6 Implementation of CoffeeStrainer

This chapter contains information about the implementation of Coffeestrainer. Section 6.1, which is derived from a technical report co-authored by André Spiegel [Bokowski, Spiegel 1998], presents the interface and implementation of BARAT, the Java front-end on which CoffeeStrainer is based. The main part of the section describes the ASG structure provided by BARAT. Section 6.2 explains how CoffeeStrainer is implemented on top of BARAT and presents a prototypical graphical user interface (GUI) which has been built for CoffeeStrainer. Finally, Section 6.3 discusses performance issues.

6.1 Barat – a front-end for Java

CoffeeStrainer has been implemented on top of BARAT, a compiler front-end for Java. BARAT supports static analysis of Java programs. BARAT parses Java source code files and class files and builds a complete abstract semantics graph (ASG) of the parsed Java program. The ASG contains name analysis information and type analysis information. During name analysis, sometimes called *name resolution*, 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 ASG built by BARAT is a passive data structure which cannot be changed. There are similar systems which allow the ASG to be changed, and which therefore enable full compile-time reflection, or even run-time reflection for Java [Welch, Stroud 1998, Tatsubori 1999]. BARAT parses source code (according to the syntax of Java 1.2 [Gosling et al. 1996]), or if no source code is available, class files (byte-code). From class files, only class and interface declarations with method signatures are accessible; the body of concrete methods is missing for classes for which no source code is available. There is no explicit distinction between the phases of loading, parsing, and analyzing Java source code. All actions that need to be performed for building the ASG of a Java program are transparent to clients of BARAT and are triggered on demand.

BARAT consists of the following packages:

- The top-level package `barat` contains, among others, the class `Barat` which is the main entry point for BARAT with methods like `getClass` or `getInterface` which return an ASG node representing a parsed class or interface. (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.)

- The package `barat.reflect` contains the node types of the abstract semantics graph, like `Class`, `Interface`, `AbstractMethod`, `Field`, `Parameter`, `Block`, `ObjectAllocation`, etc.

- The package `barat.collections` contains type-specific collection classes (lists and iterators) for ASG node objects.

- The package `barat.parser` contains BARAT's implementation classes.

The remainder of this section is organized as follows: In Section 6.1.1, the features of ASG nodes in general will be described, and Section 6.1.2 presents the individual elements of BARAT abstract semantics graphs. In Section 6.1.3, we will describe the main entry class of BARAT which may be used for looking up class and package objects by name. Sections 6.1.4 and 6.1.5 present two techniques which help implementing analyses which do not fit well in CoffeeStrainer's model of mostly local constraints. Finally, Section 6.1.6 contains information about the implementation of BARAT and lessons we have learned during its implementation.

6.1.1 ASG Nodes

The common supertype of all ASG node types is the interface `Node` (see Figure 6.1), which is part of the top-level package `barat`. All other ASG 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 ASG 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 between ASG node types start with `get`, whereas accessor methods for attributes of other – mostly primitive – types have no 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)[1].

[1]This (somewhat arbitrary) naming convention has historical reasons and cannot be changed easily because BARAT is already used by several other projects. We decided to refer to the actual implementation as opposed to an idealized system which is only similar to the one which has been implemented.

```
public interface Node {

  // container and containment aspect for this
  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 Tag[]   getTags();
  public Tag     getTag(String tagName);
  public String  getTagValue(String tagName);

  // 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);
}
```

Figure 6.1: The interface `Node`

Containment

The most important operations defined for all nodes are those involving the *containment relation*, namely, the methods `aspect`, `container`, and the methods whose names start with `containing`. These methods allow to find out, for a given ASG `Node`, what higher level program element it is contained in. For example, `return` statements are contained in methods, which are contained in classes. In BARAT, one can access the containing program elements 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 one could also write `r.containingClass()` on the last line. The method `container` returns the immediate container of a `Node`, and `aspect` describes what kind of constituent a `Node` is for that container — one 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"`. 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, one can 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;
                     }
                   };
  return null;
}
```

Suppose you want to check whether the statement `return 14;` is nested within an `if`-statement. Using `containing(barat.reflect.If.class)`, if it was implemented 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`, `containingMethod` and `containing` 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 thus stops at the anonymous class, and returns `null`.

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

There are four methods to access tags in Javadoc comments, specially formatted comments in the source file with which types, methods, and fields may be marked, as in the following example:

```
/**
 *@log-uses
 *@layer System
 */
class TagExample {

  /**@pre o!=null*/
  public void print(Object o) {
    // ...
  }
}
```

Here, the parser stores the tags "`@log-uses`" and "`@layer`" for class `TagExample`, and the tag "`@pre`" for method `doit`. Whether a tag is defined for a program element can be queried using `hasTag`, which expects as its argument a tag name (without the "`@`"). The value of a tag, i.e. the string following the tag name up to the next tag definition or the end of the comment, can be retrieved using `getTagValue`. The methods `getTag` and `getTags` return an object (or an array of objects) of class `Tag`, which is just a pair of `String` values, the tag name and the tag value. These can be retrieved using the methods `getName` and `getValue` of class `Tag`.

The interpretation of tags is left open, except for the tags defined in the Java language specification [Gosling et al. 1996], namely, `@see` for referring to related program elements, `@author` and `@version` for annotating classes and interfaces, `@param` for describing method parameters, `@return` for describing the return value of a method, and `@exception` for describing exceptions that might be thrown by a method.

Visitors and attributes

The remaining methods, `accept`, `addAttribute`, and `attributeValue`, support traversals and analyses of abstract semantics graphs according to two complementing paradigms: the method `accept` is the hook for traversals using the Visitor design pattern [Gamma et al. 1995], while `addAttribute` and `attributeValue` support the definition of on-demand traversals similar to attribute grammars [Knuth 1968, Hedin 1999]. Both ways of traversing the ASG will be explained in detail in Sections 6.1.4 and 6.1.5.

6.1.2 Elements of the abstract semantics graph

In this section, we will go through the main categories of ASG node types in detail: type nodes, nodes representing other declarative program elements, expression nodes, and statement nodes. For each type, we list all supertypes using the notation "`supertype > type > subtype`" instead of only naming the direct supertype in order to make the context of each type more explicit.

Types

The abstract supertype for all ASG nodes representing Java types is the interface `AType` (remember that the prefix `A` means that this interface is abstract - it is implemented as an abstract class):

