

# **jnome: A Java Meta Model in Detail**

*Jan Dockx  
Kristof Mertens  
Nele Smeets  
Eric Steegmans*

*Report CW 323, December 2001, version 0.2*



**Katholieke Universiteit Leuven**  
Department of Computer Science  
Celestijnenlaan 200A – B-3001 Heverlee (Belgium)

# **jnome: A Java Meta Model in Detail**

*Jan Dockx*  
*Kristof Mertens*  
*Nele Smeets*  
*Eric Steegmans*

*Report CW 323, December 2001, version 0.2*

Department of Computer Science, K.U.Leuven

## **Abstract**

jnome is an open source project that offers a meta model for Java implemented in Java. A Java meta model can be the heart of many Java development and study tools.

The meta model is kept clean of pollution by peripheral code in the interest of separation of concerns. Input and output is done through different modules. An ANTLR-based input module that populates the meta model from Java source files is available, as is a proof-of-concept XML output module.

This paper describes a number of possible uses of a Java meta model, and then discusses the jnome Java meta model in detail.

The meta model in its current incarnation lacks support for nested classes, the implementation of methods and initialization code and some minor reserved words.

**Keywords :** object oriented, object orientation, oo, programming, Java, meta model, jnome, javadoc, jml, antlr, parser.

# Contents

<b>1</b>	<b>Introduction</b>	<b>5</b>
1.1	Input . . . . .	6
1.2	Meta Model . . . . .	6
1.3	Output . . . . .	7
1.4	Outline . . . . .	7
<b>2</b>	<b>A Meta Model for Java</b>	<b>8</b>
2.1	Packages . . . . .	8
2.1.1	Short Description . . . . .	8
2.1.1.1	Packages in Java . . . . .	8
2.1.1.2	Package Declarations . . . . .	9
2.1.1.3	Hierarchical Naming Structure . . . . .	9
2.1.1.4	Unique Names . . . . .	9
2.1.2	Meta Model . . . . .	9
2.1.2.1	General . . . . .	10
2.1.2.2	Subpackages . . . . .	10
2.1.2.3	Compilation Units . . . . .	10
2.1.2.4	Void and Primitive Types . . . . .	11
2.1.2.5	Type Invariants . . . . .	11
2.2	Compilation Units . . . . .	11
2.2.1	Short Description . . . . .	11
2.2.2	Meta Model . . . . .	12
2.2.2.1	Package Declaration . . . . .	12
2.2.2.2	Import Statements . . . . .	12
2.2.2.3	Object Types . . . . .	13
2.3	Result Types - Types - Object Types . . . . .	13
2.3.1	Types in Java . . . . .	13
2.3.2	Primitive Types . . . . .	13

2.3.2.1	Short Description . . . . .	13
2.3.2.2	Meta Model . . . . .	14
2.3.3	Object Types . . . . .	14
2.3.3.1	Abstract Object Types . . . . .	15
2.3.3.2	Non Abstract Object Types . . . . .	16
2.3.3.3	Classes and Interfaces . . . . .	16
2.3.3.3.1	Classes . . . . .	17
2.3.3.3.2	Interfaces . . . . .	17
2.3.4	Array Types . . . . .	18
2.3.4.1	Short Description . . . . .	18
2.3.4.2	Meta Model . . . . .	18
2.3.4.2.1	General . . . . .	18
2.3.4.2.2	Type Invariant . . . . .	19
2.4	Variables . . . . .	19
2.4.1	Short Description . . . . .	19
2.4.1.1	Common Properties . . . . .	19
2.4.1.2	Object Type Variables . . . . .	19
2.4.1.3	Method Variables . . . . .	20
2.4.2	Meta Model . . . . .	20
2.4.2.1	General . . . . .	20
2.4.2.2	<b>ObjectTypeVariable</b> . . . . .	21
2.4.2.3	<b>MethodVariable</b> . . . . .	21
2.4.2.4	Type Invariants . . . . .	21
2.5	Methods . . . . .	21
2.5.1	Short Description . . . . .	22
2.5.1.1	Name . . . . .	22
2.5.1.2	Formal Parameters . . . . .	22
2.5.1.3	Throws Clause . . . . .	22
2.5.1.4	Access Modifier . . . . .	22
2.5.1.5	Modifier <b>abstract</b> . . . . .	22
2.5.1.6	Modifier <b>final</b> . . . . .	22
2.5.1.7	Modifiers <b>static</b> , <b>synchronized</b> , <b>native</b> , <b>strictfp</b> . . . . .	23
2.5.1.8	Result Type . . . . .	23
2.5.1.9	Additional Constraints and Properties . . . . .	23
2.5.2	Meta Model . . . . .	23
2.5.2.1	General . . . . .	23

2.5.2.2	Parent, Accessibility, Formal Parameters, Throws Clause . . . . .	25
2.5.2.3	Result Type . . . . .	25
2.5.2.4	Constructors . . . . .	25
2.5.2.5	Type Invariants . . . . .	26
2.5.2.6	Some Remarks . . . . .	26
2.6	Documentation Comments . . . . .	26
2.6.1	Short Description . . . . .	27
2.6.2	Meta Model . . . . .	27
2.7	Unresolved Elements . . . . .	28
2.7.1	General Description . . . . .	28
2.7.2	Packages . . . . .	29
2.7.3	Result Type . . . . .	30
2.7.4	Type . . . . .	30
2.7.5	Object Type . . . . .	31
2.7.6	Class . . . . .	31
2.7.7	Superclass . . . . .	31
2.7.8	Superinterface . . . . .	31
<b>3</b>	<b>Building the Object Structure: Acquire</b>	<b>34</b>
3.1	Short Description of the Second Phase . . . . .	35
3.2	Short Description of the Acquire Phase . . . . .	35
3.3	Description of the Tree . . . . .	36
3.3.1	Java Elements . . . . .	36
3.3.2	Parent-Child Relationships . . . . .	36
3.3.3	Cross References . . . . .	37
3.4	Why Two Separate Phases? . . . . .	37
3.4.1	Building the Meta Model Instance in One Step . . . . .	38
3.4.1.1	Examine the Remaining Files . . . . .	38
3.4.1.2	Examine the Classpath . . . . .	38
3.4.1.3	Conclusion . . . . .	39
3.4.2	Two Separate Phases . . . . .	39
3.5	Implementation of the Acquire Phase . . . . .	39
3.5.1	The Start of the Acquire Process . . . . .	40
3.5.2	Acquiring a Compilation Unit . . . . .	40
3.5.3	Acquiring a Package . . . . .	43
3.5.4	Acquiring an Object Type . . . . .	43

3.5.5	Acquiring a Method . . . . .	46
3.5.6	Acquiring a Variable . . . . .	47
3.5.7	Acquiring a Documentation Block . . . . .	48
3.6	A Small Example . . . . .	48
3.6.1	Description of the Files. . . . .	48
3.6.2	The Acquire Phase. . . . .	49
<b>4</b>	<b>Building the Object Structure: Resolve</b>	<b>51</b>
4.1	Short Description . . . . .	51
4.1.1	A Small Example to Illustrate the Resolve Phase . . . . .	51
4.1.2	Strategy . . . . .	52
4.2	Resolving the Import Statements . . . . .	52
4.3	Resolving the Other Unresolved Elements . . . . .	53
4.3.1	Void? . . . . .	53
4.3.2	Primitive Type? . . . . .	54
4.3.3	Object Type? . . . . .	54
4.3.3.1	Fully Qualified . . . . .	54
4.3.3.2	Not Fully Qualified . . . . .	54
4.4	Resolving Array Types . . . . .	55
4.5	Remarks . . . . .	55
4.5.1	Parsing a Limited Number of Files . . . . .	55
4.5.2	Extensions to the Meta Model . . . . .	56
4.6	A Small Example . . . . .	57
<b>5</b>	<b>The Worker Pattern</b>	<b>59</b>
5.1	Traditional Approach . . . . .	59
5.2	The Visitor Pattern . . . . .	59
5.3	The Worker Pattern . . . . .	60
5.3.1	Short Description . . . . .	60
5.3.2	Advantages . . . . .	61
5.3.3	Difficulty . . . . .	61
5.3.3.1	Description . . . . .	61
5.3.3.2	Solution: Chain of Responsibility . . . . .	62
<b>6</b>	<b>Conclusion</b>	<b>63</b>

# Chapter 1

## Introduction

In this paper, we present a meta model for Java. The meta model is implemented in Java, and made available as an Open Source project (see <http://www.jnome.org/> or <http://www.cs.kuleuven.ac.be/projects/jnome/><sup>1</sup>). We invite all interested parties to contribute to the project.

Our immediate goal with jnome is to produce a tool which generates documentation for Java code containing formal behavioral specification<sup>2</sup>; the tool should be able to generate documentation in different ways, according to the specific needs of different audiences. We believe however that an implemented Java meta model is an important resource for a large number of Java development and study tools.

In the interest of performance, development and study tools often work directly on an abstract syntax tree created from the source code. For advanced operations however, this way of working becomes extremely complex. As a result, development and study tools often lack such advanced operations.

Once a meta model is instantiated for the source code under investigation, most operations become trivial. For example, we easily succeed in generating javadoc-like documentation [javadoc] for Java projects.

The current implementation of the meta model was made with a batch approach in mind: from syntactically correct Java source files (i.e. they can be compiled without errors and comment blocks adhere to the specified syntax), a meta model is instantiated, which can then be inspected to generate the desired output. The effect of source files that contain syntax errors is not specified. Graceful degradation in this circumstance might be an interesting topic for a subproject.

In the interest of separation of concerns, we took great care not to pollute the meta model with non-semantic code. Input and output are in different modules, completely separated from the meta model code. The code is also extensively documented: each class, interface, method and variable is accompanied with a comment block, using the javadoc specification syntax. Both formal and informal specifications are given.

---

<sup>1</sup>These URL's are under construction. Please refer to <http://www.cs.kuleuven.ac.be/cwis/research/som/research/jnome/> in the meantime.

<sup>2</sup>For instance using JML [Leavens].

## 1.1 Input

To populate the meta model, an input module is necessary.

In the current code base a module exists that parses Java source files and builds the corresponding meta model instantiation by visiting<sup>3</sup> the resulting abstract syntax tree. For this input module, ANTLR [ANTLR] is used, with the provided Java syntax specification file (see package `org.jnome.input antlr.java` for the source code).

Other input modules could instantiate a meta model from XML files describing Java classes and interfaces or from a UML model [UML] of a Java project. Possibly, a (partial) meta model could be constructed from Java `.class` files using reflection. We are specifically interested in Java source files that contain formal behavioral specifications, e.g. using the JML syntax [Leavens].

## 1.2 Meta Model

In the development of the meta model, we focused on actual and pragmatic commonalities and differences of Java concepts. Therefore, in some cases, the jnome Java meta model represents concepts slightly different from the way they are explained in the Java Language Specification [Gosling Joy et al. 2000].

Currently, the jnome meta model supports the Java concepts necessary for generating javadoc-like documentation, including packages, compilation units, classes, interfaces, primitive types, array types, variables, methods and documentation comments. However, we lack support for nested classes, the implementation of methods, initialization code and some minor reserved words. An extension of the meta model with these concepts is planned.

A meta model is not only usable in a batch processing context. By making the meta model interactive, and providing a user interface for it, jnome will open the way to many more development and study tools. For example, a refactoring tool would be straightforward to make when it is crafted on top of an interactive Java meta model.

Making the meta model interactive means that the meta model objects should be mutable, and send events when their properties are changed. Such an evolution of the code can probably happen without major changes to the basic structure of the meta model.

Furthermore, the meta model could be extended with concepts that do not appear in the Java Language Specification. For example, we would like to support formal behavioral specifications (e.g. written in JML [Leavens]); to do this, the concepts of the used behavioural interface specification language should be incorporated in the meta model. Another extension of the meta model could be to model design patterns [Gamma Helm et al 1995].

We envisage different subprojects, either under the jnome umbrella (for example `org.jnome.mm.jml`) or separate, that would offer extensions to the basic jnome Java meta model. Such extensions could be accompanied by specific input modules (e.g. `org.jnome.input antlr.jml`) and output modules.

---

<sup>3</sup>In order not to pollute the meta model code with support for the Visitor Pattern [Gamma Helm et al 1995], an alternative pattern is used, which we call the Worker Pattern (see Chap. 5).

## 1.3 Output

Starting from a meta model instance, different kinds of output can be produced. We currently have code that produces XML files containing javadoc-like information, and XSLT files to display them. This output should be considered a proof-of-concept, and not production code.

We envisage adding a JDOM [Hunter McLaughlin] based XML output module. This module could produce different kinds of XML documents, which can then be processed further using XSLT to produce, for example, PDF files. Maybe the XML output presented in [JDK1.4] for persistent storage of JavaBeans might be interesting. Other examples of interesting output are files containing the results of measurements of software metrics, stylized Java source files, or source files containing test code, generated based on the meta model information. We see the creation of different interesting output modules as possible subprojects of jnome or outside projects.

## 1.4 Outline

In the rest of this paper, we will present the jnome Java meta model in detail. Chapter 2 explains how packages, compilation units, primitive types, classes, interfaces, array types, the four different kinds of variables, and the different kinds of methods are modelled in jnome. We will indicate where our approach deviates from the Java Language Specification [Gosling Joy et al. 2000] and we will give arguments for the design choices we made. Chapters 3 and 4 show that populating the meta model is tackled by a two-step-process. In a first pass, cross references are set using placeholders, which are then resolved in a second pass. In Chap. 5, a pattern, named the Worker Pattern, is explained in detail. Finally, in Chap. 6, we give a conclusion for our paper.

## Chapter 2

# A Meta Model for Java

In this chapter, the meta model is discussed in detail. In Sects. 2.1 to 2.5, the following Java elements are described: packages, compilation units, types, variables and methods. In Sect. 2.6, we describe how documentation blocks are modelled. Finally, Sect. 2.7 describes how the meta model incorporates elements to model absent Java elements (Java elements that are mentioned in some files, but that are not parsed themselves).

### 2.1 Packages

In this section, Java packages are covered. First, a short description is given about packages in Java, mainly based on the Java language specification. After that, we discuss how packages are modelled in the meta model of jnome.

#### 2.1.1 Short Description

##### 2.1.1.1 Packages in Java

An object-oriented program is a set of classes and interfaces. For small applications, controlling the names of the involved classes causes no problem: all classes and interfaces can be given a unique name. For bigger applications, a naming conflict will occur sooner or later. This is due to the fact that the reuse of existing classes is extremely important in object-oriented software development. As a result, it is unavoidable that two classes that accidentally have an identical name are needed in the same application. To resolve that kind of naming conflicts, the notion of a **package**<sup>1</sup> is introduced in Java. A package is a set of classes and interfaces. It introduces a name space in which the names of all classes and interfaces should be unique. Classes and interfaces in different packages can still have the same name.

Classes and interfaces are defined in compilation units (see Sect. 2.2). Therefore, a package can also be seen as a set of compilation units that define classes and interfaces themselves. This approach is taken in jnome.

---

<sup>1</sup>The description in Sect. 2.1.1 is based on Sects. 7.1, 7.4, 7.7 and 6.7 of the Java Language Specification ([Gosling Joy et al. 2000]) and on [Steegmans Dockx et al. 1999, p154-159].

### 2.1.1.2 Package Declarations

The package declaration in a compilation unit indicates the package to which the compilation unit belongs. An example of such a package declaration is

```
package java.util;
```

This package declaration indicates that the involved compilation unit is defined in the package with name `java.util`.

When a compilation unit does not contain a package declaration, then it belongs to the unnamed package. Unnamed packages are provided for the development of small or temporary applications.

**Fully Qualified Name** The fully qualified name of a top level package<sup>2</sup> is equal to its simple name. The fully qualified name of a named package that is a subpackage of another named package consists of the fully qualified name of the containing package, followed by `."`, followed by the simple (member) name of the subpackage. The unnamed package does not have a name, nor a fully qualified name.

### 2.1.1.3 Hierarchical Naming Structure

Packages are hierarchically ordered: each package can contain zero or more subpackages and a package cannot be a direct or indirect subpackage of itself. To avoid the possibility of two published packages having the same name, a standard naming convention has been developed (see [Gosling Joy et al. 2000, section 7.7]).

### 2.1.1.4 Unique Names

A package cannot contain two subpackages, classes or interfaces of the same name. This would result in a compile-time error.

## 2.1.2 Meta Model

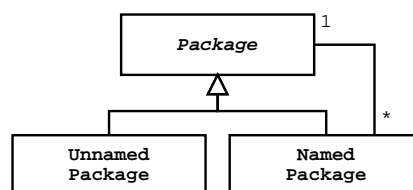


Figure 2.1: The package hierarchy.

---

<sup>2</sup>A top level package is a named package that is not a subpackage of another named package [Gosling Joy et al. 2000, chapter 7].

### 2.1.2.1 General

In the meta model, a Java package is represented by the object type<sup>3</sup> `Package` as can be seen in Fig. 2.1.

An 'ordinary' package, like `java` or `util`, is represented by the object type `NamedPackage`. The **unnamed package** is represented by the object type `UnnamedPackage`. A `NamedPackage` has a name (and a fully qualified name), the unnamed package does not have a name.

