Faculty of Automation and Computers

Bachelor Program: Calculatoare si Tehnologia Informatiei

XCore: Framework for Software Analysis Tool Development

Diploma Thesis

Alexandru Ștefănică

Supervisor
Asistent Profesor
Dr. Ing. Petru-Florin Mihancea

Timişoara June, 2015

Contents

Chapter 1

Introduction

Any fool can write code that a computer
can understand. Good programmers write
code that humans can understand
Martin Fowler

1.1 Motivation

We all want to write quality code. Code that follows the company standards, that has an amount of test coverage, that does not use system hack, etc. In order to be able to measure and enforce this characteristics we need to use software analysis tools such as:

Wala Tool Developed by IBM Research it implements a series of dataflow and type analysis algorithms.

FindBugs Widely used tool to detect common programming language hacks

Intellij IDE A platform developed by JetBrains which implements over 100 code inspector tools

CodePro A platform which implements software metrics developed by LOOSE Research Group. Goes by the name of INCODE, also [9]

Of course there are many more tools such as the ones provided. These are only some of the most used ones.

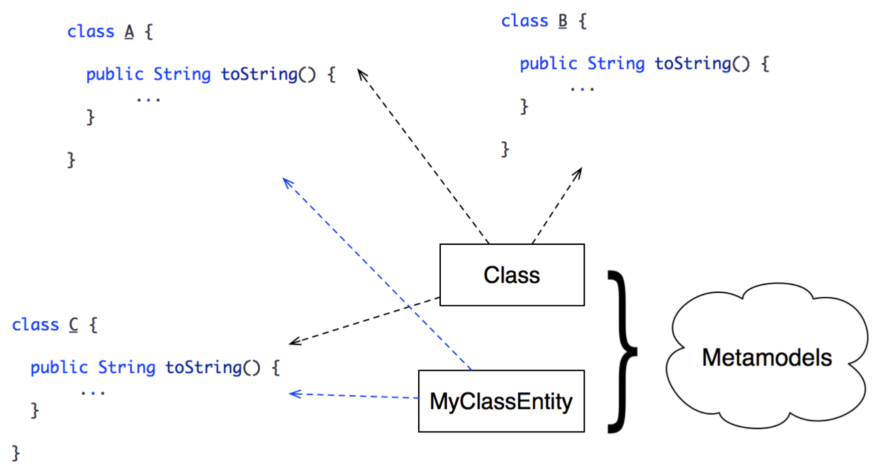

class A {
public String toString() {
...
}
}
class B {
public String toString() {
...
}
}
class C {
public String toString() {
...
}
}
Class
MyClassEntity
Metamodels

Figure 1.1: Two different metamodels for the same model

In order for the software analysis tools to be able to analyse the code they need a metamodel. A metamodel represent all the elements necessary to describe a model, in our case all the elements necessary to describe the code, i.e the classes from a program, the methods from the classes, the fields from the classes . . . etc. Consider the situation in figure 1.1

We have two valid metamodels that describe similar elements, but they cannot be integrated. If we want to build an easily extensible and integratable platform for tools we need more than a metamodel, we need a meta-metamodel. CodePro has such a meta-metamodel. In figure 1.2 we can see a glimpse of CodePro meta-metamodel and how easy it is to add new metrics. We only have to extend the `PropertyComputer` class

Unfortunately with the desire of extensibility some architectural design problems where introduced which breaks statically type safety and make it harder to understand and maintain the code. In figure 1.3 we can see an example of metric written in CodePro. With red I've marked the major problems for which we will present a solution. We can see that there are alot of magic constants present in the code. These magic constants are used to identify the tools in the system. From this a simple problem arises: What if we the id (e.g CYCLO) changes or we misspelled it ? Another problem is the generic `AbstractEntityInterface` which denotes every element from the metamodel. What would happend if for the Cyclomatic

