UNIVERSITY OF WATERLOO
Software Engineering

SE 499 Report:
Feature Modeling and Configuration of Ecore Elements within Eclipse

Prepared By
Matthew Stephan
Student ID: 20098161
Userid: mdstepha
4B Software Engineering
May 1, 2007

Report Summary

The following report summarizes the author's work during their SE 499 project term. It begins by providing an introduction to the project the student engaged in during their previous work term, specifically, the amalgamation of Ecore modelling in Eclipse with feature modelling and feature configuration. It then alludes to the fact that there were many improvements and issues outstanding with the project at the conclusion of the student's work term naturally leading into the author's participation in the SE 499 project the subsequent term.

Background information is provided that gives readers enough information on Ecore modelling, feature modelling and configuration, and the state of the project after the author's initial work term such that the reader will be able to fully appreciate the problems presented and the solutions that were devised and implemented to deal with them. A number of problems are presented during the discussion of the state of the project after the initial iteration. Most notably is the fact that there was a tight-coupling between the graphical components and logic related to displaying feature models. There is also discussion of an indirect recursion problem and incomplete implementation of some of the feature configuration options.

The solution discussion begins with the most critical change that occurred. The feature modelling component was extracted from the graphical components and was transformed into an Ecore model such that automatically generated code could be utilized. This resolved a great number of the issues and also facilitated code

generalization, a task that is also described in the report. The indirect recursion problem was fixed as a side effect of the feature Ecore model-generated code due to the ability of an Ecore object to be aware of its container/parent. Lastly, the progress made in regards to feature configuration is discussed as well any significant and outstanding issues.

The report then concludes, noting that the newly autonomous feature modelling component makes the system much more reusability and less cluttered than it was after the initial iteration. It is also noted that the recursion problem no longer exists and that feature configuration is complete barring a few exceptions.

Recommendations are made in regards to investigating and possibly improving the method in which feature code generalization is achieved and to ensure completion of the feature configuration functionality to ensure that Ecore FMP is a desirable and useful project for many people in varying domains.

Table of Contents

Table of Figures

1. Introduction

The combination of feature modelling and configuration is an extremely useful way of viewing and controlling certain features or properties of a system. It is also an important component in many generative software development methodologies in that it can be used to instantiate a system/application based on a model/specification of a system [1]. As noted in [2], a feature is any property of a system/component that is important in specifying the system. In the context of this report, feature modelling refers to viewing features in a hierarchal fashion [2], thus indicating features/properties and relationships they can have, somewhat like a template for an instance of a component. Feature configuration, on the other hand, refers to creation of an instance by utilizing a feature model by means of choosing what features and relationships will be upheld in the instance.

While feature modelling is an extremely useful tool for describing the defining properties of a system, Ecore is useful for facilitating the analysis and design of object-oriented systems. Ecore is a metamodel for object-oriented programs and is part of the Eclipse Modeling Framework (EMF) for use within the Eclipse platform [3]. It can be used to model object-oriented programs, notably, packages, classes, attributes, references, et cetera, and then facilitate dynamic instantiation and Java code generation of those programs.

During the past work term, fall 2006, a project was initiated by the author and his supervisor that provides feature modelling and configuration to Ecore models within

Eclipse through the Eclipse plugin mechanism. While the project was at a stable state at the end of the term, there were a number of possible improvements and issues that existed in the created plugin. As such, the student and the supervisor decided to continue this work through the SE 499 project course offered at the University. The following report describes the work accomplished by the author for his SE 499 project. Enough background information is provided on both feature modelling, feature configuration, and Ecore such that any reader familiar in software modelling and Eclipse should be comfortable with the material discussed. The report begins by presenting this background information and then continues by defining the problem, specifically the state of the project at the end of the work term/first iteration and the outstanding issues surrounding it. It then outlines the solutions employed to deal with these issues, draws conclusions about the work, and makes recommendations about the future of the project.