**Remark:** The notation used in the figures is based on the UML notation explained in [UML]. We have made some minor changes to that notation, leading to the following:

- The names of abstract object types (i.e. interfaces and abstract classes) are written in *italic*.
- Interfaces get the annotation `<<interface>>`.
- Classes (abstract and non abstract) get the annotation `<<class>>`.
- When we cannot explain yet whether a certain object type is a non abstract class, abstract class or interface, the object type has no annotation or special font. When we know that a certain object type is abstract, but we do not know whether it is a class or interface, the name of the object type is written in italic and no annotation is used. This notation is used in Figs. 2.1 to 2.13. The missing parts of these figures are further explained in Sect. 2.7.

### 2.1.2.2 Subpackages

In `jnome`, the top level packages are placed in the unnamed package. As a result, each named package has exactly one parent package. This parent package can be either a named package or the unnamed package. Furthermore, the unnamed package becomes less different from an ordinary package since both packages can contain subpackages now. The described properties are modelled by a 1:N relationship between `Package` and `NamedPackage`, as can be seen in Fig. 2.1. Finally, by placing the top level packages in the unnamed package, the unnamed package constitutes the root of the package hierarchy and, as a consequence, also the root of all the objects in the meta model instance. This implicates that all Java elements are accessible from one single object, namely the unnamed package of the project at hand.

### 2.1.2.3 Compilation Units

Each package can contain zero or more compilation units, and a compilation unit is defined in exactly one package. This is expressed by the 1:N relationship between `Package` and `CompilationUnit` in Fig. 2.2. The name of the package to which a compilation unit belongs can be found in the package declaration at the top of the compilation unit. A compilation unit that has no package declaration belongs to the unnamed package.

---

<sup>3</sup>In this paper, the term 'object type' is used to denote classes and interfaces.

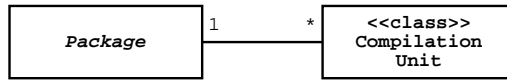


Figure 2.2: A package can contain zero or more compilation units.

#### 2.1.2.4 Void and Primitive Types

The unnamed package contains references to the result type 'void' and to each of the eight primitive types (see Fig. 2.3). In this way, it is possible to enforce that only one object of the class `Void` and one object of the classes `Char`, `Boolean`, `Byte`, `Short`,... are present in the meta model instance. The object of the class `Void`, for example, can then keep references to all methods with 'void' as result type.

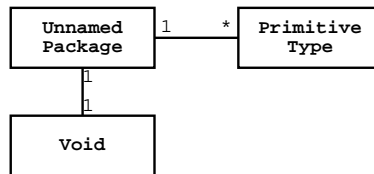


Figure 2.3: The unnamed package contains references to the result type 'void' and to each of the primitive types.

#### 2.1.2.5 Type Invariants

Finally, there are some properties of packages that cannot be expressed in the meta model leading to the following type invariants:

- A package cannot be a direct or indirect subpackage of itself. In other words: the package structure is hierarchical.
- A package cannot contain two classes, interfaces or subpackages with the same name. For example, the package `java.awt` with subpackage `image` cannot contain a class or an interface with name `image`.

## 2.2 Compilation Units

In this section, compilation units are discussed. The section starts with a short description about compilation units in Java, mainly based on the Java language specification. After that, we discuss how compilation units are modelled in the meta model of jnome.

### 2.2.1 Short Description

A compilation unit<sup>4</sup> consists of three parts. Each of them is optional:

- a **package declaration** giving the fully qualified name of the package to which the compilation unit belongs. A compilation unit that has no package declaration is part of the unnamed package.

<sup>4</sup>The description in Sect. 2.2.1 is based on Sects. 7.3, 7.5, 7.6 of the Java Language Specification ([Gosling Joy et al. 2000]).

- zero or more **import declarations**, which allow types from other packages to be referred to using their simple names
- top level **type declarations** of class and interface types

## 2.2.2 Meta Model

As can be seen in Fig. 2.4, the three parts of a compilation unit – package declaration, import declarations and type declarations – are expressed in the meta model.

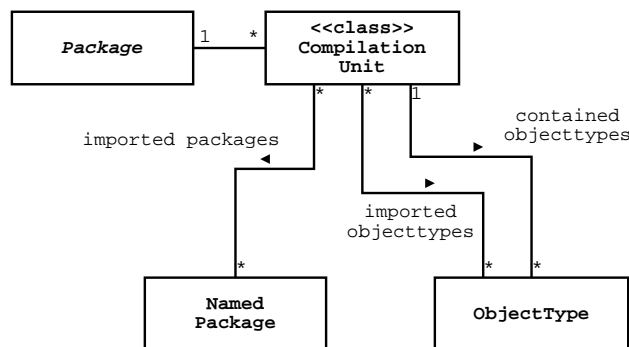


Figure 2.4: Each compilation unit is defined in exactly one package. A compilation unit can import zero or more object types and packages. In a compilation unit, zero or more object types can be defined.

### 2.2.2.1 Package Declaration

A compilation unit belongs to exactly one package. This package can be an ordinary named package, or the unnamed package (when there is no package declaration in the compilation unit). This is modelled by a 1:N relationship between `CompilationUnit` and `Package`.

### 2.2.2.2 Import Statements

There are two kinds of import statements:

- A **type-import-on-demand** imports all public object types declared in a named package. For example, the import statement `import java.util.*;` imports all public classes and interfaces contained in the package `java.util`.
- A **single-type-import** declaration imports a single class or interface giving its fully qualified name. For example, the import statement `import java.util.Vector;` imports the class `java.util.Vector`.

Each compilation unit can import zero or more packages. Each named package can be imported in zero or more compilation units. This is represented by the N:M relationship between `NamedPackage` and `CompilationUnit`.

In an analogous way, the N:M relationship between `ObjectType` and `CompilationUnit` can be understood.

### 2.2.2.3 Object Types

In a compilation unit, classes and interfaces can be defined. A class or interface is defined in exactly one compilation unit. This is expressed by a 1:N relationship between `CompilationUnit` and `ObjectType`.

## 2.3 Result Types - Types - Object Types

In this section, Java types are discussed. In a first section, the types of the Java programming language are discussed. Each of those types and the way they are modelled in jnome is then discussed in the subsequent sections.

### 2.3.1 Types in Java

There are two kinds of types in the Java programming language: primitive types and reference types. The primitive types are subdivided in the numeric types and the primitive type `boolean` (see Sect. 2.3.2). The reference types consist of classes, interfaces (see Sect. 2.3.3) and array types (see Sect. 2.3.4). Beside the primitive types and reference types, there is also the result type `void`. `Void` is used when a method does not return a value.

In jnome, the types are subdivided in a somewhat other manner: the types are divided into three parts, namely primitive types, object types and array types (where an object type is a class or an interface). This subdivision has been chosen because the usage and the possibilities of an object type differ very much from array types (as described in Sect. 2.3.4). An overall picture of the types in jnome can be seen in Fig. 2.5.

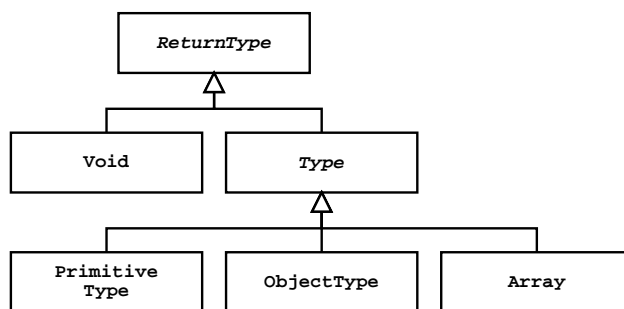


Figure 2.5: The return type hierarchy.

In the next section, the primitive types are discussed. The subsequent sections describe object types and array types.

### 2.3.2 Primitive Types

#### 2.3.2.1 Short Description

The primitive types<sup>5</sup> in the Java programming language are predefined and named by reserved keywords. There are eight primitive types, which are subdivided in the following manner:

---

<sup>5</sup>The description of primitive types in Sect. 2.3.2.1 is based on Sects. 4.2 and 6.7 of the Java Language Specification ([Gosling Joy et al. 2000]).

```

PrimitiveType :
  NumericType
  boolean
NumericType :
  IntegralType
  FloatingPointType
IntegralType: one of
  byte short int long char
FloatingPointType: one of
  float double

```

**Fully Qualified Name** The fully qualified name of a primitive type is the keyword for that primitive type.

### 2.3.2.2 Meta Model

The meta model for primitive types can be seen in Fig. 2.6.

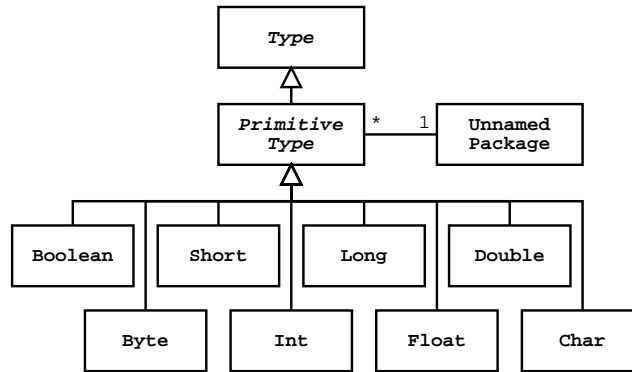


Figure 2.6: The primitive types.

The class `PrimitiveType` is a subclass of `Type`. For each of the primitive types, a separate class is provided. Each of these classes is a subclass of `PrimitiveType`.

The unnamed package keeps a reference to each of the primitive types. In this manner, it is possible to enforce that there is only one object in the meta model instance for each primitive type. Since there is only one object of the class `Int`, this object can keep references to all methods and variables in the current project that have `int` as their (result) type.

**Remark:** The classes in Fig. 2.6 should not be confused with the wrapper classes `java.lang.Boolean`, `java.lang.Byte`, etc.

### 2.3.3 Object Types

As shown in Fig. 2.7, object types (classes and interfaces) are defined in a compilation unit. They can contain methods and variables, have an accessibility modifier, can have superobjecttypes and can be accompanied with a documentation block.

An important property of object types is whether they can be instantiated or not, in other words whether objects of that object type can be created or not. This property divides the object types in two disjunct parts: the abstract and non abstract object types. A UML representation is given in Fig. 2.8.

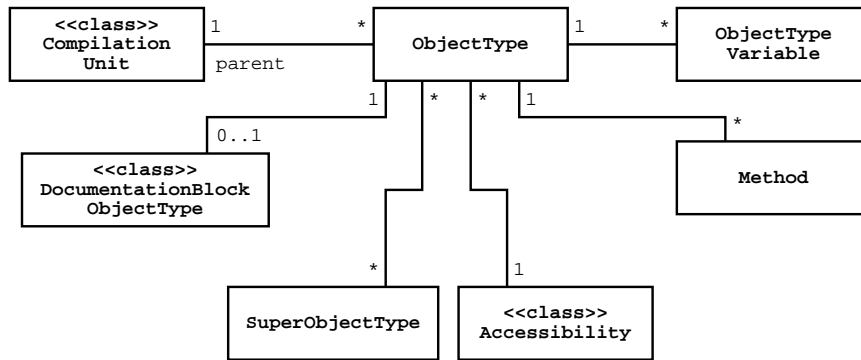


Figure 2.7: An object type is defined in a compilation unit, it contains methods and variables, has an accessibility modifier, a documentation block (optional) and zero or more superobjecttypes.

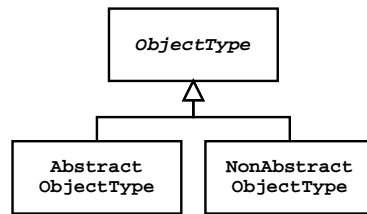


Figure 2.8: Object types are subdivided into abstract and non abstract object types.

### 2.3.3.1 Abstract Object Types

An abstract object type is an object type that cannot be instantiated. Such an object type is therefore only used to make reuse of code or specification, dynamic binding and polymorphism possible.

In Java, abstract object types are further subdivided into interfaces and abstract classes. An **interface** is an abstract object type where none of its methods has an implementation and that only contains static final variables. Interfaces are in some manners treated in a special way in the Java programming language (see below for a more detailed description of classes and interfaces).

An abstract object type can also be an **abstract class**. An abstract class can contain both abstract and non abstract methods. It is even possible that all methods have an implementation. An abstract class can contain both static variables and instance variables.

Figure 2.9 shows how interfaces and abstract classes fit into the structure of the object types.

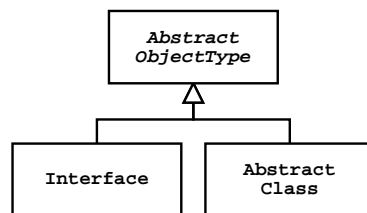


Figure 2.9: The abstract object types.

### 2.3.3.2 Non Abstract Object Types

A non abstract object type is an object type that can be instantiated. In Java, only classes can be instantiated. As a consequence, the terms 'non abstract object type' and 'non abstract class' are equivalent.

Non abstract object types can themselves be subdivided into two categories, namely the non abstract object types that can have subclasses and those who cannot. Classes that cannot have subclasses are marked with the modifier final and are called **final classes**. They cannot be the superclass of another class. Classes that are neither abstract, nor final are called **regular classes** in the meta model. This can be seen in Fig. 2.10.

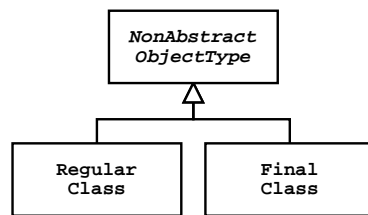


Figure 2.10: The non abstract object types.

A similar hierarchy is not meaningful for abstract object types: abstract types are always non final.

Whether an object type can be a superobjecttype or not is explicitly modelled by the type `SuperObjectType`: the object types `RegularClass` and `AbstractObjectType` are superobjecttypes. This is shown graphically in Fig. 2.11.

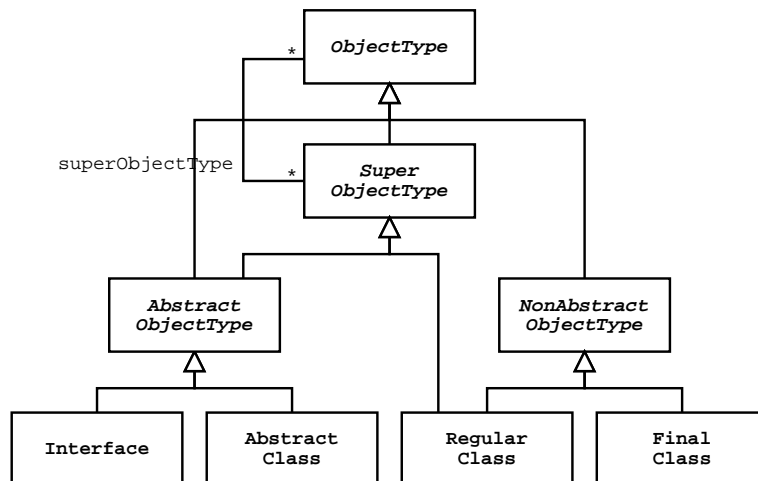


Figure 2.11: Abstract object types and regular classes can have subobjecttypes. This is modelled by letting them inherit from `SuperObjectType`.

### 2.3.3.3 Classes and Interfaces

In the Java programming language, an important difference is made between two kinds of object types, namely classes and interfaces. In our meta model, this difference takes a less prominent place. As can be read above, an interface is just

an abstract object type that can only contain abstract methods and static final variables.

The class `Interface` has already been discussed in earlier sections. To explicitly model the concept 'class' (as specified in the Java Language Specification) as well, two extra types have been added, namely `Clazz`<sup>6</sup> and `SuperClass`. How these object types fit into the meta model is shown in Fig. 2.12.

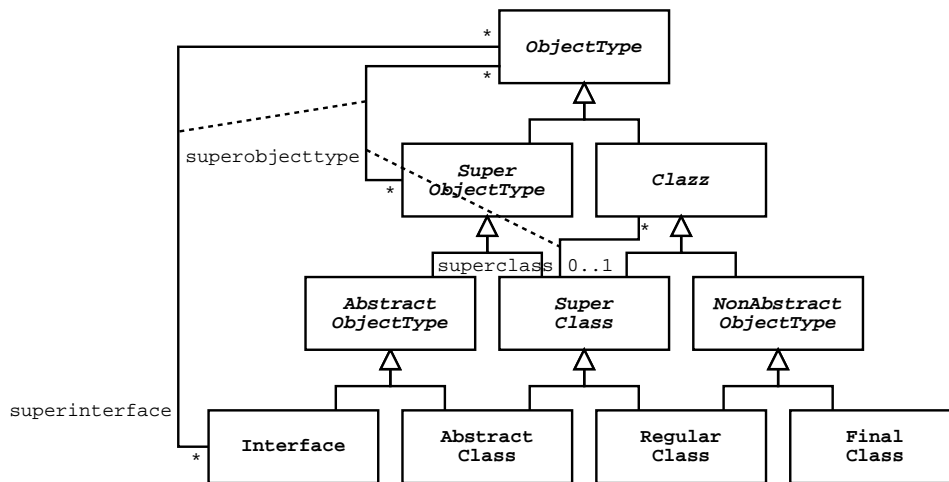


Figure 2.12: This figure extends the meta model from Fig. 2.11 with the object types `Clazz` and `SuperClass`.

