Granulated Code Generation Of Interfering Functionalities

Thesis submitted in partial fulfillment of the requirements for the M.Sc. degree in Tel-Aviv University, School of Computer Science

by

Igor Gelfgat

Prepared under the supervision of Prof. Amiram Yehudai and Dr. Shmuel Tyszberowicz

November 2011
Acknowledgements

I would like to express my gratitude to my thesis advisors. Dr. Shmuel Tyszberowicz for his throughout guidance and experience and Prof. Amiram Yehudai for his wisdom and support. This work could not been completed without their patience and appreciation.

I am grateful to Nadav Wiener for introducing me to the Model-Driven approach.

I would like to thank Ohad Barzilay and Oren Kohavi for their assistance.

To my family.
Abstract

The Model-Driven Software Development approach is becoming widely used as powerful model-driven tools are becoming available for the developer. Its primary goals are portability, interoperability and reusability, through architectural separation of concerns. Yet, it is not suitable to model, and therefore to generate the code, for all the aspects handled in the development stage. As a result, the Model-Driven Software Development approach is not as widely used as it could.

In this thesis I present a technique that extends the capabilities of Model-Driven Engineering with behavioral aspects, by modeling concerns and using them in code generation. Common concerns can be defined for design patterns, software infrastructures and other common aspects. Software architects are advised to apply those common concerns to their system models and also to create system-specific concerns and apply them at the modeling stage. I name it enriching a model with concerns. With the help of code definition for each concern, my tool automatically generates code for the enriched model. Thus, at the end of the modeling stage the developers will have the structure of the code and all the glue code ready, so they will only have to fill the business logic in the manual implementation methods created for them. They will also maintain the enriched model and not the code they would otherwise write manually. I argue that my technique decreases the required development effort and increases the quality of the software.
# Contents

1 Introduction ............................................. 5

2 Examples ................................................. 8
   2.1 Examples of Resulting Generated Code ............... 9
      2.1.1 Example 1 ........................................ 9
      2.1.2 Example 2 ........................................ 11

3 Granulated Code Generation – the Technique ............. 14
   3.1 General Description ................................ 14
   3.2 My Approach as Part of the MDE Process .......... 15
   3.3 Defining Concerns ................................... 17
   3.4 Applying Concerns ................................... 21
   3.5 Resulting Code ....................................... 25

4 GCG Tool .................................................. 26
   4.1 Design ................................................. 26
      4.1.1 Model parsing ..................................... 27
      4.1.2 Analysing concerns ............................... 28
      4.1.3 Analysing classes ................................. 28
      4.1.4 Generating the code for enriched and regular classes ................................. 30
   4.2 Design Decisions ..................................... 33
   4.3 Implementation Notes ................................. 34
   4.4 Usage Examples ...................................... 35
      4.4.1 Bank account management example ............... 35
      4.4.2 Distributed algorithms example .................. 38
5 Related Work

5.1 Comparing to AOM techniques

6 Conclusion

7 Future Work

A Usage Examples Code
### List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>Concern meta-model</td>
<td>18</td>
</tr>
<tr>
<td>3.2</td>
<td>Definition of an object pool</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>HR service example</td>
<td>24</td>
</tr>
<tr>
<td>3.4</td>
<td>Schematic view of an enriched method</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>GCG tool components</td>
<td>27</td>
</tr>
<tr>
<td>4.2</td>
<td>GCG tool object model</td>
<td>28</td>
</tr>
<tr>
<td>4.3</td>
<td>Code generation classes</td>
<td>31</td>
</tr>
<tr>
<td>4.4</td>
<td>Class Diagram for the plugins API</td>
<td>32</td>
</tr>
<tr>
<td>4.5</td>
<td>Concerns used in the bank account application</td>
<td>37</td>
</tr>
<tr>
<td>4.6</td>
<td>Bank account application implementation classes</td>
<td>37</td>
</tr>
<tr>
<td>4.7</td>
<td>Bank account application GUI</td>
<td>37</td>
</tr>
<tr>
<td>4.8</td>
<td>Concerns used in distributed algorithms example</td>
<td>38</td>
</tr>
<tr>
<td>4.9</td>
<td>Distributed algorithms implementation classes</td>
<td>39</td>
</tr>
</tbody>
</table>
Listings

2.1 Example 1 implementation code ........................................... 11
2.2 Example 2 implementation code ........................................... 12
3.1 Generated code for MySet .................................................. 25
A.1 Generated code for the method AccountDAO.withdraw .............. 45
A.2 Plugin code for the concern DistributedAgent ....................... 46
A.3 Generated code for the enriched class ChangAndRobertsNode ....... 47
Chapter 1
Introduction

The model-driven approach to software development becomes more and more popular. Its primary goals are portability, interoperability and reusability, through architectural separation of concerns [16]. When the model is the primary artifact of the development process, the user works on the model, and a tool generates all or part of the code. This raises the productivity and the quality of the process. Productivity is raised as the auto-generation takes care of all the plumbing code. The generated code is of very high quality since it was fully tested beforehand. Though the model itself still has to be tested, it requires much less effort than testing the whole system. Thus, it is effective to maintain and fix the model instead of the code when possible [1].

It is easier to understand a model rather than the code, since it is a more abstract definition of the system. Of course, when the code is automatically generated we wish its behavior to be clearly understood from its model definition, to allow us to avoid looking at this code at all. The model-driven technique obligates to work with the model itself and not with the code, keeping the model consistent. Once again, it shortens the development time and increases the implementation quality.

A software system has structural and behavioral aspects. Both should appear in various levels of abstraction in the system model, as it is more natural for a human to start from a high-level description and then gradually get down to details. Presently, using model-driven techniques mostly means constructing structural models, but not behavioral ones. Once behavior is modeled, it is often done at the same level of abstraction as
implementing, and requires the same effort.

It is hard to find a collaboration of structural elements such that it will be possible to automatically generate their implementation. There are, however, some exceptions. For example, when we have a structural model where the behavior of some of its elements is similar. In this case one can apply stereotypes to them [17], enabling code generation for this behavior. Also, there are modeling techniques that define behavior for a single type of systems, but do not suit other types. For example, state diagrams can be used to model events-based systems [14] [6], but are not suitable for, e.g., data management systems.

MDA [16], the most popular Model-Driven Engineering (MDE) initiative that separates specification of a system from the details of its platform. To enable this, it supports specifying a system and platforms separately, and transforming the system specification into one for a particular platform.

I have developed a technique to define the behavior of a general system or part of it, which enables to generate automatically as much of the code as possible, such that only the business logic will be left to implement. The intention is similar to that of Aspect-Oriented Programming (AOP) [9] that, among other things, targets at minimizing duplicated code. Here I target to minimize the effort to apply similar behavior multiple times. In my technique models are enriched with concerns, enabling automatic code generation for the structure and the behavior that concerns contribute to the applied model elements. My use of concerns in modeling is reminiscent of the use of aspects (in the sense of AOP) in implementation. I should stress that a concern may or may not be implemented partly using aspects, but this should not be a concern of the modeler.

The Oxford Dictionary defines granulated as “formed into grains or particles”. The code generation technique is named Granulated Code Generation to emphasize the fact that the generated code is built of small parts coming from many functionalities applied to the model elements.

