JTL and the Annoying Subtleties of Precise μ-Pattern Definitions

Tal Cohen   Joseph (Yossi) Gil   Itay Maman

Department of Computer Science
Technion—Israel Institute of Technology
Technion City, Haifa 32000, Israel
`ctal | yogi | imaman @ cs.technion.ac.il`

Abstract

We describe the lessons learned from our experience in translating the natural language definitions, and the actual Java implementation, of micro-patterns *into the declarative, formal, and ready for execution, yet human-readable equivalent in* JTL *(the Java Tools Language).*

1. Introduction

When theory, any theory, is brought to practice, many mundane details pop out: in economics, markets are not always perfect; different courts interpret the same law in different ways; and, honeymoons often dure less than a month. In the art of computer programming however, this phenomenon is perhaps the worst. Take notions as famous and (supposedly) well-understood such as "cohesion", "modularity", or "portability"—the software world would have been so much better if these were not so open to conflicting interpretations by different individuals, even within the same warehouse.

A telling example is the application of theoretic software metrics, e.g., the renown metric of *lack of cohesion in methods* (LCOM) in a class, due to Chidamber and Kemerer [5]. LCOM is defined as a rather simple mathematical function operating on the bi-partite graph, in which there is an edge between a method and a field iff that method uses that field. An attempt to compute this metric on actual code will yield different results, since it forces a precise definition of terms such as "method" (does this cover **`static`** methods, inherited methods, overridden methods, **`private`** methods, constructors, etc.?), "field" (what about inherited fields, arrays, nested fields, read-only fields or constants?), "use" (is indirect use included? how is aliasing treated? are messages sent to a field considered use?, etc.). Consequently, it is highly unlikely that different individuals will compute even close LCOM values for the same input, or that the same class design, implemented in different programming languages, will have the same LCOM value.

The situation is worse with the less precise theory of software architecture, design and, what's most dear to us, patterns. The broad term "patterns" includes in it the famous *design patterns* [9], architectural patterns [3], enterprise applications patterns [11], coding idioms [7] and implementation patterns [2]. These pattern kinds can be placed on an abstraction ladder, in which the lowest steps correspond to patterns closest to the code level. The code that corresponds to a pattern, what we may call its *footprint*, can usually be predicted for low-level patterns, but not for high-level patterns. For instance, the double-dispatch [7] idiom almost invariably leaves a footprint in which we see a method with a single argument, whose body invokes a method on that single argument, where the **`this`** variable is passed to the invoked method.

On the other hand, the footprint of (say) the *MVC (Model-View-Controller)* [3] architectural pattern is much more flexible and has numerous realizations.

In an earlier work, two of us [10] tried to reverse the process, instead of first defining the patterns, and then searching for their footprint; we proposed a set of patterns which are defined based also on the ability to recognize these in the code. To this end, we coined term *micro-patterns* (or μ-patterns for short) to denote patterns of class design, which are unique in being *mechanically recognizable* or *traceable*, i.e., unlike the classical design patterns, these are being defined directly in terms of the underlying programming language. In that work, we said that a pattern is *traceable* if it can be expressed as a "*simple formal condition on the attributes, type, name and body of a software module and its components*". For example, the SAMPLER μ-pattern is the condition on a class requiring that this class

> " *...has a* **`public`** *constructor, and one or more* **`static public`** *fields of the same type as the class itself ...* " [10]

This property of micro-patterns makes it possible to translate their definitions into an automatic device (i.e., a computer program) which will identify these patterns in actual code.

Yet, as it turns out, there are still annoying details that emerge even in trying to bring the theoretical definition of these patterns into practice, i.e., an actual implementation of a pattern recognizer. First, definitions, such as the one quoted above are made in natural language. There is a challenge in putting these definitions in a formal, precise, yet understandable language. Second, as in the LCOM example above, there are many subtleties in the interpretation of the basic terms which such a definition uses, regardless of whether this is definition is made in natural or formal language.

We argue that these two issues are fundamental to all patterns, regardless of their place in the abstraction level: (i) not much faith can be put in precise definitions which are not readable, and hence readily checked by humans, and, (ii) there must be no ambiguity in the meaning of basic, atomic terms which are used in higher-level definitions.