```
AType
 boolean isAssignableTo(AType);
 boolean isPassableTo(AType);
 boolean isCastableTo(AType);
```

The methods `isAssignableTo`, `isPassableTo` and `isCastableTo` allow 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. The first two of these are almost the same and allow only conversions from subtypes to supertypes and from primitive types to wider primitive types (e.g., from `float` to `double`). The exception for assignments is that an `int` value might be assigned to a formal of type `byte`, `short`, or `char`. Casts allow more conversions; they allow downcasts from a supertype to a subtype and narrowings from a primitive type to a narrower primitive type.

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

```
AType > PrimitiveType
  boolean isBoolean();
  boolean isByte();
```

```
  boolean isChar();
  boolean isDouble();
  boolean isFloat();
  boolean isInt();
  boolean isLong();
  boolean isShort();
  String  getName();
  Array   getCorrespondingArray();
```

There is a distinct `PrimitiveType` object for each primitive type in Java; one may check what actual type a `PrimitiveType` object represents by calling `isBoolean`, `isByte`, etc. There is also a method `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 one may compare these types for equality using ==. Therefore, `AType` provides a method `getCorrespondingArray` which returns a unique array node with the current type as its element type.

```
AType > AReferenceType
  boolean isSubtypeOf(AReferenceType possibleSupertype)
  AMethod getInstanceMethod(String name, AType[] argTypes)
  AMethod getStaticMethod  (String name, AType[] argTypes)
```

Java reference types, modeled by `AReferenceType`, may be subtypes of other reference types, and one can check this by calling `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 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). The method `getElementType` of `Array` returns the ASG node representing the array's element type.

```
AType > AReferenceType > Array
   AType getElementType();
```

It has proven practical to keep arrays and user-defined types separate, i.e. `Array` and `AUserType` are not in any inheritance relation. Note that one array is the subtype of another array if and only if their element types are subtypes of another.

```
AType > AReferenceType > NullType
```

Class `NullType` does not provide any methods over those inherited from `AReferenceType`. The `NullType` is the type of the literal `null` 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. invoking `getReturnType` on them yields `null`.) The singleton `NullType` object `isSubtypeOf` (and therefore, `isAssignableTo` etc.) any other `AReferenceType` object.

```
AType > AReferenceType > AUserType
   String             name();
   String             qualifiedName();
   boolean            isAbstract();
   boolean            isFinal();
   boolean            isPublic();
   boolean            isProtected();
   boolean            isPrivate();
   FieldList          getFields();
   AbstractMethodList getAbstractMethods();
   BlockList          getStaticInitializers();
   ClassList          getNestedClasses();
   InterfaceList      getNestedInterfaces();
```

The class `AUserType` provides the following methods: `name` and `qualifiedName` return the type's simple or fully qualified name, respectively. The methods `isAbstract`, `isFinal`, `isPublic`, `isProtected`, and `isPrivate` return true if the class or interface has the corresponding modifier. The method `getFields` returns a list of fields contained in the user-defined type[2]. Likewise, `getAbstractMethods` returns the list of abstract methods defined for the type (concrete methods can only occur in classes, not in interfaces), `getStaticInitializers` returns a list of `Block` nodes, and `getNestedClasses` and `getNestedInterfaces` return a list of the contained inner classes or interfaces, respectively.

```
AType > AReferenceType > AUserType > Interface
    InterfaceList getExtendedInterfaces();
    boolean       isSubinterfaceOf(Interface possibleSuperinterface);
```

Class `Interface` provides two additional methods specific to Java interfaces: `getExtendedInterfaces` returns the list of superinterfaces, which may be the empty list, and the helper method `isSubinterfaceOf` determines if two interfaces are in a subtype relationship.

```
AType > AReferenceType > AUserType > Class
    InterfaceList      getImplementedInterfaces();
    Class              getSuperclass()
    ConcreteMethodList getConcreteMethods();
    ConsructorList     getConstructors();
    BlockList          getInstanceInitializers();
    boolean            isSubclassOf(Class);
    boolean            isImplementationOf(Interface);
```

The class `Class` (which is different from the standard Java class `java.lang.Class`) contains methods for accessing the list of implemented interfaces, the direct superclass, the list of concrete methods of a class, and a list of blocks which are instance initialization blocks.

[2] BARAT makes use of a primitive kind of parameterized collection classes. For each ASG node type, there is a corresponding `List` class together with an `Iterator` class for iterating over the list, providing type-safe access to collections.

There are also two helper methods, `isSubclassOf`, which transitively follows the extends-relation between classes, and `isImplementationOf`, which transitively operates on the graph of superinterfaces implemented by this class. Note that unlike these helper methods, the methods for accessing the superclass, the implemented interfaces and (for class `Interface`) extended interfaces are not transitive, they only refer to direct superclasses and superinterfaces.

Other declarative program elements

The class `Package` provides two methods, `getClasses` and `getInterfaces`, which return the list of all classes or interfaces of a package, respectively. It should be noted that calling these methods necessarily causes all source-code and byte-code files in the package directory to be loaded at once.

