JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 9, Special Issue: TOOLS June 2007, October 2007

Contents

Page

Editorial 11

PAPERS

Aspect Refinement – Unifying AOP and Stepwise Refinement 13
By Sven Apel, Christian Kästner, Thomas Leich and Gunter Saake
Stepwise refinement (SWR) is fundamental to software engineering. As aspectoriented programming (AOP) is gaining momentum in software development, aspects should be considered in the light of SWR. In this paper, we elaborate the notion of aspect refinement that unifies AOP and SWR at the architectural level. To reflect this unification to the programming language level, we present an implementation technique for refining aspects based on mixin composition along with a set of language mechanisms for refining all kinds of structural elements of aspects in a uniform way (methods, pointcuts, advice). To underpin our proposal, we contribute a fully functional compiler on top of AspectJ, present a non-trivial, medium-sized case study, and derive a set of programming guidelines.

Direct Semantics of Extended State Machines 35
By K. Lano and D. Clark
In this paper we present semantic profiles for UML 2 state machines, which are based directly upon the structure of a state machine model, without requiring flattening or other transformations on these models. The approach addresses many of the semantic problems identified for state machines, and provides a basis for semantic analysis and proof of behavioural compatibility between state machines.

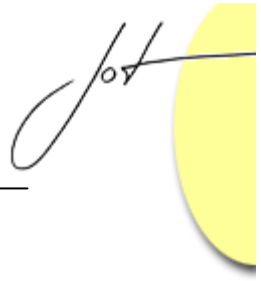
Update Transformations in the Small with the Epsilon Wizard Language 53
By Dimitrios S. Kolovos, Richard F. Paige, Fiona A.C. Polac and Louis M. Rose
We present the Epsilon Wizard Language (EWL), a tool-supported language for specifying and executing automated update transformations in the small based on existing model elements and input from the user. We discuss on EWL's requirements and relevant design decisions, as well as the infrastructure upon which the language

has been developed. We also provide concrete working examples to demonstrate how EWL can be used to automate the process of constructing and refactoring models.

Composing Tests from Examples
By Markus Gaelli, Rafael Wampfler and Oscar Nierstrasz
Understanding and maintaining complex software systems is a difficult task. In principle, tests can be a good source of information about how the system works. Unfortunately, tests are frequently unstructured and disconnected from each other and from their units under test. We propose a new approach to organizing unit tests in which tests produce examples of their units under tests which also can be reused for composing higher-level tests. The approach is based on the Eg meta-model, which classifies tests according to their granularity and their goals. We have developed the EgBrowser, an experimental tool for specifying tests that conform to the Eg metamodel while keeping track of the connection between tests themselves and their units under test. Initial usability studies suggest that the approach is both easy to learn and more efficient than the programmatic approach to developing tests. 71

DEUCE : A Declarative Framework for Extricating User Interface Concerns
By Sofie Goderis, Dirk Deridder, Ellen Van Paesschen and Theo D'Hondt 87
Evolving a software system not only affects the source code responsible for the core application, but also the user interface. Unfortunately user interface code is often scattered through and entangled with the application code. In large and complex user-interfaces, this tangling renders the implementation complex and hard to maintain. Currently, the application needs to perform both the necessary changes to the user-interface (e.g. disabling other buttons, propagating events, etc.) as well as invoke the required application logic. The Deuce framework (Declarative User Interface Concerns Extrication) intends to reduce the complexity of user-interface implementations by applying separation of concerns on three UI concerns : presentation logic, business and data logic, and connection logic. It does so by using a declarative meta-language (SOUL) on top of an object oriented language (Smalltalk) such that an adequate language is provided to describe the entire structure and behaviour of the user-interface, as well as to link it with the application.

Integrating Object Teams and OSGi: Joint Efforts for Superior Modularity 105
By Stephan Herrmann and Marco Mosconi
In central fields of software engineering, there are two competing directions of research. Component frameworks and advanced



programming languages both seek to improve fundamental software quality properties, most notably: modularity. While both directions have produced specific benefits, harvesting these benefits still requires a trade-off in selecting one of the two technologies. In this paper, we present the integration of the aspect-oriented programming language ObjectTeams/Java into the OSGi component framework. By combining these technologies, the design space for modular architectures of components and applications is enriched with fundamentally new options. As a result, the best of both worlds is available for re-using software modules at realistic scales and for evolving systems over significant periods of time.

Reuseware – Adding Modularity to Your Language of Choice
By Jakob Henriksson, Jendrik Johannes, Steffen Zschaler and Uwe Aßmann 127
The trend towards domain-specific languages leads to an ever-growing plethora of highly specialized languages. Developers of such languages focus on their specific domains rather than on technical challenges of language design. Generic features of languages are rarely included in special-purpose languages. One very important feature is modularization, the ability to formulate partial programs in separate entities, composable into a complete program in a defined manner. This paper presents a generic approach for adding modularity to arbitrary languages, discussing the underlying concepts and presenting the Reuseware Composition Framework. We walk through an example based on Xcerpt, a Semantic Web query language.

