Refactoring
-By Means of Graph Rewriting

Chen Tang

April 5th, 2004

Contents

- Review of the refactoring concept
- Overview of the current refactoring tools
- Multi-formalism modeling environment -- AToM^3
- Graph rewriting concepts & principles
- Several refactoring rules LHS, RHS, etc.
- An example
- Conclusion

Recap of the refactoring concept

- Behavior-preserving transformations are known as refactorings.
- Refactorings are changes whose purpose is to make a program more reusable and easier to understand, rather than to add behavior.
- Refactorings are specified as parameterized program transformations along with a set of preconditions that guarantee behavior preservation if satisfied.

Reference from [1]

Goal of the refactoring tools

The main purpose of a refactoring tool is to allow the programmer to refactor without having to retest the program and automate as many refactoring rules as possible. Testing is time consuming even when automated, and eliminating it can accelerate the refactoring process by a significant factor.

What refactoring tools can do?

- Can automate refactoring, free programmers from having to perform mundane restructuring tasks manually
- Can both make refactoring less a separate activity from programming and can make design mistakes less costly.
- Can reduce the amount of tests. Since All the refactoring automated by refactoring tools are provably right.
- Can compose several primitive refactoring to attain more complex refactoring. Since each primitive refactoring preserves the behavior of the program, the entire composition is itself behavior-preserving.

Currently existing refactoring tools

- Refactoring Browser
  - The Refactoring Browser is a powerful Smalltalk Browser which allows the programmer to perform various automated refactorings on Smalltalk source code
- Transmogrify
  - Transmogrify is a Java source analysis and manipulation tool. The current focus of Transmogrify is as a cross-referencing and refactoring utility.
- C# refactory
  - C# Refactory is a revolutionary new tool integrated with VS.NET.
- Resharper
  - Yet another VS.NET plug-in

Technical criteria for a refactoring tools

- Program Database
  - The ability to search for various program entities across the entire program is vital important for refactoring tool, this information is maintained in a program database.
  - For example: class, class's attributes and methods etc., should all be store in the program databse
- Parse Trees
  - Most refactorings have to manipulate portions of the system below the method level. To do this requires parse trees. A parse tree is a data structure that represents the internal structure of the method itself.
  - Will store the components in the method

Practical criteria for a refactoring tool

- Speed
  - The analysis and transformation needed to perform refactorings can be time consuming if they are very sophisticated, refactoring must be fast, if not, programmer will prefer to do the refactoring by hand.
- The refactoring tool must be integrated into the standard IDE.
  - If integrated with IDE, all the refactoring functions are at the fingertips of the programmers, it's convenient for people to use. If not, no one would tend to use it.
- Should support undo functionality
  - With undo functionality, every thing can be explored with impunity, given that we can roll back to any prior version.

Practical criteria for a refactoring tool (cont'd)

- Avoid purely automatic reorganization
  - Whenever the refactoring tool do refactoring, it should always prompt the programmer to ask for the interaction, should avoid purely automatic reorganization of the program.
- Refactorings must be reasonably correct
  - In order to to make the programmer enthusiastic about using the refactoring too, the developer of the tool should guarantee the correctness of every refactoring rules implemented.

Multi-formalism modeling environment -- AToM^3

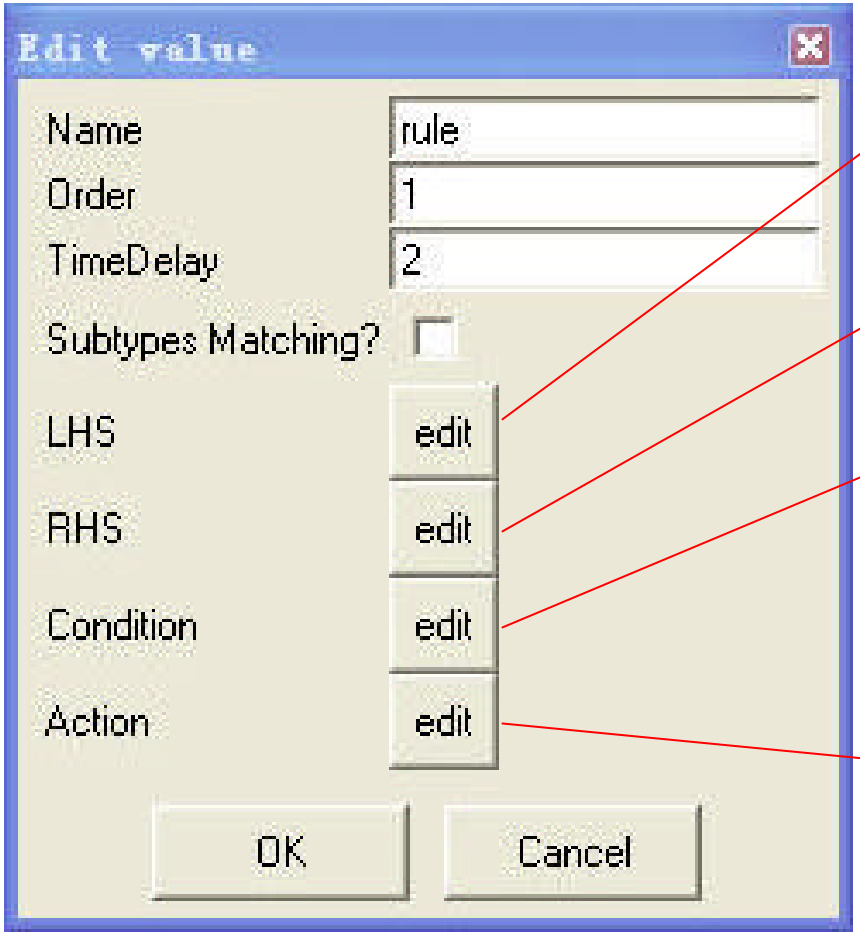
- AToM^3 stands for ``A Tool for Multi-formalism and Meta-Modeling"
- AToM^3 is a tool for multi-paradigm modeling
  - Use class diagram formalism in this project
- The two main tasks of AToM^3
  - Meta-modeling
  - Meta-transforming
- Expression of Models
  - Formalisms and models are described as graphs
  - Model transformations are performed by graph rewriting

Graph grammar & graph rewriting in AToM^3

- A graph grammar consists of several rules that specify a formalism for a given language
- Graph grammars are specified in AToM3 as model transformations
- The graph grammar using pattern

  matching to find all the patterns which match the transformation rule's LHS .

Define a new transformation



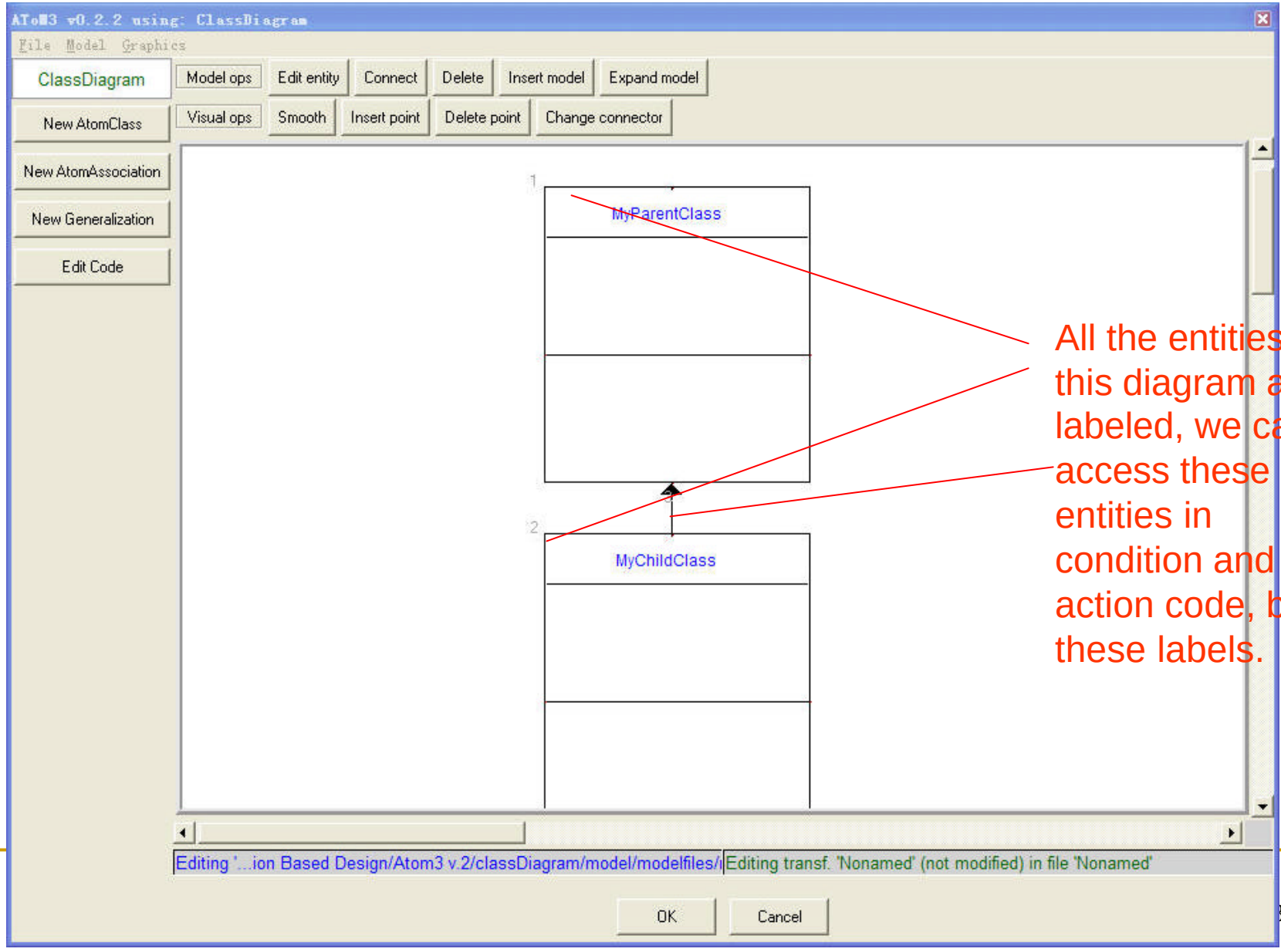
Define the LHS matching pattern of the transformation

