# Formal Patterns in Java Programs

**Itay Maman**

Technion - Computer Science Department - Ph.D. Thesis PHD-2012-05 - 2012

# Formal Patterns in Java Programs

Research Thesis

In Partial Fulfillment of the Requirements for the Degree of
Doctor of Philosophy

Itay Maman

Submitted to the Senate of
the Technion — Israel Institute of Technology

Tevet 5772 Haifa January 2012

This Research Thesis was done under the supervision of prof. Joseph (Yossi) Gil in the Department of Computer Science

The generous financial help of the Technion, and of IBM's PhD Fellowship program, is gratefully acknowledged.

*To my grandmother, Adèle Mamane née Toledano.*

# List of Publications


1. Joseph Gil and Itay Maman. Micro patterns in Java code. In Ralph Johnson and Richard P. Gabriel, editors, *Proceedings of the Twentieth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, San Diego, California, October 2005. ACM SIGPLAN Notices.

2. Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL—the Java tools language. In Peri L. Tarr and William R. Cook, editors, *Proceedings of the Twenty First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, Portland, Oregon, October 2006. ACM SIGPLAN Notices.

3. Tal Cohen, Joseph Gil, and Itay Maman. Guarded Program Transformations Using JTL. In Richard F. Paige and Bertrand Meyer, editors, *Proceedings of the Forty Sixth Conference on Objects, Models, Components, Patterns (TOOLS EUROPE 2008)*, volume 11 of *Lecture Notes in Business Information Processing*, Zurich, Switzerland, June 2008. Springer Verlag.

4. Joseph Gil and Itay Maman. Whiteoak: Introducing Structural Typing into Java. In Gail E. Harris, editor, *Proceedings of the Twenty Third Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'08)*, Nashville, Tennessee, October 2008. ACM SIGPLAN Notices.

# Contents

# List of Figures

# List of Tables

# Abstract

Acknowledging that the complexity of software is constantly growing, this work provides formal, precise tools geared towards aiding the developer in the exploration, maintenance and development of large bodies of software. These tools are centered around the notion of *formal patterns*, which, in a nutshell, are well-defined conditions on software modules.

We begin by presenting *JTL*, the Java Tools Language, which serves as the formalism for the definition of such patterns. JTL is a powerful—yet non Turing-complete—query language designed specifically for queries on JAVA programs. Its intuitive syntax and the simplicity of its underlying relational data model make JTL queries easy to write and read. A rich library of standard predicates allows JTL to examine declarations of packages, types, methods, fields, as well as the data flow within blocks of imperative statements. Using this formalism, we investigate three applications of formal patterns.

First, we present a comprehensive catalog of type-level formal patterns, *micro patterns*, each capturing a purposeful programming idiom. An empirical study allows us to make precise claims regarding the prevalence of the patterns and the ability of these to characterize the software containing them. The knowledge captured by the catalog, combined with the ability to detect these pattern mechanically, make the catalog valuable in the elicitation of design of software modules.

We then examine the utility of formal patterns in program transformation mechanisms, such as refactoring. Formal patterns are used in these as *guards*—the preconditions that decide whether/where a given transformation will be applied. Moreover, we show that an extension of JTL, which supports the production of output, is sufficiently powered for expressing transformations comparable to those of aspect-oriented programming or even generics.

Finally, we show how formal patterns can be used as first class citizens in a static type system: we describe the design and implementation of WHITEOAK, a JAVA extension that allows a programmer to define type-level formal patterns and use them just like standard types. The conformance semantics of these new types is that of structural equivalence. This allows the type system to accept a wider range of programs without compromising the language's static safety.

# Chapter 1

# Introduction

*"...Cuvier could correctly describe a whole animal by the contemplation of a single bone"*

— Sherlock Holmes, The Five Orange Pips

A skilled paleontologist is capable of deducing a great deal of information from a single piece of fossil remains. Our knowledge about the creatures that had walked on the face of this planet ages ago is derived, almost entirely, from the study of tiny variations in shape or size of body fossils.

Computer programs and dinosaurs are very different. It is therefore interesting to note that software engineers and paleontologist share a common goal: they want to learn meaningful facts about something they cannot touch, just from its physical remains. In paleontology we learn about ancient beings from their fossils. In software we learn about a program's *design* and its *runtime behavior* from its source code or from derivatives thereof (binary code).

It turns out that tracing back these properties from either source or binary code is a difficult problem. *Design* is manifested in the code, but it is covered by a layer of details so thick that the casual observer cannot usually see it. *Runtime behavior* is unambiguously specified by the code, but there is a firm limit on our ability to draw conclusions about a program's execution: ultimately, an impenetrable barrier, whose existence has been proven by Church and Turing, will be reached.

*Design patterns* [81], or even higher level patterns such as architecture-oriented patterns [45], are the obvious "suspects" for retracing design back from code. However, at these levels of abstraction, a pattern is merely a general strategy for balancing certain design forces. Such patterns are *polymorphic* in the sense that a single pattern may have multiple realizations, or as Christopher Alexander [8] puts it

> *"Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."*

No wonder that attempts to automate detection of design patterns yielded inconclusive results, typically with high rate of false negatives (see e.g., [42,105]). Indeed, as Mak, Choy and Lun [130] say, "*...automation support to the utilization of design patterns is still very limited*".

This work examines a new approach for reasoning about software. Instead of struggling with imprecision induced by inherently ambiguous patterns, we investigate, and exploit, a new kind of patterns; well defined ones.

We define the notion of *formal patterns*, which are a *simple*, ***well-defined*** *condition on the attributes, types, name and body of a software module or its components*. Relying on this notion, we

provide formal, precise tools geared towards aiding the developer in the exploration, maintenance and development of large bodies of software.

Providing the means for the formulation of such conditions is *JTL*[1], *the Java Tools Language*, a query language that allows one to effortlessly express elaborate requirements on various elements of JAVA [18] programs. Throughout this text, all formal patterns are expressed in terms of the JTL language.

With this underpinning in place, we show three different applications pertaining to a wide range of software construction activities: (*i*) exploration of existing code bases via automatic categorization of classes, using a catalog of 29 formal patterns; (*ii*) a generic mechanism for program transformation; and (*iii*) the integration of a restricted form of formal patterns into the JAVA language, yielding a more powerful type system which does not compromise JAVA's static type safety.

## 1.1 Formal Patterns

This section presents the full definition of formal patterns and discusses the various components of this definition.

**Definition 1** *A* formal pattern *is a well-defined condition on the attributes, types, name and body of a software module or its components, which is mechanically recognizable, purposeful, and simple.*

To be valuable, these patterns should not be random; they must capture a non-trivial *idiom* of the programming language which serves a *concrete purpose*. Yet, by definition, formal patterns stand at a lower level of abstraction than that of the classical collection of design patterns [81]. This is because formal patterns are tied to the implementation language and impose a condition on a single software module.

Micro patterns, presented in Chapter 3, are a special case of formal patterns in that the condition embodied in them applies to classes and interfaces. We propose the term *nano-patterns* for formal patterns which stand at the method or procedure level. The term *milli-patterns* can then be used for formal patterns at the package level (or to any other kind of class grouping).

**Recognizability.** The term *"mechanically recognizable"* means that there exists a Turing machine which decides whether any given module matches the condition. A condition such as "a method that delegates its responsibilities to others" is not recognizable, since the term "delegate" can have several interpretations. On the other hand, a predicate such as "a method which invokes a method of another class with the same name" can be automatically checked.

**Purposefulness.** By *purposeful* we mean that the condition defining a formal pattern characterizes modules which fulfill a recurring need in a specific manner. The condition that the number of methods is divisible by the number of fields does not constitute a pattern. In contrast, a pattern prescribing classes which have a single instance field that is assigned only once, at construction time, achieves programmer recognizable goals. For instance, it serves the purpose of managing a single resource by a dedicated object, a practice which Stroustrup [172] calls "resource acquisition is initialization".

**Simplicity.** The *simplicity* requirement is not only a matter of aesthetics: By sticking to first order predicate logic, we make it easier to automatically detect ill-defined formal patterns and to reason about these in general. In particular, the definition of a formal pattern should be, at the very least, decidable.

[1] Pronounced "Gee-Tel"

Together, these properties make formal patterns useful: they bring value to the manual work of the software engineer in capturing a common and meaningful idiom of the programming language. Traceability, expressed in the simplicity and recognizability properties, help in automating some of the engineer's work.

## 1.2 JTL

Our initial exploration of formal patterns started with an attempt to discover which class-level formal patterns are prevalent in real JAVA code. This attempt, which later evolved into our work on *micro patterns* (see Section 1.3.1, below), surfaced a thorny problem: the lack of means for precise description of code fragments. For instance, describing a certain family of classes as "classes that have no fields" may seem like a straightforward, unambiguous description. A careful examination reveals that this family is ill-defined, since it is not clear whether private (or even protected) fields defined at the superclass are allowed. In particular, it is not clear whether an empty class that inherits from a class that does declare fields is included in the family.

Another example is illustrated by the following criterion: "classes where methods do not call other methods". One can identify at least three sources of ambiguity in this definition. First, the definition is ambiguous with respect to inherited methods. Second, it does not specify whether constructors are also considered to be methods. Finally, if the answer to the last question is positive, then the definition leaves open the issue of initialization: a constructor may invoke another constructor, by virtue of a `this(...)` call.

This plethora of combinations and variants may be solved by defining a broad enough vocabulary. Alas, mathematical precision is based on minimalism. Formalisms that are derived from a large set atomic building blocks are usually difficult to use: precision is lost due to the inability of the human mind to accommodate more than a handful of terms.

In Chapter 2 we present a query language that was designed specifically for overcoming this predicament. This language, JTL, belongs to the family of logic programming and it sports a minimal core: it defines only a few language constructs from which a wide range of rich queries can be composed. The syntax was designed so that a JTL query often looks like the program element that it is intended to match. For instance, the JTL query `public void f()` matches public, void-returning methods, that are called "f" and take no parameters.

Aside from elegance of notation, JTL provides remarkable expressiveness and safety. First, the JTL compiler is capable of inferring the types of the parameters of every predicate. This makes JTL much like ML [139]: a statically typed language with type inference. The absence of type annotations aides in making JTL easy to write and read.

Second, JTL queries are not limited to declarations. The JTL standard library offers a set of predicates that allow one to query the *behavior* of methods. This is achieved by exposing the data flow of the method in question as a sequence of facts such as: "parameter $w$ is written to the field $f$ with receiver $x$", or "parameter $y$ is copied into temporary value $z$", etc. The JTL developer can query these facts, via standard predicates, as shown in the following query:

```
method {
  this get_field[F,V], V returned;
}
```

This query matches methods which return a value that was read from a field of the receiver object.

The features of JTL make it useful in a wide range of applications where formal patterns need to be defined. These include pointcut definition in aspect-oriented languages, smart search facilities in integrated development environments (IDEs), and even—as shown in Chapter 5—as a basis for the extension of Java (or other modern object-oriented languages).

**Publications.** JTL was originally introduced in the paper *JTL—The Java Tools Language* [57]. Our work was then cited by many others, in domains such as software-related query languages (see, e.g., [15, 62]), aspect-oriented programming [144, 162], pluggable type systems [13], etc.

Exploration of JTL's computability was the prime motivation for Cohen, Gil and Zarivach's *Datalog programs over infinite databases, revisited* [52], whose results were later used in this work. Parts of JTL's implementation were used in at least two works conducting an empirical survey of large software collections [83, 86].

## 1.3 Applications

Building on JTL as a basis, we have studied how formal patterns can be applied in various software construction activities. The results are reported in detail in Chapter 3–5. The following subsections provide a quick overview of these applications, each serving as a digest of the corresponding chapter.

### 1.3.1 Elicitation of Design

In order to better understand the actual building blocks of Java software, we defined a catalog of class-level formal patterns called *Micro Patterns*. The patterns in the catalog capture non-trivial properties of classes and thus form a mathematically-sound vocabulary for categorization of classes. As noted earlier, this was the original use-case for JTL.

A representative example is the Sampler pattern. This pattern defines classes which have a **`public`** constructor, but in addition have one or more **`static public`** fields of the same type as the class itself. The purpose of such classes is to give clients access to pre-made instances of the class, but also to let clients create their own. The Sampler pattern is realized by, e.g., class `Color` from package `java.awt` of the JAVA standard runtime environment, which offers a host of pre-defined color objects as part of its interface.

Empirical examination of a corpus of more than 70,000 classes reveals that the patterns in our catalog are abundant in production code. In particular, more then 45% of Java classes can be described by at least one of five simple micro patterns. This discovery suggests that simple, or even degenerated, classes are much more common in everyday code than what one usually expects. A direct implication is the fact that "classical" classes, carrying encapsulated mutable state and exposing rich behavior via methods, are not as common as the object-oriented programming literature suggests. These findings may change our thinking, and teaching, of modern software development.

The catalog also forms a vocabulary which comes in handy when discussing implementation strategies of classes. By using names of patterns one can express his intent, regarding the design of a class or a collaboration of a classes, in a clear, concise way.

Other results of our research in this area are statistical methods for classification of programs based on their micro pattern profile. Our approach of empirical examination of software corpora, which was inspired by Cohen and Gil's work on self calibration of code metrics [53], has gained significant popularity in recent years (see, e.g., [28, 48, 147, 160]). Perhaps this indicates that the community is acknowledging that software design has more in common with social sciences than what is commonly assumed.

**Publications.** Micro patterns were first described in *Micro Patterns in Java Code* [84]. The following is a partial rundown of works that directly use the notions and techniques from the original micro patterns paper.

Kim, Pan and Whitehead [119] examined the correlation between changes in the micro pattern of a class and the introduction of a bug into that class. The *Sourcerer* [23] project by Bajracharya, Linstead, et al. is a search engine for software collections. Its search mechanism is based on the notion of *fingerprints* that serve as a distilled, compact representation of a class. Micro patterns are one of the three kinds of information from which a fingerprint is derived. Singer and Kirkham [166] established the correlation between micro patterns and type name suffixes. They developed a tool that warns the programmer against possibly ill-named classes, by comparing the name chosen by the programmer against the micro patterns occurring in that class.

Marion, Jones, and Ryder [133] showed that certain micro patterns are highly correlated with an object's lifespan. They used this knowledge for improving the performance of generational garbage collection [177] by allowing objects that are expected to be long-lived, as determined by the micro patterns occurring in the object's class, to be allocated directly from the mature (old generation) space. This bypass around the nursery space allows the garbage collector to avoid redundant copy operations, resulting in performance improvements of 6-77%.

As was hypothesized in the original micro patterns paper, formal patterns pertaining to other kinds of software modules are also useful. For instance, Høst and Østvold [108] show that it is possible to mechanically determine whether a method's name and its implementation are likely to match each other (in a manner similar to that of Singer and Kirkham [166]). They do so by the detection of *Nano Patterns*, method-level Formal Patterns, and indeed mention our work as a "central source of inspiration".

### 1.3.2 Refactoring and Transformation

Automatic program transformation/translation mechanisms, such as the refactoring [77] engine built into modern Integrated Development Environments (IDEs), can be conceptually divided into two components: the *guard* [68], which describes the pre-requisites for the transformation, and the *transformer*, which is the clockwork behind the production of output for the matched code.

The obvious observation is that a guard is a formal pattern and thus can be expressed as a JTL expression (possibly with a custom library of standard predicates). More subtle is the observation that if JTL is augmented with output production facilities it can be used to express not only the guard, but also the transformer.

Specifically, we describe a side-effect-free technique of using the logic programming paradigm for the general task of program transformation. We show how this technique can be realized as an extension to JTL, and discuss language design issues related to this extension.

We demonstrate the utility of this extension by a variety of transformation examples, such as implementing generic structures (without erasure) in JAVA, a Lint-like program checker, and more. By allowing the transformation target to be a different language than the source (program translation), we show how one can easily solve tasks like the generation of database schemas, or XML DTDs, that match JAVA classes.

At the heart of this extensions lies the addition of one (or more) baggage parameters, specifying string values, to JTL predicates. These parameters are strictly output parameters, and thus can be thought of as the "output channels" of the predicate. A set of language-level rules dictate the manner by which the value of a baggage parameter is calculated from the terms making up a JTL predicate. The extension also includes several constructs which allow explicit manipulation of baggage values.

Together, these changes specify a formal semantics for the construction of strings from JTL calculations. The semantics is *constructive* in the sense that the baggage of an expression is a pure (side effect free) function of the enclosed terms. This lack of side effects makes it possible to marry together baggage manipulation constructs with JTL's original constructs without deviating

from the underlying logic paradigm. In particular, the extended JTL can still be reduced into DATALOG [47].

**Publications.** This extension is described in *Guarded Program Transformations Using JTL* [55], by Cohen, Gil, and Maman.

### 1.3.3 Structural Type Systems

A type in a programming language can be thought of as a condition on the set of runtime values. We argue that *a condition on the set of types, that is: a type-level formal pattern, is a useful type in its own right*.

This idea is reified by WHITEOAK, a JAVA extension that allows the user to define type-level formal patterns and use them just like standard types. Subtyping of these new types is structural: compatibility between two types is determined by their structure and not by an explicit nominal indication, a-la JAVA's `extends` or `implements` keywords. An interesting property of WHITEOAK is that every definition of a pure structural type—a structural type that does not provide default implementations for methods—is a valid JTL expression.

Structural subtyping addresses common software design problems and promotes the development of loosely coupled modules without compromising type safety. The resulting type system is capable of supporting self-referencing structural types, compile-time operators for computing new types from existing ones, as well as virtual constructors and non-abstract methods in structural types.

We describe implementation techniques, including compilation and runtime challenges that we faced (in particular, preserving the identity of objects). Measurement indicate that the performance of our implementation of structural dispatching is comparable to that of the JVM's standard invocation mechanisms.

**Publications.** WHITEOAK was first described in *Whiteoak: Introducing Structural Typing into Java* [85]. It was then used in the work of Malayeri and Aldrich [132] which conducted an empirical study of potential structural conformance of types in (nominally typed) JAVA programs. In order to demonstrate the benefits of using a hybrid structural-nominal type system, the authors have translated two of the programs in their data set into WHITEOAK.

In addition, WHITEOAK's technique for dispatching a method on a structurally typed receiver is one of the two main techniques examined in a paper by Dubochet and Odersky [69] which analyzes performance of structural dispatching in the context of the SCALA [152] language.

This work concludes (Chapter 6) with several promising directions for future research based on the ideas presented herein.

# Chapter 2

# JTL

our discussion of formal patterns begins with the tool used to define them, in an exact and, obviously, formal manner. JTL (the *Java Tools Language*) is a declarative language, belonging to the logic-programming paradigm, designed for the task of selecting JAVA program elements. Two primary applications were in mind at the time when the language was first conceived:

(a) Precise conditions on program elements, thus making it suited for defining formal patterns, and, in particular, micro patterns.

(b) Join-point selection for aspect-oriented programming, where JTL can serve as a powerful substitute of ASPECTJ [117]'s pointcut syntax.

As JTL took shape and matured it became clear that it can be used for a wider range of applications, including code lookup, generic programming's concepts, detection of coding violations, and more.

Ultimately, JTL's versatility lies in its being a *terse yet intuitive formalism for the expression of formal patterns* . Its intuitive notation makes it valuable in interactive applications where a user is expecting a quick answer for a quick question. Clearly, imposing the use of a fancy, verbose, language on such users is infeasible.

JTL's focus is on the modules in which the code is organized: packages, classes, methods and variables, including their names, types, parameters, accessibility level and other attributes. JTL can also inspect the interrelations of these modules, including questions such as which classes exist in a given unit, which methods does a given method invoke, etc. Additionally, JTL can inspect the imperative parts of the code by means of dataflow analysis.

This chapter discusses the following aspects of JTL: fundamentals, inspection of imperative instructions, semantics, applications, implementation, design considerations, and conclusions (Section 2.1—2.7, respectively). Chapter 4 presents an extension to JTL which is capable of transforming its input.

A full description of JTL will not be complete without its standard library. Unfortunately, listing all predicates of this library within the scope of this chapter may disrupt the natural flow of the discussion, so it is deferred to Appendix B. The reader is therefore invited to examine the library in order to get a better appreciation of JTL's expressiveness and elegance.

In contrast to the lavish library design, the minimal core of the language is summarized in as little as four pages of Appendix A, followed by six pages of built-in set operators which can be equated to library predicates.

**Three Introductory Examples.** JTL syntax is terse and intuitive; just as in AWK [3], one-line programs are abundant, and are readily wrapped within a single string. In many cases, the JTL *pattern* for matching a JAVA program element looks exactly like the program element itself.

For example, the JTL *predicate*[1]

```
public abstract void ();
```

matches all methods (of a given class) which are abstract, publicly accessible, return **`void`** and take no parameters. Thus, in a sense, JTL mimics the *Query By Example* [185] idea.

Even patterns which transcend the plain JAVA syntax should be understandable, e.g.,

```
abstract class {
   [long | int] field;
   no abstract method;
};
```

matches abstract classes in which there is a field whose type is either **`long`** or **`int`**, and no abstract methods.

The first line in the curly brackets is an *existential quantifier* ranging over all class members. The second line in the brackets is a negation of an existential quantifier, i.e., a *universal quantifier* in disguise, applied to this range.

JTL can also delve into method bodies, by means of intra-procedural dataflow analysis, similar to that of the class file verifier [127, Sec. 4.9.1]. Consider for example *cascading methods*, i.e., methods which can be used in a cascade of message sends to the same receiver—an idiom that is frequently used in the implementation of a fluent interface [115]. A typical example can be found in the following JAVA statement

```
(new MyClass()).f().g().h();
```

in which `f`, `g` and `h` are methods of `MyClass`. Then, the following JTL pattern matches all cascading methods:

```
instance method !void {
   all [ !returned | this ];
};
```

The curly brackets in the above pattern denote the set of values (including temporaries, parameters, constants, etc.) that the method may generate or use. The statement inside the brackets is a requirement that all such values are either not returned by the method, or are equal to **`this`**, and therefore guarantees that the only possible value that the method may return is **`this`**.

## 2.1 Fundamentals

This section covers the fundamental constructs of JTL, assuming some basic familiarity with logic programming. These constructs are related to queries on declarations of classes, methods, constructors, and fields. Constructs for queries on bodies of methods/constructors are the subject of the next section.

A JTL program is a set of definitions of named logical *predicates*. Execution begins by selecting a predicate to execute as the *goal*. As in PROLOG [66], predicate names start with a lower-case letter, while *variables* and parameters names are capitalized. Identifiers may contain letters, digits, or an underscore. Additionally, the final characters of an identifier name may be "`+`" (plus), "`*`" (asterisk), or "`'`" (single quote). Two consecutive dash signs, `--`, mark the start of a comment which spans to the end of the line.

An application of a predicate is called a *term*. Each term includes, at least, a predicate's name along with the zero or more actual parameters (either variables or literals), as per the predicate's arity. A JTL expression is a composition of terms via operators. The JTL runtime system evaluates

[1] The terms "predicate" and "pattern" are used almost interchangeably; "pattern" usually refers to a unary predicate.

an expression by (lazily) evaluating its terms and then applying the operators connecting them to get the expression's result.

As shown in the examples above, JTL extends the logic paradigm with constructs such as arguments list patterns and quantifiers which make it possible to achieve many programming tasks without recursion that is so common in logic programming.

JTL is strongly typed. Every runtime value is associated with exactly one type, henceforth: *kind*, throughout its lifetime. The two most important kinds of JTL are (*i*) **`MEMBER`**, which represents all sorts of class and interface members, including function members, data members, constructors, initializers and static initializers; and (*ii*) **`TYPE`**, which stands for JAVA **`class`**es, **`interface`**s, **`enum`**s, arrays, as well as JAVA's primitive types such as **`int`**.

Another important kind is **`SCRATCH`**, which, as the name suggests, stands for a temporary value used or generated in the course of the computation carried out by a method. Scratches originate from a dataflow analysis of a method, and are discussed below at Sec.2.2.

JTL employs a kind inference engine which makes it unnecessary to specify kind annotations when declaring variables (including parameters).

### 2.1.1 Simple Patterns

Many JAVA keywords are native patterns in JTL, carrying essentially the same semantics. For example, the JAVA keywords **`interface`**, **`public`** are also JTL patterns which match all types declared as interface, all program elements of public visibility (respectively). Henceforth, our examples shall use these keywords freely; no confusion should arise.

Not all JTL natives are JAVA keywords. A simple example is `anonymous`, defined on **`TYPE`**, which matches anonymous classes.

Some patterns (like `abstract`) are overloaded, since they are applicable both to types and members. Others are monomorphic, e.g., `class` is applicable only to **`TYPE`**.

Another example is pattern `type`, defined only on **`TYPE`**, which matches all values of **`TYPE`**. This, and the similar pattern `member` (defined on **`MEMBER`**) can be used to break overloading ambiguity.

JTL has two kinds of predicates: *native* and *compound*. Native predicates are predicates whose implementation is external to the language. In other words, in order to evaluate native predicates, the JTL processor must use an external software library that directly inspects the input. Native patterns hence are declared (in a pre-loaded configuration file) but not defined by JTL.

In contrast, compound patterns are defined by a JTL expression using logical operators. The pattern `public, int` matches all **`public`** fields of type **`int`** and all **`public`** methods whose return type is **`int`**. As in PROLOG, conjunction is denoted by a comma. In JTL however, the comma is optional; patterns separated by whitespace are conjuncted. Thus, the latter can also be written as `public int`.

As a matter of style, the JTL code presented henceforth denotes conjunction primarily by whitespace; commas are used for readability—breaking long conjugation sequences into subsequences of related terms. Disjunction is denoted by a vertical bar, while an exclamation mark stands for logical negation. Thus, the expression

```
public | protected | private
```

matches JAVA program elements whose visibility is not default, whereas `!public` matches non-**`public`** elements.

Logical operators obey the usual precedence rules, i.e., negation has the highest priority and disjunction has the lowest. Square parenthesis may be used to override precedence, as in

```
!private [byte|short|int|long]
```

which matches non-**`private`**, integral-typed fields and methods.

A *predicate definition* associates an expression with a name. After making the following definition,

```
visible := !private;
```

the newly defined predicate, `visible`, can be used anywhere a native pattern can be, as in e.g.,

```
service := visible method;
```

JTL has a rich standard library (See Appendix B) of pre-defined predicates including predicates such as `method`, `constructor`, `primitive`, `visible` (as defined above), and many more.

### 2.1.2 Signature Patterns

Signature patterns pertain to names, to types of members, and to argument lists of methods and constructors.

**Name Patterns.** A *name pattern* is a regular expression preceded by a single quote, or a previously-declared name. Standard regular expression operators are allowed, except that the wildcard character is denoted by a question mark rather than a dot. Name literals and regular expressions are quoted with single quotes. The closing quote can be omitted if there is no ambiguity.

For example, `void 'set[A-Z]?*' method` matches any **`void`** method whose name starts with "`set`" followed by an upper-case letter.

If the name pattern does not contain any regular expression operators, as in

```
toString_p := 'toString method;
```
(2.1)

then the pattern can be made clearer by using a **`name`** statement to declare `toString` as name and get rid of the quote. Thus, an alternative definition of (2.1) is

```
name toString;
toString_p := toString method;
```

In truth, the above is redundant, since an implicit **`name`** statement pre-declares all methods of the JAVA root class `java.lang.Object`. All in all, The **`name`** construct allows JTL code to match identifiers in JAVA code while avoiding the `'` symbol which will clutter the JTL text and hinder its similarity to the JAVA code it attempts to capture.

**Type Patterns.** *Type Pattern* makes it possible to specify the JAVA type of a non-primitive class member. Syntactically, a type pattern is forward slash followed by a fully qualified name (a dot separated sequence of one or more JAVA identifiers) of a JAVA type. The expression `/java.util.List method` matches all methods whose return type is `java.util.List`.

Note that a name pattern is a regular expression and thus specifies a possibly unbounded set of matching values. A type pattern, on the other hand, uniquely designates a single JAVA type, that is: a single JTL value. This makes it possible to use a type pattern also as a *literal* (see (2.3), below).

The forward slash is not necessary for type names which were previously declared as such by a **`typename`** declaration, as follows:

```
typename java.io.PrintStream;
printstream_field := PrintStream field;
```

Many of the types (including classes, interfaces and enumerations) declared in the `java.lang` package are pre-declared as type names, including `Object`, `String`, `Comparable`, and the wrapper classes (`Integer`, `Byte`, `Void`, etc.).

Here is a redefinition of `toString_p` pattern (2.1), which ensures that the matched method returns a `String`.

```
toString_p := String toString method;
```
(2.2)

**Argument List Patterns.** JTL provides special constructs which all but eliminate recursion. An important example is *arguments list patterns*, used for matching against elements of the list of arguments to a method.

The most simple argument list is the empty list, which matches methods and constructors that accept no arguments. Here is a rewrite of (2.2) using such a list:

```
toString_p := String toString ();
```

(Note that the above does not match fields, which have no argument list, nor constructors, which have no return type.)

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

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

matches members which may take any number of arguments, i.e., constructors and methods, but not fields, initializers, or static initializers. An underscore ("_") is a single-type wildcard (as it is essentially a uniquely named variable). Hence, `public (_, int, *);` matches public methods that accepts an **`int`** as its second argument, and returns any type.

Finally, type patterns can be used inside an argument list pattern, as type literals. Thus,

```
public (int,/java.util.List);                                    (2.3)
```

matches any public method or constructor that accepts an **`int`** as its first argument, and a `java.util.List` as its second parameter (and returns any type).

### 2.1.3 Variables and Higher Arity Predicates

It is often useful to examine the program element which is matched by a pattern. JTL employs variable binding, similar to that of PROLOG, for this purpose. For example, by using variable `X` twice, the following pattern makes the requirement that the two arguments of a method are of the same type:

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

**The Subject Parameter.** Most predicates presented so far were parameterless in the sense that we use them without specifying an actual parameter, as in:

```
encapsulated := private field;
```

Still, the expression `private` that appears inside `encapsulated` is not a constant expression: it will return different results based on the value passed to `encapsulated`. The ability of an expression that takes no (explicit) parameters to yield different results is achieved via the *subject* mechanism. In JTL, all predicates have a hidden argument, the *subject*, also called the *receiver* which can be referenced via the reserved symbol #.

When a predicate calls another predicate, the default is that the subject of the caller becomes the subject of the callee. Specifically, an evaluation of `encapsulated` with some subject value, $s_0$, will conjugate the results obtained from the evaluation of `private` and `field`, both with # bounded to $s_0$.

The subject parameter allows composite, non-constant, expressions to be written with minimal (or no) use of explicit parameters. We note that the semantics of JTL's subjects resembles that of the self (or **`this`**) parameters found in many object-oriented languages. When a method in some object wants to invoke another method *of the same object* it does not need to specify a receiver object for the invocation.

**Predicates with Explicit Parameters.** JTL also supports predicates which take explicit parameters (in addition to the implicit subject). Consider the library predicate `implements` which takes a single explicit parameter, `T`. This predicate holds if `T` is a direct superinterface of the subject.

Formally, the evaluation of a predicate always returns a relation—where each tuple position corresponds to one of the parameters, including the subject—of all tuples that (*i*) satisfy the criteria captured by the predicate, and (*ii*) match the values of the known parameters.

For a given subject value $s$, and an indefinite value of `T` the expression `implements T` yields a relation $\{\langle s, i_1\rangle, \langle s, i_2\rangle, \ldots \langle s, i_n\rangle\}$ where $i_1, i_2, \ldots i_n$ are all the interfaces that appear in the **`implements`** clause of $s$'s declaration[2]. Conversely, for an $s$ value of the subject and a $t$ value of the variable `T`, `implements T` yields the singular relation $\{\langle s, t\rangle\}$ if $t$ is a superinterface of $s$ or the empty relation otherwise.

A parameter in a JTL definition can be thought of as *either* input or an output parameter. The predicate will either produce all correct assignments into the parameters (if no value is specified for it) or will confirm/reject its value (if one is specified for it). This behavior is in accordance with the standard semantics of parameters in logic programming.

Here is an example of a user-defined JTL predicate with one explicit parameter holding for an abstract class and its immediate superinterface(s):

```
abstract_and_implements T := abstract implements T;
```

This predicate define a single explicit parameter, `T`, by placing it immediately before the `:=` symbol. The body of the predicate prescribes the conjunction of two terms: `abstract` which requires that the subject is not concrete, and `implements T` which selects into `T` all immediate superinterfacs of the subject.

Note that some restrictions on computability do apply. Certain predicates cannot be evaluated when one (or more) of their parameters are not specified. Computability issues are discussed in Section 2.3.2.

**Explicit Subject Application.** By default, the subject of the caller becomes the subject of the callee. This default behavior can be overruled by placing a variable in a prefix position with respect to the invoked predicate. This is illustrated by the following pattern that matches classes whose superclass is **`abstract`**:

```
extends_an_abstract_class := extends X, X abstract;
```

In this example, the term `extends X` selects into `X` the superclass of the subject (`extends` is a binary library predicate capturing the immediate subclassing relationship). In `X public` we place `X` in a prefix position thereby evaluating the `public` term with `X` being the subject.

Despite using the variable `X`, `extends_an_abstract_class` does not declare `X` is a parameter. The returned relation will be unary: It will contain a single unary tuple of the subject if such an `X` was found, or will be empty otherwise.

Square brackets are used also to wrap the list of actual parameters and formal parameters of a predicate. When the list contains a single parameter (which is the case in all examples presented so far) these brackets are optional. JTL also allows the use of an optional dot notation for separating the receiver passed to a predicate from the predicate's name. Thus,

```
abstract_and_implements[T] := #.abstract #.implements[T];
extends_an_abstract_class := #.extends[X], X.abstract;
```

are equivalent to their earlier forms, above. In these latter versions we also made explicit the default passing of the subject.

**Subject Change Operator.** The ampersand symbol, `&`, is a shorthand for repeated application of an explicit subject. Thus, the following expression:

```
extends S, S public S abstract S implements I
```

can be rewritten as

```
extends S, S public & abstract & implements I
```

[2]This type of invocation is typically described in this text as "*selects into T*".

**Literals.** Literals can be passed as actual parameters in the same manner as variables. For example, one can replace the variable `M` in the following expression:

```
interface extends M
```

with the literal `/java.io.Serializable`

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

This changes the semantics of the expression from "matches any interface that extends `M`" to "matches any interface that extends the `Serializable` interface."

**Comparison.** JTL library predicate `is` can be used for demanding the equality of the subject and the single parameter. Thus, the last expression can also be written in a more verbose way, as follows:

```
interface extends M, M is N, N is /java.io.Serializable
```

Note that `is` is merely a (native) binary predicate which computes the relation of all pairs $\langle x, x \rangle$.

**Library Predicates.** Here are some native binary predicates from the JTL standard library: `declares M`, which is also aliased as `members M`, holds when `M` is one of the subject's members (not including inherited members); `offers M`, is similar to `declares` except that it includes inherited members; `overriding M`, is true when `M` is overridden by the subject. Figure 2.1 shows some of the compound predicates built on top of these natives.

**Figure 2.1** Some of the standard predicates of JTL

```
inherits M := offers M !declares M;
declared_by T := T declares #;
precursor M := M overriding #;
extends+ C := extends C | extends C' C' extends+ C;
extends* C := C is # | extends+ C;
```

The figure makes apparent the JTL naming convention by which the reflexive transitive closure of a predicate `p` is named `p*`, while the anti-reflexive closure variant is named `p+`.

It is interesting to examine the "recursive" definition of one of these predicates, e.g., `extends+`:

```
extends+ C := extends C | extends C' C' extends+ C;
```

It may appear at first that with the absence of a halting condition, the recursion will never terminate. A moment's thought reveals that this is not the case. The semantics of this recursive definition is not of stack-based recursive calls, but rather, as customary in DATALOG, that of a work-list approach for generating facts until a fix point is reached.

**Predicate Name Aliases.** The name `extends+` suggests that it is used as a verb connecting two nouns. As mentioned above, we can even write

```
C' extends+ C
```

But, the same term can be used in situations in which it more natural to see it as a query that answers the set of *all* classes in the subclassing chain of `C'`. A more appropriate name for these situations is `ancestors`. It is possible to make another definition

```
ancestors C := extends+ C;
```

To promote meaningful predicate names, JTL offers what is known as *predicate name aliases*, by which the same predicate definition can introduce more than one name to the predicate. The definition of `extends+` has such an alias

```
extends+ C := extends C | extends C', C' extends+ C;
      Alias ancestors;
```

The use for an alias named `ancestors` will become clear with the presentation of predicate `all_parents_are_nice` below.

Native predicates can also have aliases, which are specified along with their declaration.

### 2.1.4 Quantification

Although it is possible to express universal and existential quantification with the constructs of logic programming, we found that the alternative presented in this section is more natural for the particular application domain.

Consider for example the task of checking whether a JAVA class has an **`int`** field. A straightforward, declarative way of doing that is to examine the set of all of the class fields, and then check whether this set has a field whose type is **`int`**.

The following predicate does precisely this, by employing a *quantification scope*:

```
has_int_field := class members: {
    exists int field;
  };
```

Evaluation of the expression `members: { exists int field; }` starts by generating set, $G$, of all possible members `M`, such that `# members M` holds. (The "`members:`" portion of the query is called the *generator*.)

$G$ is then passed to each of the *set queries* declares inside the curly braces scope (here, **`exists`** `int field;`). The entire scope holds if all set queries hold for the generated set.

A set query is composed of a *set condition* (here, **`exists`**) and a *subset expression* (here, `int field`). Evaluation of an individual set query goes as follows: First, the set $S$, the subset of $G$ for which the subset expression holds, is calculated. This is achieved by binding the subject to each of the values of $G$ and evaluating the subset expression with that new subject. Note that rebinding of the subject is in effect only within the curly braces scope. Outside this scope, the value of the subject remains intact.

At the second stage, the set $S$ is inspected by the set condition of the query. Here the set condition is the existential quantifier so it checks that $\|S\| \geq 1$. Given that the scope here contained only one set query, we have that the entire scope holds if an `int field` exists.

The next example shows two other kinds of set queries.

```
all_parents_are_nice := class ancestors: {
    all public;
    no abstract;
  };
```

The evaluation of this pattern starts by computing the generator. In this case, the generator generates the set of all classes that the subject **`extends`** directly or indirectly, i.e., all types `C` for which `# ancestors C` holds (recall that `ancestors` is an alias for `extends+`). The first query checks whether all members of this set are `public`. The second quantifier succeeds only if this set contains no `abstract` classes. Thus, `all_parents_are_nice` matches classes whose superclasses are all public and concrete.

Formally, the set condition **`all`** asserts that $\|S\| = \|G\|$, where **`no`** asserts that $\|S\| = 0$. The list of set conditions supported by JTL includes, among others, **`many`** $p$ that holds if the generated set has two or more elements ($\|S\| \geq 2$) for which the expression $p$ holds; and **`one`** $p$ that holds if the generated set has precisely one such element ($\|S\| = 1$).

The existential operator is the most common; hence the **`exists`** is optional. Also, a missing generator (in predicates whose subject is a **`TYPE`**) defaults to the `members:` generator. Hence, a concise rewrite of `has_int_field` is

```
has_int_field := class { int field; };
```

Some set condition are binary: they examine two subsets of $G$ and thus require two subset expressions.

```
all_fields_are_private := class { field implies private; };
```

In here, **implies** is a binary set condition whose subset expressions are `field` and `private`. Evaluation of this query calculates two subsets of $G$: $S_1$ which is all elements of $G$ for which `field` holds; and $S_2$ which is all elements of $G$ for which `private` holds.

**implies** simply checks for inclusion, that is: $S_1 \subseteq S_2$.

The last set condition that will be discussed here is **partition**. This operator has variable arity: it can operate on any number of subset expressions (greater than one). Syntactically, the subset expressions are specified as list separated by a pair of commas, "`,,`", as shown below:

```
text_book_class := class {
     partition
       private field,,
       protected abstract method,,
       public !abstract method,,
       constructor;
   };
```

In `text_book_class` we require that every member of the class will belong to exactly one of these four categories: private fields; protected abstract methods; concrete public methods; or constructors.

Formally, the **partition** operator holds for a set $G$ and a set of subsets of $G$: $\{S_1, S_2, \ldots, S_n\}$ if $S_1 \cup S_2 \cup \ldots S_n = G$ and for all pairs $1 \leq i, j \leq n$ such that $i \neq j$ we have $S_i \cap S_j = \phi$.

We conclude this discussion of quantification scope with local definitions. It is often useful to define predicates which are only visible within a quantification scope. Such local definitions promote readability and reuse (across several set queries) while avoiding the cluttering of the global namespace.

Local definitions are indicated by the **let** keyword:

```
p := class {
     let itm := method (*,int,*); -- int taking method
     one public itm;
     no private itm;
   };
```

Here we define `itm` as a local predicate. It subsequently used in the two set queries: **one** `public itm`, and **no** `private itm`.

## 2.2 Inspection of Imperative Code

Now that the bulk of the language syntax is described, we can turn to the question of inspection of *imperative entities*. To an extent, queries of these entities are mostly a matter of library design rather than a language design. Recall that JTL native predicates are implemented as part of the supporting library that the JTL processor uses for inspecting JAVA code. Extending this library, without changing the JTL syntax, can increase the search capabilities of the language.

Section 2.2.1 shows how by adding a set of native predicates, JTL *can* be extended to explore an abstract syntax tree representation of the code. This section also explains the shortcoming of this approach. We chose instead to implement a mechanism for the inspection of the dataflow graph of methods, as described in Section 2.2.2. Standard predicates exploiting this mechanism are described in Section 2.2.3.

### 2.2.1 Abstract Syntax Trees and JTL

Executional code can be represented by an abstract grammar, with non-terminal symbols for compound statement such as **`if`** and **`while`**, for operations such as type conversion, etc. One could even think of several different such grammars, each focusing on a different perspective of the code.

Code can be represented by an abstract syntax tree whose structure is governed by the abstract grammar. To let JTL support such a representation, we can add a new kind, **`NODE`**, and a host of native relations which represent the tree structure. For example, a native binary predicate `if` can be used to select **`if`** statement nodes and the condition associated with it; a binary predicate `then` can select the node of the statement to be executed if the **`if`** condition holds; another binary predicate, `else`, may select the node of the statement of the other branch, etc.

As an example of an application for such a representation, consider a search for cumbersome code fragments such as

```
if (c)
   return true;
else
   return false;
```

with the purpose of recommending to the programmer to write

```
   return c;
```

instead. The following pattern matches such code:

```
boolean_return_recommendation :=
   if _ then S1 else S2,
     S1 return V1,
     S2 return V2,
      V1 literal "true",
      V2 literal "false";
```

The above pattern should be very readable: we see that its subject must be a **`NODE`** which is an **`if`** statement, with a don't-care condition (i.e., _), which branches control to statements `S1` and `S2`; also both `S1` and `S2` must be **`return`** statements, returning nodes `V1` and `V2` respectively. Moreover, the patterns requires that nodes `V1` and `V2` are literal nodes, the first being the JAVA **`true`** literal, the second a **`false`**.

In principle, such a representation can even simultaneously support more than one abstract grammar. Two main reasons stood behind our decision not to implement the set of native patterns required for letting JTL explore such a representation of the code.

1. *Size.* Abstract grammars of JAVA (just as any other non-toy language) tend to be very large, with tens and hundreds of non-terminal symbols and rules. Each rule, and each non-terminal symbol, requires a native definition, typically more than one. The effort in defining each of these is by no means meager.

2. *Utility.* Clearly, an AST representation can be used for representing the non-imperative aspects of the code. The experience gained in using the non-AST based representation of JTL for exploring these aspects, including type signatures, declaration modifiers, and the interrelations between classes, members and packages, indicated that the abstraction level offered by an abstract syntax tree is a bit too low at times.

A third, (and less crucial) reason is that it is not easy (though not infeasible) to elicit the AST from the class file, the data format used in our current implementation.

### 2.2.2 Inspection of Dataflow via Scratches

In the course of execution of imperative entities many temporary values are generated. Dataflow analysis studies the ways that these values are generated and transferred. The idea is similar to dataflow analysis as carried out by an optimizing compiler [4, Sects. 10.5–10.6], or by the JAVA bytecode verifier [127, Sec. 4.92]. This mechanism allows JTL to inspect the behavior of methods, without running into the problems, described in Sec.2.2.1, of AST queries.

To implement dataflow analysis, we introduce a new JTL kind, **`SCRATCH`**, which represents a location in the method where a new value is computed, that is: pushed onto the operands stack or assigned to entry of the *local variable array* (LVA).

In fact, the set of scratches of a method corresponds to what is known in the compiler lingo as the *static single assignment form* [10] which is a graph that represents the computation carried out by a piece of imperative code, in which every variable is assigned-to exactly once.

The JTL runtime detects all scratches of a method as well as all computation steps involving each of these scratches. Typical steps are the assignment into a scratch (from one of the following sources: another scratch, an input parameter value, a constant, a field, a value returned from a method or a code entity, an arithmetical operation, or a thrown exception); the passing of a scratch as a parameter; assignment of a scratch into a field; a scratch being thrown as an exception; or a scratch being returned from a method.

The steps in which a scratch is involved form a set of facts about it. These facts can be inspected by JTL via a set of library predicates, in a similar manner to the inspection of class or member declarations. JTL's scratch detection algorithm takes into account merges of control flow. For example, a scratch that is arithmetically computed from a scratch on the operands stack is considered to be computed from every scratch that may be pushed onto this stack location.

The following predicate gives a quick taste of the manner of using data flow information in JTL.

```
fluent_method := !void instance method {
    returned implies this;
  };
```

The first line in the above requires that the subject is a non-void instance method. We then make the condition that every value returned by this method is equal to the **`this`** parameter. We do that by a quantification scope that asserts that the set of all scratches returned from the the method is a subset of the set of all scratches that are equal to **`this`**. Note that when the subject is a method, the default generator generates the set of all scratches of the subject.

Dataflow analysis is a large topic. Still, no new language constructs were needed to allow JTL to deal with this topic: the scratch facility is implemented strictly as a library-level facility. The remainder of this section will present some of the library predicates pertaining to this topic.

Table 2.1 lists the essential unary predicates defined on scratches. The text in the *meaning* column specifies the condition that a subject scratch must maintain in order for the predicate to yield true on this subject.

The binary predicate `scratches S` selects into `S` the scratches of the method #, and serves as the default generator for methods. The binary predicate `typed T` selects into `T` the type of the subject scratch.

The following predicate returns the set of all types that a method uses:

```
use_types T := method scratches S, S typed T;
```

The most important predicate connecting scratches is `from S`, which holds if scratch `S` is assigned to scratch #. Similarly, `func S` holds if # is computed by an arithmetical computation. As usual, `from*`, `func*` denote the reflexive transitive closure of `from`, `func`.

The `put_field[F,V]` predicate is a ternary predicate that is satisfied if the method contains

Table 2.1: Native unary predicates of scratches

| Predicate | Meaning |
|---|---|
| `parameter` | assigned from a parameter of the method |
| `constant` | is a constant |
| `null` | is the null constant |
| `this` | assigned from parameter 0 of an instance method |
| `local_var` | assigned into an LVA entry |
| `returned` | used as the return value of the method |
| `athrow` | thrown by the code |
| `caught` | obtained by catching an exception |
| `compared` | compared in the code |

a `putfield` bytecode instruction such that the receiver reference is the scratch #, the assigned field is `F` and the value assigned to the field is the scratch `V`. Similarly, `get_field[F,V]` holds if the subject is receiver in a `getfield` instruction which assigns the value of the field `F` into the scratch `V`.

Using these predicates we can now define predicates that capture the notion of setter and getter methods:

```
setter F := instance method (_) {
     this put_field[F,V], V parameter;
   };
getter F := instance method () {
     this get_field[F,V] V returned;
   };
```

`setter` looks for a single parameter instance method, performing an assignment to a field whereby (*i*) the receiver object reference is equal to **`this`** and (*ii*) the value assigned to the field is equal to the method's parameter. `getter` looks for a zero-parameter instance method, that reads the value of a field whereby (*i*) the receiver object reference is equal to **`this`** and (*ii*) the field's value is returned from the method.

Method invocations can be examined via `receives M` which selects into `M` all methods that were invoked with subject scratch playing the role of the receive object. `get_method M` selects into `M` all methods, invoked by the containing method, whose return value was assigned to the subject scratch. `put_method M` selects into `M` all methods, invoked by the containing method, where the subject scratch was passed as a parameter.

To better understand these predicates, let us first define the following sample input:

```
public class SomeClass {
   public String g() {
     return "ab";
   }

   public String f(String s) {
     String a = g();
     return s.substring(s.indexOf(a));
   }
}
```

We will now evaluate each of the predicates in Figure 2.2 with `SomeClass` as input.

Evaluating `p1` (Figure 2.2(a)) on `SomeClass` obtains all methods invoked (by methods of the class) on some object, namely: `g()`, `substring()` and `indexOf()`. Evaluating `p2` (Fig-

**Figure 2.2** Usage of standard predicates `receives`, `from`, `put_method`, `get_method`.

```
p1 M := class {
     method { receives M; };
   };
```

(a) A method of the inspected class invokes `M`.

```
p2 M := class {
     method { put_method M int; };
   };
```

(b) A method of the inspected class invokes `M` passing a parameter of type `int`.

```
p3 M := class {
     method { get_method M String; };
   };
```

(c) A method of the inspected class invokes `M` getting a result of type `String`.

```
p4[M,N] := class {
     method { get_method M, S from* #, S put_method N; };
   };
```

(d) A method of the inspected class invokes `M` and `N`, passing the return value of `M` as a parameter to `N`.

ure 2.2(b)) on `SomeClass` obtains all invoked methods to which an int parameter was passed: `substring()`.

Predicate `p3` (Figure 2.2(c)) is similar to `p2` except that it conditions the *return value* of `M` to be of type `String`. Two method calls in `SomeClass` satisfy this requirement: `g()` and `substring()`.

Finally, `p4` (Figure 2.2(d)) captures two invoked methods `M` and `N` such that the result of the former is passed as parameter to the latter. When evaluated on `SomeClass` it returns—in `M`, `N`—two pairs of methods: `g()`, `indexOf()` and `indexOf()`, `substring()`.

### 2.2.3 Pedestrian Code Predicates

Scratches allow fine-grained inspection of the computation carried out by a method. *Pedestrian Code Predicates* stand at a higher level of abstraction thereby trading precision for ease of use. Specifically, these predicates examine the method as a whole without distinguishing between individual expressions as scratches do. They are typically implemented on top of scratch-level predicates.

The following JTL definition uses the pedestrian code predicate `reads`:

```
unread_private_field F := class {
     private instance field is F;
     no [method|constructor] reads F;
   };
```

This definition looks for a private field which is never read by the methods nor the constructors

of the class[3]. `reads` is a library predicate simply implemented in term of scratches:

```
reads F := {
     get_field[F,_];
   };
```

This predicate matches either methods or constructors that contain a `getfield` bytecode instruction involving the field `F`. We can now extend this simple version such that it will also match a class if any of its method (constructors) contains such an instruction.

```
reads F := class offers M, M reads F | {
     get_field[F,_];
   };
```

This latter definition disjuncts the former with `class offers M, M reads F`. When evaluating with a class subject, JTL will first find all members of the subject via the expression `class offers M` and then select only those that satisfy the term `M reads F`. This term will trigger a (recursive) evaluation of `reads`, this time with a **`MEMBER`** subject. In this (second) evaluation, only the right hand side of the disjunction will be evaluated, as the left hand side will be short circuited to false thanks to the `class` term. This implementation is the one provide by the JTL standard library.

Figure 2.3 presents the definitions of several useful pedestrian code predicates.

**Figure 2.3** Library predicates for pedestrian code queries

```
reads F := class offers M, M reads F | {
     get_field [F,_]
   };
writes F := class offers M, M writes F | {
     put_field [F,_]
   };
calls_instance M := class offers M', M' calls_instance M | {
      receives M;
   };
calls_static M; -- Native predicate: subject, either a class or a method,
                -- calls static method M
accesses F := read F | write F;
calls M := calls_instance M | calls_static M;
uses M := accesses M | calls M;
```

Examining the figure we note that with the exception of `calls_static` all predicates therein are standard, compound, definitions. `calls_static` is unique since calls to static method may have no scratches associated with them, if the method is **`void`** and takes no parameters. Thus, a dedicated native predicate is needed to detect these. Instance methods, on the other hand, are always associated with at least one scratch (due to their receiver reference) that allows their detection by scratch-level expressions.

`writes`, `calls_instance` are implemented in a similar manner to `reads` except that they detect field assignment, instance method call (respectively). `accesses`, `calls` are simple disjunctions of `reads`, `writes` and `calls_instance`, `calls_static` (respectively). Finally, `uses` is the disjunction of `accesses`, `calls`.

[3]Note that satisfiability of `unread_private_field` is not sufficient to determine that a field is redundant, due to the existence of inner classes.

## 2.3 Semantics

Conceptually, a JTL predicate $p$ of arity $n$ (a subject and $n-1$ explicit parameter) defines an $n$-ary relation $\mathcal{R}_p$ whose value are drawn from $\mathcal{D}$: the infinite set of all JAVA elements (classes, methods, fields, scratches, etc.) in the universe. When $p$ is evaluated, the result will be a sub-relation of $\mathcal{R}_p$, restricted by the given input values.

This relational view of JTL lends itself well to logic programming, and, in particular, to the DATALOG language.

### 2.3.1 Reduction to Datalog

The following shows the translation scheme from JTL to DATALOG. This serves two purposes. First, it allows us to precisely define JTL's semantics by means of reduction. Second, it prescribes an implementation strategy for JTL. Indeed, in our implementation of JTL a parser translates the JTL source code into an equivalent DATALOG program which is then executed by the runtime system.

The basic steps in the JTL to DATALOG translation are fairly simple

- Every JTL predicate becomes a DATALOG predicate.
- The JTL (implicit) subject parameter becomes a DATALOG (explicit) parameter. All other parameters are translated as-is.
- A JTL term becomes a DATALOG term.
- JTL's conjunction operator (blank) is translated into DATALOG's comma operator.
- JTL negation operator is translated as-is.

These steps are illustrated in Figure 2.4.

**Figure 2.4** JTL-to-DATALOG translation of a predicate definition, the subject variable, conjuncted terms, and a negated term.

```
p1 := abstract !public extends X X abstract
```

(a) JTL

```
p1(This) :- abstract(This), !public(This), extends(This,X),
  abstract(X).
```

(b) DATALOG

DATALOG has no dedicated disjunction operator. Instead, disjunction is expressed by several rules of the same name, and arity. Thus, translation of disjunctive expressions requires the introduction of a new rule for each branch of the expression (see Figure 2.5).

**Figure 2.5** JTL-to-DATALOG translation of the disjunction operator.

```
p2 := public [interface | class];
```

(a) JTL

```
p2(This) :- public(This), aux1(This).
aux1(This) :- interface(This).
aux1(This) :- class(This).
```

(b) DATALOG

The signature of the auxiliary rules introduced due to disjunction must contain all parameters/variables that are accessed in either branch. This may lead to rules that declare parameters which are not used inside the body, which is considered by DATALOG to be an "unbounded parameter" error. We overcome this limitation by the use of `always(X)`: a native DATALOG predicate (an EDB), which holds for every possible `X` value (see Figure 2.6).