2. Background Information

The following section provides a brief explanation on both the EMF Ecore metamodel and feature modelling and configuration to facilitate understanding of the two technologies for any reader who may not be familiar with them. It then concludes by summarizing the state of the Ecore Feature Modelling project after the initial iteration and the issues about it that needed to be addressed during the SE 499 project course.

2.1. Ecore Metamodel within Eclipse

The Eclipse development environment facilitates object-oriented specification by means of the Ecore metamodel. By saying Ecore is a metamodel refers to the fact that Ecore is a model of model, that is, it is a model of an object-oriented specification model. Eclipse allows creation and modification of an Ecore metamodel through the sample Ecore model editor. As shown in Figure 1, the Ecore model editor allows browsing through an Ecore model through the hierarchy. Packages are at the top level, classes the next, and references, attributes, and annotations follow. At any level, siblings or children can be created if the current model allows for it. In this particular instance, the model being shown in the editor is the Ecore model itself, which is the basis for all Ecore model instances.

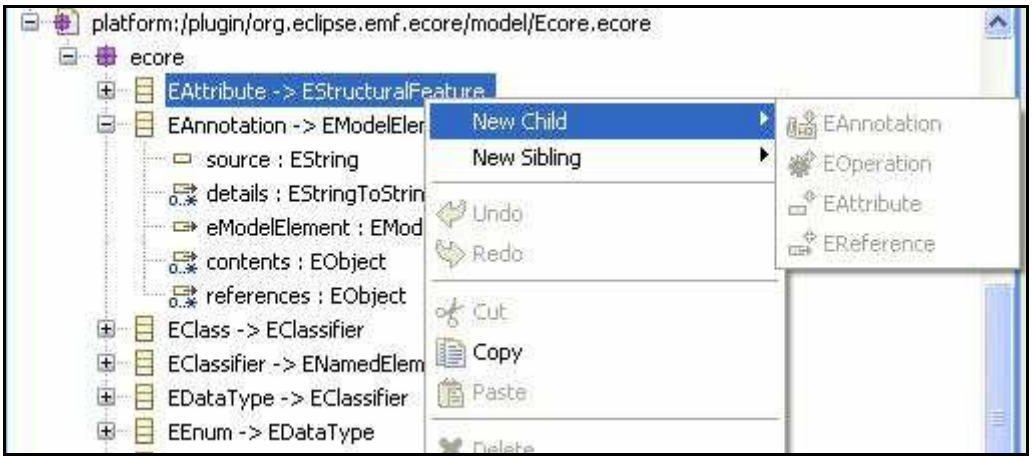

Figure 1: Example Ecore Model Editor

2.2. Feature Modelling and Configuration

As noted in [2], a feature model is a model that organizes features based on their structure within the domain being modeled. It is useful in that it provides a different way of seeing systems, specifically in terms of the features that make it unique. Furthermore, feature models can make use of cardinalities, as discussed in [4], which provide more information and power in modelling because it facilitates the addition of references, attributes, and feature cardinalities to feature models. Figure 2, taken from [2], shows a feature model in Eclipse as allowed by a feature model plugin previously created in [2]. The example shown is that of an electronic shop and also exhibits the cardinalities and the affordances they provide. Each of the symbols, such as mandatory and root features, have different meanings and are explained in [2]. This report will explain the symbols only when necessary if and when they arise throughout the document.

