Jattern: Porting Pyttern's Pattern Matching to Java

Julien Liénard[1,*], Lucie Leurquin[2] and Kim Mens[1]

[1]UCLouvain, ICTEAM, Louvain-la-Neuve, Belgium
[2]UCLouvain, EPL, Louvain-la-Neuve, Belgium

Abstract

Code pattern matching plays an important role in static program analysis, software evolution, and computing education to identify recurring design patterns, coding idioms, and programming misconceptions. Existing Program Query Languages (PQLs) often present a steep learning curve or are tightly bound to specific target languages. Pyttern was introduced as a beginner-friendly PQL for Python that expresses queries using Python-like syntax augmented with expressive wildcards. In this paper, we present *Jattern*, an adaptation of Pyttern for Java programs. We detail its syntax and technical design, including Java-specific syntax tree pruning, pushdown automaton matching, and wildcard semantics adapted for statically typed, curly-brace languages. Furthermore, we refactor the underlying matching engine into a more language-independent framework to facilitate future language ports, and provide a methodology for adding new target languages. We validated Jattern through 202 automated test cases covering structural idioms, bad practices, and architectural patterns.

Keywords

Program query language, Pattern matching, Java, Syntax tree, Pushdown automaton, Refactoring, Code analysis

1. Introduction

Code pattern matching is widely used in software evolution and maintenance to automatically analyse source code files, detect design patterns, identify anti-patterns, and track potential flaws across evolving software systems [1, 2]. In maintenance workflows, program queries assist developers in comprehending legacy codebases, verifying refactorings, and checking architectural conformance. A particularly valuable use case in educational environments is to use static analysis to support the formative evolution of student code, providing timely, personalized feedback on programming misconceptions or flaws detected in student submissions [3]. However, traditional program querying tools often rely on complex query syntaxes, formal logic specifications, or explicit syntax tree manipulation APIs, creating significant entry barriers for educators and developers. To address this complexity, we developed *Pyttern* [4], a Program Query Language (PQL) specifically designed for Python code. Pyttern minimizes the learning curve by allowing queries to be written directly in Python-like syntax augmented with wildcard abstractions that match subtrees during pattern evaluation.

Despite its usability advantages, Pyttern was initially designed exclusively for Python, a dynamically typed language with indentation-based block scoping. Porting Pyttern to statically typed languages with explicit block delimiters introduces non-trivial technical challenges. In this paper, we study how Pyttern can be refactored into a language-agnostic framework that easily adapts to programming languages like Java, and what the trade-offs are of such an adaptation.

Existing Program Query Languages for Java, such as PQL [5], PMD [6], or Checkstyle [7], require developers to master domain-specific languages, XPath expressions, or visitor APIs. In contrast, extending Pyttern's philosophy to Java would enable Java developers and educators to write expressive queries in a familiar Java syntax without mastering intricate parser details. Furthermore, decoupling Pyttern's core matching engine from language-specific grammars prevents redundant re-implementations when extending the tool to future languages.

The 25th Belgium-Netherlands Software Evolution Workshop, 26–27 November 2026, Madrid, Spain
*Corresponding author.
julien.lienard@uclouvain.be (J. Liénard); lucie.leurquin@student.uclouvain.be (L. Leurquin); kim.mens@uclouvain.be (K. Mens)
0009-0009-2966-7179 (J. Liénard); 0000-0003-0303-1630 (K. Mens)
© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

This paper makes the following main contributions:

- We design and **implement *Jattern*** [1], an adaptation of Pyttern capable of querying Java codebases by extending Java's ANTLR grammar with custom wildcard constructs, accommodating static typing, and resolving Java-specific grammar ambiguities.
- We modularize and refactor the shared pushdown automaton matching engine into a **language-independent framework**, establishing a reusable architecture for future language ports.
- We provide a step-by-step **extensibility guide** identifying language-specific hotspots and outlining how to add support for new programming languages to the refactored framework.
- We **empirically validate** Jattern's correctness through a comprehensive automated test suite of 202 test cases covering structural idioms, bad practices, and design patterns, demonstrating a 100% pass rate and 96% statement coverage.

To achieve this, Jattern leverages ANTLR parser generators [8] to construct Concrete Syntax Trees (CSTs) for Java patterns and target code. It applies specialized CST pruning to eliminate syntactic noise, translates pattern CSTs into non-deterministic pushdown automata called Tree Pattern Automata (TPA), and performs prefix traversal matching against target CSTs representing Java programs.

In the remainder of this paper we introduce Pyttern (§2), discuss related work (§3) and present the problem statement of adapting Pyttern to Jattern (§4). We then evaluate Jattern through unit tests and case studies (§6), generalise Pyttern and Jattern into a framework applicable to multiple languages (§7), outline future work (§8) and conclude (§9) the paper.