**Figure 2.6** JTL-to-DATALOG translation of disjunction where each branch uses a different set of variables. The use of `always` ensure that the auxiliary DATALOG rule will access all of its parameters.

```
p3 T := public extends T [T abstract | interface];
```

(a) JTL

```
p3(This) :- public(This), extends(This,T), aux2(This,T).
aux2(This,T) :- interface(This), always(T).
aux2(This,T) :- abstract(T), always(This).
```

(b) DATALOG

The translation of the existential set query relies on the natural semantics of DATALOG, where every predicate invocation induces an implicit existential quantifier (see Figure 2.7).

**Figure 2.7** JTL-to-DATALOG translation of the existential quantifier.

```
p4 := class { abstract; };
```

(a) JTL

```
p4(This) :- class(This), items(This,This_1), abstract(This_1).
```

(b) DATALOG

`items` is a library predicate that serves as the default generator, i.e., resolves to `declares` if the subject is a type and to `scratches` if the subject is a method. It is declared as follows:

```
items X := class declares X | method scratches X;
```

If the JTL predicate specifies a non-default generator expression (e.g., `extends:` that term will be used instead of `items` in the DATALOG translation.

Universal quantification is expressed via double negation of the existential quantifier, as shown below in Figure 2.8.

**Figure 2.8** JTL-to-DATALOG translation of the universal quantifier.

```
p5 := class { all public; };
```

(a) JTL

```
p5(This) :- class(This), !aux3(_,This).
aux3(This_1,This) :- items(This,This_1), !aux4(This_1).
aux4(This) :- public(This);
```

(b) DATALOG

In the DATALOG translation, `aux4` captures the subset expression `public`. It is used, with negation, from `aux3` which calculates the set of elements (of the generated set) that are *not* `public`. Finally, `aux3` itself is used, with negation, from `p5`. Thus, `p5` holds if there is no element that is not `public`, which is equivalent to "all elements are public".

`aux3` also illustrates how the rebinding of the subject is realized (recall that the subject within a quantification scope is different than the one outside the scope). The second parameter passed to the generator expression `items(This,This_1)` is passed as the *first* parameter to `aux4`, thus making it the subject for the subset expression.

The set condition **no** is merely the negation of the existential operator. Given that DATALOG allows negation only on terms, we had to introduce an auxiliary predicate in order to make the existential expression negatable (see Figure 2.9).

**Figure 2.9** JTL-to-DATALOG translation of set condition **no**.

```
p6 := class { no abstract; };
```

(a) JTL

```
p6(This) :- class(This), !aux5(This).
aux5(This) :- items(This, This_1), abstract(This_1).
```

(b) DATALOG

Examining the figure we see that `aux5` is merely an existential expression sporting the default generator `items` and `abstract` being the subset expression. `p6` negates `aux5`, thereby getting a "no" semantics.

The translation scheme of set condition **one** is based on the following tautology: "exists only one" is identical to "exists X and no other one exists that is different than X" (see Figure 2.10).

**Figure 2.10** JTL-to-DATALOG translation of set condition **one**.

```
p7 := public class { one abstract; };
```

(a) JTL

```
p7(This) :- public(This), class(This), aux6(This).
aux6(This) :- items(This,This_1), aux7(This_1), !aux8(This,This_1).
aux7(This) :- abstract(This).
aux8(This,This_1) :- items(This,Temp), aux7(Temp),
  is_not(This_1,Temp).
```

(b) DATALOG

`Temp` is a uniquely named synthetic variable introduced by the translator. It plays the role of the "X" variable mentioned above. `is_not` is the library predicate realizing the inequality relation. It is simply defined, in JTL syntax, as `is_not X := !is X;`.

Finally, let us consider the **`implies`** set condition. Unlike the other operators presented here, **`implies`** is a binary set condition which requires two subset expression (see Figure 2.11).

**Figure 2.11** JTL-to-DATALOG translation of set condition **`implies`**.

```
p8 := class {
    field implies private;
  };
```

(a) JTL

```
p8(This) :- class(This), !aux9(This).
aux9(This) :- items(This,This_1), aux10(This_1).
aux10(This) :- aux11(This), !aux12(This).
aux11(This) :- field(This).
aux12(This) :- private(This).
```

(b) DATALOG

The resulting DATALOG code includes two predicates, namely: `aux11` and `aux12`, that represent the subset expressions `field` and `private`.

`aux9` calculates the whole quantification expression, by calculating the generator expression and then performing conjunction with `aux10` which realizes the implication condition.

The other set conditions supported by JTL (such as **`disjoint`**, **`partition`**, etc.) are translated in a similar manner.

### 2.3.2 Computability

Even though termination is always guaranteed (on a finite database) as long as negation is stratified, it is a basic property of First-order predicate logic that other questions are undecidable. For example, it follows from Gödel's incompleteness theorem that it is impossible *in general* to determine e.g., if two queries are equivalent, a query is always empty, the results of one query is contained in another, etc. These limitations are not a major hurdle for most JTL applications.

Moreover, there are textbook results [37] stating that such questions are decidable, with concrete algorithms, if the use of quantifiers is restricted, as could be done for certain applications.

Our JTL implementation sports a top-down evaluation strategy which is optimized for *close* queries: Queries where the amount of information needed to be "seen" during evaluation is determined by the definition of the predicate and the given input, *and not* by the size of the full database. Such queries have two compelling properties.

First, in practice, the size of the input is significantly smaller than the size of the database. Thus, evaluation of such queries is faster (compared to "open" queries) simply because the amount of information needed for computing the output is smaller.

Second, evaluation of close queries produces results which are stable in the sense that they will not be affected by non-destructive changes to the database, such as: the addition of new .jar files to it. In other words, if a result was obtained with one version of the database, it will not change if more information is added to the database.

**Definition 2** *A predicate $p$ is* close with respect to a subset of its parameters $\{X_1, X_2, \ldots X_n\}$*, if its evaluation does not require information beyond that found in these class files:*

- *class files where the literals mentioned in $p$ are declared.*
- *class files where the actual values of the parameters $X_1, X_2, \ldots X_n$ are declared.*
- *class files which the former class files depend upon (transitively).*

**Definition 3** *A predicate $p$ is* open with respect to a subset of its parameters $\{X_1, X_2, \ldots X_n\}$*, if $p$ is not close with respect to $\{X_1, X_2, \ldots X_n\}$.*

In order to make this definition concrete, let us examine the predicate in Figure 2.12.

**Figure 2.12** The predicate `method_throw` is close with respect to either # or `M` but open with respect to `E`

```
methods_throw[M,E] := {
     method is M throws E;
   };
```

Predicate `methods_throw` that is defined in the figure is close with respect to the subject since `M`, and `E` can be computed from the subject by following the information found at the class file where the subject class is declared.

`methods_throw` is also close with respect to `M`: the subject can be computed from `M` since every method "knows" its declaring class; `E` can be computed from `M` since every method "knows" its thrown exceptions.

On the other hand, `method_throw` is not close with respect to `E`: an exception class has no knowledge on the methods from which it may be thrown (`M`), nor on the classes that declare these methods (#). Thus, we say that *`method_throw` is open with respect to `E`*.

Consider now an evaluation of `method_throw` where `E` is known, but #, `M` are not known, as in: `X.method_throw[Y,/java.io.IOException]`. The result of this evaluation requires finding all methods (in a given database) that throw an `IOException` and all classes declaring those methods. If class `com.sun.security.auth.PolicyFile` is contained in the database then the result will include this class as `X` and its method `getInputStream` as `Y`. Otherwise, the result will not include this pair. This illustrates the instability of open queries.

Close queries are stable. If the subject is `java.util.Date` then the result will span these two pairs of `M`, `E`, respectively: `readObject`, `IOException`; and `writeObject`,

`IOException`. Even if additional classes are added to the database the result of this close query will not change.

Note that close queries are not immune to *destructive* changes, such as: (*i*) Changes that compromise the internal integrity of the database. For instance, if class `IOException` is removed from the database but not class `Date` then the query will not be computable at all and evaluation will terminate with an error. (*ii*) changes to the definition of the involved classes. If the definition `Date` class is changed such that it no longer defines the method `readObject` then the result of the query will inevitably change.

In summary, the evaluation of open queries is time consuming. Worse, the output of these queries is non-deterministic, in the sense that it depends on the extent of the software repository available to the processor. As it turns out, JTL queries tend to be close.

The JTL processor includes a predicate analyzer, developed by Cohen, Gil, and Zarivach [52] which determines if a given query is open or close. The JTL system uses this algorithm for three purposes: First, it alerts the user whenever it tries to execute an open query. Second, it uses the algorithm for determining an evaluation order that is guaranteed to terminate.

Third, it can report the computability of a predicate, that is: which parameters need to be specified as inputs in order to ensure that other (output) parameters are computable. Computability information is expressed as a set of *calling patterns*. Each calling pattern is a pair of two sets of parameters denoted as: $I_1, I_2, \ldots I_n \rightarrow O_1, O_2, \ldots O_m$, where $\{I_1, I_2, \ldots I_n\}$ are the input parameters and $\{O_1, O_2, \ldots O_m\}$ are the output parameters.

The computability of `method_throw` is therefore denoted as follows:

$$\# \rightarrow M, E \text{ [or] } M \rightarrow \#, E$$

This indicates that if either the subject or `M` is specified then the remaining two parameters can be computed.

### 2.3.3 Kind System

JTL runtime values are categorized into several kinds.[4] These kinds are not disjoint, as shown in Figure 2.13. The figure shows the JTL kinds that are visible to the user. Implementation-level kinds, i.e., kinds that are internally used by native predicates, were omitted from the figure.

Examining the figure we see that kind **`CODE`** is either a **`SCRATCH`** or a **`UNIT`**. Under **`UNIT`** we have **`PACKAGE`** and **`ELEMENT`** which is further divided into **`MEMBER`** and **`TYPE`**. **`CODE`**'s superkind, **`ANY`**, was introduced to accommodate implementation-level kinds, and to allow future versions of JTL to support additional kinds without breaking the existing structure.

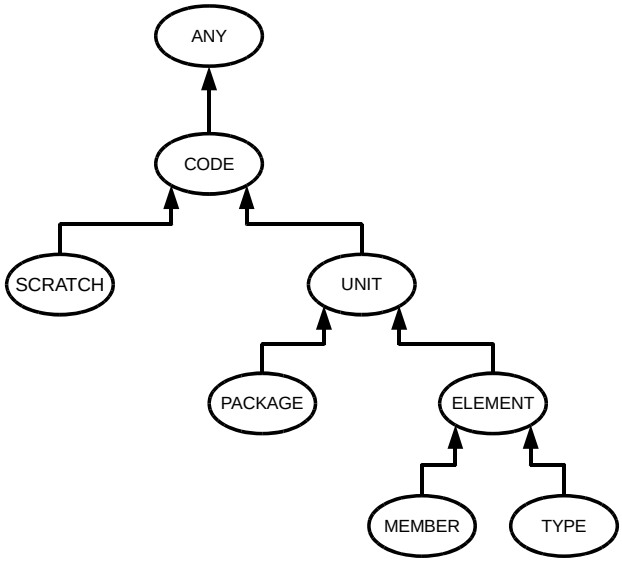
Non-native predicates in DATALOG (hence: JTL) simply apply union and/or join operations on relations returned by other predicates, or by themselves in case of recursion. Other than arity mismatch errors which are trivially detected statically, these operations are agnostic of the kinds of the values in those relations, and thus, cannot fail on grounds of kind errors.

A runtime kind error may therefore occur only if a native predicate rejects one of its inputs. Here's a typical example: `extends X, X returned`. The first term, `extends X`, will bind `X` to a **`TYPE`** value, where the second term, `X returned`, is defined only if `X` is a scratch. If evaluation of that expression were allowed, a runtime kind error should occur.

To detect these statically, JTL employs a kind checking and inferencing mechanism that follows the same ideas as those recently described by Schäfer and de Moor [163]. Every native predicate specifies its signature: a set of tuples of kinds, where each tuple is of the same arity as the predicate itself. Each tuple position denotes the kind of the corresponding parameter of the predicate.

[4] Given that JTL's domain includes JAVA types, we shall use the term "kind" to denote "a JTL type".

**Figure 2.13** JTL's hierarchy of kinds, superkinds depicted above subkinds.




A signature may contain more than one tuple because some predicates have more than one legal combination of kinds. For instance, `declared_by` can be evaluated with either ⟨**MEMBER**,**TYPE**⟩ or with ⟨**SCRATCH**,**MEMBER**⟩.

Treating a signature as a relation, one can compute the signature of a DATALOG definition as follows: The signature of a DATALOG rule is the join of the signatures of the enclosed terms. The signature of a DATALOG predicate is the union of the signatures of the rules making up the predicate. The process repeats itself until a fixed-point is reached.

We note that this algorithm is essentially identical to the evaluation of a DATALOG program where the native predicates return a relation of kinds (reflecting the signature) rather than a relation of values. Termination is guaranteed since the size of the domain, i.e., the number of possible kinds, is finite.

The result of the algorithm is a signature (relation) for each predicate. If such a signature is empty, then there is no legal combination of parameters which satisfy the requirements of the native predicates that are used, directly or indirectly, by the compound predicate. Thus, an empty signature represents an ill-defined predicate, which can never be evaluated.

When the signature is not empty it is used for checking the validity of the external input passed to the predicate when it is used as a goal predicate. Evaluation will start only if the given input matches the signature.

## 2.4 Application

Having presented the JTL syntax, the language's capabilities and its underlying semantics, we are in a good position to describe some of the applications.

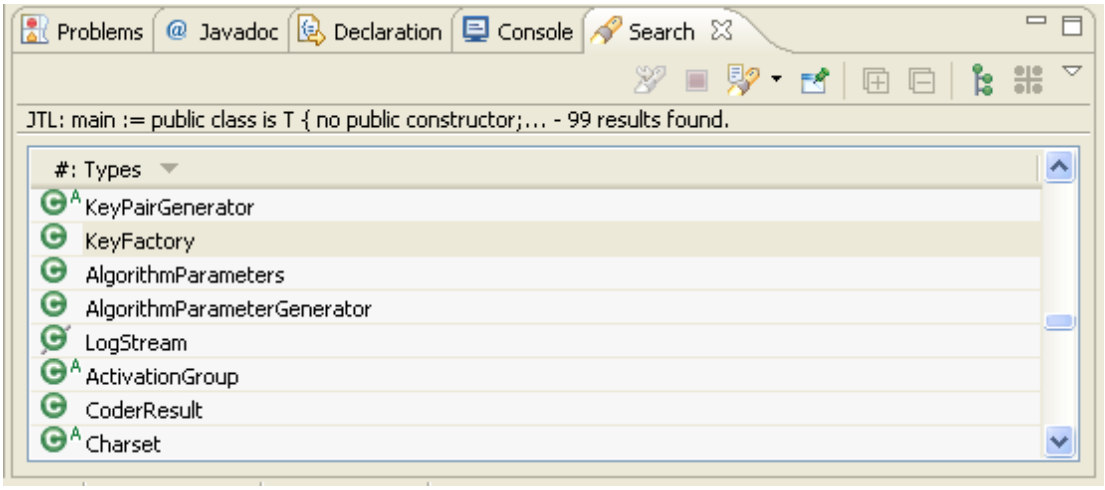
### 2.4.1 IDE Integration

In their work on JQuery, Janzen and De Volder [111] make a strong case, including empirical evidence, for the need of a good software query tool as part of the development environment.

We have developed an Eclipse plug-in that runs JTL queries and presents the result in a dedicated view. Figure 2.14 shows an example: the query (which appears, partially, above the results)

found classes from JAVA's standard library for which instances are obtained using a **`static`** method rather than a constructor. Using JTL, many searches can be described intuitively. The

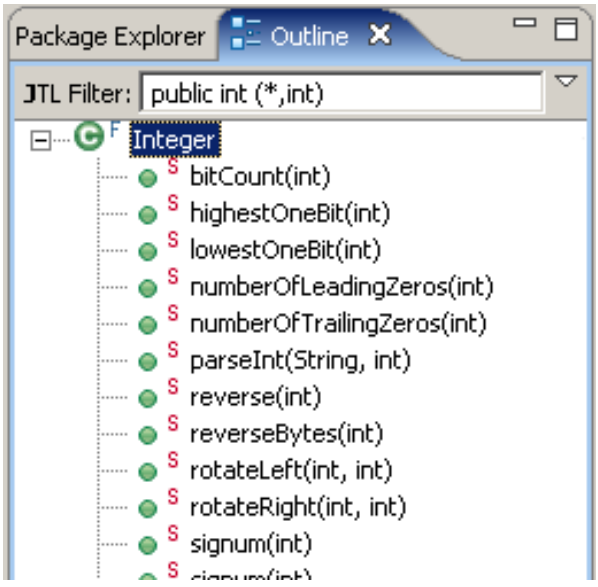
**Figure 2.14** Screenshot of the result view of JTL's Eclipse plugin




similarity between JTL syntax and JAVA declarations allows even developers who are new to JTL to easily and effectively sift through the overwhelming number of classes and class members in the various JAVA libraries.

JTL can also be used to replace the hard-coded filtering mechanism found in many IDEs (e.g., a button for showing only **`public`** members of a class) with a free-form filter. Figure 2.15 is a mock screenshot that shows how JTL can be used for filtering in Eclipse.

**Figure 2.15** Using JTL for filtering class members (mock)



### 2.4.2 Specifying Pointcuts in AOP

The limited expressive power of the pointcut specification language of ASPECTJ (and other related AOP languages, e.g., CAESAR [137] and ASPECTJ2EE [54]), has been noted several times in the literature [98, 155].

We propose that JTL is integrated into AOP processors, taking charge of pointcut specification. To see the benefits of using a JTL component for this purpose, consider the following ASPECTJ pointcut specification:

```
call(public void *.set*(*));
```

JTL's full regular expressions syntax can be used instead, by first defining `setter := public void 'set[A-Z]?*'(_);` and then writing `call(setter)`. Unlike the ASPECTJ version, The JTL version uses a proper regular expression, and therefore does not erroneously match a method whose name is, e.g., `settle()`. Even more importantly, JTL's scratch-related predicates can be used to make the pointcut examine the exact behavior of the method rather than relying on naming convention which are inherently imprecise. In particular, there is no need to define the `setter` predicate. As explained in Section 2.2.2, the predicate `setter` from JTL's standard library detects all methods where a parameter is assigned to a field.

Figure 2.16 presents an array of ASPECTJ pointcuts trapping read and write operations of primitive public fields. Not only tedious, it is also error prone, since a major part of the code is replicated across all definitions.

**Figure 2.16** An ASPECTJ pointcut definition for all read- and write-access operations of primitive public fields.

```
get(public boolean *) || set(public boolean *) ||
get(public byte *)    || set(public byte *)    ||
get(public char *)    || set(public char *)    ||
get(public double *)  || set(public double *)  ||
get(public float *)   || set(public float *)   ||
get(public int *)     || set(public int *)     ||
get(public long *)    || set(public long *)    ||
get(public short *)   || set(public short *);
```

By using disjunction in JTL expressions, the ASPECTJ code from Figure 2.16 can be greatly simplified if we allow pointcuts to include JTL expressions:

```
ppf := public primitive field;

get(ppf) || set(ppf); // JTL-based AspectJ pointcut
```

The ability to name predicates, specifically `ppf` in the example, makes it possible to turn the actual pointcut definition into a concise, readable statement.

Figure 2.17 provides an example for a condition that is impossible to specify in ASPECTJ.

Condition `field_in_plain_class`, defined in Figure 2.17, holds for **`public`** fields in a class which has no getters or setters. The above could have been implemented in other extensions of the ASPECTJ pointcut specification language, but not without a loop or a recursive call.

Our contribution puts the expressive power of JTL at the disposal of ASPECTJ and other aspect languages, replacing the sometimes ad-hoc pointcut definition language with JTL's systematic approach. There is one limitation in doing that: JTL can only be used to make queries on the program static structure, and not on the *dynamic* control flow.

**Figure 2.17** A JTL pointcut matching fields whose declaring class has neither setters nor getters.

```
plain_class := {
     no getter;
     no setter;
  };

field_in_plain_class := public field declared_by C,
  C plain_class;
```

### 2.4.3 Concepts for Generic Programming

In the context of generic programming, a *concept* is a set of constraints which a given set of types must fulfill in order to be used by a generic module. As a simple example, consider the following C++ [172] template from Figure 2.18.

Class `ElementPrinter` from the figure assumes that the provided type parameter `T` has a method called `print` which accepts no parameters. Viewing `T` as a single-type *concept* [82,173], we say that the template presents an implicit assumption regarding the concept it accepts as a parameter. Implicit concepts, however, present many problems, including hurdles for separate compilation, error messages that Stroustrup et al. term "of spectacular length and obscurity" [173], and more.

With Java generics, one would have to define a new interface

```
interface Printable { void print(); };
```

and use it to confine the type parameter. While the concept is now explicit, this approach suffers from two limitations: first, due to the nominal subtyping of JAVA, generic parameters must explicitly implement interface `Printable`; and second, the interface places a "baggage" constraint on the return type of `print`, a constraint which is not required by the generic type.

Using JTL, we can express the concept explicitly and without needless complications, thus:

```
(class | interface) {
  'print ();
};
```

There are several advantages for doing that: First, the underlying syntax, semantics and evaluation engine are simple and need not be re-invented. Second, the JTL syntax makes it possible to make useful definitions currently not possible with JAVA standard generics and many of its extensions.

The problem of expressing concepts is more thorny when multiple types are involved. The work of Garcia et al [82] evaluated genericity support in 6 different programming languages (including JAVA, C# [104] and EIFFEL [109]) with respect to a large scale, industrial strength, generic graph algorithm library, reaching the conclusion that the lack of proper support for multi-type con-

**Figure 2.18** A C++ template that expects template parameter `T` to define a zero-parameter `print()` method.

```
template<typename T>
class ElementPrinter {
public:
  void print(T element) {
    element.print();
  }
}
```

cepts resulted in awkward designs, poor maintainability, and unnecessary run-time checks.

JTL predicates can be used to express multi-type concepts, and in particular each of the concepts that the authors identified in this graph library.

As an example, consider the `memory_pool` concept. A memory pool is used when a program needs to use several objects of a certain type, but it is required that the number of instantiated objects will be minimal. In a typical implementation, the memory pool object will maintain a cache of unused instances. When an object is requested from the pool, the pool will return a previously cached instance. Only if the cache is empty, a new object is created by issuing a create request on an appropriate factory object.

More formally, the memory pool concept presented in Figure 2.19 takes three parameters: `E` (the type of elements which comprise the pool), `F` (the factory type used for the creation of new elements), and # (the pool type).

**Figure 2.19** The `memory_pool` concept

```
name create, instance, acquire, release;

factory E := {
     public constructor ();
     public E create ();
   };

memory_pool[F,E] := is T {
     public static T instance ();
     public E acquire ();
     public release (E);
   }, F factory E;
```

The body of the concept requires that the subject will provide `acquire()` and `release()` methods for the allocation and deallocation (respectively) of `E` objects, and a static `instance()` method to allow client code to gain access to a shared instance of the pool. Finally, it requires (by invoking the `factory` predicate) that `F` provides a constructor with no arguments, and a `create()` method that returns objects of type `E`.

As shown by Garcia et al., the requirements presented in Figure 2.19 have no straightforward representation in JAVA, C# or EIFFEL. In particular, using an **`interface`** to express a concept presents extraneous limitations, such as imposing a return type on `release`, and it cannot express other requirements, such as the need for a zero-arguments constructor in a factory. Using an **`interface`** also limits the applicable types to those that `implement` it, whereas the concept itself places no such requirement.

In a language where JTL concept specifications are supported, a generic module parameterized by types `X`, `Y` and `Z` can declare, as part of its signature, that `X.memory_pool[Y,Z]` must hold. This will ensure, at compile-time, that `X` is a memory pool of `Z` elements, using a factory of type `Y`.[5]

Concepts are not limited to templates and generic types. Mixins, too, sometimes have to present requirements to their type parameter. The famous `Undo` mixin example [12] requires a class that defines two methods, `setText` and `getText`, but does not define an `undo` method. The last requirement is particularly important, since it is used to prevent *accidental overloading*. However, it cannot be expressed using JAVA interfaces. The following JTL predicate clearly

[5] Thus, concepts may be regarded as the generic-programming equivalence of the *Design by Contract* [136] philosophy

expresses the required concept:

```
undo_applicable := class {
      'setText (String);
      String 'getText ();
      no 'undo ();
   };
```

In summary, we propose that in introducing advanced support of genericity and concepts to JAVA, one shall use the JTL syntax as the underlying language for defining concepts.

### 2.4.4 LINT-like tests

LINT-like tools often allow their users to enrich their set of built-in rules, with custom-made rules tailored for the specific needs of the user. JTL's expressiveness makes it an ideal device for customization of such tools.

To test this prospect, we developed a collection of JTL patterns that implement the entire set of warnings issued by Eclipse and PMD (a popular open source LINT tool for JAVA). The only exceptions were those warnings that directly rely on the program source code (e.g., unused **`import`** statements), as these violations are not represented in the binary class file, that JTL uses.

For example, consider the PMD rule *Loose Coupling*. It detects cases where the concrete collection types (e.g., `ArrayList` or `LinkedList`) are used instead of the abstract interfaces (such as `List`) for declaring fields, method parameters, or method return values—in violation of the library designers' recommendations. This rule is expressed as a JAVA class, and includes a hard-coded (yet partial) list of the implementation classes (see Figure 2.20). PMD does make a heroic effort, but it will mistakenly report (e.g.) fields of type `LinkedList` for some alien class `LinkedList` which is not a collection, and was declared outside of the `java.util` package. The JTL equivalent is:

```
loose_coupling := (class|interface) {
  [method | field] typed T | method(*, T, *);
}, T class subtypes /java.util.Collection;
```

It is shorter, more precise, and will detect improper uses of any class that implements any standard collection interface, without providing an explicit list.

## 2.5 Implementation

The main challenge in implementing JTL was in providing a robust and efficient execution environment that can be easily integrated into JAVA tools.

The JTL implementation uses the *Bytecode Engineering Library*[6], BCEL, for implementing the native predicates which extract the core facts from the input program. The compilation process starts by parsing the JTL code and translating it into DATALOG in the manner described in Section 2.3.1. We then build an in-memory representation of this DATALOG program. The kind inference algorithm then examines this representation and prevents its execution if typing errors are detected.

The runtime system takes four inputs:

- A Datalog program (in its in-memory representation)
- The name of a goal predicate

[6] http://jakarta.apache.org/bcel

**Figure 2.20** The implementation of PMD's *Loose Coupling* rule.

```
public class LooseCouplingRule extends AbstractRule {
  private Set implClassNames = new HashSet();
  public LooseCouplingRule() {
    implClassNames.add("HashSet");
    implClassNames.add("HashMap");
    implClassNames.add("ArrayList");
    implClassNames.add("LinkedList");
    implClassNames.add("LinkedHashMap");
    implClassNames.add("LinkedHashSet");
    implClassNames.add("TreeSet");
    implClassNames.add("TreeMap");
    // ...
  }

  public Object visit(ASTResultType node, Object data) {
    checkType(node, data);
    return data;
  }

  public Object visit(ASTFieldDeclaration node, Object data) {
    checkType(node, data);
    return data;
  }

  public Object visit(ASTFormalParameter node, Object data) {
    checkType(node, data);
    return data;
  }

  private void checkType(SimpleNode node, Object data) {
    if (node.jjtGetNumChildren() !=0)
      return;

    SimpleNode returnTypeNameNode = (SimpleNode)node
      .jjtGetChild(0).jjtGetChild(0);
    if (implClassNames.contains(returnTypeNameNode.getImage())) {
      RuleContext ctx = (RuleContext)data;
      ctx.getReport().addRuleViolation(createRuleViolation(ctx,
        returnTypeNameNode.getBeginLine(), MessageFormat.format(
          getMessage(),
          new Object[] {returnTypeNameNode.getImage()})));
    }
  }
}
```

- A classpath string
- A tuple of JTL values.

It evaluates the goal predicate by binding the values of the tuple to the goal's formal parameters. While the tuples need to have the same arity as the goal, they may specify `null` for some of the parameters. These parameters are considered to be unknown which effectively makes them output parameters.

If the input tuple does not provide enough input parameters the runtime system reject it on the ground of being insufficient for computing the goal (see Section 2.3.2). If the tuple if legal, the runtime system will evaluate the goal returning a relation of all output parameters. The JTL API allows client code to obtain computability constraints of a predicate, thus allowing it to determine whether a given tuple qualifies as sufficient input without actually running the query.

The remainder of this section overviews the API that allows JAVA program to evaluate JTL query, compares the performance of JTL with a similar query language, and examines the challenges in porting JTL to other languages.

### 2.5.1 API

The main gate into the JTL world is class `jtl.system.api.JTL`. Although custom instances of this class can be created, the pre-made default object, `JTL.INSTANCE` is often adequate. The simplest way to evaluate a query is to invoke one of the overloaded versions of `run()`. In these versions, the first parameter is always a string specifying a JTL program, while additional parameters are input values, in various formats (fully-qualified names, `java.lang.Class` objects, etc.).

Figure 2.21 shows a small JAVA program that evaluates a JTL query by issuing a single call to `JTL.run()`.

**Figure 2.21** Running a JTL query that finds all int-taking methods of class `JFrame`.

```
import javax.swing.JFrame;
import jtl.system.api.JTL;

public class Example {
   public static void main(String[] args) throws Exception {
      System.out.println(JTL.INSTANCE.run(
         "takes_int_method := public method (*,int,*);" +
         "main M := class { takes_int_method is M; };",
         JFrame.class));
   }
}
```

The `run()` method begins by compiling, and kind-checking, the JTL program specified in its parameters. If no errors were detect, the running phase begins: `run()` evaluates the goal predicate (defaults to `main`) using the given JAVA element (here, class `JFrame`) as the input that the goal predicate inspects (the subject). Note that `main` can compute its `M` parameter from its subject so the only input we need is the class to test.

The result is returned as an instance of class `Relation` which is a collection of `Tuple` objects of fixed arity. Here, we simply print it to standard output.

The API also support ahead-of-time compilation of JTL queries. This is illustrated in Figure 2.22.

**Figure 2.22** A JAVA program that evaluates a command-line-specified JTL query.

```
1 public class JTLEval {
2
3   public static void main(String[] args) throws Exception {
4
5     ClassRepository rep = new ClassRepository(
6       args[0].split(File.pathSeparator));
7
8     String query = "";
9     for(int i = 1; i < args.length; ++i)
10      query += args[i] + " ";
11
12    JTL jtl = new JTL(new Domain(rep));
13    Executable exec = jtl.compile(query);
14
15    for(String className : rep)
16      System.out.println(exec.run(className));
17  }
18 }
```

The program in the figure expects at least two command line parameters. The first is a class-path string specifying the class files which will be accessible to the query. The query itself is given by the space-delimited concatenation of the remaining parameters. Thus, the following command line tells the JAVA program from Figure 2.22 to run a query that finds all mutable public instance fields in the .jar file of the Mockito[7] library:

```
JTLEval /libs/mockito.jar main F := { public instance !final field is F; };
```

The program from Figure 2.22 starts by creating a class repository object corresponding to the class-path string specified by the first command line parameter (lines (5)-(6)). It then composes the query string from the remaining parameters (8)-(10).

The program continues by creating a `JTL` object, configured with that class-path (12), and by compiling the query string into an `Executable` object (13). The second loop simply iterates over the classes from the repository and evaluates the query (via `Executable.run()` on each of them (15)-(16).

### 2.5.2 Performance

We will now turn to the evaluation of the performance of this implementation. Our test machine had a single 3GHz Pentium 4 processor with 3GB of RAM, running Windows XP. All JAVA programs were compiled and run by Sun's compiler and JVM, version 1.5.0_06.

In the first set of measurements, we compared the time needed for completing the evaluation of two distinct JTL predicates. These predicates are presented at Figure 2.23.

The first predicate in Figure 2.23, `q1`, matches classes with a static method returning the same type as the declaring class. The second predicate, `q2`, holds if these two requirements are satisfied: The subject class declares a `toString()` and an `equals(Object)` method; the subject's chain of super-classes contain at least one abstract class.

Each of these predicates were evaluated over six increasingly larger inputs, formed by selecting at random 1,000, 4,000, 6,000, 8,000, 10,000 and 12,000 classes from the JAVA standard library,

[7] http://mockito.org

**Figure 2.23** JTL queries `q1` and q2. `q1` holds if # declares a public static method whose return type is #; `q2` holds if one of the super-classes of # is abstract and, in addition, # declares a `toString()` method and an `equals()` method.

```
q1 := is T { public static typed T (*) ; };

q2 := extends+: { abstract ; }
   declares: {
      public String toString () ;
      public boolean equals (Object) ;
   };
```

version 1.5.0_06, by Sun Microsystems.

The running time of `q1` and `q2` on the various inputs are shown on Figure 2.24.

Examining the figure, we see that execution time, for the given programs, is largely linear in the size of the input. The figure may also suggest that runtime is linear in program size, but this conclusion cannot be true in general, since there are programs of constant size whose output is polynomially large in the input size.

The absolute times are also quite reasonable. For example, it took just about 10 seconds to complete the evaluation of program `q1` on an input of 12,000 classes. Overall, the average execution rate for program `q1` was 1,250 classes per second.

In the second set of measurement we compared JTL's Eclipse plugin with that of JQuery [111]. In a similar manner to JTL, JQuery also tries to harness the power of declarative, logical programming to the task of searching in programs, but (unlike JTL) JQuery expressions are written in a PROLOG-like notation.

Another difference between these two systems relates to the evaluation scheme: JQuery uses a

**Figure 2.24** Execution time of a JTL program vs. input size.

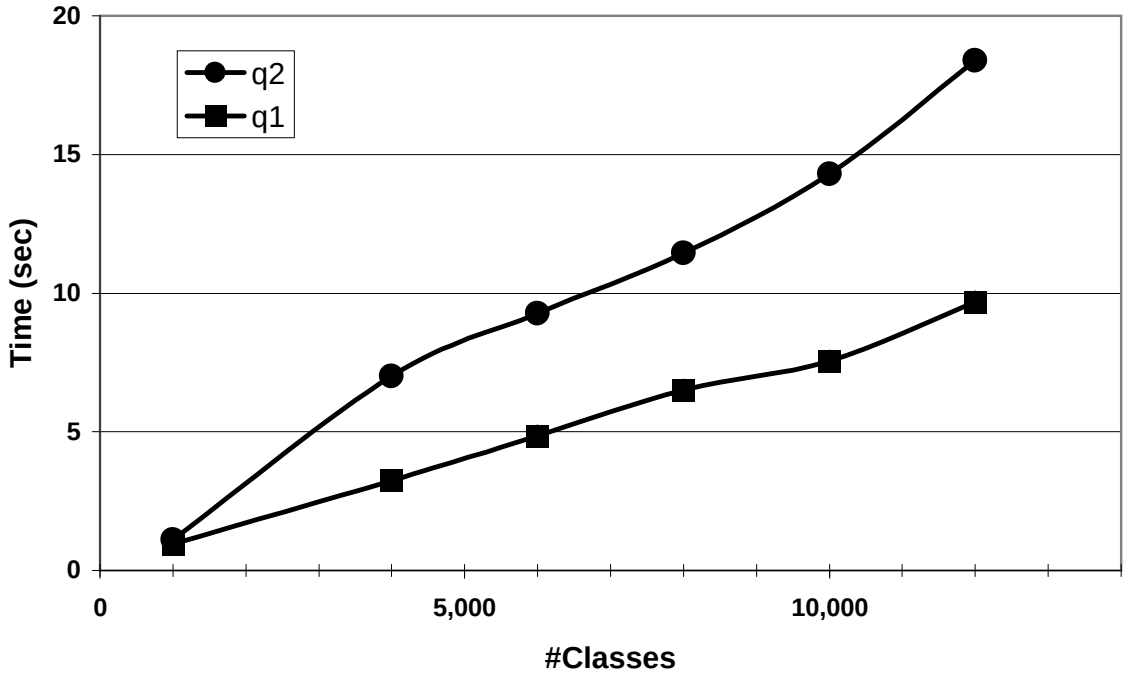


**Figure 2.25** The sequence of stages used for benchmarking.

- *Init.* One-time initialization
- *Run1.* First execution of the query
- *Run2.* Second execution of the query.
- *Update.* Updating of the internal data-structure following a slight modification of the source files.
- *Run3.* Third execution of the query.
- *Run4.* Fourth execution of the query.

**Figure 2.26** The JQuery equivalent of query `q1`. Holds for classes `C` that declare a public static method whose return type is `C`.

```
q1' ≜ method(?C,?M), returns(?M,?C),
      modifier(?M,static),
      modifier(?M,public)
```

bottom-up algorithm for the evaluation of predicates. As explained in Section 2.3.2, a bottom-up approach is far from being optimal since it needlessly computes tuples and relations even if they cannot be reached from the given input.

Specifically, JQuery initialization stage, where it extracts facts the from all classes of the program took more than four minutes on a moderate size project (775 classes), which is two orders of a magnitude slower than JTL's initialization phase. Also the first invocation of an individual JQuery query is roughly ten times slower than the corresponding time in JTL.

Therefore, in order to make the comparison fair to JQuery, we broke a user's interaction with the querying system into a sequence of six distinct stages (defined in Figure 2.25) and compared the performance of JQuery vs. JTL on a stage-by-stage basis.

When running the JTL sessions we used the query `q1` defined earlier. In the JQuery sessions we used query `q1'` (from Figure 2.26) which is the JQuery equivalent of `q1`.

We timed the JTL and the JQuery sessions on the Eclipse projects representing the source of two open-source programs: JFreeChart[8] (775 classes) and Piccolo[9] (504 classes).

The speedup ratio of JTL over JQuery is presented in Figure 2.27. The figure shows that JTL is faster in the *Init*, *Run1* and *Update* stages. JTL is about 100 times faster than JQuery at the *Init* stage, and about 25 times faster at the *Run1* stage. JTL was just slightly faster in *Run3*, while JQuery was slightly faster in the *Run2* and *Run4* stages.

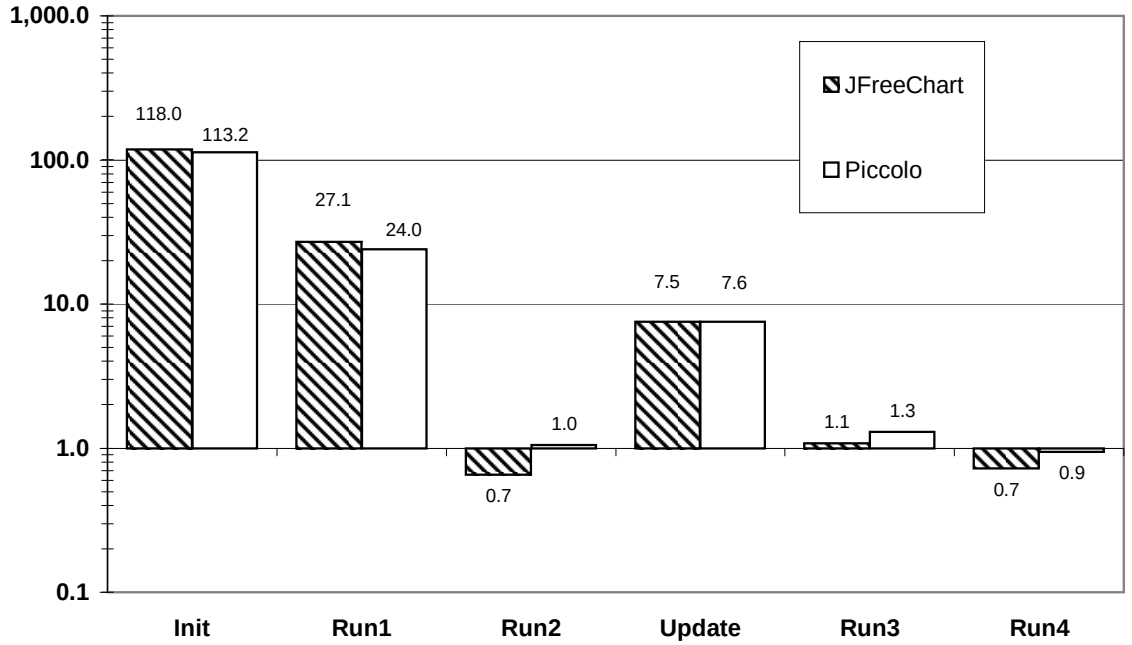
As for space efficiency, we predict that a bottom-up evaluator will be less efficient, compared to a top-down evaluator. In particular, we note that running JQuery searches on subject programs larger than 3,000 classes exhausted the memory of the benchmark machine. JTL, on the other hand, was able to process a 12,000-classes project.

### 2.5.3 Supporting Other Languages

As mentioned earlier, JTL was developed with the Java language in mind. Nonetheless, our theoretical observations, as well as many of the implementation-level concerns, are applicable to other

[8] http://www.jfree.org/jfreechart
[9] http://www.cs.umd.edu/hcil/jazz

**Figure 2.27** Speedup of JTL over JQuery, shown on a logarithmic scale. Each pair of columns represents one of the stages defined in Figure 2.25. Speedup was calculated by dividing the time needed for a stage in the JQuery session with the corresponding time measured from the JTL session.




object-oriented programming languages. This section discusses the hypothetical adaptation of JTL's specification and implementation to other programming languages by considering C# as a test case (We will henceforth refer to JTL over C# as #TL[10]).

Examining the similarities between JAVA and C#, we see that in both languages the primary programming construct is the class. A class can define methods and fields, extend another class and implement a number of interfaces. The two languages also agree on the class hierarchy being single rooted, the multiple inheritance in the interface hierarchy, and on the organization of classes in packages (namespaces in C# jargon). Many of the keywords of JAVA are also used, with the same semantics, in C#.

Based on these similarities, JTL can be ported to C# simply by replacing the native predicates in JTL's standard library. Some native predicates will have a new implementation (e.g., `class` and `abstract`), a few will be removed (e.g., `transient`), and a few others added (e.g., `const`).

Despite the fact that no other parts of the JTL system need to be changed, this simple port is surprisingly useful. For example, a query such as
`shared_state := class { no instance field };` will work on C# input if a C#-enabled version of the following native predicates is given: `class`, `members`, `static` and `field`. Obviously, most of the non-native predicates in the standard library will also work, with the same semantics, when used with the new native ones.

As in JTL, the implementation of #TL natives may take one of several input formats. In particular, a native predicate may obtain the data it needs from a source code input, or from a compiled binary input (in Microsoft's Intermediate Language, *MSIL*, format).

In order to gain further intuition regarding the porting process, let us consider two C# constructs which do not have a JAVA counterpart: **`struct`** and **`delegate`**. C#'s **`struct`**s are classes which carry value semantics. This means, for example, that instances of **`struct`**s types

[10] "Sharp-Tel"

are stored on the runtime stack, thus delivering some performance gain compared to instances of class types. The natural way to represent **`struct`** types in JTL would be to add a new native unary predicate, `struct`, that will match types iff they are declared as **`struct`**s. This new predicate is analogous to the `interface` predicate already present in JTL's standard library.

The case of delegates is slightly more complicated. A delegate is a type, parameterized by a method signature, whose values are methods of concrete runtime objects. It can be invoked in a manner similar to a function call, thereby serving as the object-oriented counterpart of C [116]'s function pointer.

It may seem that such an entity can only be represented in JTL programs by a new JTL kind, alongside the **`TYPE`** and **`MEMBER`** kinds. However, it turns out that JTL is flexible enough to cope with this new language construct by the following two changes, which are (again) limited to the standard library:

First, we will introduce a new native unary predicate, `delegate`, that will match a **`TYPE`** if it is a delegate. Second, we will change the implementation of the `call_matching` native predicate. This predicate is internally used by the JTL library for checking whether a parameter list matches a method. The new version of this predicate will extend it to match calls on delegates. These two changes allow #TL programs to acknowledge the similarity of delegates and methods.

Figure 2.28 presents a query that is written in #TL.

**Figure 2.28** A JTL query that matches C# **`struct`**s that define a compare method or a field named `compare` whose type is a two argument delegate.

```
has_compare := struct {
     public int compare(T,T);
   };
```

Looking at the body of the `has_compare` predicate (from Figure 2.28) we see that JTL's systematic semantics remains intact even if the native predicates, making up the standard library, are now examining a program in a different language.

Admittedly, notational differences between JAVA and C# may manifest some problems, which will somewhat broaden the abstraction gap between #TL and C# (compared to JTL and JAVA).

In particular, the colon symbol, `:`, is used in C# to denote inheritance. Therefore, it would have been only natural to use this symbol in #TL to denote the immediate inheritance relationship. In other words, we expect the #TL expression `A : B` to hold if `B` is the direct super-class of `A` or if `B` is a direct super-interface of `A`.

The problem that prevents #TL from supporting such a notation is that the colon symbol already carries a special meaning in #TL (and in JTL): It is the query generator operator. Therefore, if one wants to turn colon into a predicate, the query generation operator be changed accordingly.

Another issue worth examining is that of dataflow analysis. As described in Section 2.2, JTL's scratch value represent the temporary values that are created during the execution of a JAVA method. It turns out that the execution process of C# is quite similar to that of JAVA; both languages rely on an evaluation stack and on an array of local variables. A typical instruction pops one or more value off the stack and pushes the result back onto it. Finally, the MSIL specification [70] defines a verification process that is similar, in principle, to the one defined in the JVM [127] specification

However, despite these similarities, implementing support for **`SCRATCH`** values on top of the .NET runtime environment is an intricate problem. First, C# allows parameters to be passed by reference (via the **`out`** modifier), which considerably complicates the data flow analysis algorithm. Second, a C# program may use **`unsafe`** code blocks. Code in such blocks may break the guarantees which the verifier is trying to establish, thereby reducing the accuracy of the detection

of scratches.

We therefore conclude, that implementing support for scratches in #TL is considerably more difficult than in JAVA due to the inherent properties of C#.

## 2.6 Discussion and Related Work

Tools and research artifacts which rely on the analysis of program source code are abundant in the software world, including metrics [53] tools, reverse-engineering [25], smart CASE enhancements [106], configuration management [33], architecture discovery [91], requirement tracing [93], AOP) [118], software porting and migration [121], program annotation [6], and many more.

The very task of code analysis per se is often peripheral to such products. It is therefore no wonder that many of these gravitate toward the classical and well-established techniques of formal language theory, parsing and compilation [4]. In particular, software is recurringly represented in these tools in an AST.

JTL is different in that it relies of a flat relational model, which, as demonstrated in Section 2.2.1, can also represent an AST. (Curiously, there were recently two works [90, 134] in which relational queries were used in object-oriented software engineering; however, these pertained to program execution trace, rather than to its static structure.)

JTL aspires to be a universal tool for tool writers, with applications such as specification of pointcuts in AOP, the expression of type constraints for generic type parameters, mixin parameters, selection of program elements for refactoring, patterns discovery, and more.

The community has already identified the need for a general-purpose tool or language for processing software. The literature describes a number of such products, ranging from dedicated languages embedded into larger systems to attempts to harness existing languages (such as SQL or XQUERY [36]) to this purpose. Yet, despite the vast amount of research invested in this area, no single industry standard has emerged.

A well-known example is REFINE [158], part of the *Software Refinery Toolset* by Reasoning Systems. With versions for C, FORTRAN , COBOL and ADA [174], Software Refinery generated an AST from source code and stored them in a database for later searches. The AST was then queried and transformed using the REFINE language, which included syntax-directed pattern matching and compiled into COMMON LISP, with pre- and post-conditions for code transformations. This meta-development tool was used to generate development tools such as compilers, IDEs, tools for detecting violations of coding standards, and more.

Earlier efforts include *Gandalf* [100], which generated a development environment based on language specifications provided by the developers. The generated systems were extended using the ARL language, which was tree-oriented for easing AST manipulations. Other systems that generated database information from programs and allowed user-developed tools to query this data included the *C Information Abstractor* [50], where queries were expressed in the INFOVIEW language, and its younger sibling *C++ Information Abstractor* [96], which used the DATASHARE language.

A common theme of all of these, and numerous others (including systems such as *GENOA* [67], *TAWK* [97], *Ponder* [19], *ASTLog* [59], *SCRUPLE* [156] and more) is the AST-centered approach. In fact, AST-based tools became so abundant in this field that a recent such product was entitled *YAAB*, for "Yet Another AST Browser" [16]. Another category of products contains those which rely on a relational model. For example, the *Rigi* [145] reverse engineering tool, which translates a program into a stream of triplets, where each triplet associates two program entities with some relation.

Section 2.6.1 compares JTL syntax with other similar products. Section 2.6.2 says a few words

**Figure 2.29** Eichberg et. al [73] example: search for EJBs that implement `finalize` in XIRC (a) and JTL (b).

```
subtypes(/class[
      @name="javax.ejb.EnterpriseBean"])
  /method[
      @name = "finalize"
      and .//returns/@type = "void"
      and not(.//parameter)
  ]
```

(a) XIRC implementation of the query (from [73]).

```
class implements /javax.ejb.EnterpriseBean {
   public void finalize();
};
```

(b) The JTL equivalent of (a).

on the comparison of relational- rather than an AST- model, for the task of queering object-oriented languages.

### 2.6.1 Using Existing Query Languages

*"Reading a poem in translation is like kissing your lover through a handkerchief."*
H. N. BIALIK (1917)

Many tools use existing languages for making queries. YAAB, for example, uses the Object Constraint Language, OCL, by Rational Software, to express queries on the AST; the *Software Life Cycle Support Environment* (SLCSE) [171] is an environment-generating tool where queries are written in SQL; Rigi's triples representation is intended to be further translated into a relational format, which can be queried with languages such as SQL and PROLOG; etc.

BDDBDDB [181] is similar to JTL in that it uses DATALOG for analyzing software. It is different from JTL in that it concentrates on the specific objective of code optimization, e.g., escape analysis, and does not further abstract the underlying language.

In XIRC [73], program meta-data is stored in an XML format, and queries are expressed in XQUERY. JQuery [111] is an Eclipse plugin that uses a deduction engine for evaluating DATALOG queries.

Finally, ALPHA [155] promotes the use of PROLOG queries for expressing pointcuts in aspect-oriented programming. We next compare queries made with some of these languages with the JTL equivalent.

Figure 2.29(a) depicts an example (due to the designers of XIRC) of using XQUERY to find Enterprise JavaBeans (EJB) which implement `finalize()`, in violation of the EJB specification.

In inspecting the figure, we find that in order to use this language the programmer must be intimately familiar not only with the XQUERY language, but also with the details of the XIRC encoding, e.g., the names of attributes where entity names, return type, and parameters are stored. A tool developer may be expected to do this, probably after climbing a steep learning curve, but its seems infeasible to demand that an IDE user will interactively type a query of this sort to search for similar bugs.

The JTL equivalent (Figure 2.29(b)) is a bit shorter, and less foreign to the JAVA programmer.

Table 2.2: Rewriting JQuery [111] examples in JTL.

| Task | JQuery | JTL |
|---|---|---|
| Finding class "BoardManager" | `class(?C,name,BoardManager)` | `class BoardManager` |
| Finding all "main" methods | `method(?M,name,main)`<br>`method(?M,modifier,[public,static])` | `public static main(*)` |
| Finding all methods taking a parameter whose type contains the string "`image`" | `method(?M,paramType,?PT)`<br>`method(?PT,/image/)` | `(*,R,*), R '*image?*'` |

Figure 2.29 demonstrates what we call *the abstraction gap*, which occurs when the syntax of the queries is foreign to the queried items. Word-processing and other office automation applications present no (or minimal) abstraction gap. For example, the search string which a user enters in the a typical text search box is usually identical to the strings which it matches. The database users community is accustomed to Query-be-Example.

JTL strives to bring this ideal to the world of language processing tools. This makes JTL not only shorter to type and, probably, but it is also makes it almost obvious for any JAVA programmer to learn the language.

The ASPECTJ sub-language for pointcut definition, just as the sub-language used in JAM [12] for setting the requirements for the base class of a mixin, also exhibit minimal abstraction gap. The challenge that JTL tries to meet is to achieve this objective with a more general language.

We next compare JTL syntax with that of JQuery [111], which also relies on Logic programming for making code queries. Table 2.2 compares the queries used in JQuery case study (extraction of the user interface of a chess program) with their JTL counterparts.

The table shows that JTL queries are a bit shorter and resemble the code better. The JTL expression in the last row can be explained by the following: To find a method in which one of the type of parameters contains a certain word, we do a pattern match on its argument list, allowing any number of arguments before and after the argument we seek. The type of the desired argument itself is expressed by matching its name with a regular expression

Figure 2.30 is an example of using JAVA's reflection APIs to implement a query—here, finding all **`public final`** methods (in a given class) that return an **`int`**.

Examining Figure 2.30 we can observe three things:

- Figure 2.30 uses JAVA's familiar syntax, but this comes at the cost of replacing the declarative syntax in Figure 2.29 with explicit control flow.
- Despite the use of plain JAVA, Figure 2.30 manifests an abstraction gap, by which the pattern of matching an entity is very different from the entity itself.
- The code still assumes familiarity with an API; it is unreasonable to expect an interactive user to type in such code.

Again, the JTL equivalent, `public final int(*)`, is concise, avoids complicated control flow, and minimizes the abstraction gap.

**Figure 2.30** Comparison of JAVA's reflection library with JTL.

```
public Method[] pufim_reflection(Class c) {
  List<Method> list = new ArrayList<Method>();
  for (Method m : c.getMethods()) {
    int mod = m.getModifiers();
    if (m.getReturnType() == Integer.Type
      && Modifiers.isPublic(mod)
      && Modifiers.isFinal(mod))
        list.add(m);
  }
  return list.toArray(new Method[0]);
}
```

(a) Eliciting public final int methods with JAVA's reflection library

```
public final int (*)
```

(b) Eliciting public final int methods with JTL

There are, however, certain limits to JTL's similarity to JAVA, the most striking one being the fact that in JTL, an absence of a keyword means that its value is unspecified, whereas in JAVA, the absence of e.g., `static` means that this attribute is off. This is expressed as `!static` in JTL.

Another interesting comparison with JTL is given by considering ALPHA and Gybels and Brichau's [98] "crosscut" language, since both these languages rely on the logic paradigm. Both languages were designed solely for making pointcut definitions (Gybels and Brichau's work, just as ours, assumes a static model, while ALPHA allows definitions based on execution history). It is no wonder that both are more expressive in this than the reference ASPECTJ implementation.

Unfortunately, in doing so, both languages *broaden* rather than narrow the abstraction gap of ASPECTJ. This is a result of the strict adherence to the PROLOG syntax, which is very different than that of JAVA. Second, both languages make heavy use of recursive calls, potentially with "cuts", to implement set operations. Third, both languages are fragile in the sense described above

We argue that even though JTL is not specific to the aspect-oriented domain, it can do a better job at specifying pointcuts pertaining to the static structure of the program at hand.

### 2.6.2 AST vs. Relational Model

We believe that the terse expression and the small abstraction gap offered by JTL is due to three factors: (*i*) the logic programming paradigm, notorious for its brevity, (*ii*) the effort taken in making the logic programming syntax even more readable in JTL, and (*iii*) the selection of a relational rather than a tree data model.