Our initial implementation of a program to identify μ-patterns was carried out in JAVA [1]. We are now in the process of migrating this implementation to a new language, JTL (acronym for the JAVA Tools Language, pronounced "Gee-Tel"), designed specifically for the purpose of formulating queries over JAVA code. JTL is described in detail in a concurrently published paper [6]. For now it suffices to say that JTL relies on the well understood and expressive semantics of the logic-programming paradigm, as well as on its laconic mode of speaking. Further, JTL's unique syntax gives it a *query by example* flavor. In many cases, the pattern for matching JAVA code looks just like this code itself.

The purpose of this paper is to describe the lessons learned from our experience in translating the natural language definitions (backed by the actual JAVA implementation of the pattern analyzer) into declarative, formal, yet very readable JTL equivalent.

Outline. We start with a brief tutorial of JTL (Sec. 2). Sec. 3 presents an overview of micro patterns, followed by Sec. 4 which provides a precise definition, expressed as a JTL query, for each of the micro patterns. In Sec. 5 we compare some of the JTL definitions with the ones used in our previous work, highlighting the ambiguities that are inherent in natural language. Finally, conclusions and our summary are presented in Sec. 6.

2. JTL - The Java Tools Language

JTL is a DATALOG [4]-like query language, whose basic constructs are *predicates*, also called *patterns*. Many JAVA keywords are primitive predicates; each such keyword matches JAVA language elements declared by it, e.g., the JTL predicate **`public`** matches public classes, interfaces and class members, whereas predicate **`interface`** matches interfaces. Some JTL primitives, such as `method`, are not JAVA keywords.

A space denotes conjunction; therefore, **`public`** `method` matches only public methods. Disjunction is denoted by a vertical bar, negation by an exclamation mark, operator precedence is the usual, while square brackets may be used to override precedence. For example, the expression

```
[public | protected] !static int
```

matches non-**`static`** class members (both fields and methods) of type **`int`** that are either **`public`** or **`protected`**. *Predicate definitions* are used to name expressions. For example,

```
instance := !static;
service := public instance method;
```

names the `service` predicate, which can now be used just like a primitive predicate in composing expressions. Predicate `service` is in fact part of JTL's rich standard library.

The predicates presented so far are *unary*, acting on a single value and returning true if the passed-in value is matched. Semantically, this value is represented by a special *subject* variable, denoted `#`, that is implicitly passed to invoked predicates. Thus `service` can be expressed in two different, more explicit ways:

```
service := #.public !#.static #.method; −−or:
#.service := #.public !#.static #.method;
```

JTL also supports binary (and higher-arity) predicates, which accept an explicit *argument* in addition to the implicit subject.

For example, the binary primitive predicate `declares[M]` holds if `#` is a class or interface which declares the member `M`. The JAVA keyword **`implements`** has a corresponding JTL predicate, **`implements`**`[I]`, that holds if `#` is a class that implements interface `I`. Similarly, **`extends`**`[S]` holds if `S` is a superclass of `#`.

As in DATALOG, variables, including arguments, always begin with an upper-case letter, whereas predicate names must begin in lower-case. The underscore symbol ("_") represents an unnamed variable, which is useful if we do not care about a certain position in the relation "returned" from a predicate.

Developers can define their own binary predicates; e.g., given

```
interfaceof[C] := C.class C.implements[#];           (1)
```

the expression `I.interfaceof[T]` holds if `T` is a class that implements interface `I`. Example (1) can be re-written in a more elegant manner with the *subject-chaining* operator, `&`:

```
interfaceof[C] := C.class & implements[#];
```

Also, JTL can recognize the parameters passed to a predicate based on the predicate's arity. Thus, the square brackets surrounding parameters, as well as the dot symbol (separating an explicit subject from a predicate) can usually be omitted:

```
interfaceof C := C class & implements #;
```

JTL employs variable binding, similar to that of DATALOG. The `common` predicate in the following example holds if `#` and `T` have a common super-interface:

```
common T := implements X, T implements X;           (2)
```

