A Review: Analysis of Aspect Orientation and Model Driven Engineering for Code Generation

Dhiraj Gurunule\textsuperscript{a,}\textsuperscript{*}, Prof. Madhu Nashipudimath\textsuperscript{b}

\textsuperscript{a}Information Technology Department PIIT, New Panvel, India
\textsuperscript{b}Computer Department PIIT, New Panvel, India

Abstract

In the development of large and complex software applications, software engineers have to focus on many requirements other than the desired application’s requirements at the coding and design level. Code Generation is a technique which is used to automatically generate lower level executable code from higher level design artifacts. Code generation provides design of the code at a higher abstract level so that software developers can focus on a higher level design problem simultaneously meeting goals of a desired application. Aspect Orientation (AO) is characterized by identification and separation of different concerns and encapsulates them in modules. Concern is an interest which pertains to a system operation, function, development or any other thing which is important to one of the stakeholders. Model Driven Engineering (MDE) is a development paradigm which is characterized by model transformation and uses models to support various stages of the development life cycle. Model is primary artifact in MDE.

In this paper, we analyze both techniques i.e. AO and MDE, and how they can be used for code generation.

© 2015 The Authors. Published by Elsevier B.V. This is an open access article under the CC BY-NC-ND license (http://creativecommons.org/licenses/by-nc-nd/4.0/).

Peer-review under responsibility of scientific committee of International Conference on Advanced Computing Technologies and Applications (ICACTA-2015).

Keywords: Aspect Orientation, Crosscutting Concern, Aspect Oriented Software Development, Model Driven Engineering, Model Transformation, Code Generation.

---

* Dhiraj Gurunule. Tel.: +918237662511.
E-mail address: dhiraj.gurunule1991@gmail.com
1. Introduction

Aspect-orientation provides modularization by mechanism to separate the crosscutting concern. It has roots at the programming level but also can be applied over other development phases. There are approaches for dealing aspect orientation at modeling level [1]. Concepts from the programming level are often simply reused without proper adaptation. Building software applications that handle all these functional and non-functional requirements is ever more complex activity that requires appropriate programming languages and development paradigms to adequately address all these requirements throughout the entire software development lifecycle.

Model-Driven Engineering (MDE) techniques consider models as the primary development artifact and use them as a basis for obtaining an executable system in different ways[10]. Model Driven Engineering takes model as primary artifact and describes standard way to represent specification of model and its transformation. Model-driven code generation has been investigated in traditional and object-oriented design paradigms; significant progress has been made. It offers many advantages including the rapid development of high quality code; reduce error, maintaining consistency between design and code than manual approach.

Automatic code generation [5] is a well-known way of getting executable code of a system from a given design model. Code generation is the process by which higher level input is converted into the lower level. It is the process of transformation of code from one representation to another; it may be platform independent or platform specific [7]. In the process of software development for specific application, work is initialized with some high level description of given of system for understanding the outline of given application. This outline description is more generic than programming language. Here a program that writes other programs are code generators. This code generator automatically maps the high level software description into some executable code which is specific to target platform.

This Study analyzes both of the techniques i.e. Aspect Orientation and Model driven Engineering (MDE). For this we provide some introduction to be familiar with concept in section 1 we give background about both techniques in section 2. For this we describe aspect orientation and its use for code generation in section 3. Section 4 provides details about Model driven engineering (MDE) and its application to code generation. In section 5 we gives brief comparison of above two techniques and in section 6 we conclude that both techniques can be combined for code generation.

2. Literature Survey

Aspect-Orientation is basically originated at programming level but it is also useful to apply its concept to software development paradigm. Applying concept of aspect-orientation to software development is called as Aspect Oriented Software Development (AOSD). Brichau and D’Hondt [1] points out that developing software using AOSD technology is results in better implantation structure of software and increases software quality. The need of aspect-orientation for software development, its contribution to enrich implementation structure and design of software is can be found in [1] along with all the terminology of AOSD. Van den Berg, Conejero and Chitchyan [2] and Wimmer et al. [4] provides common ontology i.e. shared meaning of terms involve in AOSD.

According to Hecht, Piveta, Pimenta and Price [5] high-level programming and code generation are techniques used to accelerate the transformation process between the design of software and its implementation in executable code. Code generation reduces the effort to write programming code manually and also lessen the chances of manual programming error like spelling mistake.