The rest of the thesis is organized as follows. Chapter 2 demonstrates the expected results with examples. In Chapter 3 I describe the technique, explaining how to model with
concerns and how the generation process works. Chapter 4 describes the tool prototype implementation and usage examples. Chapters 5, 6 and 7 summarize the thesis.
Chapter 2

Examples

My technique defines concerns (or functionalities) as model entities that can be applied to regular model entities (classes, members and operations). This makes it possible to define general behavior of a class at the model stage and to generate automatically part or all of its code. I use both the terms concerns and functionalities interchangeably. Detailed descriptions of concerns are provided in Chapter 3.

There are several challenges one has to deal with when generating code from a model with functionalities applied:

- **Multiple concerns may be applied to the same class, member or method.** Various concerns appearing in different parts of the model affect the implementation, particularly same target resources\(^1\) and even interfere at lower level (code lines in the same method, etc.). Generated code for a class or a method consists of generated parts that are defined for the applied concerns. The code examples in Section 2.1 (Figures 2.1 and 2.2) demonstrate it.

- **It should be easy to add and replace concerns.** Many functionalities along different kinds of systems will be a part of the process proposed here. Already defined concerns will be changed. The challenge is that the resource generation (Java code generation, for example) should take into account each part every concern adds to a class or a method, while the part implementation is supplied by the concrete

\(^{1}\)Implementation files generated from the model, e.g., code files, database scheme, properties files, etc.
concern generation logic.

- **Order.** Each applied concern results in generated code lines, for example in a method. Since multiple concerns may affect a method, the order of the generated code lines is important. The order may be defined in the model or be automatically found by the tool, based on the model and the concerns definitions.

### 2.1 Examples of Resulting Generated Code

Following are examples of classes that I want to model using concerns and have their code generated automatically. Here I present the code as it would be written manually and explain how similar code could be generated. The examples demonstrate the results I want to achieve. See Section 3.5 for an example of generated code.

#### 2.1.1 Example 1.

This example describes the method `makeOperation` of the class `SomePrivilegedOperation` that, in addition to its main operation, implements also the following requirements: only privileged users may execute the method, it needs database connection, and its success or failure is logged.

- **Operation within a transaction.** For the operation to be successfully performed, I should locate or create a transaction and a database connection within the transaction. A transaction is an operation with external resources (for example, a database) that among other properties [10] has only two possible outcomes: a success with all needed changes performed and a failure with no change performed.

- **Privileged operation.** I want the method to run as a privileged operation, thus I use Role Based Access Control (RBAC) which allows to assign privileges to users or user groups to run this operation. For full description of how RBAC may be modeled in UML and how automatic security code may be generated from it, refer to [15].
I implement RBAC using a database, therefore, a database connection within a transaction is needed.

- **Operations log.** Every time the action is executed, the following information should be saved: the operation description, the acting user name and a success/failure/unauthorized status.

To apply the requirements for the method using concerns, the method model will look like:

```java
<<operationAccounting, currentTransactionConnection(connection),
protectedOperation(connection, currentUser)>>
makeOperation(currentUser: User): boolean
```

In the model, there are the method signature and the functionalities applied with concerns. In Sections 3.3 and 3.4 I define how concerns are modeled and applied to classes and methods. Listing 2.1 displays a Java implementation of the example.

Following is a brief explanation of how the implementation code can be generated from its model. First, the transaction code is inserted. Then permissions are checked using the parameter `currentUser`. To specify a user to be checked for privileges to run the operation, the `currentUser` is supplied as a parameter for the applied functionality at the modeling stage. The user implementation code is inserted after the permissions check. At every point before the return, the operation is logged including its success/failure status.

The structure of the example code, namely a block inside a block, reminds of aspect of AOP. Although implementation without aspects is more straightforward (and also easier and faster), aspects could be also used in the generation stage. For example, one can generate AspectJ [8] code instead of Java.

Only the code that deals with the accounting requirement differs from the others by appearing in all blocks (success, connection failure, authorization failure). It may be changed a little to log any success and any failure on the top level, (i.e., a new method that logs the result of the method in the example), but I want to show this kind of functionality.
public class SomePrivilegedOperation {
    public boolean makeOperation(User currentUser) {
        try {
            // locates current transaction and retrieves a connection within it
            Connection connection = TransactionManager.getCurrentTransaction();
            boolean authorized = checkPrivileges(connection, this, user);
            if (!authorized) {
                // removed: log "unauthorized"
                return false;
            }
            boolean success;
            // removed: here is the operation implementation
            // removed: end an operation within a transaction
            // removed: log success
            return success;
        } catch (SQLException e) {
            // sql error
            // removed: log failure
            return false;
        }
    }
}

Listing 2.1: Example 1 implementation code. Lines 4-6, 18, 23-24 and 27-28 handle database related behavior. Lines 8-9 and 12-13 check access rights to the privileged operation. Lines 11, 20 and 26 log results. Note: the "removed" lines in code fragments from here on represent code that was removed to save space.

### 2.1.2 Example 2.

The second example of granulated code generation displays how code structure may be affected and, particularly, how a class can be altered to use multiple design patterns, just by applying concerns.

- A singleton class [4]. A single instance of the class is retrieved through a getInstance() method. In modeling and generation stages every reference to the singleton class is a reference to the same object.

- An object pool [11]. In this case I use two object sets (a set of free objects and a set of objects in use) and methods to borrow them for work and to return them back to the pool.

An implementation code is shown in Listing 2.2. Both patterns are at a class level, but cause generation of different methods and fields. The singleton pattern brings in
a static method and static fields, while the object pool pattern leads to generation of regular methods and fields.

Consider a more complex case, where a concern is applied to some other class OtherClass with a link to the class in the example. Also, the link is assigned a role, such that the link should be used in the generated code for OtherClass. Any such usage should take into account the singleton pattern and access the pool object using the getInstance() method.

```java
/**
 * @concern singleton
 * @concern objectpool(MyResource)
 */

public class MyResourcesPool {
    /** @concern singleton */
    private static MyResourcesPool theInstance;
    /** @concern singleton */
    private static Object lock = new Object();
    /** @concern objectpool(MyResource) */
    private List availableList = new ArrayList();
    /** @concern objectpool(MyResource) */
    private List inuseList = new ArrayList();
    /** @concern singleton */
    private MyResourcesPool() {}
    /** @concern singleton */
    public static MyResourcesPool getInstance() {
        if (theInstance == null) {
            synchronized (lock) {
                theInstance = new MyResourcesPool();
            }
        }
        return theInstance;
    }
    /** @concern objectpool(MyResource) */
    public MyResource requestResource() {
        // removed: if there is no available object -- wait for it
        MyResource resource = null;
        synchronized (this) {
            // removed: find object in availableList, move it to inuseList
            // and return it within resource variable
            resource = availableList.remove(0);
        }
        return resource;
    }
    /** @concern objectpool(MyResource) */
    public void returnResource(MyResource resource) {
        // removed: move the object from inuseList to availableList
        synchronized (this) {
            inuseList.remove(resource);
        }
        // removed: notify that there is a free object
    }
}
```

Listing 2.2: Example 2 implementation code. Note: the @concern tag indicates either a concern is applied to a class, or a member/method is generated as a result of applying a concern.

