OOPSLA 2006

Portland, Oregon
22-26 October, 2006

Notes by Arne J. Berre, SINTEF/UiO
OOPSLA Topics

- Agile (Alistair)
- Analysis
- Architecture and SOA
- Aspects
- C++
- Collaboration
- Compilation
- Components
- Databases and Transactions
- Design
- Distribution and Concurrency
- Dynamic Languages
- Eclipse
- Frameworks
- Java
- Language Design
- MDA
- Methods
OOPSLA Topics

- Middleware
- **Modeling**
- Organization and Professional Development
- Pedagogy/teaching
- Programming Environments
- Performance
- Software Engineering
- Static Analysis/Types
- Theoretical Foundations
- Testing
- Tools
- **UML**
- Virtual Machines
- Web
Tutorials

- G1 (GPCE): Model Driven Development Basics using Eclipse
- G2 (GPCE): Building Domain Specific Languages with Eclipse and openArchitectureWare
- G4 (GPCE): Building Java Transformations with Stratego/XT
- G5 (GPCE): Engineering Software Factories for Developing Enterprise Applications Using Model-Driven
- G6 (GPCE): Feature Modularity in Software Product Lines
- G7 (GPCE): Generative Software Development
- GP1: Coherence and Community: Exploring Coherence via Group Problem Solving
- GP2: Coherence and Team-building: A "Here and Now" Experience
- PL1: PLoP Pattern Writing Bootcamp
- T01: Use-Case Model Refactoring and Improvement
- T02: Demystifying GCC: Under the Hood of the GNU Compiler Collection
- T03: Squeezing the Last Drop of Value out of your Project Management Strategies
Tutorials

- T04: Dynamic Languages for Statically-Typed Minds
- T05: Information Systems Architecture: Stakeholders, Viewpoints and Perspectives
- T06: The Art of Telling Your Design Story
- T07: Agile Software Development in the Large
- T08: Ajax: Introduction and Architecture
- T09: SOA: From Concepts to Reality
- T12: Test-Driven Development - Hands-on!
- T13: Modular AOP Using Design Rules and XPIs
- T14: Organizational Patterns: Beyond Agility to Effectiveness
- T15: Programmers are from Mars, Customers are from Venus
- T16: Building Service-Oriented Architectures with Web Services
- T17: Introduction to UML 2: A Fair and Balanced Overview
- T18: Making the Most of Eclipse
- T19: Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects
- T20: Making your Methodology Crystal Clear and More
- T21: Find Your Voice
- T22: Generate the Repetitive, Boring Code: How to Write Code Generators
- T23: Building Solid Distributed Enterprise Software Architectures: Architecture Principles and UML
- T25: Refactoring Databases: Evolutionary Database Design
- T26: Rapidly Designing and Testing Great User Interfaces
- T28: Intro to Concurrent Programming in Java 5.0
- T29: Writing Effective Use Cases for Projects Agile and Otherwise
- T30: Making RUP Agile
- T31: A Tour of Responsibility-Driven Design
- T32: Revolutionizing Software Quality through Static Analysis Tools
- T34: Domain-Driven Design: Putting the Model to Work
Tutorials

- T35: Seamless Use of Object-Oriented Models from Requirements to Software Design
- T36: Concurrent Object-Oriented Programming for Modern Architectures
- T38: Introduction to the Eclipse Modeling Framework
- T39: Understanding Security with Patterns
- T40: Ruby on Rails: A Kickstart
- T41: Techniques and Principles for Rolling Your Own Methodology
- T42: Incremental Releases Users and Stakeholders Will Love
- T43: Domain-Driven Design: Strategic Design and Modeling in Large Projects
- T44: Effective Concurrent Programming in Java
- T45: Storytelling with FIT
- T46: Software Architecture: Principles, Strategies, Qualities
- T47: Programming Mobile Devices: An Introduction
- T48: Totally Awesome Computing: Python as a General-Purpose Programming Language
- T49: Model-Driven Development of Distributed Systems
- T50: Enterprise JavaBeans and the Java Persistence API
- T51: Security Patterns and Secure Software Architecture
- T53: An Introduction to C++ Library Functionality in TR1 and Boost
- T55: Agility, for Managers, Programmers, and All
- T56: SOA from an Architectural Viewpoint
Onward!