Define RHS of the transformation, i.e., what LHS should be replaced

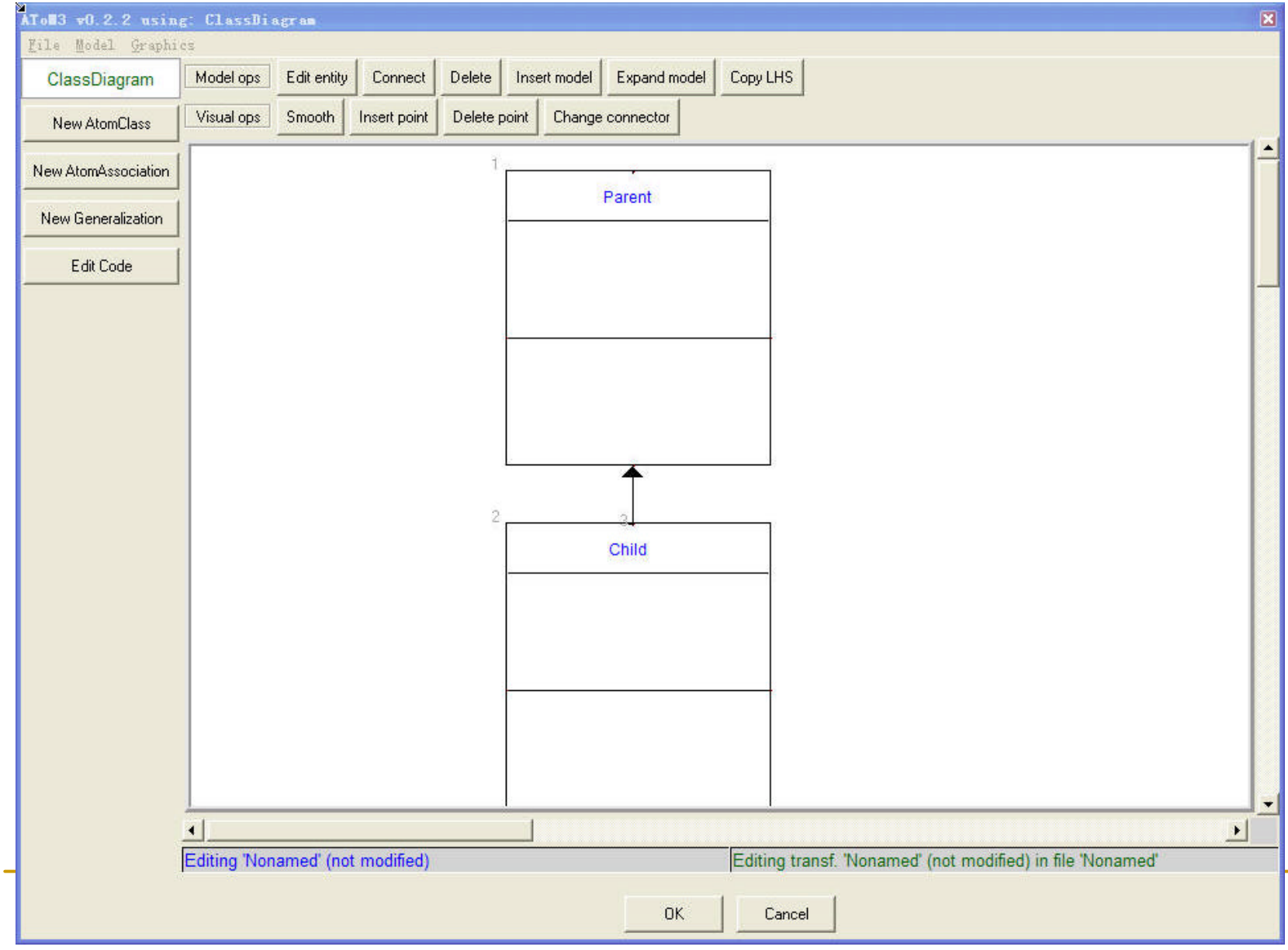
Here we place a bunch of python code, and only when these code return a true value can this transformation rule be executed!

Action, after this transformation rule be executed, the action code will be executed, normally we put update display and additional functionalities here.

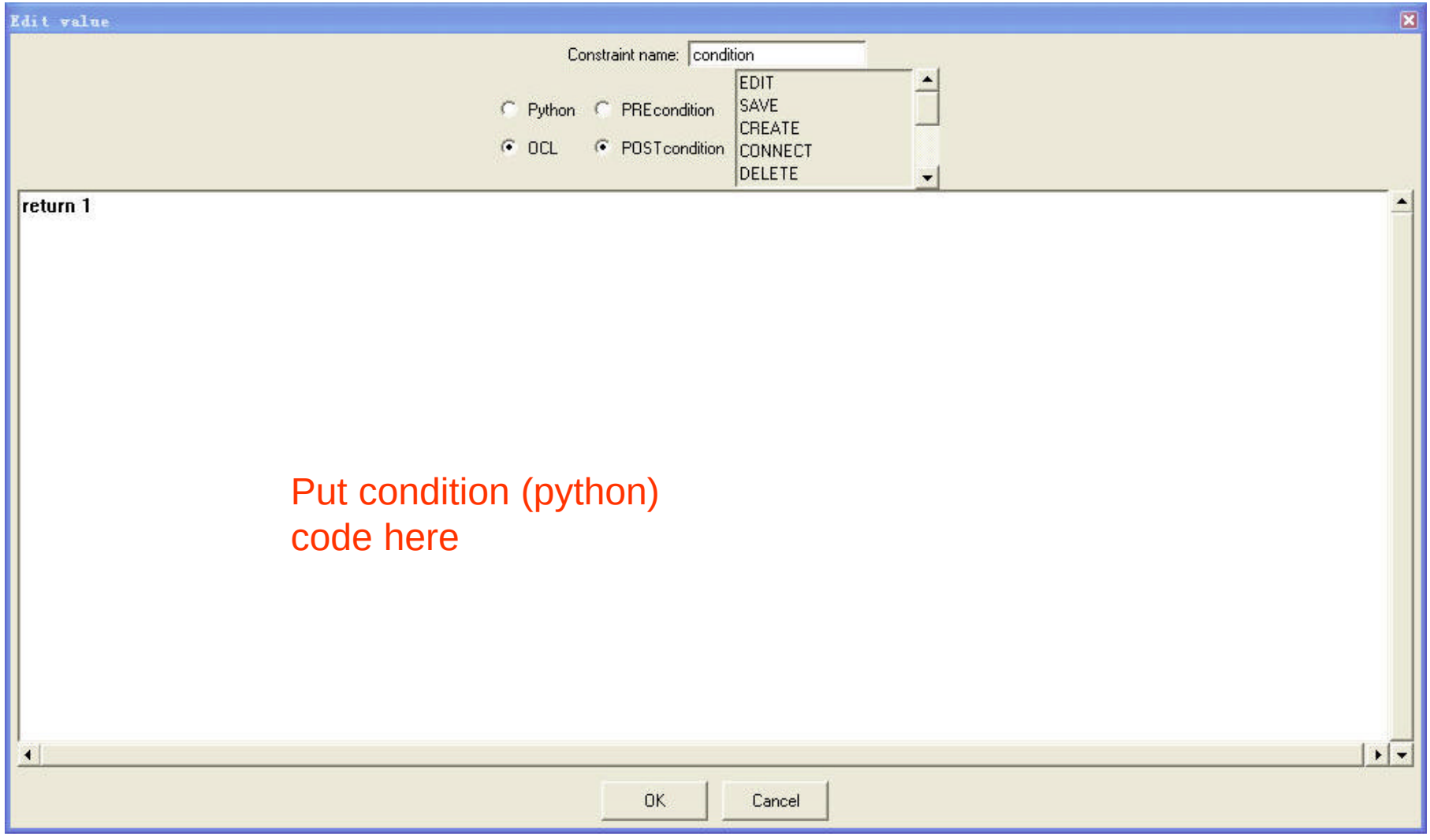
Interface for define transformation ś LHS