### 2.3.3.3.1 Classes

Classes have the following specific properties:

- All classes have exactly one superclass, except for `java.lang.Object`, which constitutes the top of the class hierarchy. We did not find it appropriate to introduce a separate meta model object type for `java.lang.Object`. Such an extra level would make the hierarchy presented in Fig. 2.12 even more complex.
- Each class has one or more constructors.
- Classes can have instance variables.
- A non abstract class can only contain methods with an implementation, in other words a non abstract class can only contain non abstract methods.
- A final class can only contain final methods.

### 2.3.3.3.2 Interfaces

Interfaces have the following specific properties:

- Interfaces are always public accessible.
- Interfaces can only contain abstract methods, which should always be public.
- Interfaces can only have interfaces as superobjecttype, no classes.
- All variables in an interface are implicitly public, static<sup>7</sup> and final.

<sup>6</sup>Clazz has been written with 'z' instead of 's' to avoid confusion with the class `Class` from the `java.lang` package.

<sup>7</sup>The fact that all variables of an interface are static is no type invariant, but has been explicitly modelled.

## 2.3.4 Array Types

### 2.3.4.1 Short Description

Arrays<sup>8</sup> are objects in the Java programming language. An array object contains zero or more components. All components of an array object have the same type, the *component type* of the array.

The component type of an array can again be an array type. If, starting from any array type, one considers its component type, and then (if that is also an array type) the component type of that type, and so on, eventually one must reach a component type that is not an array type; this is called the *element type* of the original array.

For example,

```
int [] [] []
```

is an array type with component type

```
int [] []
```

and element type

```
int.
```

**Fully Qualified Name** The fully qualified name of an array type consists of the fully qualified name of the component type of the array type followed by "[]".

### 2.3.4.2 Meta Model

**2.3.4.2.1 General** The meta model for arrays can be seen in Fig. 2.13.

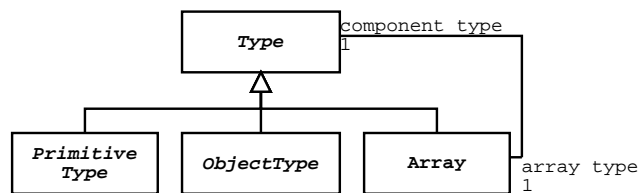


Figure 2.13: Array types.

Array types are different from object types in several ways. Array types do not have a superclass and they cannot be inherited from. As can be seen in Fig. 2.13, the class `Array` is a subclass of `Type`. Each array type has a component type. This component type can be a primitive type, an object type or again an array type. Each type corresponds to exactly one array type; these array type objects are created on demand by the factory method `getArrayType()`.

<sup>8</sup>The description in Sect. 2.3.4.1 is based on Chap. 10 and Sect. 6.7 of the Java Language Specification ([Gosling Joy et al. 2000]).

**2.3.4.2.2 Type Invariant** The following type invariant is true for array types: an array type cannot be a direct or indirect component type of itself.

## 2.4 Variables

In this section, variables are discussed. First, a short description is given about variables in the Java programming language. After that, we discuss how variables are modelled in the meta model of jnome.

### 2.4.1 Short Description

The set of variables<sup>9</sup> is partitioned into two main subsets:

- Object type variables:  
This subset consists of the variables that are declared in a class or interface; in other words, it is the set of the **static variables** and **instance variables** (also called fields in the Java Language Specification).
- Method variables:  
This subset consists of the variables that are used and declared in a method; in other words, it is the set of the **formal parameters** and **local variables**.

In the following subsections, we first discuss the properties that are shared by both kinds of variables. After that, there are two sections discussing the specific properties of each type of variable in more detail.

#### 2.4.1.1 Common Properties

All variables have a name and a type. The type of a variable can be a primitive type, an object type or an array type.

Each variable can be declared final. A final variable can be assigned to only once. For example, when a formal parameter is final, it is a compile-time error if it is assigned to within the body of the method or constructor.

#### 2.4.1.2 Object Type Variables

In this section, a number of properties are discussed that are typical for variables that are declared in a class or interface.

All variables in a class or interface have a different name.

Each object type variable has an access modifier that controls the accessibility of that variable. Variables can be public, protected, private or package accessible.

Another property of a variable is whether it is static or not. This is indicated by the modifier **static**. If a field is declared **static**, there exists exactly one incarnation

---

<sup>9</sup>The description in Sect. 2.4.1 is based on the Java Language Specification ([Gosling Joy et al. 2000]). Variables in classes are described in Sect. 8.3, variables in interfaces in Sect. 9.3, formal parameters in Sect. 8.4.1, local variables in Sect. 14.4 and the meaning of the modifier final is described in Sect. 4.5.4.

of the field, no matter how many instances (possibly zero) of the class may eventually be created. **Non-static variables** (or instance variables) introduce a totally different situation: whenever a new instance of a class is created, a new variable associated with that class instance is created for every instance variable declared in that class or any of its superclasses.

Finally, the modifiers `transient` and `volatile` are possible with a variable. For more information about these modifiers, we refer to [Gosling Joy et al. 2000, sections 8.3.1.3 and 8.3.1.4].

### 2.4.1.3 Method Variables

The class `MethodVariable` describes both formal parameters and local variables.

All **formal parameters** of the same method or constructor should have a different name. These parameter names may not be redeclared as local variables of the method either, nor as exception parameters of catch clauses in a try statement of the method or constructor.

Currently, the body of methods is not modelled yet. Consequently, **local variables** are not completely modelled either.

## 2.4.2 Meta Model

The part of the meta model describing variables can be seen in Figs. 2.14 and 2.15.

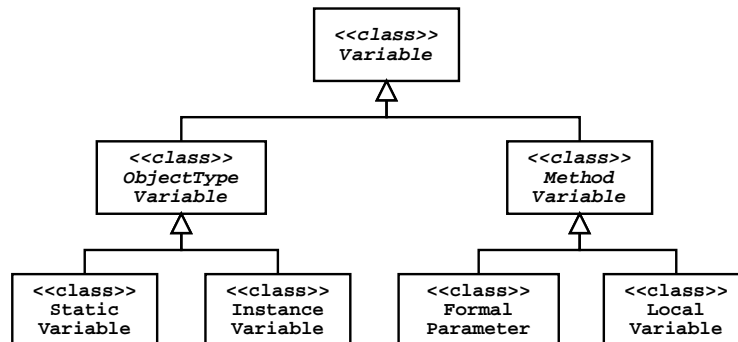


Figure 2.14: The variable hierarchy.

### 2.4.2.1 General

Figure 2.14 shows that variables are represented by the class `Variable`. This class has two subclasses, namely `ObjectTypeVariable` and `MethodVariable`. The difference between these classes has been explained in Sects. 2.4.1.2 and 2.4.1.3.

The fact that each variable has a type is expressed by a 1:N relationship between `Type` and `Variable`, as can be seen in Fig. 2.15. The other relationships in this figure are discussed in the following sections.

In the meta model, the boolean value `isFinal` expresses whether a variable is final or not.

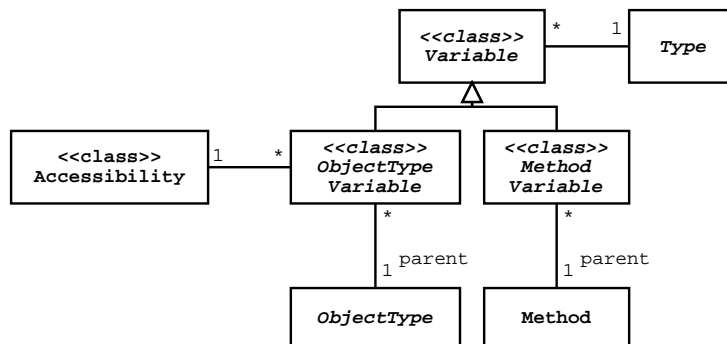


Figure 2.15: Each variable has a type. An **object type variable** has an accessibility modifier and its parent is an object type. The parent of a **method variable** is a method.

#### 2.4.2.2 ObjectTypeVariable

The accessibility of an object type variable is modelled via a 1:N relationship between **Accessibility** and **Variable**. The class **Accessibility** can have four possible values, namely public, protected, private and package accessible. These values are represented by static final variables in the class **Accessibility**.

An object type variable is declared in a class or interface. A class or interface can contain zero or more variables. These properties are expressed by a 1:N relationship between **ObjectType** and **ObjectTypeVariable**.

In the meta model, the presence or absence of the modifiers **transient** and **volatile** is expressed by the boolean values **isTransient** and **isVolatile**.

#### 2.4.2.3 MethodVariable

A method variable is declared in one method. A method can contain several formal parameters and local variables. These properties are expressed by a 1:N relationship between **Method** and **MethodVariable**.

#### 2.4.2.4 Type Invariants

Finally, there are some properties related to variables that cannot be expressed explicitly in the meta model, and that have led to type invariants:

- All variables in an object type have different names.
- All formal parameters and local variables of the same method or constructor should have a different name.
- A final (object type) variable can never be volatile.

## 2.5 Methods

In this section, methods are covered. First, a short description is given about methods in the Java programming language. After that, we discuss how methods are modelled in the meta model of jnome. This part of the meta model is not stable. Many improvements are still possible; some of them are mentioned in Sect. 2.5.2.6.

## 2.5.1 Short Description

Methods<sup>10</sup> in Java have a name, zero or more formal parameters, a throws clause with zero or more exceptions and a body containing the implementation of the method. The body of methods is not included in the meta model of jnome.

Beside the above properties, each method also has an access modifier controlling the access to the method. A method can also have zero or more modifiers from the following list: **abstract**, **static**, **final**, **synchronized**, **native**, **strictfp**. Finally, all methods except for constructors have a result type.

Each of these elements is discussed briefly in the following sections.

### 2.5.1.1 Name

Different methods in the same class or interface can have the same name. They can also have the same name as a variable in that class or interface. A class or interface may not declare two methods with the same signature. The signature of a method consists of the name of the method and the number and types of formal parameters of the method.

### 2.5.1.2 Formal Parameters

A method can have zero or more formal parameters. All formal parameters of a method have a different name.

### 2.5.1.3 Throws Clause

Each object type in the throws clause of a method should be a subclass of **Throwable** (or `java.lang.Throwable` itself).

### 2.5.1.4 Access Modifier

Each method has an access modifier that controls the access to the method. Four different modifiers are possible, namely **public**, **protected**, **private** and package accessible (a method is package accessible when its declaration is not preceded by an access modifier).

### 2.5.1.5 Modifier **abstract**

An abstract method is a method without a body, i.e. without implementation. The declaration of an abstract method can only appear in an abstract class or interface (see also Sect. 2.3.3.1).

### 2.5.1.6 Modifier **final**

When a method is declared **final**, it cannot be overridden in a subclass of the defining class.

---

<sup>10</sup>The description in Sect. 2.5.1 is based on Sect. 8.4 of the Java Language Specification ([Gosling Joy et al. 2000]).

### 2.5.1.7 Modifiers `static`, `synchronized`, `native`, `strictfp`

The correct meaning of the modifiers `static`, `synchronized`, `native` and `strictfp` can be found in [Gosling Joy et al. 2000, sections 8.4.3.2, 8.4.3.4, 8.4.3.5 and 8.4.3.6]. In the meta model, the presence or absence of these three modifiers is expressed by the boolean values `isStatic`, `isSynchronized` and `isStrictfp`.

### 2.5.1.8 Result Type

The result type of a method can be 'void', a primitive type, an array type or an object type.

### 2.5.1.9 Additional Constraints and Properties

Finally, the following additional constraints and properties are true for methods:

- An abstract method can never be private, static, final, native, `strictfp` or `synchronized`.
- A native method can never be `strictfp`.
- Each private method is implicitly final.
- A constructor has the same name as the class in which it is defined; for a default constructor, the accessibility modifier of the constructor and its defining class also agree.

## 2.5.2 Meta Model

The meta model for methods can be seen in Figs. 2.16 and 2.17.

### 2.5.2.1 General

As can be seen in Fig. 2.16, methods are modelled by the interface `Method`. The interface `Method` is implemented by the abstract class `MethodImpl` that contains the implementation for the majority of the methods in the interface. An example of a method that is not implemented in `MethodImpl` is `getName()`. This is due to the fact that the implementation of this method is different for constructors and non-constructors: the name of a constructor is equal to the name of its parent class, while the name of a non-constructor is independent of the name of its parent class.

there is an important semantical difference between constructors and non-constructors, which will be discussed in Sect. 2.5.2.4. Because of this important difference, the class of methods has two subclasses: `MethodWithReturnType` and `Constructor`.

Non abstract object types can only contain non abstract methods. An interface can only contain abstract methods. This suggests that the difference between abstract and non abstract methods should be modelled explicitly in the meta model. Therefore, `MethodWithReturnType` has two subclasses, namely `AbstractMethod` and `NonAbstractMethod`.

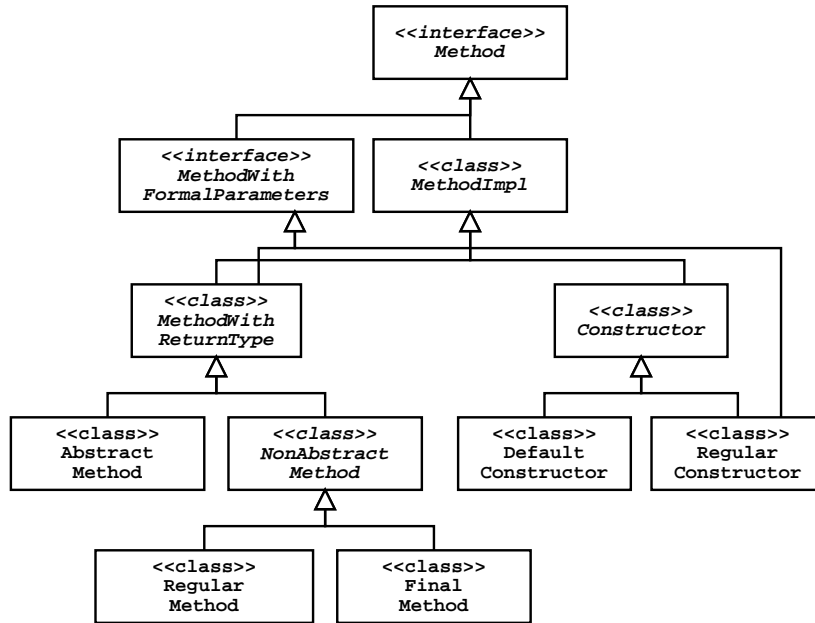


Figure 2.16: The method hierarchy.

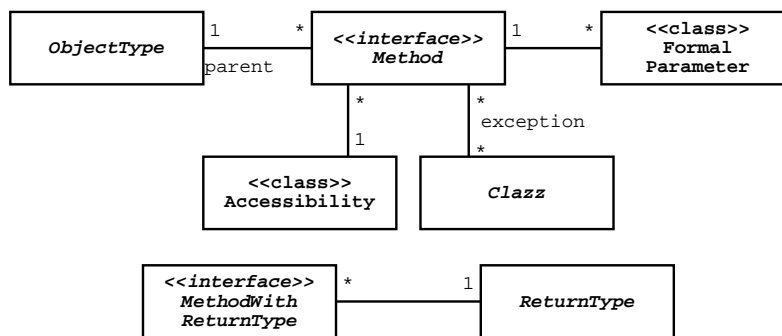


Figure 2.17: Each method has an accessibility modifier, formal parameters and a throws clause. Further, each method has exactly one object type as parent. Each class or interface can contain zero or more methods. Finally, each non-constructor has a return type.

`NonAbstractMethod` itself has two subclasses expressing the difference between final and non final methods: `RegularMethod` and `FinalMethod`. This partitioning allows, among other things, to express explicitly that all methods in an final class are implicitly final.

### 2.5.2.2 Parent, Accessibility, Formal Parameters, Throws Clause

As can be seen in Fig. 2.17, each method

- is declared in exactly one object type,
- has an access modifier,
- has zero or more formal parameters,
- has a throws clause containing zero or more exceptions.

### 2.5.2.3 Result Type

Each non-constructor has a result type. To be able to express this in the meta model, the interface `MethodWithReturnType` is introduced. This interface contains all methods necessary for setting and accessing the result type of a method. The result type of a method can be an object type, a primitive type, an array type or the result type 'void', as shown in Figs. 2.17 and 2.5.

### 2.5.2.4 Constructors

A constructor does not have a return type. Therefore, the classes `Constructor` and `MethodWithReturnType` are disjunct and stand next to each other, as can be seen in Fig. 2.16.

Furthermore, the values of the modifiers `final`, `abstract`, `static`, `native`, `strictfp` and `synchronized` are fixed for a constructor. We will explain this in the remainder of this paragraph. Since a constructor is never inherited, it is implicitly *final*. Another consequence of this fact is that a constructor is always non-*abstract*: an abstract constructor could never be implemented. A constructor is always non-*static*: it can refer to 'this' in its method body and it can call non-static methods. The lack of *native* constructors is an arbitrary language design choice in Java. A constructor cannot be declared *strictfp*. This is also an intentional language design choice; it effectively ensures that a constructor is *strictfp* if and only if its class is *strictfp*. Finally, there is no practical need for a constructor to be *synchronized* (see [Gosling Joy et al. 2000, section 8.8.3]).

