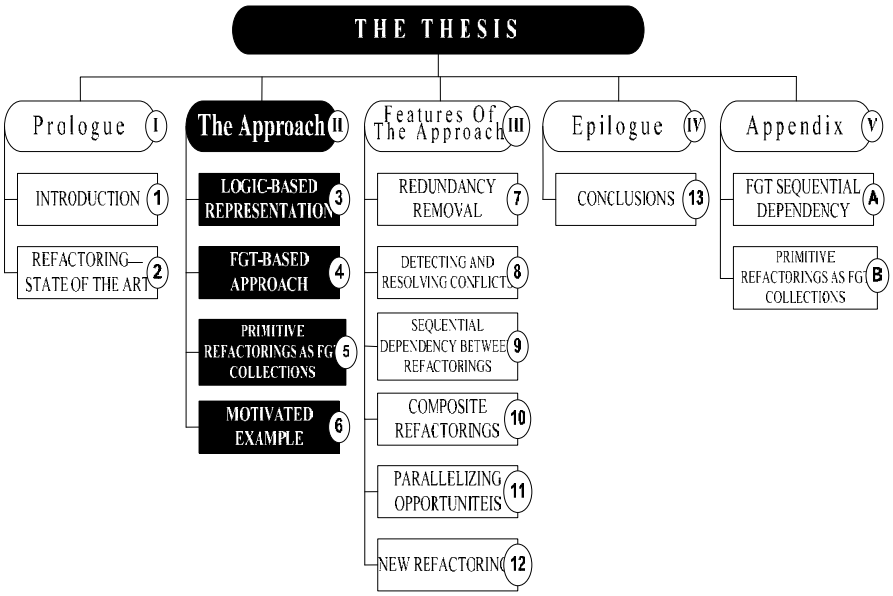
Part II

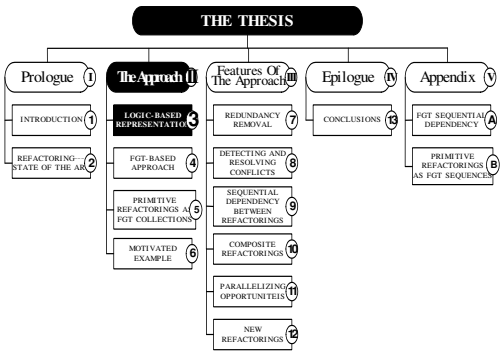
The Approach



CHAPTER 3

LOGIC-BASED REPRESENTATION

3.1 Introduction



In the proposed approach, the core abstract idea is to view refactorings of a system as FGTs, and then to transform the system in terms of these FGTs. As with author mentioned in the previous chapter, the refactoring envisaged here is at the level of the system's design. Ideally, to implement the core idea, a tool would be needed that can make user-requested refactorings on some computer-based description of the system's design. In principle, the tool could be written in any appropriate language, and the representation of the system would therefore have to be designed to match the requirements of that language.

For the purposes of the present study, a prototype tool has been built for experimental purposes. Because of its advanced search engine, and because of its overall suitability for prototyping, it was decided to build the tool in Prolog. A positive consequence of this decision is that many of the forthcoming explanations about the approach can be given by referring to the logic-terms that have been used as data for the Prolog prototype tool.

Moreover, it has been assumed that the system design is represented in standard UML [64]. The first challenge, therefore, is to represent the relevant elements of a UML class diagram as logic-terms. These logic-terms express the semantics of the standard UML modeling vocabulary. The vocabulary consists of a set of *objects* (packages, classes, attributes, methods and parameters) to represent discrete concepts in a class diagram. The vocabulary also contains a set of *relations* (extends, associations, reads, writes, calls, types) to relate the *object* elements in the UML class diagram to one another. The *object* and *relation* elements of concern here are related to the simplified UML meta-model shown in Figure 1.5. Extending the approach to represent other elements in the UML class diagram is straightforward.

In [35], a software refactoring tool called JTRANSFORMER is proposed. The tool represents the full detail of Java code as Prolog facts, and then executes refactorings by manipulating

these facts. The inspiration for representing relevant elements of UML class diagrams as logic-terms is based on the concepts described in the JTRANSFORMER tool.

It should be noted, however, that the information required to implement the full range of refactorings mentioned in the literature is not fully available in the UML class diagrams alone. Some refactorings require, in addition, basic *access-related* information—i.e. information that indicates *call* relationships between methods and *read* or *write* relationships between methods and attributes. Such information is not found at the UML class diagrams level, but will be available from sequence- and/or state diagrams, provided these are set up at the appropriate level of detail. Alternatively, it would be relatively easy to extract this information directly from the code. The design and implementation of software to do this extraction, whether directly from code or from representations of sequence- or state diagrams, is not considered further in this thesis. Instead, it is simply assumed that the required information is available.

This category of information is required because of the following two points:

a. It is needed to check preconditions of some refactorings. Refactoring precondition, as will be explained later, is important to ensure the behaviour preservation of the refactoring. For example, one of the precondition conjuncts of the primitive refactoring **deleteMethod** that is used to delete a specific method *Methn* from the class diagram is: "The method *Methn* should not be referenced (*called*) by any other *object* elements in the entire system". The information extracted from the class diagram alone is not sufficient to check such a condition. If the system has two classes A and *B* where a method in class *A calls* another method in class *B,* then, the UML class diagram may reflect an *association* relation between the two classes. However, the class diagram does not indicate the reason for the *association*.

b. Some refactorings involve a restructuring of this extra information without modifying anything in the class diagram itself. For example, a refactoring may be used to redirect direct access to a certain attribute through read and/or write methods instead (getter and/or setter). The class diagram is not affected by this refactoring, but the relations between the different members will be changed. The refactoring tool should keep track of such modifications because they are needed:

   - for future refactorings (to check preconditions, for example); or

   - to modify the code or *other* UML diagrams, such as state and sequence diagrams.

Note that the latter point suggests a theme that will not be pursued further in this thesis, namely the notion of keeping various representations of the system consistent with one another, where these may be the system's class diagrams, sequence diagrams, state diagrams, its code, etc. The focus of this thesis will remain on refactoring at the class diagram level, with the aforementioned exception related to access information, so as to address a relatively wide range of refactorings.

The set of logic-terms are accordingly classified into two groups, where each group corresponds to one of the two specific kinds of UML vocabularies. The first group is concerned with *object* elements of the UML class diagram. All the facts in this group are extracted directly from the UML class diagram of the system under consideration. Logic-terms of this group are:

- **package** logic-terms
- **class** logic-terms
- **method** logic-terms
- **attribute** logic-terms
- **parameter** logic-terms

The second group of the logic-terms is concerned with *relation* elements of the UML class diagram. Part of these logic-terms are extracted from the class diagram (*extends*, *association* and *type* relations), while the rest are assumed to have been extracted from the code-level implementation of the system (*read*, *write* and *call*). Logic-terms of this group are:

- **extends** logic-terms
- **association** logic-terms
- **read** logic-terms
- **write** logic-terms
- **call** logic-terms
- **type** logic-terms

The logic-terms of the system under consideration are represented as Prolog facts in the proposed refactoring tool. In the rest of the thesis, the concepts logic-term and Prolog fact will be regarded as exchangeable and have the same meaning.

In general, the first argument of each logic-term (fact) is a unique identifier for the model element (*object* or *relation*). The other arguments are properties of that element (name,

definition type and access mode) or are foreign identities for other model elements. In the following two sections (3.2 and 3.3), each group of logic-terms will be presented in detail.

3.2 Object Element Logic-Terms

This group of logic-terms includes all the logic-terms that are used to represent the *object* elements of the UML class diagrams. Logic-terms of this group are:

A. package(*PID, OwnerID, PName, CsList* **)** is used to represent *package* object elements of the UML class diagram. The description of arguments of the package logic-term is as follows:

- *PID* is the unique identifier of the package.
- *OwnerID* is the unique identifier of the container where the package is identified.
- *PName* is the name of the package.
- *CsList* is a list that contains the unique identifiers of all the classes defined in the package.

B. class(*CID, PID, CName, AccMode, MethsList, AttrsList* **)** is used to represent *class* object elements of the UML class diagram. The description of arguments of the class logic-term is as follows:

- *CID* is the unique identifier of the class.
- *PID* is the unique identifier of the package in which the class resides.
- *CName* is the name of the class.
- *AccMode* is the access mode of the class.
- *MethsList* is a list that contains the unique identifiers of all the methods defined in the class.
- *AttrsList* is a list that contains the unique identifiers of all the attributes defined in the class.

C. attribute(*AttrID, CID, AttrName, DefType, AccMode* **)** is used to represent *attribute* object elements of the UML class diagram. The description of arguments of the attribute logic-term is as follows:

- *AttrID* is the unique identifier of the attribute.
- *CID* is the unique identifier of the class where the attribute is identified.
- *AttrName* is the name of the attribute.
- *DefType* is the definition type of the attribute.

- *AccMode* is the access mode of the attribute.

Note: In the rest of the thesis, a distinction between two different definition types is made (*basic* and *complex* definition types) as follows:

a. Basic type: used when the definition type is basic (*int, float, etc*). It takes the following format:

type(*basic, Tname, Num*), *Num* >= 0. (Zero if the variable is not array)

- *basic* stands for basic types like *int*, *float*, *double*, *etc*.
- *Tname* is the type name (*int, float, etc*).
- *Num* stands for the dimension of an array. Zero is used for simple types (i.e. not array).

b. Complex type: used when the definition type is complex (*class*, *interface, etc*). It takes the following format:

type(*complex, ObjectID/ObjectName, Num*), *Num* >= 0.

- *complex* stands for complex types like *class* or *interface*.
- *ObjectID* is the unique identifier (ID) of that object (*class*, *interface, etc*). For example, if the definition type of an attribute *Attn* is a class *A* then this argument will be the ID of *A*. When the user specifies the definition type of an object then the user just enter the name of the object *ObjectName*. The tool then takes the responsibility of storing the ID of that object.
- *Num* stands for the dimension of an array. Zero is used for simple types (i.e. not array).

D. method(*MethID, CID, MethName, RetType, AccMode, PrmsList* **)** is used to represent *method* object elements of the UML class diagram. The description of arguments of the method logic-term is as follows:

- *MethID* is the unique identifier of the method.
- *CID* is the unique identifier of the class where the method is identified.
- *MethName* is the name of the method.
- *RetType* is the definition type of the return value of the method.
- *AccMode* is the access mode of the method.
- *PrmsList* is a list that contains the unique identifiers of all the parameters defined in the method. The order of these IDs represents the order of the parameters in the method.

E. parameter(*PrmID, MethID, PrmName, DefType* **)** is used to represent *parameter* object elements defined in methods of the UML class diagram. The description of arguments of the parameter logic-term is as follows:

- *PrmID* is the unique identifier of the parameter.
- *MethID* is the unique identifier of the method where the parameter is identified.
- *PrmName* is the name of the parameter.
- *DefType* is the definition type of the parameter.

3.3 Relation Element Logic-Terms

This group of logic-terms includes all the logic-terms that are used to represent the *relation* elements of the UML class diagrams. Each *relation* logic-term represents a specific *relation* that may exist between two object elements in the UML class diagram. All the *relation* logic-terms have the same arguments as the following:

***RelationType*(** *RID, Label, SourceID, DestinationID* **)**

Where

- *RID* is the unique identifier of the *relation*.
- *Label* is the label of the *relation*.
- *SourceID* is the unique identifier of the source *object* element of the *relation*.
- *DestinationID* is the unique identifier of the destination *object* element of the *relation*.

Logic-terms of this group are the following:

A. extends(*RID, Label, SourceID, DestinationID* **)** is used to represent an *extends* (generalization, specialization) relation that may exist between two *object* elements. For example, it may be used to represent the relation between two classes *A* and *B* where the first class *A* (with unique identifier *SourceID*) is the superclass of the second class *B* (with unique identifier *DestinationID*).

B. association(*RID, Label, SourceID, DestinationID* **)** is used to represent an *association* relation that may exist between two *object* elements. For example, it may be used to represent the relation between two classes *A* and *B* where the first class *A* (with unique identifier

SourceID) is the source of the relation and the second class *B* (with unique identifier *DestinationID*) is the destination of the relation.

C. read(*RID, _, SourceID, DestinationID* **)** is used to represent a *read* relation that may exist between two *object* elements. For example, it may be used to represent the relation between a method *Methn* and an attribute *Attn* where at the code-level one or more statements in the method *Methn* access the attribute *Attn* in a *read* mode. The method *Methn* (with unique identifier *SourceID*) is the source of the relation and the attribute *Attn* (with unique identifier *DestinationID*) is the destination of the relation.

D. write(*RID, _, SourceID, DestinationID* **)** is used to represent a *write* relation that may exist between two *object* elements. For example, it may be used to represent the relation between a method *Methn* and an attribute *Attn* where at the code-level one or more statements in the method *Methn* access the attribute *Attn* in a *write* mode. The method *Methn* (with unique identifier *SourceID*) is the source of the relation and the attribute *Attn* (with unique identifier *DestinationID*) is the destination of the relation.

E. call(*RID, _, SourceID, DestinationID* **)** is used to represent a *call* relation that may exist between two *object* elements. For example, it may be used to represent the relation between two methods *MethX* and *MethY* where at the code-level one or more statements in the method *MethX* call the method *MethY*. The method *MethX* (with unique identifier *SourceID*) is the source of the relation and the method *MethY* (with unique identifier *DestinationID*) is the destination of the relation.

F. type(*RID, _, SourceID, DestinationID* **)** is used to represent a *type* relation that may exist between two *object* elements. For example, it may be used to represent the relation between an attribute *Attn* and a class *C* where the definition type of the *Attn* is class *C*. The attribute *Attn* (with unique identifier *SourceID*) is the source of the relation and the class *C* (with unique identifier *DestinationID*) is the destination of the relation.

Note 1: The second argument *Label* in the logic-terms *read*, *write*, *call* and *type* is ignored. This is because the *read*, *write* and *call* relations do not appear in the original UML class diagram, and they just added to the logic representation of the system as extra information for refactoring purposes. In the case of the *type* relation, it simply does not have a label in the UML class diagram.

Note 2: At the code-level, if a method *Methn* accesses (*reads*) an attribute *Attn* more than once, then at the logic-based representation level these *reads* will be represented by just one *read*

relation from *Methn* to *Attn*. The same apply for the *write* and *call* relations.

3.4 Example

Figure 3.1(a) shows a UML class diagram for a simple system *SimpleSys*. The system has a package *D* with two classes *B* and *C* defined in the package.

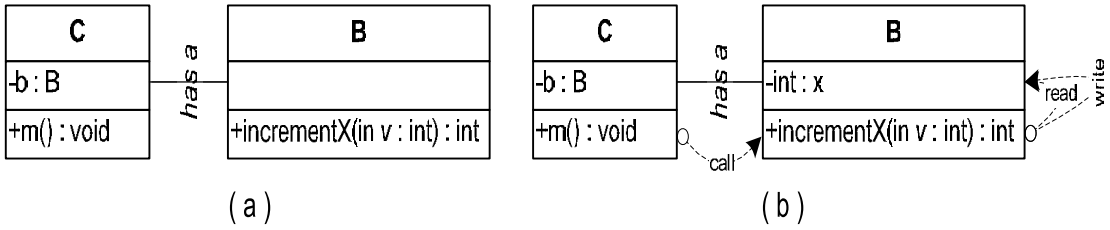


Figure 3.1: A simple UML class diagram of the *SimpleSys*

```
1.  package D;
2.  public class C {
3.    private B b;
4.    public void m(){
5.      System.out.println(b.incrementX(10));}
6.  }  //end of class C
7. class B{
8.      private int x;
9.      public int incrementX(int v){
10.     x=x+v;
11.     return x;}
12. } //end of class B
```

Figure 3.2: A code-level implementation of the *SimpleSys*

As mentioned in section 3.1, *access-related* information that describes the references between the different *object* elements in the UML class diagram is needed for refactoring. This information is extracted from the code-level of the system. For clarity, such information is represented as dashed arrows in Figure 3.1(b). Figure 3.2 shows the code-level implementation of the system from which this information is extracted. For simplicity, the main method in class *C* and the constructors in the different classes are omitted from the code.

In the following, a detailed explanation is given of where each one of the dashed arrows in Figure 3.1(b) is extracted:

- The *write* relation from the method *B.incrementX* to the attribute *B.x* (shown in Figure 3.1(b)) is extracted from line 10 of the code. The value of the attribute *B.x* is updated by the left side of the assignment statement ***x***= *x* + *v*. Representing this relation in the underlying logic-terms of the system indicates to the refactoring tool that the attribute *B.x* will be referenced (*updated*) by the code implemented in the method *B.incrementX*.

- The *read* relation from the method *B.incrementX* to the attribute *B.x* is extracted from line 10 of the code. The value of the attribute *B.x* is read by the right side of the assignment statement *x*= ***x*** + *v*. Representing this relation in the underlying logic-terms of the system indicates to the refactoring tool that the attribute *B.x* will be referenced (*read*) by the code implemented in the method *B.incrementX*.

- The *call* relation from the method *C.m* to the method *B.incrementX* is extracted from line 5 of the code. The method *B.incrementX* is called by the statement *b.incrementX(10)* which is implemented in the method *C.m*. Representing this relation in the underlying logic-terms of the system indicates to the refactoring tool that the method *B.incrementX* will be referenced (*called*) by the code implemented in the method *C.m*.

Figure 3.3 shows the list of logic-terms (*Prolog facts*) for the UML class diagram in Figure 3.1. For example, in the fact

class(2, 0, C, public, [2001], [20001]).

- The first argument represents the unique identifier of class C.
- The second argument represents the unique identifier of the container of class C, which is the package with unique identifier 0.
- The third argument is the name of the class.
- The forth argument is the access mode of the class C.
- The fifth argument is a list that contains the unique identifiers of all the methods defined in the class C. In this case, it is just one method with unique identifier 3001.
- The last argument is a list that contains the unique identifiers of all the attributes defined in the class C. In this case, it is just one attribute with unique identifier 30001.

```
package(0, 00, D, [1, 2]).
class(1, 0, B, default, [1001], [10001]).
method(1001, 1, incrementX, type(basic,int,0),  public, [100001]).
attribute(10001, 1, x, type(basic, int, 0), private).
parameter(100001, 1001, v, type(basic, int, 0)).
read(1000001, _, 1001, 10001).
write(1000002, _, 1001, 10001).
class(2, 0, C, public, [2001], [20001]).
method(2001, 2, m, type(basic, void, 0), public, []).
attribute(20001, 2, b, type(complex, 1, 0), private).
type(2000001, _, 20001, 1).
call(2000002, _, 2001, 1001).
association(2000003, has a, 2, 1).
```

Figure 3.3: Underlying logic representations of the *SimpleSys*

3.5 Reflection on this Chapter

There foregoing schema is used in the current thesis as a knowledge base to represent a UML-specified system that is manipulated by a prototype Prolog refactoring tool to refactor the system according to user-specified refactorings. The tool, therefore, contains Prolog rules to apply these refactorings. It also requires rules to check that preconditions of refactorings are satisfied before their application can be attempted. Of course, in the present thesis, all of the refactorings happen in terms of FGTs, and the tool has been designed to operate precisely at this FGT-level. The forthcoming chapters will elaborate further on these themes.

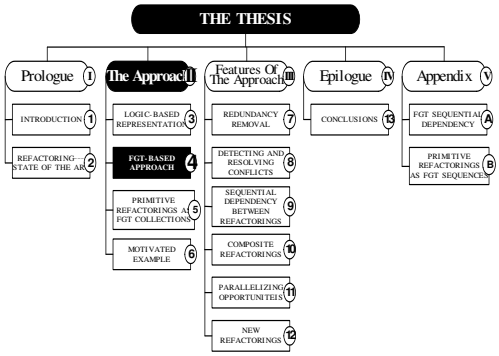
However, it might be noted in passing that the schema given above could also be used as a basis for issuing queries about a UML system—for example, for finding all classes that have a certain characteristic in the system. While this theme will not be further explored in this thesis, it appears to be a peripheral contribution of the thesis that could conceivably be exploited in developing such a Prolog-based application.

CHAPTER 4

FGT-BASED APPROACH

4.1 Introduction

The main focus of this chapter is to give detailed explanations and descriptions of the set of FGTs to represent and construct any refactorings in the proposed refactoring tool. This chapter is the ground base for the remaining chapters. Each of those later chapters may be read independently, provided that the reader is familiar with the contents of this chapter, the rest of which is organized as follows.



In section 4.2 the concept FGT is described, and details the two types of FGTs (*Object Element* and *Relational Element* FGTs) are given. Full details about the format, implementation, and the set of precondition conjuncts for each FGT are given.

In section 4.3 an algorithm is introduced that is used to allocate the collection of the FGTs that are related to one refactoring in a data structure called an FGT Directed Acyclic Graph (FGT-DAG). Since the algorithm accounts for the sequential dependencies that may occur between the different FGTs in the refactoring, the section also provides a detailed explanation of the sequential dependencies between the different FGTs.

In section 4.4 the relationship between the set of FGTs and primitive—as well as composite—refactorings is discussed. The section describes a vision in which the proposed set of FGTs constitutes the core of the refactoring system, and suggests new terminology for describing refactoring precondition.

4.2 Fine-Grain Transformations (FGTs)

An FGT is an *abstract operation* on a UML model—i.e. a UML model will always be one of the implicit operands of an FGT, and this model will always undergo an incremental *atomic*

change as a result of applying an FGT to it. The change can be regarded as atomic in the sense that it cannot be broken down into further smaller change steps from the modeling perspective. The operation is abstract in the sense that it could be specified in a wide variety of concrete syntactic representations.

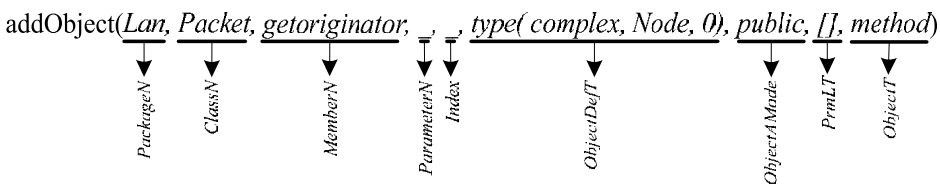
Throughout this thesis, a concrete syntax that resembles Prolog predicates will be used to specify FGTs. This choice of concrete syntax was made to support the Prolog prototype refactoring tool that has been built to illustrate the various ideas. As described in the previous chapter, the UML class diagram is itself stored as a set of facts in the Prolog database. As will be seen below, the concrete syntax of each FGT has to uniquely identify the various components of the UML class diagram that are to change, and it also has to indicate the nature of the change. In general, the nature of the change is encapsulated in the name of the FGT, and the UML components that are affected are specified as arguments of it.

The set of FGTs that have been identified are closely related to the vocabulary and semantics of standard UML mentioned in the previous chapter and they are accordingly classified into the two groups used in chapter 3.

