Reverse Engineering of Object-Oriented Code into Umple using an Incremental and Rule-Based Approach

Miguel A. Garzón

Presented to the **Faculty of Graduate and Postdoctoral Studies** in partial fulfillment of the requirements for the degree Doctor of Philosophy (Computer Science)

July 13, 2015

Research Questions

1. **RQ1**. To what extent can we achieve automated umplification?
2. **RQ2**. What transformation technology and transformations will work effectively for umplification?
3. **RQ3**. What should be the architecture, design and implementation of an umplification tool?
4. **RQ4**. What would be an effective process for improving the accuracy of the umplification tool?

Hypothesized Solution

Automated umplification can be achieved on a wide variety of systems.

The Umple Language

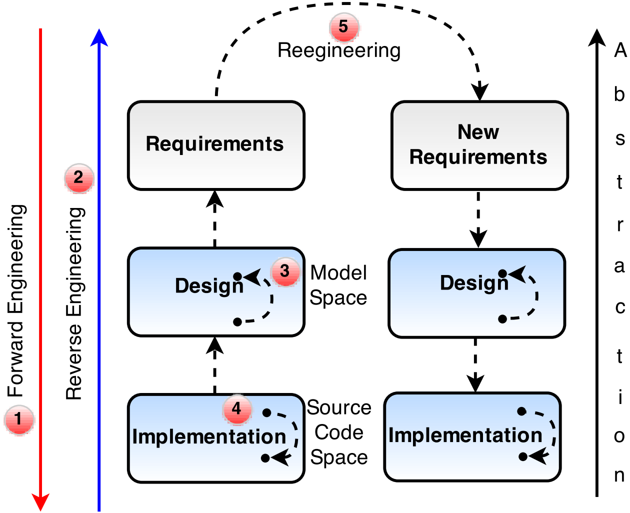
A Modeling and Programming Language

- Umple is an open-source textual modeling and programming language that adds UML abstractions to base programming languages including Java, PHP, C++ and Ruby.
- Umple was designed for modeling and developing large systems and for teaching modeling.

Umple constructs

- Associations, Attributes, State Machines
- Traits, Aspect Oriented Code Injections
- Patterns
- Tracing, Constraints, Concurrency

Transformations



Motivations

Our desire to develop our reverse-engineering approach arose for two main reasons:

Model-code duality

End-product of umplification is not a separate model, but a single artifact seen as both the model and the code.

Improving Program Comprehension

The resulting Umple code base tends to be simpler to understand as the abstraction level of the program has been '*amplified*'.

Our Approach

Umplification A play on words with the concept of '*amplification*' and also the notion of converting into Umple.

- The approach produces a program with behavior identical to the original one, but written in Umple.
- The approach eliminates the distinction between code and model.
- The approach proceeds incrementally until the desired level of abstraction is achieved.

The umplification approach is:

Incremental

Proceeds incrementally until the desired level of abstraction is achieved

Transformational

Modeling constructs are added replacing the original code

Interactive

Requires the user's feedback in order to enhance the transformations.

Extensible

Uses a mechanism that can be readily extended to refine the transformations.

Implicit-Knowledge Conserving

Preserves code comments, annotations, etc.

The Umplification Process: Refactoring Steps

The refactoring steps are the abstract transformations. The following are the refactorings steps currently implemented:

- **Transformation 0**: Initial transformation
- **Transformation 1**: Transformation of generalization/specialization, dependency, and namespace declarations.
- **Transformation 2**: Analysis and conversion of many instance variables, along with the methods that use the variables.
  - **Transformation 2a**: Transformation of variables to UML/**Umple attributes**.
  - **Transformation 2b**: Transformation of variables in one or more classes to UML/**Umple associations**.
  - **Transformation 2c**: Transformation of variables to UML/**Umple state machines**.

