# Dynamate: A Framework for Method Dispatch using invokedynamic

**Dynamate: Ein Framework für Methodendispatch mit invokedynamic**
Master-Thesis von Kamil Erhard
November 2012

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Secure Software Engineering
EC SPRIDE

SECURE SOFTWARE ENGINEERING GROUP

EC SPRIDE
EUROPEAN CENTER FOR SECURITY AND PRIVACY BY DESIGN

Dynamate: A Framework for Method Dispatch using invokedynamic
Dynamate: Ein Framework für Methodendispatch mit invokedynamic

Vorgelegte Master-Thesis von Kamil Erhard

Prüfer: Dr. Eric Bodden
Betreuer: Dr. Eric Bodden

Tag der Einreichung:

**Erklärung zur Master-Thesis**

Hiermit versichere ich, die vorliegende Master-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den 1. November 2012

(Kamil Erhard)

## Abstract

Most programming languages, extensions and tools (clients) running on the Java Virtual Machine use some form of method dispatch in their implementation. If their type of dispatch differ from Java's own type, complex implementations must enable them individually by adapting Java's semantics. Java 7 finally enables developers to reimplement their method dispatch by using the new `invokedynamic` bytecode instruction, leading to increased runtime performance and reduced complexity. Most dynamic programming languages are already extending their implementations with support for `invokedynamic`. The thesis introduces JRuby, Jython and Groovy as programming language clients, MultiJava and JCop as language extension clients and JastAdd, as a language compiler client and presents those types of their method dispatch that could profit from an `invokedynamic` integration. `invokedynamic`'s concepts are explained in detail, from a client perspective, by showing the provided mechanisms that can be used to solve method dispatch related problems. The design of the framework `Dynamate`, which was implemented in the scope of this thesis, is presented. `Dynamate` is based on the concrete requirements of the examined clients. The thesis tries to answer the question if a framework can be designed that abstracts their types of method dispatch and how such a framework compares to their individual approaches in terms of performance and usability. `Dynamate` means to reduce complexity and maintainability and integrates best practises to achieve good performance and usability. Each of the clients' implementations were modified or reimplemented prototypically to integrate `Dynamate` and allow a performance benchmarking. The concluding evaluation measures the performance of its internal components and the performance of the clients' integration by comparing it to their own `invokedynamic` and traditional approaches and discusses the usability of such a framework.

## Zusammenfassung

Die meisten Programmiersprachen, Sprachenerweiterungen und Sprachwerkzeuge (Klienten), die auf der Java Virtual Maschine ausgewührt werden, nutzen Formen von Methodendispatch in ihrer Implementierung. Falls der Dispatch von Javas eigenen Dispatchsemantiken abweicht, muss dieser durch eigene komplexe Implementierungen auf die Semantik von Java abgebildet werden. Java 7 ermöglicht es Entwicklern endlich eigene Dispatch-Semantiken durch die neue `invokedynamic` Bytecode-Instruktion abzubilden, was zu einer deutlich verbesserten Performanz und zu einer reduzierten Komplexität führt. Die meisten dynamischen Programmiersprachen haben bereits angefangen ihre Implementierungen entsprechend anzupassen, um `invokedynamic` nutzen zu können. Diese Thesis stellt die Sprachen JRuby, Jython und Groovy, die Spracherweiterungen Multijava und JCop und das Kompilierwerkzeug JastAdd vor und präsentiert diejenigen ihrer Dispatch-Arten, die von `invokedynamic` besonders profitieren können. Die Konzepte von `invokedynamic` werden im Detail, aus Sicht von Entwicklern solcher Klienten, betrachtet. Es wird das Design des Frameworks `Dynamate` vorgestellt, dass im Rahmen dieser Arbeit konzipiert und entwickelt wurde. `Dynamate` basiert auf den konkreten Anforderungen der sechs vorgestellten Klienten. In der Thesis wird die Frage beantwortet, ob ein solches Framework designt werden kann und wie es sich bezüglich Performanz und Benutzbarkeit, verglichen mit den vorhandenen Klientenimplementierungen, schlägt. `Dynamate` versucht die Komplexität und den Wartungsaufwand zu reduzieren und gleichzeitig die Benutzbarkeit und die Performanz zu erhöhen. Die Implementierungen der sechs Klienten wurden modifiziert und teilweise reimplementiert, um sie in das Framework integrieren zu können und um ein Benchmarking der Performanz zu ermöglichen. Die anschließende Evaluation geht auf die Benutzbarkeit ein und misst die Performanz von `Dynamate`s internen Mechanismen und vergleicht zusätzlich die Performanz der exisitieren Implementierungen der Klienten mit den Integrationsimplementierungen, die im Rahmen dieser Arbeit entstanden sind.

## Contents

## 1 Introduction

For years programming languages running on the Java Virtual Machine (JVM) had to conform to the bytecode instructions used by Java. The JVM had been limited by the dispatch instructions `invokevirtual`, `invokeinterface`, `invokestatic` and `invokespecial` all needing static type information to adhere to the Java object model. This has resulted in problems for languages supporting dynamic concepts like dynamic and duck typing, runtime modifications and more advanced method dispatch techniques.
Although most problems have been circumvented by using functional classes and Java Reflection, this approach has lead to poorer performance and higher complexity than a native approach would have allowed. Reflection allows the runtime invocation of arbitrary methods, but its costly method lookup and security checking is done on each invocation of a method call, whether the call changes or not.
Java Request Specification 292[1], released with Java 7 and commonly known as invokedynamic, introduces the new bytecode instruction `invokedynamic`. This instruction allows developers to invoke methods on arbitrary objects by specifying an identifier, a specific or generalized call type, arguments and most notably a class and method descriptor that will be responsible for linking the specific call to a concrete implementation at runtime. Using `invokedynamic` language developers now can implement their desired method dispatch semantics, select the appropriate one at compile-time and let it link to the target at runtime. Such linked method calls have theoretically the potential to perform just as fast as natively linked calls would, but offer great flexibility for invalidating and re-linking those targets, if needed. This makes `invokedynamic` interesting for any runtime system having to select call targets at runtime, due to not having enough linkage information at compile-time. This includes dynamic programming languages, language extensions featuring e.g. aspect- and context-oriented programming or multiple dispatch, web frameworks and general language tools.
While it is viable to implement `invokedynamic` code individually for every project featuring non-Java dispatch, it is preferable to have a configurable framework encapsulating the logic to support the most common forms of method dispatch. Such a framework offers functionality to specify method call semantics, object graph traversing and method selection and integrates best practises for `invokedynamic` related linkage, invalidation and guarding of call targets. This allows developers to focus on their own program, without the need to learn every aspect of `invokedynamic` for gaining optimal performance. This thesis presents the work on the framework `Dynamate`, which accomplishes these goals by lowering the integration complexity for developers.
Chapter 2 establishes some required background information by defining the concept of method dispatch and introducing the languages, extensions and tools (from now on called clients) selected for the purpose of abstraction with `Dynamate`.
Chapter 3 describes the usage of the `invokedynamic` bytecode instruction and shows which components play a vital role during a method dispatch.
Chapter 4 introduces the core idea of `Dynamate` and describes conceptually the composition of all relevant framework parts. A list of variability points explains the possible differences in types of method dispatch and categorizes them. It is specified, when the framework takes control of the control flow and when interaction with the client takes place. The last section introduces some further situational optimizations.
The first section of Chapter 5 explains the concrete implementation of `Dynamate`, including all its components, interfaces and classes, while its second section focuses on the implementations of a subset of the examined clients, integrating with `Dynamate`.
Chapter 6 evaluates the performance of the clients introduced in Chapter 2 by comparing the performance of their original implementation to the performance of the integration with `Dynamate`. Later sections evaluate the usability of `Dynamate` in terms of integration effort, abstraction and transparency.

[1] http://jcp.org/en/jsr/detail?id=292

Chapter 7 briefly presents other works in the area of `invokedynamic` libraries and frameworks.
Finally, Chapter 8 summarizes and concludes this thesis and looks into the future of `Dynamate` by focusing on further possible extensions and optimizations.
The following list states the main contributions of this thesis.

- Examination and explanation of `invokedynamic`'s main concepts and mechanisms.
- Analysis of the requirements variability points of different concepts of method dispatch.
- Design of an `invokedynamic` framework providing powerful expressiveness for method dispatch.
- Prototypical `Dynamate` integration of JRuby, Jython, Groovy and JastAdd and of reimplemented main dispatch concepts of MultiJava and JCop.
- Performance evaluation of the six integrated clients' original and modified implementation.

## 2 Background

The following chapter establishes some relevant background information by defining the term method dispatch and introducing the selected clients which were reviewed and integrated with `Dynamate` for identifying the required expressive power.

### 2.1 Method Dispatch

Method dispatch is the process of binding a message, at a specific call site, consisting of an identifier, arbitrary arguments and optionally argument type information, to an appropriate target implementation by examining the explicit given information and deriving further implicit information (e.g. runtime types, object composition, active layers). For the purpose of this thesis, only deterministic method dispatch is considered, i.e., targets do not change, as long as the explicit and implicit context stays the same. Otherwise, some concepts to be presented in later chapters (e.g. caching and guarding) would be rendered meaningless in non deterministic cases.
Most object-oriented languages support at least single dispatch, i.e., the target of a call is selected primarily by examining the so called 'receiver' of a message. Once this method base is found, the other arguments, beside the receiver argument, are used to select a concrete method from this base. How the other arguments are used during this selection depends entirely on the concrete language or concept: they can be completely ignored (e.g. in Ruby and Python, not allowing duplicate method names), reduced to their quantity (e.g. in dynamically typed Groovy methods), reduced to their quantity and static types (e.g. method overloading in Java) or fully examined, which is a necessity for multiple and value dispatch (e.g. in Multijava).

### 2.2 Programming Languages

This section introduces three full fledged programming languages having implementations for the JVM. All those implementations have at least started to extend their codebase with `invokedynamic` code and, as will be shown, can profit by `Dynamate`'s reduced maintainability and complexity. The introductions will primarily focus on their method dispatch patterns.

#### 2.2.1 JRuby

JRuby[1] (version 1.7.0.dev) is an implementation of the language Ruby running on the JVM and compiling to bytecode. It is object-oriented and features dynamic typing. Additionally to standard class instance methods, Ruby supports defining singleton methods for specific single instances of a class. Furthermore, modules allow the usage of different namespaces and can serve as mixin modules that can be mixed in by other classes [3].

##### Method Resolution in Ruby

The default method resolution for a call x.m() works as follows:

1. Search for m in the singleton class of x.

[1] http://jruby.org

2. Search for m in all modules included by the class of x.

3. Search for m in the class of x.

4. Repeat the steps 2 and 3 for the superclass of x.

#### Usage of `invokedynamic`

JRuby was one of the first languages embracing `invokedynamic`. Its implementation is very mature and used productively. Many of `Dynamate`'s design choices are based on JRuby's `invokedynamic` implementation. It will be compared to the `Dynamate` approach in the Chapter 6.

### 2.2.2 Jython

Jython[2] (version Jython 2.6a1, jython-pilot branch) is an implementation of the language Python and is running on the JVM and compiling to bytecode, just like JRuby. Python is an object-oriented language supporting functional programming and other paradigms. In contrast to Ruby and Java, it supports multiple code inheritance. Additionally, Python supports wrappers, decorators and metaclasses to wrap methods, instances and classes, i.e., supplying modifying logic that is executed when methods are called, instances created and classes constructed [5].

#### Method Resolution in Python

The default method resolution for a call x.m() works as follows:

1. Search for m in the runtime class of x.

2. Repeat the procedure for all superclasses of x in a depth-first manner, starting rightmost.

#### Usage of `invokedynamic`

Jython supports `invokedynamic` in a developer version, but not productively. This implementation currently supports dispatch for functions, but not methods (which are technically bindable function). The `invokedynamic` implementation will be compared to the `Dynamate` approach.

### 2.2.3 Groovy

Groovy[3] is a object-oriented language supporting dynamic typing, similar to both Ruby and Python. It compiles natively to Java bytecode and uses its syntax, but extends it with its own syntactical improvements, e.g. shorthand operators for list and map operations. It supports closures and introduces a meta object protocol allowing introspection and runtime modifications. Additionally to Java's own virtual dispatch it supports duck typing. Metaclasses with Metamethods allow the refinement of existing classes without modifying their source code.

[2] http://www.jython.org
[3] http://groovy.codehaus.org

#### Method Resolution in Groovy

The default method resolution for a call x.m() works as follows:

1. Determine the class of x.
2. Search for m in the meta class of the runtime class of x.
3. Search for m in the runtime class.
4. Repeat steps 2-3 for the superclass of x.

#### Usage of `invokedynamic`

Groovy has support for `invokedynamic`. It can be activated by a runtime switch ("–indy"). This `invokedynamic` approach will be compared to the `Dynamate` approach.

### 2.3 Language Extensions

This section introduces two programming language runtimes extending standard Java, both featuring advanced method dispatch concepts. These extensions have currently no `invokedynamic` implementations.

#### 2.3.1 Multiple and Value Dispatch / Multijava

Multijava[4] (version Version 1.3.2) is a compiler extending standard Java primarily with support for multiple dispatch and value dispatch.

#### Multiple Dispatch

In constrast to Java's single dispatch, multiple dispatch examines the runtime types of all arguments of a method call and hence selects the method most appropriate for all arguments. This enables developers to declare a family of methods for specialized subtypes in their classes without having to implement individual runtime type checking.
Consider the following example, inspired by an example from [2]. A Shape supertype as pictured in listing 2.1 contains two implementations: Rectangle and Circle. The intersects method in Rectangle could use a specialized path if the given argument's type is Circle. Standard Java code would need to have a runtime type check to determine the concrete type. This approach would work but leads to cumbersome code, especially when more than one parameter and various subtypes are involved, as depicted in listing 2.2. An alternative would be double dispatch, which is commonly used by the Visitor pattern), that exploits virtual dispatch to resolve runtime types implicitly. However, double dispatch is usually not performed on methods with more than one parameter, because it would have to be done individually for each parameter.
Multijava introduces a new syntax to specify runtime type information directly on the declared methods. As seen in listing 2.3 developers can use the `static_type@dynamic_type` syntax to add another set of methods with the same name to distinguish between different implementations of this method family. The Multijava compiler can determine at compile-time all possible combinations and will compile such extended code by inserting the required type checking and dispatch logic into a helper method capable of dispatching to the specific methods. Multijava compiles therefore to standard Java bytecode.

[4] http://multijava.sourceforge.net

```
interface Shape {
  boolean intersects(Shape s);
}

class Rectangle {
  boolean intersects(Shape s) {
    if ( shape instanceof Circle)
      return intersectsCircle(s);
    ...
  };
}
```

**Listing 2.1:** Solution with standard Java

```
  boolean intersects(Shape s0, Shape s1, Shape s2) {
    if ( s0 instanceof Circle && s1 instanceof Circle || s2 instanceof Rectangle)
      return ...;
    ...
  };
```

**Listing 2.2:** Complex type checking with standard Java

```
class Rectangle {
  boolean intersects(Shape s1, Shape s2) {
    // default logic
  };

  boolean intersects(Shape@Circle circle, Shape@Rectangle rectangle) {
    // specialized logic
  };
}
```

**Listing 2.3:** Multiple dispatch with Multijava

### Value Dispatch

Another interesting dispatch type featured in Multijava is value dispatch. Value dispatch is very similar to multiple dispatch, but instead of examining runtime types, the concrete values of the arguments are consulted to select the most appropriate method from a method family. Listing 2.4 shows a very simple example using the syntax `primitve_type@@value` for a special case if the integer argument is 0. Standard Java would once again need to check the argument and dispatch to the specialized implementation. Value dispatch in Multijava works for all primitve types, but could conceptually also work for object types having some standard evaluation function, e.g. a toString() method in Java.

```
class Fib {
  int calc(int x) {
    // default logic
  };

  int calc(int@@0 x) {
    // logic in case x is 0
  };
}
```

**Listing 2.4:** Value dispatch with Multijava

### Usage of `invokedynamic`

Multijava currently does not use `invokedynamic`. The base approach will be compared to the `Dynamate` approach, which reimplements the two mentioned concepts and replaces the instance-of and value checks with a more native mechanism.

### 2.3.2 Context-Oriented Programming / JCop

Context-oriented programming introduces layers to encapsulate behaviour and enrich the object-oriented model by allowing them to be activated, deactivated and even composed during runtime. Each layer can implement a specific behaviour and be chained with others resulting in a layered dispatch. Layers constitute to a second behavioural axis orthogonal to the inheritance axis. During dispatch the receiver is determined by the standard virtual dispatch rules of the inheritance model. This thesis uses JCop[5] (version from 2012-04-17) as a representative for the context-oriented programming model. Listing 2.5 depicts a simple example from [1] of a `Person` class with a `toString` method that basically only outputs the person's name. The new syntactical layer block, as seen in the middle part of the listing, allows methods to be refined. The refined methods will be called instead of the basic implementation when `toString` is called iff the layer is active before calling takes place. The refined method usually signals a `proceed` so that other active layers and eventually the basic implementation will still be called.

```
1  class Person {
2    ...
3    String toString() {
4      return getName();
5    }
6
7    layer Address {
8      public String Person.toString() {
9        return proceed() + ",␣" + getAddress();
10     }
11   }
12 }
13
14 with{Address} {
15   new Person("Hans", "Baker␣Street").toString();
16   //$ Hans, Backer Street
17 }
```

**Listing 2.5:** Layered Behaviour with JCop

The JCop compiler generates helper code to determine the active layers and handle the layered dispatch. Each time a layered method is called, the current layer composition, consisting of all active layers, must be traversed iteratively and with double dispatch, to determine the concrete layer type. This results in a constant performance overhead for any layered method call, even if the layer composition does not change for a layered call. It would be desirable to skip the traversing iff no layers change and to cache targets for known compositions.

#### Usage of `invokedynamic`

JCop currently does not use `invokedynamic` productively, but implementation studies using it exist [1].

## 2.4 Language Tools

The last section deals with general programming language tools.

### 2.4.1 JastAdd

JastAdd[6] is a tool for generating all kinds of language related programs, e.g. compilers, interpreters and code analysers. It is based on object-orientation and attribute grammars. Nodes of an Abstract Syntax Tree (AST) are represented as classes and their relationships implemented through composition. Abstractions are supported through inheritance. The nodes contain two distinct forms of attributes:

[5] https://www.hpi.uni-potsdam.de/hirschfeld/trac/Cop/wiki/JCop
[6] http://jastadd.org

synthesized and inherited attributes. Synthesized attributes are declared and implemented in the node itself, while inherited attributes are declared in the node, but implemented in some parent node in the AST and therefore executed in the parent's scope. Both types of attributes are implemented as methods. Reference attributes furthermore allow references to other nodes, while parametrized attributes support the usage of parameters.
When calling an inherited attribute on a node, the current AST is traversed until an ancestor with an appropriate attribute implementation is found. This method dispatch has both static and dynamic aspects. It is statically ensured that a node with an inherited attribute definitely has an ancestor with a suitable implementation. However, the AST is constructed during runtime and can still change due to rewriting rules, therefore the concrete ancestor type and instance has to be determined dynamically.
Listing 2.6 shows an example from [4] of a grammar for a domain specific language featuring a state machine. The first lines specifiy the formal grammar for the AST, resulting in the following classes: `Statemachine`, `State`, `Transition` and an abstract class `Declaration`. The second part introduces two attributes of the attribute grammar: the synthezied `source` and the (parametrized) inherited `lookup`. Listing 2.7 shows the code that JastAdd generated from the grammar and attribute specification.

### Synthesized Attribute Dispatch

The attribute `source` is generated directly in the `Transition` class as a method that just calls the second attribute `lookup` with its own source label as an argument. The method dispatch for calling `source` is just a standard JVM virtual dispatch on the receiver type. The method is found either in the receiver's class (`Transition`) or, like in this example, in its supertype (`Declaration`).

### Inherited Attribute Dispatch

The second attribute `lookup` and its method dispatch is more interesting. The method is implemented in the `Statemachine` class, but declared for all `Declaration` nodes. Calling `lookup` on `Transition` therefore must result in a traversing of the AST, instead of the inheritance tree. To support this, all Nodes inherit from `ASTNode` and have a reference to their parent node from the AST. For every inherited attribute a helper method is generated in the `ASTNode` class and its default implementation just forwards to a parent. Only the node type specified by the eq block (`Statemachine`) gets a reimplementation with the concrete specified logic.

```
1  Statemachine ::= Declaration*;
2  abstract Declaration;
3  State : Declaration ::= <Label:String>;
4  Transition : Declaration ::= <Label:String> <SourceLabel:String> <TargetLabel:String>;
5
6  aspect Analysis {
7    syn State Transition.source() = lookup(getSourceLabel());
8    inh State Declaration.lookup(String label);
9
10   eq StateMachine.getDeclaration(int i).lookup(String label) {
11     for (Declaration d : getDeclarationList()) {
12       State match = d.localLookup(label);
13       if (match != null) return match;
14     }
15     return null;
16   }
17 }
```

**Listing 2.6:** Simple JastAdd Statemachine Example

```
1  abstract class ASTNode {
2    State Define_State_lookup(ASTNode caller, ASTNode child, String label) {
3      return getParent().Define_State_lookup(this, caller, label);
4    }
5  }
6
7  abstract class Declaration extends ASTNode {
8    State lookup(String label) {
```

```
9       return getParent().Define_State_lookup(this, null, label);
10    }
11  }
12
13  class Statemachine extends ASTNode {
14    State Define_State_lookup(ASTNode caller, ASTNode child, String label) {
15      if(caller == getDeclarationNoTransform()) {
16            // as shown in the eq block in the Analysis aspect
17          }
18
19        return getParent().Define_State_lookup(this, caller, label)
20    }
21  }
```

**Listing 2.7:** Generated Code of the simple JastAdd Statemachine

In this example the `lookup` call results in just one additional call, however if the grammar specifies ASTs with a greater depth, this would lead to one call per level, just forwarding from child to parent, resulting in a possible overhead.

### Usage of `invokedynamic`

JastAdd does currently not use any form of `invokedynamic` in its implementation. For this thesis, JastAdd will be integrated with `Dynamate` for the method dispatch of inherited attributes. The approach and its results is presented in the later chapters.

# 3 invokedynamic

The following chapter introduces `invokedynamic` from a client perspective, i.e., it focusses on the concept and usage, most relevant for users developing language related applications. The technical implementation in the JVM of `invokedynamic` is beyond the scope of this thesis.

## 3.1 Bootstrapping

`invokedynamic` is a bytecode instruction without a corresponding source code instruction, i.e., it cannot be called from source code natively. The instruction takes the following input parameters:

- a unicode identifier, usually the message of the call,
- a method type, consisting of primitive or object types representing the return and parameter types of the call,
- the Java class to bootstrap the first invocation of the call,
- the name of a method in this bootstrap class
- and optionally, additional arbitrary arguments.

The instruction expects as many arguments on the call stack, as stated by the method type [6].
An exemplary call `(indexOf, (String, int)int, BootstrapHandler, boostrap)` would represent a call with the message "indexOf", two arguments of types String and int and a return type of int. The call will be handled by the method "boostrap" in the BootstrapHandler class. The term "call site" usually denotes the occurrence of any call instruction. In this thesis "call site" will refer invariably to `invokedynamic` instructions.
At runtime, when the JVM invokes an `invokedynamic` instruction for the first time, it tries to link the call to an implementation by transferring the control to the stated method in the bootstrap class. The bootstrap method gets invoked with the corresponding identifier and method type, as specified in the bytecode. This method is responsible to find an implementation for the specific call and link it to the call site. The linked call site can generally be either constant, i.e., its implementation target does never change, or it can be mutable, meaning its target could potentially change on every invocation. A bootstrap method is never invoked twice for the same call site, but the target of a mutable call site can be modified at any given time by external code.

