

# Legend

## Key to LePUS3 and Class-Z Symbols

This document describes the visual vocabulary of LePUS3 and Class-Z that captures the abstract building-blocks in object-oriented design. It gives a brief and informal summary of each term, relation and predicate symbols used in LePUS3 and Class-Z formulas. It does not include formal definitions, which are given in the [LePUS3 and Class-Z Reference Manual](#).

See also:

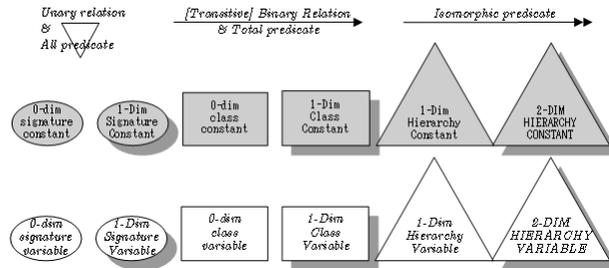
- [About LePUS3 and Class-Z](#)
- [LePUS3 and Class-Z Reference Manual](#) [pdf]

### Table of contents

1. LePUS3 Vocabulary
2. Some basic terminology
3. Modelling specific (sets of-)classes and class hierarchies
4. Modelling specific (sets of-)methods
5. Modelling properties and simple relations
6. Modelling relations between sets
7. Modelling generic motifs

## 1. LePUS3 Vocabulary

The basic set of symbols used in LePUS3 is illustrated below.



## 2. Some basic terminology

**Terms** stand for entities, such as classes (as defined in [Section 2](#)) and methods (as defined in [Section 3](#) below). **Formulas** specify constraints on the entities, where **ground formulas** specify properties and relations between individual entities (see [Section 4](#) below) and **predicate formulas** specify relations between sets of entities (see [Section 5](#) below.)

Each term has a **dimension**: 0-dimensional terms stand for individual entities (either a class, a method, or a signature) whereas 1-dimensional terms stand for sets of entities. Each term also has a **type** (see complete [list of types](#) in the reference manual), for example:

- Terms of type **CLASS** (0-dimensional class terms) stand for individual classes, whereas terms of type **PCLASS** (1-dimensional class terms) stand for sets of classes
- Terms of type **SIGNATURE** (0-dimensional signature terms) stand for individual (method) signatures, whereas terms of type **PSIGNATURE** (1-dimensional signature terms) stand for sets of signatures
- *d*-dimensional [superimposition terms](#) are of type **METHOD**

The type **HIERARCHY** is a subtype of **PCLASS**; in other words, a [hierarchy](#) is also a set of classes.

For the precise definitions of this vocabulary please refer to the [LePUS3 and Class-Z Reference Manual](#).

### 3. Modelling specific (sets of-)classes and class hierarchies

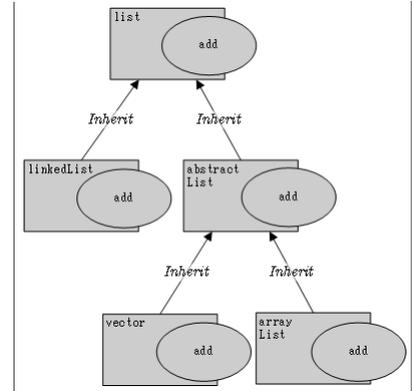
Constants are terms that stand each for a specific class, a (method) signature, or a hierarchy, or a set of these entities.