The first group is concerned with all the transformation operations whose characterising operands are *object* elements of the UML class diagram. In the rest of the thesis, these FGTs are called *Object Element FGTs.* FGTs of this group are:

- **addObject** FGT: used to add *object* elements to the class diagram.
- **renameObject** FGT: used to change the name of an *object* element.
- **changeOAMode** FGT: used to change the access mode of an *object* element.
- **changeODefType** FGT: used to change the definition type of an *object* element.
- **deleteObject**: used to delete *object* element from the class diagram.

As an example of FGTs in this group, the following FGT is used to add to the class diagram an *object* element with name *getoriginator* and access mode *public.* It is to be added to the class *Packet* that is in the package *Lan*. The *object* will return one value of type *Node* class. The last argument of the FGT tells the tool that the added *object* in this FGT is of type *method*. The empty list *PrmLT* indicates that the added method will have no parameters.



After applying this FGT, the following fact will be added to the underlying database of facts that represents the class diagram of the system under consideration:

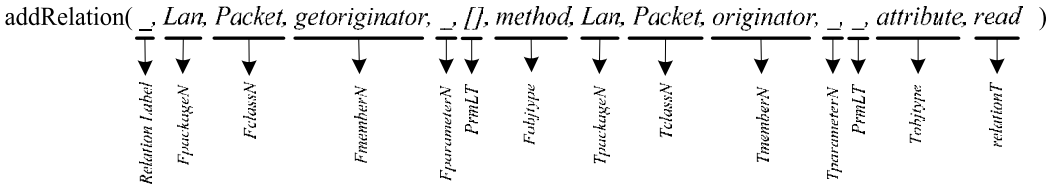
method(46, 2, getoriginator, type(complex,1, 0), public, []).

Note that from the information presented in section 3.2, number 46 will be the unique identifier of the new method. Number 2 is the unique identifier of the class *Packet* where the new method will be defined. Number 1 in the term *type* is the unique identifier of the definition type of the return value of the method, which is in this case the class *Node*.

The second group of FGTs is concerned with all the transformation operations that work on *relational* elements of a UML class diagram. These FGTs will be called *Relational Element FGTs*. FGTs of this group are:

- **addRelation** FGT: used to add a *relational* element between two *object* elements.
- **renameRelation** FGT: used to change the label of a *relational* element.
- **deleteRelation** FGT: used to delete a *relational* element that exists between two *object* elements.

As an example of FGTs in this group, the following FGT is used to add a *read* relation from the method *Lan.Packet.getoriginator* to the attribute *Lan.Packet.originator*.



After applying this FGT, the following fact will be added to the underlying database facts that represent the class diagram of the system under consideration:

read(47, _, 46, 2002).

Number 47 will be the unique identifier of the new *read* relation. Number 46 is the unique identifier of the source *object* of the relation, which is *Lan.Packet.getoriginator* method. Number 2002 is the unique identifier of the destination *object* of the relation, which is *Lan.Packet.originator* attribute. As mentioned in section 3.3, the label for the *read*, *write*, *call* and *type* relations is omitted.

Each FGT of the two groups has a set of precondition conjuncts (i.e. X and Y and Z and …) that need to be satisfied by the system in order to consider it as a legal transformation operation. In some cases, one or more of these conjuncts is itself a number of disjuncts (i.e. (X or Y)). A procedure called **FGTPrecondConj**(*FGT*) is implemented in the refactoring tool for each one of the proposed FGTs. FGTs precondition conjuncts will play an important role in preserving the behaviour of the system at the time of refactoring, as will be shown in section 4.4. For example, in order to apply the FGT:

addObject(*Lan, Packet, getoriginator, _, _, type(complex, Node, 0), public, [], method*)

The underlying system should have a class with name *Packet* in the package *Lan*; and this class should not contain a method *getoriginator* with empty parameter list. The method *getoriginator* should also not be inherited from any of the ancestors of class *Lan.Packet*. In addition, the return definition type of the method should be valid and accessible. The access mode of the created method should also be valid. The precondition conjuncts for this FGT, as implemented in the prototype tool, are specified as follows:

FGTPrecondConj(addObject(*Pn, Cn, Methn, _, _, ODefT, OAMode, PrmLT, method*) **):-**

existsObject(*Pn, Cn, class*),
not(existsObject(*Pn, Cn, Methn, PrmLT, method*)),
not(isInherited(*Pn, Cn, Methn, PrmLT, method*)),
validDefType(*ODefT*),
canAccessType(*ODefT*),
validOAMode(*OAMode, method*).

Note that the comma (,) between the two conjuncts retains the Prolog semantics of a "*logical and*" between two rules. As another example, in order to apply the FGT

addRelation(_,*Lan, Packet, getoriginator, _, [], method, Lan, Packet, originator, _, _, attribute, read*)

The underlying system should have the method *Lan.Packet.getoriginator* and the attribute *Lan.Packet.originator*. The system may not already have a *read* access between the method *Lan.Packet.getoriginator* and the attribute *Lan.Packet.originator*. In addition, the location of the source *object Lan.Packet.getoriginator* and the destination *object Lan.Packet.originator* in the model together with the access mode of the destination *object Lan.Packet.originator* play an important role in determining the applicability of the previous addRelation FGT. The

precondition conjuncts for this FGT, as implemented in the prototype tool, are specified as follows:

FGTPrecondConj(addRelation(_, *FPn, FCn, FMethn, _,FPrmLT, method, TPn, TCn, TAttn, _, _, attribute, RelT*) **):-**

existsObject(*FPn, FCn, FMethn, FPrmLT, method*),
existsObject(*TPn, TCn, TAttn, attribute*),
not(existRelation(_,*FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TAttn, attribute,RelT*)) ,
[(objectAMode(*TPn, TCn, TAttn, attribute, private*), *FPn.FCn=TPn.TCn*) |
(objectAMode(*TPn, TCn, TAttn*, *attribute, default*), *FPn=TPn*) |
(objectAMode(*TPn, TCn, TAttn, attribute, protected*), (subClass(*FPn,FCn, TPn, TCn*) |
FPn=TPn)) | objectAMode(*TPn, TCn, TAttn, attribute, public*)]**.**

Note that the comma (|) between the two conjuncts retains the Prolog semantics of a "*logical or*" between two rules.

A detailed explanation of the addRelation's precondition conjuncts, as well as those of all the other FGTs, will be discussed later in this section. In the following two subsections 4.2.1 and 4.2.2, each group of FGTs together with their set of precondition conjuncts will be presented in detail.

The presentation of each FGT will be in the following style. Firstly, the format of the Prolog term used to represent that FGT in the system is explained. The explanation includes an explanation of each of the term's arguments. Then the Prolog rule used to check the preconditions of the FGT is given. If F represent the Prolog term of some FGT, then this precondition rule has the general form:

FGTPrecondConj(F) :- C1, C2, … Cn.

where C1 … Cn are Prolog terms (containing arguments suitably derived from the arguments of F) representing the n precondition conjuncts that need to be checked against the existing system description. English narrative is given alongside these terms to explain what their meaning is. The Prolog rules used to check the truth-value of the terms C1, … Cn are not discussed here, but are fairly straightforward. Similarly, the postconditions resulting from applying each FGT to the system under consideration are not explicitly stated, but may easily be inferred from the nature of the FGT: the relevant object or relation has been added / renamed / deleted; or the access mode or definition type of an object element has been changed. In predicate logic, these could typically be represented by formulae which assert the

existence of some object or relation that previously did not exist, and / or the non-existence of some object or relation that previously existed. In Prolog, these postconditions are operationally realised by the insertion into, or deletion from the Prolog database of relevant logical terms (as discussed in Chapter 3) representing these objects and relations.

In terms of the classical notation for total correctness proposed by Hoare, the axiomatic semantics of FGT F whose precondition is C1 ^ … ^ Cn, and whose postcondition is P1 ^ … ^ Pm could be given as:

{C1 ^ C2 ... ^ Cn} F {P1 ^ P2 … ^ Pm}

i.e. if C1 and C2 and … Cn are true (of the system under consideration) before applying F to it, and F is applied to this system, then F will terminate and P1 and P2 and … Pm will be true. Although the axiomatic semantics of the various FGTs are not explicitly provided below, they are all easily derivable from the information given.

The question may be asked: is the definition of each FGT sound in the sense that its underlying axiomatic semantics correctly specifies what is intended? For example, have all the precondition conjuncts C1, C2 … Cn been correctly identified to add / delete / rename the relevant object or relation according to the rules of the language in question (in the present case, UML representing an underlying Java system)? A formal proof of this kind of soundness is beyond the scope of this thesis. Under the circumstances, the best that could be done was to manually check the soundness of each FGT. While this does not, of course, guarantee soundness, it is hoped that the explicit provision of preconditions given below will allow others to scrutinise the axiomatic semantics for the type of soundness mentioned above.

Similarly, the question may be asked: is the class of FGTs provided in the forthcoming sections complete in the sense that no other possible FGTs can be defined? Again, there does not seem to be any easy way of formally guaranteeing this. Later in this chapter, it will be seen, however, that the class of FGTs defined is sufficient for building all commonly known primitive refactorings. In this sense, the class of FGTs defined below can be said to be complete.

4.2.1 Object Element FGTs

This group of FGTs includes all FGTs that are used to manipulate *object* elements of a UML class diagram. (Recall that *object* elements in the simplified UML meta-model include classes, methods, attributes and parameters.) By using these FGTs, the developer can add, rename, change access mode, change definition type, or delete *object* elements from a UML class diagram.

4.2.1.1 addObject FGT

The addObject FGT is used to add *object* elements to the UML class diagram. It is used to add class, method, attribute, parameter *object* elements to the class diagram. In general, it takes the following format:

addObject(*Pn, Cn, Memn, Prmn, Index, ODefT, OAMode, PrmLT, OT*)

where

- *Pn* is the name of the package. It is used when the *object* to be added is a package, class, method, attribute or parameter.
- *Cn* is the name of the class. It is used when the *object* to be added is a class, method, attribute or parameter.
- *Memn* is the name of the member (method or attribute). It is used when the *object* to be added is a method, attribute or parameter.
- *Prmn* is the name of the method's parameter. It is used when the *object* to be added is a parameter.
- *Index* is the index (order) of the parameter in the method's parameter list. It is used when the *object* to be added is a parameter.
- *ODefT* is the *object* definition type. It is used when the *object* to be added is a method, attribute or parameter. If the *object* is a method, then *ODefT* refers to the return type of that method. If the *object* is an attribute or parameter, then *ODefT* refers to the definition type of the attribute or parameter.
- *OAMode* is the access mode of the *object* (*public, protected, default, private*).
- *PrmLT* is the list of method's parameters. It is used when the *object* to be added is a method or parameter. If it is a method then the list *PrmLT* will contain all the parameters that are declared in the method. These parameters will be ordered in the list according to their definition order in the method arguments. Each element in the list is represented by the pair (*Prmn, PrmDefT*) where *Prmn* is the name of the parameter and *PrmDefT* is the definition

type of that parameter. If the *object* to be added is a parameter then the list *PrmLT* will contain the definition type of all parameters that are declared in the method. These definition types will be ordered in the list according to the order of their associated parameters in the method arguments. In this case, the list *PrmLT* is used to specify the signature of the method. For the rest of the thesis, a method's signature is specified by the method name together with its associated *PrmLT*.

- *OT* is the type of the *object* (*class, method, attribute,* or *parameter*).

The set of arguments and precondition conjuncts that are used for the FGT addObject are dependent on the type of *object* element that is to be added to the UML class diagram using that FGT, as shown below:

A. addObject(*Pn, Cn, _, _, _, _, OAMode, _, class***)**

As indicated in the last argument, this FGT is used to add a new class *Cn* in the package *Pn* with access mode *OAMode*. The new class will be empty and standalone. Empty means that, it has no members (attributes or methods). Standalone means, that it has no superclass or subclasses. All the members and super- or subclass *relations* will be added to the new class at a later stage.

To apply this FGT on the underlying system the following should hold.

- The package *Pn* should be already declared in the system.
- The class name (*Cn*) should be distinct from those all classes declared in the package *Pn*.
- The access mode *OAMode* should be a valid access mode.

FGTPrecondConj(addObject(*Pn, Cn, _, _, _, _, OAMode, _, class*) **):-**

existsObject(*Pn, package*), not(existsObject(*Pn, Cn, class*)), validOAMode(*OAMode, class*).	1. *Package Pn declared in the system.* 2. *Class Pn.Cn not declared in the system.* 3. *The access mode OAMode is a valid access mode for classes.*

B. addObject(*Pn, Cn, Methn, _, _, ODefT, OAMode, PrmLT, method***)**

As indicated in the last argument, this FGT is used to add a new method *Methn* with a parameter list *PrmLT* in the class *Pn.Cn*. The new method will have an access mode *OAMode* and a return type defined by the argument *ODefT*.

To apply this FGT on the underlying system the following should hold.

- The class *Pn.Cn* should be already declared in the system.
- The signature of the method should be distinct from those all methods declared in the class *Pn.Cn* or any of its ancestor classes.
- The access mode *OAMode* and the definition type of the return value *ODefT* should be valid.
- The type of the return value *ODefT* should be accessible.

Note that the second precondition conjunct means that the method should not be inherited by the class *Pn.Cn* from one of its ancestors. This condition is used to avoid redefining inherited members. Adding a member *x* in a class *A* while it is defined in *A's* ancestors will redefine the member *x* in the class *A* and all descendants of *A* because they will use the new version of *x*, and this will therefore change the behaviour of the system. On the other hand, adding a member *x* in a class *A* while it is defined in *A's* descendants will not change the definition of *x* in *A's* descendant classes. The behaviour of the system will therefore not change.

The last precondition conjunct is important when the type of the return value is complex (not basic). For example, if the return value is of type class, then the access mode of that class should be accessible.

FGTPrecondConj(addObject(*Pn, Cn, Methn, _, _, ODefT, OAMode, PrmLT, method*))**:-**

existsObject(*Pn, Cn, class*),	1. *Class Pn.Cn declared in the system.*
not(existsObject(*Pn, Cn, Methn, PrmLT, method*)),	2. *Method Pn.Cn.Methn with PrmLT not declared in the system.*
not(isInherited(*Pn, Cn, Methn, PrmLT, method*)),	3. *Method Methn with PrmLT not declared in any Pn.Cn's ancestor classes.*
validDefType(*ODefT*),	4. *The return definition type of the method is valid.*
validOAMode(*OAMode, method*).	5. *The access mode OAMode is valid.*
canAccessType(*ODefT*)	6. *The return definition type of the method is accessible.*

C. addObject(*Pn, Cn, Attn, _, _, ODefT, OAMode, _, attribute*)

This FGT, as indicated above, is used to add a new attribute *Attn* in the class *Pn.Cn* with access mode *OAMode*. The type of the new attribute is defined by the argument *ODefT*.

To apply this FGT on the underlying system the following should hold.

- The class *Pn.Cn* should be already declared in the system.
- The attribute name *Attn* should be distinct from those all attributes declared in the class *Pn.Cn* or any of its ancestor classes.
- The access mode *OAMode* and the definition type ODefT should be valid.
- The definition *type ODefT* should be accessible.

FGTPrecondConj(addObject(*Pn, Cn, Attn, _, _, ODefT, OAMode,_, attribute*) **):-**

existsObject(*Pn, Cn, class*),	1. *Class Pn.Cn declared in the system.*
not(existsObject(*Pn, Cn, Attn, attribute*)),	2. *Attribute Pn.Cn.Attn not declared in the system.*
not(isInherited(*Pn, Cn, Attn, attribute*)),	3. *Attribute Attn not declared in any of Pn.Cn's ancestor classes.*
validDefType(*ODefT*),	4. *The definition type ODefT is valid.*
validOAMode(*OAMode, attribute*),	5. *The access mode OAMode is valid.*
canAccessType(*ODefT*)).	6. *The definition type ODefT is accessible.*

D. addObject(*Pn, Cn, Methn, Prmn, Index, ODefT, _, PrmLT, parameter*)

This FGT as indicated from the last argument is used to declare a new parameter *Prmn* in the method *Pn.Cn.Methn* with *PrmLT*. The type of the new parameter is defined by the argument *ODefT*. The new parameter will be added at the index *Index* of the list of the method parameters. If *Index* is occupied by another parameter *x*, then parameter *x* and all the subsequent parameters will be shifted one-step to the right.

To apply this FGT on the underlying system the following should hold.

- The method *Pn.Cn.Methn* with *PrmLT* should be already declared in the system.
- The parameter *Prmn* may not already be declared in the method *Pn.Cn.Methn* with *PrmLT*.
- After adding the parameter *Prmn* to the list of parameters of the method *Methn* with *ParmLT,* the method *Methn* with the modified parameters list *ParmALT* should not be declared in the class *Pn.Cn* or any of its ancestor classes.

- The definition type *ODefT* should be valid and accessible.

FGTPrecondConj(addObject(*Pn, Cn, Methn, Prmn, Index, ODefT, _, PrmLT, parameter*) **):-**	
existsObject(*Pn, Cn, Methn, PrmLT, method*),	1. *The method Pn.Cn.Methn with PrmLT declared in the system.*
not(existsObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter*)),	2. *The parameter Prmn not declared in the method Pn.Cn.Methn with PrmLT.*
not(existsObject(*Pn, Cn, Methn, ParmALT, method*)),	3. *The method Pn.Cn.Methn with ParmALT should not be declared in the class Pn.Cn , where ParmALT is the result parameter list type for the method Pn.Cn.Methn after adding Prmn to it.*
not(isInherited(*Pn, Cn, Methn, PrmALT, method*)),	4. *Method Methn with PrmALT not declared in any of Pn.Cn's ancestor classes.*
validDefType(*ODefT*),	5. *The definition type ODefT is valid.*
canAccessType(*ODefT*)).	6. *The definition type ODefT is accessible.*

4.2.1.2 renameObject FGT

This FGT is used to change the name of one of the *object* elements that is already declared in the class diagram. It takes the following format:

renameObject(*Pn, Cn, Memn, Prmn, PrmLT, OT, ONewN*)

Where:

- *ONewN* is the new name of the *object*.

Note: For the description of the other arguments in the FGT, review section 4.2.1.1.

The set of arguments and precondition conjuncts that are used for the FGT renameObject are dependent on the type of *object* element that is to be renamed, as shown below:

A. renameObject(*Pn, Cn, _, _, _, class, ONewN*)

This FGT is used to change the name of the class *Pn.Cn* to another name *Pn.ONewN*.

To apply this FGT on the underlying system, the class *Pn.Cn* should be already declared in the system and the name *ONewN* should be distinct from those all classes declared in the package *Pn*.

FGTPrecondConj(renameObject(*Pn, Cn, _, _, _, class, ONewN*) **):-**

existsObject(*Pn, Cn, class*), not(existsObject(*Pn, ONewN, class*)).	1. *Class Pn.Cn declared in the system.* 2. *Class Pn.ONewN not declared in the system.*

B. renameObject(*Pn, Cn, Methn,_, PrmLT, method, ONewN*)

This FGT is used to change the name of the method *Pn.Cn.Methn* with *PrmLT* to another name *Pn.Cn.ONewN***.**

To apply this FGT on the underlying system, the method *Pn.Cn.Methn* should be already declared in the system and the name *ONewN* should be distinct from those all methods with *PrmLT* that are declared in the class *Pn.Cn* or any of its ancestor classes.

FGTPrecondConj(renameObject(*Pn, Cn, Methn, _, PrmLT, method, ONewN*) **):-**

existsObject(*Pn, Cn, Methn, PrmLT, method*), not(existsObject(*Pn, Cn, ONewN, PrmLT, method*)), not(isInherited(*Pn, Cn, Methn, PrmLT, method*)).	1. *Method Pn.Cn.Methn with PrmLT declared in the system.* 2. *Method Pn.Cn.ONewN with PrmLT not declared in the system.* 3. *Method Methn with PrmLT not declared in any of the Pn.Cn's ancestor of classes.*

C. renameObject(*Pn, Cn, Attn, _, _, attribute, ONewN*)

This FGT is used to change the name of the attribute *Pn.Cn.Attn* to another name *Pn.Cn.ONewN***.** To apply this FGT on the underlying system, the attribute *Pn.Cn.Attn* should be already declared in the system and the name *ONewN* should be distinct from those all attributes that are declared in the class *Pn.Cn* or any of its ancestor classes.

FGTPrecondConj(renameObject(*Pn, Cn, Attn, _, _, attribute, ONewN*) **):-**

existsObject(*Pn, Cn, Attn, attribute*), not(existsObject(*Pn, Cn, ONewN, attribute*)),	1. *Attribute Pn.Cn.Attn declared in the system.* 2. *The attribute Pn.Cn.ONewN not declared in the system.*

not(isInherited(*Pn, Cn, Attn, attribute*)).	3. *Attribute Attn not declared in any of the Pn.Cn's ancestor of classes.*

D. renameObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter, ONewN*)

This FGT is used to change the name of the parameter *Prmn* that is declared in the method *Pn.Cn.Methn* with *PrmLT* to another name *Pn.Cn.Methn.ONewN*.

To apply this FGT on the underlying system, the method *Pn.Cn.Methn* should be already declared in the system and the name *ONewN* should be distinct from those all parameters that are declared in the method *Pn.Cn.Methn* with *PrmLT*.

FGTPrecondConj(renameObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter, ONewN*) **):-**

existsObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter*), not(existsObject(*Pn, Cn, Methn, ONewN, PrmLT, parameter*)).	1. *Parameter Prmn declared in the method Pn.Cn.Methn with PrmLT.* 2. *The parameter ONewN not declared in the method Pn.Cn.Methn with PrmLT.*

4.2.1.3 changeOAMode FGT

This FGT is used to change the access mode (*public, protected, default* and *private*) of class, method, or attribute *object* elements from one mode to another. It cannot be applied to the parameter *object* elements, because there is no access mode for these elements in the class diagram. The FGT takes the following format:

changeOAMode(*Pn, Cn, Memn, Prmn, PrmLT, OT, OOldAM, ONewAM*)

Where:

- *OOldAM* is the old access mode of the *object* element.
- *ONewAM* is the new access mode of the *object* element.

Note: For the description of the other arguments in the FGT, review section 4.2.1.1.

Changing the access mode of *object* X from a higher restricted access mode to a lower one can be done easily without any difficulties because none the references from the other *objects* to the *object* X will be affected. However, changing the access mode of *object* X from a lower restricted access mode to a higher one requires more attention. This is because, if *object* X is

referenced by an *object* Y and this reference is allowed only when the access mode of *object* X is that lower restricted one, then changing it to a more restricted one will not allow such a reference from *object* Y to *object* X. To compare the restriction levels of two access modes, the procedure **moreRestLevel**(*OAModex, OAModey*) is used. The procedure returns true if the access mode *OAModex* is more restricted than the access mode *OAModey* and returns false for other cases.