## 3.2 Method Resolution

The process of finding an implementation target in object-oriented languages for a call site is known as method resolution. The method resolution can be either static, i.e., it is only depended on the specified identifier and method type and completely independent from the actual arguments of the call, virtual, by depending on the concrete receiver type and dynamic by considering all arguments. `invokedynamic` provides a lookup class with finder methods to search for implementation targets. Any field, method, constructor, whether it may be static or even private, implemented in a Java class, can be looked up. The returned value of a successful lookup is an object pointing to the field/method/constructor implementation: the so called `MethodHandle`. A MethodHandle can be invoked with arbitrary arguments via its invoke method, just like it is the case with the `Method` class of the Reflection package. The two most

important lookup finders, for method dispatch, are `findVirtual` and `findStatic`, used for finding virtual and static methods. Both expect the signature of the method and the class or interface, where it is defined. Virtual method lookups result in unbound method handles, i.e., the concrete implementation will be determined prior to its invocation by examining the runtime type of the receiver argument of the call. Listing 3.1 shows an example of a simple boostrap procedure with virtual method resolution and linking to a call site.

```
1 static CallSite bootstrap(Lookup lookup, String identifier, MethodType type) {
2   // identifier="indexOf", type=(String, int)int
3   MethodHandle target = lookup.findVirtual(
4     type.parameterType(0), identifier, type.dropParameterType(0));
5   return ConstantCallSite(target);
6 }
```

**Listing 3.1:** Method Resolution and Linkage

This example is actually the equivalent `invokedynamic` instruction to a native `invokevirtual` instruction with the arguments from the prior example, as stated in the comment, and would result in the invocation of the "indexOf" method with a given index argument on a given string receiver argument.

### Dynamic Resolution

For most dynamic languages and extensions, static and virtual method resolution will not suffice, due to dependence on examinations on arguments or other runtime context. In this case, the boostrap procedure must defer the dynamic examination to another procedure, as seen in the listing 3.2.

```
1 static CallSite bootstrap(Lookup lookup, String identifier, MethodType type) {
2   // identifier="intersects", type=(Shape, Shape)boolean
3   CallSite callSite = new ConstantCallSite(type);
4   MethodHandle dynamicResolution = lookup.findStatic(Boostrap.class, "dynamicResolve",
5     MethodType.methodType(boolean.class, String.class, Shape.class, Shape.class));
6   callSite.setTarget(dynamicResolution);
7   return callSite;
8 }
```

**Listing 3.2:** Deferred Resolution and Linkage

This second procedure "dynamicResolve" will be set as the new target of the call site, meaning it will be called every time the call is invoked. This allows the method to examine the concrete runtime arguments, resolve a suitable method handle and eventually invoke it. 3.3 shows the prior example from Multijava, reimplemented with `invokedynamic`. The method determines the runtime types of both the receiver argument and the second argument, looks up the appropriate method based on the runtime type of those arguments and invokes it with the concrete arguments.

```
1 static boolean dynamicResolve(String identifier, Shape shape1, Shape shape2) {
2   // identifier="intersects", shape1=Circle, shape2=Rectangle
3   MethodHandle target = lookup.findVirtual(shape.getClass(), "intersects",
4     MethodType.methodType(boolean.class, shape2.getClass()));
5   // handle points to Circle#intersects(Rectangle)
6   return (boolean) target.invokeWithArguments(shape1, shape2);
7 }
```

**Listing 3.3:** Dynamic Resolution for the Shapes Example

Executing the dynamic resolution layer before every call generates a constant overhead that is not negligible compared to direct calls. Therefore mechanisms to guard and invalidate call sites, depending on the runtime context, exist and will be presented starting from Section 3.4.

## 3.3 Adaptation

In a real implementation it would be impossible to implement a dynamic resolution method for every imaginable call, as seen in the prior example. Adaptation allows the implementation of generic handler

methods for arbitrary call sites by adapting them to concrete call sites. `invokedynamic` provides various adaptation methods in the `MethodHandle` and `MethodHandles` classes. These methods generally accept a target method handle, wrap it with a specific adaptor and return the new adapted method handle. The most important, for `Dynamate`'s purposes, are the following ones.

#### Type Casting

Adaptors can up- and downcast primitive and object argument and return values to compatible types.

#### Inserting and Dropping

Adaptors can insert and drop arguments allowing e.g. the presetting of arguments and the injection of missing argument types with supplied external values.

#### Collecting and Spreading

Adaptors can collect arbitrary argument types into a single array argument, e.g. from (int, String) to (Object[]). Spreading can revert this process.

#### Filtering

Adaptors can process single arguments and the return value with user supplied methods (the filters) to modify arguments before method execution and return values after execution.

#### Folding

Adaptors can preprocess the set of all arguments with a user supplied method (the combiner). The result of the combiner will be used as an input argument to the target method handle. This allows especially the chained invocation of multiple method handles.

#### Permuting

Adaptors can permute the argument order, i.e., a type (int, String) can be permuted to (String, int).

#### Exception Wrapping

Adaptors can wrap calls with exception handlers.

### 3.4 Guarding

As explained in 3.2, the dynamic method resolution layer generates a constant overhead during execution by processing runtime context. To minimize this overhead it would be preferable to skip the processing the next time a call site gets invoked. This can easily be done by passing the instantiated (now mutable) call site from the bootstrap method to the dynamic resolution method. The dynamic resolution method just modifies the call site by setting the target to the newly dynamically resolved method target. Subsequent invocations of this call site will skip the dynamic resolution layer and directly execute

the target implementation. Naturally, subsequent invocations can have a different runtime context than the first (dynamically resolved) invocation. A naive invocation of the same call site with a different context would still result in the execution of the former target, although the changed context could require the resolution and execution of a new target.
`invokedynamic` provides a mechanism to guard a call site with a boolean test. Such a guarded call consists of the mentioned test and two supplied method handles: A default path and a fallback path. The guarded call site's test gets executed before every invocation and the result decides the path the call site takes. A successful test leads to the default path and a failed test leads to the fallback path. The default path is generally the dynamically resolved target method handle and the fallback path can just be the method handle to the dynamic resolution method. What differs for every `invokedynamic` client is the supplied boolean test method. It entirely depends on the client's semantics. A default test could compare the concrete runtime arguments. While such a test would work for many clients, it would generally be too specific to offer a reduced overhead and would even fail to work for clients needing to consider non argument runtime context, such as JCop for its layered dispatch. A sensibly chosen test method can reduce the overhead to a minimum, whereas a poorly chosen one can limit `invokedynamic`'s performance advantages.
Listing 3.4 shows a simple example what a user supplied test could look like in the context of Multijava to guard the Shapes example from listing 3.3.

```
1 static boolean testRuntimeTypes(Class[] savedRuntimeTypes, Object[] arguments) {
2   for (int i=0; i<arguments.length; i++) {
3     if (savedRuntimeTypes[i] != arguments[i].getClass())
4       return false;
5   }
6
7   return true;
8 }
```

**Listing 3.4:** A boolean test for the Shapes Example

## 3.5 Invalidation

While method resolution, depending not entirely on non argument runtime context (e.g. layered dispatch or runtime modification, i.e., reflection) can be guarded with the mentioned guards, performance would suffer if the tests would need to check for seldom external changes of context before every invocation. Such seldom failing guards are usually better suited for the usage of switchpoints. Switchpoints are instances of the class `SwitchPoint` and provide a method to trigger their invalidation. They are thread-safe and can be used to guard call sites. Like regular guards, they have a default and a fallback path. Unlike regular guards, no test is invoked before method invocation. The default path is used as long as the specific switchpoint is not invalidated. When a swithpoint gets invalidated (e.g. as a result of a layer activation or class reflection), it transparently switches to the fallback path. This provides guarding with no significant performance overhead. While guards with switchpoints seem to be much more worthwhile than regular guards, the invalidation itself costs performance, because threads need to be synchronized in order to ensure the switch to the fallback path. Permanent switchpoint invalidation would destroy the performance advantage over regular guards. Therefore, the choice of one of those guard types needs to be a careful consideration depending on the client's concrete needs. Conceptually switchpoints are suited for event based runtime context changes (e.g. a layer de/activation), while regular guards are better suited for pollable changes (e.g. argument runtime types or values).

# 4 Dynamate's Concept

This chapter describes Dynamate conceptually without an emphasis on the technical implementation, which will instead be detailed in Chapter 5. After outlining the core idea, requirements and responsibilities, the control flow of the framework will be presented. The last section concludes this chapter by introducing some situational optimizations, clients could profit from, in some cases, and the source code instruction.

## 4.1 Core Idea

Dynamate's core idea is to provide a framework that can be integrated with a large variety of clients by providing components and extension points to abstract from invokedynamic, and specific configurable options to support different types of method dispatch. Clients integrate Dynamate by instantiating and invoking its components in their own invokedynamic bootstrap methods, allowing the framework to be called by the JVM during the call site linkage. From then on, Dynamate takes control of handling the required method dispatch steps. During certain configurable variability points, user supplied classes with well-defined interfaces are invoked to perform client specific actions.

## 4.2 Requirements

This section lists some requirements, Dynamate has to meet, in order to be able to support a broad set of different clients. The requirements are based on the needs of the examined clients and result in, so called, "variability points". Variability points is the term, used by this thesis to describe steps during method dispatch where clients differ from each other and where they need configurable options to allow their specific semantics. The following subsections introduce the four located variability points of the examined clients.

### Call Site Semantics

Clients usually pass two kinds of arguments to their method calls: User arguments and helper arguments. User arguments are the explicit arguments specified by the user in his program source code. However, in order to resolve a method for a given method call, clients often pass additional arguments, e.g. a context or scope. These helper arguments are primarily used during method resolution, but can also be used by the target method implementation, resulting in a need for a argument specification. Additionally, the first argument position usually denominates the receiver argument, however some clients follow different rules, resulting in a need for a configurable receiver index. Finally, clients may allow the absence of specified arguments by presetting them with well defined values and may require conversions between different argument types.

### Object Graph

Due to the object-oriented nature of the examined clients, they all feature an object graph to model their language semantics and their method resolution depends explicitly on it and on the actual graph membership of the receiver argument. Generally, there are two types of relationships in an object graph: the is-a relationship and the has-a relationship. Is-a denominates the relationship between supertypes and subtypes. A type is a subtype of another type (the supertype), if it can be substituted for the supertype

without breaking the specification. Has-a in contrast, indicates a composition or containment between objects, even with different non-compatible types. While most object-oriented languages allow both relationship types, they usually use the is-a relationship for method dispatch, i.e. method implementations are executed in the context of the actual runtime type of the receiver argument, and not in the context of the type containing the implementation. On the other hand, method dispatch using has-a relationships is usually executed in the context of the object, containing the implementation. Therefore, clients need to be able to specify their required object graph semantics.

### Runtime Dynamic

The examined clients depend on some kind of runtime dynamic to achieve their method dispatches. Their dispatch algorithms have to examine the runtime context every time the call site is invoked to ensure the invocation of the correct target method. Examination of the runtime context and, in order to gain performance advantages through `invokedynamic`, tests to determine at the start of an invocation if the context has changed and if a target needs to be resolved anew, must be configurable with user supplied strategies.

### Method Implementation Locations

The actual method target implementations of the clients are either stored as compiled bytecode methods or, due to runtime modifications, added and compiled dynamically, usually with the help of tools like ASM. The compiled methods are generally contained in three possible types of classes: natural classes, functional classes and compilation classes.

#### Natural Classes

A natural class is a conventional class, modelling a concrete behaviour and containing a set of cohesive methods. In a natural class, the target implementation is one of these cohesive methods. The receiver of the method dispatch is an instance of the natural class' type.

#### Functional Classes

A functional class is a class implementing a well defined interface, consisting of methods usually called "invoke", "call" or "run". Functional classes encapsulate a single implementation and model it as an actual object. They are usually a replacement for closures, if a language does not support them natively, to allow the treatment of methods as first-class citizens. The receiver argument is just passed for one of the method's parameters.

#### Compilation Classes

Compilation classes contain a set of different methods, not adhering to a concrete object-oriented type, but rather to a concrete concern, e.g. a class representing a script file, an aspect from aspect-oriented programming or a class modelling layers for layered dispatches. As in functional classes, the receiver argument is passed for one of the method's parameters.
Therefore, clients need support for specifying target implementations in these different types of classes.

## 4.3 Responsibilities

The responsibilities of `Dynamate` are primarily based on the mechanisms provided by `invokedynamic`, as introduced in Chapter 3.

## Method Dispatching

The method dispatch with Dynamate starts in the client's bootstrap method, which is referenced in its bytecode. All required classes get instantiated and configured, depending on the concrete client call semantics. Eventually the control is passed to Dynamate's MethodDispatcher class, representing the main entry point of the framework. Listing 4.1 shows some pseudocode of a simplified bootstrap method, communicating with Dynamate. The concrete configuration is omitted, although the following sections explain some of the available configuration options, while the next chapter details complete client configurations.

```
1  // User implemented boostrap method, referenced in the client's bytecode
2  CallSite boostrap(Lookup lookup, Stirng identifier, MethodType type) {
3    dispatcher = new MethodDispatcher();
4    // configure dispatcher depending on client semantics
5    ...
6    return dispatcher.dispatch(lookup, identifier, type);
7  }
```

**Listing 4.1:** Boostrapping Pseudocode

## Adaptation

Dynamate performs all the required adaptation to support any possible call type implicitly. Developers have to supply a class implementing the ArgumentSelector interface to describe the argument position semantics, e.g. the receiver index and indices of arguments that should only be used during method resolution and not for the actual target invocation. Optional ArgumentConverter classes can be supplied to indicate possible argument conversions, e.g. a conversion from client language arrays to host language arrays, or conversions between different data structures. Finally, an optional ArgumentInjector class can be supplied to allow Dynamate to inject specified values for optional parameters. These adaptations allow the client to contain mostly generic methods, containing a broad set of parameters, by letting Dynamate adapt the call site to the concrete implementation target using these supplied policies.

```
1  // Argument Adaptation
2  Call Site: m(int, String, String, String), Target Method: m(int, String[])
3  => Adaptation by collecting the trailing String arguments to an array using implicit policies
4
5  // Argument Conversion
6  Call Site: m(int, Object[]), Target Method: m(int, List)
7  => Adaptation by converting the array to a list using a user supplied ArgumentConverter class
8
9  // Argument Injection
10 Call Site: m(int), Target Method: m(int, Block)
11 => Adaptation by injecting a block (null, constant value or new instance)
12    using a supplied ArgumentInjector class
```

**Listing 4.2:** Some Adaptation Examples

## Method Resolution

The concrete method resolution is achieved using object graph traversing and method selection, as described by the following sections.

### Object Graph Iteration

Most object-oriented clients use some form of object graph to represent their explicit object model. To determine the concrete receiver and the method base of a method call, the object graph usually has to be traversed using client specific semantics, e.g. for Multijava this would just require traversing the Java inheritance hierarchy, whereas for JastAdd this would require the traversing of its explicit AST. Clients are encouraged to implement the ObjectGraphIterator interface to supply these semantics. If

not supplied, the Java semantics are applied or the following `MethodSelectionStrategy` class must perform the traversing additionally to its main responsibility of method selection. However, if the object graph depicts a has-a relation ship, a traversing usually needs to occur not only during method resolution, but also before every subsequent invocation, in order to determine the actual containing object of the receiver argument. This can be achieved by applying an `ArgumentFilter` instance before the target method that iterates the graph and replaces the receiver.

#### Method Selection

Clients must implement the `MethodSelectionStrategy` interface, which will be called during `Dynamate`'s method resolution, to select a method depending on the actual method identifier and runtime context. If an `ObjectGraphIterator` is supplied, the selector should select a method from the method base of the passed object graph node. If not supplied, the client can use its own semantics to traverse a potential object graph. The selected method target is represented as a tuple, consisting of the method's implementation class, a method identifier and the parameter and return type and returned back to `Dynamate`, i.e., that such a selected method must be implemented in a compiled java class, in order to be selectable. The selected method does not need to conform necessarily to the concrete call site type. `Dynamate` tries an adaptation using its implicit and the user supplied adaptation policies, although a manual adaptation by the client is still possible, using a simplified interface. If `Dynamate`'s automatic adaptation fails, the client's method selection strategy is asked for a manual adaptation. Concrete client implementations will be shown in 5.
Figure 4.1 pictures `Dynamate`'s simplified interaction during method resolution with the user supplied object graph iterator and the method selection strategy. Other required steps during resolution, e.g. adaptation and guard installation are omitted in this figure.

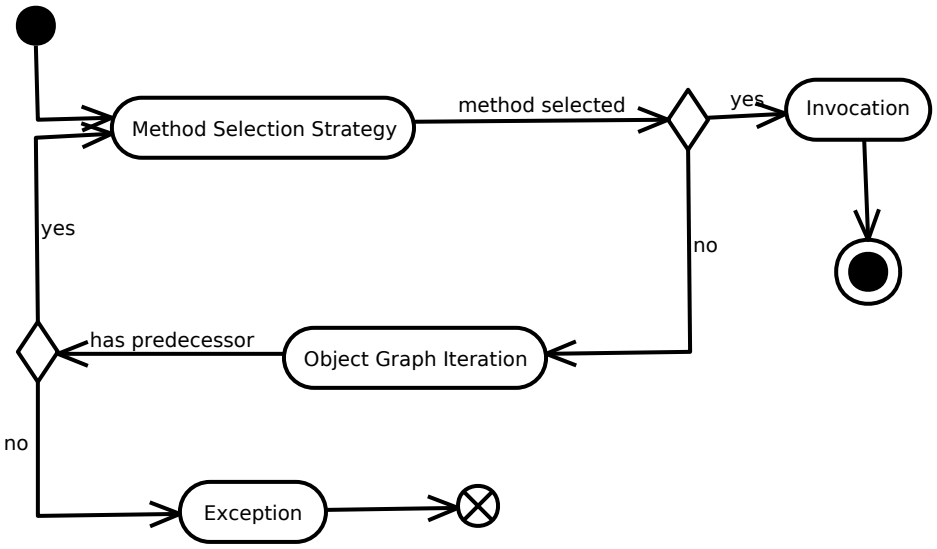


**Figure 4.1:** Simplified Method Resolution Activity

### Guarding

`Dynamate` uses the `GuardStrategy` interface to communicate with the client. The developer has to implement it with client specific semantics that will be used to guard the resolved method depending on the runtime context. The guard strategy depends on the actual call site, allowing multiple guard strategies for different call sites. The user supplied method gets called during method resolution with the actual arguments of the method call and is expected to derive a meaningful key from the runtime

context that can be stored in the guard and used during subsequent method calls to test for a changed runtime context. `Dynamate` takes care of the guard installation for the call site. Listing 4.3 shows some pseudocode depicting a user supplied guard strategy and `Dynamate`'s interaction with it during resolution and subsequent guard tests. Chapter 5 will describe in more details meaningful guard strategies for a subset of the examined clients.

```
// User implemented guard strategy
class GuardStrategy {
  Key buildKey(Object arguments[]) {
    // derive runtime context from arguments and from external context (e.g. active layers)
    return meaningfulKey;
  }
}
// During Method Resolution
key = guardStrategy.buildKey(callArguments);

// During Guard Testing
newKey = guardStrategy.buildKey(callArguments);

if (key.equals(newKey)) {
  return defaultPath;
} else {
  return fallbackPath;
}
```

**Listing 4.3:** GuardStrategy Pseudocode

## Caching

Caching refers to the technique of storing method handles resolved during method resolution under a meaningful key derived from the runtime context. Though the guarding mechanism, as described in the last section and chapter, provides a default and a fallback path, subsequent invocations of a guarded call site with changing runtime context would always lead to the fallback path and therefore to a renewed method resolution, even when a call with a specific runtime context has already been resolved once. This leads to dramatical performance overhead if the runtime context is constantly changing. This problem can be solved by using method handle caching. Once a method handle is resolved for the first time, it gets cached using a key derived from the specific runtime context semantics of the client. A caching lookup procedure can be used as the fallback path for the guards. As shown in listing 4.4, the procedure would look up the determined runtime context in a cache and, if found, return the cached method handle, or if not yet cached, return the handle pointing to the dynamic resolution method, resulting in a resolution and subsequent caching of the method. As in the case of guarding, `Dynamate` uses the same `GuardStrategy` interface to derive the caching key from the runtime context.

```
// During the fallback path, after the guard
key = guardStrategy.buildKey(callArguments);

if (cache.containsKey(key)) {
  cachedMethodHandle = cache.retrieve(key);
  return cachedMethodHandle.invoke(callArguments);
} else {
  return dynamicResolution.invoke(callArguments);
}
```

**Listing 4.4:** Caching Pseudocode

## Invalidating

`Dynamate` can be advised by the client during method selection to install a switchpoint guard for the current call site. To allow its subsequent invalidation, `Dynamate` hands out a specific `InvalidationListener` object to a user supplied `Invalidator`. Once the client notices some runtime context change, it can easily invalidate the passed invalidation listener, resulting in a renewed method resolution during the next call

site invocation. Listing 4.5 shows pseudocode depicting the invalidation interaction between Dynamate and the client, using the GlobalInvalidator class, which is a singleton implementation of an invalidator that can be used by clients for rapid prototyping. Productive clients can implement their own invalidator classes.

```
1 // During method resolution, if advised by client to install switchpoint guard
2 targetMethod.setInvalidator(GlobalInvalidator.getInvalidator(X));
3
4 // In client's code, at some later time (e.g. a runtime modification of some class X)
5 GlobalInvalidator.getInvalidator(X).invalidate();
```

**Listing 4.5:** Call Site Invalidation Example

## 4.4 Control Flow

Figure 4.2 depicts the control flow of an invokedynamic instruction integrated with Dynamate. As seen in the left path of the diagram, the first time the call site gets invoked, the client's bootstrap method is called by the JVM, which in turn passes control to Dynamate by calling its MethodDispatcher. The MethodDispatcher class starts the method resolution process, as described in more details in Section 4.3. After a tentative method target is resolved, it gets adapted using the implicit and user supplied policies. If the client specifies a guard strategy for the resolved method, Dynamate uses it to cache the method under the provided key and then proceeds to install a guard. If no guard strategy is specified by the client, these two steps are skipped. Correspondingly, if the client decides to use a switchpoint guard, Dynamate installs it, with its default path set to the regular guard, or directly to the method, if no guard strategy has been specified or Dynamate was advised to skip it, as long as the switchpoint stays valid. Finally, the adapted method is invoked with the actual call site arguments.
On subsequent call site invocations, the right path of the diagram is taken. If a switchpoint guard is installed and not yet invalidated, the regular guard gets tested using the stored guard strategy. If it was invalidated, the MethodDispatcher would get invoked, which would result in a renewed method resolution, as described before. If the regular guard succeeds, the stored method target gets directly invoked, whereas a guard failure would result in a cache lookup and a subsequent invocation, if a method could be retrieved, or a fallback to the MethodDispatcher, if not yet cached. The rightmost path is generally the fastest one. Each branch to the left side results in a performance penalty, due to increased computation efforts.