2. Pyttern

Pyttern is an open-source program query language designed for Python [4]. Patterns in Pyttern mirror Python syntax but incorporate wildcards representing matchable "holes". To illustrate Pyttern, Listing 1 shows a query designed to find functions that count values in a list based on a predicate. A Python program matching this query is shown in Listing 2.

```
def ?accumulate(?lst, ?pred):
    ?cnt = 0
    for ?item in ?lst:
        ?:*
            if ?pred(?item):
                ?cnt += 1
    return ?cnt
```

Listing 1: Pyttern Query Pattern

```
def accumulator(lst, predicate):
    count = 0
    for item in lst:
        if predicate(item):
            count += 1
    return count
```

Listing 2: Target Python Code

From the example, we can see how Pyttern's wildcards allow for flexible matching of function names, variable names, and control flow structures. Listing 3 shows what the equivalent Jattern query looks like for Java code, and Listing 4 depicts a matching Java program.

```
class #Accumulator {
    public int #accumulate(List<?> #lst,
        Predicate<?> #pred) {
        int #cnt = 0;
        for (#Object #item : #lst) {
            #*{
                if (#pred.test(#item)) {
                    #cnt += 1; }
            } }
        return #cnt; } }
```

Listing 3: Jattern Query Pattern

```
class Accumulator {
    public int accumulator(List<?> lst,
        Predicate<?> pred) {
        int cnt = 0;
        for (Object item : lst) {
            if (pred.test(item)) {
                cnt += 1; }
                    }
        return cnt; } }
```

Listing 4: Target Java Code

[1]The Jattern implementation and test battery are openly available at https://github.com/JulienLie/Pyttern.

This first example already shows some of the challenges in adapting Pyttern to Jattern. Differences in syntax, type system, and block scoping between Python and Java require careful consideration when designing the query language and its underlying matching engine. An obvious first problem is that Pyttern can use "?" as a wildcard prefix because "?" is not a valid character in Python identifiers. However, since Java allows "?" in generic type parameters like `List<?>`, we had to change the wildcard prefix to "#" in Jattern. Another challenge is that Java uses curly braces for block scoping as opposed to indentation, which requires other wildcard syntax ("`#*{}`" instead of "`?:*`") to match blocks of code.

Table 1 summarises the core wildcard syntax and semantics of both Pyttern and Jattern. The reader can observe that while the core semantics of wildcards remain consistent between Pyttern and Jattern, the syntax has been adapted to accommodate differences in language structure and identifier rules.

Table 1
Pyttern and Jattern Wildcards Comparison

Wildcard Type	Pyttern	Jattern	Semantics
Simple	`?`	`#`	Match any single syntax tree element
Named	`?var`	`#var`	Match and bind an element by name
List	`?*`	`#*`	Match 0 or more elements in a list
Number	`?{n,m}`	`#(n,m)`	Match between *n* and *m* elements
Simple Compound	`?: ...`	`#{...}`	Match one level of indentation/block nesting
Multiple Compound	`?:* ...`	`#*{...}`	Match arbitrary indentation levels
Contains	`?<...>`	`#<...>`	Match subtree containment in nesting expressions

The architecture of Pyttern and Jattern is a pipeline that operates through four main parts (Figure 1):

1. **Parsing**: The input query pattern and target program source code are independently parsed by an ANTLR-generated parser into Concrete Syntax Trees (CSTs).
2. **Pruning**: Some syntactic artifacts (like parentheses, semicolons, or commas) and intermediate pass-through wrapper nodes that do not impact program semantics are pruned from both pattern and target code CSTs.
3. **Automaton Generation**: A CST visitor traverses the pruned pattern tree to build a pushdown automaton, referred to as the Tree Pattern Automaton (TPA), where wildcards are encoded as specialized automaton transitions. In essence, the TPA describes how the pattern tree should walk through a program tree in order to find a match.
4. **Matching**: The TPA traverses the pruned target CST in prefix order, evaluating transitions against target nodes while enforcing node boundary constraints and soft statement skipping, leading to state acceptance and wildcard variable bindings.

Due to space constraints, we do not reproduce the formal state-transition definitions here; readers are referred to [4] for a complete, step-by-step worked example illustrating the Tree Pattern Automaton construction and execution trace.

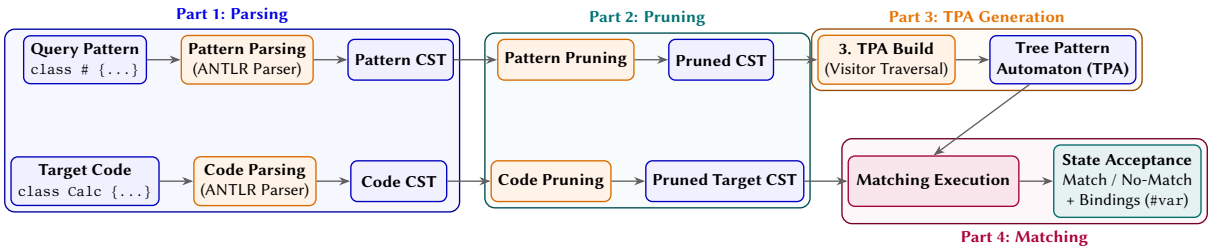


Figure 1: The 4-part pipeline architecture of Pyttern / Jattern.

3. Related Work

Different Program query languages (PQLs) and static analysis tools vary in their awareness, expressive power, and ease of use. Early text-based approaches relied on Regular Expressions (RegEx) to match raw code strings [9, 10], but lack grammar awareness, making matching nested blocks, scoping, or control flow brittle and unmaintainable.

