Faculty of Automation and Computers

Bachelor Program: Calculatoare si Tehnologia Informatiei

Universitatea Politehnica Timişoara

# XCorex
# Framework for Software Analysis Tool Development

**Diploma Thesis**

Alexandru Stefanica

*Supervisor*
Assistant Professor
Dr. Eng. Petru-Florin Mihancea

Timişoara
June, 2015

# Contents

# Chapter 1

# Introduction

## 1.1 Motivation

Software development is a very complex process in which every component of the system must developed according to a strict set of guidelines. Every component is assest based on different metrics

## 1.2 Goal and Contributions

## 1.3 Organization

# Chapter 2

# Concepts and State of the Art

> The problem with object-oriented languages is they've got all this implicit environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle.
>
> Joe Armstrong

In this section we will discuss the most important concepts that are going to be used in this paper.

## 2.1 Fundamental Concepts

### 2.1.1 Modeling, Metamodeling

Object-oriented programming is a paradigm which appeared in the early '60 [11] and is based on the concept of defining object and relationships between them. With the rapid growth of software system object-oriented design has become a non-trivial problem even for small systems, thus the need of tools that can assist us in analyzing our design. The use of object technologies in detecting problems is called object-oriented analysis.

Modeling, as defined in the dictionary [3], is the representation, often mathematical, of a process, concept, or operation of a system, often implemented by a computer program. There is no general accepted definition for this term, but for the purpose of this paper we can see a model as a simplified version of a reality. As there

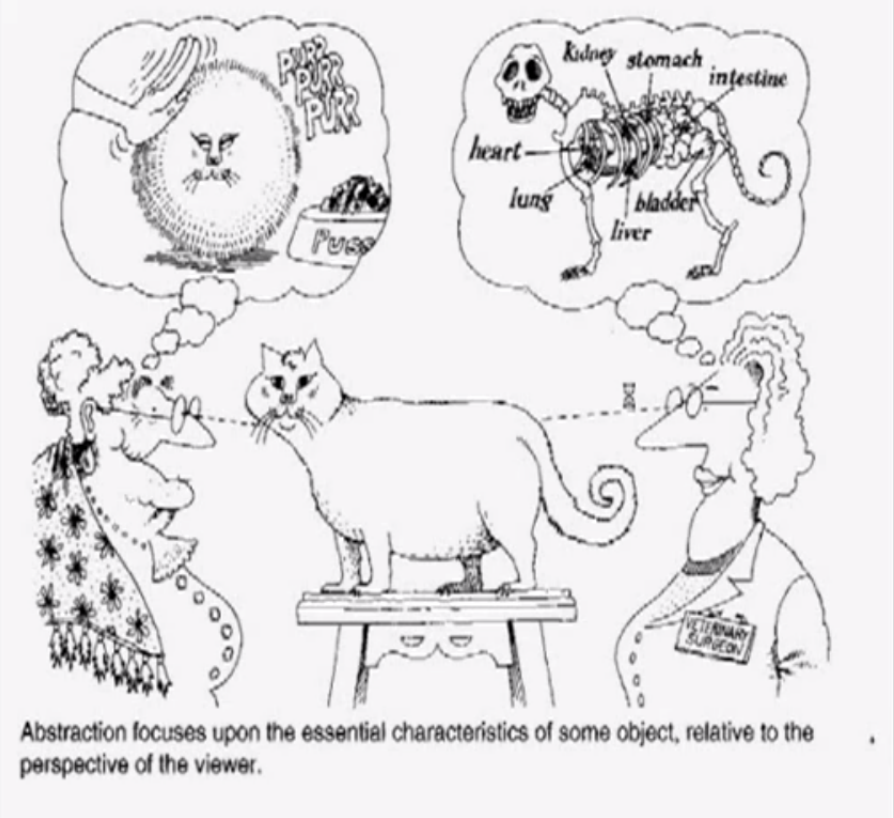



Figure 2.1: The same cat but from different perspectives [2]

can be many types of maps for the same territory depending on the purpose (riding a bike, traveling to cities, sightseeing) there can also be many types of models for the same system depending from which angle we want view the system. Figure 2.1 illustrates the idea.

