

Evolution of Java Programs to a Model-Driven Environment Using EMF

Gerald C. Gannod*†

Division of Computing Studies
Arizona State University - East Campus
7001 East Williams Field Road, Mesa AZ 85212
gannod@asu.edu

Maurice M. Carey

Dept. of Computer Science & Engineering
Arizona State University - Tempe
Box 878809, Tempe, AZ 85287-8809
mmcarey@asu.edu

Abstract

A challenge in evolving legacy systems is the need to provide continuity for both users and developers. For users, modifications of any nature must ensure integrity of usage. For developers, modifications to source code of any nature must ensure integrity of source code appearance. While changes are inevitable, the resulting product must have some link to the past (e.g., the code must be recognizable in relation to the original form). The Eclipse Modeling Framework (EMF) is a facility within Eclipse that supports certain aspects of the MDA philosophy. Accordingly, we are developing an approach for facilitating the evolution of Java programs to the EMF framework. With the popularity of Eclipse, the approach has a potential for impact, especially given the alignment with the EMF philosophy that focuses both on models and code.

1. Introduction

Recently, approaches and tools based on the use of Model Driven Architecture (MDA) have received more attention [1, 2, 3]. The primary goals of MDA are portability, interoperability, and reusability through an architectural separation of concerns between the specification and implementation of software. As new technologies, such as MDA, continue to be developed, new and innovative ways of incorporating and transforming legacy systems (e.g., systems with a long organizational history and commitment) so that they can take advantage of benefits of the new technologies are sought.

A challenge in evolving legacy systems is the need to provide continuity for both users and developers. For users, modifications of any nature must ensure integrity of usage. Specifically, the system operations must appear to be unchanged or at least consistent with the expectations that a

user has about the use of the system. For developers, modifications to source code of any nature must ensure integrity of source code appearance. While changes are inevitable, the resulting product must have some link to the past (e.g., the code must be recognizable in relation to the original form).

The Eclipse Modeling Framework (EMF) [4] is a plugin for Eclipse [5] that facilitates certain aspects of the MDA philosophy. That is, the EMF philosophy is to take the middle ground between a full MDA approach where the model is paramount and the “code-is-self-documenting” approach prevalent in many development communities. The goal of EMF is to provide an environment that facilitates the automatic generation of different kinds of UML-based models while supporting translation of those models into XML representations and Java-based source codes.

In previous work we developed *adapter* and *wrapper*-based approaches for prolonging the lifetime of legacy systems while still utilizing new technologies [6, 7]. In these approaches, the advances were of the technology sort. Specifically, we were interested in utilizing interconnection technologies such as Jini and .NET. In our current work, rather than using legacy applications as black boxes, we are developing an approach for transforming the source code of existing systems into forms suitable for supporting reverse and re-engineering. Accordingly, we are developing an approach for facilitating the evolution of Java programs to the EMF framework. The advantages of the approach are two-fold. First, with the popularity of Eclipse, the approach has a potential for impact, especially given the alignment with the EMF philosophy that focuses both on models and code. Second, the approach supports continuity for both users and developers. Specifically, we hold to the tenets that both users and developers should find transitioning to a model-driven approach to be as seamless as possible.

The remainder of this paper is organized as follows. Section 2 describes the context for our approach including information on MDA and EMF. Section 3 discusses the migration approach. A simple example illustrating the ap-

*This author supported by National Science Foundation CAREER grant No. CCR-0133956.

†Contact Author.

proach is presented in Section 4, and Section 5 concludes and suggests future investigations.

2. Background

Model Driven Architecture (MDA) [1] is a standard produced by the Object Management Group (OMG). The goal of MDA is to separate the design of application or business logic from the implementation platform. Designs are specified in a platform-independent model (PIM) that can then be translated to a platform-specific model (PSM) by an MDA tool, which is then used to generate the implementation. MDA is dependent on and makes use of several other OMG standards including the Unified Modeling Language (UML), Meta-Object Facility (MOF), XML Meta-Data Interchange (XMI), and Common Warehouse Meta-model (CWM).