I may elaborate the examples to show how I want to see the generated code of multiple functionalities applied to a class or a method. However, there are additional issues I can deal with, when implementing the technique, but I will leave them open in this thesis:
• Same parameter used many times for the same *concern* applied to different classes or methods, may be avoided to be defined many times by using annotations (see [13] for how annotations may be used in model-driven software engineering).

• Order: for independent functionalities the order can be set based (also) on performance parameters.
Chapter 3
Granulated Code Generation – the Technique

In this chapter I describe a technique that enables to automatically generate code from a model that includes behavior information.

3.1 General Description

The generic model technique and generation process that I have built enables system designers to define common functionalities and to apply them to software model entities. As the result the target system is constructed, having part of it generated from modeled structure and behavior, and the other part is manually implemented business logic.

The functionalities that I call concerns are model entities (in my examples they are UML classes) that may be associated with (applied to) regular entities (UML classes) that represent system entities and probably result in generated DB tables, classes in code, etc. Concern members may be associated with class members, and concern methods – with class methods.

Once a concern is associated with some class it affects the generated code for the class, adding some functionality (defined by a plugin, supplied with the concern), saving the developer’s time needed to write this code manually.

To make this technique useful, it should be able to apply a wide range of generation functionalities to a model. Each functionality is generally independent from others or
depends only on a small number of other functionalities. Thus, it should be easy to add new functionality definitions to the tool. I decided that the architecture of the tool will be based on a “plugins framework”, where a new functionality is added to the tool through a new plugin that is not aware of other plugins, except those it depends on.

In [13], Uhl, Koch and Weise discuss how a higher level of abstraction affects effort needed to develop and to maintain a system. Similarly, abstraction of behavior helps to make the model more intuitive. Generation of the code from the enriched model minimizes manual code, therefore increasing program quality and demanding less maintenance. Generally, the technique reduces development effort, keeping models simple.

3.2 My Approach as Part of the MDE Process

The software development process includes different phases and involves different professionals such as architects, developers, etc. Following is a description of how the granulated generation technique is incorporated into the model-driven software development process.

1. Defining generatable functionalities (e.g., of an object pool).

Architects and the software infrastructures team identify common functionalities (in addition to those widely used or already existing within the organization), prepare their definitions and code generation plugins. See Section 3.3 for details.

2. Creating high level models, using defined earlier functionalities, where needed (for example, creating a UML class for a thread pool with the object pool functionality applied).

Software system modeling is performed starting from higher level of abstraction and adding details at each level. Higher level models are automatically transformed to lower levels models, adding details by annotations, and manually at the lower level. The number of levels may vary from system to system. The transformations are system-specific, they are, however, defined using standard languages and tools.

For example, Platform-Independent Model (PIM) is a high level of abstraction,
defined in MDA [16]. There may be a number of platform-independent models with transformations between them (each is more detailed than the previous one). Functionalities can be used at all levels. It is the architect’s decision where to use a specific functionality. The decision depends on the functionality itself and on the element’s level of abstraction. The functionality may be applied to an element at the highest level it appears, when the element is not yet detailed. On the other hand, it may be preferred to apply it on some lower level, because lower levels are designated to add details. The higher the level it is introduced into the model, the earlier its effect is understood. However, if the functionality is not widely used or is very detailed, it bulks the model, making it less abstract. It may be a good practice to apply only abstract functionalities on the higher level and to create a dedicated level for applying detailed functionalities.

3. Performing model transformation automatically or semi-automatically, adding annotations or other details.

In MDA this refers to PIM-to-PIM transformations. At this stage general details about the system are specified, which are relevant for all, or most of, system perspectives (platforms, in the case of MDA).

Transformations should preserve information about functionalities used at higher levels and may also add details about how they are used. It may be convenient to supply parameters to already applied functionalities using annotations at the transformation stage. Rules about which details should be added in transformations may be part of functionalities definitions.

4. Performing transformations to lower level models.

An example is MDA PIM-to-PSM transformations (PSM stands for Platform-Specific Models). At this level we can add platform-specific details (details about implementation in the specific platform, for example, table names for database).

In case of PSM models, the added details are relevant only for some system’s
perspective. Hence it is preferred to specify most details about system behavior at higher levels. Some functionalities, however, can require platform-specific details added at this level, preferably added using annotations. More transformations may be performed, if needed, also between lower level models.

5. Running model-to-code generation of target resources that are the basis of the implementation.

Lowest level models are used for code generation. These are the models I will present in all figures and examples. Code generation is further described in Section 3.5.

6. Implementing.

In order to complete the implementation, the generated code is extended by the developers by implementing the business logic. This implies that the automatic generated code should be separated from the manual code, and at the same time its contribution at every point should be clear to the developer.

Changes at the implementation and maintenance stages, that can be done at the model level, should be performed there. This way they will affect the generated code instead of making the same changes manually in the code. Making changes manually in the code demands more effort and prevents us from taking advantage of models in the implementation and maintenance stages.

Details about MDA process and model and platform relationships can be found in [13].

3.3 Defining Concerns

Following I describe how a concern is to be defined in a model. Such a definition enables a designer to apply it to a class in the design model.

In addition to the definition, a plugin should be supplied to the generation tool for a particular concern, to enable generating the code that implements capabilities the concern is meant to provide when applied to a class. This plugin is not to be seen by
anyone during the system design or the implementation stages. Some documentation is required, however, to describe the semantics of applying the concern.

There exist different kinds of concerns. Some interfere at method level\(^1\), some at class level\(^2\), and some at both\(^3\). Some introduce parameters, others introduce fields or methods. There are also more complicated concerns that are related to other concerns (interfere at inter-object level). Following is a description of an approach to define a concern.

Defining a concern means to describe its name, fields and methods it can be applied to, and relationships with other concerns. I do it by creating a *UML class* with a &lt;&lt;concern&gt;&gt; stereotype. The name of the UML class that represents the concern is the name of the concern, its attributes and operations describe fields and methods the concern may or should be applied to within the applied class (*enriched class*). UML meta-model of a concern is presented in Fig. 3.1.

![Concern meta-model](image)

Figure 3.1: Concern meta-model.

A UML attribute in a concern (i.e., in the UML class that represents it) defines a stereotype that should be applied to an attribute in the enriched class in the design model. I call it *concern field* hereafter. The attribute it is applied to is called *enriched*

---

\(^1\)The concerns in the Example 1 in Section 2.1.1.

\(^2\)The concerns in the Example 2 in Section 2.1.2.

\(^3\)Pool example, see Figure 3.2.
A UML operation in a concern (named concern method) defines an annotation that should be applied to an operation of the enriched class (named enriched method). Concern methods may have one or more of the following stereotypes applied:

<<optional>> If not otherwise specified, a concern method is required for the concern. Only when the <<optional>>, stereotype is used for the concern method it does not have to be specified in an enriched class.

<<full implementation>> This concern method supplies full implementation for a method it is applied to. It should be the last one applied, because further applied concern methods are meaningless. Similarly, no additional manual implementation of the enriched method is required.

<<frontend>> When applied, the parameters of the enriched method should conform to the parameters of the concern method: the same order of parameters and each enriched method parameter is assignable from the corresponding concern method parameter. This kind of a concern method enables calling an enriched method from the generated code.