```
Package
 ClassList     getClasses();
 InterfaceList getInterfaces();
```

The class `Field` supports the following methods: The field's name can be retrieved using `name`. The methods `isPublic`, `isProtected`, `isPrivate`, `isStatic`, `isFinal`, and `isTransient` return `true` if the field has the corresponding modifier. Fields declared in interfaces are `public`, `static`, and `final` even if these modifiers do not occur explicitly. The method `getType` returns the field's declared type. Finally, `getInitializer` returns an initializing expression or `null` if the field has no initializer.

```
Field
 String      name();
 boolean     isPublic();
 boolean     isProtected();
 boolean     isPrivate();
 boolean     isStatic();
 boolean     isFinal();
 boolean     isTransient();
 AType       getType();
 AExpression getInitializer();
```

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. This difference is the reason 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. The class `AMethod` provides a method `name` for retrieving the method's name. The method's modifiers are available through `isPublic`, `isProtected`, `isPrivate`, `isStatic`, `isFinal`, and `isSynchronized`. Thus, to distinguish a static method from an instance method, one needs to check the method's `static` modifier. Furthermore, `AMethod` allows to retrieve the method's result type with `getResultType`, which returns `null` if the method is declared `void`. The method `getParameters` returns a list of ASG nodes representing parameters, and the helper method `getOverriddenMethod` returns the method which is overridden by the current method and `null` if there is no overridden method.

```
AMethod
 String name();
 boolean isPublic();
 boolean isProtected();
 boolean isPrivate();
 boolean isStatic();
 boolean isFinal();
 boolean isSynchronized();
 AType getResultType();
 ParameterList getParameters();
 AMethod getOverridenMethod();
```

The class `AbstractMethod` does not provide any additional methods. The method `getBody` of class `ConcreteMethod` returns the method body as a `Block` (containing a list of statements).

```
AMethod > AbstractMethod
```

```
AMethod > ConcreteMethod
  Block getBody();
```

`Constructor` objects are derived from `AMethod`, adding two methods: a method `getBody` (as in `ConcreteMethod`) and a method `getConstructorCall` which returns the constructor invocation within this constructor. In Java, any constructor first makes a call to some other constructor (by default, this is the call `super()`). This is modeled as an object of type `ConstructorCall`, which BARAT initializes to (the equivalent of) `super()` if no explicit call is found in the source code. There is one exception, the constructor of `java.lang.Object`, for which `getConstructorCall` returns `null`.

```
AMethod > Constructor
  Block           getBody();
  ConstructorCall getConstructorCall();
```

```
ConstructorCall
 AExpressionList getArguments();
 Constructor     getCalledConstructor();
```

Note also that `getCalledConstructor` in `ConstructorCall` returns the real `Constructor` object being called (the arguments of the constructor call can be retrieved with `getArguments`, which returns a list of `AExpression` objects). 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.

Formal parameters of methods and constructors are represented by nodes of type `Parameter`. Its superclass, `AVariable`, provides methods for accessing variable names and modifiers, and the method `getType` which returns the variable's declared type.

```
AVariable
 String name();
 boolean isFinal();
 AType getType();
```

```
AVariable > Parameter
```

The second subclass of `AVariable`, called `LocalVariable`, is the type of ASG nodes representing local variables. Local variables have an additional modifier which can be accessed using `isTransient`, and they may have an initializing expression that is returned by `getInitializer`. If there is no initializer for a local variable, `getInitializer` returns `null`. Local variables can only be declared within blocks, which will be explained in a following subsection on ASG nodes representing statements.

```
AVariable > LocalVariable
  boolean     isTransient();
  AExpression getInitializer();
```

Expressions

All possible kinds of Java expressions are modeled by the abstract class `AExpression`. The static type of any expression is returned by its method `getType`.

```
AExpression
 AType getType();
```

The class `Literal`, which represents literal values, supports one additional method, `constantValue`, which returns the literal's value as a Java object. Literals can be `String` objects, the value `null`, or values of primitive types, in which case the wrapper classes `java.lang.Float`, `java.lang.Boolean`, etc. are used when returning the `constantValue`.

```
AExpression > Literal
  Object constantValue();
```

The class `This` stands for the keyword `this`. Its method `getThisClass` returns the `Class` in which the keyword `this` is used. In the presence of inner classes, the keyword `this` may be qualified by a name of one of the outer classes. The method `getThisClass` returns the

`Class` which the qualified `this` expression refers to. The keyword `super`, which is very similar to `this` and may be qualified as well, is modeled by a subclass `Super` of `This`. Note that the method `getThisClass` in the case of `super` does return the same class as if the keyword had been `this` — the reason for this is that for regenerating the source code in the case of a qualified `super`, the name of the outer class is needed, which cannot be recovered in general if one only knows its superclass.

```
AExpression > This
  Class getThisClass();
```

```
This > Super
```

All binary operations except assignments are modeled by the class `BinaryOperation`. This includes arithmetic expressions as well as comparison expressions. The method `operator` returns the binary operation's operator, and `getLeftOperand` and `getRightOperand` return the binary expression's subexpressions.

