Barat – A Front-End for Java

Boris Bokowski, bokowski@inf.fu-berlin.de
André Spiegel, spiegel@inf.fu-berlin.de

Technical Report B-98-09
December 1998

Freie Universität Berlin
FB Mathematik und Informatik
Institut für Informatik

Abstract

This paper presents a front-end for Java, called Barat, that supports static analysis of Java programs. Barat builds a complete abstract syntax tree from Java source code files, enriched with name and type analysis information. It supports the complete Java language as of version 1.1. Barat is structured as a framework that supports traversals of abstract syntax trees using visitors and attributes, and provides additional features such as parsing comments as tags, access to parent nodes in the abstract syntax tree, and re-generation of source code. For users of Barat, there is no explicit distinction between phases of loading, parsing, and analyzing Java source code: All actions that need to be performed for building the AST of a Java program are transparent to clients of Barat and are triggered on demand.

Freie Universität Berlin
Institut für Informatik
Takustrasse 9
D-14195 Berlin
Germany

1. Introduction

Barat[1] is a front-end for Java which supports static analysis of Java programs. It parses Java source code files and class files and builds a complete abstract syntax tree (AST) of the parsed Java program. The AST is contains name analysis information and type analysis information. During name analysis, each use of a name (e.g., a field name in a field access expression, a class name in an `extends` clause, or a label name in a `break` statement) is associated with the name's declaration (e.g., the accessed field's declaration, the declaration of the superclass, or the labeled statement that the `break` statement refers to). During type analysis, the static type of each expression is determined. For example, the static type of a character literal is the primitive type `char`, the static type of a string concatenation expression is `java.lang.String`, and the static type of a field access is the accessed field's declared type. As usual, these static types are an approximation to the actual run-time type, which might be a subtype of the static type.

The AST built by Barat is a passive data structure which may not be changed. There are similar systems which allow the AST to be changed, and which therefore enable full compile-time reflection, or even run-time reflection for Java [OpenJava]. However, to our knowledge, Barat is the only system which builds a *complete* representation of Java programs including statements and expressions, together with full name analysis and type analysis. Barat parses source code according to the syntax of Java 1.1 [Gosling et al. 96], and is fully implemented. It parses Java source files as well as class files[2]. There is no explicit distinction between phases of loading, parsing, and analyzing Java source code. All actions that need to be performed for building the AST of a Java program are transparent to clients of Barat and are triggered on demand.

The rest of this paper is organized as follows: Section 1.1 contains installation instructions, and section 1.2 presents a simple example of how Barat can be used. Section 2 explains in detail how Barat can be used "as is" for analyzing Java programs. Section 3 describes how Barat has been implemented and gives hints how to adapt Barat's implementation if necessary. In section 4, two useful example programs demonstrate possible uses of Barat.

1.1 Installation

Barat is implemented using an extension of Java, called "Poor Man's Genericity" (PMG), that supports parameterized classes and types, but is still compatible with the Java virtual machine [Bokowski, Dahm 98]. Most of this, however, is hidden in the implementation package `barat.parser` and can be ignored by users that do not want to change or extend Barat's implementation.

Accordingly, we provide two distributions for Barat. The normal distribution contains class files for all instantiations of parameterized types and may be executed as is on any Java virtual machine, whereas the full distribution comes with PMG and source code for all parameterized types. The latter distribution is more difficult to work with, because of the need to understand some of the concepts of PMG for compiling and running Barat. Please refer to [Bokowski, Dahm 1998] for details on Poor Man's Genericity.

---

[1] "Java Barat" (West Java) is one of the three parts of the island of Java, the other two are called "Java Tengah" (Central Java), and "Java Timor" (East Java).

[2] Note that, as Barat does not perform decompilation for byte code, method bodies of class files cannot represented in the AST

For using the normal distribution of Barat, the Barat distribution main directory must be included in the classpath. Additionally, Barat makes use of JavaClass, a library for parsing class files [Dahm 98], which has to be included in the classpath as well. All files needed for installing Barat are accessible from the Barat homepage, http://www.inf.fu-berlin.de/~bokowski/barat, together with up-to-date installation instructions.

1.2 Example program using Barat

See listing 1 for a first example of how Barat can be used:

```
public class FirstExample
{
   public static void main(String[] args)
   {
        barat.reflect.Class c = barat.Barat.getClass(args[0]);
        System.out.println("accessible fields of class "+ c.getName() + ":");
        for(barat.collections.FieldIterator i=c.getFields().iterator();
             i.hasNext();)
        {
             barat.reflect.Field f = i.next();
             if(!f.isPrivate())
                  System.out.println("\t" + f.getType() + " " + f.getName());
        }
   }
}
```

Listing 1 class `FirstExample`

Barat consists of several Java packages. In this example, three packages are used:

- The package `barat.reflect` contains the node types of the abstract syntax tree, like `Class`, `Interface`, `AbstractMethod`, `Field`, `Parameter`, `Block`, `ObjectAllocation`, etc.
- The top-level package `barat` contains, among others, class `Barat` which is the main entry point for Barat with methods like `getClass()` or `getInterface()` which return an AST node representing a parsed class or interface.
- The package `barat.collections` contains type-specific collection classes (lists and iterators) for AST node objects.

In the example program, the first String argument to `main()` is expected to be a fully qualified class name. By calling `barat.Barat.getClass()`, an AST node object for this class is retrieved. (By invoking accessor methods on such "initial" node objects, on-demand name analysis, type analysis or loading of other source and class files is triggered, without the user noticing it.) After printing a short message with the class name, an iterator object is used to iterate over all fields of that class, printing types and names of each non-private field.

2. Using Barat

This section explains how to use Barat for analyzing standard Java programs, using the Barat abstract syntax tree as a passive data structure and navigating within it. Usually, even very complex analyses can be implemented this way (some examples will be given in this section and in the final section of the paper).

This section is divided into five parts. We will first describe features of AST nodes in general (section 2.1), and then explain the individual elements of Barat abstract syntax trees (section 2.2). In section 2.3, we will describe the main entry class of Barat and how to get hold of the root object of the abstract syntax tree. Finally, in sections 2.4 and 2.5, we will present visitors and attributes as techniques for navigating within a Barat syntax tree when performing a given analysis task.

2.1. AST Nodes

The common supertype of all AST node types is the interface `Node` (see listing 2), which is part of the top-level package `barat`. All other AST node types are defined by interfaces in package `barat.reflect`. They are classified into *concrete interfaces* which have a concrete class as their implementation, and *abstract interfaces* which are implemented as abstract classes. For example, there is an abstract interface `AMethod` representing Java methods in general, from which two concrete interfaces `ConcreteMethod` and `AbstractMethod` are derived. Abstract interfaces, i.e. abstractions that help structuring the AST node type hierarchy, are marked with an uppercase prefix "`A`".

For accessor methods, we have used the following naming convention: Accessor methods for association, aggregation, or containment relationships start with `get`, whereas accessor methods for attributes do not have any prefix. For example, a `BinaryOperation` object supports the method `getLeftOperand()` for accessing the left operand, which is contained within the binary operation, whereas the operation for retrieving the binary operation's operator as a string is called `operator()` (without any prefix).

```
public interface Node
{
   // container and containment aspect for this node
   public Object container();
   public String aspect();

   // helper methods for traversing the container chain
   public Object containing(java.lang.Class ofClass);
   public barat.reflect.Class containingClass();
   public barat.reflect.AUserType containingUserType();
   public barat.reflect.AMethod containingMethod();

   // line number for a node:
   public int line_number();

   // access to tags (/*:special*/ comments):
   public boolean hasTag(String t);
   public barat.collections.StringList getTags();

   // method that calls back a visitor object:
   public void accept (Visitor v);

   // defining attributes and retrieving attribute values:
   public void addAttribute(Object key, AbstractAttribute a);
   public Object attributeValue(Object key);
}
```

Listing 2 Interface of `Node`

Containment

The most important operations defined for all nodes are those involving the "containment" relation (methods `aspect()`, `container()`, and `containingXXX()`). These methods allow to find out, for a given Barat `Node`, what higher level constructs it is contained in. For example, return statements

are contained in methods, which are contained in classes. In Barat, you can access these "containers" of a statement as follows:

```
barat.reflect.Return  r = ...;
barat.reflect.AMethod m = r.containingMethod();
barat.reflect.Class   c = m.containingClass();
```

The contains-relation is transitive, so that you could also write `r.containingClass()` on the last line. The method `container()` returns the immediate container of a `Node`, while `aspect()` describes what kind of constituent a `Node` is for that container – you can think of `aspect` as the name of the instance variable of the parent container in which the `Node` is stored. In our above example, method `m` would be immediately contained in class `c`, and the containment aspect would be `"concreteMethod"` (see also the class diagram, figure 2). As another example, consider an assignment:

```
a = b
```

Both variable access expressions `a` and `b` are immediately contained within the assignment expression; but the containment aspect for `a` is `"lvalue"`, and the containment aspect for `b` is `"operand"`.

The methods `containingClass()`, `containingUserType()`, and `containingMethod()` traverse the containment hierarchy from inside to outside until an object of the requested type is found – if none exists, null is returned. These methods are the most commonly needed operations on the containment hierarchy, which is why they are provided explicitly. To search for other kinds of containers, the operation `containing(ofClass)` may be used. For example, to search for an enclosing if-statement, write:

```
barat.reflect.AStatement s = ...;
barat.reflect.If = (barat.reflect.If)s.containing(
                                          barat.reflect.If.class);
```

It turns out that in the presence of inner classes, these methods could sometimes produce counter-intuitive results. For example, consider

```
Object method() {
     if (x == 0)
          return new Object() {
               public int hashCode() { return 14; }
          };
}
```

Suppose you want to check whether the statement `return 14;` is nested within an if-statement. Using `containing(barat.reflect.If.class)`, as described above, would yield the outer if-statement (in which the entire anonymous class is contained), which is probably not the intended result. Therefore, the methods `containingClass()`, `containingUserType()`, and `containingMethod()`stop when they encounter a class or interface, except when the searched container is `CompilationUnit` or `Package`. In the example, the search for a containing if-statement would therefore stop at the anonymous class, and null be returned.

The compilation unit in which each node is contained can be accessed using

```
CompilationUnit cu = (CompilationUnit)
                        node.containing(CompilationUnit.class);
```

If the compilation unit is available as source code, `cu.hasSource()` returns `true`. By calling `line_number()` on `node`, its line number in the compilation unit's source file can be obtained. The

line number refers to the source file, whose name is returned by calling `cu.filename()`. If the line number is not available, `line_number()` returns `-1`.

Tags

The methods `hasTag()` and `getTags()` provide access to a special feature of Barat's parser: Java constructs may be marked with specially formatted comments in the source file, as in the following example:

```
/*:log-uses*/ /*:system_layer*/ class TagExample {
     /*:pre:o!=null*/
     public void print(Object o) {
          // ...
     }
}
```

Here, the parser stores "`log-uses`" and "`system_layer`" as tags for class `TagExample`, and "`pre:o!=null`" as tag for method `doit`. The example demonstrates three possible uses for tags:

- The tag "`log-uses`" could be used by a Barat-based tool that collects information about which classes reference the tagged class.
- It could be enforced that classes marked with "`system_layer`" should not be called from classes that are not marked with "`system_layer`" as well, regardless of normal Java access control.
- Tags starting with "`pre:`" and containing a precondition could be used to generate modified source code containing runtime checks. (See section 4.2 for an example of generating modified source code.)

Visitors and Attributes

The remaining methods, `accept()`, `addAttribute()`, and `attributeValue()`, support traversals and analyses of abstract syntax trees according to two complementing paradigms: the method `accept()` is the hook for traversals using the Visitor design pattern [Gamma et al. 95], while `addAttribute()` and `attributeValue()` support the definition of on-demand traversals similar to attribute grammars [Knuth 68; Kastens, Waite 94]. Both ways of traversing the AST will be explained in detail in section 2.4.

2.2. Elements of the Abstract Syntax Tree

In this section, we will go through the main categories of AST node types in detail: type nodes, structural nodes, expression nodes, and statement nodes. In the appendix, a complete UML class diagram is provided for reference. Parts of this diagram will appear in each of the following sections.

Representing Types

The abstract supertype of all Java types (shown in figure 1) is the interface `AType` (remember that the prefix `A` means that this interface is abstract – there is no implementation of it; in other words, there can never be an instance of `AType`, only of one of its concrete subtypes). The methods `isAssignableTo()`, `isPassableTo()` and `isCastableTo()` allow you to check whether an actual value of a given type may, according to the rules of the Java language, be assigned, passed or casted to a formal of another type.