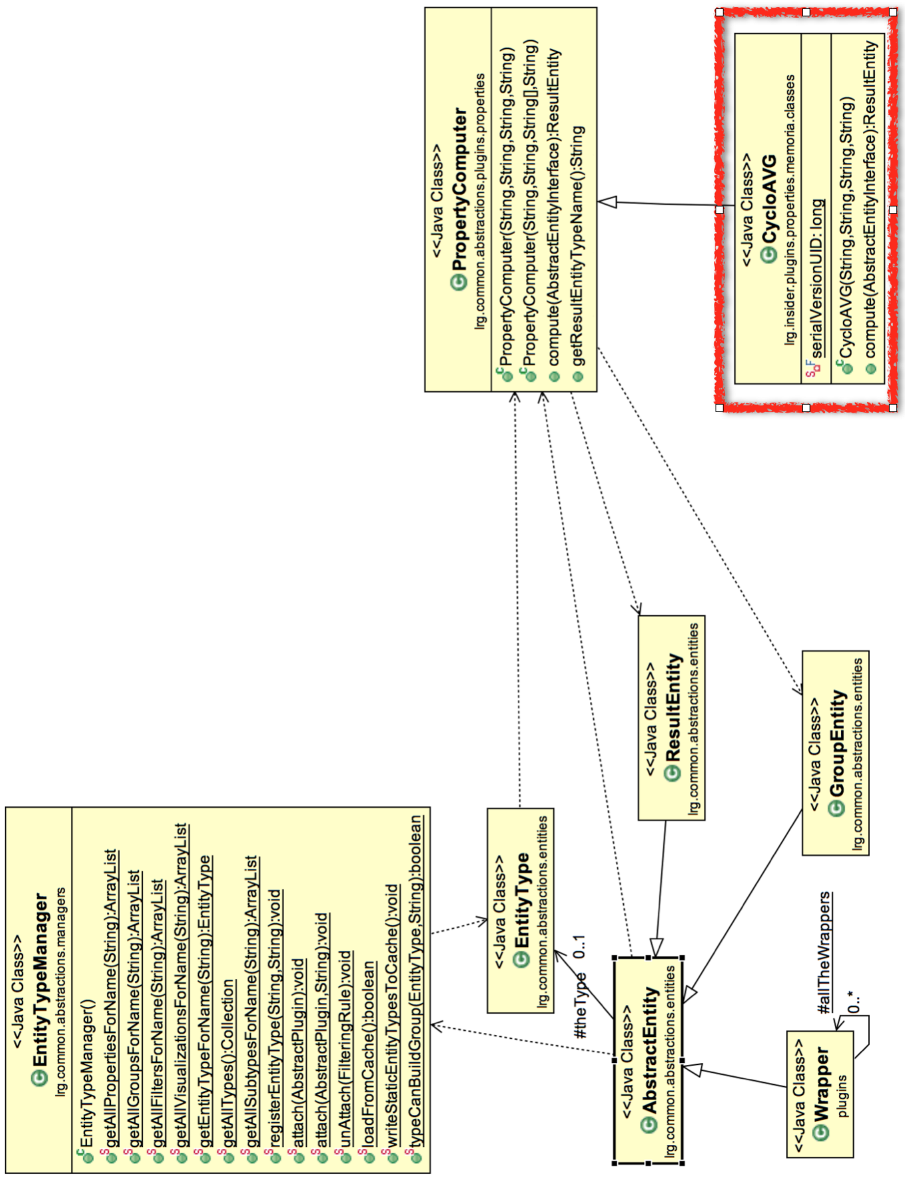
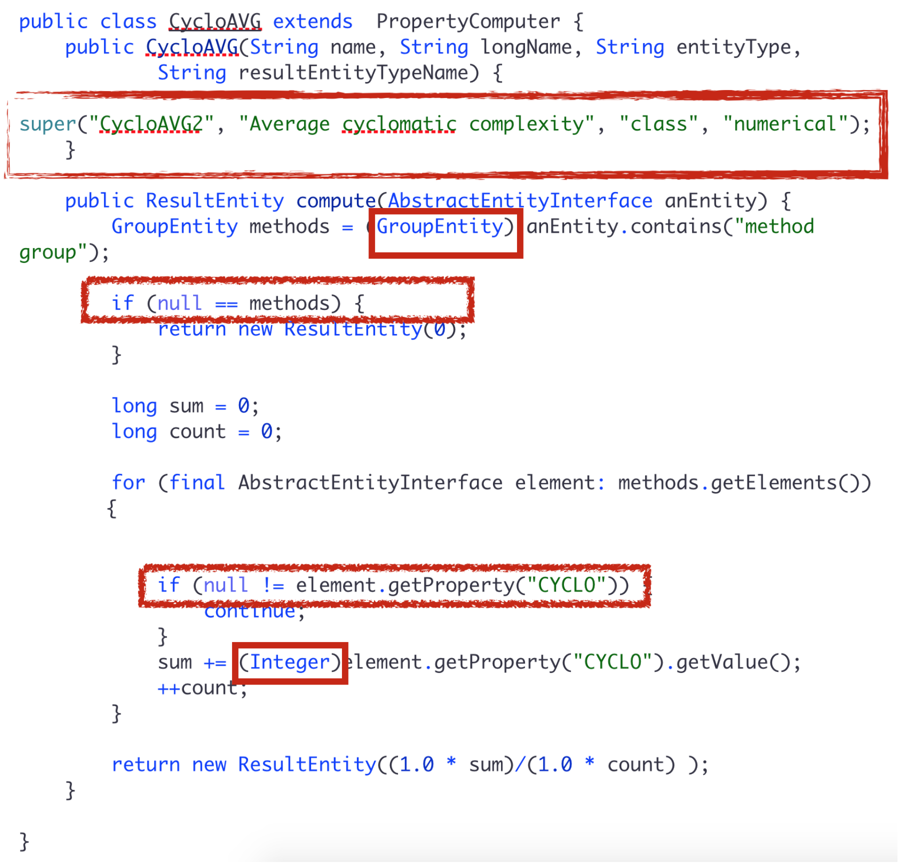

<<Java Class>>
PropertyComputer
lrg.common.abstractions.plugins.properties
PropertyComputer(String,String,String,String)
PropertyComputer(String,String,String[],String)
compute(AbstractEntityInterface):ResultEntity
getResultEntityTypeName():String
<<Java Class>>
CycloAVG
lrg.insider.plugins.properties.memoria.classes
serialVersionUID: long
CycloAVG(String,String,String,String)
compute(AbstractEntityInterface):ResultEntity
<<Java Class>>
EntityTypeManager
lrg.common.abstractions.managers
EntityTypeManager()
getAllPropertiesForName(String):ArrayList
getAllGroupsForName(String):ArrayList
getAllFiltersForName(String):ArrayList
getAllVisualizationsForName(String):ArrayList
getEntityTypeForName(String):EntityType
getAllTypes():Collection
getAllSubtypesForName(String):ArrayList
registerEntityType(String,String):void
attach(AbstractPlugin):void
attach(AbstractPlugin,String):void
unAttach(FilteringRule):void
loadFromCache():boolean
writeStaticEntityTypesToCache():void
typeCanBuildGroup(EntityType,String):boolean
<<Java Class>>
EntityType
lrg.common.abstractions.entities
<<Java Class>>
ResultEntity
lrg.common.abstractions.entities
<<Java Class>>
GroupEntity
lrg.common.abstractions.entities
<<Java Class>>
AbstractEntity
lrg.common.abstractions.entities
#theType 0..1
<<Java Class>>
Wrapper
plugins
#allTheWrappers
0..*

Figure 1.2: CodePro UML



```
public class CycloAVG extends  PropertyComputer {
    public CycloAVG(String name, String longName, String entityType,
            String resultEntityTypeName) {
super("CycloAVG2", "Average cyclomatic complexity", "class", "numerical");
    }

    public ResultEntity compute(AbstractEntityInterface anEntity) {
        GroupEntity methods = (GroupEntity) anEntity.contains("method
group");

        if (null == methods) {
            return new ResultEntity(0);
        }

        long sum = 0;
        long count = 0;

        for (final AbstractEntityInterface element: methods.getElements())
        {

            if (null != element.getProperty("CYCLO"))
                continue;
            }
            sum += (Integer)element.getProperty("CYCLO").getValue();
            ++count;
        }

        return new ResultEntity((1.0 * sum)/(1.0 * count) );
    }

}
```

Figure 1.3: CodePro UML

Complexity metric we pass a class entity instead of a method entity ? The answer to both the questions is simple, if we are lucky at runtime the application will throw an exception. Basically we managed to become **dynamically typed** even if we are programming in a statically typed language.