Constructors are always declared in a class; they cannot appear in an interface. More precisely, each class contains at least one constructor. When a class does not define a constructor explicitly, a default constructor is added automatically. To model this, the class `DefaultConstructor` is introduced as a subclass of `Constructor`. A default constructor has some special properties:

- A default constructor has the same access modifier as the class in which it is defined.
- A default constructor has no formal parameters.

- A default constructor can only be contained in a class that does not define a constructor explicitly.

An 'ordinary constructor' is represented by the class `RegularConstructor`.

#### 2.5.2.5 Type Invariants

Some properties related to methods are expressed by means of type invariants:

- An object type cannot contain two methods with the same signature.
- The classes mentioned in the throws clause of a method should be a subclass of `Throwable` or the class `Throwable` itself.
- An abstract method can never be private, static, final, native, strictfp or synchronized.
- A native method can never be strictfp.
- Each private method is implicitly final.
- A constructor is always final, non static, non synchronized, non native and non strictfp.
- A default constructor has the same accessibility modifier as the class in which it is defined and has no formal parameters.

#### 2.5.2.6 Some Remarks

In this last section, a few remaining remarks are made concerning the modelling of methods.

- Currently, the body of a method is not considered yet in the meta model. This means that programming language constructions like expressions, while-loops, method-calls, ... are not studied.
- Probably, whether a method is static or not should be modelled more explicitly in the meta model. A static method cannot be overridden in a subclass, while non-static methods can. This is important in the context of method calls or when we want to determine which methods are inherited or overridden in a certain class. A further study of the model is certainly necessary here.
- It is possible that native methods should be represented by a separate class in the meta model too, because native methods do not have a method body with Java code.

## 2.6 Documentation Comments

In this section, comment blocks are discussed. First, we discuss the format of the comment blocks that are considered. After that, the meta model for those comment blocks is described.

## 2.6.1 Short Description

The comment blocks that are considered in jnome are based on the documentation blocks that are used in the course 'Informatie- en programmastructuren' of the KULeuven (see [Steegmans Dockx et al. 1999]). The format of these documentation blocks resembles the format used in the javadoc tool [javadoc] developed by Sun.

An example of such a comment block (documenting a constructor of the class `Account`) is the following:

```
/**
 * Create a new account with the given initial
 * balance.
 *
 * @param initial
 *     The initial balance of the new account.
 * @pre   The initial balance must be greater than
 *       or equal to 0.
 *       | initial >= 0
 * @post  The balance of the new account is set to
 *       the given balance.
 *       | new.getBalance() = initial
 */
```

A comment block consists of two main parts:

- an optional short description at the beginning of the comment block
- zero or more tag blocks

A tag block has three parts:

- a tag
- an informal description (optional)
- a formal description (optional)

The short description at the beginning of a comment block and the informal descriptions are ordinary English sentences, while the formal descriptions are written in a more formal language (e.g. JML [Leavens]).

## 2.6.2 Meta Model

Figures 2.18 and 2.19 show the classes used to model documentation blocks.

A documentation block can accompany an object type, a variable or a method. In each of these three cases the documentation block is optional. This is represented in Fig. 2.18.

The different parts of a documentation block are modelled in Fig. 2.19. This figure is straightforward, so no further explanation is given.

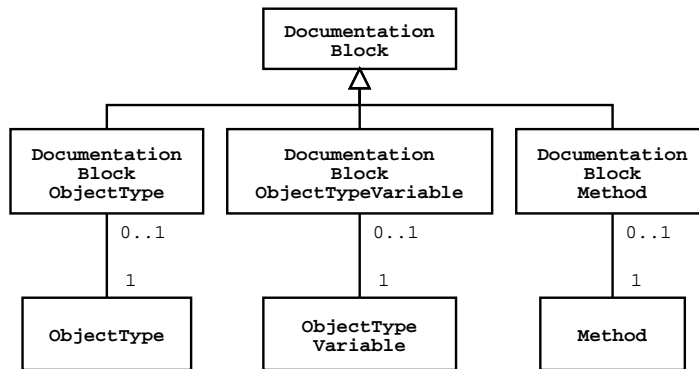


Figure 2.18: Documentation blocks can accompany an object type, a variable or a method.

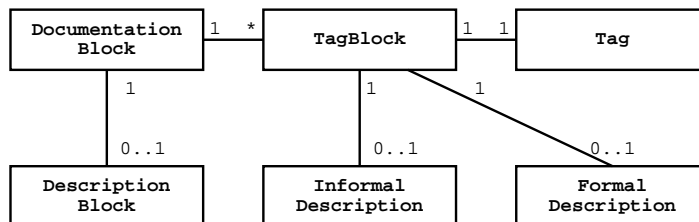


Figure 2.19: The structure of a documentation block.

The part of the meta model describing the documentation blocks is still rather superficial. It could be extended in many different ways. For example, the set of **tags** that are allowed in a comment block could be described. Probably, a difference should be made between the different kinds of documentation blocks: preconditions are necessary and important in the documentation block of a method, but they are meaningless in the documentation block of a class. The meta model could also be extended by developing a complete meta model describing the formal specifications in the documentation blocks. To realize this, a language for writing these formal specifications should be worked out first or existing languages, such as JML [Leavens] could be used.

## 2.7 Unresolved Elements

The meta model discussed in the preceding sections is theoretical and is not sufficient in practice. In theory, each class (except `java.lang.Object`) has a superclass. In practice, we read in a finite number of files sequentially, and represent them in memory using the meta model. In this case, it is possible that a certain class is handled, while its superclass is not, so it is impossible to have a reference from that class to its superclass. The problems concerning absent Java elements are modelled explicitly in the meta model.

### 2.7.1 General Description

As explained above, the superclass of a class cannot always be found. The same is true for some other elements in the meta model. An enumeration of these elements can be found here:

- an imported package
- an imported object type
- a superinterface
- a superclass
- a class in the throws clause of a method
- the result type of a method
- the type of a variable

The enumeration given above is exhaustive for the current meta model, but will have to be extended when, for instance, method calls are taken into account.

For each of the above elements, a corresponding resolved and unresolved element is introduced in the meta model. These new object types are modelled as subtypes of the more general element. For example, `ObjectType` has two subtypes, namely `ResolvedObjectType` and `UnresolvedObjectType`.

When elements are guaranteed to be present in the object structure, the resolved type is used. When this presence is not guaranteed, the more general type is used, which means that both the resolved and the unresolved type are possible there. For example, the object types of a compilation unit are always effective. As a result, a 1:N relationship exists between `CompilationUnit` and `ResolvedObjectType`. In contrast, the type of a variable is said to be of the class `Type`. This means that the concrete object representing the type of a variable can be a resolved or unresolved type.

In the following sections, the meta model for each of the unresolved elements is discussed separately in more detail.

## 2.7.2 Packages

Packages can be imported in a compilation unit using a package import statement. Since such an import statement cannot refer to the unnamed package, only the named packages have a resolved-unresolved partition. The meta model for packages containing this extension can be seen in Fig. 2.20.

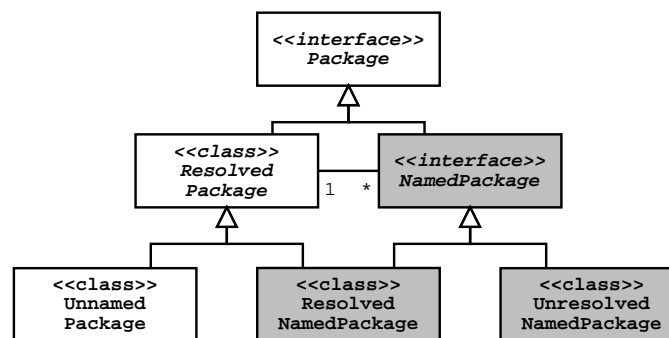


Figure 2.20: The package hierarchy (see Fig. 2.1) extended with resolved and unresolved packages.

The parent of a compilation unit can be the unnamed package or a named package (this is also true for the parent of a named package). To express this, the type

`ResolvedPackage` is introduced as superclass of `UnnamedPackage` and `ResolvedNamedPackage`.

### 2.7.3 Result Type

Each non-constructor has a result type. When the object representing the result type of a method has not been found (yet), the method keeps a reference to an unresolved return type, otherwise it refers to a resolved return type. The resulting hierarchy for return types is shown in Fig. 2.21. In general, a result type can be related to zero or more methods (non-constructors). An unresolved result type however is always connected to exactly one method. This is represented in Fig. 2.21 by using a dotted line.

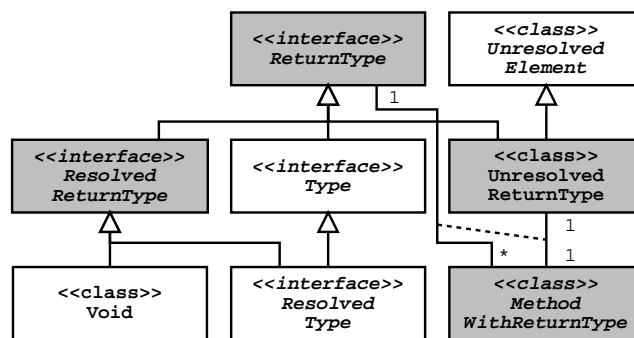


Figure 2.21: Resolved and unresolved return types.

The class `UnresolvedElement` does not represent a concept from the Java programming language. It is the superclass of the unresolved elements contained in the hierarchy of result types. This class contains the common properties and code of these unresolved elements.

### 2.7.4 Type

Each variable has a reference to its type. An unresolved type is used when the real type has not been found yet. The class hierarchy can be seen in Fig. 2.22.

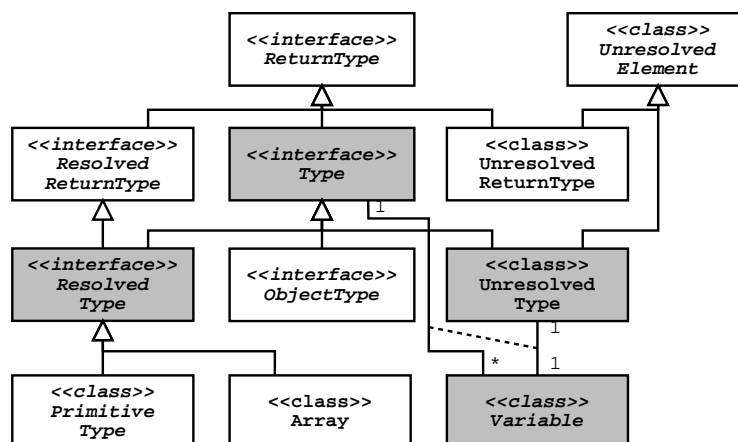


Figure 2.22: Resolved and unresolved types.

`UnresolvedType` does not inherit from `UnresolvedReturnType` because the type invariants of the second class are not satisfied in the first class: an unresolved return type is connected to a method, while an unresolved type is connected to a variable.

## 2.7.5 Object Type

An import statement in a compilation unit can refer to an object type or a package. An import declaration importing an object type, does not reveal whether a class or interface is imported. Since the object type in the import statement is not always known, the class `UnresolvedObjectType` has been introduced. The corresponding model is shown in Fig. 2.23.

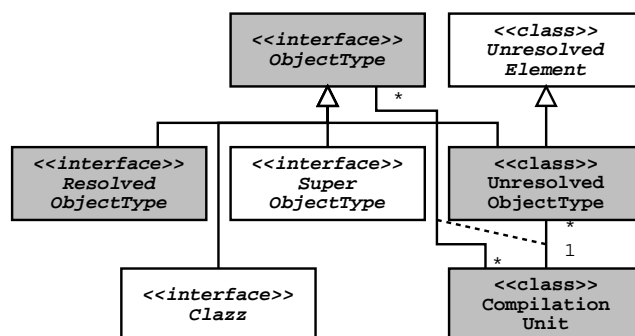


Figure 2.23: Resolved and unresolved object types.

## 2.7.6 Class

The throws clause of a method can contain zero or more classes. These classes represent the exceptions that can be thrown in the body of that method. When such a class is not found in the object structure, an object of the class `UnresolvedClass` is created, as can be seen in Fig. 2.24.

The class `ResolvedClass` is an interface because `ResolvedSuperClass`, which is an interface, inherits from it. `ResolvedSuperClass` should be an interface because its subobjecttype `RegularClass` already inherits from the class `NonAbstractObjectType` and it cannot inherit from two different classes.

## 2.7.7 Superclass

When the object representing the superclass of a class is not found, an instantiation of the class `UnresolvedSuperClass` is used. Figure 2.25 shows the class hierarchy for superclasses.

To make the drawing clearer, the interface `Clazz` has been included two times. A resolved superclass can be an abstract class or a regular class.

## 2.7.8 Superinterface

Object types can have one or more superinterfaces. When such an interface has not been found, objects of the class `UnresolvedInterface` are used as substitution. A drawing of the class hierarchy can be seen in Fig. 2.26.

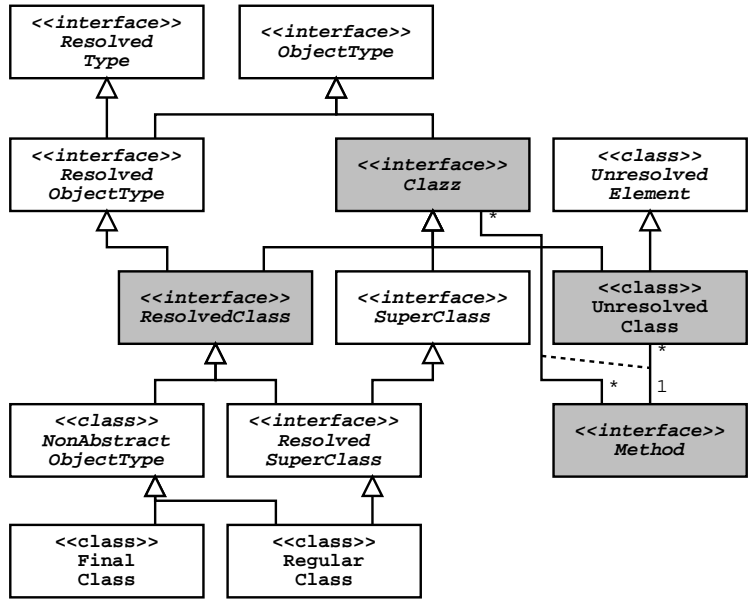


Figure 2.24: Resolved and unresolved classes.

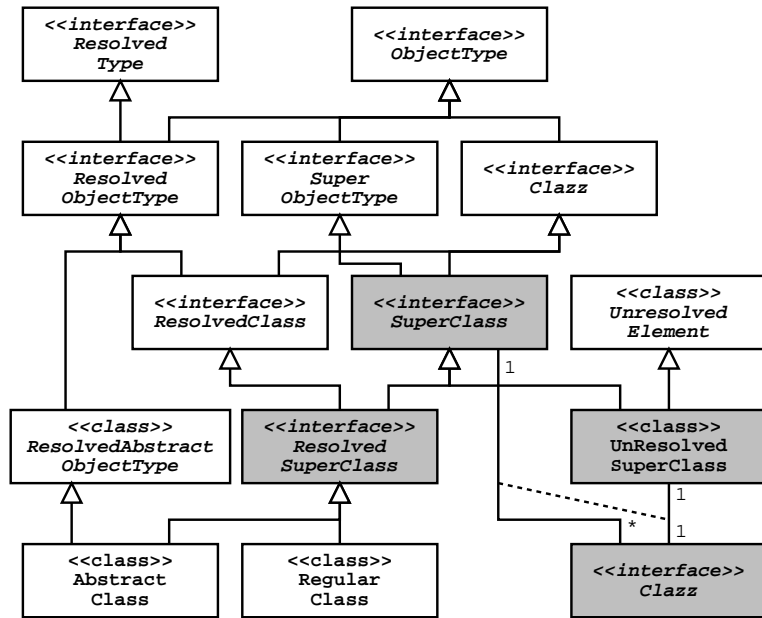


Figure 2.25: Resolved and unresolved superclasses.

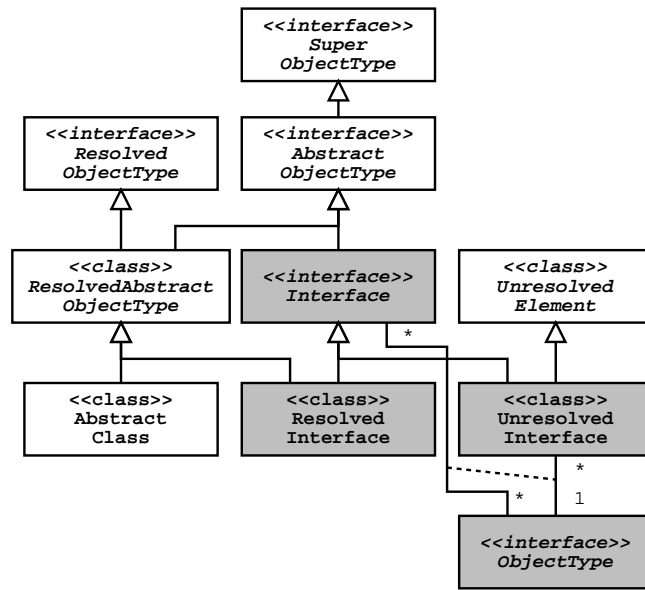


Figure 2.26: Resolved and unresolved interfaces.