Eclipse [5] is a project to develop a universal IDE. Eclipse is a platform for developing an IDE. Eclipse is an IDE and framework that supports Java development. The Eclipse Platform is designed to be flexible enough to use as the base for an IDE for any language; there are several available including the Java Development Tools (JDT). The JDT is the plug-in extension to the Eclipse Platform which supports all Java development. The Eclipse Plug-in Development Environment (PDE) supports plug-in development and is used extensively in the work described in this paper.

Eclipse Modeling Framework (EMF) [5] is a plug-in framework for modeling Java applications that includes a code-generation ability. In the EMF framework, three different kinds of models, namely UML-like *ECore* models, code models (e.g., Java), and XML-based models are all mapped to a representation that allows for translation between models. While the base EMF currently lacks the ability to generate source code for behavioral models such as UML statecharts, it does provide support for extension points (e.g., plug-ins) that allow developers to add missing MDA capabilities. In this sense, EMF is an MDA *halfway-house* of sorts. EMF is positioned in the middle of a spectrum for software development with MDA on one end and fully manual programming on the other. EMF achieves this goal by having three different methods of creating EMF models. First, an EMF *ECore* model can be created using a subset of UML modeling (EMF does not currently support all of the UML specification). Second, an EMF model can be created using an XML Schema. Finally, an EMF model can be specified using specially annotated Java code.

3. Approach

This section discusses the approach we are developing for supporting migration of Java code to the Eclipse Modeling Framework.

3.1. Philosophy

Our approach for facilitating evolution of legacy systems to MDA environments is based on two primary requirements:

R1: The user experience should be unchanged in the first generation of migration.

R2: The developer should be able to easily recognize and identify software artifacts that appeared in the input system.

Requirement *R1* forces developers of migration approaches to adjust to the user rather than having the user adjust to potential changes in a system. For instance, if the original source system is launched on the command-line, then the target system should also be launched on the command-line with the exact same parameters as the source system. To the users, the migration to the new development platform should be seamless. That is, the user should have no knowledge that the migration to the new development platform even occurred.

Requirement *R2* forces developers of migration approaches to ensure that a level of traceability exists between the original source system and the target system. Specifically, the source code of the target system should have elements that are recognizable by the developers and maintainers of the source system. For instance, if a developer applies a migration tool, software design elements present in the original system should also appear in the target system and should ideally have a structure that is similar, if not exactly the same, as the original system.

Many MDA tools produce source code in a form that is difficult for developers to comprehend. Source code for a legacy system typically has a long history and, as such, developers have intimate knowledge of the inter-workings of the system. Accordingly, our approach is based on the following underlying philosophy, that:

Source code for target systems should be directly maintainable by developers.

The benefits of this philosophy are two-fold: 1) any such approach, especially one targeted at legacy systems, can ease the process of migrating people (not just systems) to new technology, and 2) target systems can benefit from both the MDA approach and developer *know-how*.

3.2. Context and Constraints

Given these requirements and philosophies, we have found that our evolution approach most closely follows the strategy taken in the Eclipse Model Framework (EMF) [4] as opposed to approaches that might be considered more advanced such as Rose Real-Time [2], and OptimalJ [3].

Currently, the tool we are developing supports only Java, a limitation imposed by the EMF framework. We are investigating the constraints of our approach in terms of the characteristics of the codes that can be processed. To date we have been able to handle both single and multi-threaded applications as well as standalone and distributed applications (the example presented later in this paper is a simple standalone but multi-threaded application). Certain styles of programming, including the use of public inner classes, are not currently supported. Future implementations will address such limitations.

