Faculty of Automation and Computers

Bachelor Program: Calculatoare si Tehnologia Informatiei

Universitatea Politehnica Timişoara

# XCore: Framework for Software Analysis Tool Development

**Master 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

One of the goals of any software developer is to write quality code. Code that respects company policy, that has a large test coverage percentange, that uses well defined mechanism thus making it portable on different operating system . . . etc. In order to establish the quality of encode and enforce different standards analysis tools where build. Some of the most used tools are:

**NullTerminator** It represents a pseudo-automatic refactoring tool which eliminates the null checks by use of the **Null Object Design Pattern** [9]

**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

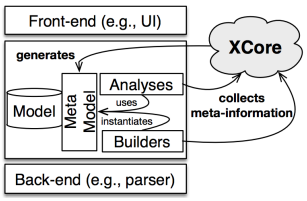


Figure 1.1: Generic Backend for analysis tools [10]

**CodePro** A platform which implements software metrics developed by LOOSE Research Group. Goes by the name of INCODE, also [5]

Most analysis tools share a generic back-end which can bee seen in 1.1. The back-end is responseble for parsing the code in the desired projects and extracting a low-level abstractions / artefacts of the code (e.g the AST of the code). The extracted low-level artefacts, though usefull, are too granular for most analysis tools, it needs to be processed furthere, resulting in high-level abstractions / artefacts called a **model**. This model is structured according to a **meta-model** defined and implemented by the developers of the tool. [10]

The construction of the **meta-model** is a recuring task when it comes to building analysis tools. **IPlasma** [8] uses the **Memoria** meta-model in order to compute metrics. Wala [11] builds its own meta-model in order to preform analysis. Eclipse, as Wala, provides its own meta-model called JDT [12] which is used by most of the Eclipse analysis plugins.

Building such a meta-model and the necessary back-end is a tedious task, at best. Another major problem that arises when building such a meta-model, and any other program for that matter, is the evolution in time. We want a meta-mode that can be easily extensible over time, but also have the ability to integrate / inter-operate with other tools / meta-models in order to obtain complex tools. All of these requirements are difficult to design and implement and, different approaches have been made propoused.[2] [10] In order to obtain the mentioned goals, some tools such as inCode[5], have introduced architectural faults which compromised the type safety of the program. This ultimately lead to poor code which is hard to maintain and understa. [1]

## 1.2 Previous Work

Previously we have build an Ecliplse plugin, called XCore which solved the previously mentioned problems. [1]. The tool works by providing the software developers with the means to describe the meta-model of their analysis tool directly into the source code of the tool, without the need to actually implement the meta-model. This is done by providing a higher level of abstraction called meta-meta-models. The meta-meta-model is implemented by using the java annotations system.[1] [3] When the tool is compiled, the java compiler will notice the annotations and invoke the XCore annotation process before actually processing the entire project. Our annotation process will generate, based on the information provided, the appropriate meta-model for the tool and will enforce all the necessary restrictions on meta-model entities. This is perfectly type safe because after the code generation phase is over the java compiler can fully typecheck the entire system !

## 1.3 Current Development

XCore provides a way to describe the necessary meta-model for the analysis tool. It also provides a way to integrated a back-end for the extraction of the "meta-model" from the code. (e.g it can integrate with the eclipse jdt for java meta-mode or eclipse cdt for cpp meta-model). In the previous version you could integrate with only one such back-end, but now we have modified the implementation such that you can integrate with multiple back-ends. Another major feature that we implemented is that one can extend another analysis tool implemented using XCore. Also, we have extended the meta-meta-model description with the concept of Action. One can now describe actions that can be executed upon meta-model entitites. This integrates nicely with the UI frontend where we could describe an action for opening the analyzed class / method entity in the editor, or concretly pinpoint the design / implementation error in the project.

## 1.4 Related Work

In essence XCore is a tool for meta-programming. It contains meta-meta-model which allows the user to describe any meta-model respecting the constraints. Thus is bears some similarity with other tools which try to solve the problems metioned above, like it such as:

**Rascal** [6]

**Soot** [7]

**Fame** [2]

Rascal is a meta-model programming language. It allows a user to express any meta-model for the development / integration of analysis tools. To the best of my knowledge there is no way to interact wit other concrete meta-models like those from Wala and Jdt. This is one it's major disadvantage, another being the time and resources need to fully learn the language before using it. Also, at time of writing, there is also the problem of running time. According to [6] the runtime environment is pretty slow.

Soot is java framework developed for code analysis. It directly analyzes jvm bytecode and can represent in several low level representations. Unlinke XCore, which provides support for high analysis tools, the goal of the framework is to provide the necessary tools in order to develope optimization for applications. Also, to the best of my understanding Soot is limited to the analysis of java byte code, whereas XCore can integrate with and language library backend, provided that the library is written in Java (or C and which can be integrated in Java applications).

Fame is similar in scope and idea of development with XCore. It uses the FM3 meta-meta-mode [2] and which is handeled by the JavaFame Tool. The meta-model for the developed tool is described in a msfe in file. By contract, XCore allows de developers to describe the meta-mode directly in the java source by use of annotations. The main disadvantage in using external files in order to provide configurations is the lack of support for refactorings (i.e easy change of the meta-model).

# Chapter 2

# Xcore Approach

> 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 mentioned in the above chapters we have implemented a tool called XCore. As we shall see in the following sections, our tools uses annotations and interfaces in order for developers to express their meta-model for the analysis tools in a type safe manaer. Based on the expressed meta-model we generate the appropriate code for the meta-model integration. 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.

