*Send Orders for Reprints to reprints@benthamscience.net*

38 *The Open Software Engineering Journal*, 2013, *7*, 38-52

Open Access

# Reflection Support: Java Reflection Made Easy

Zalia Shams* and Stephen H. Edwards

*Virginia Tech, Department of Computer Science, 114 McBryde Hall (0106), Blacksburg, VA 24061, USA*

**Abstract:** Large software projects often require the ability to load and manage new code assets that were not originally available during system compilation. Moreover, testing tools need to inspect and run code components regardless of their structures. Reflection in Java supports these tasks by providing programmers with facilities to dynamically create objects, invoke methods, access fields, and perform code introspection at runtime. These capabilities come at the cost of reduced readability and writeability, since code written using Java's reflection classes is clunky, bulky and unintuitive. Common tasks such as object creation, method invocation, and field manipulation need to be decomposed into multiple steps that require try-catch blocks to guard against checked exceptions. Type casts and explicit use of class types as parameters make development and maintenance of code difficult, time consuming and error prone. In this paper, we discuss the difficulties of using reflection in Java. We also present an open-source library called ReflectionSupport that addresses these problems and makes reflection in Java easier to use. ReflectionSupport provides static helper methods that offer the same reflective capabilities but that encapsulate the overhead of coding with reflection. This paper focuses on improving the usability of Java reflection by presenting an API that allows programmers to obtain the benefits of reflection without the hassle.

**Keywords:** Open-source Software, Software Testing, Java, Reflection, Library, API.

## 1. INTRODUCTION

Automated software testing tools inspect and evaluate programs without requiring compile-time access to the program's structure. Similarly, many software development platforms (e.g., Google Web Toolkit) manage code components in a fashion that reduces compile time dependencies between them. Moreover, educational software tools manipulate and inspect student programs regardless of their structures for various purposes, such as providing feedback, evaluating correctness, and measuring performance. Therefore, there is a need for programs to be able to inspect and execute other software without requiring any compile-time structural dependency on that code.

*Reflection* is a useful and widely adopted technique for writing code without compile-time dependencies in many programming languages, including Java. Bobrow *et al.* defines reflection this way [1]:

"Reflection is the ability of a program to manipulate as data something representing the state of the program during its own execution. There are two aspects of such manipulation: introspection and intercession. Introspection is the ability for a program to observe and therefore reason about its own state. Intercession is the ability for a program to modify its own execution state or alter its own interpretation or meaning. Both aspects require a mechanism for encoding execution state as data; providing such an encoding is called reification."

*Address correspondence to these authors at the Virginia Tech CS Dept., 114 McBryde Hall (0106), Blacksburg, VA 24061, USA;
Tel: 1-540-449-2826; Fax: 1-540-231-6075;
E-mails: zalia18@cs.vt.edu; zaliashams@gmail.com, edwards@cs.vt.edu

Reflection is a powerful tool for increasing flexibility. It provides mechanisms for assembling software pieces at runtime, profiling applications, and supporting plugin-driven architectures. JavaBeans, for example, uses reflection to manipulate software components via builder tools [2].

In Java, however, the benefits of reflection are overshadowed by the complex, unintuitive and verbose nature of the library classes that Java provides for this purpose. Java's reflection API has three main drawbacks:

1. Common tasks such as object creation, method invocation, and field manipulation require multiple steps to complete.
2. Because any of the steps may fail during execution, explicit try-catch blocks are required to handle exceptions that may be generated.
3. If code that is invoked through reflection throws an exception, Java's reflection API will wrap that exception inside a new **InvocationTargetException**, making it more difficult to handle with regular user-provided exception handlers.

Taken together, these issues cause code written using Java reflection to be more difficult to write, read, and maintain.

This paper investigates the difficulties of the Java reflection API with a focus on its usability. We present an open-source library, Reflection Support, which enable programmers to write code using intuitive API methods in a straightforward way. Reflection Support provides methods similar to those of the Java reflection library for object creation,

1874-107X/13 

```
1  import java.util.Date;
2  public class ScoreBoard
3  {
4      private String team1, team2;
5      private Date startTime, endTime;
6      private long elapsedTime, totalTime;
7      private int team1Score, team2Score;
8
9      // Constructor: sets value of team1, team2, totalTime
10     public ScoreBoard(String team1, String team2) { ... }
11
12     // Setter/getter methods
13     public void setTeam1Score(int score) { ... }
14
15     // Other setter and getter methods omitted for brevity
16     ...
17
18     // Sets startTime and initializes elapsed time
19     public void startGame() { ... }
20
21     // Sets endTime and updates elapsed time
22     public void stopGame() { ... }
23
24     // Increases team1Score by score
25     public void increaseScoreTeam1(int score) { ... }
26
27     // Increases team2Score by score
28     public void increaseScoreTeam2(int score) { ... }
29
30     // Calculates how much time remains until the end of the game
31     public String remainingTime() { ... }
32
33     // Calculates the winner of the game from the scores
34     public String getWinner() { ... }
35 }
```

**Fig. (1)**. The ScoreBoard class.

method invocation and field manipulation. These API methods internally handle all required sub-steps for each task, check for exceptions, cast types appropriately, and expose any exceptions caught during execution of the underlying code by unwrapping **InvocationTargetException** objects where necessary. By careful use of overloading and variable parameter lists, these methods can be invoked in flexible ways under different conditions. Careful use of generics allows the methods to be type-safe and prevents the need for explicit type casting in many situations. Hence, a programmer does not need to write extra steps, add try-catch blocks, cast types, or unwrap **InvocationTargetException** objects when using this library.

This paper is organized as follows. Section 2 gives an overview of reflection and discusses the difficulties that arise when using the Java reflection API. Section 3 summarizes existing approaches to simplifying Java's reflection API. Our solution to these issues in the form of the Reflection Support API is presented in Section 4. Reflection Support is evaluated in comparison with the Java reflection library and other alternative approaches in Section 5. Finally, we conclude in Section 6 with a summary of the strengths and weaknesses of Reflection Support in comparison to other existing approaches.

## 2. ISSUES WITH JAVA'S REFLECTION API

Reflection is the capability of a computer program to observe and modify its own structure and behavior. Maes et al. [3] define reflection as the activity performed by a program when doing computations about itself. Reflection in Java allows the developer to perform runtime actions based on the descriptions of the objects involved: one can create objects given their class names, call methods by their name, and access field values given their name [4]. The Java core reflection library [4, 5] provides the programmer with many reflective features. These features add flexibility by providing for runtime type information (RTTI), introspection, method invocation, and dynamic instantiation.

### 2.1. An Example

To show how reflection might be used, Fig. (**1**) presents the **ScoreBoard** class as an example that one might wish to

```
1  public class MainScoreBoard
2  {
3      public static void main(String args[])
4      {
5          ScoreBoard sb = new ScoreBoard("VT", "GT");
6          sb.setTeam1Score(20);
7          String remaining = sb.remainingTime();
8      }
9  }
```

**Fig. (2).** Manipulating a ScoreBoard using direct access.

```
1  import java.lang.reflect.*;
2  public class ReflectiveScoreBoard
3  {
4     public static void main (String args[])
5     {
6        try
7        {
8           Class<?> scBoard = Class.forName("ScoreBoard");
9           Constructor ctr = scBoard.getConstructor(String.class, String.class);
10          Object scb = ctr.newInstance("VT", "GT");
11          Method setT1S = scBoard.getMethod("setTeam1Score", int.class);
12          setT1S.invoke(scb, 20);
13          Method prt = scBoard.getMethod("remainingTime");
14          String rTime = (String)prt.invoke(scb);
15       }
16       catch (ClassNotFoundException e) {
17          System.out.println("Cannot find the class named ScoreBoard.");
18       }
19       catch (InstantiationException e) {
20          System.out.println("Failed to create a ScoreBoard object.");
21       }
22       catch (IllegalAccessException e) {
23          System.out.println("Attempted to invoke a non-public method.");
24       }
25       catch (SecurityException e) {
26          System.out.println("Insufficient security permissions.");
27       }
28       catch (NoSuchMethodException e) {
29          System.out.println("No method with the given name and parameter types was found.");
30       }
31       catch (IllegalArgumentException e) {
32          System.out.println("Actual arguments do not match "
33               + "formal parameters.");
34       }
35       catch (InvocationTargetException e) {
36          System.out.println("A method invoked via reflection threw an exception.");
37       }
38    }
39 }
```

**Fig. (3).** Manipulating a ScoreBoard using reflection.

manipulate via reflection. This rudimentary class represents a scoreboard at an athletic event and might be part of a control interface for an electronically controlled scoreboard. The **ScoreBoard** class has several fields and methods. We will focus on the constructor, the field **team1Score,** the setter method **setTeam1Score()**, and the **remainingTime()** method, since together these features represent a simplified example of the most common features that classes provide. Manipulation of such a **ScoreBoard** object using a GUI may be more realistic, but for simplicity no GUI is presented here. Instead, the **ScoreBoard** class is just an example that will allow discussion of how class features can be exercised with and without reflection.