To overcome these lexical limitations, structural pattern matchers operate directly on Abstract Syntax Trees (ASTs) or Concrete Syntax Trees (CSTs). Early work on SCRUPLE [1] introduced code search using target-language syntax augmented with entity wildcards, translated into Finite State Machines over AST nodes, a core philosophy inherited by Pyttern and Jattern. Logic-based formalisms like ASTLOG [11] treat AST nodes as predicates, while SOUL [2] used logic meta-programming to reify object-oriented program structures into logic clauses for codifying, checking, and searching programming patterns. Griswold et al. [12, 13] developed AST pattern matchers (such as AIG and TGen) for automated code transformation. Modern CST tools like Semgrep [14] re-popularised target-like syntax with metavariables and ellipsis operators, yet still lack expressiveness for fine-grained structural constraints such as bounded wildcards or explicit block depth matching. Conversely, rule-based analyzers like PMD [6] and Checkstyle [7] require writing complex XPath queries or custom Java visitor classes, creating a steeper entry barrier for rapid query writing.

To bridge the gap between low-level syntax tree navigation and text search, concrete-syntax pattern languages and Query-by-Example (QBE) prioritize authoring usability. Kellens et al. [15] introduced Example-based Program Querying using concrete code snippets as templates, while De Roover et al. [16] embedded concrete code templates inside logic queries. Subsequent systems like srcQL [17], reflexive pattern matching for code refactoring [18], concrete AST pattern matching [19], and sequence-to-tree matching over incomplete code fragments [20] further established that query languages should closely reflect surface syntax. Both Pyttern and Jattern align with this concrete-syntax movement by allowing queries to be expressed directly on top of native Python and Java syntax.

For complex program properties, advanced static analysis frameworks trade surface syntax for formal expressiveness. Systems like PQL [5] translate SQL-like queries into FSMs over execution traces, while Datalog formulations [21] and graph database queries [22] model program facts and AST dependencies declaratively. Similarly, bytecode analyzers like SpotBugs [23] inspect compiled artifacts, but cannot analyze uncompilable draft code which is common in educational settings.

In computing education, structural pattern matching is essential for detecting novice misconceptions: Lazar et al. [24] extracted AST patterns to diagnose logic errors, and Stanke et al. [25] leveraged graph queries for structural feedback. Building on Pyttern's success in detecting Python misconceptions [3], Jattern extends this concrete-syntax pattern matching approach to Java. Rather than claiming full expressive power, Pyttern and Jattern aim at a sweet spot between formal expressiveness and usability, making it easy to quickly write patterns of sufficient complexity to express and detect student programming flaws and bad idioms without requiring complex static-analysis languages.

4. Problem Statement

Adapting Pyttern from Python to Java requires addressing fundamental language differences:

- **Static vs. Dynamic Typing**: Java requires explicit type declarations for all variables, method parameters, and return types. Query patterns in Java thus must accommodate type abstractions.
- **Block Structuring**: Java uses curly braces (`{}`) and semicolons (`;`) for block delimiters and statement termination, aso opposed to Python's use of colons (`:`) and indentation.
- **Grammar Structure**: Python's ANTLR grammar features explicit named rules for constructs (e.g., `if_stmt`), whereas Java's official ANTLR grammar inlines many constructs directly inside general rules (e.g., `statement` or `expression`).

5. Jattern

Our first main contribution consists of a concrete implementation of Jattern for Java.

5.1. Implementation of Jattern

Trivial Matching Trivial matching evaluates whether two CSTs are structurally identical. The TPA (tree pattern automaton) traverses target CST nodes in prefix order. For internal nodes, the TPA issues a `Move to Left Child` (LC) action. Upon reaching a leaf node, the TPA executes several `Move to Parent` (P) transitions followed by `Move to Right Sibling` (RS) to resume prefix exploration.

Node Boundaries To prevent false matches, TPA transitions enforce *boundary conditions* specifying the minimum and maximum child counts allowed for a given CST node. For fixed-arity rules, boundaries equal the child count in the pattern. However, for variable-arity nodes (such as argument lists or block statement sequences), boundaries are adjusted dynamically. For instance, Java `if` statements have an upper bound of 2 children (the *then* and the *else* branch), whereas Python `if` statements allow arbitrary `elif` branches. The specific boundary values for each language are determined by the grammar's ANTLR definitions. This is why the `define_boundaries()` method is language-specific and must be implemented by each language adapter.

Soft Matching Student code often contains superfluous statements (e.g., print statements for debugging purposes) interspersed between essential pattern fragments. Pyttern and Jattern defaults to *soft mode*, allowing the matcher to skip un-matched statements, as long as matching statements are found further in the code. Soft mode is implemented by injecting self-transitions into the TPA for skippable CST nodes. When visiting a skippable node, the self-transition performs a `Move to Right Sibling` without changing the TPA state or stack. In Jattern, skippable nodes include:

- `BlockStatementContext`: Statements inside the method of a class.
- `PackageDeclarationContext`, `ImportDeclarationContext`, `TypeDeclarationContext`: Top-level file declarations.
- `ClassBodyDeclarationContext`, `InterfaceBodyDeclarationContext`, `RecordDeclarationContext`: Class/interface member declarations (fields, methods).

Parents of skippable nodes (`CompilationUnitContext`, `ClassBodyContext`, `BlockContext`) have their upper boundary limit set to infinity, thus allowing for an arbitrary number of child nodes to be skipped.