In order to be able to work with models software analysis tools need to be able to understand and represent them, thus the need of a way to describe models. For this the concept of metamodeling has been introduced. The prefix meta- indicates an abstraction of a concept [12]. Formally, metamodel is an abstract syntax which governs the representation of a class of models. One of the most common metamodels used in object oriented design (modeling) is UML [5]. One is able to describe it's object oriented system by simply showing and describing the most important entities, i.e objects, and how they interact with each other. In figure 2.2 we have representation of the XML metamodel described using UML.

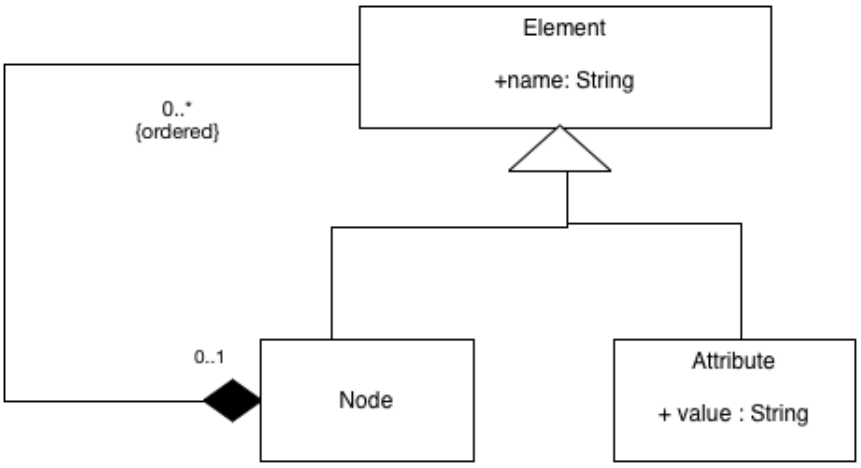



Figure 2.2: XML metamodel

A very common case with tools is that people want them aggregated in one larger tool. In order to do this we will need to define a way to describe software analysis tools. As software analysis tools use metamodels for describing models, it natural to assume that we will need a meta-metamodel in order to describe a metamodel. Thus a metamodel is also governed by a strict set of rules that form a meta-metamodel. One can easily see that the definition is recursive and we could continue with it indefinitely. In time many meta-metamodels have been implemented in order to solve different problems (e.g Ecore, GME, KM3, ... etc). For the purpose of this application we have used a simple meta-metamodel based on three elements:

- Entity — represents a generalization of a package, type, method, field.
- Property — represents a general metric for an entity
- Groups — represents a tuple of one or more entities that are connected by a relation (e.g inheritance, composition, generalization, all elements of an entity, ... etc)

The figure 2.3 represents the relationship between model, metamodel, and so on.