We now try to explain better the third factor. Examining the list of tools enumerated early in this section we see that many of these rely on the *abstract syntax tree* metaphor. The reason that ASTs are so popular is that they follow the BNF form used to define languages in which software is written. ASTs proved useful for tasks such as compilation, translation and optimization; they are also attractive for discovering the architecture of structured programs, which are in essence ordered trees.

We next offer several points of comparison between an AST based representation and the set-based, relational approach represented by JTL and other such tools. Note that as demonstrated in

Section 2.2.1, and as Crew's ASTLog language [59] clearly shows, logic programming does not stand in contradiction with a tree representation.

- *Unordered Set Support.* In traditional programming paradigms, the central kind of modules were procedures, which are sequential in nature. In contrast, in JAVA (and other object-oriented languages) a recurring metaphor is the unordered *set*, rather than the *sequence*: A program has a set of packages, and there is no specific ordering in these. Similarly, a package has a set of classes, a class is characterized by a set of attributes and has a set of members, each member in turn has a set of attributes, a method may throw a set of exceptions, etc. Although sets can be supported by a tree structure, i.e., the set of nodes of a certain kind, some programming work is required for set manipulation which is a bit more *natural and intrinsic* to relational structures.

  On the other hand, the list of method arguments is sequential. Although possible with a relational model, ordered lists are not as simple. This is why JTL augments its relational model with a built-in for dealing with lists (namely, the Argument List Pattern, Section 2.1.2)).

- *Data Model Complexity.* An AST is characterized by a variety of kinds of nodes, corresponding to the variety of syntactical elements that a modern programming language offers. A considerable mental effort must be dedicated for understanding the recursive relationships between the different nodes, e.g., which nodes might be found as children or descendants of a given node, what are the possible parent types, etc.

  The underlying complexity of the AST prevents a placement of a straightforward interface at the disposal of the user, be it a programmatic interface (API), a text query interface or other. For example, in the *Hammurapi*[11] system, the rule "*Avoid hiding inherited instance fields*" is implemented by more than 30 lines of JAVA code, including two **`while`** loops and several **`if`** clauses (see Figure 2.31). The corresponding JTL pattern is so short it can be written in one line:

  ```
  class extends S { field same_name F, S offers F; };
  ```

  The terse expression is achieved by the uniformity of the relational structure, and the fact that looping constructs are implicit in JTL queries.

- *Recursive Structure.* One of the primary advantages of an AST is its support for the recursive structures so typical of structured programming.

  Similar recursion of program information is less common in modern languages. JAVA does support class nesting (which are represented using the `inners` predicate of JTL) and methods may (but rarely do) include a definition of nested class. Also, a class cannot contain packages, etc.

- *Representation Granularity.* Even though recursively defined expressions and control statements still make the bodies of object-oriented methods, they are abstracted away by our model.

  JTL has native predicates for extracting the parameters of a method, its local variables, and the external variables and methods which it may access, and as shown, even support for dataflow analysis. In contrast, ASTs make it easier to examine the control structure. Also, with suitable AST representation, a LINT-like tool can provide warnings that JTL cannot, e.g., a non-traditional ordering of method modifiers.

[11] http://www.hammurapi.org

**Figure 2.31** The implementation of Hammurapi's *Avoid Hiding Inherited Fields* rule.

```
public class HidingInheritedFieldsRule extends InspectorBase {
  public void visit(Class clazz) {
    try {
      TypeIdentifier superclass = clazz.getSuperclass();
      if (superclass!=null) {
        Class parentClass=(Class) superclass.find();
        if (parentClass!=null) {
          Set parentFields=new HashSet();
          Iterator it=parentClass.getFields().iterator();
          while (it.hasNext()) {
            Field field = (Field) it.next();
            String cPkg = clazz.getCompilationUnit()
              .getPackage().getName();
            String sPkg = superclass.getCompilationUnit()
              .getPackage().getName();
            boolean isVisible=cPkg.equals(sPkg) ?
              !field.getModifiers().contains("private")
                : (field.getModifiers().contains("public") ||
                  field.getModifiers().contains("protected"));
            if (isVisible && field instanceof VariableDefinition)
              parentFields.add(((VariableDefinition) field)
                .getName());
          }

          it=clazz.getFields().iterator();
          while (it.hasNext()) {
            Field field = (Field) it.next();
            if (field instanceof VariableDefinition &&
              parentFields.contains(((VariableDefinition) field)
                .getName())) {
              VariableDefinition vd=(VariableDefinition) field;
              if (!("serialVersionUID".equals(vd.getName()) &&
                vd.getTypeSpecification().isKindOf("long")))
                context.reportViolation(field);
            }
          }
        }
      }
    } catch (JselException e) {
      context.warn(clazz, "Could not resolve parent class: "+e);
    }
  }
}
```

It should be said that the importance of analyzing method bodies in object-oriented software is not so great, particularly, since object-oriented methods tend to be small [53], and in contrast with the procedural approach, their structure does not reveal much about software architecture [91]. Also, in the object-oriented world, tools are not so concerned with the algorithmic structure, and architecture is considered to be a graph rather than a tree [106].

- *Theory of Searches.* Relational algebra, SQL, and DATALOG are only part of the host of familiar database searching theories. In contrast, searches in an AST require the not-so-trivial VISITOR design pattern, or frameworks of factories and delegation objects (as in the Polyglot [150] project). This complexity is accentuated in languages without *multi-methods* or *open classes* [49] but occur even in more elaborate languages.

- *Representation Flexibility.* A statically typed approach (as in Jamoos [88]) can support the reasoning required for tasks such as iteration, lookup and modification of an AST. Such an approach yields a large and complex collection of types of tree nodes. Conversely, in a weakly-typed approach (as in REFINE), the complexity of these issues is manifested directly in the code.

  Either way, changes in the requirements of the analysis, when reflected in changes to the kind of information that an AST stores, often require re-implementation of existing code, multiplying the complex reasoning toll. This predicament is intrinsic to the AST structure, since the search algorithm must be prepared to deal with all possible kinds of tree nodes, with a potentially different behavior in different such nodes. Therefore, the introduction of a new kind of node has the potential of affecting all existing code.

  In contrast, a relational model is typically widened by adding new relations, without adding to the basic set of simple types. Such changes are not likely to break, or even affect most existing queries.

## 2.7 Summary

JTL is a novel, DATALOG-based query language designed for querying JAVA programs in binary format. The JTL system can be extended to query programs written in other programming-languages (C#, SMALLTALK [89]), possibly in a different input formats. Such extensions require mostly the native predicates to be replaced with new ones which are made to inspect the desired from of input.

JTL's SCRATH values make it possible to examine the execution of blocks of imperative instructions. This allows JTL users to search for program elements by conditioning their *behavior*, in addition to their *declaration*.

We note that the detection of scratch values relies on JAVA's verification process which guarantees certain properties, of the dataflow graph, in every legal method. Therefore, the use of scratch-related predicates over a languages that has a weaker verification process, such as C#, is limited.

The relational nature of JTL leads to a simple data model that seems to be more suitable for the representation of programs than hierarchical models. This simplicity results in a notation that is terse yet intuitive.

# Chapter 3

# Micro Patterns

We all know what makes one algorithm better than another: time, space, disk access, network utilization, etc. are established, *objective* and well defined metrics [58] to be employed in making such a judgment. In contrast, the assessment of quality of software design is an elusive prospect. Despite the array of books and research articles on the topic (see e.g., [46, 51, 129, 135]), a question such as "*Is Design $A$ better than Design $B$?*" can, still, only be decided by force of the argumentation, and ultimately, by the personal and *subjective* perspective of the judge.

Medical experiments can prove that a certain medication is better than another in treating a specific ailment. We all want to carry similar controlled experiments to prove that certain design methods are more likely to produce better software than others. However, in contrast with many other natural sciences, experiments on large scale software development are so prohibitively costly that much of the research on the topic abandoned this hope [65].

Productivity in software is correlated with quality of design: Programmers working on a well-designed code are expected to be more productive than those working on ill-designed code. Our ability to estimate the (design) quality of a software product is therefore limited due to the lack of effective productivity metrics.

Martin Fowler summarizes this as follows:[1]

> "*I can see why measuring productivity is so seductive. If we could do it we could assess software much more easily and objectively than we can now. But false measures only make things worse. This is somewhere I think we have to admit to our ignorance.*"

Acknowledging this predicament, this chapter explores the utility of formal patterns for *precise elicitation of design*. Our results make the first, essential, step towards a different kind of solution to the question of evaluation of design. Instead of dealing with "is *$A$ better than $B$?*" sort of questions, we hope to be able to determine "*how is $A$ different than $B$?*". We believe that the latter question is a more tractable than the former, and that it represents a viable direction in future research of formal evaluation of design of software.

Our approach is empirical: we scan a large corpus of JAVA programs looking for traces of design by detecting code fragments (which we shall later precisely define as *micro patterns*) of interest.

This angle is made possible by two factors. First, the bountiful class structure of JAVA, together with the colossal, publicly available, base of software in the language, which opens the road for sound claims and understanding of the way people write software (more precisely, on the software written by people).

[1] http://martinfowler.com/bliki/CannotMeasureProductivity.html

Second, we were able to make our definitions of these code fragments precise by basing them on JTL's mathematically sound model. The alternative—using an informal (possibly natural) language for describing code fragments—has several flaws. Such descriptions are prune to inconsistencies and logical contradictions which compromise the ability to draw scientific conclusions from the gathered data.

## 3.1 Definition and Applications

Can design be traced and identified in software? The prime candidates of units of design to look for in the software are obviously *design patterns* [81]. However, despite the many years that passed since the original publication [80], and the voluminous research ensuing it, attempts to automate and formalize design patterns are scarce. Systems like DisCo [138], LePUS [71, 72], SPINE and HEDGEHOG [35], constraint diagrams [124], Elemental Design Patterns [167], and others did not gain much popularity. Specific research on detection of design patterns exhibited low precision, typically with high rate of false negatives (see e.g., [42,105]). Indeed, as Mak, Choy and Lun [130] say, "*... automation support to the utilization of design patterns is still very limited*".

This is a manifestation of the inherent trade-off between detectability and design information. Low level software artifact (such as: imperative statement) are easily detectable—from either binary or source code representations of the software—but readily provide only a small amount of design information. On the other side of the equation, design patterns carry substantial design information but their detection is hard.

This trade-off is not surprising. Software construction is a process of gradual refinement [182] starting from a vague idea and ending with a well-formed, ready to run executable expressed in some formal language. At each refinement step the information from the previous step is translated into some new form and/or elaborated into finer granularity. Unfortunately, the translation is not necessarily reversible. A single idea can be realized by several distinct implementation. Conversely, a piece of implementation may participate in the realization of several distinct ideas. This implies that information is lost during this process.

**Micro Patterns.** This chapter argues that a particular subset of formal patterns, namely *Micro Patterns*, strike the aforementioned trade-off at its sweet spot: they are traceable, yet still carry enough high-level information to provide substantial design details. As such they can serve as a mechanism for the extraction of design information from a program.

**Definition 4** *a micro pattern is a well-defined condition on the attributes, types, name and body of a class and its components, which is mechanically recognizable, purposeful, and simple.*

According to the definition, micro patterns are a specific kind of Formal Patterns (see Definition 1) and can be though of as "*class-level formal patterns* ". When no confusion can arise we shall, for the sake of brevity, call these just patterns.

We present (Section 3.2) a catalog of micro patterns, organized in 8 categories, including idioms for a particular and intentionally restricted use of inheritance, immutability, wrapping and data management classes, object-oriented emulation of procedural, modular and even functional programming paradigms, innovative use of class structure, and many more.

**Examples.** A simple example for concreteness is the Sampler pattern in the *Controlled Creation* category. This pattern defines classes which have a **`public`** constructor, but in addition have one or more **`static public`** fields of the same type as the class itself. The purpose of such classes is to give clients access to pre-made instances of the class, but also to create their own. The Sampler is realized by, e.g., class `Color` from package `java.awt` of the JAVA standard runtime environment, which offers a spectrum of pre-defined colors as part of its interface.

Another example of a micro pattern is the Immutable pattern [95] in the *Degenerate State* category. This pattern prescribes an object whose state cannot be changed after its construction.

The reader is invited to take a sneak preview at the entire catalog of patterns in Section 3.2 for further examples.

### 3.1.1 Micro patterns and Productivity

Other than serving for a more rigorous study of design, our catalog, just as many other collection of patterns, can help in *documentation*, in conveying a *knowledge base*, and in setting a *vocabulary* for communication among and between coders and designers.

The vocabulary that the catalog sets can come handy in the description of implementation strategies of design patterns. Terms such as Immutable, Box, Canopy, Pure Type or Implementor (all patterns from the catalog) are useful in describing the implementation of design patterns such as DECORATOR, BRIDGE, PROXY, etc. On a broader perspective, software frameworks may use this terminology to describe the various sorts of classes which take part in the framework.

Our empirical study demonstrates the consistent abundance of each of the patterns; further, the entire catalog characterizes about three quarters of the classes in our corpus. These finding support the claim that micro patterns can enhance the following aspects of software engineering productivity:

- *More Efficient Design.* The catalog captures a substantial body of knowledge gathered from a massive software corpus. The use of this knowledge base can make the design and implementation stages more efficient, by using one of the recipes in the catalog, rather than designing a class from scratch.

  The mental effort saved by using familiar, named patterns for certain classes, can be redirected to more important and difficult tasks.

- *Code Learning and Reuse.* Familiarity with the catalog makes it possible for programmers to quickly understand an overwhelming majority of the JAVA software base. They can then focus more attention to the smaller fraction of the remaining code, which presumably requires closer examination.

- *Training.* By learning the patterns in the catalog, programmers can be quickly introduced to the tools of trade of JAVA programming.

- *Automation.* Micro-patterns traceability makes it also possible to *enrich* automatically generated documentation produced by tools such as JavaDoc[2].

### 3.1.2 New Language Constructs

Micro patterns are not only patterns of class design. They are also patterns (in the information theoretical sense of the term) by which the programmer makes selections from this huge space of different combinations of class features.

Consider for instance the many different kinds of fields that a JAVA class can have: they can be **`static`** or non-**`static`**, **`final`** or not-**`final`**, inherited or introduced by the class, and they can exhibit one of four different kinds of visibility. Methods show an even greater variety, since they can also be **`abstract`**, **`final`**, overriding, or even *refining*; and, there are also constructor methods, and anonymous static initializers, etc. Our count shows that there are over forty different kinds of class members, without even considering variety due to type signature or naming.

[2] http://java.sun.com/j2se/javadoc

By recognizing that the expressive power of the programming language might be too large, we may be lead not only to a more structured system of teaching design, but also of maturing some of these combinations into full blown language constructs.

The precise definition of micro patterns makes it possible to evolve some of the patterns into language constructs, in the manner suggested by Agerbo and Cornils [2] regarding the incorporation of design patterns into a programming language (interestingly, the motivation for JAVA's new **enum** facility is reflected by the prevalence of Augmented Type and Pool micro pattern).

## 3.2 The Micro Pattern Catalog

This section presents our original catalog of 27 micro patterns. Two additional patterns, recently identified, are presented in Appendix C.[3] We start by discussing the process by which these patterns were conceived. We continue with a brief overview of the patterns which is followed by a detailed definition of each.

The search for micro patterns started by considering the various kinds of features that a JAVA class may have. We tried then to work out meaningful and useful restrictions on the freedom in using these features. To do so, we raised questions such as "*how could a class with no fields be useful?*", "*are there any classes of this sort in existence?*", "*how can these classes be characterized?*". Conversely, having thought of a useful programming practice, we tried to translate it into a condition on the code, and then inspect classes that matched this condition.

We implemented each of these initial "pre-patterns" and applied them to the classes in the corpus. Manual inspection of the code of matching classes lead to a refinement of some of the definitions, abandonment of others, merges and splits of others, until the catalog reached its current shape.

Thus, the search for patterns started from definitions, which lead to code inspection, and then to the refinement of the definitions.

It is tempting to do the converse, i.e., cluster the existing code base, and discover patterns in it, devoid of any a priori dictations. To do so we, we tried several approaches. For example, we broke down the conditions we already discovered into atomic predicates ("basic features" in the learning lingo), such as "number of instance fields is 1", "no superinterfaces", etc.

The values of these predicates on the classes in the software corpus were then fed into an associations rules analyzer [183]. In return, the analyzer generated long lists of dependencies between these predicates, sorted in descending order of strength.

Unfortunately, none of these dependencies revealed something that we could have interpreted as a purposeful pattern. Other established techniques of machine learning did not work for us.

**Catalog Overview.** Consider Figure 3.1, which shows a global map of the catalog, including the 8 categories, and the placement of the 27 micro patterns into these. (The patterns themselves are described in brief in Table 3.1.)

The $X$-dimension of Figure 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.

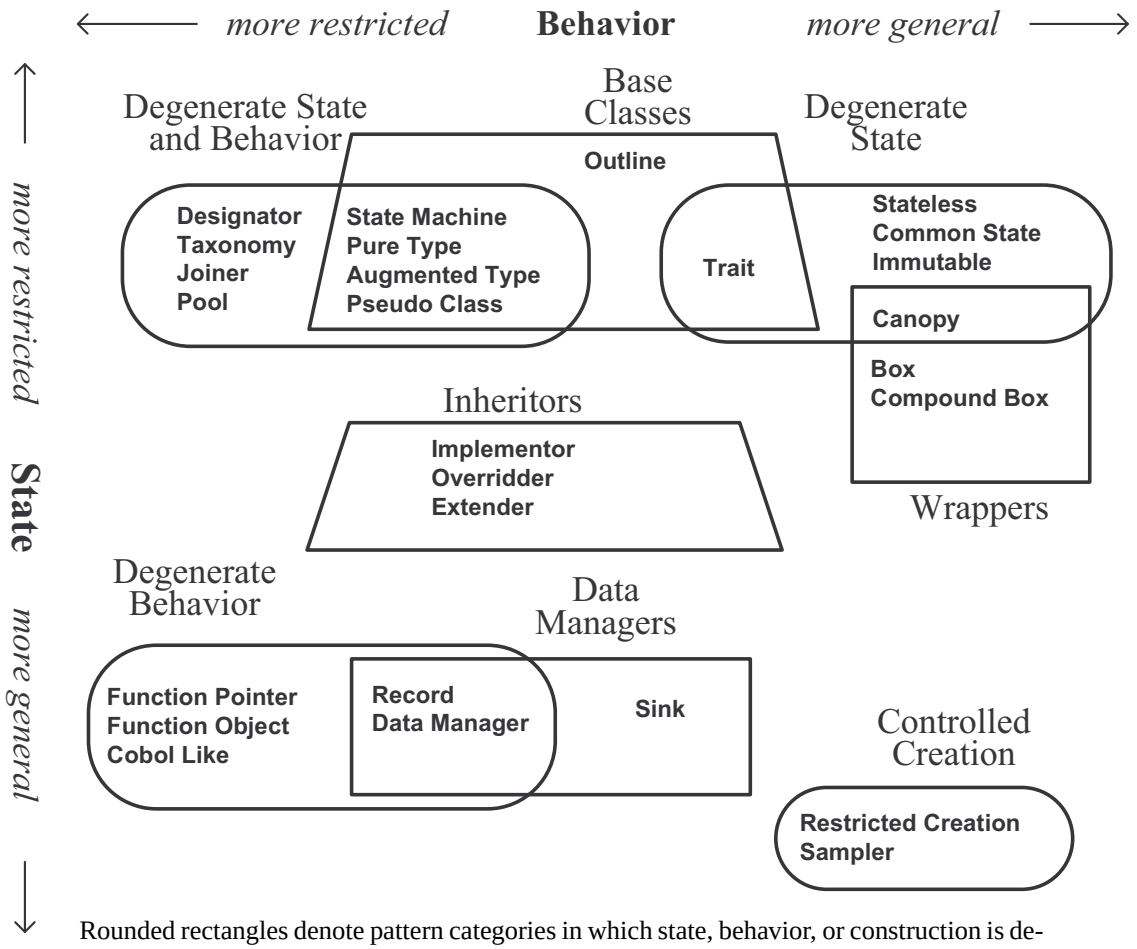
Altogether, there are four categories (depicted as rounded corners rectangles in the figure) in which the class behavioral, or creational or variability (state) aspects of a class are degenerate: *Degenerate State and Behavior*, *Degenerate State*, *Degenerate Behavior* and *Restricted Creation*.

[3] The analysis/discussion presented in this chapter pertains to our original catalog.

| | Main Category | Pattern | Short description | Additional Category |
|---|---|---|---|---|
| **Degenerate Classes** | Degenerate State and Behavior | Designator | An interface with absolutely no members. | |
| | | Taxonomy | An empty interface extending another interface. | |
| | | Joiner | An empty interface joining two or more superinterfaces. | |
| | | Pool | A class which declares only static final fields, but no methods. | |
| | Degenerate Behavior | Function Pointer | A class with a single public instance method, but with no fields. | |
| | | Function Object | A class with a single public instance method, and at least one instance field. | |
| | | Cobol Like | A class with a single static method, but no instance members | |
| | Degenerate State | Stateless | A class with no fields, other than static final ones. | |
| | | CommonState | A class in which all fields are static. | |
| | | Immutable | A class with several instance fields, which are assigned exactly once, during instance construction. | |
| | Controlled Creation | Restricted Creation | A class with no public constructors, and at least one static field of the same type as the class | |
| | | Sampler | A class with one or more public constructors, and at least one static field of the same type as the class | |
| **Containment** | Wrappers | Box | A class which has exactly one, mutable, instance field. | |
| | | Compound Box | A class with exactly one non primitive instance field. | |
| | | Canopy | A class with exactly one instance field that it assigned exactly once, during instance construction. | Degenerate State |
| | Data Managers | Record | A class in which all fields are public, no declared methods. | Degenerate Behavior |
| | | Data Manager | A class where all methods are either setters or getters. | |
| | | Sink | A class whose methods do not propagate calls to any other class. | |
| **Inheritance** | Base Classes | Outline | A class where at least two methods invoke an abstract method on “this” | Degenerate State |
| | | Mould | An abstract class which has no state. | |
| | | State Machine | An interface whose methods accept no parameters. | Degenerate State and Behavior |
| | | Pure Type | A class with only abstract methods, and no static members, and no fields | |
| | | Augmented Type | Only abstract methods and three or more static final fields of the same type | |
| | | Pseudo Class | A class which can be rewritten as an interface: no concrete methods, only static fields | |
| | Inheritors | Implementor | A concrete class, where all the methods override inherited abstract methods. | |
| | | Overrider | A class in which all methods override inherited, non-abstract methods. | |
| | | Extender | A class which extends the inherited protocol, without overriding any methods. | |

Table 3.1: Micro patterns in the catalog

**Figure 3.1** A map of the micro patterns catalog




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.

Depicted as rectangles in Figure 3.1, there are two categories pertaining to containment: The *Data Managers* category is that of patterns which directly store and manage data; The *Wrappers* category contains patterns which wrap other classes.

Finally, there are also two categories pertaining to inheritance: *Base Classes* and *Inheritors*. These categories are portrayed as trapezoids in the figure.

Table 3.1 gives an alternative, textual representation of the information depicted in Figure 3.1. As can be seen in the table (and also in Figure 3.1), the categories are not disjoint. There are a number of patterns which belong in two categories. The last column of the table shows the additional category of such patterns.

For example, Pseudo Class pattern belongs both to the *Degenerate State and Behavior* and the *Base Classes* categories; Pure Type is a *Base Class* which also exhibits *Degenerate State and Behavior*. Such patterns are described in one of their categories, and merely mentioned in the others.

This table also tersely describes each of the patterns. It is important however to note that this one line description, by nature, cannot be precise or complete. To see that, recall that there are many, not necessarily disjoint, kinds of methods which JAVA admits, including inherited methods, static methods, concrete methods, abstract methods, constructors, etc.

There are therefore several ways of translating a simple statement such as *"all methods are public"* into a precise and complete condition on the code. For example, one needs to decide

whether the universal quantification in this statement precludes inheriting protected methods.

Hence, the descriptions presented in Table 3.1 should serve merely as an intuitive summary. Precise definitions of the micro patterns are given, using JTL notation, in the subsequent pages.

The remainder of this section provides a detailed definition of each of the patterns. Presentation of each pattern starts with a short, plain-English, description followed by a precise JTL definition. We then briefly discuss the purpose of the pattern and/or typical usage scenarios. This is followed by the source code of a class (from the Java standard library) matching the pattern. Finally, we present the prevalence of this pattern in our corpus.

When possible the full source code of the matching class is presented (sans **`import`** statements). Otherwise, we omitted details inessential to the illustration of the pattern.

The JTL definitions in the catalog solely rely on the JTL Standard library (Appendix B). No other library predicates are used.

## Augmented Type

An abstract type whose list of members (excluding those defined by `Object`) contains only methods which must all be abstract. At least one such method must exist. All declared fields (other than `serialVersionUID`) must be **`static final`**, visible, and of the same type. At least two such fields must exist.

### Definition

```
augmented_type := abstract {
     field implies visible static final;
     field implies typed T | ser_ver_uid;
     many field;
   }
   non_global_members: {
     no concrete method;
     instance method;
   }  declares F, F visible & static & final & field & typed T;
```

### Purpose

There are many interfaces and classes which declare a type, but the definition of this type is not complete without an auxiliary definition of an *enumeration*. An enumeration is a means for making a new type by restricting the (usually infinite) set of values of an existing type to smaller list whose members are individually enumerated.

### Example

```
package java.sql;

public interface Statement extends Wrapper {
   int CLOSE_CURRENT_RESULT = 1;
   int KEEP_CURRENT_RESULT = 2;
   int RETURN_GENERATED_KEYS = 1;
   int NO_GENERATED_KEYS = 2;
   // Additional constants...

   int[] executeBatch() throws SQLException;
   Connection getConnection() throws SQLException;
   // Additional methods...
}
```

### Prevalence

0.5%

## Box

A class with exactly one instance field (including inherited fields). This instance field is assigned-to by at least one of the methods.

### Definition

```
box := offers: {
     one instance field;
   }
   offers Y [Y mutator | Y visible & instance & field];
```

### Purpose

Box classes provide a set of services that are based on a single piece of state. This yields a highly cohesive code as methods cannot be field-wise partitioned.

### Example

```
public class CRC32 implements Checksum {
   private int crc;

   public void update(int b) { crc = update(crc, b); }
   public void reset() { crc = 0; }
   public long getValue() { return (long)crc & 0xffffffffL; }

   public void update(byte[] b) {
     crc = updateBytes(crc, b, 0, b.length);
   }

   public void update(byte[] b, int off, int len) {
     if (b == null)
       throw new NullPointerException();
     else if (off < 0 || len < 0 || off > b.length - len)
       throw new ArrayIndexOutOfBoundsException();
     else
       crc = updateBytes(crc, b, off, len);
   }

   // Native methods ...
}
```

### Prevalence

6.0%

## Canopy

A class with exactly one instance field (including inherited fields) which must be private. None of the methods assigns a new value to this field.

### Definition

```
canopy := offers: {
     no mutator;
     one instance field;
     one private instance field;
   };
```

### Purpose

Such classes often used for modeling values (numbers, dates, etc.) from the domain space. The name Canopy draws from the visual association of a transparent enclosure set over a precious object; an enclosure which makes it possible to see, but not touch, the protected item.

### Example

```
package java.security.spec;

public class ECFieldFp implements ECField {
   private java.math.BigInteger p;

   public ECFieldFp(BigInteger p) {
     if (p.signum() != 1)
       throw new IllegalArgumentException("p is not positive");
     this.p = p;
   }

   public int getFieldSize() { return p.bitLength(); };
   public BigInteger getP() { return p; }
   public int hashCode() { return p.hashCode(); }

   public boolean equals(Object obj) {
     if (obj instanceof ECFieldFp)
       return (p.equals(((ECFieldFp)obj).p));
     return false;
   }
}
```

### Prevalence

7.7%

## Cobol Like

A class that offers a single method (excluding `Object` methods) which must be static. Instance fields are not allowed.

### Definition

```
cobol_like := non_global_members: {
     no instance field;
     no instance method;
     one static method;
   };
```

### Purpose

This particular programming style makes a significant deviation from the object-oriented paradigm. It is often used for porting algorithm from a non-object-oriented language into JAVA.

### Example

```
package java.lang;

class StringValue {
   static char[] from(char ac[]) {
     return Arrays.copyOf(ac, ac.length);
   }
}
```

### Prevalence

0.5%

## Common State

A class whose list of members (excluding those defined by `Object`) includes no instance methods, no instance fields, and at least one non-final static field.

### Definition

```
common_state := class non_global_members: {
     no instance method;
     no instance field;
     exists !final static field;
   };
```

### Purpose

Unlike Stateless classes, CommonState classes maintain state, but this state is shared by all of their instances. A CommonState with no instance methods is in fact an incarnation of the packages mechanism of ADA.

### Example

```
package java.lang;

public final class System {

   private static native void registerNatives();
   static { registerNatives(); }
   private System() {}

   public final static InputStream in = nullInputStream();
   public final static PrintStream out = nullPrintStream();
   public final static PrintStream err = nullPrintStream();
   private static volatile Console cons = null;
   private static volatile SecurityManager security = null;

   public static void setIn(InputStream in) { ... }
   public static void setOut(PrintStream out) { ... }
   public static void setErr(PrintStream err) { ... }
   public static Console console() { ... }
   ...
}
```

### Prevalence

3.8%

## Compound Box

A class whose list of members (including inherited ones) includes exactly one non-primitive instance field, and one or more primitive instance fields.

### Definition

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

### Purpose

This is a variant of a Box pattern where that most of the state is provided by the non-primitive field and auxiliary, bookkeeping data, is maintained by the primitive fields.

### Example

```
package java.util;

public class ArrayList<E> extends AbstractList<E>
     implements List<E>, RandomAccess, Cloneable, Serializable {
   private transient Object[] elementData;
   private int size;
   private static final long serialVersionUID = ...;

   public ArrayList(int initialCapacity) {
     if (initialCapacity < 0)
       throw new IllegalArgumentException(...)
     this.elementData = new Object[initialCapacity];
   }

   public ArrayList() { this(10); }

   public E get(int index) {
     RangeCheck(index);
     return (E) elementData[index];
   }

   ...
}
```

### Prevalence

4.4%

## Data Manager

A class where its list of members (including inherited, excluding those defined by `Object`) contains at least one instance field, and at least one non-private getter method. In addition, every non-private method on this list is either a setter or a getter.

### Definition

```
data_manager := non_global_members: {
     instance field;
     visible getter method;
     visible method implies [getter | setter];
   };
```

### Purpose

The Data Manager pattern is similar to the Record pattern (and thus is also useful for describing parameter objects) with the difference that the state is encapsulated. This reduces the coupling with client code, thus isolating it from future evolution of the class. Data Manager classes are typically used for parameter objects [31] that are published, as part an API, to unknown clients.

### Example

```
package java.net;

public final class PasswordAuthentication {
  private String userName;
  private char[] password;

  public PasswordAuthentication(String un, char[] pw) {
    userName = un;
    password = (char[])pw.clone();
  }

  public String getUserName() { return userName; }
  public char[] getPassword() { return password; }
}
```

### Prevalence

1.8%

## Designator

A Designator is an interface which does not declare any methods, nor static fields, nor does it inherit such members from any of its superinterfaces. A class can also be Designator if the class and all of its superclasses (excluding `Object`) declare only constructors.

### Definition

```
designator := abstract non_global_members: {
     no field;
     no method;
   };
```

### Purpose

Interestingly, vacuous interfaces are employed in a powerful programming technique, of tagging classes in such a way that these tags can be examined at runtime. For example, a class that **`implements`** the empty interface `Cloneable` indicates (at run time) that it is legal to make a field-for-field copy of instances of that class.

### Example

```
package java.lang;

public interface Cloneable {
}
```

### Prevalence

0.2%

## Extender

A class which extends the interface inherited from its superclass and super interfaces, but does not override any method. The superclass cannot be the `Object` class, thus precluding trivial matches with this pattern.

### Definition

```
extender := is T extends S, S is_not Object, !interface {
     visible instance method;
     visible instance method overrides X implies X global;
   };
```

### Purpose

A class following this pattern specifies evolution—along the inheritance chain—of the protocol but not of the behavior.

### Example

```
package java.util;

public class Stack<E> extends Vector<E> {

   private static final long serialVersionUID = ...;

   // Five non−overriding public methods:
   public E push(E item) { ... }
   public synchronized E pop() { ... }
   public synchronized E peek() { ... }
   public boolean empty() { ... }
   public synchronized int search(Object o) { ... }
}
```

### Prevalence

4.2%

## Function Object

A class that offers a single public method (excluding either `Object` methods or overridden methods) and at least one instance field.

### Definition

```
function_object := non_global_methods X, non_global_methods: {
     all same_name X, X signature_compatible #;
  }
  offers: {
    instance field is F;
  };
```

### Purpose

The Function Object pattern matches many anonymous classes in the JRE which implement an interface with a single method. These are mostly event handlers, passed as callback hooks in GUI libraries (AWT and Swing). Hence, such classes often realize the COMMAND design pattern.

### Example

```
package java.sql;

class DriverService implements PrivilegedAction {
  Iterator ps;

  public DriverService() { ps = null; }

  public Object run() {
    ps = Service.providers(java.sql.Driver.class);
    try {
      while(ps.hasNext())
        ps.next();
    }
    catch(Throwable throwable) { }
    return null;
  }
}
```

### Prevalence

5.5%

## Function Pointer

A class that offers a single public method (excluding either `Object` methods or overridden methods) and only static final fields (if any).

### Definition

```
function_pointer := non_global_methods X, non_global_methods: {
     all same_name X, X signature_compatible #
  }
  offers: {
    field implies final static;
  };
```

### Purpose

Classes following this pattern represent the equivalent of a function pointer (or a pointer to procedure) in the procedural programming paradigm, or of a function value in the functional programming paradigm. Such an instance can then be used to make an indirect polymorphic call to this function. Unlike Function Object, a Function Pointer maintains no mutable state.

### Example

```
package java.util.jar;

class JavaUtilJarAccessImpl implements JavaUtilJarAccess {

  public boolean jarFileHasClassPathAttribute(JarFile jar)
          throws IOException {
    return jar.hasClassPathAttribute();
  }
}
```

### Prevalence

1.6%

## Immutable

A class whose instance fields (at least two, including inherited ones) are only changed by its constructors.

### Definition

```
immutable := offers: {
     many instance field;
     instance field implies private;
     no mutator;
   };
```

### Purpose

Classes whose state is immutable are useful for functional style of programming: a computation does not change existing data but rather produces new data while keeping the existing data intact.

### Example

```
package javax.activation;

public class CommandInfo {
   private String verb;
   private String className;

   public CommandInfo(String verb, String className) {
     this.verb = verb;
     this.className = className;
   }
   public String getCommandName() { return verb; }
   public String getCommandClass() { return className; }

   public Object getCommandObject(DataHandler dh, ClassLoader cl)
     throws IOException, ClassNotFoundException {
     Object obj = Beans.instantiate(cl, className);
     if(obj != null)
           if(obj instanceof CommandObject)
             ((CommandObject)obj).setCommandContext(verb, dh);
           else if((obj instanceof Externalizable) && dh != null) {
             InputStream is = dh.getInputStream();
             if(is != null)
               ((Externalizable)obj).readExternal(
                 new ObjectInputStream(is));
           }
     return obj;
   }
}
```

### Prevalence

6.1%

## Implementor

A non-abstract class such that all of the public instance method it declares override an abstract method.

### Definition

```
implementor := {
     visible concrete instance method;
     visible concrete instance method implies
       precursor M M abstract;
   };
```

### Purpose

This class materializes abstract definitions without extending the protocol nor changing previously defined behavior. A common use for such classes is that of being a concrete product in creational design patterns such as ABSTRACT FACTORY.

### Example

```
package java.util.logging;

public class SimpleFormatter extends Formatter {

   Date dat = new Date();
   private final static String format = "{0,date} {0,time}";
   private MessageFormat formatter;

   private Object args[] = new Object[1];

   private String lineSeparator = ...;
   public synchronized String format(LogRecord record) {
      ...
   }
}
```

### Prevalence

21.3%

## Joiner

An interface which declares no methods, and extends two or more interfaces. A class is called a Joiner if it adds no instance fields nor methods to its superclass and implements at least one interface.

### Definition

```
joiner_inteface := interface {
     no method;
   } extends: {
     many true;
   };

joiner_class := class {
     no method;
     no instance field;
   } implements _;

joiner := joiner_inteface | joiner_class;
```

### Purpose

The Joiner pattern is used for joining together the sets of members of several interfaces. This capability is made possible by the multiple inheritance property of the interface hierarchy

### Example

```
package javax.swing.event;

public interface MouseInputListener
   extends MouseListener, MouseMotionListener {
}
```

### Prevalence

1.2%

## Mould

A class that declares: no instance fields, at least one abstract method, at least one concrete instance method.

### Definition

```
mould := abstract class offers: {
     no instance field;
     abstract method;
   } non_global_members: {
     concrete instance method;
   };
```

### Purpose

This pattern follows the *traits* construct [164] which is a collection of methods, some of which are concrete, but with no underlying state. Actual state is accessible only via the abstract methods that the actual object is required to provide.

### Example

```
package java.lang;

public abstract class Number implements Serializable {

   public abstract int intValue();
   public abstract long longValue();
   public abstract float floatValue();
   public abstract double doubleValue();

   public byte byteValue() {
     return (byte)intValue();
   }

   public short shortValue() {
     return (short)intValue();
   }

   private static final long serialVersionUID = ...;
}
```

### Prevalence

0.7%

## Outline

An abstract class where two or more declared methods invoke at least one abstract method of the current class.

### Definition

```
outline := is T {
     let candidate := concrete method;
     many candidate calls M, M abstract & declared_by T;
  };
```

### Purpose

This pattern is closely related to the TEMPLATE METHOD design pattern. The pattern conservatively requires *two* methods to call the abstract one to rule out cases of trivial overloading (such as *Resending* [83]).

### Example

```
package java.io;

public abstract class Writer
      implements Appendable, Closeable, Flushable {

  private char[] writeBuffer;
  private final int writeBufferSize = 1024;
  protected Object lock;

  public void write(char cbuf[]) throws IOException {
    write(cbuf, 0, cbuf.length);
  }

  public void write(int c) throws IOException {
    synchronized (lock) {
      if (writeBuffer == null)
        writeBuffer = new char[writeBufferSize];
      writeBuffer[0] = (char) c;
      write(writeBuffer, 0, 1);
    }
  }

  abstract public void write(char cbuf[], int off, int len)
     throws IOException;

  // Additional methods and constructors omitted ...
}
```

### Prevalence

0.9%

## Overrider

A class where each of its declared public methods overrides a non-abstract method inherited from its superclass.

### Definition

```
overrider := {
     visible instance method;
     visible instance method implies
       concrete overrides M, M concrete;
   };
```

### Purpose

A Overrider class changes the behavior of its superclass while retaining its protocol.

### Example

```
package java.io;

public class BufferedOutputStream extends FilterOutputStream {
   // Fields and constructors omitted ...

   private void flushBuffer() throws IOException { ... }

   // Reimplementations of FilterOutputStream methods:
   public void write(int b) throws IOException {
      ...
   }

   public void write(byte b[], int off, int len)
        throws IOException {
      ...
   }

   public void flush() throws IOException {
      ...
   }
}
```

### Prevalence

10.8%

## Pool

A type that declares no instance fields nor instance methods, and has at least one visible static final field (other than `serialVersionUID`).

### Definition

```
pool := {
     final static visible field !ser_ver_uid;
     no instance field;
     no instance method;
   };
```

### Purpose

Types following this pattern are typically used for grouping together a set of named constants. This pattern makes it possible to incorporate a namespace of definitions into a class by adding an **`implements`** clause to that class. Prior to the the release of JAVA 5, Pool classes were used for achieving the same effect as **`enum`** classes.

### Example

```
package javax.swing;

public interface SwingConstants {

   public static final int CENTER = 0;

   public static final int TOP = 1;
   public static final int LEFT = 2;
   public static final int BOTTOM = 3;
   public static final int RIGHT = 4;

   public static final int NORTH = 1;
   public static final int NORTH_EAST = 2;
   public static final int EAST = 3;
   // ...
}
```

### Prevalence

2.3%

## Pseudo Class

An abstract type whose list of members (excluding those defined by `Object`) contains no fields and no concrete instance methods.

### Definition

```
pseudo_class := abstract non_global_members: {
     no field;
     no concrete instance method;
   } !interface !annotation;
```

### Purpose

A Pseudo Class can be *mechanically* refactored into an interface, thus it is often considered to be an “anti-pattern”.

### Example

```
package javax.accessibility;

public abstract class AccessibleHyperlink
     implements AccessibleAction {
   public abstract boolean isValid();
   public abstract int getAccessibleActionCount();
   public abstract boolean doAccessibleAction(int i);
   public abstract String getAccessibleActionDescription(int i);
   public abstract Object getAccessibleActionObject(int i);
   public abstract Object getAccessibleActionAnchor(int i);
   public abstract int getStartIndex();
   public abstract int getEndIndex();
}
```

Note that the super-interface `AccessibleAction` declares static final fields thereby preventing the class from qualifying as a Pure Type.

### Prevalence

0.4%

## Pure Type

An abstract type whose list of members (excluding those defined by `Object`) contains only methods which must all be abstract. At least one such method must exist. The only allowed field is `serialVersionUID`.

### Definition

```
pure_type := abstract non_global_members: {
     no concrete method;
     instance method;
     field implies ser_ver_uid;
   };
```

### Purpose

Pure Type prescribes types that posses no implementation details. In particular, any interface which has at least one method, but no static definitions is a Pure Type.

### Example

```
package java.security;

public abstract class SecureRandomSpi
     implements java.io.Serializable {
   private static final long serialVersionUID = ...;
   protected abstract void engineSetSeed(byte[] seed);
   protected abstract void engineNextBytes(byte[] bytes);
   protected abstract byte[] engineGenerateSeed(int numBytes);
}
```

### Prevalence

10.6%

## Record

A class where its list of members (including inherited, excluding those defined by `Object`): (*i*) contains no methods; (*ii*) contains at least one instance field; (*iii*) contains no instance field that are not publicly visible.

### Definition

```
record := concrete class non_global_members: {
     no method;
     no !public instance field;
     public instance field;
   };
```

### Purpose

The Record pattern is often used for modeling parameter objects: objects that are used for delivering state between the methods that take part in realizing a certain algorithm.

### Example

```
package java.sql;

public class DriverPropertyInfo {

   public DriverPropertyInfo(String name, String value) {
     this.name = name;
     this.value = value;
   }

   public String name;
   public String description = null;
   public boolean required = false;
   public String value = null;
   public String[] choices = null;
}
```

### Prevalence

0.8%

## Restricted Creation

A class that declares no public constructors and at least one static field of the same type as the class or a direct supertype thereof.

### Definition

```
restricted_creation := is T {
     no public constructor;
     static field typed S, S is_not Object,
       [T is S | T subtypes S];
  };
```

### Purpose

The SINGLETON pattern is often realized as a Restricted Creation class.

### Example

```
package java.lang;

public class Runtime {
  private static Runtime currentRuntime = new Runtime();

  public static Runtime getRuntime() {
    return currentRuntime;
  }

  private Runtime() {}

  public void exit(int status) {
    SecurityManager security = System.getSecurityManager();
    if (security != null)
         security.checkExit(status);
    Shutdown.exit(status);
  }

  ...
}
```

### Prevalence

1.5%

## Sampler

A class with at least one public constructor, and at least one **`static`** field whose type is the same as that of the class.

### Definition

```
sampler := class is T {
     public constructor;
     static field typed T;
   };
```

### Purpose

These classes allow client code to create new instances, but they also provide several predefined instances.

### Example

```
package java.awt;

public class Color implements Paint, java.io.Serializable {

   public final static Color white = new Color(255, 255, 255);
   public final static Color red = new Color(255, 0, 0);
   public final static Color gray = new Color(128, 128, 128);
   ...

   transient private long pData;
   int value;
   // additional declarations: fields, methods ...

   public Color(int r, int g, int b) {
     this(r, g, b, 255);
   }

   public Color(int r, int g, int b, int a) {
     value = ((a & 0xFF) << 24) | ((r & 0xFF) << 16) |
         ((g & 0xFF) << 8) | ((b & 0xFF) << 0);
     testColorValueRange(r,g,b,a);
   }
}
```

### Prevalence

1.0%

## Sink

A class where the declared methods do not contain method calls.

### Definition

```
sink := class {
     method implies !calls _;
   };
```

### Purpose

Sink classes often realize low-level, implementation space, services. Typically the state of a Sink contains only primitive fields, but this is not mandatory: a Sink may operate on non primitive data via operations such as assignment and identity comparison. Instances of this pattern exhibit a high degree of decoupling as they never depend on method signatures.

### Example

```
package java.util.jar;

public class JarEntry extends ZipEntry {
   Attributes attr;
   Certificate[] certs;
   CodeSigner[] signers;

   public JarEntry(String name) { super(name); }
   // Additional constructors ...

   public Attributes getAttributes() { return attr;}

   public Certificate[] getCertificates() {
     return certs == null ? null
        : (Certificate[]) certs.clone();
   }

   public CodeSigner[] getCodeSigners() {
     return signers == null ? null
       : (CodeSigner[]) signers.clone();
   }
}
```

### Prevalence

13.9%

## State Machine

An interface that declares only parameterless methods.

### Definition

```
state_machine := interface {
     visible instance method implies ();
     exists visible instance method;
   };
```

### Purpose

Such an interface allows client code to either query the state of the object, or, request the object to change its state in some predefined manner.

### Example

```
package java.util;

public interface Iterator<E> {
   boolean hasNext();
   E next();
   void remove();
}
```

### Prevalence

1.8%

## Stateless

A class that has no fields at all (including inherited ones), except for fields which are both static and final.

### Definition

```
stateless := class offers: {
     field implies static final;
   };
```

### Purpose

Stateless captures classes which are a named collection of procedures, and is a representation, in the object-oriented world, of a software library in the procedural programming paradigm.

### Example

```
package java.util;

public class Arrays {

   public static void sort(long[] a) { ... }
   public static void sort(long[] a) { ... }
   // Similar sort() methods ...

   public static int binarySearch(short[] a, short key) { ... }
   public static int binarySearch(char[] a, char key) { ... }
   // Similar binarySearch() methods ...

   ...
}
```

### Prevalence

8.9%

## Taxonomy

An empty interface which extends a single interface. A class is called a Taxonomy if it implements at most one interface and adds no fields nor methods to its superclass. Since constructors are not inherited, a Taxonomy class is allowed to declare constructors.

### Definition

```
taxonomy_interface := [interface|annotation] {
     no field;
     no method;
   } extends: { one true; } non_global_members _;

taxonomy_class := class {
     no field;
     no method;
    } extends S, S is_not Object
    [!implements _ | implements: { one true; }];

taxonomy := taxonomy_interface | taxonomy_class;
```

### Purpose

A Taxonomy type is used, similarly to the Designator micro pattern, for tagging purposes. As the name suggests, a Taxonomy type is included, in the subtyping sense, in its parent, but otherwise is identical to it. This micro pattern is very common in the hierarchy of JAVA's exception classes. The reason is that selection of a **`catch`** clause is determined by the runtime type of the thrown exception, and not by its state.

### Example

```
package java.io;

public class EOFException extends IOException {
   public EOFException() { super(); }
   public EOFException(String s) { super(s); }
}
```

### Prevalence

3.5%

## 3.3 Comparison with Other Patterns

Having presented the catalog we can now discuss the differences between our micro patterns and two other pattern catalogs.

### 3.3.1 Micro Patterns vs. Design Patterns

One of the difficult tasks in software development is bridging the gap which separates the initial *imprecise and informal* system requirement from the *precise and formal* manifestation of software in code written in a specific programming language. But even the smaller steps along the bridge over this gap cannot be all formal, precise or automatic. Clearly, design patterns make one important such step.

There are several obvious relations between design patterns and micro patterns. For example, the Function Object micro pattern, is very useful for implementing the COMMAND design pattern; Sampler is one implementation of the FLYWEIGHT design pattern; most of the classes which realize the SINGLETON pattern will match the Restricted Creation micro pattern, etc.

Still, there are several consequences to the fact that micro patterns stand at a lower level of abstraction:

- *Scope.* First, micro patterns are of a single software module, namely: classes, in a particular programming language. Examining the list of micro patterns in Section 3.2 we can see that they are all about individual types. Design patterns on the other hand are not so tied to a specific language, and often pertain to two or more classes, sometimes to an entire architecture.

- *Recognizability.* Second, a crucial property of micro patterns is that they are easily recognizable by software, which renders a smooth path to automation. Florijn, Meirjer and Winsen [76] enumerate three key issues in automating design patterns: *application*, *validation* and *discovery*. As van Emde Boas argues [178], the expressiveness of the language used for defining patterns, affects the complexity of these issues, and in particular detection.

  In using a formal language, which is at a lower level than the free text description of the semantics of design patterns, automation issues become much easier. Therefore, micro patterns are, by definition, automatically recognizable.

  In contrast, distinct design patterns, presented as solutions to two different problems, may be structured similarly, but be different in their intent (a famous example is made by the STRATEGY and ADAPTER design patterns). It follows that there is an inherent ambiguity in the process of discovering design patterns in software.

- *Context Existence.* Third is the observation that micro patterns do not usually provide "a solution to a problem in a context". The design problem and the context in which it occurs are not present when an implementation is carried out. Indeed, much of the work on the automation of design forgets the problem and the context.

  Micro patterns are not different. For example, the Sink micro pattern, occurring in about a sixth of all classes, is too general to be tied to a specific high-level design problem. Nevertheless, its merits are clear: reduced coupling by avoiding dependency on methods of other classes.

  Another example is the Box micro pattern, which represents a useful programming technique. incidentally, this technique occurs in many and not very related design patterns. The Box is

therefore a term which can be used to describe many classes. Yet, it may serve a multitude of unrelated problems.

Using the semiotic approach [149] to the interpretation of patterns, we have that in formal patterns there is distinction between signifier and signified. A micro pattern is thus "a solution in search of a problem". It serves a concrete purpose, but the programmer is still required to find the right question.

- *Usability of Isolated Patterns.* A fourth difference, resulting from the loss of the problem and context in micro patterns, is the utility of individual patterns. Knowledge of the problem and context makes it possible for a design pattern to provide much more information on the proposed solution. Thus, even a single design pattern is useful on its own. In contrast, micro patterns are not as specific; their power stems from their organization in a catalog, a box of tools, each with its own specific purpose and utility.

  Given an implementation task, the programmer can choose an appropriate pattern from the catalog. Our empirical findings show that, in the majority of cases, such a micro pattern will be found. Admittedly, the nature of micro patterns is such that they do not provide as much guidance as design patterns. On the other hand, the guidance that a micro pattern does provide is suited for automatization, and does not rely as much on abilities of the individual taking that guidance.

- *Empirical Evidence.* Fifth, and perhaps most important is the fact that micro patterns carry massive empirical evidence of their prevalence, their correlation with programming practices, and the amount of information they carry. With the absence of automatic detection tools, claims of the prevalence of design patterns is necessarily limited to the yield of a manual harvest.

### 3.3.2 Micro Patterns vs. Implementation Patterns

Kent Beck presents [29, 32] an extensive discussion of implementation patterns. His books enumerate several scores of patterns, all presented in the context of the SMALLTALK and JAVA languages. These patterns touch software units of different levels: starting at patterns of message send, going through patterns for temporary variables, followed by patterns detailing method implementation, climbing up to instance variables, and ending with single class design.

Beck enumerates several roles that *implementation patterns* serve, including help in reading the code, accelerating the implementation, aid in communication between programmers and documentation.

These roles are not foreign to those of design patterns. Capturing existing lore, and means of communication are essential characteristics of all kinds of patterns.

Yet, implementation patterns come handy at a different stage of the development process. Design patterns are mostly useful at the drawing board. Implementation patterns are most effective when the programmer opens the langauge specific integrated development environment.

However, the fact that implementation patterns show up at a later stage of the development process does not mean that they are always traceable. Consider for example Beck's **Composed Method** implementation pattern. This pattern instructs the SMALLTALK programmer (indeed, a programmer in any language) to continue breaking methods into smaller parts until each method satisfies the (informal) condition that of serving a single identifiable task, and all operation in it stand at the same level of abstraction. It is difficult to fathom a simple formal predicate on the body of a method that will check whether this condition is true.

Another example is implementation pattern **Pluggable Selector** (similar to C's function pointers) which may not be easy to detect.

At the other end stand patterns such as **Query Method**, **Comparing Method**, and **Setting Method**, which are traceable. In our terminology, these are called nano-patterns.

The other important difference distinguishing micro patterns from implementation patterns is that micro patterns can be used at the late design stage as well as during the implementation. While doing class design, micro patterns can be employed to explain the kind of operations expected in inheritance, and for better characterization of the classes. At the implementation stage, the micro patterns prescribed to the class can be used as a guiding recipe, which can even be checked automatically.

## 3.4 Definitions

We denote the prevalence of a pattern $p$ by $\xi(p)$. Let $p_1$ and $p_2$ be patterns. We say that $p_1$ is *contained* in $p_2$ if $p_1 \rightarrow p_2$; they are *mutually exclusive* if $p_1 \rightarrow \neg p_2$, i.e., a module can never match more than one of them. The *co-prevalence* of the patterns (with respect to a software collection) is the prevalence of $p_1 \wedge p_2$; they are *independent* if their co-prevalence is a product of their respective prevalence levels, i.e., $\xi(p_1 \wedge p_2) = \xi(p_1)\xi(p_2)$.

Let $P = \{p_1, \ldots, p_n\}$ be a pattern catalog. Then, the *coverage* of the catalog is the $\xi(p_1 \vee \cdots \vee p_n)$, i.e., prevalence of the disjunction of all patterns in the catalog.

A catalog is more meaningful if the patterns in it are not mutually exclusive. If this is the case, each module can be described by several patterns in the catalog, and the whole catalog can present more information than the simple classification of modules into $n + 1$ categories.

We will now make precise the of amount of information that a catalog carries. First recall the definition of the information theoretical entropy.

**Definition 5** *Let $\xi_1, \ldots, \xi_k$ be a distribution, i.e., for all $i = 1, \ldots, k$ it holds that $0 \leq \xi_i \leq 1$, and $\sum_{1 \leq i \leq k} \xi_i = 1$. Then, the* entropy *of $\xi_1, \ldots, \xi_k$ is*

$$H = H(\xi_1, \ldots, \xi_k) = -\sum_{1 \leq i \leq k} \xi_i \log_2 \xi_i, \tag{3.1}$$

*where the summand $\xi_i \log_2 \xi_i$ is taken to be 0 if $\xi_i = 0, 1$.*

The entropy is maximized when the distribution is to equal parts, i.e., $p_i = \frac{1}{k}$ for all $i = 1, \ldots, k$, in which case $H = \log_2 k$.

