The “Write Once, Deploy N” MDA Case Study

Pieter Van Gorp, Dirk Janssens
Formal Techniques in Software Engineering
Pieter.VanGorp@ua.ac.be, Dirk.Janssens@ua.ac.be
http://www.fots.ua.ac.be/
Presentation Roadmap

- Problem, Context and Solution in a nutshell
- Solution Space: MDE & MDA
- The “WODN” MDA Case Study
  - Concrete example requiring more reuse in code generators
  - Could be used as benchmark for presented techniques on/after workshop...
- Conclusions & Future Work
Part I: Problem, Context and Solution in a nutshell
Problem Statement

Optimizing the performance of distributed database applications is hard to combine with middleware vendor independence since cache, transaction and cluster configuration is database and application server specific.
Vendor Independence?
EJB Essentials

- **Distributed Server Components**
  - Memory Management, Persistence, Caching, Connection Management, Transaction Management, Object Distribution, ...

- **Enterprise JavaBeans**
  - Java sources inherit / implement certain interfaces
    - Remote Home Intf, Local Home Intf
    - Remote Bean Intf, Local Bean Intf
    - Bean Class (focus on business logic)
  - Deployment Attributes for code generators / compilers
    - XML Deployment Descriptors
  - Final component accesses server specific API
    - Callbacks to Bean Class
Proposed Solution

1. **Model** platform independent business components
   ✓ PIMs

2. **Generate** platform specific implementations
   ✓ May be interactive wizard (e.g. point to database)
   ✓ PSMs, “PSC”

3. **Generate** platform independent “wrapper” code
   ✓ Generate “PIC”

4. **Write** applications using these components
   ✓ Plain Java (depends on your high-level API) or SDM

5. **Analyze** access scenario of such applications
   ✓ Model analysis

6. **Generate** delegation code
Part II:
Solution Space: MDE & MDA
Model-Driven Engineering

- **Definition**
  In MDE, developers use a set of *domain specific* modeling languages with *adaptable* relationships managed by an architect.

- **Goals**
  1. More intuitive software specifications
     ✓ Less experts required
  2. Encapsulate best practices (Performance, Modularity)
     ✓ More productive for developers
     ✓ Stricter architecture conformance
  3. Bypass vendor lock-in
     ✓ Optimizations in mappings instead of specifications
MDA

Definition
- Model-Driven Engineering with UML and MOF
  - UML: Widely known notations for diagrams (Visualize your models)
  - MOF: Repository standard (Store your models)
  - QVT: Model transformation standard
  - M2T: Code template standard

Goal
- Standard MDD
  - Tools
  - Education, ...
- Managing Evolution of Framework Standards
  - Beyond J2EE
    - Best practices as first class programming artifacts before in standard
    - Your company standards for what IBM, BEA, SUN, ... should not standardize
  - Beyond MOF (!)
Part III:
The “WODN”
MDA Case Study
Write Once, Deploy N

- Online Data Access Scenario’s
  - 85% Read for Display:
    - Lazy Loading
    - Invalidations from Writers
    - No Transactions
  - 10% Read–Write:
    - Aggressive Loading
    - Transaction Support
  - 5% Batch Update:
    - Lazy Loading
    - Transaction Support

- A “Deploy 1” Component would need conservative deployment attributes and waste resources!

  ✓ Manual Implementation is tedious => Generate It!
  ✓ Attributes vary per vendor => Generate It!

- Beyond standardized J2EE framework!
3 kinds of DSLs

- Business analysts want to model server components without platform details
  => DSL

- Architects want to encode best practices in code generator with least effort and maximal effect (e.g., round-tripping)
  => DSL

- Application programmers want a stable server component API (for “PIC”)
  => DSL

Future work, collaboration with company
Part III:
Suggested solution to the case study
Existing MDA Tools

Input Model for “contracts” package (PIM)

Generated Source for ContractBean.java (PSC)

```java
/* Autogenerated by AndroMDA (EntityBean.vsl) - do not edit */
package org.andromda.samples.carrental.contracts;
/**...
 * @ejb.interface generate="false" local-class="org.andromda.samples.carrental.contracts.Contract"
 * @ejb.home generate="false" local-class="org.andromda.samples.carrental.contracts.ContractLocalHome"
 * @ejb.pk generate = "false" class = "java.lang.String"
 * @ejb.persistence table-name="CONTRACT"
 */
public abstract class ContractBean
    extends java.lang.Object
    implements javax.ejb.EntityBean {
    /**
     * Get the driver
     * @ejb.interface-method
     * @ejb.relation
     *    name="Contract->Driver"
     *    role-name="Contract->Driver:TheContract"
     *    target-ejb="Driver"
     *    target-ejb-name="Contract->Driver:driver"
     *    target-multiple="true"
     * @jboss.relation
     *     fk-column = "DRIVER"
     *     related-pk-field = "id"
    */
    public abstract org.andromda.samples.carrental.customers.Driver getDriver();
```
package $packagename;

import javax.ejb.EntityContext;
import javax.ejb.RemoveException;

public abstract class ${entityname}BeanImpl extends ${entityname}Bean {
    private EntityContext context;

    public void setEntityContext(EntityContext ctx)
    {
        //Log.trace("${class.name}Bean.setEntityContext...");
        context = ctx;
    }

    public void unsetEntityContext()
    {
        //Log.trace("${class.name}Bean.unsetEntityContext...");
        context = null;
    }

    public void ejbRemove() throws RemoveException
    {
        //Log.trace(
        //    "${class.name}Bean.ejbRemove...");
    }
}


