Advancing Automatic Programming: Regeneration, Meta-circularity, and Two-Sided Interfaces

HERWIG MANNAERT

NOVEMBER 27, 2019

Universiteit Antwerpen
• Automatic Programming
• Toward Automatic Regeneration
• Creating Meta-Circular Regeneration
• On Meta-Programming Interfaces
• Concluding Facts and Thoughts
• Questions and Discussion
Overview

• Automatic Programming
  • Concept and Trends
  • Remaining Challenges
• Toward Automatic Regeneration
• Creating Meta-Circular Regeneration
• On Meta-Programming Interfaces
• Concluding Facts and Thoughts
• Questions and Discussion
Automatic Programming

• Automatic programming:
  • The act of automatically generating source code from a model or template
  • Sometimes distinguished from code generation, as performed by a compiler
  • Has always been a euphemism for programming in a higher-level language than was then available to the programmer [David Parnas]

• Also referred to a generative or meta-programming
  • To manufacture software components in an automated way

• It is as old as programming itself:

```java
System.out.println("Hello world.");
```

```java
System.out.println("System.out.println("Hello world.");");
```
The Need for Automatic Programming

- Goal is and has always been to improve programmer productivity
- In general, to manufacture software components in an automated way, in the same way as automation in the industrial revolution, would:
  - Increase programming productivity
  - Consolidate programming knowledge
  - Eliminate human programming errors
- Such an approach is likely to address many long-term issues:
  - Growing amount of software
  - Shortage of computer programmers
  - Increasing amount of software bugs and defects
  - Ever rising IT development and maintenance budgets
The Field of Automatic Programming

• Better known through names/trends like:
  • Model-Driven Architecture (MDA)
  • Model-Driven Engineering (MDE)
  • Model-Driven Software Development (MDSD)
  • Low-Code Development Programs (LCDP)

• The various trends share the use of models to structure requirements and/or to represent domain knowledge:

• The field is still evolving and facing challenges and criticisms:
  • Suitability for large-scale and mission-critical enterprise systems
  • Lack of intermediate representation, pervasive concepts for DSL reuse
  • Either a conceptual gap toward code, or tied to a technological solution
The issues that automatic programming is supposed to address/solve are as relevant and acute as ever:

- Software is growing in size and importance
- Shortage of tens of thousands of programmers
- Multi-trillion lines of code with billions of defects
- Gigantic IT development and maintenance budgets

Automatic programming has not yet delivered on its promises, because we believe at least these fundamental issues need to be addressed:

- Regeneration with support for additive manual code
- Meta-circularity to include the automatic programming code
- Two-sided programming interfaces to support scalable collaboration
• Automatic Programming
• Toward Automatic Regeneration
  • The NS Elements Model
  • The Need for Regeneration
  • Exploring an Implementation
• Creating Meta-Circular Regeneration
• On Meta-Programming Interfaces
• Concluding Facts and Thoughts
• Questions and Discussion
The NS Elements Model

• Automatic programming uses models that are transformed into code
• Code transformation is based on Normalized Systems Theory:
  • Seeks to provide ex-ante proven approach to build evolvable software
  • Founded on systems theoretic stability (BIBO), for the impact of changes
• NST proves a set of principles, that are necessary conditions to avoid instabilities or combinatorial effects:
  • Separation of Concerns
  • Action Version Transparency
  • Data Version Transparency
  • Separation of States
The NS Elements Model

• Automatic programming uses models to represent domain knowledge
• Our intermediate NS Elements model is a General Purpose Language:
  • At the low end of the modelling abstraction, but an executable model
  • Very basic and close to traditional software implementation concepts
  • Aligned with the fundamental primitives of a Von Neumann processor
• NS Elements model defines 5 types of elements:
  • Data element
  • Task element
  • Flow element
  • Connector element
  • Trigger element
Separating Cross-Cutting Concerns
Separating Cross-Cutting Concerns

Persistency

Transaction
The Emergence of Elements
Model Transformation: Data Element

- Invoice Nr
- Invoice Date
- Remote Access
- SOAP Connector
- REST Connector
- Access Control
- Persistency
Model Transformation: Task Element

SendInvoice Element

- SendInvoice
- Remote Access
- Transaction
- Access Control
- REST Connector
- SOAP Connector
- REST Connector
- SOAP Connector
Model Transformation: Recurring Structure
Generating Recurring Structure: Catch 22

• Structure should be recurring, as variations:
  • increase complexity of codebase
  • decrease consistency in behaviour

• Recurring structure may need to vary over time:
  • new insights
  • discovery of flaws
  • changes in technologies

N=100

Structural changes may need to be applied with retroactive effect, but the efforts increase with the frequency of change.