The output of the tool is a set of EMF Ecore models. These Ecore models act as a representation that EMF uses to translate from one form of model to another. Java source code (including behavior) is retained from the original input system and integrated into EMF but is restructured into the code standards necessary to properly support further development from within EMF. With respect to MDA terminology, the transformations we perform facilitate generation of PIM-level models from code. However, there is an underlying sequence of steps that achieve transformations from code to PSM-level models with PSM-level to PIM-level transformations being achieved by the EMF support tools.

3.3. Process

In the typical case, EMF works as follows. A developer creates a model (e.g., a PIM) using an Ecore editor such as the Omondo EclipseUML tool [8]. The model is then used to generate a number of different representations at different levels of abstraction including an XML Schema and Java code. An alternative way of generating models is to annotate Java source code with tags during development. These tags indicate to the EMF tool relationships between classes and certain constructs within those classes. Similarly, you can use an XML schema to generate Java code and Ecore models.

Table 1 provides a list of tags that are used by EMF to indicate that entities within Java source code are of a certain type. These tags are applied primarily on Java interfaces to indicate relationships between the class implementing an interface and other classes in a system. In addition, these tags allow EMF to translate Java source into corresponding model types and vice versa. From the standpoint of MDA terminology, these tags amount to *markings* that enable the creation of transformations from PIM-level Ecore models and source code.

The approach that we are currently developing is based on the following idea. Utilizing a number of refactoring operations (both built-in to Eclipse and developed by our group), we perform a number of automatic transformations on existing Java code (via tags) to ensure that the existing system is properly imported into the EMF framework.

Tag	Description
EClass	Element is a class
EAttribute	Element is an attribute of a class
EReference	Element is a reference to a class
EOperation	Element is a method of a class

Table 1. Summary of EMF Tags

Specifically, we automate the annotation task required by EMF, restructure the code into appropriate interface and implementation entities, and automatically generate necessary adapters to ensure that the application remains executable. As a result, the original source code can be incorporated into the EMF MDA environment, allowing an organization to retain a valuable asset.

The entire process can be summarized as follows: 1) *Refactor original system*, 2) *Annotate refactored code with EMF tags*, 3) *Create adapters (under certain conditions)*, 4) *Prepare execution classes*, 5) *Generate Models*, and 6) *Generate Code*. Steps 1, 5, and 6 are largely a result of using either Eclipse [5] or EMF [4] plug-ins and support tools. Our current activities are focused upon Steps 2, 3 and 4. Step 4 is specifically for addressing requirement *R1*, while the mere fact that we use the Eclipse system and EMF framework addresses requirement *R2*. From the standpoint of reverse engineering, Steps 2 thru 5 embody a design recovery activity, with Steps 2 – 4 being performed by our tools and Step 5 by the EMF facilities. As such, those steps alone, when using EMF, result in the creation of UML diagrams complete with inheritance, association, composition, and aggregation relationships, including multiplicities.

Upon completing the above steps, a system will be completely integrated into the EMF framework at which point developer will be able to use Eclipse and EMF to perform modifications at the model level to introduce structural changes into a system. In our future investigations, we intend to develop modifications as Eclipse plug-ins to allow for introducing behavioral changes.

Figure 1 shows a UML diagram depicting the design of a trivial system prior to using our tools. The diagram shows two classes, `MyClass` and `MySuperClass`, and a subclass relationship between the two. In the first step of our approach, the source is refactored in order to restructure the code so it adheres to the code standards used by EMF. Namely, classes are restructured to use an *interface-implementation* style so that every standard class in the original system becomes represented by a Java interface and Java implementation class. The refactorings are implemented using built-in eclipse refactoring tools including *renaming*, *interface extraction*, and *package movement*. Figure 2 shows a UML diagram representing a PSM-level model of the system after refactoring. It should be noted that at this stage of the process, the subject system is not executable. However, once the entire process completes, the

final target system is executable.