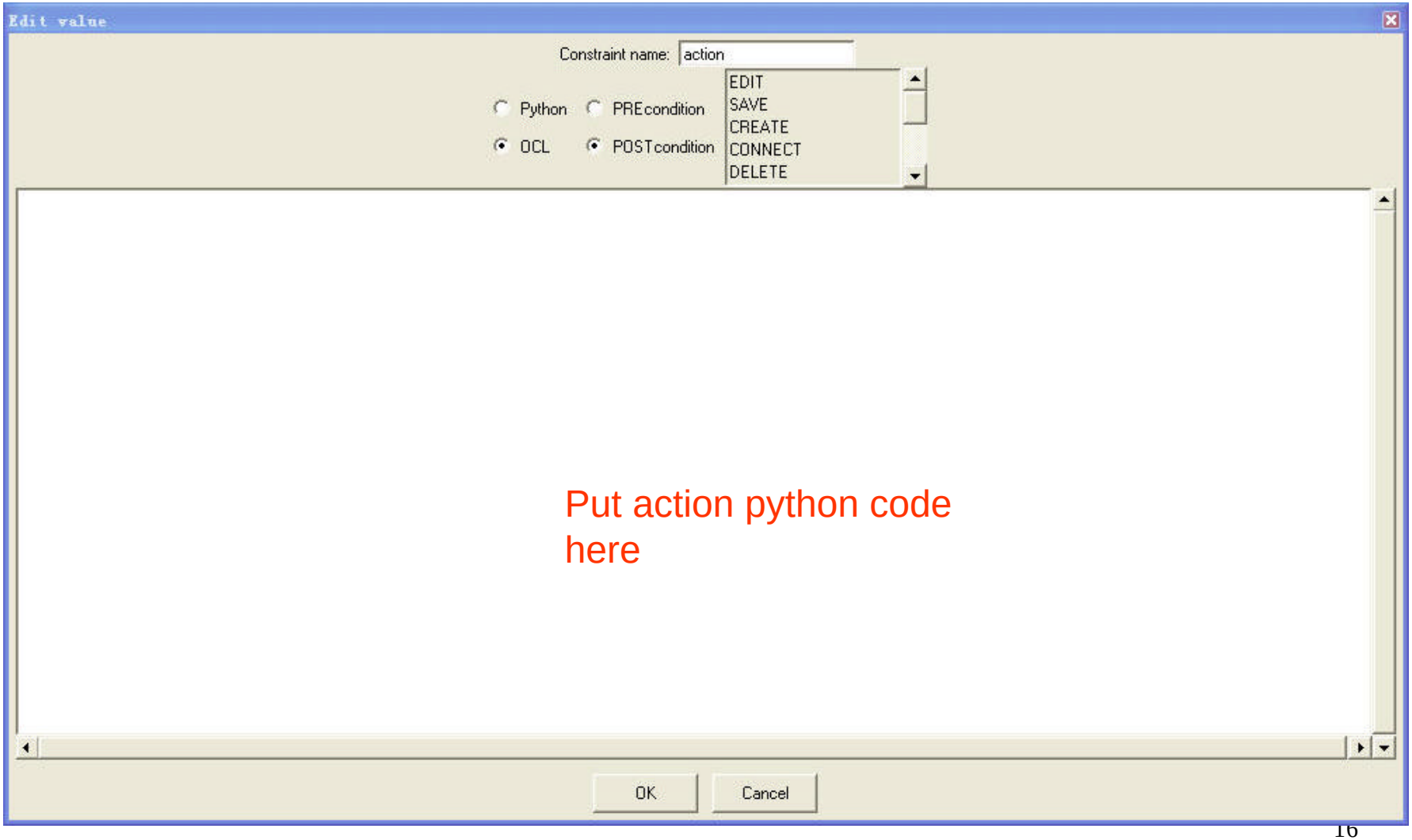
Interface for define RHS



Interface for define transformation ś condition



Interface for define transformation ś action



Transformations intend to be automated

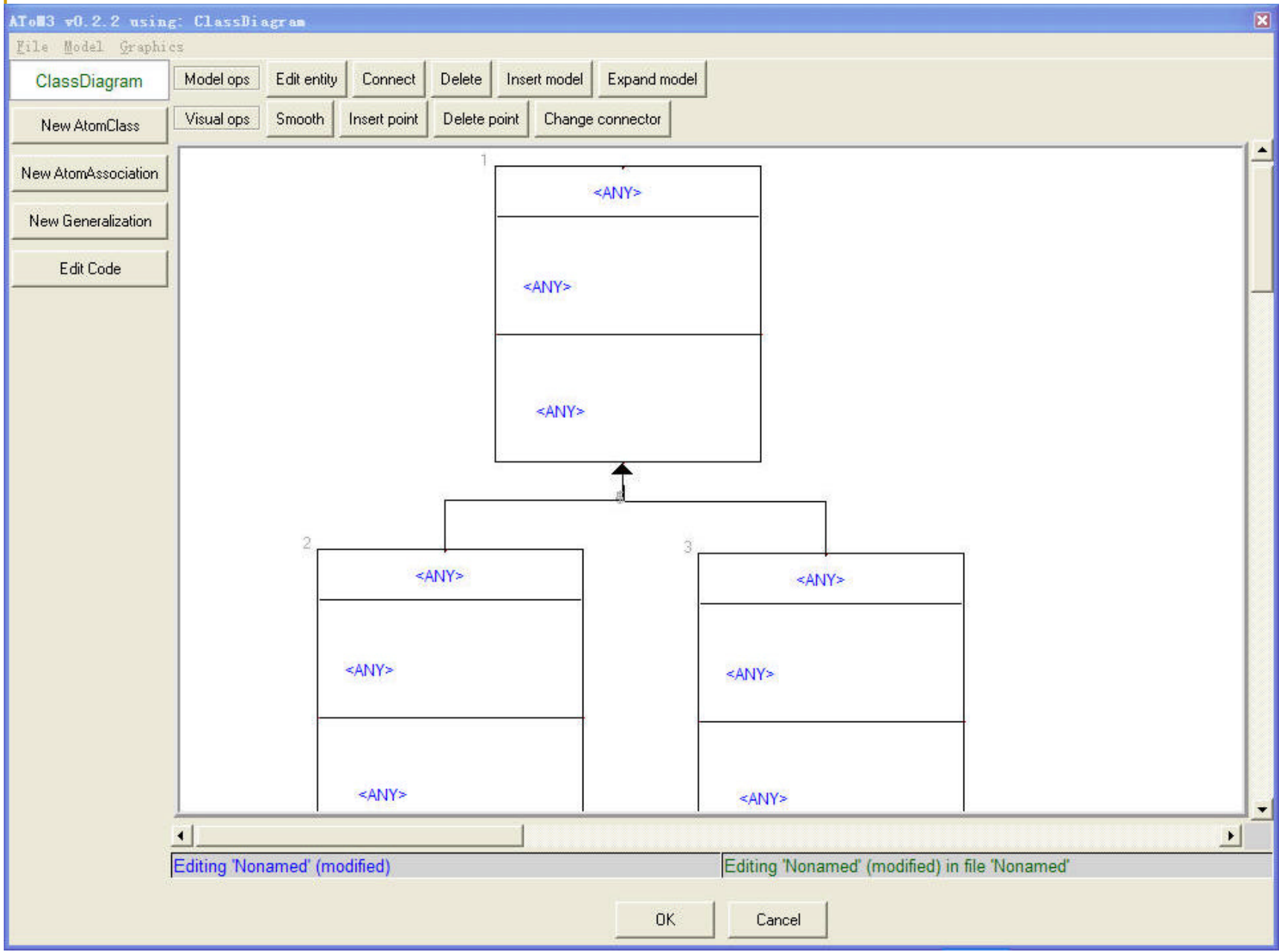
- Pull up field / pull up method

  If subclasses are developed independently, or combined through refactoring, they may have duplicate field/method. This refactoring rule removes the duplicate field/method, and move them from the subclasses to the superclass.

