Vision Spec

1

ANDROMDA.ORG

AndroMDA 4.0 Vision Document

AndroMDA – the next generation

ANDROMDA 4.0 VISION DOCUMENT

Moving to the Agile World

AndroMDA.org
Matthias Bohlen
<matthias@andromda.org>

Contents

Chapter 1

Preface

Making a powerful generator more usable…

AndroMDA has made significant progress since it started. Today, it can generate code for several different technologies and can be customized to generate project-specific or domain-specific code. This makes it already a quite powerful tool for advanced, disciplined users with a certain amount of thought up-front. Advanced users? Thought up-front? Why? Shouldn't newbies or agile users get the same advantages from AndroMDA as the experts?

Well, have a look at the development process with AndroMDA, and you will know what I mean. It is quite a bit clumsy and although I have always focused on simplicity, the user has to pay much attention to what he does and has to go forward in consistent, pre-planned steps. Otherwise the amount of rework will create friction and loss of working speed.

This paper introduces you to the problem and proposes necessary changes to the architecture. The ultimate goal is to make AndroMDA more visual, more easy to use, and make it possible to apply it in an agile project.

Agility – a simple example

Agile processes (XP, Scrum, etc.) state that you should do the simplest thing that satisfies the requirements, first. Later on, when the number of requirements increases and each requirement becomes more precise, you should refactor and extend the simple solution. Experts in the field like Kent Beck and Martin Fowler have shown brilliantly how to do this in conventional, text-based programming languages.

Imagine, you want to do the same thing (refactoring) in a UML model and re-generate the code using AndroMDA. For sake of a simple example, let's assume you only want to change the name of a class and re-generate. What would happen? Think for a moment before reading on…

Rename a class and run the generator – what happens?

After AndroMDA has generated text files (e.g. Java code) from a UML class, it forgets about them immediately. So, when you change the name of a UML class, AndroMDA thinks: "Hey, there is a new class" and happily generates text files for it. The old files still exist; the business logic in the new classes is empty. You've got two software components instead of one.

Is this the result the user would have expected? Highly unlikely!

The user's expectation about refactoring

A user who works in a project using an agile development process would expect that renaming a class at the model level would cause the generated artifacts to be renamed, too. He/she would NOT expect to get more artifacts.

Example:

Say, there is a UML class `Employee`. The user runs AndroMDA which generates `Employee.hbm.xml`, `Employee.java`, `EmployeeDao.java`, `EmployeeDaoBase.java`, `EmployeeDaoImpl.java`.

The user renames `Employee` to `Person` and runs AndroMDA again. What does he/she expect? Of course: `Person.hbm.xml`, `Person.java`, `PersonDao.java`, `PersonDaoBase.java`, `PersonDaoImpl.java`. Plain, clear and simple, right?

Chapter 2

Simple requirements – complex consequences

The rename operation in the last paragraph sounds quite simple. However, think about what really happens:

- The generator recognizes that the same UML class has a different name, now.
- The names of several classes change. This requires a change of the source text in generated as well as in hand-written files!
- The file names of several files change on disk.
- The `*Impl.java` files with the hand-written code have to be renamed in the configuration management system (e.g. CVS), too!

This is complex – how on earth should AndroMDA ever do this? On the other hand, if this would work, it would be a developer's dreams come true…

Making AndroMDA smarter

AndroMDA has to learn a lot to make the things above happen smoothly. First of all, the generator has to have a memory – it should remember what it generated in the previous run. Each UML model element has a unique id, which AndroMDA should save when it generates something for this model element. That is the only way to distinguish a renamed class from a new class.

Next, changing something (e.g. a class name) inside an existing text file is nasty. AndroMDA would have to remember exactly which part of the file contains the original name, change that part and leave the rest of the file intact. This applies only to generated files – there is no such solution for hand-written files.

Next, AndroMDA would have to change file names on disk and inside CVS (or other configuration management systems). Do you think that a decent code generator should do something like that? Probably not!