The **MainScoreBoard** class in Fig. (**2**) shows how one can use the **ScoreBoard** class by directly accessing its features (i.e., without reflection). Its **main()** method creates a **ScoreBoard** instance and then invokes **setTeam1Score()** and **remainingTime()**.

Fig. (**3**) depicts an equivalent class called **ReflectiveScoreBoard** that performs exactly the same tasks, but that uses reflection instead of directly accessing the features of

the **ScoreBoard** class. **ReflectiveScoreBoard** can be compiled without access to the source code or the compiled byte code for the **ScoreBoard** class. It resolves creation of and method invocation on **ScoreBoard** objects at execution time. However, the price of this added flexibility is clumsy code—the three lines of code in **MainScoreBoard** are transformed into about ten times as many when using Java's reflection API. Although the multiple **catch** clauses in the **try-catch** block could be combined to reduce the code size, they are shown separately here to indicate the potential for handling different problems with unique responses. With all of the exception handling, type casts, and multi-step decomposition of tasks, code written using reflection is less readable as well as less writable.

### 2.2. Three Key Problems

Code written using Java's reflection classes is verbose, clunky, and bulky because:

1. Common tasks, such as creating an object, invoking a method, setting a field, etc., take multiple steps, any one of which can fail.
2. Because the methods used to complete these steps often throw checked exceptions if they fail, explicit **try-catch** blocks are required.
3. If the underlying code invoked by reflection happens to throw an exception of its own, that exception is wrapped inside an **InvocationTargetException** before propagating out of the reflection API. When an exception is wrapped this way, it can no longer be caught directly by the handlers one would write if reflection were not used.

As an example of the first issue, consider the call to **setTeam1Score()** on line 6 of Fig. (**2**). In the reflective version shown in Fig. (**3**), this single action corresponds to two steps on lines 12–14, which involve looking up the **Method** object representing the **setTeam1Score()** method and then calling **invoke()** with the actual parameters. In other cases where a non-public method is invoked, it also is necessary to set the accessibility of the **Method** object.

On line 12 of Fig. (**3**), **getMethod()** may fail to find a method matching the given name and parameter profile, or the method may not be accessible (not declared public), since **getMethod()** only supports the retrieval of public methods. For non-public methods, **getDeclaredMethod()** must be used instead. However, **getDeclaredMethod()** does not automatically retrieve methods declared in superclasses, so the programmer must know in which specific class a method has been declared in order to retrieve it.

Because method mismatches are reported using checked exceptions, a **try-catch** block is required around **getMethod()**. Even if **getMethod()** succeeds and returns a valid **Method** object, the actual invocation via **setT1S.invoke()** could potentially throw an **IllegalArgumentException** if the actual parameter(s) did not match the formal parameter type(s) declared for the method. The call to **invoke()** returns the value produced by the method, if any, but without any type information. As a result, the programmer must manually cast the returned object to the proper type for methods that return a value (this is not necessary for **setTeam1Score()**, since it does not return a value).

Finally, the method being invoked might throw an exception while executing, which would result in an **InvocationTargetException**. Since **InvocationTargetException** is a checked exception, another **try-catch** handler is mandatory, even for methods like **setTeam1Score()** that have no declared **throws** clauses.

Because of these issues, writing code using Java's reflection API requires more work than necessary, and providing appropriate handlers for all errors that may arise in general can be tedious. Further, the exceptional messages that are produced when things go wrong can be difficult to interpret and often lack important information. For example, the call to **getMethod()** will throw a **NoSuchMethodException** if no matching method is found. However, this same exception—with the same message—is thrown irrespective of the reason for the mismatch: an incorrect method name, the wrong number of parameters, a mismatch in the type of one or more formal parameters, or if the method is not public. Terse or cryptic exception messages make correcting reflection-based code more difficult. Taken together, all of these issues often overshadow the benefits of reflection in Java.

## 3. RELATED WORK

Others have investigated the difficulties associated with Java's reflection interface. However, efforts for providing a simpler API often are pieces of larger software packages rather than products that stand on their own. Hence, such APIs are not as well known or as readily available for other programmers to use.

### 3.1. Java Beans and Reflection

Java Beans provides API methods that use reflection for object creation and method invocation. Theses capabilities are provided by the **java.beans.Expression** and **java.beans.Statement** classes. These classes are used internally within the Java Beans framework to manipulate software components via builder tools, but are also available for use in client programs.

Both classes allow one to invoke methods using reflection, where object construction is achieved by using the special name "new" as the method to invoke. Generally, an **Expression** object is used to invoke a method (or constructor) that returns a value, whereas a **Statement** object is used to call void methods. An **Expression** or a **Statement** is instantiated with three parameters: 1) a class or object that represents the receiver of the call, 2) a method name to be called, and 3) a list of actual arguments to use when invoking the specified method. Invocation of the desired method occurs when the **getValue()** method (on an **Expression**) or the **execute()** method (on a **Statement**) is invoked. The **getValue()** method returns the value produced by the method being invoked, while **execute()** ignores any such value. If **getValue()** or **execute()** encounter any errors internally, they throw an **Exception** that wraps the actual exception that occurred. Therefore, the tasks of object creation or method invocation require exactly two steps: creating an instance of **Expression** or **Statement** representing the desired action, and then call-

```
 1  import java.beans.Expression;
 2  import java.beans.Statement;
 3  public class ExpressionScoreBoard
 4  {
 5      public static void main(String args[])
 6      {
 7          try {
 8              Expression crtExpr = new Expression(ScoreBoard.class, "new",
10                 new Object[] {"VT", "GT"});
11              ScoreBoard sb = (ScoreBoard)crtExpr.getValue();
12              Statement st1ScoreStmt = new Statement(sb, "setTeam1Score", new Object[] {20});
14              st1ScoreStmt.execute();
15              Expression rTimeExpr = new Expression(sb, "remainingTime", null);
17              String rTime = (String)rTimeExpr.getValue();
18          } catch (Exception e) {
19              e.printStackTrace();
20          }
21      }
22  }
```

**Fig. (4)**. Manipulating a ScoreBoard using Java.Beans.

ing **getValue()** or **execute()** to perform that action. All internal subtasks such as finding a matching method, or handling any exceptions that occur while invoking the method, are handled internally by these classes.

Fig. (**4**) shows an example of using **Expression** and **Statement** to manipulate a **ScoreBoard** object. Each of the three actions from Fig. (**2**) is repeated in Fig. (**4**). Note that these Java Beans classes pre-date Java 1.5, and so they do not take advantage of variable arguments or generics.

One feature that is somewhat unique in the Java Beans reflection classes is that they search for the most specific matching method for the given actual parameters, where the alternative libraries discussed in this section instead look up a method using a signature that must be specified exactly. Because the API methods automatically convert actual parameters to formal parameters, executed code is usually less error prone. Moreover, the API methods unwrap any **InvocationTargetException** that may occur and re-throw the underlying exception if possible.