A defined concern method may be parameterized to pass information about the desired functionality: data structures, types, etc. Some parameters require special notation, achieved by adding a stereotype to a parameter, as following:

<<introduced>> A value of specified type, which is calculated by the concern and supplied to the proceeding concern methods and the implementation method as an additional argument. When used in a model, only a name of the new argument is specified in place of the parameter.

<<erased>> If a method argument or an introduced parameter (see Section 3.4) is passed to an <<erased>> concern parameter, the argument/introduced parameter is used only by the concern, and is not available for use by neither concern methods.
applied after it, nor the implementation method. This stereotype is for convenience to avoid developers seeing unneeded parameters (used by the concern only).

```
<<returns>> The type of the value which is calculated by the concern and returned to the caller. The concern method hides the return value it receives from the called method and returns the calculated one (of the specified type). It is helpful when the received value is processed by the concern method and a value of another type or even nothing (void) is returned by the concern method. A concern method can be defined as returning a value also when it receives void or in the case it is a
```

```
<<full_implementation>>.
```

At most one parameter of this kind may be present in a single concern method. It is possible to define it as a return type of the concern method (see Figure 3.2), except for the case of void type.

```
<<return_type_required>> Type of an enriched method that the concern method may be applied to. The concern method uses the value it accepts from the method called. At most one parameter of this kind may be present in a single concern method.
```

```
<<exception>> Exception thrown by the concern. Should be declared in the signature of the enriched method or being handled by one of the previous concerns. By default, the signature of the implementation method will not contain this exception.
```

```
<<handled_exception>> An exception of this type, if thrown from the proceeding concern method/implementation method, is handled by the concern and is not seen by the caller.
```

Note that a parameter can not be annotated with more than one of the types. The last four types do not require a parameter to be supplied with a value, so the parameter appears only in the concern definition.

An example of a concern definition is shown in Figure 3.2. It defines a concern for a pool of objects that allows reusing objects instead of recreating them (when it is ex-
pensive to create them for each use). The pool has a template parameter that defines
the type of stored objects and three methods each pool of objects has: \textit{init()} for the
first initialization of objects, \textit{borrow()} – taking an object out of the pool and \textit{return()} –
returning previously borrowed object. The initialization method is a \texttt{frontend}
method, because it is called from inside the automatically generated code. The other
two are \texttt{full implementation} – no manual implementation is required.

![Figure 3.2: Definition of an object pool (from Example 2 in Section 2.1.2)](image)

### 3.4 Applying Concerns

An element in a model can be described in various ways and by various relationships,
annotations, etc. Each description is meaningful for the code generation and should be
expressed in the generator definition. This allows the generation tool to identify the
element’s behavior.

A class can be instrumented with one or more concerns (see Figure 3.3). In this case,
every non-optional concern method defined in the concern should be applied to one of
the methods in the enriched class. Every concern field should be applied to a field in the
enriched class. When a number of concerns are applied to a class, each of them is taken
into account separately, but they should not have collisions (i.e., fields with identical
names or methods with identical names and parameter types).

Each method may have a number of concern methods applied to it. The concern
methods are treated in their order of appearance for the enriched method: on a call they
will run in that order, and on return their “after” parts will run in the reverse order.
Concern methods are applied in the following way:
• A method’s signature is defined as it should appear to the caller, i.e., it is the signature of the full method including all applied concern methods. Hence a developer, who has to call the generated code for the enriched method, sees the same method signature as it appears in the model. Please, note that it is the same signature as if the class was modeled without using concerns and implemented manually.

• From an applied concern method’s point of view, it is running in a “method” (which does not exist really) that has a list of parameters, a set of exceptions and a return type. The rest of a structure it is applied to (i.e., the following concern methods and the implementation method) is also a “method” with another list of arguments and a return type. The rules and the restrictions are as follows:

  – Every parameter supplied to a concern method should be accessible by it. Therefore it should be defined in the signature or introduced by one of the previously applied concern methods, and should not be hidden (<<erased>>) by any of them. See Equations 3.1 and 3.2.

  – The implementation method will have its abstract signature generated in the code and should be implemented manually, except for the case where the last applied concern method is a <<full.implementation>>.

  – The implementation method will contain all non-hidden parameters of the method’s signature as it is defined and all introduced (but not hidden) parameters in all applied concern methods.

  – If none of the applied concern methods requires a return parameter, the return type of the implementation method will be the same as of the enriched method. Otherwise, the return type will be as needed by the latest concern method that has a return parameter. See Equation 3.3.

  – Chain of return types. Every concern method that expects a certain return type should comply with the return type of a following concern method with a return parameter, if exists. See Equation 3.4.
The following defines a signature of a method:

\[(P, r, Ex)\]

where \(P\) is a list of parameters, \(r\) is a return type and \(Ex\) is a set of exceptions.

Enriched method signature is marked as:

\[(P_0, r_0, Ex_0)\]

Similarly, its implementation method signature is marked as:

\[(P_n, r_n, Ex_n)\]

where \(n\) is the number of applied concern methods.

The rules described beforehand are explained using the equations below. Note, the equations are defined \(\forall i = 0..n-1\).

\[P_{i+1} = P_i \cup \text{Introduced}_i \setminus \text{Erased}_i \quad (3.1)\]

\[Ex_{i+1} = Ex_i \cup \text{Handled}_i \setminus \text{Thrown}_i \quad (3.2)\]

\[r_{i+1} = \begin{cases} r_i, & \text{if } \text{returns}_i = \bot \land \text{returnTypeReq}_i = \bot \\ \bot, & \text{if } \text{returns}_i \neq \bot \land \text{returnTypeReq}_i = \bot \\ \text{returnTypeReq}_i, & \text{otherwise} \end{cases} \quad (3.3)\]

\[r_i \leq \text{returns}_i \quad (3.4)\]

Figure 3.3 presents an example of an HR service that creates, adds and removes an employee in an organization database. The service is defined as a \(<<\text{WebService}>>\) so it will be created and deployed as a Web Service and all its parameters will be presented in its Web Services Description Language (WSDL) file [21]. A connection is supplied to every method, so it will be able to perform database queries. Each method is declared
Figure 3.3: A Web Service with 3 authorized-only methods with authorization and database connection behavior, defined by applying concerns. The methods are similar to the method in Example 1 in Section 2.1.2.

with authorization checks. The only thing that is left to be implemented manually is to perform a query for each of the methods.

Figure 3.4 illustrates the enriched method `getAllEmployees()`, generated parts for concern methods applied to it, and the arguments and the return type of each part and the implementation method. These would be actual method signatures for all parts, if they were created in separate methods.

The application on the signatures is as the following. Concern method `currentTransactionConnection` adds its `<<introduced>>` parameter `connection` that becomes available for the proceeding concerns. Concern method `protectedOperation` removes the `<<erased>>` parameter `user`. Finally, both `protectedOperation` and `protectResultObject` throw `ProtectionException`.