The Umplification Process: Refactoring Steps (2)

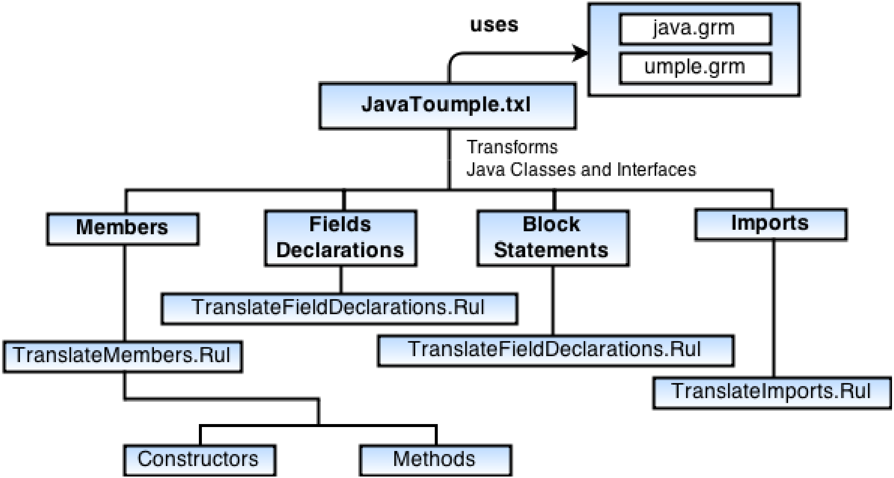
As part of each transformation step, the accessor, mutator, iterator and event methods are adapted (refactored) to conform to the Umple generated methods.

- **Classes**: None
- **Inheritance**: None
- **Attributes**: Accessor (getter) and mutator (setter) methods are removed/refactored from the original code.
- **Associations**: Accessor and mutator methods are removed or correctly injected into the umple code.
- **State Machines**: Methods triggering state change are removed if they are simple (just change state) or modified to call Umple-generated event methods.

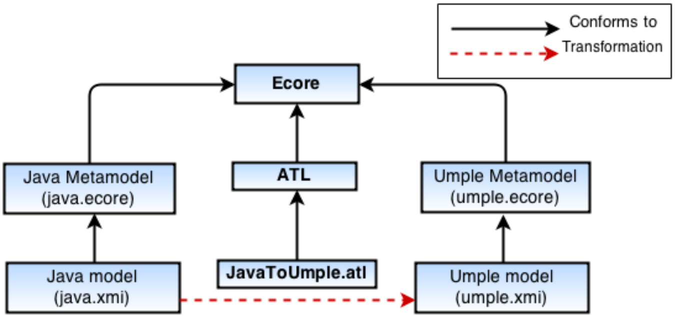
General Requirements for the Umplificator Tool

1. Support for various input languages
2. Incrementality
3. Rule execution control
4. Command-line support
5. Directionality
6. Usability of Language
7. Rule organization
8. Output Export
9. Maintainability
10. Extensibility

Alternative Approaches studied - TXL



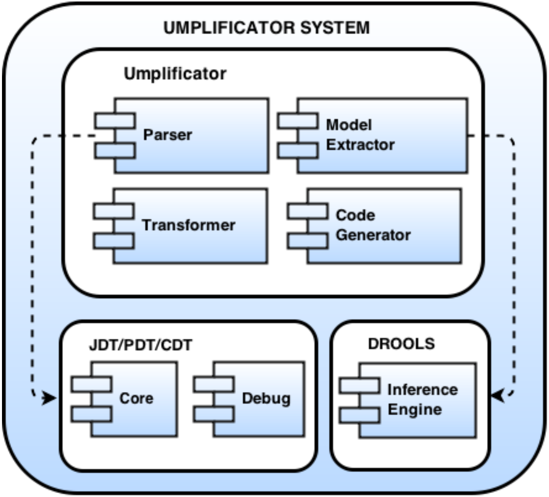
Alternative Approaches studied - ATL



ATL vs. TXL

Evaluation Criteria	ATL	TXL
Support for various input languages	-	+
Incrementality	-	+
Rule execution control	+	++
Command-line support	-	+++
Usability of Language	++	+
Rule organization	+++	+++
Output Export	-	+++
Maintainability	+	++
Extensibility	-	+

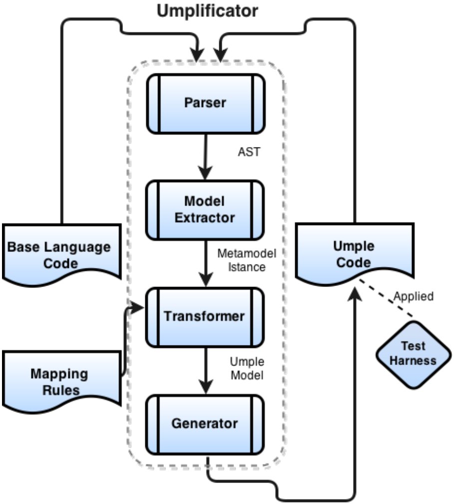
Architecture



Third Party Technologies

Technology	Targeted component(s)
JDT/CDT/PDT	Parser and Model Extractor
Drools Rule Engine	Transformer
JOpt Simple	General
Log4j	General
Perf4j	General