However, there are several limitations to these classes as well. The API does not support manipulation of fields. Explicit type casts for the method return values are usually required. Because the **getValue()** and **execute()** methods are declared to throw **Exception** objects, explicit **try-catch** blocks are required even when the underlying code being invoked cannot throw checked exceptions. Finally, each task requires two steps, although they can be combined into a single statement if desired. Overall, the Java Beans approach offers an improvement over Java's native reflection API when a programmer does not need to access fields directly, although it still has some limitations.

### 3.2. Fluent Interfaces and Reflection

Some researchers have used *fluent interfaces* [Sch07] to simplify reflection. Fluent interfaces is a technique for constructing a tiny domain-specific language formed by method call chaining and well-chosen method names. Fluent interfaces offer the opportunity to enhance readability and improve clarity when method names and chaining patterns are chosen carefully. Such an interface offers an API to maintain the instruction context for a series of method calls [1]. Chaining setter methods and factory methods to create and initialize objects is one area where the fluent interface approach has been applied, with an underlying implementation using reflection. However, a fluent interface can be designed to provide an easy way to call any method, especially those that are called often.

Stephan Schmidt in his blog [6] described a basic fluent interface for creating Java objects, powered by a reflection-driven proxy. The technique he described only focuses on object creation tasks, with the aim of replacing constructors that contain a laundry list of initialization parameters with a more readable alternative. Creating a **ScoreBoard** object using Schmidt's approach would look like this:

```
ScoreBoard scoreboard = ScoreBoard.with()
        .HomeTeam("VT").VisitorTeam("GT")
        .create();
```

Here, the with() method is a new static method that has been added to the ScoreBoard class. In effect, this static method generates a factory object that is responsible for creating the new ScoreBoard instance. This factory object supports a number of methods named after the properties of the **ScoreBoard** that can be used to set the initial values for the corresponding fields. Each of these methods modifies and then returns the factory object so that they can be chained together. Finally, the create() method on the factory object returns the newly created, properly initialized ScoreBoard instance.

This single line of code corresponds to lines 8-11 in Fig. (**3**), along with a number of associated exception handlers. The advantage of this approach is the simpler, cleaner inter-

```
1  import static org.fest.reflect.core.Reflection.*;
2  public class FestReflectiveScoreBoard
3  {
4      public static void main (String args[])
5      {
6          ScoreBoard scb = constructor().withParameterTypes(String.class, String.class)
7              .in(ScoreBoard.class).newInstance("VT", "GT");
8          method("setTeam1Score").withParameterTypes(int.class).in(scb).invoke(20);
9          String rtime = method("remainingTime")
10             .withReturnType(String.class).in(scb).invoke();
11     }
12 }
```

**Fig. (5).** Manipulating a ScoreBoard using FEST-Reflect.

face for object creation. In Schmidt's implementation, each of the property-oriented methods on the factory object is shorthand for the reflection-based invocation of a setter method on the underlying **ScoreBoard** object. However, the reflective calls are completely encapsulated and protected so that none of the complexities of interacting directly with Java's reflection API are visible.

While this approach to constructors is not a general substitute for Java's native reflection API, it does address some of the issues raised with the native API. Section 2.2 summarizes three main issues with Java's native reflection API: that common actions often take multiple steps, these steps often require handlers for checked exceptions, and if the underlying operation produces an exception it will be wrapped in an **InvocationTargetException**. Consider these issues with respect to the fluent interface approach.

First, invoking a constructor using the native Java reflection API may involve two or three steps. One must look up the constructor using the **Class** object, set its accessibility if the constructor is not public, and then invoke the constructor. Schmidt's fluent interface design encapsulates all of these actions internally so that the programmer does not have to deal with this issue. Second, with the native API any one of these steps might throw an exception under some conditions. Again, the fluent interface approach to object construction encapsulates handlers for the checked exceptions that might be produced, allowing the programmer to write object creation code without the need to write **try-catch** blocks for such exceptions. However, Schmidt's implementation is intended for use when no such exceptions would occur in practice, so it prints out and swallows any exceptions produced by the native Java reflection API without propagating them and does not provide any additional diagnostic information. Third, if the underlying constructor (or a setter method) throws an exception of its own, it is similarly printed and swallowed without propagation.

One disadvantage of this approach is that the programmer must write explicit code to support it. In this example, the programmer is responsible for providing the static **with()** method, as well as writing a factory class (or at least an interface). Since this must be done for each class one wishes to manipulate, the amount of manual work is worth considering.

While others have proposed similar ideas, Schmidt's main contribution is the idea of using dynamic proxies to provide a common implementation for the factory objects. By careful use of a single generic class and dynamic proxies, Schmidt's version of this fluent object creation technique only requires a programmer to write the interface for the factory object, with the corresponding implementation provided automatically.

Another significant disadvantage of this approach, however, is that client code must have direct access to the class to be manipulated (i.e., the **ScoreBoard**) and to the interface for the factory object. It is not possible to use this strategy directly to create objects of a class that is not available at compile-time. Also, this strategy does not address the more general problem of invoking arbitrary methods on an object using reflection, or of accessing fields.

### 3.3. Fixtures for Easy Software Testing

Fixtures for Easy Software Testing (FEST) is an open-source collection of APIs designed to make writing software tests easier [7]. It includes a reflection module called FEST-Reflect that takes the concept of a fluent interface farther to provide a more comprehensive reflection API.

FEST-Reflect provides static methods for common reflection tasks such as object creation, method invocation, and setting or getting the value of a field. These methods generate intermediate objects that provide fluent interfaces for specifying the parameters needed. Type casting, checked exception handling, and many other requirements of Java's native reflection API are handled internally by these calls, greatly simplifying the task of writing reflection-based code.

Fig. (**5**) shows an example of using FEST-Reflect to manipulate a **ScoreBoard** object. Note that each major task is rendered as a single statement consisting of a sequence of chained method calls. All the checked exceptions related to a particular task are handled within the FEST-Reflect API. Overall, the code is a significant improvement over the version using Java's native reflection API in Fig. (**3**).

FEST-Reflect has several limitations, however. First, to invoke methods, the programmer must know the specific class in which the method is declared—there is no inheritance-based lookup of methods. Second, the programmer

```
 1  import net.xoetrope.xui.helper.ReflectionHelper.*;
 2  public class ReflectionHelperScoreBoard
 3  {
 4      public static void main (String args[])
 5      {
 6          Object scb = constructViaReflection("ScoreBoard", "VT", "GT");
 7          setFieldViaReflection(scb, "team1Score", 20);
 8
 9          // remainingTime() cannot be invoked
10      }
11  }
```

**Fig. (6).** Manipulating a ScoreBoard using XUI's ReflectionHelper API.

must describe the signature exactly as declared by the operation—there is no provision for automatically handling overload resolution or potential argument conversions during method or constructor lookup (only during invocation, as provided by Java's native API). Third, there is no special support for automatic conversions provided by Java on primitive types. Fourth, FEST-Reflect does not pinpoint errors or unwrap any **InvocationTargetExceptions** when errors occur to report the exact cause of failure. This means that exception handlers written in the absence of reflection cannot be used as-is.

### 3.4. Enterprise Reflection APIs

Some open-source software products have developed reflection helper APIs for their own use, but without releasing their reflection APIs as independent libraries. For example, both XStream [8] and XUI [9] include their own reflection helper libraries. XStream is an API for serializing Java objects to and from XML. XUI is a Java and XML Rich Internet Application platform for building smart web applications. The main reason such projects include reflection helper libraries is to reduce and simplify the reflection-driven code that appears elsewhere in the project and to increase efficiency, typically through caching. These libraries include APIs to create objects, handle getter and setter methods of fields, and serialize or deserialize objects. Fig. (**6**) shows an example of creating and manipulating a **ScoreBoard** object using XUI's internal reflection API.