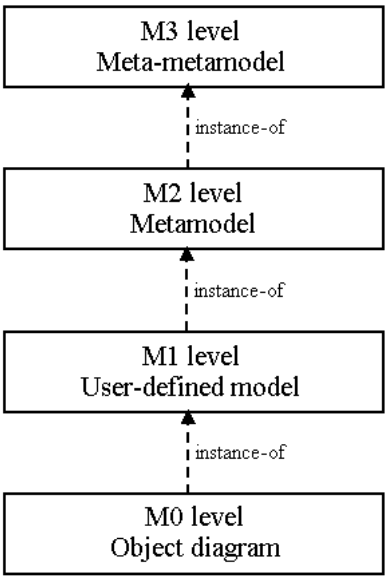



Figure 2.3: Models and Metamodels

### 2.1.2 Ecliplse plugin

In computing, a plug-in is a software component that adds a specific feature to an existing software application. When an application supports plug-ins, it enables customization. The best software analogy compares a plug-in to an object in object-oriented programming. A plug-in, like an object, is an encapsulation of behavior and/or data that interacts with other plug-ins to form a running program. Eclipse is an extremely popular Java Integrated Development Environment (IDE) and a strong competitor to NetBeans/SunOne Studio, JBuilder, and IntelliJ IDEA. Eclipse is made of a small core, with many plugins layered on top. Some plugins are nothing more than libraries that other plugins can use. These are the many tools at your disposal for the job. The base libraries used by all plugins are:

- The Standard Widget Toolkit (SWT): Graphical components used everywhere in Eclipse: buttons, images, cursors, labels, etc. Layout management classes. This library is usually meant to be used instead of Swing. JFace: Classes for menus and toolbars, dialogs, preferences, fonts, images, text files, and base classes for wizards.

- The Plugin Developer Environment (PDE): Classes to help with the data manipulation, extensions, build process, and wizards.

- The Java Developer Toolkit (JDT): Classes used to programmatically manipulate Java code.

An Eclipse plug-in has the following main configuration files. These files are defining the API, and the dependencies of the plug-in. MANIFEST.MF - contains the OSGi configuration information. plugin.xml - optional configuration file, contains information about Eclipse specific extension mechanisms. An Eclipse plug-in defines its meta data, like its unique identifier, its exported API and its dependencies via the MANIFEST.MF file. The plugin.xml file provides the possibility to create and contribute to Eclipse specific API. You can add extension points and extensions in this file. Extension-points define interfaces for other plug-ins to contribute functionality. Extensions contribute functionality to these interfaces. Functionality can be code and non-code based. For example a plug-in might contain help content.

In figure 2.4 you can see an example of a development process for a plugin.

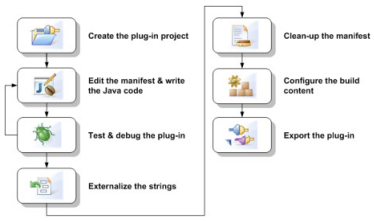



Figure 2.4: Eclipse Plugin Development [1]

### 2.1.3 Java Metadata

As mentioned above a model is nothing more or less than a simplified version of a reality/system. In case of software analysis tools this system is actually a program. In order to be able to manipulate programs (models) we need specific language support for this type of programming, also called metaprogramming [10].

Many languages have different ways and levels of supporting metaprogramming, in C++ this is done by using the template system (template metaprogramming), in Java this is done by using the reflection API provided by the compiler and/or by using the annotation language feature which allows metadata processing, i.e the ability to add information to your code so that it can be used later for generating boiler plate code or by enforcing constraints that can be verified at compile-time or run-time. [4]

The syntax for defining an annotation is quite straight forward and resembles very mucth the definition of an interface. The keyword used is `interface` prefixed with the 'at' (@) sign. Different properties can be defined inside the annotation. If you set a default value for the property it becomes optional. The type of a property can be primitive types, `Class<?>`, enum types or String.

E.g:

```
1         @Retention(RetentionPolicy.SOURCE)
2 public @interface CreateWarning {
3         String[] warning() default "Something is fishy";
4 }
```

Usage:

```
1 Set<String> names = new HashSet<>();
2 ....
3 //this message will be outputted as a warning
4 @CreateWarning("unsafe cast, must change")
5 String[] nameArray = (String[])names.toArray();
```

```
1 //the default message, will be outputted
2 @CreateWarning
3 public boolean equals(Object x) {
4 ...
5 }
```

Of course, as we love recursion, we have meta-annotations which allow us to describe important information regarding annotations such as:

- what king of elements can the current annotation annotate
- how to store the information is provides ? (in the source file, in the class file, just drop it after compilation, . . . etc)
- if it can be repeated more than one time on the same element

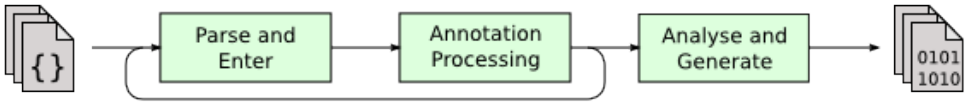



Figure 2.5: The compiler process with annotation processing[9]

- if the element is documented

An example of such an annotation is provided in the definition of `@CreateWarning`, i.e `@Retention`.

