SOLoist: A New Java-Based Framework for Rapid Model-Driven Development with Executable UML

Driving the Boundaries of Web Application Development

Dr. Dragan Milicev
Associate Professor, University of Belgrade
Founder, SOL

www.sol.rs, www.soloist4uml.com
Introduction

• The eternal question of software engineering: *how can we develop better software more efficiently?*

• Two of a few principal means to tackle this problem:
  1. Raising the level of abstraction
  2. Reducing accidental complexity
Introduction

- Problem #1: Raising the level of abstraction
  - Bran Selic: “There has been no revolutionary breakthrough in SW engineering since the invention of a compiler” (paraphrased)
  - Main idea: use more abstract languages to express the ideas/concepts/design decisions closer to the problem domain, while the compiler can translate the “program” into a lower representation (ultimately understandable by the machine)
  - Improve expressiveness: say more facts to the machine in fewer words
  - One of the main ideas of Model-Driven Development (MDD): use abstract, high-level architecture/design models instead of (direct) coding
  - A standard modeling language: UML
Introduction

• Raising the level of abstraction – simple example #1:

```
class Department {
    public List<Employee> members = new ArrayList<Employee>;
    ...
}

class Employee {
    public Department dept;
    ...
}
```

Instead of:

```
Department 0..1 assignment * Employee
+dept +members
```

Problems with the expressiveness of the OOP level:

It’s not clear that the two properties are conceptually related (tightly coupled as ends of the same association)
Introduction

• Raising the level of abstraction – simple example #2:
  `aDepartment.members.add(anEmployee);`
  Or
  `anEmployee.dept.set(aDepartment);`
  but no need for both,
  instead of (both):
  `aDepartment.members.add(anEmployee);`
  `anEmployee.dept = aDepartment;`

• Problems with the expressiveness of the OOP level:
  – It is the responsibility of the developer to keep the fields in sync – prone to error
  – More primitive semantics: unidirectional references instead of (bidirectional) links (relationships)
Introduction

• Raising the level of abstraction – simple example #3:

instead of plenty of switch-case code…

Problems with the expressiveness of the OOP level:
  – Difficult to implement and maintain
  – Error-prone
Introduction

- Raising the level of abstraction – simple example #4:

Instead of plenty of code without enforced semantics...

Problems with the expressiveness of the OOP level:

The intended meaning has to be enforced by the executable code and is the responsibility of the developer – more difficult and error-prone than a declarative approach.
Introduction

- Unfortunately, many people are disappointed by UML and MDD. Why?
- Because UML models are widely used as diagrams, simple *informal sketches* of the design that are *not* authoritative specifications with ensured implications in the running implementation.
- Consequence - *rush to code syndrome*: a pervasive unease during the early development phases that requirements definition and design models are “just documentation,” and a conviction that the ‘real work’ has not begun until code is being written.
Introduction

• Solution:
  – Use a modeling language with formal semantics – a profile of UML
  – Develop models with executable semantics

• Consequences:
  – Models are executable artifacts, just as code
  – Models are not just sketches, and are accurate documentation of design decisions
  – No rush-to-code syndrome
  – Models do not have to be complete in order to be executable (a classical compiler is much more rigid)
  – Rapid prototypes can be achieved more easily
Introduction

• Problem #2: Reducing accidental complexity
• *Essential complexity*: an inevitable component of complexity that is inherent to the very problem domain
• *Accidental complexity* arises purely from mismatches in the particular choice of tools and methods applied in the solution
• We cannot avoid essential complexity, but we can (and should!) reduce accidental complexity
Introduction

• Accidental complexity – simple example #1:
• “Every entity must have a primary key (simple, composite or auto-generated):”

```java
@TableGenerator(
    name="employeeGen",
    table="EJB_ORDER_SEQUENCE_GENERATOR",
    pkColumnName="GEN_KEY", valueColumnName="GEN_VALUE",
    pkColumnValue="EMPLOYEE_ID", allocationSize=10)
@Id
@GeneratedValue(strategy=GenerationType.TABLE, generator="employeeGen")
public Long getEmployeeID() { return employeeID; }
```

• Is this all really necessary?
• No. It is accidental complexity caused by a heavy influence of the underlying technology (relational database)
Introduction

- Accidental complexity – simple example #2:
  
  ```java
  @PersistenceContext
  EntityManager em;
  ...
  public Employee createEmployee(Department d) {
    Employee empl = new Employee();
    d.getMembers().add(empl);
    em.persist(empl);
    return empl;
  }
  ```

- The developer *must* be aware of the fact that the Java object and its database representation are separate items that have to be kept in sync.