## 2.1 Solution Overview

In figure 2.1 we can see the basic architecture of the XCore system. There are three main components to this system:

- Annotation Component
- Interface Component
- Annotation Processor

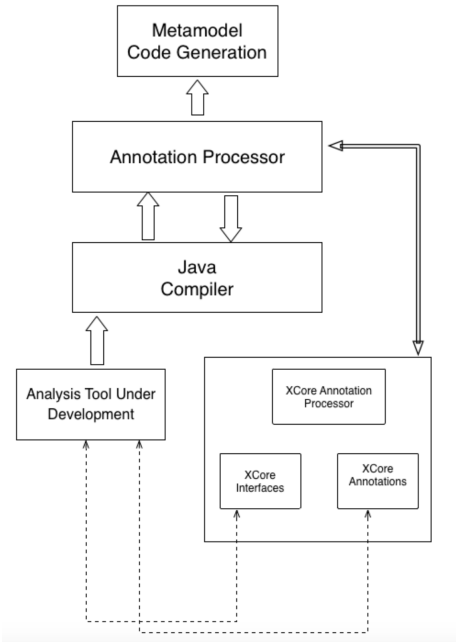


Figure 2.1: XCorex System overview [1]

### 2.1.1 Annotation Component

The annotation component, as you might have guessed, provides the necessary annotations in order to describe the meta-model. Annotations are used as markers to identify the components of the analysis tool and also to distinguish between diferent types of entities and thier relationships as described below. This is only half the story, we also use interfaces, described in the section below 2.1.2, to insure the type safety of the meta-model, and uniformity between similar components.

As it can be seen from the code below, the Java annotations are allowed only on concrete types, i.e classes.

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

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

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

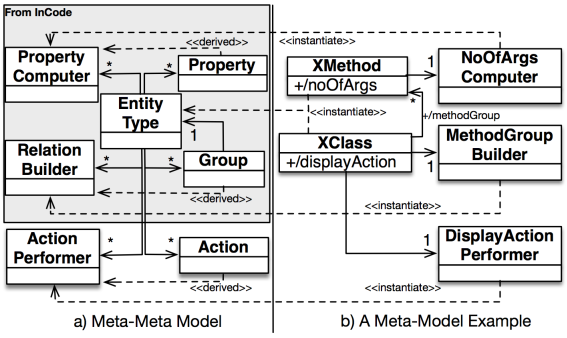


Figure 2.2: XCore meta model [10]

In the figure below 2.2 we can see the design of the meta-meta-model in the left part, which is based on the conceptual meta-meta-model described by Ganea et al [4]. In the right part we can see an example of a meta-model that can be described (ie. an instantiation of the meta-meta-model).

XMethod and XClass represent *entities* of the meta-model which describe methods and classes. Any entity has certain *properties* which represent result of analysis. For example *NoOfArgsComputer* represents a *property* associated to an *entity*, in this case the XMethod *entity*. Besides properties, we can express an relationship between entities. In our case, a we have a inclusion relationship, an XClass includes multiple XMethod elements. This can be easily seen that is represented by the *MethodGroupBuilder*. Another concept that can be expressed is the one of *action*, such as the *DisplayActionPreformer*

### 2.1.2 Interface Component

When the annotations are processed by the annotation processor, which is presented in chapter 2.1.3, it will enforce that every concrete type will implement the appropriate interface. Thus, if the concrete type represents a *property* it must implement a *IPropertyComputer* interface, a *relationship* is expressed by implementing a *IRelationshipBuilder* interface and an action requires the *IActionPreformer* interface.

As we have seen in the section above, a property represents an analysis which is done on an *entity* such as XMethod or XClass from figure 2.2. The code below, from section 2.1.2, show how the interface is defined. The code should be self explanatory. The

XEntity from the code represent a marker interface for the meta-model entities. The *ReturnType* can be anything Double, Pair of elements, List of elements, ... etc.

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

In order to express an aggregation relationship between two entities one needs to use the *IRelationshipBuilder* interface 2.1.2. The ElementType *entity* represent the aggregated type and the Entity type represents the aggregator.

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

The last interface that can be used is the *IActionPreformer*. As it can be seen from the code 2.1.2, the interface is similar to the ones presented above. The only major difference is the third type parameter, the ArgTypeList, which is an HList, i.e heterogeneous list. A heterogenous list is a an arbitrary-length tuple. This means that we can store, in a type safe maner, any numbers of objects, just like in an ordinary list, but the objects don't need to be of the same type !

Our goal is to provide a typesafe maner for building analysis tools. Using an Object array as parameters for the action, though syntacticly convinient, it is by far a safe manaer of handling such data !

```
1  public interface IActionPreformer <ReturnType, Entity
       extends XEntity, ArgTypeList extends IHList> {
2    ReturnType performAction(Entity entity, ArgTypeList
       args);
3  }
4
5  interface IHList {}
6  public final class HListEmpty implements IHList { }
       //singleton element
7  final class HList<HeadType,TailType extends IHList>
       implements IHList {
8    private final HeadType head;
9    private final TailType tail;
```

```
10
11    public HList(HeadType h, TailType t) {
12      head = h;
13      tail = t;
14    }
15
16    public HeadType head() {
17      return head;
18    }
19
20    public TailType tail() {
21      return tail;
22    }
23  }
```

### 2.1.3 Annotation Processor

In the above sections we presented the syntactic constructs which were employed for the description of the meta-meta-model. But as you have noticed all this construct have additional semantics, rule of interation, associated to them. In order for them to be fully enforced we have build a compiler plugin, called an **annotation processor**. The annotation processor will process only the java elements that we are interested in, i.e the java concrete classes annotated with the annotations presented in 2.1.1. The process of discovering the annotated java element elements is the concer of the java compiler. As we are a compiler plugin it will invoke us when such elements are found. After all the "interesting" elements where processed, the compilation of the project can begin.

Upon processing the elements it will try to see if they respect the following rules [1]:

- 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 @RelationshipBuilder must implement IRelationshipBuilder

- All elements annotated with @ActionPreformer must implement IActionPreformer
- All classes cannot be present in the default package
- The @PropertyComputer, IActionPreformer and @RelationshipBuilder annotations are mutually exclusive
- No wildcard types are allowed to be specified for the entity type parameter.