The most common annotations that are used in java are:

1. **@Override** – indicates that the current method is an (re)implementation of method from the base class.
2. **@Deprecated** – usually used in frameworks and indicates that the entity will be removed in feature updates and support for it is no longer provided.
3. **@SuppressWarnings** – stop the compiler from adding annoying warning.

The most complex part comes when we want to associate some semantics to our annotations. This is usually done by defining an **annotation processor**, i.e a compiler plugin which will be invoked when our annotation(s) have been discovered by the compiler. After each annotation processor has finished some extra code might have been generated or some annotations might have been expanded in other annotations or both, in this case the compiler starts processing the code and invoking the annotation processors again as can be seen in figure 2.5. In order to define an annotation processor one most create a class that inherits from `javax.annotation.processing.AbstractProcessor`. The most important function is `process(Set<TypeElement> annotations, RoundEnvironment roundEnv)`, here you write your code for scanning, evaluating and processing annotations and generating java files. With RoundEnviroment passed as parameter you can query for elements annotated with a certain annotation.

### 2.1.4 Mirror API

Object-oriented languages traditionally support meta-level operations such as reflection by reifying program elements such as classes

into objects that support reflective operations such as get- Superclass or getMethods. In a typical object oriented language with reflection, (e.g., Java, C#, Smalltalk, CLOS) one might query an instance for its class, as indicated in the pseudo-code below:

```
1 class Car {...}
2 Car myCar = new Car();
3 int numberOfDoors = myCar.numberOfDoors();
4 Class theCarsClass = myCar.getClass();
```

Another approach is used in many scripting languages. The language constructs themselves introduce code on the fly, modifying the program as they are executed. For example, a class comes into being when a class declaration is evaluated, and might change if another declaration of a class with the same name is executed later. The third approach is that of mirrors, and originates in Self. Mirrors have been used in class based systems such as Strongtalk, and even in the Java world. JDI, the Java Debugger Interface, is a mirror based reflective API. Here, the reflective operations are separated into distinct objects called mirrors. This seemingly minor restructuring has significant implications. Reflection is no longer tied into the behavior of every object in the system (as it is via getClass()) or (even worse) into the very syntax of the language. Instead, it resides in separable components that can be removed or replaced. Reflection is now a distinct capability, in the sense of the object capability model. The most important thing understand about mirrors is that they decouple the reflection API from the standard object API, so instead of obj.getClass() you use reflect(obj). It's a seemingly small difference, but it gives you a few nice things:

- The object API isn't polluted, and there's no danger of breaking reflection by overriding a reflective method.
- You can potentially have different mirror systems. Say, one that doesn't allow access to private methods. This could end up being very useful for tools.
- The mirror system doesn't have to be included. For compiling to JS this can be important. If mirrors aren't used then there's no out-of-band to access code and pruning becomes viable. Mirrors can be made to work on remote code, not just local code, since you don't need the reflected object to be in the same Isolate or VM as the mirror.

## 2.2 Tools: CodePro

Software analysis tools are usually used by developers when something went wrong or when someone wants to validate an idea. The developer will be running the analysis tool(s) to compute complex metrics and interpreting the abstract numbers that the tool outputted and will try, in case of abnormal values, to find the problem and propose a solution for it. The entire process is a pain kill and mostly likely comes in the most inappropriate moment. [8].