1.2 Goals and Contributions

Our purpose for this thesis is to present a solution for the problems presented in the previous section. We want to become **statically typed** again and also be easily extensible and integratable. We shall present an innovative tool called XCorex which can solve the problems. XCore provides a flexible implementation of a metamodel. The user can provide any number of plugins (e.g. metrics implementations) that are annotated in terms of the provided meta-metamodel. XCore processes the user annotations and automatically and seamlessly produces the Java code of the metamodel and the required link between the metamodel properties (e.g., metrics) and their implementation (e.g., plugin). Finally, since these metamodel is expressed in terms of Java classes, invoking a model property will be safe (because it will be checked by the compiler).

1.3 Organization

Chapter 2. A thorough presentation of the fundamental concepts that are going to be used in this work in order to avoid any ambiguities. It will cover elements regarded eclipse plugin development, meta-metamodeling, metamodeling, modeling and how they can be implemented in Java. Also we present a state of the art analysis tool CodePro and detail the problem we are solving.

Chapter 3. The anatomy of XCore will reveal the solution we purpose for the problem. All mechanisms that allow the framework to work as intended (ease of use and extensibility) will be explained in details. Also we introduce a tool that I have implemented in order to properly evaluate the tool and the actual evaluation.

Chapter 4. In this chapter we summarize all the information described in previous chapters and present the conclusions. Also, I present the future work which will be done in the development process of the tool.

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 thesis.

2.1 Fundamental Concepts

2.1.1 Modeling, Metamodeling

Object-oriented programming is a paradigm which appeared in the early '60 [12] 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.

Modeling, as defined in the dictionary [4], 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 thesis we can see a model as a simplified version of a reality. As there can be many types of maps for the same territory depending on the

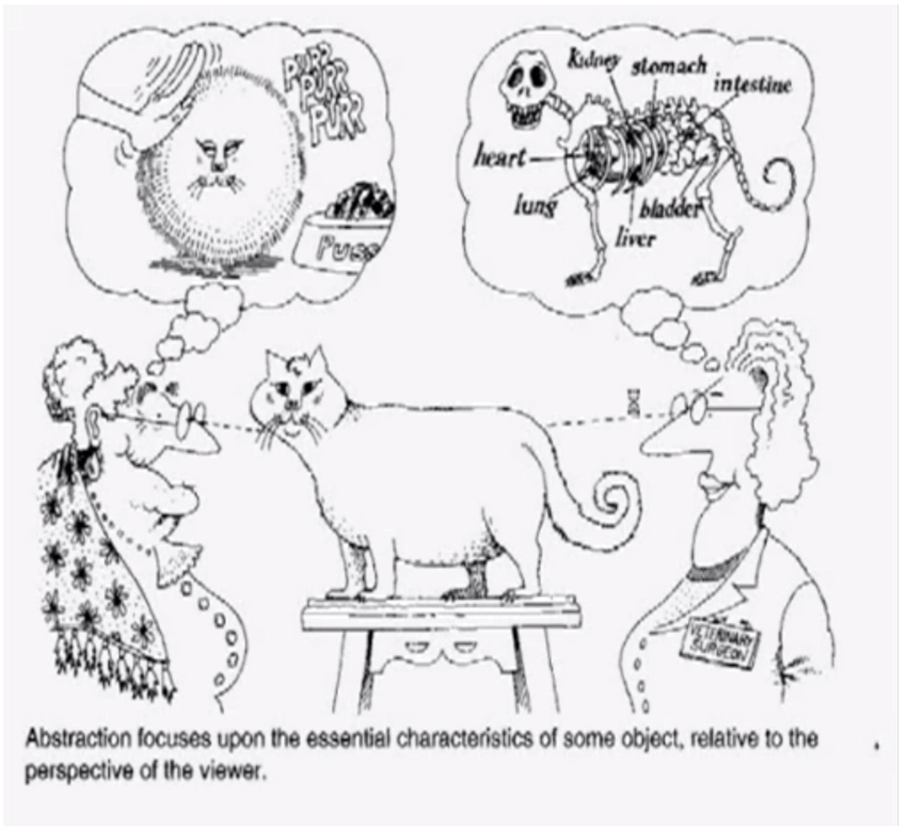

PURR PURR PURR
Puss
kidney
stomach
intestine
heart
lung
bladder
liver
VETERINARY SURGEON
Abstraction focuses upon the essential characteristics of some object, relative to the perspective of the viewer.

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

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 a way to describe models. For this the concept of metamodeling has been introduced. The prefix meta- indicates an abstraction of a concept [13]. 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 [6]. One is able to describe its object oriented system by simply showing 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.

A very common case with tools is that people want them aggre-

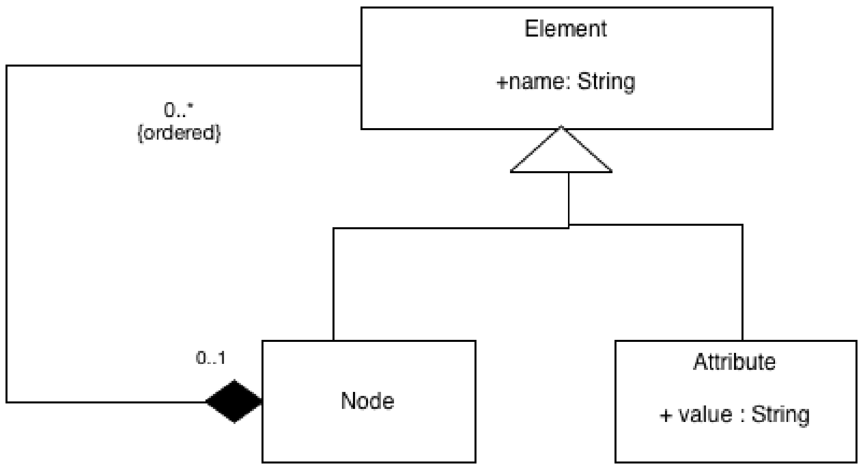


Figure 2.2: XML metamodel

gated 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 is 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[3] based on three elements:

- Entity — represents a generalization of a package, type, method, field . . . etc
- Property — represents a general characteristic computer 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.