A resolved abstract object type can be a resolved interface or an abstract class.  
 SuperInterface was the last unresolved element.

## Chapter 3

# Building the Object Structure: Acquire

Using the meta model described in Chap. 2, we have built a tool which generates documentation for Java files. The input of the program are Java source files; the output consists of XML files that can be used as documentation for the given files. The program executes in three steps, as can be seen in Fig. 3.1. In a first phase, the input files are parsed using the parser generator ANTLR [ANTLR] and an abstract syntax tree (AST) is built for them. From these ASTs, a meta model instance is built up, containing all Java elements that are present in the parsed files. Building the meta model instance is the second phase of the program. During the third phase, documentation is generated starting from the meta model instance.

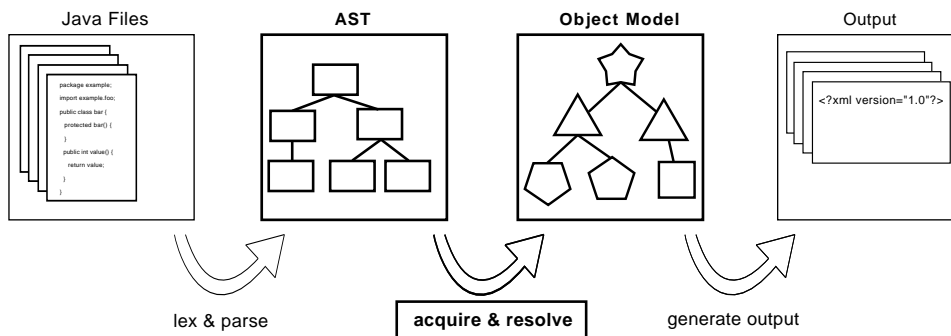


Figure 3.1: The second phase of the program handles the transformation from the ASTs to the meta model.

In this and the following chapter, a detailed description is given about the second phase of the program. This means that we will describe how a meta model instance is built, starting from a set of parsed files. In this chapter, we start with a short description of the strategy used in the second phase. It shall be made clear that this phase consists of two separate subphases, the acquire phase and the resolve phase. The acquire phase will be described in this chapter. The resolve phase will be discussed in Chap. 4.

### 3.1 Short Description of the Second Phase

The goal of the second phase of the program we developed is to build up a meta model instance, containing all Java elements that appear in the given input files, together with the necessary references between directly related elements such as a class and its superclass, a method and its return type, ... This meta model instance is built up in two subphases, as is represented in Fig. 3.2. These subphases are called the acquire phase and the resolve phase.

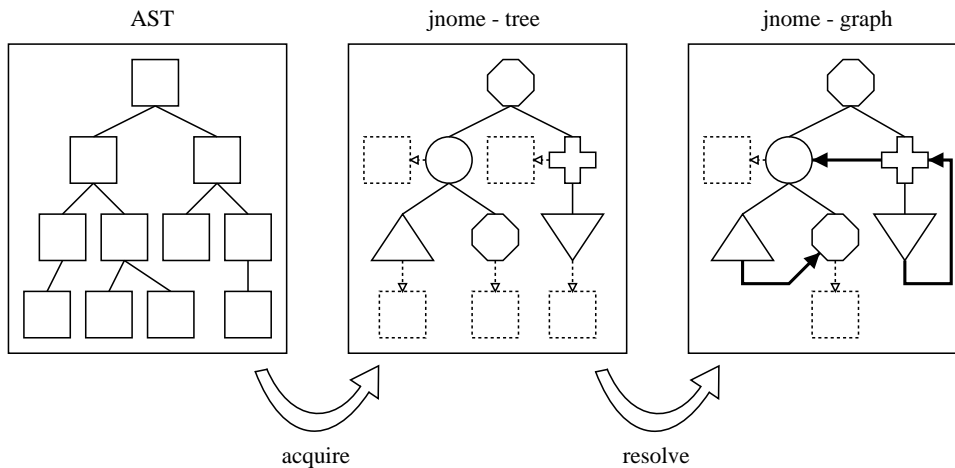


Figure 3.2: The second phase of the program (see Fig. 3.1) is composed of two subphases: the acquire phase and the resolve phase.

During the **acquire phase**, a Java object is created for each class, method, variable, ... in the parsed files. The parent-child relationships between these Java elements are also set: each package contains references to its subpackages and to the compilation units that are defined in that package, each class contains references to its methods and variables, ... No cross reference are considered, so the result of this subphase is a **tree**. The acquire phase is discussed in more detail in the rest of this chapter.

In the **resolve phase**, the cross references between the different objects in the tree resulting from the acquire phase are considered: from each variable a reference is laid to its type, from each class to its superclass, ... The result of this subphase is a **graph**. The resolve phase is discussed in more detail in Chap. 4.

### 3.2 Short Description of the Acquire Phase

During the acquire phase, a tree is created containing meta model concepts. This tree contains all *Java elements* that are present in the parsed files. In addition, also the primitive types and the result type void are present in this structure. The unnamed package constitutes the root of the tree (as mentioned in Sect. 2.1.2.2). The *parent-child relationships* between the different Java elements are also established during the acquire phase, giving rise to a tree.

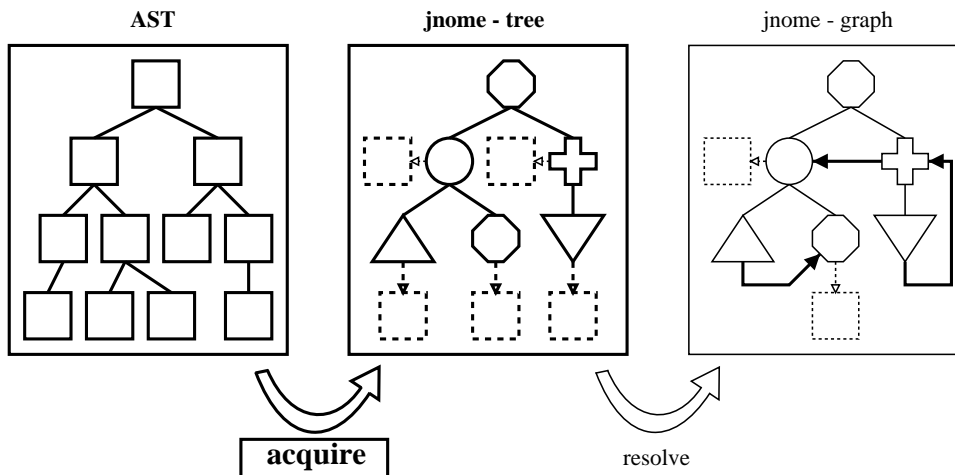


Figure 3.3: During the acquire phase, a tree is built containing instantiations of the classes from the meta model. Cross references in the parsed files are represented by unresolved elements.

### 3.3 Description of the Tree

In this section, a short description is given about the tree that results from the acquire phase.

#### 3.3.1 Java Elements

The tree resulting from the acquire phase contains all Java elements that are present in the parsed files. This means that all packages, compilation units, classes, interfaces, methods and variables in those files are inserted into the object structure; in addition, also the primitive types and the result type 'void' are present in this structure.

The unnamed package (see Sect. 2.1.2.2) constitutes the root of the tree.

#### 3.3.2 Parent-Child Relationships

The parent-child relationships between the different Java elements are also present in the tree:

- The unnamed package (which is the root of the tree) keeps references to:
  - the parsed top level packages (such as java, javax, org, ...),
  - the parsed compilation units that have no package declaration,
  - each of the primitive types,
  - the return type 'void'.
- An ordinary package (i.e. different from the unnamed package) keeps references to:
  - the direct subpackages of the package that are met when parsing the given files,

- the parsed compilation units that are defined in the package.
- A compilation unit keeps references to:
  - the classes defined in that compilation unit,
  - the interfaces defined in that compilation unit.
- An object type (a class or interface) keeps references to:
  - the methods defined in that object type,
  - the variables defined in that object type.

An example of such a tree can be seen in Fig. 3.18 on page 50.

**Remark:** all parent-child relationships that are mentioned here are modelled through bidirectional references.

### 3.3.3 Cross References

Other relationships between Java elements, such as the relationship between a class and its superclass, between a variable and its type, ... are not inserted in the tree during the acquire phase. These relationships are called *cross references*. An enumeration of all possible cross references has been given in Sect. 2.7.

When a cross reference is met during the acquire phase, the following steps are executed:

- The name of the referenced Java element is extracted from the AST.
- An unresolved element of the correct type is created.
- The name of the referenced Java element<sup>1</sup> is stored in the unresolved element.
- A bidirectional reference is set between the unresolved element and the object that defined the cross reference.
- The unresolved element is added to the resolver (see Sect. 4.1).

When the acquire phase is completed, all cross references are represented by unresolved elements. During the resolve phase (see Chap. 4), these unresolved elements will be handled by the resolver, who will try to replace each reference to an unresolved elements by a reference to the Java element corresponding with the name stored in the unresolved element.

## 3.4 Why Two Separate Phases?

In jnome, the meta model instance is built up in two separate phases, the acquire phase and the resolve phase. However, it is not necessary to work in two steps. In this section, we will first explain two possible strategies that can be used to build the meta model instance in one single step and we will show that these strategies are rather complex without any directly visible benefits. After that, we will briefly repeat the strategy used in jnome and describe its advantages.

---

<sup>1</sup>This name can be the fully qualified name of the Java element or only the simple name. Import statements always contain the fully qualified name of the imported package or object type. In all other situations both the fully qualified name and the simple name can appear.

### 3.4.1 Building the Meta Model Instance in One Step

Why two separate phases are desirable will be explained using the following example: consider a class `Elephant` with superclass `Animal`. During the parsing of the file `Elephant.java`, the `extends` clause in this file makes clear that `Elephant` has a superclass `Animal`. When building the meta model instance in one step, we should try to set a reference from the class `Elephant` to the class `Animal` immediately.

#### 3.4.1.1 Examine the Remaining Files

The most straightforward strategy is to simply search for the class `Animal` in the tree that has been built so far. If the class `Animal` is already parsed at the moment we meet the `extends` clause, then the class `Animal` is present in the tree and a reference can be set from the class `Elephant` to its superclass `Animal`. But when the class `Animal` it is not parsed yet, but will be parsed later, this strategy is not sufficient: we also want to set a reference, but we do not find the class in the tree at the moment the `extends` clause is met. In this case, we should consider the files that are not parsed yet and examine whether the class to be found (`Animal`) is defined in one of these. If it is, we can continue parsing the file in which `Animal` is defined, insert this class into the tree, and finally set a reference from `Elephant` to `Animal`. When `Animal` is not found in the considered files, we know that this Java element will not be parsed. This raises the following question: what should be done when some class is referenced from a given file, but this class will not be parsed?

When a class is searched for and not found, it could be marked (for example, by setting the reference to null). This strategy however causes some problems, since we lose some important information, in casu the name of the superclass of `Animal`. This information is very useful. It could be used by an output module to print the name of the superclass of `Elephant`. The information is also useful to retain when a meta model instance that is built at one moment can be expanded with more elements later on. This means that the files are parsed in several phases. For example, first the files `Elephant.java`, `Penguin.java` and `Tiger.java` are parsed and a meta model instance is built for them. In a second phase, the files `Mouse.java` and `Spider.java` are handled. During the first phase, possible references from the classes `Elephant`, `Penguin` and `Tiger` to the classes `Mouse` and `Spider` will not be resolved. Unfortunately, they will not be resolved during the second phase either because no information about these cross references has been kept during the first phase.

A possible solution for this problem could be to parse the classes `Elephant`, `Penguin` and `Tiger` again during the second phase. However, this is not always possible and it consumes a lot of time.

Another solution to this problem is to use a placeholder object for Java elements that are referenced from a certain file, but cannot be found when searching the other given files. This placeholder object should contain the name of the Java element that is referenced.

#### 3.4.1.2 Examine the Classpath

Another strategy is the following: when some Java element cannot be found in the tree that has been built so far, nor in the given files, one can consider the `CLASSPATH` variable. When the given files are compilable, one of the files that can be reached in this way should contain the element to be found. This means

that searching the files in the classpath will always end successfully. In case only a class file (and no Java file) is found, Java reflection can be used.

This strategy will bring along more work, because much more files will be parsed than the files that are given. On the other hand, the meta model instance that will result is complete, in the sense that all cross references are resolved.

When the given files are not compilable, it is possible that searching the classpath is not always successful. In this case, we should again work with unresolved elements as described in the previous section.

### 3.4.1.3 Conclusion

The strategies described above allow us to handle cross references in one single phase. Unfortunately, these strategies are rather complicated. This is why a more beautiful approach has been developed. The resulting strategy will be discussed in the next section.

## 3.4.2 Two Separate Phases

In jnome, the object structure is built in two separate phases. During the **acquire phase**, an object is created for each Java element in the parsed files and this object is inserted into the object structure. Cross references are not resolved during this phase. Each time a cross reference is met in a file, an unresolved element (of the correct type) is created, containing the name of the Java element to be found. In a separate **resolve phase**, the unresolved elements are considered one by one and they are replaced by the Java elements for which they are a substitution (if possible).

Working in two separate phases leads to a more easy strategy. After finishing the acquire phase, all Java elements in the parsed files are present in the meta model instance. We just have to search in this tree to find out whether a certain Java element is parsed or not.

## 3.5 Implementation of the Acquire Phase

During the acquire phase, a tree is built containing all Java elements that are present in the parsed source files. The unnamed package constitutes the root of the tree.

To build up this object structure (tree) without polluting the code of the meta model, the *Worker Pattern* is used (see Chap. 5 for a more detailed description of the Worker Pattern). In the Worker Pattern, several workers perform operations on the meta model. Each worker is responsible for one of the classes in the meta model. In the context of the acquire process, each worker is responsible for inserting a certain Java element in the object structure. Examples of such workers are the `CompilationUnitAcquirer`, the `ResolvedInterfaceAcquirer`, the `ConstructorAcquirer`, etc. The workers form a tree structure themselves: workers can use other workers to accomplish their tasks. For example, an abstract class acquirer uses a constructor acquirer to handle the constructors in the class it is responsible for.

For creating the different workers, the *Factory Pattern* [Gamma Helm et al 1995] is used.

How the acquire process works is explained step by step in the following paragraphs.

### 3.5.1 The Start of the Acquire Process

During the acquire phase, the parsed files are inserted into the object structure one by one. Figure 3.4 shows a collaboration diagram<sup>2</sup> explaining how the acquire process is started.

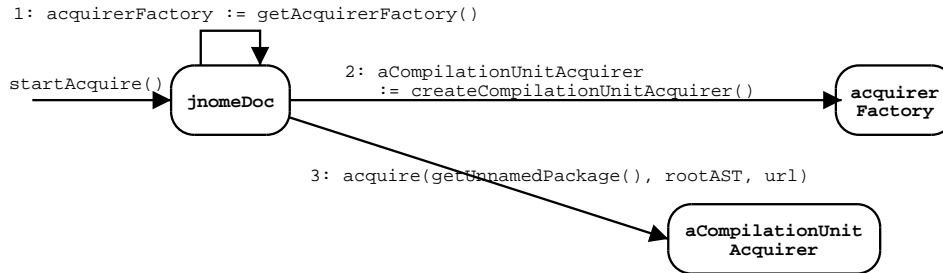


Figure 3.4: To start the acquire process, a compilation unit acquirer is created and this acquirer is started.

Figure 3.4 contains the following elements:

- **jnomeDoc**: an instance of the class containing the main method of our project.
- **acquirerFactory**: the factory that is used for creating acquirers. This factory is passed to each acquirer through their constructor. For example, an abstract class acquirer can use the factory for creating a constructor acquirer that will be responsible for inserting one of its constructors into the object structure.
- **aCompilationUnitAcquirer**: the compilation unit acquirer responsible for creating a new instance of the class `CompilationUnit` and inserting this compilation unit in the tree.
- **getUnnamedPackage()**: the unnamed package forming the root of the tree in which the ASTs should be inserted (see Sect. 3.3 for more details).
- **rootAST**: the root of the AST that should be inserted in the tree.
- **url**: the URL of the file that has led to the given AST (in other words, the URL of the file that was parsed and that gave rise to the abstract syntax tree that has to be inserted in the object structure).

The compilation unit acquirer takes care that the given AST (**rootAST**) is correctly inserted into the object structure. The acquirer creates other acquirers to help him accomplishing his task. How this is done will be described in detail in the following sections.

### 3.5.2 Acquiring a Compilation Unit

The compilation unit acquirer described above is responsible for inserting the compilation unit, described in the given AST, in the tree with the unnamed package as root. This is achieved in four steps that are schematically described below:

---

<sup>2</sup>In this text, objects are represented by rectangles with rounded corners.

1. Each compilation unit is defined in a **package**. This package is determined by the package declaration at the top of the compilation unit. When there is no package declaration, the compilation unit belongs to the unnamed package. A first task of the compilation unit acquirer is to insert the package to which the given compilation unit belongs in the object structure.
2. In a second step, the acquirer creates a new **compilation unit**, with the package from step one as its parent.
3. In a third step, the **import statements** of the compilation unit are handled.
4. In a fourth step, the **object types** defined in the compilation unit are handled.

