# Refactoring Support for Class Library Migration

Ittai Balaban
New York University
251 Mercer St., New York, NY 10012
balaban@cs.nyu.edu

Frank Tip    Robert Fuhrer
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598
{ftip,rfuhrer}@us.ibm.com

## ABSTRACT
As object-oriented class libraries evolve, classes are occasionally deprecated in favor of others with roughly the same functionality. In Java's standard libraries, for example, class `Hashtable` has been superseded by `HashMap`, and `Iterator` is now preferred over `Enumeration`. Migrating client applications to use the new idioms is often desirable, but making the required changes to declarations and allocation sites can be quite labor-intensive. Moreover, migration becomes complicated—and sometimes impossible—if an application interacts with external components, if a legacy class is not completely equivalent to its replacement, or if multiple interdependent classes must be migrated simultaneously. We present an approach in which mappings between legacy classes and their replacements are specified by the programmer. Then, an analysis based on *type constraints* determines where declarations and allocation sites can be updated. The method was implemented in Eclipse, and evaluated on a number of Java applications. On average, our tool could migrate more than 90% of the references to legacy classes.

## 1. INTRODUCTION
Class libraries evolve because it is difficult to anticipate what functionality is needed, and new usage patterns may arise after a library's initial deployment. The incorporation of additional functionality in a class library can often be accomplished in backward-compatible ways by creating additional interfaces, and/or adding methods to existing classes. However, when an existing library class provides insufficient flexibility because of an unfortunate design decision, it is often undesirable to make incompatible changes that would break existing library clients. As a result, classes are occasionally deprecated in favor of others with roughly the same functionality. In such cases, legacy classes are typically not removed from a library for backward compatibility reasons.

Several examples of the scenario described above can be found in the `java.util` collection libraries. Class `Hashtable` has been superseded by a class `HashMap` that is similar but allows unsynchronized[1] access to a hash-table's elements as well as the use of `null` keys and values. Class `ArrayList` is similar to class `Vector` except for the fact that it allows unsynchronized access to the list's elements. Furthermore, the `Iterator` interface, now favored over the similar `Enumeration` interface, uses more concise method names and provides an additional operation `remove()`. Outside of the standard collection classes, class `java.io.PrintWriter` is now preferred instead of the similar class `java.io.PrintStream` [5]. Moreover, Java 1.5 supports a class `java.io.StringBuilder` that provides the same functionality as `java.io.StringBuffer` but is not thread-safe. Migrating a client application to the new idioms is often desirable, but can be labor-intensive as it may involve updating many declarations, allocation sites, and call sites. Moreover, migration becomes complicated—and sometimes impossible—if an application interacts with external components, if a legacy class is not completely equivalent to its replacement, or if multiple interdependent classes must be migrated simultaneously.

The contribution of this paper is a technique for automatic migration of applications that use legacy library classes. In this approach, a *migration specification* defines *how* uses of legacy library classes are mapped to uses of their replacement classes. Each such mapping needs to be defined only *once*, and can then be used to migrate any number of applications. We use an analysis based on *type constraints* [25] to determine *where*, for a given migration specification, it is possible to migrate uses of legacy classes without affecting the program's type-correctness or behavior. In addition, an escape analysis (see e.g., [6, 1, 27]) serves to determine where it is safe to replace a synchronized legacy class with an unsynchronized replacement class[2]. In cases where it cannot be shown that synchronization can be removed safely for a given object $o$, our approach is to insert a synchronization wrapper around $o$. This is accomplished using an instance of the DECORATOR design pattern [15] where a collection type is implemented by delegating all operations to $o$, and where thread-safety is added by making all the forwarding methods `synchronized`. The synchronization wrap-

[1]In new Java container classes, synchronization is decoupled from a collection's functionality. Class `java.util.Collections` contains "synchronization wrapper" methods for transforming any `Map` or `List` into a synchronized `Map` or `List`, respectively.

[2]For example, consider the migration from the synchronized legacy class `Hashtable` to its unsynchronized replacement `HashMap`.

```
(1)  public class Wizard {
(2)    private static JComboBox selectionBox;
(3)    private static Hashtable h1 = new Hashtable();  /* A1 */
(4)    private static Vector v2;
(5)    public static void main(String[] args) {
(6)      for (int count = 0; count < 5; count++)
(7)        h1.put(new Integer(count),
(8)               "Item " + Integer.toString(count));
(9)      Vector v1 = new Vector();                      /* A2 */
(10)     retrieveData(v1);
(11)     v2 = runWizard(v1);
(12)     Object[] dbData = new Object[v2.size()];
(13)     v2.copyInto(dbData);
(14)     writeToDatabase(dbData);
(15)   }
(16)   private static Vector runWizard(Vector v3) {
(17)     Vector v4 = new Vector();                      /* A3 */
(18)     for (int sel = showWizardDialog(v3);
(19)          sel >= 0;) {
(20)       v4.addElement(v3.elementAt(sel));
(21)       sel = showWizardDialog(v3);
(22)     }
(23)     System.out.println("User Selections:");
(24)     printElements(v4.elements());
(25)     return v4;
(26)   }
(27)   private static void printElements(Enumeration e1) {
(28)     while (e1.hasMoreElements())
(29)       System.out.println(e1.nextElement());
(30)   }
(31)   private static void retrieveData(Vector v5) {
(32)     printElements(h1.elements());
(33)     for (Enumeration e2 = h1.elements();
(34)          e2.hasMoreElements();)
(35)       v5.addElement(e2.nextElement());
(36)   }
(37)   private static int showWizardDialog(Vector v6) {
(38)     selectionBox = new JComboBox(v6);
(39)     selectionBox.
(40)       addActionListener(new MyActionListener(v2));
       ...
(41)   }
(42)   static void writeToDatabase(Object[] data) { ··· }
(43) }
(44) public class MyActionListener implements ActionListener
(45) {
(46)   private Vector v7;
(47)   public MyActionListener(Vector v6) {v7 = v6;}
       ...
(48) }
```

(a)

```
public class Wizard {
  private static JComboBox selectionBox;
  private static HashMap h1 = new HashMap();
  private static List v2;
  public static void main(String[] args) {
    for (int count = 0; count < 5; count++)
      h1.put(new Integer(count),
             "Item " + Integer.toString(count))
    Vector v1 = new Vector();
    retrieveData(v1);
    v2 = runWizard(v1);
    Object[] dbData = new Object[v2.size()];
    Util.copyInto(v2, dbData);
    writeToDatabase(dbData);
  }
  private static List runWizard(Vector v3) {
    List v4 = Collections.synchronizedList(new ArrayList());
    for (int sel = showWizardDialog(v3);
         sel >= 0;) {
      v4.add(v3.elementAt(sel));
      sel = showWizardDialog(v3);
    }
    System.out.println("User Selections:");
    printElements(v4.iterator());
    return v4;
  }
  private static void printElements(Iterator e1) {
    while (e1.hasNext())
      System.out.println(e1.next());
  }
  private static void retrieveData(Vector v5) {
    printElements(h1.values().iterator());
    for (Iterator e2 = h1.values().iterator();
         e2.hasNext();)
      v5.addElement(e2.next());
  }
  private static int showWizardDialog(Vector v6) {
    selectionBox = new JComboBox(v6);
    selectionBox.
      addActionListener(new MyActionListener(v2));
    ...
  }
  static void writeToDatabase(Object[] data) { ··· }
}
public class MyActionListener implements ActionListener
{
  private List v7;
  public MyActionListener(List v6) {v7 = v6;}
  ...
}
```

(b)

**Figure 1: (a) Example program that uses the legacy classes `Vector`, `Hashtable`, and `Enumeration`. (b) The example program after migrating from `Vector` to `ArrayList`, `Hashtable` to `HashMap`, and `Enumeration` to `Iterator`. Underlining indicates changed code fragments. A1, A2, and A3 denote allocation sites in the program.**

pers that we use are provided by the standard library class `java.util.Collections` exactly for this purpose.