The set of arguments and precondition conjuncts that are used for the FGT changeOAMode are dependent on the type of *object* element that is to be changed, as shown below:

A. changeOAMode(*Pn, Cn, _, _, _, class, OOldAM, ONewAM*)

This FGT is used to change the class access mode from an old access mode *OOldAM* to a new one *ONewAM*. The access mode of the class can be *public* or *default*. Changing the access mode of the class from a higher restricted access mode (*default*) to a lower restricted one (*public*) can be done without any difficulties because none of the references to the class *Pn.Cn* will be affected. However, changing the access mode of the class from a lower restricted access mode (*public*) to a higher restricted one (*default)* requires more attention. This is because if the class *Pn.Cn* is referenced by any other *object* that is located outside the package *Pn*, after changing the access mode to *default* that reference will not be allowed. Thus, changing the access mode of the class from *public* to *default* requires that the class *Pn.Cn* should not be referenced by any other *object* locate outside the package *Pn*. To verify this, the procedure **referenceOutPackage**(*Pn, Cn, class*) is used. The procedure indicates whether there is any reference to that class from *objects* locates outside the package.

To apply this FGT on the underlying system the following should hold.

- The class *Pn.Cn* should be already declared in the system.
- The old access mode *OOldAM* not equal to the new access mode *ONewAM*.
- If the new access mode *ONewAM* is *default* then the class *Pn.Cn* should not be referenced from outside the package *Pn*.

FGTPrecondConj(changeOAMode(*Pn, Cn, _, _, _, class, OOldAM, ONewAM*) **):-**

existsObject(*Pn, Cn, class*), not(*OOldAM=ONewAM*),	1. *Class Pn.Cn declared in the system.* 2. *Old access mode not equal to the new access mode.*

[(*ONewAM=default*, not(referenceOutPackage(*Pn, Cn, class*))) | *ONewAM=public*].	3. *If the new access mode is default then the class Pn.Cn should not be referenced from outside package Pn.* 4. *If the new access mode is public then the class Pn.Cn can be referenced from anywhere (the condition between the two brackets [] will be true).*

B. changeOAMode(*Pn, Cn, Methn, _, PrmLT, method, OOldAM, ONewAM*)

This FGT is used to change the method access mode from an old access mode *OOldAM* to a new one *ONewAM*. The access mode of the method can be *public*, *protected*, *default or private*.

To apply this FGT on the underlying system the following should hold.

- The method *Pn.Cn.Methn* with *PrmLT* should be already declared in the system.
- The old access mode *OOldAM* not equal to the new access mode *ONewAM*.
- If the new access mode *ONewAM* is more restricted than the old one *OOldAM* then changing the access mode will be done easily without any difficulties. For the other cases, conditions 3 to 5 in the following list of precondition conjuncts are used.

FGTPrecondConj(changeOAMode(*Pn, Cn, Methn,_,PrmLT,method, OOldAM,ONewAM*) **):-**

existsObject(*Pn, Cn, Methn, PrmLT, method*), not(*OOldAM=ONewAM*),	1. *Method Pn.Cn.Methn with PrmLT declared in the system.* 2. *The old access mode not equal to the new access mode.*
[(*ONewAM=private* , not(referenceOutClass(*Pn, Cn, Methn, PrmLT, method*))) |	3. *If the new access mode is private then the method Pn.Cn.Methn should not be accessed from outside the class Pn.Cn.*
(moreRestLevel(*ONewAM, OOldAM)*, *ONewAM=protected* , not(referenceOutPckSub(*Pn, Cn, Methn, PrmLT , method*))) |	4. *If the new access mode is protected and it is more restricted than the old one then the method Pn.Cn.Methn should not be accessed from outside the subclasses of the class Pn.Cn or the package Pn.*

(moreRestLevel(*ONewAM, OOldAM)*, *ONewAM=default* , not(referenceOutPackage(*Pn, Cn, Methn, PrmLT*))) | moreRestLevel(*OOldAM*, *ONewAM*)].	5. *If the new access mode is default, and it is more restricted than the old one then the method Pn.Cn.Methn should not be accessed from outside the package Pn.* 6. *For all the other cases in which the new access mode is less restricted than the old one then the condition between the two brackets [] will be true.*

C. changeOAMode(*Pn, Cn, Attn, _, _, attribute, OOldAM, ONewAM*)

This FGT is used to change the attribute access mode from an old access mode *OOldAM* to a new one *ONewAM*. The access mode of the attribute can be *public*, *protected*, *default or private*.

To apply this FGT on the underlying system the following should hold.

- The attribute *Pn.Cn.Attn* should already declare in the system.
- The old access mode *OOldAM* may not be equal to the new access mode *ONewAM*.
- If the new access mo*de ONewAM* is more restricted than the old one *OOldAM,* then changing the access m*ode wi*ll be done easily without any difficulties. For the other cases, conditions 3 to 5 in the following list of precondition conjuncts are used.

FGTPrecondConj(changeOAMode(*Pn, Cn, Attn, _, _, attribute, OOldAM, ONewAM*) **):-**

existsObject(*Pn, Cn, Attn, attribute*),	1. *The attribute Pn.Cn.Attn declared in the system.*
not(*OOldAM=ONewAM*),	2. *The old access mode not equal to the new access mode.*
[(*ONewAM=private*, not(referenceOutClass(*Pn, Cn, Attn, attribute*))) |	3. *If the new access mode is private then the attribute Pn.Cn.Attn should not be referenced from outside the class Pn.Cn.*
(moreRestLevel(*ONewAM, OOldAM)*, *ONewAM=protected*, not(referenceOutPckSub(*Pn, Cn, Attn, attribute*))) |	4. *If the new access mode is protected and it is more restricted than the old one then the attribute Pn.Cn. should not be accessed from outside the subclasses of the class Pn, Cn or the package Pn.*

(moreRestLevel(*ONewAM, OOldAM)*, *ONewAM=default*, not(referenceOutPackage(*Pn, Cn, Attn, attribute*))) |	5. *If the new access mode is default and it is more restricted than the old one then the attribute Pn.Cn.Attn should not be accessed from outside the package Pn.*
moreRestLevel(*OOldAM*, *ONewAM*)].	6. *For all the other cases in which the new access mode is less restricted than the old one then the condition between the two brackets will be true.*

4.2.1.4 changeODefType FGT

The changeODefType FGT is used to change the definition type of the method, attribute and parameter *object* elements in the class diagram from one definition type to another. It does not apply to class *object* elements because these *object* elements do not have type definitions in the class diagram. For the method *object* elements, the changeODefType FGT changes the definition type of the return value of the method.

It takes the following format:

changeODefType(*Pn, Cn, Memn, Prmn, PrmLT, OT, OOldDT, ONewDT*)

Where:

- *OOldAM* is the old definition type of the *object* element.
- *ONewAM* is the new definition type of the *object* element.

Note: For the description of the other arguments in the FGT, review section 4.2.1.1.

The set of arguments and precondition conjuncts that are used for the FGT changeODefType are dependent on the type of *object* element that is to be changed, as shown below:

A. changeODefType(*Pn, Cn, Methn,_, PrmLT, method, OOldDT, ONewDT*)

This FGT is used to change the definition type of the method's return value from an old definition type *OOldDefT* to a new one O*NewDefT*.

To apply this FGT on the underlying system then the method *Pn.Cn.Methn* with *PrmLT* should be already declared in the system and the old definition type *OOldDefT* is not equal to the new one *ONewDefT*.

FGTPrecondConj(changeODefType(*Pn,Cn,Methn,_,PrmLT, method, OOldDT, ONewDT*))**:-**	
existsObject(*Pn, Cn, Methn, PrmLT, method*), not(*OOldD T=ONewDT*).	1. *The method Pn.Cn.Methn with PrmLT declared in the system.* 2. *The old return type not equal to the new return type.*

B. changeODefType(*Pn, Cn, Attn, _, _, attribute, OOldDT, ONewDT*)

This FGT is used to change the definition type of an attribute from an old definition type *OOldDefT* to a new one *ONewDefT*.

To apply this FGT on the underlying system then the attribute *Pn.Cn.Attn* should be already declared in the system and the old definition type *OOldDT* is not equal to the new definition type *ONewDefT*.

FGTPrecondConj(changeODefType(*Pn, Cn, Attn, _, _, attribute, OOldDT, ONewDT*))**:-**	
existsObject(*Pn, Cn, Attn, attribute*), not(*OOldDT=ONewDT*).	1. *The attribute Pn.Cn.Attn declared in the system.* 2. *The old return type not equal to the new return type.*

C. changeODefType(*Pn, Cn, Methn, Prmn, PrmLT ,parameter, OOldDT, ONewDT*)

This FGT is used to change the definition type of a parameter from an old definition type *OOldDefT* to a new one *ONewDefT*.

To apply this FGT on the underlying system then the parameter *Prmn* should be declared in the method *Pn.Cn.Methn* with *PrmLT* and the old definition type *OOldDT* is not equal to the new one *ONewDefT*.

FGTPrecondConj(changeODefType(*Pn, Cn, Methn, Prmn, PrmLT , parameter, OOldDT, ONewDT*) **):-**

existsObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter*), not(*OOldDT=ONewDT*).	1. *The parameter Prmn declared in the method Pn.Cn.Methn with PrmLT.* 2. *The old return type not equal to the new return type.*

4.2.1.5 deleteObject FGT

The deleteObject FGT is used to delete unreferenced *object* elements from the UML class diagram. It is used to delete class, method, attribute, parameter *object* elements from the class diagram. It is not allowed to delete any *object* element from the system if that *object* is being referenced by any other *object* in the system.

The deleteObject FGT takes the following format:

deleteObject(*Pn, Cn, Memn, Prmn, PrmLT, OT)*

Note: For the description of the arguments in the FGTs the reader is referred to section 4.2.1.1.

The set of arguments and precondition conjuncts that are used for the FGT changeOAMode are dependent on the type of *object* element that to be deleted from the UML class diagram using that FGT, as shown below:

A. deleteObject(*Pn, Cn, _, _, _, class*)

This FGT as indicated from the last argument is used to delete an unreferenced empty class *Cn* from the package *Pn.*

To apply this FGT on the underlying system the following should hold.

- The class *Pn.Cn* should be declared in the system.
- The class is empty—i.e. it has no members (methods or attributes). If the class to be deleted has members, then these members should first be deleted by using other FGTs. This is important to control and manage the FGTs, as will be explained later.
- The class *Pn.Cn* has neither superclass nor subclasses.

- The class *Pn.Cn* is unreferenced from any other *object.*

FGTPrecondConj(deleteObject(*Pn, Cn, _, _, _, class*) **):-**

existsObject(*Pn,Cn, class*),	1. *Class Pn.Cn declared in the system.*
not(members(*Pn, Cn, class*)),	2. *Class Pn.Cn has no members.*
not(supclass(*Pn,Cn,_ ,_*)),	3. *Class Pn.Cn does not have subclasses.*
not(subclass(*Pn,Cn,_,_*)),	4. *Class Pn.Cn does not have superclass.*
not(isReferenced(*Pn,Cn, class*)).	5. *Class Pn.Cn is unreferenced from any other object.*

B. deleteObject(*Pn, Cn, Methn, _, PrmLT, method***)**

As indicated by the last argument, this FGT is used to delete an unreferenced method *Pn.Cn.Methn* with parameter type list *PrmLT*. Note that if the method is indirectly referenced by instances of one of the *Pn.Cn's* subclasses, then the method has to be regarded as a referenced *object* and may not be deleted.

To apply this FGT on the underlying system the following should hold.

- The method *Pn.Cn.Methn* with *PrmLT* should be declared in the system.
- The method is not referenced (*directly or indirectly*) by any other *object*.

FGTPrecondConj(deleteObject(*Pn, Cn, Methn, _, PrmLT, method*) **):-**

existsObject(*Pn,Cn, Methn, PrmLT , method*),	1. *Method Pn.Cn.Methn with PrmLT declared in the system.*
not(isReferenced(*Pn,Cn, Methn, PrmLT, method*)).	2. *Method Pn,Cn.Methn with PrmLT not referenced (directly or indirectly) by any other object in the system.*

C. deleteObject(*Pn, Cn, Attn, _, _, attribute***)**

This FGT as indicated from the last argument is used to delete an unreferenced attribute *Pn.Cn.Attn.* Note that if the attribute is indirectly referenced by instances of one of the *Pn.Cn's* subclasses, then the attribute has to be regarded as a referenced *object* and may not be deleted.

To apply this FGT on the underlying system the following should hold.

- The attribute *Pn.Cn.Attn* should be declared in the system.
- The attribute is not referenced (*directly or indirectly*) by any other *object*.

FGTPrecondConj(deleteObject(*Pn, Cn, Attn, _, _, attribute*) **):-**

existsObject(*Pn,Cn, Attn, attribute*), not(isReferenced(*Pn,Cn, Attn, attribute*)).	1. *Attribute Pn.Cn. Attn declared in the system.* 2. *Attribute Pn.Cn.Attn not referenced (directly or indirectly) by any other object in the system.*

D. deleteObject(*Pn, Cn, Memn, Prmn, PrmLT, parameter***)**

This FGT as indicated from the last argument is used to delete parameter *Prmn* from the method *Pn.Cn.Methn* with *PrmLT*. Note here that we do not specify the index of the parameter because the parameter is known by a name that is distinct from all those other parameters declared in the method.

To apply the FGT on the underlying system the following should hold.

- The parameter *Prmn* should be declared in the *PrmLT* of the method *Pn.Cn.Methn*.
- If *ParmALT* denotes the type list of the method *Methn* after deleting *Prmn,* then the method *Methn* with *ParmALT* may not be declared in the class *Pn.Cn* or in any of its ancestor classes.

FGTPrecondConj(deleteObject(*Pn, Cn, Attn, _, _, attribute*) **):-**

existsObject(*Pn, Cn, Methn, Prmn, PrmLT, parameter*), not(existsObject(*Pn, Cn, Methn, ParmALT, method*)), not(isInherited(*Pn, Cn, Methn, ParmALT, method*)).	1. *Parameter Prmn declared in the method Pn.Cn.Methn with PrmLT.* 2. *Method Pn.Cn.Methn with ParmALT not declared in the system, where ParmALT is the type list of the method Methn after deleting Prmn.* 3. *Method Pn.Cn.Methn with ParmALT not declared in any of Pn.Cn's ancestor classes, where ParmALT again is the type list of the method Methn after deleting Prmn.*

4.2.2 Relational Element FGTs

This group of FGTs includes all FGTs that are used to modify *relational* elements in the system. (Recall that *relational* elements in the simplified UML meta-model include generalizations (extends), associations, reads, writes, calls and types.) The *relational* elements represent the relations that exist between two *object* elements. By using these FGTs, the developer can add, rename and delete *relational* elements that may exist between the *object* elements.

There are two types of *relational* elements. The first type includes those relations that are appeared in the class diagram and represent the relations between the different classes in the class diagram, like *extends* and *association relational* elements. The second type includes those relations that are found between the different *object* elements but are not represented in the UML class diagram. (For a detailed explanation return to chapter 3.)

4.2.2.1 addRelation FGT

The addRelation FGT is used to add a *relational* element between two different *object* elements in the UML class diagram. It is used to add *extends*, *association*, *read*, *write*, *call*, or *type relation* between two different *object* elements in the class diagram.

In general, it takes the following format:

addRelation(*RelL, SourceObject, DestinationObject, RelT*)

Where:

- *RelL* is the label of the *relation*. It is used when the *relation* to be added is an *extends* or *association* relation. It is ignored for the other types of relations.
- The *SourceObject* specified by the following parameters:
  - *FPn* is the name of the package of the source *object*. It is used when the source *object* is a package, class, method, attribute or parameter.
  - *FCn* is the name of the class of the source *object*. It is used when the source *object* is a class, method, attribute or parameter.
  - *FMemn* is the name of the member (method or attribute) of the source *object*. It is used when the source *object* is a method, attribute or parameter.
  - *FPrmn* is the name of the parameter. It is used when the source *object* is a parameter.
  - *FPrmLT* is the type list of a method's parameters. It is used when the source *object* is a method or a parameter.

  - *FOT* is the type of the source *object* (*class, method, attribute,* or *parameter*).
- The *DestinationObject* specified by the following parameters: *TPn, TCn, TMemn, TPrmn, TPrmLT,* and *TOT*. Each of these parameters has the same description as its associated parameter in the *SourceObject* as defined above.
- *RelT* is the type of the *relation* intended to be added between the source and destination *objects* (*extends, association, read, write, call or type*).

The set of arguments and precondition conjuncts that are used for the FGT addRelation are dependent on the type of *relation* that is to be added between the two *object* elements in class diagram, as shown below:

A. addRelation(_, *FPn, FCn, FMethn,* _, *FPrmLT, method, TPn, TCn, TAttn,* _, _, *attribute, RelT* **)** where *RelT* is *read* or *write.*

This FGT is used to add a *read/write* relation between the method *FPn.FCn.FMethn* with *FPrmLT* and the attribute *TPn.TCn.TAttn*. The *relation* between the method *FMethn* and the attribute *TAttn* indicates that at the code-level there will be one or more statements in the method *FMethn* that will *read/write* the attribute *TAttn*.

Because such *relations* are not part of the class diagram, the *relation* label *RelL* is not of interest.

To apply this FGT on the underlying system the following should hold.

- The source and the destination *objects* of the *relation* should be declared in the system.
- The relevant *relation* between the two *objects* is not already present.
- The location of the source *object* and the destination *object* in the model together with the access mode of the destination *object* play an important role in determining the applicability of the addRelation FGT. For example, if the access mode of the attribute *TAttn* is *private* then the method and the attribute should be allocated in the same class. This option and the other options are clarified by the conditions from 4 to 7 in the following list of precondition conjuncts.

FGTPrecondConj(addRelation(_, *FPn, FCn, FMethn,* _, *FPrmLT, method, TPn, TCn, TAttn,*_,_, *attribute, RelT*) **):-**

existsObject(*FPn, FCn, FMethn, FPrmLT, method*), existsObject(*TPn, TCn, TAttn, attribute*),	1. *Method FPn.FCn.FMethn with FPrmLT declared in the system.* 2. *Attribute TPn.TCn.TAttn declared in the system.*

not(existRelation(_,*FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TAttn, attribute, RelT*)) ,	3. *Relation RelT not found between the two objects, the relation label is execluded from the condition.*
[(objectAMode(*TPn, TCn, TAttn, attribute, private*), *FPn.FCn=TPn.TCn*) |	4. *If the access mode of the attribute TAttn is private then the method FMethn should be in the same class of the attribute.*
(objectAMode(*TPn, TCn, TAttn*, *attribute, default*), *FPn=TPn*) |	5. *If the access mode of the attribute TAttn is default then the method FMethn should be in the same package of the attribute.*
(objectAMode(*TPn, TCn, TAttn, attribute, protected*), (subClass(*FPn,FCn, TPn, TCn*) | *FPn=TPn*)) |	6. *If the access mode of the attribute TAttn is protected then the method FMethn should be in the same package of the attribute or in one of the subclasses of the class TPn.TCn.*
objectAMode(*TPn, TCn, TAttn, attribute, public*)].	7. *If the access mode of the attribute TAttn is public then the method FMethn can be anywhere "the condition between the two brackets is true".*

A. addRelation(_, *FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TMethn, TPrmLT ,method, call* **)**

This FGT is used to add a *call* relation between the method *FPn.FCn.FMethn* with *FPrmLT* and the method *TPn.TCn.TMethn* with *TPrmLT*. The *relation* between the two methods *FMethn* and *TMethn* indicates that at the code-level, there will be one or more statements in the method *FMethn* that will *call* the method *TMethn*.

Because such *relations* are not part of the class diagram, the *relation* label *RelL* is not of interest.

FGTPrecondConj(addRelation(_, *FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TMethn, TPrmLT ,method, call*) **):-**

existsObject(*FPn, FCn, FMethn, FPrmLT, method*),	1. *Method FPn.FCn.FMethn with FPrmLT declared in the system.*
existsObject(*TPn, TCn, TMethn, TPrmLT, method*),	2. *Method TPn.TCn.TMethn with TPrmLT declared in the system.*

not(existRelation(_,*FPn, FCn, FMethn, FPrmLT, method, TPn, TCn, TMethn, TPrmLT ,method, call*)),	3. *Relation call not found between the two objects.*
	4. *Note: we exclude the relation label from the condition.*
[(objectAMode(*TPn, TCn, TMethn, TPrmLT, method, private*), *FPn.FCn=TPn.TCn*) |	5. *If the access mode of the method TMethn is private then the calling method FMethn should be in the same class of TMethn.*
(objectAMode(*TPn, TCn, TMethn, TPrmLT, method, default*), *FPn=TPn*) |	6. *If the access mode of the method TMethn is default then the method FMethn should be in the same package of TMethn.*
(objectAMode(*TPn, TCn, TMethn, TPrmLT, method, protected*), (subClass(*FPn,FCn, TPn, TCn*) | *FPn=TPn*)) |	7. *If the access mode of the method TMethn is protected then the method FMethn should be in the same package of the TMethn or in one of the subclasses of the class TPn.TCn.*
objectAMode(*TPn, TCn, TMethn, TPrmLT, method, public*)].	8. *If the access mode of the method TMethn is public then the method FMethn can be anywhere "the condition between the two brackets is true".*

C. addRelation(_, *FPn, FCn, FMethn*,_, *FPrmLT, method, TPn, TCn*, _, _, _, *class, type* **)**

This FGT is used to add a *type* relation between the method *FPn.FCn.FMethn* with *FPrmLT* and the class *TPn.TCn*. The *relation* between the method *FMethn* and the class indicates that at the code-level, the method *FMethn* defines at least one local variable whose type definition is class *TCn*.

Because such *relations* are not part of the class diagram, the *relation* label *RelL* is not of interest.

FGTPrecondConj(addRelation(_, *FPn, FCn, FMethn*, _, *FPrmLT, method, TPn, TCn*, _, _, _, *class, type*) **):-**

existsObject(*FPn, FCn, FMethn, FPrmLT, method*),	1. *Method FPn.FCn.FMethn with FPrmLT declared in the system.*
existsObject(*TPn, TCn, class*),	2. *Class TPn.TCn declared in the system.*

not(existRelation(_,*FPn, FCn, FMethn, FPrmLT, method, TPn, TCn,class, type*)),	3. *Relation type not found between the two objects.*
[(objectAMode(*TPn, TCn, class, default*), *FPn=TPn*) |	4. *If the access mode of the class TCn is default then the method FMethn should be in the same package of TCn.*
objectAMode(*TPn, TCn, class, public*)].	5. *If the access mode of the class TCn is public then the method FMethn can be anywhere "the condition between the two brackets is true".*

D. addRelation(_, *FPn, FCn,* _, _, _, *class, TPn, TCn,* _, _, _, *class, type* **)**

This FGT is used to add a *type* relation between the class *FPn.FCn* and the class *TPn.TCn*. The *relation* between the two classes indicates that at the code-level, the class *FCn* defines an attribute of type class *TCn*. Because such *relations* are not part of the class diagram, the *relation* label *RelL* is not of interest.