At last, there is one more thing: Renaming a class will cause the compiler to throw error messages at the user because all the references to that particular class in the user's hand-written code would also need to be renamed. Oh boy, refactoring at the model level seems virtually impossible to implement!

Chapter 3

AndroMDA needs help from its friends

If we tried to implement all this in AndroMDA itself, we would duplicate much effort which has already been done in another project which is well known world-wide: Eclipse. Eclipse can refactor correctly in Java, so why not use this feature?

How about running AndroMDA as an Eclipse plug-in? It could generate files and tell Eclipse about them. A user could refactor something in the model, AndroMDA could tell Eclipse about the changes. Eclipse could change the necessary items (classes, interfaces, methods, etc.) at the code level. More than this, Eclipse could tell CVS that a file name has changed.

Seems pretty cool and seems to meet the user's original expectations!

How can name changes possibly work?

First of all, AndroMDA must not forget about the artifacts it has generated in the previous run. So there is a need for a persistent dictionary of generated items where an item is identified uniquely so that it can be found even if the name of the originating model element has changed.

Next, there must be some code in AndroMDA that checks each model element to be processed, whether its name has changed or not. If it has, refactoring commands should be generated to inform Eclipse what to do.

This is the key thing: AndroMDA runs in the context of an Eclipse plug-in which should request the Eclipse JDT to rename the artifacts in question (Java packages, classes, attributes, methods, etc.). Eclipse should then perform the necessary refactorings.

Changing a model element name – is that the only refactoring we need?

Well, not really. It would already be a big plus for the user but it's not the whole thing. Another example might be to move a method from one class to another. This should then be done at the code level, too.

Another nice refactoring at the model level would be to change the inheritance relationship of a class, e.g. let a class B inherit from C instead of A. What do you think would happen then?

More: How about moving a model element from one package to another?

You see, there are many refactorings at the model level which should be reflected at the code level, too. Let's start with name changes (they are complex enough to implement!) and see what comes next.

Requirements

There are two possible use cases for model change propagation:

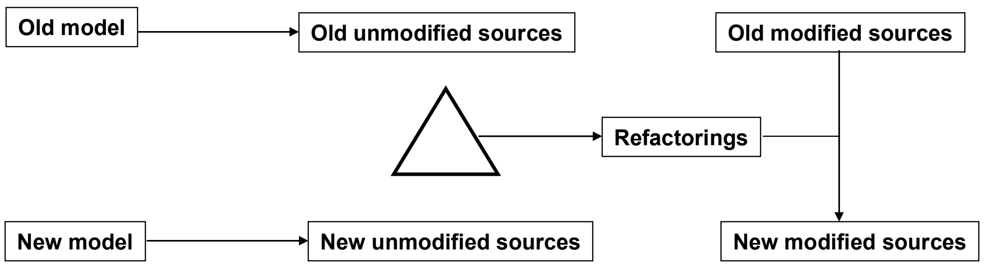
- Integrated operation inside the CASE tool. If AndroMDA were integrated in some decent CASE tool, it could react to model changes immediately. The CASE tool could notify AndroMDA about the changes the user makes to the abstract syntax tree, then AndroMDA could calculate the necessary code changes, as soon as the user presses the Save button in the CASE tool.

- Batch mode code generation as AndroMDA does it today. If we don't have CASE tool integration, we could make AndroMDA load two versions of the model: an old version, before the last run of the code generator, and a new version, before the next run of the code generator. As a first step, AndroMDA would have to calculate a "diff" between those two models and try to find out what the user changed in the more recent model version. Then, it could proceed as in the first use case.

Brainstorming for possible solutions

At the openArchitecture conference on December 1, 2004, I heard an interesting presentation by Axel Uhl, a former lead architect of the well known MDA tool *ArcStyler* by Interactive Objects. He is now with SAP to do "MDA on a large scale".