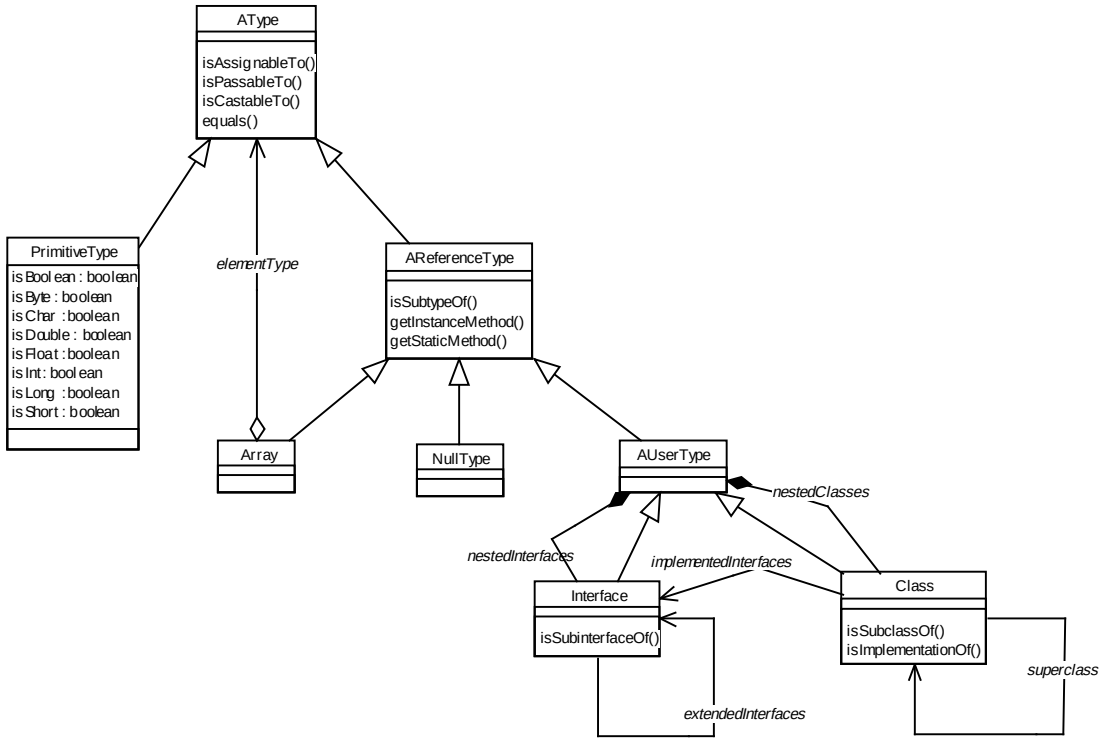


Figure 1 Types

`AType` has two subtypes: `PrimitiveType` and `AReferenceType`.

There is a distinct `PrimitiveType` object for each primitive type in Java; you may check what actual type a `PrimitiveType` object represents by calling `isBoolean()`, `isByte()`, etc. There is also an operation `PrimitiveType.getName()` which returns the Java name of that type. Note that primitive types – as all types in Barat – are represented as singletons, i.e. there is always only a single object that represents type `int` or `double` in the system, and you may compare these types for equality using `==`.

Java reference types, modeled by `AReferenceType`, may be subtypes of other reference types, and you can check this by calling `AReferenceType.isSubtypeOf()`. Following the rules of the Java language, this method returns true if, transitively, the argument type is a superclass or superinterface of this type. The methods `getInstanceMethod()` and `getStaticMethod()` allow you to find a method of a type given the method's name and the types of its arguments. These methods also search any supertypes (transitively) for an applicable method, and follow the "most-specific" rule to match parameter types.

There are three subtypes of `AReferenceType`: `Array`, `NullType`, and `AUserType`, the latter being the abstract supertype of classes and interfaces (these are sometimes also called user-defined types in Java, hence their name). `Interface` has a method `isSubinterfaceOf()`, which transitively checks whether an interface extends another interface. `Class`, on the other hand, has a method `isSubclassOf()`, which transitively follows the extends-relation between classes, while `isImplementationOf()` transitively operates on the graph of superinterfaces implemented by this class. Note that, unlike these methods, the attributes `superClass`, `implementedInterfaces` and `extendedInterfaces` of classes and interfaces are *not* transitive, they only refer to *direct* superclasses and superinterfaces.

It has proven practical to keep arrays and user-defined types separate, i.e. `Array` and `AUserType` are not in any inheritance relation. You can obtain the `Array` for any `AType` by calling `AType.getCorrespondingArray()`. Note that one array is the subtype of another array if and only if their element types are subtypes of another; this rule is implemented by the method `isSubtypeOf()` for arrays.

The `NullType` is the type of the "`null`" literal in Java – it is *not*, as one might think, the "type" returned by a `void` method. (Void methods have no return type at all in Barat, i.e. their `returnType` is null.) The singleton `NullType` object `isSubtypeOf()` any other `AReferenceType` object.

Representing Structure

The inner structure of user-defined types (classes and interfaces) is modeled as shown in figure 2. The essentials of this graph should be clear to any Java programmer, let us merely draw your attention to a few details that may not be entirely obvious.

Methods come in two flavors: `AbstractMethod` objects and `ConcreteMethod` objects, their supertype being `AMethod`. The difference between these kinds of methods is that `ConcreteMethod` objects have a body (a `Block` of statements) while `AbstractMethod` objects do not. It is this difference why both are modeled separately in Barat, whereas, for example, we do not model static methods and instance methods separately because both have the same kinds of constituents.

To distinguish a static method from an instance method, you need to check the method's modifiers: these are available through a set of boolean methods, `isStatic()`, `isPublic()`, `isFinal()`, etc. which are implemented by all Barat nodes that inherit from the interface `AHasModifier` (as can be seen in the diagram, `AMethod` does inherit from this interface).

`Constructor` objects are derived from `AMethod`, adding both a `body` (like in `ConcreteMethod`) and the additional property that any constructor first makes a call to some other constructor (by default, this is the call `super()`). This is modeled as an attribute of type `ConstructorCall`, which Barat initializes to (the equivalent of) `super()` if no explicit call is found in the source code. The exception is of course the constructor of `java.lang.Object`, this is the only case where `getConstructorCall()` returns null.

Note also that `getCalledConstructor()` in `ConstructorCall` returns the real `Constructor` object being called here (the actual parameters represented as a list of `AExpressions`, see below). This is a first example of how Barat resolves inter-class references automatically so that simply by calling the accessor function `getCalledConstructor()`, you can retrieve the constructor object that is actually called.

By convention, each Java class always has at least a single default constructor that does nothing but call `super()`. Such a constructor, if no other constructor is found in the source code of a class, is always automatically generated by Barat and inserted into the class.

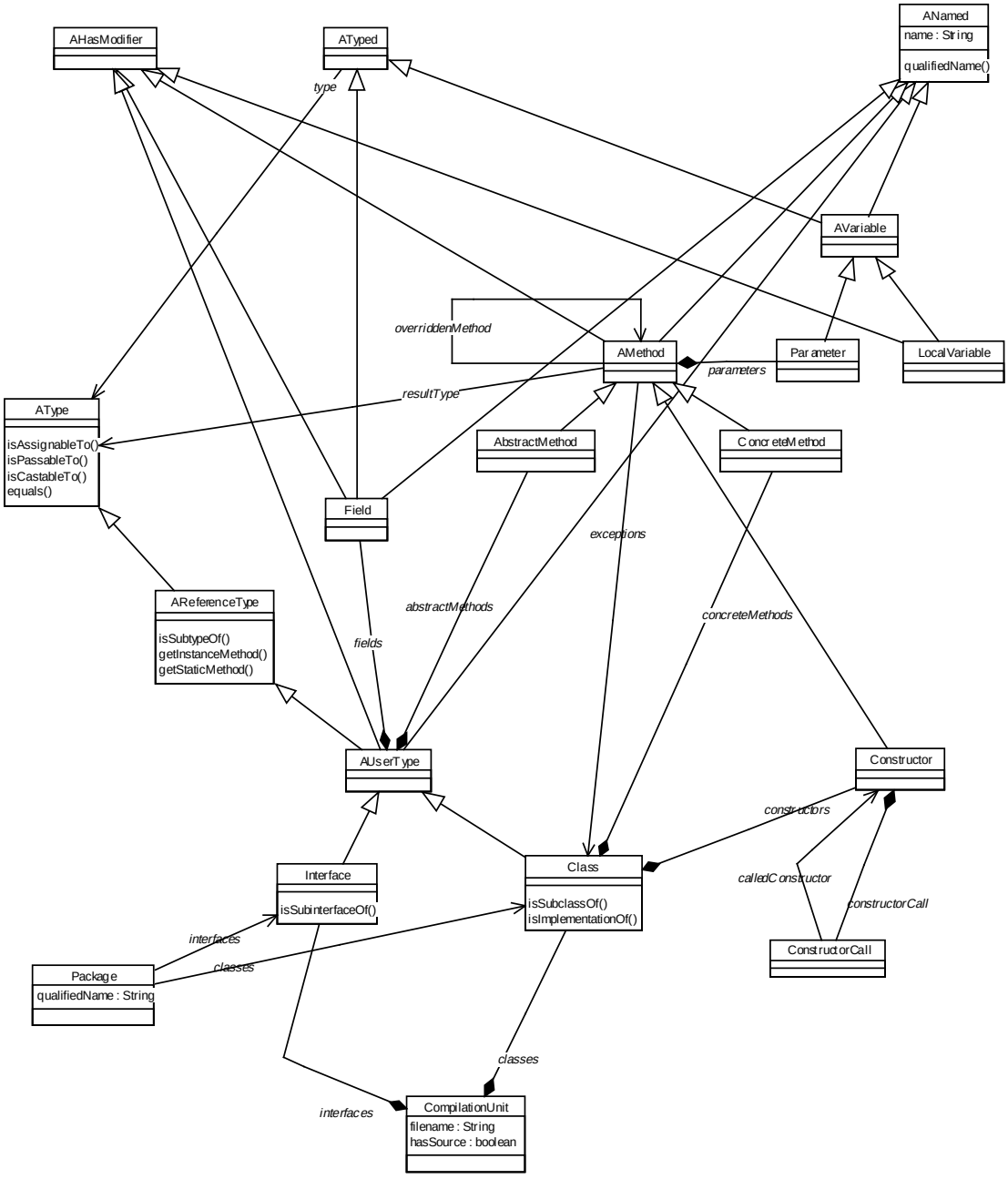


Figure 2 Structure

Representing Expressions

All possible kinds of Java expressions (shown in figure 3) are modeled by the abstract type `AExpression`. The (syntactic) type of any expression can be retrieved by reading the type attribute, using method `type()`, of the expression.

There are two kinds of method calls in Java: calls to static methods and calls to instance methods. The difference between them is that calls to instance methods contain an instance to which the call is actually directed, while calls to static methods don't have an instance – which is why we model the two as separate types, unlike static methods and instance methods (see previous section).