XUI's reflection API provides static methods for object creation and setting or getting field values. All the steps necessary for these tasks are performed within the API methods. However, the XUI library does not provide APIs for invoking methods other than constructors. Therefore, its reflection services are not adequate for many applications that need to use Java reflection. XStream's internal reflection API is similar, in that it only supports object creation and field access.

The biggest limitation to these internal libraries is that they are built for use within a single project, and as such do not systematically address all the features of Java's native reflection API. For example, these libraries often do not support invoking methods that are not setters or getters, and do not support other class introspection features. The features included are biased by the needs of the larger project, and features that are not needed in that project are usually omitted.

### 3.5. Library Support for Performance Improvement

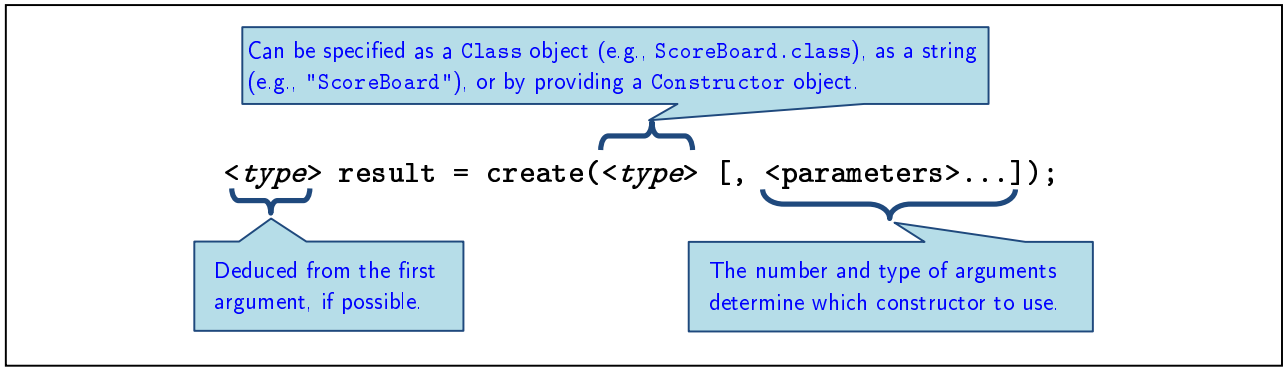
Another major issue with Java reflection is performance. Reflection in Java is much slower than direct calls to methods or constructors. This is due to the added overhead of searching for methods by name and signature, the run-time type checking required for parameters, and the additional steps involved in reflective method calls. Many attempts have been made to speed up reflective access in Java, including strategies for moving reflective actions from run-time to compile-time [4, 10] or load-time [11]. However, introspection must take place at run-time when code looks for unknown services. Smart Reflection [12] has been successful at moving most of the overhead due to dynamic introspection from run-time to compile-time with a different approach to reification. Using the stub idea from the Java RMI interface [13], Smart Reflection can transform a reflective method call into a direct call in intermediate steps of compilation and execution. This allows performance penalties related to dynamic resolution of many details such as type checking to be eliminated by performing checks statically where possible. In this paper, however, we are concerned with the API programmers use to express reflective actions. The performance problems associated with reflection are orthogonal to the issues of how one can improve the usability of the API. In principle, existing approaches to addressing the performance issues of Java reflection are just as applicable to the interface presented here.

## 4. REFLECTIONSUPPORT: A MORE USABLE JAVA REFLECTION LIBRARY

ReflectionSupport is a simplified API for performing reflection in Java. It is intended as an easier-to-use replacement for Java's native reflection API that addresses the issues discussed in Section 3. This library provides static methods for performing the basic tasks of reflection, with names similar to those in the native API. Programmers gain access to these features using a single static import statement. The main purpose of this library is to make it easy to perform object creation, method invocation, and field access through reflection when necessary. The four basic methods in the library focus on these tasks: **create()**, **invoke()**,**get()** and **set()**. Each of these operations takes care of all of the

```
1  import static student.testingsupport.ReflectionSupport.*;
2  public class ReflectionSupportScoreBoard
3  {
4      public static void main(String args[])
5      {
6         Object sb = create("ScoreBoard", "VT", "GT");
7         invoke(sb, "setTeam1Score", 20);
8         String rTime = invoke(sb, String.class, "remainingTime");
9      }
10 }
```

**Fig, (7).** Manipulating a ScoreBoard using ReflectionSupport.




**Fig, (8).** Overview of the create() method.

required steps and error checking required by Java's native reflection API.

The ReflectionSupport methods use overloading and genericity to provide for flexible usage while maintaining type safety. Programmers can perform most actions in a single statement without required **try-catch** blocks or explicit type casts. Errors and exceptions that may occur have specific internal handlers so that the appropriate cause can be reported in detail. Exceptions produced by code being invoked through reflection are unwrapped and propagated for the programmer to handle using the same approach as with non-reflection-based code.

Fig. (**7**) shows a version of the **MainScoreBoard** class from Fig. (**2**) rewritten using ReflectionSupport. Aside from the static import statement on line 1, there is a one-to-one correspondence between the lines in the **ReflectionSupport ScoreBoard** and the **MainScoreBoard** classes.

### 4.1. Creating Objects Using Create()

To create an object using reflection, the programmer uses the **create()** method as shown on line 6 of Fig. (**7**). This method takes an initial argument that identifies the class or constructor to use, followed by zero or more parameters to pass to the constructor. Fig. (**8**) summarizes the calling pattern for the **create()** method.

The initial parameter to **create()** is most commonly either a **Class** object or a class name provided as a string to indicate the type of object to create. Alternatively, one can also provide a **Constructor** object explicitly. These alternatives are supported by providing three overloaded versions of **create()** so the programmer can provide the type of object that is most appropriate in a given situation. The versions of **create()** that accept **Class** and **Constructor** objects are generic, with a return type deduced from the first parameter so that no explicit casting is necessary. A type cast is only required if the programmer supplies the class name as a string.

In addition to the parameter identifying the type of object to create, the **create()** method also takes a variable argument list of actual values to pass to the constructor it calls. It dynamically loads the class if necessary, and then searches for a constructor that will accept the given argument list. The constructor signature does not have to be specified—the **create()** method searches through the available constructors to find one that will accept the given arguments, including support for legal method invocation conversions on any parameter types.

Internally, the **create()** method handles all of the checked exceptions that might be thrown if no appropriate constructor is found—for example, if no constructor will accept the given argument list, or if the constructor is not accessible. In the Reflection Support library, a call to **create()** is treated as a *claim* that the corresponding constructor does in fact exist. As a result, an **AssertionError** is thrown when no appropriate constructor is found, with a message describing both the reason why as well as the closest matching constructor when appropriate. Thus, in cases where the programmer is confident of no errors, code can be written entirely without the

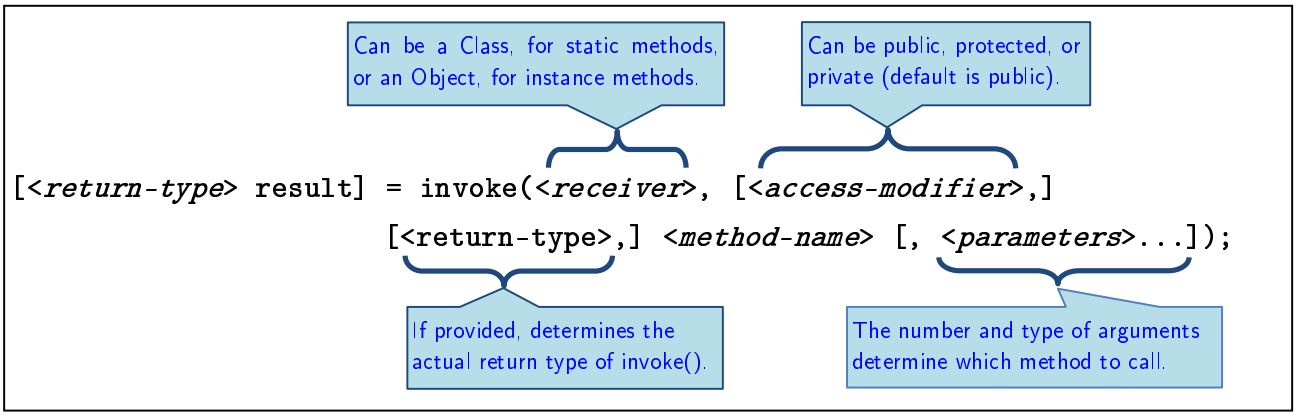



**Fig. (9).** Overview of the invoke() method.

use of **try-catch** blocks. Failures are reported in a way that will allow the problem to be identified quickly. In cases where there is no guarantee the underlying constructor exists, Reflection Support provides a **getConstructor()** method that performs the same search but returns the corresponding **Constructor** without invoking it, or null if none is found. This allows programmers to write conditional code that checks for existence in a streamlined way, and then call **create()** on the result.