N instances, update every K ➔ #updates = \( \frac{N(N+K)}{2K} \)
Catch 22: The Only Way Out

• Recurrent stable structures are required to limit complexity and to guarantee consistency
• Recurrent stable structures need to be able to adapt over time, to overcome flaws and technology changes
• Additional custom code is inevitable and needs to be maintained across updated stable structures

An automated mechanism is required, providing both code generation or expansion, and regeneration with harvesting and injection.
Integrating the Dimensions of Change

Mirrors
- Order
  - Ref
  - Product
  - ...
- ProcessOrder
- CreateInvoice
- SendInvoice

Skeletons
- Invoice
  - Number
  - Order
  - ...

Utilities
- Access Control
- Persistency

Transaction

Craftings
- Feature 1
- Feature 2
- Feature N

Codebase
Integrating the Dimensions of Variability

Mirrors

Order
-Ref
-Product
...
ProcessOrder

Invoice
-Number
-Order
...
CreateInvoice

SendInvoice

Skeletons

Craftings

Utilities

Transaction

Access Control

Persistency

Codebase

Integrating the Dimensions of Variability
Change Dimension 1: The Mirrors

Mirrors

- Order
  - Ref
  - Product
  - ...

- ProcessOrder
- CreateInvoice
- SendInvoice

Skelonts

Utilities

- Access Control
- Persistency

Transaction

Craftings

Codebase
Change Dimension 2: The Skeletons

MIRRORS

Order
-Ref
-Product
...
ProcessOrder
CreateInvoice
SendInvoice

SKELETONS

Utilities

Access Control

Transaction

Persistency

CRAFTINGS

Codebase
Change Dimension 3: The Utilities

Mirrors
- Order
  - Ref
  - Product
  - ...
- ProcessOrder
- CreateInvoice
- SendInvoice

Skeletions
- Invoice
  - Number
  - Order
  ...

Utilities
- Transaction
- Access Control
- Persistency

Craftings

Codebase
Change Dimension 4: The Craftings

Mirrors
- Order
  - Ref
  - Product
- ProcessOrder
- CreateInvoice
- SendInvoice

Skeletions

Utilities
- Access Control
- Persistency

Transaction

Craftings

Codebase
Overview

• Automatic Programming
• Toward Automatic Regeneration
• Creating Meta-Circular Regeneration
  • The Need for Meta-Circularity
  • Closing the Automatic Meta-circle
  • Exploring an Implementation
• On Meta-Programming Interfaces
• Concluding Facts and Thoughts
• Questions and Discussion
The Power of Circularity

- A transistor is switched by a transistor
- A cell is produced by a cell

- Enables rapid evolution
  - Single point of progress
    - Better transistor $\rightarrow$ better circuits
    - Improved cell $\rightarrow$ improved life forms
  - Collapses/shortcuts the design cycle
    - Even positive feedback or resonance
Meta-Circularity in Software Engineering

• Associated with *Homoiconicity*:
  • Was coined in 1965, by Mooers & Deutsch (TRAC), and traces back to McIlroy
  • Definitions use concepts like “code as data” and “program structure similar to its syntax”, and are often considered to be a bit vague and controversial
  • The concept is often associated with LISP

• The term *Meta-Circular Evaluator*:
  • Was coined by John Reynolds in 1972 for an interpreter
  • It defines each feature of the defined language by using the corresponding feature of the defining language.

• There is nevertheless a widespread belief that this kind of properties *increase the abstraction level and therefore the productivity*
Why Meta-Circularity in Meta-Programming?

- You also have to maintain the meta-code
  - Consists of several modules
  - Is in general not trivial to write
- Will face growing number of implementations:
  - Different versions
  - Multiple variants
  - Various technology stacks
- Will have to adapt itself to:
  - Evolutions of its underlying technology
    - Which even may become obsolete
- **Meta-Circularity**: meta-code that (re)generates itself
Closing the Meta-Circle: Phase 1

Reader classes
Model classes
Control classes
Expander classes
Code Templates

Read / Write
Model
Logic
Control
View

Expand
Application
Closing the Meta-Circle: Phase 2

Prime Radiant → Expand → nsx-prime → Expand → Application

Read / Write
Model
Logic
Control
View

Meta Model → Model
Reader classes
Model classes
Control classes
Expander classes
Code Templates
Closing the MetaCircle: Phase 3

Prime Radiant

Expand

nsx-prime

Model classes

Control classes

Expander classes

Code Templates

Reader classes

Model

Meta Model

Expand Application

Read / Write

Model

Logic

Control

View

Application
Closing the MetaCircle: Phase 3
Artifact = Expansion(Template, Model)
Artifact = Expansion(Template, Model)