His talk was about "Leading Edge MDA". One of the slides in it was particularly interesting to me because Axel sketched a possible approach to my "dream AndroMDA feature": refactoring at the model level. The sketch on the slide was this one: (with Axel's kind permission)



The sequence of steps as described in the picture above:

- Generate code from old model
- Make refactorings to model
- Generate code from new model
- Determine differences in generated source code to get refactorings
- Apply refactorings to manually-modified source code

Example:

- change attribute name in model
- re-generate and diff → notice change in name of get/set operation and attribute member
- apply refactoring to entire source
- results in adaptation of manually-edited code accessing the attribute

The triangle in the picture above means "delta", i.e. the "diff" between two sources.

Diffing at the PSM AST level

Now, before we start to prototype this, I thought: "Calculating the diffs between two generated PSM sources, this may be difficult. Wouldn't it be easier to produce a true intermediate PSM abstract syntax tree (AST) first, then calculate the diffs on the two PSM ASTs?". I thought this because it felt so strange to generate something and calculate diffs on it - this felt like re-calculating a knowledge that the cartridge already has.

Axel responded by email:

"Diffing the sources certainly should be done on the AST because you'll be interested in the structural, not the textual differences. The key question is how the cartridge is structured internally. In ArcStyler, for example, cartridges were able to emit whole artifacts. The cartridges would deal with a model in the way they found it, not looking at any differences. It would be awfully hard to determine the effects that a model delta would have on the cartridge's output just from "looking" at

or using the cartridge. For those kinds of approaches (not entirely declarative, no impact analysis possible at reasonable cost), I'm still convinced that first doing the mapping and then performing the diff on the mapping results is the right way."

Diffing at PIM AST level

Another idea came to my mind: How about doing the diff at the model level and translating the model changes to PIM-level change commands? Then, run the PIM-level change commands through the cartridge and let the cartridge produce PSM-level change commands? Then, apply these change commands to the generated as well as the handwritten code, using refactoring frameworks (RefactorIt) or IDEs (Eclipse, IntelliJ, etc.).

Axel also responded to this:

"As noted above, this would require an internal cartridge structure that is aligned along mapping deltas as opposed to mapping a (part of a) model to another (part of a) model. The cartridges that ArcStyler ships, the entire CARAT approach, and I assume also the AndroMDA approach are not like this. Of course, this doesn't say this isn't possible, but it would require a very different approach which I believe could be much harder to design and implement."

Choosing one approach

I agree with Axel that a diff at PIM level, producing change commands, is a too difficult approach because the cartridges would have to be rewritten completely. A cartridge would not have to translate a model but changes to a model which is a completely different thing. Our cartridge design does not allow this, and it would be too much work to change it.

So, we should choose the *diff at the PSM AST level* approach. This is not easy, either, but it is possible with a reasonable amount of work.

Now, what do we have to change to make PSM AST diffing possible? Well, first of all, we have to produce a PSM AST at all! This is a less trivial statement than it sounds: at the moment, hardly any cartridge uses a true PSM AST! Instead, almost all PSM objects used by the cartridges are of type `java.lang.String`! The metafacades return tiny little Strings to be filled into the template placeholders (see the $-syntax in Velocity).

Instead, there have to be cartridge-specific classes that represent the conceptual structure of the target artifact. For example, for the bpm4struts cartridge, there could be classes like `Page`, `Form`, `InputField`, `Button`, `Event`, `Forward`, `Action`, and so on. For the Spring cartridge, there could be classes like `Service`, `ServiceBaseClass`, `ServiceImplementation`, `DaoInterface`, `DaoBaseclass`, `QueryMethod`, and so on.

It turns out that such classes are metaclasses of the PSM metamodel! The diff has to run on two instances of a PSM metaclass, producing change commands that can be sent to an underlying infrastructure, like Eclipse or RefactorIt.