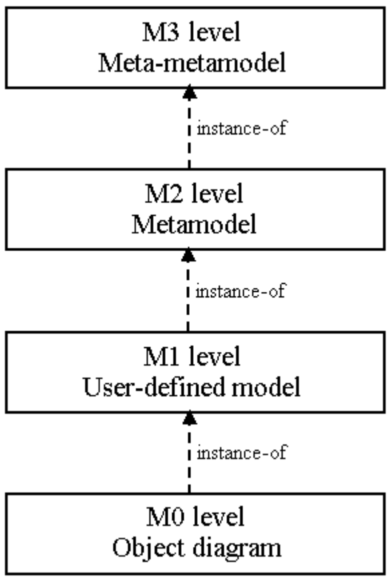


Figure 2.3: Models and Metamodels

2.1.2 Ecliplse plugin

In computing, a plugin is a software component that adds a specific feature to an existing software application. When an application supports plugins, it enables customization. The best software analogy compares a plugin to an object in object-oriented programming. A plugin, just like an object, encapsulates the behavior and/or data with which it can 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. The base libraries used by all plugins are:

The Standard Widget Toolkit (SWT) Graphical components used everywhere in Eclipse.

The Plugin Developer Environment (PDE) Classes that help us with the plugin data manipulation, extensions, build process, and wizards.

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

An Eclipse plugin has a set of main configuration files. These files define the APIs which will be used in our plugin and the dependencies to other plugins.

MANIFEST.MF - contains the OSGi configuration information.

plugin.xml - optional configuration file, contains information about Eclipse extensions.

An Eclipse plugin defines its metadata, 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. 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.

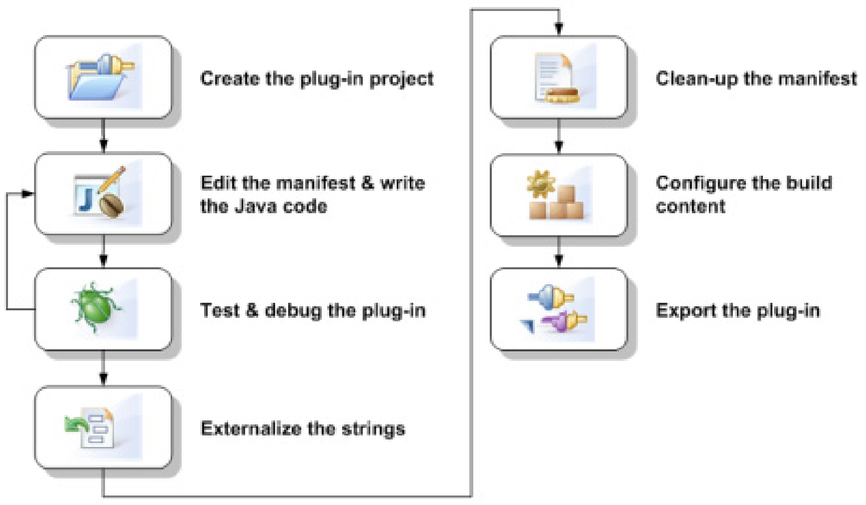

Create the plug-in project
Edit the manifest & write the Java code
Test & debug the plug-in
Externalize the strings
Clean-up the manifest
Configure the build content
Export the plug-in

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 [11].

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. [5]

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:

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

Usage:

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

```java
//the default message, will be outputted
@CreateWarning
public boolean equals(Object x) {
...
}
```

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 it provides ? (in the source file, in the class file, just drop it after compilation, . . . etc)

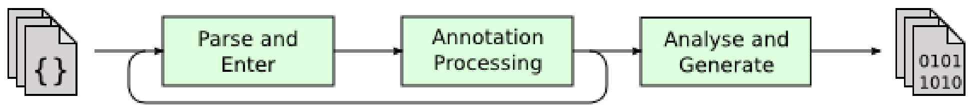


Figure 2.5: The compiler process with annotation processing[10]

- if it can be repeated more than one time on the same element
- 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 source 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 metalevel operations such as reflection by reifying program elements such as classes, methods, fields, packages, annotations ... etc, into objects that support reflective operations such as getSuperclass or getMethods. In a typical object oriented language with reflection, (e.g., Java, C#, Scala, PHP, ... etc) one might query an instance for its class, as indicated in the pseudo-code below:

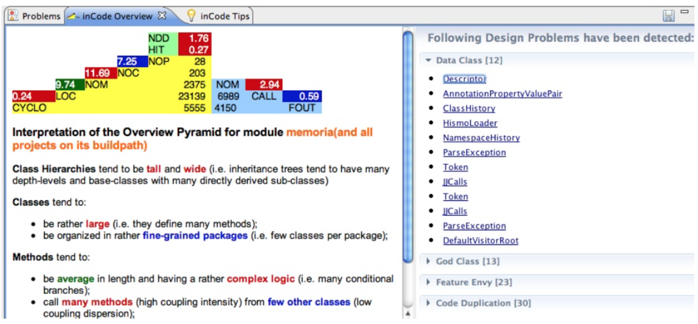
```
class TreeMap {...}
TreeMap myTreeMap = new TreeMap();
Class<TreeMap> theTreeMapClass = myTreeMap.getClass();
```

Another approach is introducing code on the fly, modifying the program as they are executed. This is used in many scripting languages and goes by the name of monkey patching or duck punching. Allows the user to locally modify the code of class or elements that was already defined.
One other approach would be the use of mirrors. Mirrors have been used in class based systems such as Strongtalk, and even in the Java world. The reflective operations are separated into distinct objects called mirrors. Reflection is no longer tied into the behavior of every object. 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 to 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 is 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. 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. [9].