The Process Flow



Rule-Based Language

The rule engine interprets and executes the mapping rules on the source/target model to produce the umplified version of the target model.

```
rule "name"
  when LHS then RHS
end
```

A rule file in Drools is a file with a .drl extension that can have the following elements:

- **Package**
- **Imports**
- **Global Variables**
- **Functions**
- **Queries**

Summary of the Umplificator technologies

The **advantages** of our mixed approach are:

1. Separation of concerns
2. Speed and Scalability
3. Centralization of Knowledge
4. Multi-level Testing
5. Robust parsing
6. Efficiency
7. Agile development
8. Extensibility
9. Reusability

4-Phase Validation Process

Testing Phase : Unit testing.

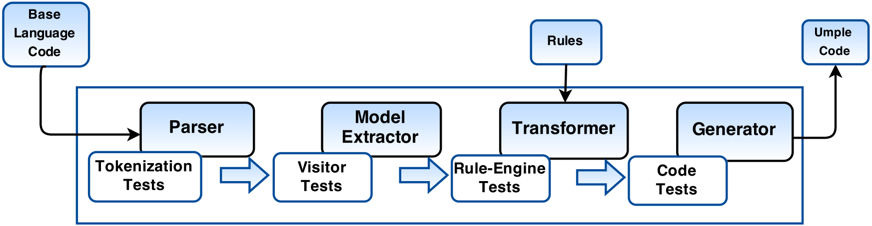
Pre-validation Phase Validation with small Java systems written in high quality Java code.

Initial Phase : Validation with Medium and large open-source projects (**‘training set’**).

Machine Learning-Based Phase : Validation with a set of randomly selected systems, the **‘testing set’**.

Testing Phase

135 tests that span all components of the tool (parser, extractor, transformer, generator) and are run as part of our automated building process.



Pre-Validation Phase

We have tested the umplificator using our own repository of **42 small Umple examples**.

Initial Validation Phase

Apply the Umplificator to various open-source systems written in Java that we have not created ourselves. that we have not created ourselves.

Name	Version	LOC	# of Classes
1. JHotDraw	7.5.1	82132	694
2. Weka	3.7.13	278642	1370
3. Java Bug Reporting Tool	1.0	2629	27
4. JEdit	1.12	59699	84
5. FreeMaker	2.3.15	39864	131
6. Java Financial Library	1.6.1	1248	28
7. Args4j	2.0.30	2223	61

Results of Initial Validation Phase - JHotDraw

	TP	FP	Expected	Precision	Recall
Attributes	383	20	363	95%	100%
Associations optional-one-to-many	22	0	49	100%	45%
Associations optional-one-to-one	115	0	185	100%	62.2%
Associations many-to-many	30	0	32	100%	93.8%

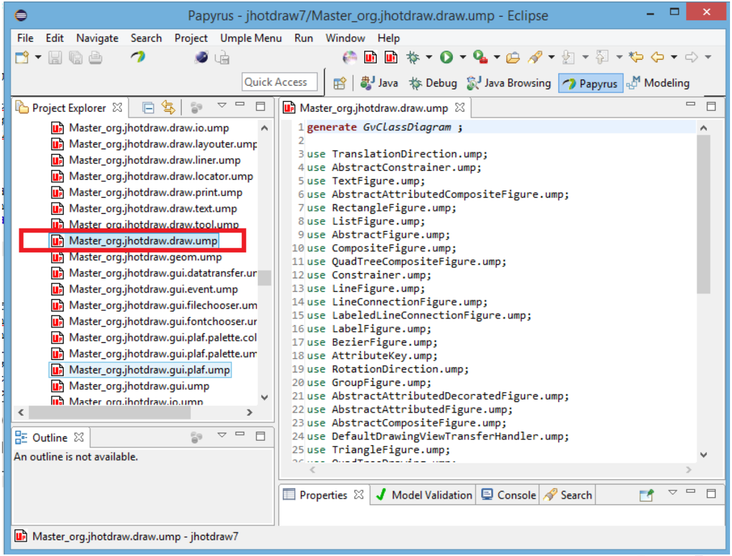
Results of Initial Validation Phase - Execution Time