Type Access Analysis:
Towards Informed Interface Design 147
By Friedrich Steimann and Philip Mayer
Programs designed from scratch often start with just a set of classes. Classes can be instantiated and so deliver the objects that are the carriers of information and function. In languages like JAVA and C++, classes also define types, so that they are sufficient to write a fully functional, type-checked program. Abstract classes and interfaces, which cannot be used for object creation, but which can serve to structure and decouple the code, are then either added later (as a result of refactoring) or never. One impediment to designing and introducing such type abstractions (generalizations) retroactively is that it is unclear how they can be used in a program, or what they should contain in order to be usable. However, this knowledge is, so we argue, completely contained in the program — it only needs to be unveiled. With our Type Access Analyzer (TAA) tool, we collect information useful for the design of type abstractions (abstract classes and interfaces) and their use, and present it to the developer for performing type-related refactorings in an informed manner.

A Metamodel Independent Approach to Difference Representation 165
By Antonio Cicchetti, Davide Di Ruscio and Alfonso Pierantonio
It is of critical relevance that designers are able to comprehend the various kinds of design-level modifications that a system undergoes throughout its entire lifecycle. In this respect, an interesting and useful operation between subsequent system versions is the model difference calculation and representation. In this paper, a metamodel independent approach to the representation of model differences which is agnostic of the calculation method is presented. Given two models which conform to a metamodel, their difference is conforming to another metamodel derived from the former by an automated transformation. Difference models are first-class entities which induce transformations able to apply the modifications they specify. Finally, difference models can be composed sequentially and in parallel giving place to more complex modifications.

Formal and Tool Support for Model Driven Engineering with Maude 187
By J. Rául Romero, José E. Rivera, Francisco Durán and Antonio Vallecillo
Models and metamodels play a cornerstone role in Model-Driven Software Development. Although several notations have been proposed to specify them, the kind of formal and tool support they provide is quite limited. In this paper we explore the use of Maude as a formal notation for describing models and metamodels. Maude is an executable rewriting logic language specially well suited for the specification of object-oriented open and distributed systems. We show how Maude offers a simple, natural, and accurate way of specifying models and metamodels, and offers good tool support for reasoning about them. In particular, we show how some basic operations on models, such as model subtyping, type inference, and metric evaluation, can be easily specified and implemented in Maude, and made available in development environments such as Eclipse.

Hygienic methods — Introducing HygJava 209
By Jaroslaw D. M. Kúsmierek and Viviana Bono
One of the base concepts of object-oriented programming is that of “method”.
In languages supporting inheritance, this concept is realized by three different actions: (i) the introduction of a new method; (ii) the implementation/override of an existing method; (iii) the method call.
The bindings between (ii) and (i), and between (iii) and (i) are typically based on method names, which are not guaranteed to be unique, thus such bindings might cause some ambiguities. As a result, modifications of existing code may cause errors in some other parts of the code,

especially in programs written by third party developers; overall, a programmer cannot predict the moment in the execution when such ambiguities will arise.
In this paper, we describe the nature of these problems and propose a general mechanism to overcome ambiguities in a safe, straightforward, and flexible way. To study the details of this mechanism, and make the reader more familiar with it, we show how to apply this mechanism to Java, and also to a mixin-oriented language called MixedJava.

Rule-based Assessment of Test Quality 231
By Stefan Reichhart,Tudor Girba and Stephane Ducasse
With the success of agile methodologies more and more projects develop large test suites to ensure that the system is behaving as expected. Not only do tests ensure correctness, but they also offer a live documentation for the code. However, as the system evolves, the tests need to evolve as well to keep up with the system, and as the test suite grows larger, the effort invested into maintaining tests is a significant activity. In this context, the quality of tests becomes an important issue, as developers need to assess and understand the tests they have to maintain. In this paper we present TestLint, an approach together with an experimental tool for qualifying tests. We define a set of criteria to determine test quality, and we evaluate our approach on a large sample of unit tests found in open-source projects.

Applying Triple Graph Grammars For Pattern-Based Workflow Model Transformations 253
By Carsten Lohmann, Joel Greenyer, Juanjuan Jiang and Tarja Systä
Workflow and business process modeling approaches have become essential for designing service collaborations when developing SOA-based systems. To derive actual executable business process descriptions from the high-level workflow models, model transformation techniques can be used. Various service composition and business process languages are available for describing the executable processes. They have been developed having slightly different aims and requirements in mind. They do, however, share common key constructs, called workflow patterns that recur in descriptions given in these languages.
We propose a model-driven approach for transforming workflow models given as UML activity diagrams into service composition descriptions. This paper will show how to realize a transformation from UML to BPEL and XPDL with a technology based on Triple Graph Grammars (TGGs). TGGs allow structural relationships between the different model elements to be elegantly expressed in graphical, declarative rules. We will show, in particular, how the commonly known workflow patterns recurring in the different business process

languages can act as a guideline for designing the transformation rules. Based on the experiences in this application domain, we furthermore outline ways to enhance the usability and applicability of TGG for this purpose.