TaskElement.xml

ArtifactExpander.xml

```xml
<expander name="TaskInterfaceExpander" xmlns="http://nsx." <taskElement name="PrimeRadiantUpdater">
  <packageName>net.democritus.expander.common.taskElement</packageName>
  <layerType name="SHARED_LAYER"/>
  <technology name="COMMON"/>
  <sourceType name="JAVA"/>
  <elementType name="TaskElement"></elementType>
  <artifactName>$taskElement.name$.java</artifactName>
  <artifactPath>$componentRoot.directory/$artifactSubFolder</artifactPath>
  <isApplicable>true</isApplicable>
  <active value="true"/>
  <anchors/>
  <customAnchors>
    <customAnchor name="custom-imports"/>
    <customAnchor name="custom-methods"/>
  </customAnchors>
</expander>
```

```xml
<packageName>net.democritus.settings</packageName>
<targetClass>net.democritus.settings.NsfBaseDetails</targetClass>
<targetElement component="elements" name="NsfBase"/>
<paramClass/>
<description/>
<taskElementType name="Updater"/>
<transactionType name="noTransaction"/>
<taskOptions>
  <taskOption name="PrimeRadiantUpdater:includeDelegation">
    <value/>
    <taskOptionType name="includeDelegation"/>
  </taskOption>
  <taskOption name="PrimeRadiantUpdater:includePerform">
    <value/>
    <taskOptionType name="includePerform"/>
  </taskOption>
  <taskOption name="PrimeRadiantUpdater:includeRemoteAccess">
    <value/>
    <taskOptionType name="includeRemoteAccess"/>
  </taskOption>
</taskOptions>
</taskElement>
```
Artifact = Expansion(Template, Model)