CST Pruning Jattern's tree pruner removes concrete syntax artifacts that introduce structural noise:

1. Terminal punctuation tokens: parentheses `"()"`, switch colons `":"`, commas `","`, dots `"."`, braces `"{}"`, and semicolons `";"`.
2. Intermediate pass-through nodes: Single-child non-terminal nodes that exist solely to enforce operator precedence in the grammar. For example, parsing a simple literal like `"42"` produces a chain of nested wrapper nodes (such as `Expression` → ⋯ → `Primary` → `Literal`). Jattern prunes these intermediate wrappers to directly connect the expression to its leaf, eliminating unnecessary structural depth.
3. Semicolon Ambiguity: Semicolons in Java terminate single statements but are absent in compound statements (e.g., loops). Retaining semicolons in the CST causes simple wildcard matches against loops to fail because no semicolon exists to match the trailing token. Pruning semicolons eliminates this ambiguity cleanly across all statement types (see Section 5.2).

Jattern Wildcards As shown in Table 1, Jattern wildcards are very similar to those of Pyttern except for the use of "#" as a wildcard prefix and the introduction of curly brace-based wildcards to match Java's block-structured code. Jattern's wildcards thus becomes:

- **Simple Wildcard (#)** Matches any single CST element by skipping traversal over the subtree of that element. In Java's grammar, # is also allowed in identifier and type rules. Array dimensions can be constrained (e.g., `#[]` matches 1D arrays). Furthermore, Java's static typing allows distinguishing local variable declarations (`# #;`) from assignments (`# = #;`), an expressiveness unavailable in dynamic languages like Pyttern for python.
- **Named Wildcard (#var)** Binds the subtree matched CST element to identifier `var`. Subsequent occurrences of `#var` require subtree isomorphism with the already bound subtree.
- **List Wildcard (#*)** Matches 0 or more comma-separated elements in parameter lists, argument lists, or array initializers.
- **Number Wildcard (#(n,m))** matches between n and m elements, where n is the minimum number of mandatory TPA transitions and m is the maximum number of branching transitions.
- **Simple Compound Wildcard (#{...})** Matches a single level of block nesting. Because Java constructs (such as `try-catch` blocks) introduce variable sub-block depths, Jattern pushes a `body` marker onto the TPA stack upon block entry. After sub-statement matching, the TPA pops parent nodes until returning to the `body` stack frame, dynamically handling variable block depths.
- **Multiple Compound Wildcard (#*{...})** Matches arbitrary block nesting levels (≥ 0) by pushing a `body` marker and maintaining a self-loop traversal over nested block children.
- **Contains Wildcard (#<...>)** Pushes a `body` marker to search for expression subtree containment within arbitrary nested expressions.

5.2. Ambiguity

Porting Pyttern to a statically typed, curly-brace language like Java introduces two primary parsing and structural ambiguity challenges:

Semicolon and Statement Ambiguity In Java, statement syntax introduces a dual challenge for pattern matching: semicolons (;) are mandatory terminators for simple statements (such as variable declarations and assignments), but are absent on compound statements (such as `for`, `while`, and `if` blocks). Furthermore, a standalone semicolon constitutes a valid empty statement in Java.

This duality creates significant syntactic ambiguity when writing wildcard queries. Consider the pattern on the left of Figure 2.

```
class Calculator {
    void hello_world() {
        # #;
    }
}
```

```
class Calculator {
    void hello_world() {
        println("A");
        println("B");
        ;
    }
}
```

```
class Calculator {
    void hello_world() {
        int a;
    }
}
```

Figure 2: Example of semicolon ambiguity in Jattern. The ambiguous pattern `# #;` (left) could match either the two consecutive statements followed by a semicolon (middle) or the local variable declaration (right).

In Java's grammar, this pattern can be interpreted in two fundamentally different ways:

1. *A single local variable declaration*: The first # matches the type (e.g., `int`), and the second # matches the declared variable name (e.g., a), terminated by a semicolon (Figure 2, right).
2. *A sequence of statements*: The first # matches an entire statement (e.g., `println("A");`), while the second `#;` matches another statement followed by an empty semicolon statement (Figure 2, middle).

Because these two interpretations yield distinct automata and represent semantically different programs, the parser must resolve this ambiguity deterministically. The Jattern grammar resolves this at parse time by defining `# #;` as a variable declaration rule.

However, resolving the parse interpretation exposes a secondary matching challenge: if statement wildcards were required to match a trailing semicolon token, matching against compound statements (such as loops or branches) would fail because compound statements contain no trailing semicolon token in their CST. Conversely, if statement wildcards did not require a semicolon, parsing statement sequences would become ambiguous.

Jattern resolves this tension cleanly through *semicolon pruning*: during the CST pruning phase (applied equally to the pattern and target code), all semicolon tokens are stripped from the syntax trees. Consequently, statements are demarcated during automaton matching solely by their structural node boundaries (`statement` contexts) rather than by terminal punctuation tokens. This allows a single wildcard statement to match both simple statements (whose original terminating semicolons have been pruned) and compound statements uniformly, eliminating token mismatch errors without requiring special-case grammar exceptions.

Named Wildcard and Lexer Ambiguity Named wildcards (`#var`) introduce ambiguity between simple wildcards followed by an identifier (`# a;`, matching a local variable definition) and named wildcards (`#a;`, matching and binding a statement to a). Furthermore, top-down ANTLR parsing rules prefer method definitions over constructor definitions, causing patterns like `class #name { #name() {} }` to misinterpret `#name()` as a method return type `#` and method name `name()`.