To gain a bit of intuition into Definition 5, let us apply it to a single pattern with prevalence $\xi$ (with respect to some software collection). We can say that the pattern occurs with probability $\xi$, and not occur at probability $1 - \xi$, giving rise to the following entropy

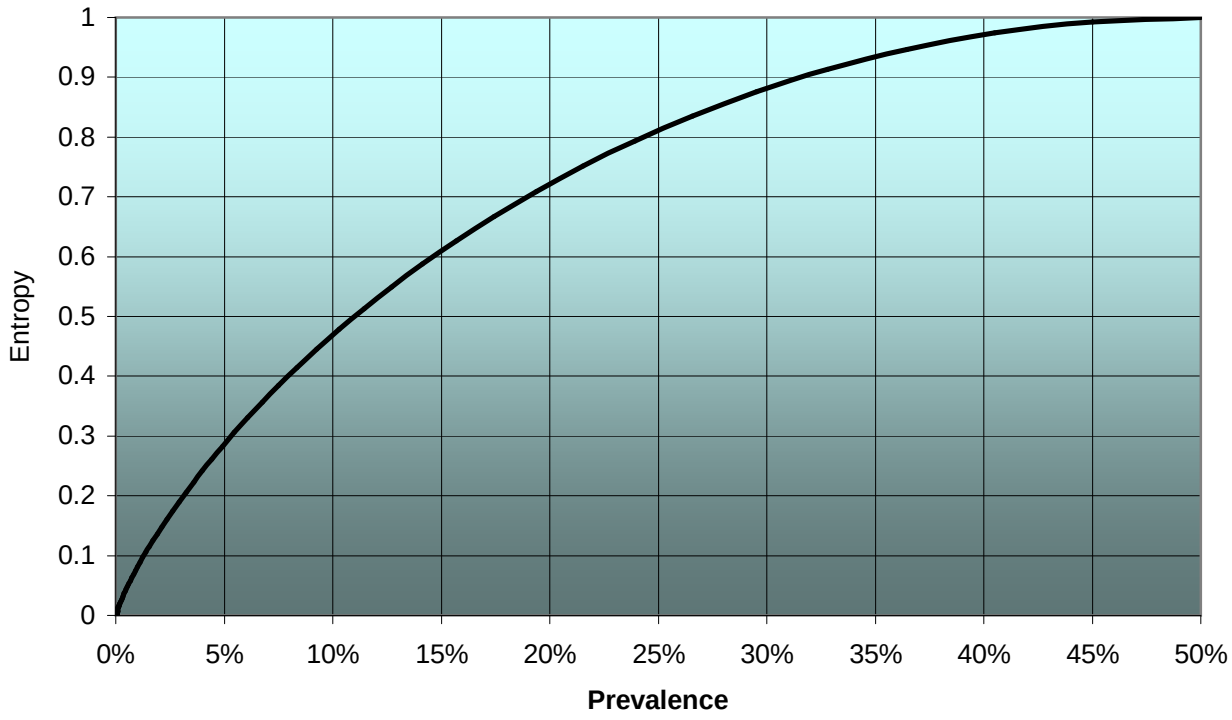
$$-\xi \log_2 \xi - (1 - \xi) \log_2 (1 - \xi).$$

Suppose that $\xi = \frac{1}{2^n}$. Then, the first summand states that the fact that pattern does occur carries $n$ bits of information (the event occurs in only $\frac{1}{2^n}$ of all cases), but these bits have to be weighted with the "probability" of the event. The second summand corresponds to the complement event, i.e., that the pattern does not occur.

Figure 3.2 shows the entropy of a single pattern as a function of the prevalence.

As the figure shows, the entropy achieves its maximal value of 1 when the prevalence is 50% and drops to zero when the prevalence is zero. The entropy is 0.72 if $\xi(p) = 20\%$, drops to 0.47 when $\xi(p) = 10\%$, to 0.29 when $\xi(p) = 5\%$, to 0.08 when $\xi(p) = 1\%$, and to 0.01 when $\xi(p) = 0.1\%$.

**Figure 3.2** Entropy vs. prevalence level of a single pattern.




The entropy of an entire catalog is defined as the entropy of the distribution of the many different combinations of patterns in the catalog

**Definition 6** *The* entropy of a catalog $P$ *(with respect to a certain software collection) is*

$$H(P) = -\sum_{Q \in \wp P} \xi(Q) \log_2 \xi(Q),$$

*where $\wp P$ is the power set of $P$ and $\xi(Q)$ is the prevalence of the event that all patterns in Q occur and all the patterns in $P \setminus Q$* do not *occur, i.e.,*

$$\xi(Q) = \xi\left(\bigwedge_{p \in Q} p \quad \text{and} \quad \bigwedge_{p \in P \setminus Q} \neg p\right).$$

An entropy of (say) 4 of a catalog with respect to a certain software collection can be understood as equivalent to the amount of information obtained by a partitioning of the collection to $16 = 2^4$ equal parts. We can think of $2^{H(P)}$ as the *separation power of the catalog.*

Information theory tells us that entropy is an *additive* property in the sense that the entropy of a catalog of independent patterns is the *sum* of the entropies of each of these events. If patterns in a catalog are mutually exclusive, then the entropy of the catalog is *less* than the sum of the individual entropies.

As mentioned before, the patterns in our catalog are not mutually exclusive, which makes the catalog more informative. On the other hand, we do not expect software patterns to be truly independent. In order to evaluate the contribution of each pattern to the expressive power of the catalog, we can examine its marginal contribution to the entropy of the catalog.

Table 3.2: The JAVA class collections comprising the corpus.

| Collection | Domain | Packages | Classes | Methods |
|---|---|---|---|---|
| Kaffe$^{1.1}$ | JRE impl. | 75 | 1,220 | 10,945 |
| Kaffe$^{1.1.4}$ | JRE impl. | 152 | 2,511 | 22,022 |
| Sun$^{1.1}$ | JRE impl. | 67 | 991 | 9,448 |
| Sun$^{1.2}$ | JRE impl. | 131 | 4,336 | 36,661 |
| Sun$^{1.3}$ | JRE impl. | 170 | 5,213 | 44,747 |
| Sun$^{1.4.1}$ | JRE impl. | 314 | 8,216 | 73,834 |
| Sun$^{1.4.2}$ | JRE impl. | 330 | 8,740 | 76,675 |
| Scala | Lang. tools | 96 | 3,382 | 32,008 |
| MJC | Lang. tools | 41 | 1,141 | 10,927 |
| Ant | Lang. tools | 120 | 1,970 | 17,902 |
| JEdit | GUI | 23 | 805 | 6,110 |
| Tomcat | Server | 280 | 4,335 | 43,868 |
| Poseidon | GUI | 594 | 10,052 | 77,988 |
| JBoss | Server | 998 | 18,699 | 157,460 |
| Total | | 3,391 | 71,611 | 620,595 |

**Definition 7** *The* marginal entropy *of pattern $p \in P$ with respect to a catalog $P$, written $H(p/P)$ is*

$$H(P) - H(P \setminus \{p\}).$$

The sum of the marginal entropies can be greater, smaller or equal to the entropy of the whole catalog.

If a pattern is identical to one of the other patterns in the catalog, or to any combination of these, then its marginal entropy is 0. Conversely, suppose that a certain pattern partitions every combination of the other patterns in the catalog into two equal parts. Then, the marginal entropy of this pattern is 1.

## 3.5 Data set

In the experiments, we measured the prevalence level of each of the patterns in the catalog in large collections of JAVA classes, available in the `.class` binary format. As explained in Section 3.2 the analysis was carried out by invoking a set of predicates over all classes in the collection.

A corpus of fourteen large collections of JAVA classes, totaling over three thousand packages, seventy thousand classes and half a million methods served as data set for our experiments. Table 3.2 summarizes some of the essential size parameters of these collections. The table does not include a line count of the collections in the corpus, since many of the collections are available in binary format only.

As can be seen in the table, the collections, although all large, vary in size. The smallest collection (JEdit) has about 800 classes and 6,000 methods, while the largest (JBoss) has almost a thousand packages, 18,699 classes and 157,460 methods. The median number of classes is about 4,000.

These collections can be partitioned into several groups

1. *Implementations of the standard* JAVA *runtime environment.* The JAVA runtime environment (JRE) is the language standard library, as implemented by the language vendor, which pro-

vides to the JAVA programmer essential runtime services such as text manipulation, input and output, reflection, data structure management, etc.

We included several different implementations of the JRE in our corpus for two purposes. First, to examine the stability of patterns in the course of evolution of a library, we used SUN's standard implementations of versions 1.1, 1.2, 1.3, 1.4.1, and 1.4.2 of the JRE specification. These are denoted respectively by $\text{Sun}^{1.1}$, $\text{Sun}^{1.2}$, $\text{Sun}^{1.3}$, $\text{Sun}^{1.4.1}$ and $\text{Sun}^{1.4.2}$ in Table 3.2. We also used two other JRE implementations, $\text{Kaffe}^{1.1}$ and $\text{Kaffe}^{1.1.4}$, included in the Kaffe project[4]: a non-commercial JVM implementation.

We had also tried to expand our corpus with three commercial JRE libraries supplied with theses JVM products: (*i*) *IBM 32-bit Runtime Environment for* JAVA *2*, version 1.4.2; (*ii*) *J2SE for HP integrity*, version 1.4.2; and (*iii*) *Weblogic JRockit 1.4.2* by BEA. Eventually, these three collections were *not* included in the corpus since they all exhibited an overwhelming similarity with $\text{Sun}^{1.4.2}$. Our experiments indicated that these three were in many ways a port of the Sun implementation. Obviously, no significant data can be drawn from the analysis of these.

2. *GUI Applications.* The corpus includes two GUI applications: JEdit—which is version 4.2 of the programmer's text editor written in JAVA with a Swing GUI, and Poseidon–a popular UML modeling tool delivered by Gentleware[5]. (We used version 2.5.1 of the community edition of the product.)

3. *Server Applications.* There were two collections in this category: JBoss—the largest collection in our corpus is version 3.2.6 of the famous JBoss[6] application server (JBOSS AS) which is an open source implementation of the J2EE standard, Tomcat—part of the *Apache Jakarta Project*[7], which is a servlet container used by http servers to allow JAVA code to create dynamic web pages (version 5.0.28).

4. *Compilers and Langauge Tools*. This category includes Ant—another component of the Apache project[8]–a build tool which offers functionality that is similar, in principle, to the popular make utility (version 1.6.2), and Scala—version 1.3.0.4 of the implementation of the SCALA multi-paradigm programming language; and, MJC—version 1.3 of the compiler of MULTIJAVA, a language extension which adds open classes and symmetric multiple dispatch to the language.

Thus, the corpus represents a variety of software origins (academia, open source communities and several independent commercial companies), interaction modes (GUI, command line, servers, and libraries), and application domains (databases, languages, text processing).

Note that the totals in the last line of Table 3.2 include multiple and probably not entirely independent implementations of the same classes. For experiments and calculations which required independence of the implementation, we used only collection $\text{Sun}^{1.4.2}$ out of all JRE implementations. Also, as many as 5,979 classes recurred in several collections since software manufacturers tend to package external libraries in their binary distribution.

To assure independence, all such classes were pruned out of their respective collections and included in a pseudo-collection named Shared. (Interestingly, the 100 or so classes comprising the famous JUnit [30] library, were found in several collections in our corpus, thus turning Shared

[4]http://www.kaffe.org
[5]http://www.gentleware.com
[6]http://www.jboss.org
[7]http://jakarta.apache.org
[8]http://ant.apache.org

Table 3.3: The JAVA class collections in the pruned corpus.

| Collection | Packages | Classes | Methods |
|---|---|---|---|
| Sun$^{1.4.2}$ | 272 | 7,525 | 66,676 |
| Scala | 68 | 2,678 | 25,186 |
| MJC | 32 | 945 | 8,607 |
| Ant | 45 | 421 | 3,883 |
| JEdit | 21 | 676 | 4,653 |
| Tomcat | 132 | 1,434 | 14,367 |
| Poseidon | 477 | 8,162 | 61,645 |
| JBoss | 750 | 13,623 | 110,820 |
| Shared | 346 | 5,979 | 55,431 |
| Total | 2,143 | 41,443 | 351,268 |

into a super set of the Junit library.) This process defined a **Pruned** software corpus by

$$\textbf{Pruned} \quad = \quad \{\mathsf{Sun}^{1.4.2}, \mathsf{Scala}, \mathsf{MJC}, \mathsf{Ant}, \mathsf{JEdit}, \mathsf{Tomcat}, \mathsf{Poseidon}, \mathsf{JBoss}, \mathsf{Shared}\}$$

The total size of this corpus and each of the (pruned) collections in it is reported in Table 3.3.

We can see that the elimination of duplicates and dependent implementations halved the size of the corpus. In total, more than 41,000 independent class comprise the pruned corpus.

## 3.6 Experimental Results

The experimental results of running the pattern analyzer on the pruned corpus are summarized in Table 3.4.

The table shows the prevalence, coverage, entropy and marginal entropy of the patterns in the corpus. The body of the table presents, for each micro pattern and each software collection, the *prevalence* of the micro pattern in the collection, that is, the percentage of classes in this collection which match this micro pattern. The two last rows give a summary of each collection. (Note that due to overlap between the patterns, columns do not add up to the total coverage in the last row.) The seven last columns give summarizing statistics on each of the patterns.

In this section we take mostly a broad perspective in the inspection of this information, and will be interested in the more global properties of the catalog, including coverage, entropy, and marginal entropy. In the next section, we will march on to a deeper *statistical analysis* of this information.

**Coverage.** The most important information that this table brings is in the penultimate line, which shows the coverage of our catalog. We see that $79.5\%$ of all classes in Sun$^{1.4.2}$ are cataloged. The collection with least coverage is Ant, but even for it, one in two classes is cataloged. The total coverage of the (pruned) corpus is 74.9%. The fluctuation in coverage level is not very great—the standard deviation (penultimate column) is 11%.

**Conclusion 1** *Three out of four classes match at least one micro pattern in the catalog*[9].

**Prevalence.** Examining Table 3.4 in greater detail, we see that the most prevalent group is this of the inheritors micro patterns. About 35% of all classes not only inherit from a parent, they also

[9]The above, just as all subsequent conclusions, refer to what can be observed in our corpus.

| Collection | $Sun_{1.4.2}$ | Scala | MJC | Ant | JEdit | Tomcat | Poseidon | JBoss | Shared | Total | Average | Median | Min | Max | $\sigma$ | $H(p/P)$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Designator | 0.2% | 0.1% | 0.2% | 0.0% | 0.0% | 0.2% | 0.1% | 0.3% | 0.3% | 0.2% | 0.2% | 0.2% | 0.0% | 0.3% | 0.1% | 0.05 |
| Taxonomy | 4.4% | 2.7% | 3.2% | 1.4% | 1.2% | 2.6% | 3.8% | 3.2% | 3.5% | 3.5% | 2.9% | 3.2% | 1.2% | 4.4% | 1.1% | 0.13 |
| Joiner | 0.7% | 1.8% | 0.0% | 0.0% | 0.0% | 0.6% | 0.3% | 2.2% | 0.9% | 1.2% | 0.7% | 0.6% | 0.0% | 2.2% | 0.8% | 0.09 |
| Pool | 1.9% | 1.0% | 4.6% | 1.7% | 1.0% | 1.5% | 1.7% | 2.9% | 2.7% | 2.3% | 2.1% | 1.7% | 1.0% | 4.6% | 1.1% | 0.15 |
| Sink | 20.6% | 14.0% | 10.7% | 14.3% | 9.0% | 12.1% | 11.3% | 12.7% | 13.5% | 13.9% | 13.1% | 12.7% | 9.0% | 20.6% | 3.3% | 0.67 |
| Record | 0.4% | 0.3% | 0.2% | 0.2% | 0.6% | 0.3% | 0.4% | 1.1% | 1.5% | 0.8% | 0.6% | 0.4% | 0.2% | 1.5% | 0.5% | 0.08 |
| Data Manager | 1.8% | 0.2% | 1.2% | 4.0% | 1.5% | 1.7% | 1.9% | 1.8% | 2.4% | 1.8% | 1.8% | 1.8% | 0.2% | 4.0% | 1.0% | 0.04 |
| Function Pointer | 2.0% | 0.9% | 1.8% | 1.2% | 1.2% | 2.8% | 1.7% | 1.7% | 1.0% | 1.6% | 1.6% | 1.7% | 0.9% | 2.8% | 0.6% | 0.11 |
| Function Object | 7.7% | 0.8% | 9.1% | 1.4% | 24.1% | 2.4% | 6.3% | 4.2% | 5.2% | 5.5% | 6.8% | 5.2% | 0.8% | 24.1% | 7.1% | 0.23 |
| Cobol Like | 0.4% | 0.6% | 0.5% | 0.7% | 0.1% | 1.0% | 0.5% | 0.7% | 0.4% | 0.5% | 0.5% | 0.5% | 0.1% | 1.0% | 0.2% | 0.07 |
| Stateless | 9.8% | 14.6% | 7.6% | 5.7% | 6.1% | 10.3% | 6.8% | 9.6% | 6.8% | 8.9% | 8.6% | 7.6% | 5.7% | 14.6% | 2.8% | 0.38 |
| CommonState | 2.4% | 0.3% | 2.1% | 0.2% | 3.4% | 1.3% | 1.8% | 7.1% | 3.6% | 3.8% | 2.5% | 2.1% | 0.2% | 7.1% | 2.1% | 0.14 |
| Canopy | 9.8% | 3.9% | 11.0% | 4.5% | 26.5% | 4.6% | 10.3% | 6.3% | 4.5% | 7.7% | 9.0% | 6.3% | 3.9% | 26.5% | 7.1% | 0.28 |
| Immutable | 7.6% | 5.6% | 7.0% | 2.1% | 12.0% | 4.0% | 6.2% | 6.1% | 4.6% | 6.1% | 6.1% | 6.1% | 2.1% | 12.0% | 2.7% | 0.28 |
| Box | 4.6% | 14.5% | 3.3% | 3.1% | 1.3% | 8.6% | 2.5% | 7.8% | 5.1% | 6.0% | 5.6% | 4.6% | 1.3% | 14.5% | 4.1% | 0.22 |
| Compound Box | 6.0% | 5.1% | 3.6% | 10.0% | 5.8% | 3.1% | 3.8% | 3.7% | 4.4% | 4.4% | 5.0% | 4.4% | 3.1% | 10.0% | 2.1% | 0.24 |
| Implementor | 26.0% | 10.5% | 17.8% | 17.1% | 37.1% | 12.7% | 22.1% | 23.1% | 15.8% | 21.3% | 20.2% | 17.8% | 10.5% | 37.1% | 8.1% | 0.63 |
| Overrider | 12.4% | 4.1% | 8.1% | 4.0% | 23.1% | 20.2% | 16.8% | 7.0% | 9.4% | 10.8% | 11.7% | 9.4% | 4.0% | 23.1% | 6.9% | 0.23 |
| Extender | 4.3% | 1.6% | 5.3% | 4.8% | 4.9% | 5.9% | 4.5% | 4.2% | 4.2% | 4.2% | 4.4% | 4.5% | 1.6% | 5.9% | 1.2% | 0.23 |
| Outline | 1.8% | 0.2% | 1.1% | 1.0% | 0.4% | 0.3% | 1.3% | 0.6% | 0.6% | 0.9% | 0.8% | 0.6% | 0.2% | 1.8% | 0.5% | 0.09 |
| Mould | 1.3% | 0.3% | 0.8% | 0.2% | 0.0% | 0.7% | 0.8% | 0.4% | 0.6% | 0.7% | 0.6% | 0.6% | 0.0% | 1.3% | 0.4% | 0.08 |
| State Machine | 1.5% | 1.8% | 1.0% | 0.7% | 0.3% | 1.7% | 1.7% | 2.1% | 1.8% | 1.8% | 1.4% | 1.7% | 0.3% | 2.1% | 0.6% | 0.09 |
| Pure Type | 7.7% | 20.5% | 6.7% | 3.1% | 2.5% | 5.6% | 11.9% | 11.2% | 10.1% | 10.6% | 8.8% | 7.7% | 2.5% | 20.5% | 5.5% | 0.15 |
| Augmented Type | 0.6% | 0.0% | 0.3% | 0.5% | 0.0% | 0.1% | 0.2% | 0.4% | 1.0% | 0.5% | 0.4% | 0.3% | 0.0% | 1.0% | 0.3% | 0.06 |
| Pseudo Class | 0.7% | 1.6% | 0.3% | 0.0% | 0.0% | 0.3% | 0.3% | 0.2% | 0.4% | 0.4% | 0.4% | 0.3% | 0.0% | 1.6% | 0.5% | 0.06 |
| Sampler | 1.2% | 3.5% | 1.0% | 0.0% | 0.6% | 1.7% | 1.0% | 0.5% | 1.0% | 1.0% | 1.2% | 1.0% | 0.0% | 3.5% | 1.0% | 0.10 |
| Restricted Creation | 2.3% | 0.5% | 1.0% | 0.0% | 0.4% | 1.3% | 1.5% | 1.7% | 0.7% | 1.5% | 1.0% | 1.0% | 0.0% | 2.3% | 0.7% | 0.14 |
| Coverage | 79.5% | 79.4% | 64.3% | 48.0% | 83.7% | 67.3% | 76.9% | 76.2% | 65.7% | 74.9% | 71.2% | 76.2% | 48.0% | 83.7% | 11.1% | |
| Entropy | 5.27 | 4.32 | 4.27 | 3.32 | 4.51 | 4.22 | 4.74 | 4.96 | 4.83 | 5.08 | 4.50 | 4.51 | 3.32 | 5.27 | 0.56 | |

Table 3.4: The prevalence, coverage, entropy and marginal entropy of micro patterns in the collections of the pruned corpus.

adhere to a specific, particularly restrictive style of inheritance. The most common micro pattern, in this category and overall, is Implementor which occurs in about 21% of all classes. This finding indicates wide spread use of the technique of separating type and implementation, by placing the implementation in a concrete class.

Also large is Overrider, which occurs in about 11% of all classes.

A large group is also that of classes with degenerate state, whose total prevalence is about 24%.

**Conclusion 2** *One in four classes is degenerate in respect to the data it maintains.*

In this group, the largest pattern is Stateless (8.9% prevalence), which is unique in that it has no instance fields.

The base class category is also quite significant, occupying about 15% of all classes. The largest pattern there is Pure Type with 10.6% prevalence.

It is interesting to see that the Sink, a class which essentially does not communicate with any other class, is also very frequent, with prevalence of 13.9%.

Together, the five leading patterns (Implementor, Sink, Overrider, Pure Type and Stateless) describe 23,848 classes, which are 58% of the classes in our pruned corpus.

**Conclusion 3** *The majority of classes are cataloged by one of the five leading patterns.*

**Separation Power.** Conclusion 3 does not mean that we can make do with only five patterns. The other patterns in the catalog contribute to the information it provides. One of the reasons is that the micro patterns are not mutually exclusive. There are classes in the corpus which match more than one micro pattern. Figure 3.3 depicts the number of classes in the pruned corpus for each multiplicity level.

We see that 31% of the classes matched a single pattern, 30% matched two patterns, 13% matched three patterns. Out of the total 41,443 classes in this corpus there was also a significant number of classes which matched more than three patterns: 558 classes with four patterns, 89 with five patterns. There were even 18 classes which matched six patterns!

It is interesting to examine some of the classes with multiple classifications. There were 12 classes which matched the same six patterns: Canopy, Restricted Creation, Overrider, Sink, Function Object, and Data Manager. All of these classes are exceedingly similar: they all have a series of pre made instances represented as **`public static`** fields, a **`private`** constructor which accepts the name of the created instance (passed as a `String`), a **`private`** variable to store that name, and a `toString()` method that returns the name of that instance. Here is one of these for example.

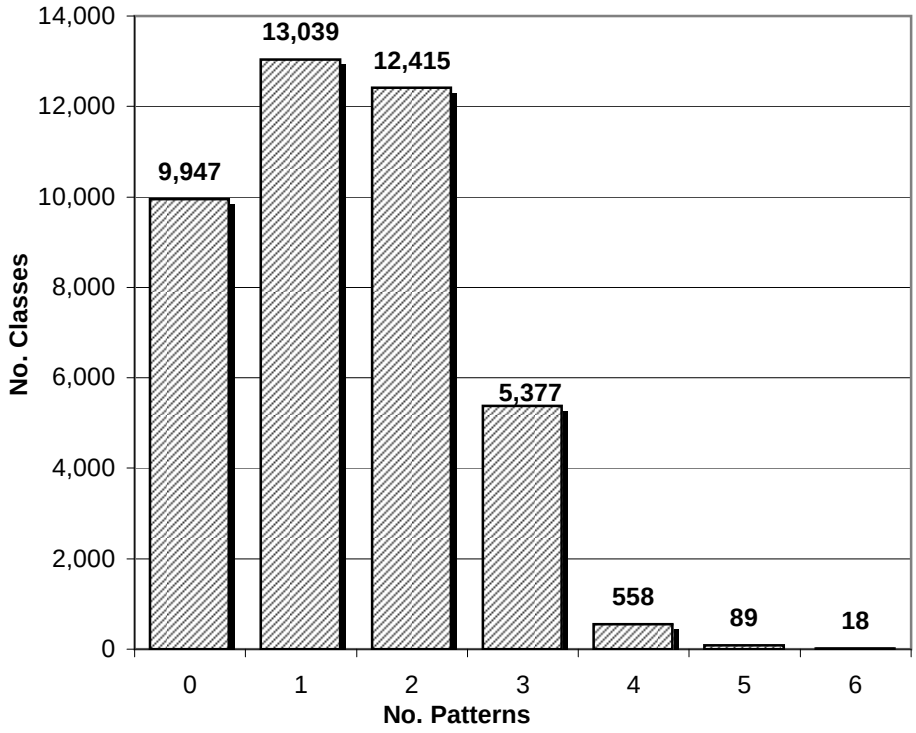
```
package javax.xml.rpc;
public class ParameterMode {
   public static final ParameterMode
       IN = new ParameterMode("IN"),
       INOUT = new ParameterMode("INOUT"),
       OUT = new ParameterMode("OUT");

   private String mode;
   private ParameterMode(String mode) { this.mode = mode; }
   public String toString() { return mode; }
}
```

Interestingly, there is nothing else in all these classes (except for an array of the samples, in the case that the number of samples is large.)

There are 30 classes which match Joiner, Pure Type, Stateless and Sink but no other patterns. Here is one of them for example.

**Figure 3.3** Multiplicity of pattern classification in the classes of the pruned corpus.




```
package org.freehep.swing.graphics;

public abstract class AbstractPanelArtist
   implements PanelArtist, GraphicalSelectionListener {

   public AbstractPanelArtist() { }
}
```

Again, all 30 classes are very similar in structure. They *join* together several empty classes or interfaces, thus helping in enriching the classification hierarchy. In the process, they add a single empty method.

The examples we checked indicated that a multiple patterns match is very precise, yet very narrow. We can think of each pattern combination as a *new* pattern which is more focused than any of its components.

We analyzed the classes where multiple patterns were detected, and found out that there are more than 600 different combinations of multiple patterns (when a combination is a set of patterns detected in a single class). While this number merely provides some vague intuition to the power of the catalog, the entropy measurement can formally describe the catalog's separation power, or: the amount of information that the catalog provides on average. Examining the last row of Table 3.4, we learn that the entropy fluctuates between 4.28 and 5.27. By raising these values to the power of two, we obtain,

**Conclusion 4** *The separation power of the catalog is equivalent to that of a partitioning into 19–39 equal and disjoint sets.*

**Marginal Entropy.** The last column of Table 3.4 gives the marginal entropy of each of

the micro patterns with respect to the entire catalog and all classes in the pruned corpus. In other words, this column specifies the "additional separation power" or added information, when classes which were already matched by the rest of the catalog are matched against this micro pattern.

We see that the marginal entropy of none of the patterns is 0. Therefore, we can state:

**Conclusion 5** *All patterns contribute to the separation power of the catalog.*

In examining the last column we also find that patterns with high prevalence usually exhibit higher marginal entropy, and vice versa, patterns with low prevalence tend to have low marginal entropy. Maximal marginal entropy is achieved by Sink; Implementor follows. In other words, we may argue that Sink contributes the most to the separation power of the catalog. This is in spite of the fact that there are patterns with higher prevalence. In a sense, Sink is more "independent" of the rest of the catalog than other patterns.

The sum of marginal entropies is 5.02, while the entropy of the entire catalog stands at a slightly higher, 5.06. This finding is the basis of our claim that the information brought by the catalog is greater than the sum of its parts.

**Variety in Prevalence.** Following the table body, there are six columns that give various statistics on the distribution of prevalence of each pattern in the different collections. The first of these columns gives the prevalence of each pattern in the entire (pruned) corpus, i.e., a weighted average of the preceding columns. The two following columns give the (straight) average and median prevalence. Note that in the majority of micro patterns, these three typical values are close to each other.

The next three columns are indicative of the variety in prevalence, giving its minimal and maximal values, as well as the standard deviation. Examining these, we can make the following qualitative conclusion:

**Conclusion 6** *There is a large variety in the prevalence of patterns in different collections.*

For example, Function Object occurs in 24.1% of all classes in JEdit (probably since it is used to realize the COMMAND pattern in this graphic environment), but only in 0.8% of the classes in the Scala compiler. On the other hand, 20.5% of Scala classes match the Pure Type pattern, while the prevalence of this pattern in JEdit is only 2.5%. In JEdit, 37.1% of all classes are instances of Implementor, while only 10.5% of Scala classes match Implementor.

## 3.7 Prevalence Differences and Purposefulness

The previous section ended with Conclusion 6 making the qualitative statement that differences between prevalence levels are "large". In this section, we will make this statement more precise by showing that these differences are *statistically significant*. Concretely, we prove that random fluctuations of prevalence are improbable to generate differences of this magnitude. Thus, we will infer that there exists a non-random mechanism which governs the extent by which patterns are used in different collection.

The statistical validation of Conclusion 6 can be taken as supporting evidence to our claim that *the patterns in the catalog are purposeful*. One such purpose could be that different software collections serve different needs, and therefore employ different patterns at different levels. Yet another explanation of this non-random process is the difference in programming style and practice between different vendors and their various software teams. We shall discuss these possible explanations in greater detail in the following section.

**Statistical Inference.** The statistical inference starts by making a *null hypothesis* $\mathcal{H}_0$, by which *patterns are a random property of* JAVA *code*. According to this hypothesis, each pattern

has some fixed (yet unknown) probability of occurring in the code, regardless of context or programming style. The number of occurrences of a certain pattern in a collection of $n$ classes is therefore the sum of the $n$ independent random binary variables, one for each class in the collection. The binary variable of a class is 1 precisely when the pattern occurs in that class. If the null hypothesis is true, then changes in prevalence of the pattern across different collections are due to normal fluctuations of the $n$-sum.

Our objective here is to *reject* the null hypothesis. As usual in statistical inference, we assume $\mathcal{H}_0$ and check the probability that such changes occur under this assumption. More specifically, let $\mathcal{H}_0(p)$ denote the null hypothesis for a pattern $p$. For each pattern $p$, we examine the values found in the corresponding row of Table 3.4, i.e, the prevalence level of this pattern in the different collections, and check whether the variety in these can be explained by $\mathcal{H}_0(p)$.

For example, the prevalence of the rarest pattern, Designator, is distributed as follows: $0.0\%$ in two of the collections, $0.1\%$ in two collections, $0.2\%$ in three collections, and $0.3\%$ in the three remaining collections. Are these rather tiny differences which occur in such minuscule prevalence values, meaningful at all?

A precise answer to this question is given by the application of the standard $\chi^2$-test[10] to this row. This test checks whether random fluctuations in prevalence values can give rise, with reasonable probability, to these differences.

Perhaps surprisingly, the test shows that null hypothesis is rejected with confidence level of more than $99\%$, i.e., $\alpha < 0.01$. (More precisely, the confidence level with Designator is $99.75\%$.) In other words, the probability that the changes in the prevalence level of Designator can be explained by $\mathcal{H}_0(p)$ is less than $0.01$.

In applying the test to each of the patterns we find that hypothesis $\mathcal{H}_0(p)$ is rejected with confidence level of $99\%$, i.e., $\alpha < 0.01$ for all the patterns in the collection, with only one exception: Cobol Like, for which the confidence level is 96%.

**Conclusion 7** *With the exception of Cobol Like, changes in prevalence of each of the micro patterns in the collections of the pruned corpus are significant.*

**Pair-wise Separation.** The above conclusion does not provide means of *understanding* the nature of the changes. It merely says that these changes as a whole are (statistically) significant. Furthermore, the rejection of $\mathcal{H}_0(p)$ does not mean that *every change in prevalence level of each pattern in any two collections* is significant.

Conclusion 7 only says that *not all* changes in the collections are a matter of coincidence. Despite the great variety, some patterns exhibit the same prevalence level in different collections. For example, the prevalence of State Machine in Tomcat and Poseidon is almost the same (round $1.7\%$); its (rounded) prevalence in Scala and Shared is 1.8%. Is each of these differences significant?

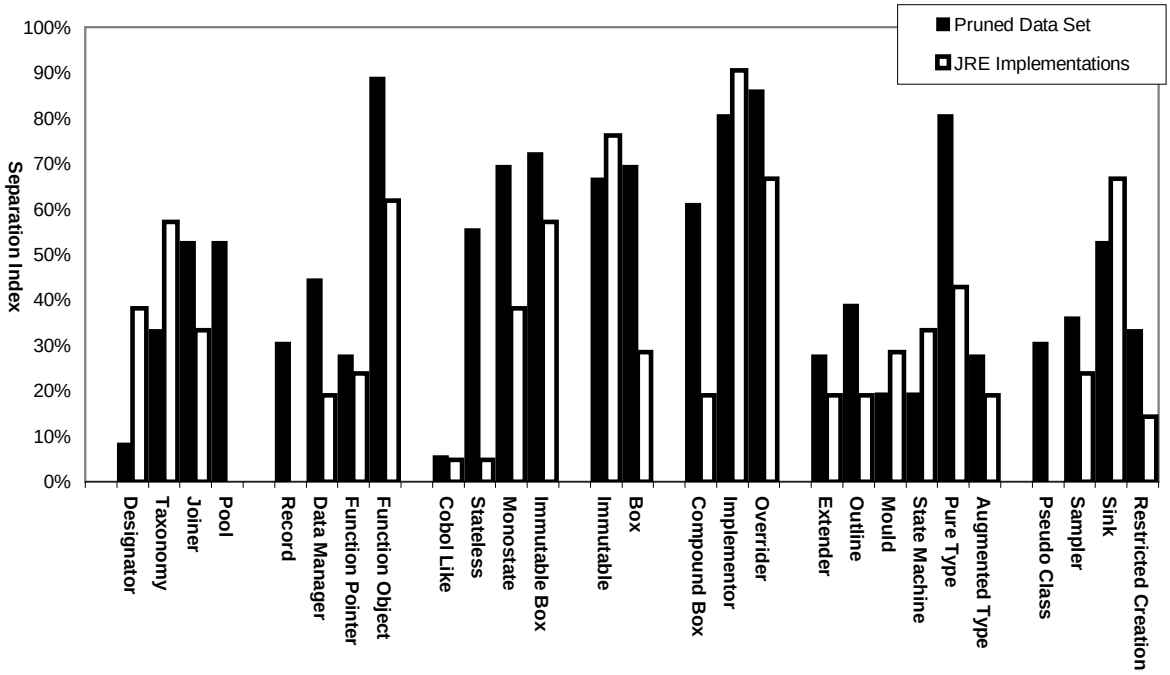
Let $\mathcal{H}_0[c_1, c_2](p)$ be the null hypothesis that the prevalence of a pattern $p$ in collections $c_1$ and $c_2$ is the same. To check this hypothesis, we apply a $\chi^2$-test to determine whether the difference in proportions in the two collections is significant. The result is that hypothesis $\mathcal{H}_0[\text{Tomcat}, \text{Poseidon}](\text{State Machine})$ cannot be rejected by the test. The test similarly fails to reject the hypothesis $\mathcal{H}_0[\text{Scala}, \text{Shared}](\text{State Machine})$.

**Definition 8** *A pattern $p$* separates *the collections $c_1$ and $c_2$ if $\mathcal{H}_0[c_1, c_2](p)$ is rejected.*

**Corpus Separation Index.** Let us now define a metric of the average extent by which a pattern distinguishes between different collections.

[10] Read "Chi-squared-test".

**Figure 3.4** The separation index of the patterns with respect to the pruned corpus and the different implementations of the JRE ($\alpha < 0.01$).




**Definition 9** *Assume some fixed confidence level. Let $\mathcal{C}$ be a software corpus. Let $p$ be a pattern. Then, the notation $\Upsilon(p, \mathcal{C})$ (or for short just $\Upsilon(p)$ when $\mathcal{C}$ is clear from context), stands for the* separation index of $p$ *(with respect to $\mathcal{C}$) is the fraction of rejected null hypotheses $\mathcal{H}_0[c_1, c_2](p)$ (at the fixed confidence level) out of all such hypotheses where $c_1$ and $c_2$ vary over $\mathcal{C}$, $c_1 \neq c_2$.*

The separation index becomes useful because the $\chi^2$-test is sensitive to outliers: Suppose that the prevalence of a pattern $p$ in a single collection $c \in \mathcal{C}$ is distant from the average prevalence, while the prevalence in the other collections in $\mathcal{C}$ is very close to the average prevalence. Then, the test will reject the null hypothesis $\mathcal{H}_0(p)$. In contrast, $\mathcal{H}_0[c_1, c_2](p)$ will be rejected only if $c_1 = c$ or $c_2 = c$.

Low separation index of a pattern indicates that the pattern prevalence is more stable in the different collections.

Figure 3.4 shows the separation index of the patterns in the catalog with respect to the pruned corpus (black columns) and the JRE corpus (white columns), which is defined below (Section 3.8).

Not surprisingly, the minimal value is that of Cobol Like, which separates only 2 out of the 36 pairs of the pruned corpus **Pruned**

$$\Upsilon(\text{Cobol Like}, \textbf{Pruned}) = 5.6\%.$$

It is followed by $\Upsilon(\text{Designator}) = 8.33\%$. The highest separation index, $89\%$, is achieved by Function Object, where the second highest value is $\Upsilon(\text{Overrider}) = 86\%$. The median is 44%, while the average separation index is $47\%$.

**Conclusion 8** *The difference in prevalence levels between two collections is significant in one out of two cases.*

Together Conclusion 7 and Conclusion 8 are the statistically sound counterpart of the qualitative statement in Conclusion 6.

**Discussion.** The above arguments established the significance of the variety in prevalence level of the same pattern in different collections. In this section, we shall discuss several ways of interpreting this significance.

We first note that the corpus size makes it possible to establish the significance of even relatively small differences in prevalence levels. But, the nature of the statistical tests we employ is that they take an appropriate account of both the corpus size but also of the expected prevalence. Even with large data sets, not every difference in prevalence level is significant:

- Consider the difference in prevalence level of an individual pattern between two specific collections. Figure 3.4 shows that in only about half of the cases there was significance to the difference.

- As mentioned in Section 3.5 we tried to expand the corpus with three other ports of Sun's implementation of the JRE, due to IBM, HP, and BEA. As it turned out, the differences in prevalence level of these ports were not statistically significant.

We can think of at least five different phenomena which can explain, alone or together, the findings of Section 3.7

1. *Requirement Variety.* The different collections serve different needs, which call for different patterns.

2. *Style Variety.* The different collections are implemented by different vendors employing different programming policies, styles, and individuals, all reflected by patterns prevalence.

3. *Replication.* We know that programmers tend, or are at least encouraged, to reuse both design and code. Programmers may copy classes, changing only a few lines of codes, instead of factoring out similarities[11]. If this happens often, the number of "independent" classes in a collection is smaller than the actual number of classes. A random fluctuation in a pattern prevalence is amplified by this replication, and interpreted to be significant even if it is not so.

4. *Population Contamination* Our experiments cannot tell a difference in prevalence level is a result of a moderate global change to the entire population, or of an accentuated change to a subpopulation.

   To understand this better, let us assume that CommonState occurs naturally in JAVA code with probability of 7%. Then, if we take a set of $10,000$ classes, about $700$ of these will be a CommonState. Finding instead $767$ classes, is, so tells us the $\chi^2$-test, statistically significant. What the test fails to say is whether the increase in the number of occurrences is a result an increased global tendency to use CommonState, or of (say) having a sup-population of 350 classes, whose specific domain is such that the prevalence of CommonState is $26\%$.

5. *Dormant Abstraction.* It could be the case that the micro-patterns found here are a reflection of some *deep* patterns which are still not known to us. The difference in prevalence of micro patterns could be a reflection of difference in prevalence of the "deep patterns", which capture the "true" differences between collections.

We would like to attribute changes in the use of patterns primarily to requirement variety, and only then to style variety. But, these changes could be a result of code replication, or population contamination. These two explanations represent in fact the two faces of the same phenomena,

[11] In some cases, code duplication cannot be avoided due to the absence of advanced abstraction mechanisms, (such as multiple inheritance, mixins, anonymous functions, and traits) in the language.

i.e., that different classes are not independent of each other. Finally, dormant abstraction may mean that we are examining the wrong patterns.

In an initial experimentation with a bunch of "pseudo-patterns", which are not expected to carry any purpose, we made some interesting discoveries.

- Pseudo-patterns computed by hashing the class pool into a single bit showed, at times, significance, although not as strong as we found for micro patterns. This finding indicates that the extent of code replication in the corpus is small, but probably measurable.
- The statistical tests can trace in the corpus more than design information. For example, the use of a code obfuscator in parts of Poseidon, generated short named classes, which made significant changes to the prevalence of a "non-sense" pattern occurring whenever the length of the class name is a prime number. The dormant abstraction of naming convention could be detected by significant changes to the same "pattern".
- We were able to find dormant abstraction, of (so we guess) our patterns, in examining classes with exactly one method, and no instance fields. In other meaningless patterns, e.g., requiring that a class has precisely two methods and two instance fields, significance was found.

## 3.8 The Evolution of Software Collections

We now turn to the quest of checking the persistence of micro patterns across different implementations of the same design, and in the course of the software life cycle. To this end we consider the seven different implementations of the JRE as discussed in Section 3.5.

Table 3.5 is structured similarly to Table 3.4 except that in Table 3.5 we compare the micro pattern prevalence in the seven implementations of the JRE, i.e., in the corpus defined by

$$\textbf{JRE} \quad = \quad \{\mathsf{Kaffe}^{1.1}, \mathsf{Kaffe}^{1.1.4}, \mathsf{Sun}^{1.1}, \mathsf{Sun}^{1.2}, \mathsf{Sun}^{1.3}, \mathsf{Sun}^{1.4.1}, \mathsf{Sun}^{1.4.2}\}$$

Comparing the two tables we see that the values in the average, total, and median lines in Table 3.5 are close, just as they are in Table 3.4.

In comparing the standard deviation column ($\sigma$) in the two tables, we see that the variety in coverage level and entropy is much smaller in the related collections (Table 3.5) than the variety in the related collections (Table 3.4). For the majority of patterns (18 out of the 27), the variety in prevalence level in Table 3.5 is smaller than the variety in Table 3.5.

The variety of four patterns is about the same in both corpora. Only five patterns, Designator, Taxonomy, State Machine, Immutable and Sink showed a greater variety in the JRE-collections than in the unrelated collections.

We can therefore make the following qualitative conclusion:

**Conclusion 9** *Pattern prevalence tends to be the same in software collections which serve similar purposes, independent of the size of the collection.*

Examining the patterns whose variety is greater in the **JRE** corpus, we see that there was a large drop in their prevalence level with the progress of JRE implementations.

The drop in Immutable is explained by a change in the root of the exceptions hierarchy of JRE, `Throwable`, which broke the immutability of all of the classes in it.

The drop in Designator, Taxonomy, State Machine and Sink, is not so much in relative numbers but rather due to the fact that the development of new branches of the standard library did not make much new use of these patterns. In particular, the introduction of the fairly large and complex Swing library in $\mathsf{Sun}^{1.2}$, has induced a corresponding decrease in the ratio of Sink classes.

| Collection | Kaffe$^{1.1}$ | Kaffe$^{1.1.4}$ | Sun$^{1.1}$ | Sun$^{1.2}$ | Sun$^{1.3}$ | Sun$^{1.4.1}$ | Sun$^{1.4.2}$ | Total | Average | Median | Min | Max | $\sigma$ |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Designator | 1.3% | 0.8% | 0.8% | 0.4% | 0.4% | 0.3% | 0.2% | 0.4% | 0.6% | 0.4% | 0.2% | 1.3% | 0.4% |
| Taxonomy | 11.0% | 5.6% | 13.8% | 6.7% | 5.8% | 5.1% | 4.4% | 5.9% | 7.5% | 5.8% | 4.4% | 13.8% | 3.5% |
| Joiner | 0.4% | 0.3% | 0.0% | 0.8% | 0.9% | 1.2% | 0.7% | 0.8% | 0.6% | 0.7% | 0.0% | 1.2% | 0.4% |
| Pool | 1.9% | 1.9% | 1.4% | 1.6% | 1.8% | 2.3% | 1.9% | 1.9% | 1.8% | 1.9% | 1.4% | 2.3% | 0.3% |
| Sink | 21.9% | 27.5% | 23.5% | 17.0% | 17.6% | 17.3% | 20.6% | 19.4% | 20.8% | 20.6% | 17.0% | 27.5% | 3.9% |
| Record | 0.2% | 0.2% | 0.5% | 0.5% | 0.6% | 0.6% | 0.4% | 0.5% | 0.4% | 0.5% | 0.2% | 0.6% | 0.2% |
| Data Manager | 2.7% | 2.8% | 2.3% | 1.5% | 1.9% | 1.9% | 1.8% | 1.9% | 2.1% | 1.9% | 1.5% | 2.8% | 0.5% |
| Function Pointer | 1.0% | 1.1% | 0.5% | 2.1% | 1.2% | 1.7% | 2.0% | 1.6% | 1.4% | 1.2% | 0.5% | 2.1% | 0.6% |
| Function Object | 1.1% | 1.2% | 1.8% | 8.0% | 7.7% | 6.9% | 7.7% | 6.5% | 4.9% | 6.9% | 1.1% | 8.0% | 3.3% |
| Cobol Like | 0.9% | 0.6% | 0.5% | 0.4% | 0.4% | 0.4% | 0.4% | 0.4% | 0.5% | 0.4% | 0.4% | 0.9% | 0.2% |
| Stateless | 9.0% | 8.9% | 7.0% | 9.3% | 8.6% | 9.5% | 9.8% | 9.2% | 8.9% | 9.0% | 7.0% | 9.8% | 0.9% |
| CommonState | 2.0% | 1.3% | 1.9% | 1.4% | 2.8% | 3.6% | 2.4% | 2.5% | 2.2% | 2.0% | 1.3% | 3.6% | 0.8% |
| Canopy | 5.1% | 4.8% | 2.9% | 10.2% | 9.1% | 9.1% | 9.8% | 8.7% | 7.3% | 9.1% | 2.9% | 10.2% | 2.9% |
| Immutable | 13.3% | 5.1% | 15.1% | 17.6% | 16.7% | 6.8% | 7.6% | 10.7% | 11.7% | 13.3% | 5.1% | 17.6% | 5.2% |
| Box | 8.0% | 4.1% | 4.7% | 4.5% | 4.6% | 5.3% | 4.6% | 4.9% | 5.1% | 4.6% | 4.1% | 8.0% | 1.3% |
| Compound Box | 7.0% | 6.0% | 8.4% | 5.4% | 5.5% | 5.7% | 6.0% | 5.9% | 6.3% | 6.0% | 5.4% | 8.4% | 1.1% |
| Implementor | 9.7% | 17.0% | 9.3% | 27.6% | 27.1% | 21.5% | 26.0% | 23.2% | 19.7% | 21.5% | 9.3% | 27.6% | 7.9% |
| Overrider | 6.6% | 5.7% | 7.7% | 9.8% | 9.2% | 12.3% | 12.4% | 10.5% | 9.1% | 9.2% | 5.7% | 12.4% | 2.6% |
| Extender | 6.5% | 4.9% | 4.5% | 4.1% | 4.1% | 4.1% | 4.3% | 4.3% | 4.6% | 4.3% | 4.1% | 6.5% | 0.9% |
| Outline | 1.9% | 2.8% | 2.3% | 1.7% | 1.7% | 1.8% | 1.8% | 1.9% | 2.0% | 1.8% | 1.7% | 2.8% | 0.4% |
| Mould | 1.6% | 2.3% | 1.4% | 1.9% | 1.9% | 1.3% | 1.3% | 1.6% | 1.7% | 1.6% | 1.3% | 2.3% | 0.4% |
| State Machine | 1.6% | 3.1% | 2.5% | 1.3% | 1.4% | 1.8% | 1.5% | 1.7% | 1.9% | 1.6% | 1.3% | 3.1% | 0.7% |
| Pure Type | 10.1% | 14.9% | 12.4% | 8.4% | 8.7% | 9.5% | 7.7% | 9.3% | 10.2% | 9.5% | 7.7% | 14.9% | 2.6% |
| Augmented Type | 0.9% | 1.3% | 1.0% | 0.6% | 0.7% | 0.7% | 0.6% | 0.7% | 0.8% | 0.7% | 0.6% | 1.3% | 0.3% |
| Pseudo Class | 1.1% | 1.2% | 0.5% | 1.1% | 1.0% | 0.8% | 0.7% | 0.9% | 0.9% | 1.0% | 0.5% | 1.2% | 0.3% |
| Sampler | 2.5% | 1.2% | 1.6% | 1.3% | 1.2% | 1.1% | 1.2% | 1.3% | 1.5% | 1.2% | 1.1% | 2.5% | 0.5% |
| Restricted Creation | 2.0% | 2.5% | 1.3% | 1.2% | 1.8% | 2.2% | 2.3% | 2.0% | 1.9% | 2.0% | 1.2% | 2.5% | 0.5% |
| Coverage | 74.8% | 82.9% | 73.3% | 79.5% | 80.3% | 79.5% | 79.5% | 79.5% | 78.5% | 79.5% | 73.3% | 82.9% | 3.3% |
| Entropy | 4.98 | 5.08 | 4.68 | 5.17 | 5.25 | 5.25 | 5.27 | 5.34 | 5.10 | 5.17 | 4.68 | 5.27 | 0.21 |

Table 3.5: The prevalence, coverage and entropy of micro patterns in different implementations of the JRE.

Note that the two most largest differences are 19% in Implementor, between $\text{Sun}^{1.1}$ and $\text{Sun}^{1.2}$, and an 11% drop in Immutable between $\text{Sun}^{1.3}$ and $\text{Sun}^{1.4.1}$. The first difference can be attributed to the introduction of large interface-based libraries in $\text{Sun}^{1.2}$ (such as the *Swing* library and the `sun.java2d.*` packages). The latter difference is, as explained above, due to the change of class `Throwable` in $\text{Sun}^{1.4.1}$.

To appreciate the greater similarity in prevalence values, we can recheck the null hypothesis $\mathcal{H}_0[p]$. This time with respect to the collections in the **JRE** corpus. As it turns out, the hypothesis cannot be rejected as often as in the pruned corpus. The difference in the prevalence levels of Cobol Like were insignificant here just as in the pruned corpus, but there were five additional patterns for which the differences in prevalence levels are not significant: Outline, Augmented Type, Pseudo Class, Pool, Stateless, and Record.

Indeed, as indicated by Figure 3.4, the average separation index with respect to the pruned corpus is $47\%$, while the average with respect to the JRE's is $33\%$.

## 3.9 Related Work

Cohen and Gil [53] supplied some statistical evidence to the existence of *common programming practice*, which "good" programmers will follow in their coding. Their conclusions were obtained from a set of simple metrics, such as: number of parameters of a method, bytecode size of a method, number of static method calls, etc. Given the somewhat "technical" nature of these metrics, the deduction of meaningful conclusions regarding the design of a program, from a given vector of metrics values, is not an easy task.

In this paper, we took the natural challenge of bridging the gap: finding micro patterns which are at a slightly higher level than e.g., the number of parameters to a method, but at a lower level than design patterns.

Van Emde Boas [178] describes the trade off of expressivity (of the language used for describing design patterns) vs. the complexity of the pattern detection problem. He showed that lack of syntactic constraints on the design pattern definitions, results in the detection problem being undecidable.

Kraemer and Prechelt [157] developed the Pat system which detects structural design patterns (ADAPTER, BRIDGE, COMPOSITE, DECORATOR, PROXY) by inspecting a given set of C++ header files (`.h`), and storing extracted data as *Prolog* facts. Identification is carried out by invoking a Prolog query against a set of predefined Prolog rules describing the identifiable design patterns. This system had a detection precision of 14 – 50%. As the authors claim, the precision can be significantly improved by checking method call delegation information, which cannot be obtained from header files. Our approach is yields better results:

- The richer information available at **`.class`** file will allow our tools to inspect method call delegations, detect more types of patterns, and reduce the number of false positives.
- We are not restricting our research to Gamma et al.'s [81] design patterns. Any micro pattern is applicable.

Heuzeroth et al. [105] combine static and dynamic analysis for detection of design patterns (Behavioral: OBSERVER, MEDIATOR, CHAIN OF RESPONSIBILITY, VISITOR; Structural: COMPOSITE) in JAVA applications. The static analyzer applies various predicates over the source code (`.java` files) to obtain a set of candidates. The dynamic analyzer employs code instrumentation techniques to trace the behavior of the candidates at runtime. A candidates whose behavior is not conforming with the expected behavior of the relevant design pattern is filtered out. This technique's dependency on runtime information is a major drawback.

Brown [42] uses dynamic analysis of *Smalltalk* programs for the detection of Gamma et al. patterns. His technique is based on tracing of messages sent between objects.

The need for enhanced documentation tools has been stated by several works in the area of software visualization [74, 169, 170]. Another similar research is the work of Lanza and Ducasse [123], which suggest a technique for classifying methods of *Smalltalk* classes to one of five categories by inspecting their implementation. These authors' classification algorithm is partially based on common naming conventions.

Micro patterns are related also to a number of systems which allowed the programmer to add auxiliary, automatically checkable rules to code. Examples include Minsky's *Law Governed Regularities* [141, 142], or Aldrich, Kostadinov and Chambers's work on alias annotation [7]. Micro patterns are different in that they restrict the programmer's freedom in choosing such rules (unless the user comes with a new pattern), but on the other hand give a set of simple, pre-made, well defined rules backed up by extensive empirical support.

Statistical inference in the context of software was used in the past. For example, Soloway, Bonar, and Ehrlich [168], employed the $\chi^2$-test to answer questions such as the extent by which advanced programmers have greater tendency to use certain programming idioms, and the extent by which language support for the preferred idioms promotes program correctness.

## 3.10 Summary

People use patterns without thinking. This phenomenon is a consequence of the recognition built into every one of us, that *routine* is easier and safer than the time consuming and error-prone process of *decision making*. As demonstrated so many times in the past, patterns exist also in the programming world. In languages with rich system of attributes such as JAVA it is clear there are many (statistical) correlations between these attributes. For example, we expect classes which define static fields to define static methods, etc.

Micro patterns step further beyond the simple conclusion that there are many inter-correlations between the setting of (say) attributes and selection of types. Our experiments show that there are distinct patterns which the majority of JAVA software follows. In fact, we gave meaning, name and significance to many of these correlations.