## 4.5 Situational Optimizations

This section introduces two optimizations that offer only situational performance gains. They depend on concrete client programs and even on concrete call sites and therefore can be enabled for each call site individually. A potential profiler could try to identify situations, where those optimizations could be applied. Chapter 5 will state, which of those optimizations are used by the examined clients.

### Cascading Guards

Generally for any given call site, only one regular guard gets installed with its default path leading to the method target and its fallback path leading to the caching lookup procedure. Cascading guards are instead call site guards with multiple internal regular guards. Figure 4.3 shows a workflow of a cascading guard with three internal guards. Each internal guard's default path is set a different method target. The fallback path points to next internal guard in the cascading. The last guard's fallback path leads to the caching lookup procedure, as in the regular case. The number of cascading internal guards can be arbitrary and entirely depends on the concrete call site. Call sites that change their target often,

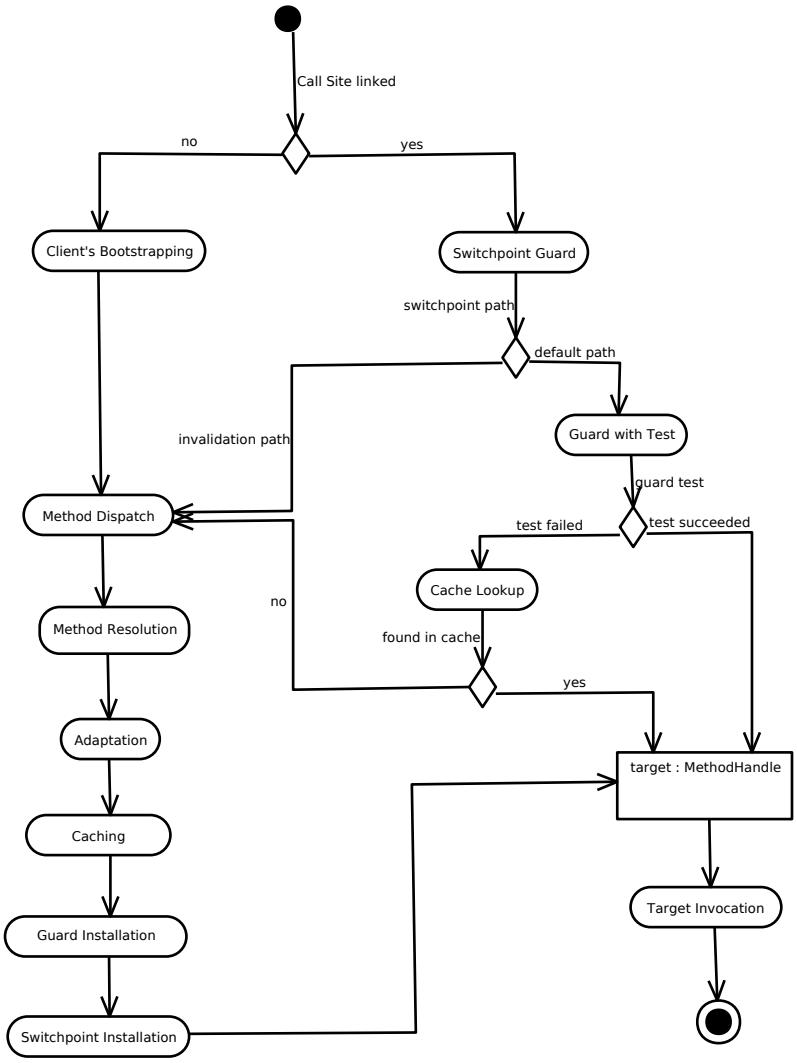


**Figure 4.2:** Control Flow Diagram

due to changing runtime context, can profit from a cascading guard if their targets are rotating. Consider a call site, where the target depends on the runtime type of a given receiver object. Each invocation with a different runtime type would force another method resolution. A regular non cascading guard would test for just one stored runtime type, usually the most recent encountered one. If the call site would constantly be invoked with two different receiver types, the guard would fail in 50% of the invocations, resulting in the usage of the much more expensive, in relation to the guard tests, caching lookup. Such a call site would profit from cascading guards if the number of different receiver types could be guessed or derived. Call sites, where such assumptions can not be derived, usually do not profit from cascading guards with more than two internal guards. The potential performance loss of using a cascading count of Two is negligible in the long run, therefore it is the default size of `Dynamate`'s call sites. Clients are free to increase the default count during the call site's configuration, or decrease it, which would result in the usage of regular non cascading guards.

### Call Site Caching

Additionally to the described caching of resolved method targets, call site caching describes the technique to reuse whole call sites. Call sites contain the cache of method targets and their installed guards. `invokedynamic` instructions, occurring in different places in the program, create naturally different call

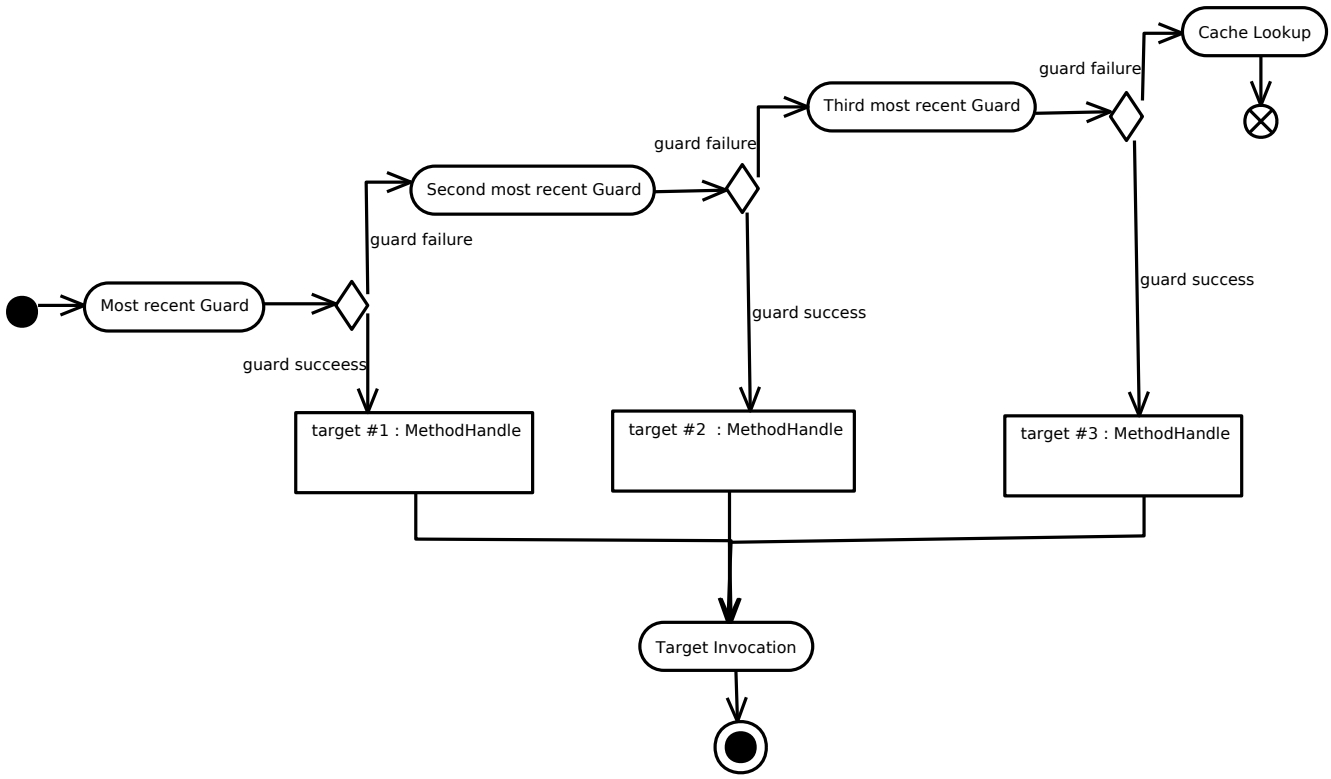


**Figure 4.3:** Cascading Guards Optimization

sites, although they can still specify identical method identifiers and call types and would therefore usually point to the same target iff the static (e.g. the scope) and runtime context stays the same. This results in situations where already resolved call sites could be reused. If clients enable this option, the `MethodDispatcher`, invoked by the client's bootstrap method, would try to lookup a call site with the same identifier and call type. If found, it would directly return a copy of the existing cached call site, including the method target cache and installed guards. If not found, the usual method resolution process would start. This optimization should usually be considered for clients with a very large codebase, potentially consisting of many call sites featuring identical identifiers and call types, or for clients whose method resolution is very expensive.

## 4.6 Source Code Instructions

As stated in 2, there is no native support for source code `invokedynamic` instructions. In fact, there was an official syntax for specifying `invokedynamic` in source code, but it was removed prior to Java 7's release. `Dynamate` includes the `SourceCodeInvoker` class allowing `invokedynamic` calls from source code. This source code support allows rapid prototyping and facilitates debugging efforts. Such source code support can generally be done in two ways: constructing bytecode with tools like ASM and compiling it to Java bytecode at runtime, or routing from one physical call site to multiple logical ones. Both options come with a performance penalty. The first option suffers from dynamic compiling and class loading, while the second option, used by `Dynamate`, suffers from the necessary routing. Therefore, this feature should not be used productively, but should be replaced with native bytecode eventually. The following sequence details `Dynamate`'s source code invoking concept from a developer's perspective. The concrete usage is shown in the following chapter.

# 5 Implemantations

The following chapter shows the implementation of `Dynamate` and the prototypical integration of a subset of the examined clients.

## 5.1 Dynamate's Implemantation

The following sections introduce the relevant components and interfaces of `Dynamate`. First, the components handling the core mechanisms are presented, before Section 5.1 shows the central component and its main entry point: `MethodDispatcher`.

### Guard and Caching Component

The guard and caching mechanism is located in its own component. Figure 5.1 shows the class diagram of all its interfaces and classes.

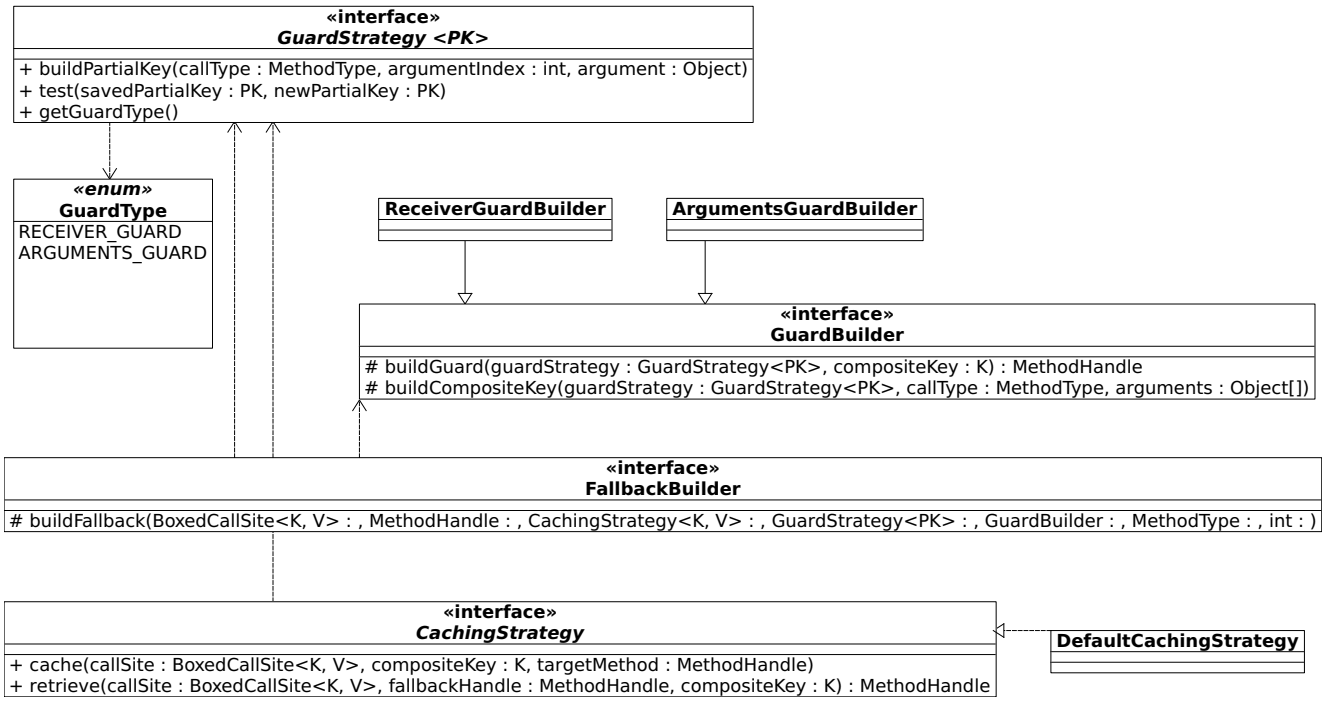


**Figure 5.1:** Guard and Caching Class Diagram

#### GuardStrategy

The `GuardStrategy` interface contains three methods. The generic type PK represents a partial key of arbitrary type. For JRuby, this could be a RubyClass, representing the metaclass of an argument, while for JCop this could be the current layer composition. The `buildPartialKey` method is used to derive a guard and caching key from the runtime context for the current method call. The method is usually called for each argument of the call, individually. It has three parameters: `callType`, represents the static return type and the static parameter types of the current method call, the `index` argument depicts the actual position of the argument in the call type and the `argument` is one of the actual arguments of the call. The method is meant to construct a partial key for the specific argument. Later on, the composite

key, denoted by the generic type K, is build from all partial keys. This concept, inspired by Groovy's `invokedynamic` implementation, allows skipping those arguments, that do not need to be guarded. The final guard is build by cascading individual partial guards, instead of relying on a loop structure, that produces overhead if arguments are skipped. Skipping can be denoted by returning `null` for a specific partial key.

The `test` method is meant to provide a user defined boolean test to compare two partial keys. Usually this could either be an identity test or an equality test, depending on the concrete partial keys.

Finally, the `getGuardType` method is used to denote how the guard should be applied. `GuardType` is a Java enum, consisting of two values: `RECEIVER_TYPE` signals that the guard should only be applied once, on the specific receiver argument, while `ARGUMENTS_GUARD` signals that it should be applied for all arguments, including the receiver.

Guard strategies are bound to a target method during the method selection step in the client's `MethodSelectionStrategy`. Section 5.2 will present some concrete guard strategies of the examined clients.

#### GuardBuilder

The `GuardBuilder` interface denotes a class capable of building a concrete guard for a call site. Its `buildCompositeKey` method is used to construct a composite key with thel help of a guard strategy. The `buildGuard` method can be called to retrieve a method handle pointing to a new guard that offers protection using the passed guard strategy. Two concrete implementations of this interface exist: `ReceiverGuardBuilder` handles the composite key building and guarding if the guard type is set to `RECEIVER_GUARD`, while `ArgumentsGuardType` builds guards for the `ARGUMENTS_GUARD` type.

#### CachingStrategy

A `CachingStrategy` is used to cache and retrieve `MethodHandles`. In both its methods it uses a passed guard strategy to derive the composite key that is used for both caching method handles and retrieving them. `DefaultCachingStrategy` is the concrete caching strategy that should be applicable for most clients, including all examined ones. It caches method handles under exactly one composite key, supplied by the guard strategy. Other client specific implementations could cache the handle under multiple keys, e.g. a caching strategy that splits a key "ABC" into multiple prefix keys "ABC", "AB", "A".

#### FallbackBuilder

A `FallbackBuilder` interface is used to build a concrete fallback handle that is capable of looking up method handles using the passed caching and guard strategies. The `buildFallback` method takes all necessary parameters to construct such a fallback handle. If a cache lookup fails, the fallback instead invokes the initial method resolution method, which will be presented in detail in Section 5.1.

### Invalidation Component

Figure 5.2 depicts the class diagram of the invalidation component.

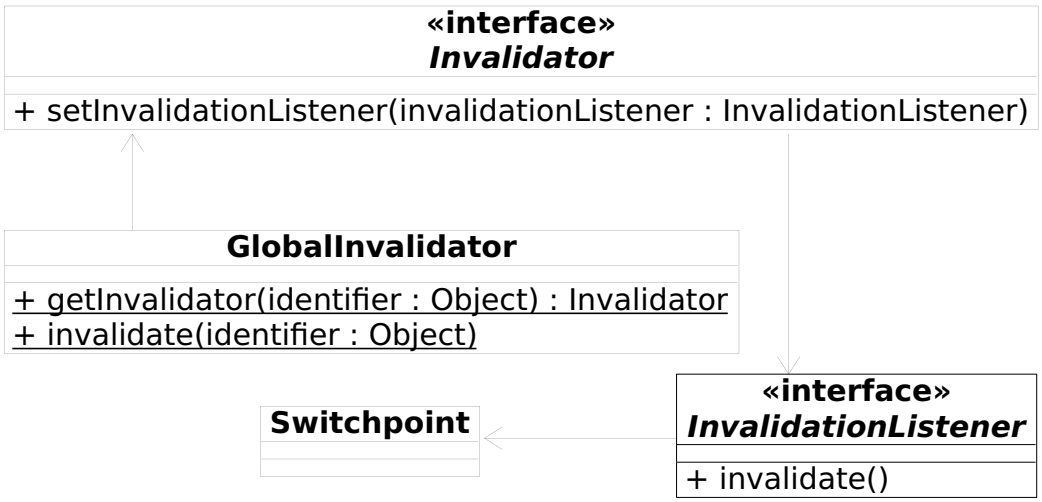


**Figure 5.2:** Invalidatation Class Diagram

#### InvalidationListener

`InvalidationListener` is a class holding a reference to a concrete `SwitchPoint` for a call site. The class provides the method `invalidate` to signal the required invalidation of the switchpoint. Once invalidated, the switchpoint switches transparently to its fallback path.

#### Invalidator

An `Invalidator` is bound to a concrete target by the client during its method selection. `Dynamate` creates the required switchpoint and passes it to the bound invalidator, which can use it from this point on to invalidate the switchpoint. `GlobalInvalidator` is a global registry for invalidator instances, containing two static methods. Clients can request an invalidator instance for any arbitrary identifier using the `getInvalidator` method. `GlobalInvalidator` creates a new invalidator, stores it under the given identifier and returns it to the client, which can use it during its method selection, instead of its own invalidator implementation. Using the `invalidate` method and the specified identifier, clients can invalidate designated switchpoints. The `GlobalInvalidator` is meant for implementation studies and rapid prototyping and should be replaced with custom `Invalidator` implementations for productive usage.

### Adaptation Component

Figure 5.3 shows the class diagram of the adaptation component.

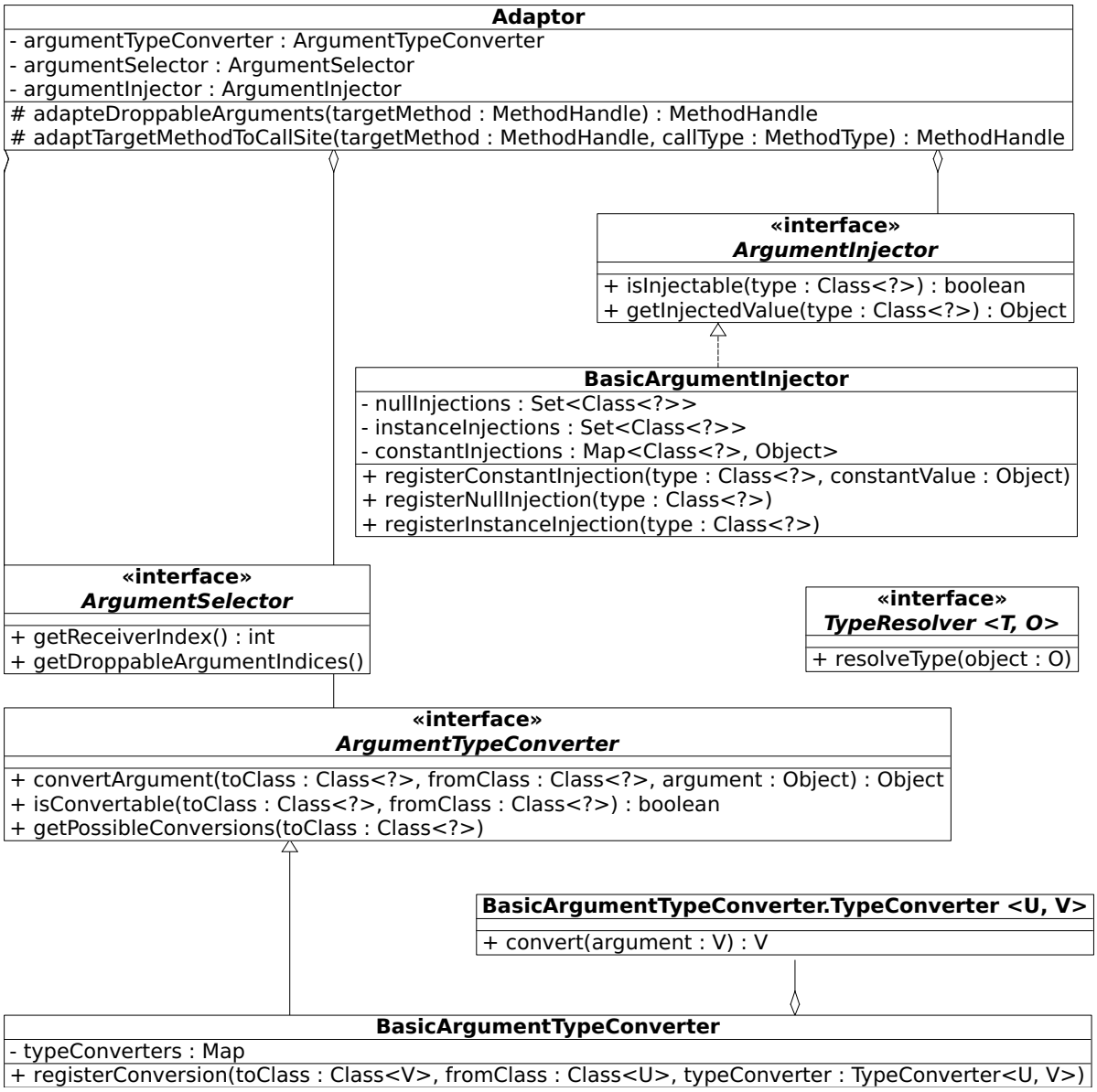


**Figure 5.3:** Adaptation Class Diagram

### ArgumentSelector

The `ArgumentSelector` interface is used to specify the argument position semantics of a method dispatch. Clients should implement this interface with a concrete class if their argument list is different from Java' semantics. The `getReceiverIndex` method denotes the position of the receiver argument in the call site's type, while the `getDroppableArgumentsIndices` method denotes the set of argument positions that must be dropped after the method is selected, i.e., the arguments specified by these indices are not used for the target method's invocation.

### ArgumentTypeConverter

An `ArgumentTypeConverter` is used to convert between different incompatible argument types. Argument type conversions usually convert between types of the host language (in this case Java) to types of the client language (e.g. a conversion of a Java IRubyObject[] array to a JRuby RubyArray object, or a conversion between arrays and lists). The argument type converter is used during method adaptation iff the target method's signature differs from the call site's type. The `isConvertible` and `getPossibleConversions` methods are used to check whether a conversion between two types is possi-

ble, while the convertArgument method is used to actually convert an argument. An argument conversion is applied using a method handle filter, as introduced in Chapter 3. A BasicTypeConverter class is provided that offers a registerConversion method to register TypeConverter instances, converting between two concrete types. However, clients can implement ArgumentTypeConverter with own subclasses.