```
AExpression > BinaryOperation
  String      operator;
  AExpression getLeftOperand();
  AExpression getRightOperand();
```

The ternary conditional operator '`?:`' is modeled by expressions of type `Conditional`. The method `getCondition` returns the first subexpression, `getIfTrue` returns the expression that will be evaluated if the first subexpression is `true`, and `getIfFalse` returns the expression that will be evaluated if the first subexpression is `false`.

```
AExpression > Conditional
  AExpression getCondition();
  AExpression getIfTrue();
  AExpression getIfFalse();
```

There is an abstract class `ALValue` whose subclasses may be used as l-values in an assignment. However, such expressions can of course occur in other contexts as well.

```
AExpression > ALValue
```

The class `VariableAccess` represents accesses to either local variables or parameters. The accessed variable is returned by the method `getVariable`.

```
AExpression > ALValue > VariableAccess
   AVariable getVariable();
```

Accesses to array elements are modeled as `ArrayAccess` objects. Mostly, these are based on field or variable accesses. Consider

```
int[] a = new int[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 = new int[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`. The subexpression of an `ArrayAccess` that represents the accessed array is returned by the method `getArray`, and the method `getExpression` returns the subexpression representing the array index. The former subexpression has an array type, and the latter subexpression is of type `int`.

```
AExpression > ALValue > ArrayAccess
   AExpression getArray();
   AExpression getExpression();
```

Accesses to object fields are represented by the abstract class `AFieldAccess`, which has a method `getField` which returns the ASG node for the accessed field. The two concrete subclasses of `AFieldAccess` are `StaticFieldAccess` and `InstanceFieldAccess`. The class `InstanceFieldAccess` has an additional method `getInstance` which returns the subexpression which represents the object instance whose field is accessed.

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 at the beginning of Section 6.1.1 for an example.

```
AExpression > ALValue > AFieldAccess
   Field getField();
```

```
AExpression > ALValue > AFieldAccess > StaticFieldAccess
```

```
AExpression > ALValue > AFieldAccess > InstanceFieldAccess
    AExpression getInstance();
```

There are a number of expressions in Java which operate on a single operand expression, such as unary operations, `instanceof` expressions, casts, *etc*. These are modeled by the abstract class `AOperandExpression` which allows to retrieve the operand subexpression using the method `getOperand`.

```
AExpression > AOperandExpression
  AExpression getOperand();
```

The class `UnaryOperation` represents unary operations. The operator of an unary operation can be retrieved with the method `getOperator`. The distinction between prefix and postfix unary expressions is made by means of the method `isPostfix`.

```
AExpression > AOperandExpression > UnaryOperation
   String  operator();
   boolean isPostfix();
```

Cast expressions, modeled by class `Cast`, have an operand and a type to which the cast expression casts. The ASG node for the cast type can be retrieved using the method `getCastType`.

```
AExpression > AOperandExpression > Cast
   AType getCastType();
```

The `instanceof` expression in Java has an operand and tests whether the operand's runtime type is assignable to a certain reference type which can be retrieved by method `getReferenceType`. This is not an `AUserType` because one can also cast to array types in Java.

```
AExpression > AOperandExpression > Instanceof
   AReferenceType > getReferenceType()
```

Parenthesized expressions could be replaced by their operand without changing the semantics of a parsed program. However, in BARAT, these expressions are kept in the ASG as operand expressions of type `ParenExpression` because regenerating the source code is easier if parenthesized expressions are not optimized away.

```
AExpression > AOperandExpression > ParenExpression
```

Assignments are modeled as operand expressions which also have an l-value. Thus, the class `Assignment` has a method `getLValue` which returns the assignment's l-value.

```
AExpression > AOperandExpression > Assignment
   AExpression > getLValue();
```

There is a common abstract superclass `AArgumentsExpression` for expressions that contain a list of argument subexpressions, which can be retrieved with the method `getArguments`.

```
AExpression > AArgumentsExpression
  AExpressionList > getArguments();
```

Method calls are the first example of argument expressions. 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 the two are modeled as separate types, unlike static methods and instance methods (see previous section).

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

  public methodC() {
    Target.methodA();

    Target t = new Target();
    t.methodB();

    methodB();
  }
}
```

The three method calls in `methodC` are resolved as follows: The call `Target.methodA()` is a `StaticMethodCall` whose called method is `Target.methodA`. The call `t.methodB()` is an `InstanceMethodCall` whose `instance` is the expression `t` and whose called method is `Target.methodB`. Finally, the call `methodB()` is an `InstanceMethodCall` whose called method is the same as before, and whose `instance` is the (implicit) expression `this`, represented by an ASG node of type `This`. The method `getThisClass`, when called on this ASG node, returns the ASG node of type `Class` representing the class `Target`.