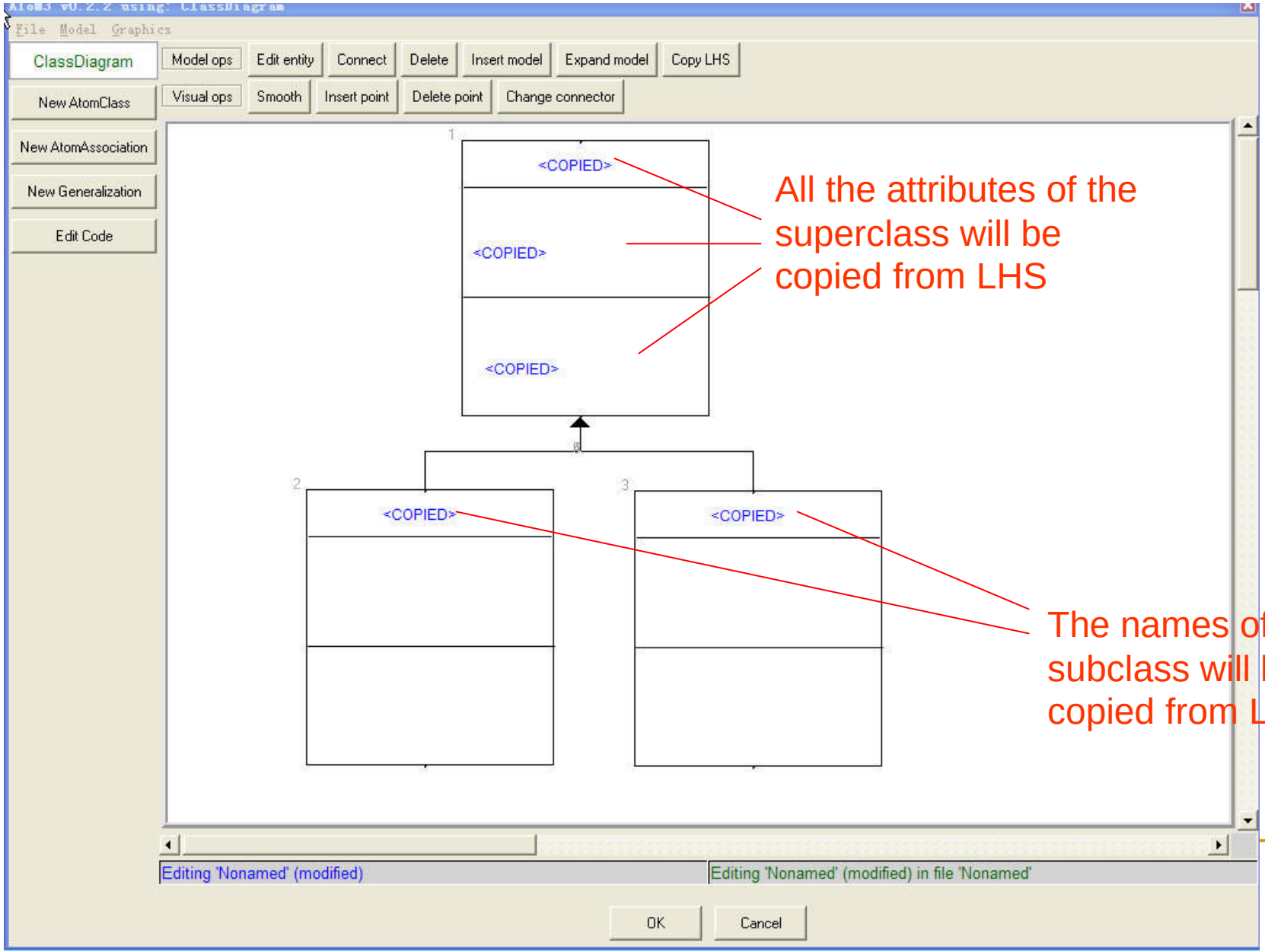
- Collapse hierarchy

  If a superclass and the corresponding subclass are not very different “merge them together”

Pull up field /pull up mehod (LHS)



Pull up field /pull up mehod (RHS)



Pull up field / pull up method action

When load and execute pull up field/method transformation, AToM^3 will begin doing pattern matching and when it finds there is a pattern matching the LHS of the pull up field/method LHS, it will execute the following action:

1. Enumerate all the entities in the matching pattern; if it a class then locate all the generalization relationships entities connected to this class entity, and according to the connection we can determine the type of the class, whether it's a superclass or a subclass. At the same time keep down the parent/child class this class.

2. Start another round of enumeration. According to this class's parent pointer locate the parent, and according to parent's child pointer locate all the sibling of this class. Then search all the sibling's attribute/method list to find the same attribute/method as this class, if find any, pull up the attribute/method to the superclass, delete the attribute/method from all sibling class and this class. Iterates until can't find any attribute/method which has the same signature as this class's attributes/methods.