Jattern resolves named wildcard ambiguities in the lexer phase by leveraging the *longest match* principle. The lexer explicitly defines separate token rules for wildcards:

```
fragment HASH: '#';
WILDCARD_SPACE: HASH WS+;
VAR_WILDCARD: HASH IDENTIFIER;
WILDCARD: HASH;
```

Because `VAR_WILDCARD` matches `#` immediately followed by an identifier without intervening whitespace, it is tokenized deterministically as a named wildcard before parsing, eliminating rule order ambiguities between constructors and methods.

6. Validation

We validated Jattern's functional correctness using an automated test battery comprising 202 test cases, all of which pass successfully (100% pass rate). This test battery was adapted directly from the test suite of the original Pyttern implementation [4], which was designed by the original Pyttern developers to cover all conceivable syntactic cases based on unit examples and student code collected from introductory programming courses, translated and extended for Java. The tests are categorised into four suites:

Trivial Suite 10 tests validate exact structural matching between identical Java pattern and code files.

Soft Suite 27 tests validate soft matching, i.e., allowing for statement skipping across class bodies, method bodies, and imports.

Wildcard Suite 139 tests systematically test all wildcard types (`#`, `#var`, `#*`, `#(n,m)`, `#{...}`, `#*{...}`, `#< ...>`) in all valid syntactic contexts where they can appear, including identifiers, types, expressions, statements, and parameter lists.

Advanced Suite 26 tests evaluate more advanced realistic queries combining multiple wildcards for structural idiom and design pattern detection.

For illustrative purposes, two examples of these advanced tests are presented below:

Hard-Coded List Anti-Pattern Listing 5 depicts a Jattern pattern designed to detect hard-coded array initialisations containing at least 3 elements, which is a common flaw in student submissions who try to hard-code expected answers in arrays.

```
#* {
  # #(#*) {
    # #[] = new #[]{ #(3,) };
  }
}
```

Listing 5: 'Hard-coded list' Pattern in Jattern

```
?:*
    def ?(?*):
        ? = [?{3,}];
```

Listing 6: Corresponding 'Hard-coded list' Pattern in Pyttern

When evaluated against Listing 7, Jattern matches the 4-element array `{2, 3, 5, 7}`. Conversely, a target program containing only a 2 element list (`{0, 1}`) would not produce a match.

```
class IsPrime {
  boolean isPrime(int n) {
    Integer primes[] = new Integer[]{ 2, 3, 5, 7 };
    for (int i = 0; i < primes.length; i++) {
      if (primes[i] == n) return true;}
    return false; } }
```

Listing 7: Target Java Code which matches the hard-coded list query pattern

Observer Design Pattern Listing 8 illustrates a query to detect the Observer design pattern [26]. The query verifies that a class maintains an observer list, initialises it in the constructor, updates it in an `attach` method, and iterates over observers to invoke `update()` inside a `notify` method. We evaluated this query against a benchmark of 12 candidate Java implementations: 7 compliant implementations displaying varying syntactic idioms (such as using different collection types like `List` vs. `ArrayList`, enhanced `for` loops vs. iterator traversal, and supplementary helper methods skipped via soft matching), and 5 non-compliant variants (e.g., omitting constructor initialization of the list, lacking an `attach` update, or omitting the notification loop). Jattern achieved 100% precision and recall on this benchmark, matching all 7 compliant implementations while rejecting all 5 non-compliant variants because the TPA strictly enforces required structural components while accommodating benign syntactic diversity.

```
class #name {
  # #observerList;
  #name(#*) {
    #observerList = new ArrayList<>();
  }
  void attach(# #observer) {
    #observerList.add(#observer);
  }
  void notify(#*) {
    #{
      #<#observerList>.update(#*);
    }
  } }
```

Listing 8: Observer Design Pattern Query in Jattern

Validation Limits and Automated Test Generation While this test suite covers representative syntactic constructs and anti-patterns, validating functional correctness solely through a curated test battery is not sufficient to fully stress-test the level of correctness required to handle messy student code "in the wild". Real-world student submissions frequently feature irregular idioms, unconventional formatting, and syntactic near-misses. To address this limitation, in the future random test cases could

be generated automatically directly from the Java ANTLR grammar definition (grammar-based fuzzing) to stress-test the parser, tree pruner, and matching automaton against edge-case syntactic combinations.

7. Generalising Pyttern to Other Languages

Our second contribution is the generalization of Pyttern into a language-independent framework. Refactoring Pyttern and Jattern into shared engine classes represents an initial step and proof-of-concept towards a language-agnostic pattern matching engine, simplifying the process of extending Pyttern to additional target languages.

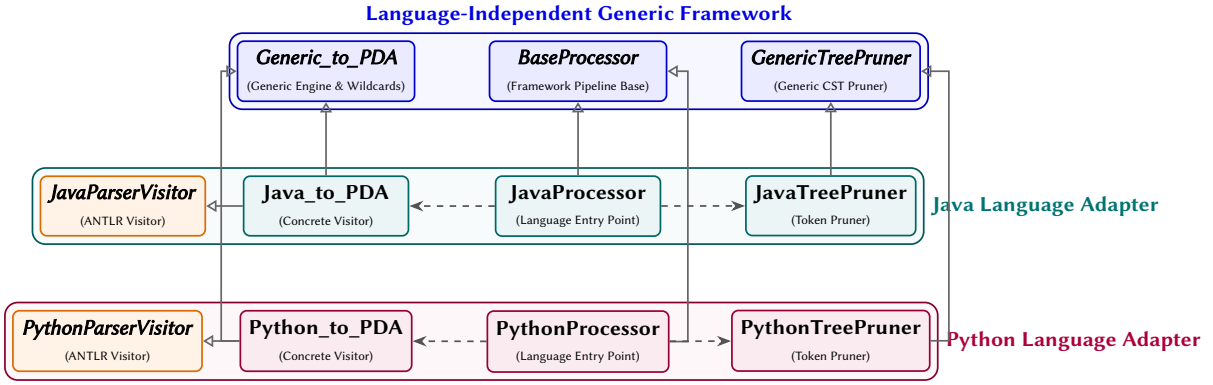


Figure 3: Multi-Language Refactored Class Hierarchy organized in a grid layout to eliminate line crossings.