```
AExpression > AArgumentsExpression > AMethodCall
   AMethod getCalledMethod();

AExpression > AArgumentsExpression > AMethodCall > StaticMethodCall

AExpression > AArgumentsExpression > AMethodCall > InstanceMethodCall
    AExpression getInstance();
```

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();
```

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 subexpressions, while the number of free dimensions can be retrieved using the method `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 `getAnonymousClass` of the `AnonymousAllocation` object.

```
AExpression > AArgumentsExpression > ObjectAllocation
   Constructor > getCalledConstructor();
```

```
AExpression > AArgumentsExpression > ObjectAllocation > AnonymousAllocation
   Class getAnonymousClass();
```

```
AExpression > AArgumentsExpression > ArrayAllocation
   int               freeDimensions();
   ArrayInitializer getInitializer();
```

An `ArrayAllocation` may have an array initializer that can be retrieved using `getInitializer`. If there is no array initializer, this method returns `null`. An aray initializer may also occur as the initializer of field or local variable declarations. It is modeled by the class `ArrayInitializer`, where the arguments of the `AArgumentsExpression` superclass are the array initializer's element expressions. In the case of multidimensional arrays, these in turn can be `ArrayInitializer`.

```
AExpression > AArgumentsExpression > ArrayInitializer
```

Statements

Java statements are modeled as subtypes of `barat.reflect.AStatement`, which does not support any methods.

```
AStatement
```

The empty statement is modeled by class `EmptyStatement`. Empty statements, written ";", may occur, for example, just after the `while` statement in "`while(i++<=100) { out.println(i); };`" because of the extra semicolon.

```
AStatement > EmptyStatement
```

A `VariableDeclaration` is a statement that declares a variable. However, the actual variable is modeled as a `LocalVariable` contained within the `VariableDeclaration`, returned by the method `getVariable`, making the distinction between the statement and the declaration explicit. A `VariableAccess` (see "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; in this case the declaration would be modeled as two consecutive `VariableDeclaration` statements.

```
AStatement > VariableDeclaration
  LocalVariable getVariable();
```

Some statements may have a preceding label. These statements are modeled by the abstract class `ATargetStatement` which has a method `label` that returns the statement's label. Subclasses of `ATargetStatement` are `Block`, `If`, `Switch`, `Try`, `Synchronized`, and `ALoopingStatement`, the abstract superclass for loop statements.

```
AStatement > ATargetStatement
  String label();
```

A block essentially is a list of statements. Only statements within a block (and in the initializing statement of a `For`) may be `VariableDeclaration` statements. The list of statements is returned by `getStatements`.

```
AStatement > ATargetStatement > Block
  AStatementList getStatements();
```

Java, like C, allows expressions to be statements, modeled by class `ExpressionStatement` with a method `getExpression` that returns the contained `AExpression`. For example, 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 "Expressions").

```
AStatement > ExpressionStatement
  AExpression getExpression();
```

Return statements are modeled by class `Return` with a method `getExpression` that returns the returned `AExpression`. If the return statement does not have an expression — in the case of `void` methods — the method `getExpression` returns `null`.

```
AStatement > Return
  AExpression getExpression();
```

Throw statements, modeled by class `Throw` with a method `getExpression` always have a valid containing expression, the static type of which must be assignable to `java.lang.Throwable`.

```
AStatement > Throw
  AExpression getExpression();
```

The `synchronized` statement is modeled by class `Synchronized` with a method `getExpression` that returns the expression which determines the object on which the synchronization is performed. The block of statements that are executed under synchronization can be retrieved by calling `getBlock`.

```
AStatement > ATargetStatement > Synchronized
   AExpression getExpression();
   Block       getBlock();
```

The `if` statement is modeled by class `If`. The method `getExpression` returns the condition expression, and the two methods `getThenBranch` and `getElseBranch` return the statements that are to be executed if the condition expression evaluates to `true` or `false`, respectively. Very often, the type of the statement that is returned will be `Block`. If an `if` statement has no `else` branch, `getElseBranch` returns `null`.

```
AStatement > If
   AExpression getExpression();
   AStatement  getThenBranch();
   AStatement  getElseBranch();
```

The `switch` statement selects one or more branches depending on the value of the contained expression which is returned by `getExpression`. The list of branches (`ASwitchBranch` will be explained below) is returned by method `getBranches`.

```
AStatement > Switch
   AExpression getExpression();
   ASwitchBranchList getBranches();
```

The `do`, `while`, and `for` statements are subtypes of the abstract supertype `ALoopingStatement`. By calling `getExpression`, the continuation condition can be retrieved, and the method `getBody` returns the statement (often a `Block`) that is the loop's body. Objects of type `For` support two additional methods: `getForInit`, returning the initialization part of the `for` loop, and `getUpdateExpressions`, returning the list comma-separated expressions that are evaluated after each execution of the loop body.

```
AStatement > AStatementWithExpression > ALoopingStatement
   AExpression getExpression();
   Block getBody();
```

```
AStatement > AStatementWithExpression > ALoopingStatement > Do
```

```
AStatement > AStatementWithExpression > ALoopingStatement > While
```

```
AStatement > AStatementWithExpression > ALoopingStatement > For
    AForInit getForInit();
    AExpression getUpdateExpression();
```

Both `continue` and `break` are statements that may refer to a (sometimes labeled) target statement. Because targets of `continue` statements may only be looping statements, the method `getTarget` of `Continue` returns an object of type `ALoopingStatement`. The method `getTarget` of class `Break` returns an object of type `ATargetStatement`.

```
AStatement > Continue
  ALoopingStatement getTarget();
```

```
AStatement > Break
  ATargetStatement getTarget();
```

The `try` statement consists of three parts: A `Block` of statements, returned by method `getBlock`, which is mandatory, a list of `catch` clauses which may be empty, returned by method `getCatchClauses`, and an optional `finally` clause, returned by method `getFinallyClause`. There must be at least one `catch` clause or a `finally` clause.

```
AStatement > ATargetStatement > Try
  Block     getBlock();
  CatchList getCatchClauses();
  Finally   getFinallyClause();
```