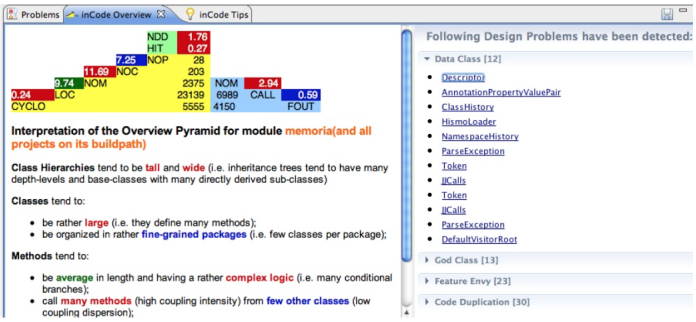



Figure 2.6: inCode overview [8]

CodePro, or by its comercial name INCODE [6], is an eclipse plugin which plans to solve this shortcomings of analysis tools by providing continuous analysis of the code, marking problems solved and detecting new ones, also providing arguments and tips on why was the problem detected, what causes it and how it can be solved. In the figures 2.7 and 2.6 you can see examples of the features mentioned above.

From an architectural point of view CodePro defines its own meta-metamodel ,very similar to the one presented in 2.1.1, but more complex. The metamodel which is used is the one provided

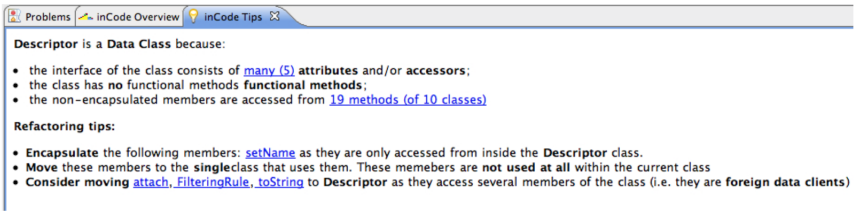



Figure 2.7: inCode tips [8]

by Eclipse JDT library, wrapped to fit the meta-metamodel defined. In 2.8 you can see an overview of the general architecture. From analyzing the diagram we can spot some architectural problems related to type safety such as:

- using strings in order to identify different metrics, types, filters ... etc. What will happen when someone changes the name of the entity ? What if we write Class instead of class ?
  ```
  aSystem.getGroup("class group").applyFilter("model
  class");
  ```
- property computers accept abstract entities instead of specific elements for which the computers are defined. The semantic definition is not preserved. If we use the Cyclomatic Complexity metric on a class or package instead of a method ? What is the expected return type ?
  ```
  public ResultEntity compute(AbstractEntityInterface anEntity)
  ```

The tool that I have created is aimed at solving this shortcoming by providing a way in which you can define a metamodel and it will generate, based on the provided metadata, a model, everything done in a type safe manner.