FGTPrecondConj(addRelation(_, *FPn, FCn,* _, _, _, *class, TPn, TCn,* _, _, _, *class, type*) **):-**

existsObject(*FPn, FCn, class*),	1. *Class FPn.FCn declared in the system.*
existsObject(*TPn, TCn, class*),	2. *Method TPn.TCn declared in the system.*
[(objectAMode(*TPn, TCn, class, default*), *FPn=TPn*) |	3. *If the access mode of the class TCn is default then the class FCn should be in the same package of TCn.*
objectAMode(*TPn, TCn, class, public*)].	4. *If the access mode of the class TPn.TCn is public then the class FCn can be anywhere "the condition between the two brackets is true".*

E. addRelation(*RelL, FPn, FCn,* _, _, _, *class, TPn, TCn,* _, _, _, *class, extends* **)**

This FGT is used to add an *extends relation* (*generalization/specialization*) with label *RelL* between the two classes *FPn.FCn* and *TPn.TCn*. The source *object* of the *relation FPn.FCn* will be the superclass while the destination *object TPn.TCn* will be the subclass.

To apply this FGT on the underlying system the following should hold.

- The source and the destination *objects* of the *relation* should be declared in the system.
- The *extends* relation between the two *objects* may not already exist.
- To avoid multiple inheritances between classes, the class *TCn* may not already be a subclass of any other class.
- To avoid circular *extends-relations* between classes, the class *FCn* should not be one of the descendants of the class *TCn.*

FGTPrecondConj(addRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, extends*) **):-**

existsObject(*FPn, FCn, class*),	1. *Class FPn.FCn declared in the system.*
existsObject(*TPn, TCn, class*),	2. *Method TPn.TCn declared in the system.*
not(existRelation(_,*FPn, FCn, class, TPn, TCn, class, extends*)),	3. *Relation extends not found between the two classes*
not(subclass(*TPn, TCn*,_,_)),	4. *Class TPn.TCn is not a subclass of any other classes. This condition is to avoid multiple inheritance.*
not(subclass(*FPn, FCn*, *TPn, TCn*)).	5. *Class FPn.FCn is not one of the descendants of the TPn.TCn. This condition is to avoid circular extends between the classes.*

F. addRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, association* **)**

This FGT is used to add an *association relation* with label *RelL* between the two classes *FPn.FCn* and *TPn.TCn*. The first class *FPn.FCn* will be the source of the *relation* while the second class *TPn.TCn* will be the destination of the *relation*. Note that if there is any *read*, *write*, or *type relation* between the class *FCn* (or any of its members) and the class *TCn* (or any of its members) then there should be an *association* relation from class *FCn* to class *TCn*.

FGTPrecondConj(addRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, association*) **):-**

existsObject(*FPn, FCn, class*),	1. *Class FPn.FCn declared in the system.*
existsObject(*TPn, TCn, class*),	2. *Method TPn.TCn declared in the system.*
not(existRelation(*RelL, FPn, FCn, class, TPn, TCn, class, association*)).	3. *Relation association with label RelL not found between the two classes.*

4.2.2.2 renameRelation FGT

This FGT is used to change the label of the *relation* that exists between two *objects* from an old label *RelOldL* to a new one *RelNewL*. It has the following format:

renameRelation(*RelOldL, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT, RelT, RelNewL*)

Since the label of *extends* and *association* relations appear in the class diagram, this FGT can be used to change the label of the *extends* and *association* relations.

To apply this FGT on the underlying system the following should hold.

- The *relation* with label *RelOldL* and type *RelT* should already be found between the two object elements.
- The *relation* with label *RelNewL* and type *RelT* should not be found between the two object elements.

FGTPrecondConj(renameRelation(*RelOldL, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT, RelT, RelNewL*) **):-**

existRelation(*RelOldL, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT*)	1. *A relation with the old label RelOldL already exists between the two objects.*
not(existRelation(*RelNewL, FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT*))	2. *A relation with the new label RelNewL should not be found between the two objects.*

4.2.2.3 deleteRelation FGT

The deleteRelation FGT is used to delete a *relation* element that may exist between two different *object* elements in the class diagram.

A. deleteRelation(_, *FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT, TOT, RelT* **)** where *RelT* is *read, write, call,* or *type.*

This FGT is used to delete a *relation* between two *objects*. However, the *relation* has to be *read, write, call* or *type.*

FGTPrecondConj(deleteRelation(_, *FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT, RelT*) **):-**

existRelation(_, *FPn, FCn, FMemn, FPrmn, FPrmLT ,FOT, TPn, TCn, TMemn, TPrmn, TPrmLT ,TOT, RelT*).	1. *The relation RelT between the two objects already exists in the system.*

B. deleteRelation(*RelL, FPn, FCn, _, _, _ ,class, TPn, TCn, _, _, _, class, association***)**

This FGT is used to delete an *association* relation that may exist between two classes in the class diagram.

FGTPrecondConj(deleteRelation(*RelL, FPn, FCn, _, _, _,class, TPn, TCn, _, _, _,class, association*) **):-**

existRelation(*RelL, FPn, FCn, _, _, _ ,class, TPn, TCn, _, _, _,class, association*)	1. *The relation between the two classes has already existed in the system.*

C. deleteRelation(*RelL, FPn, FCn, _, _, _, class, TPn, TCn, _, _, _, class, extends***)**

This FGT is used to delete an *extends* relation (*generalization/specialization*) with label *RelL* that is found between the two classes *FPn.FCn* as the superclass and *TPn.TCn* as the subclass.

To apply this FGT on the underlying system, the *relation* between the two classes has to be already found between the two classes. Furthermore, instances of class *TCn* (or any of *TCn's* descendant) may not reference any member which is inherited from the class *FCn* (or any of its ancestors). Clearly, if such a reference to such a member is found, then after deleting the *extends* relation that member will not be accessible to instances of *TCn* and its descendants. Trying to reference the member will therefore cause an error. This is checked by condition two of the following list of precondition conjuncts.

FGTPrecondConj(deleteRelation(*RelL, FPn, FCn,_, _, _, class, TPn, TCn, _, _, _, class, extends*) **):-**

existRelation(*RelL, FPn, FCn, FPrmLT ,class, TPn, TCn, TPrmLT ,class, extends*)) ,	1. *The relation between the two objects has already existed in the system.*

[existsObject(*FPn, FCn, FMemn, FPrmLT, member*), (objectAMode(*FPn, FCn, FMemn, FPrmLT, member*, *OAMode*), isElement(*OAMode, [protected, public]*), not(useInheritenceMem(*TPn, TCn*, *FMemn,FPrmLT* , *member*))], [existsObject(*FPn, FCn, FMemn, FPrmLT, member*), objectAMode(*FPn, FCn, FMemn, FPrmLT, member*, *OAMode*), isElement(*OAMode, [protected, public]*) , subclass(*TTPn, TTCn*, *TPn, TCn*) ,not(useInheritenceMem (*TTPn, TTCn*, *FMemn* , *FPrmLT*, *member*))], [supclass(*FFPn, FFCn*, *FPn, FCn*), existsObject(*FFPn, FFCn, FMemn, FPrmLT*, *member*) ,objectAMode(*FPn, FCn, FMemn, FPrmLT, member*, *OAMode*), isElement(*OAMode, [protected, public]*), not(useInheritenceMember(*TPn, TCn*, *FMemn*, *FPrmLT*, *member*))], [supclass(*FFPn, FFCn*, *FPn, FCn*), existsObject(*FFPn, FFCn, FMemn, FPrmLT, member*), objectAMode(*FPn, FCn, FMemn, FPrmLT, member*, *OAMode*), isElement(*OAMode, [protected, public]*) , subclass(*TTPn, TTCn*, *TPn, TCn*), not(useInheritenceMember (*TTPn, TTCn*, *FMemn*, *FPrmLT*, *member*))].	2. *If the class FCn or any of its superclasses "ancestors" have a member X and at the same time the member X is inherited and used from outside through instances of the class TCn or any of its descendants then the extends relation between the two classes (FCn and TCn) cannot be deleted.*

4.3 FGT Sequential Dependency

In the foregoing, the notion of a postcondition of an FGT has not been discussed. Nevertheless, it is evident that whenever an FGT is applied to a system, one or more of its precondition conjuncts will be negated. For example:

- In adding an object, the precondition that the object may not exist is negated, and the object now exists in the system.

- In renaming an object, the precondition requires that an object of the old name may exist, and an object of the new name may not exist.

After application of the relevant FGT, then requirements are negated. The conjunction of these negated precondition conjuncts after applying an FGT will be considered to be its postcondition conjuncts.

4.3.1 Definition

FGT_j is said to be potentially sequentially dependent on FGT_i if and only if the postcondition conjuncts of FGT_i satisfied one or more precondition conjuncts of FGT_j. The sequential dependency between the two FGTs is represented by:

$$FGT_i \rightarrow FGT_j$$

For example, the FGT

addObject(*P, A, m1, _, _, type(basic, void, 0), public, [], method*)

that is used to add the method *m1* in the class *P.A* is sequentially dependent on the FGT

addObject(*P, A, _, _, _, _, public, _, class*)

that is used to add the class *A* in the package *P*, because one first has to add the container (class *P.A*) before adding members in it. The sequential dependency between the two FGTs is represented as:

addObject(*P, A, _, _, _, _, public, _, class*) → addObject(*P, A, m1, _, _, type(basic, void, 0), public, [], method*)

Note that, as defined above, the potential sequential dependency between two FGTs does not depend on the description of the system under consideration. (This is in contrast with Robert [70], who defines sequential dependency between two refactorings relative to program or system.) This means that if $FGT_j \rightarrow FGT_i$ and there is a need to apply FGT_i to some given system, S, one of the following scenarios may occur:

- S is such that it already satisfies all precondition conjuncts of FGT_i. Thus, FGT_i may be directly applied to S. In this case, it would not be possible to apply FGT_j to S, since the satisfaction of all FGT_i's precondition conjuncts indicates that at least one of FGT_j's precondition conjuncts is not satisfied by S.

- S does not satisfy all precondition conjuncts of FGT_i. In this case it will be necessary to select one or more FGTs upon which FGT_i sequentially depends, to apply it to S, and then to apply FGT_i. Whether or not FGT_j is to be included in this selection depends on S.

All the potential sequential dependencies between all the FGTs in the earlier sections of this chapter have been catalogued. These are shown in Figure 4.1. The figure shows two types of sequential dependency between the different FGTs. Each one of the two types will be explained in detail in the following two subsections.

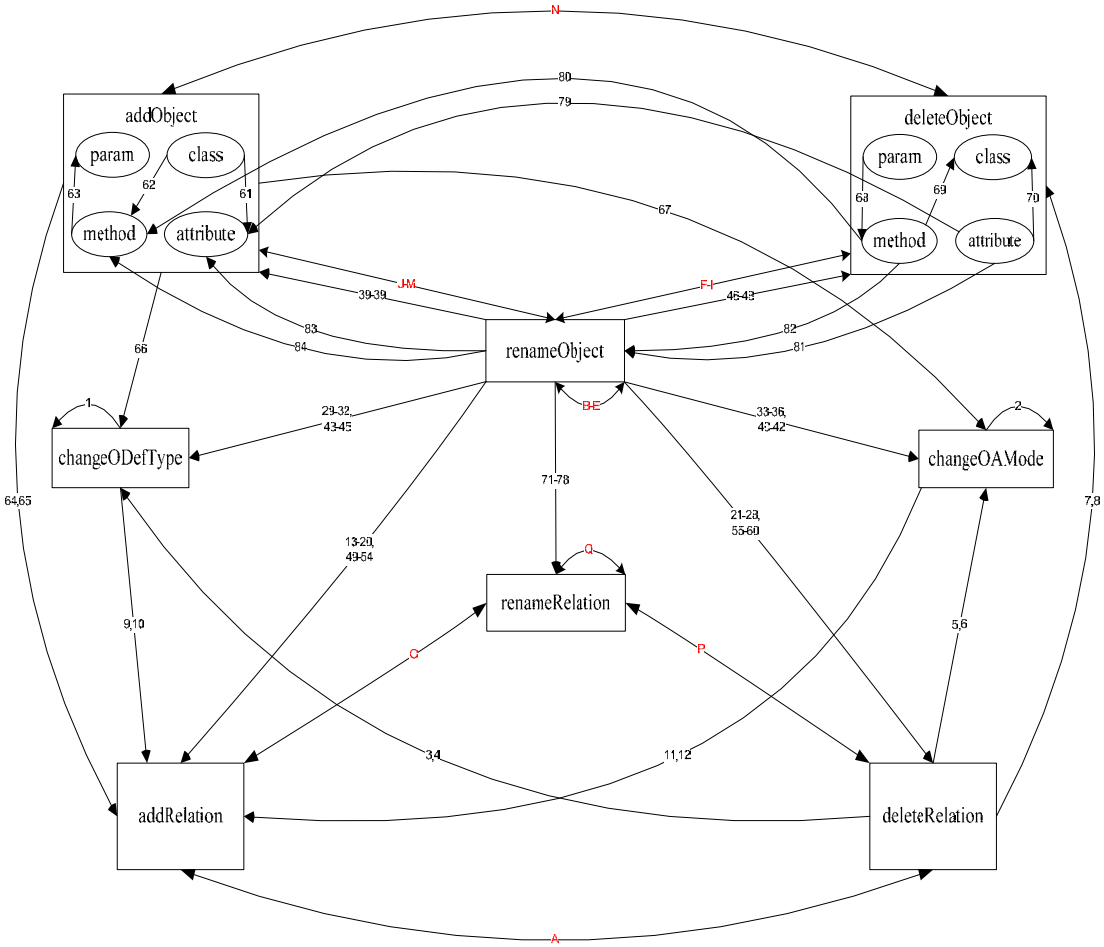


Figure 4.1: Potential sequential dependencies between FGTs

4.3.2 Uni-Directional Sequential Dependencies

The first kind includes sequential dependencies that occur in one direction between the two FGTs (FGT_i and FGT_j). This means that FGT_j is sequentially dependent on FGT_i but FGT_i is

not sequentially dependent on FGT_j. The uni-directional sequential dependency between FGTs is represented as arcs with a single arrow at one end in Figure 4.1. This category of sequential dependencies is represented as **uniDirSD** facts in the Prolog database of the refactoring tool.

All the uni-directional sequential dependencies in Figure 4.1 are discussed in more detail in Appendix A.1. It will be seen that each numbered sequential dependency corresponds to a numbered arc in Figure 4.1 that represents a uni-directional sequential dependency.

For example, consider the following FGTs:

FGT_i : addObject(*P, C, _, _, _, _, public, _, class*) and

FGT_j : addObject(*P, C, att1, _, _, type(basic, int, 0), private, _, attribute*)

The information represented in Figure 4.1 (the arrow labelled 61) shows that FGT_j is sequentially dependent on FGT_i ($FGT_i \rightarrow FGT_j$). This is because the class *P.C* has to be added to the system first. Only after that can the attribute *att1* be added in that class. On the other hand, FGT_i is not sequentially dependent on FGT_j.

4.3.3 Bi-Directional Sequential Dependency

The second type of sequential dependency includes sequential dependencies that can occur in the two directions of the two FGTs. This means that the first FGT is sequentially dependent on the second one and that the second FGT is sequentially dependent on the first one—i.e. for FGT_i and FGT_j, $FGT_i \leftrightarrow FGT_j$. The bi-directional sequential dependencies are represented as arcs with arrows at both ends in Figure 4.1. This category of sequential dependencies is represented as **biDirSD** facts in the Prolog database of the refactoring tool.

All the bi-directional sequential dependencies in Figure 4.1 are discussed in more detail in Appendix A.2. It will be seen that each numbered sequential dependency corresponds to a numbered arc in Figure 4.1 that represents a bi-directional sequential dependency.

For example, consider the following FGTs:

FGT_i : addObject(*P ,A, f1, _, _, type(basic, int, 0), private, _, attribute*) and

FGT_j : renameObject(*P, A, f1, _, _, attribute, f2*)

Two forms of sequential dependencies can occur between the two FGTs in this example:

a. $FGT_i \rightarrow FGT_j$: This is the case when class *A* does not contain attribute *f1*. The attribute *f1* then has to be added in class *A* by the addObject FGT. Thereafter, the added attribute can be renamed from *f1* to *f2* by the renameObject FGT. Thus, here the renameObject FGT is sequentially dependent on the addObject FGT.

b. $FGT_j \rightarrow FGT_i$: This is the case when attribute *f1* is originally declared in class *A* so adding another attribute with same name *f1* will cause duplication. Here the renameObject FGT has to be used to change the name of *f1* to *f2* and thereafter the addObject FGT can be used to add the attribute *f1* in class *A*. In this case the FGT addObject is sequentially dependent on the renameObject FGT.

To decide which sequential dependency applies in a given situation, the state of the underlying system has to be taken into consideration. This will be discussed in more detail in chapter 9.

4.3.4 Mapping Feasible FGT-Lists to FGT-DAGs

This section takes as a starting point a **feasible FGT-list**. By this is meant a list of FGTs for which at least one system exists, such that the FGT elements in the list can feasibly be applied to the system, starting at the head of the list and applying each successive FGT until the tail of the list has been applied. A consequence of applying the list to an appropriate system is that a set of objects and a set of relations (each possibly empty) will be guaranteed to exist in the system; and a set of objects and a set of relations (each possibly empty) will be guaranteed *not* to exist in the system. The conjunction of the assertions about the existence and non-existence of these entities can be regarded as the list's postcondition.

Of course, not every list of FGTs is feasible. For example, any FGT-list that specifies two successive deletions of the same object cannot be feasible, since the precondition of the second—the object's existence—cannot be met. Nevertheless, for the purposes of describing the algorithm given later in this section, the origin of such a feasible list of FGTs is currently not of concern. It may, for example, be an FGT-list proposed by a developer who wishes to transform a given system design in some particular way. The transformation may or may not retain the original system behaviour—i.e. it may or may not be a refactoring.

This section is also concerned with the notion of an **FGT Directed Acyclic Graph (FGT-DAG)**. An FGT-DAG is a directed acyclic graph in which each node represents an FGT, and there is an arc between two nodes, say from node FGT_j to FGT_i, if and only if:

1. FGT_i is sequentially dependent on FGT_j;

2. FGT_i is not sequentially dependent on any successor of FGT_j; and

3. no ancestor of FGT_j has an arc to FGT_i (even if FGT_i is sequentially dependent on that ancestor).

An FGT-DAG is feasible if some system exists to which it can feasibly be applied. An FGT-DAG is applied to a system by applying the FGTs in any order that respects the sequential dependency relationships represented by the arcs. This means that an FGT may only be applied after all its ancestors have been applied. As in the case of a feasible FGT-list, a feasible FGT-DAG is characterised by a postcondition—the conjunction of predicates asserting what objects and relations exist and/or do not exist as a result of applying the FGT-DAG. Similarly, the postcondition of a set of feasible FGT-DAGs is simply the conjunction of the postconditions of its constituent FGT-DAGs, and is attained by applying these in any order.

Clearly, if a feasible FGT-list is to be applied to some system, the system should comply with certain requirements that ensure that the FGTs in the list can indeed be applied in the given order—i.e. the feasible FGT-list has a certain precondition conjuncts to which the system should conform. The conjuncts of the precondition of this feasible FGT-list are not simply the conjuncts of all precondition conjuncts of its constituent FGTs. Indeed, it consists of the conjunction of FGT precondition conjuncts that are not negated as a result of applying the FGTs. For example, consider the feasible FGT-list [FGT_1, FGT_2]. Suppose the precondition of FGT_1 is P_1^P_2 and the precondition of FGT_2 is P_3^P_4. Suppose, also, that the postcondition of FGT_1 is P_3 (or, more generally, that it logically implies P_3, but not P_4). Then the precondition of the list is P_1^P_2^P_4. By similar argumentation, a set of feasible FGT-DAGs also has a precondition.

In the remainder of this thesis, it should be assumed that the reference to an FGT-list or set of FGT-DAGs would be taken to mean a **feasible** FGT-list or set of FGT-DAGs, unless otherwise stated. Furthermore, a sequence of FGTs should be regarded as equivalent to a list of FGTs, the latter simply indicating the concrete implementation of a sequence in the Prolog context.

The question then arises: How can a feasible FGT-list can be mapped to a set of FGT-DAGs that has the same postcondition as the feasible FGT-list? An algorithm, called **build-FGT-DAG** has been implemented in the prototype tool to do that. Algorithm 4.1 provides the pseudo-code for the build-FGT-DAG algorithm. The algorithm derives from a feasible FGT-list, *FGTList,* a set of FGT-DAGs, DSET, that has the same postcondition as *FGTList*.

It does this by setting DSET to the empty set, and then processing the FGTs in *FGTList* from first to last. Each next FGT, FGT_i, to be processed begins as a new singleton FGT-DAG in DSET. All paths in the other FGT-DAGs in DSET are then traversed in a bottom-up fashion,

searching for the first FGT upon which FGT_i sequentially depends. If such a node, FGT_j, is found in a path, it is connected to FGT_i and all ancestors of FGT_j are eliminated as candidates for further consideration.

Algorithm 4.1 (Building FGT-DAGs algorithm)

build-FGT-DAG (*FGTList* **)**

Input: *FGTList*: A feasible list of FGTs
uniDirSD: Uni-directional sequential dependencies between FGTs
biDirSD: Bi-directional sequential dependencies between FGTs

Output: DSET: A set of FGT-DAGs whose postcondition is the same as that of *FGTList*

Set DSET to the empty set
For each FGT_i in *FGTList* **do** { //FGTs should be selected in order from first to last
  Mark each FGT in each FGT-DAG of DSET as *unchecked*
  Insert FGT_i into DSET as a single node of a new FGT-DAG and mark it as *checked*
  While there are *unchecked* FGTs in DSET **do** {
    Select an *unchecked* FGT with no *unchecked* children, say FGT_j
    Mark FGT_j as *checked*
    If $FGT_j \rightarrow FGT_i$ (as determined from **uniDirSD** and **biDirSD**) **then** {
      Mark all ancestors of FGT_j as *checked*
      Insert an arc from FGT_j to FGT_i
    } //enf **If**
  } //end **While**
}//end **For**
Return DSET

The algorithm will build the same set of FGT-DAGs from a given feasible FGT-List. Firstly, note the comment in the For-each loop: FGTs are selected in the order in which they appear in the list. Secondly, note that there is no possibility of non-determinism because of the potential alternative selections in the While-loop. To see this, suppose that FGT_j and FGT_k are both candidates for selection as FGTs with no unchecked children. If FGT_j is selected before FGT_k, then a link may (because of sequential dependency) or may not be established from FGT_j to FGT_i. However, it can easily be seen that this selection will not cause FGT_k to be checked. Instead, FGT_k will then be a candidate for selection in the next iteration.

When the algorithm completes, each FGT in *FGTList* will have been inserted into one and only one FGT-DAG in the set of FGT-DAGs. Each FGT node will have inbound arcs from the closest FGTs that precede it in *FGTList* upon which it sequentially depends. It will have outbound arcs to the closest FGTs following it in *FGTList* and which are sequentially dependent on it. Note that the algorithm has been designed to ensure that whenever a candidate bi-directional sequential dependency relationship between two elements in *FGTList* is found, the direction reflected in the FGT-DAG corresponds to the intended execution order dictated by *FGTList*.

