Model Driven Architecture
Code Generation

Prof. Dr. Peter Thiemann

Universität Freiburg

05.07.2006
Contents

1 Code Generation
   • Code Generation Techniques

2 Pragmatics of Code Generation
   • Interfacing Generated with Non-Generated Code
   • Splitting in Technical Subdomains
   • Metaobjects
Code Generation
Reasons

• Performance
• Code size
• Analyzability
• Early detection of errors
• Portability
• Restrictions in the programming language
• Aspects
• Introspection/Reflection
Programs that generate programs (base programs)

Staging of metaprograms

- Independent of base programs (usually earlier)
  - base program and metaprogram are kept separate
  - Examples: MDE generators
- During compilation of the base program
  - static metaprogramming: generated program is unaware of the generation process
  - Examples: C++ preprocessor, C++ templates
- At run-time of the base program
  - dynamic metaprogramming: base program can be extended and modified at run time
  - Examples: metaobject protocol of CommonLisp

Homogeneous vs heterogeneous metaprogramming
Code Generation Techniques
Templates and Filtering

Filter

apply to

Specification

Subset of Specification

apply to

Templates

generated code
Code Generation Techniques
Templates and Filtering/Example

- Code to be generated from templates
- Template variables may be bound to model values
- Example: generate JavaBean from XML specification
Bean specification

```xml
<class name="Person" package="de.unifrei">
  <attribute name="name" type="String"/>
  <attribute name="age" type="int"/>
</class>
```

expected generated code

```java
package de.unifrei;
public class Person {
  private String name;
  public String getName () {return name;}
  public void setName (String name) {this.name=name;}
  private int age;
  public int getAge () {return age;}
  public void setAge (int age) {this.age=age;}
}
```
<xsl:template match="/class">
 package <xsl:value-of select="@package"/>;
 public class <xsl:value-of select="@name"/>
 { <xsl:apply-templates select="attribute"/> } 
</xsl:template>

<xsl:template match="attribute">
 <xsl:variable name="capname" select="concat( translate(substring(@name, 1, 1),
 'abcdefghijklmnopqrstuvwxyz',
 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'),
 substring(@name, 2))"/>
 private <xsl:value-of select="@type"/>;
 <xsl:value-of select="@name"/>;
 public <xsl:value-of select="@type"/>
 get<xsl:value-of select="$capname" /> () 
 {return <xsl:value-of select="@name"/>;} 
 public void set<xsl:value-of select="$capname" />
 (<xsl:value-of select="@type"/> <xsl:value-of select="@name"/>)
 {this.<xsl:value-of select="@name"/>=<xsl:value-of select="@name"/>;}
</xsl:template>
- parse XML and map to user-defined metamodel
- generate code from template and metamodel
A *frame* is an object consisting of slots and a code template

Control iterates over frame instantiation

Exporting of the final frame structure generates the code
Frame specification

.Frame GenNumberElement (Name, MaxValue)
  .Dim vIntQual = (MaxValue > 32767) ? "long" : "short"
  .Dim sNumbersInitVal
  <!vIntQual!> int <!Name!> <?= <!sNumbersInitVal!?>>;
.End Frame

Frame instantiation

.myNumberElem = CreateFrame ("GenNumberElement", "aShortNumber", 100)

Code generation

.Export myNumberElem
**Code Generation Techniques**

API-based Generators

- Client-Program applies API modifies creates or instanceof, corresponds expressed by the base of Grammar AST/CST
- API creates or modifies Code
Code Generation Techniques

Inline Generation

- Configuration
  - Source-Code contains the version-specification
  - Optional preprocess
- Integrated Compiler
  - Source-Code preprocess
    - some versions resolved
    - [else]
    - [all resolved]
- Source-Code all versions resolved
- compilation
- Machine- or Bytecode
Annotate code with active comments

Examples: JavaDoc, XDoclet (supported by Eclipse)

```java
/**
 * @ejb:bean type="Stateless"
 * name="vvm/VVMQuery"
 * local-jndi-name="/ejb/vvm/VVMQueryLocal"
 * jndi-name="/ejb/vvm/VVMQueryRemote"
 * view-type="both"
 */

class VVMQueryBean
{

    /**
     * @ejb:interface-method view-type="both"
     */

    List getPartsForVehicle ( VIN theVehicle ) { 
        return super.getPartsForVehicle ( theVehicle );
    }
}
```
.NET supports attributes that can be attached to parts of C# programs

```csharp
[AttributeUsage(AttributeTargets.All,
    Inherited=true,
    AllowMultiple=true)]
public class MyCustomAttribute: System.Attribute
{
    private string desc;
    private string name;
}
```

Similar feature: Metadata (aka Annotations) in Java5
Many APIs require extra data that must be kept in sync with the code.