Code generation is the process of transforming design level constructs to programming constructs. Automatic code generation is converting software design into executable code without or little programmer intervention. Aspect Oriented code generation is the technique which combines aspect oriented programming [3] and automatic code generation techniques. According to Hecht, Piveta, Pimenta and Price [5] aspect oriented code generation is automatic code generation methods with aspect oriented concepts within it. The advantages of aspect orientation can be applied to code generation. To apply Aspect-orientation to code generation, separation of crosscutting behavior should be present at the design and coding level. Groher & Schulze [15] explained an approach to separate and encapsulate design of crosscutting behavior by using UML extension mechanism while an approach which provides separation and encapsulation for code of crosscutting behavior is explained by Dijkstra [3].
The maturity [5] of aspect-oriented software modeling approaches provides support for the automatic generation of aspect-oriented code. In [5] Hecht, Piveta, Pimenta and Price describe several means for automatic code generation from Theme/UML models, and discuss some difficulties involved in this process. They describe how to generate code in the AspectJ [13] language from UML models extended according to Theme/UML with all the detail about generator, their specification, requirement etc.

Hemel [8] identify techniques for improving separation of concerns in the implementation of code generators. They propose the core technique called as code generation by model transformation, that is, the generation of a structured representation (model) of the target program instead of plain text. To explain practical use of code generation as a form model transformation Kundu, Debasis, and Rajib [9] propose a method for generation of code from UML sequence diagram, they propose a method which start with systematical conversion of sequence diagram into Sequence Integration Graph (SIG) then to message node table and finally to code. They validate their approach by giving example for “book Issue” use case of “Library Management System”.

So the combination of Aspect-Orientation and Model Driven Engineering can be used for improving software development process. According to Mehmood and Jawani [14] the integration of Aspect-Orientation and Model driven Engineering can be done in two ways (1) using aspect orientation at modelling level producing object oriented code, or (2) producing aspect oriented code from aspect oriented design model.

3. Aspect Orientation

Aspect-orientation provides a new way of modularization by clearly separating crosscutting concerns from non-crosscutting ones. Although originally emerged at the programming level, aspect-orientation meanwhile stretches over other development phases. In the following section we describe the basic functionality of Aspect Orientation and its use to code generation.

3.1. Separation of Concern

The idea of separation of concern is stated by Dijkstra [3] is the identification of different concerns in software and their separation by encapsulating them in appropriate modules or parts of the software. Separation of concerns simplifies system development by allowing the development of specialized expertise and by producing an overall more comprehensible arrangement of elements [2].

A concern is something that interests stakeholders (programmer, developer, user etc.) because it is important or affects system under development. The software application consists of various concerns which should be identified and implement by developer. Concerns of software application can be related to its functionality, behaviour or any other software qualities that matters to the stakeholders. For example, in a mobile phone application implementation of calling function and contact list are required to be in phone, here both implementation of the calling function and maintaining contact list are two separate concerns.

According to principle of separation of concerns [1] every concern which affects the software application should be encapsulate in separate module for entire development process. To explain above principle, Brichau and D’Hondt [1] give an example of designing of the building plan as shown in figure 1a [1]. Here architect creates different blueprints describing single part of the building such as pipeline, electricity etc. Due to this separation the blueprints are easy to understand and easy to construct given building. So in software engineering this principle suggests that each concern of the software application must be implemented in separate software entity or module.

3.2. Crosscutting Concern

Object-orientation provides support for separation of concerns by dividing an application into individual objects. In best case, each object consists of only single concern. Object-oriented technology (OOT) has sufficiently developed to achieve this. Though OOT has developed so far, there are many concerns whose implementation will spread along different objects which is show in figure 1b [1].This spreading of concerns along different objects is called crosscutting of concerns. Crosscutting concerns creates trouble because their implementations causes tangling [2] and/or scattering [2] in software application.
3.3. Aspectual Decomposition

To eliminate the problems of tangling and scattering there is need to modularize crosscutting concern and therefore software developers require different (de) composition techniques. Modules in contemporary programming languages and paradigms are all based on some form of functional (de)composition (e.g. subroutines, functions, objects, components). Aspect-orientation proposes a fundamentally new kind of modularization that goes beyond generalized procedures: an aspect [5] and base [5]. Where the crosscutting concerns are modularize in Aspect and Base.