Our work was implemented as a *refactoring* [13, 22] in Eclipse (see `www.eclipse.org`), a widely used open-source development environment for Java, using an existing refactoring infrastructure [2]. Thus far, our experiments have concentrated on migrations involving classes from the Java Standard Collections Framework, but our techniques can also be used for migrations involving other library classes, and for migrations between user-defined classes. At present, the framework requires that the migration of method calls does not change the exceptions that may be thrown, and migrations are simply disallowed if this is not the case. This has not posed any problems in practice, because we did not encounter situations that required nontrivial migrations of exception types while conducting our experiments. Section 2 presents a simple workaround mechanism that is sufficient to handle most cases of migrations between methods that differ in the thrown exceptions. In the long term, it does not seem especially difficult to support the migration of exception types, as is discussed in Section 7. Furthermore, we expect that handling other statically typed object-oriented languages (e.g., C#) would only require minor modifications of the framework.

We evaluated our technique on a number of moderate-sized Java applications in which we migrated a number of heavily used legacy classes from the standard collection libraries. Specifically, we considered migration from `Vector` to `ArrayList`, `Hashtable` to `HashMap`, and `Enumeration` to `Iterator`. In these benchmarks, our tool could automatically migrate an average of 90.4% of the declarations that refer to legacy classes, 96.6% of the call sites, and 92.0%

| | | | |
|---|---|---|---|
| (1) | **new** Vector(), *unsynchronized* | → | **new** ArrayList() |
| (2) | **new** Vector(), *synchronized* | → | Collections.synchronizedList(**new** ArrayList()) |
| (3) | int Vector:receiver.size() | → | int receiver.size() |
| (4) | Object Vector:receiver.firstElement() | → | Object receiver.get(0) |
| (5) | Object Vector:receiver.setElementAt(Object: value, int: index) | → | Object receiver.set(index, value) |
| (6) | void Vector:receiver.copyInto(Object: array) | → | void Util.copyInto(receiver, array) |
| (7) | Enumeration Vector:receiver.elements() | → | Iterator receiver.iterator() |
| (8) | **new** Hashtable(), *unsynchronized* | → | **new** HashMap() |
| (9) | **new** Hashtable(), *synchronized* | → | Collections.synchronizedMap(**new** HashMap()) |
| (10) | Object Hashtable:receiver.put(Object: key, Object: value) | → | Object receiver.put(key, value) |
| (11) | Enumeration Hashtable:receiver.elements() | → | Iterator (Collection receiver.values()).iterator() |

**Figure 2: Specification used for migrating the example program of Figure 1.**

of the allocation sites. Moreover, for the migrated allocation sites, only a small fraction needed to be surrounded with synchronization wrappers. Code inspection revealed that the remaining occurrences of legacy classes could not be migrated due to interaction with external libraries such as *Swing*.

The remainder of this paper is organized as follows. Section 2 presents a motivating example that illustrates our approach. Section 3 presents the model of type constraints that will be used to determine where migrations are allowed, and Section 4 describes how type constraints are solved. Section 5 discusses the evaluation of our approach on a set of moderate-sized benchmarks. Related work is discussed in Section 6. Finally, conclusions and directions for future work are presented in Section 7.

## 2. MOTIVATING EXAMPLE

Figure 1(a) shows an example program that implements a simple "wizard" using a number of components from the *Swing* class library. The program displays a list of values in a drop-down menu, and prompts the user to select one or more of these. Then, the selected values are written to a database via a call to the method `writeToDatabase(Object[])`, the definition of which is omitted. Our goal is to transform this program so that it uses `ArrayList` instead of the legacy class `Vector`, `HashMap` instead of `Hashtable`, and `Iterator` instead of `Enumeration`. Figure 1(b) shows the desired refactored program.

`ArrayList` was introduced in the standard libraries to replace `Vector`, and is considered preferable because its interface is minimal and matches the functionality of the `List` interface. More importantly, it allows for unsynchronized access to a list's elements whereas all of `Vector`'s methods are `synchronized`[3] which results in unnecessary overhead when `Vector`s are used by only one thread. Similarly, `HashMap` was introduced to replace `Hashtable` and allows for unsynchronized access to a hash-table's elements. Moreover, `HashMap` allows for `null` values and it does not contain any of `Hashtable`'s methods that produce `Enumerations`. The `Iterator` interface that replaces `Enumeration` features shorter method names and supports an additional `remove()` method for the safe removal of elements during iteration.

Figure 2 shows a migration specification, as would be written by a user, for migrating the example program. This specification consists of a set of rewrite rules that express *how* allocation sites and method calls on legacy classes[4] `Vector`, `Hashtable` and `Enumeration` can be rewritten. For example, rule (3) states that migrating calls to method `Vector.size()` requires no modification, since `ArrayList` defines a syntactically and semantically identical method. If methods in the legacy class are not supported by the replacement class, rewriting method calls becomes more involved. For example, `Vector` supports a method `firstElement()` not defined in `ArrayList`. Rule (4) states that a call `receiver.firstElement()`, where `receiver` is an expression of type `Vector`, should be transformed into `receiver.get(0)`. In cases where it is not possible to express the effect of a method call on the legacy class in terms of calls to methods on the replacement class, calls may be mapped to methods in user-defined classes. For example, `Vector` has a method `copyInto()` that copies the contents of a `Vector` into an array. Since `ArrayList` does not provide this functionality, rule (6) transforms `receiver.copyInto(array)` into a call to method `Util.copyInto(receiver, array)` of the `Util` class shown in Figure 3. This strategy can also be used to migrate between methods that throw different types of exceptions. Specifically, a user-defined class method can be used to wrap a method in a replacement class in order translate between exception types. Finally, when an operation in a legacy class that is not supported by a replacement class cannot be modeled using an auxiliary class, migration may become impossible.

```
public class Util {
  public static void copyInto(ArrayList v,
                              Object[] target) {
    if (target == null)
      throw new NullPointerException();
    for (int i = v.size() - 1; i >= 0; i--)
      target[i] = v.get(i);
  }
}
```

**Figure 3: Auxiliary class that contains a method used for migrating `Vector.copyInto()`.**

Rules (1) and (2) are both concerned with rewriting allocation sites of the form `new Vector()`. The former applies in cases where thread-safety need not be preserved, and transforms the allocation site into an expression `new ArrayList()`. The latter applies in situations where thread-safety must be preserved, and transforms

[3]As noted in e.g., [6, 1, 27], synchronization operations in Java programs may incur significant performance overhead.

[4]In this discussion we use the term *class* to refer to Java classes as well as to interfaces.

the allocation site into `Collections.synchronizedList(` **new** `ArrayList())` using a standard synchronization wrapper in the class `java.util.Collections`. Our tool relies on an escape analysis to determine which of the two rules should be applied, and prefers (1) over (2) whenever possible.

It is important to realize that, while the specification of Figure 2 describes *how* program fragments are transformed, it does not state *when* the transformation is allowed. We will now examine the program of Figure 1 to illustrate the limitations on migration.

Line (38) of the example program contains a call to a constructor `javax.swing.JComboBox(Vector)` in the *Swing* library, where `v6` (declared on line (37)) is used as an actual parameter. The type of `v6` cannot be changed to `ArrayList` because: (i) the methods of library class `JComboBox` cannot be changed, and (ii) changing `v6`'s type to `ArrayList` would result in a type error because `ArrayList` is not a subtype of `Vector`[5]. Furthermore, observe that method `showWizardDialog(Vector)` is invoked on line (18), using `v3` as the actual parameter. `v3`'s type cannot be changed to `ArrayList` because: (i) we just argued that the type of its formal parameter `v6` cannot be changed, and (ii) changing `v3`'s type to `ArrayList` would result in a type-error because `ArrayList` is not a subtype of `Vector`. The type of `v1` and of the allocation site labeled `A2` cannot be changed to `ArrayList` for similar reasons.

Next, we turn our attention to the migration of allocation sites. Suppose that allocation site `A3` is changed to "`new ArrayList()`". Objects allocated at `A3` are eventually passed to the constructor of `MyActionListener`, a listener that is passed to a `JComboBox` object allocated on line (39). As `JComboBox` occurs in an external library, we conservatively assume that the listener may be used in a separate thread and hence preserve thread-safety by rewriting `A3` to "`Collections.synchronizedList(new ArrayList)`". The assignment to `v4` on line (17) requires that we also change `v4`'s type to `List` because a type-error occurs otherwise. Using similar arguments, it can be seen that `runWizard()`'s return type and `v2`'s type must be changed to `List` as well.