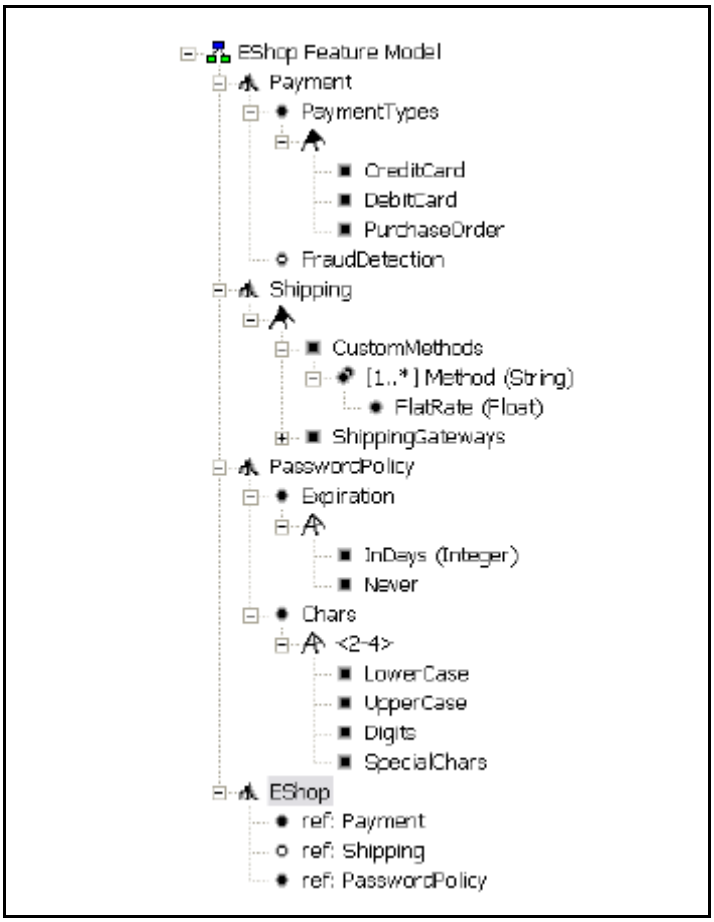

Figure 2: A sample Feature Model. Taken from [2]

Feature configuration refers to instantiating a feature model by selecting which features and relationships will be upheld in the specific instance being configured. So, continuing from the example above, an actual EShop would pick the features applicable to them, assuming they are optional, in order to configure its specific instance. Feature configuration also allow for feature cloning/replicating and attribute selection [2]. Once a configuration is complete, the idea is that an application or subset of an application can be generated automatically using knowledge and inference about the domain based on the features and options configured. It is similar in structure and appearance to that of a feature model, however, it has check boxes for optional features and attributes, shows

only non-abstract classes, and shows all inherited references and attributes. The values of a specific instance are shown and are also editable within that configuration view.

2.3. First Iteration of the Project

During the previous academic term, fall 2006, the supervisor of the author of this report and one of the supervisor's graduate students came up with the idea of amalgamating the concept of feature modelling and configuration with that of Ecore modelling within Eclipse. Thus, the Ecore Feature Modelling Plugin (Ecore FMP) project was born. The aim of the project is to have Ecore models created, viewed, and managed through a feature modelling view. Furthermore, configuration of an instance of feature model can be achieved through the feature configuration editor/view. The work was assigned to the author of this report, who was doing a cooperative work term under the supervisor for that term. At the conclusion of the work term, the project was at a stable state. Specifically, feature modelling was complete and feature configuration was well on its way. While it was stable, there were a number of serious improvements to the project that needed to be made. As such, the author of this report and their supervisor began the SE 499 project course with the intention of making these improvements and furthering the project. This section outlines the various problems that existed with the project after the work term that needed to be addressed during the SE 499 term.

2.3.1. Tight Coupling with Graphical Component

In order to create a tree-like viewer or editor within Eclipse, one must implement the interfaces of both the ITreeItemContentProvider and IItemLabelProvider interfaces for each of the entry types within the tree. That is, each Java object type that is to be

represented in the tree structure must have an ItemProvider that implements both of these interfaces. Each time the tree is created/drawn, an adapter is called that maps each object to a specific ItemProvider. The first of the required interfaces, ITreeItemContentProvider, allows an ItemProvider to retrieve its children and its parent, which is necessary because a tree entry must be able to contain and display zero or more children. The latter of these interfaces, the IItemLabelProvider, gives the ItemProvider the ability to retrieve the text and image that should be displayed for a specific entry within the tree structure. As such, the three main methods that an implementer should be concerned with are the ones that retrieve the children of the object, retrieve the text, and retrieve the image to display in the tree for the object.