Symbol name	Meaning in Java	Sample Class-Z schema	Sample LePUS3 chart	Illustration
0-dimensional class constant	A class, an interface, or a primitive type	<p><i>IndividualClass</i> _____</p> <p>linkedList : <b>CLASS</b></p>		<pre>public class LinkedList {...}</pre>
1-dimensional class constant	A set of classes, interfaces, or primitive types	<p><i>ManyClasses</i> _____</p> <p>ConcreteCollections : <b>PC</b>CLASS</p>		
1-dimensional hierarchy constant	A set of classes which contains one class such that all other classes inherit (possibly indirectly) from it	<p><i>A Class Hierarchy</i> _____</p> <p>Collections : <b>HIERARCHY</b></p>		
2-dimensional hierarchy constant	A set of hierarchies	<p><i>A Set Of Hierarchies</i> _____</p> <p>PRODUCTS : <b>PHIERARCHY</b></p>		

# 4. Modelling specific (sets of-)methods

A method or (a set thereof) is represented by superimposing a signature term over a class term, a combination called a **superimposition term**:

Term name	Meaning in Java	Sample Class-Z schema	Sample LePUS3 chart	Illustration
0-dimensional superimposition constant	A method with signature <i>sig</i> which is a member of (or inherited by) class <i>cls</i>	<p><i>AnIndividualMethod</i></p> <hr/> <i>LinkedList</i> : CLASS add : SIGNATURE <hr/> <i>Method</i> (add⊗LinkedList)		<pre>public interface LinkedList {     void add(); }</pre>
1-dimensional superimposition constant	A tribe (a set of methods with signatures <i>Signatures</i> ) that are members of (or inherited by) class <i>cls</i>	<p><i>ATribe</i></p> <hr/> <i>LinkedList</i> : CLASS ListOps : PSIGNATURE <hr/> <i>ALL</i> (Method, ListOps⊗LinkedList)		
	A clan (a set of methods with signature <i>sig</i> ) that are members of (or inherited by) classes in <i>Classes</i>	<p><i>AClan</i></p> <hr/> <i>Lists</i> : PCLASS add : SIGNATURE <hr/> <i>ALL</i> (Method, add⊗Lists)		
	A clan (a set of methods with signature <i>sig</i> ) that are members of (or inherited by) classes in the hierarchy <i>Hrc</i>	<p><i>AClanInAHierarchy</i></p> <hr/> <i>Lists</i> : HIERARCHY add : SIGNATURE <hr/> <i>ALL</i> (Method, add⊗Lists)		



## 5. Modelling properties and simple relations

Properties of individual classes and methods are modelled using **unary relation symbols**, modelled in LePUS3 as inverted triangles. Relations between individual classes and methods are modelled using **binary relation symbols**, A formula consisting of a unary or a binary relation symbol and 0-dimensional arguments is called a **ground formula**.

For the precise meaning of each unary and binary relation symbol in Java see: [Abstract Semantics for Java 1.4 Programs](#)

Relation Symbol	Meaning in Java	Sample Class-Z schema	Sample LePUS3 chart	Illustration
<i>Abstract</i>	The class/method represented by the argument abstract.	<i>AnAbstractClass</i> _____ abstractList : <b>CLASS</b> Abstract(abstractList)		<pre>public abstract class AbstractList ...</pre>
		<i>InterfaceAndAbstractMethod</i> _____ collection : <b>CLASS</b> size : <b>SIGNATURE</b> Abstract(collection) Abstract(size@collection)		<pre>interface Collection { ...   int size();   ... }</pre>
<i>Inherit</i>	One static type extends, implements, or is a subtype-of another.	<i>InheritRelation</i> _____ vector, abstractList, list : <b>CLASS</b> Inherit(vector, abstractList) Inherit(vector, list)		<pre>public class vector   extends AbstractList   implements List   ...</pre>
<i>Inherit*</i>	The class represented by the 'from' term inherits (possibly indirectly) from the class represented by 'to' term.	<i>TransitiveInheritance</i> _____		<pre>public interface Collection ... public interface List   implements Collection ...</pre>