From the known interfaces, IPropertyComputer, IRelationshipBuilder and IActionPreformer, we extract the the meta-model type (i.e the type parameter which extends the XEntity marker interface) that is used in that analysis component. For each of the different meta-model types we generate an interface in which every IPropertyComputer / IRelationshipBuilder / IActionPreformer correspond to a method in the appropriate interface. Of course, to be of any use, the interfaces are fully implemented. The methods from each interface are very easy to implement, they instantiate the concrete "*Computer" element and call the appropriate method (i.e this is the reason we used interfaces, we have a simple way of guaranteeing the each for this classes have the necessary methods, with the correct definitions). The end user has no access to this classes, one can instantiate an entity type (such as XMethod or XClass from 2.2) by using our provided *Factory*. The factory also includes a cache which prevents us from double instantiation of the entities. (this is mostly for performence concers).

"In Figure 2.3 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 2.4 we can see an implementation of metric and what code is generated for it by the tool." [1]

From 2.4 you should see that the XClass entity, and for that matter any entity from XCore, has an extra method
`Object getUnderlyingObject`.

As metioned in 1 an analysis tool has, at it's core, a back-end which converts the raw project files into meta-model entities. As we have seen in this chapter XCore provides us with way to describe the meta-mode, but no way of directly extracting it from the raw project files. For this we rely, as most analysis tools actually do, on external back-end. **We don't rely on a specific such backend**.

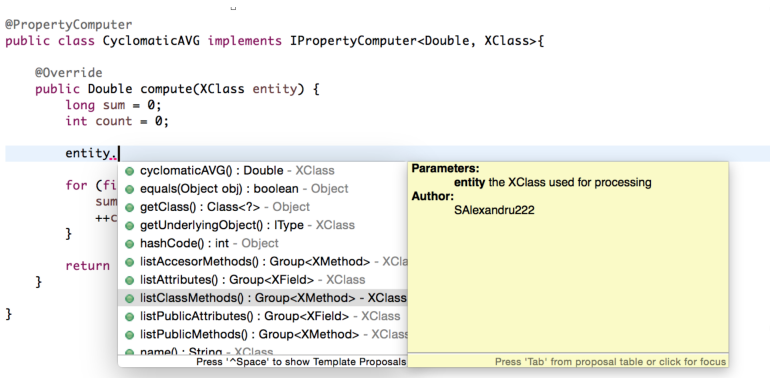


Figure 2.3: XCore Intellisense [1]

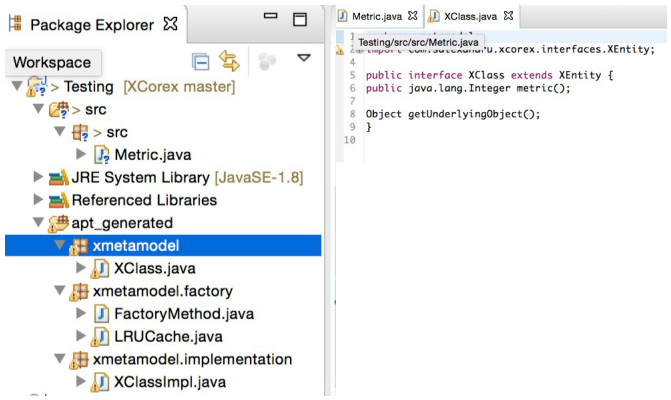


Figure 2.4: XCore Type Specification [1]

We offer the ability of the developer to choose one, or more such backends and map the back-end entities to our meta-model. This can be done through the projects configuration page, where we provided a nice type as you go search facility for all the possible types in the project as you can see in figure 2.5