ItemProviders should be seen as purely graphical components such that they should only be concerned with how to display an object for a given context. The first iteration of the project did not abide by this. Two different FeatureItemProviders were created for both modelling and configuration that were meant to deal with features in the two different domains. Each of these contained a significant amount of logic that pertained to determining the properties of the features, such as their mapping to Ecore and the calculation of their children. Not only is this a problem because of the tangling of feature logic and display code, but there was a significant amount of duplication between the two FeatureItemProviders.

2.3.2. No Ability to Deal with Features Alone

A somewhat implicit side effect of the issue described in the previous section is that features are not autonomous. That is, if one wanted to utilize the feature logic and

deal with features that map to Ecore without being tied to ItemProviders, it would not be possible using the first iteration of the project. This issue became apparent during the first iteration work term when a fellow member of the author's group was interested in manipulating Ecore components as features without utilizing the GUI. In order to deal with features in the way they needed, the group member was forced to use the ItemProviders method that retrieved children and had to extend it to get more functionality. This is clearly incorrect because the intent of the ItemProviders' children retrieval method is for children in the tree structure, not necessarily the children of a feature. It became quite clear that the logic related to features needed to be extracted and localized as a separate part of the project.

2.3.3. Recursion Deeper than One Level (Indirect Recursion)

A relatively important problem with the first iteration was its inability to deal with Ecore models that had indirect recursion, that is, recursion that occurs deeper than one reference level. Figure 3 shows an example of both direct and indirect recursion. The Ecore class "R" has a direct containment reference of type R, which would yield an infinite recursion from a feature perspective because R would contain R, which would contain R, et cetera. The indirect recursion is represented in the remaining Ecore classes. The class AContainer has a reference to A, which has a reference to B, which has a reference back to A. From a feature hierarchy perspective, this is infinite recursion.

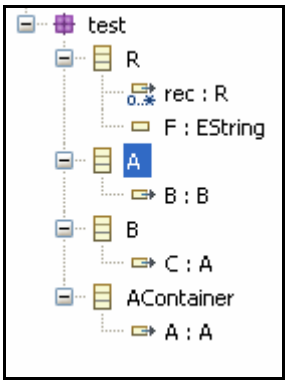

Figure 3: Example of Simple and Complex Recursion

In the first iteration of the project, the feature modelling and feature configuration views could both support the direct form of recursion by having an Ecore class check if any of its immediate references are of the same type or subtype of itself. The indirect recursion support was not in place for either modelling or configuration and caused an error in the program because the children retrieval occurred infinitely. This error was non-trivial as it seemed to occur in many examples derived by the author and the group members.

2.3.4. Feature Configuration Incomplete

As noted earlier, the feature configuration view/editor needs to be able to allow a user to configure an instance of a feature model by selecting, clicking, cloning, and editing an instance of that model. It needs to show all the same elements of the feature that are applicable to configuration, for example, only non-abstract classes and inherited features. The viewing aspect of feature configuration was completed after the first iteration, but only a subset of the interaction part was finished. Specifically, only editing of an attribute was completed at the end of the first iteration.

3. Upgrading Ecore FMP

Given all the shortcomings from the first iteration of the project, it seems logical to have the SE 499 project focus around improving the Ecore FMP system and dealing with these issues. This section describes the various solutions that deal with the problems presented in the past section and other general improvements to the system that were made.