		<pre> linkedList, collection : CLASS ----- Inherit*(linkedList, collection) </pre>	<pre> graph LR   linkedList -- Inherit* --&gt; collection </pre>	<pre> public class LinkedList implements List ... </pre>
Create	The 'from' method may create instances of the 'to' class (or subtypes thereof).	<pre> CreateRelation ----- string : CLASS toLowerCase : SIGNATURE Create(toLowerCase ⊗ string, char[]) </pre>	<pre> graph LR   string -- toLowerCase -- Create --&gt; char["char[]"] </pre>	<pre> public String toLowerCase(Locale locale) { ... if (...) { ... } else { ... for (...) { ... char[] result2 = new char[...]; ... } } </pre>
Member	The 'from' class has a field of the type of the 'to' class (or subtypes thereof).	<pre> MemberRelation ----- package, url : CLASS ----- Member(package, url) </pre>	<pre> graph LR   package -- Member --&gt; url </pre>	<pre> public class package { ... private URL theURL; } </pre>
Create	The 'from' class is (or has a field of) a subtype of Collection, or has an array of (possibly a subtype of) the 'to' class.	<pre> AggregateRelation ----- linkedList, object : CLASS ----- Aggregate(linkedList, object) </pre>	<pre> graph LR   LinkedList -- Aggregate --&gt; object </pre>	<pre> public class LinkedList implements List // which implements Collection ... </pre>
Call	The 'from' method may call the 'to' method.	<pre> CallRelation ----- test, printStream : CLASS main, print : SIGNATURE ----- call(main ⊗ test, print ⊗ printStream) </pre>	<pre> graph LR   test -- main -- Call --&gt; print["print printStream"] </pre>	<pre> public class Test { public static void main (...) { ... if (...) { System.out.print (...); ... } } </pre>
<p>Any other relation symbol designating a decidable relation in code is also permitted and modelled in the same manner, including:</p> <ul style="list-style-type: none"> <li>• <i>Throws</i>, indicating a method contains a return statement with a given type (or subtypes thereof);</li> <li>• <i>Produce</i>, indicating a method returns new instances of a given type (or subtypes thereof);</li> <li>• <i>Throws</i>, indicating a method may throw instances of a given type (or subtypes thereof);</li> </ul>				

## 6. Modelling relations between sets

Relations between sets of entities are represented using **predicates**:

Predicate name	Meaning	Sample Class-Z schema	Sample LePUS3 chart	Illustration
ALL	All the entities represented by the 'to' term are in the relation <i>UnaryRelation</i> .	<pre> AllAbstract ----- AbsCollections : PCLASS ----- ALL(Abtract, AbsCollections) </pre>		

TOTAL	Each one of the entities (except abstract methods) in the set represented by the 'from' term is in the relation <i>BinaryRelation</i> with some entity represented by the 'to' term.	<p><i>TotalInherit</i></p> <hr/> <pre>collection : CLASS ConcreteCollections : PCLASS TOTAL(Inherit, ConcreteCollections, collection)</pre> <hr/>	
ISOMORPHIC	Each one of the non-abstract entities in the set represented by the 'from' term can be paired with a unique non-abstract entity in the set represented by the 'to' term so as to satisfy the ground formula with the relation <i>BinaryRelation</i> .	<p><i>IsomorphicForward</i></p> <hr/> <pre>httpServlet,webAppServlet : CLASS servletOps : PSIGNATURE ISOMORPHIC(Forward, BufferOps@lineNumberReader, BufferOps@bufferedReader)</pre> <hr/>	

Note that in LePUS3 we do not distinguish between the ground formula  $BinaryRelation(t_1, t_2)$  and the predicate formula  $TOTAL(BinaryRelation, t_1, t_2)$ . Since both formulas are satisfied under the same conditions, there is no ambiguity here.

## 7. Modelling generic motifs

Design patterns and generic elements of application frameworks are not tied in to a particular implementation. Their specification therefore requires *variables* rather than *constants*. The difference between constants and variables is as follows:

- **Constants** represent specific entities. Constants are modelled in LePUS3 as filled shapes and in Class-Z using *fixed-width typeface*.
- **Variables** range over entities. Constants are modelled in LePUS3 as empty shapes and in Class-Z using *italicized typeface*.

In other words, variables are used to specify generic design constraints that are not tied in to any specific implementation. For example, specifications of design patterns use only variables, as demonstrated in the "Gang of Four" Companion document.