Interpretation of the Overview Pyramid for module memoria(and all projects on its buildpath)

Class Hierarchies tend to be **tall** and **wide** (i.e. inheritance trees tend to have many depth-levels and base-classes with many directly derived sub-classes)

Classes tend to:

- be rather **large** (i.e. they define many methods);
- be organized in rather **fine-grained packages** (i.e. few classes per package);

Methods tend to:

- be **average** in length and having a rather **complex logic** (i.e. many conditional branches);
- call **many methods** (high coupling intensity) from **few other classes** (low coupling dispersion);

Following Design Problems have been detected:

Data Class [12]

- Descriptor
- AnnotationPropertyValuePair
- ClassHistory
- HismoLoader
- NamespaceHistory
- ParseException
- Token
- JJCalls
- Token
- JJCalls
- ParseException
- DefaultVisitorRoot

God Class [13]

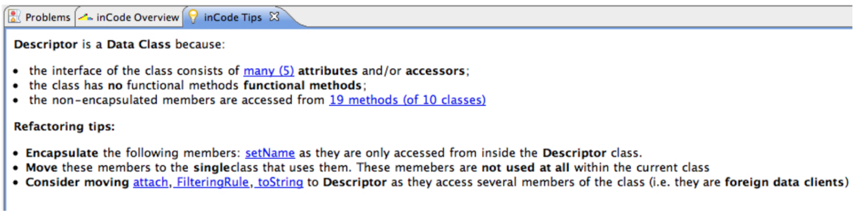
Feature Envy [23]

Code Duplication [30]

Figure 2.6: inCode overview [9]

CodePro, or by its comercial name INCODE [7], 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 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



Descriptor is a **Data Class** because:

- the interface of the class consists of many (5) **attributes** and/or **accessors**;
- the class has **no** functional methods **functional methods**;
- the non-encapsulated members are accessed from 19 methods (of 10 classes)

Refactoring tips:

- **Encapsulate** the following members: setName as they are only accessed from inside the **Descriptor** class.
- **Move** these members to the **single**class that uses them. These memebers are **not used at all** within the current class
- **Consider moving** attach, FilteringRule, toString to **Descriptor** as they access several members of the class (i.e. they are **foreign data clients**)

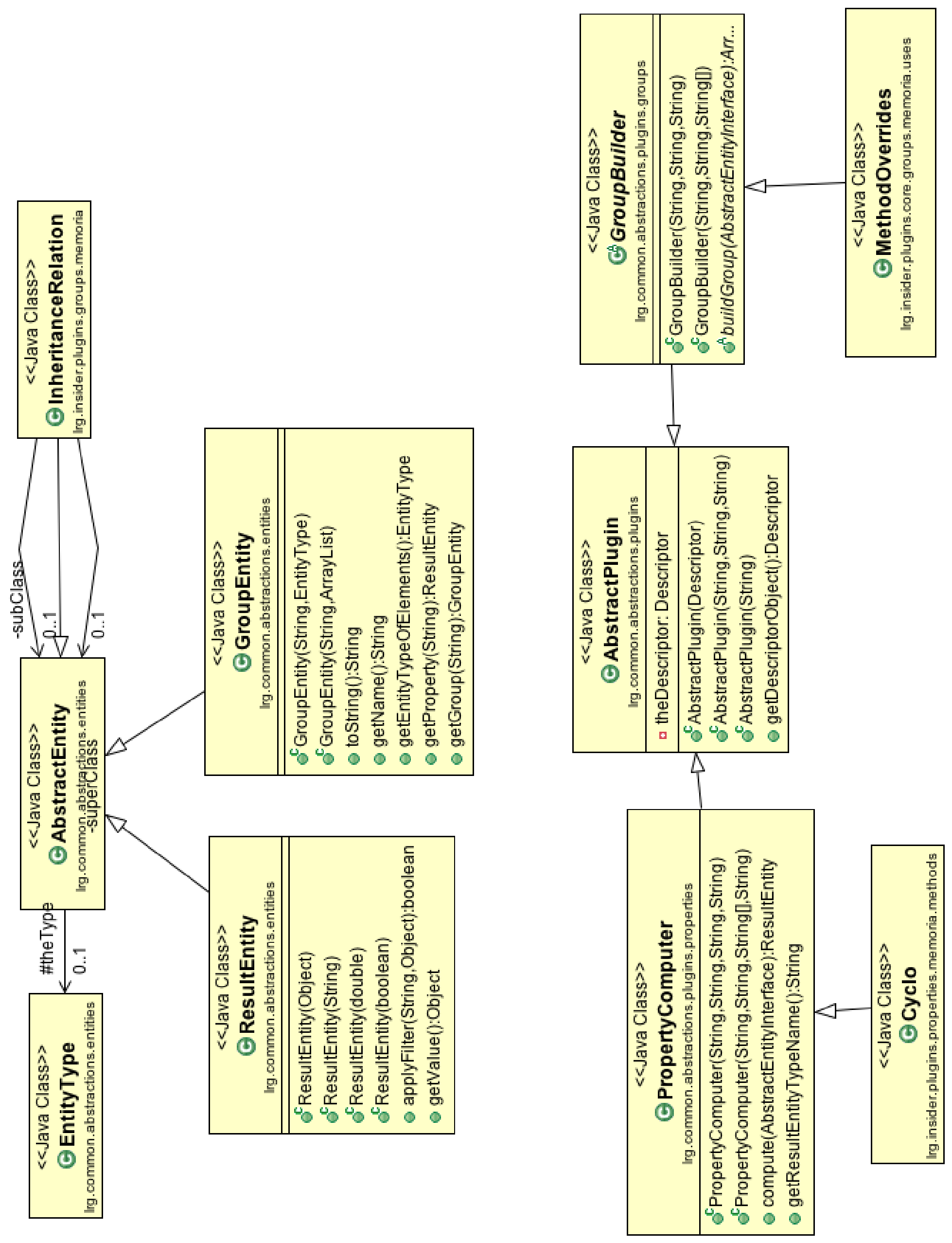
Figure 2.7: inCode tips [9]

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:

- Consider the following code snippet: `aSystem.getGroup("class group").applyFilter("model class");`. We are 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 ?