The next new features, *cartridge-specific metamodels* and *model to model transformations*, which are described in the next chapter, will make it a lot easier to implement a true PSM AST than today. Let me describe them first before we go into the details of the necessary changes to AndroMDA.

Chapter 4

Cartridge-specific metamodels

A model is always an abstraction of a target domain. Metamodels are models that describe the target domain of *modeling*. For example, the UML metamodel contains metaclasses that are able to describe UML models, like `Classifier`, `ActionState`, `Operation`, and so on.

Cartridge-specific metamodels are designed to describe other models: models for the generated artifacts. For example, for a cartridge that generates JSPs for a web application, there could be metaclasses like `Page`, `Form`, `InputField`, `Button`, `Event`, `Forward`, `Action`, and so on. Instances of these metaclasses represent the artifacts which the cartridge is about to generate.

Metamodels are based on MOF, the metaobject facility designed by the OMG. Netbeans MDR which is an implementation of MOF, is able to load any MOF-based metamodel, followed by any model based on that loaded metamodel. Pretty cool.

The Netbeans MDR people wrote a special utility called *UML2MOF*, a Java program that can translate a UML model, designed with special stereotypes and tagged values from the MOF profile, to a true MOF metamodel in MDR. MDR can save that metamodel as XMI. Using UML2MOF, we'll be able to design cartridge-specific metamodels and package them with a cartridge so that they can be instantiated at runtime when AndroMDA invokes the cartridge.

Model to model transformations

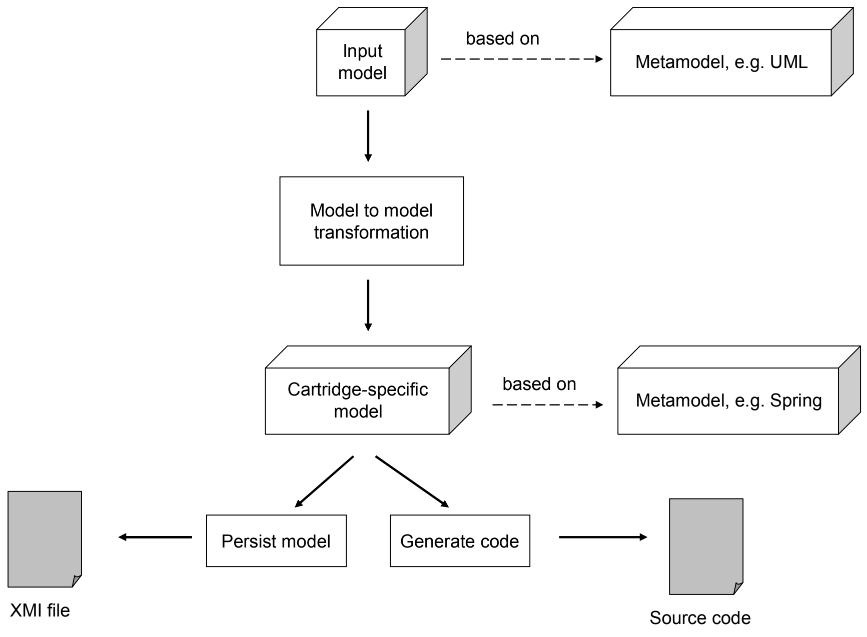
With MDA, it is possible to transform a model to another. The metamodels of the input model and the output model of such a transformation need not be (and most of the time: will not be) the same. Imagine a new way how cartridges could work:

- Transform from UML to a (less abstract) cartridge-specific metamodel
- For very complex cartridges, transform into the next metamodel which transforms into the next (even less abstract) metamodel, and so on, until you transform into a metamodel which closely resembles your output artifacts
- From the least abstract metamodel, generate the output artifacts, e.g. code