What happens if the constructor that is invoked throws an exception? For simplicity, the three versions of **create()** discussed above have no **throws** clauses in their declarations, so they do not throw checked exceptions. If an **InvocationTargetException** is produced when the underlying constructor is invoked, it is intercepted and the inner exception is inspected. When the constructor throws any kind of unchecked exception, the original exception is unwrapped and re-thrown by **create()**, allowing any client code written to intercept the original exception to be used without change. If the underlying constructor throws a checked exception, however, it cannot be re-thrown directly. Instead, **create()** unwraps the **InvocationTargetException** to find the original exception object, and then re-throws it after wrapping it inside a **RuntimeException**.

In those cases where the programmer expects that the underlying constructor may throw a checked exception, the **createEx()** method ("Ex" for exception) can be used instead. This method behaves exactly the same as **create()**, but its signature declares that it may throw an instance of **Exception** (or any of its subclasses). This requires the programmer to include an explicit **try-catch** block, but allows **createEx()**to re-throw checked exceptions produced by the underlying constructor without any wrapping. The programmer can then write **try-catch** blocks in terms of the native exception types that may be produced by the underlying constructor without regard to the wrapping performed by the native reflection API.

### 4.2. Invoking Methods Using Invoke()

Reflective method calls are implemented using **invoke()**, as shown on lines 7 and 8 of Fig. (**7**). The **invoke()** method is similar in many respects to **create()**. Fig. (**9**) summarizes the parameters to **invoke()**.

The first argument to **invoke()** determines the receiver of the method call, which can either be a **Class**—for static methods—or an **Object**—for instance methods. Optionally, the programmer can also specify the visibility of the desired method, which defaults to public if omitted. Next, the programmer specifies the expected return type of the method. Note that this need not be the declared return type, but is instead the programmer's expectation of what kind of value will be returned. The **invoke()** method is generic, so that if a return type is specified by the programmer, that type will also be used as the declared return type for **invoke()** itself, so no explicit casting is required. If the return type is omitted, the method is presumed to be a void method that produces no return value.

The next parameter to **invoke()** is the name of the method to call, provided as a string. Following the method name, the programmer can provide a variable number of actual arguments to pass to the method being called. As with **create()**, the exact parameter profile of the method is not specified. Instead, **invoke()** searches for a method with the given name that will accept the given actual parameters, including support for appropriate argument conversions. This search takes inheritance into account, searching through all methods that can be called on the receiver, regardless of how far up the inheritance chain they are declared.

Just like the **create()** method, **invoke()** handles all of the checked exceptions that might be thrown if no appropriate method is found. Specific diagnostics are provided in the form of an **AssertionError** containing a message describing the reason no match was found and including the closest (inexact) match where possible. For programmers who wish to check for the presence of methods, **getMethod()** is provided. It performs the same search but returns null if no method is found. An overloaded version of **invoke()** that accepts a **Method** object instead of string name can then be used to call the result if desired.

The **invoke()** method handles exceptions thrown by the underlying operation the same way as **create()**. Any **InvocationTargetException** that arises is automatically unwrapped. If the original exception is an unchecked exception, it is re-thrown in its original form. If it is a checked exception, it is re-thrown inside a **RuntimeException**. If the programmer expects a method to throw a checked exception,

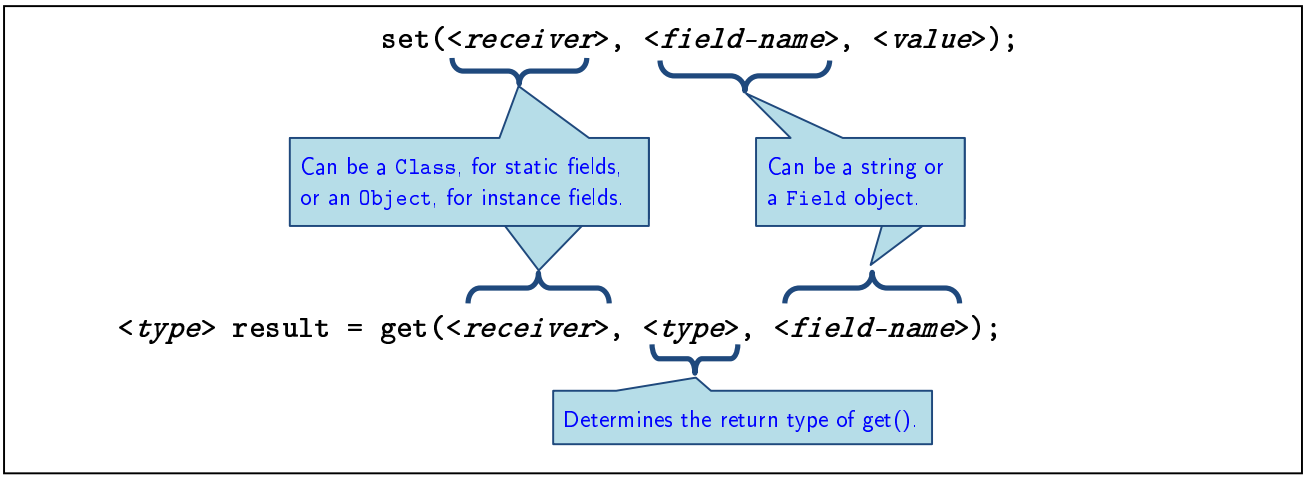



**Fig. (10).** Overview of the set() and get() methods.

the **invokeEx()** method will unwrap and re-throw such exceptions in their original form so that **try-catch** blocks written with the underlying method in mind can be used to handle the exceptions.

### 4.3. Accessing Fields Using Set() and Get()

ReflectionSupport also allows programmers to get or set fields via its **get()** and **set()** methods, which are similar in style to **create()** and **invoke()**. Field access is not shown in Fig. (**7**), since it is more common for programmers to utilize the available public methods on a class rather than to access fields directly. For completeness, however, reflective access to fields is necessary. Further, there are some situations where it is used in practice. When writing unit tests, programmers sometimes find it easier to directly access fields that are normally encapsulated, either to set up an object in an appropriate state for running a specific test, or to check that internal conditions meet expectations after some method executes. That is one reason why the FEST-Reflect library, which was designed to provide the support needed for writing unit tests, includes its own reflection-based field access support.

Fig. (**10**) summarizes the structure of calls to **get()** and **set()**. Both methods transparently handle inherited and declared fields, as well as take care of automatic boxing and unboxing conversions for primitive and wrapper types. The **set()** method takes the receiver as its first parameter—either a **Class**, when accessing a static field, or an **Object**, when accessing an instance field. In addition to the receiver, **set()** takes the field name and the value to store in the field. The field name can be specified as either a string value or as a **Field** object. The method searches for the field, including inherited fields if necessary, and checks that the actual value is assignable to the field's declared type. The **get()** method's parameters include the receiver (class or object), the expected type of the field, and the field's name. As with **create()** and **invoke()**, these methods throw an **AssertionError** if problems occur.