Updating the types of variables may necessitate the transformation of method calls. For example, changing `v4`'s type to `ArrayList` requires that we transform the calls `v4.addElement(···)` on line (20) and `v4.elements()` on line (24) because these operations are not supported by `ArrayList`. Using the specification of Figure 2, we transform the former into `v4.add(···)` and the latter into `v4.iterator()`. Moreover, changing `v2`'s type requires transforming the call `v2.copyInto(dbData)` on line (13) into `Util.copyInto(v2, dbData)`. Note that the transformation of expression `v4.elements()` on line (24) necessitates changing the type of the formal parameter `elements` of `printElements()` from `Enumeration` to `Iterator`. This, in turn, entails migrating `elements.hasMoreElements()` on line (28) and `elements.nextElement()` on line (29), and all callers of `printElements()` must be updated accordingly, which involves further changes on lines (3) and (32)–(35).

Figure 1(b) shows the refactored program, using underlining to highlight transformed program fragments. It is evident from the above discussion that migrating references of legacy classes is both labor-intensive (many declarations, calls, and allocation sites are affected) and error-prone (not all uses of legacy classes can be migrated, and dependencies between migrations exist). The limitations on class library migration can be categorized as follows:

**interaction with external libraries.** If an application calls a method of an external library, opportunities for migration may be limited if the method signature refers to a concrete class such as `Vector`.

**type correctness requirements due to assignments.** Program constructs such as assignments require that certain subtype-relationships hold between the types of variables and expressions. As a result, migrating one variable often requires migrating others.

**dependencies between migrations.** If a method in one legacy class refers to another, one migration may depend on another.

**thread safety.** A major difference between legacy collection classes and their replacements is the fact that the latter do do not perform automatic synchronization on every method call. If thread-safety is required, synchronization wrappers must be inserted.

# 3. THE FRAMEWORK

We now present the formal framework for expressing migrations between classes. In what follows, we will use $P$ and $P'$ to refer to the original program, and to the refactored program, respectively. We will first make precise the concept of a migration between classes. Then, we present the type constraints model.

## 3.1 Migrations

Fix two classes, $T$ and $T'$. We say that a mapping $m$ over fragments of Abstract Syntax Trees (AST) is a *migration* from $T$ to $T'$ under the following conditions:

- $m$ maps allocation expressions of type $T$, i.e. expressions of the form **new** $T(\cdots)$, into allocation expressions of type $T'$.
- $m$ maps method call expressions on methods of $T$ to expressions not written in terms of $T$.

Given a migration from $T$ to $T'$, we refer to $T$ and to $T'$ as a *legacy class*, and a *replacement class*, respectively. An expression $E$ is said to be a *migration candidate* if it is either an allocation expression of type $T$, or if it is a call to a method of $T$.

The input to the framework, in addition to the program being refactored, is a set of migrations, each of which is specified by a set of rules. The example in Figure 2 consists of three migrations: From `Vector` to `ArrayList`, from

[5]The use of *concrete* container classes such as `Hashtable` (as opposed to *abstract* container classes such as `Map`) in the signature of public methods is often an indication of poor design, because it unnecessarily exposes implementation details. Nonetheless, this practice is pervasive. For example, we counted 215 public methods in the JDK 1.4.2 standard libraries whose signature refers to `Hashtable` or `Vector`.

| | |
|---|---|
| $M, M'$ | methods (signature, return type, and a reference to the declaring type are assumed to be available) |
| $F, F'$ | fields (name, type, and declaring type are assumed to be available) |
| $C, C'$ | classes |
| $I, I'$ | interfaces |
| $T, T'$ | types (in this paper, a type is either a class or an interface) |
| $E, E', E_1, E_2, \cdots$ | expressions (at a specific point in the program, corresponding to a specific node in the program's AST) |
| $[E]$ | the type of expression or declaration element $E$ |
| $[M]$ | the declared return type of method $M$ |
| $[F]$ | the declared type of field $F$ |
| $Decl(M)$ | the type that declares method $M$ |
| $Decl(F)$ | the type that declares field $F$ |
| $Param(M, i)$ | the $i$-th formal parameter of method $M$ |
| $T' \leq T$ | $T'$ is equal to $T$, or $T'$ is a subtype of $T$ |
| $RootDefs(M)$ | $\{ M' \mid M$ overrides $M'$, and there exists no $M''(M'' \neq M')$ such that $M'$ overrides $M'' \}$ |

**Figure 4: Notation used for defining type constraints.**

| | |
|---|---|
| $\alpha = \alpha'$ | type $\alpha$ must be the same as type $\alpha'$ |
| $\alpha \leq \alpha'$ | type $\alpha$ must be the same as, or a subtype of type $\alpha'$ |
| $\alpha \leq \alpha_1$ **or** $\cdots$ **or** $\alpha \leq \alpha_k$ | $\alpha \leq \alpha_i$ must hold for at least one $i$, $(1 \leq i \leq k)$ |
| $\alpha \not\leq \alpha'$ | type $\alpha$ must not be a subtype of type $\alpha'$ |
| $(\alpha = \alpha') \rightarrow c$ | Constraint $c$ must hold if constraint $\alpha = \alpha'$ holds |

**Figure 5: Syntax of type constraints. Here, constraint variables $\alpha$, $\alpha'$, ... represent the types associated with program constructs and must be of one of the following forms: (i) a type constant $T$, (ii) the type of an expression $[E]$, (iii) the type declaring a method $Decl(M)$, or (iv) the type declaring a field $Decl(F)$.**

`Hashtable` to `HashMap`, and from `Enumeration` to `Iterator`. Note that while Figure 2 does not specify how to migrate all methods declared in these classes, it *can* be used with any application. In general, the use of a partial specification has the disadvantage that the framework may be able to migrate smaller portions of a program than with a more complete one. The grammar of the specification language, as well as an example of a complete migration from `Vector` to `ArrayList`, can be found in Appendix A.

## 3.2 Type Constraints

Given a set of migrations $\mathcal{M}$ and a program $P$, a system of *type constraints* [25] is generated. Type constraints are a formalism for expressing relationships between the types of declarations and expressions that has traditionally been used for type-checking, type inference, and more recently for refactoring [31, 30, 14]. We extend the type constraints framework of [31] to determine where declarations and allocation sites can be transformed, as per migrations in $\mathcal{M}$, without affecting program behavior. Type constraints are generated from the AST of a program in a syntax-directed manner, and encode relationships between the types of declarations and expressions that must be satisfied in order to preserve type correctness or program behavior. Figure 4 shows the notation used to formulate type constraints. Figure 5 shows the syntax of type constraints. The type constraints used in this paper can be categorized as follows:

1. Constraints that express relationships between types of declarations and expressions that must be preserved in order to preserve type-correctness. These constraints are generated by rules (1)–(24) in Figure 6, and are largely the same as in [30, 31], the main difference being special treatment for legacy classes. Section 3.2.1 presents some of these constraints in more detail.
2. Constraints that express relationships between the types of declarations and expressions that must be preserved in order to preserve a program's run-time behavior. These constraints are generated by rules (25)–(29) in Figure 6, and are discussed in Section 3.2.2.
3. Constraints that preserve type-correctness of migration candidates. These are shown in Figure 7, and will be discussed in Section 3.2.4.

### *3.2.1 Preserving Type Correctness*

Rules (1)–(24) in Figure 6 generate type constraints for preserving type-correctness. Each rule generates, for a given program construct, one or more type constraints that express the subtype-relationships that must exist between the declared types of the construct's constituent expressions, in order for that program construct to be type-correct. By definition, a program is *type-correct* if the type constraints for all its constructs are satisfied. We will now study a few of the constraint generation rules in detail, referring the reader to [30, 31] for further details.

For example, rule (1) states that an assignment $E_1 = E_2$ is type correct if the type of $E_2$ is the same as or a subtype of the type of $E_1$. For a field-access expression $E.f$, rule (2) defines the type of this expression to be the same as the declared type of $F$ and rule (3) requires that the type of expression $E$ be a subtype of the type in which $F$ is declared.