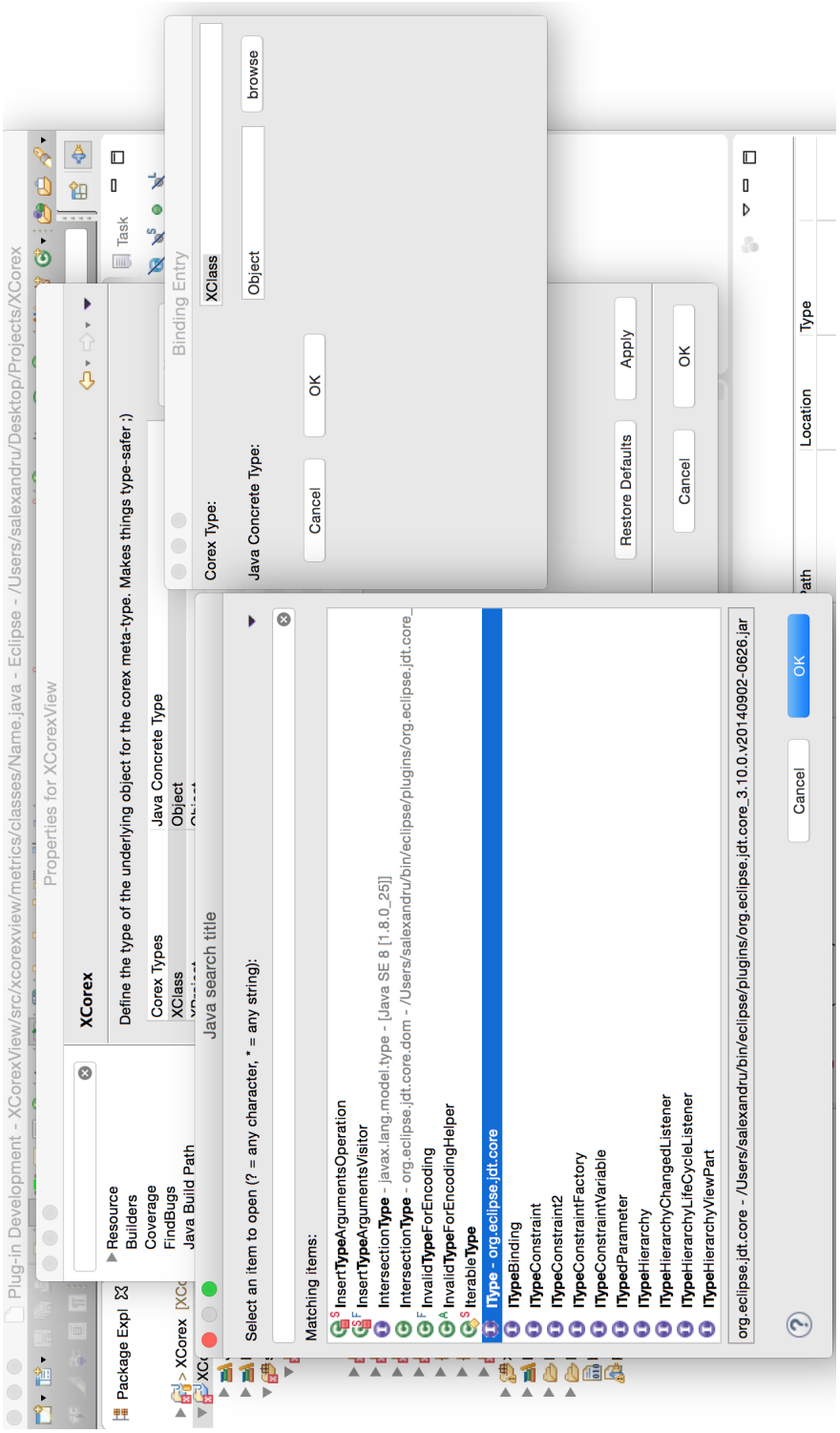


Figure 2.5: XCore Type Dialog Box [1]

# Chapter 3

# Building a Tool with XCore

> Software is like entropy. It is difficult to grasp, weighs nothing, and obeys the second law of thermodynamics; i.e. it always increases.
>
> Norman Ralph Augustine

XCore is used for building, in a type safe manaer, analysis tools. Why it is important has been presented in the chapters above.

In the current chapter we will build an analysis tool which is capable of:

1. computing the cyclomatic complexity of a method
2. displaying the method in the editor
3. computing how many calls are done to the function using a call graph.

## 3.1 Createing the Project

Before we start we need to download XCore and install it into Eclipse. The instalation is very easy, simply clone the XCore repository or download it directly as a zip from here. After you have downloaded the repository go to the XCore/deploy/plugings folder and copy ther XCore jar into the Eclipse instalation folder called *dropins*. At the time of writing this document we are at version 1.2.0 and the jar is called *XCore_1.2.0.jar*.

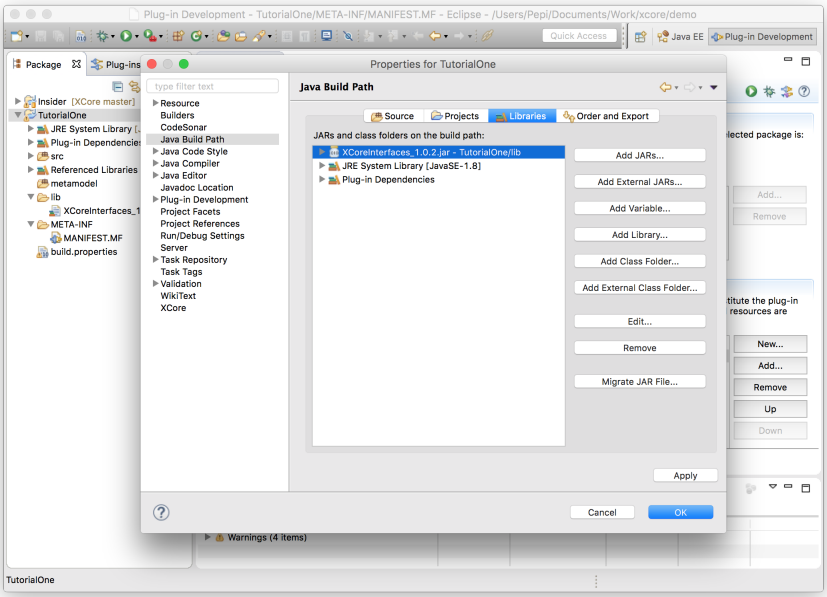


Figure 3.1: Adding the jar to the dependecy

For the scope of this article please create a new Eclipse workspace and import from the XCore repository the *Util/XCoreView* (a.k.a Insider View) project. This is a nice ui we have build in order to test tools build with XCore.

Now, in the new workspace, create a Plug-in Project. In this project create a lib folder and copy the jar that you find in *XCore/deploy/lib/XCoreInterface_<version>.jar*. Current version for this jar is 1.1.0. After this we have to add the library to the dependecy path using *Project Properties / Java Build Path / Libraries* as can be seen in figure 3.1. We also have to make the jar available at runtime from *MANIFEST.MF/Runtime/Classpath* as can be seen in figure 3.2.

After this we should enable the annotation processor. This is done by accessing *Project Properties/Java Compiler/Annotation Processing/Factory Path* and ticking the *Enable Specific Settings*. Also, for clarity we can make the generate content easily accessable by modifing the path from *Project Properties/Java Compiler/Annotation Processing*. Each step is ilustrated in figures 3.3 and 3.4.