3.1. Feature Metamodel

The most basic and prevalent problem from the first iteration was the entanglement of feature code within the ItemProviders as discussed early. In order to use ItemProviders the correct way, the solution chosen is to refactor by way of modelling the feature concept in Ecore and then using the automatically generated code. Figure 4 shows the Ecore model of the feature concept underneath the feature model (fm) package. It begins with an abstract class, Node, which both Feature and FeatureGroup inherit from. A Feature is a feature in the traditional sense as discussed in this report so far. A FeatureGroup, on the other hand, is a derived or annotated grouping of features under a class such that it contains a grouping of them abiding by some cardinality constraints. The only difference, with respect to the model, between a Feature class and a FeatureGroup class, is the representation enumeration, which indicates the type of feature or feature group it is.

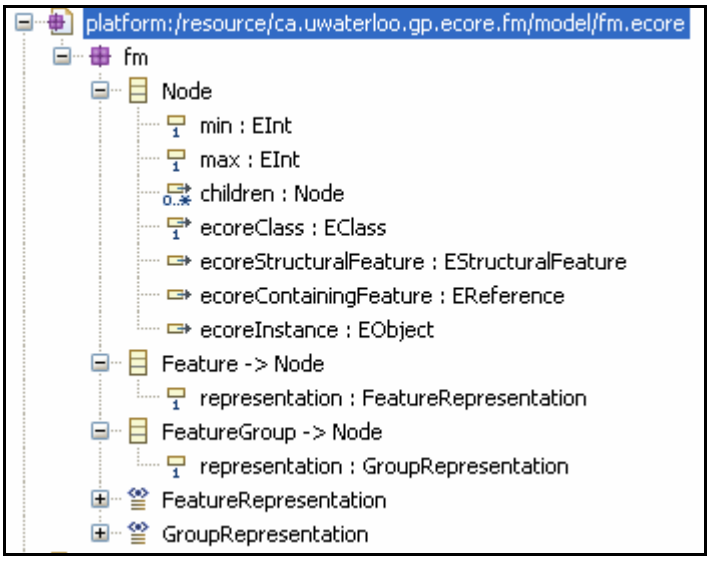

Figure 4: Feature Ecore Model

The Node abstract class contains all the information required to perform the mapping from feature model to Ecore as accomplished in the first iteration. The structural feature is used in the case the feature or group is representing an attribute or reference. The Instance variable is used only in the case where an actual instance of the feature exists, for example, during feature configuration, instances are created and thus this attribute would take on the value of that instance. The children reference contains all Nodes that are logical children of the Node. Min and max refer to cardinality constraints and the containing feature is the Ecore reference that contains the node. Lastly, the class attribute refers to the Class that the feature represents in the case of a class feature. In the case of an attribute or reference it represents the containing class. A more detailed and technical version of the mapping between Ecore and the feature Ecore model will be provided in a technical report to be done by the author of this report in the near future.

After generating the code, it became necessary to modify the ItemProviders to utilize these new Node objects. A FeatureItemProvider and FeatureGroupItemProvider were created in both the feature modelling view and feature configuration view contexts. Both of these contain relatively zero feature logic and only make calls to retrieve attributes of the Node class in question in order to take care of the main three methods; retrieving children, retrieving text to display on the tree, and retrieving image to display on the tree. The modelling and configuration versions of the ItemProviders differ in the way they use these 3 methods to provide content to the tree.

This is ideal because, as shown in Figure 5, there is now only a one-way dependence between the graphical ItemProviders and the features being modelled or configured. The feature modelling component/package has no dependencies to any other part of the system, as such, it can be used autonomously. This includes users who want to have the use of features and feature groups that correspond to the definition in the feature model, but not necessarily use the ItemProviders or any graphical interface at all.