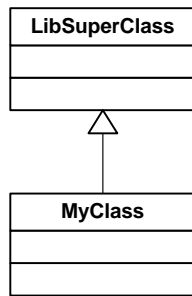


Figure 1. UML Depiction of Original Structure

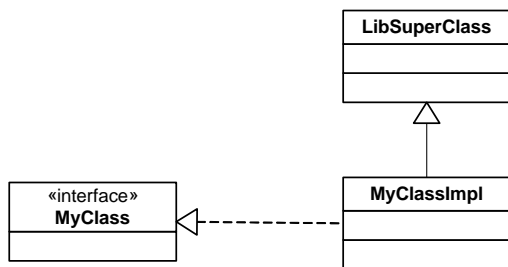


Figure 2. After Refactorings

In addition to annotating the source code (e.g., interfaces) with EMF tags to indicate relationships between classes, subclass-superclass relationships between developer classes and library classes must be mediated with an adapter to allow for the introduction of an EMF-specific class called an EObject, as shown in Figure 3. EObject is conceptually at the PSM-level and is necessary to achieve proper integration of library classes (e.g., Java SDK libraries) into EMF. Specifically, every class in EMF should implement EObject (it is an interface). EObjectImpl should be extended to get the implementation of EObject that EMF expects. The problem is when an existing developer class already extends another class (like Thread). Since Java does not allow multiple inheritance, a delegate class is inserted into the hierarchy. The delegate extends the base and implements the interface to EObject by passing calls to the member EObjectImpl. The need for the EObject interface is to allow EMF to make certain assumptions about objects and how they are created, initialized, and serialized.

The diagram in Figure 3 is conceptually considered a PSM-level model in the sense that it represents the system according to Java-specific constructs that are necessary to implement the system. By contrast, Figure 1 is a PIM-level model. Upon completing the process described above, including using EMF to complete the model generation process by importing the transformed Java code into an EMF

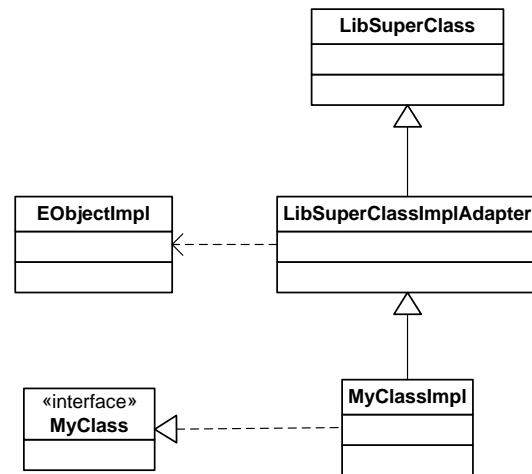


Figure 3. After Annotation and Adapter Generation

ECore model, Eclipse will present a user with a model similar to that of Figure 1.

4. Example

As a simple example, we created a multi-threaded “Hello World” program to illustrate the before and after-effects of applying our tool. Figure 4 shows the code prior to performing transformations and refactorings while Figure 5 shows one of the generated interfaces that has been annotated to allow for inclusion in EMF. Note that the HelloWorld class extends the Thread class, which is part of the Java libraries.

```

public class Hello {
    public static void main(String[] args) {
        HelloWorld hw = new HelloWorld();
        ((Thread)hw).start();
    }
}

public class HelloWorld extends Thread {
    private String helloMessage =
        "Hello world!";

    private boolean printed = false;

    public boolean isPrinted() {
        return printed;
    }
    public String getHelloMessage() {
        return helloMessage;
    }
    public void printHelloMessage() {
        System.out.println(getHelloMessage());
        printed = true;
    }
    public void run() {
        printHelloMessage();
    }
}
  
```

Figure 4. Sample Code

```

/**
 * @model
 * EClass
 * @author Maurice
 */
public interface HelloWorld {
    /**
     * @model
     * EAttribute
     */
    public abstract boolean isPrinted();
    /**
     * @model
     * EReference
     */
    public abstract String getHelloMessage();
    /**
     * @model
     * EOperation
     */
    public abstract void printHelloMessage();
    /**
     * @model
     * EOperation
     */
    public abstract void run();
}

```

Figure 5. Portion of Generated Interface

The code for the HelloWorld class in the lower portion of Figure 4 is retained except that the class is renamed (HelloWorldImpl), the class it extends is changed (HelloWorldImplAdapter), and it acts as an implementation of an interface (HelloWorld). The HelloWorldImplAdapter class is created in order to allow for the inclusion of the EObject class in the inheritance path.