One can also use the `==` operator, or its alias **`is`**, to explicitly equalize two variables. This allows (2) to be written as:

```
common T := implements X, T implements Y, X is Y;
```

Quantification. Logic programming often uses recursion to realize existential quantifiers. JTL has a unique mechanism for carrying out such computations. For example, the following predicate matches classes which implement a non-**`public`** interface:

```
has_nonpublic_interface := implements: {
  exists !public;
};
```

The computation here involves two stages: (a) generating a set, and (b) applying a quantified condition to the entire set. The "`:`" character that follows the binary predicate **`implements`** turns this predicate into a *generator*, which returns the set of all `X`'s such that the expression `#.implmenets[X]` holds. Quantifier application is then carried out inside the curly brackets. Specifically, **`exists`** `!`**`public`** is a *set condition* which checks that the set of interfaces contains at least one member for which `!`**`public`** holds.

The curly brackets can be thought of as a loop iterating over the elements of the generated set. The current element of the iteration serves as the subject of the quantified condition. The internal curly brackets scope hides the subject variable of the external scope; therefore, the condition `!`**`public`** from the previous example will be evaluated against each of the values in the generated set.

JTL's has five basic quantifiers: **`has`** (also aliased as **`exists`**), the existential quantifier; **`all`**, the universal quantifier; **`no`** (alias: **`empty`**), negated existential quantifier; **`one`**, exactly one match; **`many`**, more than one match. A missing quantifier defaults to **`has`**. In addition, JTL offers set operators which compare two (or more) sets, e.g.,

```
good_encapsulation := class members: {
  field => private;
};
```

holds if every field has private visibility (a containment between the corresponding sets).

If a curly brackets scope has no preceding generator, then an (implicit) `members:` generator predicate is inferred. This generator produces all the fields, methods, constructors as well as the static initializer that are declared within the body of a class, regardless of their visibility level. This includes members which either override or hide inherited members (but excludes all inherited members).

Other standard predicates which are useful as generators include `protocol`—all non-**`private`** members of a class, including inherited ones, that were not overridden (or hidden) due to inheritance; `holds`—all members (including **`private`**-, inherited-, overridden- and hidden- members) that a class has; `offers`—similar to `holds`, but excludes the members that were declared in `java.lang.Object`.

Signature predicates. Signature predicates pertain to the signature of program elements, including the name, type, argument list, declared thrown exceptions and annotations (meta-data).

An *argument list predicate* is used for matching against elements of the list of arguments to a method. (Internally, such lists are stored using standard PROLOG [8]-like `head` and `tail` relations.) The most simple argument list is the empty list, which matches methods and constructors that accept no arguments. For example,

```
defaultCtor := constructor ();
```

matches only no-arguments constructors.

An asterisk ("`*`") in an arguments list predicate matches a sequence of zero or more types. Thus, the standard-library predicate

```
invocable := (*);
```

matches members which may take any number of arguments, i.e., constructors and methods, but not fields.

By using variable binding we write a predicate which requires that the two arguments of a protected method are of the same type:

```
  firstEq2nd := protected (X,X);
```

A *name predicate* is a name (or a regular expression) enclosed in single quotes. The closing quote can be omitted if there is no ambiguity. For example, the library predicate

```
pure_static := static !['serialVersionUID field];
```

matches static members, except the `serialVersionUID` field[1].

A *type predicate* specifies the JAVA type of a non-primitive class member. A type predicate is a regular expression preceded by a forward slash; e.g., predicate `/java.util.?*/ method` matches all methods with a return type from the `java.util` package (or its sub-packages). The closing slash is optional.

Hence, the expression `public _ (_, /String, *)` matches any public method that accepts a `String` as its second argument, and returns any type (but not constructors, which return no type).