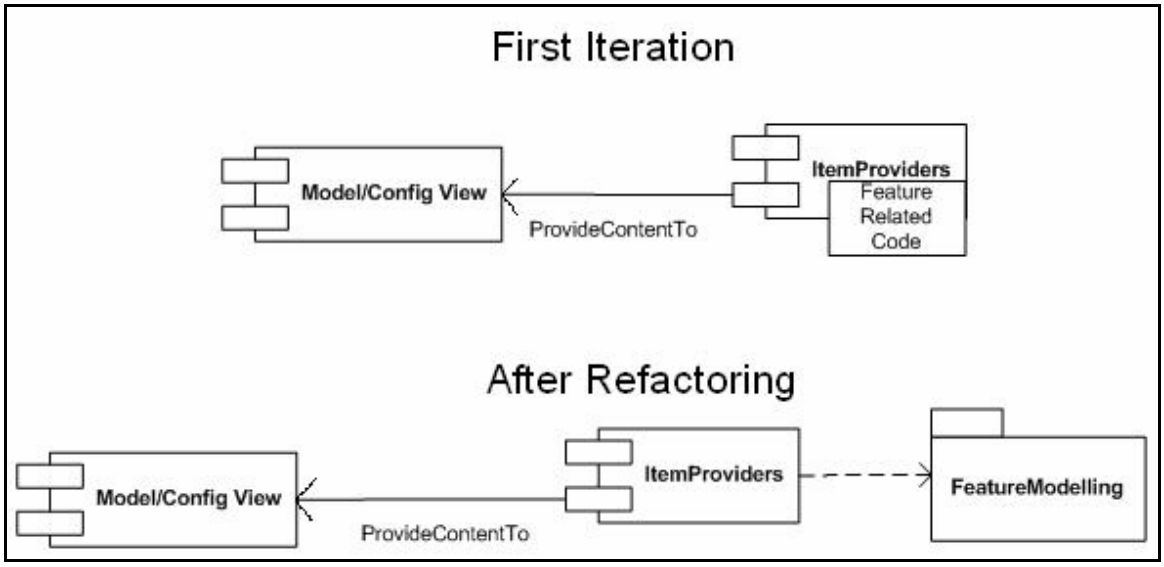

Figure 5: Comparison of the Feature Logic Location

3.2. Feature Code Generalization

In order to provide a single code base for features that would support the requirements of both feature modelling and feature configuration, it was necessary to generalize the feature logic code. The main example of this is the children retrieval method of a feature. The concept of children/sub features in a hierarchy differs depending on the context one is interested in. There are two main differences in feature modelling and feature configuration regarding the notion of children; abstract classes and inherited features. The following two sections outline how these differences were dealt with in order to make the code more general.

3.2.1. Abstract Classes

The problem of including abstract classes is solved by allowing the user of the update/get children method to determine if they would like to include abstract classes or not. This flag is given to the feature and is used when retrieving the metadata (non-instance) children. Figure 6 shows the code snippet that is called to facilitate the generalization of allowing abstract classes or not. Every time a call is made to add either a class or a subclass of a reference to the children list, there is a check to see if the class being added to the children list is abstract or if abstract classes are being included in the addition. This ensures that abstract classes are only shown when the flag indicates so.

```
if (includeAbstractClasses || !classOfreference.isAbstract()){
    //children.add(...,includeSuperStructuralFeatures,includeAbstractClasses));
}
```

Figure 6: Logic for Generalizing Abstract Classes

3.2.2. Inherited Features

Inherited features refer to features that are inherited by a subclass from a super class. So, if a super class has an attribute, then a sub class should also have the same attribute. The same applies for references. In the first iteration of the project, this was accomplished by iterating different lists; one that includes the inherited features and one that does not. In the newer version, this basic idea remains, however it is accomplished slightly differently. In order to achieve this cleanly, the project uses the "for each" facility provided by Java 5.0 that allows a user to iterate through a given list and get the value placed in an object pointer. This is used in conjunction with a flag provided by the feature method caller that indicates if super/inherited structural features should be included. As exhibited in Figure 7, the list that is chosen is dependent on the flag that indicates if inherited features should be included. If the flag is true, then the list returned is one that contains all inherited structural features, otherwise, only the local structural features are returned. The "for each" loop then iterates through that list, proceeding identically from that point on irrespective of which list is used.

```
EList structuralFeatures = includeSuperStructuralFeatures? this.getEcoreClass().getEAllStructuralFeatures()
                                                         : this.getEcoreClass().getEStructuralFeatures();
for (Object aux : structuralFeatures) {
    //...
}
```