- property computers accept abstract entities instead of specific elements for which the computers are defined. The semantics 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 since there is no Cyclomnatic Complexity defined for them?
  `public ResultEntity compute(AbstractEntityInterface anEntity)`

The tool that we have created is aimed at solving this shortcoming by providing a way in which you can define a meta-metamodel and it will generate, based on the provided meta-metadata, a concrete metamodel, everything becomes statically typed.



<<Java Class>>
InheritanceRelation
lrg.insider.plugins.groups.memoria
-subClass
0..1
0..1
-superClass
<<Java Class>>
AbstractEntity
lrg.common.abstractions.entities
#theType
0..1
<<Java Class>>
EntityType
lrg.common.abstractions.entities
<<Java Class>>
GroupEntity
lrg.common.abstractions.entities
GroupEntity(String,EntityType)
GroupEntity(String,ArrayList)
toString():String
getName():String
getEntityTypeOfElements():EntityType
getProperty(String):ResultEntity
getGroup(String):GroupEntity
<<Java Class>>
ResultEntity
lrg.common.abstractions.entities
ResultEntity(Object)
ResultEntity(String)
ResultEntity(double)
ResultEntity(boolean)
applyFilter(String,Object):boolean
getValue():Object
<<Java Class>>
GroupBuilder
lrg.common.abstractions.plugins.groups
GroupBuilder(String,String,String)
GroupBuilder(String,String,String[])
buildGroup(AbstractEntityInterface):Arr...
<<Java Class>>
MethodOverrides
lrg.insider.plugins.core.groups.memoria.uses
<<Java Class>>
AbstractPlugin
lrg.common.abstractions.plugins
theDescriptor: Descriptor
AbstractPlugin(Descriptor)
AbstractPlugin(String,String,String)
AbstractPlugin(String)
getDescriptorObject():Descriptor
<<Java Class>>
PropertyComputer
lrg.common.abstractions.plugins.properties
PropertyComputer(String,String,String)
PropertyComputer(String,String,String[],String)
compute(AbstractEntityInterface):ResultEntity
getResultEntityTypeName():String
<<Java Class>>
Cyclo
lrg.insider.plugins.properties.memoria.methods

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

Chapter 3

The anatomy of XCore

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 We 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 we have also reimplemented a front-end tool called InsiderView [8] 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 XCore

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

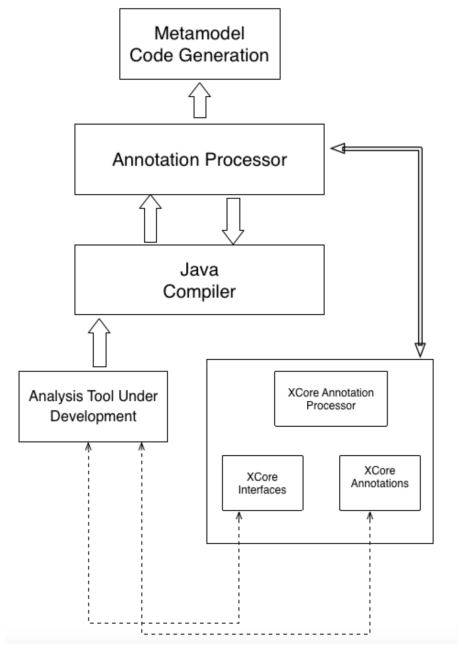

Metamodel Code Generation
Annotation Processor
Java Compiler
Analysis Tool Under Development
XCore Annotation Processor
XCore Interfaces
XCore Annotations

Figure 3.1: XCorex System overview

system:

- 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 metamodel.

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. The annotations 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.

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

```java
@Target(ElementType.TYPE)
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.

```java
public interface IGroupBuilder <ElementType extends
    XEntity,
                    Entity extends XEntity> {
        Group<ElementType> buildGroup(Entity entity);
}
```

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:

```java
@GroupBuilder
public class ListOfMethods implements
    IGroupBuilder<XMethod, XClass> {
        @Override
        public Group<XMethod> buildGroup (XClass entity) {}
}
```

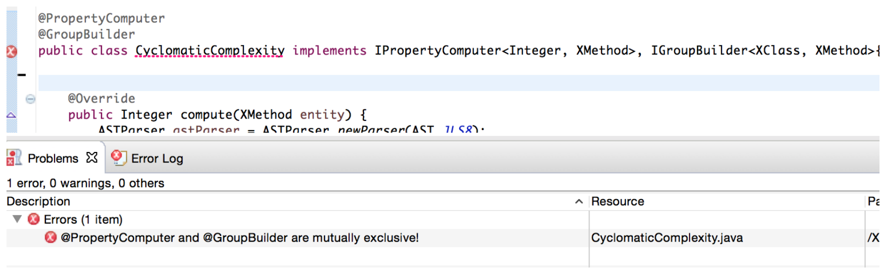
```
public interface IPropertyComputer <ReturnType, Entity
   extends XEntity> {
        ReturnType compute(Entity entity);
}
```

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



```
@PropertyComputer
@GroupBuilder
public class CyclomaticComplexity implements IPropertyComputer<Integer, XMethod>, IGroupBuilder<XClass, XMethod>{