A `UserTypeDeclaration` is a declaration of an inner class or interface that occurs inside a method. Java permits this wherever a statement is allowed. The method `getUserType` returns the inner class or interface. An inner class or interface declaration that does not occur inside a method is modeled as a nested class or nested interface of the enclosing `AUserType`, see the section on "Other declarative program elements".

```
AStatement > UserTypeDeclaration
  AUserType getUserType();
```

The abstract class `AForInit` represents the initialization part of a `for` loop. It has two subclasses: The class `ForInitDeclaration` represents the case that the `for` loop contains a list of variable declarations the scope of which is the `for` loop. This variable declaration list can be retrieved by method `getDeclarations`. The second subclass of `AForInit` is `ForInitExpression`, containing a comma-separated list of expressions that are to be evaluated just before entering the `for` loop.

```
AForInit
```

```
AForInit > ForInitDeclaration
  VariableDeclarationList getDeclarations();
```

```
AForInit ForInitExpression
  AExpressionList getExpressions();
```

Branches of `switch` statements are modeled as objects of the abstract class `ASwitch-Branch`, supporting a method `getStatements` which returns the list of statements of the branch. In Java, execution of `switch` branches “falls through” if a `switch` branch does not end with a `break` statement. `ASwitchBranch` class has two concrete subclasses: `Case-Branch` and `DefaultBranch`. The class `CaseBranch` has one additional method, `get-ConstantExpression`, which returns the constant expression the value of which is compared to the `switch` expression at runtime.

```
ASwitchBranch
 AStatementList getStatements();
```

```
ASwitchBranch > CaseBranch
  AExpression getConstantExpression();
```

```
ASwitchBranch > DefaultBranch
```

A `catch` clause, which appears as part of the `try` statement, is modeled by class `Catch` which has two methods: `getParameter` returns an object of type `Parameter` that represents the exception parameter of the `catch` clause, and `getBlock` returns the block of statements of the `catch` clause.

```
Catch
 Parameter getParameter();
 Block getBlock();
```

Finally, the class `Finally` represents a `finally` clause of a `try` statement. Its method `getBlock` returns the block of statements of the `finally` clause.

```
Finally
 Block getBlock();
```

6.1.3 Retrieving ASG root objects

The single point of access to the entire BARAT system is the class `barat.Barat`, shown in Figure 6.2. One can retrieve the root object of a BARAT ASG for a given Java class simply by calling the method `getClass`:

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

The method `getClass` expects the fully qualified name of the class that should be parsed. The entire parsing process and internal analysis is handled by BARAT transparently and on demand. At any time, one may call methods of ASG node objects to access other parts of the ASG, or other ASGs of other classes or interfaces that are referred to.

One may either use the method `getClass` or `getInterface` to retrieve ASG nodes for a Java class or interface, respectively. If it is not clear whether a given name refers to a class or an interface, the method `getUserType` may be used, which returns an object of `AUserType`, the common abstract supertype of interfaces and classes. There are also some convenience methods that allow 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 semantics graph based on that source code. However, if only a byte code file exists, BARAT parses that, although it doesn't decompile any of the actual instructions in it. Only field and method signatures will therefore be visible for analysis; method bodies of concrete methods are nonexistent (`null`).

The method `registerAttributeAdder` will be explained in Section 6.1.5.

```
package barat;

import barat.reflect.*;
import barat.reflect.Class; import barat.reflect.Package;

public class Barat {
  // runtime flag:
  public static boolean debugLoading = false;

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

  // accessing ASG roots:
  public static AUserType getUserType(java.lang.String);
  public static Class getClass(java.lang.String);
  public static Interface getInterface(java.lang.String);

  // accessing prominent types:
  public static Class getObjectClass();
  public static Class getStringClass();
  public static Interface getThrowableInterface();
  public static Interface getCloneableInterface();

  // registering an attribute adder:
  public static void registerAttributeAdder(Visitor adder);

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

Figure 6.2: Interface of class `barat.Barat`

6.1.4 Visitors

Traversing a BARAT ASG 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. 1995] that allows you to formulate common analysis algorithms in a much easier way. This framework is independent of CoffeeStrainer's built-in visitor-like traversal and may be used for implementing global analyses that do not fit well in the single traversal model of CoffeeStrainer.

The Visitor design pattern lets programmers traverse hierarchical structures of objects such that the code which does the traversal is separated from the actions to be performed at each visited object. The pattern is illustrated in Figure 6.3.

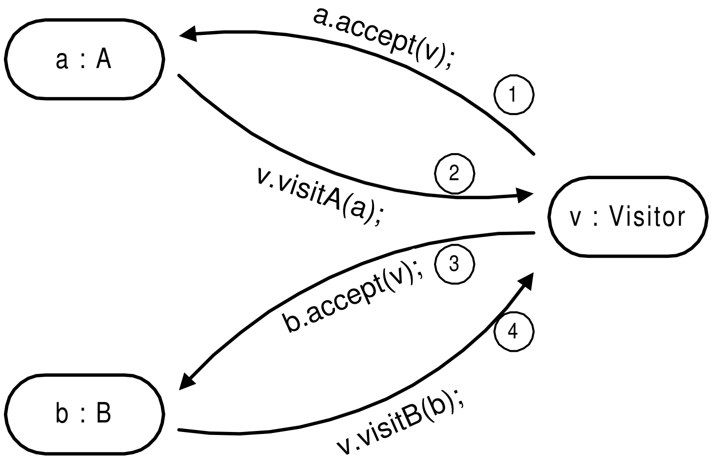

Figure 6.3: Example for visitor pattern

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, calling a method specific for the type being visited. Thus, for class `A`, `accept` would be implemented as follows:

```
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 class `Visitor`, rather than scattered all over the program. We will also see how this pattern allows us to abstract from traversal algorithms in a convenient way.

