**CSE 460/598**

**Computer Science Program**

**School of Computing, Informatics, and Decision Systems Engineering**

**Arizona State University, Tempe**

**Term Paper on Virtual Vending Machine Modeling using the Eclipse Modeling Framework**

**Spring 2015**

**Paper By:**

**Kranthi Rekala – ASU ID: 1207663258**

**Anurag Agarwal – ASU ID: 1207687399**

1. **Abstract:**

*EMF is a framework that can take a model (as described in a .ecore file) and generate the supporting java files that can be used with EMF to build, load, manipulate and save instances of that model.It means you use EMF when you are writing a java application and want to manipulate a model (an abstract syntax tree for a language, a model that can be displayed by your View, a domain that needs to be manipulated, etc).*

*There are a lot of modeling projects, and they add support for more and more functionality but most of it is based on them. It Supports the definition of meta models using a dedicated metamodeling language (Ecore).It Supports various functionalities such as built in xmi serialization and deserialization, validation etc.*

**Introduction:**

**What is EMF?**

The EMF is a modeling framework that is based on the “Eclipse” IDE (*Integrated Development Environment*). It provides the facility for generating codes that are used for building tools and applications under the structured data-model. These tools & runtime supports are provided by EMF for producing sets of **Java Classes** to your models, **Adapter Classes** to view and edit these models via command base, and for producing **Basic Editors.**

Using annotated Java, XML, UML documents, or the modeling tools your models can be created and imported into the EMF. And the important factor of EMF is that it provides a foundation for inter-operation between different EMF based tools or applications.

**Why should we Model?**

**Modeling** There are a lot of advantages in using modeling; of which, producing high quality results at quicker pace is a key factor, then its tried & tested solutions can be reused for better efficiency and effectiveness overall. Making complex and structured information is concise and simplified, while creating rich textual/graphical remarks are easy as well.

Also, implementation of runtime solutions is to the point, and making use of its inter-operation functionality to exploit the industry’s standards is rewarding.

**What are the Fundamental pieces of Modeling in EMF?**

**EMF** There are two meta-model upon which EMF is based on, they are Ecore and Genmodel models. Ecore framework includes the meta-model describing the models, and runtime support for these models, which includes changing of notifications, persistence in support using the default XMI serialization and an effective reflective API that’s manipulating the EMF objectives. The Genmodel on the other hand contains information that is needed in addition to the core content for code-generation; such as path information, file information, etc.

**EMF.Edit**

This EMF.Edit framework comprises of reusable generic classes that are used for building the editors of the EMF models. They Provide the Following:

* Content and the label providing classes, support for property sources, and related classes that conveniently support EMF models in order to be displayed under the standard desktop viewers (JFace) and the property sheets.
  + The command framework that includes generic commands implementing classes that are used for building the editors that ultimately support the fully automated undoes and redoes.

**EMF.Codegen**

The facility offered by EMF code generator makes it possible for generating everything related to the building of a complete EMF model editor. This includes the GUI (Graphical User Interface) with which the options for generators can be specified and invoked. The Java Development Tooling (JDT) components of the Eclipse are given leverage by this generation facility.

**Code Generation levels in EMF:**

**There are three levels of Code Generation supported by EMF, They are as Follows:**

* **Model** - This provides the Java interfaces & the implementation classes that are available to all the classes in your model, additionally the factory & package implementation class, also known as the meta-data.
* **Adapters** - This generates the implementation classes, which are called the Item-Providers, adapt to the model class requirements and the editing/display.
* **Editor** - The EMF produces a clearly structured editor which matches the style recommendation of Eclipse EMF editor serving as the starting point from where you will start the customization
* Test level plugin was generated during our execution of the genmodel through the generateall() method

All of these generators support code regeneration while at the same time keeping user modifications unchanged. These generators can be accessed and activated either by using the User Interface or by creating a headless command line.

1. **Contribution Table:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Module** | **%Cont.**  **(Kranthi)** | **%Cont.**  **(Anurag)** | **Team**  **Evaluation** |
| **VM**  **Ecore**  **Model** | **60** | **40** | **50-50** |
| **Code**  **Generation** | **60** | **40** | **50-50** |
| **Implementations**  **To The VM**  **Generated**  **Code** | **60** | **40** | **50-50** |
|  |  |  |  |

1. **Ecore Model and Development Steps:**

Building the Ecore model entailed studying the code given by the professor and subsequently proceeding with building an Ecore entity. The Eclipse modeling framework was downloaded for this purpose as an add-on. The basic elements used during the course of constructing an Ecore model included:

**EPackage:**

The top-level object of an Ecore file

In Ecore terminology metamodel is equal to EPackage

•Semantically similar to a Java package

•Each EPackage has a name, a namespace URI and a prefix

–nsURI (Namespace URI): Global EPackage identifier

•Ideally, each EPackage in the world should have a unique nsURI

–Prefix: No semantics. Only used for XMI serialisation

•An EPackage can contain

EClasses and EDataTypes (both inherit from EClassifier

**EClasses:**

•Semantically similar to a Java class

•Can be instantiated to produce model elements in the same way that Java classes can be instantiated to produce objects except if its **abstract** or **interface** properties are set to true (similar to Java)

•Can inherit from multiple EClasses

Unlike Java classes

•Can contain

- (E) Structural Features (EAttributes and EReferences)

–EOperations

**EDatatype:**

Represent primitive types

•Ecore provides built-in support for some EDataTypes (EString, EInteger, EBoolean, EDouble)

•Each EDataType needs to define the fully-qualified name of the Java class that can convert strings to instances of the data type and vice-versa

**EOperations:**

•Semantically equivalent to a Java method

•Has a name, typed parameters and a return type

–The multiplicity of an EOperation (Lowe Bound, upper Bound inherited from ETypedElement) affect its return type

•For example if the multiplicity of an operation is 0...\* and its return type is EString, it means that the operation will return a list of Estrings

•Ecore can only define the signatures of operations not their logic!

**EAnnotations**

These are syntactical metadata that can be added to the source code where deem necessary with classes, methods, variables and parameters. Ecore annotations “can be [reflective](http://en.wikipedia.org/wiki/Reflection_(computer_science)) in that they can be embedded in [class files](http://en.wikipedia.org/wiki/Class_(file_format)) generated by the compiler and may be retained by the Java VM to be made retrievable at [run-time](http://en.wikipedia.org/wiki/Run_time_(program_lifecycle_phase))”

**EStructuralFeature:**

•EAttributes and EReferences are collectively known as EStructuralFeatures

–EAttributes: Provide slots for values of primitive types (EDataTypes, EEnum)

–EReferences: Provide slots for model elements

–They only denote structure, in contrast to EOperations which denote behaviour

•Each EStructuralFeature has

–a name

–a type (eType)

–a lower-bound and an upper-bound multiplicity e.g.

•0..-1 (-1 means unbounded/\*)

•2..5

•0..1

•EStructuralFeatures can be marked as

–derived: the value of the feature is computed and cannot be changed directly by the user

–volatile: the value of the feature is not persisted when the model is saved

–transient: the value of the feature is recomputed every time it is requested

•Containment

–If a model element is deleted from the model, any elements under its containment EReferences are also deleted

–Each model element can belong to **at most one** containment EReference slot

•eOpposite

–Enables 2-way synchronisation between EReferences

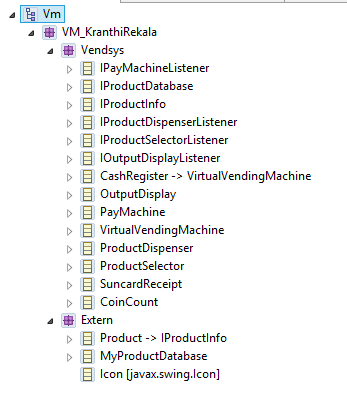
–Opposites need to be specified on both participating EReferences

–The opposite of the opposite of an EReference must be the EReference itself

Ecore Design Process:

The process of designing Ecore had the Following Steps:

1. Identifying the packages and the classes, interfaces under each of them
2. Identifying the relationships, multiplicities and their associated roles for each relationship and their types.
3. Opening the aird perspective for modeling in Eclipse and drawing each of the attributes, setting return types and defining their respective interfaces and Listeners.
4. After this step a genmodel is defined to drive the code generation.
5. The ecore is loaded into genmodel and then the code is generated using GenerateAll to generate Editor, Test, and Edit Codes.
6. Using Editor the Runtime IDE is generated using Run as “Eclipse Application” and then it opens another Eclipse Instance where a Model can be defined to interact with the Code and validate the constraints.
7. Validator is setup for the insertDime() using the Evalidator which would alert in case the total sum of Money is less than 3 dollars.
8. Other Methods which were generated are provided implementation wherever appropriate.
9. GUI methods and classes are removed from the ecore as EMF is not good in desingning UI. The provided UI code has been used for this purpose.
10. **VM\_KranthiRekala.Ecore**
11. **extern.Ecore**



**Constraint Inclusion in the insertDime method:**

This was achieved by editing the ecore file and providing the Evalidator function called hasMaxDimes. The value for the variable m\_dMaximumBalance of PayMachine Class has been set to 3.00 and in case the total deposited value reaches 3.00 dollars, an error is generated.