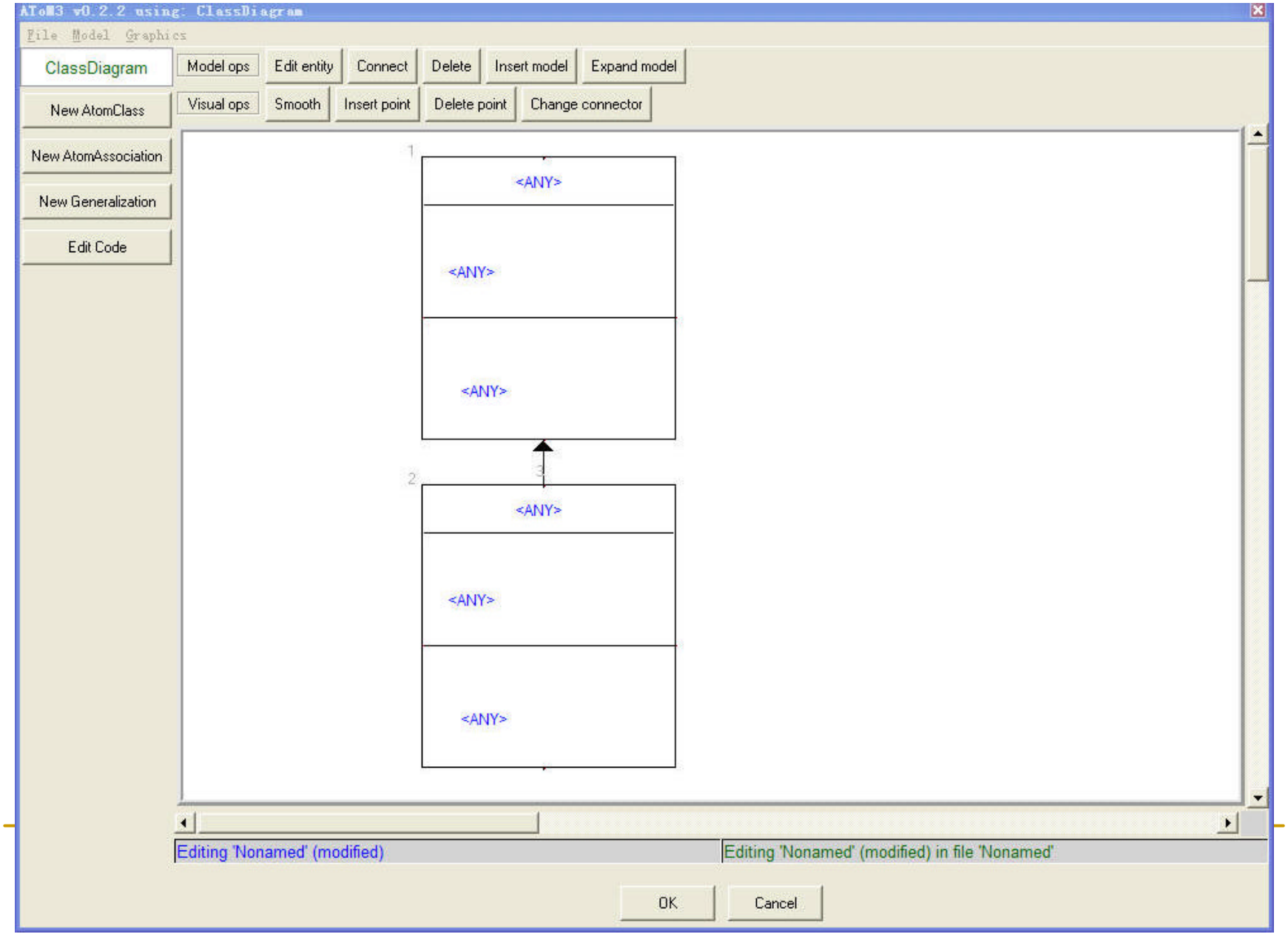
3. Update the display

Collapse hierarchy overview

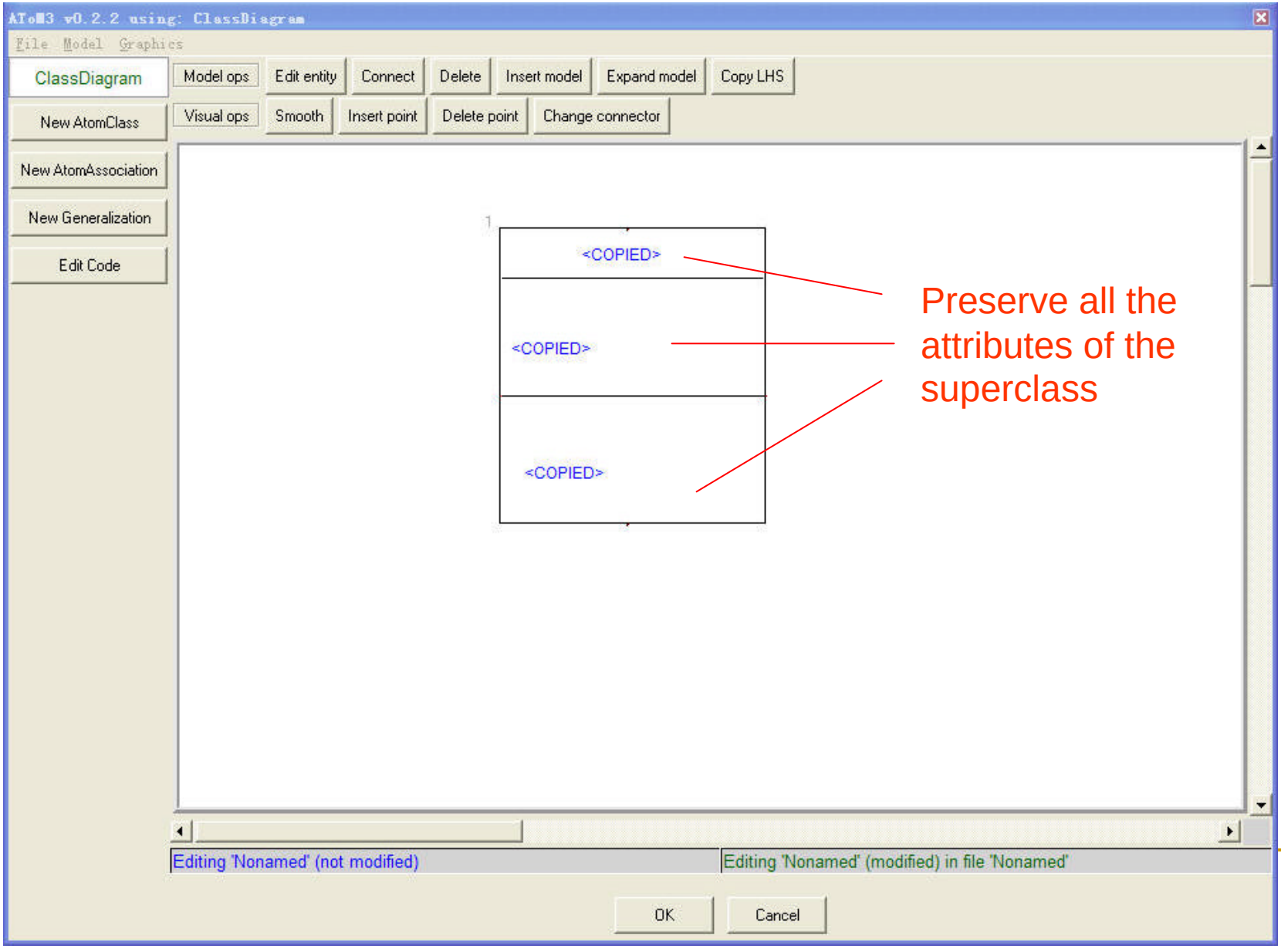
Collapse hierarchy is a composition transformation:

- Pull up field
- Pull up method
- The actual collapse rule

Collapse hierarchy rule (LHS)



Collapse hierarchy rule (RHS)



Collapse hierarchy rule condition code

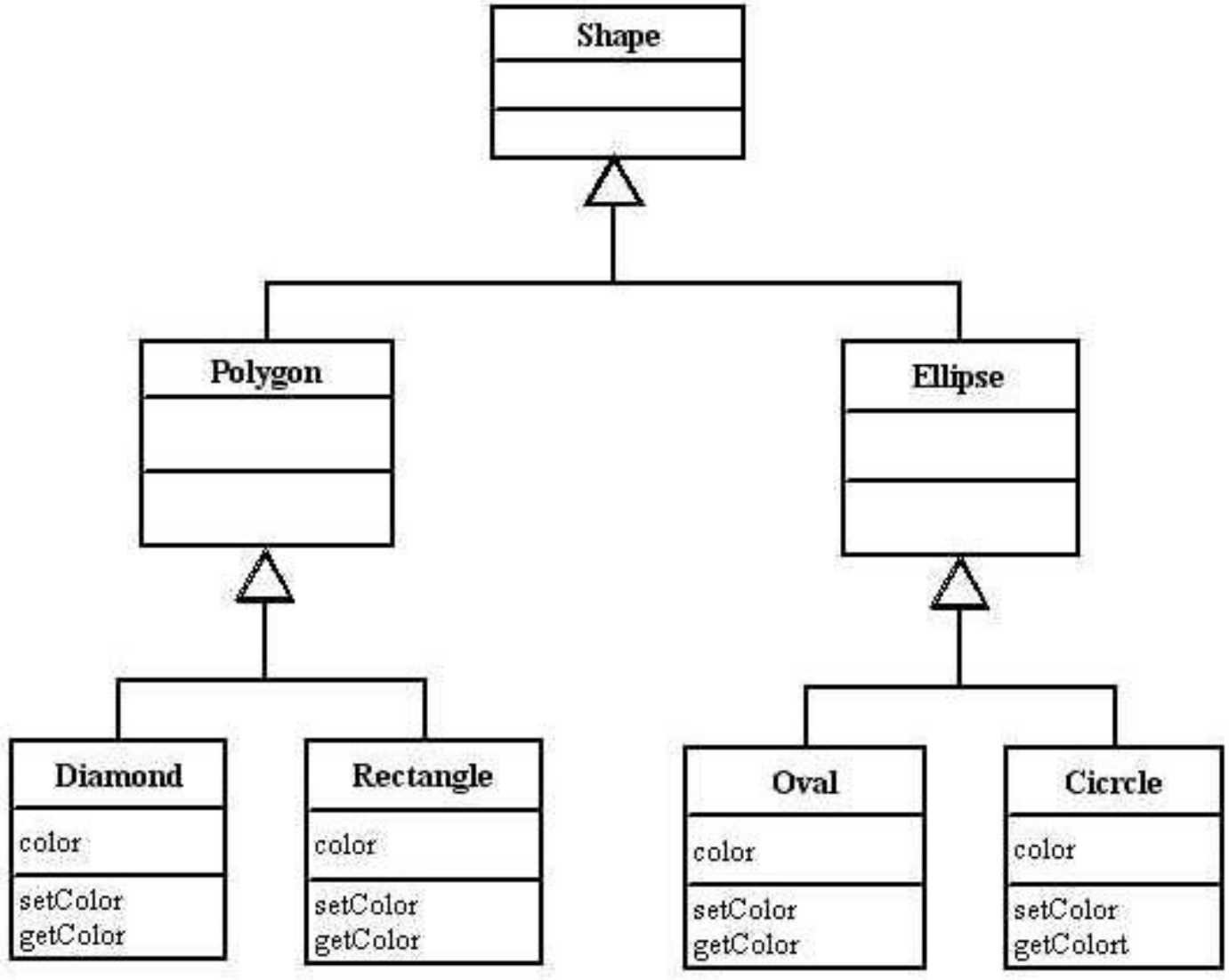
Before the actual collapse hierarchy rule be executed, it should check that the child class's attribute field and method field and other fields should be empty, in addition to that, the subclass should be the final class, in other word, the subclass should not have subclass. This is the precondition code to execute before apply the condition code:
this rule.