Finally, Type predicates can also be used as actual parameters to binary (or higher-arity) predicates:

```
  interface extends /java.io.Serializable
```

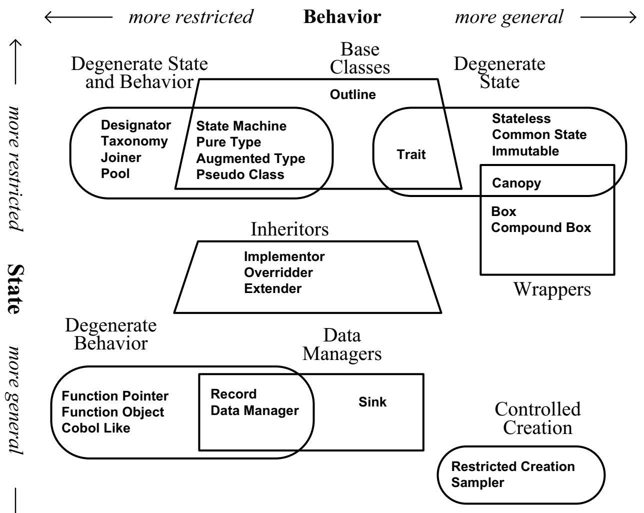
matches any interface that extends the standard `Serializable` interface.

Further definitions of predicates, as well as some language constructs, will be presented, as needed, in Sec. 4.

3. Micro Patterns

In a previous work [10] we defined 27 micro patters and studied their abundance in JAVA programs. In that work, the patterns were defined in a natural language (English) where each definition attempted to capture the lookup algorithm (of each pattern) in a few short sentences.

An overview of our catalog of micro patterns is depicted by the map in Fig. 3.1. The map presents the 8 categories of micro patterns and the placement of the 27 micro patterns into these.



Rounded rectangles denote pattern categories in which state, behavior, or construction is degenerate, rectangles denote categories of patterns for containment, while trapezoids denote patterns used for inheritance.

Figure 3.1. A map of the micro patterns catalog

The X-dimension of Fig. 3.1 corresponds to class behavior. Categories at the left hand side of the map are those of patterns which restrict the class behavior more than patterns which belong to categories at the right.

Similarly, the Y-dimension of the figure corresponds to class state: Categories at the upper portion of the map are of patterns restricting the class state more than patterns which belong to categories at the bottom of the map.

Examining the figure we see that the smallest category, with respect to the number of patterns, is the *Controlled Creation* which contains two patterns. The largest category is the *Degenerate State and Behavior* which has eight patterns.

The map highlights the fact that the categories are not mutually exclusive: Some patterns (E.g, Trait) belong to more than one category (*Base Classes* and *Degenerate State*.

Another issue, which is not evident from the map, is that of overlapping between patterns. A class may belong to two or more patterns at the same time. Class `java.beans.BeanDescriptor`[2], for example, is both an Extender and a Sink. In the data set that we used in [10] we found classes that were matched by as many as six(!) patterns.

4. Precise Description of Micro Patterns

We can now turn to a detailed description of each pattern, using JTL queries. This section is largely organized according to the categories of patterns which were presented in Fig. 3.1.

In order to eliminate overlapping between categories we decided that patterns that belong both to *Base Classes* and to any other category will be presented inside *Base Classes*. We also merged the *Degenerate State* and *Wrappers* categories, as well as the *Degenerate Behavior* and *Data Managers* categories. The resulting categorization has no overlaps.

For each category we give a short summary and then the JTL queries defining the patterns belonging to the category. We start with the inheritance-related categories, then move to the *Controlled Creation* category, followed by the degenerated classes categories

Inheritors. The conditions embodied by the patterns in this category examine the relationship between a class and its superclass.

An Implementor class implements abstract methods, an Overrider class overrides existing methods and Extender enriches the inherited interface. Note that the definitions of these patterns are mutually exclusive.

```
implementor := !abstract class {
    service => overriding M, M abstract;
    exists service;
  };

overrider:= !abstract class {
    service => overriding M, M concrete;
    exists service;
  };

extender := type {
    introduced[T] := T introduces #;
    service => !introduced _;
    exists service;
  };
```

Figure 4.1. Inheritors

The definition of `introduced` in the Extender pattern defines an auxiliary predicate which is visible only inside the scope of the curly brackets. Specifically `introduced T` holds if `#` was declared in `T`, without either hiding or overriding an inherited member.

Base Classes. This category includes six micro patterns capturing different ways in which a base class can make preparations for its subclasses.