The four steps are also shown in the collaboration diagram in Fig. 3.5.

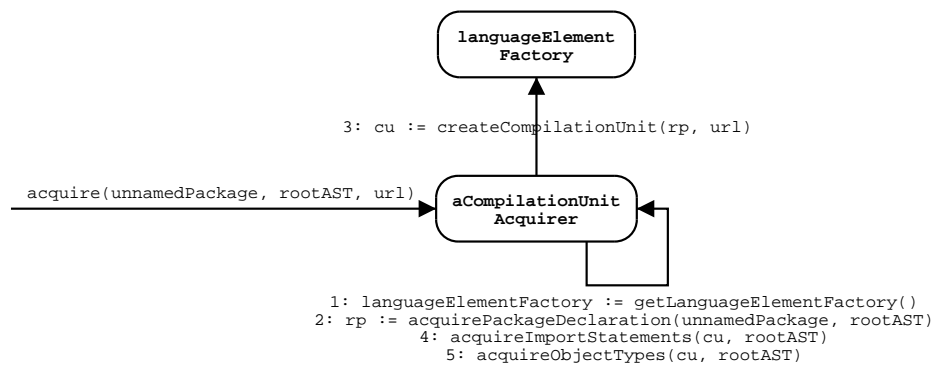


Figure 3.5: Collaboration diagram for inserting a compilation unit in the object structure.

For accomplishing the tasks described above, the compilation unit acquirer uses other acquirers. This is shown in Fig. 3.6.

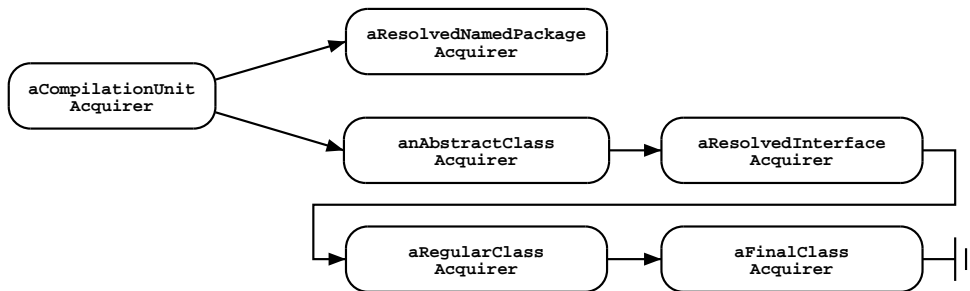


Figure 3.6: The compilation unit acquirer uses two other acquirers to accomplish its task. The resolved named package acquirer takes care of the package declaration in the compilation unit. A chain of resolved object type acquirers makes sure that all the classes and interfaces that are defined the compilation unit are inserted into the object structure.

For inserting the **package** from the package declaration into the object structure, the compilation unit acquirer creates a resolved named package acquirer. This acquirer makes sure that the package is correctly inserted into the tree. How this package acquirer works is discussed in more detail in Sect. 3.5.3.

The **object types** that are defined in the compilation unit are handled one after the other. To acquire an object type, a chain of 'resolved object type acquirers' is

used (see Fig. 3.7). This Chain of Responsibility [Gamma Helm et al 1995] makes sure that the object type is correctly inserted into the tree.

A Chain of Responsibility is used because an object type can be an instantiation of several classes: `ResolvedInterface`, `AbstractClass`, `RegularClass` or `FinalClass`. Each of the acquirers in the chain is responsible for exactly one of these types. How such a resolved object type acquirer accomplishes its task is described in detail in Sect. 3.5.4. A Chain of Responsibility is often used in the Worker Pattern (see Sect. 5.3.3.2).

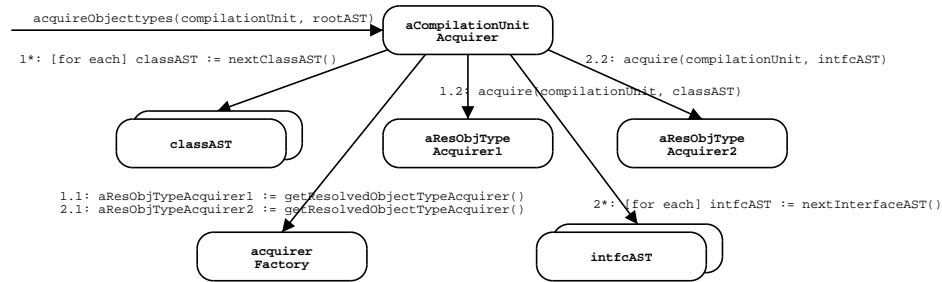


Figure 3.7: For acquiring a class or interface, a chain of resolved object type acquirers is used.

For each **import statement**, the compilation unit acquirer creates an unresolved object type or an unresolved package. Furthermore, a bidirectional reference is set between the newly created unresolved element and the compilation unit (created above). Finally, the unresolved element is added to the resolver (see Sect. 4.1). This procedure is shown in more detail in Fig. 3.8.

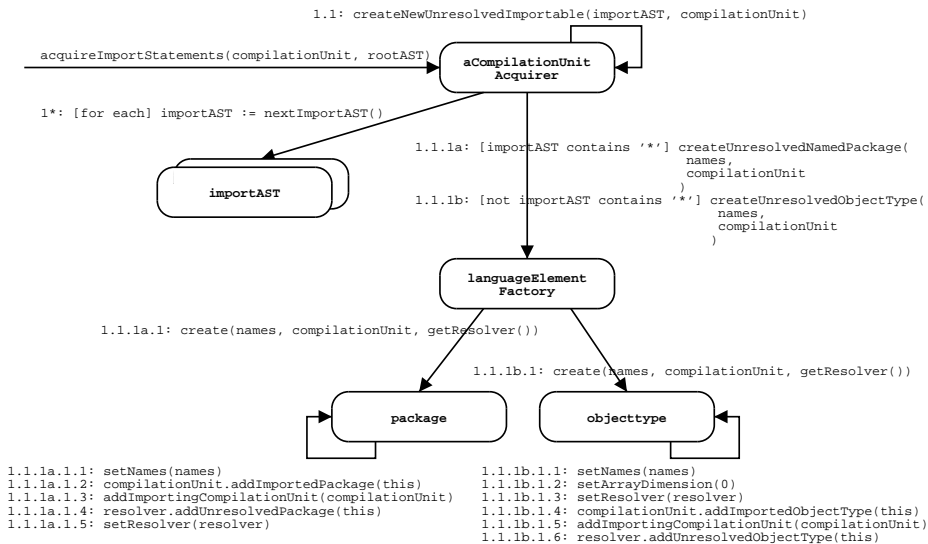


Figure 3.8: For each import statement in a given compilation unit, an instance of the classes `UnresolvedObjectType` or `UnresolvedPackage` is created and a bidirectional reference is set between this instance and the given compilation unit. The unresolved element is also added to the resolver.

### 3.5.3 Acquiring a Package

A package acquirer is created by a compilation unit acquirer to insert the package from the package declaration into the tree. The package acquirer checks whether the concerned package and its parent packages are already present in the tree; when they are not, they are created and added to the tree. The package acquirer receives the following information from the compilation unit acquirer:

- The fully qualified name of the package in the **package declaration**. When there is no package declaration, an empty list is given.
- A reference to the the **unnamed package** (forming the root of the tree in which the package mentioned above should be inserted).

There is no need to insert the **unnamed package** into the object structure: this package is already present and constitutes the root of the tree.

For inserting an **ordinary package** (different from the unnamed package) into the tree, algorithm 1 is used. This algorithm is written in pseudo-code.

---

**Algorithm 1** Algorithm for inserting a package into the object structure.

---

```
current = getUnnamedPackage();
for each element in the fully qualified name:
    if (current.containsSubPackageWithName(element)) {
        current = getSubpackageWithName(element);
    }
    else {
        make a new subpackage s for the current package
        with name element;
        current = s;
    }
```

---

### 3.5.4 Acquiring an Object Type

The acquirers that are responsible for acquiring an object type in the object structure are shown in Fig. 3.9. Each acquirer is responsible for exactly one kind of object type, as reflected in the names of the acquirers.

All resolved object type acquirers use the same scheme:

1. The **name** of the object type is taken from the AST. Furthermore, the acquirer examines whether the object type is **public accessible** or not.
2. An **object type** of the correct type is created, with the name and accessibility modifier determined in step one, and with the compilation unit created by the compilation unit acquirer as its parent.
3. The (optional) **documentation block** is handled.
4. In a fourth step, the **superobjecttypes** (superinterfaces and/or superclass) are inserted into the object structure.
5. In a fifth step, the **methods** of the object type are considered.

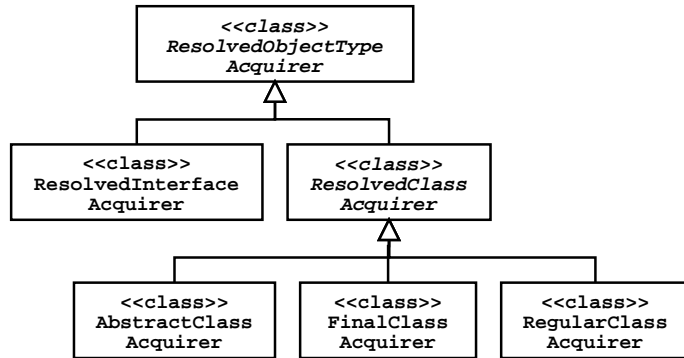


Figure 3.9: The hierarchy of the acquirers that are responsible for inserting an object type into the object structure.

6. In a sixth and last step, the **variables** of the object type are handled.

These six steps are also represented in the collaboration diagrams in Figs. 3.10, 3.11 and 3.12.

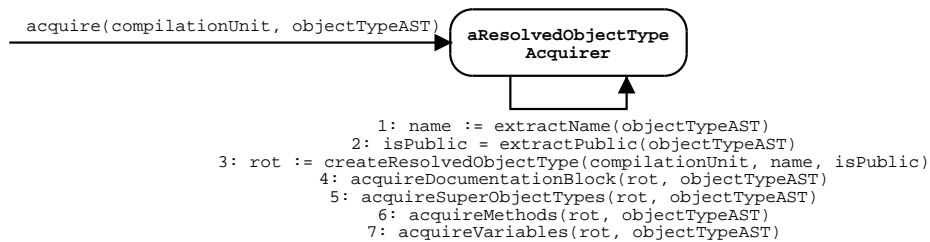


Figure 3.10: Collaboration diagram for inserting an object type into the object structure.

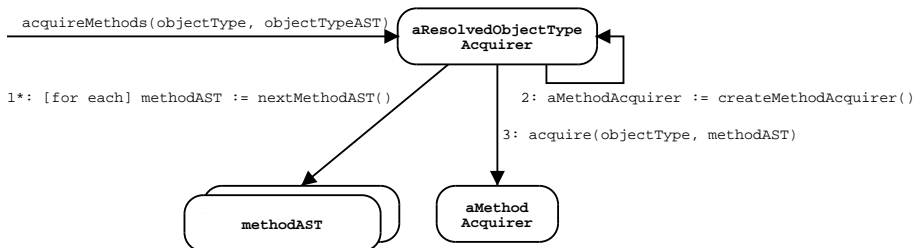


Figure 3.11: Collaboration diagram for inserting the methods of an object type into the object structure.

A resolved object type acquirer uses other acquirers to accomplish its task. This is shown in Figs. 3.13 and 3.14.

The **documentation block** of an object type is handled by an instance of the class `DocumentationBlockObjectTypeAcquirer`. How such an acquirer works is described in Sect. 3.5.7.

For acquiring the **superobjecttypes** of an object type, a strategy is used that is analogous to the one used for acquiring the import statements of a compilation unit (see Fig. 3.8). For each superinterface, an object of the class `UnresolvedInterface`

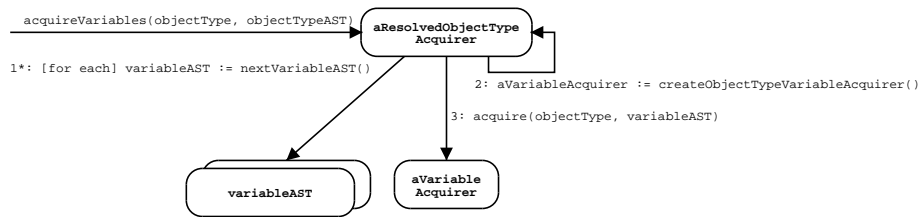


Figure 3.12: Collaboration diagram for inserting the variables of an object type in the object structure.

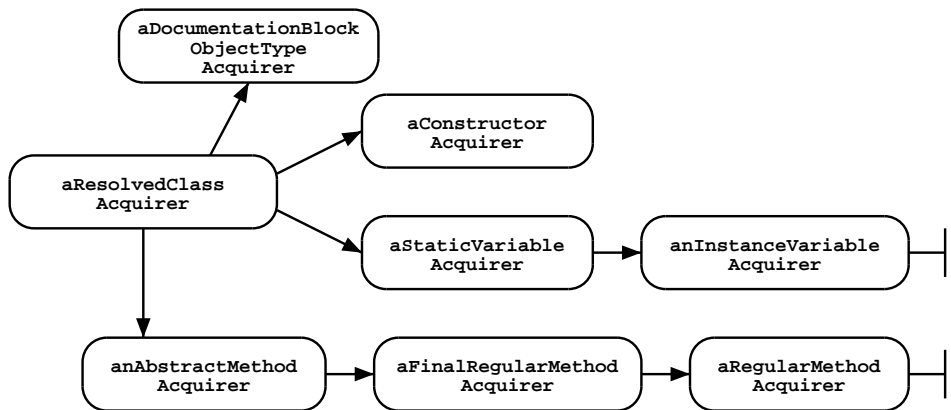


Figure 3.13: The resolved class acquirer uses a documentation block object type acquirer, a constructor acquirer, a class method acquirer and a class variable acquirer to accomplish its task.

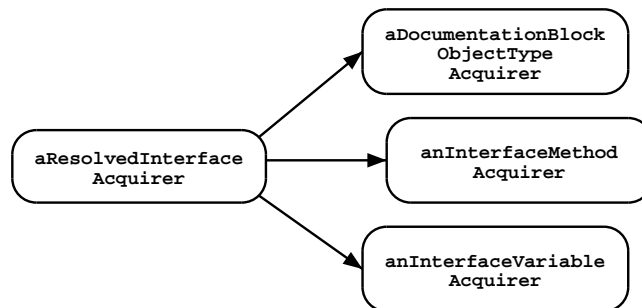


Figure 3.14: The resolved interface acquirer uses a documentation block object type acquirer, an interface method acquirer and an interface variable acquirer to accomplish its task.

is created; for the superclass of a class, an object of the class `UnresolvedSuperClass` is created.

Only the resolved class acquirers can meet **constructors**. These are inserted into the tree by an instance of the class `ConstructorAcquirer`.

A resolved object type acquirer should also handle the other **methods** (the non-constructors) of the concerned object type. A method is inserted into the tree by an instance of the class `InterfaceMethodAcquirer` or by a chain of class method acquirers. How a method acquirer works exactly is discussed in Sect. 3.5.5.

The **variables** of an object type are handled by an instance of the class `InterfaceVariableAcquirer` or by a chain of class variable acquirers. How such variable acquirers work exactly is discussed in Sect. 3.5.6.

**Remark:** for classes and interfaces, a different method acquirer is used. This is due to the fact that methods in an interface are always public and abstract; their declaration does not have to contain the keywords `public` and `abstract`. The interface acquirer must take this into account when creating a new (interface) method.

Analogously, classes and interfaces use a different variable acquirer, since variables in an interface are implicitly static, final and public.

### 3.5.5 Acquiring a Method

The acquirers that are responsible for inserting a method in the object structure are shown in Fig. 3.15.

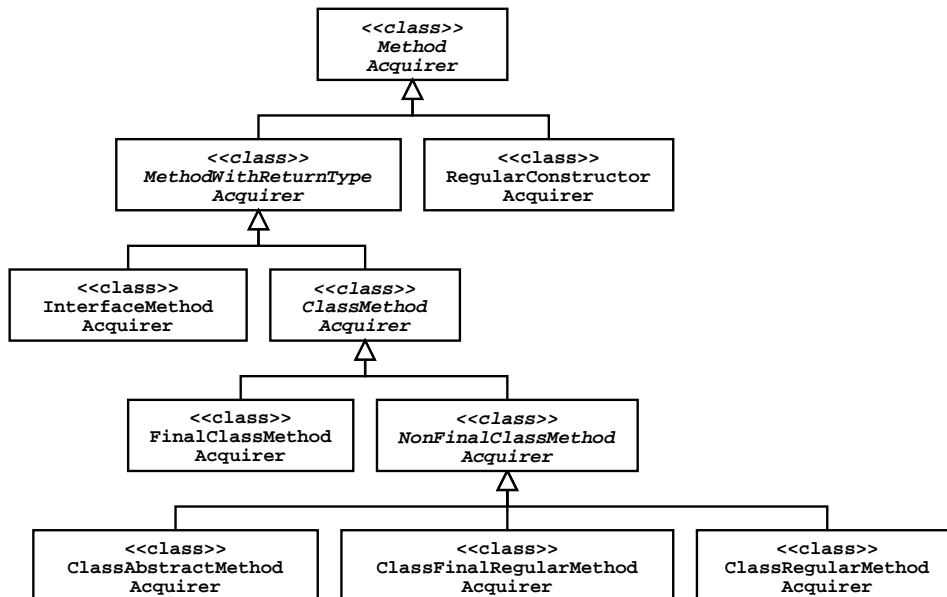


Figure 3.15: The hierarchy of the acquirers that are responsible for inserting a method into the object structure.