	Execution Time (in ms)	
Component	**JHotDraw**	**Args4J**
Parsing	50899	12500
Extractor	21025	3204
Transformer	339327	920
Generating Umple Code	1700	450
Total Time:	412951	14074

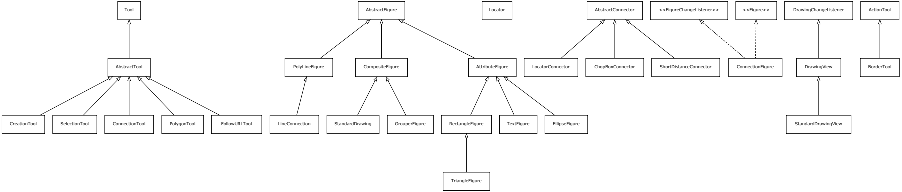
Second Validation Phase

1. **Download** 100 projects (randomly).
2. **Umplify** the projects.
3. **Report** the scores.
   - (Doesn't Exist)
   - F.Umplify.Score.log
   - {012}.Umplify.Score.log

JHotDraw


Papyrus - jhotdraw7/Master_org.jhotdraw.draw.ump - Eclipse
File Edit Navigate Search Project Umple Menu Run Window Help
Quick Access
Java Debug Java Browsing Papyrus Modeling
Project Explorer
Master_org.jhotdraw.draw.io.ump
Master_org.jhotdraw.draw.layouter.ump
Master_org.jhotdraw.draw.liner.ump
Master_org.jhotdraw.draw.locator.ump
Master_org.jhotdraw.draw.print.ump
Master_org.jhotdraw.draw.text.ump
Master_org.jhotdraw.draw.tool.ump
Master_org.jhotdraw.draw.ump
Master_org.jhotdraw.geom.ump
Master_org.jhotdraw.gui.datatransfer.ur
Master_org.jhotdraw.gui.event.ump
Master_org.jhotdraw.gui.filechooser.um
Master_org.jhotdraw.gui.fontchooser.ur
Master_org.jhotdraw.gui.plaf.palette.col
Master_org.jhotdraw.gui.plaf.palette.um
Master_org.jhotdraw.gui.plaf.ump
Master_org.jhotdraw.gui.ump
Outline
An outline is not available.
Master_org.jhotdraw.draw.ump
1 generate GvClassDiagram ;
2
3 use TranslationDirection.ump;
4 use AbstractConstrainer.ump;
5 use TextFigure.ump;
6 use AbstractAttributedCompositeFigure.ump;
7 use RectangleFigure.ump;
8 use ListFigure.ump;
9 use AbstractFigure.ump;
10 use CompositeFigure.ump;
11 use QuadTreeCompositeFigure.ump;
12 use Constrainer.ump;
13 use LineFigure.ump;
14 use LineConnectionFigure.ump;
15 use LabeledLineConnectionFigure.ump;
16 use LabelFigure.ump;
17 use BezierFigure.ump;
18 use AttributeKey.ump;
19 use RotationDirection.ump;
20 use GroupFigure.ump;
21 use AbstractAttributedDecoratedFigure.ump;
22 use AbstractAttributedFigure.ump;
23 use AbstractCompositeFigure.ump;
24 use DefaultDrawingViewTransferHandler.ump;
25 use TriangleFigure.ump;
Properties Model Validation Console Search
Master_org.jhotdraw.draw.ump - jhotdraw7

JHotDraw


Tool
AbstractTool
CreationTool
SelectionTool
ConnectionTool
PolygonTool
FollowURLTool
AbstractFigure
PolyLineFigure
CompositeFigure
AttributeFigure
LineConnection
StandardDrawing
GrouperFigure
RectangleFigure
TextFigure
EllipseFigure
TriangleFigure
Locator
AbstractConnector
LocatorConnector
ChopBoxConnector
ShortDistanceConnector
<<FigureChangeListener>>
<<Figure>>
ConnectionFigure
DrawingChangeListener
DrawingView
StandardDrawingView
ActionTool
BorderTool

WEKA

Results

Initial Umplification results for Weka nonetheless had a precision of 85% when it comes to attributes and 38% for 1-to-many associations.

Why

Note that a precision of 38% doesn't mean that the Umplificator has missed 62% associations of this type. It means that some of them were not correctly transformed into Umple (e.g. incorrect navigability, role names or transformation of accessor/mutator methods).