Rules (4)–(6) are concerned with a virtual method call $E.m(E_1, \cdots, E_k)$ that refers to a method $M$. Rule (4) defines the type of the call-expression to be the same as $M$'s return type. Further, the type of each actual parameter $E_i$ must be the same as or a subtype of the type of the corresponding formal parameter $Param(M, i)$ (rule (5)). Rule (6) states that

| program construct(s)/analysis fact(s) | implied type constraint(s) | |
|---|---|---|
| assignment $E_1 = E_2$ | $[E_2] \leq [E_1]$ | (1) |
| access $E.f$ to field $F$ | $[E.f] = [F]$<br>$[E] \leq \mathit{Decl}(F)$ | (2)<br>(3) |
| method call<br>$E.m(E_1, \cdots, E_n)$<br>to a virtual method $M$<br>$[E]_P$ is not a migration candidate | $[E.m(E_1, \cdots, E_n)] = [M]$<br>$[E_i] \leq [\mathit{Param}(M, i)]$<br><br>$[E] \leq \mathit{Decl}(M_1)$ **or** $\cdots$ **or** $[E] \leq \mathit{Decl}(M_k)$<br>where $\mathit{RootDefs}(M) = \{ M_1, \cdots, M_k \}$ | (4)<br>(5)<br><br>(6) |
| `return` $E$ in method $M$ | $[E] \leq [M]$ | (7) |
| Constructor call `new` $C(E_1, \cdots, E_n)$ to constructor $M$<br>$[\texttt{new}\ C(E_1, \cdots, E_n)]_P = C$<br>Class $C$ is not a legacy class | $[E_i] \leq [\mathit{Param}(M, i)]$ | (8) |
| direct call<br>$E.m(E_1, \cdots, E_n)$<br>to method $M$ | $[E.m(E_1, \cdots, E_n)] = [M]$<br>$[E_i] \leq [\mathit{Param}(M, i)]$<br>$[E] \leq \mathit{Decl}(M)$ | (9)<br>(10)<br>(11) |
| down-cast<br>$(C)E$ | $[(C)E] \leq [E]$<br>if $[E]$ is a class | (12) |
| for every type $T$ | $T \leq \texttt{java.lang.Object}$<br>$[\texttt{null}] \leq \texttt{T}$ | (13)<br>(14) |
| implicit declaration of `this` in method $M$ | $[\texttt{this}] = \mathit{Decl}(M)$ | (15) |
| declaration of method $M$ (declared in type $T$) | $\mathit{Decl}(M) = T$ | (16) |
| declaration of field $F$ (declared in type $T$) | $\mathit{Decl}(F) = T$ | (17) |
| explicit declaration of $i$'th parameter $v$ of method or constructor $M$ | $\mathit{Param}(M, i) = [v]$ | (18) |
| down-cast expression $(T)E$<br>$T$ is a legacy type | $[(T)E] \leq T^{\top}$<br>$T_{\perp} \leq [(T)E]$ | (19)<br>(20) |
| down-cast expression $(T)E$<br>$T$ is not legacy type | $[(T)E] = T$ | (21) |
| expression `new` $C(E_1, \cdots, E_n)$<br>$C$ is a legacy class with replacement $C'$ | $[\texttt{new}\ C(E_1, \cdots, E_n)] \leq C^{\top}$<br>$C_{\perp} \leq [\texttt{new}\ C(E_1, \cdots, E_n)]$ | (22)<br>(23) |
| expression `new` $C(E_1, \cdots, E_n)$<br>$C$ is not a legacy class | $[\texttt{new}\ C(E_1, \cdots, E_n)] = C$ | (24) |
| $M'$ overrides $M$,<br>$M' \neq M$ | $[\mathit{Param}(M', i)] = [\mathit{Param}(M, i)]$<br>$[M'] = [M]$ | (25)<br>(26) |
| actual parameter $E$ in call to method in external library,<br>$[E]_P = T$ | $[E] = T$ | (27) |
| for each down-cast expression $(C)E$, and each allocation expression $E' \in \mathit{PointsTo}(P, E)$ such that $[E']_P \leq [(C)E]_P$ | $[E'] \leq [(C)E]$ | (28) |
| for each down-cast expression $(C)E$, and each allocation expression $E' \in \mathit{PointsTo}(P, E)$ such that $[E']_P \not\leq [(C)E]_P$ | $[E'] \not\leq [(C)E]$ | (29) |

**Figure 6: Type constraints for a set of core Java language features.**

| program construct(s)/analysis fact(s) | implied type constraint(s) | |
|---|---|---|
| Expression $E$ of the form **new** $C(\cdots)$<br>$[E]_P = C$<br>$C$ is a legacy class with replacement $C'$<br>$E'$ is a migration of $E$ | $([E] = C) \rightarrow c$<br>$([E] = C') \rightarrow c'$<br>for every $c \in \mathit{consts}(E)$, $c' \in \mathit{consts}(E')$ | (30)<br>(31) |
| Expression $E$ of the form $E_1.m(\cdots)$<br>$[E_1]_P = T$<br>$T$ is a legacy type with replacement $T'$<br>$E'$ is a migration of $E$ | $([E] = T) \rightarrow c$<br>$([E] = T') \rightarrow c'$<br>for every $c \in \mathit{consts}(E)$, $c' \in \mathit{consts}(E')$ | (32)<br>(33) |

**Figure 7: Type constraints for migration candidates.**

a declaration of a method with the same signature as $M$ must occur in a supertype of the type of $E$. The complexity in this rule stems from the fact that $M$ may override one or more methods $M_1, \cdots, M_k$ that are declared in supertypes $T_1, \cdots, T_k$ of the type $Decl(M)$. Here, the type-correctness of the method call only requires that the type of receiver expression $E$ is a subtype of one of these $T_i$. This is expressed by way of a disjunction in rule (6) using auxiliary function *RootDefs* of Figure 4.

Rules (22) and (23) are concerned with allocation expressions of legacy classes. The need for rules specific to legacy classes is illustrated as follows: In [31], the following constraint is generated for an allocation expression of the form `new` $C(E_1, \cdots, E_n)$:

$$[\texttt{new}\ C(E_1, \cdots, E_n)] = C$$

stating that the type of the expression must be $C$. For non-legacy classes, this rule is still needed (rule (24)). However, in the case of legacy classes such as `Vector`, it is too restrictive because we want to consider the alternative solution in which the expression is changed to allocate an `ArrayList` instead of a `Vector`. To this end, we augment the class hierarchy with auxiliary classes as well as subtype relationships. Let $T$ and $T'$ be a legacy and a replacement, class, respectively. As in [30], we define the auxiliary classes $T^\top$ and $T_\perp$ where $T^\top$ is defined to be a supertype of both $T$ and $T'$, and $T_\perp$ is defined to be a subtype of both $T$ and $T'$. For example, in migrating class `Vector` to `ArrayList`, the type $\texttt{Vector}^\top$ is introduced as a common supertype of both classes, and $\texttt{Vector}_\perp$ as a common subtype. Now, rules (22) and (23) allow the type of an allocation expression to be either `Vector` or `ArrayList`. (note that the presence of other constraints may still restrict the expression's type to `Vector`). It should be noted that $T^\top$ and $T_\perp$ are only used during constraint solving; the refactored program will not refer to these auxiliary classes. The rules for down-cast expressions have been adapted from [31] similarly.

### 3.2.2 Preserving Run-Time Behavior

The constraints discussed so far are only concerned with type-correctness. Additional constraints are needed to ensure that program behavior is preserved. Rules (25) and (26) state that overriding relationships in the original program $P$ must be preserved in the refactored program $P'$.

Rules (28) and (29) state that the execution behavior of a down-cast $(C)E$ must be preserved. Here, the notation $PointsTo(P, E)$ refers to the set of objects (identified by their allocation sites) that an expression $E$ in program $P$ may point to, and $[(C)E]_P$ denotes the type of the down-cast expression $(C)E$ in the original program $P$. Any of several existing algorithms [19, 28] can be used to compute points-to information. Rule (28) ensures that for each $E'$ in the points-to set of $E$ for which the down-cast succeeds, the down-cast will still succeed in $P'$. Likewise, Rule (29) enforces that for each $E'$ in the points-to set of $E$ for which the down-cast fails, the down-cast will still fail in $P'$. The treatment of up-casts is completely symmetrical to that of down-casts and requires rules similar to (12), (19)–(20), and (28)–(29). We omitted these rules, but our implementation fully supports up-casts.