We described a catalog of micro patterns, which can be used as a mental skeleton to mold mundane modules, allowing programmers to become more productive. For example, by using the catalog, much of the coding work is reduced to the mere issue of selecting a pattern for a class (often dictated by the system design), and then laboriously filling in the missing details.

We showed (Conclusion 1) that an overwhelming majority of JAVA classes follows one or more of the patterns in the catalog. (The remaining classes either fit yet unknown patterns, or represent code locations which required more skill than routine.) We used statistical methods to increase the confidence that these patterns capture sound ideas.

Despite the fact that more than half the classes can be described by one of the five leading patterns (Conclusion 3), we found that each of the patterns in the catalog contributes (Conclusion 5) to the 4–5 bits or so of design information that the catalog as a whole reveals (Conclusion 4).

After noticing (Conclusion 6) that there is a considerable variety in the use of patterns in different domains, statistical analysis was carried out to understand better the nature of this variety. This analysis has shown that in almost half of the cases, changes in a pattern's prevalence levels, between two software collections, were not an artifact of random fluctuation. This indicates that the choice of patterns is not merely a follow up of the language constraints and that it is effective for distinguishing programming context.

# Chapter 4

# Program Transformation

Software maintenance encompasses various activities which generally fall into these two broad categories: (*i*) Enhancing the capabilities of existing code, including defect fixing (debugging), the introduction of new functionality, and the leveraging the existing functionality by integration with external systems; (*ii*) Reworking the structure of existing code in order to reduce its naturally growing entropy.

Clearly, code transformation mechanisms, and in particular refactoring, are instrumental to supporting the latter category [77]. More subtle is the observation that many techniques that pertain to the former category—such as aspect-oriented programming, meta-programming, or even object-relational mapping (which allows the integration with external storage systems)—can also be realized by code transformers.

A code transformation mechanism can be conceptually divided into two components: the *guard* [68], which describes the pre-requisites for the transformation, and the *transformer*, which is the clockwork behind the production of output for the matched code.

Taking aspect-oriented programming languages as an example, we note that the *pointcut* plays the role of the guard, while the *advice*, along with the underlying weaver, plays the role of the transformer. As we already showed (Section 2.4.2), JTL is an effective means for the specification of pointcuts. Generalizing this, we note that in transformation systems that work on a static representation of a program, *a guard is usually a formal pattern*. The only exceptions to this observation are guards that are built on a language that is too powerful to meet the simplicity criterion of formal patterns.

This part of our work focuses on the transformer component. We describe *JTL*$^*$ , a backward-compatible extension of JTL which augments the base language with output generation capabilities. We argue that transformation tasks often correspond to the structure of the code element matched by the guard. Thus, the association of transformer expressions with distinct parts of the formal pattern expressing the guard, greatly simplifies the authoring of such transformers and streamlines the integration of guards and transformers.

**Background.** The logic paradigm is attracting increasing attention in the software engineering community, e.g., Hajiyev, Verbaere and de Moor's CodeQuest system [101] and Janzen and De Volder's JQuery [111] and even dating back to the work of Minsky [142] and law governed systems [143]. The aspect-orientation school is also showing a growing interest in this paradigm, with work on aspect-oriented logic meta programming [64], the ALPHA system [155], Carma [41], LOGICAJ [159], the work of Gybels and Kellens [99] and more. There is even a dual line of research concentrating in the application of aspects to PROLOG programs [22].

Most of this previous art is characterized by use of the logic paradigm for *querying* code, rather than code generation. JTL$^*$ makes the next step. It further utilizes the logic paradigm by showing that it is useful for the expression of both guards *and* transformers.

Chapter 4.1 describes the key constructs of JTL*. Underlying these constructs is the association of a "baggage" string variable (or more generally, an array of such variables) with each predicate. This variable can be interpreted as the query's output, or *result*.

There are clear and simple rules for the implicit derivation of the result of a compound predicate from the results of its components. Programmer intervention is therefore required only where the defaults are not appropriate.

The JTL* native- and library-predicates are designed such that their string baggage is the JAVA code fragment that best describes their inputs. Thus, native predicate `final` returns the string `"final"` if the subject is indeed final. The result of the expression `final abstract`, which is the string `"final abstract"`, is the concatenation of the results of the enclosed terms. Moreover, many *tautologies*, such as `visibility` (returning either of `"public"`, `"protected"`, `"private"` or the empty string, depending on the argument's visibility) were added to the library.

The applications presented in Section 4.3 show how JTL* programs (and formal patterns in general), can be used for *rephrasing* purposes—target language is the same as the source language—such as refactoring or aspect-oriented programming, as well as for *translation* purposes—target language is different than the source language—such as object relational mapping, detection of coding convention violations in Lint-like tools and more.

## 4.1 The JTL* language

This section describes JTL* by presenting the differences between JTL and JTL*.

In a nutshell, JTL is a simply typed high level language in the logic paradigm, whose underlying model is that of DATALOG augmented with well founded negation, but without function symbols. That is to say that JTL (unlike PROLOG) uses variable *binding*, rather than *unification*, and predicates's parameters just as other variables, are atomic. Abstraction mechanisms over plain DATALOG offered by JTL include features such as set predicates, quantification, and scoped definitions.

The principle behind JTL* is simple: every predicate implicitly carries with it an unbounded array of baggage **`STRING`** variables, which are computed by the predicate in a constructive manner. These variables are output only—an invocation of a predicate cannot specify an initial value for any of them. The compilation process translates into DATALOG only baggage which is actually used. Thus, plain JTL programs are not changed, since no baggage variables are used.

In most output-producing applications, only the first baggage variable, called the *standard output* or just the *output* of the predicate, is used. The output parameter is sometimes called the "returned value" in the context of program transformation.

### 4.1.1 Simple Baggage Management

The essence of the baggage extension is that the output of a compound predicate is constructed by default from its component predicates. Since the initial purpose of our extension was the production of JAVA code, the library was so designed that the output of JTL* predicates that are also JAVA keywords (e.g., `synchronized`) return their own name on a successful match. Other primitive predicates (e.g., `method`) return an empty string. Type and name patterns return the matching type or name. The fundamental principle is that whenever possible, any predicate returns the text of a JAVA code fragment that can be used for specifying the match.

The returned value of the conjunction of two predicates is the concatenation of the components. By default, this concatenation trims white spaces on both ends of the concatenated components,

**Figure 4.1** Definitions of common tautologies.

```
header := modifiers declarator '?*' parents;
modifiers := concreteness strictness visibility ...;
concreteness := abstract | final | nothing;
strictness := strictfp | nothing;
declarator := class | interface | enum | @interface;
parents := superclass optional_interfaces;
superclass := extends T T is_not Object | nothing;
```

and then injects a single space between these. Disjunction of two predicates returns the string returned by the first predicate that is satisfied. Thus, for example, the predicate

```
public static [int | long] field 'old?*'
```

can be applied to some field called `oldValue`, in which case it will generate an output such as "`public static int oldValue`".

String literals are valid predicates in JTL*, except that they always succeed. They return in the output their own value. By using strings, predicates can generate output which is different from echoing their building blocks. For example, the pattern

```
class '?*' "extends Number"
```

generates, when evaluated with class `Complex` as its input, the following output "`class Complex extends Number`". The string literal in the pattern does not present any requirement to the tested program element, and the string result need not be an echo of that element. The pattern above, for example, will successfully match class `String`, which does *not* extend `Number`.

String literals are just one example of what we call *tautologies*: predicates which hold for any value of their parameters. Tautologies are used solely for producing output. The most simple tautology is the predicate `nothing`, which returns the empty string, i.e., `nothing := "";`. The JTL* library offers many such tautologies, e.g., `visibility` mentioned earlier, or `multiplicity`, defined as `static | nothing`.

Other tautologies in the library include `modifiers`, returning the string of all modifiers used in the definition of a JAVA code element; `signature`, returning the type, name, and parameters of methods, or just the type and name of fields; `header`, including the modifiers and signature; the (primitive) `torso`, returning the body (without the head and embracing curly brackets) of a method or a class; and `preliminaries`, returning the package declaration of a class, etc. Tautology `declaration`, whose baggage is the full definition of a program element, is useful for the exact replication of the matched element. We have (for classes and methods, but not for fields)

```
declaration := preliminaries header "{" torso "}";
```

Figure 4.1 demonstrates how tautology `header` and several other auxiliary tautologies are defined.

Examining the definitions in Figure 4.1 we see the recurring pattern of a disjunction expression whose last branch is `nothing`. In some definitions, such as `declarator`, all possible options are enumerated, thereby obviating the need to a "default" option. `optional_interfaces` that is used by `parents` is subsequently defined in Section 4.1.5. Finally, note the actual definitions of these tautologies are a bit more involved, since they have to account for annotations and generic parameters, and must have overloaded versions for elements of kind **`MEMBER`**.

The negation operator, `!`, discards any output generated by the expression it negates. For example, `!static` will generate an empty string when successfully matched. Thus `multiplicity`

can be also defined as `static | !static`.

Finally, if a JTL* main query returns several answers, then the output of the whole program is obtained by concatenating the outputs of each result, but the order is unspecified. Thus, if a JTL* query matches all method called by a given method, then the order by which these methods are generated is unspecified, and hence the concatenation of the string results can be in any order. In most cases however, JTL* programs are written to produce a single output, or be applied in a setting where only one output makes sense.

### 4.1.2 Multiple Baggage

It is sometimes desirable to suppress the output of one or more constituents of a pattern, even if they are not negated. This can be done by prepending the percent character, `%`, to the expression. For example, the predicate

```
%public %final '?*';
```
(4.1)

will match any public final element, but print only its name, without the modifiers that were tested for. Predicate (4.1) can also be written using square brackets:

```
%[public final] '?*';
```
(4.2)

The suppression syntax is in fact one facet of a more complex mechanism, which allows predicates to generate multiple string results, directed to different *output streams*. By default, any string output becomes part of string result 1, which is normally mapped to the standard output stream (`stdout` in Unix jargon). Also defined are string result 0, which discards its own content (`/dev/null`), and string result 2, the standard error stream (`stderr`).

To direct an expression's string output to a specific string result, prepend a percent sign and the desired string result's number to the expression. A percent sign with no number, as used above in (4.1) and (4.2), defaults to `%0`, i.e., a discarded string result.

For example, consider the following predicate:

```
testClassName := %2[
    class '[a-z]?*' "begins with a lowercase letter."
];
```

If matched by a class, it will send output to string result 2, i.e., the standard error stream; possible output can be "`class badlyNamed begins with a lowercase letter`". If, however, the expression is not matched (in this case, because the class does not begin with a lowercase letter), no output is generated.

By using disjunction, we can present an alternative output for those classes that do not match the expression; for example:

```
testClassName :=
 %2[class '[a-z]?*' "begins with a lowercase letter." ]
 | [class '?*' "is properly named."];
```
(4.3)

Because it is not directed to any specific stream, the string result of the second part of the predicate is directed to the standard output. As explained below (Section 4.2) the disjunction operator's output is evaluated in a non-commutative manner, so that its right-hand operand can generate output only if its left-hand one yielded false. Thus, predicate (4.3) will generate exactly one of two possible messages, to one of two possible output streams, when applied to a class. The query `testClassName[X]` is a tautology for classes, matching any class `X`; its *output*, however, depends on `X`'s name.

A configuration file binds any string result generated by a JTL* program to specific destinations (such as files). The default destinations of string results 0, 1 and 2 can therefore be overridden, and additional string destinations (unlimited in number) can also be defined.

We stress that these multiple output streams are all used in a single translation job, handling a single JAVA file. To process a large codebase with a large number of classes, one needs not define an output stream per file; rather, the same JTL* program is executed repeatedly, once per class. Normally, only one output stream (plus the null stream for suppression, if relevant) is used in each such execution. The task of executing the same JTL* program on repeatedly, once per class, should itself be managed by some configuration tool, such as Ant[1].

JTL* also includes mechanisms for redirecting the string result generated by a subexpression into a different string result in the enclosing expression, or even to bind string results to variables. The syntax `%`$n$`>`$m$ $p$ will redirect the string result of predicate $p$ in output stream $n$ into output stream $m$ of the caller. For example, the expression

```
[%2>1 p1] | "failed"
```
(4.4)

will yield the string result that `p1` sends to output stream 2, in output stream 1. If `p1` fails, (4.4) will generate the output `failed`. However, if `p1` succeeds without generating any output in stream 2 (e.g., it generates no output at all, or output to other streams only) then (4.4) will generate no output. To bind string results to a variable, the syntax `%`$n$`>`$V$ $p$ can be used, binding the output of predicate $p$ in output stream $n$ to variable $V$ of the caller. Thus, writing

```
%2>Error refactor
```

will assign the output stream 2 of predicate `refactor` to variable `Error`. Redirection into a variable is only permitted if the variable is output-only—that is, if no other assignments into it occur in the predicate. To determine whether a variable is output-only, we rely on JTL's pre-analysis stage [52] designed to decide whether a query result is close (e.g., returning all class ancestors), or open in the sense that it depends of whole program information (e.g., returning all class's descendants). A by-product of this analysis is the decision whether a variable is output only.

If necessary, file redirection can also be achieved from within JTL* just as in the AWK programming language. For example,

```
%2>"/temp/error.log" refactor
```

will redirect the standard error result of `refactor` to the file named `/tmp/error.log`. It is also possible to redirect output into a file whose name is computed at runtime, as explained at the conclusion of the following Section 4.1.3.

Admittedly, redirection syntax degrades from the language elegance, but it is expected to be used rather rarely.

### 4.1.3 String Literals

Baggage programming often uses string literal tautologies. Escaping in these for special characters is just as within JAVA string literals. For example, `"\n"` can be used to generate a newline character. An easier way to generate multi-line strings, however, is by enclosing them in `\[ ... \]`, which can span multiple lines.

When output is generated, a padding space is normally added between every pair of strings. However, if a plus sign is added directly in front of (following) a string literal, no padding space will be added before (after) that string. For example, the predicate

[1] http://ant.apache.org

```
class "New"+ '?*'
```

will generate the output "`class NewList`" when applied to the class `List`.

The character # has a special meaning inside JTL* strings; it can be used to output the value of variables as part of the string. For example, the predicate

```
class "ChildOf"+ ['?*' is T] "extends #T"          (4.5)
```

will yield "`class ChildOfInteger extends Integer`" when applied to `Integer`. The first appearance of `T` in predicate (4.5) captures the name of the current class into the variable; its second appearance, inside the string, outputs its value.

When applied to JAVA types, a name emits returns (as a string value) the short name of a class, whereas a value of kind **`TYPE`** emits the fully-qualified name of the type. We can therefore write (4.5) as

```
class "ChildOf"+ '?*' "extends" #
```

to obtain "`class ChildOfInteger extends java.lang.Integer`".

The sharp character itself can be generated using a backslash, i.e., `"\#"`. To output the value of # (the current receiver) in a string, just write `"#"`. For example, the following binary tautology, when applied to an element of kind **`MEMBER`**, outputs the name of that element with the parameter prepended to it:

```
prepend Prefix := "#Prefix" +"#";
```

In case of ambiguity, the identifier following # can be enclosed in square brackets. More generally, # followed by square brackets can be used to access not only variables, but also output of other JTL* expressions.[2] For example, the following tautology returns a renamed declaration of a JAVA method or field:

```
rename Prefix := modifiers typed _ prepend Prefix [
      method (*) throwsList "{ #[torso] }"
    | field "= #[torso];"
  ];
```

Useful redirections can be achieved via predicate embedding. Consider for example predicate `fix_equals` designed to add an `equals()` method to classes which fail to define it. Then, relying on a tautology named `path` whose output is the file system directory path as computed from the full class name, the following predicate regenerates its input in a new directory structure.

```
regen_equals := %1>"/home/new_root/#[path]" fix_equals
```

Applying `regen_equals` to all classes in a directory will regenerate, under `/home/new_root/`, the directory structure applying `fix_equals` to each class in it.

### 4.1.4 List Conditions

In JTL, the only construct that had some notion of "order" was the method signature pattern (Section 2.1.2). Other than this dedicated construct there is neither language nor library support for manipulation of lists (ordered sets) in user-defined predicates.

In JTL* list of program elements are first class values in the language. While this extension is not directly related to output generation, we found that in many program transformation scenarios—such as: the transformation of the list of arguments a method—list manipulation capabilities are needed.

[2] Note that using JTL* expressions inside a string literal may mean that the literal is not a tautology, e.g., `"#[public|private]"` is not a tautology.

Specifically, JTL* supports list values by the introduction of kinds such as **MEMBERS**, **TYPES**, etc. which prescribe an immutable list of the values of the corresponding atomic kind (respectively: **MEMBER**, **TYPE**).

The unary predicate `nil` ensures that its subject is the empty list. New lists can be created by the ternary predicate `cons` which creates a new list by prepending an element to an existing list: `X cons[Xs,Ys]` will select into `Ys` the list formed by prepending the element `X` to the list `Xs`.

As common in logic programming these predicates can be used in multiple ways, as determined by their computability specification: If subject is unknown, `nil` will select the empty list into the subject. If the subject is known, it will simply check that the subject is the empty list. `cons` can either compose a new list (if the last parameter is unknown and the other two are known) or it can decompose a list into its first element and its longest proper suffix (if the last parameter is known, and the other two are unknown). Together, these two predicate form the JTL* equivalent of the standard LISP [94] system of representing lists.

A standard library predicate that "extracts" a list from a JAVA program is the binary predicate `args`. Specifically, `M args Ts` holds if `M` is a method and `Ts` of kind **TYPES** is the list of types of the formal parameters of `M`. Arbitrary processing of `Ts` can then be achieved by eliciting individual elements via a recursive application of `cons` and `nil`.

Using list conditions, however, provides a simpler alternative. List conditions are very similar to quantification scopes (Section 2.1.4). They are specified using regular parenthesis, "(" and ")", rather than curly braces. Insides these parenthesis appear quantifiers (**`exists`**, **`all`**, etc.) just as in set conditions.

The difference in evaluation is that with list conditions, JTL* searches for a disjoint partitioning of the list into consecutive sublists, such that these sublists satisfy the enclosed list queries, in order. Therefore, list queries are meaningful only for generators that provide an ordered list rather than an unordered set.

A concrete example is given in the following expression:

```
args: (many abstract, int, exist final, one public)
```

This expression is evaluated in two steps: (a) list generation; and (b) application of the quantified conditions to the list—this is achieved by searching for a disjoint partitioning of the list into consecutive sublists that satisfy the specified quantifiers. In this example, we search for $T_s$ such that `# args` $T_s$ holds, and then apply the four quantifiers to it. The predicate holds if there are sublists $T_1$, $T_2$, $T_3$, and $T_4$, such that $T_s$ is the concatenation of the four, and it holds that:

- There is more than one `abstract` type in $T_1$,
- Sublist $T_2$ has precisely one element, which matches `int`,
- There is at least one `final` type in $T_3$, and
- There is exactly one `public` type in $T_4$.

If one of the list queries specifies no quantifier (as `int` in the last example), the semantics is that the respective sublist has exactly one element matching the query.

The default generator for list conditions is `args:` (compared with `members:` for set conditions ). Now, the argument list pattern `()` (matching functions with no arguments) is shorthand for `args: (no true)`, while `(*)` is shorthand for `args: (all true)`. Similarly, the expression `(_,String,*)` is shorthand for `args: (one true, String, all true)`.

### 4.1.5 Baggage Management in Quantifiers

In the `rename` predicate example above (Section 4.1.3), the term `(*)` outputs the list of all parameters of a method. Quantifiers scopes generate output like any other compound predicates.

Different quantifiers used inside the generated scope generate output differently. In particular, **one** will generate the output of the set or list member that was successfully matched; **many** and **all** will generate the output of every successfully matched member; and **no** generates no output. The extension introduces one additional quantifier, which is a tautology: writing **optional** $p$; in a quantification context prints the output of $p$, but only if $p$ is matched.

For example, the following predicate will generate a list of all fields and methods in a class that were named in violation of the JAVA coding convention:

```
badlyNamedClassMembers := %class %{
 [field|method] '[A-Z]?*' "is badly named.";
%}
```
(4.6)

By default, the opening and closing characters (`()` or `{}`) print themselves; their output can be suppressed (or redirected) by prepending a `%` to each character, as above.

Two (pseudo) quantifiers, **first** and **last**, are in charge of producing output at the beginning or the end of the quantification process. The separator between the output for each matched member (as generated by the different quantifiers) is a newline character in set quantifiers, or a comma in the case of list quantifiers. This can be changed using another pseudo-quantifier, **between**. The tautology `optional_interfaces` used in the above definition of `header` in Figure 4.1 requires precisely this mechanism:

```
optional_interfaces := implements: %{
     first "implements";
     exists "#"; -- and names of all super interfaces
     between ","; -- separated by a comma
     last nothing; -- and no ending text
   %}
   | nothing;
```

Since we use the **exists** quantifier, the entire predicate in the curly bracket fails if the class implements no interfaces—in which case the "**first**" string `"implements"` is not printed; if this is the case, then the `| nothing` at the end of the definition ensures that the predicate remains a tautology, printing nothing if need be.

## 4.2 Reduction to Datalog

This section describes how the baggage extension is realized by the JTL*-to-DATALOG compilation process. Assuming familiarity with Section 2.3.1 we will simply describe the two main deviations from the JTL-to-DATALOG translation scheme.

Output generation does not require the introduction of any side-effect into JTL*. Rather, when compiling JTL* predicates to DATALOG, we have that the string output is presented as an additional "hidden", or implicit parameter to DATALOG queries. This parameter is used for output only.

**Conjunction.** The translation of a conjunctive expression, simply concatenates the baggage of the enclosed terms using the DATALOG ternary predicate `str` which concatenates two input strings (first two parameters) into an output string (third parameter). This is illustrated by Figure 4.2.

**Figure 4.2** JTL*-to-DATALOG translation showing the construction of the baggage results in a conjunction expression.

```
pa := public abstract;
```

(a) JTL*

```
pa(This,Result) :- public(This,Result1), abstract(This,Result2),
  str(Result1,Result2,Result).
```

(b) DATALOG

The JTL* predicate in Figure 4.2(a) is a simple disjunction of terms `public` and `abstract`. The corresponding DATALOG predicate in Figure 4.2(b) shows how the value of the implicit parameter, `Result`, is constructed, via predicate `str`, from results of the enclosed terms.

Multiple output streams (Section 4.1.2) mandate the use of multiple baggage variables. The "redirection" of output from one stream to another involves changing the order in which the baggage variables are passed from one DATALOG predicate to another; redirection to a variable (the %$n$>$V$ syntax) implies binding the JTL variable $V$ to a baggage parameter.

**Disjunction.** In a JTL* disjunction expression, output will be generated only for the first matched branch. To this end, each branch of the disjunction is considered true only if all previous branches are false, as shown in Figure 4.3.

**Figure 4.3** JTL*-to-DATALOG translation showing the construction of the baggage results in a disjunction expression.

```
p_or_a := public | abstract;
```

(a) JTL*

```
p_or_a(This,Result) :- public(This,Result).
p_or_a(This,Result) :- !public(This,_), abstract(This,Result).
```

(b) DATALOG

The DATALOG predicate in Figure 4.3(b) contains two rules. The first one is a straightforward translation of the left-hand side term of the disjunction in the JTL predicate (Figure 4.3(a)). The second rule, starts by negative the left-hand side term, and then continues with a straightforward translation of the right-hand side term.

Note that the operation remains commutative with regard to the question of which program elements match it; the pattern `abstract | public` will match exactly the same set of elements. Commutativity is compromised only with regard to the string output, where it is undesired.

To better appreciate this design choice consider predicate `add_equals`, which unconditionally adds an `equals()` method to its implicit class argument (see Figure 4.4). Then, there is a straightforward implementation of tautology `fix_equals` which only adds this method if it is not present:

```
fix_equals := has_equals declaration | add_equals;
```

(We assume a standard implementation of the obvious predicate `has_equals`.) However, this implementation will fail if baggage is computed commutatively. The remedy is in the more verbose version

```
fix_equals := has_equals declaration | !has_equals add_equals;
```

More generally, it was our experience that in the case of alteration between several alternatives only one output is meaningful. Our decision then saves the overhead of manual insertion of code (which is quadratically increasing in size) to ensure mutual exclusion of these alternatives. Con-

versely, if several alternatives are satisfied, we found no way of combining their output commutatively.

## 4.3 Transformation Examples

This section shows how JTL*'s baggage can be used for various tasks of program transformation. The description ignores the details of input and output management; the implicit assumption is that the transformation is governed by a build-configuration tool such as Ant, which directs the output to a dedicated directory (without overriding the originals), orchestrates the compilation of the resulting source files, etc. This makes it possible to apply a JTL* program in certain cases to replace an existing class, and in others, to add code to an existing software base.

### 4.3.1 Using JTL* in an IDE and for Refactoring

Baggage output makes it possible for JTL* to not only detect violations of coding practices (Section 2.4.4), but also provide useful error and warning messages. Pattern (4.6) in the previous section shows an example.

JTL* can also be put to use in refactoring services supplied by the IDE [3]. The following pattern extracts the public protocol of a given class, generating an **`interface`** that the class may then implement:

```
elicit_interface := %class −− Guard: applicable to classes only
   modifiers "interface" prepend["P_"] −− Produce header
      { −− iterate over all members
        optional %public !static method header ";" ;
      };
```

We see in this example the recurring JTL* programming idiom of having a *guard* [68] which checks for the applicability of the transformation rule, and a *transformer* which is a tautology. (Note that by convention, the output of guards is suppressed, using the percent character.) The interface is generated by simply printing the header declaration of all public, non-static methods.

The converse IDE task is also not difficult. Given an interface, the following JTL* code shows how, as done in Eclipse, a prototype implementation is generated:

```
defaultVal := %boolean "false"
   | %primitive "0"
   | %void nothing
   | "null";
gen_class := %interface −− Guard: applicable to interfaces only
   modifiers "class" prepend["C_"] "implements #" {
      header \[ { return #[defaultVal]; } \]
   };
```

The above also demonstrates how JTL* can be used much like output-directed languages such as PHP and ASP: output is defined by a multi-line string literal, into which, at selected points, results of evaluation are injected. Here, the value of the tautology `defaultVal` is used to generate a proper default returned value.

Another common IDE/refactoring job is the addition of standard, yet nontrivial methods to classes. For example, in JAVA, the `equals` method is important for proper usage of classes in, e.g., the collection framework. The predicate in Figure 4.4 adds a proper implementation of `equals` to its argument class, without changing anything else.

[3] We note, however, that some refactoring steps exceed JTL*'s expressive power.

**Figure 4.4** A JTL transformation that generates a proper `equals` method.

```
1 add_equals := %class header %[# is T] declares: {
2     ![boolean equals (Object)] declaration;
3     last \[
4       @Override boolean equals(Object obj) {
5         if (obj == null) return false; if (obj == this) return true;
6         if (!obj.getClass().equals(this.getClass())) return false;
7         #T that = (#T)obj; // downcast the parameter to the current type
8         #[T compareFields] // invoke helper predicate for field comparison
9         return true; // all field comparisons succeeded
10      }
11    \];
12
13    let compareFields := { -- generate field-comparison code
14      %[primitive field, '?*' is Name] -- guard for primitive fields
15        "if (this.#Name != that.#Name) return false;";
16
17      %[!primitive field, '?*' is Name] -- guard for reference fields
18        "if (!this.#Name.equals(that.#Name)) return false;";
19    }
20  };
```

The term `%class` (Figure 4.4, Line 1) is used as the first guard, filtering non-class elements. Next, the `header` tautology outputs the class's header. The expression "`%[# is T]`" captures the implicit parameter into the variable `T`, thereby making it accessible inside the generated scope.

We then use `declares` as a generator, and iterate over all members defined in this class. Members that do not match the signature of `equals` (as presented in Line 2) will be copied without change, using the `declaration` predicate. In effect, this filters out any existing definition of the method we are about to add. Then, using the pseudo-quantifier **`last`**, code for the `equals` method is added at the end of the class body. This code meets the strict contract prescribed for `equals` by the JAVA language specification.

In Line 8, the predicate invokes another predicate, `compareFields`. Looking at `compareFields`, we find that it provides two different possible outputs for fields, each with its own guard: one for primitive fields and one for reference-type fields, using the appropriate comparison mechanism in each case.

Comparing `gen_class` with `add_equals` brings this subtle point. `gen_class` uses the default **`exists`** quantifier which implies iterating over every matching element; but since no condition is presented – "header" is a tautology – all elements will be visited. `add_equals` does likewise, iterating over every matching element, but does include a condition (see Line 2) so that only certain elements will be sent to output. Then, the pseudo-quantifier **`last`** generates extra output.

### 4.3.2 JTL* as a Lightweight Aspect-Oriented Language

With its built-in mechanism for iterating over class members, and generate JAVA source code as output, it is possible to use JTL* as a quick-and-dirty aspect-oriented language. The basic technique is illustrated by Figure 4.5.

The predicate in Figure 4.5 is in fact an "aspect" which generates a new version of its class

**Figure 4.5** A JTL* predicate that weaves a logging aspect to its input.

```
1 loggingAspect := %class header declares: {
2     -- pointcut:
3     let targetMethod := public !abstract method;
4
5     -- advice:
6     %targetMethod header \[ {
7           System.out.println("Entering method #");
8           #[torso]
9       }
10    \]
11    | declaration;
12  };
```

parameter. This new version is enriched with a simple logging mechanism, attached to all public methods by means of a simple "before" advice.

The local predicate `targetMethod` defines the kinds of methods which may be subjected to aspect application—in other words, it is a guard serving as a pointcut definition. The advice is the existential quantifier whose condition is a tautology; therefore, output will be generated for every element in the set.

The first branch in this condition starts with the the guard expression `%targetMethod` (Line 6), which effectively binds the pointcut to the advice. If the member is matched against the guard, the method's header is printed, followed by a *modified* version of the method body.

If, however, the member does not match the guard, the disjunction alternative `declaration` will be used—i.e., class members that are not concrete public methods will be copied unchanged.

Having seen the basic building blocks used for applying aspects using JTL*, we can now try to improve our logging aspect. For example, we can change the logging aspect so that it prints the actual arguments as well, in addition to the invoked method's name. To do so, we define the tautology depicted in Figure 4.6.

**Figure 4.6** A JTL* predicate that generates a string with the values of the actual arguments of the subject method.

```
actualsAsString := %(
    first "(";
    last ")";
    between ", ";
    argName; -- at least one; iterate as needed
  %)
  | "()"; -- no arguments
```

When the predicate from Figure 4.6 is evaluated with a method with signature such as `void f(int p1, String p2, char p3)` it will generate as output the following JAVA expression

```
"(" + p1 + ", " + p2 + ", " + p3 + ")"
```

which is exactly what we need to print the actual parameter values.

The code generated is specific per method to which the advice is applied. Note that implementing an equivalent aspect with ASPECTJ requires the usage of runtime reflection in order to iterate over each actual parameter in a method-independent manner.

**Figure 4.7** A JTL* predicate realizing an aspect that logs parameters, and return values.

```
1  loggingAspect2 := %class header declares: {
2      -- pointcut definition:
3      let targetMethod := public !abstract method;
4
5      -- rename matching methods:
6      %targetMethod rename["original_"]
7         | declaration; -- other elements copied unchanged.
8
9      -- reiterate over matching methods:
10     %targetMethod %[typed R] header "{" -- generate header, ‘‘{’’
11     [
12        -- generate torso, differently for void and non-void methods:
13        %!void -- guard for non-void methods
14        \[
15           System.out.println(”Entering #” + #[actualsAsString]);
16           #R result = #[prepend[”original_”]] #[argNames];
17           System.out.println(”Returned ” + result);
18           return result;
19        \]
20        | -- deal with void methods
21        \[
22           System.out.println(”Entering #Name” + #[actualsAsString]);
23           #[prepend[”original_”]] #[argNames];
24        \]
25     ]
26     "}"; -- generate closing ‘‘}’’
27  };
```

JTL* can be used to define not only **`before`**, but also **`around`**, **`after returning`** or **`after throwing`** advice, by renaming the original method and creating a new version which embeds a call to the original. Figure 4.7 is a version of the logging aspect which also reports returned values.

The predicate in the figure preforms two “iterations” over the members declared in a class. In the first iteration (Lines 6–7), methods which match the pointcut (defined in Line 3) are renamed. The second iteration (Lines 11–26) regenerates matching methods with a new body that calls their renamed version, while adding the appropriate logging instructions.

It is interesting to see how guards and transformers are nested in the second iteration. At first, the phrase `%targetMethod header` has two components: the guard, which lets through only members that match the pointcut, and a tautology which regenerates the header of matching items.

Subsequently, we see the compound expression (Lines 12–25) which is in charge of printing the method’s new torso. This disjunction has two parts: in the first (Lines 13–19), a guard is applied to produce output for non-void methods; the second part (Lines 22–24) comes into action if the first fails, and produces output solely for void methods. In the aspect-oriented terminology, we see that pointcuts and advices can be intermixed and nested.

Note how the arguments list is copied to create the method invocation, using `#[argNames]`. This tautology (for methods or constructors) is defined thus:

```
argNames := ( optional argName );
```

Because the default separator in list generators is a comma, the result is formatted exactly as we

**Figure 4.8** A JTL* producing a SINGLETON version of its subject.

```
singleton :=
  %[!{ one constructor(); public constructor (); }] −− guard
    %2 "# must have a single constructor (public, zero-args)."
  | "public" class "Singleton"+ '?*', %[# is T] {
      constructor is C;
      last \[
        private #T() { C.torso } // constructor is private
        private static #T instance = null;
        public static #T getInstance() {
          if (instance == null) instance = new #T();
          return instance;
        }
      \];
    };
```

need it.

Unlike ASPECTJ aspects, which use `Object` references to capture return values in a type-independent manner and must therefore rely on the boxing and unboxing of primitive values, the aspect presented above involves no boxing of primitives. For example, if the method being processed is of type **`int`**, then the generated replacement method will include a local variable `result` of the primitive type **`int`**.

The main limitation of writing aspects in JTL* is that we have no way to traverse and modify the internals of method bodies. JTL* is therefore limited to **`execution`** pointcuts only. In particular, advice that should be applied to each access to a variable (**`get`** and **`set`** pointcuts), or advice that should be applied to exception catch blocks, etc., cannot be created with JTL*. This limitation is not unique to JTL*, however; several other aspect-oriented solutions take the same approach, some (such as the Spring framework[4]) by a well-reasoned, explicit choice.

In the examples above, the generated class has the same name as the original class, i.e., weaving-by-replacement is used. However, JTL* can just as easily be used for weaving-by-subclassing, and in particular can be used for implementing shakeins [56]. Indeed, this is but one example of how JTL* can be used to implement aspects; Czarnecki and Eisenecker [61] explain that the "striking similarity" between code transformations and aspects stems from the fact that both "may look for some specific code patterns in order to influence their semantics," and "[f]or this reason, transformations represent a suitable implementation technology for aspects or aspect languages".

The following section discusses additional uses for JTL* that can be reached by replacing, augmenting, or subclassing existing classes.

### 4.3.3 Templates, Mixins and Generics

Since JTL* can generate code based on a given JAVA type (or list of types), it can be used to implement generic types. The `singleton` predicate presented in Figure 4.8 is a simple example: it is a generic that generates a SINGLETON class [81] from a given base class. Given class, e.g., `Connection`, this predicate will generate a new class, `SingletonConnection`, with the regular singleton protocol.

[4] http://www.springframework.org

**Figure 4.9** A program that performs a mixin-like transformation, after verifying that the target class meets some basic requirements.

```
undoMixin := "public" class %[# is T] "Undoable#T extends #T" {
  %[!private void 'setName (String)]
  %[!private String 'getName ()]
  %[no !private 'undo ()]
  last \[
    private String oldName;
    public void undo() { setName(oldName); }
    public void setName(String name) {
      oldName = getName();
      super.setName(name);
    }
  \];
}
```

The seemingly trivial predicate from Figure 4.8 cannot be implemented using JAVA's generics, because those rely on type-erasure [40]. It takes the power of NEXTGEN [9], with it's first-class genericity, to define such a generic type.

JTL* expressions are also superior to the C++ template approach, because the requirements presented by the class (its *concept* of the parameter) are expressed explicitly. The lack of concept specification mechanism is an acknowledged limitation of the C++ template mechanism [173]. With the JTL* example above, in case the provided type argument does not include an appropriate constructor (i.e., does not match the concept), a straightforward error message is printed to `stderr`. This will be appreciated by anyone who had to cope with the error messages generated by C++ templates.

Due to type erasure, a JAVA class cannot specify a (generic) type parameter as its superclass. This prevents JAVA programmers from imitating mixins [39] via generics. JTL* does not suffer from this limitation, and can be used to achieve a mixin-like effect. Figure 4.9 shows an example that implements the classic mixin `Undo` [12].

As with previous examples, the code in Figure 4.9 explicitly specifies its expectations from the type argument—including not only a list of those members that must be included, but also a list of members that must *not* be included (to prevent accidental overriding [12]).

### 4.3.4 Translation

There is nothing inherent in JTL* that forces the generated output to be JAVA source code. Indeed, some of the most innovative uses generate non-JAVA textual output by applying JTL* programs to JAVA code.

A classic nonfunctional concern used in aspect-oriented systems is persistence, i.e., updating a class so that it can store instances of itself in a relational database, or load instances from it. In most modern systems (such as Hibernate[5] and JAVA EE v5[6]), the mapping between classes and tables is defined using annotations. For example, Figure 4.10 shows two classes, mapped to different tables, with a foreign key relationship between them.

In this simplified example, the annotation `@Table` marks a class as persistent, i.e., mapped to a database table. If the `name` element is not specified, the table name defaults to the class name.

[5] http://www.hibernate.org/

[6] http://java.sun.com/javaee/

**Figure 4.10** Two JAVA classes with annotations that details their persistence mapping.

```
@Table class Account {
  @Id @Column long id; // Primary key
  @Column float balance;
  @ForeignKey @Column Owner accountOwner;
}

@Table class Owner {
  @Id @Column long id;
  @NotNull @Column String firstName;
  @NotNull @Column String lastName;
}
```

Similarly, the annotation `@Column` marks a persisted field; the column name is the same as the field's name. The special annotation `@Id` is used to mark the primary-key column.

Given classes annotated in such a manner, we can use the `generateDDL` predicate (Figure 4.11) to generate SQL DDL (Data Definition Language) statements, which can then be used to create a matching database schema. Using the **`first`**, **`last`**, and **`between`** directives, this query generates a comma-separated list of items, one per field in the class, enclosed in parenthesis. The program also includes error checking, e.g., to detect fields with no matching SQL column type.

When applied to the two classes presented above, `generateDDL` creates the output shown in Figure 4.12.

In much the same way, JTL$^*$ can be used to generate an XML Schema or DTD specification, describing an XML file format that matches the structure of a given class.

## 4.4 Output Validation

The trust we can put in any code-generation mechanism can be increased by the assurance that it will always generate valid code in the target language. Let us assume that $p$ is a JTL$^*$ predicate that was designed to produce output in a language $\mathcal{L}$, where $\mathcal{L}$ can stand for concrete languages such as JAVA, XML, SQL, C++, etc. Then, we would like to automatically prove that the output of $p$ is a valid word in $\mathcal{L}$, where validity includes both *syntactical* and *semantical* aspects. In this section, we argue that doing that is impossible, but still, using JTL$^*$ can help in somewhat ameliorating this predicament.

Note that the impossibility claim is for general, arbitrary $\mathcal{L}$. The situation may be better for specific values of $\mathcal{L}$: still, as the literature demonstrates, guaranteeing correct output even for specific languages is still very difficult.

Compared to JTL, ASPECTJ and other aspect-oriented languages are safe in the sense that it is guaranteed that an application of an aspect to a valid JAVA program yields a syntactically correct program.

Deep inside, this safety is achieved by a "proof system" (so to speak) that automatically determines, for any predicates $a$ and $p$, whether $a$ follows from $p$, where $a$ denotes the demands and assumptions that an advice makes of the advised code, and $p$ the demands that the pointcut definition makes of the same code. Clearly, the difficulty of writing such a proof system increases with the expressive power of the languages in which $a$ and $p$ are written. In the case that $a$ and $p$ use the full power of first order predicate logic, then the problem becomes undecidable. Safety in ASPECTJ is achieved by minimizing the expressiveness of both the pointcut and the advice

**Figure 4.11** Predicates for generating SQL DDL statements for annotated persistent JAVA classes.

```
generateDDL := %class "CREATE TABLE " tableName %{
   first "("; last ")"; between ",";

   %[ @Column field !sqlType ] "Unsupported type, field #";
   columnName sqlType sqlConstraints;
%};

qlType := %String "VARCHAR"
   | %integral "INTEGER"
   | %real "FLOAT"
   | %boolean "ENUM('Y','N')"
   | %BigDecimal "DECIMAL(32,2)"
   | %Date "DATE"
   | foreignKey;

sqlConstraints :=
   [ %@NotNull "NOT NULL" | nothing ]
   [ %@Id "PRIMARY KEY" | nothing ]
   [ %@Unique "UNIQUE" | nothing ];

foreignKey := %[ field typed T ] ––target class
    "FOREIGN KEY REFERENCES" T tableName;

tableName := [ %@Table '?*' ] ––Table name = class name
   | %2 "Class # is not mapped to a DB table.";

columnName := [ %@Column '?*' ] ––Column name = field name
   | %2 "Field # is not mapped to a DB column.";
```

**Figure 4.12** The DDL statements generated by applying the `generateDDL` predicate (Figure 4.11) to the classes from Figure 4.10 (shown pretty-printed for easier reading).

```
CREATE TABLE Account (id INTEGER PRIMARY KEY,
   balance FLOAT,
   accountOwner FOREIGN KEY REFERENCES Owner);

CREATE TABLE Owner (id INTEGER PRIMARY KEY,
   firstName VARCHAR NOT NULL,
   lastName VARCHAR NOT NULL);
```

languages; complex situations, e.g., iteration over parameters of advised methods, are deferred to runtime and must be implemented by JAVA code (*cf.* the JTL aspect in Figure 4.7, which performs this iteration ahead-of-time).

Unlike ASPECTJ code, code generated by a JTL program is never executed directly; it must first be compiled by the target language's compiler. Thus, the lack of output syntax safety in JTL will never manifest itself at runtime.

Another point that mitigates the severity of the validation issue in JTL* is its declarative nature. In many cases, a human can infer the grammar, $G_p$, that describes the language of the output of a JTL* predicate, $p$, just by following the structure of the predicate. For example, the predicate `[public | protected] static` has two possible outputs, `public static` or `protected static`. Note that the grammar of this language is quite close to the originating predicate.

The ability to easily infer a grammar of a predicate, paves the way for the following method for validating a JTL* predicate: *for a specific target language, it may be possible to prove that the grammar of that language, $G_t$, contains the grammar of the originating predicate, that is: $G_p \subseteq G_t$.*

This method is based on the work of Minamide and Tozawa [140] who showed that it is possible (and practical) to decide for a given context-free grammar $G$, whether $L(G) \subseteq L(G_{\text{XML}})$, where $G_{\text{XML}}$ is the grammar of XML. Minamide and Tozawa used this result for checking that a PHP [126] program produces, as output, correct XHTML. This was achieved by inferring a conservative grammar of the output language from the code of the PHP program at hand.

## 4.5 Related Work and Discussion

The work on program transformations is predated to at least D. E. Knuth's call for "program-manipulation systems" in his famous "Structured programming with go to statements" paper [120]. Shortly afterwards, Balzer, Goldman and Wile [24] presented the concept of *transformational implementation*, where an abstract program specification is converted into an optimized, concrete program by successive transformations.

The JTL* system can be categorized using Wijngaarden and Visser's taxonomy of transformation systems [179], consisting of three dimensions:

1. *Scope* pertains to the extent of the portion of an object program covered by a single transformation step, which can range from a single instruction to an entire program. Most examples given here are **local-local** transformations, since both input and output do not consult global information. Still, it is possible to write programs with more global scope for either input or output. As a minor example, the SQL DDL generation program examines the annotations attached to classes other than the input class—as directed by the types of fields marked as foreign keys.

2. The *direction* of a transformation is either forward (source driven) or reverse (target driven). JTL* is primarily **source driven**, in that the input structure orchestrates the generation of output. The reverse translation mode is one in which, just as being done in the ASP and PHP languages, the output (normally HTML text in these two languages) is a template, with placeholders ready to be filled by functions of the input. As some of the examples indicate, reverse direction transformation is possible in JTL*, by embedding output predicates in string literals.

3. Different transformation engines use a different number of *stages*. Wijngaarden and Visser make the distinction between *single-stage*, *multi-stage modify* and *multi-stage generate*

techniques. JTL* applications are a **single-stage** approach, since the target is generated in one single traversal over the source. It is future research to evaluate the benefits of using JTL* in a multi-stage-generate approach, where every traversal generates a piece of output which is then merged to create the final output.

A more interesting direction for future research is the multi-stage-modify approach, by which the target is generated incrementally by making several traversals over the source, which corresponds to famous questions of aspect interferences, priority, etc.

In a sense, JTL* sides with the perspective by which aspects are thought of as transformations of a software base; aspect application is a transformation of the rephrasing kind, which also includes inlining, specialization, and refactoring. This perspective was presented earlier by Fradet and Südholt [78], whose work focused on "aspects which can be described as static, source-to-source program transformations". It was in fact one of the earliest attempts to answer the question, "what exactly *are* aspects?". Unlike JTL*, the framework presented by Fradet and Südholt utilizes AST-based transformations, thereby offering a richer set of possible join-points, enabling the manipulation of method internals.

Lämmel [122] also represents aspects as program transformations, whereas the developers of LOGICAJ [159] go as far as claiming that "the feature set of ASPECTJ can be completely mapped to a set of conditional program transformations".[7] LOGICAJ uses program transformations as a foundation for AOP, and in particular for extending standard AOP with generic aspects. More recently, Lopez-Herrejon et al. [128] developed an algebraic model that relates aspects to program transformations.

JTL* is not the first system to use logic-based program transformation for weaving aspects. Indeed, De Volder and D'Hondt's [64] coin the term *aspect-oriented logic meta programming* (AOLMP) to describe logic programs that reason about aspect declarations. The system they present is based on TYRUBA [63], a simplified variant of PROLOG with special devices for manipulating JAVA code. However, whereas JTL* presents an open-ended and untamed system for manipulating JAVA code, De Volder and D'Hondt's system presents a very orderly alternative, where output generated not by free-form strings but rather using quoted code blocks.

We therefore find that, compared to other AOP-by-transformation systems, JTL* is limited in the kind of transformations it can apply for weaving aspects, and in the level of reasoning about aspects that it provides—which is why we view it as a "quick-and-dirty" AOP language. The windfall, however, is that program transformation in JTL* is not limited to AOP alone, as evident from some of the examples provided in this paper—the generation of stub classes from interfaces, the generation of SQL DDL to match classes, the definition of generic classes, etc.

The ELIDE system for Explicit Programming [43] defines *modifiers* that are placed, somewhat like annotations, in JAVA code; programs associated with these modifiers can then change the JAVA code in various ways, including the generation of new methods and classes or the insertion of code before/after methods.

For example, in the following declaration: `private property<> String name;` the handler associated with the `property` marker will add accessor methods to the containing class. The markers can be parameterized, e.g., `property<read_only>` will cause only a getter method to be generated. By using queries that match standard Java annotations, JTL* transformations can be used to a similar effect.

With a pattern matching `@Property` as a guard, the transformation can create accessor methods. And since annotations can be parameterized, the equivalent of marker parameters can also be achieved.

[7] http://roots.iai.uni-bonn.de/research/tailor/aop

Unlike JTL*, ELIDE handlers use JAVA-like code to modify the base JAVA code; yet similarly to JTL*, ELIDE's code can include multi-line strings (enclosed in `%{ ... }%`) and has an "escape" syntax for quoting variables inside such strings.

ELIDE markers can be applied not only to class members, but to whole classes too. It is thus possible to mark a class with `allAccessors<>`, which generates accessor methods for all private data members. However, because ELIDE handlers are written in JAVA, the task of *finding* all such members and *iterating* over them is significantly more complex than in JTL*, where a guard such as `private field` is all that is needed.

The Stratego system [180] is a generic term rewriting tool. As such it is useful in a wider range of applications. By focusing on the domain of Java programs, JTL* sports a nicer and more intuitive syntax, thus making it more user friendly.

# Chapter 5

# Whiteoak

So far, we discussed the role that formal patterns play in capturing design knowledge (Chapter 3) and in code transformation mechanisms (Chapter 4). Such applications are often related to activities such as maintenance and refactoring (respectively) whose starting point is an existing codebase that needs to be understood and simplified before it can further grow. In this chapter we explore the utility of formal pattern in the activity that is "responsible" for the existence of such large codebases: *implementation*.

We do that by showing that formal patterns can be a first-class construct in an object-oriented programming language. Given that a type is a condition on the set of runtime values, we argue that *a condition on the set of types, that is: a type-level formal pattern, is a useful type in its own right*.

This idea is reified by WHITEOAK, a JAVA extension that allows the user to define type-level formal patterns and use them just like standard types. Subtyping of these new types is structural: compatibility between two types is determined by their structure and not by an explicit nominal indication, a-la JAVA's **`extends`** and **`implements`** keywords.

An interesting property of WHITEOAK is that every definition of pure a structural type—a structural type that does not provide default implementations for methods—is a valid JTL expression.

Structural subtyping addresses common software design problems, and promotes the development of loosely coupled modules. This additional flexibility is achieved without compromising JAVA's static type safety. Measurement indicate that the performance of our implementation of structural dispatching is comparable to that of the JVM's standard invocation mechanisms.

## 5.1 Introduction

A restaurant accounting library $A$ purchased from an American vendor expects parameters that conform to **`interface`** `Check`, but, the British maker of a large software module $B$ in charge of serving orders from the kitchen to customers, chose to produce objects which are instances of **`class`** `Bill`. Now, a `Bill` offers essentially the same set of services demanded by `Check`. How can components $A$ and $B$ be coerced to work together without modifying any one of them?

In programming languages which obey *nominal typing* rules, such retrofitting is not easy. Nominal typing dictates that two types are equivalent only if they have the same name. Accordingly, types `Check` and `Bill` are nominally unrelated; one must use techniques such as those offered by the ADAPTER design pattern [81] to make the necessary plumbing code.

How is retrofitting done in languages such as ML and HASKELL [113] where *structural typing* is the rule? Recall that structural typing means that two types are *equivalent* if they have the same structure. Also, structural subtyping follows from inclusion of structure. Thus, in these languages

compatibility can be achieved by the observation that `Bill` is a subtype of `Check` (or vice versa), or even by using the minimal super-type of the two types. The compatibility is therefore due to the overlap between the set of members of two types.

### 5.1.1 The Case for a Dual Nominal-Structural Typing

The failure of nominal typing system in situations demanding retroactive type abstraction is discussed in depth in the literature [26, 27, 44, 125, 131] making the case for using structural typing in mainstream languages. We argue further that the importance of retrofitting even increases with the advent of "*compile once run everywhere*" languages such as JAVA, RUBY [176] and PHP: The fact that hardware speed and memory constraints are not as stifling as they used to be, complemented by the huge body of open source modules accumulated in the world wide web, has led to the emergence of a new kind of hybrid programs [79] that contain numerous modules, written by many different programmers. Crucial to such architectures is the concept of *interoperability*—the ability to integrate modules written by independent authors. Dynamic type checking of RUBY and PHP may make interoperability in these easier—it is more difficult to achieve this goal while preserving the safety, clarity, and other benefits of static typing. To make interoperability possible in a statically typed environment, independent modules must agree on the *type* of exchanged data; and structural subtyping contributes to the ability of reaching such an agreement.

Another famous crucial issue in which nominal type systems fail is in interaction with external data, be it persistent (e.g., residing in relational or XML databases) or originating from a distributed computing environment (e.g., a query to a web service). Such data is described by its structure, and attaching globally agreed names to it is difficult. This is precisely the reason that languages designed for data interchange such as ASN.1 [148] are structurally typed. Difficulties in implementing conflicting types and class hierarchies is also mentioned in the literature [27] as a failure point of nominal typing systems.

On the other hand, it is also acknowledged that structural typing has its limitations, most notably, "accidental conformance" (discussed, together with some prospective solutions by Laüfer, Baumgartner and Russo [125]), and the difficulty of defining recursive types. Other pros of nominal type systems include (to use the words of Malayeri and Aldrich [131]): the fact that these systems support the "*explicit expression and enforcement of design intent, simplify declaration of recursive types, and make it possible to produce more comprehensible error messages*".

These, and the fact that dispatching is less efficient in structural type systems may explain the fact that nominal typing is the dominating scheme in (statically typed) mainstream languages designed for large projects, including JAVA, C++ and EIFFEL, while at the same time, the community seeks ways of combining these two typing paradigms so that their respective benefits can be used in tandem. The Unity language [131] is a recent example of a language design which combines the two concepts. The contribution of Unity is in formally showing a language model with a sound and complete type system which combines the two paradigms. However, Unity leaves open issues such as separate compilation, dynamic loading of classes and multiple inheritance of nominal types (a-la JAVA's/C$^{\#}$'s interfaces), and a host of problems arising in actual language implementation. The recent release of SCALA introduced structural typing into the language [153]. through the notion of *Refinement of Compound Types*. It appears though that the performance of this feature is still poor.

In their paper describing the Continuum project [103], Harrison Lievens and Walsh also arrive at the conclusion that nominal subtyping makes mainstream object-oriented languages inflexible. They even go further and claim that the standard dispatching mechanism, where a method is looked-up in the context of a single receiver object, creates an involuntary coupling between the client and the structure of the service provider.

Prior work on integrating structural typing with concrete, non-research languages, includes

*signatures*, a C++ language extension which combines structural typing with the existing nominative features of the languages [26, 27], *safe structural types for* JAVA [125] and the work of Büchi and Weck on *Compound Types* [44].

More generally, many aspects of the question of merging the two paradigms can be thought of as the problem of integrating such nominal languages with external, structurally typed data. Work on this dates back to the work of Schmidt on Pascal-R [165] going through the work of Andrews and Harris [14] in the context of C. (See also surveys in [20, 21]). Another related work is that of Jorgensen on Lasagne/J [114] who solved the retrofitting problem by means of language mechanisms that allow for automatic wrapping, but stays short of structural typing.

### 5.1.2 Overview

This chapter describes the design and implementation of WHITEOAK, a system that introduces structural type equivalence and structural subtyping into JAVA. This addition relies on a new keyword, **`struct`**, used to define structural types whose subtyping relation is determined solely by structure.

In a sense, **`struct`** types are similar to JAVA's interfaces, and in particular support "multiple inheritance", except that subtyping among **`struct`** types is, as expected, structural rather than nominal. Thus, the type `XYZ` defined by

```
struct XYZ { int x,y,z; }
```

is a subtype of `XY` defined by

```
struct XY { int x,y; }
```

and of the unnamed structural type

```
struct { int y,x; }
```

Names of structural type are optional, but once defined, they can be used as a shorthand for the full type declaration: The function definition

```
int innerProduct(XY a, XY b) {
  return a.x * b.x + a.y * b.y;
}
```

would be no different if it used the more verbose signature

```
int innerProduct(struct {int x, y;} a, struct {int y, x;} b) {
  return a.x * b.x + a.y * b.y;
}
```

Any nominal (**`class`** or **`interface`**) type that conforms to the protocol of a structural type can be upcast into that structural type: The method `innerProduct` can thus be applied to any conventional type which has public, non-final integer fields named `x` and `y`. A field access in the method, e.g., `a.x`, is mapped to the field declared by the referenced object's runtime type. Thus, fields declared by a structural type follow a late binding semantics, just as methods.