Java 5 defines a general purpose annotation facility that permits the definition and use of customized annotation types (generalizing javadoc, @deprecated, transient, etc).

Java 5 annotations consist of:
- syntax for declaring annotation types,
- a syntax for annotating declarations,
- APIs for reading annotations,
- a class file representation for annotations,
- an annotation processing tool (apt).

Annotations do not affect semantics directly, but may influence the execution context.

Annotations can be read from source files, class files, or reflectively at run time.

(Used in EJB 3.0)
/**
 * Describes the Request-For-Enhancement (RFE) that led to the presence of the annotated API element.
 */

public @interface RequestForEnhancement {
    int id();
    String synopsis();
    String engineer() default "[unassigned]";
    String date() default "[unimplemented]";
}
Code Generation Techniques
Excursion: an annotation type use

@RequestForEnhancement(
    id = 2868724,
    synopsis = "Enable time-travel",
    engineer = "Mr. Peabody",
    date = "4/1/3007"
)

public static void travelThroughTime(Date destination) { ... }

- annotation is special kind of modifier
- precedes all other modifiers
## Code Generation Techniques

### Summary

<table>
<thead>
<tr>
<th>Staging</th>
<th>program/metaprogram</th>
<th>generated/manual</th>
</tr>
</thead>
<tbody>
<tr>
<td>Templates and Filtering</td>
<td>before</td>
<td>separate</td>
</tr>
<tr>
<td>Template and Metamodel</td>
<td>before</td>
<td>separate</td>
</tr>
<tr>
<td>Frame Processors</td>
<td>before</td>
<td>separate</td>
</tr>
<tr>
<td>API-based Generators</td>
<td>before/during/after</td>
<td>separate</td>
</tr>
<tr>
<td>Inline Generation</td>
<td>before/during</td>
<td>mixed</td>
</tr>
<tr>
<td>Code Attributes</td>
<td>before/during</td>
<td>(mixed)</td>
</tr>
</tbody>
</table>
Pragmatics of Code Generation

- Which functionality to generate
  - not provided by the platform
  - describable with a DSL
- Generating the final application
  - one build process which regenerates all generated and transformed artifacts
  - without manual intervention or fixing
- Exploiting the model beyond generated code
  - Component tests
  - Simple GUIs
  - Database generation scripts
  - Component configurations
Code Generation
Examples for Configurations

Software
- EJB deployment descriptors
- Behavior for web frameworks like Struts
- Hibernate configurations
- CORBA IDL

Hardware (from deployment diagrams)
- Installation of components on particular machines
- Generation of database tables
- Infrastructure like load balancers
People look at generated code
- They do not trust the generator (initially)
- Debugging
- Checking the configuration of the generator

How to improve acceptance
- Generate comments with information from the models
- Pretty printer for code formatting
- Use “location strings”

Exception: portions optimized for performance
Keep generated and hand-written code separate as much as possible

Use a suitable software architecture for this task
  - what is generated
  - what is written manually
  - how the two are combined
  - tools: interfaces, abstract classes, delegation, design patterns (Factory, Strategy, Bridge, Template Method)

Generated code should be a throw-away product!
Generated vs Non-Generated

Standard Solution: Protected Regions

- complex generation
- not always possible to preserve contents
- weak separation between generated and non-generated code
**Generated vs Non-Generated**

**Alternative Solution: Layered Implementation**

- **Platform Layer**
  - Manually programmed, abstract base class (part of platform)
  - generated, abstract "Middle" class
  - manually implemented concrete class

Three layers of functionality

- identical for all components of a certain kind
- different for each component, but can be generated from the model
- manual implementation
Generated vs Non-Generated
Combination a

- Generated code calls non-generated code
- Advice: only generate a small portion of code at a time and integrate with existing, tested code
Generated vs Non-Generated
Combination b

- Manual code calls generated code
- Requires knowledge of generated code
- May generate dependencies in the build process
Generated vs Non-Generated

Combination c

- **Generated code**
  - inherits from manual code or
  - implements a manual interface

- **Manual code**
  - has some interface to program against
  - can instantiate generated code via Factory pattern
- Manual code inherits from generated code
- Implementation may override generated, generic behavior
- Factory
Generated code inherits from manual code
Invokes operations in manual code
Manual class invokes operations of generated subclass
Template Method pattern
Superclass defines abstract operations
Generated subclass implements these operations
Multi-layer generation may be necessary because of dependencies