- A Commensalistic Software System
- Applying a UML-based Agent Modeling Language to the Autonomic Computing Domain
- Collaborative Diffusion: Programming Antiobjects
- Conscientious Software
- Intentional Software
- JDA: A Step Towards Large-Scale Reuse on the Web
- Keeping Track of Crosscutting Requirements in UML Models via Context-Based Constraints
- Model-Based DSL Frameworks
- Pegasus: First Steps Towards a Naturalistic Programming Language
- The Geography of Programming
As agent technology practitioners, some time ago we decided to develop an extension to UML 2.0 that addressed our specific needs, such as modeling autonomicity, proactivity and role-based behavior. We called this extension the Agent Modeling Language (AML) and have recently published the metamodel and specification for public use. In a recent project, we realized that AML could also be applied to the domain of autonomic computing and so decided to publish some of our findings in this paper. AML can be directly used by designers of autonomous and autonomic computing systems to visually model their architectures and behaviors. Herein we provide an overview of the scope, approach taken, the specific language structure and optional extensibility. The core modeling constructs of AML are explained using a series of didactic examples describing the IBM Unity architecture, a well-grounded exemplar of an autonomic system. We thus focus on the features of AML that differentiate it from UML 2.0 with a specific focus on those aspects that support the autonomic principles of self-healing and survivability.

Ivan Trencansky, Whitestein Technologies
Radovan Cervenka, Whitestein Technologies
Dominic Greenwood, Whitestein Technologies
Separation of concerns is a well established technique in software engineering. Similar to how WYSIWYG editors separated the data model that the editor operated on from the editor itself, Intentional Software uses Structured Source Code for source. A projective Structured Editor separates the source format from the notations. Generative Programming is employed to transform the intentional source to executable code. We apply the technique to an example from the domain of syntax equations, and discuss relationships with present techniques and future directions.

- Charles Simonyi, Intentional Software (previously responsible for MS Word/Excel)
- Magnus Christerson, Intentional Software
- Shane Clifford, Intentional Software
Model-based DSL Frameworks

- MDE is more general than the set of standards and practices recommended by the OMG's MDA proposal. The main idea is to consider models as first class entities. In MDE the concept of model designates not only OMG models but a lot of other artifacts like XML documents, Java programs, RDBMS data, etc. Today we observe another evolutionary step. A convergence between MDE and DSL (Domain Specific Language) engineering is rapidly appearing. The notion of domain is common to these two study areas and they share a lot of common goals and practices. In the same way as MDE is a generalization of MDA, the DSL engineering may be viewed as a generalization of MDE.

- Jean Bezivin, INRIA and LINA, University of Nantes
  Frederic Jouault, INRIA and LINA, University of Nantes
  Ivan Kurtev, INRIA and LINA, University of Nantes
  Patrick Valduriez, INRIA and LINA, University of Nantes
Invited Talks

- Linda Northrop  “Scale Changes Everything”
- Guy Steele
- Joshua Bloch
- Martin Rinard
- Philip Wadler
“Scale Changes Everything”

Many systems of the future will be of ultra-large size on one or many dimensions – number of lines of code; number of people employing the system for different purposes; amount of data stored, accessed, manipulated, and refined; number of connections and interdependencies among software components; number of hardware elements to which they interface. They will be ultra-large-scale (ULS) systems. Is the software community ready to tackle ULS systems? Will incremental changes in our current software development and management practices be sufficient?

This talk is based on the results of a year-long study on ULS systems, documented in *Ultra-Large-Scale Systems: The Software Challenge of the Future* (ISBN 0-9786956-0-7).
Joshua Block