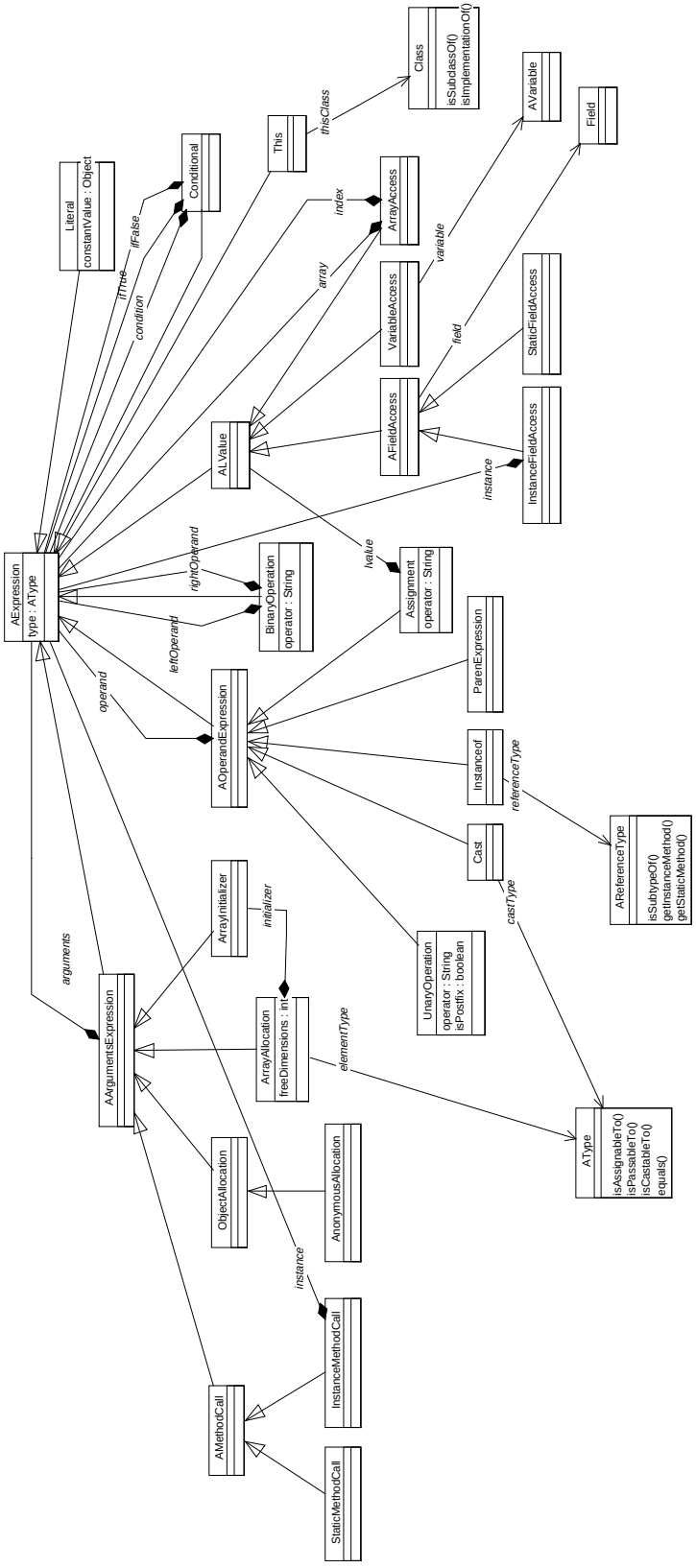


Figure 3 Expressions

As we have already seen for constructors, Barat automatically resolves method calls, though only based on the static types of expressions (for `InstanceMethodCall` objects, it is not generally possible to resolve polymorphism statically, and Barat makes no attempt to do so). The following examples may be helpful to understand Barat's modeling of method calls:

```
public class Target {
     public static void methodA() { ... }; // ConcreteMethod,
                                           // isStatic()==true
     public        void methodB() { ... }; // ConcreteMethod,
                                           // isStatic()==false
}

Target.methodA();           // StaticMethodCall
                            //   calledMethod: Target.methodA()

Target t = new Target();
t.methodB();                // InstanceMethodCall
                            //   instance: t (VariableAccess)
                            //   calledMethod: Target.methodB()

     // inside class Target
     methodB();             // InstanceMethodCall
                            //   instance: This (thisClass = Target)
                            //   calledMethod: Target.methodB()
```

There is a similar distinction between accesses to static fields and instance fields, modeled as `StaticFieldAccess` and `InstanceFieldAccess`. Both are different from objects of type `VariableAccess` (variables are defined locally within methods; formal method parameters are also modeled as variables). To find out whether an access to a variable or field is a read or write access, you need to find out whether it occurs on the left hand side of an assignment (in which case it is a write access) or elsewhere (in which case it is a read). See the paragraph on "Containment" in section 2.1 for an example.

Accesses to array elements are modeled as a combination of `ArrayAccess` objects and field or variable accesses. Consider

```
int a[4] = ...;

a[3] = 17;
```

Here, a[3] is an `ArrayAccess` where the `array` is a `VariableAccess` referring to `a`, and the `index` is a `Literal` object with `constantValue` equal to `3`. Multi-dimensional arrays are modeled likewise:

```
int b[3][4] = ...;

    b[2][3] = 12;
```

Here the left hand side of the assignment is an `ArrayAccess`, where the `index` is the `Literal` "3" and the `array` is an `ArrayAccess`, in which the `array` is a `VariableAccess` and the `index` is the `Literal` "2".

There are three different kinds of object allocations (`new` expressions) in Java, and hence, in Barat: `ObjectAllocation`, `ArrayAllocation`, and `AnonymousAllocation`. An `ObjectAllocation` is an expression such as

```
new Integer (4)
```

The attribute `calledConstructor` in the `ObjectAllocation` object refers to the constructor used for the new object; to find out its class, you therefore write

```
barat.reflect.ObjectAllocation o = ...;
barat.reflect.Class c = o.getCalledConstructor().containingClass();
```

The arguments passed to the constructor can be accessed as a list of `AExpression` objects.

An `ArrayAllocation` is an expression of the form

```
new Integer[4][]
```

As the example suggests, such an allocation may have an arbitrary number of dimensions, starting with a sequence of "definite dimensions", i.e. dimensions for which a length expression is provided (`[4]` in the example), followed by an arbitrary number of "free dimensions", for which there is no length expression ([]). In Barat, the definite dimensions are modeled as a list of argument expressions, while the number of free dimensions is stored in the attribute `freeDimensions`. To get the total number of dimensions of a given array allocation, you could thus write:

```
barat.reflect.ArrayAllocation a = ...;
int dimensions = a.getArguments().size() + a.freeDimensions();
```

The third kind of allocation in Java and Barat is called `AnonymousAllocation`. Here, an object of a given type is allocated, but the type of the object is anonymously extended by code provided as part of the allocation expression. For example:

```
Object o = new Object() {
     public int hashCode() { return 14; }
};
```

Here, an anonymous (implicitly declared) class inheriting from `Object` that overrides `hashCode()` is instantiated. In Barat, this allocation is modeled as a node of type `AnonymousAllocation`, which is a subtype of `ObjectAllocation`. Thus, everything that has been said about `ObjectAllocation` above also applies here, with the additional property that the anonymous extension code is accessible through the accessor method `anonymousClass()` of the `AnonymousAllocation` object.

Representing Statements

Java statements are modeled as subtypes of `barat.reflect.AStatement`, as shown in figure 4. Some less obvious details about statements include:

- A `UserTypeDeclaration` is a declaration of an inner class or interface that occurs inside a method. Java permits this wherever a statement is allowed. An inner class or interface declaration that does not occur inside a method is modeled as a `nestedClass` or `nestedInterface` of the enclosing `AUserType`, see section "Representing Structure".

- Do-, While-, and For-Statements are subtypes of the abstract supertype `ALoopingStatement`. By calling `getExpression()`, the continuation condition can be retrieved. Objects of type `For` support two additional methods: `getForInit()` and `getUpdateExpressions()`.

- In Java, there is no explicit assignment statement, because assignments are themselves expressions and therefore may occur nested within a complex expression. Thus, a top-level assignment is modeled as an `ExpressionStatement` with the expression being an `Assignment` (see section "Representing Expressions").

- A `VariableDeclaration` is a statement that declares a variable, however the actual variable is modeled as a `LocalVariable` contained within the `VariableDeclaration`. A `VariableAccess` (see "Representing Expressions") always refers to that `LocalVariable`, not to the `VariableDeclaration`.

- Also note that short-hand variable declarations such as "`int a, b;`" are "canonicalized" by Barat; thus the above statement would be modeled as two consecutive `VariableDeclaration` statements.

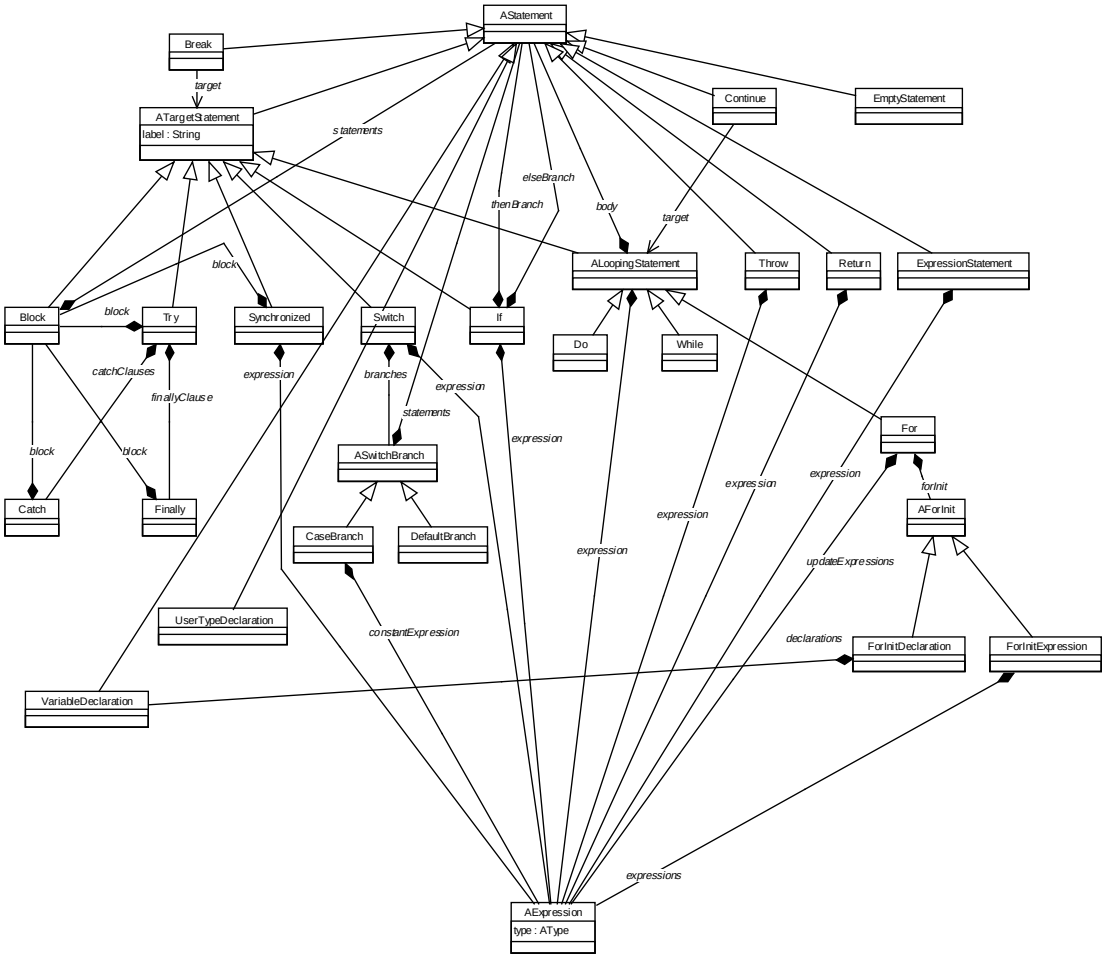


Figure 4 Statements

2.3. Getting the Abstract Syntax Tree of a Java program

The single point of access to the entire Barat system is the class `barat.Barat`, shown in listing 3. You can retrieve the root object of a Barat AST for a given Java class simply by calling

```
barat.reflect.Class c = barat.Barat.getClass ("java.lang.Object");
```

i.e. you only need to call the `getClass()` method with the fully qualified name of the class you are interested in. The entire parsing process and internal analysis is handled by Barat transparently, on demand. At any time, you may simply call the methods of the object that Barat supplied you with to access the internal elements of the class, or other classes it refers to.

```
public class Barat
{
   // runtime flags:
   public static boolean debugLoading = false { ... }
   public static boolean preferByteCode = false { ... }

   // initialization:
   public static void setClassPath(java.lang.String) { ... }

   // accessing AST roots:
   public static barat.reflect.AUserType getUserType(java.lang.String) { ... }
   public static barat.reflect.Class getClass(java.lang.String) { ... }
   public static barat.reflect.Interface getInterface(java.lang.String) { ... }

   // accessing prominent types:
   public static barat.reflect.Class getObjectClass() { ... }
   public static barat.reflect.Class getStringClass() { ... }
   public static barat.reflect.Interface getThrowableInterface() { ... }
   public static barat.reflect.Interface getCloneableInterface() { ... }

   // registering an attribute adder:
   public static void registerAttributeAdder(Visitor adder) { ... }

   // main:
   public static void main(java.lang.String[]) { ... }
}
```

Listing 3 Interface of Barat

You may either use the method `getClass()` or `getInterface()` to analyze a Java class or interface, respectively. If you are not certain whether a given name refers to a class or an interface, use the method `getUserType()`, which returns an object of the common abstract supertype of interfaces and classes. There are also some convenience methods that allow you to access frequently needed Java types: `Object`, `String`, and the interfaces `Throwable` and `Cloneable`.

Just like the tools of the JDK, Barat uses the CLASSPATH environment variable to search for the classes or interfaces you request from it — if the search fails, Barat throws a `RuntimeException`. To use an alternate classpath for analysis, use the method `setClassPath()`.

Usually, if the source code for a given class is found, Barat constructs the abstract syntax tree based on that source code. However, if only a byte code file exists, Barat parses that, although it doesn't disassemble any of the actual instructions in it. Only field and method signatures will therefore be visible for analysis; method bodies are left empty. If for any reason you do prefer loading the byte code even if source is available, set `Barat.preferByteCode` to true.