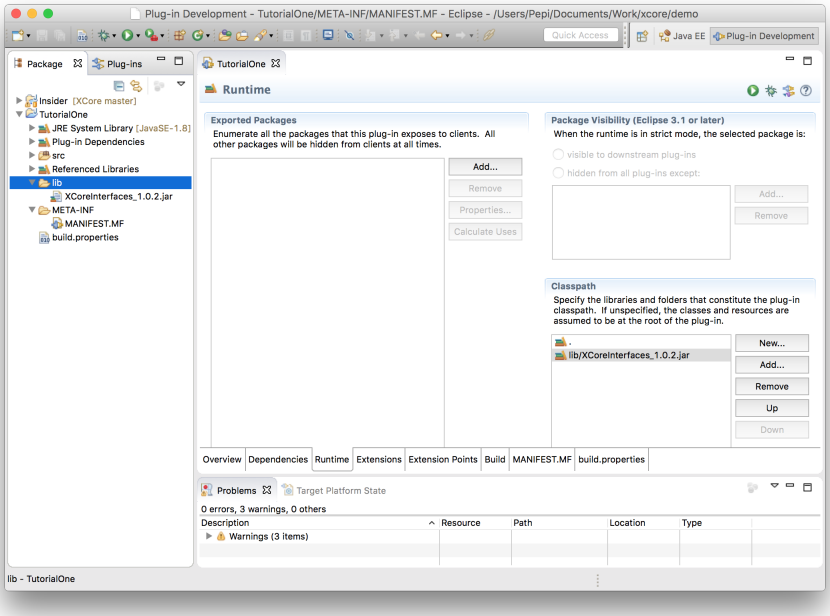


Figure 3.2: Making the jar available at runtime

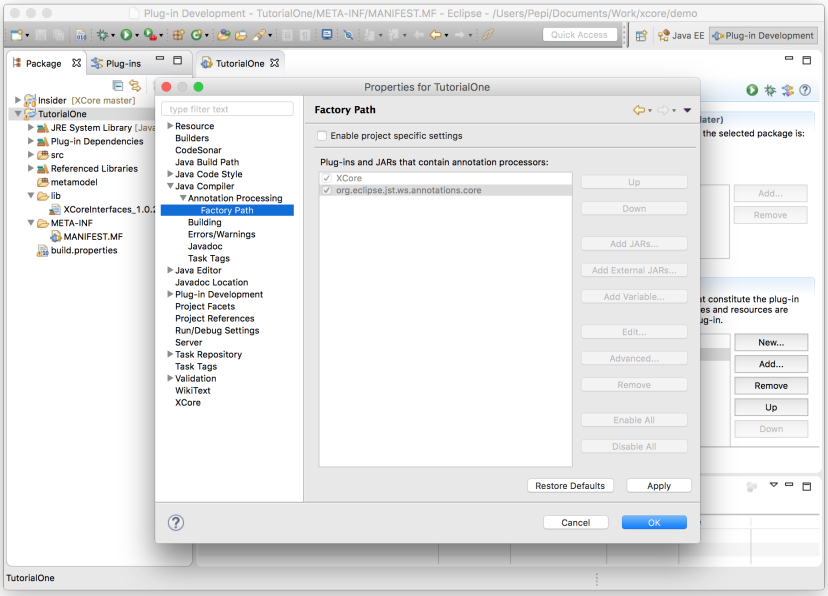


Figure 3.3: Enable XCore Annotation Processor

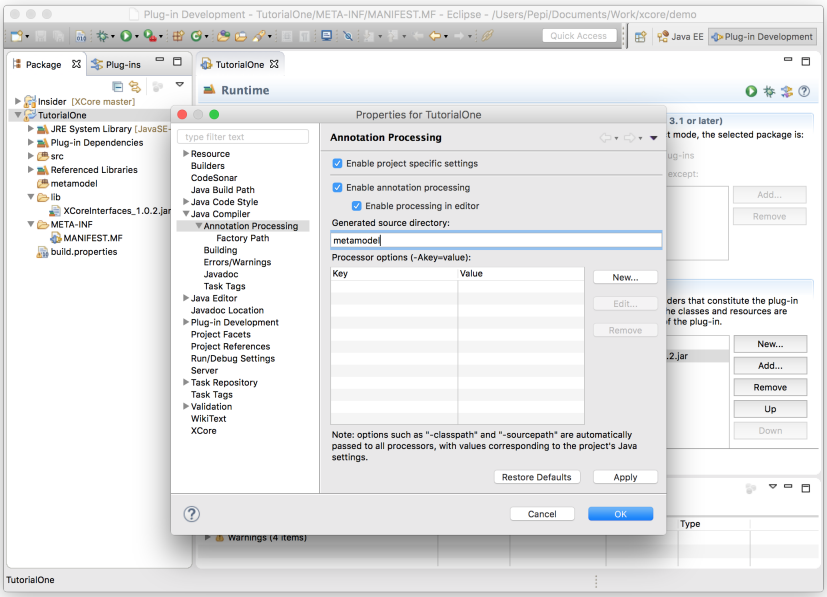


Figure 3.4: Change Code Generation Directory

## 3.2 Building the Analysises

After setting-up the project we continue implementing the analysis mentioned above. We will start implementing the simplest possible tool, which is the displaying the methond in the editor. We will also show the new feature that have been developed.

### 3.2.1 Displaying a method in the editor

First create a class called `OpenInEditorAction`. The code for this class is presented in the section 3.2.1. After reading chapter 2 the code should be quite easy to read. We have defined an action class, this can be easily seen by presence of the annotation `@ActionPreformer` and its coressponding interface `IActionPerformer<Void, XClass, HListEmpty>`.

This feature of adding actions on entities has been introduced quite recently into the project. When you first implement this class, you will most likely see that XClass cannot be imported because it doesn't exist yet. It doesn't exist because we haven run the annotation processor which will generate the code for the XClass entity.

To invoke the processor, simply build the entire project. Once you have rebuild you can go to the source directory set in the first section of this chapter and inspect the code.

