automated tests that document how the system functions are rarely available.

Second, the *process* for implementing changes ceases to be effective. Simple changes take too long. A continuous stream of bug fixes is common. Maintenance dependencies make it difficult to implement changes or to separate products.

Finally, the *code* itself will exhibit various disagreeable symptoms. Large amounts of duplicated code are common, as are other "code smells" such as violations of encapsulation, large, procedural classes, and explicit type checks.

Concretely, the code will manifest *architectural problems* such as improper layering and lack of modularity, as well as *design problems* such as misuse of inheritance, missing inheritance and misplaced operations. Excessive build times are also a common sign of architectural decay.

Since the bulk of a (successful) software system's life cycle is known to reside in maintenance, and "maintenance" is known to consist largely in the introduction of new functionality [14], identifying and resolving these problems becomes critical for the survival of legacy systems.

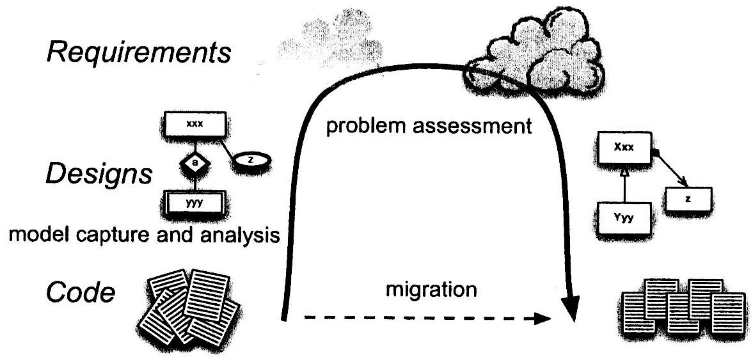

Fig. 1. The Reengineering life cycle

To this end, it is useful to distinguish *reverse engineering* from *reengineering* of software systems [2]. By "reverse engineering", we mean the process of analyzing a software system in order to expose its structure and design at a higher level of abstraction, *i.e.*, the process of extracting various *models* from the concrete software system. By "reengineering" we refer to the process of transforming the system to a new one that implements essentially the same functional requirements, but also enables further development.

The process of reverse- and re-engineering consists of numerous activities, including architecture and design recovery, test generation, problem detection, and various high and low-level refactorings. In Figure 1 we see an ideal depiction of the reverse- and re-engineering life cycle [3,10].

Although the motivations for reengineering a legacy system may vary considerably according to the business needs of the organization, the actual technical steps taken tend to be very similar. As a consequence, it is possible to identify a number of generally useful *process patterns* that one may apply while reverse- and re-engineering a legacy system. We provide a brief overview of these patterns in Section 2. By the same token, there exist various tools that can help support the reengineering process. In Section 3 we present a brief outline of some of the tools we have developed and applied to various legacy systems.

2 Reengineering Patterns

The term "pattern" used in the context of software usually evokes the notion of "design patterns" — recurring solutions to design problems. *Reengineering patterns* are not design patterns, but rather *process patterns* — recurring solutions to problems that arise during the process of reverse- and re-engineering.

We distinguish patterns from "rules" or "guidelines" because each pattern must be interpreted in a given context. Patterns are not applied blindly, but entail tradeoffs. Just as one would never deliberately implement a software system applying all of the GOF patterns [7], one should not blindly apply reengineering patterns without considering all the consequences.

We were able to mine a large number of reengineering patterns during the course of FAMOOS, a European project[1] whose goal was to support the evolution of first-generation object-oriented software towards object-oriented frameworks. FAMOOS focussed on methods and tools to analyse and detect design problems in object-oriented legacy systems, and to migrate these systems towards more flexible architectures. The main results of FAMOOS are summarized in the FAMOOS Handbook [4] and in the book "Object-Oriented Reengineering Patterns" [3].