The method `registerAttributeAdder()` will be explained in section 2.5.

2.4. Working with Visitors

Traversing a Barat structure can be quite complicated if you must write the entire code for such a traversal yourself. Barat therefore provides a framework based on the Visitor design pattern [Gamma et al. 95] that allows you to formulate common analysis algorithms in a much easier way.

The Visitor design pattern lets programmers traverse hierarchical structures of objects in a way where the code that does the traversal is separated from the actions to be performed at each visited object. The pattern is illustrated by the example in figure 5.

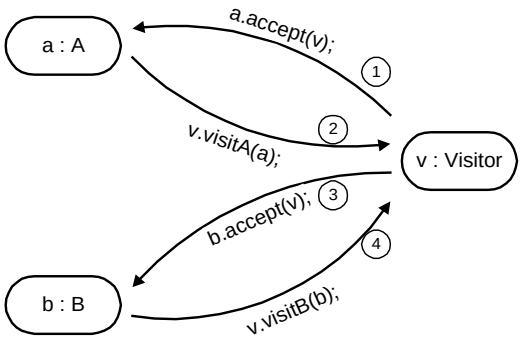


Figure 5 example for visitor pattern

Each of the objects to be visited (having types `A` or `B` in the example) implements a method `void accept (Visitor v)`. To visit an object, the visitor calls this method with itself as the argument. The implementation of `accept` is simply to make a callback to the visitor, however to a method specific for the type being visited. Thus, for class `A`, accept would be implemented as

```
class A {
     ...
     void accept (Visitor v) {
          v.visitA (this);
     }
}
```

The consequence is that the code for visiting `A` objects and `B` objects is bundled in the `Visitor` class, rather than scattered all over the program. We will also see how this pattern allows us to abstract from traversal algorithms in a nice way.

In Barat, all elements of abstract syntax trees (i.e. all `Node` objects) implement an `accept()` method in the way described above. Consequently, there is an interface `barat.Visitor` which declares all the appropriate `visit...` methods:

```
package barat;

public interface Visitor {
     public void visitArrayAccess(ArrayAccess o);
     public void visitArrayAllocation(ArrayAllocation o);
     public void visitAssignment(Assignment o);
     public void visitBinaryOperation(BinaryOperation o);
     ...
}
```

One implementation of `Visitor` provided by Barat is the `DescendingVisitor`. In this class, all visiting methods are implemented so that the constituents of a given class are traversed in a depth-first order. For example, the implementation of `visitClass()` looks roughly like this:

```
public void visitClass (Class o) {
     for (ConstructorIterator i=o.getConstructors().iterator();
                                                  i.hasNext();) {
          i.next().accept (this);
     }
     for (FieldIterator i = o.getFields().iterator();
                                                  i.hasNext();) {
          i.next().accept (this);
     }
     for (ConcreteMethodIterator i = o.getConcreteMethods().iterator();
                                                  i.hasNext();) {
          i.next().accept (this);
     }
}
```

The nice property of the `DescendingVisitor` is that it is guaranteed to traverse all syntactic elements of a given class. To use this for your own analyses, you can subclass `DescendingVisitor`, overriding only those methods where something meaningful should be done. For example, to find out how often a given class refers to `java.lang.System.out`, write:

```
public class MyVisitor extends barat.DescendingVisitor {
     public int result = 0;
     public void visitStaticFieldAccess (StaticFieldAccess o) {
          Field f = o.getField();
          if (f.qualifiedName().equals ("java.lang.System.out"))
               result++;
          super.visitStaticFieldAccess (o);
     }
}
```

Note how the superclass method is called on the last line: this is to make sure the traversal remains a complete one (the sub-nodes of the static field access will be traversed by this call). As static field accesses cannot be nested in Java (i.e., the AST node type `StaticFieldAccess` has no children), this would not strictly be necessary here, but it is always a good idea to follow this convention.

To use the above visitor on a class, write:

```
barat.reflect.Class c = barat.Barat.getClass ("example.MyClass");
MyVisitor v = new MyVisitor();
c.accept (v);
System.out.println ("Result: " + v.result);
```

Another useful class implementing the visitor interface is `DefaultVisitor`, which implements all visit methods by an empty method. This is useful for cases in which only some of the AST node types need to be considered. Subclassing `DefaultVisitor` and overriding only some methods yields a visitor class that acts like a switch statement that switches over the visited object's actual type, as in:

```
ALoopingStatement s = ...;
s.accept(new DefaultVisitor() {
     public void visitDo(Do d) {
          // ...
     }
     public void visitFor(For f) {
          // ...
     }
     public void visitWhile(While w) {
          // ...
     }
});
```

In this example, an anonymous inner class is used as a visitor. The visitor object does not traverse the tree, it is used only once to distinguish between certain possible actual types of a node object. Using `DefaultVisitor` avoids using a nested if and explicit downcasts, and is more efficient when the number of cases is large, as can be seen when comparing it to the more conventional code below:

```
ALoopingStatement s = ...;
if(s instanceof Do) {
     Do d = (Do)s;
     // ...
}
else if(s instanceof For) {
     For f = (For)s;
     // ...
}
else if(s instanceof While) {
     While w = (While)s;
     // ...
}
```

A variant of `DefaultVisitor`, called `AbstractingVisitor`, provides even more flexibility. `AbstractingVisitor` defines additional visit methods for abstract interfaces (such as `AMethodCall`, `AFieldAccess`, `AExpression`). In `AbstractingVisitor`, each visit method for a type `T` has a default implementation that calls the visit method(s) for `T`'s supertype(s). (If there is more than one supertype, one or more of `ANamed`, `ATyped`, or `AHasModifier` are involved, see figure 2.) In the following example, abstract interface types may be separate cases as well:

```
AExpression e = ...;
e.accept(new AbstractingVisitor() {
     public void visitInstanceMethodCall(InstanceMethodCall o) {
          System.out.print("instance ");
          super.visitInstanceMethodCall(o);
     }
     public void visitStaticMethodCall(StaticMethodCall o) {
          System.out.print("static ");
          super.visitStaticMethodCall(o);
     }
     public void visitAMethodCall(Cast o) {
          System.out.println("method call");
     }
     public void visitAExpression(AExpression o) {
          System.out.println("not a call expression");
     }
});
```

It is instructive to compare this example to a hypothetical switch statement that selects cases based on actual node types:

- Calling visit methods of super is similar to a fall-through (omitting `break`) in a case branch. However, note that *both* `visitInstanceMethodCall` and `visitStaticMethodCall` "fall through" to `visitAMethodCall`, which would not be possible in a switch statement. Note that unlike in switch statements with fall-through, the order of "cases" does not matter in our example.

- Visit methods for more abstract types are like default branches in switch statements, but there may be several levels of defaults.

Another useful visitor provided by Barat is the `OutputVisitor`. It is similar to the `DescendingVisitor` in that it traverses an entire user type in natural order, however it also prints

that type's source code to an arbitrary file (effectively re-generating the source code). By subclassing `OutputVisitor` and overriding certain methods of it, all sorts of source code modifications and transformations can be implemented. As an example, see the `InstrumentingVisitor` in section 4.2.

It must be noted, though, that the `OutputVisitor` *re-generates* a classes' source code based on the information in the Barat AST. This newly generated source code is guaranteed to be semantically equivalent to the original source code, but it is mildly canonicalized with respect to formatting and some other issues, such as the order of declarations inside a class. Also, all comments in the original code are lost.

2.5. Working with Attributes

For some purposes that involve a non-standard traversal of the AST, visitors may not be adequate. By means of an example, we will explain user-defined node attributes, an alternative way of structuring traversals of the AST.

In Barat, the attribute concept allows to store user-defined data for each node, and to cache data that is calculated automatically on-demand. Rather than storing user-defined data directly, so-called *attribute objects* that return user-defined data objects can be stored for each node object. Attribute objects are instances of classes that implement the interface `AbstractAttribute`:

```
public interface AbstractAttribute {
   public Object objectValue();
}
```

Listing 4 interface `AbstractAttribute`

Objects of type `AbstractAttribute` can be stored in a node object `n` by calling `n.addAttribute(k, a)`, where `k` is a key object and `a` is an attribute. By calling, on the same node object, the method `attributeValue(k)`, providing the key object `k`, the result of calling `objectValue()` on the stored attribute object will be returned.

Two implementations of `AbstractAttribute` are already provided: The first, called `ConstantAttribute`, can be used for storing a constant value as an attribute. For storing an object o in node n under key k, use:

```
n.addAttribute(k, new ConstantAttribute(o));
```

The stored value o can be retrieved using:

```
n.attributeValue(k);
```

The second implementation, `CachedAttribute`, can be used for on-demand calculated attributes whose values will be cached once they have been calculated. Cached attributes are usually added to newly created AST nodes by registering a Visitor (usually, a subclass of `DefaultVisitor` or `AbstractingVisitor`) using `Barat.registerAttributeAdder()`.

Assume that you want to compute, for a number of classes, the set of interfaces implemented by each class. It is relatively straightforward to write a recursive algorithm for computing the set of interfaces for one class. However, if the result of this calculation is to be used several times, e.g. for computing the set of interfaces that are implemented by subclasses of the current class, it will be desirable to maintain a cache of already computed sets.

Using attributes, we can write a concise solution to this problem:

```
final Object implementors = new Object(); // used as key
Barat.registerAttributeAdder(new DefaultVisitor() {
     public void visitClass(final Class c) {
          c.addAttribute(implementors,
                         new CachedAttribute() {
            protected Object calculate() {
               Set result = new HashSet();
               for(InterfaceIterator i=c.getImplementedInterfaces();
                  i.hasNext();) {
                  result.addAll(
                    (Collection)i.next().attributeValue(implementors));
               }
               if(c!=Barat.getObjectClass()) result.addAll(
                  (Set)c.getSuperclass().attributeValue(implementors));
               return result;
            }
          });
     }
     public void visitInterface(final Interface c) {
          c.addAttribute(implementors,
                         new CachedAttribute() {
            protected Object calculate() {
               Set result = new HashSet();
               for(InterfaceIterator i=c.getExtendedInterfaces();
                  i.hasNext();) {
                  result.addAll(
                    (Set)i.next().attributeValue(implementors));
               }
               return result;
            }
          });
     }
});
```

By registering an attribute adder visitor, visit methods will be called for every newly created AST node object. As these node objects are not yet properly inserted into the AST, the only method that can safely be called on them is `addAttribute()`. The added attribute's code, however, can invoke arbitrary methods on the node, because it will be called only if the attribute's value is to be computed, which can only happen after proper initialization. Note that by creating anonymous attribute classes that inherit from `CachedAttribute`, the attribute's code will be called only once per AST node object.

To get the value of an attribute as defined in this example, you can use the following code:

```
AUserType ut = ...;
Set s = (Set)ut.attributeValue(implementors);
```

Note that calls of `attributeValue(implementors)` occur during calculation of the attributes' values due to their recursive definition. Of course, there should be no cycles in recursive definitions of attribute calculations. From our experience, cycles normally do not occur; however, if they do, the class `CachedAttribute` will detect this at runtime.

3. Implementation of Barat

Barat's public interface consists of the three packages `barat`, `barat.reflect`, and `barat.collections`. The fourth package, `barat.parser`, contains the implementation that is normally hidden from users of Barat: There is an explicit distinction between interface and imple-

mentation parts of AST node types. For each AST node type, there is a public interface in `barat.reflect` and a class implementing the interface in package `barat.parser`. The names of implementation classes are derived from the names of the implemented interfaces and end with "`Impl`". Implementation details are not exposed by Barat's public interface: The interfaces in `barat.reflect` contain read-only accessor methods with parameter and return types that reference only other interfaces in `barat.reflect`.

Package `barat.parser` also contains the actual parser, which is generated by JavaCC (version 0.7.1) from the grammar file `BaratParser.jj`. This file consists of a BNF-based grammar from which a scanner is derived for transforming the input file into a sequence of tokens, and of a LL(k) grammar augmented with tree-building Java code that specifies Java's syntax based on the tokens. Class files are parsed by `ClassFileParser`; name analysis is defined in class `NameAnalysis` and type analysis in class `TypeAnalysis`.

For certain advanced uses, accessing Barat using the public interface – where the internals of Barat are hidden – may not be sufficient. For example, changing the syntax of the parsed language would require changing the implementation of the parser part of Barat, or parsing comments other than tag comments would require changing the implementation of the scanner part of Barat

This section, which assumes familiarity with basic compiler construction techniques, guides the reader through the inner workings of Barat, and points out areas where changes might be needed for certain advanced uses of Barat.