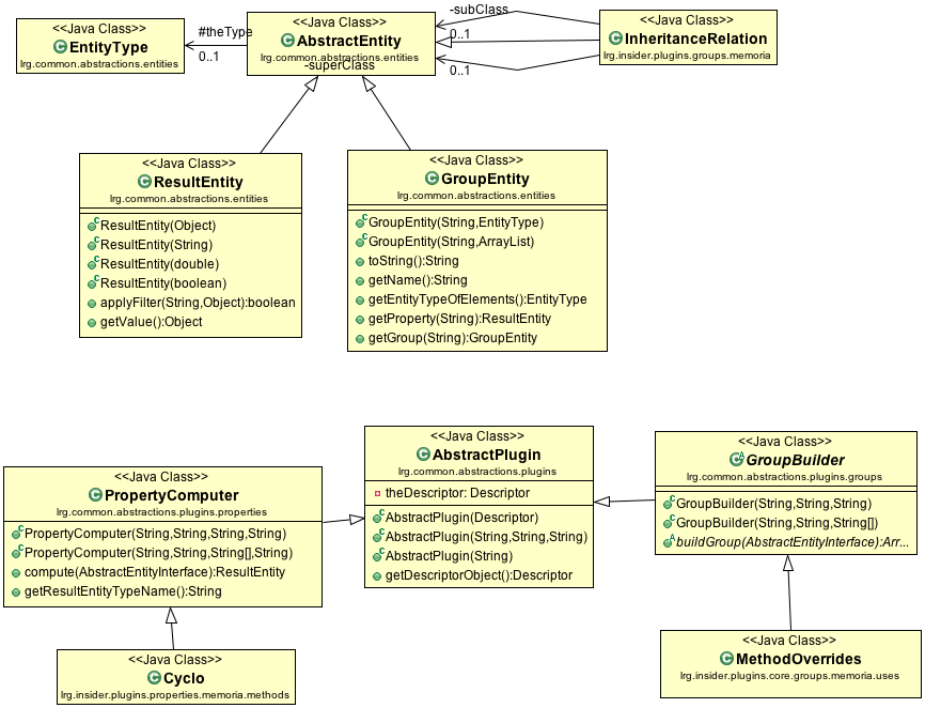



Figure 2.8: CodePro meta-metamodel and meta-model overview

# Chapter 3

# Yet another tool: XCorex

> Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.
>
> Rich Cook

For the problems presented in chapter 2.2 I have implemented a tool that defines a meta-metamodel as described in chapter 2.1.1 which will allow you to describe the metamodel for the tool you want to implement and it will generate a model based on the data you provided. In order to evaluate the application I have also reimplemented a front-end tool called InsiderView [7] which allows you to integrate different metrics based on the meta-metamodel from CodePro. During the implementation phase there were a number of different ideas that were proposed and/or implemented in order to fully understand the limitations of annotation processing and code generation in java and also to provide the most general solution which can be used by as many users as possible.

## 3.1 XCorex

### 3.1.1 Solution Overview

Figure 3.1 represent the general system architecture. As it can be seen there are three main components that describe the XCorex system:

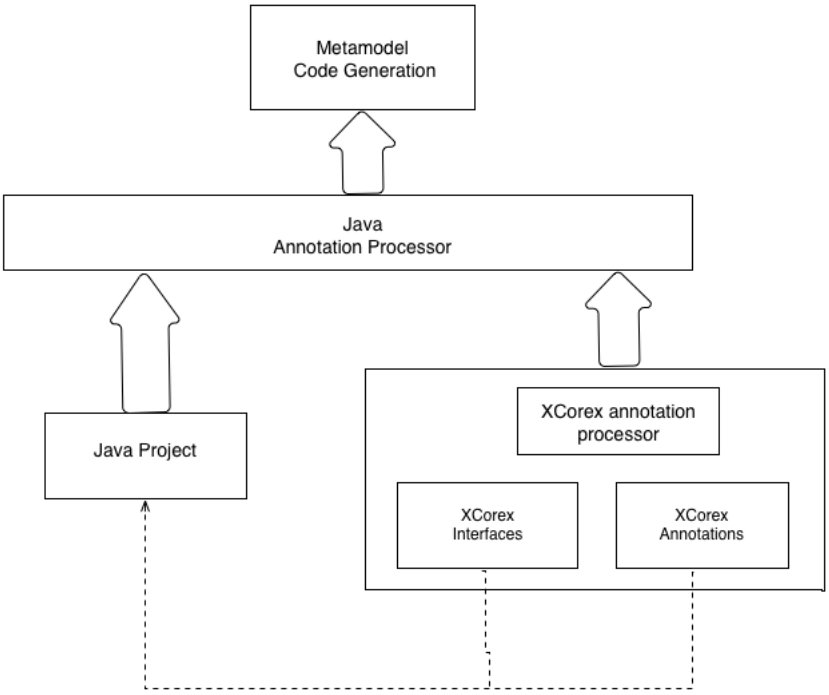



Figure 3.1: XCorex System overview

- Annotation Component
- Interface Component
- Annotation Processor

The XCore annotation component represent the implementation of the meta-metamodel as presented in chapter 2.1.1 and provides the necessary metadata to describe the metamodel which is implemented by the client. The XCorex interface component helps the client in describing the necessary elements for the model of the application and also enforces a type safe environment. The above two components describe the semantics of our system which is enforced by the annotation processor. The processor is, if you will, the brain of system which analysis the metadata provided by client with the help of the two components and generates the appropriate model.

### 3.1.2 Implementation Details

#### Annotation Component

The annotations we have defined combined with the interfaces, which will be presented in the section below form the meta-metamodel of our program as describe in section 2.1.1. As you can see in the code sections below the annotations are only allowed to annotate java types, to be more specific only classes, and help us distinguish between group builders, which describe any type of relation between a series of elements of the same type, and property computers which represent a general description of a metric.

```
1 @Target(ElementType.TYPE)
2 public @interface GroupBuilder {}
```

```
1 @Target(ElementType.TYPE)
2 public @interface PropertyComputer {}
```

#### Interface Component

Both annotations presented above force the annotated type to implement a specific interface which assures us type safety through generics. For the GroupBuilder annotation we have the IGroupBuilder interface and for the PropertyComputer annotation we have the IPropertyComputer interface, both definitions can be seen below.

```
1 public interface IGroupBuilder <ElementType extends
     XEntity,
2                 Entity extends XEntity> {
3         Group<ElementType> buildGroup(Entity entity);
4 }
```

For the GroupBuilder the ElementType represents the type for the aggregated elements and the entity represent the aggregator type. For example if we want to define the group of all methods for a class we would have something very similar to:

```
1 @GroupBuilder
2 public class ListOfMethods implements
     IGroupBuilder<XMethod, XClass> {
3         @Override
4         public Group<XMethod> buildGroup (XClass entity) {}
5 }
```

```
1 public interface IPropertyComputer <ReturnType, Entity
     extends XEntity> {
2         ReturnType compute(Entity entity);
3 }
```

The ReturnType can be any type which makes sens for the current metric to return (Double, Integer, String, Pair<>, ... etc). The XEntity represents a markup interface which define the elements of our metamodel which we will be generating when the compilation process starts and the annotation processor is invoked. The Entity type which is present in both interfaces must not exists before, because it will be generated.

### Annotation Processor

All of the elements presented above, annotations and interfaces, are defined by using the java syntax and thus we are confident that they will be used correctly from that point of view, but they also have semantics attached which cannot be enforced by syntax alone. In order to enforce the appropriate semantics we have defined an annotation processor which will parse every relevant project files, java files which contain elements annotated with the elements presented in the sections above, it will analyze every elements and, if needed, will generate the appropriate errors or warnings. The process of discovering which elements are annotated and with what kind of annotations is the responsibility of the compiler. Once our requested annotations are identified the compiler will invoke the processor by calling the `process(Set<? extends TypeElement> annotations,RoundEnvironment roundEnv)` method with the annotations it identified and are relevant for us, and the corresponding java elements that are attached to them. Each element is processed, first starting with the PropertyComputers and continuing with the GroupBuilders, and the following rules are enforced:

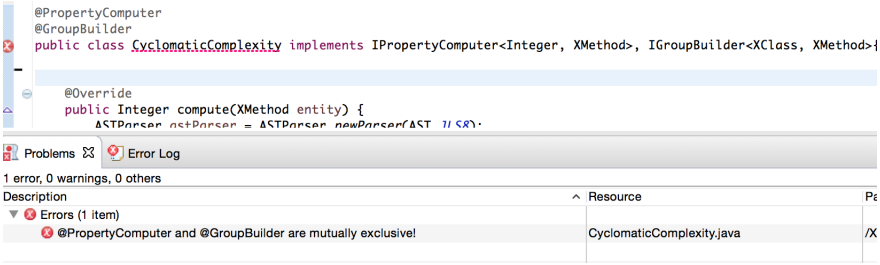
- All elements that are annotated must be classes
- All annotated classes must have a default constructor.
- All elements annotated with @PorpertyComputer must implement IPropertyComputer
- All elements annotated with @GroupBuilder must implement IGroupBuilder