The different method acquirers all use the following scheme:

1. The **name** of the method is extracted from the given AST. Furthermore, the acquirer checks whether the method is **native**, **static** or **synchronized**, and determines its **accessibility modifier**.

2. In a second step, the **formal parameters** of the method are considered.
3. A new **method** of the correct type is created, with the name, accessibility and other modifiers determined in step one and the formal parameters determined in step two. The parent of the new method is the previously created object type (see Sect. 3.5.4). When creating a non-constructor, also the result type of the method should be extracted from the AST before the new method can be created.
4. In a fourth step, the (optional) **documentation block** of the method is considered.
5. Finally, the **throws clause** of the method is inserted into the object structure.

The **documentation block** of a method is handled by an instance of the class `DocumentationBlockMethodAcquirer` (see Sect. 3.5.7).

To extract the information about the **formal parameters** of the method from the AST, the method acquirer uses zero or more formal parameter acquirers, each responsible for one formal parameter.

To represent the **return type** of a method and the different exceptions in the **throws clause** of that method, unresolved elements are created.

### 3.5.6 Acquiring a Variable

The hierarchy of the variable acquirers is shown in Fig. 3.16.

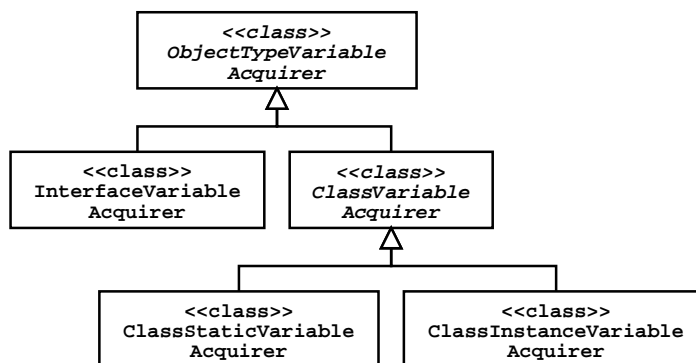


Figure 3.16: The hierarchy of the acquirers that are responsible for inserting a variable into the object structure.

All variable acquirers work according to the following scheme:

1. The **name** of the variable is extracted from the AST. Furthermore, the acquirer determines the **accessibility modifier** of the variable and verifies whether the variable is **final**, **transient** or **volatile**. Finally, the name of the **type** of the variable is extracted from the AST. The acquirer checks whether this type is an array type or not; if it is, the dimension of the array type is determined.
2. A new **variable** of the correct type is created with the name, modifiers and the type determined in step one. The parent of the variable is the object type created earlier (see Sect. 3.5.4).

3. In a third and last step, the **documentation block** of the variable is considered.

For the **type** of a variable, an object of the class `UnresolvedType` is created.

The **documentation block** of a variable is handled by an instance of the class `DocumentationBlockObjectTypeVariableAcquirer` (see Sect 3.5.7).

### 3.5.7 Acquiring a Documentation Block

The hierarchy of the documentation block acquirers is shown in Fig. 3.17.

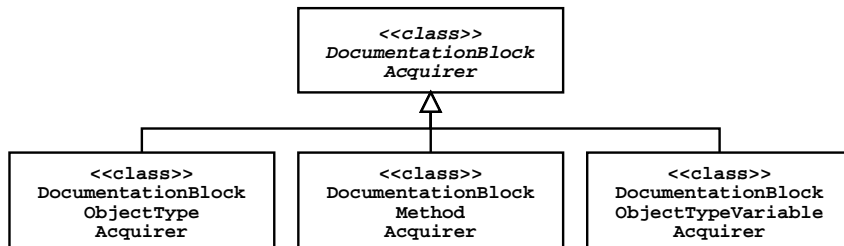


Figure 3.17: The hierarchy of the acquirers that are responsible for inserting documentation block into the object structure.

The documentation block acquirers use the following scheme:

1. The **short description** (standing at the beginning of a documentation block) is extracted from the AST.
2. A **documentation block** of the correct type is created.
3. The **tag blocks** of the documentation block are considered.

The **tag blocks** are handled by an instance of the class `TagBlockAcquirer`. This acquirer extracts the name of the tag and the informal and formal specification from the AST and creates an object of the class `TagBlock` with this information.

## 3.6 A Small Example

In this section, the acquire phase is illustrated with a small example. This example will return in Sect. 4.6 to illustrate the resolve phase.

### 3.6.1 Description of the Files.

First, the source code of two files, `Polygon.java` and `Rectangle.java` is shown. These files will be parsed and a meta model will be built for them. How this happens will be illustrated here and in Sect. 4.6.

The file `Polygon.java` describes a class of polygons. This class contains only one variable `$vertices` containing the vertices of the polygon and a method `dimension` returning the number of vertices of the polygon.

The class `Rectangle` is a class of rectangles. It is a subclass of `Polygon` and overrides the method `dimension`.

The source code of the two files is shown here:

- Polygon.java

```

package mathematics.geometry;
/**
 * A class of polygons manipulated through their vertices.
 * @invar A polygon must have at least three vertices.
 *       | dimension() >= 3
 */
public class Polygon {
    /**
     * Return the number of vertices of this polygon.
     */
    public int dimension() {
        return $vertices.length;
    }
    /**
     * Array of references to the vertices of this polygon.
     * @invar All elements in the array are effective.
     *       | for each i in 0..$vertices.length-1:
     *       |   $vertices[i] <> null
     */
    private Coordinate[] $vertices;
}

```

- Rectangle.java

```

package mathematics.geometry;
/**
 * A class of rectangles manipulated through their vertices.
 * @invar A rectangle has exactly four vertices.
 *       | dimension() = 4
 */
public class Rectangle extends Polygon {
    /**
     * Return the number of vertices of this polygon.
     * @see superclass
     */
    public int dimension() {
        return 4;
    }
}

```

### 3.6.2 The Acquire Phase.

Parsing and acquiring the Java files `Polygon.java` and `Rectangle.java` produces the tree shown in Fig. 3.18.

As described at the beginning of Sect. 3.3.1, the root of the tree is formed by the unnamed package. This unnamed package keeps references to each of the primitive types and to the result type 'void'.

The only packages appearing in the parsed files are `mathematics` and `geometry`. The package `mathematics` is a subpackage of `geometry`; the top level package `mathematics` is placed in the tree as subpackages of the unnamed package.

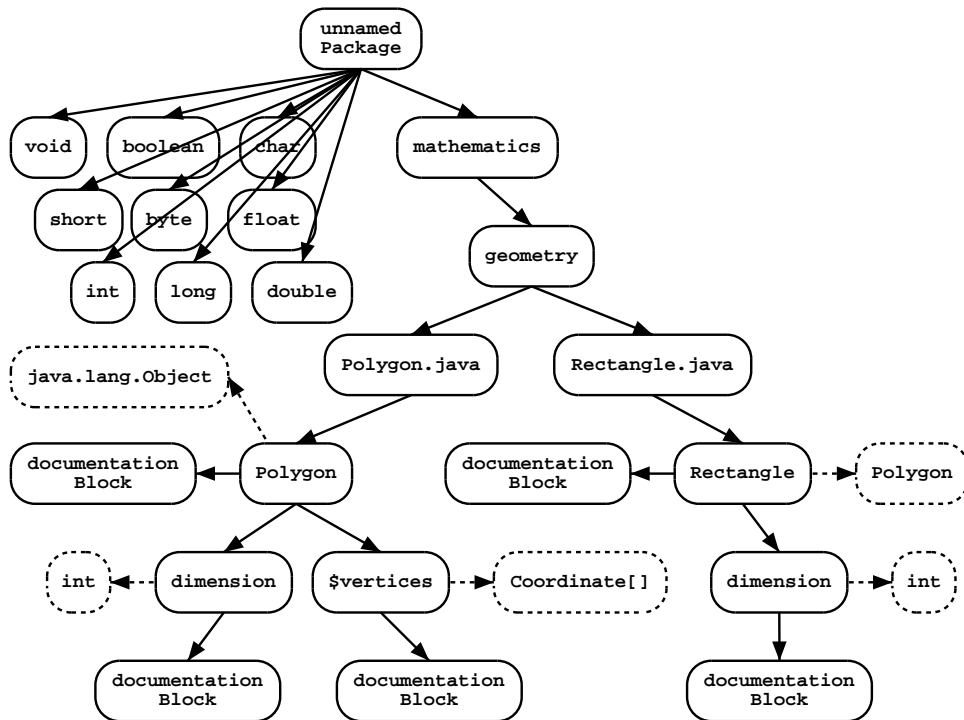


Figure 3.18: The object structure (tree) resulting from acquiring the files `Polygon.java` and `Rectangle.java`.

The package `geometry` contains the compilation units defined in the files `Polygon.java` and `Rectangle.java`.

Both compilation units contain only one class, namely `Polygon` and `Rectangle`.

The class `Polygon` contains one method (`dimension`) and one variable (`$vertices`). The class `Rectangle` only contains one method (`dimension`).

The classes `Rectangle` and `Polygon`, the methods `dimension` and the variable `$vertices` contain references to a documentation block.

Beside the parent-child relationships, a few cross references are present in the tree. As described in Sect. 3.3.3, these relationships are represented by unresolved elements. In Fig. 3.18, the following cross references appear:

- the method `dimension` in the class `Polygon` contains a reference to its type (`int`)
- the variable `$vertices` in the class `Polygon` contains a reference to its type (`Coordinate[]`)
- the method `dimension` in the class `Rectangle` contains a reference to its type (`int`)
- the class `Rectangle` contains a reference to its superclass (`Polygon`)
- the class `Polygon` contains a reference to its superclass (`java.lang.Object`).

During the next phase (i.e. the resolve phase, see Chap. 4), an attempt will be made to replace the references to unresolved elements by references to the real objects. Doing this, the tree changes into a graph (see Fig. 4.2).

## Chapter 4

# Building the Object Structure: Resolve

In this chapter, the resolve phase is discussed. First, the goal of this phase is described briefly. After that, the phase is discussed in more detail. Afterwards, a short description is given about the way references are handled that cannot be resolved and about the way the possibilities for an unresolved object can be determined. Finally, the resolve phase is illustrated with an example.

### 4.1 Short Description

During the acquire phase (see Chap. 3), unresolved elements are created for all cross references (see Sect. 3.3.3). During the **resolve phase**, these unresolved objects are handled by the resolver. The **resolver** is a Java object responsible for resolving cross references. When an unresolved element is created during the acquire phase, a reference to this element is added to the resolver. As a consequence, the resolver always keeps references to all unresolved elements in the meta model instance. During the resolve phase, the resolver considers the unresolved elements one after the other and verifies whether the object structure contains the object represented by the unresolved element. When this object is found, the unresolved element is replaced by the found object.

#### 4.1.1 A Small Example to Illustrate the Resolve Phase

Suppose that, in a compilation unit `Foo.java`, the class `java.util.Vector` is imported through the statement:

```
import java.util.Vector;
```

When this statement is met during the acquire phase, an object of the class `UnresolvedObjectType` is created. In this object, the name “`java.util.Vector`” is stored. Furthermore, a bidirectional reference is set up between the unresolved object type and the object representing the compilation unit `Foo.java`. The resolver also gets a reference to the unresolved object.

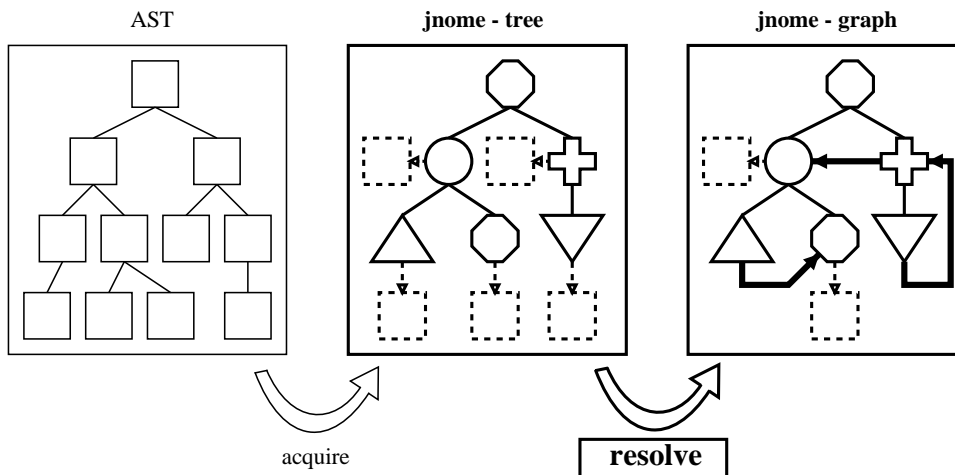


Figure 4.1: During the resolve phase, the unresolved elements are replaced by the objects for which they are a placeholder (if possible).

During the resolve phase, the resolver initiates a search for the object representing the class `java.util.Vector`. If this object is found, the unresolved object is replaced by the found object; further, the reference to the unresolved object is removed from the resolver (making the object ready for garbage collection). When the class `java.util.Vector` is not found, nothing changes: both the compilation unit `Foo.java` and the resolver keep a reference to the unresolved object. In this way, no information is lost and future resolve phases stay possible (for example, when extra Java source files are parsed).

#### 4.1.2 Strategy

During the resolve phase, the unresolved elements created during the acquire phase are considered one by one by the resolver and, if possible, they are replaced by the real object they represent. This means that the return type or package represented by the unresolved element should be searched for in the object structure built so far.

The resolve process is composed of two sequential steps:

1. Resolving the import statements.
2. Resolving the other unresolved elements.

Two separate steps (in the given order) are necessary since the resolved import statements are used during the second step.

In the next section, the first step is discussed. The second step is described in Sect. 4.3.

## 4.2 Resolving the Import Statements

Resolving the import statements involves the following two cases:

- *package import statement*: a package import statement contains the fully qualified name of the imported package. The first word in this name represents a subpackage of the unnamed package (a top level package), the second part represents a subpackage of the first package, the third package is a subpackage of the second, etc.
- *object type import statement*: the name included in an object type import statement is the fully qualified name of the imported object type. The last part of the name is the name of the object type. All words before it represent the fully qualified name of the package in which the object type is defined. When the fully qualified name consists of only one part, the imported object type belongs to the unnamed package.

Resolving import statements is straightforward, because the fully qualified name of the package or object type to be found is given. To explain the resolve process, the example from Sect. 4.1.1 is reused. When a class with fully qualified name `java.util.Vector` has to be found, the following strategy is used: first, one should check whether the unnamed package (the root of the tree) contains a package with name `java`. If this is the case, the found package is considered and we check whether it contains a subpackage with name `util`. Finally, we verify whether this package contains an object type with name `Vector`. This is the class we looked for. When one of the described steps fails (for example, when the package with name `java` does not have a subpackage with name `util`), this means that the class `java.util.Vector` is not present in the tree, i.e. its compilation unit is not parsed. In this case, the attempt to resolve this imported object type has failed and the unresolved object cannot be replaced.

## 4.3 Resolving the Other Unresolved Elements

After resolving the import statements, the other unresolved elements are considered. Such an unresolved element can represent one of the following things: the return type of a method, the type of a variable, a superinterface, an exception in the throws clause of a method or a superclass (see Sect. 2.7). In this paragraph, we describe how these references are resolved.

In a compilation unit, a reference can be made to a class or interface that does not belong to the package the compilation unit is defined in. Such a reference can be made in two manners:

- The fully qualified name of the class or interface is given.
- The object type or the package containing that object type is imported in the compilation unit through an import statement. In this case, the simple name of the object type can be used to refer to it in the compilation unit. The package `java.lang` is always imported implicitly.

When resolving an unresolved element, the knowledge described above should be used. The steps described in the next sections are performed consecutively.

### 4.3.1 Void?

When resolving the return type of a method, the resolver should first verify whether the unresolved element represents the result type 'void'. This is the case when the name of the element consists of only one part and equals `void`.

### 4.3.2 Primitive Type?

When resolving the result type of a method different from `void` or the type of a variable, we should examine whether the unresolved object represents a primitive type. This is the case when the name of the object consists of only one part and equals `boolean`, `char`, `short`, `byte`, `int`, `long`, `float` or `double`.

### 4.3.3 Object Type?

When the result type of a method is different from `void` and different from each of the primitive types, it should represent an object type. The same is true for the type of a variable that is verified not to be a primitive type. The other unresolved elements (unresolved class, unresolved superclass and interface) always represent an object type.

Resolving an unresolved element that represents an object type requires two actions, described in the following sections.

#### 4.3.3.1 Fully Qualified

If the name of the unresolved object consists of more than one part, it is the fully qualified name of the object type that is being represented. In this case, searching the object type in the object structure is performed in the same way as for an import statement (see Sect. 4.2).