Note that the structure (FGT-DAGs) produced by the algorithm are indeed acyclic, and not cyclic. This can be verified by considering the following two points:

a. The resulting set of DAGs represents a feasible FGT list. As suggested at the beginning of this section, the FGTs of a feasible FGT list are ordered according to their sequential dependencies in such a way that their overall precondition does not evaluate to false.

b. Logically, the set of DAGs have been set up in such a way that they encapsulate the sequential dependency between the FGTs. The sequential dependency conveys the nature of the pre/post conditions of the FGTs. Suppose that one FGT-DAG contained a cycle of sequential dependencies, for example: A→B→C and C→A. This would mean that part of the post conditions of C is needed to satisfy the preconditions of A and at the same time part of the postcondition of A is needed to satisfy the preconditions of C, which leads to a contradiction. Such a contradiction could only arise if the input FGT list was not feasible.

As a toy example, Figure 4.2 shows the FGT-DAGs that are produced for the following collection of FGTs of a refactoring X. The result shows that the FGTs of refactoring X are allocated inside three different FGT-DAGs which are sequentially independent:

- renameObject(*lan, A, _, _, _, class, B*),
- addObject(*lan, C, t, _, _, _, public, [], method*),
- addObject(*lan, B, y, _, _, _, public, _, attribute*),
- renameObject(*lan, B, y, _, _, attribute, x*),
- changeODefType(*lan, B, x, _, _, attribute, int, float*),
- deleteObject(*lan, S, m, _, [], method*),
- addObject(*lan, S, m, _, _, _, private, [], method*),
- changeOAMode(*lan, S, m, _, [], method, private, public*),
- deleteObject(*lan, Super, x, _, _, attribute*),
- renameObject(*lan, C, m, _, [], method, n*),
- addRelation(*l, lan, C, n, _, [], method, lan, S, m, _, [], method, call*),

- deleteRelation(*l, lan, C, n, _, [], method, lan, S, m, _, [], method, call*),
- deleteObject(*lan, C, n, _, [], method*),
- renameObject(*lan, C, t, _, [], method, h*),
- addRelation(*l1, lan, S, m, _, [], method, lan, B, x, _, _, attribute, write*).

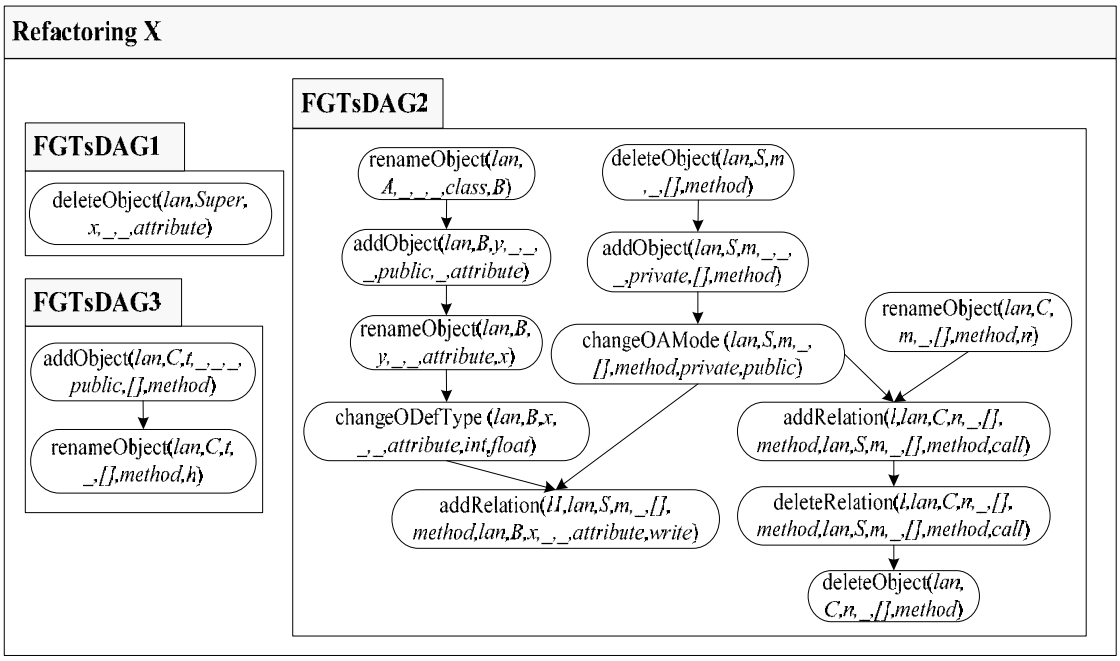


Figure 4.2: FGT-DAGs of refactoring X

4.4 FGTs for Primitive and Composite Refactorings

This section discusses in overview how to deal with primitive and composite refactorings in terms of their transformation operations and their preconditions. In particular, the section shows the relationship between the previously identified set of FGTs and refactorings, whether primitives or composites. A more complete discussion of primitive refactorings is taken up in chapter 5, and of composite refactorings in chapter 10.

4.4.1 Definitions

Definition 1: A **primitive refactoring** is an atomic refactoring that cannot be split into more refactorings. In the refactoring literature, researchers agree that there exists a finite set of primitive refactorings [65, 70]. The list of primitive refactorings that are commonly agreed upon is shown in Table 4.1.

A primitive refactoring may be said to be sound if its application to a system, say S1, which complies with its precondition results in a system, say S2, whose behaviour is the same as that of S1. Of course, S1 and S2 have the same behaviour if and only if for all possible input their resulting output is the same.

Furthermore, the collection of primitive refactorings in Table 4.1 can be regarded as complete with respect to the FGTs in this thesis if and only if it is not possible to use some set these FGTs to define a new primitive refactoring, that has not been mentioned in Table 4.1.

The question may be asked whether the primitive refactorings in Table 4.1 are sound and complete. It is beyond the scope of this thesis to provide a formal answer to this question. For their soundness, we appeal to their appearance in the literature. Should they be incomplete (in the sense mentioned above), then, per definition, it will be possible to use FGTs to add to the menu of primitive refactoring given in the table.

For each primitive refactoring, a precondition exists that will guarantee behaviour preservation of the system. This precondition is implemented inside the refactoring tool and need to be checked before applying the related refactoring.

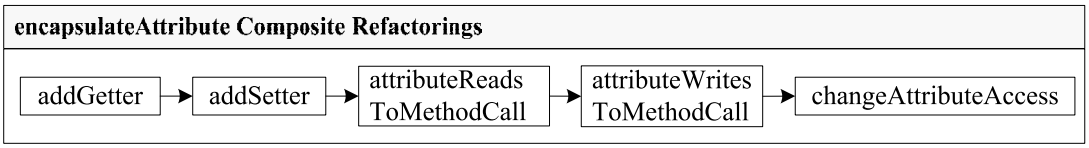
Table 4.1: Primitive refactorings

<table>
<tr><th rowspan="2">Add Element Refactorings</th><th rowspan="2">Delete Element Refactorings</th><th colspan="2">Change Element Refactorings</th></tr>
<tr><th>Change Characteristics</th><th>Change Structure</th></tr>
<tr><td rowspan="2">addClass
addMethod
addAttribute
addParameter</td><td rowspan="4">deleteClass
deleteMethod
deleteAttribute
deleteParameter</td><td rowspan="2">renameClass
renameMethod
renameAttribute
renameParameter</td><td>changeSuper</td></tr>
<tr><td>moveMethod
moveAttribute</td></tr>
<tr><td>addGetter
addSetter</td><td>changeClassAccess
changeMethodAccess
changeAttributeAccess</td><td>attributeReadsToMethodCall
attributeWritesToMethodCall</td></tr>
<tr><td></td><td>changeMethodType
changeAttributeType
changeParameterType</td><td>pullUpMethod
pullUpAttribute
pushDownMethod
pushDownAttribute</td></tr>
</table>

Definition 2: A **composite refactoring** is a collection of primitive refactorings that are applied on the model as one unit. In part of the composite refactoring, the execution order of some of, but not necessarily all, its primitive refactorings may be specified. Each composite refactoring has its own precondition. This precondition may not simply be the conjunction of its constituent primitive refactorings preconditions. Instead, it should articulate system conditions

that make it possible to apply the primitive refactorings in the order required by the composite refactoring.

An example of a composite refactoring is the **encapsulateAttribute** composite refactoring. This composite, as will be shown in more detail in chapter 6, consists of the following sequence of primitive refactorings:



The current approach shifts the granularity of transformation one level down: primitive refactorings are constructed from a collection of FGTs ordered in FGT-DAGs. Thus, FGTs are the most fine-grained type of transformations under consideration. The relationship between primitive refactorings, composite refactorings and FGTs is intuitively reflected in Figure 4.3(b). Figure 4.3(a) shows that a composite refactoring is a collection of primitive ones, and each primitive refactoring can be defined as a collection of FGTs. Thus, each composite refactoring can be carried out as a collection of FGTs.

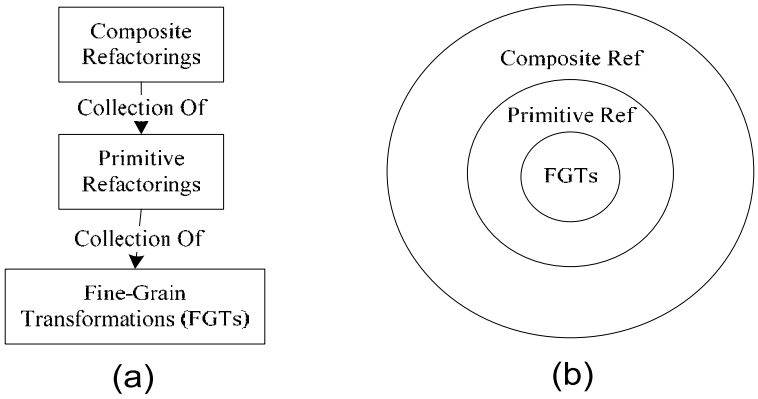


Figure 4.3: Primitive, composite refactorings and FGTs

4.4.2 FGT-Enabling Preconditions in an FGT-DAG

It is evident an FGT in an FGT-DAG has the property that the postcondition of each of its parents logically entails of one or more of that FGT's precondition conjuncts. If

$$Pre = \{P_i \mid i = 1, \ldots n\}$$

is the set of the FGT's precondition conjuncts, and

$$Post = \{Q_j \mid j = 1, \ldots m\}$$

is the set of postconditions of all of its parents, then

$$En = \{P_i \in Pre \mid \forall Q_j \in Post : \sim(Q_j \Rightarrow P_i)\}$$

defines the set of precondition conjuncts of the FGT that are not entailed by its parents' postconditions. This will be called the FGT's set of **enabling precondition conjuncts**.

The union of the enabling precondition conjuncts of all FGTs in an FGT-DAG is the FGT-DAG's set of enabling precondition conjuncts. The conjunction of all enabling precondition conjuncts of all the FGT-DAGs in a refactoring is called the **FGT-enabling** precondition of the refactoring (and also of the set of FGT-DAGs).

Clearly, if a system complies with the FGT-enabling precondition of an FGT-DAG, then the FGTs in the FGT-DAG can be systematically applied to the system in the order determined by the FGT-DAG, with the assurance that all FGT preconditions will be fulfilled by the system when they are to be applied to the system.

4.4.3 FGTs and Primitive Refactorings Preconditions

In much of the literature on refactoring, precondition conjuncts for each respective primitive refactoring are specified. Figure 4.4(b) shows how the current refactoring approaches deal with such refactorings preconditions. All the precondition conjuncts of the refactoring in these approaches are installed as one unit at the level of the whole refactoring. If the system complies with all of the primitive's precondition conjuncts, then the refactoring is applied to the system, and behaviour is guaranteed to be preserved. (Note that this application occurs "atomically", which is why Figure 4.4(b) represents the refactoring as a black box.)

However, a primitive refactoring can be represented as a collection of FGTs ordered within a set of FGT-DAGs. This will be discussed in chapter 5. As seen in section 4.4.2, associated with each FGT-DAG is a specific FGT-enabling precondition. This raises the following question:

Suppose that a primitive refactoring is represented as a set of FGT-DAGs. Will behaviour of a system be preserved if all the FGT-enabling preconditions of all the FGT-DAGs are satisfied before their individual FGTs are applied (in the appropriate order) to the system? The answer is NO. To justify this claim consider the following point.

For some primitive refactorings, there are special precondition conjuncts that cannot be inferred from the precondition conjuncts of the FGTs included in the primitive refactoring. For example, one of the precondition conjuncts of the refactoring **pullUpAttribute** that is used to pull up an attribute *Attn* to the superclass from all subclasses where it is defined, is that the attribute *Attn* should be declared identically (have the same definition type) in all the subclasses where it is defined. Consideration of the FGTs used in such a **pullUpAttribute** refactoring (not given here, but in chapter 5) will show that it is impossible to infer such a precondition conjunct from the preconditions of the included FGTs.

In general, it is therefore necessary to isolate the set of precondition conjuncts of a primitive refactoring that are not logically entailed by any of the FGT-enabling preconditions of the FGT-DAGs from which the primitive refactoring is constructed. These isolated conjuncts will be referred to as **refactoring-level precondition conjuncts**. In principle, therefore, a refactoring that is specified as a set of FGT-DAGs will preserve a system's behaviour if the system initially complies with the refactoring-level precondition conjuncts, and also complies with the FGT-enabling preconditions of all FGT-DAGs in the set. Figure 4.4(a), which shows the FGT-DAGs for a fictitious primitive refactoring, thus also portrays the refactoring-level precondition, as well as FGT-enabling preconditions.

In the present text, the focus is on precondition conjuncts. However, postconditions can also be viewed as being at the refactoring-level as well as at the FGT-level. These notions are abstractly portrayed in Figures 4.4(a) and 4.4(b) with respect to an FGT approach and previous approaches respectively.

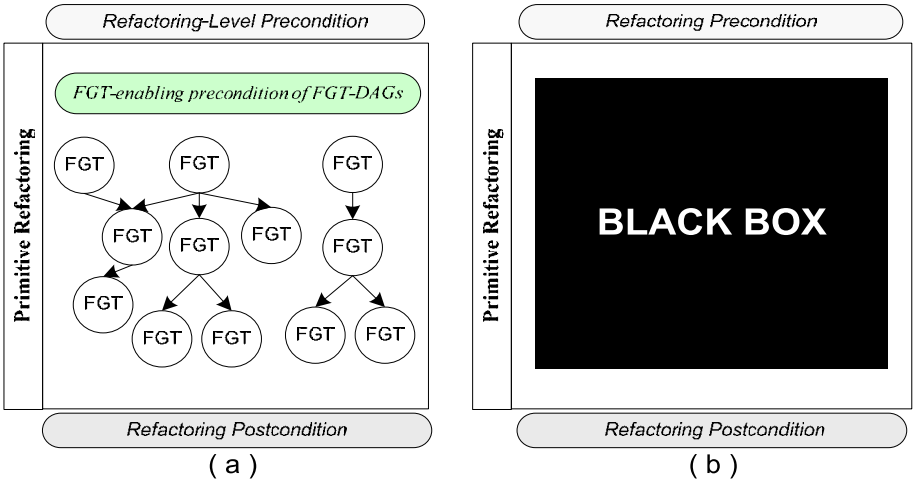


Figure 4.4: Primitive refactoring different considerations

4.4.4 Applying Refactorings

Using a tool to apply a specific refactoring to the system is done in two phases: in the first phase, the tool has to check both the refactoring-level precondition as well as all the FGT-enabling preconditions of the various FGT-DAGs. If these are satisfied then it proceeds to the second phase in which the refactoring itself is applied to the system—i.e. the tool's code that updates the tool's representation of the UML model.

Dealing with two levels of precondition conjuncts introduces the followings themes:

a. As explained in chapter 7, when an FGT is cancelled or absorbed by the reduction process, then its set of precondition conjuncts will also be cancelled or absorbed, which means that the overall number of refactoring precondition conjuncts may potentially be reduced. The overall effect will be to reduce the number of precondition conjuncts that need to be checked, potentially enhancing the performance of the refactoring tool.

b. Consideration should be given to the parallelizing opportunity at the time of checking the precondition conjuncts of FGTs and also at the time of applying that FGTs. (Addressed in chapter 11)

c. Because the precondition conjuncts of the FGTs are predefining and pre-implemented in the refactoring tool, an end user of the refactoring tool who chooses to define a new refactoring merely has to be concerned with the precondition conjuncts at the refactoring-level. (Addressed in chapter 12)

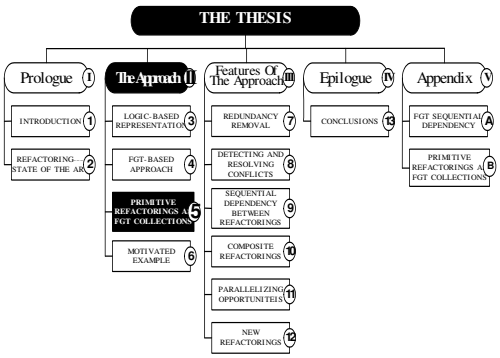
4.5 Reflection on this Chapter

This chapter has introduced the notion of FGTs and catalogued those relevant to this thesis, together with their associated precondition conjuncts. It has suggested that a collection of such FGTs, ordered in a set of FGT-DAGs, can be used to transform a system. It has also suggested that where such transformations constitute a refactoring, certain refactoring-level precondition conjuncts can be isolated from FGT-level precondition conjuncts and be processed separately. The remainder of the thesis elaborates on the consequence of using FGTs in this fashion.

CHAPTER 5

PRIMITIVE REFACTORINGS AS FGT COLLECTIONS

5.1 Introduction



This chapter elaborates on the feasibility of representing primitive refactorings as a collection of FGTs. The term "collection" is used here to designate either an FGT-list or an equivalent set of FGT-DAGs, as discussed in the previous chapter. Twenty-nine well-known primitive refactorings that are frequently defined and used in refactoring literatures will be introduced [22, 60, 65, and 66]. Each primitive refactoring is represented as a collection of FGTs instead of implementing it as a piece of code (*black box*). The chapter shows that some of these primitive refactorings can be represented by a single FGT while others need the application of several FGTs in an FGT-list. The chapter also discusses the relationship between the precondition conjuncts of the primitive refactorings and the precondition conjuncts of the associated FGTs.

The concern in this thesis is to propose a new approach to formalize model refactorings. It is beyond of the scope of this thesis to discuss in detail the theme of so-called ``*code-smells*``— i.e. to identify opportunities for refactoring in the system and to propose suitable refactorings to be use in the presence of such code-smells. This is, in fact, an entirely different area of research in the field of refactorings. There exist a number of detection tools that automatically detect opportunities for refactorings on the system [76, 85]. Such tools are based on various metrics of software quality and other techniques.

The primitive refactorings discussed in this chapter are categorized into three groups according to the kind of transformations they make on the underlying system: the first group, 'Add Element Refactorings', includes all refactorings that, when executed, will add elements to the system. These elements may be *object* or *relational* elements. The second group, 'Change Element Refactorings', includes all refactorings that, when executed, will change the characteristics of the element such as name, access mode or definition type. Alternatively, they

will change the structure of the elements in the system by moving the element from one place to another. The third group, 'Delete Element Refactorings', includes all refactorings that will delete element or elements from the system under consideration. The list of primitive refactorings in each group is:

1. Add Element Refactorings
   a. addClass
   b. addMethod
   c. addAttribute
   d. addParameter
   e. addGetter
   f. addSetter
2. Change Element Refactorings

2.1 Change Characteristics
   a. renameClass
   b. renameMethod
   c. renameAttribute
   d. renameParameter
   e. changeClassAccess
   f. changeMethodAccess
   g. changeAttributeAccess
   h. changeMethodRetType
   i. changeAttributeType
   j. changeParameterType

2.2 Change Structure (Restructuring)
   a. changeSuper
   b. moveMethod
   c. moveAttribute
   d. attributeReadsToMethodCall
   e. attributeWritesToMethodCall
   f. pullUpMethod
   g. pushDownMethod
   h. pullUpAttribute ()
   i. pushDownAttribute

3. Delete Element Refactorings
   a. deleteClass

b. deleteMethod
c. deleteAttribute
d. deleteParameter

These twenty-nine primitive refactorings have been stored in the Prolog prototype tool as generic (i.e. uninstantiated) FGT-lists. In order to generate a particular refactoring that relates to elements in the system's representation of an UML class diagram, the name of the refactoring and its instantiated parameters are provided to the tool. The tool then instantiates the relevant stored FGT-list. Subsequently, the algorithm given in 4.3.4 may be used to build the corresponding set of FGT-DAGs. The refactoring may then be applied, and the system's representation will be changed accordingly.

In what follows, selected primitive refactorings and their mappings to FGTs will be discussed. In each case, the following headings will be used: Parameters; Description; Precondition Conjuncts; FGTs in the order of the stored FGT-list; followed by a note that relates the FGT and primitive refactoring precondition conjuncts. Primitive refactorings that map to a single FGT will not be given here, but—for completeness—will be discussed under the same headings in Appendix B.

In some cases, it will be seen that sequential compliance with the FGT precondition conjuncts that make up a primitive refactoring is sufficient to guarantee system behaviour as well. In this case, the FGT precondition conjuncts are said to cover the primitive refactoring precondition conjuncts.

Note that by sequential compliance is meant that the FGTs precondition holds at the point at which the FGT is about to be applied—not that the conjunction of all FGTs making up a primitive refactoring hold from the start. The claim that FGT precondition conjuncts cover the primitive refactoring precondition conjuncts is therefore not the same as the claim that the conjunction of FGT precondition conjuncts logically entails the precondition conjuncts of the associated primitive refactoring.

It will also be seen that in some cases, the precondition conjuncts of the FGTs do not cover the precondition conjuncts of the associated primitive refactoring. Mere compliance with FGT precondition conjuncts will therefore not necessarily guarantee behaviour preservation. In such cases, it is necessary to define so-called refactoring-level precondition conjuncts. To guarantee behaviour preservation, compliance with these should be checked before checking as mentioned in 4.4 and applying the constituent FGTs.

5.2 Add Element Refactorings

Refactorings in this group are used to add new elements to the software. The first four refactorings in the group are used to add class, method, attribute and parameter object elements to the system, while the last two refactorings are used to add getter and setter methods for specific attributes in the system. From a formal point of view, the first four refactorings are a behaviour-preserving—they do not change the behaviour of the system after refactoring—because none of the elements that they add are referenced in the system. The last two refactorings, addGetter and addSetter, are also behaviour-preserving because, even though the added methods (getter and setter) reference one of the existing attributes in the system, these methods (getter and setter) themselves are unreferenced from anywhere in the system.

5.2.1 addClass(*ClassName, AccessMode*)

The refactoring adds a new class to the system under consideration. The created class will be empty and standalone (no members, super or subclasses). (*For more details see Appendix B.1.1*)

5.2.2 addMethod(*MethodName, ReturnDType, AccessMode, ParameterList*)

The refactoring adds a new method in one of the classes of the system under consideration. (*For more details see Appendix B.1.2*)

5.2.3 addAttribute(*AttibuteName, AttributeDType, AccessMode*)

The refactoring adds a new attribute in one of the classes of the system under consideration. (*For more details see Appendix B.1.3*)

5.2.4 addParameter(*Prmname, PrmDType, Index, MethTList*)

The refactoring declares a new parameter in one of the methods of the system under consideration. (*For more details see Appendix B.1.4*)

5.2.5 addGetter(*AttributeName*)

Where *AttributeName* has the following format: *Pn.Cn.Attn* (*Pn* is the name of the package, *Cn* is the name of the class and *Attn* is the name of the attribute).