To see how these methods might be used, consider setting the **team1Score** field in a **ScoreBoard** object:

```
set(scoreboard, "team1Score", 20);
```

Similarly, the field value can be retrieved by reflection as well:

```
int score = get(scoreboard, int.class, "team1Score");
```

As with **create()** and **invoke()**,**get()** is generic, taking its actual return type from the type specified in the parameter list. A **getField()** method is also provided for programmers who wish to check for the presence of a field or retrieve a **Field** object for further manipulation.

## 5. EVALUATING THE IMPACT OF REFLECTION SUPPORT ON CODE SIZE

ReflectionSupport is designed to provide a simpler interface for manipulating objects with reflection. It resolves the issues described in Section 2.2 more effectively than other existing approaches. When writing code using ReflectionSupport, a programmer no longer needs to split basic actions into multiple steps, employ mandatory **try-catch** blocks when calling code that cannot throw checked exceptions, explicitly cast the types of parameters or return values, or set accessibility. When executing code that might throw exceptions, handlers can be written directly in terms of those native exceptions, without regard for **InvocationTargetException** wrappers.

To evaluate ReflectionSupport, we focused on how its use affects program size. The amount of coding effort, number of bugs, and readability of a program are generally proportional to its length. Writing code using Java's native reflection features increases code size drastically, which negatively affects programmer productivity and increases the likelihood of defects. ReflectionSupport requires much less additional code. This makes it more intuitive to learn and makes the resulting code easier to maintain. Section 5.1 considers an analytical comparison of relative source code sizes,

while Section 5.2 presents simple measures of relative code bloat based on a representative class.

### 5.1. Comparing Features Line-by-Line

As discussed in Section 2, the three primary object manipulation tasks are creating new objects, invoking methods, and manipulating fields. We can examine the relative cost of using ReflectionSupport in terms of "extra code" by comparing it against the same tasks performed using Java's native reflection API. For example, consider simple object creation:

```
ScoreBoard scoreboard =
   new ScoreBoard("VT", "GT");
```

To accomplish this same task using Java's reflection API requires two method call statements, a **try-catch** block, and an explicit type cast:

```
try {
   Constructor ctor = ScoreBoard.class
      .getConstructor(
         String.class, String.class);
   ScoreBoard scoreboard = (ScoreBoard)ctor
      .newInstance("VT", "GT");
}
catch (Exception e) {
   // perform some action
}
```

Here, the one line from the original becomes seven with Java's reflection API. Even more code would be needed if one wished to pinpoint the source of the error. With ReflectionSupport, the equivalent code segment is:

```
ScoreBoard scoreboard =
   create(ScoreBoard.class, "VT", "GT");
```

This form is much closer in size and readability to the original. Because of the generic nature of **create()**, no type cast is needed if a **Class** is provided as the first parameter. No **try-catch** block or additional method calls are required.

Now consider the same situation, but where the constructor may throw a checked exception:

```
try {
   ScoreBoard scoreboard =
      new ScoreBoard("VT","GT");
}
catch (CustomException e) {
   // perform some action
}
```

To accomplish this same task using Java's reflection API requires two method-call statements, a dynamic type check, and an explicit cast:

```
try {
   Constructor ctor = ScoreBoard.class
      .getConstructor(
         String.class, String.class);
   ScoreBoard scoreboard = (ScoreBoard)ctor
      .newInstance("VT", "GT");
}
catch (InvocationTargetException e) {
   if (e.getCause() != null && e.getCause()
      instanceof CustomException) {
      CustomException ce =
       (CustomException)e.getCause();
      // perform some action
   }
}
```

The six lines from the original become ten with Java's reflection API. With ReflectionSupport, the equivalent code segment is:

```
try {
   ScoreBoard scoreboard = createEx(
         ScoreBoard.class, "VT", "GT");
}
catch (CustomException e) {
   // perform some action
}
catch (Exception e) {
   // just to keep the compiler happy
}
```

ReflectionSupport is the same size as the original, except for an extra catch clause for Exception, needed because createEx() is declared to throw this more general type.

When examining methods, a similar pattern emerges. The following statements call methods on a ScoreBoard:

```
scoreboard.setTeam1Score(20);
String remaining = scoreboard.remainingTime();
```

Repeating these same actions with Java's reflection API gives the following:

```
try {
   Method setT1S = scBoard.getMethod(
         "setTeam1Score", int.class);
   setT1S.invoke(scoreboard, 20);
   Method prt =
      scBoard.getMethod("remainingTime");
   String rTime = (String)prt.invoke(scoreboard);
}
catch (Exception e) {
   // perform some action
}
```

Here, the number of methods required doubles, a try-catch block is required, explicit type casts are required for methods that return values, and the two-line sequence grows to nine lines. Using ReflectionSupport, however, produces a code segment similar in size to the original:

```
invoke(scoreboard, "setTeam1Score", 20);
String remaining = invoke(scoreboard,
   String.class, "remainingTime");
```

If we imagine that one of the methods might throw a checked exception, the **invokeEx()** results in the same effect shown above for **createEx()** when creating an object.

Finally, we can consider getting and setting a field. For normal classes, one does not typically have direct access to fields due to encapsulation choices commonly employed by programmers. Under some circumstances, however, such as when writing white-box software tests, direct access to normally encapsulated fields can be helpful. Given the correct visibility constraints, a field within a scoreboard could be read and written this way:

```
String oldValue = scoreboard.team1;
scoreboard.team1 = "Dodgers";
```

With Java's native reflection API, the following code would be necessary:

```
try {
   Field team1 = ScoreBoard.class
            .getDeclaredField("team1");
   team1.setAccessible(true);
   String oldValue =
      (String)team1.get(scoreboard);
   team1.set(scoreboard, "Dodgers");
}
catch (Exception e) {
   // perform some action
}
```

Note that two steps are required to gain access to the field, in addition to the one required to read or write its value. Here, the original two-line sequence has grown to nine, including a required **try-catch** block and a required type cast on the result of reading from the field. With ReflectionSupport, however, the code is the same size as the original:

```
String oldValue = get(
   scoreboard, String.class, "team1");
set(scoreboard, "team1", "Dodgers");
```

### 5.2. Measures of Code Size in a Representative Class

Section 5.1 compares ReflectionSupport code to equivalent code written using Java's native reflection API, showing in principle that code written using ReflectionSupport is similar in size to equivalent non-reflective code, while Java's native reflection API requires two to six times as much code. However, such an analytical comparison does not necessarily translate directly to realistic uses of reflection. Unfortunately, there are no generally accepted benchmarks of how reflection is typically used in programs. Indeed, many frameworks take advantage of reflection, but in stylized ways depending on their particular needs.

To address this issue and attempt to measure (on a small scale) how ReflectionSupport affects code size in an actual class, we selected a specific scenario to examine. We selected the context of writing software tests for a simple class, since software tests are intended to exercise all of the features of the class under test. A JUnit test class, for example, would provide a clean, self-contained "client example" that attempts to fully utilize all of the methods and behaviors in the class under test.

We selected a target class for the unit test that provides getters and setters for basic properties, in addition to methods that implement behaviors using those properties. In all, the target class provides eight public methods and one constructor, none of which are declared to throw checked exceptions. In addition, a corresponding JUnit test class was written to check the behaviors of all public features. It included eight separate test methods, each intended to double-check all the behaviors of a specific public method in the class under test. The test case also included a **setUp()** method that created a fresh instance of the class under test for use in each test method.

We then proceeded to create a purely reflective version of the JUnit test that accessed the class under test using only Java's native reflection API. Reflection was used to create instances and to invoke any public methods of the class under test. Since all fields in the class under test were private, and client code would normally utilize only the class' public features, we explicitly avoided using direct field access in this example in order to be more representative of typical practices. We also created a purely reflective version of the JUnit test that used ReflectionSupport only, rather than Java's native reflection API. We wrote equivalent versions of the same code using each of the other reflection APIs described in Section 3 where possible—several of the APIs were not expressive enough to perform all of the required actions, however. Table **1** numerically summarizes the comparison among all the approaches, while Fig. (**11**) graphically summarizes the comparison.