Please note that you should still get a compilation error on the line where you call the methond `getUnderlyingObject()`. This is because we haven't associated any back-end to our tool. For that please se the subsection 3.2.3

```
1  import org.eclipse.jdt.core.JavaModelException;
2  import org.eclipse.jdt.ui.JavaUI;
3  import org.eclipse.ui.PartInitException;
4
5  import com.salexandru.xcore.utils.interfaces.HListEmpty;
6  import
       com.salexandru.xcore.utils.interfaces.IActionPerformer;
7  import
       com.salexandru.xcore.utils.metaAnnotation.ActionPerformer;
8
9  import exampletool.metamodel.entity.XClass;
10
11 @ActionPerformer
12 public class OpenInEditorAction implements
       IActionPerformer<Void, XClass, HListEmpty> {
13
14  @Override
15  public Void performAction(XClass entity, HListEmpty args)
       {
16  try {
17     JavaUI.openInEditor(entity.getUnderlyingObject(),
          true, true);
18  } catch (Exception e) {
19     e.printStackTrace();
20  }
21 return null;
22 }
23
24 }
```

### 3.2.2 Cyclomatic Complexity

In this section we will continue to enrich our small analysis tool by adding a simple metric which is applied onto methods.
A property, as explained in the sections above, is nothing more that

an analysis / processing that returns a value. The value isn't restricted to any specific type, it can be anything!
The metric that we will try to build is called the *Cyclomatic Complexity* of a method. It requires us to compute the number of different paths that can be executed within the method. Take for example the method presented in 3.2.2. The cyclomatic complexity is 3. We have one path then goes through the *then* branch, one that goes through the *else* branch and doesn't enter the while loop and one that goes through the *else* branch and enters the while loop.

```
1  int method (int x) {
2     if (x % 2) return 1;
3     else {
4         while (x % 2 == 0) x /= 2;
5         return x;
6     }
7  }
```

The code for implementing it cyclomatic complexity property can be found below in 3.2.2. It should be quite strait forward to read. We take the AST of a the methond in question and for each statement: `if`, `else case`, `default`, `for`, `while`, `do {} while`, `try`, `catch`, `finally` we increase the cyclomatic complexity.

```
1  @PropertyComputer
2  public class CyclomaticComplexity implements
      IPropertyComputer<Integer, XMethod> {
3  @Override
4  public Integer compute(XMethod entity) {
5  ASTParser astParser = ASTParser.newParser(AST.JLS8);
6
7  astParser.setSource(entity.getUnderlyingObject().getCompilationUnit());
8
9
10 NodeVisitor visitor = new NodeVisitor(entity.toString());
11 astParser.createAST(null).accept(visitor);
12
13 return visitor.getCount();
14 }
15
16 private static final class NodeVisitor extends ASTVisitor {
17 private int count = 0;
18 private final String methodName;
19
```

```
20 public NodeVisitor(final String methodName) {
21     this.methodName = methodName;
22 }
23
24 @Override
25 public boolean visit(MethodDeclaration d) {
26     return d.getName().toString().equals(methodName);
27 }
28
29 @Override
30 public boolean visit(IfStatement node) {
31     ++count;
32     if (null != node.getElseStatement()) {
33         ++count;
34     }
35     return true;
36 }
37
38 @Override
39 public boolean visit(ForStatement node) {
40     ++count;
41     return true;
42 }
43
44 @Override
45 public boolean visit(SwitchCase node) {
46     ++count;
47     return true;
48 }
49
50 @Override
51 public boolean visit(ConditionalExpression node) {
52     ++count;
53     if (null != node.getElseExpression()) {
54         ++count;
55     }
56     return true;
57 }
58
59 @Override
60 public boolean visit(InfixExpression node) {
61     InfixExpression.Operator type = node.getOperator();
62
```

```
63      if (InfixExpression.Operator.CONDITIONAL_AND == type ||
64          InfixExpression.Operator.CONDITIONAL_OR == type) {
65          ++count;
66      }
67
68      return true;
69  }
70
71  @Override
72  public boolean visit(WhileStatement node) {
73      ++count;
74      return true;
75  }
76
77  @Override
78  public boolean visit(DoStatement node) {
79      ++count;
80      return true;
81  }
82
83  @Override
84  public boolean visit(CatchClause node) {
85      ++count;
86      return true;
87  }
88
89  @Override
90  public boolean visit(ThrowStatement node) {
91      ++count;
92      return true;
93  }
94
95  @Override
96  public boolean visit(BreakStatement node) {
97      ++count;
98      return true;
99  }
100
101 @Override
102 public boolean visit(ContinueStatement node) {
103     ++count;
104     return true;
105 }
```

```
106
107 public int getCount() {return count + 1;}
108 }
109 }
```

### 3.2.3 Associate A Backend

Currently we have implemented a property and an action. Both shouldn't compile as we have not associated any back-end. As mentioned in the previous chapters XCore is dealing with high-level entities, we do not provide a back-end for extracting the entities from the source code. But we do provide a way of integrated the tools you build with a real back-end.

To do this, simply go to *Project Properties/XCore* and you will see a table just like in figure 3.5 , just look at the first one for now ! Select XMethod hit the edit button. You should see a search bar where you should be able to write the corresponding the entity name from JDT, it's *IMethod*, see figure 3.6 and hit enter. Do the same thing for XClass and the table should look like 3.7. Now if you recompile everything it should compile.

What we have done here is associated every meta-model entity with an corresponding entity from the Eclipse JDT back-end. This will help us greatly in implementing more analysis tools. Of course, we are not restricted to mapping them directly to a JDT entity, you could have provided your own implementation of the *IMethod* if you wished.