- …Yet another instance of accidental complexity caused by the influence of the underlying implementation
Introduction

• Accidental complexity – simple example #3:
• For each Web page/form, there should be a backing bean which has to be configured and implemented…

```html
<h:inputText id= "emplName"
    value="#{EmployeeBean.emplName}" />
```
• Move the value from the form (Web page) to a backing bean field, then to an entity field, and back…
• Good, because it keeps the form and the entity “isolated and independent.”
• It is good if it is necessary, but is it always necessary? And are these really independent and isolated, anyway?
• Example: Which components are affected if, e.g.:
  (a) another property is added to the entity and should be displayed in the form,
  (b) an entity’s property has changed its type?
• Yet another instance of accidental complexity…
Introduction

- SOLoist is an open-source framework for model-driven development (MDD) based on UML,
- rapid prototyping and application development,
- execution of object-oriented information systems (OOIS), such as business and other kinds of enterprise database Web applications, based on high-level, executable UML models.
Model Execution in SOLoist

- Build a conceptual UML model in a third-party UML modeling tool:
Model Execution in SOLoist

• Generate the application:
  – Generate Java code using a SOLoist plug-in
  – Generate relational database schema using a SOLoist utility

• Effects without any manual interventions:
  – Direct and transparent object persistence
  – UML action semantics
  – Generic object space browser in SOLoist Application Control Center
Model Execution in SOLoist
Model Execution in SOLoist

- Direct and transparent object persistence, with the full UML semantics and without any additional coding:

```java
Message::send(to: Addressee[*]):
for (Addressee addr : to) {
    this.addressees.add(to);
    to.receive(this);
}
```
Model Execution in SOLoist

• Reduced accidental complexity:
  – No accidental complexity in manual coding and annotations (everything is generated)
  – No need for primary keys in classes
  – No confusion wrt. associations
  – Plain and natural coding of actions, without the burden of persistence management coding:

  ```java
  Employee emp = new Employee();
  ...
  emp.dept.set(dept);
  ...
  emp.destroy();
  ```

  `emp` is already persistent.
  No need for “persist” or “save”.

  `dept.members` immediately includes `emp`.

  `emp` is removed from the object space (database).
GUI Building in SOLoist

• **GUI Components**: elementary GUI building blocks
• Hierarchical containment of GUI components, as usual
• However, GUI components are ordinary SOLoist objects with UML semantics, and can be persisted (in the database) and are interpreted by the GUI runtime engine
• Consequences: GUI can be configured at runtime, dynamically, and even interactively
GUI Building in SOLoist

• A new GUI building paradigm:
  – pins and bindings
  – data flow
  – data retrieval
  – notification
GUI Building in SOLoist

- Components are directly bound to the elements of the object space:

```java
GUIElementComponent nameSlotComponent = GUISlotEditorComponent.create(detailsPanel, Department.FQPropertyNames.name, ...);

GUIFactory.createBinding(departmentTreeView.value, nameSlotComponent.element)
```
Conclusions

• SOLoist Data Sheet:
  – Development:
    • UML modeling tool: StarUML™ (an open-source, free UML modeling tool) and others under development
    • Target language: Java
    • IDE: Eclipse
  – Execution:
    • Application server: Tomcat
    • DBMS: MySQL, PostgreSQL, Oracle, MS SQL Server, Sybase, or any other SQL-92 compatible RDBMS
    • GUI framework: Google Web Toolkit (GWT)

• Dual licensing: GPL (for open-source projects) and commercial (for proprietary products)
Conclusions

• Theoretical background:

SOLoist is based on an executable UML profile (named OOIS UML) thoroughly described in the book:

Dragan Milicev,
*Model-Driven Development with Executable UML*

Wiley (Wrox), July 2009, 816 pages,
Conclusions

• Empirical background:
  – Ten years of successful application of the overall approach
  – Dozens of industrial projects done with the framework
  – This is the 4\textsuperscript{th}, completely redesigned and reconstructed version of the framework (for Java and Web)
  – The new framework has already been successfully applied to a few complex industrial systems
  – Size of applications: a few thousands of GUI components (SACC: \~5000 GUI components)
Conclusions
Conclusions
Conclusions

- Visit www.soloist4uml.com to see how it works! (Demos, tutorials, news, etc.)
- Q&A
- Thank you for your attention