Description

The refactoring adds a getter method in the class *Pn.Cn*. This method is used to return (*get*) the value of the attribute *Attn* that is defined in the class *Pn.Cn*. Hence, the definition type of the return value of the getter method is the same as that of the attribute *Attn*.

Figure 5.1 shows the effect of the refactoring addGetter when it is applied to the *private* attribute *A.x* using: addGetter(*A.x*).

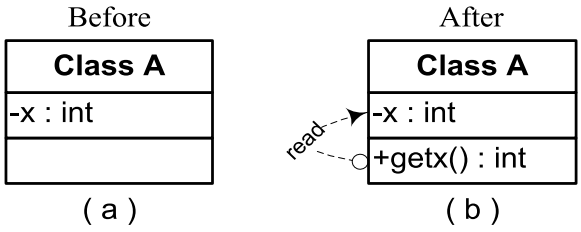


Figure 5.1: Class A before and after addGetter(*A.x*)

Precondition Conjuncts

(1) The signature of the getter method is distinct from those of all methods declared already in the class *Pn.Cn* and of any of its ancestors.

(2) The attribute *AttributeName* is declared in the class *Pn.Cn*.

FGT-List

1. addObject(*Pn,Cn, Methn,_,_,AttType ,public,[],method*)
2. addRelation(*_,Pn,Cn,Methn,_,[],method,Pn,Cn,Attn,_,_,attribute,read*)

Note

- FGT 1 in the FGT-list is used to add the getter method with no parameters. The name of the getter method *Methn* is formulated automatically by using the procedure **concat**(*'get', Attn, Methn*). The procedure concatenates the word *'get'* with the attribute *Attn*. The return type of the method is the same as the definition type of the attribute *Attn* because the intention of the getter method is to retrieve the value of that attribute. The procedures **getType**(*Pn,Cn, Attn, attribute, AttType*) is used to retrieve the definition type of the attribute under consideration.

- FGT 2 in the FGT-list is used to add a *read* relation between the created getter method as a source of the *relation* and the attribute *Attn* as a destination. The *read* relation between the two *objects* is an indication that the getter method has *read* access to the attribute *Attn.* This means that one or more statements in *Methn* will have a *read* access on the value of the attribute *Attn*. This will be reflected at the code-level.

- Precondition conjunct (1) is covered by precondition conjuncts of the FGT 1 in the FGT-list (*section 4.2.1.1.B*). Precondition conjunct (2) is covered by precondition conjuncts of the FGT 2 in the FGT-list (*section 4.2.2.1.A*). There is no need to add precondition conjuncts at the refactoring-level.

Note that this refactoring indeed preserves system behaviour, but is matter futile if applied on its own. Normally, it will be applied in a context where *Attn* is being accessed directly, and there is a need to encapsulate it. To do this, several more primitive refactorings need to be applied. Chapter 6 provides an example of how such encapsulation may be achieved by the application of various primitive refactorings, which together may be viewed as an example of a composite refactoring called **encapsulateAttribute**.

5.2.6 addSetter(*AttributeName*)

Where *AttributeName* has the following format: *Pn.Cn.Attn*.

Description

The refactoring adds a setter method in the class *Pn.Cn*, the intention of this method is to be used to set the value of the attribute *Attn* that is defined in the class *Pn.Cn*. For that the setter method has a parameter whose definition type is the same as the definition type of the attribute *Attn*.

Figure 5.2 shows the effect of the refactoring addSetter when it is applied on the *private* attribute *A.x* using: addSetter(*A.x*) .

Precondition Conjuncts

(1) The signature of the setter method is distinct from those all methods declared already in the class *Pn.Cn* or any of its ancestors.

(2) The attribute *Attn* is declared in the class *Pn.Cn*.

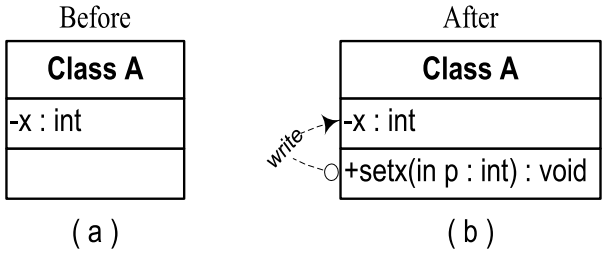


Figure 5.2: Class A before and after addSetter(*A.x*)

FGT-List

1. addObject(*Pn,Cn, Methn,_,_,type(basic,void,0),public,[(p,AttType)],method*)
2. addRelation(*_,Pn,Cn,Methn,_,[Tname],method,Pn,Cn,Attn,_,_,attribute,write*)

Note

- FGT 1 in the FGT-list is used to add the setter method, the name of the setter method *Methn* is formulated by using the procedure **concat**(*'set', Attn, Methn*) that is used to concatenate the word *'set'* with the attribute *Attn*. The return type of the method is *void* because the setter method returns no values. The setter method has only one parameter which has the same definition type as the definition type of the attribute *Attn* because the intention of the setter method is to set (change) the value of that attribute by using this parameter. The procedure **getType**(*Pn,Cn, Attn, AttType*) is used to retrieve the definition type of the attribute under consideration.

- FGT 2 in the FGT-list is used to add a *write* relation between the created setter method as a source of the *relation* and the attribute *Attn* as a destination. The *write* relation between the two *objects* is an indication that the setter method has a *write* access on the attribute *Attn*. The procedure **typeName**(*AttType, Tname*) is used to retrieve the type name (*int, float, …*) of the *AttType*. *Tname* is used in FGT 2 to specify the signature of the method *Methn*.

- Precondition conjunct (1) is covered by precondition of the FGT 1 in the FGT-list (*section 4.2.1.1.B*). Precondition conjunct (2) is covered by precondition conjuncts of the FGT 2 in the FGT-list (*section 4.2.2.1.A*). There is no need to add precondition conjuncts at the refactoring-level.

5.3 Change Element Refactorings

These refactorings can be divided into two groups. The first group includes refactorings that are used to change the characteristics of the *object* elements in the system by changing the name, access mode and definition type of *object* elements. Note that one of the features of the proposed refactoring tool is that all the references to the *object* elements are done through the ID of the *object* elements and not through their names, so when we change the name of the *object* element, for example, then there is no need to change any references to that *object*.

The second group includes refactorings that are used to restructure *object* elements in the system by changing the hierarchal *relations* between *objects* or by moving *object* elements from one place to another or by redirecting member's accesses from one *object* element to another.

5.3.1 Changing Characteristics

5.3.1.1 renameClass(*ClassName, NewName*)

The refactoring changes the name of a class. (*For more details see Appendix B.2.1*)

5.3.1.2 renameMethod(*MethodName, MethTList, NewName*)

The refactoring changes the name of a method. (*For more details see Appendix B.2.2*)

5.3.1.3 renameAttribute(*AttributeName, NewName*)

The refactoring changes the name of an attribute. (*For more details see Appendix B.2.3*)

5.3.1.4 renameParameter(*ParameterName, MethTList, NewName*)

The refactoring changes the name of a parameter. (*For more details see Appendix B.2.4*)

5.3.1.5 changeClassAccess(*ClassName, NewAcces*)

The refactoring changes the access mode of a class. (*For more details see Appendix B.3.1*)

5.3.1.6 changeMethodAccess(*Methname, MethTList, NewAccess*)

The refactoring changes the access mode of a method. (*For more details see Appendix B.3.2*)

5.3.1.7 changeAttributeAccess(*AttributeName, NewAccess*)

The refactoring changes the access mode of an attribute. (*For more details see Appendix B.3.3*)

5.3.1.8 changeMethodReturnType(*Methodname, MethTList, NewRType*)

The refactoring changes the definition type of the return value of a method. (*For more details see Appendix B.3.4*)

5.3.1.9 changeAttributeDefType(*AttributeName, NewDType*)

The refactoring changes the definition type of an attribute. (*For more details see Appendix B.3.5*)

5.3.1.10 changeParameterDefType(*Parametername, MethTList, NewDType*)

The refactoring changes the definition type of a parameter. (*For more details see Appendix B.3.6*)

5.3.2 Change Structure (Restructuring)

5.3.2.1 changeSuper(*ClassName, NewSuper*)

Where

- *ClassName* has the following format: *Pn.Cn*
- *NewSuper* has the following format: *NewPn.NewCn*

Description

The refactoring changes the superclass of the class *Pn.Cn* to a new class *NewPn.NewCn*.

Precondition Conjuncts

(1) Members of the old superclass or any of its ancestors are not referenced by instances of the class *Pn.Cn* or any of its descendents.

FGT-List

1. deleteRelation(_,*OldPn, OldCn*,_,_,_,*class, Pn,Cn*, _,_,_,*class, extends*)

2. addRelation(*isa, NewPn, NewCn,_,_,_, class, Pn,Cn, _,_,_,class, extends*)

Note

- In order to find the superclass of the class *Pn.Cn* we use the procedure **supClass**(*OldPn, OldCn, Pn,Cn*)

- FGT 1 is used to delete the *extends* relation between the old superclass *OldPn.OldCn* and the class *Pn.Cn*.

- FGT 2 is used to add the *extends* relation between the new superclass *NewPn.NewCn* and the class *Pn.Cn*.

- Note that even if class *Pn.Cn* or any of its descendants have a member *x* that is defined in the class *NewPn.NewCn* or any of its ancestor classes, adding the *extends* relation between the two classes will not cause a redefining of the member *x*. *Pn.Cn* and its descendants will still use their version of *x*. Thus, member *x* that is defined in the class *Pn.Cn* or one of its descendants is not affected by adding the *extends* relation.

- Also note that the new members that the class *Pn.Cn* and its descendants will inherit from the new superclass will not affect the behaviour of the system because these inherited members are not referenced by any instance or member of the class *Pn.Cn* or its descendants.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 in the FGT-list (*section 4.2.2.3.B*). There is no need to add precondition conjuncts at the refactoring-level.

5.3.2.2 moveMethod(*MethodName, NewClassName, MethTList*)

Where

- *MethodName* has the following format: *Pn.Cn.Methn*
- *NewClassName* has the following format: *NPn.NCn*
- *MethTList* has the following format: *[Tname$_1$, Tname$_2$,…., Tname$_n$]*

Description

The refactoring moves a method from one class to another. The developer may need to do this when the two classes are highly coupled and the method to be moved *Methn* is extensively accessed members that are defined in the destination class. In this case, the developer concludes that the method is more related to the destination class and moving it will make the system more readable and simple. For example, in Figure 5.3(a), method *B.m* accesses the *private* attribute *A.x* through its getter and setter methods. The figure shows that method *B.m* does not access any members in the class *B*. It is reasonable to conclude that the method *B.m* is more related to the class *A* than class *B* and a developer might therefore prefer to move the method to the class A. For this, refactoring **moveMethod**(*B.m, A, [int]*) may be used. Note that in order to serve all the accesses (*calls*) to *B.m* from the other *object* elements in the system, the tool adds a method with the same signature in the source class *B*. Then a *call* relation is created between the two methods in the two classes. All the existing accesses to the method *B.m* will be now redirected to method *m* in its new location *A.m*.

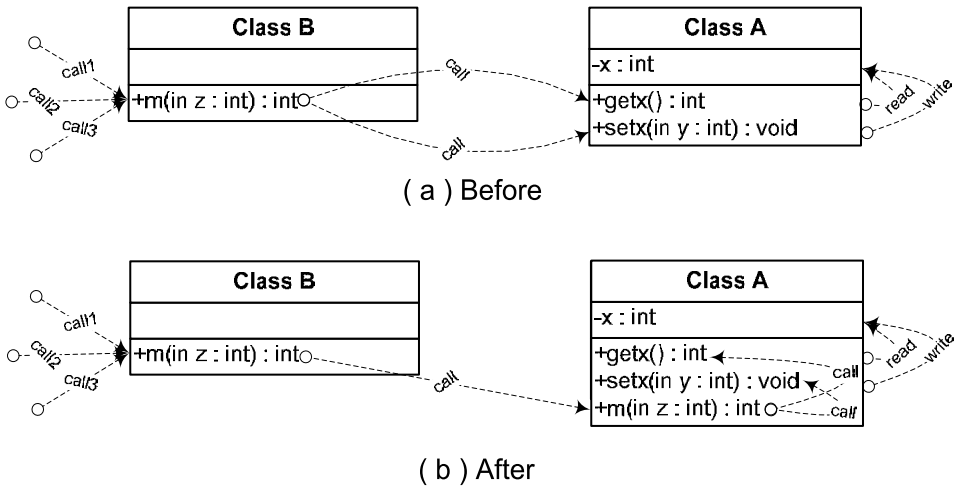


Figure 5.3: Class A & B before and after moveMethod(*B.m, A, [int]*)

Precondition Conjuncts

(1) The signature of the method *Methn* is distinct from those all methods declared already in the class *NPn.NCn* or any of its ancestors.

FGT-List

1. **For** each relational element that the *Methn* is the source object of **do** {

deleteRelation(_,*Pn,Cn,Methn,_,PrmLT,method,*TPn_i,TCn_i,$TMem_i$,$TPrm_i$, $TPrmLT_i$, TOT_i,$RelType_i$) }

2. addObject(*NPn, NCn, Methn,_,_, RetDefT, public, MethTList, method*)
3. **For** each relational element deleted in stage 1 **do** {

   addRelation(_,*NPn,NCn,Methn,_,PrmLT,method,*TPn_i,TCn_i,$TMem_i$,$TPrm_i$, $TPrmLT_i$, TOT_i,$RelType_i$) }

4. addRelation(_,*Pn,Cn,Methn,_,PrmLT,method,NPn,NCn,Methn,_,PrmLT, method, call*)

Note

- Stage 1 is used to delete all the *relational* elements that exist between the method *Methn* and any other *object* elements in the system, where the method *Methn* is the source of the *relation.* Thus, all *Methn* accesses to the other *objects* will be deleted. Note that this stage will generate a deleteRelation FGT for each existing *relation*. In Figure 5.3(a) the two *call* relations from the method *B.m* to the methods *A.gets* and *A.setx* will be deleted at this stage.

- FGT 2 is used to add a new method with the same signature as *Methn* to the destination class *NPn.NCn*. Figure 5.3(b) shows that the method *m* is added to the class *A*.

- All the *relational* elements that were deleted during stage 1 will be added by stage 3 with the newly created method in the destination class *NPn.NCn* as a source of these *relational* elements. Figure 5.3(b) shows that the two *call* relations that were deleted during stage 1 are added between the method *A.m* as a source and the two methods *A.setx* and *A.getx* as destinations.

- FGT 4 is used to create a *relational* element of type *call* between the method *Pn.Cn.Methn* and the new method *NPn.NCn.Methn*. The purpose of this *relation* is to forward all the accesses from all *object* elements to the method *Methn* in its old location to its new location. Figure 5.3(b) shows that a new *call* relation is created between the method *B.m* and the method *A.m*, so all the accesses to the method *B.m* is still valid and forwarded to the method *A.m*.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 2. There is no need to add precondition conjuncts at the refactoring-level.

5.3.2.3 moveAttribute(*AttributeName, NewClassName*)

Where

- *AttributeName* has the following format: *Pn.Cn.Attn*
- *NewClassName* has the following format: *NPn.NCn*

Description

The refactoring moves an attribute from one class to another. This primitive refactoring is typically used if the attribute under consideration is intensively accessed—through its getter and setter methods—by *object* members defined in another class than by members defined in its own class. As shown in Figure 5.4(a) below, there are many accesses from methods in the class *B* to the attribute *A.x*. In this case it is preferred to move the attribute from class *A* to class *B*, and for this refactoring **moveAttribute**(*A.x, B*) is used.

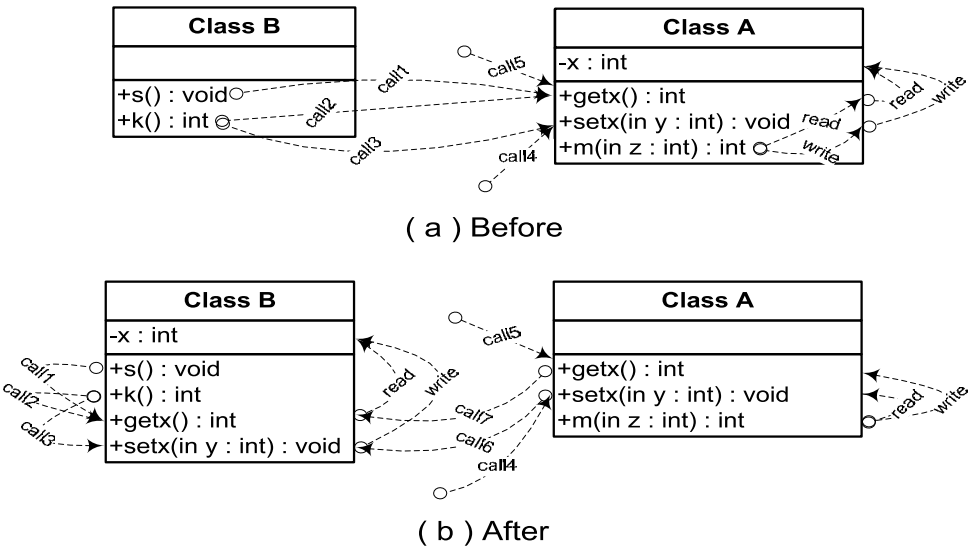


Figure 5.4: Class A & B before and after moveAttribute(*A.x, B*).

Precondition Conjuncts

(1) The attribute *Attn* is distinct from those all attributes that are declared already in the class *NPn.NCn* or any of its ancestors.

FGT-List

1. deleteRelation(_,*Pn,Cn, getMethn,_,[],method,Pn,Cn,Attn,_,_,attribute,read*)
2. deleteRelation(_,*Pn,Cn, setMethn,_,[Tname],method,Pn,Cn, Attn,_,_,attribute, write*)
3. deleteObject(*Pn,Cn, Attn,_,_,attribute*)
4. addObject(*NPn,NCn, Attn,_,_,AttType, private,_,attribute*)
5. **addGetter**(*NPn.NCn.Attn*)
6. **addSetter**(*Npn.NCn.Attn*)
7. addRelation(_,*Pn,Cn, getMethn,_,[],method,NPn,NCn, getMethn,_,[], method, call*)
8. addRelation(_,*Pn,Cn, setMethn,_,[Tname],method,NPn,NCn, setMethn,_, [Tname],method, call*)
9. deleteRelation(_,*NPn,NCn, $NMem_i$,_,_,NOT_i,Pn,Cn, getMethn,_,[],method,call*)
10. addRelation(_,*NPn,NCn, $NMem_i$,_,_,NOT_i,NPn,NCn, getMethn,_,[],method, call*)
11. deleteRelation(_,*NPn,NCn, $NMem_i$,_,_,NOT_i,Pn,Cn, setMethn,_,[Tname], method,call*)
12. addRelation(_,*NPn,NCn, $NMem_i$,_,_,NOT_i,NPn,NCn, setMethn,_,[Tname], method, call*)

Note

- FGTs 1 and 2 are used to delete the *read/write* relations in the source class *Pn.Cn* between the getter/setter methods and the attribute *Attn*. In Figure 5.4(a) the two *read/write* relations from the methods *A.getx/A.setx* to the attribute *A.x* will be deleted at this stage.

- FGT 3 is used to delete the attribute *Attn* from the source class *Pn.Cn*.

- FGT 4 is used to add the attribute *Attn* to the destination class *NPn.NCn*.

- Then in stages 5 and 6, the two refactorings **addGetter** (describe in section 5.1.5) and **addSetter** (describe in section 5.1.6) are used to create a getter and a setter methods for the attribute *NPn.NCn.Attn*. Figure 5.4(b) shows that *read/write* relations are created from the methods *B.getx/B.setx* to the attribute *B.x*.

- FGTs 7 and 8 are used to create a *call* relations between the getter/setter methods in the source and the destination classes. Figure 5.4(b) shows two *call* relations (*call6* and *call7*) are created from *A.getx/A.setx* to *B.getx/B.setx*.

- FGTs 9 to 12 are used to redirect all the *call* relations from the class *NPn.NCn* to the getter/setter methods in the class *Pn.Cn.* These calls are redirected to the new getter/setter methods in the class *NPn.NCn*. In Figure 5.4(b) the relations *call1, call2* and *call3* are redirected to the methods *B.getx/B.setx*

- Precondition conjunct (1) is covered by the set of precondition conjuncts of the FGT 4**.** There is no need to add precondition conjuncts at the refactoring-level.

5.3.2.4 attributeReadsToMethodCall(*AttributeName, MethodName, MethTList*)

Where

- *AttributeName* has the following format: *Pn.Cn.Attn*
- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[Tname$_1$, Tname$_2$,…., Tname$_n$]*

Description

The refactoring redirects all *read* accesses to a specific attribute *Attn* to be through a getter method *Methn*. The method *Methn* will return the value of the attribute *Attn* to the calling *object* element**.**

Precondition Conjuncts

(1) The access mode of the method *Methn* is equal to or less restricted than the access mode of the attribute *Attn*. This ensures that all the *read* accesses to the attribute *Attn* will be within the access scope of the method.

(2) The method *Methn* acts as a getter method to the attribute *Attn*. This means that when the method *Methn* is called it will return the value of the *Attn* to the calling object.

FGT-List

For each relational element of type *read* with *Pn.Cn.Attn* as the destination object **do** {

1. deleteRelation(_,*SPn$_i$, SCn$_i$,SMethn$_i$*,_,*PrmLT$_i$,SOT$_i$, Pn,Cn, Attn*,_,_, *attribute, read*)
2. addRelation(_,*SPn$_i$, SCn$_i$,SMethn$_i$*,_,*PrmLT$_i$,SOT$_i$, Pn,Cn,Methn*,_, *MethTList, method,call*) **}**

Note

- The destination of all *relational* elements—whose destination *object* is the attribute *Pn.Cn.Attn* and whose type is *read*—will be changed to be *Pn.Cn.Methn* instead of *Pn.Cn.Attn*.

- Precondition conjunct (1) is covered by the precondition of FGT 2 because in order to create the *call* relation, the destination method *Pn.Cn.Methn* should be accessible. Precondition conjunct (2) is not covered by precondition conjuncts of the FGTs in the FGT-list because there is no guarantee that the method *Methn* acts as a getter method to the attribute *Attn*. In order for the method *Methn* to be a getter method of the attribute *Attn,* the return type of the *Methn* should be the same as the return type of the *Attn. In addition* there has to be a *read* relation between the *Methn* and the *Attn*. These precondition conjuncts will therefore need to be specified at the refactoring-level of this primitive refactoring.