- **When in doubt, leave it out.** If you are in doubt whether a feature should be included in the API, leave it out. It's better to add it later if you need it, rather than having functionality that you would like to remove, but cannot because others have started using it.
- **Boilerplate code.** A clean API will not force you to create a lot of boilerplate code. My interpretation of this, is that you should avoid inappropriate intimacy between your code and the client code.
- **Don't violate the principle of least astonishment.** Your functionality, behaviour and conventions should always be the ones that astonishes your users least.
- **Fail fast.** Report errors to the programmer as soon as possible.
- **Avoid fragile base class problem.** If you allow users to extend your classes, document well how your methods are related. If extending the classes in the API is not something that you don't want the users to do, make the classes final.
- **Overload with care**
- **Do not use floating point to represent monetary values.** The thing here, is that you will often not get the values and behaviour you expect.
- **Never use strings if a better option exists.**
- **Use builder pattern to reduce number of method arguments.** A method should have three or less arguments. A higher number makes you forget what they are. Create a builder to gather the arguments into a holder class and pass that to the method.
- **Return empty objects instead of null.** For instance, take a method that returns a set of objects. If a method is unable to do this, return an empty set object instead of a null pointer. This way, you avoid having your caller always have to check for null before using the return value.
- **Don't use exceptions for flow control.** This is a personal "favorite" of mine. I have seen this on several occasions, and it upsets me every time.
- **Only use checked exceptions if there are any way to recover.** If the system cannot recover from the error, there is no point spending time to try to catch the exception in your code. It just clutters your code.
An important design principle was to create a language that would grow over time. To accomplish this, the focus is to keep the language and the corresponding compiler lean, and relying on libraries to extend the language.

Support for unit testing, make (build tool), and Subversion is built into the language (or libraries, I don't remember which).

It uses notation from math as much as possible, since this is what mathematicians know. In order to find notational constructs, they studies what mathematicians wrote on whiteboards, and try to incorporate this into the syntax. Furthermore, they based the syntax on Unicode to incorporate the special characters that mathematicians do. In order to make the code more readable they run it through Latex to get a full mathematical notation. In this respect, Fortress seems to be a kind of a Domain-specific language, or DSL.
Panels

- Objects And Databases
- Ultra-Large-Scale System Panel session
Practitioners reports

- **Applications Crossing Software / Hardware Boundary**
  - OO Techniques Applied to a Real-time, Embedded, Spaceborne Application
  - Secure Trade Lane: A Sensor Network Solution for more reliable and more secure Container Shipments
  - Using Model-Driven Engineering to Complement Software Product Line Engineering in Developing Software

- **Changing the Language of Programming**

- **Enterprise Architecture Modeling**
  - Architectural improvement by use of strategic level Domain-Driven Design (Statoil)
  - Software Architecture at a Large Financial Firm
  - Using Domain-Driven Design to evaluate Commercial Off-The-Shelf software (Statoil)

- **Incremental Software Engineering Techniques**
Demonstrations

- **D01**: Writing Truly Efficient Smalltalk
- **D03**: Web Testing Made Easy
- **D04**: Tools for a Successful Refactoring
- **D05**: A Simple Edit-Time Metaobject Protocol: Controlling the Display of Metadata in Programs
- **D06**: **OO Design Methodology of a DSL using EMF**
- **D07**: DigitalAssets Manager, sharing and managing software development assets.
- **D08**: Bringing Ownership Domains to Mainstream Java
- **D09**: **Mutual Satellites: Round-trip modeling for complete applications**
- **D10**: Programming the World with Sun SPOTs
- **D11**: A Visualization of the Frame Representation Language
- **D12**: **Language Integrated Query In Visual Basic 9 And C# 3.0**
- **D13**: Fluid AOP Join Point Models
- **D14**: Aspect Mining For Large Systems
Demonstrations

- D15: DEMOS: A Tool for Declarative Executable Modeling of Object-Based Systems
- D16: Using Framework Interfaces with Design Fragments
- D17: ATL: a QVT-like Transformation Language
- D18: Advanced issue tracker in Smalltalk
- D19: SAFARI: A Platform for Generating Language-Specific IDE Support
- D20: Flexible Event Scheduling for Data-flow Audio Processing
- D21: Integrated Solution Engineering
- D22: The Silver Extensible Java Compiler and Modular Language Extensions
- D23: APProVer: Managing Conditional Compilation Through Object and Aspect-Oriented Modeling
- D24: Refactoring-aware Software Configuration Management
- D25: A Static Analysis for Extracting Runtime Views from Annotated Object-Oriented Code
- D26: Language Integrated Query - Unified Querying across Data Sources and Programming Languages
- D28: Developing Embedded Software Product Lines with AspectC++
Research Papers – session topics