Section 3.1 explains how Barat is implemented. Section 3.2 describes hooks that are provided for white-box users of Barat, making it possible to adapt Barat without changing its implementation. Section 3.3 lists possibilities for changing Barat's implementation.

3.1. Implementation of Barat

A first version of Barat was designed using a conventional architecture for parsers: After building an explicit abstract syntax tree, name and type analysis were performed by several passes, each of which was defined as a traversal of the abstract syntax tree. Experiences with this first version showed two main drawbacks of the chosen architecture:

- It turned out that name and type analysis for Java is a non-trivial problem that cannot easily be divided into a small number of passes (we ended up with six passes: registering names, resolving type names, establishing inheritance links, building lists of all methods per class/interface, resolving remaining names, and type analysis). Moreover, each of the required passes had to produce complex intermediate results which were then used as input to other passes. This lead to a situation where debugging and testing became extremely difficult: Often, a bug in one of the passes manifested itself in a later pass, when the incorrect intermediate result was being used.

- During name and type analysis for Java source files, other source files need to be parsed and partly analyzed on demand. Because it is difficult to predict in advance how much analysis is needed for these other source files, we had to maintain information about each source file's parsing and analysis status, and we needed a complex recursive algorithm that triggered parsing and different analysis passes based on that information. Because the algorithm at certain points made conservative decisions about which files needed to be parsed, the number of files that were parsed starting from a certain file was much greater than would have been needed for name and type analysis of the first file. Worse still, because Barat is used as the basis for other analyses, it is not possible to tell to what extent name and type analysis is needed for other source files. Because client code should not be concerned with problems of how much name and type analysis has been performed already on needed source files, we decided to parse all source files that are transitively referenced from the starting source file, and to perform full name and type analysis on all those files. This lead to enor-

mous startup times (five to ten minutes) for Barat, before any user-defined static analysis could proceed.

Due to the performance and stability problems we encountered with the first version, we decided to redesign Barat, supporting on-demand parsing, on-demand name analysis, and on-demand type analysis. Central to the new architecture of Barat are lazily-evaluated attributes of AST node objects similar to attributes as known from attribute grammars [Knuth 68; Kastens, Waite 94]. In some cases, attributes are sometimes used to calculate parts of the AST itself, like in higher order attribute grammars [Vogt et al. 89]. We also chose to use parameterized types – at least internally – to gain more type safety when building the AST, and to support type-safe attribute objects.

In attribute grammars, attributes can be defined for each terminal or nonterminal of a grammar, where an attribute's value may depend on values of other attributes of possibly different terminals or nonterminals. As parsers are usually used for transforming a sentence of an input grammar into a sentence of an output grammar, in the ideal case, a complete parser could be generated from an attribute grammar, specifying the output of the parser as the value of a distinguished top-level attribute. There are systems for automatically generating efficient parsers based on attribute grammars, which usually avoid generating an explicit abstract syntax tree with explicit attributes at the nodes of the tree – both attribute values and parts of the abstract syntax tree are stored only if they will be needed to calculate the value of other attributes.

Because Barat should support arbitrary static analyses on Java source code, there is no "main" or "top-level" attribute as in an attribute grammar. Thus, an explicit abstract syntax tree is still built, and attributes are used as a means for structuring name and type analysis in a declarative way.

Attributes have been described already in section 2.5. In the implementation part of Barat, we use type-parameterized classes for attributes that allow to access an attribute's value without downcasts. Attributes in package `barat.parser` are instances of subclasses of the generic abstract class `Attribute<A>`, defined in package `barat.parser`, with two methods: the abstract method `calculate()` must be implemented in subclasses of `Attribute<A>` to return a value of type `A`, and the public final method `value()` should be called to retrieve the value of an attribute. The implementation of `value()` always performs caching: it calls `calculate()` only if there is no cached value yet. For compatibility with the attribute classes defined in package `barat`, `Attribute<A>` implements the interface `barat.AbstractAttribute` by returning the attribute's value as a value of type java.lang.Object. However, as this way of accessing the attribute's value does not retain type information and thus would require downcasts, the generic method `value()` returning an object of type `A` is used instead.

There are two subclasses of `Attribute<A>`: `Constant<A>` is used for constant values that need to be wrapped in an attribute, and `CastingAttribute<A,B>` is necessary for some cases where a typecast on the level of attributes is needed. `CastingAttribute<A,B>` inherits from `Attribute<B>` and expects in its constructor an attribute of type `Attribute<A>`. Its calculate method calls `value()` on this attribute, yielding a value of type `A`, and then casts this value to type `B` and returns it.

As attribute objects should be invisible for users of Barat, the accessor methods defined in interfaces in package `barat.reflect` return values rather than attribute objects, i.e., on calling an accessor method, the underlying attribute's `value()` method will be called. For example, in interface `StaticFieldAccess`, an accessor method `getField()` is defined that returns the AST node object for the accessed field's declaration. Clearly, this involves name analysis, and thus, the implementation of `getField()` in class `StaticFieldAccessImpl` returns the result of calling `value()` on the corresponding attribute.

Rather than being separate objects, the desired lazy evaluation and caching of attribute values could be achieved by implementing the caching scheme in each of the accessor methods explicitly. We chose

the first alternative for two reasons: First, it factors out common code that manages caching, so that for example provisions for detecting cyclic dependencies between attributes are handled in one class rather than in each accessor method. Second, and more important, it allows to separate the calculation code for attributes from the classes that have attributes. Similar to the visitor pattern, this allows attribute calculation code to be collected in separate classes; for example, all attribute calculations that implement name analysis are contained in a single class `barat.parser.NameAnalysis`.

We now sketch the steps that are performed when a user of Barat calls the top-level method `barat.Barat.getClass(qn)`. The parameter `qn`, of type `String`, is the fully qualified name of the class that will be returned by the call.

1. The implementation in class `Barat` delegates the call to class `barat.parser.NameAnalysis`, converting the passed string to an object of class `barat.QualifiedName`, which allows iterating over a qualified name's components and easy access to the base and qualifier parts of a qualified name.

2. The called method in class `NameAnalysis` iterates over the qualified name, maintaining a prefix qualified name (initially empty), a current simple name, and the remaining qualified name. For each prefix, it retrieves an object representing the package with the prefix name. (In the case of the empty prefix, this is the global, unnamed package.) It then tries to find a class or interface with the current simple name in that package. If it finds such a class or interface, the remaining qualified name must denote an inner class of this class or interface. Otherwise, if no class or interface is found, the next iteration is performed by appending the current simple name to the prefix and fetching the first simple name of the remaining qualified name.

3. To retrieve an object representing a package, an internally maintained table is searched. If such an object does not yet exist, it is created and inserted into the internal table. Thus, for each qualified name, there is a single unique package object, allowing to compare package objects by identity comparison (using '`==`' rather than `equals()`)

4. To search for a class or interface within a package, the list of already loaded classes and interfaces of that package is searched. If no class or interface is found, the classpath is searched for a Java source file or a Java class file in a directory with the package's name. Whether Java source files or class files are considered first can be determined by the property "`barat.preferByteCode`", which is false by default, but may be set to true by either setting `barat.Barat.preferByteCode` to `true` or by the command line switch "`–Dbarat.preferByteCode`". There is a second property called `barat.debugLoading` which, when set to true (the default is false), causes messages to be printed to `System.out` whenever a Java source file or class file is read.

5. Parsing of Java source files is performed by `barat.parser.BaratParser`, a parser class generated by JavaCC [JavaCC] (version 0.7.1), based on the Java 1.1 grammar distributed with JavaCC. The grammar input file, called `BaratParser.jj`, contains code for creating the abstract syntax tree for a given compilation unit. Since the abstract syntax tree generated by the parser should be fully typed and be based on names that correspond to the Java language specification rather than generated names like `f0`, `f1`, ... , we did not use parse tree generator tools [JJTree, JTB].

6. Parsing of class files is performed using JavaClass [Dahm 98], a package for reading and writing byte-code files. The class `barat.parser.ClassFileParser` is responsible for transforming JavaClass's internal representation of byte-code files into AST node objects of Barat.

For implementing Barat, we have used type-parameterized versions of the new (JDK 1.2) collection classes. In Barat, these come in two flavors: In package `barat.collections`, we have placed source-level instantiations of such classes, in order to keep things simple for normal users of Barat.

The package `barat.parser`, which contains the implementation part of Barat, makes use of a version of the collection classes modified to work with Poor Man's Genericity, an extension of Java that supports parameterized types by automatically generating byte-code level instantiations of generic classes.

Most of the classes in `barat.reflect` and `barat.parser` have been generated from a UML class diagram using a custom-made code generator. (A copy of the diagram is provided in the appendix.) In fact, everything about package `barat.reflect` is obvious from the class diagram, so that it is usually easier to look in the class diagram for names of methods rather than in the source files. Note that, at least in package `barat.parser`, there are some methods and constructors that have been generated for the sake of completeness, although they are not needed by the current implementation.

3.2. Hooks for Adapting Barat

Factory

In both the source file parser and the class file parser, AST node objects are never created directly using the `new` operator; rather, a factory object is used, which can be set using the static method `barat.parser.Factory.setInstance()`. For this purpose, the default Factory class `barat.parser.Factory` should be subclassed. Note that in the current implementation, only one (singleton) factory instance is supported. Because the factory object stores a list of package objects that have been created so far, changing the factory object starts a new "session", and all previously accessed Barat objects should be discarded. Thus, in a program that wishes to create objects differently, setting the appropriate factory object should be the first statement.

When overriding factory methods in subclasses of `barat.parser.Factory`, care should be taken not to break assumptions made by other parts of Barat. The most important assumption is that name and type analysis relies on attributes to be set right after object creation; thus, if an overridden factory method does not call the overridden method, it should call static methods in both `NameAnalysis` and `TypeAnalysis` analogously to the original implementation to have the required attributes created. Furthermore, user-defined attributes that can be set by registering a callback with the factory will not be set if the callbacks are not performed by an overriding factory method.

Even more care is needed if the new factory allocates objects of classes which implement the required interfaces of `barat.reflect`, but do not inherit from the default implementation classes in `barat.parser`, as the current implementation in some methods relies on downcasts to implementation classes.

Notification on Parsed Files

Because loading and parsing of Java source code is performed on-demand only, some clients of Barat (e.g., a code generator) might be interested in being notified whenever a file has been parsed. For this purpose, the class `barat.parser.NameAnalysis` provides a method for registering a callback. This method is called `addSourceParsedObserver()`. It has one argument of type `barat.parser.SourceParsedObserver`, an interface with the callback method `sourceParsed()` (see listing 5). As an argument to `sourceParsed()`, Barat provides the AST node object for the newly parsed compilation unit.

```
public interface SourceParsedObserver
{
   public void sourceParsed(CompilationUnit cu);
}
```

Listing 5 interface `SourceParsedObserver`

3.3. Possible Changes and Extensions

As with any framework, there are many possibilities for improving Barat. In this section, we list some of the possibilities we already have thought about, and try to assess how much effort would be needed for the improvements. We have classified possible improvements into the categories changes, extensions, and optimizations.

Grammar Changes

For experimenting with Java language extensions or derivations, Barat currently only supports parsing of special comments that can be used to tag classes, interfaces, methods, and declarations of fields, parameters, and local variables. Certainly, for some applications, this is not sufficient, and changing the grammar that Barat uses would be desirable. However, the generated parser cannot easily be substituted by a user-defined parser, and changing the syntax often will cause changes in the structure of the ASTs as well. Thus, if you change the grammar, you will create a different branch in the development history of Barat, and it will be difficult to incorporate bug-fixes and improvements of Barat itself in the changed version of Barat. After these necessary warnings have been said, this is how you would proceed: The parser within Barat has been generated using the parser generator JavaCC 0.7.1, and the input file for the generator is called BaratParser.jj in directory barat/parser. The grammar file also contains the scanner definition; e.g., comments are handled by the scanner. If you change the grammar part itself, you will probably need different AST node types which should then be introduced in `barat.reflect` and implemented in `barat.parser`.

Allowing Write Access to the AST

Barat's on-demand name and type analysis has been designed with the assumption that the parsed program will not change over time. Thus, the attributes used for name and type analysis will only be evaluated once. If changing the AST were allowed, re-evaluation would be required for those attributes which relied on parts of the AST that have changed. Such a re-evaluation could be incorporated into the implementation, by causing an invalidation and eventual re-evaluation of affected attributes. This would be possible if, during evaluation of one attribute, accesses to other attributes would be logged, so that if one of these attributes was invalidated, the dependent attribute could be invalidated as well. This behavior could be implemented globally in class `Attribute`. Unfortunately, this simple modification would allow only certain kinds of changes, because some references in the AST are stored directly rather than within an attribute. Currently, only references to expression nodes and type nodes are consistently wrapped by an attribute, because these are results of name and type analysis. If all computed references were wrapped in attributes, including all calculations a user of Barat has made, this scheme would work in the general case as well.