Collapse hierarchy rule condition code (cont'd)

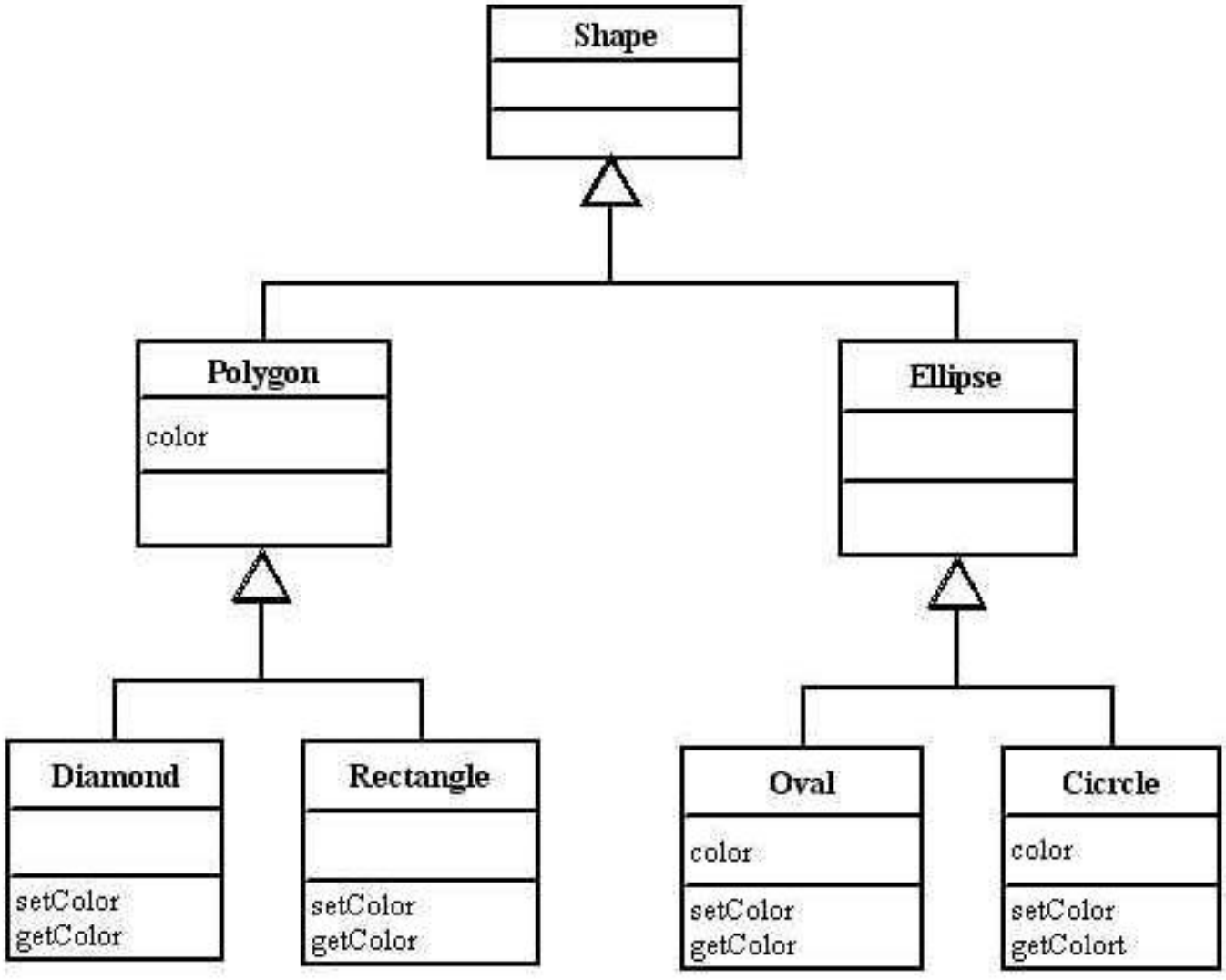
1. Enumerate all the entities in the matching pattern; if it a class then locate all the generalization relationships entities connected to this class entity, and according to the connection we can determine the type of the class, whether it's a superclass or a subclass.
2. Start another round of enumeration, if the entity we find is a subclass, and if the subclass attribute field or method field are not empty or if the subclass has subclass then, we return 0, end execution.
3. After the enumeration of all the entities in a graph, return 1

Load collapse hierarchy transformation and execute this transformation on the following class diagram graphical model

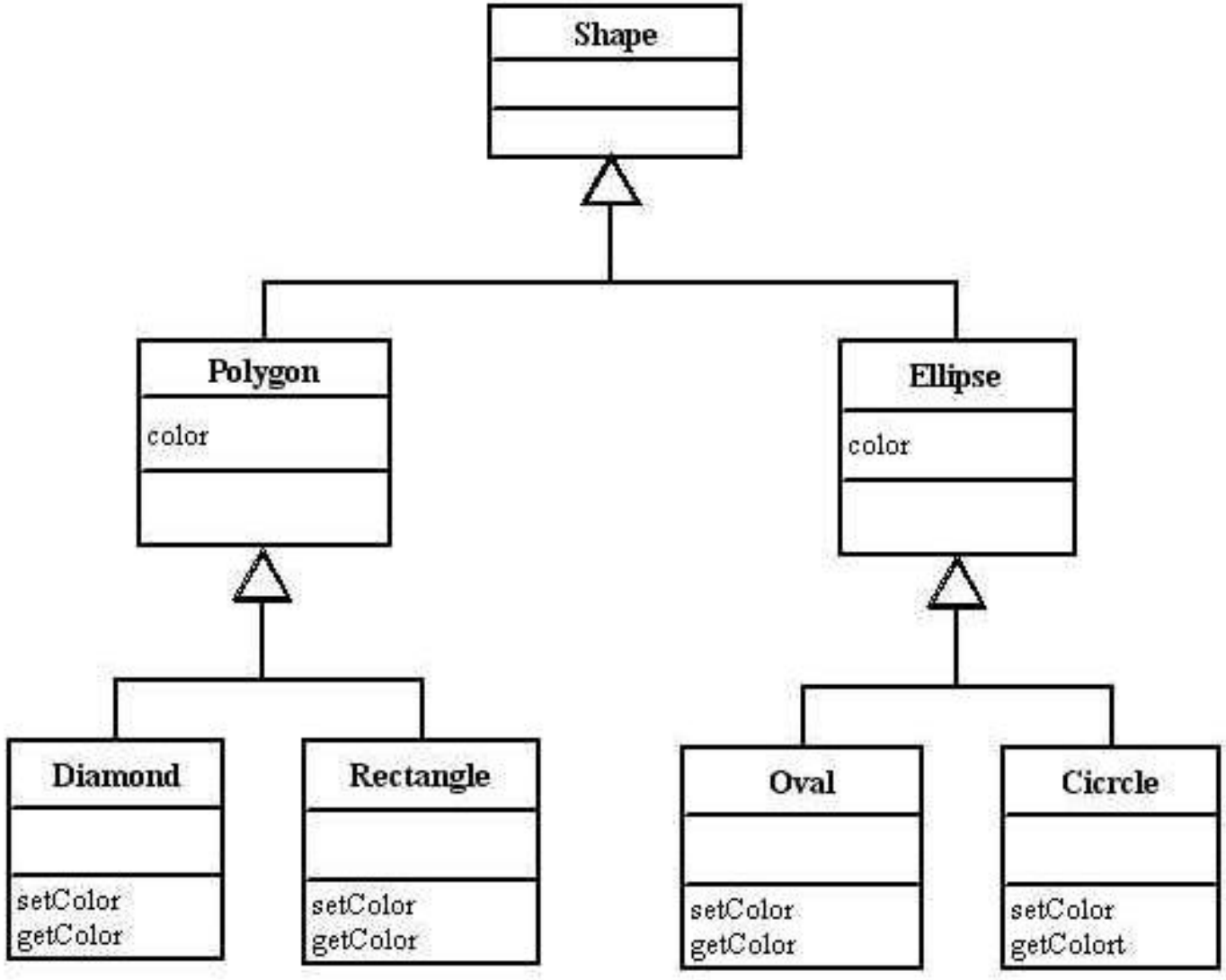
Example


Shape
Polygon
Ellipse
Diamond
color
setColor
getColor
Rectangle
color
setColor
getColor
Oval
color
setColor
getColor
Cicrcle
color
setColor
getColort