Figure 7: Inherited Features Generalization Code

3.2.3. Possible Alternatives

While flags are sometimes considered bad practice, they seemed to be appropriate in this instance, providing the user of the feature component some ability to control what children will be returned. A possible alternative, however, that may be superior is to use

something like the strategy pattern or an analogous design pattern that allows for different ways of accomplishing a similar goal. The author of this report felt as though the overhead in classes and calls inherent in the strategy pattern was not worth the trade off in good design. Given more time, however, a deeper analysis should be performed.

3.3. Identifying Direct and Indirect Recursion

Now that the system is utilizing the generated feature Ecore model code, dealing with both types of recursion is much less troublesome and borderline trivial. By using the generated code, all instances of Nodes, which are Ecore Objects or "EObjects" because they are from an Ecore model, have access to their container through the retrieval method eContainer() that can be called on any EObject. Figure 8 contains the correct feature model of the Ecore model presented in Figure 3. Because features can know their containers, this diagram/feature model is possible.

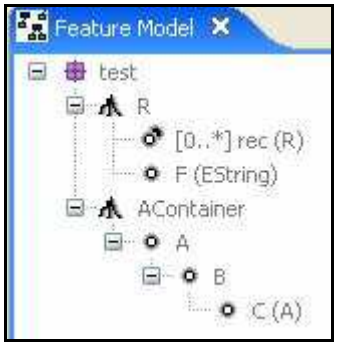

Figure 8: Feature Model of Recursive Examples

Both indirect and direct recursion is checked for and dealt with in the same manner. It is checked for by continually traversing a feature container and comparing it with the current reference type until recursion is discovered or the top level has been hit. In the direct recursion case, as shown in feature R in the figure, it checks the very first

reference type against the first container, that being R. Recursion is discovered and the chain is broken. In the indirect case, as shown in feature AContainer in the figure, the reference C of type A that B has is first checked against B. A is not equivalent to B, so this search continues by checking reference type A and all its subtypes against B's container A. Here, recursion has been detected and the chain is broken. This traversal through containers can continue indefinitely until the highest level has been reached.

3.4. Advancement in Feature Configuration

At the conclusion of the first iteration there were a number of outstanding issues remaining in the feature configuration aspect. After utilizing the feature Ecore model, the correctness of the metadata (non-instance) feature configuration view aspect remained intact. So, only the non-metadata configuration component remained. The two main components of this aspect are the instance viewing and the instance modification. The following section will indicate the work that was done for both.

3.4.1. Instance Viewing

The main idea behind instance viewing is that the configuration view needs to be aware of the instances of features that have already been instantiated by the user. The feature model accommodates instances through the Ecore instance attribute found in the Node abstract class. Regarding the actual coding that is required to accomplish this, the amount of additions is quite significant. Every time a feature has its children retrieved, an instance must be checked for. This will work in both feature modelling and feature configuration because there is no way of an instance being set in the case of feature modelling. If an instance is found, then it needs to be displayed as either a checkbox if it

is optional or a black dot if it is a mandatory feature or clone, as was done in the previous feature plugin from [2]. There must also be a check of cardinality when an instance is present because if there are a number of instances to a reference equal to the maximum amount allowed, then no modelling/metadata should be displayed. This functionality is now present in the Ecore FMP project with very few issues left to tackle. Some of them include displaying feature group instances, refreshing instances that have been added recently, and others. All of which can be tackled given time.

3.4.2. Instance Configuration

All of the instance configuration methods are quite similar. Selection, cloning, and removal all require analogous modification of the feature. Before modification is allowed however, the cardinalities of the references or attributes must be checked against the number of instances. If the number of instances is equal to the max, then there will be no option to select or clone. If the number of instances is equal to the min, then there will be no option to remove.