Extensions

Whenever a class has been parsed from a class file instead of a Java source file, method bodies of concrete methods are empty blocks, although more information could be recovered from the byte code. As has been shown by several decompilers for Java (e.g., [Proebsting, Watterson 97]), recovering Java source code from byte code is possible. Integrating a decompiler in Barat would be an attractive extension, as it would enable normal Barat-style static analyses on byte code classes as well. This would

allow to use Barat not only at compile-time, but also at load time, e.g. for checking security-related constraints.

Optimizations

So far, we have focused on the functionality aspect of Barat. The current version has been in use by several persons that use Barat for different purposes each. However, as we have neither optimized performance nor space requirements, Barat tends to be slow and sometimes needs a larger heap than provided by the JVM by default. From our experience, there are two areas of possible optimizations that would be relatively easy to implement:

1. Currently, as soon as an object of type `ClassImpl` or `InterfaceImpl` is created, the corresponding source file or class file is parsed. It turned out that some classes and interfaces are parsed unnecessarily: During type analysis, when method lookup is performed, simple identity comparisons between method argument types could be performed before actually parsing the corresponding classes or interfaces. Only if subtype relationships have to be considered, parsing needs to be performed. We hope that by decoupling creation of class or interface objects from the actual parsing, a significant number of classes and interfaces would not need to be parsed.

2. As Barat does not support changing the AST once it has been created, all attributes that are used for name and type analysis are only computed once, and from then on, the calculated result is cached. However, the attribute object used for calculating this result is still referenced, and so are all intermediate attributes or objects that are needed only for this calculation. Space requirements of Barat could be reduced significantly if references to already computed attribute objects were deleted and the resulting garbage objects were collected by the garbage collector.

4. Examples

In this section, we present two complete examples of using Barat. In the first example, a simple static analysis task is performed, while in the second example, Barat is used to instrument Java source code.

4.1. Immutable Classes

As a first example, we want to determine whether a given Java class is immutable, i.e. whether the state of objects of this class cannot be changed after creation. We define this a little more rigorously as follows:

- For an immutable class, we require all fields to be private. Hence, only the methods of the class itself could possibly alter the state.

- We allow the fields to be written in constructors, or by field initializers, but not in ordinary methods.

- For simplicity, we do not check for write accesses into arrays – that is, we regard arrays as separate objects: if an immutable object has an instance variable that is an array, the object is allowed to change the array's elements, but it may not replace the entire array.

- Also for simplicity, we do not consider inheritance. In practice, objects could only be immutable if their superclasses were also immutable (transitively).

Given this definition, it is straightforward to write a Visitor that checks for immutability:

```
public class ImmutabilityVisitor extends barat.DescendingVisitor {

   public boolean isImmutable = true;

   public void visitConstructor (Constructor o) {
        // Do nothing, i.e. don't check for write accesses in constructors.
   }

   public void visitField (Field o) {
        if (!o.isPrivate())
             isImmutable = false;
        super.visitField (o);
   }

   private boolean isWriteAccess (AFieldAccess o) {
        return (o.container() instanceof Assignment)
                    && (o.aspect().equals ("lvalue"));
   }

   public void visitInstanceFieldAccess (InstanceFieldAccess o) {
        if (o.getField().containingClass() == o.containingClass())
             if (isWriteAccess (o))
                  isImmutable = false;
        super.visitInstanceFieldAccess (o);
   }

   public void visitStaticFieldAccess (StaticFieldAccess o) {
        if (o.getField().containingClass() == o.containingClass())
             if (isWriteAccess (o))
                  isImmutable = false;
        super.visitStaticFieldAccess (o);
   }
}
```

Listing 6 class `ImmutabilityVisitor`

The Visitor inherits from the DescendingVisitor, i.e. it is guaranteed to traverse the entire class on which it is started. The analysis result is stored in the field `isImmutable`, initialized to `true`. If any violation of the immutability condition is found, that variable is set to `false`.

The method `visitConstructor()` does *not* contain a call to `super.visitConstructor()`, which means that the body of constructors is skipped by the analysis (writing to fields is allowed in constructors). The method `visitField()` makes sure that all fields of the class are private, while `visitInstanceFieldAccess()` and `visitStaticFieldAccess()` ensure that fields are only read, but not written (remember that field accesses inside constructors are not seen by these methods). To distinguish a read access from a write access, the visitor checks (in method `isWriteAccess()`) whether the access occurs on the left hand side of an assignment.

To use this visitor on a class, write

```
barat.reflect.Class c = barat.Barat.getClass ("java.lang.String");
ImmutabilityVisitor v = new ImmutabilityVisitor();
c.accept (v);
if (v.isImmutable)
     System.out.println ("java.lang.String is immutable");
else
     System.out.println ("java.lang.String is not immutable");
```

4.2. Logging method calls

This example shows how Barat can be used to instrument a Java program. Assume you want to log method calls, and you have written a runtime support class `log.Log` which implements two methods `enterMethod()` and `exitMethod()`. These methods should be called on each method entry and method exit, respectively. Now, although the AST built by Barat cannot be changed, there is a way to generate programs that are instrumented with calls to `enterMethod()` and `exitMethod()`: We can write a subclass of `OutputVisitor` that, whenever a method body is written, inserts the appropriate calls.

Before we describe our solution, we examine what exactly needs to be inserted into method bodies: It is easy to make sure that the call to `enterMethod()` is performed by inserting a method call statement before the first original statement of the method's body. But where should we insert the call to `exitMethod()`? Things are more complicated with method exits because the method may return in the middle of its body, using a `return` statement, and it may exit by throwing an exception. Fortunately, we can make use of the `try` ... `finally` construct which guarantees that the `finally` block is executed after the `try` block, regardless of how the `try` block exits.

An obvious solution would be to override `visitConcreteMethod()` in our subclass of `OutputVisitor`, called `InstrumentingVisitor`. This, however, would require that we copy the original code for `visitConcreteMethod()` into our subclass. But there is a better solution: By overriding `visitBlock()` instead, and checking whether the block is directly contained within a `ConcreteMethod` object, we can reuse both the original implementation of `visitConcreteMethod()` and of `visitBlock()`: If the visited block is not directly contained in a `ConcreteMethod` object, we call the superclass implementation of `visitBlock()`. Otherwise, the visited Block is a method body and needs to be instrumented. Thus, we first print the call to `enterMethod()`. After printing the keyword `try`, we call the superclass implementation of `visitBlock()`, which will output the original method body. Then, we print the `finally` clause that contains the call to `exitMethod()`. In listing 7, the complete source code for `InstrumentingVisitor` is shown.

To use `InstrumentingVisitor` on a class, write:

```
barat.reflect.Class c = barat.Barat.getClass ("example.MyClass");
InstrumentingVisitor v = new InstrumentingVisitor();
c.accept (v);
```

For example, applying `InstrumentingVisitor` to itself would print the following code for the method `visitBlock()` (some lines of the output have been omitted):

```
public void visitBlock(Block o) {
     log.Log.enterMethod("barat.test.InstrumentingVisitor.visitBlock");
     try {
          if(o.container() // ... etc
          // ... etc
     }
     finally {
          log.Log.exitMethod(
                    "barat.test.InstrumentingVisitor.visitBlock");
     }
}
```

```
package barat.test;

import barat.reflect.*;
import barat.*;

public class InstrumentingVisitor extends OutputVisitor
{
   public void visitBlock(Block o)
   {
         if(o.container() instanceof ConcreteMethod)
         {
               String methodName = ((ConcreteMethod)o.container())
                                                      .qualifiedName();
               println("{");
               currentIndent++;
               indent();
               println("log.Log.enterMethod(\"" + methodName + "\");");
               indent();
               print("try ");
               super.visitBlock(o);
               nl();
               indent();
               println("finally {");
               indent();
               println("\tlog.Log.exitMethod(\"" + methodName + "\");");
               indent();
               println("}");
               currentIndent--;
               indent();
               println("}");
         }
         else
         {
               super.visitBlock(o);
         }
   }
}
```

Listing 7 class `InstrumentingVisitor`

References

[Bokowski, Dahm 98] B. Bokowski, M. Dahm, *Poor Man's Genericity for Java*, Proceedings of JIT'98, Springer Verlag, 1998

[Dahm 98] M. Dahm, *Byte Code Engineering with the JavaClass API*, Freie Universität Berlin, Institut für Informatik, Technical Report B-98-17

[Gamma et al. 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns - Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995

[Gosling et al. 96] J. Gosling, B. Joy, G. Steele, *The Java Language specification*, Addison-Wesley, 1996

[JavaCC] JavaCC, the Java Parser Generator, Sun Microsystems Inc., http://www.suntest.com/JavaCC/

[JJTree] JJTree, Sun Microsystems Inc., http://www.suntest.com/JavaCC/DOC/JJTree.html

[JTB] K. Tao, J. Palsberg, The Java Tree Builder Homepage, Purdue University, http://www.cs.purdue.edu/homes/taokr/jtb/index.html

[Kastens, Waite 94] U. Kastens, W. M. Waite, *Modularity and Reusability in Attribute Grammars*, Acta Informatica, Vol. 31, pp. 601-627, 1994.

[Knuth 68] D. E. Knuth, *Semantics of Context-Free Languages*, Mathematical Systems Theory, 2(2):127-145, June 1968

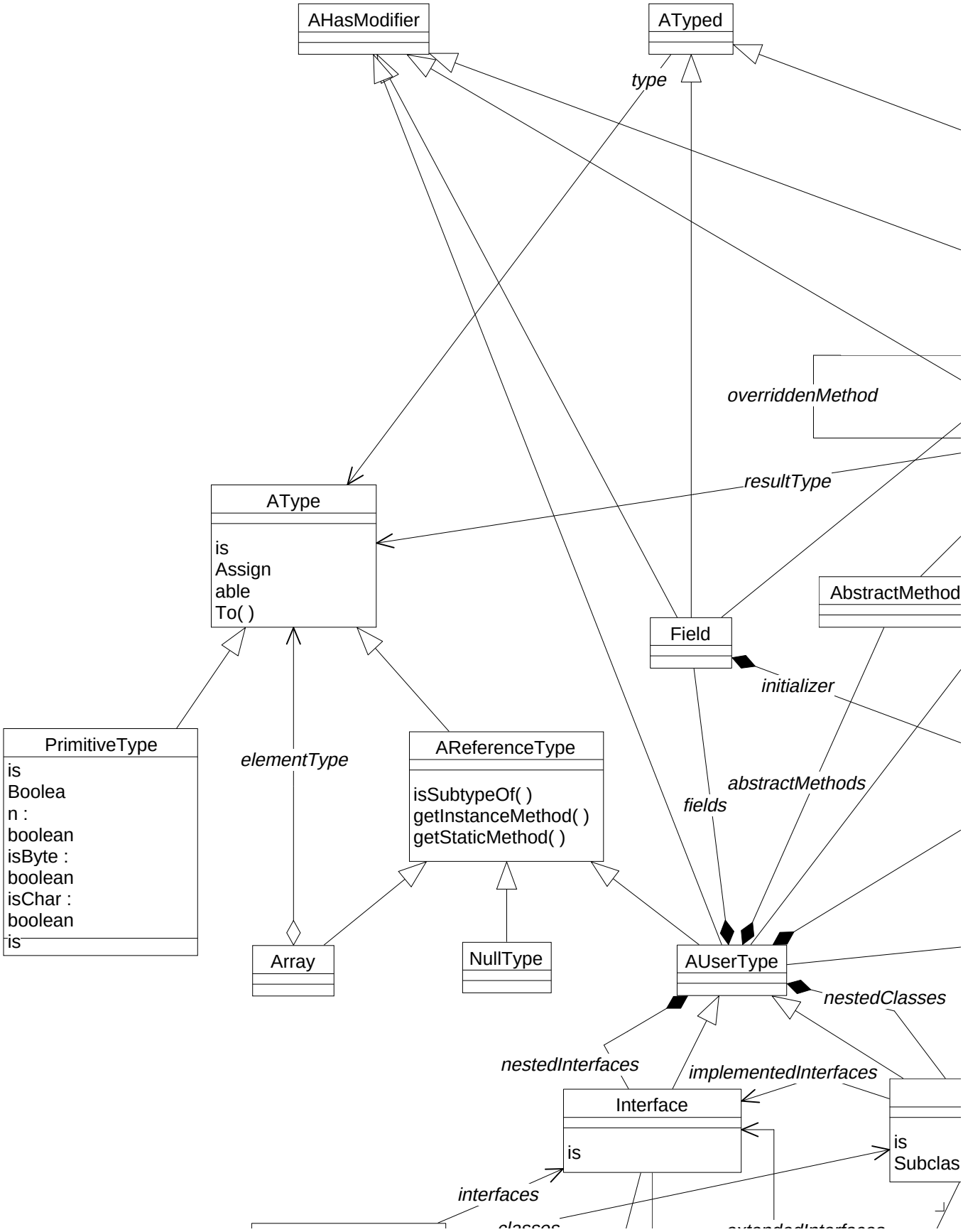
[OpenJava] M. Tatsubori, S. Chiba, OpenJava WWW Page, http://www.softlab.is.tsukuba.ac.jp/~mich/openjava/