5.3.2.5 attributeWritesToMethodCall(*AttributeName, MethodName, MethTList*)

Where

- *AttributeName* has the following format: *Pn.Cn.Attn*
- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[Tname$_1$, Tname$_2$,…., Tname$_n$]*

Description

The refactoring redirects all the *write* accesses to a specific attribute *Attn* to be through a setter method *Methn*. The setter method *Methn* will receive a value from the calling *object* element and set the value of the attribute accordingly.

Precondition Conjuncts

(1) The access mode of the method *Methn* is equal to or less restricted than the access mode of the attribute *Attn.* This ensures that all the accesses to the attribute will be within the access scope of the method.

(2) The method *Methn* acts as a setter method to the attribute *Attn*. This means that when the method *Methn* is called it will receive a value of the same definition type as the *Attn* and the method will set the value of the *Attn* to this value.

FGT-List

For each relational elements of type *write* and *Pn.Cn.Attn* is the destination object **do** {

1. deleteRelation(_,SPn_i,SCn_i,$SMehN_i$,_,$PrmLT_i$,SOT_i, *Pn,Cn,Attn,_ ,_, attribute, write*)
2. addRelation(_,SPn_i,SCn_i,$SMehN_i$,_,$PrmLT_i$,SOT_i, *Pn,Cn,Methn,_,[Tname],method, call*)

}

Note

- The destination of all *relational* elements—whose destination object is attribute *Pn.Cn.Attn* and whose *relation* type is *write*—will be changed to be *Pn.Cn.Methn* instead of *Pn.Cn.Attn*.

- Precondition conjunct (1) is covered by FGT 2 because in order to create the *call* relation, the destination method *Pn.Cn.Methn* should be accessible. Precondition conjunct (2) is not covered by precondition conjuncts of the FGTs in the FGT-list because even though the FGT 2 ensures that the *Methn* has a parameter of the same type as the attribute *Attn*, there is no guarantee that the method *Methn* has *write* access to the attribute *Attn*. For this we need to check if there is a *write* relation between the method *Methn* and the attribute *Attn*. This precondition conjunct will be defined at the refactoring-level of this primitive refactoring.

5.3.2.6 pullUpMethod(*SubClassesNames, Methn, MethTList*)

Where

- *SubClassesNames* has the following format: *[($SubPn_1$,$SubCn_1$), ($SubPn_2$, $SubCn_2$),.., ($SubPn_n$, $SubCn_n$)]* where items in the list represent the names of the subclasses that the refactoring will pull the method from.
- *MethTList* has the following format: *[$Tname_1$, $Tname_2$,…., $Tname_n$]*

Description

The refactoring pulls up a method *Methn* from a list of subclasses *SubClassesNames* to their common superclass. If all subclasses in list have the same method with the same signature and the same effect. Then inconsistencies caused by not changing all these methods equally can be avoided by pulling up this method to their common superclass. It is clear that pulling the

method up will not affect the behaviour of the system because all the subclasses after refactoring will have this method by inheritance.

The access mode of the method *Methn* should not be more general than the access modes of the corresponding versions of the method in the various subclasses—i.e. it should be *protected* if it is *protected* in one or more subclasses, and otherwise (if it is *public* in all subclasses) it should be *public*.

Precondition Conjuncts

(1) The method *Methn* should not be declared in the superclass nor in any of its ancestors.

(2) The access mode of the method *Methn* in the subclasses is not *private*.

(3) All the references made by *Methn* to the other *object* elements should be visible from the superclass.

(4) The signature of *Methn* in all the subclasses in the list *SubClassesNames* should be the same.

Note 1: Precondition conjunct (4) is not necessarily sufficient to legitimate a pull up refactoring. In addition in should be the case that the postcondition conjuncts of the various methods in the subclasses are compatible. Technically, one might say that the postcondition of at least one method should logically entail the postcondition conjuncts of all the others. In this case, the method with the strictest postcondition should be pulled up. (Further explanation of this point is beyond the scope of this thesis.)

However, the prototype refactoring tool built for the purposes of this thesis does not require that postcondition information should be available. It merely considers information embedded in UML class diagrams as well as some limited information embedded in the code. It is therefore the responsibility of the tool user to ascertain the compatibility of method postcondition conjuncts before carrying out a pull up refactoring. The tool will, however, check compliance with precondition four as an approximation of the more rigorous requirement of postcondition compatibility.

FGT-List

1. addObject(*SupPn,SupCn,Methn,_,_,MethRType,OAMode,MethTList, method*)
2. **For** each subclass in the *SubClassesNames* list **do** {

deleteObject(*SubPn$_i$,SubCn$_i$, Methn,_, MethTList, method*) }

Note

- FGT 1 is used to add the method *Methn* in the superclass with the same signature as in the subclasses. The method access mode *OAMode* is calculated according to the rule mentioned above. For this, the procedure **objectAMode**(*SubPn$_i$, SubCn$_i$, Methn, MethTList, method, SubOAmode*) is used.

- In stage 2, method *Methn* will be deleted from each one of the subclasses that is found in the list *SubClassesNames*.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 in the FGT-list (*section 4.2.1.1.B*). Precondition conjuncts (2) and (3) are not covered by precondition conjuncts of FGTs in the FGT-list and should be defined as refactoring-level precondition conjuncts for this refactoring. Precondition conjunct (4) is covered by FGTs in stage 2.

5.3.2.7 pushDownMethod(*SuperClassName, MethodName, MethTList*)

Where

- *SuperClassName* has the following format: *SupPn.SupCn*
- *MethodName* has the following format: *Pn.Cn.Methn*
- *MethTList* has the following format: *[Tname$_1$, Tname$_2$,…., Tname$_n$]*

Description

The refactoring pushes down a method *Methn* from a superclass to all its subclasses. This refactoring can be used if the *Methn* is not referenced in some of the subclasses. In such a case, this refactoring is used to push down the method to all the subclasses. It is thereafter deleted from those subclasses where it is not referenced, using the **deleteMethod** refactoring. The access mode of *Methn* in all the subclasses will be the same as its access mode in the superclass.

Precondition Conjuncts

(1) The method *Methn* should not be declared in any of the subclasses of the superclass.

(2) The method *Methn* should not be referenced by members of the superclass, since it will be deleted from the superclass and these referenced will be not defined anymore.

(3) The method *Methn* should not access any of the *private* members of the superclass.

(4) The access mode of the method *Methn* in the superclass should not be *private*.

FGT-List

1. **For** each subclass of the class *SuperClassName* **do** {

   addObject(*SubPn*$_i$, *SubCn*$_i$, *Methn*,_,_,*MethType, OAMode, MethTList, method*) }

2. deleteObject(*SupPn, SupCn, Methn, _, MethTList, method*)

Note

- Stage 1 is used to add the method *Methn* in all the subclasses of the class *SupClassName*. The signature of the method will be the same as defined in the superclass.

- FGT 2 is used to delete the method *Methn* from the superclass.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 in the FGT-list (*section 4.2.1.1.B*). Precondition conjuncts (2), (3) and (4) are not covered by precondition conjuncts of FGTs in the FGT-list and should be defined as refactoring-level precondition conjuncts.

5.3.2.8 pullUpAttribute(*SupClassName, Attn*)

Where *SuperClassName* has the following format: *SupPn.SupCn*

Description

The refactoring pulls up an attribute *Attn* to the superclass *SupClassName* from all subclasses where it is defined. If the access mode of the attribute *Attn* where it is currently defined is *public* then it will be *public* in the superclass class; otherwise, the access mode of the *Attn* in the superclass will be *protected*. None of the references to the attribute in the subclasses and their descendants will be affected because they will inherit the attribute from the superclass. The refactoring is thus behaviour-preserving.

Precondition Conjuncts

(1) The attribute that to be pulled up *Attn* should not be declared in the superclass or one of its ancestors.

(2) The attribute *Attn* should be declared identically (have the same definition type) in all the subclasses where it is defined.

(3) The access mode of the attribute *Attn* in the subclasses may not be *private.*

FGT-List

1. addObject(*SupPn,SupCn,Attn,_,_, AttType, OAMode,_,attribute*)
2. **For** each subclass of *SupPn.SupCn* where *Attn* is defined **do {**

   deleteObject($SubPn_i$,$SubCn_i$, *Attn,_,_, attribute*) **}**

Note

- FGT 1 is used to add the attribute *Attn* into the superclass. The definition type of the attribute will be found by the procedures **getType**(…). The attribute access mode *OAMode* is calculated according to the rule mentioned above. For this, the procedure **objectAMode**($SubPn_i$, $SubCn_i$, *Attn, attribute,SubOAmode*) is used.

- In stage 2, the attribute *Attn* will be deleted from each one of the subclasses in which it is defined.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 1 in the FGT-list (*section 4.2.1.1.C*) . Precondition conjunct (2) is not covered by precondition conjuncts of FGTs in the FGT-list. It requires that *Attn* be defined identically in all the subclasses. For checking this, the procedure **checkIdentically**(*Attn, ClassList, Identical*) may be used. The procedure takes as input the name of the attribute and the list of all subclasses where the attribute is defined. It then return true by the parameter *Identical* if the attribute *Attn* has the same definition type in all the classes in the list *ClassList*. Precondition conjunct (3) is not covered by precondition conjuncts of FGTs in the FGT-list. Precondition conjuncts (2) and (3) should be defined as refactoring-level precondition conjuncts.

5.3.2.9 pushDownAttribute(*SupClassName, AttributeName*)

Where

- *SuperClassName* has the following format: *SupPn.SupCn*
- *AttributeName* has the following format: *Pn.Cn.Attn*

Description

The refactoring pushes down an attribute *Attn* from a superclass to all its subclasses. This refactoring is useful if the *Attn* is not referenced in some of the subclasses. In such a case, this refactoring is used to push down the attribute to all the subclasses. It is thereafter deleted from those subclasses where it is not referenced, using the **deleteAttribute** refactoring. The access mode of *Attn* in all the subclasses will be the same as its access mode in the superclass.

Precondition Conjuncts

(1) The attribute *Attn* should be not referenced by members or instances of the superclass.

(2) The access mode of the attribute *Attn* in the superclass should not be *private*.

FGT-List

1. **For** each subclass of the *SupClassName* **do** {

   addObject(*SubPn*$_i$, *SubCn*$_i$, *Attn,_,_, AttType, OAmode,_,attribute*) }

2. deleteObject(*SupPn, SupCn, Attn,_,_, attribute*)

Note

- In stage 1 the attribute *Attn* will be added to all the subclasses of the class *SupClassName*. The definition type and access mode of the attribute will be the same as in the superclass.

- FGT 2 is used to delete the attribute *Attn* from the superclass.

- Precondition conjunct (1) is covered by precondition conjuncts of FGT 2 in the FGT-list (*section 4.2.1.1.C*). Precondition conjunct (2) is not covered by precondition conjuncts of FGTs in the FGT-list and should be defined as a refactoring-level precondition.

5.4 Delete Element Refactorings

These refactorings are used to delete unreferenced *object* elements from system.

5.4.1 deleteClass(*ClassName*)

Where *ClassName* has the following format: *Pn.Cn*

Description

The refactoring deletes unreferenced class *Cn* from the package *Pn*. For this refactoring the class *Pn.Cn* may have a superclass but it should not have any subclasses.

Precondition Conjuncts

(1) The class *Pn.Cn* should not be referenced by any other *object* elements outside the class.

(2) The class *Pn.Cn* has no subclasses.

FGT-List

1. **If** the class to be deleted has a superclass **then**

   deleteRelation(_,*SupPn,SupCn,* _,_,_,_,*Pn,Cn,*_,_, _,_,*extends*)

2. Delete all the relational elements between any two object elements defined in the class *Pn.Cn*

   deleteRelation(_,*Pn,Cn,* _,_,_,_,*Pn,Cn,*_,_, _,_,_)

3. Delete all the methods defined in the class *Pn.Cn*

   deleteObject(*Pn, Cn, Methn*$_i$,_,_, *method*)

4. Delete all the attributes defined in the class *Pn.Cn*

   deleteObject(*Pn, Cn, Attn*$_i$,_,_,*attribute*)

5. deleteObject(*Pn, Cn,*_,_,_, *class*)

Note

- In order to delete a class by using the FGT deleteObject, the class should be empty (no members) and should also stand alone (no superclasses or subclasses). For the refactoring **deleteClass**, one of the precondition conjuncts is that the class should not have subclasses,

although it may have a superclass and it may also have members defined in it. Therefore, to delete the class *Pn.Cn*, the tool should first check if there is a superclass for the class *Pn.Cn* by using the procedure **supClass**(*SupPn,SupCn,Pn,Cn*); if there is then FGT 1 is used to delete the *extends* relation between the classes *SupPn.SupCn* and *Pn.Cn*.

- Although the members of the class *Pn.Cn* are unreferenced by any *object* elements defined outside the class (this is ensured by one of the refactoring precondition conjuncts), references between the different object elements in the class *Pn.Cn* may exist. All these references have to be deleted. Stage 2 in the FGT-list is used for this purpose.

- In stage 3 and 4, all members of the class *Pn.Cn* are deleted.

- FGT 5 is used to delete the class *Pn.Cn*.

- Precondition conjunct (1) is covered by the set of precondition conjuncts of FGTs 3, 4 and 5 of the FGT-list (*section 4.2.1.5*). Precondition conjunct (2) is covered by the set of precondition conjuncts of the FGT 5 because the procedure **isReferenced**(...) will also check if there is any *extends* relation with the class *Pn.Cn*. There is therefore no need to add precondition conjuncts at the refactoring-level for this refactoring.

5.4.2 deleteMethod(*MethodName, MethTList*)

The refactoring deletes an unreferenced method from specific class. (*For more details see Appendix B.4.1*)

5.4.3 deleteAttribute(*AttributeName*)

The refactoring deletes an unreferenced attribute from specific class. (*For more details see Appendix B.4.2*)

5.4.4 deleteParameter(*Prmname, MethTList*)

The refactoring deletes a parameter from the parameter's list of specific method *Methn*. This refactoring is beneficial when, for example, a method's purpose is changed and there is a need to remove (and perhaps later add) parameters from (to) the method. (*For more details see Appendix B.4.3*)

5.5 Reflection on this Chapter

Note that for each primitive refactoring presented in this chapter, there is a corresponding procedure in the prototype tool that receives a set of parameters as input and generates an FGT-list. It should be emphasized that the list of generated FGTs is dependent on the system. For example, the **deleteClass** primitive refactoring (which deletes an unreferenced class from the system) has a corresponding procedure that builds a *system-dependent* FGT-list by carrying out the following steps:

1. **If** the class to be deleted has a superclass **then**

   deleteRelation(_,*SupPn,SupCn,* _,_,_,_,*Pn,Cn,*_,_, _,_,*extends*)

2. Delete all the relational elements between any two object elements defined in the class *Pn.Cn*

   deleteRelation(_,*Pn,Cn,* _,_,_,_,*Pn,Cn,*_,_, _,_,_)

3. Delete all the methods defined in the class *Pn.Cn*

   deleteObject(*Pn, Cn, Methn*$_i$,_,_, *method*)

4. Delete all the attributes defined in the class *Pn.Cn*

   deleteObject(*Pn, Cn, Attn*$_i$,_,_,*attribute*)

5. deleteObject(*Pn, Cn,*_,_,_, *class*)

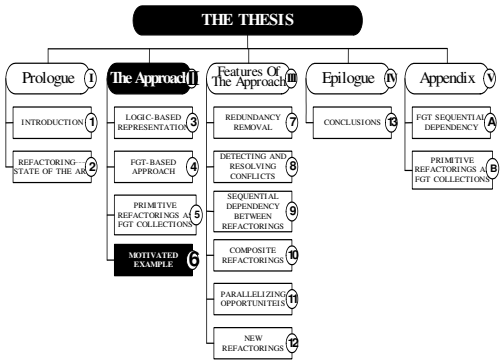
After executing this procedure the list of FGTs is returned. As a second stage a set of FGT-DAGs corresponding to this FGT-list may be generated, as discussed in chapter 4. Recall that there may be one or more FGT-DAGs for that primitive, depending on the sequential dependencies between FGTs in the produced list.

In later chapters, it will be seen that various operations can be performed on arbitrary FGT-DAG sets. For example, it might be possible to reduce them (see chapter 7), to detect conflicts in them (see chapter 8), or to establish sequential dependencies between arbitrary FGT-DAGs (see chapter 9), or representing different composite refactorings from primitive ones (see chapter 10), perhaps process them in parallel (see chapter 11). However, before proceeding to these matters, a motivating example is provided in the next chapter.

CHAPTER 6

MOTIVATED EXAMPLE

6.1 LAN Simulation



To illustrate the approach outlined above, an example is presented that is frequently used for teaching refactoring: the simulation of a Local Area Network (LAN) [13]. Initially there are five classes: *Packet*, *Node* and the three subclasses: *Workstation*, *PrintServer* and *FileServer*. The idea is that all *Node* objects are linked to each other in a token ring network (via the *NextNode* variable) and that they can send or accept a *Packet* object. *PrintServer*, *FileServer* and *Workstation* refine the behaviour of *Node* objects. A *Packet* object can only originate from a *Workstation* object, and sequentially visits every *Node* object in the network until it reaches its receiver that accepts the *Packet,* or until it returns to its originator *Workstation* object (indicating that the *Packet* cannot be delivered).

The UML class diagram for the LAN example is shown in Figure 6.1. The dashed arrows represent the extra information extracted from the code-level of the LAN system which is shown in Figure 6.2. Recall that in the approach the interest of the code-level is limited to the *access-related* information that exists between the different object elements in the class diagram.

Suppose that it is required to enhance the structure of the LAN model as follows:

1. Encapsulate the attribute *originator* in the *Packet* class. This refactoring is useful for increasing modularity, by avoiding direct accesses of the local state of a packet. For this restructuring, the composite refactoring **encapsulateAttribute** will be used.

2. As another enhancement, it has been decided to create a new class *Server* to be a superclass of the *PrintServer*, *FileServer* classes and subclass of the *Node* class. The purpose of this refactoring is to show that the classes *PrintServer* and *FileServer* are similar in nature.

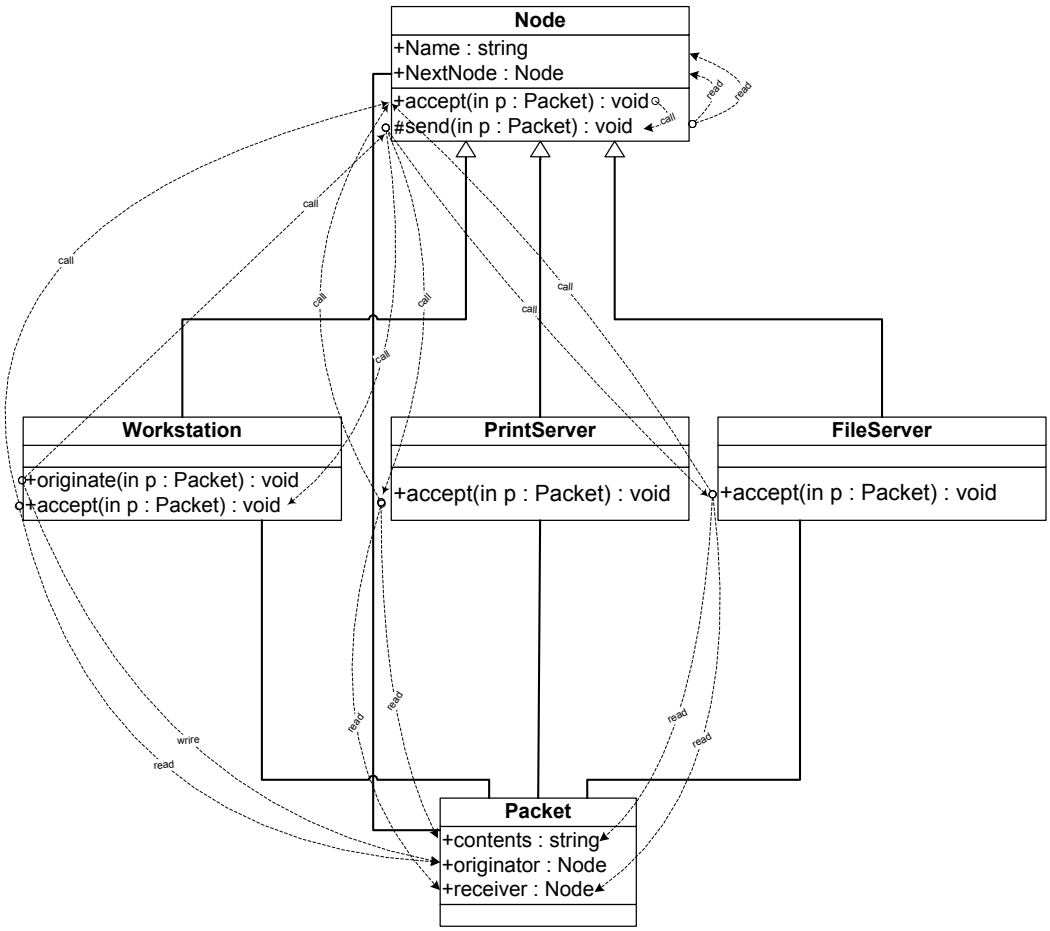


Figure 6.1: A UML class diagram of the LAN simulation before refactoring

```
package Lan;
class Node {
 public String Name;
 public Node NextNode;
 public void accept(Packet p) {
 this.send(p); }
 protected void send(Packet p) {
 System.out.println(Name - NextNode.Name);
 this.NextNode.accept(p); }
}
class Packet {
 public String contents;
 public Node originator;
 public Node receiver;
}
class PrintServer extends Node {
 public void accept(Packet p) {
 if(p.receiver == this)
   System.out.println(p.contents);
 else super.accept(p); }
}
class FileServer extends Node {
 public void accept(Packet p) {
 if(p.receiver == this)
   System.out.println(p.contents);
 else super.accept(p); }
}
class Workstation extends Node {
 public void originate(Packet p) {
 p.originator = this;
 this.send(p); }
 public void accept(Packet p) {
 if(p.originator == this)
   System.err.println("no destination"};
 else super.accept(p); }}
```

Figure 6.2: A code-level implementation of the LAN simulation before refactoring

They can both accept a packet sent by another node in the network and process it in the same way. For this restructuring, the composite refactoring **createClass** will be used.

3. Thereafter, it is intended to pull up the method *accept* from *FileServer*, *PrintServer* classes to the class *Server* that was created in the previous stage. For this restructuring, the primitive refactoring **pullUpMethod** will be used.

The following sections show how the system is represented as a set of logic-terms; and how each of the above composite refactorings can be seen as a sequence of primitive refactorings, each of which can be represented as an FGT-list. Each composite refactoring therefore has a corresponding FGT-list associated with it. The chapter does not focus on mapping these refactoring's FGT-lists to FGT-DAG sets, since the forthcoming chapters will pay considerable attention to FGT-DAG sets. Here, instead, the FGT-lists are assumed to transform the original system to the refactored one. A number of subtleties relating to system representation are pointed out.