An Aspect composes of Aspect Applicability and Functionality code [1]. In aspect orientation Joint Points [1] and Pointcuts are Aspect Applicability Code while Advice [1] is Aspect Functionality code. In Aspect Oriented paradigm composition of Aspect with Base (business logic) is called weaving [1].

3.4. Aspect Oriented Code Generation (AOCG)

The process of code generation requires: 1) a model of the desired system described within a meta-model having details of the structure to be generated. 2) The specification of the elements of the target domain and 3) the set of rules to combine above two. A code generator specification can be divided in two main pieces: the code fragments and the production rules as shown in figure 1c [5]. Code fragments are pieces of programs or programming construct written the generator’s target language. Production rules define how the elements of meta-model (model of the desired system or source model) should be converted into the code fragments in the output files. Thus above code fragments are assembled to generate source code in the output file by using production rules and model of the desired system.

Aspect-oriented code generation (AOCG) can be achieved by applying concepts of aspect orientation to the automatic code generation. To achieve above combination there are two approach [5]: 1) code generator itself is developed by using aspect oriented programming, or 2) a code generator generates aspect-oriented code. This paper discuss the second approach.

3.5. Application of Aspect Orientation for Code Generation (AOCG)

Following are the approaches in which aspect orientation can be apply for code generation.

• First Approach: In first approach [5] code generator is developed by using aspect orientation. The code of code generator application will be developed by using aspect oriented programming. Some ideas from aspect-orientation may be useful for the creation of code generator. It can, for example, be used to parameterize a generator, so that it...
can generate different outputs according to the aspects selected to be weaved with it. Each functional and non-functional requirement of generator is encapsulated in separate concern i.e. production rule, code fragments etc. The weaving of all concerns results in code generator [5].

- Second Approach: In the second approach code generator generates aspect-oriented code. To realize the given approach target language must support Aspect Orientation for example AspectJ is the modified form of Java to support Aspect-orientation. For this approach, crosscutting behavior of application must be separated at design and coding level. For above constraints aspect oriented model should be present at design level. Existing modelling language (UML) does not support separation of crosscutting behavior. Following paragraphs explains how crosscutting concerns are separated at design and coding level.

To design aspect-oriented model of the desired system Groher and Schulze [15] suggest a design notation based on standard UML. This design notation separates crosscutting behaviour of application at the modelling level by using UML extension mechanism. In this mechanism aspect and base elements are separated such that there is no direct connection between them. Given UML extension mechanism provides the techniques to customize the UML for a specific domain, e.g. aspect-oriented modelling. In this extension mechanism the model elements can be customized and extended with new semantics by using stereotypes [15], constraints [15], tag definitions [15], and tagged values [15]. The following paragraphs explain design of aspect-oriented model with example.

To represent crosscutting behaviour in UML Groher & Schulze [15] suggest package level decomposition in which the notation includes a base package (containing the business logic), an aspect package (containing the crosscutting concern) and a connector to link aspect and base elements as shown in figure 2a [15].

Fig.2 a) package level decomposition b) security design example

Here aspect and base are separated and they communicate only through connector. The connector consists of require implementation of aspect oriented technology. [15] The example in Figure 2b [15] shows how to model an aspect related to security (authentication) which describes guidelines and indications on how to use these notation. In figure 2b every time the user performs an invocation on the Server, he is authenticated. Both base package and aspect package are independent from each other so no connection is modelled inside. The connector, specifying the weaving rules, includes program execution points (pointcuts) and actions to be performed at those points (advices). The pointcuts are triggered every time when client invokes the Server.getValues() method, the actions to be performed before the method call are reading and checking the user data to authenticate him. So in this way crosscutting behaviour is supported at design level by UML extension mechanism.

The requirement, code of crosscutting behaviour is fulfilled by AspectJ [13] language. AspectJ is an AOP language with low-level support for specifying and composing crosscutting code into a core system. AspectJ is a simple and practical aspect-oriented extension to Java. With just a few new constructs, AspectJ provides support for modular implementation of a range of crosscutting concerns. All the terminology of aspect orientation is well supported by AspectJ. Kiczales et al. [13] demonstrate the support of AspectJ language towards aspect orientation.
So by using UML extension mechanism aspect oriented model of the system can be developed at design level and AspectJ language provide the required aspect-oriented code. In this the crosscutting behaviour can be modelled at both design and code level.