Figure 3.4: Schematic view of the enriched `getAllEmployees()` method, the arguments and the return type of each generated part (if they were separate methods) and the implementation method.
3.5 Resulting Code

In this section I present a MySet example, a set of objects, combined with a Visitor design pattern: the method checkAllObjects() passes over all elements in the set and performs a manually implemented operation on each object. Listing 3.1 shows its generated code in the class MySetBase. The manual code is placed in the class MySet, implementing the abstract method, and adding any fields and methods not mentioned in the model (private methods, as the rest should be in the model by best practice). Thus, the manual code is not affected by regeneration of the automatic code, since they are in separate files. Moreover, all the automatic code is in a separate source folder.

```java
/** Generated base class of example1.MySet class in the model. */
public abstract class MySetBase {

private MyObject[] myObjects;

/** Generated method for the one specified in the model. */
public final void checkAllObjects() {
    for (MyObject element : myObjects) {
        this.checkAllObjects(element);
    }
}

/**
 * It is a generated method that should be implemented.
 * @param object the parameter introduced by the visitor
 */
protected abstract void checkAllObjects(MyObject object);
}
```

Listing 3.1: Generated code for MySet
Chapter 4

GCG Tool

In this chapter I describe the GCG Tool design and implementation.

4.1 Design

The tool is based on the Eclipse Modeling Framework (EMF), specifically on its ability to parse UML models. Please, note that I describe the tool design as it was prepared, some of the features were not yet implemented, as stated in the Section 4.3.

The GCG Tool gets as an input one or several UML models in XMI format that contain the following:

- concerns definitions;
- enriched classes;
- regular classes.

The tool makes its work in four phases:

1. Parses the files, loading the UML models into the memory.
2. Finds and analyses concerns definitions.
3. Finds and analyses enriched as well as regular classes.
4. Automatically generates the code for the classes.

Figure 4.1 presents the tool’s components design. The `ggt.engine` component contains all the analysis classes (including `umlutils` for UML model parsing, `platform` and `exceptions` subcomponents). The `ggt.generator` contains all the code generation logic. The `ggt.generator.parts` is the plugins API.

Figure 4.1: GCG tool components: the main class, components and subcomponents.

If there are errors in some concerns, the code for the classes they are applied to will not be generated. The progress of the generation as well as the errors are shown to the user.

The GCG Tool is implemented to enable generation of the code for various platforms (e.g., Java, C#, DB, etc.). To enable a platform, some additions are required to the GCG Tool. After that, plugins that wish to provide code for this platform, should supply code parts for it.

### 4.1.1 Model parsing

The parsing phase is fully based on the EMF framework. At the end of the parsing there is a tree of EMF objects that represents the UML model. At this point the tool does not have the knowledge about neither concerns nor enriched classes. All it has are raw UML classes with stereotypes/annotations.

In order for the `<<concern>>` stereotype (see Section 4.1.2) to be recognized, the GCG profile has to be applied to a model.
**4.1.2 Analysing concerns**

In this phase, the tool makes a pass on the UML models, finding concerns and representing them in the object model that it uses later for analysing enriched classes and generating the code. The design of the concerns object model is shown in the Figure 4.2.

To identify a concern, it searches for the stereotype `<<concern>>` applied to a UML class. It then analyzes operations and their parameters and build concern’s object model: a `Concern` object containing `ConcernMethod` objects, each with its `ConcernMethodParameter` objects. Especially, the tool analyses applied stereotypes and annotations, as described in Section 3.3. Note that no additional objects are created for the UML attributes, because they have no additional properties yet beyond the basic UML properties (e.g., visibility, type, etc.).

**Figure 4.2: Classes used to represent concerns and how they are applied**

---

**4.1.3 Analysing classes**

After concerns data is prepared, the tool proceeds with a search for enriched classes and generates their code. I could implement it in two ways: either load all enriched and regular classes and only then generate their code; or generate the code for each class at
the time I encounter it. Currently it is simpler for me do it one by one, but in the future, when complex concerns are introduced, I will probably have to load all classes at first and only then to generate the code.

When I encounter an enriched class, I analyse it and create its object model, as shown in Figure 4.2. I recognize concerns applied for the class, functional methods applied for the class methods and parameters supplied to the functional methods. I do it by analyzing annotations as described in Section 3.4. I use annotations and not stereotypes as the UML stereotypes do not have parameters. Stereotypes/annotations are simple and do the work, but they are textual and not so intuitive for use in large and complex models. In future I will consider using a graphical way to apply concerns.

At this point I check the concerns are applied in a way consistent with their definition, as described in Section 3.4. The following situations are recognized as errors (in addition to syntax errors) and prevent generating the code for the enriched class:

1. A concern or a functional method is not found, or a number of functional parameters supplied for a functional method is not as required.

2. A concern’s template parameter is not assigned with a specific type.

3. Ambiguous functional method: when a functional method is applied other than with its fully qualified name (e.g., method instead of concern.method), and a functional method with this name exists in more than one concern.

4. Illegal or void attribute is supplied as a functional parameter.

5. A <<full_implementation>> functional method is not the last one applied to an enriched method.

6. When a class enriched with some concern lacks a method enriched with a non-optional functional method of the concern.
4.1.4 Generating the code for enriched and regular classes

Code generation for the enriched classes are based on concerns definitions in the UML model and their plugins. A concern’s plugin supplies information about code parts that should be generated for classes and methods enriched with the concern.

For an enriched class the generation process performs the following:

1. Generates the class-level code (for Java target platform those are the class definition and its members defined in UML).

2. Passes over applied concerns and combines all their class-level code parts (for Java - mostly class members and imports).

3. For an enriched method of the class:

   (a) creates the method signature;

   (b) analyses the first applied functional method, its application on the parameters, exceptions and the return parameter;

   (c) generates the code for the functional method’s code parts;

   (d) recursively analyses and generates the code (3b - 3d) for the following functional method;

   (e) creates an abstract method for the manual implementation of the enriched method (if required);

   (f) handles special code parts.

4. If requested by specifying a parameter for the tool, generates also a stub for the manual implementation class. The stub includes empty implementations of all manual implementation methods generated by the tool for the class.

Figure 4.3 presents the code generation module.
To implement a plugin, one should use the API described below. At first, a `ConcernGenerationData` interface should be implemented. Its methods return code parts (objects of classes that implement `GeneratedCodePart` interface). Following is the description of basic code parts that a plugin can use. Plugins may also extend the existing code parts. Figure 4.4 illustrates the API.

- **SimpleGeneratedCodePart** A text inserted as is into the generated code.
- **EnrichedClassName** The name of the enriched class.
- **EnrichedMethodName** The name of the enriched method.
- **TemplateParameterValue** A value of a template parameter (i.e., a type assigned to this template parameter).
- **ParameterExpression** A value of a functional method parameter. It may be a method argument, a class member or an expression.
- **EnrichedMethodCall** A call for the implementation (or the next applied functional method). This is a special code part and is essential for combining together code
parts of functional methods applied on an enriched method.

- **ReturnExpression** Defines an expression represented by one or more code parts that will be returned by the functional method. This code part requires special handling in the GCG Tool: depending on the position of this code part in the generated method, the value of the expression will be returned immediately or saved and returned later.

- **ResultLocalVariableName** A name of a local variable that holds the intermediate result (of *EnrichedMethodCall* and *ReturnExpression*).

**Generalization** For any UML class *SubClass* that generalizes (extends) UML class *SuperClass*, the following generalization chain applies for the generated platform classes: *SubClass* extends *SubClassBase* extends *SuperClass* extends *SuperClassBase*.