[1] This special field is used by the JAVA serialization mechanism.

[2] taken from JAVA's standard library

```
trait := abstract offers: {
   abstract method;
   !abstract method;
   no instance field;
  };

state_machine := interface offers: {
   service => ();
   has service;
  };


augmented_type := abstract {
   constant := visible pure_static final field;
   constant => type_is T;
   many constant;

   instance method;
   no instance field;
   no private field;
   no !final real_static field;
  }
  offers: {
   no !abstract instance method;
  };

pure_type := abstract offers: {
   abstract method;
   no !abstract method;
   no field;
  };

pseudo_class := abstract class offers: {
   no instance field;
   no !abstract method;
  };

outline := abstract class is T {
   candidate := !abstract instance method
     | static method ! 'main;
   dispatch[M,M'] := holds M, protocol M',
     [ M' overrides M | M' hides M ];
   candidate invokes_virtual M,
     T.dispatch[M,M'], M' abstract;
  };
```

Figure 4.2. Base Classes

The `type_is` predicate in Augmented Type associates a member `#` with its type (a field's type or a method's return type). Thus, the primary requirement of the Augmented Type pattern is that a class will have two or more static final fields, which are all of the same type.

A Pseudo Class is a class which could be rewritten as an interface (ignoring static methods). A Pure Type is either an interface or a class which define only a set of operations. This implies that a class offers neither concrete methods nor static methods, nor fields, and that an interface offers no static fields.

The Outline patterns tries to capture the famous *template method* design pattern, whose intent is:

> "*Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure*" [9]

This was realized in JTL using two auxiliary predicates, `candidate` and `dispatch`.

Controlled Creation. The two patterns in this category match classes in which there is a special protocol for creating objects.

The first pattern prevents clients from creating instances directly. The second pattern provides clients with ready made instances.

Degenerate State and Behavior. This category includes those interfaces and classes in which both state and behavior are extremely degenerate. This degeneracy means, in most cases, that the class (or interface) does not define any variables or methods.

```
restricted_creation := # is T {
   no public constructor;
   static field type_is X,
     [T extends X | T implements X | T is X];
  };

sampler := # is T {
   public constructor;
   static field type_is X,
     [T extends X | T implements X | T is X];
  };
```

Figure 4.3. Controlled Creation

```
designator := abstract type {
   no method | field;
  };

taxonomy := type {
   no method;
   no field;
  }
   [ interface, interfaces: { one }
    | class, interfaces: { empty } ]

joiner := type {
   no field;
   no method;
  }
   [ interface, interfaces: { many }
      | class, implements _ ];

pool := type {
   no instance [field | method];
   no visible !final field;
   exist visible real_static field;
  };
```

Figure 4.4. Degenerate State and Behavior

Despite the severe restrictions imposed by these definitions, classes and interfaces which fall into this group are useful in tasks such as making and managing global definitions, class tagging, and more generally for defining and managing a taxonomy.

Degenerate State, Wrappers. The *Degenerate State* category pertains to classes whose instances have no state at all, or that their state is shared by several objects, or that they are immutable.

Wrappers are classes which wrap a central instance field with their methods. They tend to delegate functionality to this field. The main pattern in *Wrappers* is Box. The case that the wrapper protects the field from changes is covered by Canopy. In a Compound Box most of the state is maintained by a single non primitive field, where additional primitive fields holds auxiliary information.

The `putfield` predicate in Immutable associates a method with an instance field if the method puts a value in this field. It is conveniently named after the corresponding JVM instruction.

Degenerate Behavior, Data Managers. The degenerate behavior category relates to classes with no methods at all, classes that have a single method, or classes whose methods are very simple.

Data managers are classes whose main purpose is to manage the data stored in a set of instance variables.

The definition of the `getter` and `setter` auxiliary predicates (inside the Data Manager pattern) relies on a unique JTL mechanism, the **`SCRATCH`** values, which will be telegraphically explained here.