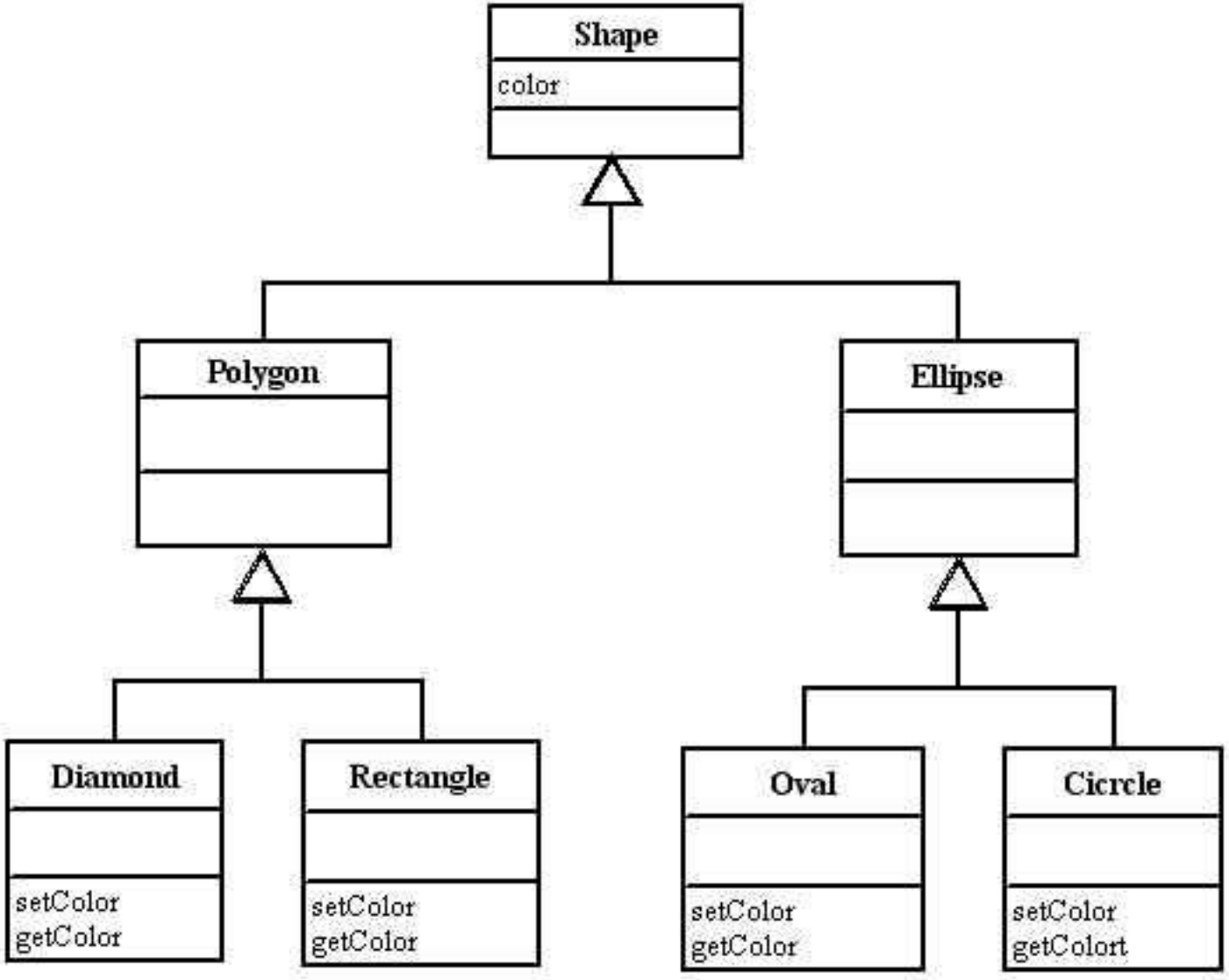
Example after pull up field step 1



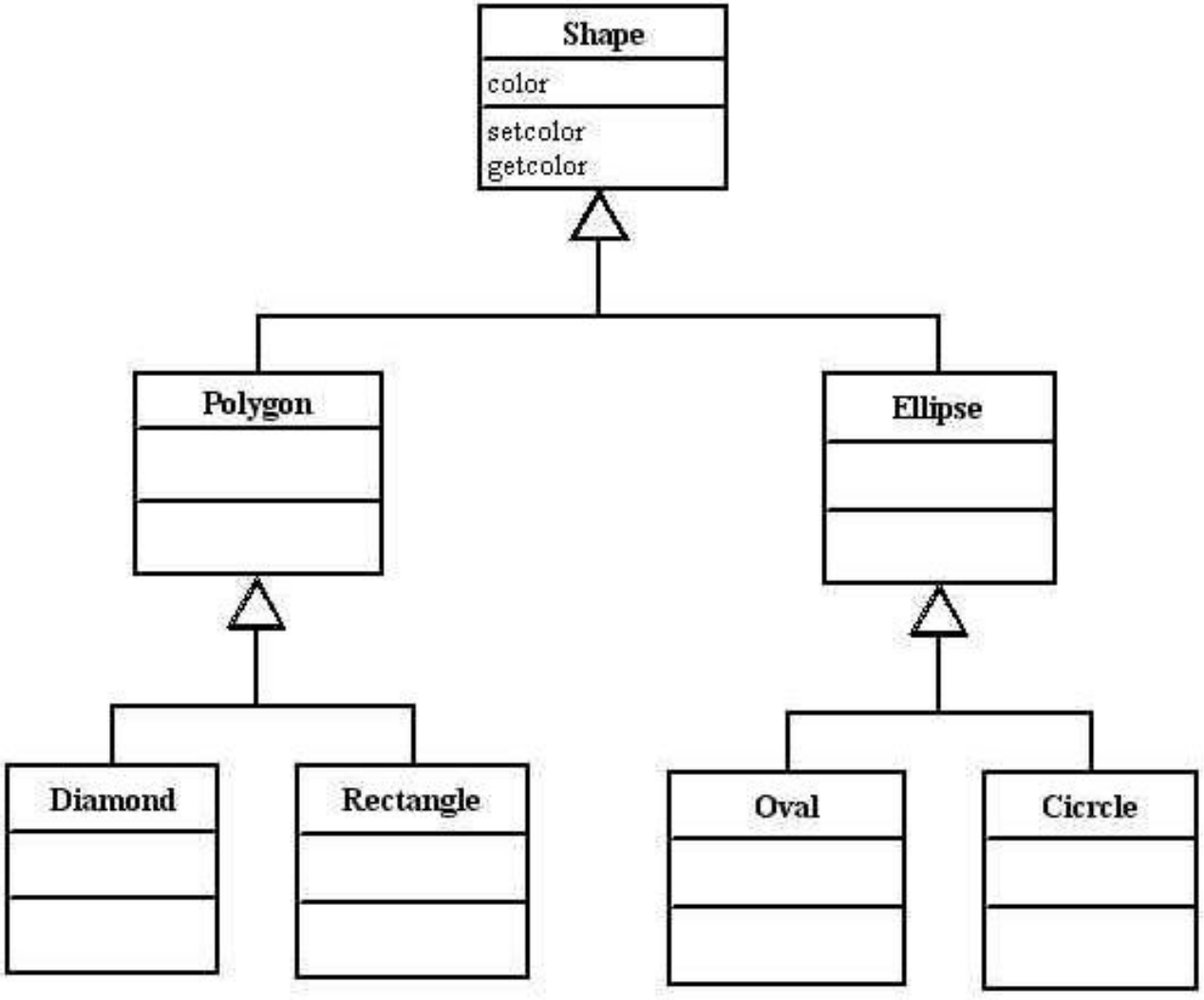
Example after pull up field step 2



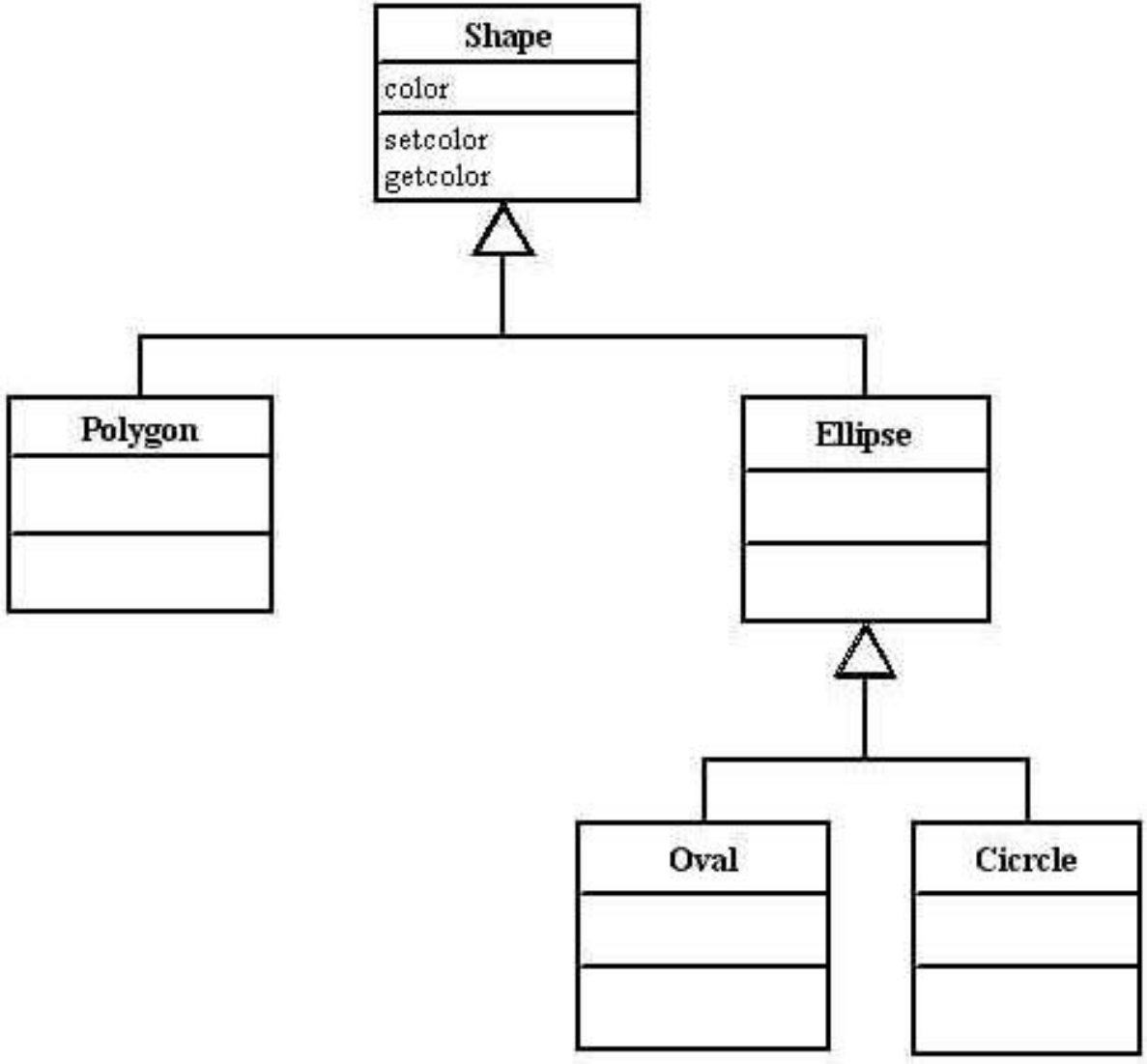
Example after pull up field step 3



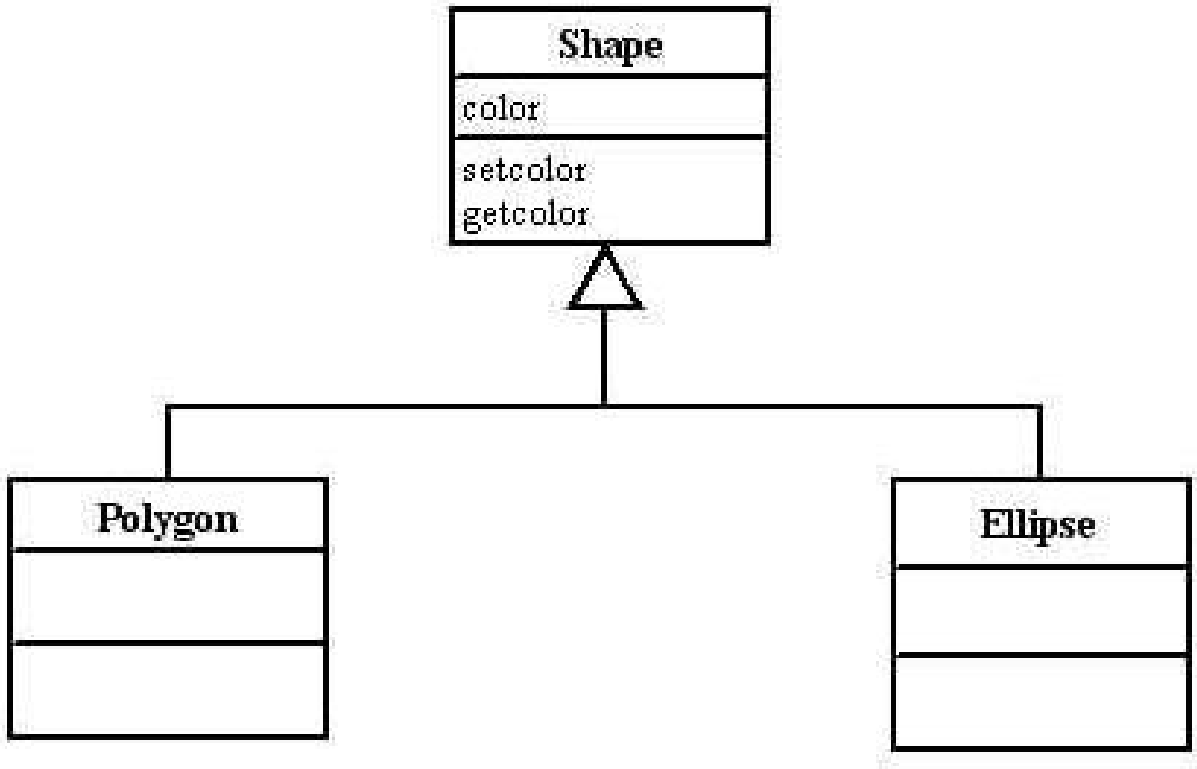
Example after pull up method (nearly the same three steps)



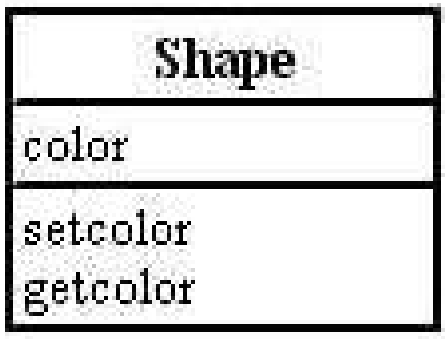
Example after collapse hierarchy step 1



Example after collapse hierarchy step 2



Example after collapse hierarchy step 3



Conclusion

- Refactoring is a common operation in software lifecycle and this small project intend to implement some refactoring tools in AToM^3 multi-formalism meta-modeling environment using graph rewriting.

- Some refactoring rules can’t be implemented using graph rewriting alone, since some refactoring can’t be completely automated, they need human interaction, a lot of things should be specified by a human being! (e.g. At lease need the specification of which matching formalism, i.e., LHS be substituted by RHS. It’s unsuitable to substitute every matching ones!)

References

1. Don Roberts, John Brant, and Ralph Johnson, A Refactoring Tool for Smalltalk, Department of Computer Science, University of Illinois at Urbana-Champaign
2. Refactoring object-oriented frameworks, William F. Opdyke, Department of Illinois at Urbana_Champaign, 1992
3. Opdyke, William F. “Refactoring Object-Oriented Frameworks.” Ph.D. diss., University of Illinois at Urbana-Champaign. Department of Computer Science, University of Illinois at Urbana-Champaign.
4. Introduction to graph grammar of AToM^3 http://moncs.cs.mcgill.ca/people/mprovost/tutorial_a/tut_a_main.html
5. Martin Fowler, Refactoring improving the design of existing code, Addison Wesley, 1999

Thanks!
Questions ?