We saw that WHITEOAK provides for polymorphic abstraction and retrofitting over fields. The following WHITEOAK function demonstrates retrofitting over methods:

```
struct Source { int read(); }
void exhaust(Source s) {
  while (s.read() >= 0) ;
}
```

Function `exhaust` is applicable to objects of both class `Reader` (the superclass of classes capable of digesting Unicode input) and `InputStream` (the superclass of classes for processing byte-oriented input), despite the fact that the two classes are unrelated. Further, the function call

`s.read()` dispatches correctly in all cases, even though the dispatch target is not necessarily stored in the same location in the virtual functions table.

We saw that unlike interfaces, **`struct`** types allow, just like abstract classes, the declaration of fields. Another similarity to abstract classes is that **`struct`** types may define concrete method implementations: abstract methods and fields declared in a **`struct`** define the *requirements* that any conforming type must provide, where the concrete methods define *default behavior* which can be specialized by the conforming type.

Revisiting our opening dilemma, we can say that if `Check` was declared as a **`struct`** type rather than an **`interface`** type, then a `Check` variable can receive its value from `Bill` objects. But, even if this was not the case, the bridging code is simplified by using the most specific structural type to which both `Check` and `Bill` conform.

Unlike classes, **`struct`** types have no constructors, although they may pose a constraint on the protocol of the class constructors—a feature which naturally admits "virtual constructors" into the language. Also, unlike classes, fields cannot be initialized as part of their definition.

There are no direct means for defining **`struct`** literals, but anonymous classes provide a substitute. One can therefore invoke function `innerProduct` with two ad-hoc types and their values

```
int xmas = innerProduct(new Object() { int x = 5, y = 2; },
      new Object() { int x = 3, y = 5, a = 3;});
```

Conversely, we argue that user control over anonymous classes is enhanced with structural types.

Two more features should be mentioned at this stage: (*i*) support for dynamic run-time subtyping tests and downcasts even against structural types; and (*ii*) operators for the *intersection*, commutative- and non-commutative- (i.e., overriding) *union* of structural types.

The implementation of WHITEOAK comprises two components:

- A modified JAVA compiler (based on Sun's JAVA 5 compiler). This compiler generates standard bytecode, and conforming `.class` files.
- A small runtime library of functions realizing the dynamic dispatching semantics of WHITEOAK.

Unlike many research languages, WHITEOAK's design had to deal with the issue of integrating with and supporting existing language features (including genericity, reflection, annotations, dynamic loading of classes, etc.), as well as preserving the semantics of existing, previously compiled code.

A primary challenge that WHITEOAK faced was that of an efficient realization of the structural typing addition on top of the standard Java Virtual Machine [127] (JVM), which is nothing else than a *nominally* (and strongly typed) machine model. In this respect, our work was more difficult than that of e.g., Baumgartner and Russo classical implementation of signatures in C++, in which both translation to untyped assembly and the use of advertised loopholes in the type system of C++ could be used to support structural typing.

## 5.2 The WHITEOAK Language

Backward compatibility, efficiency, simplicity, uniformity and expressive power where principal guidelines in the design of WHITEOAK. This section describes the main language design alternatives encountered, explains the decisions we took in light of these causes, and elaborates on the challenges that these decisions entailed. The section concludes with a detailed comparison of WHITEOAK's realization of structural types within a nominative type system with previous efforts of this sort.

### 5.2.1 Definition of Structural Types

A structural type can be used anywhere a nominal type can be used, including variable-, parameter-, return type-, and field- definition just as in constraints on type parameters to generics. Structural types cannot be used in the throw-list of exceptions, since all exceptions must inherit from the nominal library type `Throwable`. We also stayed short of allowing generic structural types.

A structural type may be defined in place, or refer to a named structural type definition. Such named definitions may be made in any location a non-anonymous JAVA class can be defined, including the outer package scope, in a class, or in a function.

Which member kinds are allowed in structural types? Figure 5.1 demonstrates WHITEOAK's answer.

**Figure 5.1** Structural type `ErrorItem` demonstrating the variety of member kinds allowed in WHITEOAK

```
1 struct ErrorItem {
2
3   // bodiless method:
4   int severity();
5
6   // a field:
7   String description;
8
9   // a requirement on a field (read−only access)
10  final int lineNumber;
11
12  // a method with default implementation:
13  String where() {
14    return lineNumber + ": " + description;
15  }
16
17  // constraints on constructors:
18  constructor(int l);
19  constructor(String d, int l);
20 }
```

In the figure we see that **`struct`** types may have bodiless functions (Line 4), data members (Line 7) which may even be **`final`** (Line 10), functions with body (Line 13), and constructor specification (Line 18). WHITEOAK does not allow **`static`** members[1], initialized data members, or constructors with a body.

Bodiless (abstract) methods, representing a constraint on the actual type, are obviously essential.

*Data members* were added for uniformity and in support of interaction with external databases; The mechanisms for supporting these are no different than those required for bodiless functions.

*Constructor constraints* were added, again for uniformity, but also in support of virtual constructors and more expressive generics. For example, given the above definition of `ErrorItem`, if one needs a function that takes an existing `ErrorItem` object and returns a new one that differs only in the line number, one may write:

[1] Still, we shall see that a field or method declaration in **`struct`** can be realized by a **`static`** field or method of a nominal type.

```
ErrorItem bump(ErrorItem e, int diff) {
  return e.constructor(e.description, e.lineNumber + diff);
}
```

Note that constructor specifications are always bodiless: a structural type cannot provide a default implementation for a constructor.

*Static members.* WHITEOAK does not allow structural types to define **`static`** members. In particular, we cannot make the demand that a function or a field is implemented as **`static`**. Still, a **`static`** member in a nominal type is allowed to realize a **`struct`** member specification, thus allowing uniform treatment of static and non-static features.

For example, an instance of class `A` defined by

```
class A {
   public static void f() {}
   public static int d;
}
```

may be assigned to a **`struct`** requiring a **`void`** function `f` and an **`int`** data member `d`:

```
struct {
   void f();
   int d;
} a = new A();
```

and `a.f()` will be bound dynamically to `A.f` while the member reference `a.d` will be dynamically delegated to the **`static`** data member `A.d`.

*Virtual Objects* allow client code to obtain an object reference that offers a different set of methods than the actual (referenced) object. This is achieved by assigning an object into a variable of structural type `S`, where `S` defines *non-abstract methods*. Each such method provides a default implementation that will be invoked if the actual object does not provide its own implementation for that method. This is demonstrated by Figure 5.2.

**Figure 5.2** A structural type with a default function implementation.

```
1 struct LineReader {
2
3   int read() throws Exception;
4
5   String readLine() throws Exception {
6     String s = "";
7     for (int c = read(); c >= 0 && c != '\n'; c = read())
8       s += (char) c;
9     return s;
10  }
11 }
12
13 void f(Reader r) throws Exception {
14  LineReader lr = r;
15  System.out.println(lr.readLine());
16 }
```

In the figure we see the structural type `LineReader`. This type requires an **`int`** `read()` method and provides a `String readLine()` service based on this method. Assigning an instance of any class with an appropriate `read` function to a variable whose type is `LineReader`

will effectively attach the implementation of this function to the object, as demonstrated in function `f` in the figure: The assignment in Line 14 in this function is legal, since the class `Reader` declares the method `int read()`.[2]

If function `f` is invoked with an object whose dynamic type is `FileReader` (a subclass of `Reader` that does not offer a `readLine()` method) then the `readLine()` call in Line 15 is dynamically bound to the default implementation found in `LineReader`. In contrast, if the function is invoked with an instance of a class such as `BufferedReader`, which happens to implement this function, then Line 15 is bound to the object's own implementation.

Summarizing this program we note that the virtual object reference, `lr`, provides its own static protocol and its own run-time behavior which differ from the ones provided by the referenced object, `r`. This means that in WHITEOAK protocol and behavior can be associated with *references* (i.e.: variables) and not just with *objects*. This provides the means for achieving better localization of concerns: there is no need to define the full behavior of the object at the class definition point. If a certain concern is needed only in a certain part of the program, we can package the relevant code as a virtual object and use it only when needed.

One reservation applies: a structural type in which a certain function is unimplemented cannot be assigned from a structural type which offers a default implementation for this function. Hence, the following fails to compile

```
Reader r = ...;
LineReader lr = r;
struct { String readLine(); } x;
x = lr; // Compilation error!
```

In the assignment `x = lr` we obtain a new reference, `x`, from an existing reference `lr`. Recalling that the implementation of `readLine()` is associated with the reference `lr` and not with the actual object we note that there is no assurance that the actual object will contain an implementation for `readLine()`. Therefore, the assignment `x = lr` is ill-typed and is rejected by the compiler. Specifically, in an assignment from a virtual object, non-abstract methods are treated as if they were abstract.

WHITEOAK also supports *recursive structural types* as demonstrated by Figure 5.3,

The `List` type in the figure is (self) recursive in a covariant position: the return type of `List.tail()` is `List` itself. This allows `MutableList` to be a structural subtype of `List`. Examining `MutableList` we see that it is (self) recursive in a *contra-variant* position: the second `tail` method, `MutableList.tail(MutableList)`, defines a parameter of type `MutableList`. Recursion in a contra-variant position prohibits subtyping, so the type `ReversableMutableList` is not a subtype of `MutableList`. The formal criteria for subtyping of recursive types are realized in the subtyping algorithm presented in Algorithm 1.

### 5.2.2 Composition

This part examines the composition techniques available in WHITEOAK. As we shall see shortly, the combination of these composition techniques along with the ability to define virtual objects (that is: structural types with non-abstract methods), allows the WHITEOAK programmer to attach units of behavior to existing objects. This allows greater flexibility than traditional, statically-typed, code reuse mechanisms—inheritance, mixins and traits—which manipulate classes but not objects.

WHITEOAK offers three type composition operators: If $T_1$ and $T_2$ are structural types,

[2] The fact that `Reader.read()` is an abstract method is not a problem. The JAVA language semantics forbidding instantiation of abstract classes guarantees that the dynamic type of variable `r` will offer a concrete implementation for all its methods.

**Figure 5.3** Recursive structural types. `MutableList` and `ReversableMutableList` are subtypes of `List`. `ReversableMutableList` is not a subtype of `MutableList`.

```
struct List {
  int head();
  List tail();
}

struct MutableList {
  int head();
  MutableList tail();
  void tail(MutableList t);
}

struct ReversableMutableList {
  ReversableMutableList reverse();
  int head();
  ReversableMutableList tail();
  void tail(ReversableMutableList t);
}
```

then $T_1$`*`$T_2$ is the type obtained by the *intersection* of the set of members defined in $T_1$ and $T_2$, $T_1$`+`$T_2$ is the *commutative union* of these sets, and $T_1$ $T_2$ (type $T_1$ concatenated with type $T_2$) is the *overriding union* of these sets. If one of $T_1$ and $T_2$ is nominal, then it is cast to a corresponding structural type prior to the operation. In order to explain the semantics of these let us first make the following definitions.

**Definition 10** *A method in $T_1$* conflicts *with a method in $T_2$ if both have the same name and the same arguments, but a different return type or a different exception specification.*

**Definition 11** *A constructor in $T_1$* conflicts *with a constructor in $T_2$ if both have the same arguments, but a different exception specification.*

**Definition 12** *A field in $T_1$* conflicts *with a field in $T_2$ if both have the same name, but different type, or* **`final`** *specification.*

**Definition 13** *Two non-conflicting members $m_1, m_2$ are* synonymous *if either of the following condition holds*

- *$m_1$ and $m_2$ are methods with the same name, signature, return type, and exception specification.*
- *$m_1$ and $m_2$ are constructors with the same signature and exception specification.*
- *$m_1$ and $m_2$ are both fields of the same name, type and* **`final`** *specification*

When either of the composition operators are used with operands $T_1$ and $T_2$, conflicts between members of $T_1$ and $T_2$ are reported as errors.

**Intersection.** Type $T_1$`*`$T_2$ is obtained by taking an element of each synonymous pair of $T_1$ and $T_2$. This element is included only once in the result. The resulting type is actually the most specific supertype that is common to both $T_1$ and $T_2$.

For a pair of synonymous methods $m_1 \in T_1$, $m_2 \in T_2$, the method included in the result will have the same body as $m_1$ ($m_2$) if $m_1$ ($m_2$) is non-abstract, and $m_2$ ($m_1$) is abstract. In the other cases—$m_1$ and $m_2$ are both abstract, $m_1$ and $m_2$ are both non-abstract—the method in the result will be abstract.

**Commutative Union (Traits).** The "+" operator can be used to compose a structural type from smaller building blocks. For instance, in writing

```
struct Input {
  int read() throws Exception;
  int read(char[] a) throws Exception;
}
struct ImprovedInput = Input + LineReader;
```

we define a new structural type whose specification is the union of `LineReader` and `Input`. More formally, the result of $T_1$+$T_2$ is a structural type that contains these members:

- All members of $T_1$ that do not have a synonymous member in $T_2$.
- All members of $T_2$ that do not have a synonymous member in $T_1$.
- All members of $T_1$*$T_2$

This semantics resembles that of *Traits* that were proposed [164] as a code reuse mechanism. In a trait-enabled language, a class definition can use one or more traits as building blocks that supply part (or even all) of its behavior. Just like WHITEOAK's structural types, traits provide behavior but they cannot carry any state. If two or more traits (used by a single class) offer an implementation for the same method, that method becomes abstract in the class.

Figure 5.4 compares composition of traits with commutative union composition.

Figure 5.4(a) shows a standard example for traits using the syntax offered by Hill, Quitslund and Black [146]. Specifically, class `RedCircle` is built from traits `TRed` and `TCircle`, where the former contributes the `diameter()` method and the latter contributes the `color()` method. The resulting class, `RedCircle`, is concrete since it provides an implementation for the remaining, abstract, method(s) of these traits, namely: `radius()`.

Figure 5.4(b) shows the equivalent WHITEOAK code. Structural type `RedCircle` uses the "+" operator, to compose the `TCircle` and `TRed` structural types. This composition yields a type with two implemented methods, `diameter()` and `color()`, and one abstract method: `radius()`. We then assign an instance of an anonymous class, that implements all the abstract methods, into a variable of type `RedCircle`. The resulting variable can respond to any of those three methods.

**Non-Commutative Union (Mixins).** Given two structural types, $T_1$, $T_2$, their non-commutative union, denoted by concatenation: $T_1T_2$, is a structural type that is similar to $T_1$+$T_2$ except for its handling of non-abstract methods. Specifically, the result of $T_1T_2$ is a structural type that contains these members:

- All members of $T_1$ that do not have a synonymous member in $T_2$.
- All members of $T_2$ that do not have a synonymous member in $T_1$.
- $m_1$ from each pair of synonymous members $m_1 \in T_1$, $m_2 \in T_2$ where $m_1$ is non-abstract and $m_2$ is abstract.
- $m_2$ from each pair of synonymous members $m_1 \in T_1$, $m_2 \in T_2$ where $m_1$ is abstract or $m_2$ is non-abstract.

**Figure 5.4** (a) A JAVA with traits [146] code realizing a Red Circle class, and (b) corresponding WHITEOAK code.

```
abstract class TCircle {
  abstract double radius();
  double diameter() return 2*radius();
}

abstract class TRed {
  Color color() { return Color.RED; }
}

class RedCircle uses TCircle, TRed {
  double radius() { return 3.0; }
}
```

(a) JAVA with traits [146] code realizing a Red Circle class via the composition of traits `TCircle`, `TRed`.

```
struct TCircle {
  double radius();
  double diameter() { return 2*radius(); }
}

struct TRed {
  Color color() { return Color.RED; }
}

struct RedCircle = TCircle + TRed;

RedCircle rc = new Object() {
  double radius() { return 3.0; }
};
```

(b) WHITEOAK code realizing a Red Circle object via the commutative union of `TCircle`, `TRed`.

This semantics resembles that of *Mixins* [38]. Just like traits, mixin composition allows classes to be composed from smaller building blocks. Unlike traits, a mixin composition is non-commutative, i.e.: the order of the composition is important. This order imposes a natural overriding relation on the defined methods.

Figure 5.5 shows how non-commutative union can be used to achieve mixin-like semantics.

The composition `MCircle MRed` yields a type with just one abstract method, `radius()`. Therefore, any conforming nominal type needs to specify only this method. A call to the `diameter()` method on the `cr` variable will dispatch the (only) implementation from `MCircle`. A `cr.name()` call will dispatch the implementation from `MRed`, since the methods of the second operand override the methods of the first operand.

### 5.2.3 Grammar

Figure 5.6 gives a grammatical specification of the four kinds of members allowed in structural types. The productions in the figure rely on nonterminals such as *Identifier* and *FormalParameterList* defined elsewhere in JAVA's grammar [92].

Examining the figure we see that structural types may not be generic (generic recursive struc-

**Figure 5.5** A mixin composition of the `Circle`, `Red` classes. The methods of the second operand, `Red` override those of the first one.

```
struct MCircle {
  abstract double radius();
  double diameter() { return 2*radius(); }
  String name() { return "Circle"; }
}

struct MRed {
  Color color() { return Color.RED; }
  String name() { return "Red"; }
}

struct CircleRed = MCircle MRed;
CircleRed cr = new Object() {
  double radius() { return 3.0; }
};

System.out.println(cr.name()); // Output is: "Red"
```

tural types are known to be a difficult and elusive problem; see e.g., Hosoya et al. [107]). Also note that with the exception of **`final`** for field declarations, no modifiers are allowed. All members of a structural type are implicitly **`public`**, and as discussed below, they can be realized by both **`static`** and non-**`static`** implementations.

Figure 5.7 defines how compound structural types are created, and how these types combine with the rest of JAVA.

The first production in the figure augments JAVA's grammar by stating that a structural type (nonterminal *StructType*) can be used *anywhere* a reference type can be used. This includes e.g., arguments to generics, bounds on such arguments, etc. The second production augments JAVA's grammar by admitting every **`struct`** declaration as a type declaration.

We then state that a *StructType* is either an identifier of a named structural type, defined by a *StructDeclaration*, or an *UnnamedType* which allows a direct use of a structural type expression. Such expressions are composed by applying the three structural type operators, (*i*) union, specified by operator `+` (lowest priority), (*ii*) intersection (operator `*`), and (*iii*) concatenation, whose

**Figure 5.6** Grammar specification of the body of structural types.

*StructBody: StructMember*$_{opt}$ | *StructMember StructBody*
*StructMember:* *MethodDeclaration* `;`
| *MethodDefinition*
| *ConstructorDeclaration* `;`
| *FieldDeclaration* `;`
*MethodDeclaration: Type Identifier* `(` *FormalParameterList*$_{opt}$ `)` *Throws*$_{opt}$
*MethodDefinition: MethodDeclaration MethodBody*
*ConstructorDeclaration:* **`constructor`** `(` *FormalParameterList*$_{opt}$ `)` *Throws*$_{opt}$
*FieldDeclaration :* **`final`**$_{opt}$ *Type Identifiers*
*Identifiers: Identifier* | *Identifiers* `,` *Identifier*

**Figure 5.7** Grammar specification of the uses of structural types

*ReferenceType:* $\cdots \mid \cdots \mid$ *StructType*
*TypeDeclaration:* $\cdots \mid \cdots \mid$ *StructDeclaration*
*StructType: StructId* | *UnnamedType*
*StructId: Identifier*
*UnnamedType:* **`struct`** *StructUnion*
*StructDeclaration:* **`struct`** *StructId* { *StructBody* } | **`struct`** *StructId* = *StructUnion* **`;`**
*StructUnion: StructIntersection* | *StructIntersection* + *StructUnion*
*StructIntersection: StructConcatenation* | *StructConcatenation* **`*`** *StructIntersection*
*StructConcatenation: StructTerm* | *StructTerm StructConcatenation*
*StructTerm:* **`(`** *StructUnion* **`)`** | *StructAtom*
*StructAtom:* { *StructBody* } | *ReferenceType*

semantics is reminiscent of inheritance with overriding (highest priority) to combine *StructAtom*s, i.e., atomic structural types.

Named types are more than shorthand for an unnamed type declaration; they make it possible to define recursive structural types. Also note that the production

*StructDeclaration:* **`struct`** *StructId* = *StructUnion* **`;`**

states that names can be also given to structural type expressions.

Nonterminal *StructAtom* is in turn either a *StructBody* enclosed inside a pair of curly braces, or a *ReferenceType*. The semantics of this production should be clear if this *ReferenceType* happens to be a structural type. However, if the reference type is a nominal type, the derivation effectively computes the *structural equivalent* of the nominal type, defined as the set of all **`public`** method declarations and all **`public`** fields declarations. In computing this set, all modifiers except for **`final`** data member modifiers, method bodies, initialization expressions of data members, just as annotations are eliminated.

### 5.2.4 Type Checking Algorithm

This section presents the algorithm used by the WHITEOAK compiler for determining whether one type is a subtype of another. This algorithm is used by the type checker to verify that the type of the right-hand side value in an assignment is compatible with the type of the left-hand side variable.

The algorithm is based on the algorithm of Amadio and Cardelli [11]. The presentation uses the following notations and symbols.

- $x = y$ indicates the trivial type equality relation, that is: $x$ and $y$ are the same type. We naturally extend this notation for denoting equality of sets of types and of sequences of types.
- $x \preceq y$ indicates WHITEOAK's type compatibility relation: $x$ is a subtype of $y$.
- As a convention we use the variable $r$, "required", to denote a candidate supertype (or a member thereof); we use the variable $f$, "found", to denote a candidate subtype (or a member thereof).

The algorithm for computing $x \preceq y$ is presented below (Algorithm 1). Following it are auxiliary procedures that are called (either directly or indirectly) from Algorithm 1.

Note that these algorithms are inherently recursive: In order to determine a type compatibility question we need to determine member compatibility questions, which in turn rely on the results of further type compatibility questions. A cache (from a pair of types to a Boolean value) is used to break this otherwise infinite recursion.

---

**Function:** $IsAssignable(f, r)$

---

**Input:** $f, r$ types (either structural or nominal)
**Output:** $True$ if $f \preceq r$, $False$ otherwise

```
1:  if f = r then
2:      return True
3:  if r is a nominal type then
4:      return IsNominallyAssignable(f, r)
5:  if f is a non-anonymous JAVA class then
6:      if f's visibility is not public then
7:          return False
8:  if cache[f, r] is initialized then
9:      return cache[f, r]
10: cache[f, r] ← True
11: cache[f, r] ← MemberSetTest(f, r)
12: return cache[f, r]
```

Algorithm 1: Testing that $f \preceq r$: $f$ is compatible to $r$. Function $IsNominallyAssignable(f, r)$ realizes JAVA's standard nominal subtyping test. Function $MemberSetTest(f, r)$ checks that every member of $r$ has a compatible member in $f$. $cache[x, y]$ is a cache slot that holds the result of $x \preceq y$. Initially all cache slots are uninitialized.

Examining Algorithm 1 we see that in step 3 the algorithm falls back to Java's standard (nominal) typing scheme if the candidate supertype, $r$, is a nominal type.

Steps 5—7 ensure proper visibility of the candidate subtype. Access to public classes is allowed. Access to other classes is allowed only if they are anonymous. This poses no security risks: During compilation, the only expressions that carry an anonymous-class type are the anonymous-class instantiation expressions. Therefore, $f$ will be an anonymous class only if the developer deliberately assigns such an instantiation expression directly into a structurally-typed variable.

Step 8 breaks the recursion: if the result for the current type-compatibility question is already cached, we return that result.

The last part of the algorithm, realizes the actual computation of the structural compatibility question. We then (Step 10) cache a (tentative) $True$ answer for the $f \preceq r$ question.

In (11) the auxiliary function $MemberSetTest()$ is called. This function will return $False$ if there is a member of $r$ that has no compatible member in $f$. Hence, we assign the result returned by this function to the appropriate cache slot and then return this result (12).

The algorithm that realizes function $MemberSetTest()$ is depicted in Algorithm 2.

The algorithm goes over every member of the $r$ ("required") type, finds the compatible members from $f$ ("found"), and stores these in the set $s$ (5) If no such member exists (6) the algorithm returns a $False$ answer. On the other hand, if such members were found the algorithms makes sure (8) that there is one member in $s$ which is more specialized than all other members in $s$. This check (which is essentially identical to Java's standard check for ambiguity due to overloading) ensures that every member of $r$ is unambiguously mapped to a member of $f$. Figure 5.8 shows why this check is essential.

class A from Figure 5.8 defines two overloaded methods. If A were compatible with S then a

**Function:** $MemberSetTest(f, r)$

**Input:** $f, r$ types
**Output:** $True$ if every member of $r$ has a compatible member in $f$.

```
1:  for all m_r member of r do
2:      s ← φ
3:      for all m_f member of f do
4:          if MemberPairTest(m_f, m_r, r) then
5:              s ← s ∪ {m_f}
6:      if s = φ then
7:          return False
8:      if s has overloading conflicts then
9:          return False
10: return True
```

Algorithm 2: Testing that every member of $r$ has exactly one compatible member in $f$. Function $MemberPairTest(x, y)$ checks that $x$ can replace $y$.

**Figure 5.8** Class A is not compatible with S due to overloading ambiguity.

```
public struct S {
   public void m(String o, Integer n);
}

public class A {
   public void m(Object o, Integer n) { ... }
   public void m(String o, Object n) { ... }
}
```

statically legal call such as `m("", 0)`, on a receiver of type `S`, will have no reasonable runtime behavior. It can be dynamically bounded to either of the two methods of `A`, with no method being a better match than the other. The check at step 8 prevents these situations.

The details of function $MemberPairTest()$ , called from step 4 of Algorithm 2, are presented in Algorithm 3.

The body of Algorithm 3 is straightforward: it delegates to one of three other functions, each handling a different set of inputs (a pair of constructors, a pair of methods, or a pair of fields). The function for deciding the compatibility of constructors, is presented in Algorithm 4.

The first check performed by Algorithm 4 ensures that $f$, the type of the object created by the candidate constructor $m_f$, is compatible with the required type, $r$, that initiated this constructor compatibility test. This check (Step 2) is carried out by a issuing a (recursive) subtyping test.

The algorithm then (4—8) verifies that $f$ is instantiable. In particular, constructors of non-**`static`** inner classes are rejected due to their extra, implicit, parameter. Finally, we require no-variance of the parameters (9) and covariance or no-variance of the throws clauses (11).

Note that the type compatibility decision issued by step 11 is essentially a nominal subtyping test: the candidate super-type is (per the JAVA language specification) a subclass of `Throwable` which is a nominal type. Therefore it is safe to use JAVA's standard test for conformance of **`throws`** clauses, $ThrowsClauseTest()$ .

One may think that checking that $f \preceq r$ at step 2 is redundant:

> $f$, the type declaring $m_f$, is the same as $f$ in Algorithm 1, Step 11, which indirectly invokes $ConstructorCompatibilityTest$. Prior to that call a $True$ answer is tenta-

**Function:** $MemberPairTest(m_f, m_r, r)$

**Input:** $m_f$, $m_r$ members; $r$ type
**Output:** $True$ if $m_f$ can replace $m_r$, $False$ otherwise
1: **if** $m_f$'s visibility is not **`public`** **then**
2: **return** $False$
3: **if** both $m_f$ and $m_r$ are constructors **then**
4: **return** $ConstructorCompatibilityTest(m_f, m_r, r)$
5: **if** the names of $m_f$ and $m_r$ are not identical **then**
6: **return** $False$
7: **if** both $m_f$ and $m_r$ are methods **then**
8: **return** $MethodCompatibilityTest(m_f, m_r)$
9: **if** both $m_f$ and $m_r$ are fields **then**
10: **return** $FieldCompatibilityTest(m_f, m_r)$
11: **return** $False$

Algorithm 3: Testing the compatibility of two members $m_f$, $m_r$, where $f$ is the originating required type. The three auxiliary functions: $ConstructorCompatibilityTest()$ (see Algorithm 4 below), $MethodCompatibilityTest()$ (Algorithm 5) and $FieldCompatibilityTest()$ (Algorithm 6) determine the compatibility of a pair of constructors, methods and fields (respectively).

tively cached for the $f \preceq r$ question. Therefore when Algorithm 4 checks if $f \preceq r$ it is bound to get a positive answer.

This (incorrect) reasoning overlooks the following point: the enumeration of all members of $f$ (Algorithm 2 Step 3) includes inherited members. In particular, it includes constructors of superclasses. When such a constructor is examined by Algorithm 4 its declaring type is not the same as $f$ in Algorithm 1 thereby making it possible for the $f \preceq r$ question, in Algorithm 4, to yield $False$ .

The second algorithm which Algorithm 3 relies on is Algorithm 5, which determines compat-

**Function:** $ConstructorCompatibilityTest(m_f, m_r, r)$

**Input:** $m_f$, $m_r$ constructors; $r$ type
**Output:** $True$ if $m_f$ can replace $m_r$, $False$ otherwise
1: $f \leftarrow m_f$'s declaring type
2: **if** $\neg(f \preceq r)$ **then**
3: **return** $False$
4: **if** $f$ is a nominal type **then**
5: **if** $f$ is **`abstract`** **then**
6: **return** $False$
7: **if** $f$ is a non-**`static`** inner class **then**
8: **return** $False$
9: **if** $Params(m_f) \neq Params(m_r)$ **then**
10: **return** $False$
11: **return** $ThrowsClauseTest(m_f, m_r)$

Algorithm 4: Testing the compatibility of two constructors. Function $Params()$ returns the sequence of the types of the formal parameters of its operand. Function $ThrowsClauseTest()$ realizes JAVA's standard (nominal) test for conformance of the **`throws`** clauses of its operands.

ibility of methods.

---

**Function:** $MethodCompatibilityTest(m_f, m_r)$

---

**Input:** $m_f$, $m_r$ methods
**Output:** $True$ if $m_f$ can replace $m_r$, $False$ otherwise
1: **if** $\neg(Type(m_f) \preceq Type(m_r))$ **then**
2: **return** $False$
3: **if** $Params(m_f) \neq Params(m_r)$ **then**
4: **return** $False$
5: **return** $ThrowsClauseTest(m_f, m_r)$

Algorithm 5: Testing the compatibility of two methods. Function $Type()$ returns the return type of its operand. Function $Params()$ returns the sequence of the types of the formal parameters of its operand. Function $ThrowsClauseTest()$ realizes JAVA's standard (nominal) test for conformance of the `throws` clauses of its operands.

Looking at Algorithm 5 we see that compatibility of methods is established if we have: no-variance or co-variance of the return type (Step 1); no-variance of the parameters (3); no-variance or co-variance of the exceptions declared in the throws clauses (5).

Finally, the compatibility of a pair of fields is determined by Algorithm 6.

---

**Function:** $FieldCompatibilityTest(m_f, m_r)$

---

**Input:** $m_f$, $m_r$ fields
**Output:** $True$ if $m_f$ can replace $m_r$, $False$ otherwise
1: **if** $m_r$ is a read-only field **then**
2: **return** $Type(m_f) \preceq Type(m_r)$
3: **if** $m_f$ is a read-only field **then**
4: **return** $False$
5: **return** $Type(m_f) = Type(m_r)$

Algorithm 6: Testing the compatibility of two fields. Function $Type()$ returns the type of its operand.

Checking Algorithm 6 we see that if the required field, $m_r$, is a read-only field, it can be matched with either a read-only or a mutable field, with a possibly co-variant type (Step 2). On the other hand, if the required field is mutable, the matched field must be mutable (3), and of the exact same type (5).

Having presented WHITEOAK's type compatibility algorithm we can discuss a few implications. The first point to note is that of constructor calls on a receiver typed with a type parameter. This situation is depicted by Figure 5.9.

The structural type `U` from Figure 5.9 prescribes a zero-argument constructor and a void method `m1()`. Class `N1` is trivially compatible with `U`. While class `N2` does not define a zero argument constructor, its superclass, `N1`, does. Therefore, we can get a fresh `U`-compatible object from an instance of `N2` by invoking the constructor of the superclass, `N1`. Therefore, both `N1` and `N2` should be subtypes of the structural type `U`.

This means that a constructor call on a structural type, $X$, is not guaranteed to return an object that is of the same dynamic type as the receiver. Instead, it makes the weaker guarantee that the returned object's dynamic type is a subtype of $X$.

To see the implication of this consider the Line 24 in Figure 5.9. `N2` is a subtype of `U` and thus

**Figure 5.9** Constructor calls on a type parameter bounded by a structural type, are typed by the upper bound.

```
1 struct U {
2   U constructor();
3   void m1();
4 }
5
6 public class N1 {
7   public N1() { }
8   public void m1() { }
9 }
10
11 public class N2 extends N1 {
12   public N2(int n) { }
13   public void m2() { }
14 }
15
16 static<T extends U>
17 T f(T t) {
18   T result = t.constructor(); // Compiler error!!
19   result.m1();
20   return result;
21 }
22
23 static void g() {
24   N2 n2 = f(new N2());
25   n2.m2();
26 }
```

it meets the constraints (Line 16) imposed on type parameter `T` of method `f()`.

In this instantiation of the generic method, `f()`, type parameter `T` is bound to `N2`, and thus the return value of the method, `T`, can be assigned to a variable of type `N2`. Therefore, Line 24 is well-typed.

Inside the method `f()`, we see that the return value is obtained (Line 18) by issuing a constructor call on the parameter `t` whose type is specified by the type parameter `T`. This call is also type correct, since type erasure replaces `T` with its upper-bound, `U`, which defines such a constructor.

However, as explained above, the object returned by a constructor call is guaranteed to be as specific as the static type but not a as specific as the dynamic type. Thus, the object assigned into `result` is as specific as `U`, but not necessarily as specific as `T`. Specifically, in the instantiation depicted by the figure, `result` will be assigned with an instance of `N1` which is not as specific as `T` (i.e., `N2`).

Therefore, constructor calls on a type parameter (bounded by a structural type) are typed by the WHITEOAK compiler by the upper-bound and not by the type parameter itself[3]. This makes the compiler reject the assignment on Line 18.

If such calls were legal (i.e., typed by the structural type) then the object returned from `f()`, which is an instance of `N1`, would have been assigned into the variable `n2` in Line 24 leading to an inevitable runtime error at the following line.

[3]This resembles the special handling of `getClass()` calls in the standard JAVA compiler [92, Sec. 4.3.2].

Another subtle issue is related to the recursive nature of the algorithms presented here. These algorithms are recursive in the sense that in order to check that one type is a subtype of another, one needs to check subtyping relationship of individual methods. Method subtyping is defined (as usual in JAVA) by the demand that the return type changes co-variantly, and the argument types are the same. The recursive call may therefore yield more subtyping problems that need to be decided. We deviate from the usual implementation of Amadio and Cardelli's algorithm [11] in that if, in these generated problems, the candidate supertype is nominal, we apply JAVA's nominal type testing algorithm. Also, the subtyping test invariably fails if the candidate subtype is structural and the candidate supertype is nominal. Recursion proceeds only if the candidate supertype is structural.

This ensures that a method that declares a parameter of a nominal type can safely assume that the dynamic type of this parameter will be a nominal subtype. Otherwise, every invocation of a method, on a parameter, had to be treated as a structural invocation. Even more seriously, the method's code may use reflection in ways that will break if the dynamic type is not a nominal subtype (e.g.: a parameter of type `Serializable` is sent to a serializing data stream).

### 5.2.5 Comparison with Related Work

Tab. 5.1 shows a feature based comparison of WHITEOAK with some of the main work on introducing structural typing into nominative languages, or for producing synergetic language, including C++ Signatures [27], Brew [125], Compound [44], Unity [131] and Scala.

| | | Signatures [27] | Brew [125] | Compound [44] | Unity [131] | Scala | Whiteoak |
|---|---|---|---|---|---|---|---|
| Member Kinds | Method declaration | + | + | | + | + | + |
| | Method definition | + | | | + | | + |
| | Mutable Field declaration | | | | + | + | + |
| | Readonly Field declaration | + | | | + | + | + |
| | Field definition | + | | | | | |
| | Constructor declaration | | | | | | + |
| | Constructor definition | | | | | | |
| Capabilities | Parametrization | + | | | | + | |
| | Bounds on type parameters | | | | | + | + |
| | Unnamed types | + | | | + | + | + |
| | Type operators | | | + | | + | + |
| | Recursive types | + | + | | + | | + |
| Impl. | Invariance of identity | + | | + | + | + | + |
| | Separate compilation | + | + | + | | + | + |
| Formalization | | | | + | + | | |

Table 5.1: A comparison of recent work on introducing structural typing into nominally typed languages.

The first section in the table compares the admissible member kinds. We see that WHITEOAK's repertoire of member kinds is fairly complete. Still, unlike WHITEOAK and all the other compared languages, only C++ signatures support field definition, that is, fields together with a default initialization expressions. But since the C++ signatures' semantics is restricted, the problem of finding appropriate semantics to default initialization expressions remains open. We know of no support in current work of constructor bodies in structural types.

The next section compares integration with other linguistic mechanisms and composability of structural types through recursion or type operators. As expected, most languages chose to provide support for anonymous types on top of structural types (unlike *instances* of anonymous classes, *anonymous types* are practically useless in a purely-nominal setting). Note that Scala's parametrization of structural types in restricted in the sense that type parameter can only be used in co-variant positions (e.g.: return types).

The third section pertains to implementation. Examining preservation of object identity we

see that C++ signatures implement this (by enhancing the compiler existing mechanism for comparison of **this**-adjusted references), while Brew, a previous addition of structural types to JAVA fails to preserve object identity.

Compound also preserves object identity, by relying on a fundamental property of this JAVA language extension, by which structural types can only be defined as the union of existing nominal JAVA interfaces. Every structural reference in Compound is represented as multiple variables, one for each of what the authors call "constituent type".

Finally, as the last table row indicates, only Compound and Unity enjoy a formalized basis.

## 5.3 Implementation and Performance

This section describes the compilation and run-time techniques that we used in order to augment JAVA with WHITEOAK's additional constructs while preserving these fundamental principles:

1. WHITEOAK classes should run on any standard, JAVA 5 compliant JVM; hence the run-time system must be implemented as a JAVA library.

2. Object identities must be preserved; a structural type reference to an object must be the same as a nominal type reference to it.

3. Compilation should be separate without relying on a whole-world analysis; WHITEOAK compiler cannot consult all uses of a given reference type.

4. No executable blowup; the compiler cannot generate a nominal type system reflecting the structural hierarchy by generating a nominal type for each subset of the set members of all structural types.

### 5.3.1 The Object Identity Problem

To better understand the difficulty in preserving the identities of objects, let us consider the following code fragment:

```
struct S { Object me() { return this; } }

Object o = new Object();
S s = o;
assert o == s && o == s.me();
```

In this fragment we have three object references, `o`, `s` and `s.me()`. Given that `s` is assigned from `o` and that `S.me()` returns **this**, it is only natural to expect the two equalities `o == s` and `o == s.me()` to hold. Following JAVA's standard semantics, one can also conclude that `s == s.me()`.

Examining the standard techniques for adding behavior to existing objects, such as the DECORATOR pattern [81], we note they prescribe the use of a wrapper object whose identity is inevitably different than the identity of the wrapped object. Specifically, naïve wrapper-based techniques stop identities such as `o == s` and `o == s.me()` from holding.

Alternatively, object identity could also have been assured by overriding the compiler default implementation of reference comparison operation. This seems to be the desired alternative in C++, where the frequent use of **this**-adjustment [87] forces such specialized comparison code even for nominal types. In JAVA however, such an addition is not only foreign to the language, but would have come at the cost of breaking previously compiled code. For example, the invariants of libraries relying on reflection may be invalidated even if the wrapper is hidden using a **this**-adjustment technique.

WHITEOAK's design insists on both binary compatibility and on semantics of object identity (in particular, the aforementioned assertions hold in WHITEOAK). This is achieved by a technique which may be called *Invisible Wrapper* described below.

### 5.3.2 Compile Time Representation

The WHITEOAK compiler represents every structural type, $S$, as an interface $I_s$. This interface is generated as soon as the parsing of $S$ is complete and therefore it serves as the only representation of $S$ during compilation. In other words, from the compiler's point of view, the structural type exists only as $I_s$; there is no representation for $S$ per-se.

The mapping from $S$ to $I_s$ is straightforward: Each method (either abstract or non-abstract) declared in $S$ is represented as an abstract method declaration in $I_s$. $I_s$ also declares an abstract, inner, static class, $C_s$, which **`implements`** the interface $I_s$. Each definition of a non-abstract method in $S$ is translated into a similar definition in $C_s$. Hence, $C_s$ is referred to as the *Partial Implementation Class* of $I_s$.

Similarly, a constructor with a certain signature is represented as a method named `constructor` with this signature whose return type is $I_s$.

A field $f$ declared in $S$ is translated into a **`static final`** field of the same name and type in $I_s$. Also, a uniquely named getter method—corresponding to $f$—is declared in $I_s$. If $f$ is a non-final field, $I_s$ also contains a corresponding uniquely named setter method.

Finally, The $I_s$ interface carries a special annotation that distinguishes $I_s$ from "normal" interfaces (that is: interfaces that are not an artifact of a structural type declaration).

Under this translation scheme, $I_s$ is a faithful representation of $S$: all elements of the structural type $S$ are represented, without loss of information, as standard JAVA entities in $I_s$ and its inner class $C_s$. Also note that $I_s$ is the primary representation of $S$: whenever $S$ is specified in the program as the type of a variable, a return type of a method, or a bound on a type parameter, it is $I_s$ that will replace it in the compiler's internal representation. The class $C_s$ is merely a vehicle which "carries" the implementations of non-abstract methods from $S$: there is no way for the programmer to specify $C_s$ as a type in the program.

**Type Checking.** Given that $I_s$ and $C_s$ are plain JAVA definitions, type checking can largely follow JAVA's familiar semantics. In particular, access to a method of a structural type is type-checked in the same manner as access to an interface-declared method. Access to a constructor of a structural type is syntactically identical to a method call (e.g.: `x = y.constructor(5)`). Consequently, constructor calls are type-checked like method calls. Finally, a read operation from a field of a structural type is handled as a read operation from a (**`static final`**) interface field.

The primary changes to the type checker are as follows:

- Assignment to a field of a structural type is allowed only if the field has a corresponding setter method.
- When a type conformance test is in order (e.g.: assignments, instantiation of generics) WHITEOAK's type checking algorithm (Section 5.2.4) is used in place of JAVA's nominal subtyping rules.

### 5.3.3 Code Generation and Invisible Wrappers

In order to support WHITEOAK constructs the bytecode generator has to deal with three primary issues: assignments into variables, run-time type tests and method invocation.

**Assignments.** The structural type `S` defined by

```
struct Subbable {
  String substring(int i);
}
```

is a supertype of the nominal type `String`, so if `sub` is a variable of type `Subbable` the assignment

```
sub = "whiteoak"
```

type checks correctly by the compiler. The JVM verifier [127, Sec. 4.9.1] however will reject the assignment if `sub` is a field or a method argument[4] since two the types **`class`** `String` and **`interface`** `Subbable` (representing internally the structural type) are nominally incompatible. The difficulty is resolved by a technique similar to the implementation of type erasure of generic classes: variables of structural types take the type `Object` in their **`.class`** representation. The real (structural) type of these is preserved in an annotation so the symbol table can still reflect the correct type of definitions from separately compiled modules.

**Type tests.** Typecasts and runtime subtyping tests [184] executed at runtime are another difficulty. The JVM does not acknowledge that `String` is a subtype of `Subbable`. This is the reason that the WHITEOAK compiler replaces such tests by a call to a library function which executes, at runtime, the structural subtyping algorithm to compare the object's dynamic type with the given structural type. As it turns out, typecasts in the JVM are no different than subtyping tests.

**Dispatching.** In order to understand the challenges in dispatching, let us consider the call `sub.substring(5)`.

This call type checks correctly since interface `Subbable` has a public `String substring(int)` method. The emitted code must however dispatch the call to `substring(int)` from class `String` despite the fact that type `Object` has no such method (as a direct result of the erasure process described above, the JVM considers the `sub` variable to be of type `Object`). Even if the JVM type of `sub` was interface `Subbable` which includes such a method, and even if the assignment of a `String` to such a variable would have been possible, dispatching would be difficult since we have no access to the mechanism by which the JVM dispatches interface calls to the correct class implementation.

Dispatching is realized in our implementation by class `WrapperFactory` of WHITEOAK's runtime library. This class effectively augments the JVM with an ability to virtual dispatch a method through a structural method selector. The method

```
public final<I> I wrap(Object content, Class<I> description);
```

of this class produces an *invisible wrapper* around its `content` argument, through which dispatching is carried out. Method `wrap` requires these two preconditions:

- The actual type passed to `I` is some $I_s$ interface representing a structural type $S$.
- The `content` object's runtime type, $R$, is structurally conforming to $S$.

The method's contract guarantees that it will return an instance of a "wrapper class", $W$, that is a *nominal* subclass of class $C_s$ corresponding to $I_s$. Given that class $C_s$ always **`implements`** $I_s$ (ensured by the way it is built by the compiler) we have that $W$ is a nominal subtype of $I_s$. Moreover, the fact that $W$ subclasses $C_s$ elegantly injects into the wrapper object the default implementation declared by the structural type.

The creation of $W$ by `wrap()` follows this pattern: for each method $m$ of $I_s$ that has a compatible method $m'$ in $R$, class $W$ defines a corresponding non-abstract method with the same

[4] Unlike local variables, the verifier is aware of the declared type of fields and arguments.

signature as $m$ whose body delegates to $m'$ on the `content` object. This ensures that methods provided by the `content` object will override default implementations of methods from $S$.

Method dispatching is realized by first creating an invisible wrapper around the receiver, and then using a nominal `invokeinterface` call to dispatch the call correctly. More specifically, the compiler emits bytecodes patterned after the algorithm depicted in Algorithm 7.

---

**Procedure:** $StructuralDispatch(S, r, m)$

---

**Input:** $S$ a structural type, $r$ an object of static type $s$, $m$ a method ($m \in S$)

1: Load a per-thread `WrapperFactory` object onto the stack.
2: Load $r$ onto the stack.
3: Load the class literal $I_s$ onto the stack.
4: Call `WrapperFactory.wrap()` via `invokespecial`.
5: Downcast the object at the top of the stack to $I_s$.
6: Load `m`'s parameters onto the stack.
7: Call `m()` via `invokeinterface`.

Algorithm 7: JVM code to dispatch method $m$ on a receiver $r$ of structural type $I_s$.

The first four steps of Algorithm 7 issue a `WrapperFactory.wrap()` call with the receiver $r$ passed as the content parameter, and $I_s$ passed as the description parameter.

This call is issued on a per-thread instance of class `WrapperFactory`. Although a per-thread instance requires the introduction of an extra local variable, it is preferred over a globally shared instance: a per-thread instance can safely assume that its methods are always invoked from the same thread, thereby eliminating the performance penalty incurred by inter-thread synchronization.

The downcast at the $5^{th}$ step always succeeds (since $W$ is a nominal subtype of $I_s$). This ensures that the `invokeinterface` instruction at the $7^{th}$ step of the algorithm verifies correctly.

A concrete bytecode-level incarnation of the pattern depicted by Algorithm 7, for method `Subbable.substring()`, is shown in Figure 5.10.

In the figure, each of the instructions at lines 1—7 represents the corresponding step from Algorithm 7. The local variable `_wrapper_factory_` is generated by the compiler in every method that performs structural dispatching. It is initialized with a per-thread instance of class `WrapperFactory`, as explained above. This local variable is used for all structural dispatching taking place within the enclosing method. The initialization of this variable (via JAVA's `ThreadLocal` API) is happening at most once, per method execution, and only in methods that perform structural dispatching.

The invocation depicted in Figure 5.10 is of a method that takes only one parameter of type

---

**Figure 5.10** The bytecodes generated for the invocation `sub.substring(5)`, where `sub` is a variable of a structural type `Subbable` that declares the method `String substring(int)`.

```
1 aload_1 // Load variable _wrapper_factory_
2 aload_2 // Load variable sub
3 ldc_w #3 // class Subbable
4 invokespecial #4 // Method WrapperFactory.wrap(Object,Class)
5 checkcast #3 // class Subbable
6 bipush 5 // Push the constant 5
7 invokeinterface #5 // Method Subbable.substring(int)
8 pop // Returned value is ignored
```

---

**int**, namely: `Subbable.substring()`. Thus, the $6^{th}$ step of Algorithm 7 becomes a single `bipush` instruction in the figure. Also note that the instructions at lines 2, 6, 7, and 8 are the same as any virtual function call, and that the invisible wrapper generated at line 4, vanishes when the call is finished.

The WHITEOAK compiler generates the aforementioned sequence of instructions only when it generates the code for an invocation of a method on a receiver of a (static) structural type. Assignments to such variables are allowed only if the static type of the assigned value is structurally conforming to the type of the variable. This guarantees that in each compiler-generated invocation of `WrapperFactroy.wrap()` the preconditions required by the method are met.

We further argue that class $W$ is always a concrete class: If the `content` object does not provide a suitable implementation for $m$ and $m$ is non abstract in $S$, then class $C_s$ will have a concrete $m$ method (again, this is ensured by its building process). class $W$ will inherit this implementation from its superclass $C_s$.

In the dual case—$R$ does not provide a suitable implementation for $m$ and $m$ is *abstract* in $S$—we have that in all the nominal supertypes of $R$ (including $R$ itself) there is no declaration of a method $m$, not even an **`abstract`** declaration. If there were such an abstract declaration then $R$ were an abstract class which cannot, by definition, be the dynamic type of an object. Given that no definition for $m$ exists in the hierarchy above $R$ we have that $R$ is not structurally conforming to $S$ (see Section 5.2.4) in contradiction to the precondition of the `wrap()` method.

$W$ being a concrete class guarantees the following: (*i*) `WrapperFactory.wrap()` will be able to create an instance of $W$; (*ii*) the method invocation in the $7^{th}$ step of Algorithm 7 will succeed.

**Constructors and Fields.** Dispatching to constructors is no different than methods. A call to a **`constructor()`** method on a structurally typed receiver is handled by a corresponding **`constructor()`** method in class $W$ whose body carries out the standard bytecode sequence for creating a new object of type `S`.

Access to fields of a structural type is made possible by invocation of the appropriate getter or setter method introduced by the compiler into $I_s$. In the wrapper class $W$ the implementation of such methods uses either a `GETFIELD` or a `PUTFIELD`) instruction to carry out the actual operation on the field of the `content` object.

**`this` References.** In a structural type that defines only abstract methods, the identity of the wrapper object is never leaked to the actual program code: the methods of the wrapper object merely delegate to the methods of the `content` object without exposing the **`this`** reference.

Things are more complicated when considering methods with default implementations. When such a method gets executed (that is: the `content` object does not provide a suitable implementation), the actual program code runs in the context of the wrapper object thereby exposing the identity of the wrapper.

To solve this difficulty, the compilation of methods with default implementations into the partial implementation class, $C_s$, is altered so that each **`this`** reference is replaced by a reference to the wrapped object. To achieve this, the partial implementation class defines an abstract function `getReceiver()` whose return type is $I_s$. In methods of a partial implementation class, the compiler inserts a `getReceiver()` call immediately after every "load **`this`**" instruction (bytecode: `aload_0`).

The net effect is two fold: First, the reference to the wrapper object is replaced by a reference to the wrapped content, thus maintaining the transparency of the identity of the wrapper object. Second, in self calls, such as $m_1$ invoking $m_2$, the static type of the receiver becomes $I_s$ (since `getReceiver()` returns $I_s$) so the compiler will treat the call as a structural invocation thereby ensuring correct dispatching semantics of self calls.

We note that this process however does not need to generate a new wrapper class, since both

the dynamic nominal type and the structural static description are the same as in the context which generated the wrapper object.

### 5.3.4 Performance

The implementation approach described above realizes structural dispatching by these steps: (*i*) creation of an invisible wrapper, (*ii*) using the JVM's native `invokeinterface` instruction to dispatch to a wrapper object, and (*iii*) delegation to the actual implementation. As it turn out, the first such step is the most time-intensive.

This step involves two operations which are notoriously slow: the examination of a reflection object, and the creation, at runtime, of a new class. In fact, in an early implementation of WHITEOAK we found that dispatching based on reflection information was at least three order of magnitude slower than native dispatching.

The current WHITEOAK implementation achieves good performance by relying on two levels of caching inside method `wrap()`: First, a fixed size cache containing 8 entries and implemented with no looping instructions is used to check whether an invisible wrapper was previously created for the specified value of the receiver and the static (structural) type of the receiver. Second, if the pair is not found in this cache, a hash table containing all previously returned wrappers is examined.

In both caches the searching strategy is similar: we first check whether an exact match is found, that is, match in both receiver value and in the receiver's (structural) type. If this fails, the dynamic type of the receiver is fetched, and the cache is examined again to see whether a wrapper class appropriate for the receiver's dynamic type and the receiver's static type exists already and can be instantiated to create the invisible wrapper.

In addition, within any single thread the cache may even recycle wrappers by changing their contained wrapped objects. Such recycling is restricted to plain structural types though.

Figure 5.11 compares the timings of runs of a reference program in WHITEOAK with that of the JAVA equivalent using runtime interface dispatch. (Experiments were conducted on a single processor Pentium-4 3GB RAM, 3GHz, Windows XP machine.)

The curve marked "*Interface*" refers to the plain JAVA run. The "*Hit-1*" curve describes the WHITEOAK timing when the program mostly hits the primary cache. The curve marked "*Hit-2*" corresponds to the situation where structural dispatching mostly hits the secondary cache, while the "*Hit-1/2*" curve denotes an intermediate situation.