| Line | Constraint | Rule |
|---|---|---|
| (9) | $[\mathbf{A2}] \leq [\texttt{v1}]$, $[\mathbf{A2}] \leq \texttt{Vector}^\top$, $\texttt{Vector}_\perp \leq [\mathbf{A2}]$ | (1), (22), (23) |
| (11) | $[\texttt{v1}] \leq [Param(\texttt{runWizard(Vector)}, 1)]$ | (10) |
| (16) | $[Param(\texttt{runWizard(Vector)}, 1)] = [\texttt{v3}]$ | (18) |
| (49) | $[\texttt{v3}] \leq [Param(\texttt{showWizardDialog(Vector)}, 1)]$ | (10) |
| (37) | $[Param(\texttt{showWizardDialog(Vector)}, 1)] = [\texttt{v6}]$ | (18) |
| (38) | $[\texttt{v6}] \leq [Param(\texttt{JComboBox(Vector)}, 1)]$ | (8) |
| n/a | $[Param(\texttt{JComboBox(Vector)}, 1)] = \texttt{Vector}$ | (27) |
| (17) | $[\mathbf{A3}] \leq [\texttt{v4}]$, $[\mathbf{A3}] \leq \texttt{Vector}^\top$, $\texttt{Vector}_\perp \leq [\mathbf{A3}]$ | (1), (22), (23) |

**Figure 8: Some of the generated type constraints.**

External libraries that cannot be modified raise several issues. First, rule (27) states that the types of actual parameters in calls to methods in external libraries cannot be changed. Second, since source code is unavailable, we must make conservative assumptions about casts that may be performed inside external libraries, and also for computing points-to information. Rule (27) satisfies these assumptions by effectively fixing the type of any allocation site that may flow to an external library.

### 3.2.3 Example

Figure 8 shows some of the basic constraints generated for the example of Figure 1 using the rules of Figure 6. This set of constraints, although not complete, is sufficient to demonstrate the migration possibilities and limitations of allocation sites `A2` and `A3`. Note that `A3` is only constrained to be a subtype of $\texttt{Vector}^\top$, and a supertype of $\texttt{Vector}_\perp$, and can therefore be migrated. However, `A2` cannot be migrated due to a call to a constructor of the external class `JComboBox`. Rule (27) fixes the parameter of this constructor to be `Vector`, and the other constraints effectively propagate this fact to `A2`, whose type accordingly must remain `Vector`.

### 3.2.4 Implication Constraints

Since a migration is a transformation over expressions, a migration candidate and its transformed counterpart often generate distinct sets of constraints. To illustrate this, consider the migration in Figure 2 and the expression `v2.copyInto(dbData)` on line (13) of Figure 1. The transformed method call is `Util.copyInto(v2, dbData)`. Here, the original expression generates (by rules (5) and (6)) the constraints

$$[\texttt{v2}] \leq \texttt{Vector}$$
$$[\texttt{dbData}] \leq [Param(\texttt{Vector.copyInto}, 1)]$$

because `v2` is a receiver of the method call, and `dbData` is an actual parameter. However, in the transformed expression `v2` and `dbData` are both actual parameters. Hence the transformed expression generates (by rule (10)) the constraints

$$[\texttt{v2}] \leq [Param(\texttt{Util.copyInto}, 1)]$$
$$[\texttt{dbData}] \leq [Param(\texttt{Util.copyInto}, 2)]$$

Therefore, depending on the type assigned to certain constraint variables (in this case the variable $[\texttt{v2}]$), different sets of constraints need to be considered by a solving algorithm. To express this, we use *implication* constraints, rules for which are given in Figure 7.

```
 1 : repeat until fixpoint
 2 : ⌈ repeat until fixpoint
 3 : | ⌈ for every D, E such that (D≤E) ∈ C
 4 : | |     S_D := S_D − {T | ∀T' ∈ S_E : T≰T'}
 5 : | |     S_E := S_E − {T' | ∀T ∈ S_D : T≰T'}
 6 : | |   for every D, E such that (D≰E) ∈ C
 7 : | |     S_D := S_D − {T | ∀T' ∈ S_E : T≤T'}
 8 : | ⌊     S_E := S_E − {T' | ∀T ∈ S_D : T≤T'}
 9 : |   if there exists D such that |S_D| > 1
10 : ⌊     S_D := {choose(S_D)}
```

**Figure 9: Algorithm** Solve-Simple **for a set $\mathcal{C}$ of simple constraints over $\leq, \not\leq$**

```
1 : Invoke Solve-Simple
2 : while I ≠ ∅
3 : ⌈ Let c ∈ I be a constraint of the form ([E] = T) → ψ
4 : |   I := I − {c}
5 : |   If T ∈ S_E then
    |   ⌈ (S_E, C) := (S_E ∩ {T}, C ∪ {ψ})
6 : |   |     or
    |   ⌊ (S_E, C) := (S_E − {T}, C)
7 : |   Invoke Solve-Simple
8 : |   If ∃E.S_E = ∅ then
9 : ⌊     backtrack
```

**Figure 10: Algorithm** Solve-Complete **for the sets $(\mathcal{I}, \mathcal{C})$ of implication and simple constraints. The notation $[V := A$ or $V := B]$ denotes a nondeterministic assignment.**

In rules (30)–(33), $E$ denotes a migration candidate, and $E'$ denotes a transformed form of $E$. The notation $consts(E)$ (respectively $consts(E')$) denotes the set of constraints generated by subexpressions of $E$ (respectively $E'$) according to the rules in Figure 6. Rules (30)–(31) apply to legacy allocation sites, while rules (32)–(33) apply to method calls in which the type of the receiver is a legacy class.

For example, let $E$ be the expression `v2.copyInto(dbData)`. Then applying rules (32) and (33), we get the following constraints for $E$:

$$([E] = \texttt{Vector}) \rightarrow [\texttt{v2}] \leq \texttt{Vector}$$
$$([E] = \texttt{Vector}) \rightarrow [\texttt{dbData}] \leq [Param(\texttt{Vector.copyInto}, 1)]$$
$$([E] = \texttt{ArrayList}) \rightarrow [\texttt{v2}] \leq [Param(\texttt{Util.copyInto}, 1)]$$
$$([E] = \texttt{ArrayList}) \rightarrow [\texttt{dbData}] \leq [Param(\texttt{Util.copyInto}, 2)]$$

# 4. SOLVING CONSTRAINTS

We will present the algorithm for solving systems of type constraints in two parts. First, an algorithm for solving basic constraints is presented. This is then embedded in a solver that accommodates implications. Our goal is to find a solution to the constraints that maximizes the number of migration candidates that are transformed to use replacement classes.

## 4.1 Solving Simple Constraints

An initial step in solving a system of simple constraints is to simplify all disjunction constraints into equivalent or stronger constraints over the relation $\leq$. Disjunctions are generated by rule (6) for dealing with virtual method calls. We first remove any disjunction that trivially follows from some other constraint in the system. Then, any remaining disjunction is strengthened to a $\leq$-constraint by replacing it with one of its disjuncts. If we assume that none of the disjunctions is derived from a migrated call site[6], it is guaranteed that a solution to the constraint system will still exist after strengthening the disjunctions, because the original program satisfies all disjuncts of all disjunctions. The manner in which a disjunct is chosen to replace the constraint is the same as in [30].

To solve a system of simple constraints over relations $\leq$ and $\not\leq$, we partition the set of constraint variables into equivalence classes according to equality constraints. That is, any solution to the system must assign an identical type to all members of a single equivalence class. The relations $\leq$ and $\not\leq$ are extended to equivalence classes in the obvious way. Then the set of possible types for each class is computed using an optimistic algorithm. The algorithm associates with each equivalence class $E$ a set $S_E$ of types, initialized as follows: (i) if $E$ contains an allocation expression **new** $C(\cdots)$, then $S_E = \{C\} \cup \{C' \mid C$ is a legacy class with replacement $C'\}$; otherwise (ii) $S_E$ contains all types except $T^{\top}$ and $T_{\perp}$, for all $T$.

The algorithm Solve-Simple in Figure 9 solves a system of simple constraints by iterating until a fixpoint is reached, in which no set $S_E$ contains more than one type. The algorithm consists of a *pruning* phase (lines 2-8), where invalid types are pruned from sets, and a *heuristic* phase (lines 9-10) where a non-singleton set is reduced to a single member. The heuristic phase is defined in terms of some heuristic `choose` that, given a set of classes, selects a *most preferred* class. In our implementation it greedily prefers replacement classes over legacy classes, though it does not consistently prefer any one replacement class over another. Space limitations prevent us from presenting a proof that Solve-Simple always computes a solution, and that the heuristic phase does not eliminate all solutions from the search space. Termination follows from the fact that every loop body performs a strictly monotone step, reducing the size of at least one set.