Args4j - Modernization

- **Original** Args4j Java source code is composed of 61 classes and 2223 lines of code.
- **Umplified** Args4j source code is composed of 122 (2 per input class) umple files and 1980 lines of code in total.
- Total number of lines of code in files containing modeling constructs (X.ump) is 312 LOC.
- Total number lines of code in files with algorithmic/logic code (X_code.ump) is 1668 LOC. If we exclude the umple class declarations and curly brackets, the number becomes 1424 LOC.

Translating source code base

To achieve the goal of translating Args4j into C++, a developer must translate 1518 lines of code (rather than 2223 lines of code).

Tuning of the Umplificator

The complexity of the tuning depends on the number of false positives and false negatives that the tool generates.

1. **Umple construct missed (false negative)**: we may add a new mapping rule to cover this case.
2. **Umple construct incorrectly identified (false positive)**: we may edit the corresponding mapping rule.
3. **Method incorrectly refactored**: we may review and correct the refactoring action (a function in Drools language) that led to the incorrect piece of code.

Related Work

Author-Ref	Scalability	Incrementality	Validation	Usability	Target Lang.	Attributes?	Associations?
Barowski et al. - 2002	N	N	P	N	Java	P	P
Grant et al. - 2011	P	N	P	F	Java	F	P
Gueeheneuc - 2004	P	P	F	F	Java	F	F
Vinita et al. - 2008	N	N	N	N	Cpp	P	P
Kang et al. - 2007	N	N	N	F	Java	F	P
Sutton et al. - 2007	N	N	P	F	Cpp	F	F
Korshunova et al. - 2006	P	N	P	F	Cpp	F	P
Umplification	**P**	**F**	**P**	**F**	**Java,Cpp**	**F**	**P**

Summary of Contributions

1. The overall concept of umplification and the defined levels of refactoring;
2. An understanding of how umplification compares with other reverse engineering techniques;
3. The implementation and analysis of integrating different transformation technologies; resulting in the Umplificator tool itself;
4. Case studies of Umplification, demonstrating strengths, weaknesses and opportunities. Results presented in this thesis are reproducible and repeatable;
5. Mapping rules for Umplification and the language for expressing these;
6. Detection of associations in a body of source code;
7. **We also made improvements to Umple compiler to support abstract classes, interfaces and top-level enumerations, as needed to support the case studies.**

Summary of Tools Developed

Command Line Tool

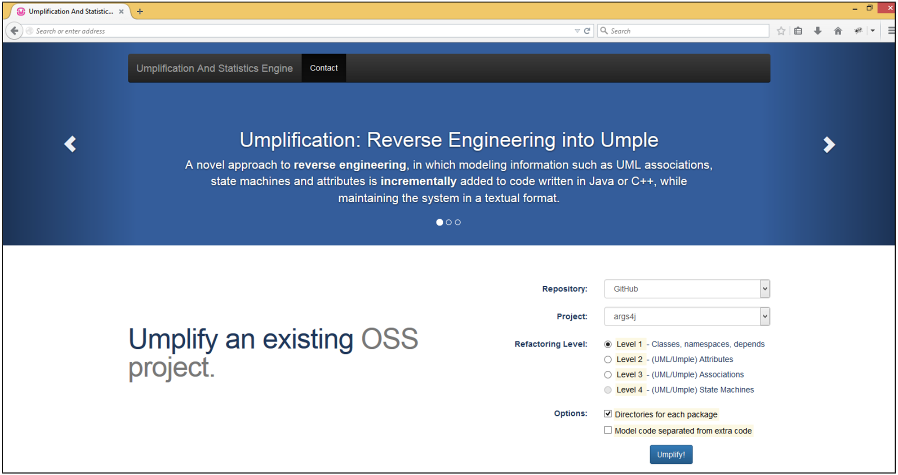
1. **cruise.umplificaror.eclipse_vX.X.X.jar** : Plug-in for the Eclipse IDE
2. **umplificator_X.Y.Z.jar** : Command-Line tool for umplification
3. **validator_X.Y.Z.jar** : Command-Line tool that checks whether the input Umple code generates compilable base language code.

Umplifying source code by means of the command-line tool can be done using the following command:

java -jar umplificator.jar inputFile -level=1,2,3 -splitModel -dir -path

Summary of Tools Developed

Web-App



Summary of Tools Developed

Various

Script

The code for the script is published online and can be found in our **code repository** in the following directory:
`cruise.umplificator/scripts/umplificator_all_projects.sh`.