Figure 6 shows a resulting Platform Independent Model (PIM) for the example program. The diagram was generated using the Omondo EclipseUML [8] plug-in, which supports EMF. The diagram was generated after conversion of code shown above. Note that the structure of the diagram is the same as would appear for the code shown in Figure 4. That is, the recovered diagram retains the structure of the original system.

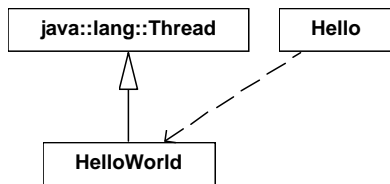


Figure 6. Example PIM

Figure 7 shows the Platform Specific Model (PSM) for the example program. Again, this diagram was generated using the EclipseUML plug-in. In this case, the diagram reveals dependencies and relationships that were introduced by migrating the code into EMF and would generally be hidden from a developer. However, given our philosophy

of allowing a developer to directly maintain code as well as diagrams, the ability to access the PSM is ultimately necessary and desired.

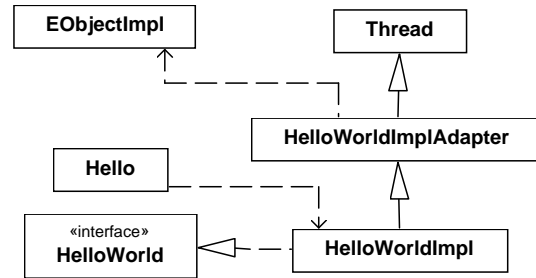


Figure 7. Example PSM

5. Conclusions and Future Investigations

To date we have performed proof of concept application of our approach to programs of limited size but with good success. Given our use of Eclipse, we are currently limited to the Java programming language. We are in the process of fully automating the migration process. Currently, each step of the process is automated; the transitions between the steps has yet to be integrated. Our immediate plans for evaluation of the approach will include systems of small, medium, and large sizes. In addition, we are currently investigating the development of tools that support operations necessary for manipulating systems once they have been integrated into EMF including behavioral code synthesis.

References

- [1] Joaquin Miller and Jishnu Mukerji et al. MDA Guide Version 1.0.1. Technical Report omg/2003-06-01, Object Management Group, June 2003.
- [2] IBM. Rose real-time. [Online] Available <http://www-136.ibm.com/developerworks/rational/products/rose>.
- [3] Compuware. OptimalJ. [Online] Available <http://www.compuware.com/products/optimalj/>.
- [4] F. Budinsky et al. *Eclipse Modeling Framework : A Developer's Guide*. Addison-Wesley, August 2003.
- [5] S. Shavor et al. *The Java Developers Guide to Eclipse*. Addison-Wesley, May 2003.
- [6] Gerald C. Gannod, Sudhakaran V. Mudiam, and Timothy E. Lindquist. Automated Support for Service-Based Software Development and Integration. *Journal of Software and Systems Special Issue on Automated Component-Based Software Engineering*, 2004 (in press).
- [7] Gerald C. Gannod, Huimin Zhu, and Sudhakaran V. Mudiam. On-the-fly Wrapping of Web Services to Support Dynamic Integration. In *Proc. of the 2003 Working Conference on Reverse Engineering*, pages 175–184. IEEE, November 2003.
- [8] Omondo. EclipseUML. [Online] Available <http://www.omondo.com>.