## 4.2 Solving Implication Constraints

We now consider a general constraint system consisting of implications as well as simple constraints. Let $\mathcal{I}$ be a set of implication constraints of the form $([E] = T) \rightarrow \psi$ and $\mathcal{C}$ a set of simple constraints. The algorithm Solve-Complete for solving $(\mathcal{I}, \mathcal{C})$ is given in Figure 10. Its basic step is to select an implication constraint $([E] = T) \rightarrow \psi$, and to nondeterministically satisfy or violate its left side by removing types from the set $S_E$. Subsequently, the simple solver is applied with a possibly larger set of simple constraints. The requirement on the nondeterministic choice, expressed in line (8), is that it does not eliminate all solutions. If such a state is reached then the algorithm backtracks to the last nondeterministic choice point at which not all choices have been exhausted, and makes a different choice. Such a point is guaranteed to exist, since the original program is a valid solution to the constraint system. In practice the backtracking mechanism is coupled with a policy for directing nondeterministic choice. For example, given an implication $([E] = T) \rightarrow \psi$, the implemented policy is to attempt to satisfy $([E] = T)$ if $T$ is a replacement class.

[6]This assumption is satisfied if for any method $M$ of a replacement class, we require that $|RootDefs(M)| = 1$.

The running time of SOLVE-COMPLETE is worst-case exponential in the number of call sites. However, such behavior is rare, since implication constraints are rarely independent of one another (i.e., multiple implications tend to have identical left hand sides). More importantly, due to the choice-directing policy, worst case behavior is only exhibited in programs where a majority of call sites cannot be migrated. In programs encountered in our evaluation this has not been the case.

## 4.3 Preserving Thread-Safety

Suppose that we migrate from a class $T$ to a class $T'$, where $T$ is designed to be thread-safe by only allowing `synchronized` access to its internal state, but where $T'$ is not. Unless countermeasures are taken, the refactored program may exhibit different behavior if it uses multiple threads whose executions are interleaved in ways that could not arise in the original program. The migration from `Vector` to `ArrayList` is an example of this case[7], because all of `Vector`'s public methods are `synchronized`, thus preventing multiple threads from concurrently accessing the same object, but none of the methods in class `ArrayList` are. The JDK library designers deliberately omitted synchronization from class `ArrayList` so as to avoid unnecessary overhead in clients that do not access a list concurrently. To handle cases where thread-safety is required, the Java collection classes provide a mechanism known as *synchronization wrappers*: An object of type `ArrayList` can be made thread-safe by "wrapping" its allocation site with a call to the static method `Collections.synchronizedList(List)`[8]. Hence, instead of writing:

```
new ArrayList()
```

one would write[9]:

```
Collections.synchronizedList(new ArrayList())
```

We support migrations from classes that are thread-safe to classes that are not by introducing synchronization wrappers. However, synchronization wrappers are only introduced for objects that may be accessed by multiple threads. To this end, we employ a simple escape analysis (see, e.g., [6, 1, 27]) to compute a conservative approximation of legacy allocation sites that may escape their allocating thread. If the analysis indicates that a legacy allocation site may escape, a synchronization wrapper is introduced. Otherwise, the allocation site is left unwrapped.

Figure 11 shows the algorithm used to determine allocation sites that may escape their allocating thread. It uses the notations $\mathsf{SF}(P)$ and $\mathsf{fields}(\mathcal{S})$ to denote, respectively, the set of all static field declarations in program $P$, and the set of all fields declared by types in the set $\mathcal{S}$. It uses the notation $\mathit{PointsTo}(P, E)$, discussed in Section 3.2.2, to denote the set of objects that an expression $E$ may point to. Additionally, for every allocation site $A$, $[A]$ denotes its type.

[7]Migration from `Hashtable` to `HashMap` is another example of migrating from a class that is thread-safe to a class that is not.

[8]Class `Collections` provides similar synchronization wrappers for `Collections`, `Sets`, and `Maps`.

[9]This can result in different behavior in the presence of race conditions, due to the use of iterators that, unlike `Enumerations`, raise exceptions on detection of concurrent changes to an underlying collection. If a more strict notion of preservation is needed, one can use custom synchronization wrappers.

$1: \mathcal{S} := \{\text{class } C \mid C \leq \texttt{Thread}\} \cup$
$\quad \{\text{class } C \mid C \leq \texttt{Runnable}\} \cup$
$\quad \bigcup_{F \in \mathsf{SF}(P)} \{[A] \mid A \in \mathit{PointsTo}(P, F)\}$
$2:$ **repeat until fixpoint**
$3:$ **for each** $F \in \mathsf{fields}(\mathcal{S})$
$4: \quad \mathcal{S} := \mathcal{S} \cup \{[A] \mid A \in \mathit{PointsTo}(P, F)\}$
$5:$ **for each allocation site** $A$
$6: \quad \mathsf{escaping}(A) :=$
$\quad \text{LIB-ESCAPING}(A) \vee$
$\quad \exists F \in \mathsf{fields}(\mathcal{S}) \,.\, A \in \mathit{PointsTo}(P, F)$

**Figure 11: Escape Analysis Algorithm.** LIB-ESCAPING($A$) **is informally defined as true if** $A \in \mathit{PointsTo}(P, E)$, **where** $E$ **is an actual parameter in a call to a method in an external library.**

The rationale behind this algorithm is that static fields, subclasses of `Thread`, and implementors of the interface `Runnable`, which is often used to pass callback procedures to `Thread` objects, are the sole vehicles by which objects can escape their allocating thread. Therefore, any escaping object must be transitively referenced by one or more fields of these classes. Furthermore, since external libraries cannot partake in this analysis, the algorithm conservatively assumes that any allocation site flowing into a method of an external library class is thread-escaping. As we shall see in Section 5, this naïve algorithm has proved itself remarkably effective on the benchmark applications that we have analyzed.

# 5. EVALUATION

Our work is implemented in *Eclipse*, a widely-used open-source development environment for Java, using existing infrastructure for building refactorings [2] and type constraints [31, 14]. Class Hierarchy Analysis [7] is used to compute a call graph, followed by a variation on 0-CFA [29] to compute points-to information. Care is taken to ensure that correct points-to information is computed in the presence of reflection and external class libraries (similar to [32]).

We evaluated the refactoring tool on a number of Java applications of up to 53 KLOC that we migrated from `Vector` to `ArrayList`, from `Hashtable` to `HashMap`, and from `Enumeration` to `Iterator`. Table 1 states the essential characteristics for each benchmark program. From left to right, columns of the table indicate: (i) the benchmark name, (ii) the number of lines of source code (in thousands), (iii) the number of classes, (iv) the legacy classes used (here, $V$, $HT$, and $E$ indicate `Vector`, `Hashtable`, and `Enumeration`, respectively), (v) the number of declarations of variables, parameters, and fields that refer to legacy classes, (vi) the number of allocation sites referring to legacy classes, and (vii) the number of call sites referring to methods in legacy classes.

The results are shown in Table 2. The columns of the table show, for each benchmark: (i) the number of legacy declarations that were migrated and those that could not be migrated for reasons discussed in Section 2 (these numbers are separated by a '/' symbol), (ii) the number of legacy allocation sites that were migrated without synchronization

| benchmark | KLOC | #classes | legacy classes used | # legacy declarations | # legacy allocation sites | # legacy call sites |
|---|---|---|---|---|---|---|
| *Hanoi* | 4.0 | 41 | *V* | 3 | 3 | 26 |
| *JUnit* | 5.3 | 100 | *V*, *HT*, *E* | 62 | 24 | 118 |
| *JLex* | 7.9 | 26 | *V*, *HT*, *E* | 39 | 16 | 185 |
| *JavaCup* | 10.6 | 36 | *HT*, *E* | 56 | 14 | 153 |
| *Cassowary* | 12.2 | 68 | *V*, *HT*, *E* | 139 | 46 | 728 |
| *Azureus* | 13.9 | 160 | *V* | 13 | 12 | 51 |
| *HTML Parser* | 17.1 | 115 | *V*, *HT* | 144 | 23 | 467 |
| *JBidWatcher* | 22.9 | 154 | *V*, *HT* | 71 | 36 | 294 |
| *SpecJBB* | 31.3 | 110 | *V*, *HT*, *E* | 28 | 15 | 88 |
| *Jax* | 53.1 | 309 | *V*, *HT*, *E* | 251 | 94 | 706 |