**Errors** If there is an error for some generation part, the code for the enriched class will not be generated. The major generation errors reported by the tool are:

- An error locating a plugin or loading one of classes it uses.
• A functional method has multiple EnrichedMethodCall parts.

• A <<full_implementation>> functional method has EnrichedMethodCall as one of its parts.

4.2 Design Decisions

In this section I analyze two techniques of generating automatic code from the enriched model: regular code and aspects. Concerns may appear in two ways in the code: generating the relevant code in each place it is needed (separate, but tied code) and aspects. It may seem that aspects are natural representation of such concerns, but each of the two ways has its benefits and disadvantages.

My aim is to tie the concern (and its implementation) to the classes and methods it is used in. Also, the order constraints should be addressed in the generated code. Note that the modeling part of the development process is unaffected by implementation way: the meta-model is not aware of any class/method it is used in; and an enriched class can be easily separated from relations to applied concerns.

Implementing a concern can be done in a way such as a class or a library, thus minimizing the weaving code. In the model, noticing a concern applied to an element will cause generating a code that implements the behavior the concern contributes to the element. The element itself is generated as a class or a method, of course. The weaving between the concern and the elements it is used in can be done in two ways as mentioned above:

• Code for the concern is placed in the generated code at every place it is needed. Generated and user code are separated, as regularly in MDA to preserve consistency between the model and the generated code and to hide the generated code from developer. This approach is better for performance-critical classes, for structural concerns (for example, design patterns) and for concerns that are to be kept with the code for its readability.
• The more natural way is to create an aspect for weaving between the concern and places it is used in. The concern here is separated from the code and all its usage is concentrated in the aspect, so it can be easily generated again with no additional code separation needed.

The advantages of the regular way are: the generated code structure is like the model and is accessible for review and debugging (but is placed in a separate place to avoid confusion); performance at runtime; also for non-crosscutting concerns, aspects do not fit as they hide the behavior away. An additional, yet very important, issue is the inability of aspects to handle some cases (introduced and redundant parameters, etc.). One known disadvantage of the regular way is its interfering with class hierarchy, as powerful IDE tools are required to hide it so as not to confuse the developer.

Aspects separate concerns completely from the code, which is sometimes helpful and sometimes confusing. Making concerns more abstract could be helpful for maintenance (as it appears once and not at every use), but on the other hand, the generated code usually does not require maintenance or even looking at. It is easier to implement concerns logic (to guide the generator) with an aspect or aspect-like language. It is not always possible to define precedence for each case with widely defined aspects.

The conclusion is that in most cases (except crosscutting concerns) I should use the regular way, so I stick to it in my research. Combining the two ways together is left for future work.

4.3 Implementation Notes

Features not implemented yet in the tool prototype as it was prepared only for proof-of-concept:

• multiple models as an input;

• \texttt{\textless\textless full\_implementation\textgreater\textgreater} static enriched methods;

• further error handling, among them:
I do not support fully qualified names for the designer to avoid error 3 in Section 4.1.3.

Non-optional functional methods described in error 6 in Section 4.1.3 are not enforced yet.

- declaring an $<\text{exception}>$ parameter to appear in the signature of the implementation method (see Section 3.3);

- automatically generated interface for a concern with $<\text{frontend}>$ methods (see Section 4.4.2);

- incremental code generation;

- only Java is implemented as a target platform.

### 4.4 Usage Examples

To demonstrate the usage of the GCG Tool, I implemented two examples with the help of the tool: a small data management application and a mechanism for distributed algorithms, based on asynchronous events. I picked these examples to demonstrate that my technique is applicable to different types of systems. Data management, distributed and event-based cover large portion of applications in the software world.

#### 4.4.1 Bank account management example

The first example is a bank account management application. It uses concerns for database connectivity, security, multithreading, error checking and design patterns.

The application manages bank accounts, users, their operations access rights and an access log in a MySQL database. The following operations are implemented:

- Check balance

- Withdraw

- Deposit
• Show access log

• Committing business day operations

The sample implementation uses the following concerns (see Figure 4.5):

• *Singleton* Singleton pattern.

• *Pool* Pool of objects

• *JDBCConnection* JDBC connection supplier.

• *PooledConnection* Supplies pooled JDBC connection. In its generated code uses the *SingletonConnectionPool* class that combines the three previous concerns.

• *Protection* Access control checks.

• *GeneralUtils* Contains a <<full_implementation>> *methodCall* functional method that gives an ability to call a method as an implementation of an enriched method.

• *AccountUtils* Contains *legalAccountState* functional method that asserts a bank account is in legal state after an operation, otherwise it throws an exception.

• *ScheduledTask* A recurring cancellable task, executed with a predefined time interval.

Figure 4.6 displays the model of the AccountDAO and ProtectionDAO classes (and also some auxiliary classes) that implement the operations. Each operation retrieves a JDBC connection, checks user rights to perform the operation and ensures the account state is legal after performing the operation.

Almost all of the manual code in the implementation (not including the GUI) consists of SQL queries which are the business logic of the application. Furthermore, there are MDE tools [23] that enable RDB Schema / SQL code generation from a UML model. Thus, after integrating GCG tool with other MDE tools, I will be able to implement this and more complex applications with minimal or without any manual code.

A simple GUI for executing those operations (see Figure 4.7) is implemented manually.
Figure 4.5: Concerns used in the bank account application

Figure 4.6: Bank account application implementation classes

Figure 4.7: Bank account application GUI
4.4.2 Distributed algorithms example

The second example is a mechanism for distributed algorithms, based on asynchronous events. The mechanism uses concerns for multithreading and communicating with events. With its help I have implemented a ring-based Chang and Roberts election algorithm [2]. Its manual code consists only of the algorithm logic and code that builds the ring.

The example introduces DistributedAgent concern for communicating with events in a graph of computation nodes. Additionally, the implementation also uses Singleton and ScheduledTask concerns described in Section 4.4.1. Figure 4.8 displays concerns definitions.

Figure 4.8: Concerns used in distributed algorithms example

![Concerns Diagram](image)

Figure 4.9 displays the implementation model. EventsProcessingThread and EventsManager classes, and the EventsHandler interface provide mechanism for transporting events between nodes. They are part of the general mechanism and are used in the generated code for the DistributedAgent concern.

Note also the DistributedAgent interface. It is created manually and used in the DistributedAgent concern model and generated code. Actually, it would be probably better for such an interface automatically generated for any concern having its <<frontend>> methods. I leave it for future tool versions.

ChangAndRobertsNode is the algorithm implementation class that sends events and reacts on events received from other nodes. Particularly, there are generated methods for receiving and sending ELECTION and ELECTED events, each of them having an integer
parameter holding some node id. The manual implementation code only fills them with the algorithm logic. Also, the class has completely generated method `setNeighbours()` for connecting it to other nodes.

Figure 4.9: Distributed algorithms implementation classes

See Appendix A for samples of concerns plugins and generated code for the example applications.
Chapter 5

Related Work

Basin et al. [15] describe a way to model security requirements for a software system that enables generating security code automatically. This approach is based on applying roles to UML classes. It does not address other types of requirements besides security.