Sub-Method Reflection 275
By Marcus Denker, Stéphane Ducasse, Adrian Lienhard and Philippe Marschall
Reflection has proved to be a powerful feature to support the design of development environments and to extend languages. However, the granularity of structural reflection stops at the method level. This is a problem since without sub-method reflection developers have to duplicate efforts, for example to introduce transparently pluggable type-checkers or fine-grained profilers. In this paper we present Persephone, an efficient implementation of a sub-method meta-object protocol (MOP) based on AST annotations and dual methods (a compiled method and its meta-object) that reconcile AST expressiveness with bytecode execution. We validate the MOP by presenting TreeNurse, a method instrumentation framework and TypePlug, an optional, pluggable type system which is based on Persephone

Modularizing constructors 297
By Viviana Bono and Jarosław D. M. Kúsmierek
Object-oriented class-based languages provide mechanisms for the initialization of newly created objects. These mechanisms specify how an object is initialized and what information is needed to do so. The initialization protocol is usually implemented as a list of constructors. It is often the case that the initialization protocol concerns some orthogonal properties of objects. Unfortunately, if those properties have more than one option of initialization, the total number of constructors becomes exponential in the number of properties.
In this paper, we propose an alternative protocol. In our approach, instead of defining a list of constructors, it is possible to split blocks of definitions into smaller and composable pieces, in order to obtain reduction in the size of the code, better reusability, more expressiveness, and easier maintenance.

Reflecting on an Existing Programming Language 319
By Andreas Leitner, Patrick Eugster, Manuel Oriol and Ilinca Ciupa,
Reflection has proven to be a valuable asset for programming languages, especially object-oriented ones, by promoting adaptability and extensibility of programs. With more and more applications exceeding the boundary of a single address space, reflection comes in very handy for instance for performing conformance tests dynamically.

The precise incentives and thus mechanisms for reflection vary strongly between its incarnations, depending on the semantics of the considered programming language, the architecture of its runtime environment, safety and security concerns, etc. Another strongly influential factor is legacy, i.e., the point in time at which the corresponding reflection mechanisms are added to the language. This parameter tends to dictate the feasible breadth of the features offered by an implementation of reflection.
This paper describes a pragmatic approach to reflection, consisting in adding introspection to an existing object-oriented programming language a posteriori, i.e., without extending the programming language, compiler, or runtime environment in any specific manner. The approach consists in a pre-compiler generating specific code for types which are to be made introspectable, and an API through which this code is accessed.
We present two variants of our approach, namely a homogeneous approach (for type systems with a universal root type) and a heterogeneous approach (without universal root type), and the corresponding generators. We discuss limitations such as infinite recursion, and compare the code generated with these two approaches by Erl-G, a reflection library generator for the Eiffel programming language, thereby quantifying the benefits of a universal root type. Erl-G is being used by several tools for development in Eiffel.

AspectScope: An Outline Viewer for AspectJ Programs 341
By Michihiro Horie and Shigeru Chiba
This paper presents the AspectScope tool for AspectJ. It displays how aspects affect the existing module interfaces in the program. Because an aspect is not explicitly invoked, some developers claim that it is difficult to understand the behavior of their code within local reasoning. Although this difficulty should be mitigated by appropriate tool support, the support by current tools such as AJDT is not sufficient. We have developed AspectScope for providing another visualization of AspectJ programs so that developers can more easily understand crosscutting structures in their programs.

The Message-Oriented Mobility Model 363
By Jorge Vallejos, Tom Van Cutsem, Elisa Gonzalez Boix, Stijn Mostinckx, Jessie Dedecker and *Wolfgang De Meuter*
Mobile networks composed of devices interconnected by wireless communication media frequently suffer from partitions. If mobile devices depend on software services running on remote devices, such partitions may render the software services unavailable. We propose the use of code mobility to mitigate the unavailability of software services in mobile networks. We discuss the issues of existing mobility

mechanisms, identify four characteristics necessary to support code mobility in mobile networks, and propose a model for code mobility, the Message-Oriented Mobility (MOM) model, that features such characteristics.

ClassSheets - model-based, object-oriented design of spreadsheet applications 383

By Jan-Christopher Bals, Fabian Christ, Gregor Engels and Martin Erwig

Using spreadsheets is the preferred method to calculate, display or store anything that fits into a table-like structure. They are often used by end users to create applications. But they have one critical drawback - they are very error-prone.