In BARAT, all possible elements of the abstract semantics graphs (i.e. all `Node` objects) implement a method `accept` 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` in class `DescendingVisitor` 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. The intended use is to have the constraint programmer 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 of `visitStaticFieldAccess`: 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

ASG node type `StaticFieldAccess` has no children), this would not strictly be necessary here, but it is always a good idea to follow this convention. Note how this corresponds to a pre-order traversal; a post-order traversal would call `super.visitStaticFieldAccess` from the first statement as opposed to this example, where it is called from the last statement.

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 ASG 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) {
             // code that deals with Do
           }
           public void visitFor(For f) {
             // code that deals with For
           }
           public void visitWhile(While w) {
             // code that deals with While
           }
        });
```

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 representing a looping statement. 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;
  // code that deals with Do
} else if(s instanceof For) {
  For f = (For)s;
  // code that deals with For
} else if(s instanceof While) {
  While w = (While)s;
  // code that deals with While
}
```

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, abstract interfaces of named objects, typed objects, and objects with modifiers.) 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 a common case `visitAMethodCall`, which would not be possible in a `switch` statement. Note also 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 with `AbstractingVisitor` 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 could be implemented.

It must be noted, though, that the `OutputVisitor` re-generates a classes' source code based on the information in the BARAT ASG. This newly generated source code is guaranteed to be semantically equivalent to the original source code except minor changes with respect to formatting and some other issues, such as the order of declarations inside a class. Also, all ordinary (i.e., non-formal) comments in the original code are lost.

6.1.5 Attributes

For some purposes that involve a non-standard traversal of the ASG, 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 ASG which can be used if the built-in traversal of CoffeeStrainer is not sufficient. Note that for all example constraints in this thesis, neither visitors nor attributes were needed for their implementation. They are, however, used in the implementation of BARAT itself.

In BARAT, the attribute concept allows to store user-defined data for each ASG 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();
}
```

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 ASG 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 is desirable to maintain a cache of already computed sets.

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

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

By registering an attribute adder visitor, `visit` methods will be called for every newly created ASG node object. As these node objects are not yet properly inserted into the ASG, 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 ASG 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(implementing);
```

Note that calls of `attributeValue(implementing)` 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.

6.1.6 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 implementation parts of ASG node types. For each ASG 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`.

The package `barat.parser` also contains the actual parser, which is generated by JAVACC (version 0.7.1) from a grammar file that consists of two parts: a BNF-based grammar that defines a scanner for transforming the input file into a sequence of tokens, and a LL(k) grammar augmented with tree-building Java code that specifies Java's syntax based on the tokens. Class files are parsed using JAVACLASS [Dahm 1999], a class library for reading, manipulating and writing Java byte code.

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 what 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 enormous 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 ASG node objects similar to attributes as known from attribute grammars [Knuth 1968, Hedin 1999]. We also chose to use parameterized types - at least internally - to gain more type safety when building the ASG, 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 6.1.5. 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 within package `barat.parser`.

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 ASG 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, 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 that allows 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 classes and interfaces of that package that are already loaded is searched. If no class or interface is found, the class path 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 [Metamata 1999a] (version 0.7.1), based on the Java 1.1

grammar distributed with JAVACC with slight modifications for 1.2[3]. The grammar input file 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 [Metamata 1999b, Wang et al. 1999].

6. Parsing of class files is performed using JAVACLASS [Dahm 1999], a package for reading and writing byte-code files.

For implementing BARAT, we have used type-parameterized versions of the 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 GJ [Bracha et al. 1998], an extension of Java that supports parameterized types.

Most of the classes in `barat.reflect` and `barat.parser` have been generated from a UML class diagram using a custom-built code generator.

6.2 From Barat to CoffeeStrainer

CoffeeStrainer is just a small layer on top of BARAT. Before building the ASG with BARAT, CoffeeStrainer parses Javadoc comments in a separate step and generates constraint classes. These classes are then loaded dynamically using Java reflection, and a CoffeeStrainer visitor traverses the ASG, calling constraint methods from constraint classes as appropriate. For every node in the ASG, first, applicable definition constraints associated with the containing method are called. Next, definition constraints are called which are associated with overridden methods. After that, definition constraints associated with the containing class and with its supertypes are called. Calling of usage constraint methods proceeds in a similar way.

Before every call of a constraint method, the variable `rationale` is initialized to the empty string. If a constraint method returns `false`, a message is printed which lists the ASG node causing the constraint violation, its line number and file name, together with the string which may have been assigned to `rationale`.

In addition to the command-line interface of CoffeeStrainer which has been described so far, we have developed a prototype of a graphical user interface for CoffeeStrainer which could be integrated into an integrated software development environment system (IDE). IDEs usually have a notion of the current project, or the working set of packages and classes in the system. To simulate this, our prototype expects, at startup, a list of package names. All classes and interfaces in these packages are then parsed and checked by CoffeeStrainer. In Figure 6.4, a screendump of the prototype is shown which lists all constraint violations in the upper part of the window. The lower part of the window contains a class browser, with a tree view of packages, classes, and method signatures on the left hand side and a source code

[3]The only change in the language between versions 1.1 and 1.2 is the additional keyword `strictfp` for declarations of floating point variables which differentiates between two different floating point semantics.