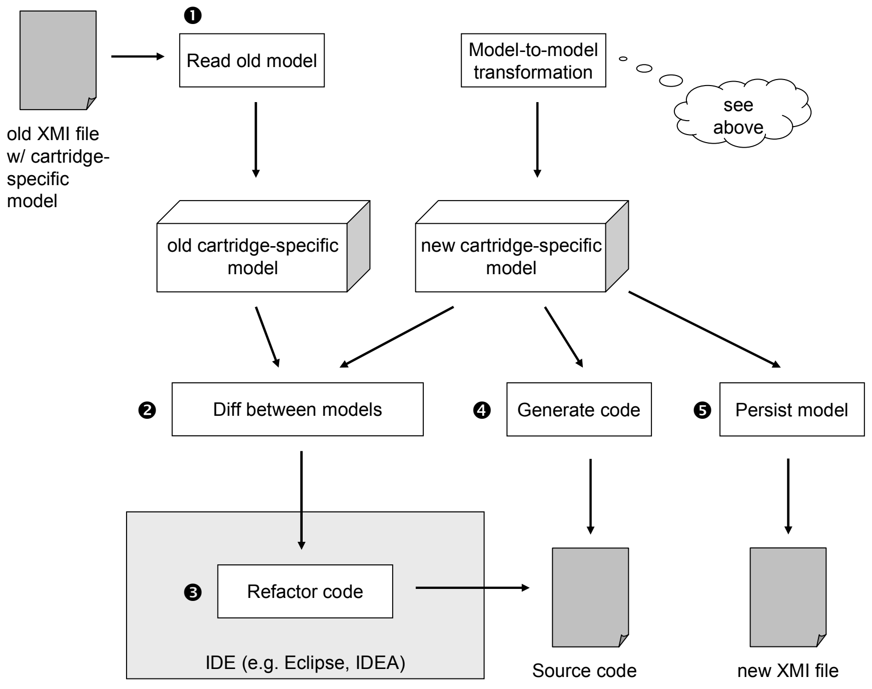
The advantage of using such a stack of metamodels is that you always only bite off as much as you can chew: each step leads you from a higher to a lower abstraction level with precisely defined semantics. You end up with a metamodel (and hence with its instances, abstract syntax trees) that closely resemble your target artifacts, *minus the complexity of their concrete syntax*. This is exactly the right point where we should run the diffs which we need to calculate the refactorings mentioned earlier!

Integrating transformations into a cartridge

The first code generation in a new AndroMDA 4.0 cartridge looks like this, if the generator has never been run before:



Each subsequent run of AndroMDA 4.0 with the same cartridge works like this:



In each run of the code generator, the new cartridge-specific model is compared against the old. Then, the differences are translated into refactoring commands for the IDE which executes them. After that, the usual source code generation takes place, using the template engine.

Necessary metamodels

There must be at least one metamodel per cartridge plus one metamodel for the global AndroMDA configuration with `Namespace`, `Property` and `Mapping` metaclasses. That way, the transformation scripts will have access to the global configuration. There may be intermediate metamodels that span multiple cartridges, too, because they contain common abstractions. An example for this could be common metaclasses for all cartridges which generate code for web controller frameworks (cartridges like bpm4struts, jsf, etc.).

Architectural changes in AndroMDA

Significant changes have to be made to AndroMDA's architecture to make model transformations work:

- Our `RepositoryFacade` interface has to learn about extents. A repository can have multiple extents, just like directories inside a file system. In each extent, there can now reside a model or a metamodel. There must be a new method `createExtent(name:String):ExtentFacade`, and all the other methods `readModel()`, `writeModel()` and `getModel()` should be moved to the new interface `ExtentFacade`.

- A cartridge may now include model transformations, in addition to metafacades and templates. Model transformations are `*.asm` files from ATL or scripts from any other transformation engine. They operate on one or more source model(s), stored in (a) source extent(s) and produce a target model, stored in a target extent. This means that a cartridge can now reference multiple extents in the repository. The cartridge descriptor needs an improvement so that you can specify which transformations from which extents and metamodels should be executed in which order. The cartridge descriptor could specify this as a pipes-and-filters architecture.