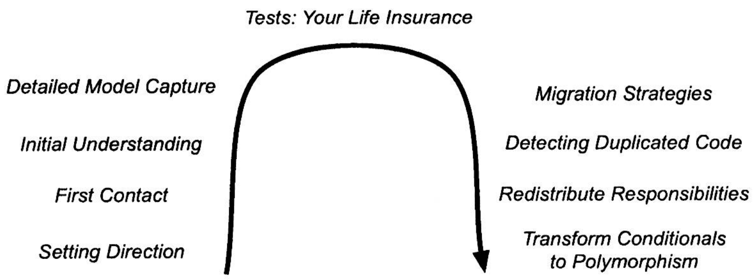
Tests: Your Life Insurance
Detailed Model Capture
Initial Understanding
First Contact
Setting Direction
Migration Strategies
Detecting Duplicated Code
Redistribute Responsibilities
Transform Conditionals to Polymorphism

Fig. 2. Reengineering pattern clusters

In Figure 2 we see how various clusters of reengineering patterns can be mapped to our ideal reengineering life cycle. Each name represents a collection of process patterns that can be applied at a particular stage during the reengineering of a legacy system.

Setting Direction contains several patterns to help you determine where to focus your re- engineering efforts, and make sure you stay on track. *First Contact* consists of a set of patterns that may be useful when you encounter a legacy system for the first time. *Initial Understanding* helps you to develop a first simple model of a legacy system, mainly in the form of class diagrams.

[1] ESPRIT Project 21975: "Framework-based Approach for Mastering Object-Oriented Software Evolution". `www.iam.unibe.ch/~scg/Archive/famoos`

Detailed Model Capture helps you to develop a more detailed model of a particular component of the system. *Tests: Your Life Insurance* focusses on the use of testing not only to help you understand a legacy system, but also to prepare it for a reengineering effort. *Migration Strategies* help you keep a system running while it is being reengineered, and increase the chances that the new system will be accepted by its users. *Detecting Duplicated Code* can help you identify locations where code may have been copied and pasted, or merged from different versions of the software. *Redistribute Responsibilities* helps you discover and reengineer classes with too many responsibilities. *Transform Conditionals to Polymorphism* will help you to redistribute responsibilities when an object-oriented design has been compromised over time.

Since a detailed description of the patterns is clearly out of the scope of a short paper, let us just briefly consider a single pattern cluster. *First Contact* consists of patterns that can be useful when first encountering a legacy system. There are various *forces* at play, which one must be conscious of. In particular, legacy systems tend to be *large* and complex, so it will be difficult to get an overview of the system. *Time is short*, so it is important to gather quality information quickly. Furthermore, *first impressions are dangerous*, so it is important not to rely on a single source of information.

One has various resources at hand: the source code, the running system, the users, the maintainers, documentation, the source code repository, the changes log, the list of bug requests, the test cases, and so on. Even if some of these are missing or unreliable, one must take care to not reject anything out of hand.

In Figure 3 we see a map of the patterns in this cluster, and how they relate to each other. As with each pattern cluster, patterns support each other to resolve the forces at play. The *First Contact* cluster resolves the forces by balancing what you learn from the users and maintainers with what you learn from the source code.

In Figure 4 we see a capsule summary of one of the better-known patterns of this cluster. The *name* is typically an action to be performed, that expresses the key idea of the pattern. Not every pattern is always relevant in every context, so one must be clear about the *intent* of each pattern, the *problem* it solves, the key idea of the *solution*, and the *tradeoffs* entailed. In this particular pattern, the context of a demo is used as a device to help the user to focus on concrete rather than abstract qualities of the application, while communicating typical use cases and scenarios to the engineer. Each pattern may also include *hints*, *variants*, *examples*, *rationale*, *related patterns*, and an indication of *what to do next*. *Known uses* are very important, since only established best practices can truly be considered "patterns".

3 Reengineering Tools and Techniques

It is easy to put too much faith into tools. For this reason the reengineering patterns put more emphasis on process than tools. (As a popular saying puts it: "A fool with a tool is still a fool.")