Framework Architecture As depicted in Figure 3, the architecture extracts common engine logic into three abstract base classes:

Generic_to_PDA implements language-independent PDA (TPA) construction, stack management, lookahead/lookbehind helpers, and generic wildcard handlers.

GenericTreePruner provides general CST pruning routines based on customizable token suppression sets.

BaseProcessor coordinates pipeline execution (lexer/parser invocation, tree pruning, PDA conversion, and file extension mapping).

Multiple Inheritance and linearization In our Python implementation, concrete visitors (`Java_to_PDA` and `Python_to_PDA`) inherit from both `Generic_to_PDA` and ANTLR's generated visitor (`JavaParserVisitor` and `PythonParserVisitor`). To ensure proper Method Resolution Order (MRO) under Python's C3 linearization, language adapters must list `Generic_to_PDA` first in the inheritance declaration.

Language-Specific Hotspots While most PDA logic is language-agnostic, three components remain inherently language-specific and must be implemented by each language adapter:

1. `define_boundaries()`: Specifies child boundary constraints tailored to the target language's ANTLR grammar.
2. `visitSimple_compound_wildcard()`: Handles language-specific block scoping constructs as each grammar might have unique ways of defining them. For example, in Java, the `try-with-resources` statement has a specific structure different from other control flow statements.
3. List wildcard parent visitors: Overrides visitor methods for language rules that host list wildcards (e.g., `FormalParameters`, `ExpressionList`). In Python, list of expressions are often separated by commas, while in Java, they can be separated by semicolons or commas depending on the context. Each language adapter must handle these differences to ensure accurate matching.

Steps for Adding a New Language To extend the framework to a new language, developers should follow a four-step process:

1. **Grammar Setup**: Integrate wildcard rules (`WILDCARD`, `VAR_WILDCARD`) into the ANTLR lexer/parser grammar of the new target language. Add wildcards to identifier, type, expression, parameter, and statement rules.
2. **Tree Pruner Definition**: Subclass from `GenericTreePruner`, specifying punctuation characters in `TO_REMOVE` and essential CST nodes in `TO_KEEP`.
3. **PDA Converter Setup**: Subclass `Generic_to_PDA`, set `self.grammar`, define `skippable_nodes`, implement `define_boundaries()`, and map list wildcard parent visitors.
4. **Processor Implementation**: Subclass `BaseProcessor` to orchestrate the pipeline execution. This requires implementing three core methods:
    - `generate_tree_from_stream(stream)`: Invokes the language's ANTLR lexer and parser on the input stream to construct and return the concrete syntax tree.
    - `create_pda(root_node)`: Instantiates the concrete visitor (from step 3) to convert the pruned pattern CST into a pushdown automaton.
    - `get_language_extensions()`: Returns the set of file extensions (e.g., `{".java"}`) associated with the processor to enable dynamic dispatching of target source files.

Validation After refactoring, we validated the new framework by successfully running all 202 Jattern test cases and all pre-existing Pyttern test cases. Using `pytest-cov` [27], we measured code coverage across Jattern's components, achieving 96% statement coverage. Uncovered lines correspond exclusively to defensive error handling for malformed syntax.

When designing our test suite, we paid particular attention to three key aspects to ensure the robustness and correctness of the validation. First, to prevent errors in boundary definitions or tree pruning logic, we paired comprehensive test coverage with step-by-step visual verification of the pushdown automata transitions using Pyttern's web visualizer. Second, we focused on ensuring the test suite comprehensively covers standard Java constructs. While advanced language features (e.g., complex lambda expressions or anonymous inner classes) are left for future empirical evaluation, standard constructs are fully covered. Finally, pattern matching correctness was validated against concrete ground-truth positive and negative target code samples to ensure both precise matches and appropriate rejections. While the framework has been validated on only two languages so far, making it premature to conclude full portability across all OOP languages, these results demonstrate that decoupling the matching engine is a viable first step towards broader cross-language portability.

8. Future Work

Based on our implementation of Jattern and its refactoring into a more general language-independent framework, several promising directions for future work emerge:

Support for sub-pattern calls Pyttern recently introduced sub-pattern calls, enabling query authors to define modular sub-patterns that can be reused across queries. A natural continuation is to port such sub-patterns to Jattern and the underlying `Generic_to_PDA` engine. This will require extending the stack-based PDA transitions to support parameter expansion and sub-tree query composition in Java without causing state-space explosion.

Language-Specific and Type-Aware Wildcards While Jattern adapts standard Pyttern wildcards to Java, static typing offers opportunities for domain-specific wildcards. Future work includes introducing type-aware wildcards, such as matching specific primitive types (`int`, `boolean`, etc.), interface contracts, or generic type bounds (e.g., `? extends T`), enabling deeper static analysis queries beyond purely structural matching.