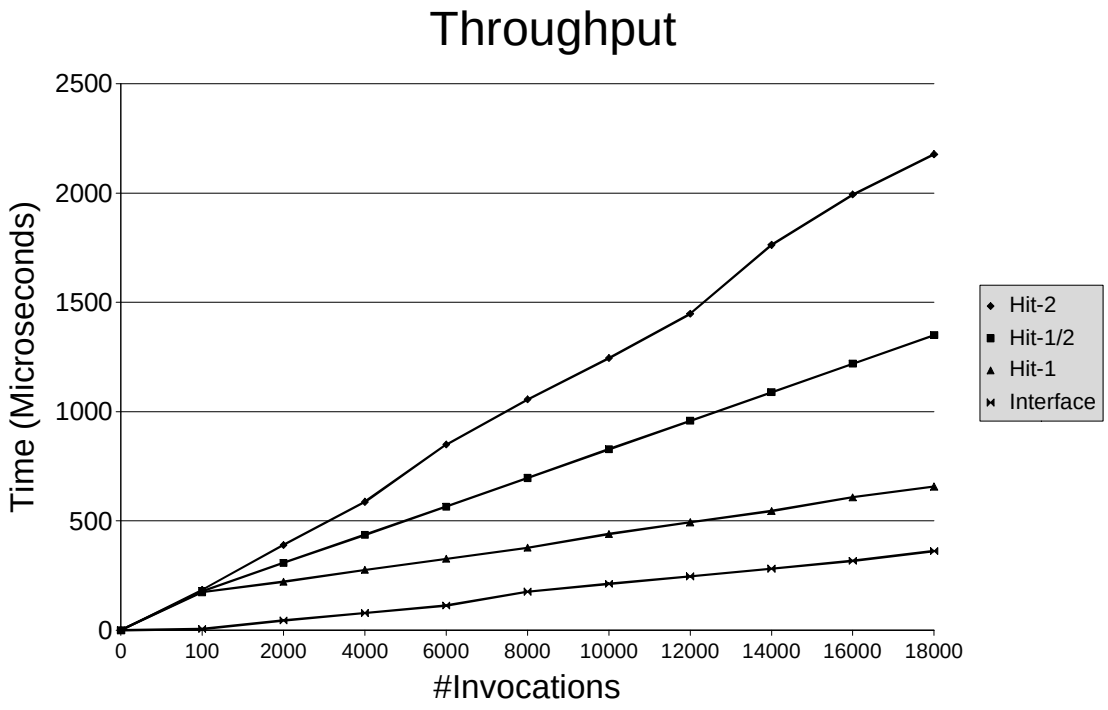
As expected, the curves are linear at large, indicating that the invocation time is constant. Hits on the primary cache incur a slowdown of a factor of about two in comparison to the reference `invokeinterface` implementation. Note that we cannot hope for more; the dispatching algorithm described above, even if caching requires no resources, replaces each structural dispatch with one interface dispatch and two ordinary class based dispatches. The hit-1/2 scenario is more than three times slower where the hit-2 scenario is about seven times slower.

The experiment depicted in this figure represents an unrealistic situation where the program spends an overwhelming majority of its time invoking an empty method. A more realistic program is likely to take additional computation which will mitigate the influence of dispatching on the overall execution time.

To evaluate performance under these circumstances we took two standard benchmarks, JESS and JAVALEX[5] from the famous SpecJVM98 suite and changed them such that every **`interface`** was replaced by a corresponding **`struct`**. Of course, this change meant that **`implements`** clauses were eliminated from all classes.

[5]We use the term JAVALEX instead of the formal name of this benchmark, JAVAC, which is confusing in the context of this paper.

**Figure 5.11** Execution time of a program vs. # number of method invocations.




The JESS program is a rule-based inference system. The JAVALEX benchmark measures the time needed for a JAVA compiler to compile a 60,000 lines program. In the original benchmark, the compiler whose compilation time is measured is one of the early versions of Sun's JAVA compiler. We changed the benchmark to measure the compilation time of Sun's JAVA 5 compiler against that of the bootstrapped WHITEOAK compiler (The bootstrapped version uses **`struct`**s instead of **`interface`**s). The use of interfaces in the remaining SpecJVM benchmark programs was minimal, so applying this techniques in these did not yield useful data.

In both benchmarks the performance of the WHITEOAK program was less than 5% slower than JAVA program. Specifically, In JESS WHITEOAK time was 1.95 seconds while the JAVA time was 1.89 seconds, while in JAVALEX the respective times were 3.9 and 3.75 seconds;

Finally, we compared WHITEOAK's structural dispatching implementation with that of SCALA, in the simple case of repeatedly invoking a method on a single receiver. The slowdown in SCALA was more than two orders of a magnitude, compared to less than one order of a magnitude in WHITEOAK (as shown in Figure 5.11). Note, that we chose such a simple case since some of the features that are needed for a more complicated test, e.g., array access, incur substantially different run-time costs in the two languages, thereby biasing the method invocation measurement that we seek.

The differences between SCALA's and WHITEOAK's dispatching mechanisms were also studied by Dubochet and Odersky [69].

## 5.4 Summary

The introduction of structural types into a nominally typed language has two major implication on the style of programming and design in that language.

First, the use of structural types increase locality: If a type is needed in some place in the

program we only need to define it in the point where it is used. There is no need to change the definition of classes (residing elsewhere in the code) in order to make this new type viable.

This enables concerns to be confined to a well-defined region of the program, resulting in increased coherence of the program's modules. Locality is particularly important in maintaining the balance of power between supplier's and client's code. A change in a supplier module due to a client's needs may eventually lead to deterioration of the design of the supplier, especially when there are multiple clients that inflict changes onto one supplier. Retroactive abstraction obviates many of these changes, thereby preserving the quality of the supplier's design.

The second effect is that of increased uniformity. As argued by Meyer's *principle of uniform access* similar services should be accessible through a single syntactic device carrying a simple, uniform semantics. Looking at JAVA, we see several breaches of this principle, for example: methods and fields are accessed through the same syntax, but with different binding semantics; Procedural abstraction is realized by two kinds of methods, **`static`** and instance methods accessible by slightly different notations, and carrying different binding semantics; finally, there is a dedicated syntax for constructor calls. WHITEOAK allows the programmer to abstract away most of these differences, thus making it easier to read, write, and maintain the source code.

Moreover, the on-going struggle of augmenting the precision and expressiveness of static type systems leads to the emergence of multi-paradigm languages which exhibit a large, feature-rich core that only emphasizes the point in favor of uniform access (In contrast, the elegant, minimal structure of dynamically typed languages enables these to better maintain Meyer's principle).

The semantics of dispatching methods on plain structural types is quite close to that of the proposed `invokedynamic` instruction[6]. Indeed every language implementation with structural types could rely on this new instruction when it becomes available. Alternatively, the performance data reported in this paper provide a reference point for evaluating future implementations of `invokedynamic`.

Support for **`struct`**s with non-abstract methods exceeds the abilities of `invokedynamic`. The implementation details presented here are therefore important for any language that will attempt to provide similar services, without changing the underlying virtual machine. We assume that the research effort aimed at better utilization of the JVM will become more and more important as the JVM gradually turns into a safe and powerful computing platform that can host a large array of languages. In fact, at this point one can reasonably predict that the JVM will actually outlive the JAVA language.

[6] http://www.jcp.org/en/jsr/detail?id=292

# Chapter 6

# Summary

As the complexity of modern software grows [79], so grows the complexity of the code, the complexity of auxiliary information (test scenarios, build scripts, specifications, etc.) that needs to be maintained alongside the code, and the complexity of the tools that should help us safely navigate through this ocean of information.

This work defines the notion of formal patterns and shows their utility in understanding pre-existing code, in formulating powerful code transformation mechanisms, and in expressing fine-grained JAVA types. We believe that this wide range of applications is indicative of the future role of formal patterns in streamlining the management of programming-related information.

To make this vision more concrete let us describe the architecture of a formal pattern-based IDE and its resulting benefits.

**Architecture.** The IDE will exposes a relational model that reflects the information maintained in the raw artifacts (source files, etc.). A query engine, built around JTL or a similar language, will allow clients, either code or users, to retrieve this information. This relational layer will not hide the standard hierarchical (AST) layer. Instead, it is a dual relational-hierarchical view of the same data that allows clients to choose which of the two models is more appropriate for a specific task.

**Simpler Implementation.** Many standard IDE facilities will be implemented via JTL/JTL$^*$ queries. This includes views (such as *Package Explorer* or *Class Outline*), source code warnings, and refactoring. A JTL-based Implementation of these services will be much simpler than an implementation based on a standard programmatic interface, due to the higher level of abstraction offered by JTL.

**Enhanced Functionality.** The synopsis of a class or a method will be extended to include the micro or nano patterns detected in it, in addition to the (standard) declaration and associated documentation. A developer using such an IDE will be able quickly grasp the intent of a class without delving into its protocol, let alone its implementation.

The search facility will be augmented with a JTL-based search. This will allow precise navigation through the code since JTL queries examine the semantics, which even includes runtime behavior, of program elements. Traditional textual queries inspect only the notational aspect of the code, and thus are not as powerful (and also are more sensitive to changes in the formatting/style of the code).

Moreover, in languages that support structural typing, a semantic search query can often be automatically translated into a type, and vice versa. A developer will be able to prototype the definition of a new type by trying it as a search query before introducing it into the code.

**Customization.** Having a canonical language for manipulating its data model makes the IDE much more open and extensible. An IDE based on formal patterns will allows its users to

interactively introduce new kind of views, warnings, refactoring steps or code generators, simply by typing in a new JTL/JTL* expression. In contrast, in current IDEs the complexity of the internal representation of the code is so high, that the only way to introduce new facilities is through the authoring of plugins in a Turing-complete language.

## 6.1 Directions for Future Research

Here are a few directions for future evolution of the work presented here.

**JTL.** It would be interesting to see if JTL could be enhanced to provide first class support for control flow inspection. Currently, JTL supports data flow inspection (via SCRATCH values) which carry some control flow information. A complete solution would make execution branches, and their mutual relationships, explicit in the JTL data model. Even more challenging is the combination of the two perspectives.

Second, it might be useful to extend JTL to make queries on the program trace, similarly to PQL [134] or PTQL [90]. This extension could perhaps be used for pointcut definitions based on execution stack.

Finally, there is an interesting challenge of finding a generic tool for making language type extensions, for implementing e.g., non-null types [75], read-only types [34], and alias annotations [7]. The difficulty here lies in the fact that the dataflow analysis we presented is a bit remote from the code. Perhaps the grand challenge is the combination of the brevity of expression offered by JTL with the pluggable type systems of Andreae, Markstrum, Millstein and Noble [13].

**Micro Patterns.** The current micro pattern definitions are binary in the sense that they accept or reject an input class. It may be interesting to add weights to part of the definitions, which will make it possible to measure the proximity of a class to a pattern. Weights should also make it possible to build systems which not only discover the use of micro patterns, but also help the user correct his software—by offering concrete recommendations of how to make certain classes a better match to the acquired knowledge base.

With the development of automatic tools for tracing patterns, and the evidence of their significance, it is possible and interesting to expand the notion of micro patterns by studying kinds of interactions between classes obeying various micro patterns and even developing patterns to specify sorts of such interaction. Such a research direction may even mature to tools making more global advice. For example, in a hierarchy where a Pure Type class is subclassed by several Implementor classes, the root class can possibly be turned into a Mould or an Outline class, thus capturing some of the similarities of its subclasses.

Finally, perhaps the most challenging work is in exploring the utility of micro patterns to programmers. This will probably require empirical experimentation with a large group of programmers. We believe that objective assessment of productivity in software is beyond the state of the art, and may likely to be so indefinitely. Thus, a survey of the developer's opinion, possibly backed by qualitative evaluation, may be the correct path for such future work.

**Program Transformation.** Currently, there is no mechanism of automatically proving that the JTL* output is correct, i.e., that it conforms to the specification of the output language. This is the price that powerful output generation techniques collect: the problem at its general form is undecidable. We believe that (in certain cases) there is still hope, by augmenting JTL* with an auxiliary type system, which reflects the type system of the parse tree of the host language. Thus, the process of type checking of predicate definitions in JTL* will also prove that the output is correct.

Another future evolution of JTL* is that of automating the process of writing a transformer. Ideally, an automatic tool will be able to take two JAVA code fragments and suggest several JTL*

programs that translate one to the other. Obviously, this process is likely to be based on heuristics. Even if such a tool can provide only an approximated answer, users will be able to use its output as a skeleton which will be manually refined later. This will allow users to quickly define new refactoring steps, simply by providing a set of "before" and "after" samples.

**Whiteoak.** A promising direction for future research is the use of structural types in conjunction with first-class support for data queries. One of the major predicaments facing the integration of (say) SQL queries into JAVA is the correct typing of the query results. If two different program-embedded SQL queries return similarly structured data it is only natural to expect this data to be of the same type. Locally inferring the nominal type of the result is only a partial solution since the queries may be located at two separately compiled modules. In the absence of whole-program analysis, the inferred types will be nominally different. A structural typing scheme is therefore the natural solution for allowing the interplay of such two types.

A second direction is that of concepts [82]. Currently, JAVA offers a very limited language for expressing constraints on generic types. This work provides two ingredients that are essential for allowing JAVA to support full-blown concepts: (*i*) a rich language for conditions on types (JTL); (*ii*) structural typing (WHITEOAK).

# Appendix A

# The JTL Manual

## A.1 General

- `:=`

  The Query Definition Operator. The `:=` symbol is used to define a new JTL query. It separates the query's head, which specifies the name and the parameters, from the query's body.

  The example defines a `grandparent` query that takes the explicit parameter `X` (note that every query also takes an implicit subject parameter):

  ```
  garndparent X := extends Y, Y extends X;
  ```

- `#`

  The Subject Parameter Symbol. In JTL, every query always takes at least one parameter. This parameter is implicitly defined for all queries and is denoted by the # symbol. As a convention, the # parameter is used as the *inspected value* of the query, that is: the primary input which will be examined by the query. Additional parameters need to be explicitly define, if needed.

  In most queries the # does not appear inside the query's definition. This is due to this JTL rule: the subject of a query becomes (by default) the subject of terms contained within the predicate. For example, in:

  ```
  public_static := public static;
  ```

  the # of the `public_static` query is passed to the `public` and `static` terms invoked by it. Recalling that # stands for the inspected value and that blank stands for disjunction we can read this definition as follows: *The predicate holds if its inspected value is `public` and `static`.*

  The default behavior described above can be overruled by placing a variable in a prefix position with respect to a term:

  ```
  extends_a_public_class := extends X, X public;
  ```

  In this example, the term `extends X` selects into `X` the superclass of the inspected value. In `X public` we place `X` in a prefix position thereby evaluating the `public` term with `X` being the inspected value.

  Finally, here is a rewrite of these two queries this time with the # parameter being explicitly defined and used:

```
# public_static := # public # static;
# extends_a_public_class := # extends X, X public;
```

## A.2 Composition of Terms

- **(blank) or , (comma)**

  The Conjunction Operator. In JTL, conjunction is denoted by either the comma symbol or the blank symbol. Thus, if we want to express the fact that two terms must concurrently hold we will separate them with a comma (“,”) or a space (“ ”) as follows:

  ```
  public_and_static := public static;
  static_and_final := static, final;
  ```

- **&**

  The Subject Change Operator. When a term is evaluated, it inspects, by default, the same value as the one inspected by the containing predicate (*an implicit subject*). One can overrule this default by prefixing a term with a variable that denotes its inspected value (*an explicit subject*), as in: `extends X, X public X abstract;`.

  The Subject change operator, & is a shorthand for repeated application of an explicit subject. Specifically, if we have two terms separated with & as in: `<term-a> & <term-b>` then the subject of `<term-a>` will also be used as the subject of `<term-b>`. Using this operator, complex queries can be rewritten in a more readable form as illustrated in the following example:

  ```
  super_and_its_interface[S,I] := extends S,
        S public S abstract, S implements I;
  ```
  *−− rewritten using &:*
  ```
  super_and_its_interface[S,I] := extends S,
        S public & abstract & implements I;
  ```

- **|**

  The Disjunction Operator. The vertical bar indicates that either the left hand side term or the right hand side term must hold.

  ```
  public_or_static := public | static;
  ```

- **!**

  The Negation Operator. The exclamation point symbol can be attached to a term to negate its semantics. Thus, the term `!p X` will hold only if the subject and `X` do not satisfy the query `p`.

  The query in the following example is satisfied only if Y is *not* a super class of the inspected class:

  ```
  not_super Y := ! extends Y;
  ```

  Note that a negated term cannot compute (or obtain) values for the variables that appear inside it. Specifically, in order to evaluate `! extends Y` the value of `Y` must be known. This requirement propagates to the site where `not_super` is used: a term such as `not_super Z` can be evaluated only if the inspected value and the variable `Z` are known. To summarize, a negated term can only verify that the given values uphold a certain property but it cannot find these values.

- **`[ ... ]`**

  The Grouping Operator. Squared brackets are used for two different purposes:

  First, they are used for enclosing the list of parameters of a query. This happens both at the definition of the query (formal parameters) and when the query is used (actual parameters):

```
parent_grandparent[X,Y] := extends X, X extends Y;
parent_and_grandparent_are_abstract :=
  parent_grandparent[A,B], A abstract, B abstract;
```

  Note that in queries that have only one explicit parameter, JTL allows the squared brackets to be omitted. Thus the following two queries are equivalent:

```
grandparent[X] := extends[Y], Y extends[X];
grandparent X := extends Y, Y extends X;
```

  Second, squared brackets can be used to overrule the default operator precedence. JTL employs the standard behavior where conjunction has higher precedence than disjunction. This behavior can be overruled by enclosing an expression in a pair of squared brackets. Therefore the following queries, `p` and `q`, are equivalent:

```
p := [public | protected] final;
q := public final | protected final;
```

## A.3 Special Queries and Literals

- **`( ... )`**

  The Method Signature pattern. A pair of parenthesis denotes a query on the signature of a method. Inside the enclosed scope we can place values, that is: variables or type literals, that will be matched against the signature of the inspected value (implies that the inspected value is a method):

```
public_method_two_int_parameters := public (int,int);
public_method_two_parameters_of_same_type := public (T,T);
method_one_primitive_parameter := (T) T primitive;
```

  The `(...)` operator supports the `*` (asterisk) wild-card: a special symbol that will match any sequence of parameters of any type. In the following example we capture, using the asterisk symbol, all parameters of the inspected method but the last one:

```
public_method_last_param_is_int := public (*,int);
```

  Similarly, in:

```
method_taking_two_ints := public (*,int,*,int,*);
```

  The three asterisks capture three sequences of parameters that are separated with a single `int` parameter. The net effect is query that is matched by any public method taking at least two `int` parameters.

- **’...’ (single quote)**

  The Regular Expression pattern. A sequence of characters that enclosed inside a pair of single quotes denotes a regular expression query: it will be satisfied by the inspected value only if the inspected value’s name matches the regular expression.

```
p := method ’toString’;
q := method ’eq?*’;
```

The `p` query matches methods whose name is `toString`. The `q` query matches methods whose name is `eq` followed by any sequence of characters.

The regular expression constructs supported by JTL are depicted in Tab.A.1.

| Construct | Matches |
|---|---|
| $x$ | The character $x$ |
| $?$ | Any character |
| $[abc]$ | $a$, $b$, or $c$ (characters) |
| $[a-zA-Z]$ | Characters $a$ through $z$ or $A$ through $Z$, inclusive |
| $X*$ | $X$, zero or more times |
| $X+$ | $X$, one or more times |

Table A.1: Summary of regular-expression constructs

Note that this use of the question mark symbol is somewhat unconventional: usually, the dot symbol — and not the question mark symbol — is used for denoting the *any character* match. By diverting from this convention, JTL's regular expressions allow users to easily match fully qualified names, as in:

```
class_from_java_util := class 'java.util.?*';
```

If the regular expression contains only plain alphabetic characters the terminating single quote can be omitted. Thus, query `p` can be rewritten as:

```
p := method 'toString ;
```

- **/...**

  Type Literal. A sequence of characters that starts with a forward slash denotes a type literal: a JTL value representing the Java type whose fully qualified name is specified:

  ```
  extends_date := extends /java.util.Date;
  ```

## A.4 Quantification and Set Conditions

- **`{ ... }`**

  The Quantification Scoping Operator. A pair of curly braces defines a scope which encloses set-queries and local definitions. A scope of set-queries evaluates to true if each of its set queries evaluates to true.

  Each set query is composed of a set condition (e.g.: **`exists`** or **`implies`**) and of one or more JTL expressions known as: *subset expressions*. Here are two examples for set queries:

  ```
  all static final;
  field implies private;
  ```

  In the first set query the set condition is **`all`** and the subset expression is `static final`. In the second set query the set condition is **`implies`** which prescribes two subset expressions. In here the two subset expressions are `field` and `private`.

  Also, a scope of set-queries is always associated with a generator term. If the generator term is omitted, the default generator is used. The default generator generates the set of all declared members (see *declares*) if the inspected value is a type, or the set of all scratches if the inspected value is a method.

Evaluation of a single set query begins by obtaining a set of values (*the generated set*) from the generator associated with the enclosing scope. Then, JTL computes a subset of the generated set that contains only those values that satisfy the subset expression embodied in the set query. Finally, JTL checks that this subset satisfies the requirement imposed by the set condition (embodied in the set query) with respect to the generated set.

Thus, the set query `one public;` builds a subset containing all *public* elements of the generated set and then checks that this subset contains exactly one element.

Some set conditions (such as: **`implies`** or **`partition`**) operate on more than one set. Therefore they contain several subset expression which yield several subsets, one for each subset expression. The set condition is evaluated against all these subsets. For example, the set query `field implies private;` builds two subsets from the generated set: (1) all fields, and (2) all private elements. The set condition **`implies`** compares these two subsets by checking that the former is included in the latter.

In the following example

```
p := class { all public; one static method; }
```

The quantification scope in `p` contains two set queries: `all public;` and `one static;`. The set conditions are **`all`** and **`one`**, where the respective subset-expressions are `public` and `static method`. No generator expression is specified so the default generator will generate the set of all members of the inspected class. Therefore, the query `p` will hold if all elements in the generated set (that is: all members of the class) are *public* and there is exactly one *static method* in this set.

Note that the subject inside the quantification scope is different than the one outside the scope. Specifically, inside the scope the subject iterates over each element of the generated set.

The complete list of JTL's set conditions includes: **`exists`**, **`all`**, **`no`**, **`one`**, **`many`**, **`implies`**, **`differ`**, **`equal`**, **`disjoint`**, **`partition`**.

- `:`

  The Generation Operator. A quantification scope (see above) is always evaluated against a generator expression. One can specify a generator expression by placing the name of a binary query (that is: a query that takes one explicit parameter) followed by a colon (`:`).

  For example, the generator expression `implements:` will generate the set of all direct super-interfaces of the inspected class. the generator expression `throws:` will generate the set of all exceptions that are listed in the inspected method's throws clause.

  In the following query:

  ```
  all_supers_are_concrete := extends+: { all !abstract; }
  ```

  we use the generator `extends:` to generate the set of all super-classes of the inspected class (including indirect super-classes). The set-query `all !abstract` places the condition that every such super-class is not an *abstract* class.

  Note that a generator expression can be omitted in which case the quantification scope will use the default generator that generates the set of all declared members if the inspected value is a type or the set of all scratches if the inspected value is a method.

- **`exists`**

  The Existential Set Condition. This set-condition asserts that at least one element in the generated set that satisfies the specified expression.

The following query uses the **exists** condition to verify that the inspected class declares at least one non-*final* non-*static* field:

```
has_mutable_instance_field := class {
     exists !final !static field;
   };
```

The query uses the default generator to generate the set of declared members of the inspected class. The **exists** condition specifies the expression `!final !static field` so during evaluation JTL finds the subset of the generated set where every element is not final, not static and is a field. Finally, this subset is checked for existence, that is: that there is at least one element in the set.

The existential condition is the default condition that is used if a set query does not specify a condition. Thus, the query can also be rewritten as:

```
has_mutable_instance_field := class {
     !final !static field;
   };
```

Note that JTL's natural semantics is that of existence. Thus, the query can also be rewritten as:

```
has_mutable_instance_field := class declares X,
     X final X static X field;
```

- **all**

  The Universal Set Condition. This set-condition asserts that every element in the generated set satisfies the specified expression.

  The following query uses the **all** condition to verify that every member of the class (default generator is used here) is final.

  ```
  every_member_is_final := class {
       all final;
     };
  ```

- **no**

  The Empty Set Condition. This set-condition asserts that no element in the generated set satisfies the specified expression.

  The following query uses the **no** condition to verify that there are no public fields in the inspected class:

  ```
  no_public_fields:= class {
       no public field;
     };
  ```

- **one**

  The Singluar Set Condition. This set-condition asserts that exactly one element in the generated set satisfies the specified expression.

  The following query uses the **one** condition to verify that the inspected class declares exactly one constructor:

  ```
  one_ctor := class {
       one constructor;
     };
  ```

- **`many`**

  The Plural Set Condition. This set-condition asserts that there are at least two elements in the generated set that satisfy the specified expression.

  The following query uses the **`many`** condition to verify that the inspected class implements at least two interfaces of public visibility:

```
has_two_or_more_interfaces := class implements: {
    many public;
  };
```

- **`implies`**

  The Implication Set condition. This set-condition compares two sets: it requires that every element of the set on the left is contained in the set on the right.

  The following query uses the implication condition to assert that every public field (in the inspected class) is final:

```
every_public_field_is_final := class {
    public field implies final;
  };
```

- **`differ`**

  The Inequality Set Condition. This set-condition requires the inequality of two sets: the set on the left must have at least one element that is not included in the set on the right or vice-versa.

  The following query uses the **`differ`** condition to assert the existence of a non-static final field or of a non final static field.

```
p := class {
    final field differ static field;
  };
```

- **`equal`**

  The Equality Set condition. This set-condition requires the equality of the set on the left with the set on the right.

  The following query uses the **`equal`** condition to assert that every final field is also a static field and vice versa:

```
p := class {
    final field equal static field;
  };
```

- **`disjoint`**

  The Disjointness Set Condition. This set-condition requires that several sets are disjoint: no element in these sets appear in two (or more) sets.

  This condition is a multi-set condition: it can check an unlimited number of sets. Sets are defined by a sequence of expression that are separated with the double-comma, `,,`, symbol.

  The following query uses the **`disjoint`** condition to assert that no field is public:

```
p := class {
    disjoint field,, public;
  };
```

- **`partition`**

  The Partitioning Set Condition. This set-condition requires that several set form a partition of the generated set: every element of the generated set belongs to exactly one of the subsets.

  This condition is a multi-set condition: it can check an unlimited number of sets. Sets are defined by a sequence of JTL expressions that are separated with the double-comma, `,,`, symbol.

  The following query uses the **`partition`** condition to assert that every member declared by the inspected class is either a protected abstract method, a public non abstract method, a constructor or a private field.

```
p := class {
     partition
       protected abstract method,,
       public !abstract method,,
       constructor,,
       private field;
   };
```

- `,,`

  The Multiple Expression Separation Operator. Some set conditions (namely: **`partition`** and **`disjoint`**) operate on more than two subsets. In these, the subset expressions are given as a list of expressions separated by the comma-comma symbol (`,,`).

```
p := class {
     partition
       protected abstract method,,
       public !abstract method,,
       constructor,,
       private field;
   };
```

- **`let`**

  The Local Definition Keyword. A scope of curly braces can contain definitions of local queries. These queries will be recognized only inside the scope thereby avoiding the cluttering of the global namespace. A local definition hides similar definitions from enclosing scopes.

  In the example, the set query **`one`** `int_taking_method;` uses the locally defined query `int_taking_method`:

```
p := class {
     let int_taking_method := method (*,int,*);
     one int_taking_method;
   };
```

# Appendix B

# The JTL Standard Library

This chapter enumerates all the predicates in the JTL standard library. Presentation follows a uniform template, explained by the following (annotated) example.

- ***some_predicate M,S***

  *Name of the predicate and its formal parameters, excluding the subject*

  SIGNATURE: TYPE,MEMBER,SCRATCH [or] TYPE,MEMBER,TYPE

  *Arity and type specification. This predicate computes a ternary relation that is a subrelation of* $\{\langle x,y,z\rangle \mid (x \in TYPE \wedge y \in MEMBER \wedge z \in SCRATCH) \vee (x \in TYPE \wedge y \in MEMBER \wedge z \in TYPE)\}$*. In the subrelation, a tuple* $\langle x,y,z\rangle$ *must maintain a certain property captured by the predicate. Parameter binding goes as follows: The (implicit) subject parameter,* #*, is bound to* $x$*, the first explicit parameter,* M*, is bound to* $y$ *and the second explicit parameter,* S*, is bound to* $z$*.*

  COMPUTABILITY: $\#, M \rightarrow S$ [or] $\#, S \rightarrow M$

  *Constraints on parameters. The predicate can compute the the result if either* # *and* M *or* # *and* S *are known.* $\phi$ *on the left-hand side of the arrow (e.g.,* false*,* default_package*) indicates that the predicate computes a constant, finite, relation and thus needs no input. The computability field is omitted if the predicate requires all parameters to be known (as in:* $\#, M \rightarrow \#, M$*).*

  Select into M the member that the inspected value ....

  *A description of the relation embodied by the predicate. The term* inspected value *refers to the subject parameter's actual value.*

  SEE ALSO: *some_other_predicate*

  *Other predicates pertaining to the same topic.*

Following pages depict standard JTL predicates, alphabetically.

- ***@interface***

  SIGNATURE: TYPE

  An alias of *annotation*

  SEE ALSO: *annotation*

- ***abstract***

  SIGNATURE: ELEMENT

  The inspected value is **`abstract`**.

- ***accesses F***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow F$

  Select into `F` the fields that are either read from-, or written to-, by the subject method/constructor or by methods/constructors of the subject class.

  Defined as:

  ```
  accesses F := reads F | writes F;
  ```

  SEE ALSO: *reads*, *writes*

- ***annotated_by T***

  SIGNATURE: TYPE,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  Obtain the annotations of a type. Selects into `T` the annotations that are attached to the inspected type.

  The JTL standard library operates on the binary (classfile) representation of classes so annotations with `RetentionPolicy.SOURCE` will not be detected by this query.

  Evaluating the JTL expression

  ```
  class {
    public method is M annotated_by /java.lang.Override;
  };
  ```

  will assign into `M` all public methods (of the inspected class) that are tagged with `@Override`.

  SEE ALSO: *annotation*

- ***annotation***

  SIGNATURE: TYPE

  The inspected value is an annotation.

  SEE ALSO: *class*, *interface*, *enum*, *array*

- ***anonymous***

  SIGNATURE: TYPE

  The inspected value is an unnamed class.

  SEE ALSO: *class*, *inner*

- ***array***

  SIGNATURE: TYPE

  The inspected value is an array.

  SEE ALSO: *class*, *interface*, *enum*, *annotation*

- ***athrow***

  SIGNATURE: SCRATCH

  The inspected scratch is thrown. Implies that the type of the subject scratch is either `java.lang.Throwable` or a subtype thereof.

  SEE ALSO: *caught*

- ***boolean***

  SIGNATURE: ELEMENT

  Select boolean elements. Holds if the inspected value is the primitive type **`boolean`**; or it is a field of type **`boolean`**; or it is a method with return type **`boolean`**.

- ***byte***

  SIGNATURE: ELEMENT

  Select byte elements. Holds if the inspected value is the primitive type **`byte`**; or it is a field of type **`byte`**; or it is a method with return type **`byte`**.

- ***calls M***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Select into `M` the methods that are called by the subject method/constructor or by methods/-constructors of the subject class.

  Defined as:

  ```
  calls M := calls_instance M | calls_static M;
  ```

  SEE ALSO: *calls_instance*, *calls_static*

- ***calls_instance M***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Select into `M` the instance methods that are called by the subject method/constructor or by methods/constructors of the subject class.

  Defined as:

  ```
  calls_instance M := class offers M, M calls_instance M | {
       receives M;
    };
  ```

  SEE ALSO: *calls*, *calls_static*, *receives*

- ***calls_static M***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

Select into `M` the static methods that are called by the subject method/constructor or by methods/constructors of the subject class.

SEE ALSO: *calls_instance*, *calls*

- ***caught***

  SIGNATURE: SCRATCH

  The inspected scratch is caught. Satisfied by scratches that are explicitly caught by a **`catch`** handler.

  SEE ALSO: *athrow*

- ***char***

  SIGNATURE: MEMBER

  Select char elements. Holds if the inspected value is the primitive type **`char`**; or it is a field of type **`char`**; or it is a method with return type **`char`**.

- ***class***

  SIGNATURE: TYPE

  The inspected value is a class.

  SEE ALSO: *interface*, *array*, *enum*, *annotation*

- ***compared***

  SIGNATURE: SCRATCH

  The inspected scratch is compared.

  Satisfied by scratches that participate on comparison operations such as: greater than, less than, equal-to, etc.

  SEE ALSO: *from*

- ***concrete***

  SIGNATURE: ELEMENT

  The inspected value is not **`abstract`**.

  SEE ALSO: *abstract*

- ***constant***

  SIGNATURE: SCRATCH

  The inspected scratch is a constant. Satisfied by scratches that represent constant values like primitive values, string literals, or **`null`**.

  Assuming `SomeClass` is defined as

```
public static class SomeClass {
  private void f(Object o) { }
  public void g() { f(this); }
  public void h() { f("ab"); }
}
```

  Then evaluating the JTL query

```
class { public method is M { constant; }; };
```

with `SomeClass` as input, will select the method `SomeClass.h()` into `M`, since `h()` uses the constant value `"ab"`.

The method `SomeClass.g()` does not use any constants so it is not selected by the query.

SEE ALSO: *null*

- ***constructor***

  SIGNATURE: MEMBER

  The inspected value is a constructor.

- ***declared_by C***

  SIGNATURE: MEMBER,TYPE [or] SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow C$ [or] $C \rightarrow \#$

  Obtain the declaring type or method.

  If the inspected value is a `MEMBER` selects into `C` the declaring type. If the inspected value is a `SCRATCH` selects into `C` the declaring method.

  The following JTL expression

  ```
  public class {
    field is F declared_by T, typed T;
  }
  ```

  Will assign into `F` fields whose type is the class that declared the fields. If we evaluate this expression over the input `java.awt.Color` then `F` will be assigned with these fields:

  ```
  java.awt.Color#WHITE
  java.awt.Color#LIGHT_GRAY
  java.awt.Color#RED
  java.awt.Color#ORANGE
  ...
  ```

  SEE ALSO: *declares*

- ***declares C***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,SCRATCH

  COMPUTABILITY: $\# \rightarrow C$ [or] $C \rightarrow \#$

  Obtain the members declared by a type.

  Selects into `M` the members that were explicitly declared by the inspected value. Inherited members are excluded.

  Assuming `SomeClass` is defined as

  ```
  package p1;
  public class SomeClass {
    public void f() { };
    private int n;
    public String toString() { return null; }
  }
  ```

  then evaluating the JTL expression

  ```
  declares X;
  ```

over the input `SomeClass` assigns the following members into `X`:

```
p1.SomeClass#f()
p1.SomeClass#n
p1.SomeClass#toString()
```

SEE ALSO: *offers, declared_by*

- ***default_access***

  SIGNATURE: ELEMENT

  The inspected value has default visibility.

- ***default_package***

  SIGNATURE: PACKAGE

  COMPUTABILITY: $\phi \rightarrow \#$

  The inspected value is the default package

- ***double***

  SIGNATURE: MEMBER

  Select double elements. Holds if the inspected value is the primitive type **`double`**; or it is a field of type **`double`**; or it is a method with return type **`double`**.

- ***enum***

  SIGNATURE: TYPE

  The inspected value is an enum.

  SEE ALSO: *class, interface, array, annotation*

- ***extends T***

  SIGNATURE: TYPE,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  Realize the **`extends`** relationship. If the inspected type is a class selects its direct superclass into `T`; If the inspected type is an interface selects its direct superinterfaces into `T`.

  Evaluating the JTL expression `extends T` over the input `java.util.ArrayList` assigns `java.util.AbstractList` into `T`.

  Evaluating the JTL expression `extends T` over the input `java.util.List` assigns `java.util.Collection` into `T`.

  SEE ALSO: *implements, extedns*, extends+*

- ***extends+ T***

  SIGNATURE: TYPE,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  Recursive application of **`extends`**. Defined as

  ```
  extends+ T := extends T | extends T' T' extends+ T;
  ```

Evaluating the JTL expression **`extends+ T`** over the input `java.util.ArrayList` assigns the following types into `T`:

```
java.util.AbstractList
java.util.AbstractCollection
java.lang.Object
```

SEE ALSO: *implements*, *extends*, *extends**

- ***extends* T***

  SIGNATURE: TYPE,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  **`extends+`** or me.

  Produces the same result as **`extends+`** but includes the inspected type itself in the result. Defined as

  ```
  extends* T := is T | extends+ T;
  ```

  Evaluating the JTL expression **`extends* T`** over the input `java.util.ArrayList` will assign the following types into `T`:

  ```
  java.util.ArrayList
  java.util.AbstractList
  java.util.AbstractCollection
  java.lang.Object
  ```

  SEE ALSO: *implements*, *extends*, *extends+*

- ***false***

  SIGNATURE: ANY

  COMPUTABILITY: $\phi \rightarrow \#$

  Always reject the inspected value.

  SEE ALSO: *true*

- ***field***

  SIGNATURE: MEMBER

  The inspected value is a field.

- ***final***

  SIGNATURE: ELEMENT

  The inspected value is **`final`**.

- ***float***

  SIGNATURE: MEMBER

  Select float elements. Holds if the inspected value is the primitive type **`float`**; or it is a field of type **`float`**; or it is a method with return type **`float`**.

- ***from S***

  SIGNATURE: SCRATCH,SCRATCH

  COMPUTABILITY: $\# \rightarrow S$ [or] $S \rightarrow \#$

  Obtain the source scratch of the subject.

  Select into `S` the scratches from which the inspected scratch was assigned.

  SEE ALSO: *func*, *from+*, *from**

- ***from+ S***

  SIGNATURE: SCRATCH,SCRATCH

  COMPUTABILITY: $\# \rightarrow S$ [or] $S \rightarrow \#$

  Recursive application of `from`. Defined as

  ```
  from+ S := from S | from S' S' from+ S
  ```

  Selects into `S` all scratches from which the inspected scratch was copied either directly or indirectly.

  SEE ALSO: *from*, *from**

- ***from* S***

  SIGNATURE: SCRATCH,SCRATCH

  COMPUTABILITY: $\# \rightarrow S$ [or] $S \rightarrow \#$

  from+ or me. Defined as

  ```
  from* S := is S | from+ S
  ```

  Produces the same result as `from+` but includes the inspected scratch itself.

  SEE ALSO: *from*, *from+*

- ***func S***

  SIGNATURE: SCRATCH,SCRATCH

  COMPUTABILITY: $\# \rightarrow S$ [or] $S \rightarrow \#$

  Obtain the scratch from which the inspected scratch was computed.

  Select into `S` the scratches from which the inspected scratch was computed using simle arithmetical or logical operator.

  SEE ALSO: *from*

- ***get_field M,S***

  SIGNATURE: SCRATCH,MEMBER,SCRATCH

  COMPUTABILITY: $\# \rightarrow M, S$ [or] $S \rightarrow \#, M$

  Obtain scratch that was loaded from a field and the field itself.

  If the inspected scratch was the receiver in a load-from-field operation (via the JVM's `getfield` or `getstatic` instructions) select that field into `M`, and select the loaded scratch into `S`.

  In the case of a static field the system produces a "fake scratch" that plays the role of the receiver. This allows `get_field` to handle both kinds of fields (static and non-static).

A JTL query can distinguish the two by applying the *static* query on `get_field`'s `F` parameter.

Example 1. The following JTL predicate selects into `F` the fields of the inspected class that are not read by methods of this class.

```
indifferent_to F := class declares F, F field {
  no method {
    get_field[F,_];
  };
};
```

Example 2. We now want to find only "getter" methods. That is: methods that return a value that was loaded from one of the fields of the current object. To do that we pass the receiver scratch through the *this* predicate and the value scratch (the scratch loaded from the field) through the *returned* predicate:

```
getters [M,F] := class {
  let my_getter F := method {
    this get_field[F,V], V returned;
  };
  is M my_getter F;
};
```

Assuming `SomeClass` is defined as:

```
public class SomeClass {
  private int x;
  public int getX() { return x; }
  public void setX(int arg) { x = arg; }
  public void inc() { x = x + 1; }
  public void copyFrom(SomeClass that) { this.x = that.x; }
}
```

then evaluating the JTL expression `getters M` over the input `SomeClass`, yields a single pair of results for `M`, `F`:

`SomeClass#getX()`, `SomeClass#x`

In this execution the `copyFrom()` method failed to pass the *this* condition. The `inc()` method failed to pass the *parameter* condition.

SEE ALSO: *put_field*, *receives*

- ***get_method M***

  SIGNATURE: SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Select the method whose return value is the inspected scratch.

  Given the following method

  ```
  public void f() { toString(); wait(); }
  ```

  the JTL expression

  ```
  method { get_method M; }
  ```

  will select the method `toString()` into `M` because there is a scratch that takes the value returned from the `toString()` call. This scratch exists despite the fact the `toString()`'s result is dicarded and not assigned into a variable.

On the other hand, `wait()` will not be selected by this query: it is a void method which has no return value.

SEE ALSO: *put_method*

- ***getter F***

  SIGNATURE: MEMBER,MEMBER

  The subject is a getter method of the `F` field. A getter method is an *inspector* returning the value of a field of the receiving object.

```
getter F := !void instance method () {
     this get_field[F,V], V returned;
  };
```

  SEE ALSO: *get_field*, *inspector*, *setter*

- ***global M***

  SIGNATURE: MEMBER

  COMPUTABILITY: $\phi \rightarrow \#$

  The inspected value is declared by `Object`. Defined as:

```
global := # declared_by Object;
```

  SEE ALSO: *non_global_members*

- ***implements T***

  SIGNATURE: TYPE,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  Obtain directly implemented interfaces. Selects into `T` all interfaces directly implemented by the inspected class. This predicate realizes the relationship induced by the Java keyword **`implements`**.

  Evaluating the JTL expression

```
implements T
```

  over the input `java.util.ArrayList` assigns the following types into `T`:

```
java.util.List
java.util.RandomAccess
java.lang.Cloneable
java.io.Serializable
```

  SEE ALSO: *extends*, *extends+*, *extends**

- ***inner***

  SIGNATURE: TYPE

  The inspected value is an inner class. An inner class may be unnamed in which case it will satisfy the *anonymous* predicate.

  SEE ALSO: *class*, *anonymous*

- ***int***

  SIGNATURE: MEMBER

  Select int elements. Holds if the inspected value is the primitive type **`int`**; or it is a field of type **`int`**; or it is a method with return type **`int`**.

- ***inspector***

  SIGNATURE: MEMBER

  The inspected value is an inspector method. A method is considered to be an inspector if it reads the value of a field of the declaring class.

  SEE ALSO: *get_field*

- ***interface***

  SIGNATURE: TYPE

  The inspected value is an interface.

  SEE ALSO: *class*, *enum*, *array*, *annotation*

- ***is A***

  SIGNATURE: ANY,ANY

  COMPUTABILITY: $\# \rightarrow A$ [or] $A \rightarrow \#$

  Require equality. Asserts that the inspected value and `A` are the same value. If the subject is known and `A` is unknown then this predicate has the effect of assigning the subject into `A`, and vice-versa.

  The following query:

  ```
  class is T {
      public method void (T);
    };
  ```

  will match a classes if it has method taking the class itself as a parameter. We use `is T` to assign the inspected class into the variable `T` which we then use in the condition `method void (T)` inside the curly braces.

  SEE ALSO: *is_not*

- ***is_not A***

  SIGNATURE: ANY,ANY

  Require inequality. Satisfied only if the inspect value and `A` are different.

  SEE ALSO: *is*

- ***items X***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,SCRATCH

  COMPUTABILITY: $\# \rightarrow X$ [or] $X \rightarrow \#$

  The default generator. Obtain the members of a class or the scratches of a method.

  Defined as:

  ```
  items X := class declares X | method scratches X;
  ```

  SEE ALSO: *declares*, *scratches*

- ***local_var***

  SIGNATURE: SCRATCH

  The inspected scratch is assigned into a local variable. Satisfied by scratches that are explicitly assigned into the local variable array (LVA) of the enclosing Java method.

  SEE ALSO: *temp*, *locus*

- ***locus S***

  SIGNATURE: SCRATCH,SCRATCH

  COMPUTABILITY: $\# \rightarrow S$ [or] $S \rightarrow \#$

  Require two scratch to be stored at the same local variable.

  Satisfied if the inspected scratch and `S` are stored at the same local variable.

  SEE ALSO: *local_var*, *this*

- ***long***

  SIGNATURE: MEMBER

  Select long elements. Holds if the inspected value is the primitive type **`long`**; or it is a field of type **`long`**; or it is a method with return type **`long`**.

- ***member***

  SIGNATURE: MEMBER

  The inspected value is a member.

- ***members C***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,SCRATCH

  COMPUTABILITY: $\# \rightarrow C$ [or] $C \rightarrow \#$

  An alias of *declares*

  SEE ALSO: *declares*

- ***method***

  SIGNATURE: MEMBER

  The inspected value is a method.

- ***mutator***

  SIGNATURE: MEMBER

  The inspected value is a mutator method. A method is considered to be a mutator if it makes an assignment into a field of the declaring class.

  SEE ALSO: *put_field*

- ***native***

  SIGNATURE: MEMBER

  The inspected member is a native JAVA method.

- ***non_global_members M***

  SIGNATURE: TYPE,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain all members offered by a type excluding global ones. Selects into `M` all members that exist in the inspected type, including inherited members, excluding those that were declared by the `Object` class.

  Defined as follows:

  ```
  non_global_members M := offers M, M !global;
  ```

  SEE ALSO: *offers*, *global*

- ***non_global_methods M***

  SIGNATURE: TYPE,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain all non-overridden, non-global methods of a type. Selects into `M` all non global methods (as per *non_global_members*), retaining only the most specific version of each method. This means that a method will be selected into $M$ if it is neither declared by `Object` nor is it overridden in $\#$ inheritance chain.

  SEE ALSO: *non_global_members*

- ***null***

  SIGNATURE: SCRATCH

  The inspected scratch is the **`null`** constant.

  SEE ALSO: *constant*

- ***offers M***

  SIGNATURE: TYPE,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain all members offered by a type. Selects into `M` all members that exist in the inspected type, including inherited members.

  Assuming `SomeClass` is defined as

  ```
  package p1;
  public class SomeClass {
     public void f() { };
     private int n;
     public String toString() { return null; }
  }
  ```

  then evaluating the JTL expression

  ```
  offers X;
  ```

  over the input `SomeClass` assigns the following members into `X`:

  ```
  p1.SomeClass#f()
  p1.SomeClass#n
  p1.SomeClass#toString()
  java.lang.Object#wait(long,int)
  ```

```
java.lang.Object#registerNatives()
java.lang.Object#hashCode()
java.lang.Object#wait(long)
java.lang.Object#toString()
java.lang.Object#clone()
java.lang.Object#wait()
java.lang.Object.<clinit>
java.lang.Object#notifyAll()
java.lang.Object#getClass()
java.lang.Object#finalize()
java.lang.Object#Object()
java.lang.Object#notify()
java.lang.Object#equals(java.lang.Object)
```

SEE ALSO: *declares*

- ***overrides M***

  SIGNATURE: MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain all overridden methods. Selects into `M` all methods overridden (directly or indirectly) by the inspected method.

  ```
  overrides M := method declared_by T, T substypes+ S,
    S declares P, P method,
    same_name P, signature_compatible P;
  ```

  SEE ALSO: *precursor*

- ***package***

  SIGNATURE: PACKAGE

  The inspected value is a package.

- ***packaged_in P***

  SIGNATURE: TYPE,PACKAGE

  COMPUTABILITY: $\# \rightarrow P$

  Obtain the package of a type. Selects into `P` the package declaring the inspected type.

- ***parameter***

  SIGNATURE: SCRATCH

  The subject scratch is a parameter.

  Satisfied if there is a series of assignments from one of the parameters of the enclosing method (including the **`this`** parameter) into the subject scratch.

  SEE ALSO: *this*, *local_var*

- ***precursor M***

  SIGNATURE: MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain most specific overridden method. Selects into `M` the method that the inspected method directly overrides.

```
precursor M := overrides M, M declared_by T, overrides: {
    all is M | declared_by S, T substypes+ S;
  };
```

  SEE ALSO: *overrides*, *subtypes+*

- ***primitive***

  SIGNATURE: ELEMENT

  Select primitive elements. Holds if the inspected value is one of the primitive types; or it is a field of a primitive type; or it is a method returning a primitive. This is equivalent to:

```
primitive := boolean | byte | short | int | long
  | float | double | void;
```

  SEE ALSO: *boolean*, *byte*, *short*, *int*, *long*, *float*, *double*, *void*

- ***private***

  SIGNATURE: ELEMENT

  The inspected value is private.

- ***protected***

  SIGNATURE: ELEMENT

  The inspected value is protected.

- ***public***

  SIGNATURE: ELEMENT

  The inspected value is public.

- ***put_field M,S***

  SIGNATURE: SCRATCH,MEMBER,SCRATCH

  COMPUTABILITY: $\# \rightarrow M, S$ [or] $S \rightarrow \#, M$

  Obtain written to field and the written value.

  If the inspected scratch was the receiver in a field assignment operation (via the JVM's `putfield` or `putstatic` instructions) select that field into `M` and select the stored scratch into `S`.

  In assignment to static fields the system produces a "fake scratch" that plays the role of the receiver. This allows `put_field` to handle assignments to both kinds of fields (static and non-static). A JTL query can distinguish the two by applying the *static* query on `put_field`'s `F` parameter.

  Example 1. The following JTL expression finds all methods (of the inspected class) that assign a value into a field.

```
class {
  method is M {
    put_field[F,_];
  };
};
```

When this query is evaluated against the following input class

```
public class SomeClass {
  private int x;
  public void setX(int arg) { x = arg; }
  public int getX() { return x; }
  public void reset() { x = 0; }
  public void copyTo(SomeClass that) { that.x = this.x; }
}
```

we get three pairs of results for `M`, `F` (respectively):

`SomeClass#setX(int)`, `SomeClass#x`

`SomeClass#reset()`, `SomeClass#x`

`SomeClass#copyTo(SomeClass)`, `SomeClass#x`

Example 2. We now want to find only "setter" methods. That is: methods that assign to fields of the current object. To do that we pass the receiver scratch through the *this* predicate, and the value scratch (the scratch assigned the field) through the *parameter* predicate:

```
class {
  method is M {
    this put_field[F,V], V parameter;
  };
};
```

Evaluating this expression against `SomeClass` we get a single pair of results for `M`, `F`:

`SomeClass#setX(int)`, `SomeClass#x`

In this execution the `copyTo()` method failed to pass the *this* condition. The `reset()` method failed to pass the *parameter* condition.

SEE ALSO: *get_field*, *receives*

- ***put_method M***

  SIGNATURE: SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Select the methods to which the subject scratch is passed as a parameter.

  In the following method

  ```
  public void f() { g(5); h(); }
  ```

  The scratch that corresponds to the constant value `5` is passed to the method `g()`.

  Evaluating the JTL expression `method { put_method M; }` with `f()` as an input, selects `g()` into `M` because there is a scratch that is passed in the `g()` call. `h()` is not selected because no scratch that is passed to it.

  SEE ALSO: *get_method*, *parameter*

- ***reads F***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow F$

  Select into `F` the fields read by the subject method/constructor or by methods/constructors of the subject class.

  Defined as:

```
reads F := class offers M, M reads F | {
     get_field [F,_]
  };
```

  SEE ALSO: *accesses*, *writes*

- ***receiver_get M***

  SIGNATURE: SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain the field that was read-from with the subject scratch being the receiver.

  A `getfield` instruction is the JVM instruction that fetches the value of a non-static field. It uses a scratch to obtain a reference to the object holding the field.

  SEE ALSO: *get_method*, *put_method*, *receiver_put*, *get_field*

- ***receiver_interface M***

  SIGNATURE: SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain interface method invoked on the subject scratch.

  This query allows low-level examination of method invocations and is thus sensitive to subtle issues related to byte code generation. Users are advised to use the *receives* query which is indifferent to these subtleties.

  `receiver_interface` is satisfied if the subject was used as the receiver in an `invokeinterface` instruction. The invoked method is assigned into `M`.

  method invocations.

  SEE ALSO: *receiver_special*, *receiver_virtual*, *get_method*, *put_method*, *receives*

- ***receiver_put M***

  SIGNATURE: SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain the field that was written-to with the subject scratch being the receiver.

  A `putfield` instruction is the JVM instruction that fetches the value of a non-static field. It uses a scratch to obtain a reference to the object holding the field.

  SEE ALSO: *get_method*, *put_method*, *receiver_get*, *put_field*

- ***receiver_special M***

  SIGNATURE: SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain method invoked on the subject scratch.

This query allows low-level examination of method invocations and is thus sensitive to subtle issues related to byte code generation. Users are advised to use the *receives* query which is indifferent to these subtleties.

`receiver_special` is satisfied if the insepcted scratch was used as the receiver in an `invokespecial` instruction. The invoked method is assigned into `M`.

SEE ALSO: *get_method*, *put_method*, *receives*

- ***receiver_virtual M***

  SIGNATURE: SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain an instance method invoked on the subject scratch.

  This query allows low-level examination of method invocations and is thus sensitive to subtle issues related to byte code generation. Users are advised to use the *receives* query which is indifferent to these subtleties.

  `receiver_virtual` is satisfied if the insepcted scratch was used as the receiver in an `invokevirtual` instruction. The invoked method is assigned into `M`.

  SEE ALSO: *get_method*, *put_method*, *receives*

- ***receives M***

  SIGNATURE: SCRATCH,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Obtain method invoked on the subject scratch.

  If a method was invoked on the inspected scratch select that method into `M`. This query captures method calls made by these JVM instructions: `invokevirtual`, `invokeinterface` or `invokespecial`.

  Example 1. Evaluating the JTL expression

```
class {
  method {
    receives M;
  };
};
```

  over this input

```
public static class SomeClass {
  public void g() { }
  public void f(List<String> items) {
    g();
    items.add("a");
  }
}
```

  Produces the following result:

```
SomeClass#g()
java.util.List#add(java.lang.Object)
```

  Example 2. This example is similar to the former but it imposes one additional requirement: the receiver scratch must now satisfy the **`this`** query. This will reject the `List.add()` call as is its receiver is `items` rather than **`this`**.

Specifically, evaluating

```
class {
   method {
      this receives M;
   };
};
```

over the input `SomeClass` yields the following output:

```
SomeClass#g()
```

SEE ALSO: *calls*, *receiver_special*, *receiver_interface*, *receiver_virtual*

- ***returned***

  SIGNATURE: SCRATCH

  The subject scratch is returned from the method.

  Satisfied if there is a series of assignments from the inspected scratch to the method's **`return`** instruction.

  The following JTL predicate

  ```
  parameter_returning := method {
     returned parameter;
  };
  ```

  Matches methods where there is a *parameter* that is returned from the method (possibly via a chain of assignments).

  SEE ALSO: *parameter*, *local_var*, *this*, *constant*

- ***same_args M***

  SIGNATURE: MEMBER,MEMBER

  Compare argument lists.

  Holds if both the subject and `M` are methods and the their argument lists are identical (pair-wise comparison of types).

  SEE ALSO: *same_name*

- ***same_name M***

  SIGNATURE: MEMBER,MEMBER

  Compare names. Holds if both the subject and `M` have the same name.

  SEE ALSO: *same_args*

- ***scratch***

  SIGNATURE: SCRATCH

  The inspected value is a scratch.

- ***scratches S***

  SIGNATURE: MEMBER,SCRATCH

  COMPUTABILITY: $\# \rightarrow S$ [or] $S \rightarrow \#$

  Obtain the scratches of a method. Selects into `S` all scratches of the subject method.