#### ArgumentInjector

The interface ArgumentInjector is used to inject specified values for missing parameters of a target method. Injection can be used if a target method differs from the call site's type in terms of the argument count, and if all missing arguments are located at the end of the method's parameter list. The isInjectable method checks, whether a value for a given type can be injected and the getInjectedValue returns the value that will be used for argument injection. The implementing class BasicArgumentInjector provides three methods to register injections: registerNullInjection is used to inject the null value for a missing parameter, the registerConstantInjection is used to inject a specific constant value, which can be either a primitive value or an object reference, and registerInstanceInjection is used to inject a new instance of a given class, each time the injection is applied.

#### Adaptor

The Adaptor class is used to adapt the selected target method to the call site's type. It holds references to the described argument adaptation interfaces and uses them in its methods. The adaptDroppableArguments method is used to ensure that the target method is invoked without those arguments that are specified as droppable, while the adaptTargetMethodToCallSite performs the actual adaptation. The method iterates through each argument pair of both the target method's type and the call site's type and performs either an assignment, a conversion, a cast or an injection, in this order, using the supplied (conversions and injections) and implicit (assignments and casts) adaptation policies.

### Method Selection and Object Graph Iteration Component

Figure 5.4 shows the class diagram of the method selection and object graph iterating component.

#### ObjectGraphIterator

The ObjectGraphIterator interface is used to traverse client specific object graphs, as explained in Chapter 4. Clients can implement this interface using their own semantics. The generic type O denotes the type of the receiver argument, e.g. IRubyObject for JRuby and Object for Java. T denotes the type of the receiver's type, e.g. RubyClass for JRuby and Class for Java. The hasNext method is used to check if the current receiver object has a parent in its object graph and next is used to retrieve the parent object and its type. Both methods are invoked with the current receiver and its type. Section 5.2 will present exemplary client object graph iterators.

#### ObjectGraphIterationHistory

The ObjectGraphIterationHistory contains the history of the object graph iteration, i.e., after each invocation of the object graph iterator's next method, the returned (object, type) pair is added to the history. This allows the MethodSelectionStrategy to browse the history and use arbitrary pairs from it for the actual target invocation, which is, in fact, required for has-a dispatch.

#### ArgumentFilter

The ArgumentFilter interface can be implemented by user defined classes that need to filter arguments prior to the target method's invocation. Has-a dispatch can be implemented with the help of this interface. The filter instance is attached to the simplified method handle during method selection and invoked

with the actual call arguments when the method gets called. The filter can manipulate the arguments and return a new list that will be used for the invocation, instead.

### SimplifiedMethodHandle

The `SimplifiedMethodHandle` is a class primarily for storing target method location. It resembles the `MethodHandle` class, but has a simplified interface. Pointing to the desired method is done by specifying an implementation class, the method's name and its type, either as an `MethodType` instance or as an `Class` array. Additionally, a `GuardStrategy`, an `Invalidator` and a directive, specifying the execution order of both regular and switchpoint guard, can be attached to the simplified method handle. Furthermore, it provides two methods to bind and drop parameters of the target method. This has to be done if the target method's type differs from the call site so significantly that an automatic adaptation would be impossible.

### MethodSelectionStrategy

The `MethodSelectionStrategy` is an interface used to select a concrete target method depending on the call site's type, name, receiver, arguments and external runtime context. Clients must implement this interface with their own classes, handling their client specific method selection semantics. Its `selectMethod` method is invoked with the mentioned arguments and expected to return an instance of `SimplifiedMethodHandle`, pointing to the target Method. A suitable guard strategy and, if it makes sense, an invalidator can be bound to the handle for later use.

### ResolvedMethodHandle

The `ResolvedMethodHandle` class represents a simple container for a fully resolved method handle, some context information, e.g. whether the target method is static or virtual and the guard strategy and invalidator. It is constructed by invoking the `resolveMethod` method on the concrete simplified method handle and is used by the `MethodDispatcher`.

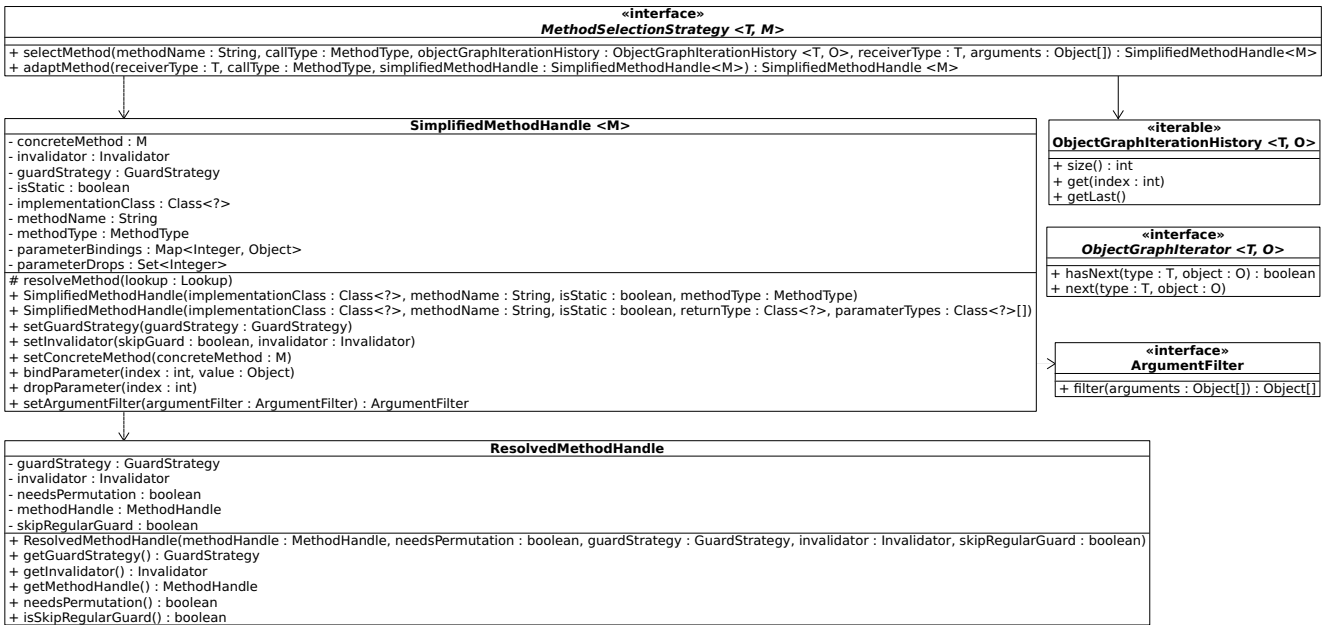


**Figure 5.4:** Method Selection and Object Graph Iteration Class Diagram

Figure 5.5 shows the class diagram of the method dispatch component.

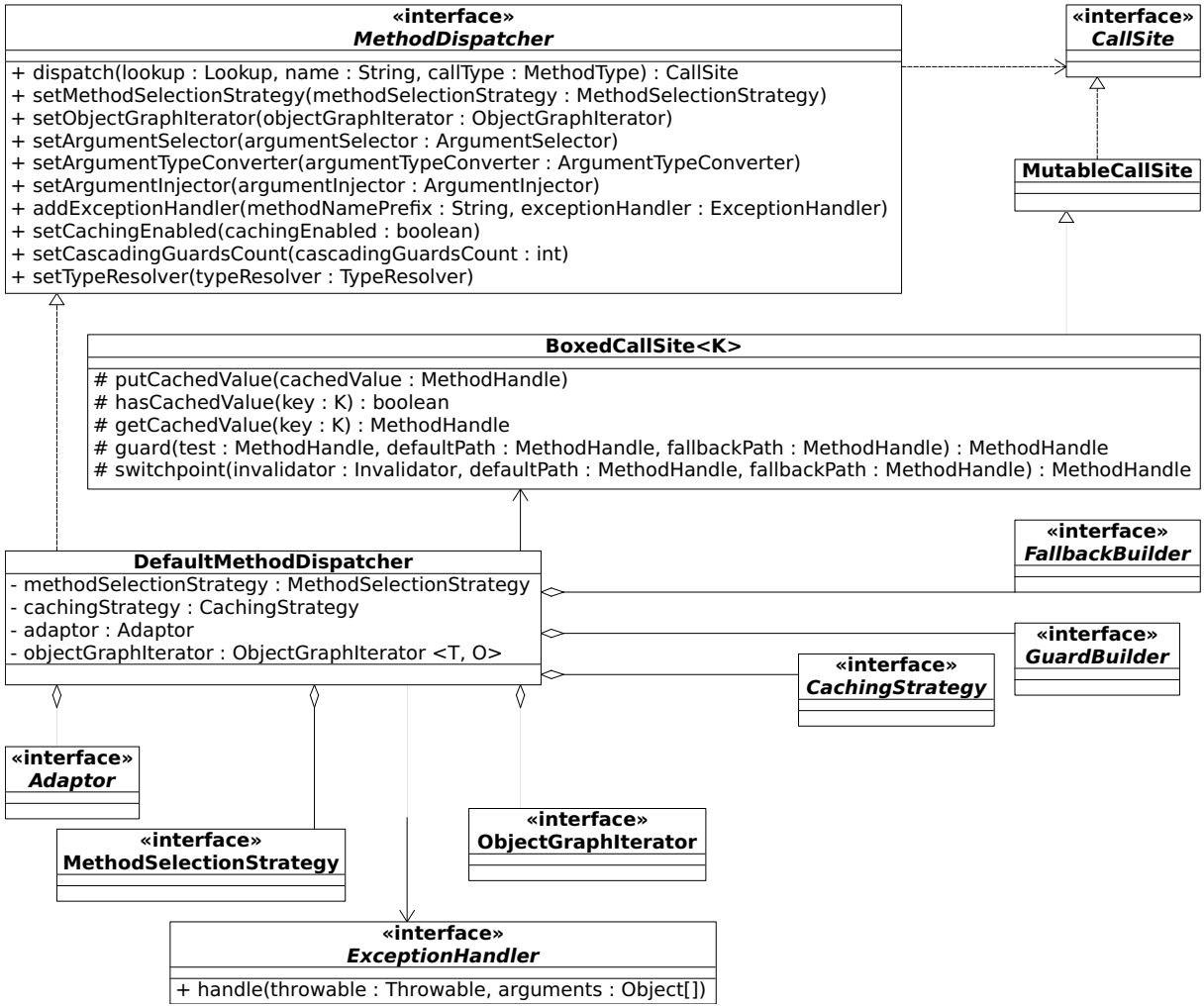


**Figure 5.5:** Method Dispatch Class Diagram

### BoxedCallSite

A `BoxedCallSite` is a subclass of `MutableCallSite`, representing a mutable call site holding a method handle cache as its boxed value. The `putCachedValue` method is used to store a method handle under a given key, while the `hasCachedValue` and `getCachedValue` methods are used to check if a method handle is cached and to retrieve it from the cache. Additionally, the `guard` method is called with concrete test, default path and fallback path method handles, used to install a new guard protecting these handles in the call site, optionally as a cascading guard. The `switchpoint` method installs a switchpoint guard in the call site by creating a new switchpoint and an `InvalidationListener` instance, encapsulating the swichtpoint, and eventually passing this listener to the given `Invalidator` instance, which can use it to invalidate the switchpoint guard.

### ExceptionHandler

The `ExceptionHandler` interface can be implemented by clients if they need to wrap certain call sites with exception handlers. The interface's sole `handle` method is called if an exception occurs. The exception is passed as an `Throwable`, in addition to the actual call arguments. The client can examine the exception and the arguments and react by returning a value that will be passed along to the caller.

Clients can add exception handlers for specific method name prefixes, e.g. the adding of an handler for the prefix "callIter" would result in an exception wrapping for any call, whose method identifier starts with the prefix, e.g. "callIter:msg".

#### MethodDispatcher

`MethodDispatcher` is an interface representing the central entry point for the client integration. Its `dispatch` method has to be invoked in a client's boostrap method to integrate `Dynamate`'s method dispatch. This method returns a new `CallSite` object, which will be returned by the bootstrap method and installed by the JVM. Additionally, the interface provides setter methods to set the desired adaptation policies, the method selection strategy, the object graph iterator, exception handlers and some further options, configuring the guard and caching parameters.

#### DefaultMethodDispatcher

`DefaultMethodDispatcher` is the concrete implementation of `MethodDispatcher`. Its private `resolveMethod` method is the initial target for any call site, that has not been invoked yet. Once invoked, call sites use this method as a fallback method if both the guarding and caching mechanism fail. This method is a dynamic resolution method, as introduced in Chapter 3, i.e., it has not only access to the method identifier and call type, but also to the concrete call arguments. The method performs the following steps to perform the method dispatch.

1. Locate the receiver argument and resolve its type, using `ArgumentSelector` and `TypeResolver`.
2. Invoke `MethodSelectionStrategy` to find a suitable method.
3. If no method is found, traverse the object graph and repeat Step 2 with the new receiver.
4. If a method is found, resolve the `SimplifiedMethodHandle` instance to a `ResolvedMethodHandle` instance.
5. Compare the target's type to the call type and try to adapt between them using `Adaptor`.
6. If adaptation failed, use `MethodSelectionStrategy` to let the client handle the adaptation and then repeat from Step 4
7. Install an object graph iterator in front of the target method for subsequent invocations if needed by the client.
8. Install exception handlers if provided by the client.
9. Use the `GuardStrategy` instance, bound to the target method, to derive a composite key.
10. Cache the target method using `CachingStrategy` and the composite key.
11. Install a guard using the `GuardStrategy` and the composite key.
12. Install a switchpoint guard if an `Invalidator` instance is provided.
13. Invoke the target method using the call arguments.

### Source Code Invoker Component

Figure 5.6 shows the class diagram of the source code invoker component, as introduced in Section 4.6.

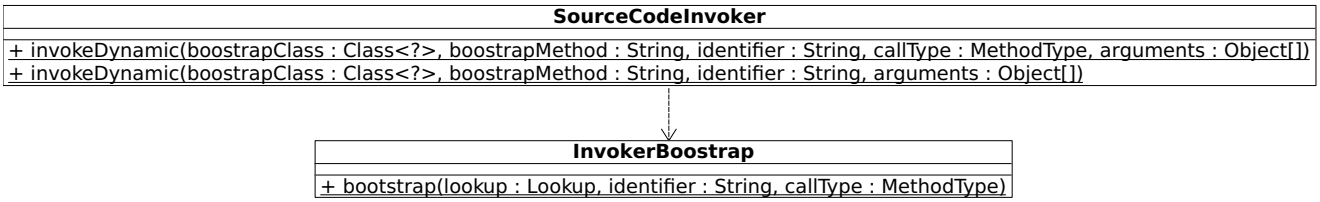


**Figure 5.6:** Source Code Invoker Class Diagram

The `SourceCodeInvoker` class contains two static methods to allow an `invokedynamic` invocation from source code. Its `invokeDynamic` methods both accept the user specified bootstrap class and method, which will be used for bootstrapping the call, and the identifier and concrete call arguments as an `varargs` array. Additionally, one of these methods accepts the specification of a static call type using `MethodType`, while the other implicitly assumes a generic type (e.g. (Object, Object)Object) depending on the count of the arguments. The call is bootstrapped by the `InvokerBootstrap` class that looks up the user specified bootstrap class and method and forwards it the call. Technically, two call sites get created: a physical call site in the `invokeDynamic` method and a logical call site in the `InvokerBootstrap` class. One physical call site can route to multiple logical call sites depending on the call's identifier and type. All the introduced guard and caching mechanisms are used to speed up this forwarding. Listing 5.1 shows a simple example of using the source code invoker to invoke the method "length" on a `String` instance. The call gets bootstrapped by both `InvokerBootstrap` and eventually `ClientBootstrap`.

```
1 int length = SourceCodeInvoker.invokeDynamic(ClientBootstrap.class, "bootstrap", "length",
2   MethodType.methodType(int.class, String.class), "Hello World");
3 // length = 11, assuming Java dispatch semantics
```

**Listing 5.1:** Source Code Invoker Example

## 5.2 Exemplary Client Implemantations

This section will introduce concrete implementations of client integrations. The set of clients consists of JRuby, MultiJava, JCop and JastAdd. Jython and Groovy are omitted, because they are conceptually similar to JRuby. Instead, their implementations can be looked up in `Dynamate`'s source code. Each of the following subsections will detail one client by depicting their following implementations:

1. their argument selector, describing call site argument semantics if any configured,
2. their argument converters and injectors, adapting from call type to method type if any configured,
3. their object graph iteration, specifying graph traversing if any configured,
4. their method selection strategy, detailing concrete dispatch semantics,
5. their guard strategies, protecting the call site's against poll changes,
6. their invalidators, protecting the call site's against push changes,
7. and their bootstrap, showing client specific instantiation and configuration of `Dynamate`.

### JRuby

The prototypical `Dynamate` integration was performed for JRuby 1.7.0 (development version), which already uses `invokedynamic` productively for its method dispatch. Its `InvokeDynamicSupport` class, used to generate bytecode containing `invokedynamic` instructions, was modified for this integration to allow

the usage of a new bootstrap class, separate from the usual JRuby boostrap class. This allowed an integration without modifying any other JRuby classes. JRuby's compiler generates a unified call type for its `invokedynamic` instructions. Every call type is always a prefix of the following `(ThreadContext, IRubyObject, IRubyObject, IRubyObject)IRubyObject)` call type. `ThreadContext` denotes a helper parameter, containing state information and handling the frames. The first `IRubyObject` type represents the class instance encapsulating the call, while the second type depicts the receiver of the call. Additional call arguments are always inserted after the receiver's position and also represented as `IRubyObject` instances, which is actually the interface of the Ruby `Object` class, from which all other types inherit. Although the call type is unified, the actual method implementations have no unified parameter set, which enforces a required adaptation between the call type and the target type. Listing 5.2 shows the argument selector that is used for integrating JRuby with `Dynamate`. As stated, the receiver argument is not the first call argument, but rather the third one, thus the argument selector's `absoluteReceiverIndex` method returns 2. The second call argument represents the encapsulating class instance. This argument is used during method selection, but not for the actual target invocation, therefore the `absoluteDroppableArgumentsIndices` method denotes it as a droppable argument.

```
1  public class JRubyArgumentSelector implements ArgumentSelector {
2    private final static int[] droppableArgumentsIndices = new int[] { 1 };
3
4    @Override
5    public int absoluteReceiverIndex() {
6      return 2;
7    }
8
9    @Override
10   public int[] absoluteDroppableArgumentsIndices() {
11     return droppableArgumentsIndices;
12   }
13 }
```

**Listing 5.2:** Argument Selector for JRuby's Integration

To access the actual class of an `IRubyClass` object, the `getMetaClass` method can be invoked on it, which is part of its interface. Classes are instances of the type `RubyClass`, e.g. a user defined `Car` class would also be an instance of `RubyClass`. The `Dynamate` implementation uses a custom class implementing `Dynamate`'s `TypeResolver` interface to retrieve the type of an object, as shown in Listing 5.3.

```
1  public class JRubyTypeResolver implements TypeResolver<RubyClass, IRubyObject> {
2    @Override
3    public RubyClass resolve(MethodType type, int index, IRubyObject object) {
4      return object.getMetaClass();
5    }
6  }
```

**Listing 5.3:** Argument Type Resolver for JRuby's Integration

JRuby's object graph is quite similar to Java's graph, although the concrete access methods differ. The method `getSuperClass` can be used to get the superclass of a given `RubyClass` instance, which is, of course, also a `RubyClass` instance. Using the object graph iterator depicted in Listing 5.4, `Dynamate` can iterate through the hierarchy of any Ruby object.

```
1  public class JRubyHierarchyResolver implements ObjectGraphIterator<RubyClass, IRubyObject> {
2    @Override
3    public Pair<RubyClass, IRubyObject> next(RubyClass type, IRubyObject object) {
4      return new Pair<RubyClass, IRubyObject>(type.getSuperClass(), object);
5    }
6
7    @Override
8    public boolean hasNext(RubyClass type, IRubyObject object) {
9      return type.getSuperClass() != null;
10   }
11 }
```

**Listing 5.4:** Object Graph Iterator for JRuby's Integration

To facilitate automatic adaptation between call type and target type, the `Dynamate` implementation consists of two argument converters. The `RubyArrayToHostArrayConverter` class is used to convert a

RubyArray instance, representing a Ruby specific array and providing various array operations, to an IRubyObject[] array. Additionally, a SingletonArrayConverter class generates a new IRubyObject[] array holding a given single IRubyObject instance. Listing 5.5 shows the two classes.

```
1  public class SingletonArrayConverter implements TypeConverter<IRubyObject, IRubyObject[]> {
2    @Override
3    public IRubyObject[] convert(IRubyObject object) {
4      return new IRubyObject[] { object };
5    }
6  }
7
8  public class RubyArrayToHostArrayConverter implements TypeConverter<RubyArray, IRubyObject[]> {
9    @Override
10   public IRubyObject[] convert(RubyArray object) {
11     return (IRubyObject[]) object.getList().toArray();
12   }
13 }
```

**Listing 5.5:** Argument Type Converters for JRuby's Integration

Methods of an JRuby class (RubyClass) can be accessed by the method getMethods, which returns a map, associating from the method's name to a concrete method. These methods' parameter types can differ completely from the call type. The so called "direct path" denotes the situation where a call can be matched directly to a retrieved method, either because the types are identical, or because adaptations can be applied automatically, while an "indirect path" must take an indirection to a functional class capable of adapting the call. Additionally, only calls that do not require frame manipulation can take the direct path. The functional class instance representing the indirect path can be retrieved and invoked using a generic "call" method, passing it the call's method name and arguments. Furthermore, some methods require the ThreadContext instance, while others do not, making it a semi droppable argument, complicating an automatic adaptation.

Instance variables are stored in a map in every IRubyObject instance. The setting and accessing of these variables are implicitly modelled as calls on a getter and setter method, using the variable's name as an parameter of the methods.