Downloader

The project surrounding the umplificator that automatically downloads, tracks and reports on various projects can be found at:
`https://github.com/mgarzon/dlproj`.

Future Work

1. Umplify more Projects
2. State Machines Support
3. IDE Integration
4. Evaluation with real developers
5. Exploring other types of software
6. Umplifying code that matches other model elements

The End

Thank you!

Additional Notes

Answers to Research Questions (1)

RQ1. To what extent can we achieve automated umplification?

1. The evaluation results showed that our approach and its current implementation are effective and efficient enough to be applied to real systems.
2. We were able to successfully umplify many real open-source systems at varying degrees of effectiveness, and were able to iteratively improve this effectiveness by adjusting rules.

Answers to Research Questions (2)

RQ2. What transformation technology and transformations will work effectively for umplification?

1. We discussed our experiments with ATL and TXL approaches.
2. We selected an approach that uses:
   (a) **the mature xDT** family of parsing libraries (JDT, CDT, etc.) to process both code and textual models, and build in memory models;
   (b) the **Drools tool** to transform the in-memory models, accomplishing the core umplification task and;
   (c) **our own generator technology** to produce revised versions of the system.

Answers to Research Questions (3)

RQ3. What should be the architecture, design and implementation of an umplification tool? The advantages of the selected approach enabled the following aspects, each of which we discussed in Chapter 5:

1. Separation of concerns
2. Speed and Scalability
3. Centralization of Knowledge
4. Multi-level Testing
5. Robust parsing
6. Efficiency
7. Agile development
8. Extensibility
9. Reusability

Answers to Research Questions (4)

RQ4. What would be an effective process for improving the accuracy of the umplification tool?

1. An approach adapted from the model often used in machine learning considerably improved the precision and recall of our transformations.
2. This validation approach takes one set of systems as the '**training set**' and then takes another set as the **'testing set'** to see how well the Umplificator performs on unknown systems.
3. The Umplificator is tuned based on problems found when working with the testing set.

The Umple Architecture

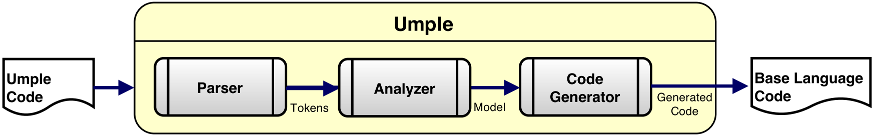


Figure: Umple Architecture

Umple is available as an Eclipse plugin, command-line tool and web-based application.

EXAMPLE

1. **Task:** *Umplify* a small system written in Java.
2. **Initial Input:** Three Java Classes *(Student.java, Person.java, Mentor.java*).
3. **Final Output:** An *Umple model* containing three Umple Classes (which contain Umple Attributes, associations, etc).
    - This Umple Model can also be viewed and edited as an UML Class Diagram.

Tranformations 0 and 1 (Student.java)

- One-to-one direct and simple mappings between constructs.
- The final output after execution, is an Umple model/program that can be compiled.
- Three files created at this point: *Student.ump*, *Mentor.ump*, *Person.ump*.

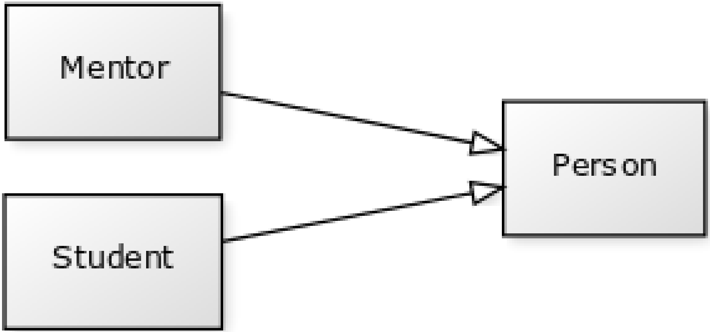
Java code:

```
package university;
import java.util.*;
public class Student extends Person { ... more code }
```

Umple code:

```
namespace university;
class Student {
   depend java.util.*;
   isA Person;
  /*The rest of the code*/
}
```

UML Class Diagram After Tranformation 1



Transformation 2: Refactoring to Create Attributes

- We analyze all instance variables for their presence in constructor and get/set methods.