Both the fluent interface approach and the enterprise reflection APIs were omitted from this comparison because neither provides a mechanism for invoking general methods. As a result, the example class used in this comparison could not be expressed reflectively using either approach.

Writing code using Java's native reflection API increases code size by a factor of three. The growth in source code size is due primarily to the addition of **try-catch** blocks and the use of multiple statements to carry out each basic action. Although the native reflection API version of the example combines **try-catch** blocks where possible and uses just a single **catch** clause per **try**, it still required nine such blocks (essentially, one per test method) whereas the version using ReflectionSupport required none. Among all the operations, object creation and method invocation are the most frequent and most significant in this example.

In Table **1**, "operations invoked" refers to the raw number of underlying method calls or constructor calls on the unit under test, which is the same across all five versions (hence, it does not appear in Fig. (**11**). "Source lines" in Table **1** and Fig. (**11**) refers to a count of non-comment, non-blank lines of source code (NCSLOC), giving a measure of the length of each version. The chart in Fig. (**11**) uses the native reflection version as a baseline for comparison, so that impact of each version can be seen relative to that of Java's native reflection API. The source code for the ReflectionSupport version was only one line longer than the original—

**Table 1. Numerical Comparison of the Impact of Each Approach to Simplifying Reflection**

| Reflection API | Operations Invoked | Source Lines | Method Calls | Required Try/ Catch Blocks | Type Casts | getClass()/.class References |
|---|---|---|---|---|---|---|
| Original (no reflection) | 12 | 34 | 20 | 0 | 0 | 0 |
| Java Reflection API | 12 | 93 | 49 | 9 | 13 | 17 |
| ReflectionSupport | 12 | 35 | 24 | 0 | 4 | 8 |
| Java Beans | 12 | 84 | 37 | 9 | 13 | 1 |
| FEST-Reflect | 12 | 37 | 60 | 0 | 4 | 15 |
| Fluent Interfaces | N/A | N/A | N/A | N/A | N/A | N/A |
| Enterprise Reflection APIs | N/A | N/A | N/A | N/A | N/A | N/A |

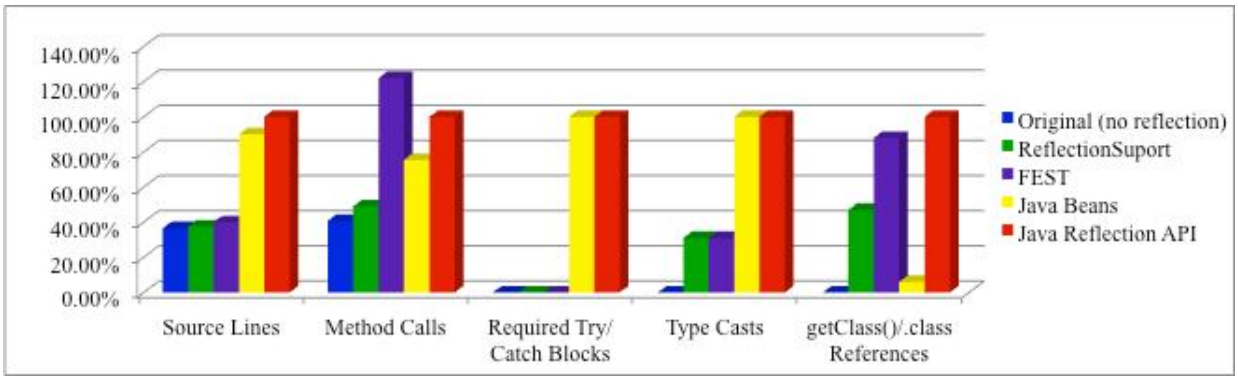



**Fig. (11).** A graphical comparison of the impact of each approach to simplifying reflection.

that line being the extra import statement needed for accessing ReflectionSupport's features. The version using Java's native reflection API was almost three times as long as the original, however, giving a better feel for the average code bloat over a variety of operations. This size is consistent with the two-to-six range described in Section 5.1. The FEST-Reflect version was only three lines longer than the original. The Java Beans version of the example class was almost as long as the native reflection version, because it requires its methods to be wrapped in **try-catch** blocks.

The "Method Calls" column in Table **1** and Fig. (**11**) is a count of method calls or constructor calls appearing in the source code. This shows a different measure of length than measuring lines of source code. Interestingly, FEST-Reflect employed the largest number of method calls in this example. That is due to the use of chained method calls to form argument lists in each basic action. Native reflection employed the second largest number of calls. ReflectionSupport was closest to the original code in terms of the number of method and constructor calls.

The "Required Try/Catch Blocks" column in Table **1** and Fig. (**11**) refers to the number of **try-catch** blocks that were added to the code due to checked exceptions potentially thrown from API methods. In all cases, exception-handling statements were combined where possible resulting in the minimum number of **try-catch** blocks. More **try-catch** blocks would be necessary in many cases if differentiating the cause or source of errors were necessary. Java's reflection API required the most **try-catch** blocks because it uses checked exceptions in most methods. Java Beans requires the same number of **try-catch** blocks since it also uses checked exceptions for most methods. Both ReflectionSupport and FEST-Reflect required no **try-catch** blocks, since they both avoid checked exceptions in method signatures where possible.

Table **1** and Fig. (**11**) also compare the number of type casts necessary under each approach. When type casts are required, they are usually for converting method return values to a more specific type. The values reported here include both explicit type casts and also manual unboxing operations on wrapper types (e.g., calling the **intValue()** method on an **Integer**, for example). While Java will automatically unbox wrapped primitive values, in some contexts (such as when method overloading prevents a unique interpretation) it is necessary to manually unbox primitive values. Both ReflectionSupport and FEST-Reflect required four type casts, while Java Beans and the native reflection API required three times this number. This is because both Java Beans and the native reflection API use **Object** as the return type when invoking methods by reflection, making type casts commonplace. Both ReflectionSupport and FEST-Reflect use generics for inferring return types where possible. All four type casts required by these two libraries were situations where manual unboxing was necessary because of context—where the return value of a method was being passed to a second, overloaded method, so that the compiler could not automatically determine if unboxing was desired. The count of thirteen type casts for both native Java reflection and Java Beans includes these same four manual unboxing method calls, together with nine explicit type casts on method return values.

**Table 2. Summary of Features of Reflection APIs, where (+) Indicates a Strength and (−) Indicates a Weakness**

| Features | Native Reflection API | Java Beans | Fluent Interfaces | FEST-Reflect | Enterprise Reflection APIs | Reflection-Support |
|---|---|---|---|---|---|---|
| Statements per task | 2 or more (−) | 2 (−) | 1 (+) | 1 (+) | 1 (+) | 1 (+) |
| Methods per task | 2 or more (−) | 2 (−) | Many (−) | Many (−) | 1 (+) | 1 (+) |
| Requires try-catch | Yes (−) | Yes (−) | No (+) | No (+) | No (+) | No (+) |
| Always wraps exceptions | Yes (−) | No (+) | Swallows (−) | Yes (−) | Yes (−) | No (+) |
| Allows catching inner exceptions | No (−) | Yes (+) | No (−) | No (−) | No (−) | Yes (+) |
| Requires type casts | Yes (−) | Yes (−) | No (+) | No (+) | No (+) | No (+) |
| Allows field access | Yes (+) | No (−) | No (−) | Yes (+) | Yes (+) | Yes (+) |
| Allows method invocation | Yes (+) | Yes (+) | No (−) | Yes (+) | No (−) | Yes (+) |
| Inheritance-based method lookup | No (−) | Yes (+) | No (−) | No (−) | No (−) | Yes (+) |
| Overload resolution | No (−) | Yes (+) | No (−) | No (−) | No (−) | Yes (+) |
| Informative errors | No (−) | No (−) | No (−) | No (−) | No (−) | Yes (+) |

The final column in Table **1** indicates the number of times **Class** objects were explicitly accessed in the resulting code, either to look up features or to provide type specifications as parameters in API methods. ReflectionSupport provides for using **Class** objects as parameters in **create()** and **invoke()** to customize the return type, in effect taking the place of explicit type casts on results. Java's reflection API uses these objects for many more purposes, including explicitly specifying the formal argument signatures of methods to search for, identifying the declaring class of a method, and so on. As a result, Java's reflection API required roughly twice as many uses of **Class** objects. Java Beans requires only one **Class** object since it does not use generics. On the other hand, FEST-Reflect requires all the parameter and return types to be specified using a series of **Class** objects, leading to a count much closer to that of the native reflection API.