Listing 5.6 depicts the method selection strategy used for various types of dispatch using Dynamate. In Line 11 the method is looked up in the receiver's class. If no such method exists, the method selection returns and is invoked again on the super class. Otherwise, the retrieved DynamicMethod, which is JRuby's interface for various types of methods, including compiled methods and jit-compiled methods, is examined. If it has no native target, but if is is a an instance field, setting or accessing a variable, an indirect call to the appropriate setter or getter method has to be done, as shown by the Lines 23-27 and 63-85. In other cases, absence of a native target method results in an indirect call, as depicted by the Lines 30 and 97-110. Such absence represents non compiled methods, which can not be invoked directly using invokedynamic. If the dynamic method requires frame manipulation, an indirection has to be taken, in order to call a helper method capable of manipulating it, as shown by the Lines 33 and again 97-110. If a native target was found, a direct path to the method can be taken, presumably. Lines 35-55 depict this case. The simplified method handle is constructed pointing to the native target. If the native target method does not accept a ThreadContext instance, the argument must be declared droppable, as shown by Lines 44-45. Additionally, if the containing class of the target method is a script object, the script instance has to be retrieved and bound to the target. Line 40 stores the dynamic method in the simplified handle for the case that an adaptation could not be performed successfully. In this case, Dynamate would call the adapt method in the method selection strategy with the constructed handle. An indirect call can then be made by the client. All simplified handles are configured with an invalidator, which protects the receiver class and invalidates the switchpoint guard when the class changes, e.g. when a method is replaced with a new implementation. Listing 5.7 shows the receiver guard strategy, which just uses the class of the receiver argument and an identity test to guard the call site. This guard strategy is attached to every handle, apart from handles pointing to functions, because function call targets are independent of concrete arguments and therefore the target does not change.

```
1  @Override
```

```
public SimplifiedMethodHandle<DynamicMethod> selectMethod(GraphIteratingHistory<RubyClass, IRubyObject> resolveHistory,
  MethodType callType, RubyClass receiverType,
    String method, Object[] arguments) {
  String[] identifierParts = method.split(":");

  if (identifierParts.length > 1) {
    method = JavaNameMangler.demangleMethodName(identifierParts[1]);
  }

  DynamicMethod dynamicMethod = receiverType.getMethods().get(method);

  // check of current receiver class contains the method
  if (dynamicMethod != null) {
    // use no guard for function invocations, those cannot change
    GuardStrategy<RubyClass> guardStrategy = !identifierParts[0].startsWith("call") ? null : this.guardStrategy;

    // try to find the native direct path
    NativeCall nativeCall = dynamicMethod.getNativeCall();

    if (nativeCall == null) {
      // method is a setter/getter
      if (dynamicMethod instanceof AttrReaderMethod) {
        return this.getGetterHandle(receiverType, dynamicMethod);
      } else if (dynamicMethod instanceof AttrWriterMethod) {
        return this.getSetterHandle(receiverType, dynamicMethod);
      }

      // no direct path available, must use indiraction
      return this.buildIndirectionHandle(dynamicMethod, receiverType, callType, method, guardStrategy);
    } else if (dynamicMethod.getCallConfig() != CallConfiguration.FrameNoneScopeNone) {
      // cannot use direct path, because frame manipulation required
      return this.buildIndirectionHandle(dynamicMethod, receiverType, callType, method, guardStrategy);
    } else {
      // construct handle to native target method
      SimplifiedMethodHandle<DynamicMethod> handle = new SimplifiedMethodHandle<DynamicMethod>(
        nativeCall.getNativeTarget(), nativeCall.getNativeName(), nativeCall.isStatic(),
        nativeCall.getNativeReturn(), nativeCall.getNativeSignature());

      handle.setConcreteMethod(dynamicMethod);
      handle.setGuardStrategy(guardStrategy);
      handle.setInvalidator(false, GlobalInvalidator.getInvalidator(receiverType));

      // drop ThreadContext iff it is not part of the method's type
      if (this.mustDropThreadContext(nativeCall.getNativeSignature())) {
        handle.drop(0);
      } else if (nativeCall.isStatic() && AbstractScript.class.isAssignableFrom(nativeCall.getNativeTarget())) {
        // retrieve the script instance which contains the native target method
        try {
          handle.bind(0, dynamicMethod.getClass().getMethod("getScriptObject").invoke(dynamicMethod));
        } catch (Exception e) {
          throw new RuntimeException(e);
        }
      }
      return handle;
    }
  }
  // method not contained in current receiver's class, traverse object graph
  return null;
}
private SimplifiedMethodHandle<DynamicMethod> getGetterHandle(RubyClass receiverType, DynamicMethod dynamicMethod) {
  AttrReaderMethod attrReader = (AttrReaderMethod) dynamicMethod;
  return this.getXetterHandle(receiverType, dynamicMethod, attrReader.getVariableName(), "getVariable",
    MethodType.methodType(Object.class, int.class));
}
private SimplifiedMethodHandle<DynamicMethod> getSetterHandle(RubyClass receiverType, DynamicMethod dynamicMethod) {
  AttrWriterMethod attrWriter = (AttrWriterMethod) dynamicMethod;
  return this.getXetterHandle(receiverType, dynamicMethod, attrWriter.getVariableName(), "setVariable",
    MethodType.methodType(void.class, int.class, Object.class));
}
private SimplifiedMethodHandle<DynamicMethod> getXetterHandle(RubyClass receiverType, DynamicMethod dynamicMethod,
  String varName, String xetterType, MethodType type) {
  RubyClass.VariableAccessor accessor = receiverType.getRealClass().getVariableAccessorForWrite(varName);
  SimplifiedMethodHandle<DynamicMethod> handle = new SimplifiedMethodHandle<DynamicMethod>(
    IRubyObject.class, xetterType, false, type);
  handle.drop(0);
  handle.bind(1, accessor.getIndex());
  handle.setGuardStrategy(this.guardStrategy);
  handle.setInvalidator(false, GlobalInvalidator.getInvalidator(receiverType));
  return handle;
}
private boolean mustDropThreadContext(Class<?>[] args) {
  for (Class<?> arg : args) {
    if (arg.equals(ThreadContext.class)) {
      return false;
    }
  }
```

```
88    return true;
89  }
90  private SimplifiedMethodHandle<DynamicMethod> buildIndirectionHandle(DynamicMethod dynamicMethod,
91    RubyClass receiverType, MethodType methodType, String methodName, GuardStrategy<?> guardStrategy) {
92    MethodType modifiedType = methodType.dropParameterTypes(1, 2).insertParameterTypes(2, RubyModule.class, String.class);
93
94    // invoke the functional class instance capable of adapting the method call
95    SimplifiedMethodHandle<DynamicMethod> handle = new SimplifiedMethodHandle<DynamicMethod>(
96      dynamicMethod.getClass(), "call", false, modifiedType);
97    handle.bind(0, dynamicMethod);
98    handle.bind(3, dynamicMethod.getImplementationClass());
99    handle.bind(4, methodName);
100   handle.setGuardStrategy(guardStrategy);
101   handle.setInvalidator(false, GlobalInvalidator.getInvalidator(receiverType));
102   return handle;
103 }
104 @Override
105 public SimplifiedMethodHandle<DynamicMethod> adapt(RubyClass receiverType, MethodType callType,
106   SimplifiedMethodHandle<DynamicMethod> handle) {
107   // Dynamate adaptatatio failed, use indiraction
108   return this.buildIndirectionHandle(handle.getConcreteMethod(),
109     callType, handle.getMethodName(), handle.getGuardStrategy());
110 }
```

**Listing 5.6:** Method Selection Strategy for JRuby's Integration

```
1   private GuardStrategy<RubyClass> guardStrategy = new GuardStrategy<RubyClass>() {
2     @Override
3     public boolean test(RubyClass savedClass, RubyClass receiverClass) {
4       return savedClass == receiverClass;
5     }
6     @Override
7     public RubyClass buildPartialKey(MethodType type, int index, Object argument) {
8       return ((IRubyObject) argument).getMetaClass();
9     }
10    @Override
11    public library.GuardStrategy.GuardType getGuardType() {
12      return GuardType.RECEIVER_GUARD;
13    }
14  };
```

**Listing 5.7:** Guard Strategy for JRuby's Integration

Finally, Listing 5.7 shows the new bootstrap class, used by JRuby's `Dynamate` integration. Additionally to the introduced argument selector and converter classes, an `BasicArgumentInjector` is instantiated, used to inject a default `Block` instance (the NULL block) for missing `Block` parameters. Blocks are optional arguments for many methods in Ruby, therefore this injection happens used quite often. A `JRubyExceptionHandler` instance is created and added to the dispatcher component for three different call prefixes. This exception handler will handle break instructions in JRuby's looping structures. Apart from that, the `MethodDispatcher` instance is instantiated and configured with the mentioned strategies and policies.

```
1   public class Bootstrap {
2     private static MethodDispatcher<RubyClass, IRubyObject> methodDispatcher;
3
4     static {
5       ArgumentInjector argumentInjector = new ArgumentInjector();
6       argumentInjector.registerConstantInjection(Block.class, Block.NULL_BLOCK);
7
8       ArgumentTypeConverter argumentTypeConverter = new ArgumentTypeConverter();
9       argumentTypeConverter.registerTypeConversion(IRubyObject[].class, RubyArray.class, new RubyArrayToHostArrayConverter());
10      argumentTypeConverter.registerTypeConversion(IRubyObject[].class, IRubyObject.class, new SingletonArrayConverter());
11
12      methodDispatcher = new DefaultMethodDispatcher<>();
13      methodDispatcher.setArgumentSelector(new JRubyArgumentSelector());
14      methodDispatcher.setMethodSelectionStrategy(new JRubyMethodSelectionStrategy());
15      methodDispatcher.setObjectGraphIterator(new JRubyHierarchyResolver());
16      methodDispatcher.setTypeResolver(new JRubyTypeResolver());
17      methodDispatcher.setArgumentInjector(argumentInjector);
18      methodDispatcher.setArgumentTypeConverter(argumentTypeConverter);
19
20      JRubyExceptionHandler jRubyExceptionHandler = new JRubyExceptionHandler();
21      methodDispatcher.getExceptionHandlers().put("callIter", jRubyExceptionHandler);
22      methodDispatcher.getExceptionHandlers().put("fcallIter", jRubyExceptionHandler);
23      methodDispatcher.getExceptionHandlers().put("vcallIter", jRubyExceptionHandler);
24    }
25
26    public static CallSite bootstrap(Lookup lookup, String name, MethodType type)
```

```
27      throws NoSuchMethodException, IllegalAccessException {
28      return methodDispatcher.dispatch(lookup, name, type);
29    }
30  }
```

**Listing 5.8:** Bootstrap for JRuby's Integration

## Multijava

As stated in Section 2.3.1, the following implementation is not a concrete Multijava implementation, but rather a prototypical reimplementation using the same semantics for multiple and value dispatch.
Listing 5.12 shows the method selection strategy code of the Multijava reimplementation for multiple dispatch. Lines 5-7 determine the concrete runtime types of the arguments. The runtime type of the receiver was already determined by `Dynamate`. Lines 10-25 iterate through all methods of the receiver's type and, because Java class inheritance semantics are used, through all methods of all supertypes. The applicable methods, i.e., the methods with parameters that are assignable to the call type, are stored temporarily under a parameter type tuple. Line 27-29 denotes that an exception should be thrown if no method was found. Lines 31-41 try to find the most applicable tuple from all considered applicable tuples by iteratively checking whether the currently most applicable tuple is assignable from another tuple. Lines 44-51 setup a `SimpleMethodHandle` with the information of the most applicable tuple, including its concrete implementation class and the method signature. Line 54 attaches a suitable guard strategy to the resolved target handle. The guard strategy is shown in Listing 5.10. It is implemented as an anonymous class instance and uses the runtime type of an argument to construct a partial key and an identity test for comparing partial keys. An invalidator is not sensible for this semantics, because there are no event like changes.

```
1   @Override
2   public SimplifiedMethodHandle<Method> selectMethod(GraphIteratingHistory<Class<?>, Object> resolveHistory,
3     MethodType callType, Class<?> receiverType, String method, Object[] arguments) {
4
5     // determine runtime types of arguments
6     List<Class<?>> argumentTypes = new ArrayList<>(arguments.length + 1);
7     for (Object arg : arguments) {
8       argumentTypes.add(arg.getClass());
9     }
10
11    // find methods in the receiver's class hierarchy
12    Method[] declaredMethods = receiverType.getMethods();
13
14    Map<List<Class<?>>, Method> applicableMethods = new LinkedHashMap<>();
15
16    // find all applicable methods, i.e., all methods stating the same name and argument count
17    for (Method m : declaredMethods) {
18      if (m.getName().equals(method)) {
19        List<Class<?>> parameterTypes = new ArrayList<Class<?>>(Arrays.asList(m.getParameterTypes()));
20        parameterTypes.add(0, m.getDeclaringClass());
21
22        // check if the method's parameter types are compatible to the callType
23        if (this.isApplicable(parameterTypes, argumentTypes)) {
24          applicableMethods.put(parameterTypes, m);
25        }
26      }
27    }
28
29    if (applicableMethods.isEmpty()) {
30      throw new RuntimeException();
31    }
32
33    Entry<List<Class<?>>, Method> mostApplicableEntry = null;
34
35    // out of all applicable methods, determine the most applicable one, i.e.,
36    // the one with the tightest of runtime type matching
37    for (Entry<List<Class<?>>, Method> entry : applicableMethods.entrySet()) {
38      if ((mostApplicableEntry == null) || isApplicable(mostApplicableEntry.getKey(), entry.getKey())) {
39        mostApplicableEntry = entry;
40      }
41    }
42
43    // setup the simplified method handle pointing to the most applicable method
44    Class<?> implementationClass = mostApplicableEntry.getValue().getDeclaringClass();
```

```
45    String methodName = mostApplicableEntry.getValue().getName();
46    boolean isStatic = false;
47    Class<?> returnType = mostApplicableEntry.getValue().getReturnType();
48    List<Class<?>> parameterTypes = mostApplicableEntry.getKey().subList(1, mostApplicableEntry.getKey().size());
49
50    SimplifiedMethodHandle<Method> handle = new SimplifiedMethodHandle<Method>(implementationClass, methodName,
51      isStatic, returnType, parameterTypes);
52
53    // attach guard to protect the target method
54    handle.setGuardStrategy(this.guardStrategy);
55
56    return handle;
57  }
58  private boolean isApplicable(List<Class<?>> typesA, List<Class<?>> typesB) {
59    if (typesA.size() != typesB.size()) {
60      retu  rn false;
61    }
62    for (int i = 0; i < typesA.size(); i++) {
63      if (!typesA.get(i).isAssignableFrom(typesB.get(i))) {
64        return false;
65      }
66    }
67    return true;
68  }
```

**Listing 5.9:** Method Selection Strategy for Multijava's reimplemented Multiple Dispatch

```
1   private GuardStrategy<Class<?>> guardStrategy = new GuardStrategy<Class<?>>() {
2     @Override
3     public Class<?> buildPartialKey(MethodType type, int index, Object argument) {
4       return argument.getClass();
5     }
6     @Override
7     public boolean test(Class<?> savedArgumentClass, Class<?> argumentClass) {
8       return savedArgumentClass == argumentClass;
9     }
10    @Override
11    public library.GuardStrategy.GuardType getGuardType() {
12      return GuardType.ARGUMENTS_GUARD;
13    }
14  };
```

**Listing 5.10:** Guard Strategy for Multijava's reimplemented Multiple Dispatch

Listing 5.12 shows the method selection strategy code of the Multijava reimplementation for value dispatch. It is assumed that value dispatchable methods are annotated using a `@ValueDispatchable` annotation and a special method name syntax specifying all existing sub methods, as Listing 5.11 shows for the method family `foo`, containing one default method and two sub methods. `@ValueDispatchable` specifies all indices of sub methods, while `@OnValue` specifies which values a sub method should handle.

```
1   @ValueDispatchable({1, 2})
2   public void foo(int x, int y) {
3     // default, x and y arbitrary
4   }
5   public void foo$1(@OnValue("1") int x, @OnValue("1") int y) {
6     // x = 1, y = 1
7   }
8   public void foo$2(@OnValue("1") int x, @OnValue("2") int y) {
9     // x = 1, y = 2
10  }
```

**Listing 5.11:** Example of Methods modelling Multijava's reimplemented Value Dispatch

The method selection strategy looks up the default method in Line 9, specified by the call name, and retrieves the annotation pointing to the sub methods. If no annotation is found, the default method gets selected, resulting in regular virtual dispatch that does not need guarding. Lines 19-42 loop through all sub methods and try to find the most applicable one. This is done by examining all parameters of each sub method and checking whether they have an `@OnValue` annotation and whether the annotations value matches the concrete argument. If all annotated parameters match the appropriate arguments, the sub method is selected. Lines 51-52 build a simplified handle pointing to the sub method and attach in Line 55 a guard strategy, protecting the concrete argument values. The guard strategy is detailed in Listing 5.13. The partial key is, in fact, just the argument itself and the test method is an equality test.

```
@Override
public SimplifiedMethodHandle<Method> selectMethod(GraphIteratingHistory<Class<?>, Object> resolveHistory,
  MethodType callType, Class<?> runtimeClass, String method, Object[] arguments) {

  MethodType receiverlessType = callType.dropParameterTypes(0, 1);

  try {
    // lookup the default method
    Method defaultMethod = runtimeClass.getMethod(method, receiverlessType.parameterArray());
    Method target = defaultMethod;

    // try to fetch the ValueDispatchable annotation to find sub methods
    ValueDispatchable annotation = defaultMethod.getAnnotation(ValueDispatchable.class);

    if (annotation != null) {
      // loop through all sub methods denoted by the ValueDispatchable annotation
      for (int methodIndex : annotation.value()) {
        boolean foundMethod = true;

        // lookup the sub method
        Method subMethod = runtimeClass.getMethod(method + "$" + methodIndex, receiverlessType.parameterArray());

        Annotation[][] parameterAnnotations = subMethod.getParameterAnnotations();
        // loop through each parameter's annotation
        for (int i = 1; i < arguments.length; i++) {
          // get the value of the OnValue annotation if set, or null if parameter is not annotaed with it
          Object value = this.getDispatchValue(parameterAnnotations[i - 1]);

          if (value != null) {
            // check whether the concrete argument matches the annotated value,
            // otherwise this sub method must not be considered
            if (!value.equals(arguments[i].toString())) {
              foundMethod = false;
              break;
            }
          }
        }

        // if a suitable sub method is found, set it as the target method
        if (foundMethod) {
          target = subMethod;
          break;
        }
      }

      SimplifiedMethodHandle<Method> handle = new SimplifiedMethodHandle<Method>(target.getDeclaringClass(),
        target.getName(), false, target.getReturnType(), target.getParameterTypes());

      // setup a guard stratgy for the indices that need to be protected, i.e., the ones with an OnValue annotation
      handle.setGuardStrategy(new ValueGuardStrategy()));

      return handle;
    } else {
      // method is not value dispatchable, use always the default one (no guard needed in this case)
      return new SimplifiedMethodHandle<Method>(target.getDeclaringClass(), target.getName(), false,
        target.getReturnType(), target.getParameterTypes());
    }
  } catch (Exception e) {
    throw new RuntimeException(e);
  }
}
// get the value of the OnValue annotation from a set of parameter annotations
private Object getDispatchValue(Annotation[] annotations) {
  for (Annotation a : annotations) {
    if (a.annotationType().equals(OnValue.class)) {
      return ((OnValue) a).value();
    }
  }

  return null;
}
```

**Listing 5.12:** Method Selection Strategy for Multijava's reimplemented Value Dispatch

```
private static class ValueGuardStrategy implements GuardStrategy<Object> {
  @Override
  public Object buildPartialKey(MethodType type, int index, Object argument) {
    // use the concrete value as an partial key
    return argument;
  }
  @Override
  public boolean test(Object savedPartialKey, Object argumentPartialKey) {
    return savedPartialKey.equals(argumentPartialKey);
  }
```

```
11    @Override
12    public library.GuardStrategy.GuardType getGuardType() {
13      return GuardType.ARGUMENTS_GUARD;
14    }
15  };
```

**Listing 5.13:** Guard Strategy for Multijava's reimplemented Value Dispatch

Listing 5.14 shows the boostrap class of the Multijava reimplementation integrating `Dynamate`. The configuration is pretty simple, because Multijava's general semantics are equal to Java's semantics, resulting in the absence of argument selectors, adaptors and object graph iterators. The class instantiates two static `DefaultMethodDispatcher` fields and attaches to them the introduced multiple dispatch and value dispatch strategies. Two bootstraps are implemented, each one dispatching using its concrete selection strategy.

```
1  public class Bootstrap {
2    private static MethodDispatcher<Class<?>, Object> multiDispatcher, valueDispatcher;
3
4    static {
5      multiDispatcher = new DefaultMethodDispatcher<Class<?>, Object, Method>();
6      multiDispatcher.setMethodSelectionStrategy(new MultiDispatchStrategy());
7
8      valueDispatcher = new DefaultMethodDispatcher<Class<?>, Object, Method>();
9      valueDispatcher.setMethodSelectionStrategy(new ValueDispatchStrategy());
10   }
11
12   public static BoxedCallSite bootstrapMultiDispatch(Lookup caller, String name, MethodType type)
13     throws NoSuchMethodException, IllegalAccessException {
14     return multiDispatcher.dispatch(name, type);
15   }
16
17   public static BoxedCallSite bootstrapValueDispatch(Lookup caller, String name, MethodType type)
18     throws NoSuchMethodException, IllegalAccessException {
19     return valueDispatcher.dispatch(name, type);
20   }
21 }
```

**Listing 5.14:** Bootstrap for Multijava

## JCop

As it is the case for Multijava, the following implementation is also not a concrete JCop implementation, but rather a prototypical reimplementation using the same semantics for layered dispatch. It is heavily inspired by the implementation study from [1].