Harel and Politi [6] show how reactive systems may be modeled using statecharts. This technique enables the creation of a complete model of event-based systems, but in non event-based systems, statecharts are useful only for a small part of the system.

The Rhapsody tool enables full cycle model-driven development [5]. In Rhapsody, model and code are strongly associated, thus the model is never outdated, while the code remains the most important artifact at the development stage as it handles all the details. Some model elements also include code. Rhapsody enables model execution and model-based testing. Although Rhapsody is one of the most powerful MDE tools, it does not include an ability to effectively reuse common design patterns and logic, besides some built-in structures.

Comparing my technique to Aspect-Oriented Modeling (AOM) [22] [20] [18] [12] and Feature-Oriented Modeling (FOM) [19], AOM and FOM do not change method’s signature (except for exceptions) and do not allow passing parameters between two aspects/features applied on the same method. Moreover, AOM does not provide aspects that may be applied to a class, enriching the class along with its methods with a complex behavior at once. Thus, my technique is more powerful, meaning more code is easier generated for non crosscutting concerns. Therefore it results in applications with higher
quality, demanding less development and maintenance effort.

Compared to non-MDE techniques such as AOP and Feature-Oriented Programming (FOP), my technique makes the development process easier and more straightforward, as the association is done in the model stage.

5.1 Comparing to AOM techniques

There are various existing AOM techniques. Here I compare my technique with some of them in particular.

AspectOPTIMA [10] is a powerful language independent, aspect-oriented framework that was built as a case study for Reusable Aspect Models (RAM) [12]. To apply aspects to an application model, RAM uses Sequence Diagrams that define pointcuts and advices (the aspect behavior at pointcuts).

RAM, similarly to other AOM techniques, has the following disadvantages:

- Weaving (the term used in RAM for models, although the actual weaving is done at the runtime by AOP implementation) of aspects with application models, is done at the aspects level, using interfaces to describe entities that should be affected. This way it serves only crosscutting concerns.

- Weaving is done at pointcuts (methods) as in AOP. My technique enables applying a concern as a whole, thus adding a general behavior that takes into account the enriched class with all its enriched methods.

- There is no change in signature for application methods.

- To pass parameters between two applied aspects, or to make them aware one of another for some other purpose, it is required to create an aspect that unites the two (see Figure 5 in [12]).

A UML profile for AOD [3] and other aspect-oriented UML extentions [7] enable to create aspect-oriented models using UML profiles (i.e., stereotypes). They model pointcuts, advices, etc., therefore, they have the same disadvantages as RAM. Moreover,
in some of them the complexity of applying an aspect to an application model (multiple
pointcuts, advices and other constructs the profiles have) bulks the model and may affect
abstraction.
Chapter 6

Conclusion

I have described an MDE technique to define common functionalities (concerns) and use them in modeling to get automatically generated code for an enriched model. It reduces the effort needed for software system development. I showed how different concerns can be applied to a single model element and how they can solve different programming and design problems (security, design patterns, exceptions handling, etc.).

My technique handles both behavioral and structural concerns, allows to define generally any functional or non-functional concern and to combine multiple concerns at a single class or method. Other MDE techniques either do not handle the behavioral aspect, or do not suit for all software systems (data management, real-time, distributed, etc.). I successfully applied my technique using a tool prototype to generate and implement examples of data management and distributed event-based systems.
Chapter 7

Future Work

In the future I plan to deal with complex concerns and automatic ordering, to check alternative ways to represent concerns and to prepare a case study.

I can think about concerns that are aware of each other, for example generalizing of another concern, aggregation of concerns and other dependencies. As described above, when enriching a model it is required to state the full order for applying concerns. It may not be necessary in every case. Moreover, frequently, two unrelated and non colliding concerns are applied together and may be executed in any order. Further research may be done on how it is possible to automatically determine (for a particular use) necessary order of concerns, even if it is not stated explicitly (also for colliding concerns). I plan to conduct a wide case study to assess the contribution of the technique to the development process and to further check its applicability.

Also, I seek for an alternative representation of concerns model with expression strength similar to the one described. I actually want to define a similar graphical representation instead of a textual one to make the model more intuitive.

I aim to integrate with other AOM techniques such as RAM [12] to benefit both and better integrate crosscutting and non crosscutting concerns.

Also, I would like to integrate my tool with other MDE tools (e.g., AndroMDA [23], model editing tools, testing tools, etc.).
Appendix A

Usage Examples Code

Following are samples of code that was used for or generated for the implementation examples presented in Sections 4.4.1 and 4.4.2.

Listing A.1 presents the generated code for the `AccountDAO.withdraw()` method as defined in Figure 4.6.

```java
public abstract class AccountDAOBase
{
    ...

    public void withdraw(java.lang.String accountNumber, java.lang.String user, double amount)
        throws example1.bankaccount.AccountException, java.sql.SQLException,
            example1.protection.ProtectionException
    {
        Connection connection = SingletonConnectionPool.getInstance().getUnallocatedConnection();
        connection.setAutoCommit(false);
        boolean success = false;
        try
        {
            java.sql.ResultSet resultSet = connection.createStatement().executeQuery("SELECT count(∗) , users.id_user , operations.id_operation
                FROM users , operations
                WHERE users.id_user = operations.id_user
                AND operations.name = "AccountDAO.withdraw"");
            if (!resultSet.next())
                throw new ProtectionException();
            if (resultSet.getInt(1) < 0)
                throw new ProtectionException();
            int userId = resultSet.getInt(2);
            int operationId = resultSet.getInt(3);
            connection.createStatement().execute("INSERT INTO operations_log VALUES(∗ + userId + ∗ + operationId + ∗ , now())");
            withdrawImpl(accountNumber, user, false, connection);
            if (getAccountBalance(accountNumber, user, false, connection) < 0.0)
                throw new AccountException();
            success = true;
        }
        finally
        {
            if (success)
                connection.commit();
            else
                try
                {
                    connection.rollback();
                } catch (Throwable e)
                {
                }
        }
        SingletonConnectionPool.getInstance().returnConnection(connection);
    }
}
protected abstract void withdrawImpl(java.lang.String accountNumber, double amount, java.sql.Connection connection) throws example1.bankaccount.AccountException, java.sql.SQLException;

Listing A.1: Generated code for the method AccountDAO.withdraw

Listing A.2 displays the plugin code for concern DistributedAgent. For each concern method, it returns code parts for code generation. There are simple text code parts as well as special ones. See Section 4.1.4 for the description of code parts.

public class DistributedAgentGenerationData implements ConcernGenerationData {
    private final GeneratedCodePart[] generalCreationData = new GeneratedCodePart[] { new SimpleGeneratedCodePart("private DistributedAgent[] neighbours; protected Integer id;"); }
    private final GeneratedCodePart[] startAgentMethodCreationData = new GeneratedCodePart[] { new EnrichedMethodCall(); }
    private final GeneratedCodePart[] endAgentMethodCreationData = new GeneratedCodePart[] { new EnrichedMethodCall(); }
        EventsManager.getInstance().sendEvent("",
            new ParameterExpression("message"),
            new SimpleGeneratedCodePart("+neighbour.getId()", "")
        ,
            new ParameterExpression("messageParameter"),
            new SimpleGeneratedCodePart("")"); }
    private final GeneratedCodePart[] onMessageGeneralMethodCreationData1 = new GeneratedCodePart[] { new SimpleGeneratedCodePart("{ EventsManager.getInstance().registerHandler(new EventsHandler()
            +"public void onTopicEvent(Object eventType, Object eventData) {
            if(eventType.equals("getId")
            .equals(eventType))
        );
    private final GeneratedCodePart[] onMessageGeneralMethodCreationData2 = new GeneratedCodePart[] { new SimpleGeneratedCodePart("(eventData);""); }
    private final GeneratedCodePart[] onMessageMethodCreationData = new GeneratedCodePart[] { new EnrichedMethodCall(); }
    private final GeneratedCodePart[] setNeighboursMethodCreationData = new GeneratedCodePart[] { new SimpleGeneratedCodePart("this.neighbours = neighbours;"); }
    private final GeneratedCodePart[] setIdMethodCreationData = new GeneratedCodePart[] { new SimpleGeneratedCodePart("return id;")"); }
    @Override
    public GeneratedCodePart[] getEnrichedMethodGeneratedCodeParts(String functionalMethodName, String enrichedClassName, String enrichedMethodName) {
        if ("startAgent".equals(functionalMethodName)) {
            return startAgentMethodCreationData;
        }
        if ("endAgent".equals(functionalMethodName)) {
            return endAgentMethodCreationData;
        }
        if ("sendMessage".equals(functionalMethodName)) {
            return sendMessageMethodCreationData;
        }
        if ("onMessage".equals(functionalMethodName)) {
            return onMessageMethodCreationData;
        }
        if ("setNeighbours".equals(functionalMethodName)) {
            return setNeighboursMethodCreationData;
        }
        if ("getId".equals(functionalMethodName)) {
            return setIdMethodCreationData;
        }
        return null;
    }
}
```java
@Override
public GeneratedCodePart[] getGeneralGeneratedCode()
{
    return generalCreationData;
}