**Table 1: Characteristics of the Java applications used to evaluate our techniques. From left to right, the columns of the table show: (i) the name of the benchmark, (ii) the number of lines of source code (in thousands), (iii) the number of classes in the benchmark, (iv) the legacy classes used in this benchmark (here, *V* denotes `Vector`, *HT* denotes `Hashtable`, and *E* denotes `Enumeration`), (v) the number of declarations that refer to these legacy classes, (vi) the number of allocation sites that refer to these legacy classes, and (vii) the number of call sites that refer to these legacy classes.**

| benchmark | declarations (migr./unchanged) | allocation sites (migr.-desync/migrated/unchanged) | call sites (migr./unchanged) | time (min:sec) |
|---|---|---|---|---|
| *Hanoi* | 3/0 | 3/0/0 | 26/0 | 2:07 |
| *JUnit* | 55/7 | 23/1/0 | 111/7 | 8:29 |
| *JLex* | 29/10 | 12/0/4 | 167/18 | 9:38 |
| *JavaCup* | 56/0 | 14/0/0 | 153/0 | 16:56 |
| *Cassowary* | 121/18 | 44/0/2 | 692/36 | 63:45 |
| *Azureus* | 13/0 | 6/6/0 | 51/0 | 7:29 |
| *HTML Parser* | 141/3 | 21/0/2 | 461/6 | 27:35 |
| *JBidWatcher* | 67/4 | 32/1/3 | 291/3 | 32:17 |
| *SpecJBB* | 22/6 | 13/0/2 | 78/10 | 9:36 |
| *Jax* | 208/43 | 81/0/12 | 706/0 | 125:05 |

**Table 2: Results of applying our refactoring tool to the benchmarks. For each benchmark, the table shows: (i) the number of declarations that could be migrated and the number of declarations that had to be left unchanged, (ii) the number of allocation sites that could be migrated and desynchronized, the number of allocation sites that could be migrated but not desynchronized, and the number of allocation sites that could not be migrated, and (iii) the number of call sites that could be migrated and that could not be migrated. (iv) the time required to process the benchmark (minutes:seconds).**

wrappers, the number of legacy allocation sites that were migrated with synchronization wrappers, and the number of legacy allocation sites that could not be migrated (using '/' symbols to separate these numbers), and (iii) the number of legacy call sites that were migrated, and the number of legacy call sites that could not be migrated (again, using '/' for separation), and (viii) the time required to process the benchmark. For example, for the *Cassowary* benchmark we found that: (a) 121 of the original 139 legacy class declarations were migrated, but 18 could not be migrated, (b) 44 of the 46 legacy allocation sites could be migrated without inserting synchronization wrappers, and the remaining 2 legacy allocation sites could not be migrated at all, and (c) 692 of the 728 legacy call sites could be migrated, and the remaining 36 call sites could not be migrated. Processing times are currently quite slow—processing *Jax* takes about 2 hours, but we have not done performance tuning yet, and many optimizations have yet to be implemented. In addition, our implementation does not yet make use of the new optimized type constraint infrastructure in Eclipse.

Figure 12 shows a chart that visualizes the percentage of legacy declarations and legacy call sites that were successfully migrated by our tool. From this chart, it can be seen that, on average, 90.4% of legacy declarations and 96.6% of legacy call sites were migrated successfully. Figure 13 shows a chart visualizing the effectiveness of the tool at migrating allocation sites. In this chart, the bottom part of each bar shows the percentage of allocation sites that were migrated without the insertion of synchronization wrappers, and the top part of each bar shows the percentage of allocation sites that could be migrated but that required the insertion of synchronization wrappers. As can be seen in Figure 13, an average of 92.0% of all allocation sites can be migrated (82.8% without the insertion of synchronization wrappers and 9.1% with the insertion of synchronization wrappers). We now discuss a few cases out of the experiments illustrating a number of nontrivial aspects of migration.

**Nontrivial rewriting.** *JBidWatcher*, *JUnit*, and *SpecJBB* contain calls to the previously discussed method `Vector.copyInto()` which requires nontrivial rewriting and introduction of an auxiliary class. In *JBidWatcher*, *JUnit*, *SpecJBB* and *Jax*, the percentages of migrated call sites for which the method's name or signature was changed are 30%, 75%, 73%, and 47%, respectively. Clearly, manual migration of these applications would involve a significant amount of error-prone editing work.

**Interaction with external libraries.** In *JUnit*, one of the `Vectors` is eventually passed to the constructor of the exter-

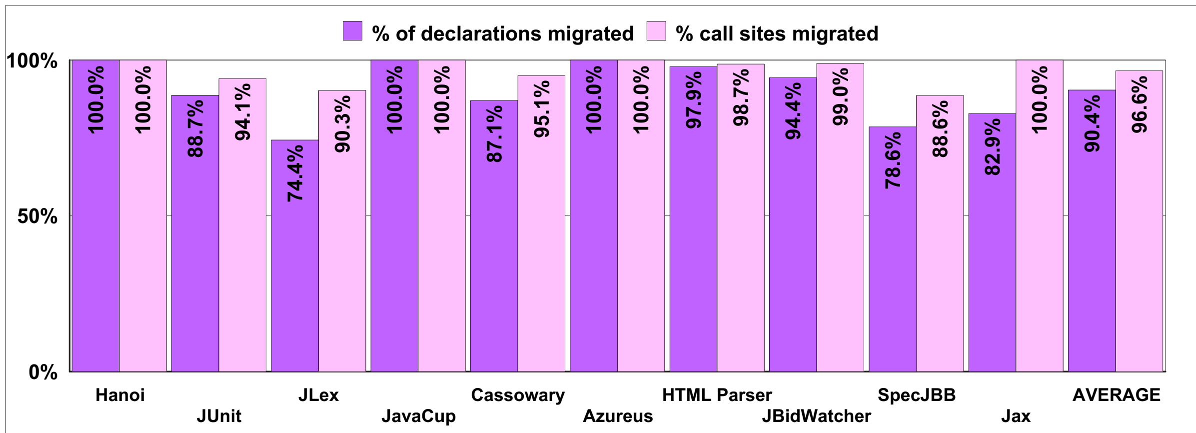


**Figure 12: Percentage of declarations and call sites that can be migrated.**

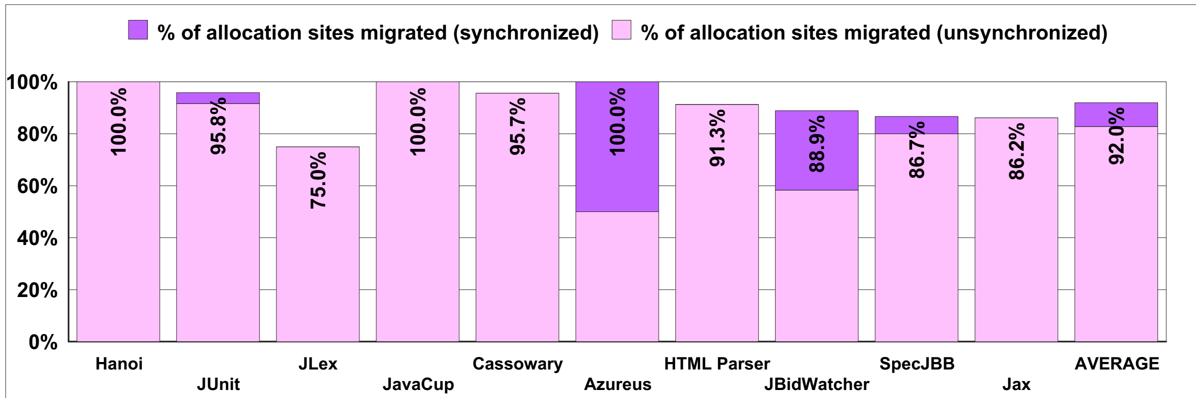


**Figure 13: Percentage of allocation sites can be migrated. The bottom part of each bar indicates the percentage of allocation sites that can be migrated without inserting synchronization wrappers, and the top part depicts the additional allocation sites that can be migrated with the insertion of synchronization wrappers.**

nal *Swing* library class `JList`, whose formal parameter is of type `Vector`. This flow of objects is not immediately evident from the code, as the allocated `Vector` is assigned to a variable that is elsewhere passed to the constructor. Similar cases occur in *SpecJBB* where various `Vectors` are not migrated because they are stored in other `Vectors`. With more insight into the implementation of `Vector`, it is evident that concrete types of its elements are irrelevant, which could in principle be utilized by a more precise analysis[10].