Every time an instance is created, through selection or cloning for example, the containing reference of that instance must be accessed through the pre-existing instance of the container. In other words, if feature A has a reference, C, to feature B and the user selects to clone or select B under A the following must transpire: C must be accessed through reference C's container, namely A. Once A is acquired and the reference C is retrieved from the actual instance of A, a new instance of B must be created and added to C's reference list in the context of instance A.

For an instance to removed, the same principle applies in acquiring the appropriate reference. The only difference here is that removal of an instance must remove recursively all features that are contained by the feature or somewhere underneath its hierarchy. This is because those feature instances have no context outside that of the feature if the containment property is true.

One problem that has arisen that has not been significantly dealt with is that of creating new instances of objects or attributes that are not primitive. Primitive objects and attributes have a default value that can be put effect for all new instances. Many non-primitive objects and attributes, however, do not have any default value or settings. This is typically for non-primitive objects and attributes that do not have an empty/default constructor such as the Integer class. For the time being, the make-shift solution is to provide a string containing the character '0'. This works for the majority of classes that take a single string parameter, but it is clearly not the correct solution. One possible solution is some kind of switch statement or analogous design pattern that acts on all of the Ecore types that are non-primitive. Then a default value will be chosen appropriately. Another solution is to create a very simple wizard that pops up when a new instance of a non-primitive object or attribute is created. This will then have a field for each parameter of the base constructor that the user can fill in order to configure the newly created/cloned instance of the non-primitive object or attribute.

4. Conclusions

It comes as no surprise that separating out the feature domain logic from the graphical/display logic is extremely beneficial. Having the feature logic as its own autonomous component means that code reuse is increased and code duplication is decreased. Users of the Ecore FMP project who want to deal solely with the logic surrounding features and feature groups are able to do so. The ItemProviders are now fulfilling their appropriate roles and code throughout the project is cleaner and more concise.

The recursion problem from the first iteration of not being able to handle indirect recursion is dealt with as a pleasant side effect of having the feature and feature group logic modeled as an Ecore model. The ability for each feature to detect its container allows for easy parent traversal, thus reducing the complexity of detecting recursion.

Much progress is made from the first iteration to the current version in terms of feature configuration. Instances now display correctly, with a few minor exceptions. Configuration through addition and removal of instances is at a much better state then it was after the first iteration. There is still a large issue remaining however in the instantiation of non-primitive objects and attributes.

5. Recommendations

The first recommendation is to have the author of the report or someone familiar with the work investigate the alternative solutions put forth in the feature code generalization section. While the flags do work quite nicely and the author believes at this time that the trade off between using them or a design pattern not warrant it, more time may reveal the opposite to be true.

The next recommendation is that the instance viewing component of feature configuration be completed. There are a number of minor but somewhat difficult cases remaining and it is important that all of them are covered for the plugin to be a quality product.

The last recommendation is to determine which solution to the new instance of non-primitive object or attribute problem is the best and to implement that solution. This will arise frequently while someone is using the feature configuration editor/viewer so this task is critical and should definitely be attended if the Ecore FMP project is to move on.

6. References

[1] K. Czarnecki and U. W. Eisenecker, *Generative Programming: Methods, Tools, and Applications,* Addison-Wesley, 2000.

[2] Michał Antkiewicz and Krzysztof Czarnecki, "FeaturePlugin: Feature Modeling Plug-In for Eclipse" in *OOPSLA'04 Eclipse Technology eXchange (ETX) Workshop*, 2004.

[3] The Eclipse Foundation, "Eclipse Modelling Framework (EMF), 2007, http://www.eclipse.org/modeling/emf/?project=emf (last accessed March 26, 2007)

[4] K. Czarnecki, S. Helsen, and U. Eisenecker, "Formalizing cardinality-based feature models and their specialization" in *Software Process Improvement and Practice*, 10(1), pp. 7–29, January/March 2005.