public GeneratedCodePart[] getEnrichedMethodGeneralGeneratedCodeParts(String functionalMethodName, String enrichedClassName, String enrichedMethodName)
{
    if ("onMessage".equals(functionalMethodName))
    {
        List<GeneratedCodePart> result = new ArrayList<GeneratedCodePart>();
        result.addAll(Arrays.asList(onMessageGeneralMethodCreationData1));
        result.add(new SimpleGeneratedCodePart(enrichedMethodName));
        result.addAll(Arrays.asList(onMessageGeneralMethodCreationData2));
        return result.toArray(new GeneratedCodePart[0]);
    }
    return new GeneratedCodePart[0];
}

@Override
public String[] getImports()
{
    return new String[] {
        DistributedAgent.class.getName(),
        EventsManager.class.getName(),
        EventsHandler.class.getName()};
}
```

Listing A.2: Plugin code for the concern DistributedAgent

Listing A.3 presents the generated code for the ChangAndRobertsNode class as defined in Figure 4.9.

```java
public abstract class ChangAndRobertsNodeBase implements example2.distributed.DistributedAgent
{
    public Boolean participant;
    public Integer leader;
    private DistributedAgent[] neighbours;
    protected Integer id;

    public ChangAndRobertsNodeBase(Integer id)
    {
        // generated to ensure constructor in the implementation class;
        // it should be implemented there
    }

    public void start()
    {
        startImpl();
    }

    protected abstract void startImpl();

    public void end()
    {
        endImpl();
    }

    protected abstract void endImpl();

    public void election(java.lang.Integer idToPass)
    {
        for (DistributedAgent neighbour : neighbours)
            EventsManager.getInstance().sendEvent("ELECTION" + neighbour.getId(), idToPass);
    }

    EventsManager.getInstance().registerHandler(new EventsHandler()
    {
        public void onTopicEvent(Object eventType, Object eventData)
        {
            if ("ELECTION" + getId()).equals(eventType)
                onElection(eventData);
        }
    });

    public void onElection(java.lang.Object idToPass)
    {
        onElectionImpl(idToPass);
    }
```

protected abstract void onElectionImpl(java.lang.Object idToPass);

public void elected(java.lang.Integer idToPass)
{
    for (DistributedAgent neighbour : neighbours)
        EventsManager.getInstance().sendEvent("ELECTED" + neighbour.getId(), idToPass);

    EventsManager.getInstance().registerHandler(new EventsHandler()
    {
        public void onTopicEvent(Object eventType, Object eventData)
        {
            if (("ELECTED" + getId()).equals(eventType))
                onElected(eventData);
        }
    });
}

public void onElected(java.lang.Object idToPass)
{
    onElectedImpl(idToPass);
}

protected abstract void onElectedImpl(java.lang.Object idToPass);

public abstract void setNext(example2.distributed.Election.ChangAndRobertsNode neighbour);

public void setNeighbours(example2.distributed.DistributedAgent[] neighbours)
{
    this.neighbours = neighbours;
}

public Integer getId()
{
    return id;
}

Listing A.3: Generated code for the enriched class ChangAndRobertsNode
Bibliography


תקציר
נֶדֶסֶת הַתּוֹכָה מֶנְחָית מֶדוֹלִים נֶפֶבַּת לַשְּׁיוֹמָשֶׁית זִיְר כָּכָל שֻׁלֶּלֶכֶת הֶיֶלֶלֶם נְוִיְיָמָא זְמִיתַמַּס. נָעֲדָה עַקְרָיִירָאָה לַשְּׁיוֹמָשֶׁית נֶדֶסֶת הַתּוֹכָה מֶנְחָית מֶדוֹלִים מַעְיָמָא נְוִיְיָמָא. נִיהַוֶּה, קַוְּשַרֵיתָיו וְשְׁמוֹשָׁתָיו עַיְיֵי פַּרְדֶּה עַיְיִיִּי. אוֹאְלָא לְגָנָה לַמֶּדֶל אַחַת כֶּל הַבּוֹטְסִיָּמָא מֶסֶפְּלִמָא בַשְּׁלָלָה הָפְּמָשָׁת, אוֹחַזַּואֵה מַכֶּה אַל מֵשְׁתָּבְסָיָהּ בָּהָנְדֶסֶת הַתּוֹכָה מֶנְחָית מֶדוֹלִים בִּוּהַבָּ רְוָחְתָּ מְסָפְּקָא.
תודה

ברצוני להודותelmanhof שליד שמות וכישורוני על הה⚓ות ונסיוני, לפורפ
עמרו יהודאי על הנ戗ות וה italia. אלא سبيلות והערהות הרב לא היה נית
להשלים עבדה זו.

תודה לנדב ויינה על היכרותי עם התחום.

תודה לאוהד ברזילי ואוורן כוכבי על עזרתם.

תודות לנדב ויינה על היכרותי עם התחום.

תודה לאוהד ברזילי ואוורן כוכבי על עזרתם.

מקודש למישפחת.
יצירת אוטומטית של קוד עי″י שילוב קטעי פונקציונליות

hibor הוחפש חתול מחדרישות מהсмерт אשינורסיטה (M.Sc) באוניברסיטת תל-אביב

מאת
איגור גלפגט

הועברה הוכנה בהדרכה של פרופ' עמירם יהודאי
ודר' שמואל טישברוביץ

חשון תשע"ב