In `getter` we require that all values returned from a non-void, 0-parameters method are copied from `S`, where `S` is a value that is obtained (via `getfield`) from a field `F` of the enclosing class. The `from*` predicate is the reflexive-transitive closure of the scratch copying operator. Thus, the condition **`all`** `from* S` ensures that

```
box := !immutable, offers: { one instance field };

canopy := immutable, offers: { one instance field };

compound_box := offers: {
   one !primitive instance field;
   primitive instance field;
  };

immutable := class offers: {
   has instance field;
   no visible instance field;
   no instance method putfield _;
  };

stateless := class offers: {
   field => static final;
  };

common_state := class is T {
   !final real_static field
  } offers: {
   no instance [field | method];
  };
```

Figure 4.5. Degenerate State, Wrappers

```
function_pointer := !abstract class offers: {
   no field;
   one public instance method;
  };

function_object := !abstract class offers: {
   field;
   one public instance method;
  };

cobol_like := class offers: {
   no instance [field | method];
   one static method;
  };

record := !abstract offers: {
   public instance field;
   no !public instance field;
   no method;
  };

data_manager := class is C offers: {
   exist instance field;
   exist service;
   service => [setter | getter];

   getter := !void () returned: {
      all from* S, S getfield F, C holds F;
     };

   setter := void(_), C offers F, F field {
      putfield _ => putfield F, from* P,
        P parameter;
      exists putfield;
     };
  };

sink := class {
   no method invokes _;
  };
```

Figure 4.6. Degenerate Behavior, Data Managers

there is a chain of copy instructions from the value read from the field `F` to every value that is returned from the method.

The workings of `setter` are similar, but relate to the flow of data from the method's parameter to a field of the class.

Summarizing this section we note that the definitions presented here are much more concise and readable than those in [10]. This is further discussed in the next section.

5. Definition of Patterns

This section discusses the difficulties that arise when one defines patterns in a natural language. We will examine several micro pattern definitions, taken from the original paper [10] and compare them with the corresponding JTL queries.

First, we will consider the Function Pointer pattern, which represents an action that can be stored in a variable, passed to method, etc. In that sense, instances of Function Pointer classes are equivalent to pointer to functions in procedural languages, or to function values in functional languages.

This pattern was originally defined as:

> *"...classes which have no fields at all, and only a single* **`public`** *instance method"* [10]

A closer examination of this definition reveals a small problem: in JAVA every class is a subclasses of `java.lang.Object`. Therefore, every JAVA class recognizes methods such as `wait()`, `toString()` or `hashCode()` which are inherited from `Object`. In other words, there is no class with less than nine[3] public methods.

Another source of confusion arises from the statement "*no fields at all*". Is it ok for a Function Pointer class to extend a class that defines a private instance field? If we turn to the JTL definition for Function Pointer (in Fig. 4.6) we see that it is clear, concise, and unambiguous.

We will now examine the State Machine pattern which was originally defined as:

> *"An interface that defines only parameterless methods"* [10]

This definition is (again) ambiguous since it is not clear how inherited methods should be handled. However, there is another problem that is related to the logical condition expressed by the definition. This problem becomes obvious when we write the corresponding JTL definition

```
wrong := interface offers: { service => () };
```

Looking at the body of the JTL definition it is easy to see that the condition `service => ()` will trivially hold for empty sets. Obviously, this was not the intention of the State Machine pattern so we arrive at this correct version:

```
state_machine := interface offers: {
   service => ();
   has service;
};
```

Finally, we want to compare the JTL definition for the Restricted Creation pattern with the original JAVA code that we used in our pattern detector. The JAVA code is a 35-lines long method, which checks a JAVA class (passed in as a `JavaClass`[4] parameter) and returns **`true`** if the class is a Restricted Creation class.

First, we note that this code fragment is error prone since there is no support in JAVA for quantification of sets. Thus, the programmer has to manually code this logic via loops and other control-flow primitives. This poses difficulties not only during development, but also at the maintenance stage: if the conditions making the pattern needs to be changed it is not easy to modify the code correctly.

Second, it is very difficult to understand the pattern that this JAVA method embodies. In other words, we cannot use this piece of code to communicate the essence of the pattern to a human reader. This is due to the non-declarative nature of JAVA.