  SEE ALSO: *scratch*

- ***ser_ver_uid***

  SIGNATURE: MEMBER

  The inspected element is the `serialVersionUID` field prescribed by JAVA's serialization mechanism. Defined as:

  ```
  ser_ver_uid := static final long 'serialVersionUID;
  ```

- ***setter F***

  SIGNATURE: MEMBER,MEMBER

  The subject is a setter method of the `F` field. A setter method is a void returning *mutator* that takes a single parameter and assigns it to a field of the receiving object.

  ```
  setter F := instance method (_) {
      this put_field[F,V], V parameter;
    };
  ```

  SEE ALSO: *getter*, *mutator*, *put_field*

- ***signature_compatible M***

  SIGNATURE: MEMBER,MEMBER

  The subject is a method whose signature is compatible to that of `M`. This means that the list of types of the formal parameters of # and `M` are pair-wise identical, and that the return type of # is identical to the return type of `M` or is a subtype thereof.

  SEE ALSO: *overrides*

- ***short***

  SIGNATURE: MEMBER

  Select short elements. Holds if the inspected value is the primitive type **`short`**; or it is a field of type **`short`**; or it is a method with return type **`short`**.

- ***static***

  SIGNATURE: ELEMENT

  The inspected value is static.

- ***static_initializer***

  SIGNATURE: MEMBER

  The inspected value is a static initializer. A static initializers is a JAVA block that is executed once when the class is loaded:

  ```
  class A {
    static int n;
    static { n = new Random().nextInt(); }
  }
  ```

- ***strictfp***

  SIGNATURE: ELEMENT

  The subject is declared as **`strictfp`**.

- ***subtypes T***

  SIGNATURE: TYPE,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  The subject type either implements or extends `T`.

  ```
  subtypes T := extends T | implements T;
  ```

  SEE ALSO: *extends*, *implements*, *subtypes**, *subtypes+*

- ***subtypes+ T***

  SIGNATURE: TYPE,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  Recursive application of `subtypes`. Defined as

  ```
  subtypes+ T := subtypes T | subtypes T', T' subtypes+ T;
  ```

  SEE ALSO: *subtypes*, *subtypes**

- ***subtypes* T***

  SIGNATURE: TYPE,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  Produces the same result as `subtypes+` but includes the subject itself in the result. Defined as:

  ```
  subtypes* T := is T | subtypes+ T;
  ```

  SEE ALSO: *subtypes*, *subtypes+*

- ***synchronized***

  SIGNATURE: ELEMENT

  The subject is declared as **`synchronized`**.

  SEE ALSO: *plain*, *native*

- ***synthetic***

  SIGNATURE: ELEMENT

  The subject is declared carries the synthetic modifier.

- ***this***

  SIGNATURE: SCRATCH

  The subject scratch is the **`this`** parameter.

  Satisfied if there is a series of assignments from the method's **`this`** parameter to the inspected scratch.

  The following JTL predicate

  ```
  fluent_method := method {
    returned this;
  };
  ```

  Matches methods where there is a *returned* scratch that is assigned from the **`this`** variable (possibly through a chain of assignments).

  SEE ALSO: *parameter*, *local_var*, *from*, *returned*, *receives*, *put_field*, *get_field*

- ***throws T***

  SIGNATURE: MEMBER,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  Obtain the exceptions on a method's **`throws`** clause.

  Selects into `T` the exceptions that are mentioned in the inspected method's **`throws`** clause. This implies that `T` will be a subclass of `java.lang.Throwable`.

- ***transient***

  SIGNATURE: MEMBER

  The inspected value is **`transient`**.

- ***true***

  SIGNATURE: ANY

  The truth predicate. Holds for every subject value. SEE ALSO: *false*

- ***type***

  SIGNATURE: TYPE

  Holds if the subject is a type.

- ***typed T***

  SIGNATURE: TYPE,TYPE [or] MEMBER,TYPE [or] SCRATCH,TYPE

  COMPUTABILITY: $\# \rightarrow T$

  Obtain the type of the subject, according to these rules:

  If inspecting a type select the inspected type into T

  If inspecting a field select the field's type into T

  If inspecting a method select the method's return type into T

  If inspecting a scratch select the scratch's type into T

- ***uses M***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow M$

  Select into M the fields that are accessed by the subject or the methods that are called by it.

  Defined as:

  ```
  uses M := accesses M | calls M;
  ```

  SEE ALSO: *accessses, calls*

- ***varargs***

  SIGNATURE: MEMBER

  The subject is a method taking a variable number of arguments

- ***visible***

  SIGNATURE: ELEMENT

  The inspected element is not **`private`**.

  SEE ALSO: *private*

- ***void***

  SIGNATURE: MEMBER

  Select void elements. Holds if the subject is the primitive type **`void`**; or it is a field of type **`void`**; or it is a method with return type **`void`**.

- ***volatile***

  SIGNATURE: MEMBER

  The subject is declared as **`volatile`**.

- ***writes F***

  SIGNATURE: TYPE,MEMBER [or] MEMBER,MEMBER

  COMPUTABILITY: $\# \rightarrow F$

  Select into `F` the fields written by the subject method/constructor or by methods/constructors of the subject class.

  Defined as:

  ```
  reads F := class offers M, M writes F | {
      put_field[F,_]
    };
  ```

  SEE ALSO: *accesses*, *reads*

# Appendix C

# Micro Pattern Catalog—Addendum

## Limited Self

A class that the methods it invokes on itself are only those that are inherited or overridden. In addition, the class has no instance fields.

### Definition

```
limited_self := class is T extends S {
     let self_invokes M := calls M, M declared_by T;
     self_invokes M implies S declares M',
       M same_name M', M signature_compatible M';
     no instance field;
   };
```

### Purpose

A class realizing this pattern captures an extension to the behavior of the super-class. Typically, such classes can be refactored into the DECORATOR design pattern, thus allowing the extension to be selectively applied to pre-existing objects.

### Example

```
package java.awt.event;

public class TextEvent extends AWTEvent {

   // static fields, constructors ...

   public String paramString() {
      // id is an inherited protected field
      return id == TEXT_VALUE_CHANGED ? "TEXT_VALUE_CHANGED"
        : "unknown type";
   }
}
```

### Prevalence

28.0%

## Recursive

A class that has at least one instance fields whose type is the same as that of the class.

### Definition

```
recursive := class is T {
     instance field typed T;
   };
```

### Purpose

This pattern is typically used in implementation of (recursive) data structures where a node is linked with at-least one more similarly typed node.

### Example

```
package java.util;

public abstract class ResourceBundle {

   protected ResourceBundle parent = null;

   private Locale locale = null;
   private String name;
   private volatile boolean expired;

   ...

}
```

### Prevalence

1.0%

# Bibliography


[1] ACM Press, New York, NY, USA. *Proc. of the Sixteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'01)*, Tampa Bay, Florida, October 14–18 2001. ACM SIGPLAN Notices 36(11).

[2] Ellen Agerbo and Aino Cornils. How to preserve the benefits of design patterns. In OOPSLA'98 [154], pages 134–143.

[3] Alfred Vaino Aho, Brian W. Kernighan, and Peter J. Weinberger. *The AWK programming language*. Addison-Wesley series in Computer Science. Addison-Wesley Publishing Company, Reading, Massachusetts, 1988.

[4] Alfred Vaino Aho, Ravi Sethi, and Jeffry D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1986.

[5] Mehmet Akşit and Satushi Matsuoka, editors. *Proc. of the Eleventh European Conference on Object-Oriented Programming (ECOOP'97)*, volume 1241 of *Lecture Notes in Computer Science*, Jyväskylä, Finland, June 9-13 1997. Springer Verlag.

[6] Jonathan Erik Aldrich and Craig Chambers. Ownership domains: Separating aliasing policy from mechanisms. In Odersky [151], pages 1–25.

[7] Jonathan Erik Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proc. of the Seventeenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, pages 311–330, Seattle, Washington, November 4–8 2002. ACM SIGPLAN Notices 37(11).

[8] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A pattern language: towns, buildings, construction*. Center for Environmental Structure series. Oxford University Press, 1977.

[9] Eric Allen, Jonathan Bannet, and Robert Cartwright. A first-class approach to genericity. In Crocker and Jr. [60], pages 96–114.

[10] B. Alpern, M. N. Wegman, and F. K. Zadeck. Detecting Equality of Variables in Programs. In *Proceedings of the 15th symposium on Principles Of Programming Languages (POPL'88)*, pages 1–11, New York, NY, USA, 1988. ACM.

[11] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[12] Davide Ancona, Giovanni Lagorio, and Elena Zucca. Jam—designing a Java extension with mixins. *ACM Transactions on Programming Languages and Systems*, 25(5):641–712, 2003.

[13] Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. In Tarr and Cook [175].

[14] Tim Andrews and Craig Harris. Combining language and database advances in an object-oriented development environment. In Norman K. Meyrowitz, editor, *Proc. of the Second Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, pages 430–440, Orlando, Florida, October 4-8 1987. ACM SIGPLAN Notices 22(12).

[15] Michal Antkiewicz, Thiago T. Bartolomei, and Krzysztof Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *22nd IEEE/ACM International Conference on Automated Software Engineering*, pages 214–223. ACM, November 2007.

[16] Giuliano Antoniol, Massimiliano Di Penta, and Ettore Merlo. YAAB (Yet Another AST Browser): Using OCL to navigate ASTs. In IWPC'03 [110], pages 13–22.

[17] *Proc. of the Second International Conference on Aspect-Oriented Software Development (AOSD'03)*, Boston, Massachusetts, USA, March 17-21 2003. ACM Press, New York, NY, USA.

[18] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison-Wesley Publishing Company, Reading, Massachusetts, 1996.

[19] Darren C. Atkinson and William G. Griswold. The design of whole-program analysis tools. In *Proc. of the Eighteenth International Conference on Software Engineering (ICSE'96)*, pages 16–27, Berlin, Germany, March 25-30 1996.

[20] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

[21] Malcolm P. Atkinson and Ray Welland. *Fully Integrated Data Env.: Persistent Prog. Lang., Object Stores, and Prog. Env.* Springer Verlag, New York, Secaucus, NJ, USA, 2000.

[22] Douglas M. Auclair. Aspect-oriented programming in prolog, December 2005. http://www.bigzaphod.org/whirl/dma/docs/aspects/aspects-man.html.

[23] Sushil Bajracharya, Trung Ngo, Erik Linstead, Yimeng Dou, Paul Rigor, Pierre Baldi, and Cristina Lopes. Sourcerer: a Search Engine for Open Source Code Supporting Structure-Based Search. In *Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications (OOPSLA'06), October 22-26, 2006, Portland, Oregon, USA*, pages 681–682. ACM Press, New York, NY, USA, 2006.

[24] Robert Balzer, Neil M. Goldman, and David S. Wile. On the transformational implementation approach to programming. In *Proc. of the Second International Conference on Software Engineering (ICSE'76)*, pages 337–344, San Francisco, California, United States, 1976. IEEE Computer Society Press.

[25] Larry A. Barowski and James H. Cross II. Extraction and use of class dependency information for Java. In *Proc. of the Ninth Working Conference on Reverse Engineering (WCRE'02)*, pages 309–318, Richmond, Virginia, USA, October 2002. IEEE Computer Society Press.

[26] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. In *Proc. of the Sixth USENIX C++ Conference*, pages 37–56, Cambridge, MA, April 1994. USENIX Association.

[27] Gerald Baumgartner and Vincent F. Russo. Implementing signatures for C++. *ACM Transactions on Programming Languages and Systems*, 19(1):153–187, January 1997.

[28] Gareth Baxter, Marcus R. Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, and Ewan D. Tempero. Understanding the Shape of Java Software. In Tarr and Cook [175], pages 397–412.

[29] Kent Beck. *Smalltalk: best practice patterns*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, first edition, 1997.

[30] Kent Beck. *JUnit Pocket Guide*. O'Reilly, 2004.

[31] Kent Beck. *Implementation Patterns*. Addison-Wesley, Upper Saddle River, NJ, USA, 2006.

[32] Kent Beck. *Implementation Patterns*. Addison-Wesley Professional, 2007.

[33] L. Bendix, A. Dattolo, and F. Vitali. Software configuration management in software and hypermedia engineering: A survey. In *Handbook of Software Engineering and Knowledge Engineering*, volume 1, pages 523–548. World Scientific Publishing, 2001.

[34] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In John M. Vlissides and Douglas C. Schmidt, editors, *Proc. of the Nineteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'04)*, pages 35–49, Vancouver, BC, Canada, October 2004. ACM SIGPLAN Notices 39 (10).

[35] Alex Blewitt, Alan Bundy, and Ian Stark. Automatic verification of Java design patterns. In *Proc. of the Sixteenth IEEE Conference on Automated Software Engineering (ASE'01)*, pages 324–327, San Diego, California, 2001. IEEE Computer.

[36] Scott Boag, Don Chamberlin, Mary F. Fernández, Daniela Florescu, Jonathan Robie, and Jérôme Siméon. *XQuery 1.0: An XML Query Language*. W3C, 2005.

[37] E. Börger, E. Grädel, and Y. Gurevich. *The Classical Decision Problem*. Perspectives of Mathematical Logic. Springer Verlag, 1997.

[38] Gilad Bracha. *The Programming Language Jigsaw: Mixins, Modularity and Multiple Inheritance*. PhD thesis, Department of Computer Science, University of Utah, 1992.

[39] Gilad Bracha and William R. Cook. Mixin-based inheritance. In Norman K. Meyrowitz, editor, *Proc. of the Fifth Object-Oriented Programming Systems, Languages, and Applications / European Conference on Object-Oriented Programming OOPSLA/ECOOP'90*, pages 303–311, Ottawa, Canada, October 21-25 1990. ACM SIGPLAN Notices 25(10).

[40] Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In OOPSLA'98 [154], pages 183–200.

[41] Johan Brichau, Andy Kellens, Kris Gybels, Kim Mens, Robert Hirschfeld, and Theo D'Hondt. Application-specific models and pointcuts using a logic meta language. In *LNCS*, Prague, Czech Republic, 2006.

[42] Kyle Brown. *Design Reverse-Engineering and Automated Design Pattern Detection in Smalltalk*. Masters thesis, North Carolina State University, 1996.

[43] Avi Bryant, Andrew Catton, Kris De Volder, and Gail C. Murphy. Explicit programming. In *Proc. of the First International Conference on Aspect-Oriented Software Development (AOSD'02)*. ACM Press, New York, NY, USA, 2002.

[44] Martin Büchi and Wolfgang Weck. Compound types for java. In OOPSLA'98 [154], pages 362–373.

[45] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*. John Wiley & Sons, August 1996.

[46] David N. Card and Robert L. Glass. *Measuring software design quality*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, Englewood Cliffs, New Jersy, 1990.

[47] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic programming and databases*. Springer Verlag, New York, 1990.

[48] Patrice Chalin and Perry R. James. Non-null references by default in Java: Alleviating the nullity annotation burden. In Erik Ernst, editor, *Proc. of the Twenty First European Conference on Object-Oriented Programming (ECOOP'07)*, volume 4609 of *Lecture Notes in Computer Science*, pages 227–247, Berlin, Germany, July/August 2007. Springer Verlag.

[49] Craig Chambers. Object-oriented multi-methods in Cecil. In Ole Lehrmann Madsen, editor, *Proc. of the Sixth European Conference on Object-Oriented Programming (ECOOP92)*, volume 615 of *Lecture Notes in Computer Science*, pages 33–56, Utrecht,the Netherlands, June29–July3 1992. Springer Verlag.

[50] Yih-Farn Chen, Michael Nishimoto, and C.V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[51] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[52] Sara Cohen, Yossi (Joseph) Gil, and Evelina Zarivach. Datalog programs over infinite databases, revisited. In *Database Programming Languages, 11th International Symposium, DBPL 2007*, pages 32–47, 2007.

[53] Tal Cohen and Joseph Gil. Self-calibration of metrics of Java methods. In *Proc. of the Thirty Seventh International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'00 Pacific)*, pages 94–106, Sydney, Australia, November 20-23 2000. Prentice-Hall, Englewood Cliffs, New Jersy 07632.

[54] Tal Cohen and Joseph Gil. AspectJ2EE = AOP + J2EE: Towards an aspect based, programmable and extensible middleware framework. In Odersky [151], pages 219–243.

[55] Tal Cohen, Joseph Gil, and Itay Maman. Guarded Program Transformations Using JTL. In Richard F. Paige and Bertrand Meyer, editors, *Proc. of the Forty Sixth Conference on Objects, Models, Components, Patterns (TOOLS EUROPE 2008)*, volume 11 of *Lecture Notes in Business Information Processing*, pages 100–120, Zurich, Switzerland, June 2008. Springer Verlag.

[56] Tal Cohen and Joseph (Yossi) Gil. Shakeins: Nonintrusive aspects for middleware frameworks. *Transactions on Aspect-Oriented Software Development II*, November 2006.

[57] Tal Cohen, Joseph (Yossi) Gil, and Itay Maman. JTL—the Java tools language. In Tarr and Cook [175].

[58] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms*. McGraw Hill and MIT Press, first edition, June 1990.

[59] Roger F. Crew. ASTLOG: A language for examining abstract syntax trees. In S. Kamin, editor, *Proc. of the First USENIX Conference Domain Specific Languages (DSL'97)*, pages 229–242, Santa Barbara, October 1997.

[60] Ron Crocker and Guy L. Steele Jr., editors. *Proc. of the Eighteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'03)*, Anaheim, California, USA, October 2003. ACM SIGPLAN Notices 38 (11).

[61] Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Publishing Company, June 2000.

[62] Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-Oriented Queries Made Easy. In *Generative and Transformational Techniques in Software Engineering II*, volume 5235 of *Lecture Notes in Computer Science*, pages 78–133. Springer, 2007.

[63] Kris De Volder. *Type-Oriented Logic Meta Programming*. PhD thesis, Vrije Universiteit Brussel, 1998. Adv: Prof. Dr. Theo D'Hondt.

[64] Kris De Volder and Theo D'Hondt. Aspect-oriented logic meta programming. In *Proc. of the Ninetieth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 1616 of *Lecture Notes in Computer Science*, pages 250–272. Springer Verlag, 1999.

[65] Tom DeMarco. Software Engineering: An Idea Whose Time Has Come and Gone? *IEEE Software*, 26(4):96–95, 2009.

[66] Pierre Deransart, Laurent Cervoni, and AbdelAli Ed-Dbali. *Prolog: The Standard: reference manual*. Springer-Verlag, London, UK, 1996.

[67] Premkumar T. Devanbu. GENOA—a customizable, front-end-retargetable source code analysis framework. *ACM Trans. on Soft. Eng. and Methodology*, 8(2):177–212, 1999.

[68] Edsger W. Dijkstra. Guarded commands, non-determinancy and a calculus for the derivation of programs. In Friedrich L. Bauer and Klaus Samelson, editors, *Language Hierarchies and Interfaces*, volume 46 of *LNCS*, pages 111–124, Marktoberdorf, Germany, July 1975. Springer Verlag.

[69] Gilles Dubochet and Martin Odersky. Compiling structural types on the JVM: a comparison of reflective and generative techniques from Scala's perspective. In *Proc. of the Fourth Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems (ICOOOLPS '09)*, pages 34–41. ACM, 2009.

[70] ECMA International. Common Language Infrastructure (CLI) Partitions I to VI, 3rd edition. Technical report, ECMA, Rue du Rhône 114, CH-1204, Geneva, Jun 2005.

[71] Amnon H. Eden. Formal specification of object-oriented design. In *Proc. of the International Conference on Multidisciplinary Design in Engineering (CSME-MDE'01)*, Montreal, Canada, November 21-22 2001.

[72] Amnon H. Eden. A visual formalism for object-oriented architecture. In *Proc. of the Sixth Biennial World Conference on Integrated Design and Process Technology (IDPT'02)*, California, June 23-28 2002. Society for Design and Process Science.

[73] Michael Eichberg, Mira Mezini, Klaus Ostermann, and Thorsten Schäfer. XIRC: A kernel for cross-artifact information engineering in software development environments. In *Proc. of the Eleventh Working Conference on Reverse Engineering (WCRE'04)*, pages 182–191, Delft, Netherlands, November 8-12 2004. IEEE Computer Society Press.

[74] Stephen G. Eick, Joseph L. Steffen, and Eric E. Jr. Sumner. Seesoft-a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[75] Manuel Fähndrich and K. Rustan M. Leino. Declaring and checking non-null types in an object-oriented language. In Crocker and Jr. [60], pages 302–312.

[76] Gert Florijn, Marco Meijers, and Pieter van Winsen. Tool support for object-oriented patterns. In Akşit and Matsuoka [5], pages 472–495.

[77] Martin Fowler, Kent Beck, John Brant, William Opdyke, and Don Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[78] Pascal Fradet and Mario Südholt. An aspect language for robust programming. In Ana M. D. Moreira and Serge Demeyer, editors, *Proc. of the ECOOP'99 Workshops, Panels, and Posters*, volume 1743 of *Lecture Notes in Computer Science*, pages 291–292, Lisbon, Portugal, June 1999. Springer Verlag.

[79] Richard P. Gabriel, Linda Northrop, Douglas C. Schmidt, and Kevin Sullivan. Ultra-large-scale systems. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 632–634. ACM Press, New York, NY, USA, 2006.

[80] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar M. Nierstrasz, editor, *Proc. of the Seventh European Conference on Object-Oriented Programming (ECOOP'93)*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431, Kaiserslautern, Germany, July 26-30 1993. Springer Verlag.

[81] Erich Gamma, Richard Helm, Ralph E. Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Professional Computing series. Addison-Wesley Publishing Company, Reading, Massachusetts, 1995.

[82] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, and Jeremiah Willcock. A comparative study of language support for generic programming. In Crocker and Jr. [60], pages 115–134.

[83] Joseph Gil and Keren Lenz. The Use of Overloading in Java Programs. In *Proceedings of the 24th European Conference on Object-Oriented Programming*, volume 6183 of *Lecture Notes in Computer Science*, pages 529–551. Springer, June 2010.

[84] Joseph Gil and Itay Maman. Micro patterns in Java code. In Johnson and Gabriel [112], pages 97–116.

[85] Joseph Gil and Itay Maman. Whiteoak: Introducing Structural Typing into Java. In Harris [102], pages 73–90.

[86] Joseph Gil and Tali Shragai. Are We Ready for a Safer Construction Environment? In *Proceedings of the 23rd European Conference on Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 495–519. Springer, July 2009.

[87] Joseph Gil and Peter F. Sweeney. Space- and time-efficient memory layout for multiple inheritance. In *Proc. of the Fourteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'99)*, pages 256–275, Denver, Colorado, November1–5 1999. ACM Press, New York, NY, USA, ACM SIGPLAN Notices 34 (10).

[88] Joseph Gil and Yuri Tsoglin. JAMOOS—a domain-specific language for language processing. *J. Comp. and Inf. Tech.*, 9(4):305–321, 2001.

[89] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1984.

[90] Simon Goldsmith, Robert O'Callahan, and Alex Aiken. Relational queries over program traces. In Johnson and Gabriel [112], pages 385–402.

[91] Ian Gorton and Liming Zhu. Tool support for *Just-in-Time* architecture reconstruction and evaluation: An experience report. In Roman et al. [161], pages 514–523.

[92] James Gosling, Bill Joy, Guy L. Jr. Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, June 2005.

[93] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. In *Proc. of the First International Conference on Requirements Engineering (ICRE'94)*, pages 94–101, Colorado Springs, Colorado, April 1994. IEEE Computer Society Press.

[94] Paul Graham. *ANSI Common LISP*. Prentice Hall, 1995.

[95] Mark Grand. *Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with Uml,Volume 1*. John Wiley & Sons, New-York, 2002.

[96] Judith E. Grass and Yih-Farn Chen. The C++ information abstractor. In *Proc. of the USENIX C++ Conference*, pages 265–277, San Fransisco, CA, April 1990. AT&T Bell Laboratories, USENIX Association.

[97] William G. Griswold, Darren C. Atkinson, and Collin McCurdy. Fast, flexible syntactic pattern matching and processing. In *Proc. of the Fourth Workshop on Program Comprehension (WPC '96)*, pages 144–153, Washington, DC, 1996. IEEE Computer Society Press.

[98] Kris Gybels and Johan Brichau. Arranging language features for more robust pattern-based crosscuts. In AOSD'03 [17], pages 60–69.

[99] Kris Gybels and Andy Kellens. An experiment in using inductive logic programming to uncover pointcuts. In *European Interactive Workshop on Aspects in Software (EIWAS'04)*, Berlin, Germany, September23-24 2004.

[100] A. N. Habermann and David Notkin. Gandalf: Software development environments. *IEEE Transactions on Software Engineering*, 12(12):1117–1127, December 1986.

[101] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. CodeQuest: Scalable source code queries with Datalog. In Dave Thomas, editor, *Proc. of the Twentieth European Conference on Object-Oriented Programming (ECOOP'06)*, volume 4067 of *Lecture Notes in Computer Science*, Nantes, France, July 3–7 2006. Springer Verlag.

[102] Gail E. Harris, editor. *Proc. of the Twenty Third Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'08)*, Nashville, TN, October 19-23 2008. ACM SIGPLAN Notices.

[103] William Harrison, David Lievens, and Tim Walsh. Using recombinance to improve modularity. Technical Report 104 Software Structures Group, Trinity College Dublin, Dublin, Ireland, March 2007.

[104] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, October 2003.

[105] Dirk Heuzeroth, Thomas Holl, Gustav Höström, and Welf Löwe. Automatic design pattern detection. In IWPC'03 [110], page 94.

[106] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In Roman et al. [161], pages 117–125.

[107] Haruo Hosoya, Alain Frisch, and Giuseppe Castagna. Parametric polymorphism for XML. In Jens Palsberg and Martín Abadi, editors, *Proc. of the Thirty Second ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*, pages 50–62. ACM Press, New York, NY, USA, 2005.

[108] Einar W. Høst and Bjarte M. Østvold. Debugging method names. In Sophia Drossopoulou, editor, *Proc. of the Twenty Third European Conference on Object-Oriented Programming (ECOOP'09)*, volume 5653 of *Lecture Notes in Computer Science*, pages 294–317, Genoa, Italy, July 2009. Springer Verlag.

[109] ISE. *ISE EIFFEL The Language Reference*. ISE, Santa Barbara, CA, 1997.

[110] *Proc. of the Eleventh International Workshop on Program Comprehension (IWPC'03)*, Portland, Oregon, USA, May 10-11 2003.

[111] Doug Janzen and Kris De Volder. Navigating and querying code without getting lost. In *Proc. of the Second international conference on Aspect-Oriented Software Development (AOSD'03)*, pages 178–187, New York, NY, USA, 2003. ACM Press.

[112] Ralph Johnson and Richard P. Gabriel, editors. *Proc. of the Twentieth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'05)*, San Diego, California, October 2005. ACM SIGPLAN Notices.

[113] Simon Peyon Jones. *Haskell 98 Language and Libraries: The Revisited Report*. Cambridge University Press, 2003.

[114] Bo Nørregaard Jørgensen. Integration of independently developed components through aliased multi-object type widening. *Journal of Object Technology*, 3(11):55–76, 2004.

[115] Jevgeni Kabanov and Rein Raudjärv. Embedded typesafe domain specific languages for Java. In *Proceedings of the 6th international symposium on Principles and Practice of Programming in Java (PPPJ'08)*, pages 189–197, New York, NY, USA, 2008. ACM.

[116] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Software Series. Prentice-Hall, Englewood Cliffs, New Jersy 07632, second edition, 1988.

[117] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In Jørgen Lindskov Knudsen, editor, *Proc. of the Fifteenth European Conference on Object-Oriented Programming (ECOOP'01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–355, Budapest, Hungary, June 2001. Springer Verlag.

[118] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Akşit and Matsuoka [5], pages 220–242.

[119] Sunghun Kim, Kai Pan, and E. James Jr. Whitehead. Micro Pattern Evolution. In *Proceedings of the 2006 international workshop on Mining Software Repositories (MSR'06), May 22-23, 2006, Shanghai, China*, pages 40–46. ACM Press, New York, NY, USA, 2006.

[120] Donald Ervin Knuth. Structured Programming with go to Statements. *ACM Computing Surveys (CSUR)*, 6(4):261–301, 1974.

[121] Kostas Kontogiannis, Johannes Martin, Kenny Wong, Richard Gregory, Hausi A. Müller, and John Mylopoulos. Code migration through transformations. In Stephen A. MacKay and J. Howard Johnson, editors, *Proc. of the Conference of the Centre for Advanced Studies on Collaborative research (CASCON'98)*, page 13, Toronto, Ontario, Canada, November 1998. IBM Press.

[122] Ralf Lammel. Declarative aspect-oriented programming. In *Partial Evaluation and Semantic-Based Program Manipulation*, pages 131–146, 1999.

[123] Michele Lanza and Stéphane Ducasse. A categorization of classes based on the visualization of their internal structure: the class blueprint. In OOPSLA'01 [1], pages 300–311.

[124] Anthony Lauder and Stuart Kent. Precise visual specification of design patterns. In Eric Jul, editor, *Proc. of the Twelfth European Conference on Object-Oriented Programming (ECOOP'98)*, volume 1445 of *Lecture Notes in Computer Science*, pages 114–134, Brussels, Belgium, July 20–24 1998. Springer Verlag.

[125] Konstantin Läufer, Gerald Baumgartner, and Vincent F. Russo. Safe structural conformance for Java. *The Computer Journal*, 43(6):469–481, 2001.

[126] Rasmus Jay Lerdorf, Kevin Tatroe, Bob Kaehms, and Ric McGredy. *Programming PHP*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, March 2002.

[127] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, Reading, Massachusetts, second edition, 1999.

[128] Roberto Lopez-Herrejon, Don Batory, and Christian Lengauer. A disciplined approach to aspect composition. In *Proc. of the ACM SIGPLAN Simposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'06)*, pages 68–77, Charleston, South Carolina, 2006. ACM Press, New York, NY, USA.

[129] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: a practical guide*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, Englewood Cliffs, New Jersy, 1994.

[130] Jeffrey K. H. Mak, Clifford S. T. Choy, and Daniel P. K. Lun. Precise modeling of design patterns in UML. In *Proc. of the Twenty Sixth International Conference on Software Engineering (ICSE'04)*, pages 252–261, Edinburgh, Scotland, United Kingdom, May 23-28 2004. IEEE Computer Society Press.

[131] Donna Malayeri and Jonathan Aldrich. Integrating nominal and structural subtyping. In Jan Vitek, editor, *Proc. of the Twenty Second European Conference on Object-Oriented Programming (ECOOP'08)*, volume 5142 of *Lecture Notes in Computer Science*, pages 260–284, Paphos, Cyprus, July 7–11 2008. Springer Verlag.

[132] Donna Malayeri and Jonathan Aldrich. Is Structural Subtyping Useful? An Empirical Study. In Giuseppe Castagna, editor, *Proc. of the Eighteenth European Symposium on Programming (ESOP 2009)*, volume 5502 of *Lecture Notes in Computer Science*, pages 95–111, York, United Kingdom, March 2009. Springer Verlag.

[133] Sebastien Marion, Richard Jones, and Chris Ryder. Decrypting the Java Gene Pool. In *Proceedings of the 6th International Symposium on Memory Management (ISMM'07), October 21-22, 2007, Montreal, Quebec, Canada*, pages 67–78. ACM Press, New York, NY, USA, 2007.

[134] Michael Martin, Benjamin Livshits, and Monica S. Lam. Finding application errors and security flaws using PQL: a program query language. In Johnson and Gabriel [112], pages 365–383.

[135] Bertrand Meyer. *Object-Oriented Software Construction*. International Series in Computer Science. Prentice-Hall, Englewood Cliffs, New Jersy 07632, 1988.

[136] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Englewood Cliffs, New Jersy 07632, Englewood Cliffs, New Jersy, second edition, 1997.

[137] Mira Mezini and Klaus Ostermann. Conquering aspects with Caesar. In AOSD'03 [17], pages 90–100.

[138] Tommi Mikkonen. Formalizing design patterns. In *Proc. of the Twentieth International Conference on Software Engineering (ICSE'98)*, pages 115–124, Kyoto, Japan, April 19-25 1998. IEEE Computer Society Press.

[139] R. Milner, M. Tofte, Robert Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[140] Yasuhiko Minamide and Akihiko Tozawa. XML validation for context free grammars. In *Proc. of the Fourth Asian Symposium on Programming Languages and Systems (APLAS'06)*, volume 4279 of *LNCS*, pages 357–373. Springer Verlag, 2006.

[141] Naftaly H. Minsky. Law-governed Linda communication model. Technical Report LCSR-TR-221, Department of Computer ScienceLaboratory for Computer Science ResearchUniversity of Rutgers, March 1994.

[142] Naftaly H. Minsky. Towards alias-free pointers. In Pierre Cointe, editor, *Proc. of the Tenth European Conference on Object-Oriented Programming (ECOOP'96)*, volume 1098 of *Lecture Notes in Computer Science*, pages 189–209, Linz, Austria, July 8–12 1996. Springer Verlag.

[143] Naftaly H. Minsky and Jerrold Leichter. Law-governed linda as a coordination model. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Proc. of the Object-Based Models and Languages for Concurrent Systems, ECOOP'94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, volume 924 of *Lecture Notes in Computer Science*, pages 125–146, Bologna, Italy, July 1994. Springer Verlag.

[144] Clint Morgan, Kris De Volder, and Eric Wohlstadter. A static aspect language for checking design rules. In *Proceedings of the 6th International Conference on Aspect-Oriented Software Development*, ACM International Conference Proceeding Series, pages 63–72. ACM, March 2007.

[145] H. A. Müller and K. Klashinsky. Rigi—A system for programming-in-the-large. In *Proc. of the Tenth International Conference on Software Engineering (ICSE'88)*, pages 80–86, Singapore, April 1988. IEEE Computer Society Press.

[146] Emerson R. Murphy-Hill, Philip J. Quitslund, and Andrew P. Black. Removing duplication from java.io: a case study using traits. In *OOPSLA '05: Companion to the 20th ACM SIGPLAN conference on Object-oriented programming languages, systems, and applications*, pages 282–291, New York, NY, USA, 2005. ACM Press, New York, NY, USA.

[147] Radu Muschevici, Alex Potanin, Ewan Tempero, and James Noble. Multiple dispatch in practice. In Harris [102], pages 563–582.

[148] Gerald W. Neufeld and Son T. Vuong. An overview of ASN.1. *Computer Networks and ISDN Systems*, 23(5):393–415, February 1992.

[149] James Noble and Robert Biddle. Patterns as signs. In Boris Magnusson, editor, *Proc. of the Sixteenth European Conference on Object-Oriented Programming (ECOOP'02)*, number 2374 in Lecture Notes in Computer Science, Malaga, Spain, June 10–14 2002. Springer Verlag.

[150] Nathaniel Nystrom, Michael R. Clarkson, and Andrew C. Myers. Polyglot: An extensible compiler framework for Java. In *Proc. of the Twelfth International Conference on Compiler Construction (CC'03)*, pages 138–152, Warsaw, Poland, April 2003. Springer Verlag.

[151] Martin Odersky, editor. *Proc. of the Eighteenth European Conference on Object-Oriented Programming (ECOOP'04)*, volume 3086 of *Lecture Notes in Computer Science*, Oslo, Norway, June 2004. Springer Verlag.

[152] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the Scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[153] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala: A Comprehensive Step-by-step Guide*. Artima Inc, 1st edition, November 2008.

[154] *Proc. of the Thirteenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, Vancouver, British Columbia, Canada, October18-22 1998. ACM SIGPLAN Notices 33(10).

[155] Klaus Ostermann, Mira Mezini, and Christoph Bockisch. Expressive pointcuts for increased modularity. In Andrew P. Black, editor, *Proc. of the Nineteenth European Conference on Object-Oriented Programming (ECOOP'05)*, volume 3086 of *Lecture Notes in Computer Science*, pages 214–240, Glasgow, UK, July 25–29 2005. Springer Verlag.

[156] Santanu Paul and Atul Prakash. Querying source code using an algebraic query language. In Hausi A. Müller and Mary Georges, editors, *Proc. of the Tenth IEEE International Conference on Software Maintenance (ICSM'94)*, pages 127–136, Victoria, BC, Canada, September 1994. IEEE Computer.

[157] Lutz Prechelt and Christian Krämer. Functionality versus practicality: Employing existing tools for recovering structural design patterns. *Journal of Universal Computer Science*, 4(12):866–882, 1998.

[158] Reasoning Systems. *REFINE User's Manual*, 1988.

[159] Tobias Rho, Günter Kniesel, Malte Appeltauer, and Andreas Linder. LogicAJ home page, 2006. http://roots.iai.uni-bonn.de/research/logicaj/.

[160] Dirk Riehle. Design pattern density defined. In Shail Arora, editor, *Proc. of the Twenty Fourth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'09)*, pages 469–480, Orlando, Florida, October 25-29 2009. ACM SIGPLAN Notices.

[161] Gruia-Catalin Roman, William G. Griswold, and Bashar Nuseibeh, editors. *Proc. of the Twenty Seventh International Conference on Software Engineering (ICSE'05)*, New York, NY, USA, May 15-21 2005. ACM Press, New York, NY, USA.

[162] S. Hossein Sadat-Mohtasham and H. James Hoover. Transactional pointcuts: designation reification and advice of interrelated join points. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, pages 35–44. ACM, October 2009.

[163] Max Schäfer and Oege de Moor. Type Inference for Datalog with Complex Type Hierarchies. In Manuel V. Hermenegildo and Jens Palsberg, editors, *Proceedings of the 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2010, Madrid, Spain, January 17-23, 2010*, pages 145–156. ACM, 2010.

[164] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz, and Andrew P. Black. Traits: Composable units of behavior. In Luca Cardelli, editor, *Proc. of the Seventeenth European Conference on Object-Oriented Programming (ECOOP'03)*, volume 2743 of *Lecture Notes in Computer Science*, pages 248–274, Darmstadt, Germany, July 21–25 2003. Springer Verlag.

[165] Joachim W. Schmidt. Some high level language constructs for data of type relation. *ACM Transactions on Data Base Systems*, 2(3):247–261, September 1977.

[166] Jeremy Singer and Chris C. Kirkham. Exploiting the Correspondence between Micro Patterns and Class Names. In *Eighth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM'08), September 28-29, 2008, Beijing, China*, pages 67–76, 2008.

[167] Jason McC. Smith and David Stotts. Elemental design patterns: A formal semantics for composition of OO software architecture. In *Proc. of the Twenty Seventh Annual NASA Goddard Software Engineering Workshop (SEW'02)*, pages 183–190, Greenbelt, Maryland, December 5-6 2002. IEEE Computer Society Press.

[168] Elliot Soloway, Jeffrey Bonar, and Kate Ehrlich. Cognitive strategies and looping constructs: An empirical study. *Communications of the ACM*, 26(11):853–860, 1983.

[169] John T. Stasko. Tango: A framework and system for algorithm animation. *The Computer Journal*, 23(9):27–39, 1990.

[170] Margaret-Anne D. Storey and Hausi A. Müller. Manipulating and documenting software structures using SHriMP views. In *Proc. of the Eleventh IEEE International Conference on Software Maintenance (ICSM'95)*, page 275, Opio (Nice), France, October 1995. IEEE Computer.

[171] Tom Strelich. The Software Life Cycle Support Environment (SLCSE): a computer based framework for developing soft. sys. In *Proc. of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments (SDE'88)*, pages 35–44, Boston, Massachusetts, 1988. ACM Press, New York, NY, USA.

[172] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Massachusetts, third edition, 1997.

[173] Bjarne Stroustrup and Gabriel Dos Reis. Concepts—design choices for template argument checking. ISO/IEC JTC1/SC22/WG21 no. 1522, 2003.

[174] S. Tucker Taft and Robert A. Duff, editors. *Ada 95 Reference Manual, Language and Standard Libraries, International Standard ISO/IEC 8652: 1995(E)*, volume 1246 of *LNCS*. Springer Verlag, 1997.

[175] Peri L. Tarr and William R. Cook, editors. *Proc. of the Twenty First Annual Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'06)*, Portland, Oregon, October22-26 2006. ACM SIGPLAN Notices.

[176] David Thomas and Andrew Hunt. *Programming Ruby: the pragmatic programmer's guide*. Addison-Wesley Publishing Company, 2000.

[177] David Ungar. Generation Scavenging: A Non-Disruptive High Performance Storage Reclamation Algorithm. In *Software Development Environments (SDE)*, pages 157–167, 1984.

[178] Peter E. van Emde Boas. Resistance is Futile; formal linguistic observations on design patterns. Technical Report ILLC-CT-1997-03, The Institute For Logic, Language, and Computation (ILLC), University of Amsterdam, February 1997.

[179] Jonne van Wijngaarden and Eelco Visser. Program transformation mechanics. a classification of mechanisms for program transformation with a survey of existing transformation systems. Technical Report UU-CS-2003-048, Institute of Information and Computing Sciences, Utrecht University, 2003.

[180] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. In Aart Middeldorp, editor, *Proc. of the Twelfth International Conference on Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–362, Utrecht, The Netherlands, May 2001. Springer Verlag.

[181] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proc. of the Conference on Programming Language Design and Implementation (PLDI'04)*, pages 131–144, New York, NY, USA, June 9-11 2004. ACM Press.

[182] N. Wirth. The programming language Pascal. *Acta Informatica*, 1:35–63, 1971.

[183] Ian H. Witten and Eibe Frank. *Data mining: practical machine learning tools and techniques with Java implementations*. Morgan Kaufmann, 2000.

[184] Yoav Zibin and Joseph Gil. Efficient subtyping tests with PQ-encoding. In OOPSLA'01 [1], pages 96–107.

[185] Moshé M. Zloof. Query By Example. In *Proceedings of the National Computer Conference*, pages 431–438, Anaheim, CA, May 1975.

## שימושים

כעת נתאר בקצרה שלושה שימושים שונים של מושג התבניות הפורמליות ושל הפורמליזם של JTL. שימושים אלה מתוארים בפרקים 3-5 (בהתאמה).

**אחזור תכן.** בכדי להבין טוב יותר את אבני הבניין של תכניות ג׳אווה, הגדרנו קטלוג של מיקרו תבניות: תבניות פורמליות המתייחסות לטיפוסים. התבניות בקטלוג מתארות תכונות משמעותיות של מחלקות ולכן מהוות אוצר מילים, וגוף ידע, מוגדרים היטב שמקלים על הבנה ותיאור של תכן.

בחינה אמפירית של יותר מ 70,000 מחלקות מגלה שהקטלוג שלנו נפוץ בקוד אמיתי. בפרט, יותר מ-45% מהמחלקות מתאימות ללפחות לאחת מתוך חמש תבניות פשוטות יחסית. הגילוי הזה רומז שמחלקות פשוטות, ואולי אפילו מנוונות, נפוצות יותר ממה שנהוג לחשוב.

מעבר לממצאים הבסיסיים, למסקנות העולות מהן ולידע המבוטא ע״י הקטלוג, השיטות הסטטיסטיות שפתחנו לאבחון תכנה ע״ס המיקרו תבניות המזוהות בה, הפכו לכלי שימושי לביצוע מדידות על תכנה. בפרט חוקרים אחרים השתמשו במיקרו תבניות בכדי לאפיין תופעות שבאופן אחר הינן קשות למדידה [25,125,173].

**התמרות.** מנגנונים אוטומטיים להתמרות קוד כוללים בתוכם שני מנגנוני משנה: *השומר*, אשר מתאר את התנאים המוקדמים להתמרה, *והמתמיר* אשר מפיק את הפלט מתוך קוד הקלט שזוהה ע״י השומר.

תפקידן של תבניות פורמליות בשומר הינו ברור. ברם, מעבר לכך, אנו טוענים כי ניתן להוסיף ל-JTL יכולות להפקת פלט, כך שניתן יהיה להשתמש בה לא רק לתיאור השומר, אלא גם לתיאור המתמיר.

אנו מתארים כיצד ניתן להפיק פלט בשפה לוגית, אך עדיין לשמור על כך שהיא נטולת תוצרי לוואי. ההרחבות הללו פותחות קשת חדשה של יישומים עבור JTL, כגון: תמיכה בתכנות מונחה אספקטים, מימוש של מבני תכנות כלליים (generics), בדיקה כי נהלי תכנות מומלצים נשמרים, תרגום מבני נתונים בג׳אווה לסכמות תואמות של בסיסי נתונים יחסיים, ועוד.

העובדה כי גם השומר וגם המתמיר מבוטאים באותה שפה מקלה על כתיבת מנגנוני התמרה חדשים ומפשטת את המבנה שלהם. אנו טוענים, כי ברבות ממהתמרות, המבנה של הפלט תואם למבנה של הקלט, ולכן שיוך רכיבים של הפלט עם רכיבים של השומר מאפשר לכתוב התמרות ברורות ותמציתיות.

**מערכות טיפוסים מבניות.** טיפוס בשפת תכנות הוא תנאי על ערכי זמן הריצה אשר מגדיר למעשה קבוצה של ערכים. אנו טוענים כי תנאי על טיפוסים, כלומר: תבנית פורמלית המתייחסת לטיפוסים, הוא טיפוס שימושי בפני עצמו.

רעיון זה ממומש ע״י Whiteoak, הרחבה של שפת התכנות ג׳אווה שמאפשרת למשתמש להגדיר תבניות פורמליות על טיפוסים ולהשתמש בהן כטיפוסים רגילים. תאימות בין טיפוסים כאלה נקבעת על סמך המבנה שלהם ולא על סמך הצהרות מפורשות בהגדרתם. יכולות אלה מקלות על הקטנת התלויות בין רכיבי תכנה שונים ועל אפשרויות השימוש החוזר ברכיבים קיימים.

אנו מתארים את טכניקות המימוש שאותן פתחנו בכדי לפתור בעיות כגון: שמירה על זהות העצמים, שיוך מתודות לאובייקטים לאחר יצירתם, כבילה מאוחרת של בנאים ועוד. המדידות שבצענו מצביעות על כך שהביצועים של תכנית Whiteoak להשוואה לאלו של תכנית ג׳אווה מקבילה.

בכדי להיות בעלות ערך תבניות אלה צריכות להביע מבנה תכנותי של שפת התכנות שאינו תפל, ואשר משרת מטרה מוחשית. תבניות פורמליות נמצאות ברמת הפשטה נמוכה יותר, לעומת תבניות תכן, וזאת משום שהן מוקשרות לשפת תכנות ספציפית ומתייחסות, כל אחת, ליחידת תכנה בודדת.

מיקרו-תבניות, המוצגות בפרק 3, הן מקרה פרטי של תבניות פורמליות: התנאי של התבנית מתייחס אך ורק לטיפוסים. אנו מציעים את השם *נמו-תבניות* לתיאור תבניות פורמליות המתייחסות למתודות, ואת השם *מילי-תבניות* לתיאור תבניות פורמליות המתייחסות לחבילות (או כל יחידה אחרת שמהווה אוסף של טיפוסים).

**זיהוי באופן מכני.** משמעותה של דרישה זו היא שקיימת מכונת טורינג אשר מחליטה האם יחידת תוכנה נתונה מקיימת את התנאי המובע בתבנית. תנאי כגון "המחלקה חולקת אחריות עם מחלקה אחרת" אינו ניתן לזיהוי באופן מכני משום שהביטוי "חולקת אחריות" אינו חד משמעי. לעומת זאת התנאי "במחלקה קיימת מתודה אשר קוראת למתודה עם שם זהה ממחלקה אחרת" ניתן לבדיקה באופן אוטומטי.

**תכליתיות.** התנאי המובע בתבנית צריך ללכוד יחידות תכנה אשר מספקות צורך מסוים בצורה ייחודית. לכן, התנאי לפיו מספר המתודות מתחלק ללא שארית במספר השדות אינו מהווה תבנית פורמלית. מאידך, תבנית המתארת מחלקה שלה שדה אחד בלבד שערכו נקבע בזמן בניית העצם ואינו משתנה לאחר מכן, ממשת מטרה החשובה למתכנת.

**פשטות.** ע"י הגבלת השפה שבה כתוב התנאי לכזאת שמבוססת על כלים לוגיים חלשים יחסית, אנו מקלים על היכולת לזהות באופן אוטומוטי תבניות שגויות, ועל היכולת לנתח תבניות פורמליות באופן כללי. בפרט, אנו דורשים כי ההגדרה תבנית פורמלית צריכה להיות כריעה.

## JTL

הריבוי בסוגי הרכיבים השונים הקיימים בשפת תכנות עכשווית (שדות, מתדות, בנאים, מחלקות, ממשקים, מערכים, וכ'ו) בשילוב עם הריבוי בסוגי הקשרים ביניהם (הרחבה, מימוש, שימוש, יצירה, וכ'ו), במגוון הסיווגים השונים שניתן לשייך אליהם (רמת נראות, חתום לעומת לא חתום, מופשט לעומת מוחשי, וכ'ו) והעובדה שלעתים קיימת יותר מהגדרה אחת עבור רכיב נתון (ואז מתקיימים יחסי העמסה, דריסה או הסתרה) יוצרים אוסף גדול של אפשרויות להגדרת רכיב תכנה כלשהו.

הגדרת תנאים על רכיבים כאלה הינה למעשה בחירת פלח מסוים מתוך מרחב אפשרויות גדול מאד, מה שמקשה מאד על ניסוחו של תנאי מדויק שכן מנסח התנאי צריך לעתים קרובות להתחשב במספר רב של שילובים אפשריים.

בפרק 2 אנו מציגים שפת שאילתות שתוכננה במיוחד בכדי להתגבר על קושי זה. שפה זו, JTL, משתייכת למשפחת השפות הלוגיות והיא כוללת ליבה קטנה מאד: מספר המבנים בשפה נמוך יחסית, אך שילובם מאפשר הגדרה של קשת רחבה של שאילתות עשירות. התחביר בנוי כך, ששאילתה ב JTL נראית במקרים רבים בדיוק כמו הרכיב בג'אווה שאותו היא אמורה למצוא. למשל, השאילתה `public void f()` תמצא מתודות ציבוריות, ללא פרמטרים, ללא ערך החזרה, אשר שמן "f".

JTL מבצעת בדיקות טיפוסים סטטיות אך אינה דורשת הגדרת טיפוסים של משתנים או פרמטרים: מערכת הסקת טיפוסים מאפשרת למהדר של JTL לחשב את הטיפוס של משתנה באופן אוטומטי (בדומה למה שקיים בשפת התכנות ML) וכך חוסכת מהמתכנת את הצורך לציינם, ובד בבד, מפשטת את הקוד ועושה אותו לקריא יותר.

# תקציר


פלאנתולוגים מסוגלים להסיק רבות אודות תכונותיהם של יצורים קדמונים רק מתוך המאובנים שלהם. הידע שלנו אודות היצורים שחיו על פני כדור הארץ בעידנים רחוקים נגזר, רובו ככולו, מתוך לימוד וניתוח של הבדלים קטנים בגודל ובצורה של מאובנים.

תכניות מחשב ודינוזאורים הם שני מושגים שונים בתכלית. למרות זאת, אנו טוענים כי מהנדסי תכנה ופלאנתולוגים חולקים מטרה משותפת: הרצון להסיק עובדות משמעותיות בנוגע למשהו שבו לא ניתן לגעת. פלאנתולוגים לומדים על יצורים קדמונים. בעולם התכנה, אנו מעוניינים ללמוד על *התכן* ועל *ההתנהגות בזמן ריצה* של תכנית נתונה מתוך קוד המקור שלה או נגזרות שלו (כגון, הקוד הבינרי).

מסתבר שהסקה כזאת איננה משימה קלה. קוד המקור אכן מבטא את התכן, אך הוא כולל, בד בבד, רבדים של פרטים מסוגים שונים שמקשים על המתבונן בזיהוי אלו שרלבנטיים לתכן. ההתנהגות בזמן ריצה מוגדרת בצורה חד משמעית ע״י קוד המקור, אך יכולתנו לנתח התנהגות זו מוגבלת: המגבלות שתוארו ע״י טורינג וצ׳רץ מונעות מאיתנו מלהסיק מסקנות משמעותיות על התנהגותה הדינמית של תכנית.

נסיונות לזהות תכן של תכנית ע״י איתור תבניות תכן [87] בקוד המקור, הציגו עד עתה תוצאות בלתי מספקות. לב הבעיה הוא העובדה שבהגדרתן, תבניות תכן, הינן לא יותר מאשר אסטרטגיה כללית לאיזון בין מספר כוחות שאיתם נאלץ המהנדס להתמודד. לכן, לכל תבנית תכן יש מספר צורות שבהן היא יכולה להופיע מה שמקשה על זיהויה בדיעבד. מצב עניינים זה תואר ע״י קריסטופר אלכסנדר [9]:

> ״כל תבנית מתארת בעיה חוזרת בסביבה שלנו, ואת ליבת הפתרון לבעיה זו. הפתרון מתואר כך שניתן להשתמש בו פעמים רבות, אך בכל פעם לעשות זאת בדרך ייחודית ששונה מקודמותיה.״

חוסר ההצלחה בזיהוי תבניות תכן, הביא אותנו, בעבודה זו, לבחון גישה חדשה לניתוח תכן של תכנה. בכדי להמנע מחוסר הדיוק שנובע מתבניות שאינן חד משמעיות, אנו בוחנים סוג חדש של ״תבנית מוגדרות היטב״.

אנו מציעים כלים פורמליים שמיועדים לסייע למתכנת לחקור, לתחזק ולפתח מאגרי תכנה גדולים. הכלים הללו מבוססים על מושג *התבניות הפורמליות* שהן *תנאים פשוטים, מוגדרים היטב על התכונות, טיפוסים ושמות של יחידת תכנה, או של תת היחידות שלה*. מי שמספקת את הבסיס המתמטי לתנאים הללו היא JTL, השפה לכלי ג׳אווה, שהיא שפת שאילתות שמאפשרת להביע בתמציתיות תנאים מורכבים על רכיבים של תכנית ג׳אווה. אנו משתמשים ב JTL בכדי להגדיר את כל התבניות הפורמליות המופיעות בעבודה זו.

ע״ג הבסיס הנ״ל אנו מציגים שלושה שימושים שונים של תבניות פורמליות המתייחסים לקשת רחבה של פעילויות פיתוח תכנה: א. חקירת קוד קיים ע״י סווג אוטומטי של מחלקות ע״פ קטלוג של תבניות פורמליות; ב. מנגנון כללי לביצוע התמרות על קוד; ג. שימוש בתת סוג של תבניות פורמליות לשם הגדרת סוג חדש של טיפוסים הנתמכים ע״י המהדר.


## תבניות פורמליות

נסקור כעת את ההגדרה המלאה של תבניות פורמליות ונדון במשמעותה.

**הגדרה 1** תבנית פורמלית *היא תנאי מוגדר היטב, פשוט, תכליתי וניתן לזיהוי באופן מכני, על התכונות, טיפוסים ושמות של יחידת תוכנה אל של תת היחידות שלה.*

לסבתי, אדל ממן לבית טולידאנו

המחקר נעשה בהנחיית פרופסור יוסף גיל בפקולטה למדעי המחשב.

אני מודה לטכניון, ולתכנית מלגות הדוקטורט של י.ב.מ, על התמיכה הכספית הנדיבה בהשתלמותי.

# תבניות פורמליות בתכניות ג׳אווה

חיבור על מחקר

לשם מילוי חלקי של הדרישות לקבלת התואר
דוקטור לפילוסופיה

**איתי ממן**

הוגש לסנט הטכניון – מכון טכנולוגי לישראל

טבת תשע״ב חיפה ינואר 2012

# תבניות פורמליות בתכניות ג׳אווה

איתי ממן