- A cartridge can now reference multiple metamodels. The cartridge descriptor needs an improvement so that you can specify which metamodels should be loaded when a certain cartridge is being initialized.

- Templates now trigger on PSM metafacades, not on UML metafacades any more.

- Metafacades now operate on arbitrary extents. So, `metafacade.xml` needs an improvement so that you can specify in which extent the metafacades will find their metaclasses that they are supposed to wrap.

- Metafacades will not operate on the UML metamodel any more. They will operate on PSM metamodels supplied by cartridges or used independently from a cartridge (e.g. the 3GL metamodel). There will be less metafacades than before because many of their responsibilities will be moved to model transformations.

- `cartridge.xml` will now map PSM metafacades to templates.

- Transformation libraries may now include global ATL transformations, too. These will be used for more than one cartridge at a time, that's why these "global" transformations are not kept inside a particular cartridge.

- There will be new metamodels for PSM purposes. They will be able to use and extend each other, e.g. the Hibernate metamodel extends the 3GL metamodel. Metamodels will be generated from UML models, using the UML profile for MOF and the UML2MOF utility. The UML models will be MagicDraw modules that may reference each other.

- Model elements inside PSM metamodels must have a *flavor*. Flavors must be kept in a global dictionary. When the Spring cartridge runs a model transformation from UML to the Spring metamodel and then to the 3GL metamodel, it must tell the 3GL Class that it has the "SpringService" flavor (for example). The Spring cartridge must register its "services" outlet with the "SpringService" flavor. The Java cartridge will notice later on that has to generate code for a Class with the "SpringService" flavor and must redirect the output to the outlet which it finds in the global flavor dictionary.

- `cartridge.xml` will have to be modified so that a template does not care about the outlet any more. Outlets will now be chosen by the flavor of the generated artifact, instead.

All this must be done to satisfy the following use case examples:

- Imagine the Spring cartridge: It starts a model transformation from the UML metamodel to the Spring metamodel and translates a UML `<<Service>>` class to an instance of the `Service` metaclass inside the Spring metamodel. Then, it starts a second model transformation from the Spring to the 3GL metamodel and translates the Spring `Service` to a 3GL `Interface`, a 3GL *ServiceBase `Class` and 3GL *ServiceImpl `Class`. It also runs a template that translates the Spring Services to `applicationContext.xml`. Java code for the `Services` is not produced inside the Spring cartridge any more - this is done in the Java cartridge, now.

- Imagine the Java cartridge: It generates Java code from an extent where a 3GL model is stored. It does not care about where the `Class` elements came from (i.e. which other cartridge did the model transformation). It runs templates on instances of the `Class` and `Interface` metaclasses of the 3GL metamodel.

- Imagine a C# cartridge: It generates C# code from the same extent where the 3GL model is stored. It does not care about where the `Class` elements came from (i.e. which other cartridge did the model transformation). It runs templates on instances of the `Class` and `Interface` metaclasses of the 3GL metamodel and produces C# code from them.