Listing 5.15 shows the example from Section 2.3.2, depicting a `Person` class with a `toString` method, another method `toString$$base` acting as the default method if no layers are active and a `toString$$Address1` method, representing a layered method of the `Address1` layer. The identifier "Person$$toString" in the regular method denotes that either the base method or one of the layered ones should be called at the start of the dispatch, while "proceed:Person$$toString:Address1" denotes that the dispatch should proceed to either base or another layered method, depending on the current layer composition. The proceed identifier contains the current layer name.

```
1  @Override
2  public String toString() {
3    return SourceCodeInvoker.invokeDynamic(Bootstrap.class, "bootstrap", "Person$$toString",
4      MethodType.methodType(String.class, Person.class), this);
5  }
6  private String toString$$base() {
7    return getName();
8  }
9  private String toString$$Address1() {
10   return SourceCodeInvoker.invokeDynamic(Bootstrap.class, "bootstrap", "proceed:Person$$toString:Address1",
11     MethodType.methodType(String.class, Person.class), this) + ", " + getAddress();
12 }
```

**Listing 5.15:** Example of JCop's Reimplementation

The layer composition is stored as a list of layer objects in a `ThreadLocal`, i.e., a composition is only valid for a specific thread. The static methods `Composition.with` and `Composition.without` allow the addition and removal of layers and model the syntax from the example of Listing 2.5. Listing 5.16 shows the method selection strategy code of the JCop reimplementation for layered dispatch. Lines 5-6 determine the actual method and current layer from the passed identifier (e.g. "proceed:Person$$toString:Address1"), decide whether its a `proceed` call and fetch in Line 8 the next layer from the current composition using the determined layer name by iterating through the layer list. Line 10 derives the concrete method name including the suffix and Lines 14-15 construct the appropriate method handle. Line 16 attaches a guard strategy, which is shown in Listing 5.17. This guard strategy constructs partial keys for only the receiver argument and, in fact, by only considering external context: the current layer composition. The layer list is therefore used for the composite key and tested for equality. Additionally, Line 17 in Listing 5.16 registers an invalidator to use for an invalidation guard. The boolean value `true` is used to specify that the regular guard should be skipped as long as the switchpoint is not invalided. The (thread specific) invalidator is used in the `Composition.with` and `Composition.without` methods to invalidate the swichtpoint, i.e., when the layer composition is changed and the switchpoint invalidated, the guard is used to check if the call site's target is still valid.

```
1  @Override
2  public SimplifiedMethodHandle<Method> selectMethod(GraphIteratingHistory<Class<?>, Object> resolveHistory,
3    MethodType callType, Class<?> receiverClass, String identifier, Object[] arguments) {
4
5    String methodName = this.getMethodName(identifier);
6    String layerName = this.getLayerName(identifier);
7
8    Layer nextLayer = this.determineNextLayer(layerName);
9
10   String targetMethodSuffix = (nextLayer == null) ? "base" : nextLayer.getClass().getSimpleName();
11
12   MethodType receiverlessType = callType.dropParameterTypes(0, 1);
13
14   SimplifiedMethodHandle<Method> handle = new SimplifiedMethodHandle<Method>(
15     receiverClass, methodName + "$$" + targetMethodSuffix, false, receiverlessType);
16   handle.setGuardStrategy(this.guardStrategy);
17   handle.setInvalidator(true, GlobalInvalidator.getInvalidator( "cop$" + Thread.currentThread().getId()));
18
19   return handle;
20 }
21 private String getMethodName(String identifier) {
22   String[] split = identifier.split(":");
23   String methodIdentifier;
24
25   if ((split.length == 3) && split[0].equals("proceed")) {
26     methodIdentifier = split[1];
27   } else {
28     methodIdentifier = split[0];
29   }
30
31   return methodIdentifier.split("\\$\\$")[1];
32 }
33 private String getLayerName(String identifier) {
34   String[] split = identifier.split(":");
35   String layerName;
36
37   if ((split.length == 3) && split[0].equals("proceed")) {
38     layerName = split[2];
39   } else {
40     layerName = null;
41   }
42
43   return layerName;
44 }
45 private Layer determineNextLayer(String layerName) {
46   Layer nextLayer = null;
47
48   List<Layer> currentComposition = Composition.getCurrentComposition();
49   if (layerName != null) {
50     for (int i = 0; i < currentComposition.size(); i++) {
51       if (currentComposition.get(i).getClass().getSimpleName().equals(layerName)) {
52         nextLayer = (i + 1) >= currentComposition.size() ? null : currentComposition.get(i + 1);
53         break;
54       }
55     }
56   } else {
57     nextLayer = currentComposition.isEmpty() ? null : currentComposition.get(0);
58   }
```

```
59
60    return nextLayer;
61  }
```

**Listing 5.16:** Method Selection Strategy for JCop's Reimplementation

```
1  private GuardStrategy<List<Layer>> guardStrategy = new GuardStrategy<List<Layer>>() {
2    @Override
3    public List<Layer> buildPartialKey(MethodType type, int index, Object argument) {
4      return Composition.getCurrentComposition();
5    }
6    @Override
7    public boolean test(List<Layer> object, List<Layer> argument) {
8      return object.equals(argument);
9    }
10   @Override
11   public library.GuardStrategy.GuardType getGuardType() {
12     return GuardType.RECEIVER_GUARD;
13   }
14 };
```

**Listing 5.17:** Guard Strategy for JCop's Reimplementation

Listing 5.18 shows the bootstrap class of JCop's reimplementation using `Dynamate`. The general semantics for arguments and inheritance are equal to Java's semantics, therefore no additional configuration is required, apart from the method selection strategy.

```
1  public class Bootstrap {
2    private static final Logger logger = LoggerFactory.getLogger(Bootstrap.class);
3
4    private static MethodDispatcher<Class<?>, Object> layeredDispatcher;
5
6    static {
7      layeredDispatcher = new DefaultMethodDispatcher<Class<?>, Object, Method>();
8      layeredDispatcher.setMethodSelectionStrategy(new LayeredDispatchStrategy());
9    }
10
11   public static BoxedCallSite bootstrap(MethodHandles.Lookup caller, String name, MethodType type)
12     throws NoSuchMethodException, IllegalAccessException {
13     return layeredDispatcher.dispatch(name, type);
14   }
15 }
```

**Listing 5.18:** Bootstrap for JCop's Reimplementation

## JastAdd

The following implementation is based upon the original JastAdd library version R20110902 and incorporates some changes. The original library does not use `invokedynamic`, but rather virtual dispatch, as described in 2.4.1. The modifications were added to the JastAdd code generator (JragCodeGen.jrag) and enable the usage of `invokedynamic` for the inherited attributes dispatch. Listing 2.7 showed that JastAdd generates three call sites, in Lines 3, 9 and 19, for every inherited attribute. The modification consists of replacing two of those three calls with suitable `invokedynamic` calls and removing the method containing the call from Line 3. The method has to be removed, in order to achieve the stated goal of removing unnecessary forwarding to parent nodes. Listing 5.19 states the new calls for that concrete example.

```
1  // Instead of Line 9: generated for the Declaration class (contains attribute declaration)
2  SourceCodeInvoker.invokeDynamic(Bootstrap.class, "bootstrap", "Define_State_lookup",
3    MethodType.methodType(State.class, ASTNode.class, ASTNode.class, ASTNode.class, String.class), this, null, label);
4
5  // Instead of Line 19: generated for the Statemachine class (contains attribute implementation)
6  SourceCodeInvoker.invokeDynamic(Bootstrap.class, "bootstrap", "Define_State_lookup",
7    MethodType.methodType(State.class, ASTNode.class, ASTNode.class, ASTNode.class, String.class), this, caller, label);
```

**Listing 5.19:** Replaced Calls in JastAdd's modified Implementation

Just removing the intermediate calls results in an invocation of the attribute implementation's method with wrong arguments. As Listing 2.7 shows, the intermediate calls change the argument list by using

their own instance for the `caller` parameter and the former `caller` argument for the `child` parameter, effectively dropping the old `child` argument. This is necessary to allow the attribute implementation's node to determine a call's AST path, in order to decide if a call should be handled or forwarded to a parent, e.g. suppose that `Statemachine` has not only `Declaration` children, but also another child of type `X`, which contains the same inherited attribute `lookup`, whose implementation (equation) is instead located in some parent of `Statemachine`. Therefore `Statemachine` must know whether the call came through a `Declaration` child or an `X` child.
Generally, the dispatch of inherited attributes is similar to a has-a relationship dispatch. Additionally, the target implementation must be called with arguments of the last two intermediate nodes. Consider an AST "D@d->C@c->B@b->A@a". A small letter denotes an instance and a capital letter denotes its type. The type D contains the inherited attribute, A (the AST's root) contains the inherited attribute's implementation and C and B are intermediate node types. To ensure the introduced call semantics, prior to the target's invocation, the receiver node (d) has to be substituted by the parent node containing the target method (a). Additionally, the target's call arguments (caller and child) must reflect the two intermediate nodes (b and c).
Technically, a `MethodHandle` filter has to be applied on the target method to modify the call arguments. The modified JastAdd implementation uses the `ArgumentFilter` interface, provided by `Dynamate`, to achieve this. During the method selection, an instance of this filter is attached to the simplified method handle, replacing the arguments depending on the concrete object graph iteration history, passed by `Dynamate`. Listing 5.20 shows the implented object graph iterator, which enables the traversing of JastAdd's ASTs.

```
1  public class JastAddAstTraversing implements ObjectGraphIterator<Class<?>, ASTNode> {
2
3    @Override
4    public Pair<Class<?>, ASTNode> next(Class<?> type, ASTNode node) {
5      return new Pair<Class<?>, ASTNode>(node.getParent().getClass(), node.getParent());
6    }
7
8    @Override
9    public boolean hasNext(Class<?> type, ASTNode object) {
10     return object.getParent() != null;
11   }
12 }
```

**Listing 5.20:** Object Graph Iterator for JastAdd's modified Implementation

Listing 5.21 depicts the method selection strategy used for the JastAdd's modified implementation. Lines 4-6 check if the method selection is requested for the original receiver node, as the inherited attribute's implementation can not be contained in the same node containing the attribute's declaration. The mentioned object graph iterator is used to fetch the parent node of the receiver, which passes this test. Lines 11-15 look up the requested method in the current node's class. If it does not exist, the node must be an intermediate node and can be skipped. Eventually, the method selection is requested for the node containing the attribute implementation method, thus in Lines 17-18 a simplified method handle pointing to it, is constructed.

```
1  @Override
2  public SimplifiedMethodHandle<Method> selectMethod(GraphIteratingHistory<Class<?>, ASTNode> history,
3    MethodType callType, Class<?> receiverType, String methodName, Object[] arguments) {
4    if (resolveHistory.size() == 0) {
5      return null;
6    }
7
8    MethodType receiverlessType = callType.dropParameterTypes(0, 1);
9    Method method = null;
10
11   try {
12     method = receiverType.getMethod(methodName, receiverlessType.parameterArray());
13   } catch (NoSuchMethodException | SecurityException e) {
14     return null;
15   }
16
17   SimplifiedMethodHandle<Method> handle = new SimplifiedMethodHandle<Method>(receiverType, methodName, false,
18     method.getReturnType(), method.getParameterTypes());
19   handle.setArgumentFilter(new JastAddArgumentFilter(this.objectGraphIterator, history));
20   handle.setGuardStrategy(this.guardStrategy);
```

```
21
22    return handle;
23  }
```

**Listing 5.21:** Method Selection Strategy for JastAdd's modified Implementation

Line 18 of Listing 5.21 attaches an instance of the mentioned argument filter to the handle, referencing the object graph iterator and the actual object graph iteration history. The filter is shown in Listing 5.22 and modifies the arguments by traversing their AST accordingly to the history. The filter is called prior to every invocation of the target.

```
1  public class JastAddArgumentFilter<T, O> implements ArgumentFilter {
2
3    private JastAddAstTraversing astTraversing;
4    private GraphIteratingHistory<T, O> history;
5
6    public JastAddArgumentFilter(JastAddAstTraversing astTraversing, GraphIteratingHistory<T, O> history) {
7      this.astTraversing = astTraversing;
8      this.history = history;
9    }
10
11   @Override
12   public Object[] filter(Object[] arguments) {
13     Object[] modifiedArguments = new Object[arguments.length];
14     System.arraycopy(arguments, 0, modifiedArguments, 0, arguments.length);
15
16     Class<?> currentType = arguments[0].getClass();
17     ASTNode currentObject = (ASTNode) arguments[0];
18
19     LinkedList<Object> list = new LinkedList<>();
20     list.addFirst(currentObject);
21
22     // iterate the object graph according to the object graph history of the resolved call
23     for (int i = 0; i < (this.history.size() - 1); i++) {
24       Pair<Class<?>, ASTNode> next = this.astTraversing.next(currentType, currentObject);
25       currentType = next.getX();
26       currentObject = next.getY();
27       list.addFirst(currentObject);
28     }
29
30     // change original node to target node containing the attribute method
31     modifiedArguments[0] = list.get(0);
32
33     // change 'caller' parameter if more than one intermediate node between original node and target node
34     if (list.size() > 1) {
35       modifiedArguments[1] = list.get(1);
36     }
37
38     // change 'child' parameter if more than two intermediate node between original node and target node
39     if (list.size() > 2) {
40       modifiedArguments[2] = list.get(2);
41     }
42
43     return modifiedArguments;
44   }
45 }
```

**Listing 5.22:** Argument Filter for JastAdd's modified Implementation

Line 19 of Listing 5.21 attaches a guard strategy to the handle. For JastAdd, choosing a sensible guard is no trivial matter. The resolved target depends not only on the receiver's node type, but also on the concrete path to the target node. Consider the following two ASTs: (C@c1->B@b1->A@a1) and (C@c2->B@b2->A@a2). If a call on an inherited attribute in c1 gets resolved to the target method in A (using the context of a1), another invocation of this call site with c2 does not need to be resolved again, because their path types are equal, although the path instances are not. However, the introduced argument filter automatically ensures that the method is invoked in the context of a2. Therefore, a guard is needed that builds the key from the nodes path to the AST's root. JastAdd's code generator is modified (in JaddCodeGen.jrag) by adding an instance field `astPredecessors` to the `ASTNode` class, thus making it available in all node types. This field is a string representing the current path of a node to its root parent node. The string has the mentioned abstract form "C@c->B@b->A@a" and is reconstructed every time the node moves in its AST, either due to initial tree construction or rewrite rules. Using this, always correct, string for composite keys, a receiver guard strategy can be attached, as seen in 5.23. The

strategy uses, in fact, only the type information from the receiver's path string, because, as mentioned, the argument filter already ensures the invocation with the correct instances.

```
@Override
public boolean test(String savedNodePath, String argumentNodePath) {
  return savedNodePath.equals(argumentNodePath);
}
@Override
public String buildPartialKey(MethodType type, int index, Object argument) {
  if (index == 0) {
    ASTNode<?> node = (ASTNode<?>) argument;
    return node.astPredecessors.replaceAll("@\\d+", "");
  }

  return null;
}
@Override
public library.GuardStrategy.GuardType getGuardType() {
  return GuardType.RECEIVER_GUARD;
}
```

**Listing 5.23:** Guard Strategy for JastAdd's modified Implementation

Listing 5.18 finally shows the bootstrap class of JastAdd's modified implementation using `Dynamate`. The configured semantics are similar to Java's semantics, apart from the object graph iteration, which is therefore provided.

```
public class Bootstrap {
  private static MethodDispatcher<Class<?>, ASTNode> methodDispatcher;

  static {
    JastAddAstTraversing objectGraphIterator = new JastAddAstTraversing();

    JastAddMethodSelectionStrategy methodSelectionStrategy = new JastAddMethodSelectionStrategy(objectGraphIterator);

    methodDispatcher = new DefaultMethodDispatcher<Class<?>, ASTNode, Method>();
    methodDispatcher.setObjectGraphIterator(objectGraphIterator);
    methodDispatcher.setMethodSelectionStrategy(methodSelectionStrategy);
  }

  public static BoxedCallSite bootstrap(Lookup lookup, String name, MethodType type)
    throws NoSuchMethodException, IllegalAccessException {
    return methodDispatcher.dispatch(name, type);
  }
}
```

**Listing 5.24:** Bootstrap for JCop's Reimplementation

# 6 Evaluation

This chapter focuses on the evaluation of Dynamate by means of evaluating its runtime performance and usability. The first part of Section 6.1 describes the performance of some of Dynamate involved components, while the second part presents actual benchmarks and results used to measure the performance of the examined clients with and without Dynamate's integration. Section 6.2 discusses briefly Dynamate's usability and required integration effort.

## 6.1 Performance

The following two subsections evaluate the performance of Dynamate. Section 6.1.1 details the overhead and gains of Dynamate's internal mechanisms, while Section 6.1.2 presents actual benchmarks and results some of the examined clients.

### 6.1.1 Performance of Dynamate's Components

This section measures the performance of Dynamate's guard and caching mechanisms for different types of call sites.

#### Guards

Figure 6.1 depicts the normalized performance gains of Dynamate's guard mechanism in relation to the probability of a call site mutation, due to runtime context changes. The measurement was performed with 10 possible runtime context states and 1.000.000 call site invocations. At x=0, the context never changes, and at x=100, the context changes on each invocation to one of its possible states. The base function represents a call site with no guards and no caching, resulting in a renewed method resolution, each time the call site gets invoked. Because no guard is installed, the resolution has to occur every time, independently from the mutation probability, leading to a constant runtime. The other functions represent call sites with a varying count of cascading guards.

The performance gains of the guard mechanism is enormous, but it is anti proportional to the mutation probability, resulting in decreasing gains and even a slight overhead if the call site's target changes too often. Usage of cascading guards reduce the overhead in these cases. However, the benefit from cascading guards is not linear, but declines with each additional internal guard. Cascading guards with two or three internal guards seem to be a good choice that does not result in too much memory overhead.

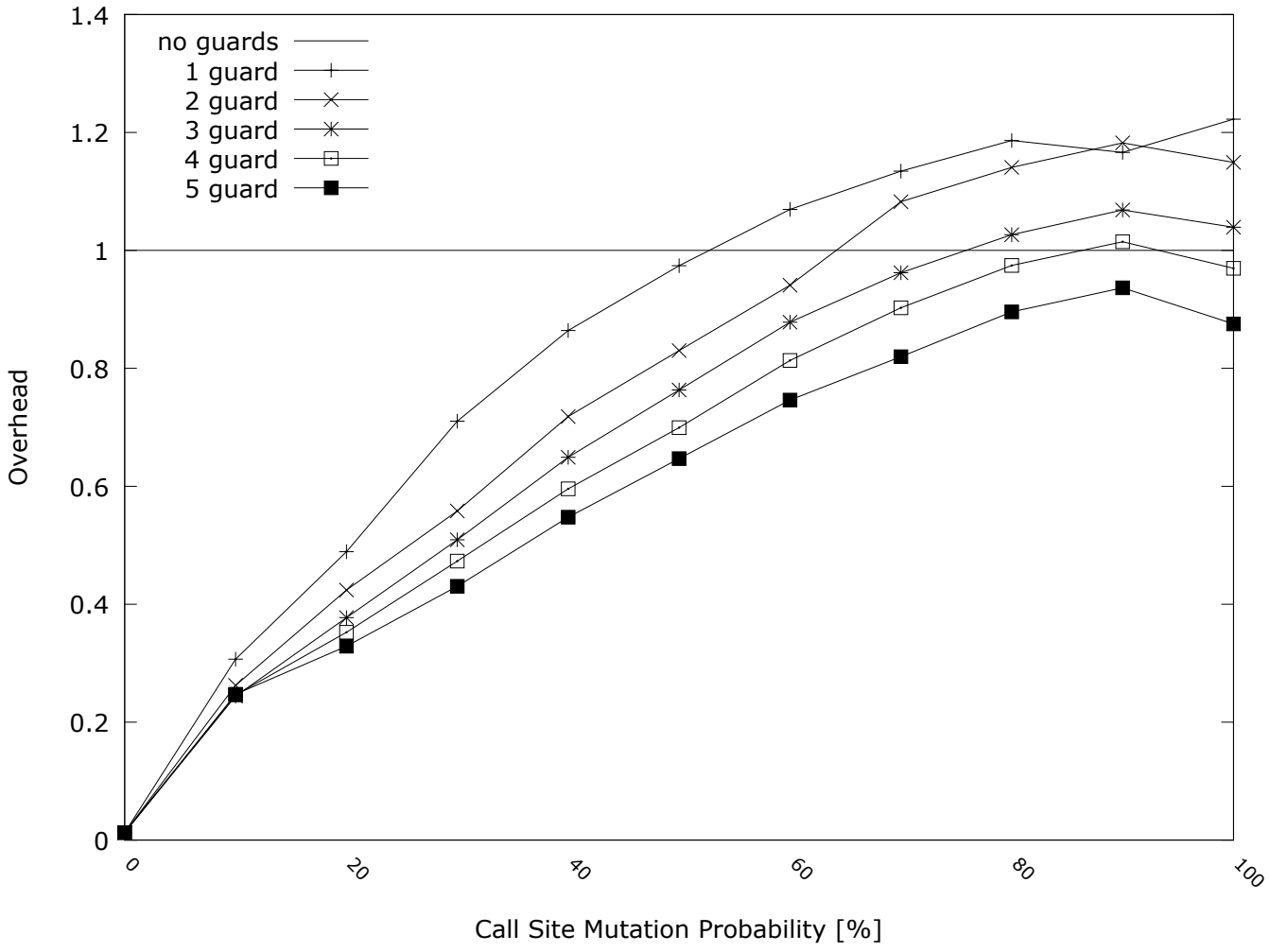


**Figure 6.1:** Normalized performance overhead of `Dynamate`' guard mechanism with a varying number of cascading guards

## Caching

Figure 6.2 shows the normalized performance gains of `Dynamate`'s caching mechanism in combination with the guard mechanism. The assumptions are the same as in Figure 6.1. Both its base function and the function depicting a call site with one guard, is shown. Additionally, a call site with no guard, but enabled caching, and a call site combining both mechanism, is pictured. The caching mechanism performs individually better than the guard mechanism for increasing mutation probability, because more resolved targets get cached and can be looked up. The combination of both mechanism performs much better than the individual scenarios and is therefore indispensable for a good performance. It is even no longer proportional to the mutation probability, making it applicable for any situation. Further usage of cascading guards improve the performance only slightly at lower mutation probability and is therefore only useful in situations where assumptions about the runtime context can be made.

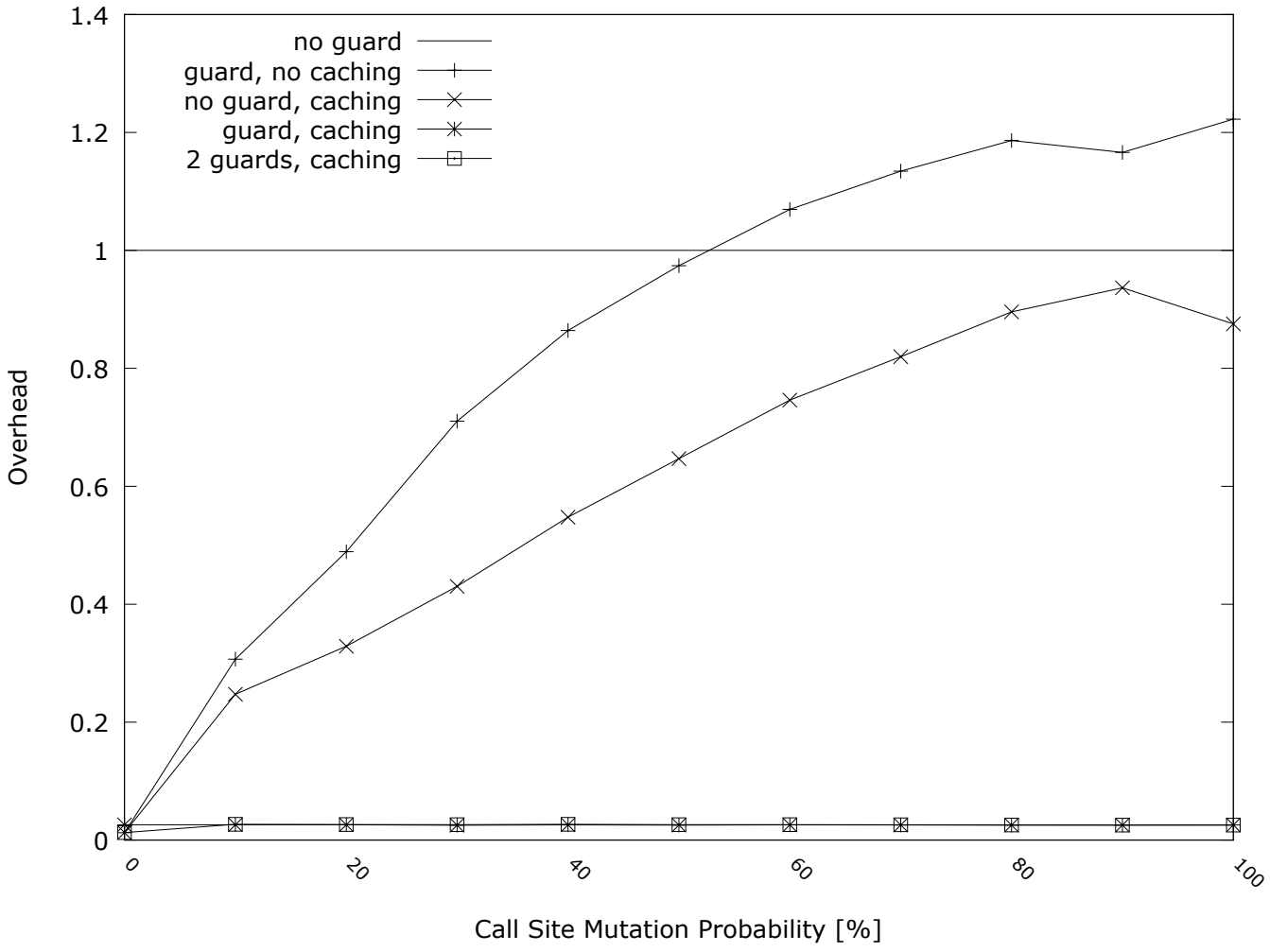


**Figure 6.2:** Normalized performance overhead of `Dynamate`' caching mechanism individually and in combination with a guard

### 6.1.2 Performance of Clients

The performance of the clients is measured in terms of total run time for a specific task. Most of the benchmarks used for measurement of the three programming language clients originate from "The Computer Language Benchmarks Game"[1] (CLBG) and represent common algorithmic problems. The benchmarks for the two language extension clients are self implemented and the benchmark for JastAdd consists of measuring the runtime of JastAddJ's[2] enclosed compiler program. Each client's performance is evaluated by introducing the concrete benchmarks, presenting and analysing the results and finally concluding, if the client profits from integrating `Dynamate`.