It should be noted that because the **encapsulateAttribute** and **createClass** composite refactorings could occur quite commonly, they have been implemented as procedures in the prototype tool. Sections 6.3 and 6.4 will show how they are to be invoked.

6.2 Logic-Based Representation

Before doing any refactoring, the class diagram and the extra information extracted from the code-level should be represented as collection of logic-terms as discussed in chapter 3. Figure 6.3 shows the collection of logic-terms for the LAN simulation example. All refactorings will be done on this underlying representation of the model.

```
package(0,00,Lan,[1,2,3,4,5]).
class(1,0,Node,public,[1001,1002],[10001,10002]).
method(1001,1,send,type(basic,void,0),protected,[100001]).
method(1002, 1, accept,type(basic,void,0),public,[100002]).
attribute(10001,1,Name,type(basic,string,0),public).
attribute(10002,1,NextNode,type(complex,1,0),public).
parameter(100001,1001,p,type(complex,2,0)).
parameter(100002,1002,p,type(complex,2,0)).
call(1000001,_,1002,1001).
call(1000002,_,1001,3002).
call(1000003,_,1001,4002).
call(1000004,_,1001,5002).
read(1000008,_,1001,10001).
read(1000009,_,1001,10002).
extends(10000010,isa,1,3).
extends(10000011,isa,1,4).
extends(10000012,isa,1,5).
class(2,0,Packet,public,[],[20001,20002,20003]).
attribute(20001,2,contents,type(basic,string,0),public).
attribute(20002,2,originator,type(complex,1,0),public).
attribute(20003,2,receiver,type(complex,1,0),public).
```

```
class(3,0,FileServer,public,[3002],[]).
method(3002,3,accept,type(basic,void,0),public,[300002]).
parameter(300002,3002,p,type(complex,2,0)).
read(3000001,_,3002,20001).
read(3000003,_,3002,20003).
call(3000005,_,3002,1002).
class(4,0,PrintServer,public,[4002],[]).
method(4002,4,accept,type(basic,void,0),public,[400002]).
parameter( 400002,4002,p,type(complex,2,0)).
read(4000001,_,4002,20001).
read(4000003,_,4002,20003).
call(4000005,_,4002,1002).
class(5,0,Workstation, public,[5001,5002],[]).
method(5001,5,originate,type(basic,void,0),public,[500001]).
method(5002,5,accept,type(basic,void,0),public,[500002]).
parameter(500001,5001,p,type(complex,2,0)).
parameter(500002,5002,p,type(complex,2,0)).
write(5000001,_,5001,20002).
call(5000002,_,5001,1001).
read(5000004,_,5002,20002).
call(5000005,_,5002,1002).
```

Figure 6.3: Underlying logic representations of the LAN simulation before refactoring

6.3 encapsulateAttribute Refactoring

The refactoring **encapsulateAttribute** is a composite refactoring that is used to avoid direct access to a specific attribute. It was briefly mentioned in section 5.2.5. It includes the following actions:

1. Add getter and setter methods. This is done by using the primitive refactorings **addGetter** (*section 5.2.5*) and **addSetter** (*section 5.2.6*).

2. Replace accesses to the attribute by calls to the newly created methods. This is done by using the primitive refactorings **attributeReadsToMethodCall** (*section 5.3.2.4*) and **attributeWritesToMethodCall** (*section 5.3.2.5*) primitive refactorings.

3. Make the attribute *private*. This is done by using the primitive refactoring **changeAttributeAccess** (*section 5.3.1.7*).

To encapsulate the attribute *Packet.originator* we call the **encapsulateAttribute** procedure:

encapsulateAttribute(*'Lan','Packet',originator*)

The function will produce a collection of FGTs which represent the transformation actions that are needed to perform the encapsulation of the attribute as shown in the right column of Table 6.1. The collection of primitive refactorings used in the function is shown in the middle column of the Table.

For example, in the primitive refactoring **attributeReadsToMethodCall** that has the following format:

attributeReadsToMethodCall($Dest_x$, $Dest_y$)

any *read* access from anywhere in the system to the destination $Dest_x$ will be redirected to a new destination $Dest_y$. This means that for each *read* access, two FGT operations will be produced, one to delete the original *read* access "*read relation*" from the source *S* to the destination $Dest_x$, this is done by FGT:

deleteRelation(_, *S*, $Dest_x$, *read*)

and the other to add a new *read* access from the source *S* to the new distention $Dest_y$, this is done by FGT:

addRelation(_, *S*, $Dest_y$, *read*).

In the LAN example, there is one *read* access from *Workstation.accept* method to the *Packet.originator* attribute. Accordingly, the primitive refactoring

attributeReadsToMethodCall(*'Lan','Packet',originator,'Lan','Packet', getoriginator,[]*)

will produce two FGTs:

deleteRelation(_,*Lan,Workstation,accept,_,[Packet],method,Lan,Packet,originator,_,_, attribute,read*)

addRelation(_,*Lan,Workstation,accept,_,[Packet],method,Lan,Packet,getoriginator,_,[], method,call*)

Table 6.1: encapsulateAttribute refactoring

Com-posite Ref.	Seq. Of Primitive Refactorings	Seq. Of FGTs For Each Primitive Refactoring
encapsulateAttribute(*'Lan','Packet',originator*)	addGetter(*'Lan','Packet', originator*)	**FGT1:** addObject(*Lan,Packet,getoriginator,_,_,type(complex, Node,0), public,[],method*) **FGT2:** addRelation(*_,Lan,Packet,getoriginator,_,[],method, Lan, Packet,originator,_,_,attribute,read*)
	addSetter(*'Lan','Packet', originator*)	**FGT3:** addObject(*Lan,Packet,setoriginator,_,_,type(basic, void,0), public,[(p, type(basic,Node,0))],method*) **FGT4:** addRelation(*_,Lan,Packet,setoriginator,_,[Node], method, Lan,Packet,originator,_,_,attribute,write*)
	attributeReadsToMethodCall(*'Lan','Packet',originator, 'Lan','Packet',getoriginator, []*)	**FGT5:** deleteRelation(*_,Lan,Workstation,accept,_,[Packet], method, Lan,Packet,originator,_,_,attribute,read*) **FGT6:** addRelation(*_,Lan,Workstation,accept,_,[Packet], method, Lan,Packet,getoriginator,_,[],method,call*)
	attributeWritesToMethodCall(*'Lan','Packet',originator, 'Lan','Packet',setoriginator, ['Node'])*	**FGT7:** deleteRelation(*_,Lan,Workstation,originate,_,[Packet], method,Lan,Packet,originator,_,_,attribute,write*) **FGT8:** addRelation(*_,Lan,Workstation,originate,_,[Packet], method ,Lan,Packet,setoriginator ,_,[Node], method,write*)
	changeAttributeAccess(*'Lan', 'Packet',originator,private*)	**FGT9:** changeOAMode(*Lan,Packet,originator,_,_,attribute, public, private*)

When the tool applies the nine FGTs that are produced/extracted from the composite refactoring **encapsulateAttribute** on the LAN system, the representation of the *Packet* and *Workstation* classes will be affected. Figure 6.4 shows the underlying logic representation of the two classes before and after applying the refactoring. The figure also shows the ID of each FGT alongside each logic-terms that is affected by it.

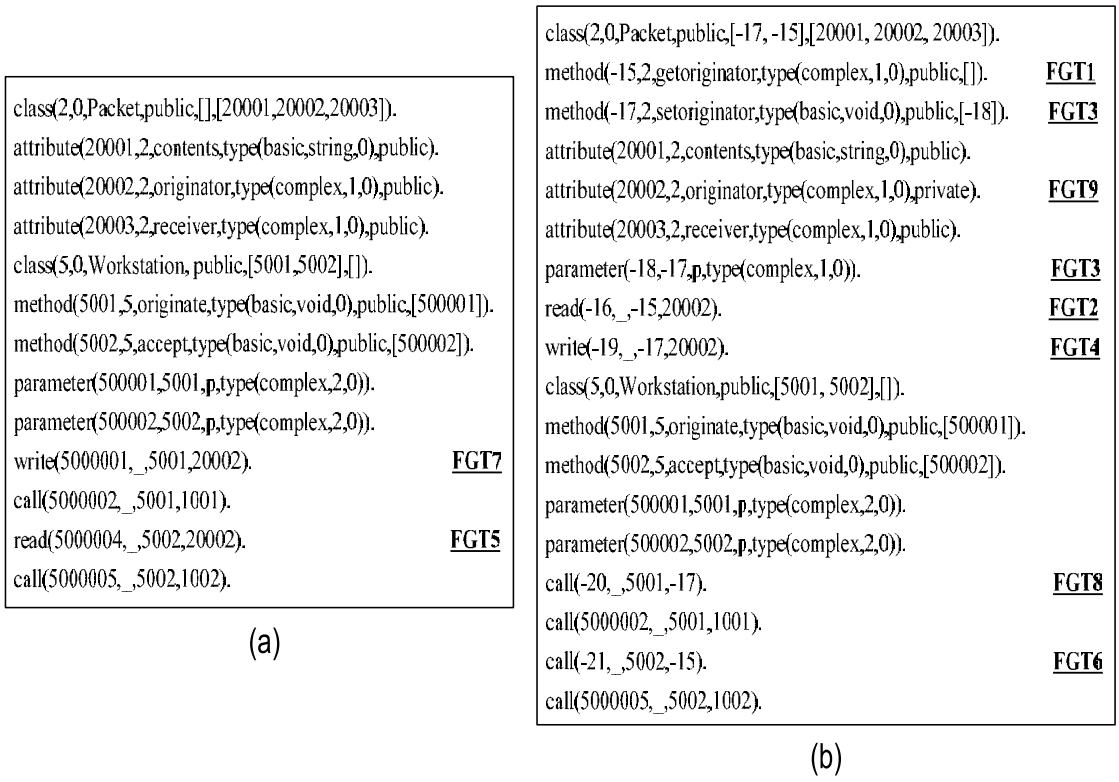


Figure 6.4: Packet & Workstation classes before and after encapsulateAttribute refactoring

6.4 createClass Refactoring

The refactoring **createClass** is a composite refactoring that is used to create a new class. The new class may be a standalone class or a super/sub (or both) of other classes, depending on the parameters that are used in the refactoring. It includes the following actions:

1. Add a new class. This is done by using the primitive refactoring **addClass** (*section 5.2.1*).

2. Change the superclass of the specific class from one class to another. This is done by using the primitive refactoring **changeSuper** (*section 5.3.2.1*).

In the motivated example, to create the class *Server* the composite refactoring procedure **createClass** is invoked:

createClass(*'Lan', 'Server', public,* *'Lan', 'Node',* *['Lan', 'FileServer', 'Lan', 'PrintServer']*)

The New Class *Superclass* *List Of Subclasses*

As indicated in the list of parameters, the refactoring will create a new class, *Lan.Server*, with access mode *public*. The new class will be subclass of the class *Lan.Node* and superclass of the classes *Lan.FileServer* and *Lan.PrintServer*. Note that the subclasses are included in a list which can have as many subclasses as required. If the list is empty this means that the new class will not have any subclasses. The same also for the superclass parameter: if it is null then the new class will not have a superclass. The middle column of Table 6.2 shows the list of primitive refactorings that are used to construct the **createClass** refactoring, while the right column shows the collection of FGTs that are produced for each refactoring.

Table 6.2: createClass refactoring

Com-posite Ref.	Seq. Of Primitive Refactorings	Collection Of FGTs
createClass(*'Lan','Server',public,'Lan','Node', ['Lan','FileServer','Lan','PrintServer'])*	addClass(*'Lan','Server',public*)	**FGT1:** addObject(*Lan,Server,_,_,_,public,_, class*)
	changeSuper(*'Lan','Server','Lan','Node'*)	**FGT2:** addRelation(*isa,Lan,Node,_,_,_,class, Lan, Server, _,_,_,class,extends*)
	changeSuper(*'Lan','FileServer', 'Lan','Server'*)	**FGT3:** deleteRelation(*_,Lan,Node,_,_,_,class, Lan, FileServer, _,_,_,class,extends*) **FGT4:** addRelation(*isa,Lan,Server,_,_,_,class, Lan, FileServer,_,_,_,class,extends*)
	changeSuper(*'Lan','PrintServer', 'Lan','Server'*)	**FGT5:** deleteRelation(*_,Lan,Node,_,_,_,class, Lan, PrintServer,_,_,_,class,extends*) **FGT6:** addRelation(*isa,Lan,Server,_,_,_,class, Lan, PrintServer,_,_,_,class,extends*)

6.5 pullUpMethod Refactoring

The refactoring **pullUpMethod** is a primitive refactorings that is used to pull up a method from a list of subclasses to a superclass. For more details return to section *5.3.2.6*.

In the motivated example, to pull up the method *accept* from *FileServer*, *PrintServer* classes to the class *Server* the procedure

pullUpMethod(*['Lan','FileServer','Lan','PrintServer'],accept,['Packet']* **)**

is called. The parameters show that the subclasses from which to pull up the method are inserted in a list. Thus, as many subclasses as desired can be given. The procedure will produce a collection of FGTs as show in the right column of Table 6.3.

Table 6.3: pullUpMethod refactoring

Prim-itive Ref.	Collection Of FGTs
pullUpMethod(*['Lan','FileServer', 'Lan','PrintServer'],accept,['Packet']*)	**FGT1**: addObject(*Lan,Server,accept,_,_,type(basic,void,0),public,[(p, type(complex, Packet,0))],method*) **FGT2**: deleteObject(*Lan,FileServer,accept,_,[Packet],method*) **FGT3**: deleteObject(*Lan,PrintServer,accept,_,[Packet],method*)

6.6 LAN after Refactorings

Figure 6.5 shows the produced collection of logic-terms for the LAN motivated example after applying the three refactorings **encapsulateAttribute**, **createClass** and **pullUpMethod**. Figure 6.6 shows the resulting UML class diagram based on the refactored version of the logic-terms. Figure 6.7 shows the modified code-level implementation of the UML class diagram after refactoring. Note that, in principle, the process of modifying the code from the refactored UML class diagram can be automated. However, details have not been investigated in this research.

The reader's attention is drawn to the reasons for colour-coding various entries in Figure 6.5. At first sight, it might appear strange that the **pullUpMethod** refactoring retained the *call*- and *read* relationships between the *Node* class on the one hand and the *FileServer* and *PrintServer* classes on the other. One might have expected that these should be relocated in the UML diagram (and logic-based representation thereof) to the new superclass in which the *accept* method has now been physically located.

However, these relationships refer to code-level activity. Despite the **pullUpMethod** refactoring, at the code-level the *call*- and *read* relationships have not been changed. All that has happened is that a *call* to method physically present in a class has become a *call* to an inherited method. This has happened, even though the actual code has not changed.

In terms of the visual representation of the UML class diagram that has been augmented with relationship information, all that needs to change is that the pulled up method should be shown in the superclass, and removed from the subclasses. However, Figure 6.6 shows the *accept* method in subclasses in blue, and provides a special note to indicate that this is for illustrative purposes, and by way of exception.

Notwithstanding these observations, the concrete representations of the *call*- and *read* relationships in the logic database have to be modified. This is because the *accept* method to which they refer is no longer in the respective subclasses, but in the superclass. To this end, at the logical level, a representation of the inherited method is retained for each inheriting subclasses. All relationship information is specified in terms of this representation.

Thus, in the specific example given, the *accept* method is represented in Figure 6.5 by three different entries. The first is a normal method with ID 57 in the class with ID 53. However, the method is also represented as an inherited method in the two subclasses. In these cases, special IDs are used for these two representations, namely 53_90, and 53_91 respectively. The 53 references the class ID in which the inherited method is to be found.

Similarly, the *read*- and *call* information has to be changed to reflect these new IDs. In Figure 6.5, all relations changed that relate to *FileServer* are given in blue, and those relating to *PrintServer* are given in red. The new method is given in green.

The reason for retaining this information is clear: it may be needed for a future refactorings, involving, for example, the **deleteMethod** refactoring. Recall that the precondition conjuncts for such a refactoring require that there should be no reference to the method to be deleted. The information reflected in Figure 6.5 will ensure that such precondition conjuncts may be properly checked.

```
package(0,00,Lan,[53, 1, 2, 3, 4, 5]).
class(1,0,Node,public,[1001, 1002],[10001, 10002]).
method(1001,1,send,type(basic,void,0),protected,[100001]).
method(1002,1,accept,type(basic,void,0),public,[100002]).
attribute(10001,1,Name,type(basic,string,0),public).
attribute(10002,1,NextNode,type(complex,1,0),public).
parameter(100001,1001,p,type(complex,2,0)).
parameter(100002,1002,p,type(complex,2,0)).
call(1000001,_,1002,1001).
call(1000002,_,1001,53_90).
call(1000003,_,1001,53_91).
call(1000004,_,1001,5002).
read(1000008,_,1001,10001).
read(1000009,_,1001,10002).
extends(10000012,isa,1,5).
extends(54,isa,1,53).
class(2,0,Packet,public,[48, 46],[20001,20002,20003]).
method(46,2,getoriginator,type(complex,1,0),public,[]).
method(48,2,setoriginator,type(basic,void,0),public,[49]).
attribute(20001,2,contents,string,1,public).
attribute(20003,2,receiver,type(complex,1,0),public).
attribute(20002,2,originator,type(complex,1,0),private).
parameter(49,48,p,type(complex,1,0)).
read(47,_,46,20002).
write(50,_,48,20002).
```

```
class(3,0,FileServer,public,[53_90],[]).
method(53_90,3,accept,type(basic,void,0),public,[61]).
read(3000001,_,53_90,20001).
call(3000003,_,53_90,1002).
read(3000004,_,53_90,20003).
class(4,0,PrintServer,public,[53_91],[]).
method(53_91,4,accept,type(basic,void,0),public,[61]).
read(4000001,_,53_91,20001).
call(4000003,_,53_91,1002).
read(4000004,_,53_91,20003).
class(53,0,Server,public,[57],[]).
method(57,53,accept,type(basic,void,0),public,[61]).
parameter(61,57,p,type(complex,2,0)).
extends(55,isa,53,3).
extends(56,isa,53,4).
class(5,0,Workstation,public,[5001, 5002],[]).
method(5001,5,originate,type(basic,void,0),public,[500001]).
method(5002,5,accept,type(basic,void,0),public,[500002]).
parameter(500001,5001,p,type(complex,2,0)).
parameter(500002,5002,p,type(complex,2,0)).
call(5000002,_,5001,1001).
call(5000005,_,5002,1002).
call(51,_,5001,48).
call(52,_,5002,46).
```

Figure 6.5: Underlying logic representations of the LAN simulation after refactorings

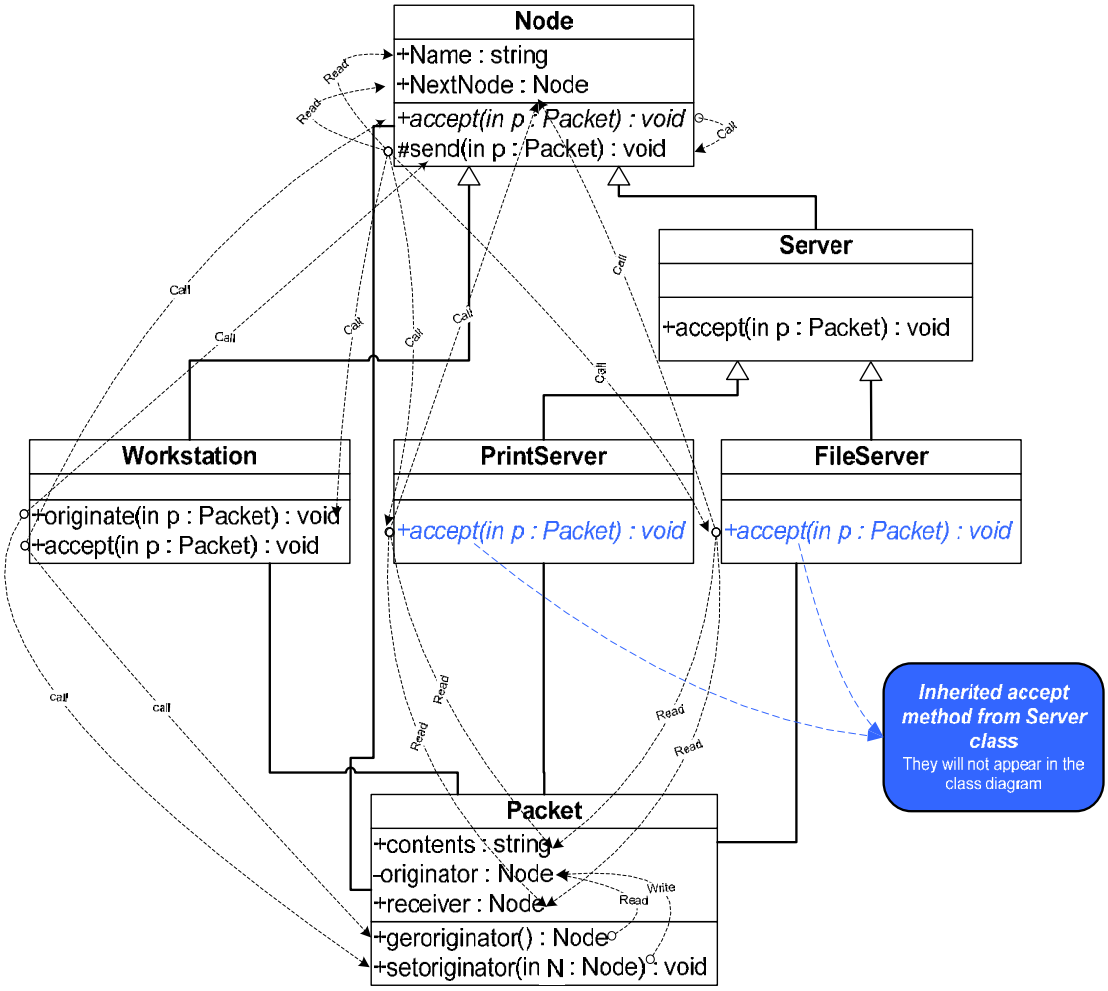


Figure 6.6: A UML class diagram of the LAN simulation after refactoring

```
package Lan;
class Node {
 public String Name;
 public Node NextNode;
 public void accept(Packet p) {
 this.send(p); }
 protected void send(Packet p) {
 System.out.println(name + nextNode.name);
 this.nextNode.accept(p); }}
class Packet {
 public String contents;
 private Node originator;
 public Node addressee;
 public void setOriginator(Node originator){
 this.originator = originator;}
 public Node getOriginator() {
 return originator;}}
```

```
class Server extends Node{
 public void accept(Packet p) {
 if(p.addressee == this)
   System.out.println(p.contents);
 else super.accept(p); }}
class PrintServer extends Server {}
class FileServer extends Server {}
class Workstation extends Node {
 public void originate(Packet p) {
 p.setOriginator(this);
 System.out.println(this.name+
this.nextNode);
 this.send(p); }
 public void accept(Packet p) {
 if(p.getOriginator() == this)
 System.err.println("no destination");
 else super.accept(p); }}
```

Figure 6.7: A code-level implementation of the LAN simulation after refactoring