Aspect oriented code generator is implanted as an automatic code generator capable of transforming an aspect-oriented model into aspect-oriented code. The model transformation process is explained in section 4.4.

4. Model Driven Engineering

The model-driven engineering (MDE) suggests a software development approach in which first the abstract model of the desired system is developed, which is then transformed into real, executable systems (e.g. source code) in several iterative steps.

Figure 3a [10] shows the process of model driven engineering. It shows that first Model n is created which is more abstract and less platform specific. Then Model n goes through series of transformation with gradually decrease in abstraction level and increase in platform specificity up to Model 1. Here Model 1 is platform specific model. Each model conforms to its respective modelling language. At the end Model 1 is transformed to source code (e.g. Java code).

![Fig 3a](image)

MDE is a promising approach to deal with the complexity of platforms (which is not effectively decreased by using third-generation languages), as well as express domain-specific concepts [10]. Transformation engines and code generators are meant to understand the information contained in the model. The purpose is to produce other types of artefacts (semi-)automatically (for e.g. More detailed models, source code, simulation inputs, components configuration files, and other models). Hence these are combined with modelling languages in MDE [6].

The name MDE stresses the fact that models are the focal point of MDE. In present scenario models related to software development are considered. A model needs to possess three features / meet three criteria otherwise it is not a model [12]:

- **Reflection/Mapping**: A model is based on an original. The original (system) might be something yet to be built or it may remain completely imaginary.
- **Abstraction/Reduction**: Not all the properties of the subject are mapped onto the model, but the model is somehow reduced. However, a model must mirror at least some properties of the subject.
- **Substitute/Pragmatic**: A model needs to be usable in place of a subject with respect to some purpose. A model always has some relation with a subject (mostly a system, but it can also be another model).

In general MDE consists of the following two main artefacts: Modelling languages, and model transformations which are describe in next section.

4.1. Modeling Language

In MDE models are described by modelling language so that the meaning of a model would be well-defined and different developers can understand and work with the models easily. Because if meaning of the model is not clear
then it is difficult to transform it into another model. Modelling language consists of an abstract syntax [10], at least a concrete syntax [10] and semantics [10]. The hierarchy of modelling language is explained in next section 4.2.

4.2. Four Layer Meta-modeling Architecture

The meta-modelling provides definition of modelling language. Model Driven Engineering (MDE) is uses modelling languages which it is based on the four-layer modelling architecture [10]. Four-layer meta-modelling architecture is specification for modelling standardize by Object Management Group (OMG) [17]. It consists of four layers M0, M1, M2, and M3. The components or specification of model at M1 level are defined by Meta-model at M2 level. This meta-model describes what model should contain. Similarly components or specification of meta-model at M2 level are defined by Meta-meta-model which is at M3 level. Instances at M0 level are the objects which are used in final design model. The meta-models are abstract syntax of modelling language while objects at M0 level are instances of model which are concrete syntax of modelling language. Table 1 [18] articulated the four layer meta-modelling architecture with example.

Table 1: Four layer meta-modelling architecture

<table>
<thead>
<tr>
<th>OMG levels</th>
<th>OMG Standards/examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>M3: Meta-meta-model</td>
<td>MOF</td>
</tr>
<tr>
<td>M2: Meta-model</td>
<td>UML language</td>
</tr>
<tr>
<td>M1: Model</td>
<td>A UML model: Class &quot;Car&quot; with attributes &quot;name&quot; and &quot;ModelNo&quot;</td>
</tr>
<tr>
<td>M0: Instance</td>
<td>An instance of &quot;Car&quot;: &quot;GT&quot; Model No. &quot;GT150&quot;</td>
</tr>
</tbody>
</table>

4.3. Model Transformation

A model transformation automatically generates a target model from a source model, according to a transformation definition [7]. An overview of all relevant artefacts which are involved in defining and implementing model transformation is shown in figure 3b [10]. Transforming a model into another model means that a source model is transformed into a target model based on some transformation rules [7]. Different methods can be used for defining the transformation rules. In 2007 the OMG (Object Management Group) [17] released QVT [17], a model transformation language specification.