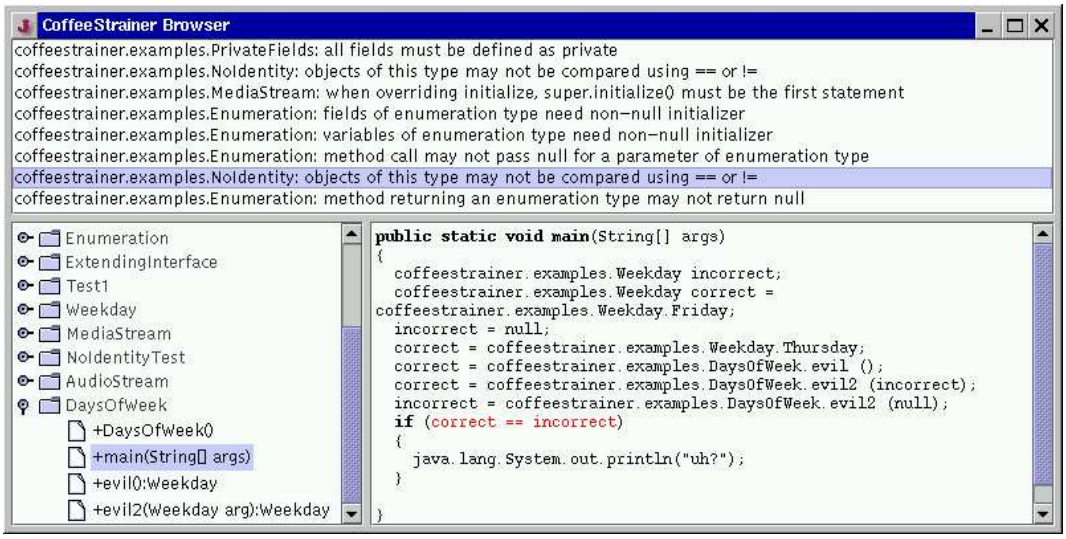

Figure 6.4: Screendump of graphical user interface

view of the currently selected class or method on the right hand side. When the user clicks on one of the constraint violation entries in the upper part of the window, the class browser part updates to show the part of the program which caused the constraint violation. In our example, eight violations are listed in the upper part of the window. The user has clicked on the constraint violation entry:

```
CoffeeStrainer.examples.NoIdentity:
            objects of this type may not be compared using == or !=
```

This is a violation of the constraint given in `CoffeeStrainer.examples.NoIdentity` which disallows object reference identity operations (see Section 4.3.2). In the lower part of the window, the class browser displays the part of the program which caused the violation: On the left hand side, method `main` of class `DaysOfWeek` is selected, and on the right hand side, the object reference identity operation is highlighted in red in the source code of method `main` ("`correct == incorrect`"). In an integrated development environment, the user could then change the source code of main appropriately, which would cause the constraint violation entry to be removed from the list of currently found violations. This latter functionality is not part of the GUI prototype of CoffeeStrainer.

6.3 Performance Evaluation

Checking constraints with CoffeeStrainer usually involves a single traversal of the abstract syntax tree of the checked program. Because CoffeeStrainer is an open framework, no upper bound of the complexity of checking a program can be given. For example, a constraint method could be written that searches the whole ASG instead of just checking a local property of a certain ASG node, resulting in quadratic complexity instead of linear complexity. However, experience showed that most of the constraints do not depend on the size of the

ASG, so that in the normal case, checking constraints has complexity $O(n)$ where n is the size of the ASG. Thus, it can be expected that checking constraints with CoffeeStrainer is dominated by parsing the program and performing name and type analysis, which certainly cannot be less than $O(n)$ because it at least builds a tree of size n. Consequently, the time for checking constraints should be comparable to the time for compiling the program.

Performance measurements on example inputs confirm these considerations: When compared to running the standard java compiler, Sun Microsystem's JAVAC, on various input files, the running time of CoffeeStrainer is between 1.4 and 2.6 times larger than for JAVAC. See Figure 6.5 for measurements taken on a PC (266 MHz Pentium, 64 MB RAM) running Linux and Sun's JDK 1.1.7. Thus, the time needed for checking constraints with CoffeeStrainer is similar to the time needed for compiling the same program (taking into account that so far almost no effort has been spent on optimizing Barat, the Java front-end on which CoffeeStrainer is built). The example inputs were the following packages: `coffeestrainer.examples`, containing examples of CoffeeStrainer constraints, `pos2`, containing a simple point-of-sale application for which CoffeeStrainer constraints enforce a layered architecture, `de.fub.bytecode.util` and `de.fub.bytecode.classfile`, containing a library for reading and writing byte-code files [Dahm 1999], and `javaparser`, containing a Java parser generated by JAVACC [Metamata 1999a]. For the last three examples, no constraints were defined, so that only the time for a full traversal (calling empty constraint methods) was measured.

input	**LOC**	**javac [s]**	**CoffeeStrainer [s]**	**factor**
`coffeestrainer.examples`	274	1.6	2.3	1.4
`pos2`	843	3.1	8.5	2.7
`de.fub.bytecode.util`	1419	5.1	11.0	2.2
`de.fub.bytecode.classfile`	6489	11.0	16.2	1.5
`javaparser`	5950	5.8	15.0	2.6

Figure 6.5: Comparing the performance of CoffeeStrainer and Sun's JAVAC