#### 4.3.3.2 Not Fully Qualified

If the name of the unresolved object consists of only one part, the represented object type is imported in the compilation unit, or it is located in the package in which the compilation unit is defined (the 'own package'). The following actions are taken consecutively:

1. We verify whether the name of one of the imported object types equals the name of the unresolved object.
2. We examine whether the own package contains an object type with the given name.
3. We investigate whether one of the imported packages (including the package `java.lang` that is implicitly imported) contains an object type with the given name.

As soon as one of these steps is accomplished successfully, the wanted object type is found and the resolve phase can be finished.

It is necessary to perform the above steps in the described order, because object type import statements have priority over package import statements. Furthermore, object types in the own package have priority over object types in imported packages. This last rule does not originate from the Java Language Specification [Gosling Joy et al. 2000], but it follows from our own experience with certain compilers (including JDK [JDK1.4] from Sun). The footnote in the next section also covers this topic.

After (successfully) resolving a cross reference, the bidirectional association with the unresolved element is removed and a bidirectional association is set with the found 'real' object. The resolver removes its reference to the unresolved object making this object ready for garbage collection.

## 4.4 Resolving Array Types

The result type of a method and the type of a variable can be array types. Resolving an array type consists of two steps:

- In a first step, the element type is resolved as described in the preceding sections (the element type of an array type is the array type without the square brackets).
- After resolving the element type, the array type can be determined in a second step. The array type can be obtained starting from the element type: from each type  $T$ , a reference to the array type  $T[]$  can be obtained. This array type is not yet present in the meta model instance, but is created at the time of the first request. By repeating this one or more times (depending on the depth of array nesting), the wanted array type can be reached.

## 4.5 Remarks

### 4.5.1 Parsing a Limited Number of Files

Since only a limited number of files are parsed, it is possible that not all import statements in a compilation unit can be resolved, or that not all object types in a certain package are parsed. These observations have consequences for the resolve phase.

In Sect. 4.3.3.2, we explained how an unresolved element represented by a simple name can be resolved. This explanation is not complete, as we will show here.

When resolving an unresolved element that is not described with its fully qualified name, first all imported object types are considered. If one of these object types has the same name as the unresolved object type, and the imported object type has been resolved, then the wanted object type is found. If one of the imported object types has the same name as the unresolved object type but is unresolved itself, then it is certain that the object type cannot be resolved. The imported object types whose name is different from that of the wanted object type (either resolved or unresolved) can be skipped.

When the unresolved element is not found in the imported object types, the own package should be considered.

If the own package contains an object type with the same name as the unresolved element, the wanted object type is found.

When the name of the unresolved element is not found in the imported object types, nor in the own package, the imported packages should be considered.

It is sufficient to examine only the imported packages that are resolved. If the name of the unresolved element equals the name of one of the object types in one of the resolved imported packages, then the wanted object type is found. If this name

cannot be found there, then the object type has not been parsed and the unresolved element remains unresolved.

In the reasoning above, one fault is made: when the own package is considered and this package does not contain such an object type, we have a problem. That no such object type is present in the meta model instance can have two causes:

1. The package really does not contain an object type with the same name as the unresolved element.
2. The package does contain such an object type 'in reality', but this object type has not been parsed.

In case one, it is correct to continue with the package import statements to see whether the name of the unresolved element is there.

In case two, continuing with the package import statements leads to problems in the case that one of the imported packages also contains an object type with the same name as the unresolved element. When this object type is present in the meta model instance, a reference will be made to this object type instead of the object type in the own package. This is incorrect, because in practice, object types in the own package have priority over object types in imported packages, i.e. when some cross reference is ambiguous in the sense that it can represent an object type in the own package, as well as an object type in one of the imported packages, then the name will be considered as a reference to the object type in the own package.

A consequence of this all is that in practice an object type can never be resolved with certainty when package import statements are used, since a non-parsed object type with the same name can always exist in the own package.

## 4.5.2 Extensions to the Meta Model

The meta model developed so far does not contain all elements of the Java programming language. For example, the body of a method is not considered. As a consequence, concepts like method calls, control structures, ... are not included in the meta model. Furthermore, nested object types (object types defined in another object type) and the initialization of variables are not considered.

When these extra concepts are included in the meta model, a few changes of the resolve phase will be necessary. Some of these adjustments are the following:

- When resolving a method call, a reference has to be made to the method that is being called. Mechanisms such as inheritance and overloading should be taken into account when doing this.
- When the variables in the body of a method are resolved, scopes should be considered.
- When including nested object types in the meta model, the Java Language Specification [Gosling Joy et al. 2000] should be consulted to learn about their scope, the object types, methods and variables they can access, the structure of their fully qualified name, etc.

## 4.6 A Small Example

Performing the resolve phase on the tree in Fig. 3.18 results in the graph shown in Fig. 4.2.

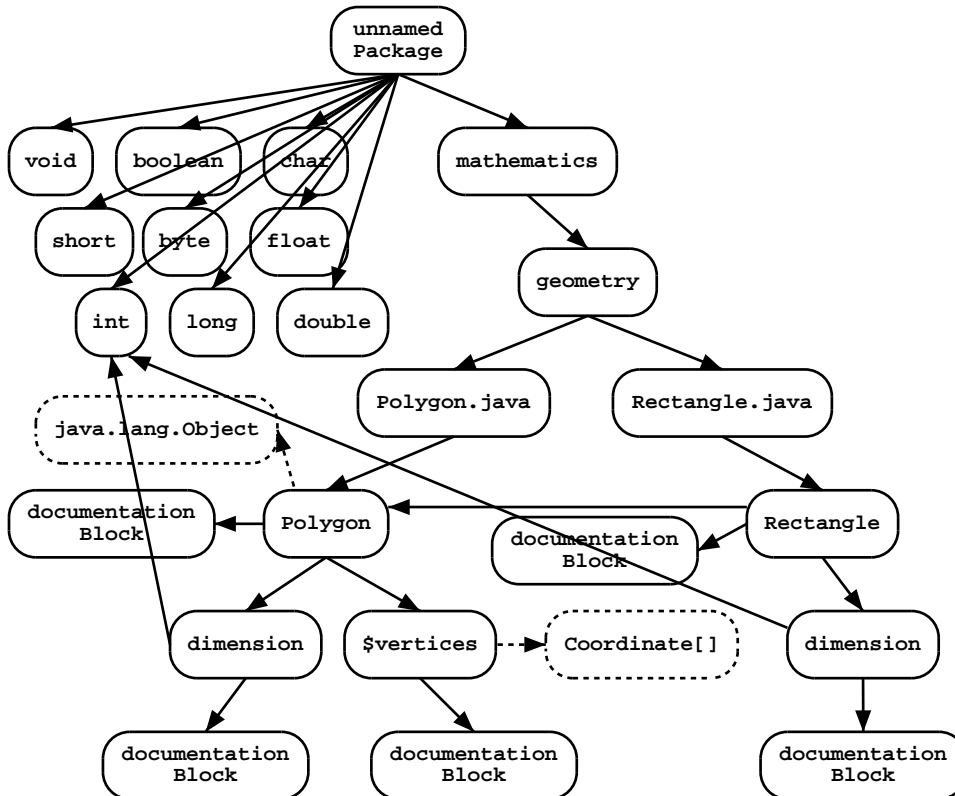


Figure 4.2: The object structure (graph) resulting from acquiring and resolving the files `Polygon.java` and `Rectangle.java`.

In Sect. 3.6, the cross references that were present in the tree resulting from the acquire phase were discussed. After the resolve phase, three of the five cross references are replaced by the real object:

- the method `dimension` in the class `Polygon` has a reference to the object representing the primitive type `int`
- the same is true for the method `dimension` in the class `Rectangle`
- the class `Rectangle` has a reference to its superclass `Polygon`

Two unresolved elements are still in the graph:

- The type of the variable `$vertices` is the array type `Coordinate[]`. An array type is present in the object structure if and only if its element type (in casu `Coordinate`) exists. Since the class `Coordinate` has not been parsed, it does not appear in the object structure, so no reference can be made from the variable `$vertices` to the object representing the array type `Coordinate[]`.

- The superclass of the class `Polygon`, i.e. the class `java.lang.Object` has not been parsed.

How to deal with these remaining unresolved elements is at the discretion of the users of the instantiated meta model. For instance, in a HTML output, an unresolved element could be replaced by a hyperlink to another document containing more information about the unresolved element.

## Chapter 5

# The Worker Pattern

In this chapter, the Worker Pattern is discussed in detail. In jnome, this pattern is used during the acquire phase (see Chap. 3) to build up the meta model instance without polluting the code of the meta model. We also use the same pattern when generating documentation from the meta model instance.

Building up a meta model instance and generating output from a meta model instance are examples of 'operations on an object structure'. To solve this kind of problems, two techniques are commonly used: spreading the code over all elements in the object structure, or the Visitor Pattern [Gamma Helm et al 1995]. These two techniques will be discussed in Sects. 5.1 and 5.2. In jnome, an alternative approach is used to accomplish this kind of operations, namely the Worker Pattern. This pattern will be discussed in Sect. 5.3. The Worker Pattern has a number of advantages over the two commonly used techniques, as will be discussed in Sect. 5.3.2.

### 5.1 Traditional Approach

Traditionally, the code for executing a certain operation on an object structure is spread over the different elements in that object structure. For example, to generate output from a meta model instance, a 'write' method could be added to each of the classes in the meta model. In the main program, the write method of the root of the object structure (the unnamed package) could be called. The write method of the unnamed package uses the write methods in the subpackages and object types of the unnamed package. The write method of an object type in its turn uses the write methods of the variables and methods of the object type, etc.

This approach has several disadvantages. By using this approach, code that does not really belong there is put into a class, making the class less reusable. In this way, the principle of high cohesion is violated.

### 5.2 The Visitor Pattern

A second approach that is often used to execute operations on an object structure is the Visitor Pattern [Gamma Helm et al 1995]. This pattern has a number of disadvantages:

- The classes in the object structure have to be adjusted before the Visitor Pattern can be used: an accept method should be added to these classes. In some situations, this is impossible, namely when there's no access to the source code.
- Each visitor is dependent on each element in the object structure. This implicates, among others, that all visitor classes should be adapted as soon as one of the classes in the object structure is changed, added or removed.
- The Visitor Pattern prevents inheritance. In object oriented programming, inheritance is often used to put functionality that is shared by different classes in one common superclass. In the Visitor Pattern, functionality belonging to different classes is put together in one single visitor class. For example, it is not possible to put the common code for printing a `RegularClass` and an `AbstractClass` in a superclass.  
In addition, when code for one particular Java element has to be changed, it is not possible to override this code partially and selectively. This is a consequence of the fact that the code is written in one single method.
- When the implementation of a visitor for one particular class from the object structure is changed, this happens in a class that also contains code for the other classes in the object structure. This is something that is better not done, to decrease risk of spoiling correct, tested code.

## 5.3 The Worker Pattern

To avoid the problems with the traditional approach and the Visitor Pattern, a new approach is used in jnome. Because we did not meet our approach anywhere else, and because we needed a means to easily talk about the strategy, we decided to give this approach a name. Since the pattern is used to perform some operations, some work on an object structure, we chose the name Worker Pattern.

### 5.3.1 Short Description

The Worker Pattern is similar to the Visitor Pattern. A difference between these two patterns is that, in the Worker Pattern, the code for a certain class in the object structure is put in a separate class instead of in a method, as is done in the Visitor Pattern. This means that, in the Worker Pattern, for each class in the object model, a corresponding 'worker' class is created, that is responsible for executing an operation on that class. This is different from the Visitor Pattern, where only one class exists, the Visitor class, containing a different method for each class in the object model.

When a certain operation has to be executed on the object structure, a worker is created for the root of the object structure. This worker is responsible for executing the operation on the root and all of its subelements. To start the execution, a method of the newly created worker (for example called the work method) should be called. Workers can use other workers to accomplish their tasks: every time an operation has to be executed on a subelement, a new worker is created for that element and the execution of that worker is started. In this way, the complete object structure can be traversed.

### 5.3.2 Advantages

A first problem that is avoided by the Worker Pattern is pollution of the object model. The code for performing an operation on the object structure is now located outside the object model.

The problems we observed in the Visitor Pattern (see previous section) are avoided by providing a separate worker class for each class in the object model instead of placing all code in one single visitor class. Here, we discuss briefly how the observed problems are solved:

- The classes in the object model do not have to be adjusted to let the workers perform their job. As a consequence, the object model is not polluted. This means, among others, that the Worker Pattern can also be used when the object model already exists and its source code is not available.
- A worker, a class responsible for performing a certain operation on a class in the object model, is only dependent on a limited number of classes in the object model. For example, the worker that has to print a regular class is dependent on `RegularClass` and on the direct subelements of a regular class that have to be printed, such as its methods and variables. There's no dependency, for example, on packages or compilation units.
- Inheritance can be used, because the code for a certain class in the object model is put in a separate worker class. Worker classes that have common code can put this code in a superclass. Code can be overridden partially and selectively.
- When the implementation of the operation on one particular class from the object structure has to change, these changes occur in one single worker class. In this way, introducing errors in existing, tested code is avoided.

### 5.3.3 Difficulty

In the Worker Pattern, a certain difficulty arises. This problem, together with its solution, is described in the following sections.

#### 5.3.3.1 Description

When some operation is executed on an object structure using the Worker Pattern, subelements are accessed using inspectors. Through the use of polymorphism, these subelements can have different concrete types. Often, it is necessary to perform a different operation on a subelement depending on its concrete type. Normally, this causes no problems, because dynamic binding can be used. In the Worker Pattern, this is no option, because the code for executing an operation on the object structure is outside the object model.

For example, using the inspector `getClasses()` of the class `CompilationUnit`, we get all classes that are defined in a certain compilation unit. Of these classes, we only know that they are of type `Clazz`. Their concrete type can be `AbstractClass`, `RegularClass` or `FinalClass`. Depending on the concrete type, a class has to be printed differently. For example, an abstract class is always preceded by the keyword `abstract`, while a final class needs the modifier `final`.

### 5.3.3.2 Solution: Chain of Responsibility

As described above, the concrete type of a subelement is not always known exactly due to the use of polymorphism. This causes problems when a different action has to be taken depending on the concrete type of a subelement. A solution for this is the use of the Chain of Responsibility Pattern [Gamma Helm et al 1995].

When using the Chain of Responsibility Pattern, a chain of handlers is created. Each of these handlers is responsible for objects of a certain type. The subelement that has to be treated is given to the first handler in this chain. This handler checks whether it is responsible for the element or not. If he is, he handles the element. If the handler is not responsible, he passes the subelement to the next handler in the chain, etc.

By using a chain of responsibility, a nested if is avoided. A nested if based on the type of the subelement causes problems when a new type is added or when a type is removed. In that situation, every nested if should be extended with an extra case, or an existing case should be removed. This requires changes to a non separate part of the code. As a consequence, errors could be introduced into correct and tested code.

## Chapter 6

# Conclusion

In this paper, we presented a meta model for Java, implemented in Java. The source code is made available as an Open Source project, called jnome.

The meta model is kept clean of pollution by peripheral code in the interest of separation of concerns. An input module that populates the meta model from Java source files is available, as is a proof-of-concept XML output module. Further, the code is well documented, containing both informal and formal specifications.

We invite the Java community to participate in the further development of jnome. The meta model should be completed and adapted to support interactivity; meta model extensions beyond the Java syntax are also welcome, and new in- and output modules can be added to the project. Several possible extensions and uses of jnome were discussed.

The meta model features support for the Java package hierarchy, including the unnamed package. Packages are containers of compilation units, and ultimately, object types. The unnamed package is also the container of the Java primitive types, and 'void'. Types, variables and methods are modelled as an extensive type hierarchy, which focuses on the actual commonalities and differences rather than on how the concepts are presented in the Java Language Specification [Gosling Joy et al. 2000]. The meta model in its current incarnation lacks support for nested classes, the implementation of methods, initialization code and some minor reserved words.

Populating the meta model happens in two separate phases. During the first phase, a tree is built containing all Java elements that are met in the parsed files; cross references are handled using placeholder objects, which are instances of some subclass of `UnresolvedElement`. During the second phase, an attempt is made to replace the unresolved elements by the Java elements they represent. The unresolved elements that cannot be resolved stay in the graph and can be used in subsequent resolve phases or by output modules.

# Bibliography

- [ANTLR] Terence Parr. *ANTLR Reference Manual*. <http://www.antlr.org>.
- [Gamma Helm et al 1995] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman, Inc, 1995.
- [Gosling Joy et al. 2000] James Gosling, Bill Joy, Guy Steele, Gilad Bracha. *The Java Language Specification, Second Edition, 2000*. <http://java.sun.com/docs/books/jls/>.
- [Hunter McLaughlin] Jason Hunter, Brett McLaughlin. *JDOM*. <http://jdom.org>.
- [javadoc] Javadoc. <http://java.sun.com/j2se/javadoc/index.html>.
- [JDK1.4] Sun. Java 2 SDK, Standard edition, v1.4.0 Beta 4 (J2SE). <http://java.sun.com/j2se/1.4>.
- [Leavens] Gary T. Leavens. *JML*. <http://www.cs.iastate.edu/~leavens/JML.html>.
- [Steegmans Dockx et al. 1999] E. Steegmans, J. Dockx, S. De Backer. *Object-gericht programmeren met java*. Acco Leuven, 1999.
- [UML] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999. ISBN 0-201-30998-X.