- All classes cannot be present in the default package
- The @PropertyComputer and @GroupBuilder annotations are mutually exclusive
- No wildcard types are allowed to be specified for the entity type parameter.

When an invalid element is encounter an error similar to the one in figure 3.1.2 is presented. For the rest of the elements, which are valid, the entity type defined in the interface as a type parameter is extracted by using the mirror api provided by the compiler and the appropriate code. For each unique entity type found, an interface is generated which implements the XEntity markup interface and which contains a method for every PropertyComputer that uses this type and a method for every GroupBuilder which uses this entity for generating the elements of the group. Also if any comments are associated with the PropertyComputers or GroupBuilders they are preserved in the generated code. Figure 3.1.2 and 3.1.2 show an example on how the code must be written and what is generated.

Each interface generated in the previous step is implemented by defining the appropriate class in an impl package. The class will implement every method defined in the interface by simply instantiating and object of the appropriate PropertyComputer or GroupBuilder and call the compute method by passing a reference to `this`. Also, there is one addition method that is implemented by each entity and this is the `getUnderlyingObject()` method which returns the equivalent object from the framework we are using. For example if we are using Eclipse JDT framework an equivalent for a XClass entity would be IType. By default the method returns Object, but if the user specifies the type as shown in figure 3.1.2 the method will return the specified type as can be seen in figure 3.1.2

The implementation classes are not publicly available. In order to be able to instantiate an element a FacetoryMethod class is generated which also has cache support to increase speed and avoid useless object instantiation.




## 3.2 XCorexView

### 3.2.1 Overview

XCorexView is a tool based on XCorex which implements a series of software metrics

### 3.2.2 What has changed

# Chapter 4

# Evaluation

## 4.1 The Good and Bad

## 4.2 Insider View Reimplemented

### 4.2.1 Insider View overview

### 4.2.2

# Chapter 5

# Future Work and Conclusions

## 5.1 Future Work

## 5.2 The Good and Bad

## 5.3 Conclusions

# List of Figures

# List of Tables

# Bibliography


[1] Chris Aniszczyk. Plug-in development 101, part 1: The fundamentals. http://www.ibm.com/developerworks/library/os-eclipse-plugindev1/, 2008.

[2] Grady Booch. http://www.evinw.com/w/oop-concepts-php/.

[3] English Dictionary. http://dictionary.reference.com/browse/modeling.

[4] Bruce Eckel. *Thinking in Java*. Prentice Hall, 2006.

[5] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3 edition, 2003.

[6] LOOSE Reasearch Group. http://loose.upt.ro/reengineering/research/codepro.

[7] C. Marinescu, R. Marinescu, P. F. Mihancea, and R. Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume)*, pages 77–80. Society Press, 2005.

[8] Radu Marinescu, George Ganea, and Ioana Verebi. Incode: Continuous quality assessment and improvement. *2011 15th European Conference on Software Maintenance and Reengineering*, 0:274–275, 2010.

[9] neildo. Project lombok - trick explained. http://notatube.blogspot.ro/2010_11_01_archive.html, 2010.

[10] Abdelmonaim Remani. The art of metaprogramming in java. http://www.slideshare.net/PolymathicCoder/the-art-of-metaprogramming-in-java, 2012.

[11] Wikipedia. http://en.wikipedia.org/wiki/Object-oriented_programming.

[12] Wikipedia. http://en.wikipedia.org/wiki/Meta.