The Following is the Code for the same.

**public** **boolean** validatePayMachine\_hasCrossedValue(PayMachine payMachine, DiagnosticChain diagnostics, Map<Object, Object> context) {

// **TODO** implement the constraint

// -> specify the condition that violates the constraint

// -> verify the diagnostic details, including severity, code, and message

// Ensure that you remove @generated or mark it @generated NOT

**double** deposit=payMachine.getM\_dBalance();

**double** max\_deposit = payMachine.getM\_dMaximumBalance()-0.10;

**if** (!(deposit<max\_deposit) && (payMachine.isM\_bOutOfOrder()==**false**)) {

**if** (diagnostics != **null**) {

diagnostics.add

(createDiagnostic

(Diagnostic.***ERROR***,

***DIAGNOSTIC\_SOURCE***,

0,

"\_UI\_GenericConstraint\_diagnostic",

**new** Object[] { "hasCrossedValue", *getObjectLabel*(payMachine, context) },

**new** Object[] { payMachine },

context));

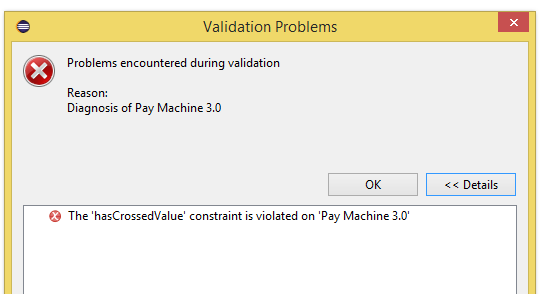
}

**return** **false**;

}

**return** **true**;

}



**Comparison of Code:**

**The Following are some of the inferences from the Code that was provided to us and the Code Generated by EMF.**

To maximize the code generated by the Model and to ensure maximum code is covered by the Model, interfaces which are part of code are also designed as part of Ecore.

Java code will be added to the project containing your Ecore metamodel

–Java interfaces and implementations for each EClass in your metamodel

•3 new projects

–**.editor**: A dedicated tree-based editor for your metamodel

–**.edit**: Controls the behavior of the Properties view

•You will rarely need to worry about this one

–**.tests**: JUnit tests for testing

* For each class in the model, a Java interface and corresponding implementation class will be generated
* All the Setter and getter methods for the Attributes defined are generated with @generated Tags. These are to be overridden using the NOT to provide our Implementation.
* Regeneration of Code post modifications is easy using the .editor file. It would not regenerate the code that is having **@generated NOT** before it and so it would not disturb the already implemented methods.
* Reusability can be achieved as this model can be used to generate any similar projects by modifying the ecore.
* Metamodel defined can directly interact with the generated code and hence can validate constraints which is not achieved by used normal Modeling Tools like UML etc. Hence it exerts a more control over the code.
* Our metamodel contains an operation and a derived reference
* Node.isStart() operation
* Node.nodes derived reference
* •By default, the code generated by EMF for these, throws an UnsupportedOperationException
* –i.e. we need to implement the logic of these methods

**Advantages of EMF:**

* Easy Code generation for huge Software
* Reusability of Code wherever required
* Less Bugs as the code generated is by the Machine and eliminates normal Human made errors
* Multiple times code can be generated by simply editing model which takes care of editing at appropriate places and replacing appropriate variables.
* Appropriate setting of visibility for data types automatically.
* Factory is used to create instances of model classes and while the package provides static constants
* Observing EMF methods are used that provide notifications when an attribute or reference are changed
* In Built Exception Handling Techniques.

**Conclusion**

A fresh approach that focuses on great level design and rerouting of menial tasks to the tools and frameworks is what concisely describes the **Eclipse Modeling Framework**. It ultimately offers smarter, faster and better solutions to your modern requirements of a model.

Source Code:

The Source code for the implementation has been submitted to blackboard.

**References**:

1. <http://www.eclipse.org/modelingemf/>
2. <https://wikipedia.org/>
3. <http://vogella.com>
4. <http://help.eclipse.org/luna/>
5. <http://www.slideshare.net/>
6. [www.eclipsesource.com](http://www.eclipsesource.com)
7. EMF Text Book
8. [www.youtube.com](http://www.youtube.com)
9. <http://refcardz.dzone.com/refcardz/essential-emf>
10. http://www.philmann-dark.de/EMFDocs/tutorial.html