Pieter Van Gorp, The “Write Once, Deploy N” MDA Case Study

Code Templates

Too Abstract Input Models => Complex scripting
Evolving today’s MDA tools

Problem

Input Metamodel (UML) is too General Purpose
(Too abstract for code generation)

Solution:
1. Code Templates on very concrete Metamodels
   ✓ Refactor input metamodel to stack of metamodels
2. Solution: Stepwise refinement
   ✓ Refactor code templates to model transformations
Domain Specific Metamodels

- Metamodel for business analysis
- Metamodel for transactional caching
- Metamodel for object-to-relational mapping
- Metamodel for WL EJB
- Metamodel for JBoss EJB
Metamodels for “WODN” (I/III)

Note

Reuse
Metamodels for “WODN” (II/III)
Metamodels for “WODN” (III/III)
Recall: 3 kinds of DSLs

- Business analysts want to model **server components** without platform details
  => DSL

- Architects want to encode best practices in **code generator** with least effort and maximal effect (e.g. round-tripping)
  => DSL

- Application programmers want a **stable server component API**
  => DSL

Future work
Reuse requirements for Evolution and of DSLs?
- Reusable aspect modules
  - Contract: maintain consistency relation defined between metamodels
  - Parallellism to manage complexity
  - Sequencing to enable reuse of refinement of WODN pattern across JBoss and WebLogic cartridge
    - “Pipes & Filters”
  - Reuse with specialization of individual transformations
    - “Polymorphism”
    - Confirms observation from Marjan Mernik, Xiaoqing Wu, Barrett R. Bryant
Code Templates with Reuse & Specialization (I/II)

abstract Transformation Components2xDoclet
   from: {TransactionalCaching, ORDB},
   to: {Text}
{

   Rule Generate() {
      postcondition:
         TransactionalCaching::Component.allInstances->forall(
            c | ORDB::Entity.allInstances->forall(
               e | (c.Classifier = e.Classifier)
               implies this^Join2JavaFile(c, e)
          )
   )
}

   Rule Join2JavaFile (TransactionalCaching::Component c,
                     ORDB::Entity e) {
      // code template fragment for Java imports
      // code template fragment for conventional Javadoc
      #call Join2ClassTags(c,e);
      ...
      // code template fragment for iterating over methods
      ...
      #call Join2MethodTags(c,e);
      ...
      ...
   }

}
Code Templates with Reuse & Specialization (II/II)

Transformations ~ Classes
Rules ~ Methods

- Abstract transformations and rules
- Inheritance and Overriding
- Polymorphism

Transformation Components2WL

```
inherit Components2xDoclet {

// Join2JavaFile inherited, not overriden

Rule Join2ClassTags (TransactionalCaching::Entity e1, ORDB::Entity e2) {
    #call super.Join2ClassTags(e1,e2);

    // code template for WebLogic specific xDoclet class
    // tags like @weblogic.persistence, @weblogic.cache,
    // @weblogic.invalidation-target, ...
}

...}
```
Model to Model Transformations as reused generator components

!!! Conflict Resolution (in first increment, during maintenance, ...) !!!

Transformation Data2TransactionableCaching
from: {Data},
to: {TransactionableCaching} {
...
Rule Entity2RO_Component () {
  postcondition:
  Entity.allInstances->forAll(e |
    Component.allInstances->exists(c |
      e.Classifier = c.Classifier and
      c.lockingStrategy = LockingStrategy::noLock and
      c.transactionDemarcation = TransactionDemarcation::optional
    )
  )
}
...
}
Part IV:
Conclusions
& Future Work
Conclusions

- **Reuse in DSL MMs**
  - Data (PIM), TransactionalCaching (PSM), … reuse from UML Core MM
  - Java (or C#, or sequence diagrams, ...) syntax reused for application developers

- **DSL supporting code generator evolution:**
  - **M2C**
    - Integration with code templates
    - Inheritance with overriding (reuse, specialize)
  - **M2M**
    - **Engine** based on Design By Contract (OCL’)
      - Generate constructive code from declarative (logic) rules (auto-satisfy postcondition)
      - Framework for launching manually written reconciliation code

- Code generation and architectural style checking can be integrated
- Activity Diagram & Class Diagram useful for documenting code generator design!
- Link to MDA: QVT Standard!
Future work on the case study: Refactoring Code Generators...

- **PIM**
  
  A model is said to be independent of a set of platforms
  
  1. if its metamodel abstracts from those platforms and
  
  2. if for each abstracted platform there is a sequence of mapping techniques from its metamodel to a metamodel describing this platform

- **New Platforms**
  
  1. New Mappings
     - Reuse of existing mappings is desirable
     - Refactor platform refinements
       - Remove Duplication
       - Improve Simplicity
       - ...
     - Transformation Language Requirements
       - Inheritance with overriding
       - Stepwise Refinement (PIM and PSM per level)

  2. Adapt MM of PIM
     - Required for unanticipated platform characteristics
     - Backward compatibility with existing mappings!