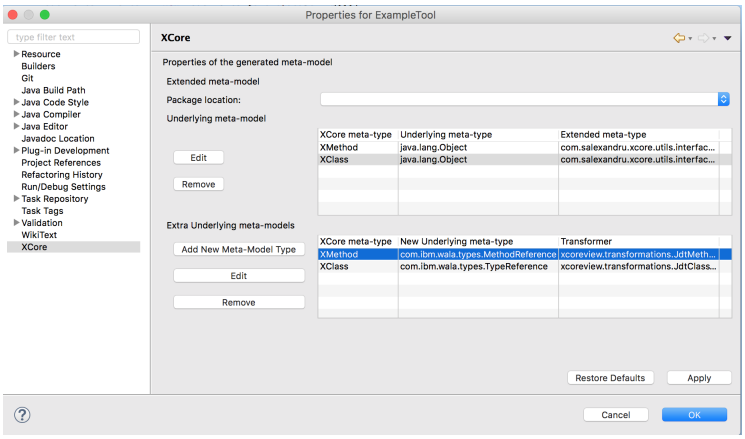


Figure 3.5: XCore Project Configuration

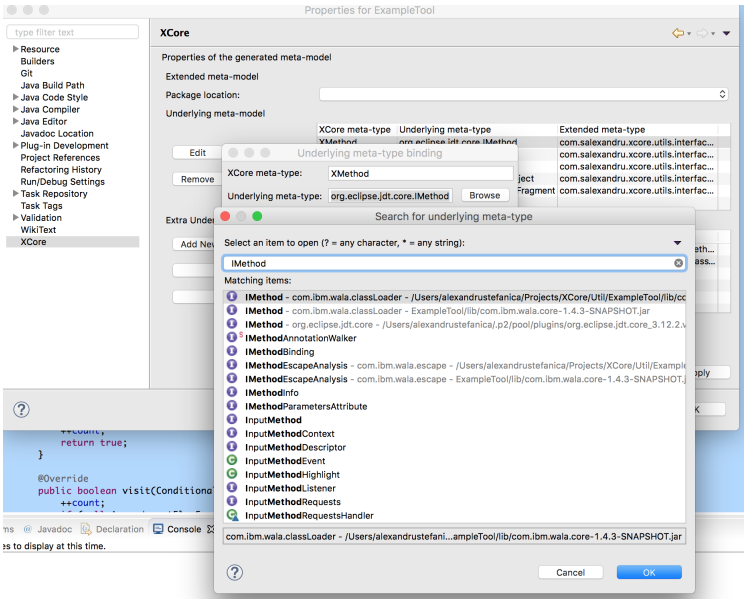


Figure 3.6: XCore Search Type Widget

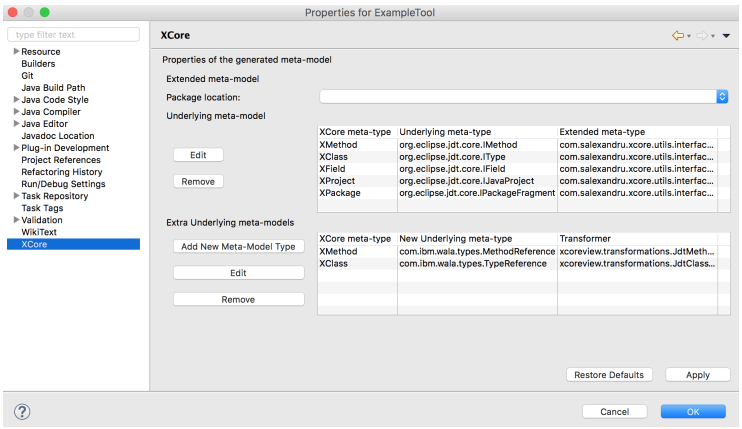


Figure 3.7: XCore Project Configuration

### 3.2.4 Number of calls to the method

For the last metric that we want to implement, we will want to computer the number of times this method is called from within the project. For this we need to build want is known as the Control Flow Graph of the project. "A control flow graph (CFG) in computer science is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The control flow graph is due to Frances E. Allen, who notes that Reese T. Prosser used boolean connectivity matrices for flow analysis before." [13]

This is quite hard to using only the Eclipse JDT infrastructure because it will imply to building it from scratch. Simply writing our algorithm for building the CFG of the code is tidious and wrong as we would go about reinventing the wheel. What we will due is use another back-end called WALA.

With XCore, now, we can add multiple back-end to it, not just one. We have implemented this new feature for easier integration with multiple tools that use quite different back-ends.
In order to integrate with another back-end we first have to add the necessary dependencies and implement a transformer from the original entity, i.e JDT IMethod in our case, to, in our case, WALA IMethod. A transformer is simply to implement. We have to impelment the `ITransform <OriginalUnderlyingType, NewUnderlyingType>`. An example of the implementation that transforms JDT IMethod to WALA IMethod can be found in the code section below 3.2.4

```
1 class JdtMethodToWala implements ITransform<IMethod,
```

```
    MethodReference> {
@Override
public MethodReference transform(IMethod origObj) {
final ASTParser parser = ASTParser.newParser(AST.JLS8);
final MethodBindingFinder finder = new
    MethodBindingFinder(origObj);

parser.setSource(origObj.getCompilationUnit());
parser.setKind(ASTParser.K_COMPILATION_UNIT);
parser.setProject(origObj.getJavaProject());
parser.setResolveBindings(true);

final ASTNode node = parser.createAST(null);

node.accept(finder);

final JDTIdentityMapper mapper = new
    JDTIdentityMapper(ClassLoaderReference.Primordial,
    node.getAST());

 return mapper.getMethodRef(finder.getBinding());

}

@Override
public IMethod reverse(MethodReference newObj) {
return null;
}


private static final class MethodBindingFinder extends
    ASTVisitor {
private IMethod method_;
private IMethodBinding methodBinding_ = null;

public MethodBindingFinder (IMethod method) {
    if (null == method) {
        throw new
            IllegalArgumentException("MethodBindingFinder
            was given to find a null method ...");
    }

    method_ = method;
```

```
38 }
39
40 public IMethodBinding getBinding() {
41     if (methodBinding_ == null) {
42         throw new RuntimeException("Method Binding for " +
              method_.toString() + " wasn't found in the
              AST");
43     }
44     return methodBinding_;
45 }
46
47 @Override
48 public void endVisit(MethodDeclaration m) {
49     if (methodBinding_ == null &&
          m.getName().toString().equals(method_.getElementName()))
          {
50         methodBinding_ = m.resolveBinding();
51     }
52 }
53
54 @Override
55 public boolean visit(MethodDeclaration m) {
56     if (methodBinding_ == null &&
          m.getName().toString().equals(method_.getElementName()))
          {
57         methodBinding_ = m.resolveBinding();
58     }
59
60     return true;
61 }
62
63 @Override
64 public void endVisit(MethodInvocation m) {
65     if (methodBinding_ == null &&
          m.getName().toString().equals(method_.getElementName()))
          {
66         methodBinding_ = m.resolveMethodBinding();
67     }
68 }
69
70 @Override
71 public boolean visit(MethodInvocation m) {
72     if (methodBinding_ == null &&
```

```
            m.getName().toString().equals(method_.getElementName()))
            {
73           methodBinding_ = m.resolveMethodBinding();
74       }
75
76       return true;
77   }
78   }
79   }
```

After we have the transformer we can now go back to the *Project Properties / XCore* and edit the second table (this is why the second table exists, add more elements to the backend). It is similar to what we have done before but now we have to add the transformer too, not only the type. Figure 3.8 shows you an example.