Framework Generalisation across Target Languages To further evaluate the generality of our framework classes `Generic_to_PDA`, `GenericTreePruner`, and `BaseProcessor`, future work should apply our developer guidelines to build adapters for additional target languages:

- *Class-based OOP languages* (e.g., C#, C++): to evaluate the reusability of skippable class/interface body declarations and structural boundary rules.
- *Multi-paradigm and dynamic languages* (e.g., JavaScript/TypeScript, Go): to test whether the CST pruning guidelines and PDA transition logic generalise to non-class-centric or dynamically typed paradigms.
- *Functional languages* (e.g., Lisp, Scala): to explore whether the PDA-based matching engine can accommodate functional constructs such as higher-order functions, pattern matching, and algebraic data types.

Pedagogical Tooling and Interactive Visualisation Leveraging the web-based visualisation tool that traces PDA stack operations step-by-step, future work involves developing dedicated IDE plugins (e.g., for VS Code and IntelliJ). Integrating Jattern into learning environments can provide real-time, interactive feedback to student programmers to highlight misconceptions (such as accumulator anti-patterns or hard-coded collections) as they code.

Usability Studies with Educators Whereas usability studies were conducted for Pyttern to evaluate its query authoring efficiency among instructors, there is not yet a formal usability study of Jattern. Conducting an empirical usability evaluation is a critical endeavour to validate that Jattern can effectively and intuitively be used by programming language educators and teaching assistants for the tasks targeted by the tool, such as automated grading and structural misconception detection.

PDA Search Optimisation and Traversal Heuristics Non-deterministic PDA state space exploration can become computationally expensive when matching complex compound wildcards (`#{}`, `#*{}`) over large repositories. Future work includes implementing search heuristics, tree-indexing memoisation, and state-pruning strategies to accelerate pattern matching over larger codebases.

9. Conclusion

In this paper, we presented **Jattern**, a program query language for Java similar to Pyttern for Python. Jattern enables developers and educators to write intuitive code patterns using Java syntax augmented with powerful wildcards. We described the design choices behind Java-specific CST pruning, boundary enforcement, and stack-based PDA wildcard matching. Furthermore, we refactored the core engine into a language-generic architecture (`Generic_to_PDA`, `GenericTreePruner`, and `BaseProcessor`) and provided structured steps for adding new languages. Experimental evaluation demonstrated a 100% test pass rate across 202 unit tests and 96% code coverage, confirming Jattern's capability to detect coding flaws, idioms, and design patterns effectively, and providing an encouraging first step towards porting Pyttern to other OOP languages. Future work includes conducting usability studies with educators, exploring grammar-based automated test generation, extending Jattern for sub-pattern calls and type-aware wildcards, and IDE integration for interactive feedback in educational settings.

Acknowledgments & Declaration on Generative AI Use

This paper is a condensed version of the Master's thesis by the second author, Lucie Leurquin [28], conducted under the supervision of the third and first authors, Kim Mens and Julien Liénard, respectively. During the preparation of this manuscript, the authors used Gemini 3.5 and 3.6 for grammar and spelling correction, rephrasing, stylistic improvements, and text condensation. All AI-assisted modifications were extensively reviewed and iteratively refined by the authors, who take full responsibility for the final manuscript and its contents.

References

[1] S. Paul, A. Prakash, A framework for source code search using program patterns, IEEE Trans. Software Eng. 20 (1994) 463–475. doi:10.1109/32.295894.

[2] K. Mens, I. Michiels, R. Wuyts, Supporting software development through declaratively codified programming patterns, Expert Systems with Applications 23 (2002) 405–413. URL: https://www.sciencedirect.com/science/article/pii/S0957417402000763. doi:https://doi.org/10.1016/S0957-4174(02)00076-3.

[3] Q. Colla, K. Mens, J. Liénard, A Comparison of Three Program Query Languages to Detect Python Programming Misconceptions, in: J. Edwards, R. Perera, T. Petricek (Eds.), Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025), volume 134 of *Open Access Series in Informatics (OASIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2025, pp. 21:1–21:15. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.Programming.2025.21. doi:10.4230/OASIcs.Programming.2025.21.

[4] J. Liénard, K. Mens, S. Nijssen, The Pyttern Program Query Language, in: J. Edwards, R. Perera, T. Petricek (Eds.), Companion Proceedings of the 9th International Conference on the Art, Science, and Engineering of Programming (Programming 2025), volume 134 of *Open Access Series in Informatics (OASIcs)*, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl, Germany, 2025, pp. 23:1–23:15. URL: https://drops.dagstuhl.de/entities/document/10.4230/OASIcs.Programming.2025.23. doi:10.4230/OASIcs.Programming.2025.23.

[5] M. Martin, B. Livshits, M. S. Lam, Finding application errors and security flaws using PQL: a program query language, SIGPLAN Not. 40 (2005) 365–383. URL: https://doi.org/10.1145/1103845.1094840. doi:10.1145/1103845.1094840.

[6] PMD Development Team, PMD source code analyzer documentation, 2026. URL: https://docs.pmd-code.org/latest/index.html.

[7] Checkstyle Development Team, Checkstyle, 2026. URL: https://checkstyle.org/index.html.

[8] T. Parr, ANTLR parser generator, 2026. URL: https://www.antlr.org/.