To reduce the error-proneness, we purpose a new way of object-oriented modeling of spreadsheets prior to using them. These spreadsheet models, termed *ClassSheets*, are used to generate concrete spreadsheets on the instance level.

By this approach sources of errors are reduced and spreadsheet applications are easier to understand.

A Tool for Supporting and Automating the Development of Component-based Embedded Systems 399

By Rafael L. Cancian, Marcelo R. Stemmer, Alexandre Schulter and Antônio A. M. Fröhlich

Embedded systems are comprised of hardware and software and usually run dedicated applications in environments with highly restricted resources, such as memory constrained devices, microcontrollers with low processing power, and wireless sensors running on batteries. These systems must exactly match applications' requirements, with minimum support. The growth in application complexity and even more strong constraints demand new approaches, methodologies, and tools to assist embedded systems development. Usually, in this domain, reuse of components, architectural transparency, low overhead, and reconfigurability are essential features. The Application-Oriented System Design (AOSD) method was created to deal with these issues, and aims at guiding the development of embedded systems that exactly match application requirements. To deliver each application a tailored run-time support system calls for a good combination of object oriented techniques, the separation of functional and non-functional aspects, some implementation techniques, and a sophisticated tool that helps the developer in managing component configurations and automating the generation of embedded systems.

This paper describes a configuration and system generation tool that is being successfully used with EPOS (Embedded and Parallel Operating System), an OS developed using AOSD. This tool receives the application source-code (using EPOS API) as input and, after a few

mouse clicks, builds the entire computational support, comprised by software and, if hardware is reconfigurable, the FPGA configuration file. This paper also shows that the design of these tools allows them to be used for configuring and building several other systems, not only EPOS. The development of this tool enabled the configuration and generation of several embedded systems instances for several different architectures in an automatic way. To illustrate this process and the tool's usage, this paper describes a case study of the generation of an embedded system that supports a simple audio decoder application.

Improving Alignment of Crosscutting Features with Code in Product Line Engineering 417

By Christine Hundt and Katharina Mehner

Feature models used in product line engineering often include features that crosscut other features. These features cannot be cleanly modularized using object-oriented techniques and are the source of scattering and tangling in implementation modules. This significantly complicates the traceability of features during the development and maintenance of a product line and during the instantiation of a product. This paper proposes a model-driven approach for mapping features to a design in the aspect-oriented role-based language Object Teams. The approach has been evaluated in an industrial case study for developing a security product line that can be applied to several applications using aspect bindings.

REMM-Studio: an Integrated Model-Driven Environment for Requirements Specification, Validation and Formatting 437

By Cristina Vicente-Chicote, Begoña Moros and Ambrosio Toval

In order to integrate requirements into the current Model-Driven Engineering (MDE) approach, the traditional document-based requirements specification process should be changed into a requirements modelling process. To achieve this we propose a requirements metamodel called REMM (Requirements Engineering MetaModel) which includes the elements that should appear in a requirements model (requirements, stakeholders, test cases, etc.) together with the relationships that may appear between them. This metamodel is the basis of the REMM-Studio environment which enables: (1) to build graphical requirements models, (2) to validate them against the metamodel and against a set of additional OCL constraints, and (3) to automatically generate a navigable Software Requirements Specification (SRS) document as the main deliverable of the Requirements Engineering process. REMM-Studio is expected to ease the integration of requirements with other development models (e.g. component models) and to facilitate the validation of the SRS thanks to its navigability.

Pluggable checking and inferencing of nonnull types for Java 455
By Torbjörn Ekman and Görel Hedin
We have implemented a non-null type checker for Java and a new non-null inferencing algorithm for analyzing legacy code. The tools are modular extensions to the JastAdd extensible Java compiler, illustrating how pluggable type systems can be achieved. The resulting implementation is compact, less than 230 lines of code for the non-null checker and 460 for the inferencer. Non-null checking is a local analysis with little effect on compilation time. The inferencing algorithm is a whole-program analysis, yet it runs sufficiently fast for practical use, less than 10 seconds for 100.000 lines of code. We ran our inferencer on a large part of the JDK library, and could detect that around 70% of the dereferences, and around 24% of the method return values, were guaranteed to be non-null.

Incremental Lock Selection for Composite Objects 477
By John Potter and Abdelsalam Shanneb
With the trend towards multi-core processors, support for multi-threaded programming is increasingly important. We are interested in providing development and deployment options to allow programmers to select minimal locks, achieving maximal concurrency, at different levels of granularity within a composite system. We explore local properties of the fixpoint lattice of a Galois connection between exclusion requirements and concurrency potential of a composite object. This allows us to develop incremental algorithms for lock selection. Implemented within integrated development environments, such algorithms will allow programmers to interactively select minimal locks with safety.

OUTLOOK

A brief outlook to the next issue 495