First generation step:
- generates set of base classes from certain model elements
- yields “API” for the manual part

Second generation step:
- generation involves all model elements
- references (potentially) manually written parts
### Generated vs Non-Generated

#### Constraints

<table>
<thead>
<tr>
<th>Account</th>
<th><code>context Account.increase (int amount)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>balance: int</code></td>
<td><code>pre: amount &gt; 0</code></td>
</tr>
<tr>
<td><code>increase (int amount): void</code></td>
<td><code>post: balance &gt; balance@pre + amount</code></td>
</tr>
</tbody>
</table>

The programmer should not be able to subvert the model constraints.
Generated vs Non-Generated
Constraints and Protected Regions

// generated
class Account {
    int balance;
    public void increase ( int amount ) {
        assert (amount > 0);
        // check precondition
        int balance_atPre = balance;
        // saved for postcondition
        // --- protected region begin ---

        // --- protected region end ---
        assert (balance = balance_atPre + amount);
        // check postcondition
    }
}

- insufficient protection
- simple inheritance does not help, either
// generated
class Account {
    int balance;
    public final void increase ( int amount ) {
        assert (amount > 0);       // check precondition
        int balance_atPre = balance; // saved for postcondition
        increase_internal (amount);
        assert (balance = balance_atPre + amount); // chk postcondition
    }
    protected abstract void increase_internal (int amount);
}

* no way to subvert the dynamic contract monitoring *

// manually written code
class AccountImpl extends Account {
    protected void increase_internal (int amount) {
        balance += amount;
    }
}
The programmer still has to follow some conventions in manually written code

- Naming conventions
- Class must inherit from a certain generated class and must override certain operations
- Class must implement certain interfaces
- Class must implement certain operations

Check these conventions by generating code that tests them
Generated vs Non-Generated
Consistency Example

// generated
public abstract class SomeGeneratedBaseClass
extends SomePlatformClass {
protected abstract void someOperation ();
public void someOtherOp() {
    someOperation();
}
}

Obligations of the developer

- must inherit from this class
- must override someOperation()
- must name the class ...Impl
- must implement IExampleInterface
public class SomeGeneratedBaseClassImpl
    extends SomeGeneratedBaseClass
    implements IExampleInterface {
    protected void someOperation () {
        // do something
    }
    public void anOperationFromExampleInterface() {
        // ...
    }
}
public abstract class SomeGeneratedBaseClass extends SomePlatformClass {
    // (see above)
    private void dontCallMe () {
        new SomeGeneratedBaseClassImpl();
        // checks that class is present
        // and not abstract
        SomeGeneratedBaseClass a =
            new SomeGeneratedBaseClassImpl();
        // checks that class is subclass
        IExampleInterface x =
            new SomeGeneratedBaseClassImpl();
        // checks that class implements
    }
}
Splitting in Technical Subdomains

- Large systems have a multitude of aspects
- Consequently
  - models become large
  - one single DSL not adequate
  - splitting of tasks for multiple teams hard
- Multiple DSLs with different modeling required
- Generator unifies the different models
- Must communicate via *gateway metaclasses*
Metamodel elements which are used in multiple metamodels

May result in information duplication because multiple definitions of a modeling element must be kept consistent

Solved via proxy elements that reference modeling elements in another metamodel
The Problem

Some applications need model information at runtime
  - for scripting
  - for debugging

How can model information be transported to runtime?

Example: Logging of generated objects should happen with attribute names and attribute values

Reflection helps only partially, it still cannot provide info from the underlying model (before model transformation)

Solution: generate metaobjects that contain the desired information

Association with concrete objects
  - via generated `getMetaObject()` operations
  - via central registry
Metaobjects

Example

```
SomeClass

<<pk>> name: String
{label="Nachname"}
firstname: String
{label="Vorname"}
age: int
{label="Alter", min=0, max=100}
zip: String
{label="PLZ", regex="99999"}

SomeClass

name: String
firstname: String
age: int
zip: String

generated Code

StringAttributeMetaObject

getAttributeNames(): String[]
getAttribute(name: String): AttributeMetaObject

NumAttributeMetaObject

getMin(): int
getMax(): int

AttributeMetaObject

getName(): String
getValue(): Object
setLabel(): String
setValue (Object newVal): void

ClassMetaObject

getAttributeNames(): String[]
getAttribute(name: String): AttributeMetaObject
```