[9] Python Software Foundation, Regular expression HOWTO, 2026. URL: https://docs.python.org/3/howto/regex.html.

[10] F. Dib, regex101: build, test, and debug regex, 2026. URL: https://regex101.com/.

[11] R. F. Crew, ASTLOG: A language for examining abstract syntax trees, in: C. Ramming (Ed.), Proceedings of the Conference on Domain-Specific Languages, DSL'97, Santa Barbara, California, USA, October 15-17, 1997, USENIX, 1997, p. 18. URL: http://www.usenix.org/publications/library/proceedings/dsl97/crew.html.

[12] W. G. Griswold, D. C. Atkinson, C. McCurdy, Fast, flexible syntactic pattern matching and processing, in: 4th International Workshop on Program Comprehension (WPC '96), March 29-31, 1996, Berlin, Germany, IEEE Computer Society, 1996, pp. 144–153. doi:10.1109/WPC.1996.501129.

[13] D. C. Atkinson, W. G. Griswold, Effective pattern matching of source code using abstract syntax patterns, Softw. Pract. Exp. 36 (2006) 413–447. doi:10.1002/spe.704.

[14] Semgrep Inc., Semgrep documentation, 2026. URL: https://semgrep.dev/docs/.

[15] A. Kellens, J. Brichau, D. R. Coen, Example-based program querying, 2008. URL: https://hdl.handle.net/20.500.14017/e6be211b-6e4b-4395-8f0c-4cf587316931, international Workshop on Query Technologies and Applications for Program Comprehension (QTAPC 2008).

[16] C. D. Roover, T. D'Hondt, J. Brichau, C. Noguera, L. Duchien, Behavioral similarity matching using concrete source code templates in logic queries, in: G. Ramalingam, E. Visser (Eds.), Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007, ACM, 2007, pp. 92–101. doi:10.1145/1244381.1244398.

[17] B. Bartman, C. D. Newman, M. L. Collard, J. I. Maletic, srcQL: A syntax-aware query language for source code, in: M. Pinzger, G. Bavota, A. Marcus (Eds.), IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February

20-24, 2017, IEEE Computer Society, 2017, pp. 467–471. doi:10.1109/SANER.2017.7884655.

[18] J. Lecerf, J. Brant, T. Goubier, S. Ducasse, A reflexive and automated approach to syntactic pattern matching in code transformations, in: 2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018, IEEE Computer Society, 2018, pp. 426–436. doi:10.1109/ICSME.2018.00052.

[19] A. J. Mooij, P. van de Laar, AST matching based on concrete syntax patterns: Exploration of the specification challenges, in: G. Perrouin, B. Vanderose, X. Devroey (Eds.), Proceedings of the 23nd Belgium-Netherlands Software Evolution Workshop, Namur, Belgium, November 21-22, 2024, volume 3941 of *CEUR Workshop Proceedings*, CEUR-WS.org, 2024, pp. 35–41. URL: https://ceur-ws.org/Vol-3941/BENEVOL2024_TECH_paper5.pdf.

[20] G. Matute, W. Ni, T. Barik, A. Cheung, S. E. Chasins, Syntactic code search with sequence-to-tree matching: Supporting syntactic search with incomplete code fragments, Proc. ACM Program. Lang. 8 (2024). URL: https://doi.org/10.1145/3656460. doi:10.1145/3656460.

[21] M. S. Lam, J. Whaley, V. B. Livshits, M. C. Martin, D. Avots, M. Carbin, C. Unkel, Context-sensitive program analysis as database queries, in: Proceedings of the Twenty-Fourth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, PODS '05, Association for Computing Machinery, New York, NY, USA, 2005, p. 1–12. URL: https://doi.org/10.1145/1065167.1065169. doi:10.1145/1065167.1065169.

[22] R.-G. Urma, A. Mycroft, Source-code queries with graph databases—with application to programming language usage and evolution, Science of Computer Programming 97 (2015) 127–134. URL: https://www.sciencedirect.com/science/article/pii/S0167642313002943. doi:https://doi.org/10.1016/j.scico.2013.11.010, special Issue on New Ideas and Emerging Results in Understanding Software.

[23] SpotBugs team, Spotbugs, 2026. URL: https://spotbugs.github.io/.

[24] T. Lazar, M. Možina, I. Bratko, Automatic extraction of ast patterns for debugging student programs, in: E. André, R. Baker, X. Hu, M. M. T. Rodrigo, B. du Boulay (Eds.), Artificial Intelligence in Education, Springer International Publishing, Cham, 2017, pp. 162–174.

[25] L. Stanke, C. Olbricht, M. Goedicke, J. Krugel, Using graph-based queries to detect patterns in program structures for introductory programming, in: A. Greubel, S. Strickroth, M. Striewe (Eds.), Proceedings of Seventh Workshop on Automatische Bewertung von Programmieraufgaben (ABP 2025), 2025, pp. 13–22. doi:10.18420/abp2025_02.

[26] Refactoring.Guru, Observer design pattern, 2026. URL: https://refactoring.guru/design-patterns/observer.

[27] pytest-cov team, pytest-cov documentation, 2026. URL: https://pytest-cov.readthedocs.io/en/latest/reporting.html.

[28] L. Leurquin, Jattern: A Program Query Language for Detecting Coding Idioms in Java Programs, Master's thesis, Université catholique de Louvain (UCLouvain), 2026.