    @Override
    public Integer compute(XMethod entity) {
        ASTParser astParser = ASTParser.newParser(AST.JLS8);
```

Problems | Error Log

1 error, 0 warnings, 0 others

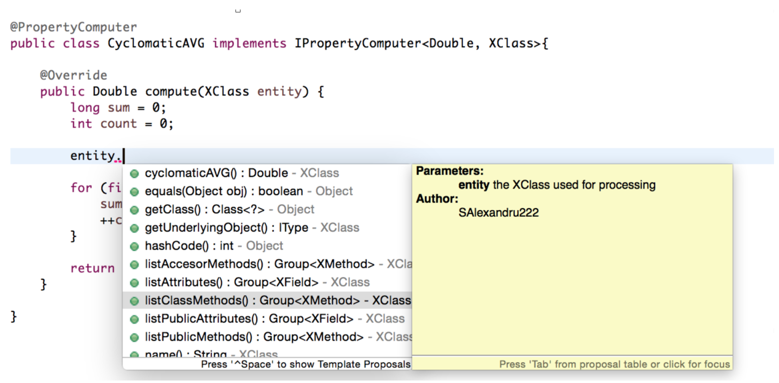
Description	Resource	Pa
Errors (1 item)		
@PropertyComputer and @GroupBuilder are mutually exclusive!	CyclomaticComplexity.java	/X

Figure 3.2: XCore Error Messages

- 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 3.2 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. The interface will contain a method for every PropertyComputer that uses this type. Also, it will contain a method for every GroupBuilder which uses this entity for generating the elements of the group. If any comments are associated with the PropertyComputers or GroupBuilders they are preserved in the generated code.
In Figure **??** we can see how easily it is for any user to find out which property computers or group builders are implemented for the entity. The intellisense does all the work. In Figure **??** we can see an implementation of metric and what code is generated for it by the tool.

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



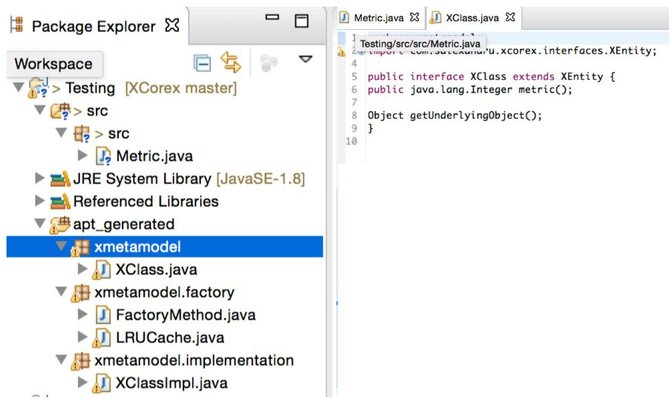
```
@PropertyComputer
public class CyclomaticAVG implements IPropertyComputer<Double, XClass>{

    @Override
    public Double compute(XClass entity) {
        long sum = 0;
        int count = 0;

        entity.

        for (fi
            sum
            ++c
        }

        return
    }

}
```

```
cyclomaticAVG() : Double - XClass
equals(Object obj) : boolean - Object
getClass() : Class<?> - Object
getUnderlyingObject() : IType - XClass
hashCode() : int - Object
listAccesorMethods() : Group<XMethod> - XCla
listAttributes() : Group<XField> - XClass
listClassMethods() : Group<XMethod> - XClass
listPublicAttributes() : Group<XField> - XClass
listPublicMethods() : Group<XMethod> - XClass
name() : String - XClass
Press '^Space' to show Template Proposals
```

Parameters:
entity the XClass used for processing
Author:
SAlexandru222

Press 'Tab' from proposal table or click for focus

Figure 3.3: XCore Intellisense



```
import com.salexandru.xcorex.interfaces.XEntity;

public interface XClass extends XEntity {
public java.lang.Integer metric();

Object getUnderlyingObject();
}
```

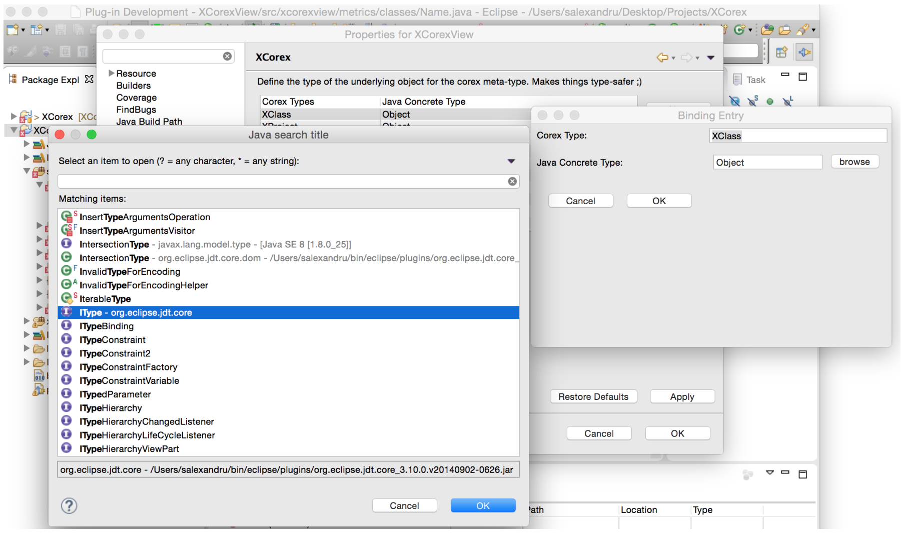
Figure 3.4: XCore Type Specification

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 **??** the method will return the specified type. The menu can be easily access by right on the Java project -> XCore.

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

3.2 Evaluation

In order to properly evaluate the framework usability and also to discover additional improvements I have reimplemented the tool In-


Plug-in Development - XCorexView/src/xcorexview/metrics/classes/Name.java - Eclipse - /Users/salexandru/Desktop/Projects/XCorex

Properties for XCorexView

XCorex

Define the type of the underlying object for the corex meta-type. Makes things type-safer ;)

Corex Types	Java Concrete Type
XClass	Object

Java search title

Select an item to open (? = any character, * = any string):

Matching items:

```
InsertTypeArgumentsOperation
InsertTypeArgumentsVisitor
IntersectionType - javax.lang.model.type - [Java SE 8 [1.8.0_25]]
IntersectionType - org.eclipse.jdt.core.dom - /Users/salexandru/bin/eclipse/plugins/org.eclipse.jdt.core_
InvalidTypeForEncoding
InvalidTypeForEncodingHelper
IterableType
IType - org.eclipse.jdt.core
ITypeBinding
ITypeConstraint
ITypeConstraint2
ITypeConstraintFactory
ITypeConstraintVariable
ITypedParameter
ITypeHierarchy
ITypeHierarchyChangedListener
ITypeHierarchyLifeCycleListener
ITypeHierarchyViewPart
```

org.eclipse.jdt.core - /Users/salexandru/bin/eclipse/plugins/org.eclipse.jdt.core_3.10.0.v20140902-0626.jar

Binding Entry

Corex Type: XClass

Java Concrete Type: Object

siderView by using the XCore framework. InsiderView is an Eclipse plugin of CodePro. It makes available all the metric system of CodePro to Eclipse by using a view in which the entities and their properties will be shown. You can operate on the view by expanding a group defined by the entity or simple reveling the a new property for the current entities. This entire structure was preserved in the implementation of XCoreView.

The properties, metrics, implemented by XCoreView are limited and have a direct equivalent in CodePro. Some of this metrics are:

Name Each entity has a computer which returns its name

Cyclomatic Complexity (Cyclo, CC) This is a metric for methods which indicates the linearly independent paths

Number of Children (NOC) Measures the number of direct descendants of a class.

Lines of Code (LOC) Measure the number of written code lines of a given entity (class, method)

Weight of Class (WOC) Identifies if a class is a data class.

We have implemented 13 different metrics. Each metric was compared based on the number of lines written, the cyclomatic complexity of the compute method, the number of casts used and the number of magic strings used. All of this values can be seen in table **??**.

The aggregated values of the evaluation can be seen in 3.1. It is often the case with framework that alot of boiler plate code must be written in order to properly configure them. In this case this is not true. In green you can see that the number of lines has remained the same, showing that no boiler plate code is needed.

One of the most important things is that the overall complexity of the code has been halved when using XCore. This means that the code is now easier to understand and maintain. Another confirmation of this is that number of casts and magic strings has been reduced to zero. Our goal for getting rid of this elements and having statically typed plugins has been reached.

Table 3.1: Aggregated values of the comparison. The relevant values have been highlighted

	Number of Lines	Cyclomatic Complexity	Number of Casts	Number of Magic Strings
Avg CodePro	34.153	4.307	1.796	2.230
Avg XCoreView	31.384	2.384	0	0
Sum CodePro	444	56	23	29
Sum XCoreView	408	31	0	0

Table 3.2: 13 different metrics from CodePro and XCoreView compared

Framework Name	Metric Name	Number of Lines of Code	Number of Casts	Cyclomatic Complexity	Number of Magic Strings
Code Pro	Cyclomatic Complexity	83	2	4	1
XCoreView		110	0	1	0
CodePro	LOC	44	2	5	1
XCoreView		29	0	3	0
CodePro	Name	12	0	1	1
XCoreView		8	0	1	0
CodePro	IsConstructor	27	2	2	1
XCoreView		15	0	1	0
CodePro	IsAccessor	43	3	12	6
XCoreView		19	0	4	0
CodePro	WOC	19	0	2	6
XCoreView		13	0	2	0
CodePro	NOC	9	0	1	2
XCoreView		32	0	1	0
CodePro	Cyclomatic Avg	32	3	4	3
XCoreView		17	0	2	0
CodePro	Method Group	36	3	4	2
XCoreView		23	0	2	0
CodePro	Public Method Group	15	0	1	2
XcoreView		19	0	3	0
CodePro	Attributes Group	39	3	4	2
XCoreView		19	0	3	0
CodePro	Classes Group	65	3	13	2
XCoreView		25	0	5	0
CodePro	Package Group	35	2	4	2
XCoreView		79	0	3	0

Chapter 4

Conclusion

One worthwhile task carried to a
successful conclusion is better than 50
half-finished tasks

B.C.Forbes

4.1 Conclusion

As stated in the beginning of the work, we wanted to ease and improve the way analysis tools are developed and integrated in IDEs. Currently they are mostly developed by creating a complex metamodel which is able to analyze the model which we want to evaluate. This procedure can cause major problems when we try to extend the platform with different tools because we have to merge different metamodels (architectures) which also have different semantics. Another problem is the procedure itself, it is repetitive, thus it can, as we have seen in the case of CodePro, it actually did, cause design and implementation problems.

The main problem with CodePro is the lack of static type safety due to the need of generalization and ease of extensibility. Another problem is the manipulation of magic constants in order to identify and describe the plugins that are created based on the framework.

Our solution for this problems is to implement a meta-metamodel, a model which will allow us to describe an analysis tool (metamodel), thus giving us the possibility to easily extend the platform. As we have seen the meta-metamodel is implemented by using specific Java metadata programming, annotations, and we enforce their semantics be defining an annotation processor. Also, the annotation processor has the important role of generating the appropriate code for the model based on the user specifications. Because eveyrthing is done at compiletime, and not at runtime, the compiler can enforce type safety and IDEs can provide nice intellisense which will ease us in the development process.

For evaluation we have reimplemented InsiderView, an eclipse plugin that exposes CodePro to eclipse users. Table 3.1 shows that our tools has helped in removing all the magic strings and casts from the code making it statically typed. It also shows that the complexity of the code has decreased, making the code easier to understand and maintain.

4.2 Future Work

In the future we would like to extend the framework to be integrate with the incremental build systems that are present in major IDEs such as Eclipse and IntelliJ. Currently when using the tool and have added or removed entities that are annotated with elements from the framework you will need to rebuild the entire projects. This is due to the lack of partial information provided by the compiler when incremental builds are used.

Major improvements would be the possibility of integrating multiple models in the same tool. For example we may want to design a software tool that uses some model as JDT, but we would like to take advantage of the Wallace model also (because we want to provider a larger acceptance or optimize our application).
Another improvement would be the possibility of sandboxing new add-ons. When we develop a tool we usually develop a core, formed from a series of elementary elements which rarely are changed, and add as we need different plugins. It will be great for the user if he can develop the new plugin in isolation with respect to other developers so it does not interfere with them and also if it does not affect the core.

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] Cristian Caloghera-Ianeș. Evolutionary integrated analysis environment for software systems, 2004.

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

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

[6] 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.

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

[8] 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.

[9] 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.

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

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

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

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