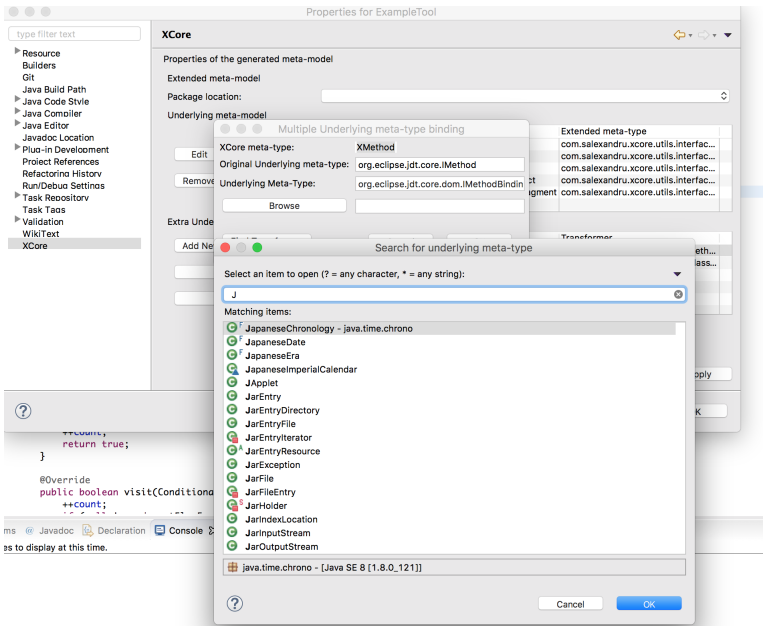


Figure 3.8: Adding another back-end to XCore

Now that we have everything ready in place we can write the code for this highly complex metric. The code from the section below 3.2.4 should now be easy to understand!

```
1  @PropertyComputer
2  public class SomethingAboutCG implements
       IPropertyComputer<Integer, XMethod> {
3
```

```
4           @Override
5           public Integer compute(XMethod entity) {
6                   IJavaProject p =
                       entity.getUnderlyingObject().getJavaProject();
7                   try {
8                           JavaEclipseProjectPath path =
                               JavaEclipseProjectPath.make(p,
                               AnalysisScopeType.SOURCE_FOR_PROJ_AND_LINKED_PROJS);
9                           AnalysisScope scope =
                               path.toAnalysisScope(new
                               JavaSourceAnalysisScope());
10                          scope.setExclusions(null);
11                          JDTClassLoaderFactory factory =
                               new
                               JDTClassLoaderFactory(scope.getExclusions());
12
13
14                          ClassHierarchy cha =
                               ClassHierarchyFactory.make(scope,factory);
15
16
17
18                          Iterable<Entrypoint> entryPoints =
                               Util.makeMainEntrypoints(scope,
                               cha);
19                          AnalysisOptions opts = new
                               AnalysisOptions(scope,
                               entryPoints);
20                          AnalysisCache cache = new
                               AnalysisCache(null, null, null);
21                          SSAPropagationCallGraphBuilder
                               cgBuilder =
                               Util.makeZeroCFABuilder(opts,
                               cache, cha, scope);
22                          CallGraph cg =
                               cgBuilder.makeCallGraph(opts,
                               null);
23
24                          return
                               cg.getNodes(entity.asMethodReference()).size();
25                  } catch (IllegalArgumentException |
                       CallGraphBuilderCancelException |
                       IOException | CoreException |
```

```
                    ClassHierarchyException e) {
26                      e.printStackTrace();
27
28                      return null;
29                      //throw new RuntimeException(e);
30                  }
31          }
32
33  }
```

# Chapter 4

# Conclusions

# List of Figures

# List of Tables

# Bibliography


[1] S. Alexandru and P. F. Mihancea. Xcore: Framework for software analysis tool development. June 2015.

[2] Stéphane Ducasse, Tudor Gîrba, Adrian Kuhn, and Lukas Renggli. Meta-environment and executable meta-language using smalltalk: an experience report. *Software and Systems Modeling*, 8(1):5–19, 2009.

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

[4] George Ganea, Ioana Verebi, and Radu Marinescu. Continuous quality assessment with incode. *Science of Computer Programming*, 2015.

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

[6] Paul Klint, Tijs van der Storm, and Jurgen Vinju. *EASY Meta-programming with Rascal*, pages 222–289. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[7] Patrick Lam, Eric Bodden, Ondrej Lhoták, and Laurie Hendren. The soot framework for java program analysis: a retrospective. 2011.

[8] Cristina Marinescu, Radu Marinescu, Petru Florin Mihancea, and R Wettel. iplasma: An integrated platform for quality assessment of object-oriented design. In *In ICSM (Industrial and Tool Volume)*. Citeseer, 2005.

[9] Ş. Medeleanu and P. F. Mihancea. Nullterminator: Pseudo-automatic refactoring to null object design pattern. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 601–603, Oct 2016.

[10] Alexandru Ştefănică and Petru Florin Mihancea. Xcore: Support for developing program analysis tools. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 462–466. IEEE, 2017.

[11] IBM Watson. Watson libraries for analysis, wala. sourceforge. net/wiki/index. php. *Main Page*.

[12] J Wiegand et al. Eclipse: A platform for integrating development tools. *IBM Systems Journal*, 43(2):371–383, 2004.

[13] Wikipedia. `https://en.wikipedia.org/wiki/Control_flow_graph`.