Comparing Fig. 5.1 with the JTL definition, we see that the JTL version is easier to develop, maintain and understand:

```
restricted_creation := # is T {
   no public constructor;
   static field type_is X,
     [T extends X | T implements X | T is X];
```

[3] as of JDK 1.5

[4] This type is defined by the Apache BCEL library.

```
boolean isRestrictedCreation(JavaClass jc) {
  HashSet supers = new HashSet();
  supers.add(jc);

  String spr = jc.getSuperclassName();
  if (spr != null)
    supers.add(spr);

  String[] is = jc.getInterfaceNames();
  for (int i = 0; i < is.length; ++i)
    supers.add(is[i]);

  boolean found = false;
  Field[] fs = jc.getFields();
  for (int i = 0; !found && i < fs.length; ++i) {
    if (!fs[i].isStatic())
      continue;

    if (supers.contains(fs[i].getSignature()))
      found = true;
  }
  if (!found)
    return false;

  Method[] ms = jc.getMethods();
  for (int i = 0; i < ms.length; ++i) {
    if (!ms[i].getName().endsWith(("init>")))
      continue;

    if (ms[i].isPublic())
      return false;
  }

  return true;
}
```

Figure 5.1. The JAVA code for detecting Restricted Creation classes

```
  };
```

6. Summary

Having seen the JTL definitions for micro patterns and some of the problems induced by natural language definitions, we are now in a good position to summarize the insights derived from the translation process.

We identify two primary problems that are caused by non formal definitions of patterns:

Imprecise terminology. The basic terms that are used in a natural language are not well defined. Thus we do not know whether a simple term such as "methods" include inherited methods, private methods, hidden methods, etc. The design of JTL overcomes this problem by these two complementing rules:

(i) Most JAVA keywords have a JTL predicate (with same name) capturing the semantics of the keyword (e.g., **`private`**, **`final`**, **`extends`**). Even if a keyword is implicit (such as **`abstract`** for interfaces), from JTL's point of view it is present;

(ii) Other relationships within the JAVA program are captured by the standard library's predicates. Most library predicates are a single-line JTL expression, thereby making the library simple and self explanatory.

Complicated Logic. Expressing complicated logical conditions (and especially quantification) in a natural language is difficult. Even the short description *"a static field or a method"* can be interpreted in two different ways. This problem is eradicated in JTL, since we have clear precedence rules and logical operators with well-defined semantics. Unlike other languages from the logical paradigm, complex logical conditions in JTL are terse and readable:

(i) JTL's implicit subject passing, combined with the subject chaining operator ("&"), eradicates more than half of the variables that are mentioned in a predicate's body (compared to the corresponding DATALOG predicate);

(ii) Built in quantifiers eliminate most of the recursive calls that are typical in logical programming.

Acknowledgements. We thank Grigory Fridberg for carrying out part of the implementation, and for his work on JTL's standard library manual.

References

[1] K. Arnold and J. Gosling. *The Java Programming Language*. Addison-Wesley Publishing Company, 1996.
[2] K. Beck. *Smalltalk: best practice patterns*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, first ed., 1997.
[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, 1996.
[4] S. Ceri, G. Gottlob, and L. Tanca. *Logic programming and databases*. Springer Verlag, New York, 1990.
[5] S. R. Chidamber and C. F. Kemerer. Towards a metrics suite for object oriented design. In *OOPSLA'91*.
[6] T. Cohen, J. Y. Gil, and I. Maman. JTL—the Java tools language. To appear in the proc. of OOPSLA'06, 2006.
[7] J. O. Coplien. *Advanced C++ Programmings Styles and Idioms*. Addison-Wesley Publishing Company, 1992.
[8] P. Deransart, L. Cervoni, and A. Ed-Dbali. *Prolog: The Standard: reference manual*. Springer-Verlag, 1996.
[9] E. Gamma *et al. Design Patterns: Elem. of Reusable OO Software*. Addison-Wesley Publishing Company, 1995.
[10] J. Gil and I. Maman. Micro patterns in Java code. In *OOPSLA'05*.
[11] M. Volter, A. Schmid, and E. Wolff. *Server Component Patterns: Component Infrastructures Illustrated with EJB*. John Wiley & Sons, Inc., 2002.