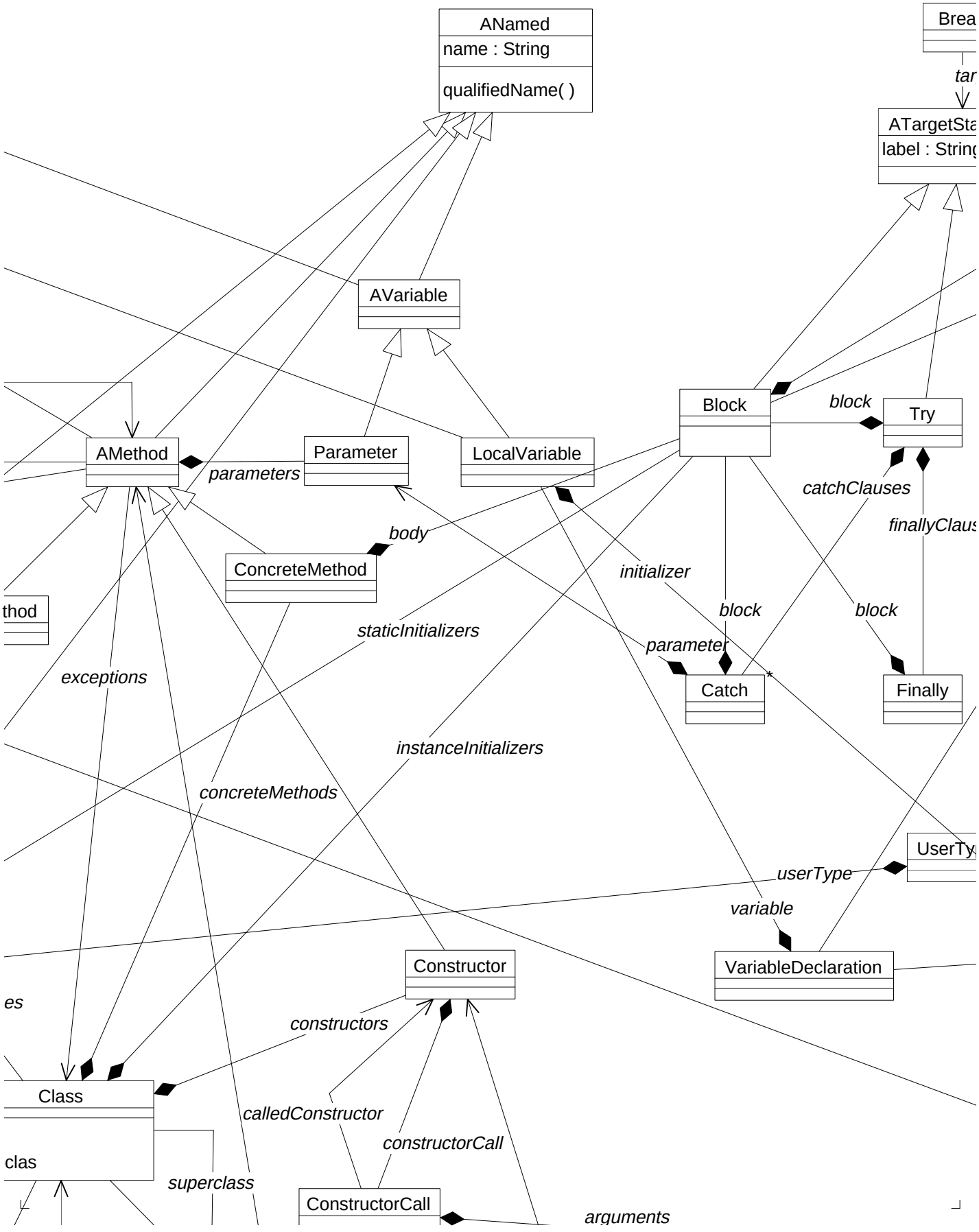
[Proebsting, Watterson 98] T.A. Proebsting, S.A. Watterson, *Krakatoa – Decompilation in Java*, Proceedings of COOTS'97, Usenix, 1997

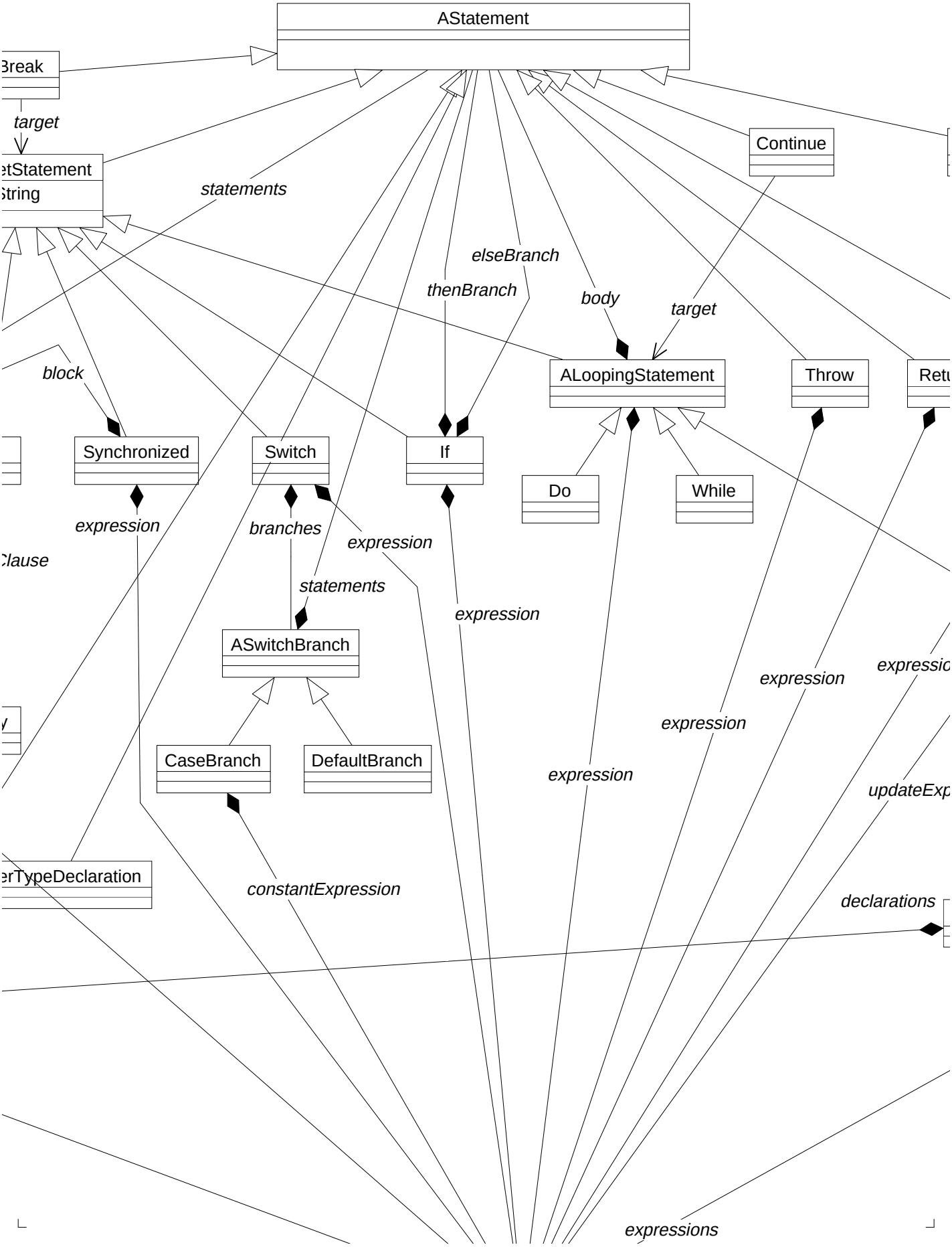
[Vogt et al. 89] H.H. Vogt, S.D. Swierstra, M.F. Kuiper, *Higher-order Attribute Grammars*, Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, ACM SIGPLAN Notices, 24(7), June 1989

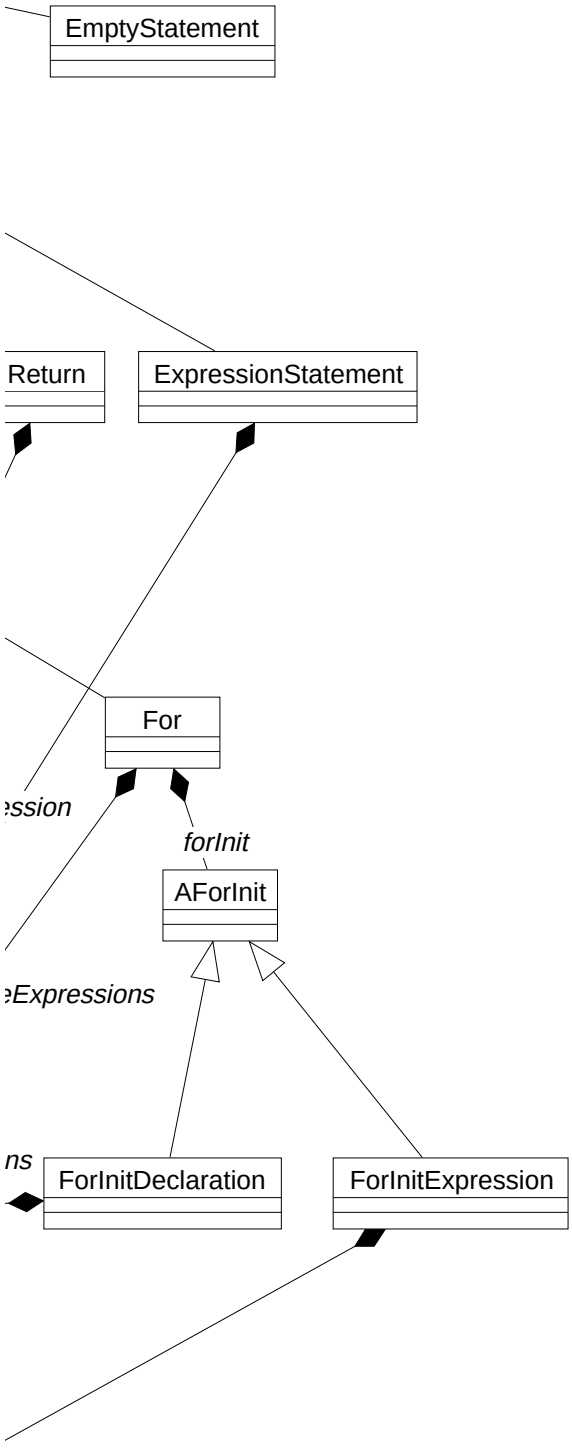
APPENDIX: class diagram for barat.reflect

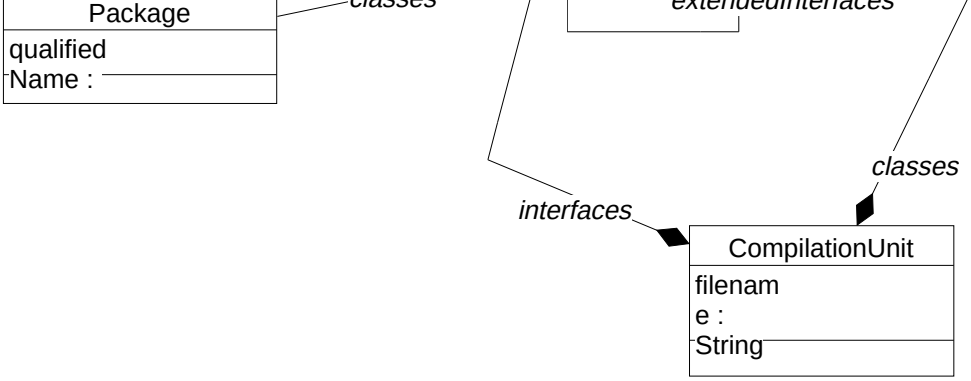
The following pages can be glued together to form a large UML class diagram of all AST node types.