Constructor	Setter	Getter	Attribute (probability)
Yes	Yes	Yes	High
Yes	Yes	No	Low
Yes	No	Yes	High
Yes	No	No	Low
No	Yes	Yes	High
No	Yes	No	Low
No	No	Yes	Medium
No	No	No	Medium Low

- We culminate this refactoring step by removing or refactoring getters and setters of the previously identified attributes.

Refactoring to Create Attributes - The Input code

Umple code after transformation 1 (INPUT):

```
class Student {
 depend java.util.*;
 isA Person;
 public Mentor mentor;
 public static final int MAX_PER_GROUP = 10;
 private int id;
 private String name;
 private boolean isActive;

 public Student(int id,String name){
      id = id; name = name;
 }
 public String getName(){
  String aName = name;
  if (name == null) { throw new RuntimeException("Error");}
  return aName;
}
```

Refactoring to Create Attributes - The Input code

```
public Integer getId() {
   return id;
}
public void setId (Integer id) {
   this.id = id;
}
public boolean getIsActive() {
   return isActive;
}
public void setIsActive ( boolean  aIsActive) {
  isActive = aIsActive;
}
public Mentor getMentor() { return mentor; }
public void setMentor(Mentor mentor) { this.mentor = mentor;
    }
}
}
```

Refactoring to Create Attributes - Analyzing the code:

For the class Student, we obtain the following results:

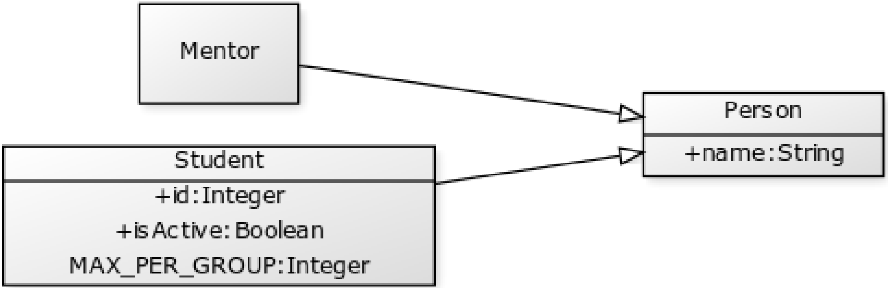
Member Variable	**Constructor?**	**Getter?**	**Setter?**	**Type?**
id	Yes	Yes	Yes	Yes
isActive	No	Yes	Yes	Yes
name	Yes	Yes	No	Yes
MAX_PER_GROUP	No	No	No	Yes

Refactoring to Create Attributes- The Output code

Umple code after transformation 2a (OUTPUT):

```
class Student {
 Integer id;
 lazy Boolean isActive;
 immutable name;
 const Integer MAX_PER_GROUP = 10;
 after getName {
  if (name == null) {
   throw new
     RuntimeException("Error");}
 }
 /*DEVELOPER CODE - PROVIDED AS-IS*/
  public Mentor mentor;
  public Mentor getMentor() { return mentor; }
  public void setMentor(Mentor mentor)
  {
     this.mentor = mentor;
  }
}
```

UML Class Diagram After Tranformation 2a.



Refactoring to Create Associations

- In order to guarantee the correct extraction of an association and to avoid false-negative cases, we consider not only the getter and setter of the fields but also the iteration call sequences (iterators).
- A variable represents an association if all of the following conditions apply:
  1. Its declared type is a **Reference type** (generally a class in the current system).
  2. The variable field is simple, or the variable field is a container (also known as a collection).
  3. The class in which the variable is declared, stores, access and/or manipulates instances of the variable type.

Refactoring to Create Associations (2)

Umple code before transformation 2b (INPUT):

```
class Student {/*The rest of the code*/ }

class Mentor {
 depend java.util.Set;
 isA Person;
 public Set<Student> students;
 public Set<Student> getStudents()
  { return students; }

 public void  setStudents (Set<Student>students)
  { this.students = students; }

 public void addStudent( Student  aStudent)
  { students.add(aStudent); }

 public void removeStudent(Student   aStudent)
  { students.remove(aStudent);}
}
```

Refactoring to Create Associations (3)

Umple code after transformation 2b (OUTPUT):

```

class Mentor {
 0..1 -- 0..* Student;
}
class Student {/*The rest of the code*/}
```

UML Diagram After Tranformation 2b.