The Umplification Process: Refactoring Steps

The refactoring steps are the abstract transformations. The following are the refactoring 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 from the original code if they are simple since Umple generated code replaces them.
  - **Custom** accessors and mutators are refactored so Umple generates code that maintains the original semantics.
- **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.
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:

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

Umple code:

```umple
namespace university;
class Student {
    depend java.util.∗;
    isA Person;
    /*The rest of the code*/
}
```
UML Class Diagram After Transformation 1
Transformation 2: Refactoring to Create Attributes

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

<table>
<thead>
<tr>
<th>Constructor</th>
<th>Setter</th>
<th>Getter</th>
<th>Attribute (probability)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Low</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>Yes</td>
<td>No</td>
<td>No</td>
<td>Low</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>High</td>
</tr>
<tr>
<td>No</td>
<td>Yes</td>
<td>No</td>
<td>Low</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>Yes</td>
<td>Medium</td>
</tr>
<tr>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Medium Low</td>
</tr>
</tbody>
</table>

- We culminate this refactoring step by removing or refactoring getters and setters of the previously identified attributes.
Refactoring to Create Attributes - The Input code

Uml 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 - Analyzing the code:

For the class `Student`, we obtain the following results:

<table>
<thead>
<tr>
<th>Member Variable</th>
<th>Constructor?</th>
<th>Getter?</th>
<th>Setter?</th>
<th>Type?</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>isActive</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>name</td>
<td>Yes</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>MAX_PER_GROUP</td>
<td>No</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Refactoring to Create Attributes - The Output code

Umple code after transformation 2a (OUTPUT):

```java
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 Transformation 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):

```java
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):

```java
1 class Mentor {
2     0..1 −→ 0..* Student;
3 }
4 class Student { /*The rest of the code*/ }
```
UML Diagram After Transformation 2b.