CHAPTER 5

Chapter 5

Model Element Differs and Change Commands

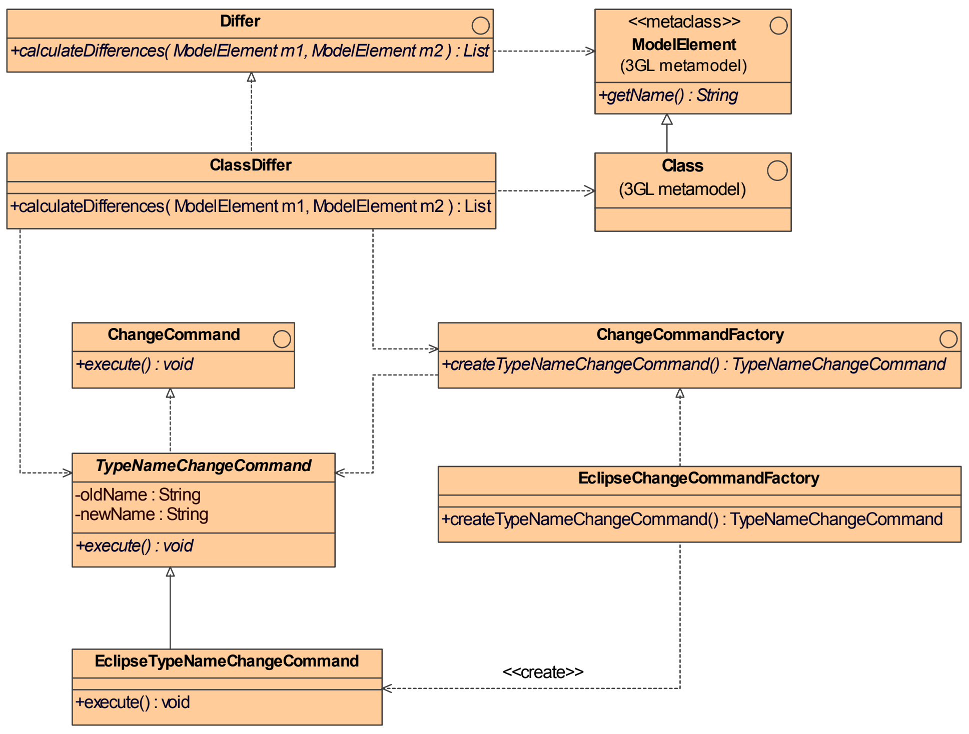
Once model transformations are in place and intermediate models are persisted and reloaded, we can start to think about calculating differences between an old and a new version of an intermediate model. I'll show you this using an example.

Let's say, we're inside the Java cartridge that usually generates Java classes. In AndroMDA 4.0, it generates a Java class from an instance of `Class` from the 3GL metamodel.

The Java cartridge contains a new class `ClassDiffer` which is responsible for calculating differences between two instances of `Class`. A `ClassDiffer` should return a `List` of `ChangeCommands` which will (among others) contain instances of `TypeNameChangeCommand`.

A factory is responsible for creating instances of `ChangeCommand` because those will be dependent on the particular refactoring environment, e.g. an IDE like Eclipse or a refactoring framework like RefactorIt.

The relationships between all those classes are shown in this UML diagram:


Differ
+calculateDifferences(ModelElement m1, ModelElement m2) : List
<<metaclass>>
ModelElement
(3GL metamodel)
+getName() : String
ClassDiffer
+calculateDifferences(ModelElement m1, ModelElement m2) : List
Class
(3GL metamodel)
ChangeCommand
+execute() : void
ChangeCommandFactory
+createTypeNameChangeCommand() : TypeNameChangeCommand
TypeNameChangeCommand
-oldName : String
-newName : String
+execute() : void
EclipseChangeCommandFactory
+createTypeNameChangeCommand() : TypeNameChangeCommand
EclipseTypeNameChangeCommand
+execute() : void
<<create>>

The basic pattern is always the same:

- For each relevant metaclass, there is a `Differ`.
- For each concrete example of a refactoring, there is a `ChangeCommand`.
- For each IDE or refactoring framework, there is a `ChangeCommandFactory`.
- Differs operate on metaclasses and use a `ChangeCommandFactory` to create and return instances of `ChangeCommand`.
- Plugins for typical IDEs will interpret instances of `ChangeCommand` and will execute them in their particular environment.

In the example above, a `ClassDiffer` compares two instances of `Class`, the old instance and the recently generated instance. If the name of the recently generated instance differs from the name of the old instance, the `Differ` creates a `TypeNameChangeCommand`.

Chapter 6

Other features for AndroMDA 4.x

There are more ideas for features in 4.x than model-level refactoring only:

Feature name…
Describe it here…