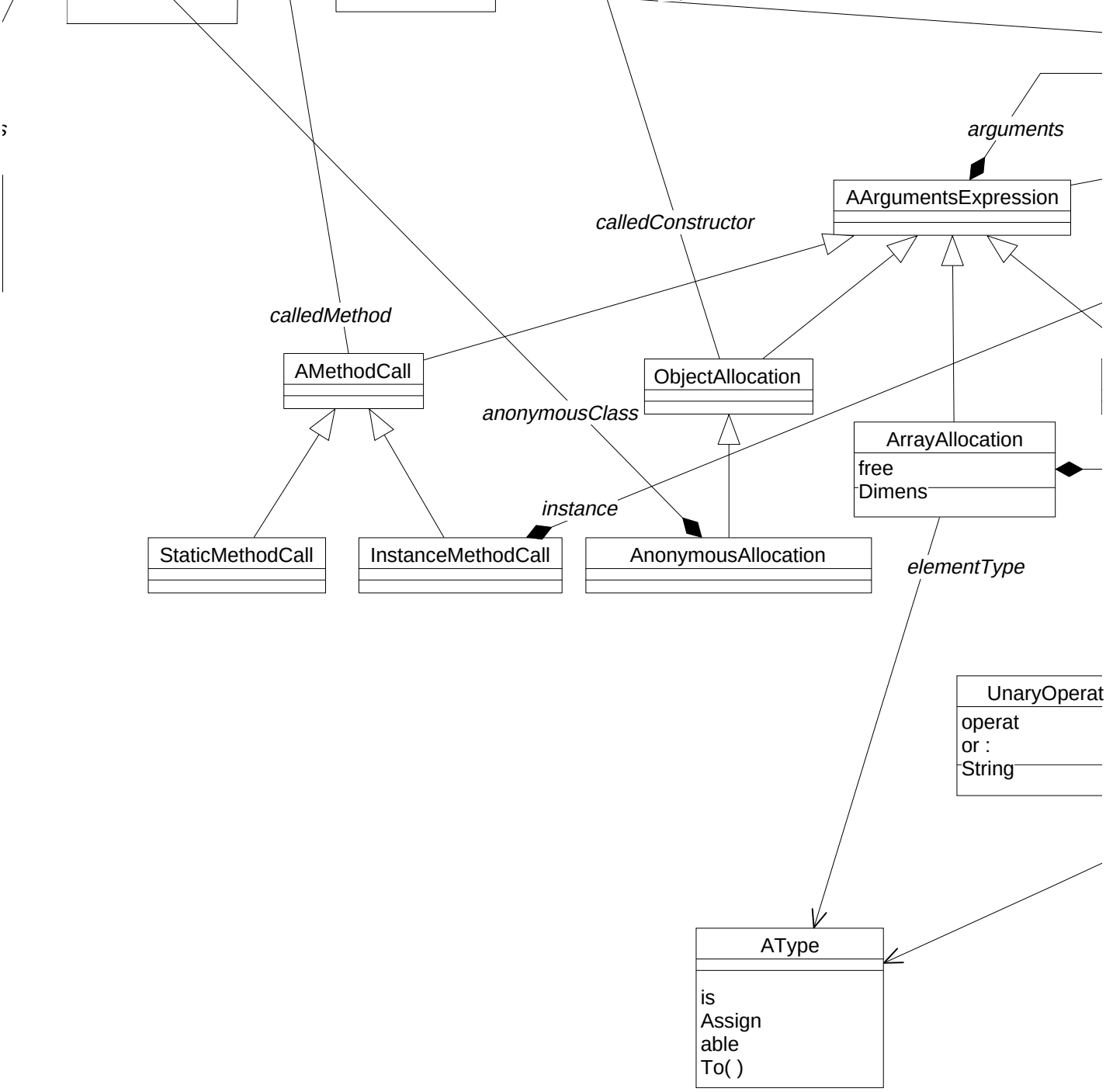

AHasModifier
ATyped
type
overriddenMethod
resultType
AType
is
Assign
able
To( )
AbstractMethod
Field
initializer
PrimitiveType
is
Boolea
n :
boolean
isByte :
boolean
isChar :
boolean
is
elementType
AReferenceType
isSubtypeOf( )
getInstanceMethod( )
getStaticMethod( )
abstractMethods
fields
Array
NullType
AUserType
nestedClasses
nestedInterfaces
implementedInterfaces
Interface
is
is
Subclas
interfaces

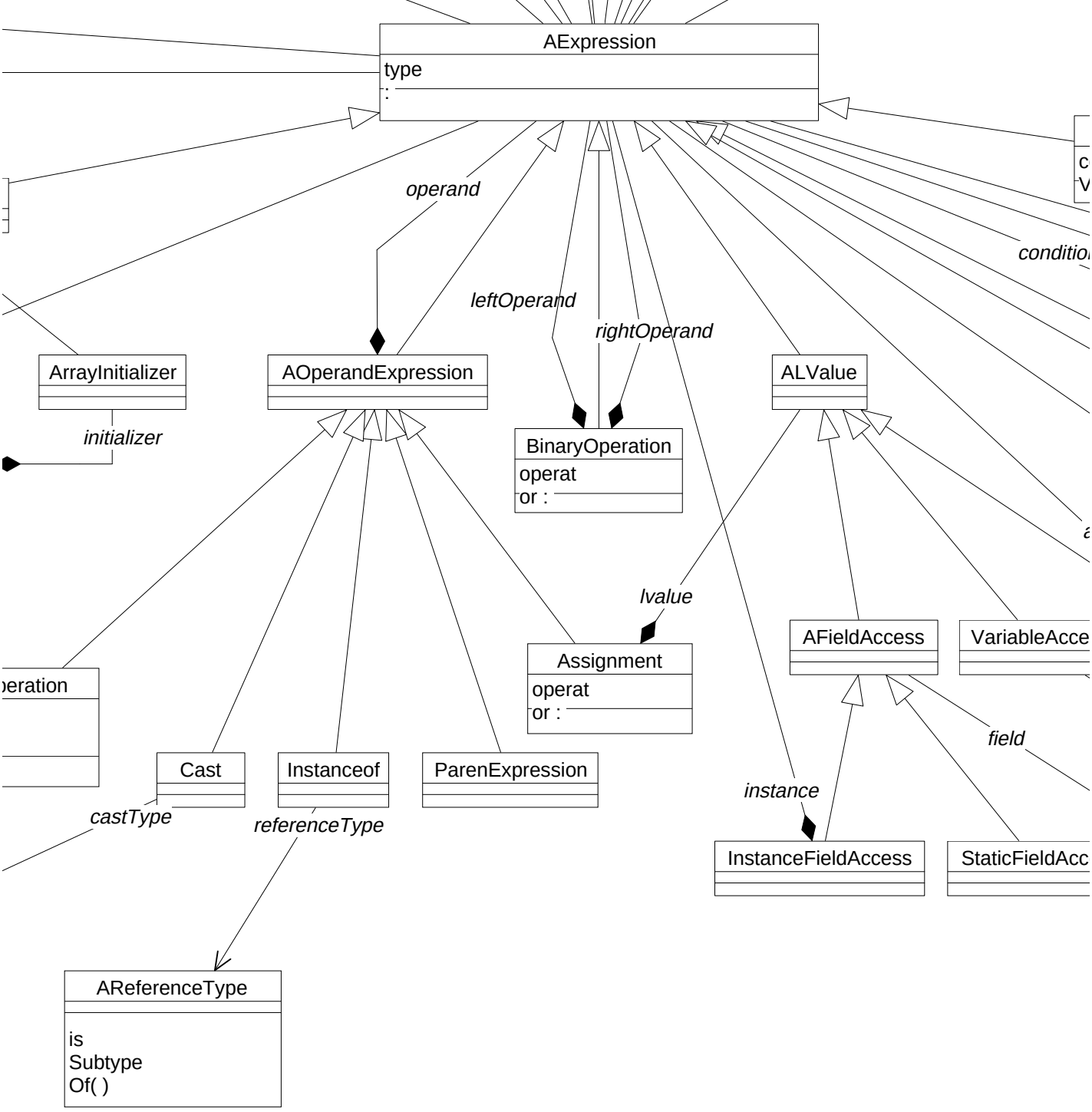

ANamed
name : String
qualifiedName( )
Brea
tar
ATargetSta
label : String
AVariable
Block
block
Try
AMethod
Parameter
LocalVariable
parameters
catchClauses
finallyClaus
body
ConcreteMethod
initializer
thod
block
block
staticInitializers
parameter
exceptions
Catch
Finally
instanceInitializers
concreteMethods
UserTy
userType
variable
Constructor
VariableDeclaration
es
constructors
Class
calledConstructor
constructorCall
clas
superclass
ConstructorCall
arguments

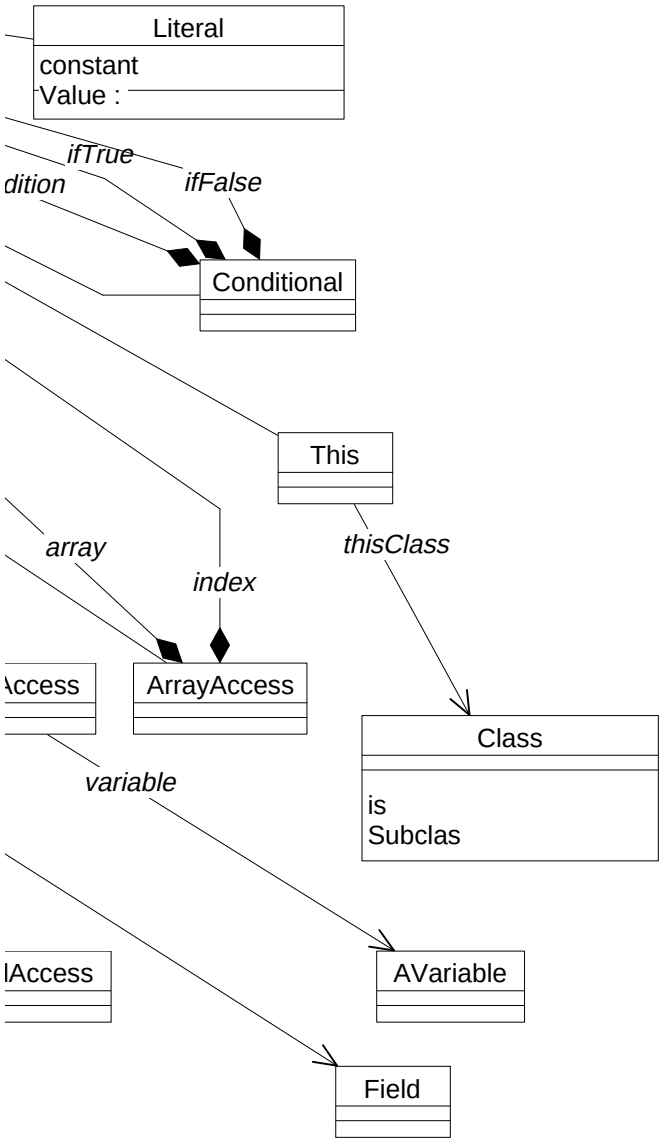

AStatement
Break
target
etStatement
String
statements
Continue
elseBranch
thenBranch
body
target
block
ALoopingStatement
Throw
Retu
Synchronized
Switch
If
Do
While
expression
branches
expression
Clause
statements
expression
ASwitchBranch
expression
expressio
expression
CaseBranch
DefaultBranch
expression
updateExp
erTypeDeclaration
constantExpression
declarations
expressions


EmptyStatement
Return
ExpressionStatement
For
ession
forInit
AForInit
eExpressions
ns
ForInitDeclaration
ForInitExpression


Package
qualified
Name :
classes
extendedInterfaces
classes
interfaces
CompilationUnit
filenam
e :
String


arguments
AArgumentsExpression
calledConstructor
calledMethod
AMethodCall
ObjectAllocation
anonymousClass
ArrayAllocation
free
Dimens
instance
StaticMethodCall
InstanceMethodCall
AnonymousAllocation
elementType
UnaryOperat
operat
or :
String
AType
is
Assign
able
To( )


AExpression
type
operand
leftOperand
rightOperand
ArrayInitializer
AOperandExpression
ALValue
initializer
BinaryOperation
operat
or :
lvalue
Assignment
operat
or :
AFieldAccess
field
Cast
Instanceof
ParenExpression
castType
referenceType
instance
InstanceFieldAccess
AReferenceType
is
Subtype
Of( )


Literal
constant
Value :
ifTrue
dition
ifFalse
Conditional
This
array
index
thisClass
ArrayAccess
Class
is
Subclas
variable
AVariable
Field