Although this example is small, it shows how code bloat caused by Java's native reflection API can affect (and complicate) even simple classes where reflection is employed. It also shows that ReflectionSupport allows one to write purely reflection-based code that is similar in size and structure to equivalent direct-access code written without reflection. By avoiding the extra code required by the native API, code written using ReflectionSupport is simpler to read and easier to maintain in comparison. In terms of code bloat, FEST-Reflect is the approach closest to ReflectionSupport relative to the size reduction over native reflection.

## 6. CONCLUSIONS

Reflection is a practical tool with many uses, although in Java, the native API for reflection is cumbersome to use. Various projects have provided alternative APIs, attaining partial success in resolving the three basic issues of native reflection in Java: 1) Common tasks require multiple steps to complete, 2) explicit **try-catch** blocks are required around all the steps, and 3) reflection methods wrap any exceptions thrown by the invoked code inside **InvocationTargetException** objects, making it more difficult to employ user-provided handlers. The ReflectionSupport library addresses all of these limitations, while trying to maximize ease of use and minimize the amount of source code the programmer must write.

Table **2** summarizes the features of the various approaches to Java reflection discussed in this paper, in comparison with the native Java reflection API. The native reflection support provided as part of Java's standard library is expressive, but otherwise has weaknesses in all other issues. Each approach discussed in Section 3 addresses some of the weaknesses with the native library, but none of the alternative approaches address all of the weaknesses listed in Table **2**. In comparison, ReflectionSupport addresses all of the weaknesses listed.

For the key tasks of object creation, method invocation, and field access, some approaches are bulkier, requiring multiple conceptual steps that are typically written as separate program statements. Even when only one statement is required, some approaches rely on chaining multiple method calls together, making the statement longer and more divergent from non-reflective code that performs the same task. Some approaches require the programmer to write **try-catch** blocks because the API operations are declared to throw checked exceptions, which can lead to bulkier code that is more difficult to maintain. Except for ReflectionSupport and Java Beans, however, all of the approaches in Section 3 always wrap any exceptions thrown by the code that is being invoked reflectively (the fluent interface approach described in Section 3 swallows such exceptions without propagating them at all). Some approaches, such as the native reflection API and the Java Beans API, declare the possibility of checked exceptions, which is why those approaches require

**try-catch** blocks. Others, such as FEST-Reflect and the enterprise reflection APIs, wrap any exceptions in an unchecked exception, such as **RuntimeException**, allowing **try-catch** blocks to be omitted. Wrapping strategies prevent the programmer from writing **catch** clauses for the actual type(s) of exceptions thrown by the code being invoked, however. Instead, the programmer must write reflection-aware handlers that catch the wrapper exception, unwrap it, and then dispatch appropriately. ReflectionSupport, on the other hand, actively unwraps thrown exceptions that are unchecked, and if the programmer indicates (by using an "Ex" method), will also unwrap checked exceptions. This allows the programmer to write **catch** clauses in terms of the actual exception type(s) thrown by the code being invoked.

As discussed in Section 3, the native reflection and Java Beans approaches use the type **Object** as the return type when one is needed in a particular task. This requires the programmer to use explicit type casts. Other approaches avoid explicit type casts, often through the use of generics. Also, only three of the approaches—the native API, FEST-Reflect, and ReflectionSupport—provide support for the full set of tasks: object creation, method invocation, and field access. The other approaches leave out one or more of these tasks because they were not intended to be comprehensive reflection libraries. In addition, most approaches require the programmer to know the specific class declaring a field before it can be accessed, or know the declaring class and exact signature of a method before it can be invoked. Only ReflectionSupport and Java Beans find methods using an inheritance-based lookup process so that all methods on an object can be invoked without regard for which (super) class declares them, and perform argument matching with overloading in mind in order to find the best method that will accept the given arguments without requiring the programmer to restate the exact method signature.

Finally, ReflectionSupport is the only approach discussed here that takes steps to produce more informative exception messages when problems arise. Instead of just reporting a failure to find some feature (a method, constructor, or field), it reports the *reason* no match was found when possible (i.e., what property of the desired feature could not be matched), and will also describe the *closest match* it was able to locate. This aids programmers in diagnosing the cause of reflective binding failures when they do happen.

Reflection is an integral part of Java. In spite of its benefits, programmers often avoid using reflection due to the complex, verbose nature of code written using Java's reflection API. Though there have been several efforts at creating improved APIs for reflection, they are sometimes project-specific and often do not support the full range of tasks one can perform with reflection. The ReflectionSupport library provides a straightforward mechanism for using reflection to access and manipulate objects that addresses the full range of shortcomings in Java's native API.

We have discussed how the static methods provided by Reflection Support help in writing clean code, while being intuitive and easy to learn. These methods handle the three basic tasks of object creation, method invocation, and field reading/writing. The abstraction layer provided by ReflectionSupport hides the complexities of Java reflection from developers. Therefore, ReflectionSupport improves the readability, writeability, and maintainability of Java programs that use reflection. Source code for this project is available at: http://sourceforge.net/projects/web-cat/files/Reflection-Support/

## CONFLICT OF INTEREST

The authors confirm that this article content has no conflicts of interest.

## ACKNOWLEDGEMENT

Declared none.

## REFERENCES

[1] D. G. Bobrow, R. P. Gabriel, and J. L. White, "CLOS in context: the shape of the design space," In: *Object-oriented programming*, MIT Press, USA, 1993, pp. 29-61.

[2] S. Chiba, "A metaobject protocol for C++," *SIGPLAN Not.,* vol. 30, pp. 285-299, 1995.

[3] P. Maes, "Concepts and Experiments in Computational Reflection," In: *Proceedings of the 2nd Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'87)*, Orlando, Florida, USA, 1987, pp. 147–156.

[4] I. R. Forman and N. Forman, Eds., *Java Reflection in Action*. Manning Publications,USA, 2004.

[5] S. Microsystems, "JavaTM Core Reflection API and Specification," February 1997.

[6] S. Schmidt, *Code Monkeyism-The Blog for Developers*. Available: http://codemonkeyism.com/fluent-interface-and-reflection-for-object-building-in-java [Last Accessed on August 31, 2012].

[7] A. Ruiz, *FEST-Reflect*. Available: http://docs.codehaus.org/-display/FEST/Reflection+Module [Last Accessed on August 31, 2012].

[8] *XStream*. Available: http://xstream.codehaus.org/index.html [Last Accessed on August 31, 2012].

[9] *XUI*. Available: http://www.xoetrope.com/xui [Last Accessed on August 31, 2012].

[10] J. Gosling, B. Joy, and G. Steele, Eds., *The Java Language Specification*. Addison-Wesley Longman Publishing Co., Inc: Massachusetts, 1996.

[11] S. Liang, Ed., *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, Massachusetts, 1999.

[12] W. Cazzola, "Smart Reflection: Efficient Introspection in Java," *J. Object Technol,* vol. 3, pp. 117-132, 2004.

[13] S. Microsystems, *Java TM Remote Method Invocation - Distributed Computing for Java,* S.Microsystem: USA,1998.

Received: February 15, 2013 Revised: April 08, 2013 Accepted: April 11, 2013