Artifact: ExpanderMapping.xml

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<mapping xmlns="http://nx.normalizedsystems.org/201806/expanders/mapping/package
<let name="helper" eval="new net.democritus.expander.common.taskElement">
  // <expanderComment>
  |</!-- EXPANSION CONTEXT -->
  <value name="componentName" eval="expansionContext.componentExpansionComponentImport net.democritus.sys.TaskPerformer;">
  <value name="logicSecurity" eval="taskElement.component.getOption('user')">
    <if paramClass && paramClass.packageName.empty()>
      paramClass
    </if>
  </value name="useRemoteEJB" eval="!globalOptionSettings.beanInterface">
  <value name="includePerform" eval="taskElement.getOption('includePerform')">
    <value name="includeDelegation" eval="taskElement.getOption('includeDelegation')">
      <value name="hasResultClass" eval="taskElement.getOption('hasResultClass')">
        <value name="targetClass" eval="classBuilder.from(taskElement.targetClass)">
          <value name="includePerformOnTarget" eval="taskElement.getOption('includePerform').defined">
            and !isDetailsOnly
          </value name="targetProjection" eval="taskElement.getOption('targetProjection').defined"/>
        </value>
        <option if="taskElement.getOption('isBranchingTask').defined">
          <value name="builder" eval="classBuilder.from(taskElement.targetElement, 'State')">
            <option if="taskElement.getOption('hasResultClass').defined">
              <value name="builder" eval="classBuilder.from(taskElement.getOption('hasResultClass'))">
                <defaultOption eval="classBuilder.from('Void')"/>
              </value>
            </option>
          </value>
        </option>
      </value>
    </value>
  </value>
  <option if="resultClass && !resultClass.packageName.empty()">
    import <resultClass.packageName/>
    <resultClass.packageName/>
    if (resultClass && !resultClass.packageName.empty())
      import <resultClass.packageName/>
      <resultClass.packageName/>
      // anchor:custom-imports:start
      import <targetClass.packageName/>
      // anchor:custom-imports:end
      interface $Interface$ to access the implementation of the task element <class.className>,
    */
    public interface <class.className> extends TaskPerformer<resultClass.packageName>, <targetClas
    if (includeParameters)
      public void setParameters(<paramClass.className>Details <paramClass.varName>Details);
      <option if="taskElement.getOption('hasResultClass').defined">
        <value name="builder" eval="classBuilder.from('Void')"/>
      </value>
    </option>
    // anchor:custom-methods:start
    // anchor:custom-methods:end
    }
  </option>
</conditionalValue>
```

Artifact: Expander.stg
• Automatic Programming
• Toward Automatic Regeneration
• Creating Meta-Circular Regeneration
• On Meta-Programming Interfaces
  • Need for Meta-Level Interfaces
  • Exploring Two-Sided Interfaces
• Conclusions and Discussion
• Questions
Need for Meta-Level Interfaces

• Programming interfaces enable *scalable collaboration:*  
  • Within companies  
  • Across companies  
  • In open source communities  

• Scalable collaboration is needed in software for:  
  • Rich application offering (desktop applications, apps, ...)  
  • Convenient hardware support (drivers for modems, screens, disks, ...)  

• Defining *meta-level interfaces is still a subject of research* in 2019:  
  • Novel conceptual model for the systematic reuse of textual DSLs [Wortmann]  
  • An intermediate representation to be used for code generation [Gusarovs]
Two-Sided Meta-Level Interfaces

• Automatic programming performs a *transformation*
  • From domain and/or intermediate models
  • To code generators and programming code

• Need to define open *interfaces at both ends*
  • To add or extend domain models
  • To add or replace code generators

• The proposed meta-circular structure
  • Simplifies the definition of the interfaces
  • Avoids the *non-scalable burden* on the meta-code
    • To integrate, or at least accommodate, ever more extensions at both ends
Closing the Meta-Circle: Expander API

Extend meta-model

Define expanders
Provide templates

Prime Radiant Expand

Expander MetaCircle

Expand

Model

Logic

Control

View

Read / Write

Model

Logic

Control

View

Read / Write

Application

Expander Model

Code Templates

Model

Logic

Control

View

Application
• Automatic Programming
• Toward Automatic Regeneration
• Creating Meta-Circular Regeneration
• On Meta-Programming Interfaces
• Concluding Facts and Thoughts
• Questions and Discussion
(Re)Generated Code in Production (or ½)

- Enterprise applications (JEE)
  - Budget follow-up tool
  - Master thesis evaluation
  - Diplomatic card services
  - Data centre management
  - Solar panels monitoring
  - Beverage product lifecycle
  - Energy datahub management
  - Real estate estimation tool
  - IoT data inflow engine
  - Privacy and digital vault
  - ...

<table>
<thead>
<tr>
<th>Applications</th>
<th>± 20</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components</td>
<td>43</td>
</tr>
<tr>
<td>Data elements</td>
<td>1546</td>
</tr>
<tr>
<td>Attributes</td>
<td>7094</td>
</tr>
<tr>
<td>Task elements</td>
<td>535</td>
</tr>
<tr>
<td>Flow elements</td>
<td>133</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Skeletons</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>± 40.000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Extensions</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data layer</td>
<td>6</td>
</tr>
<tr>
<td>Logic layer</td>
<td>1731</td>
</tr>
<tr>
<td>Shared layer</td>
<td>250</td>
</tr>
<tr>
<td>Proxy layer</td>
<td>5</td>
</tr>
<tr>
<td>Control layer</td>
<td>218</td>
</tr>
<tr>
<td>View layer</td>
<td>1186</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Insertions</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data element</td>
<td>1436</td>
</tr>
<tr>
<td>User connector</td>
<td>146</td>
</tr>
<tr>
<td>Task element</td>
<td>401</td>
</tr>
<tr>
<td>Flow element</td>
<td>0</td>
</tr>
</tbody>
</table>

Data 2018
(Re)Generated Meta-Application in Use
(Re)Generated Meta-Application in Use

• Prime Radiant, supporting:
  • CRUDS for 87 data elements of the meta-model
  • Invocation of operations using 31 task elements
  • Expansion, build, and deployment of applications

• Expander or code generation software:
  • Has been available for 8 years
  • Has been refactored to meta-circular architecture in 2019
  • Contains 181 expanders or code generators for JEE application stack

• New expander bundles are being released:
  • REST/Swagger bundle of 39 expanders by internal developer
  • Additional bundles by one of the early customers
Releasing a Expander Developer Kit
Starting exchange.stars-end.net
**Remember:** The Power of Circularity

- A transistor is switched by a transistor
- A cell is produced by a cell

- Enables rapid evolution
  - Single point of progress
    - Better transistor → better circuits
    - Improved cell → improved life forms
  - Collapses/shortcuts the design cycle
    - Even *positive feedback* or *resonance*
Closing the Meta-Circle: Resonance

Extend meta-model

Define expanders
Provide templates

Expand

Prime Radiant

Expand

MetaCircle

Model

Logic

Control

View

Developer MetaCircle

Model

Logic

Control

View

Application
Closing the Meta-Circle: Resonance

- Extend meta-model
- Define expanders
- Provide templates
- Prime Radiant
- Model
- Logic
- Control
- View
- Read / Write
- Model
- Logic
- Control
- View
- Application
- Expander Model
- Expander MetaCircle
- Developer MetaCircle
- Meta

Diagram:

- Meta Model
- Expand
- Model
- Logic
- Control
- View
- Read / Write
- Logic
- Control
- View
- Application
- Expand

Legend:
- Expander Model
- Meta
- Model
- Logic
- Control
- View
- Read / Write
- Application
“Let us turn every programmer into a meta-programmer, and create a meta-circular instability or resonance effect.”
• Automatic Programming
• Toward Automatic Regeneration
• Creating Meta-Circular Regeneration
• On Meta-Programming Interfaces
• Concluding Facts and Thoughts
• Questions and Discussion
QUESTIONS?
herwig.mannaert@uantwerp.be