#### JRuby

JRuby is evaluated by performing 12 benchmarks on the original implementation, already using `invokedynamic`, as stated in Chapter 2, and on `Dynamate`'s implementation, introduced in Chapter 5.

[1] http://shootout.alioth.debian.org
[2] http://jastadd.org/web/jastaddj

Ten of these benchmarks originate from CLBG and are performed using the most common input parameters. The final two benchmarks are self implemented and represent very simple programs invoking the same call site numerous times with different arguments. Their shared declaration is shown in Listing 6.1, which just declares three different non related classes X, Y and Z with the same named method `foo`. Additionally, a function `bar` is declared, which invokes the mentioned `foo` method on the passed argument, resulting in a duck typed method dispatch. 6.2 shows the concrete program of the sequential dispatch. The `foo` method gets invoked the specified times on an instance of X and then sequentially in a second and third loop on the instances of Y and Z. 6.3 in contrast, invokes the `foo` method on all instances of X, Y and Z in the same loop alternatingly. These two benchmarks represent call sites that change their target seldom (in the sequential benchmark) and always (in the alternating benchmark) and therefore test both the call sites' default and fallback paths extensively.

```
1  class X
2    def foo
3      "X.foo()"
4    end
5  end
6
7  class Y
8    def foo
9      "Y.foo()"
10   end
11 end
12
13 class Z
14   def foo
15     "Z.foo()"
16   end
17 end
18
19 def bar(obj)
20   obj.foo()
21 end
```

**Listing 6.1:** Sequential and alternating dispatch prelude

```
1  N = Integer(ARGV[0]) / 3
2
3  x = X.new
4  y = Y.new
5
6  for i in 0..N
7    bar(x)
8  end
9
10 for i in 0..N
11   bar(y)
12 end
13
14 for i in 0..N
15   bar(z)
16 end
```

**Listing 6.2:** Sequential dispatch program

```
1  N = Integer(ARGV[0]) / 3
2
3  x = X.new
4  y = Y.new
5
6  for i in 0..N
7    bar(x)
8    bar(y)
9    bar(z)
10 end
```

**Listing 6.3:** Alternating dispatch program

Figure 6.3 shows the normalized runtime overhead of `Dynamate`'s integration against the original JRuby implementation and Table 6.6 states the absolute run times. Overall, `Dynamate` performs quite equal to the original implementation with some benchmarks having a slight overhead and some having a low performance gain. This result shows that `Dynamate` is quite suited for integration with JRuby. Although

JRuby would not gain any real performance advantage, it would not lose any either, eventually profiting from a central framework's other advantages, usability and maintainability.

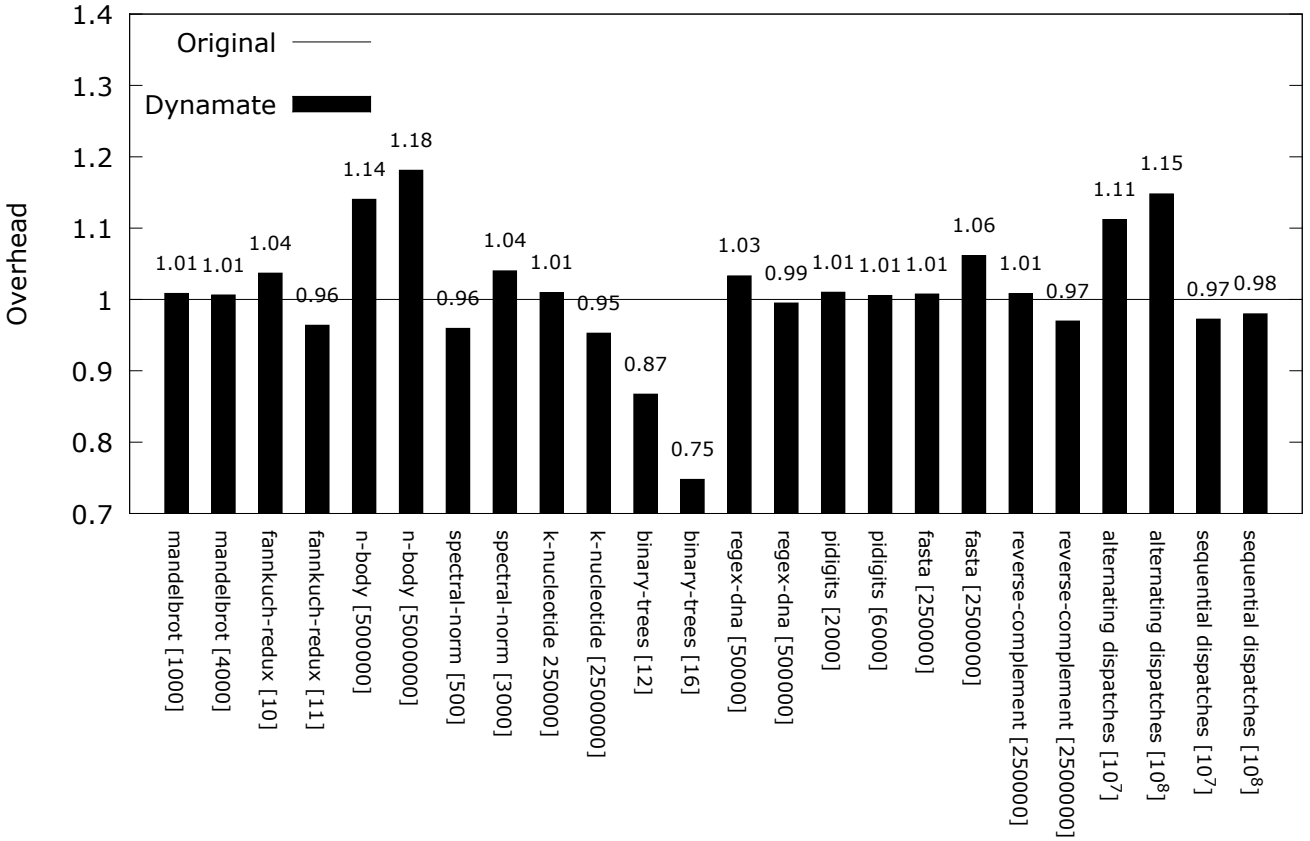


**Figure 6.3:** Normalized performance results of the `Dynamate` implementation against the original JRuby implementation

## Jython

Jython is evaluated by executing benchmarks on the jython-pilot implementation, which uses `invokedynamic` as introcuded in Chapter 2, and on the `Dynamate` prototypical implementation. The benchmarks are mostly implementations from CLBG. Additionally, the two self implemented sequential and alternating dispatch benchmarks are used, this time, adapted for the Python language. Figure 6.4 shows the normalized runtime overhead of `Dynamate`'s integration against the jython-pilot implementation and Table 6.2 the absolute benchmark run times. The overall performance of `Dynamate` is fairly equal to the original implementation, making it a good candidate for a real integration.

| Benchmark | Original [ms] | Dynamate [ms] |
|---|---|---|
| mandelbrot [1000] | 3554 | 3584 |
| mandelbrot [4000] | 40275 | 40530 |
| fannkuch-redux [10] | 5170 | 5361 |
| fannkuch-redux [11] | 53583 | 51645 |
| n-body [500000] | 4202 | 4792 |
| n-body [5000000] | 30850 | 36436 |
| spectral-norm [500] | 1705 | 1636 |
| spectral-norm [3000] | 19476 | 20256 |
| k-nucleotide 250000] | 1967 | 1986 |
| k-nucleotide [2500000] | 6123 | 5834 |
| binary-trees [12] | 1152 | 999 |
| binary-trees [16] | 9847 | 7363 |
| regex-dna [50000] | 1061 | 1096 |
| regex-dna [500000] | 1969 | 1959 |
| pidigits [2000] | 1096 | 1107 |
| pidigits [6000] | 3557 | 3577 |
| fasta [250000] | 2642 | 2662 |
| fasta [2500000] | 12889 | 13680 |
| reverse-complement [250000] | 1335 | 1346 |
| reverse-complement [2500000] | 2408 | 2335 |
| alternating dispatches [10000000] | 937 | 1042 |
| alternating dispatches [100000000] | 3042 | 3492 |
| sequential dispatches [10000000] | 1200 | 1167 |
| sequential dispatches [100000000] | 4896 | 4797 |

**Table 6.1:** Absolute benchmark runtimes of the `Dynamate` implementation and original JRuby implementation

| Benchmark | Original [ms] | Dynamate [ms] |
|---|---|---|
| simple [10000] | 2788 | 2726 |
| simple [100000] | 9725 | 8867 |
| fannkuch-redux [10] | 8599 | 7708 |
| fannkuch-redux [11] | 68380 | 75851 |
| n-body [500000] | 10514 | 10462 |
| n-body [5000000] | 87934 | 88562 |
| sequential dispatch 10000] | 1770 | 1750 |
| sequential dispatch [100000] | 1859 | 1879 |
| alternating dispatch [10000] | 1759 | 1699 |
| alternating dispatch [100000] | 1855 | 1815 |

**Table 6.2:** Absolute benchmark runtimes of the `Dynamate` implementation and original Jython implementation

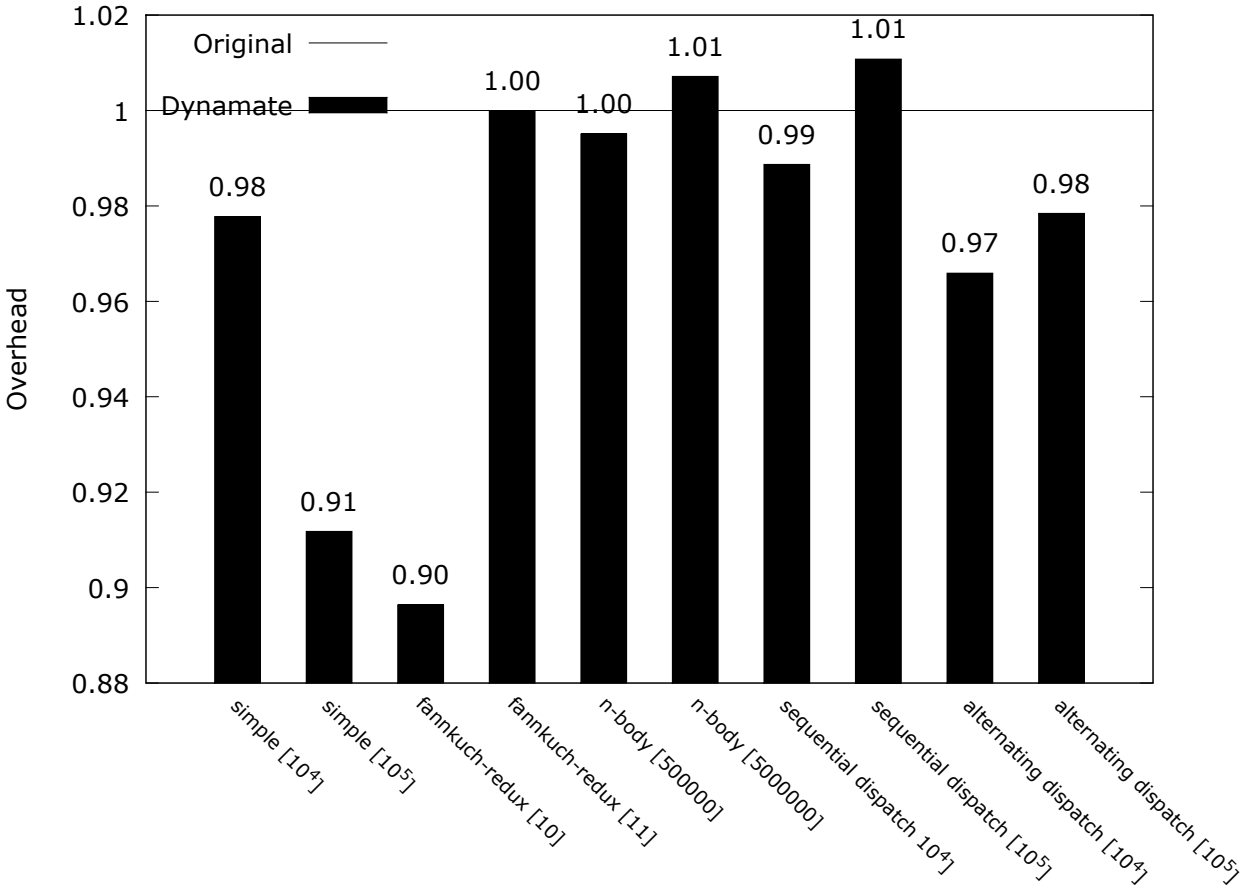


**Figure 6.4:** Normalized performance results of the `Dynamate` implementation against the original Jython implementation

### Groovy

Groovy is evaluated by executing benchmarks on the Groovy implementation, which also already uses `invokedynamic` quite extensively, as introduced in Chapter 2, and on the `Dynamate` implementation. The benchmarks are mostly implementations from CLBG. Once again, the two self implemented sequential and alternating dispatch benchmarks are used, adapted for the Groovy language. Figure 6.4 shows the normalized runtime overhead of `Dynamate`'s integration against the Groovy implementation. Table 6.3 states the concrete run times. The overall performance of `Dynamate` is unfortunately fluctuating. Some benchmarks perform better, while some perform worse than the original Groovy implementation. Likely, this is due to an insufficient guarding design in the `Dynamate` integration implementation that should be redesigned. However, particularly the "alternating dispatch" benchmark shows that `Dynamate` has advantages if the call site's target changes permanently.

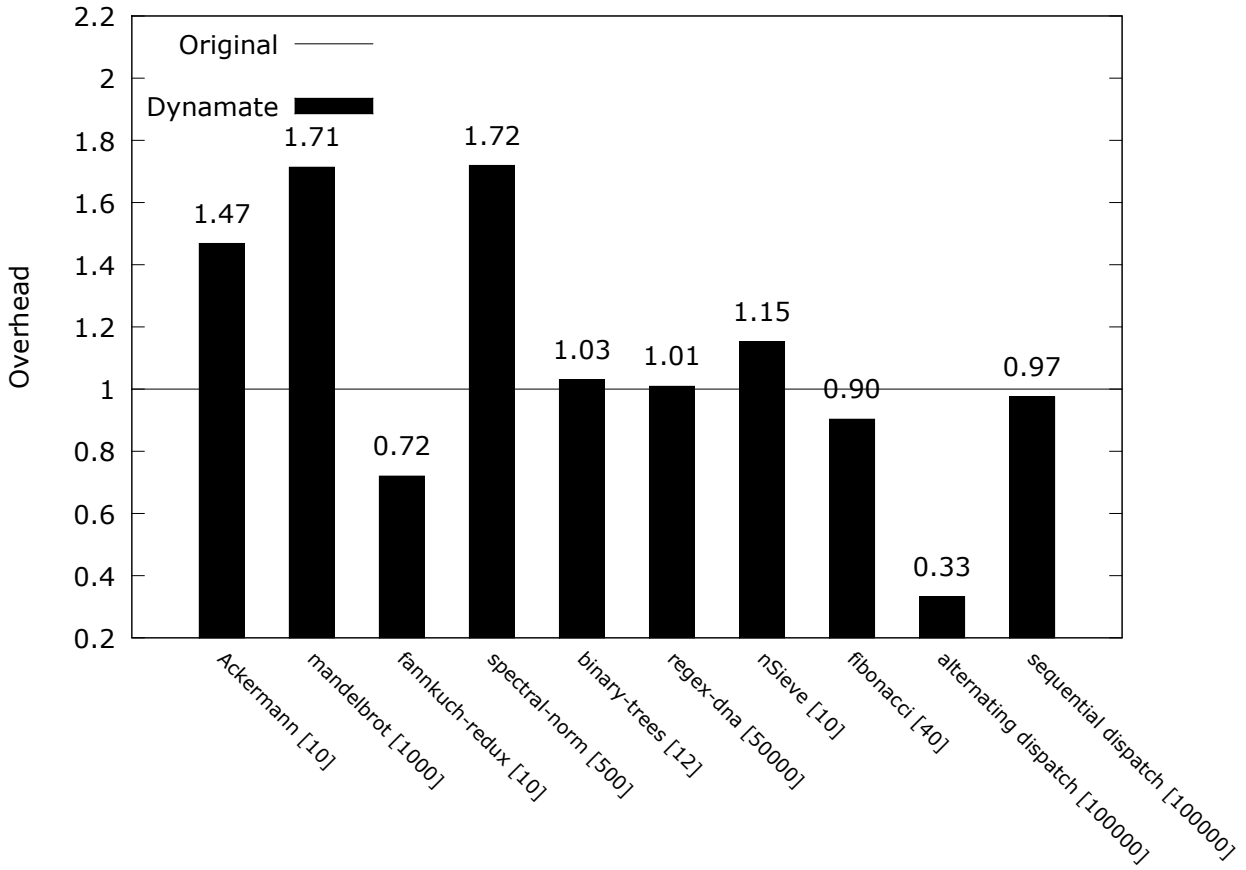


**Figure 6.5:** Normalized performance results of the `Dynamate` implementation against the original Groovy implementation

### Multijava

Multijava is benchmarked using the Shapes hierarchy example, introduces in Chapter 2. Both implementations, the original Multijava, not using any `invokedynamic`, and `Dynamate`'s reimplementation

| Benchmark | Original [ms] | Dynamate [ms] |
|---|---|---|
| Ackermann [10] | 2271 | 3335 |
| mandelbrot [1000] | 6293 | 10784 |
| fannkuch-redux [10] | 101817 | 73301 |
| spectral-norm [500] | 2438 | 4192 |
| binary-trees [12] | 1139 | 1174 |
| regex-dna [50000] | 1030 | 1039 |
| nSieve [10] | 4373 | 5038 |
| fibonacci [40] | 2926 | 2642 |
| alternating dispatch [100000] | 2300 | 764 |
| sequential dispatch [100000] | 791 | 771 |

**Table 6.3:** Absolute benchmark runtimes of the `Dynamate` implementation and original Groovy implementation

execute three simple self written benchmarks, all invoking one call site, but with different argument lists. The first benchmark, named "Constant List", creates one list of arguments and repeatedly invokes the call site with this list, resulting in exactly one resolved target using multiple dispatch. The second benchmark, called "Shuffled Arguments", uses a shuffled list of arguments, resulting eventually in many different resolved targets using multiple dispatch, depending on the actual argument combination. The last benchmark "Random Value Dispatch" is the scenario for value dispatch and is based upon the example from Section 5.11. A method gets invoked a specified number of times with two random argument values, resulting in four different target methods, depending on the concrete values. The containing class is a concrete class without subclasses. Furthermore, the first two benchmarks are varied by using two different method families. Method "m2" represents a method with just two parameters, while method "m5" represents a method with five parameters. Figure 6.6 shows the normalized runtime overhead of `Dynamate`'s integration against the original Multijava implementation, while Table 6.6 shows the absolute benchmark run times. It is immediately clear that `Dynamate`'s reimplementation is much inferior to the original Multijava. As explained in Section 2.3.1, Multijava uses instance-of checks in the default method to examine the concrete types and dispatch to more appropriate methods. For value dispatch, Multijava does the same with equality tests. These simple mechanisms produce only minimal overhead, compared to `Dynamate`'s heavier approach. Both mechanisms use `invokevirtual` calls that the JVM can optimize effectively, e.g. by inlining the concrete methods into the default method. Overall, `Dynamate` is not suitable for Multijava, because its mechanisms are too simple to profit from `invokedynamic`.

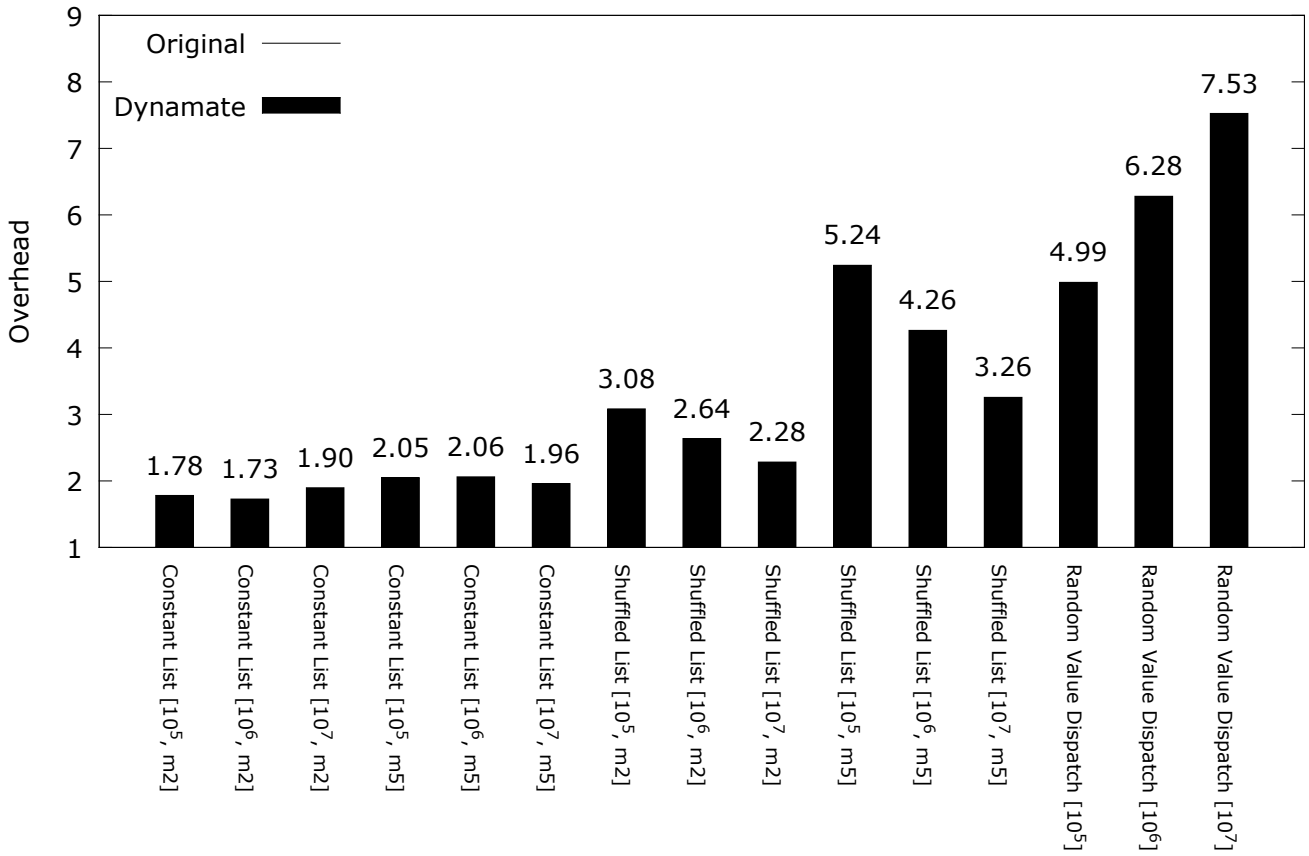


**Figure 6.6:** Normalized performance results of the `Dynamate` implementation against the original Multi-java implementation

### JCop

JCop is evaluated by comparing the original non `invokedynamic` implementation to the `Dynamate` reimplementation using a set of self written benchmarks. All benchmarks represent the example from Section 2.3.2 by performing a statically typed method dispatch with different sets of layers, resulting in layered dispatch. The Benchmark "Direct" depicts method dispatch without active layers (no with/without statements). Benchmark "Constant" features method dispatch with three active layers, which are activated prior to the first dispatch and are active during all invocations. "Sequential" represents the sequential activation of up to five layers. After each layer addition, the method is invoked the specified times. Finally, "Alternating" resembles "Sequential", such that the layers are activated sequentially from zero to five, but additionally, an outer loop resets the layer composition each time the last loop finishes. The total number of method invocations, ranging from $10^3$ to $10^5$, stays always equal across all four benchmark scenarios.