- Session 1: Language Design
- Session 2: Frameworks / tools
- Session 3: Virtual Machines
- Session 4: Performance
- Session 5: Compilation
- Session 7: Concurrency
- Session 8: Generics
- Session 9: Types
- Session 10: Software Engineering
Workshops

- GPCE WS1: First Workshop on Aspect-Oriented Product Line Engineering (AOPLE)
- GPCE WS2: Software Transformation Systems Workshop (STS)
- GPCE WS3: Domain-Specific Aspect Languages Workshop (DSAL)
- GPCE WS4: Generative Programming and Component Engineering for QoS Provisioning in Distributed Systems
- W01: Library-Centric Software Design
- W02: Eclipse Technology eXchange
- W03: Building Software for Sensor Networks
- W04: 1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)
- W05: The 6th OOPSLA Workshop on Domain-Specific Modeling
- W06: Event Driven Architecture
- W07: Fifth "Killer Examples" for Design Patterns Workshop
- W08: Escaped from the Lab: Software Practices in Large Organizations
- W09: Creating an Informative Workspace
- W10: Fourth International Workshop on SOA & Web Services Best Practices
- W11: From Embedded to Enterprise: How to Get Application Portability
- W12: NetBeans plug-in/RCP application development workshop
- W16: Ultra Large Scale Systems
W10: Fourth International Workshop on SOA & Web Services Best Practices

Semantic SOA workgroup

Challenges
Current status
Future directions/challenges
Overview – from Business process to SOA
## State of the Practice (not semantics)

<table>
<thead>
<tr>
<th>(Semantics)</th>
<th>Modeling</th>
<th>Specification, Development</th>
<th>Publication/Discovery</th>
<th>Composition</th>
<th>Deployment/Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data / Information</td>
<td>UML Class diagrams</td>
<td>XML/XMLS</td>
<td>UDDI</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Services/ Function</td>
<td>UML Interfaces</td>
<td>WSDL</td>
<td>UDDI (not commercially)</td>
<td>BPEL</td>
<td>BPEL engine</td>
</tr>
<tr>
<td>3. Rules</td>
<td>UML profile for Rules</td>
<td>SWRL(?)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Processes</td>
<td>BPMN, UML AD</td>
<td>BPEL</td>
<td></td>
<td>BPEL</td>
<td>BPEL engine</td>
</tr>
<tr>
<td>5. NFA (QoS, Perf.)</td>
<td>UML QoS profile</td>
<td>WS-Policy</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## State of the Practice (semantics)

<table>
<thead>
<tr>
<th>(Semantics)</th>
<th>Modeling</th>
<th>Specification Development</th>
<th>Publication /Discovery</th>
<th>Composition. Depl/Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data / Information</td>
<td>UML ODM Ontology pr.</td>
<td>OWL</td>
<td>sem web</td>
<td></td>
</tr>
<tr>
<td>2. Services/ Function</td>
<td>UML and OCL*</td>
<td>OWL-S, WSMO, WSDL-S*</td>
<td>Enterprise service repository, WSRF, Meteor-S*</td>
<td></td>
</tr>
<tr>
<td>3. Rules</td>
<td>UML profile for rules</td>
<td>SWRL, SBVR</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Processes</td>
<td>BPMN UML AD</td>
<td>OWL-S, WSMO, WSDL-S</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. NFA (QoS, Perf.)</td>
<td>UML QoS profile</td>
<td>WS-Policy</td>
<td>Meteor-S*</td>
<td>Meteor-S*</td>
</tr>
</tbody>
</table>
## Future directions/challenges

<table>
<thead>
<tr>
<th>(Semantics)</th>
<th>Modeling</th>
<th>Specification Development</th>
<th>Publication/Discovery</th>
<th>Composition/Depl/Exec</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Data/Information</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2. Services/Function</td>
<td></td>
<td></td>
<td></td>
<td>Subsumption reasoning does not always work</td>
</tr>
<tr>
<td>3. Rules</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4. Processes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5. NFA (QoS, Perf.)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>