In the transformation tool many elements are involved in performing the transformation. Inside the tool there lies definition that describes how a model should be transformed. In the following, we present the idea of model transformations definition and transformation rule.

4.4. Code Generation As a Form of Model Transformation (CGAFMT)

The essence of the approach is to shift the knowledge about implementation details from the minds of programmers to the templates of code generators that automatically translate models into implementations [8]. Since even the generated code has a structured model representation to which transformations can be applied, any restrictions in modularity of the target language can be alleviated by extending it with new constructs to support better modularity. For example [9] target language java is extended by adding some partial classes, interface extraction, and name generation in order to simplify code generation rules.

In code generation as form of model transformation we transfer source model to target model or code with the help of code generator. This process is shown in following figure 4[16].
Source model is a model that should be transformed to code. It is defined by its language specification and may be a model as a result of various transformations and is ready for final conversion to the code. Target is the final target code in some language specification for which all transformations have been done. Both target and source model conform to the respective meta-language. Target code is platform-specific and is also referred to as program. Transforming a source model to target code is the responsibility of the code generator and code generator is dependent on the code generator definition for correct transformation. The generator relates the source model to target code. Code generation definition consists of a set of transformation rules which are explained in model transformation. In simple terms, a practical generator can be represented as in Figure 4 [16]. The application of Model transformation for code generation is explained in the next section.

4.5. Application of Model Transformation for Code Generation

Kundu, Samanta, and Mall [9] provide an approach to generate executable code from UML sequence diagram. We will see how these approaches are applicable for CGAFMT. In their approach, first a graph is generated from a UML sequence diagram (SD) and then it is converted to source code. Diagrams in Figure 5 [9] show how a sequence diagram is represented in graph. In a sequence diagram (Figure 5a) there is a condition fragment in which if-condition is true then one message is executed otherwise the second message is executed. This behavior of the SD is represented in graph as if-condition is true then the left branch is traversed else the right branch is traversed.

Now source code can be generated from the graph or a subset of the graph by using the set of transformation rules mentioned as a transformation definition as shown in Figure 6 [9]. The transformation rules of transformation definition specify how the contents of the graph should be converted into the programming code artefacts. As shown in Figure 6 [9] transformation definition consists of four columns. The first column is the rule number, the second column has elements of the graph, the third column has a branch reference i.e., given node is at which branch (left, right, or middle) of the graph. The elements in the fourth column are programming constructs that should be mapped for corresponding elements in the second column.
The algorithm for code generation takes graph as an input and generates source code by using transformation definition. The algorithm traverse the graph in depth-first-search order and the elements of graph are mapped to programming constructs according to branch reference. For example in rule 1 the given graph element is mapped to code “if (“, if it is at left branch of the graph. This process can be summarized in figure 6 [9].

How the subset of graph model is converted into code artefact has been explained in above paragraph, now each subset will produce different code artefact and assembling these artefacts will make target program. In this ways the techniques of model transformation can be applied to automatic code generation.

5. Comparison

Table 2: Comparison of Aspect Orientation and Model Driven Engineering

<table>
<thead>
<tr>
<th>Aspect Orientation</th>
<th>Model Driven Engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Application logic is split up by principle of “Separation Of Concern”</td>
<td>Models are described by modeling languages</td>
</tr>
<tr>
<td>Various concerns are makeup whole application</td>
<td>Different models go through MDE to make final product.</td>
</tr>
<tr>
<td>Important process is “Aspectual Decomposition”</td>
<td>Important process is “Model Transformation”</td>
</tr>
</tbody>
</table>

Weaver perform same role in both

6. Conclusion

Research in the area of aspect-oriented modelling and model driven code generation can result in significant advancement in development of software systems that are more maintainable, extensible and reusable. Aspect orientation is a technique for separating crosscutting concerns and MDE is an approach to automatically transform model. In this paper we defined how MDE is used to specify the representation of model and how to transform one model to another. With the example of sequence diagram we have shown that MDE can be used for code generation.
The research on integration of aspect-oriented software development and model-driven engineering process can result in enabling software development to produce more maintainable, extensible and reusable systems. In this paper existing Aspect-oriented and Model Driven approaches are evaluated which can be used for integration of both techniques and use for code generation.

References