Figure 6.7 shows the benchmark results. `Dynamate` is generally superior in all scenarios. In the "Direct" case, JCop still has to forward the call to the base method, even when no layer is active. `Dynamate` in contrast, can guard and shortcut such calls, resulting in better performance. The "Constant" benchmarks shows that `Dynamate` performs better, compared to JCop, for an increasing amount of invocations. The

| Benchmark | Original [ms] | Dynamate [ms] |
|---|---|---|
| Constant List [100000, m2] | 87 | 155 |
| Constant List [1000000, m2] | 99 | 171 |
| Constant List [10000000, m2] | 232 | 440 |
| Constant List [100000, m5] | 82 | 168 |
| Constant List [1000000, m5] | 98 | 202 |
| Constant List [10000000, m5] | 277 | 543 |
| Shuffled List [100000, m2] | 97 | 299 |
| Shuffled List [1000000, m2] | 185 | 488 |
| Shuffled List [10000000, m2] | 1009 | 2304 |
| Shuffled List [100000, m5] | 99 | 519 |
| Shuffled List [1000000, m5] | 186 | 793 |
| Shuffled List [10000000, m5] | 1072 | 3490 |
| Random Value Dispatch [100000] | 68 | 339 |
| Random Value Dispatch [1000000] | 82 | 515 |
| Random Value Dispatch [10000000] | 293 | 2205 |

**Table 6.4:** Absolute benchmark runtimes of the `Dynamate` implementation and original Multijava implementation

| Benchmark | Original [ms] | Dynamate [ms] |
|---|---|---|
| Direct, No Layers [10000] | 164 | 123 |
| Direct, No Layers [100000] | 216 | 131 |
| Constant, 3 Layers [10000] | 221 | 237 |
| Constant, 3 Layers [100000] | 1133 | 367 |
| Sequential, 1-5 Layers [10000] | 1010 | 156 |
| Sequential, 1-5 Layers [100000] | 15412 | 201 |
| Alternating, 1-5 Layers [1000] | 285 | 203 |
| Alternating, 1-5 Layers [10000] | 1233 | 286 |

**Table 6.5:** Absolute benchmark runtimes of the `Dynamate` implementation and original JCop implementation

"Sequential" and "Alternating" benchmarks show one of the main advantages of `Dynamate`: its dynamic. JCop clearly struggles with the five layer composition changes, while `Dynamate` can reuse already resolved methods and only has to resolve the latest added layer method. In the "Alternating" benchmark, `Dynamate` can profit from its caching mechanism. Targets for specific layered compositions get cached and can be reused without much overhead. Table 6.6 shows the absolute benchmark runtimes. Overall, this results show that `Dynamate` is quite suited for a JCop integration, offering better performance in both static and dynamic scenarios.

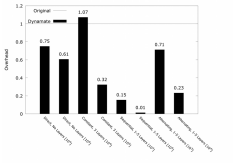

**Figure 6.7:** Normalized performance results of the `Dynamate` implementation against the original JCop implementation

## JastAdd

JastAddJ is a Java compiler, implemented with JastAdd. It can be extended with own modules to support other language related tasks, besides compiling, e.g. static analysis. Its `org.jastadd.jastaddj.JavaCompiler` class can be used to compile most Java programs to standard Java bytecode, executable on every JVM. To evaluate the performance of `Dynamate`'s JastAdd implementation, it is compared to the original JastAdd implementation by using the JastAddJ compiler to compile some external Java programs, i.e., JastAddJ is invoked once with the original library and once with the library integrated with `Dynamate`. Each benchmark features a different Java program. The first two benchmarks represent the compilation of one of AspectJ's[3] libraries and of the Apache commons library[4]. The other benchmarks omit JastAddJ and introduce their own simple abstract language, which is meant, once again, to explain the overall performance results. Listing 6.4 shows the formal grammar of the language and an inherited grammar attribute v. The grammar has one Root A, which contains one node of type S. S is abstract and can be either U, or V. Nodes of type U are just intermediate nodes in the AST. A node of type V eventually finalizes the AST.

```
1 A ::= S;
2 abstract S;
3 U : S ::= S;
4 V : S;
5
6 aspect Analysis {
7   inh int S.v();
8   eq A.getS().v() = 0;
9 }
```

**Listing 6.4:** Degenerated Tree Grammar

Therefore, this grammar allows concrete trees consisting of one A node at the head, one V node at the tail and an arbitrary number of intermediate U nodes, resulting in a linked-list-like structure: a degenerated tree. The concrete tree has a varying depth, depending on the number of U nodes. Each of the final benchmark consists of a constructed concrete tree of a specified depth and the repeated invocation of its attribute v, resulting in method dispatch along the AST, as explained in Chapter 2.

Figure 6.8 shows the normalized runtime overhead of `Dynamate`'s integration against the original JastAdd implementation. The results of the benchmarks show that `Dynamate`'s `invokedynamic` integration is clearly inferior to the original implementation, at least for real world examples. On average, its ten times slower than the original implementation. The reason is, the depth of concrete trees of grammar languages is usually fairly low. A typical Java program (as represented by the AspectJ and Apache commons libraries) represents a very broad, but small tree, generally with a depth of well under ten nodes. Such trees do not profit, in terms of performance, from using `invokedynamic` for their method dispatch, because the resolution needs, thus, at most ten method calls, for any given inherited call, to invoke the desired target. Furthermore, these method calls are statically typed `invokevirtual` calls, which can be highly optimized by the JVM and are therefore faster, than any `invokedynamic` call could ever be. However, the other benchmarks in Figure 6.9 show that `Dynamate` can offer superior performanc in some exotic scenarios. The "degenerated tree" benchmarks depict that the original JastAdd implementation struggles with increasing tree depth, while `Dynamate` offers constant performance. Table 6.6 shows the absolute benchmark run times. For languages with such exotic grammar trees, `Dynamate` or at least `invokedynamic` calls should definitely be considered.

[3] http://www.eclipse.org/aspectj
[4] http://commons.apache.org

| Benchmark | Original [ms] | Dynamate [ms] |
|---|---|---|
| AspectJ Lib Compiling | 1476 | 11733 |
| Apache commons Lib Compiling | 3520 | 44203 |
| Degenerated Tree [depth=5] | 86 | 91 |
| Degenerated Tree [depth=50] | 186 | 99 |
| Degenerated Tree [depth=100] | 319 | 106 |
| Degenerated Tree [depth=500] | 1937 | 149 |

**Table 6.6:** Absolute benchmark runtimes of the `Dynamate` implementation and original JastAdd implementation

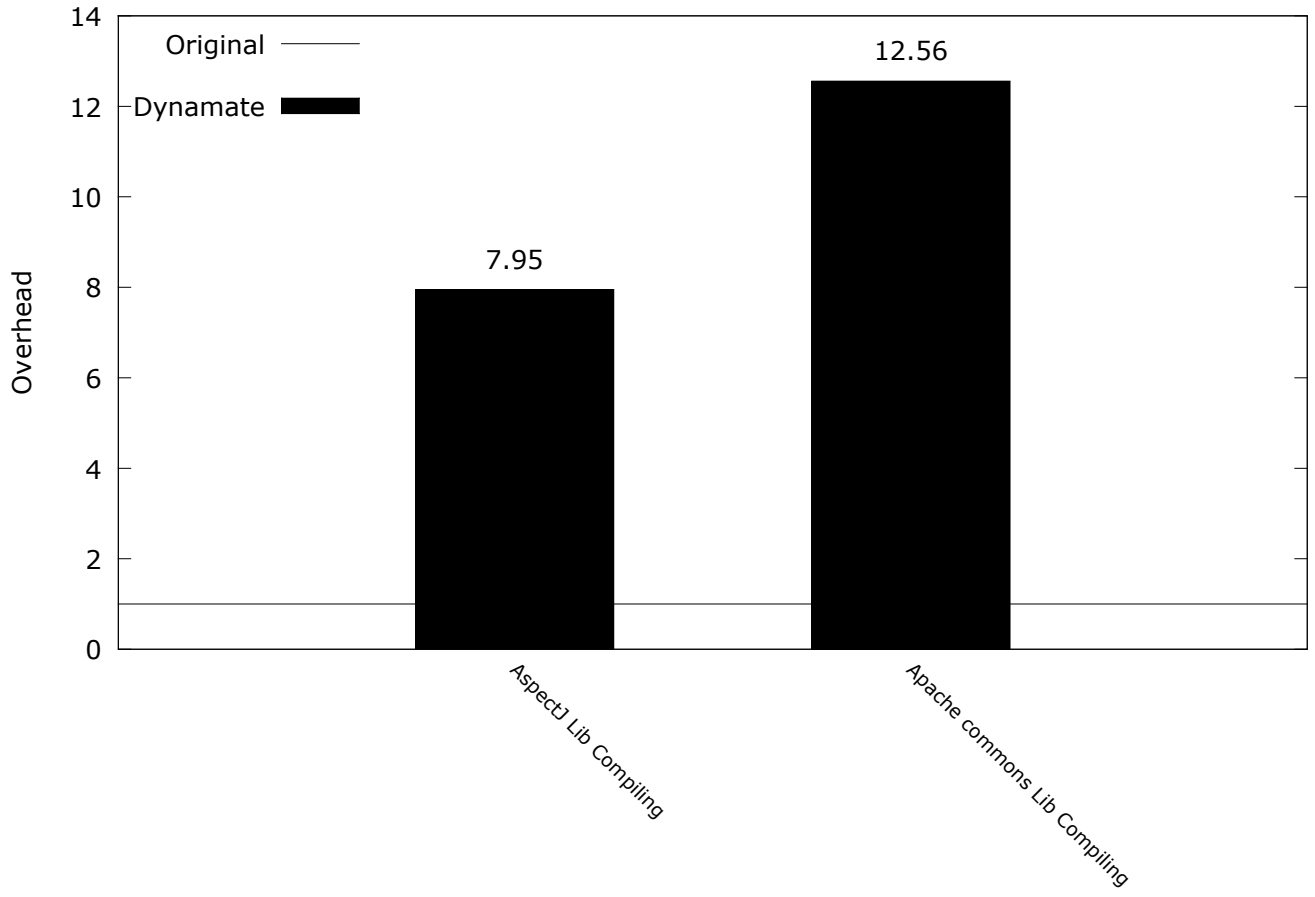


**Figure 6.8:** Normalized performance results of the `Dynamate` implementation against the original JastAdd implementation performing the library compiling benchmarks

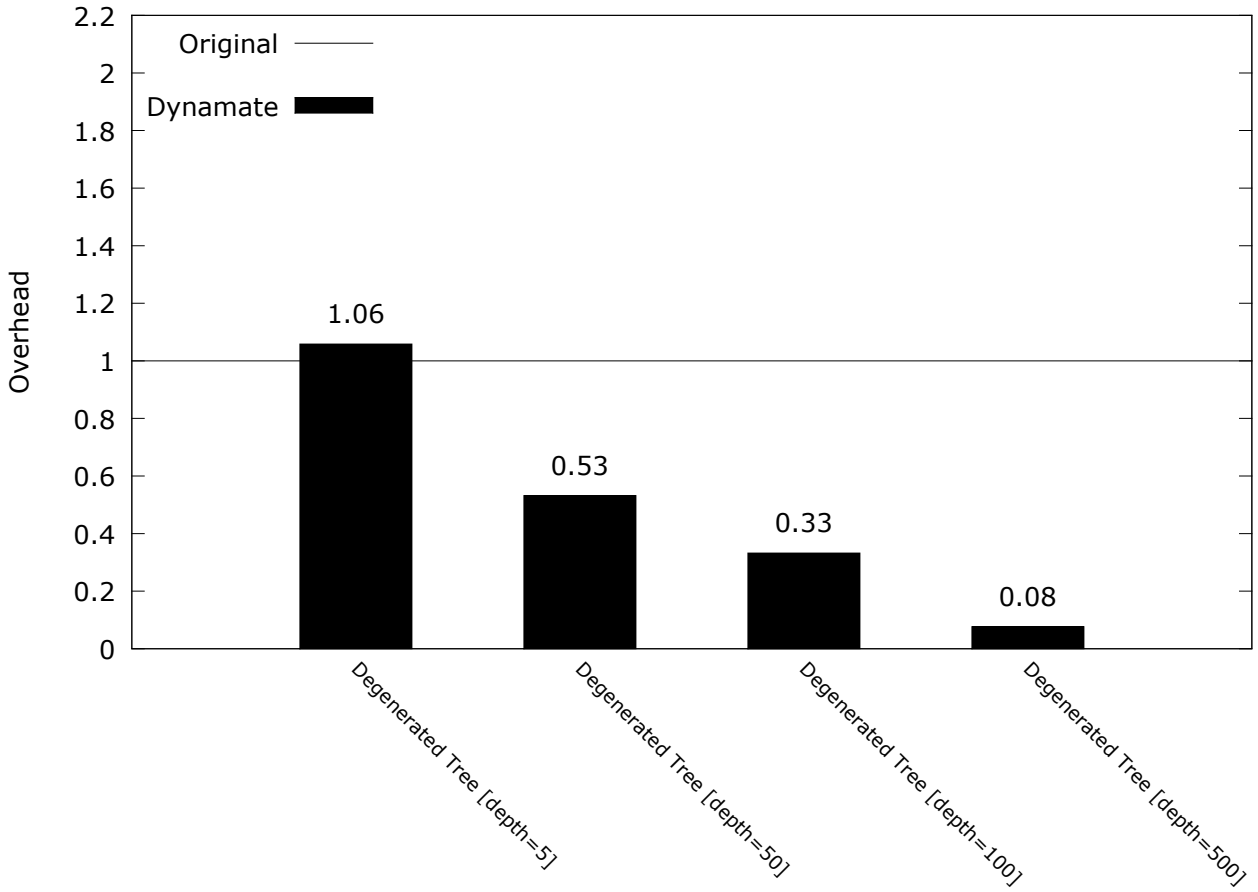


**Figure 6.9:** Normalized performance results of the `Dynamate` implementation against the original JastAdd implementation performing the degenerated tree benchmarks

## 6.2 Usability

`Dynamate` can increase the usability of both clients already using `invokedynamic`, and clients so far using traditional techniques to enable their method dispatch. The following sections address shared advantages and those specific to the two mentioned client types.

### Maintainability

Integrating a framework handling `invokedynamic` mechanisms lowers the required maintainability on one's client code. Conceptual improvements to the applied `invokedynamic` techniques can be integrated centrally into the framework by `invokedynamic` specialists, instead of forcing every developer to update and maintain his implementation individually.

### Integration Effort

The effort to use `invokedynamic` for one's language or tool is lowered significantly. Writing runnable code that integrates `Dynamate` can be done with very low effort, which is especially beneficial for rapid

prototyping and implementation studies of new dispatch concepts. The existence of guard and caching mechanisms enable the performance evaluation of new dispatch semantics in real world scenarios.

### Abstraction

Learning `invokedynamic` and Java bytecode comprehensively is no longer a necessity, as `Dynamate` abstracts from this means by using common object-oriented framework practises. The complexity of `invokedynamic` is hidden behind simplified interfaces, in which cohesive concepts are grouped together, unlike `invokedynamic`'s native library, which is instead procedure oriented.

### Inversion of Control

Because `Dynamate` is a framework, its main property is the inversion of control. Developers implement and instantiate the supplied extension points with their own semantics and set up their desired configuration options in the bootstrap process. From then on, `Dynamate` takes control and ensures that all user defined policies and strategies are called during certain steps of the method dispatch.

### Composability

`Dynamate`'s well-defined interfaces allow full usage of object-orientation. Guard strategies, method selection strategies, object graph traversing strategies and all adaptation policies can be implemented for each semantic responsibility individually and combined arbitrarily to facilitate advanced dispatch semantics. Each call site can be bound to a specific type of dispatch by simply instantiating and composing the desired user defined policies.

### Transparency and Tool Support

Clients like JCop and Multijava modify the bytecode of user classes directly during their own compiling process, in order to add their client specific dispatch logic. Generally, this is an acceptable non intruding approach, transparent to end users. However, tools, e.g. debuggers and analysers, have more problems with this approach, due to the discrepancy between source- and bytecode. Custom mappings between both code types would be needed to fully support existing tools. Clients like JastAdd are even problematic for the user, because their compilers not only modify bytecode, but actual source code. This generated source code contains both user defined code and helper code, burdening the user with polluted interfaces. These problems can be handled better by using `invokedynamic` and `Dynamate`, thus segregating dispatch logic from user code.

# 7 Related Work

This chapter briefly discusses two existing projects using `invokedynamic`.

## 7.1 invokebinder

invokebinder[1] is a library for constructing method handles in a more intuitive and readable way, compared to Java's own library. It is extensively used by JRuby. All the usual method handle operations are modelled as transformation classes (e.g. Fold, Filter, Spread transformations) that can be chained to execute the desired method handle adaptation. Due to this model, the actual transformations do not happen immediately, as it's the case in Java's library. Instead, they are applied at the end, when a special method is called. This can potentially save performance if some transformations are redundant (e.g. two drop transformations dropping single arguments could be combined to a single transformation dropping both arguments). Listing 7.1 shows an example from invokebinder's website, where the transformations are applied prior to the execution of the `invoke` method.

```
1 MethodHandle mh = Binder
2     .from(String.class, String.class, String.class) // String w(String, String)
3     .drop(1, String.class) // String x(String)
4     .insert(0, 'hello') // String y(String, String)
5     .cast(String.class, CharSequence.class, Object.class) // String z(CharSequence, Object)
6     .invoke(someTargetHandle);
```

**Listing 7.1:** invokebinder Example

## 7.2 Dynalink

Dynalink[2] is a relatively young framework using `invokedynamic` for a metaobject protocol for cross-language invocations, i.e., it provides the infrastructure to call arbitrary methods on objects from different languages and receive calls from other languages on own objects, as long as all languages run on the JVM. Using a special identifier pattern, denoting the object's type and the requested method, for the call site's identifier, Dynalink searches the classpath for the participating argument types and invokes the desired method. A call "dyn:getProp:temperature" with the call type (Object)Number and one (receiver) argument would result in the invocation of the field "temperature", while the call "dyn:getTemperature" would invoke the method "getTemperature" on the passed argument. This uniform identifier syntax allows the invocation of fields, methods and constructors on objects of different languages. Dynalink has guard and switchpoint mechanisms to protect call sites against changes, using user supplied policies.
The biggest difference between `Dynamate` and Dynalink is a conceptual one. `Dynamate` wants to support developers writing their own dispatch semantics by providing well-defined interfaces and coordinating the dispatch. Dynalink in constrast, brokers primarily between dispatches to other languages, i.e., developers still need to implement their own dispatch semantics and implement an endpoint for Dynalink's brokering.

[1] https://github.com/headius/invokebinder
[2] https://github.com/szegedi/dynalink

# 8 Conclusion and Outlook

The following chapter's first section summarises and concludes the work on Dynamate and this thesis, while the second section looks into the future of invokedynamic and Dynamate by outlining possible optimizations and extensions.

## 8.1 Conclusion

Chapter 2 introduced the examined clients with their requirements on method dispatch and Chapter 3 introduced and explained the new invokedynamic instruction, which is the core mechanism in Dynamate. Chapter 4 presented the concept of Dynamate by showing shared and variable points of different types of method dispatch and by designing a common framework to bridge these client specific semantics. Chapter 6 evaluated the performance of Dynamate, compared to the examined client's own existing implementations by means of measuring runtime on a set of common and specific benchmarks. Additionally, it discussed Dynamate's usability, primarily in terms of integration effort and abstraction. Finally, Chapter 7 introduced briefly two existing libraries in the field of invokedynamic.
Overall, this thesis has showed that a common framework for method dispatch using invokedynamic is possible and generally preferable to individual solutions. Using techniques for call site guarding and caching, a performance, similar to existing client invokedynamic implementations, can be achieved, while gaining the advantage to be intergratable with different clients, by providing well-defined interfaces and configuration options. A high value is set on lessening the burden on developers by simplifying and abstracting invokedynamic and conducting the control flow.

## 8.2 Outlook

A common framework may be too late for the examined language clients (JRuby, Jython, Groovy), considering that invokedynamic implementations exist and developers are already familiar with them. However, clients not yet using invokedynamic and new clients can definitely profit from Dynamate, especially developers, unfamiliar with invokedynamic or Java bytecode, who want to prototype new dispatch semantics more easily and rapidly.
Dynamate will be open sourced and hosted on GitHub[1] eventually to allow and facilitate collaboration and development.
The following final section describes further possible optimizations and extensions, that would have gone beyond the scope of this thesis, but could be candidates for implementation in the future.

### 8.2.1 Possible Optimizations and Extensions

Dynamate could make use of a profiler to find an optimal guard and caching configuration for each client. Such a profiler could be enableable for any call site and profile its invocations under extensive load. This would produce optimal configurations for each call site, individually.
Furthermore, Dynamate could try to support different types of specifying the desired method dispatch. As of now, only imperative specifications, using user implemented classes, are supported. Dynamate could integrate other means of specifying the dispatch, e.g. using a declarative language and Java annotations, to let Dynamate discover the required components (e.g. classes and methods), wire them automatically and use them. This could reduce the effort and facilitate tool supported analysis.

[1] http://github.org

## Bibliography


[1] Malte Appeltauer, Michael Haupt, and Robert Hirschfeld. Layered method dispatch with invokedynamic: an implementation study. In *Proceedings of the 2nd International Workshop on Context-Oriented Programming*, COP '10, pages 4:1–4:6, New York, NY, USA, 2010. ACM.

[2] Curtis Clifton, Todd Millstein, Gary T. Leavens, and Craig Chambers. MultiJava: Design rationale, compiler implementation, and applications. Technical Report 04-01b, Iowa State University, Dept. of Computer Science, December 2004. Appears in ACM TOPLAS, vol 28, number 3, May 2006.

[3] David Flanagan and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly, first edition, 2008.

[4] Görel Hedin. An introductory tutorial on jastadd attribute grammars. In *Proceedings of the 3rd international summer school conference on Generative and transformational techniques in software engineering III*, GTTSE'09, pages 166–200, Berlin, Heidelberg, 2011. Springer-Verlag.

[5] Mark Lutz. *Learning Python*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2 edition, 2003.

[6] John R. Rose. Bytecodes meet combinators: invokedynamic on the jvm. In *Proceedings of the Third Workshop on Virtual Machines and Intermediate Languages*, VMIL '09, pages 2:1–2:11, New York, NY, USA, 2009. ACM.