**Synchronization preservation.** The migration of *JUnit* includes a synchronization-wrapped allocation site. It is detected as escaping since it is assigned to a field whose declaring class declares a `Runnable` that references the field. The `Runnable` object is passed to *Swing*, which would cause any escape analysis without access to the *Swing* code to declare it as escaping. In *Azureus*, escape analysis reports that synchronization wrappers need to be introduced for certain `ArrayLists`, but the program already performs explicit synchronizations. Hence, in principle, a more precise escape analysis could enable migration without synchronization wrappers in *Azureus*.

[10] Such information could be provided in the form of *stub* implementations that approximate the behavior of selected library methods.

## 6. RELATED WORK

Several categories of related work can be distinguished, as will be discussed below.

### 6.1 Refactoring

The field of refactoring and related semantics-preserving program transformations was pioneered in the early 90s by Opdyke and Johnson [23, 24] and by Griswold [16, 17]. Opdyke's Ph.D. thesis [23, page 27–28] catalogs a number of refactorings, and informally specifies a number of invariants that any refactoring must respect in order to preserve program behavior. For example, the *Type-Safe Assignments* invariant states that "The type of each expression assigned to a variable must be an instance of the variable's defined type, or an instance of one of its subtypes. This applies both to assignment statements and function calls". The same constraints (encoded by rules (1), (5), (8), and (10)) are used in the present paper. The refactorings proposed by Opdyke et al. were implemented in the Smalltalk Refactoring Browser [26], which was the first tool to provide automated support for refactoring.

In recent years, refactoring has been popularized by the emergence of light-weight design methodologies such as "extreme programming" [3] that advocate continuous improve-

ment of a program's design. For an overview of the field, the reader is referred to books by Fowler [13] and Kerievsky [20], to a recent survey article by Mens and Tourwé [22], and to Martin Fowler's `www.refactoring.com` site. Popular development environments such as eclipse (see `www.eclipse.org`) and IntelliJ IDEA (see `www.jetbrains.com/idea/`) currently offer automated support for a wide range of refactorings. However, we are not aware of any IDE that supports refactorings for migrating between functionally equivalent classes.

Dig and Johnson [8] conducted an empirical study on the role of refactorings in API migration. In their work, successive versions of several frameworks and libraries are examined, and the API-breaking changes are classified as refactorings or as other changes. In each of these frameworks, over 80% of the API-breaking changes are refactorings, thus providing evidence that automated tool support for API migration is highly desirable. Such tool support is provided by Henkel and Diwan [18], who developed a tool that records how refactorings are used to evolve a library. Then, applications that use the library under consideration can be refactored accordingly by applying the recorded refactorings. The work by Henkel and Diwan can be viewed as a tool for automatically deriving migration specifications from refactorings that have been applied to a library. These specifications capture a class of transformations that is more limited than the one we consider, because we consider transformations that cannot be expressed by refactorings.

## 6.2 Refactorings based on Type Constraints

Tip, Kiezun, and Bäumer [31] use a subset of the constraint generation rules presented in this paper to express the preconditions and to compute the allowable source code modifications for several refactorings related to generalization, including Extract Interface, Pull Up Members, Generalize Type, and Use Supertype Where Possible. Specifically, the refactorings in [31] restructure the type hierarchy and change declarations (but not allocation sites). This work was implemented in Eclipse, and our implementation reuses key parts of this infrastructure.

Donovan et al. [11] and Fuhrer et al. [14] present refactorings that convert Java applications to use Java 1.5 generics [4]. In this work, the assumption is that an application uses a class library (e.g., the Java Standard Collections Framework) for which a generic version has become available, and type inference is used to determine concrete classes that can be used as actual type parameters to instantiate generic library classes. An important benefit of this refactoring is that the use of generic container classes often makes it unnecessary to use down-casts when retrieving elements, and [11, 14] both report that a significant number of down-casts can be removed from the analyzed benchmark applications. Similar to the work in the present paper, Fuhrer et al. extend the basic type constraints of [31], but in a way that is completely different from the rules in the present paper. The work by Donovan et al. [11] uses type constraints that are similar to those used in the present paper, but augmented with type variables and type substitutions to support the inference of generic types. Interestingly, Donovan et al. also use a form of implication constraints (referred to as guarded constraints in [11]), in order to handle situations where the generation of a constraint depends on whether or not a declaration is left raw (i.e., not parameterized). To solve these, their solver also employs a form of backtracking.

Von Dincklage and Diwan [33] present a constraint-based approach for converting non-generic Java classes to use generics and updating non-generic usages of generic classes. Von Dincklage's tool employs a suite of heuristics that resulted in the successful parameterization of several classes from the Java standard collections.

Most closely related to the work in the present paper is the work by De Sutter et al. [30], who present an optimization method in which Java applications are rewritten to use customized versions of library classes. These custom classes incorporate a number of high-level optimizations that are likely to result in reduced execution speed and heap consumption, and a combination of static analysis and profile information is used to select the optimizations to be applied. The program transformations of [30] are more limited than the ones we consider, because the methods supported by the generated custom classes are always a strict subset of those in the library classes that they replace, and a custom class always has the same superclass as the original class that it replaces. The type constraints used in [30] therefore do not need to allow for situations where auxiliary classes are needed, and hence there is no need for implication constraints (and a backtracking solver). There are several other significant differences between the work in the present paper and [30], most notably the fact that we permit non-trivial user-specified interface mappings between source and target classes, and that we allow the simultaneous migration of interdependent classes.

## 6.3 Support for Migration and Evolution

Although migration is an important theme in the maintenance of long-lived applications, limited tool support appears to exist, much of which is focused on data and interlanguage migration. In [12], type inference is used to reverse-engineer date-related types in a given Cobol program, and migrate them to Year 2000-compliant ones. Unlike the present work, the interface mapping is trivial, since the source and target types are equivalent. A tool to migrate PL/IX programs to C++ programs appears in [21]. Curiously, that work focuses exclusively on language translation, and ignores API migration.

Dmitriev [9] presents a migration tool for converting persistent objects after certain kinds of changes (e.g., replacing a single field with two fields, and the renaming of classes). In this work, changes are specified using a simple specification language, and the programmer supplies converter classes that perform the data conversion. In later work, Dmitriev [10] presents a language-specific make facility for Java, in which various types of dependences between classes are distinguished in order to avoid unnecessary recompilation.

# 7. CONCLUSIONS AND FUTURE WORK

Even small applications may contain large numbers of references to legacy classes. Migrating these involves a significant amount of error-prone editing work, so automation is highly desirable. We have presented a framework for the automatic migration of (library) classes that has been implemented in

the Eclipse environment. We evaluated the approach on a number of moderate-sized Java applications and found that, in the benchmarks we studied, over 90% of declarations, allocation sites, and call sites were migrated successfully. Moreover, a careful analysis of the results revealed that nontrivial rewriting of method names and signatures was required in many cases. The simple escape analysis we use has proved sufficient to avoid insertion of synchronization wrappers in all but a few cases. Hence, there does not appear to be a need for a more precise escape analysis such as [6].

Plans for future work include supporting the migration of user-defined subclasses of legacy classes, and the introduction of the ADAPTER design pattern [15] to wrap allocation sites and method parameters that could otherwise not be migrated. We also plan to support migrations between methods that differ in terms of thrown exceptions. This situation is similar to the one in Section 2 where the return type of a method in a legacy class (e.g., `Vector.elements()`) is another legacy class (in this case, `Enumeration`), and can be handled similarly. An additional factor, however, is the fact that determining all the `catch` clauses that need to be migrated requires interprocedural analysis, but we do not foresee any major problems here. Lastly, our refactoring tool could be made more useful by providing explanations in cases a construct could not migrated.

## Acknowledgments

We would like to thank Robert O'Callahan for suggesting the idea of refactoring `Vector`s into `ArrayList`s and Julian Dolby for suggesting an easily-implemented escape analysis algorithm. We are also grateful to Adam Kiezun and the anonymous reviewers for comments on drafts of this paper.

## 8. REFERENCES

[1] ALDRICH, J., CHAMBERS, C., SIRER, E. G., AND EGGERS, S. J. Static analyses for eliminating unnecessary synchronization from Java programs. In *Static Analysis Symposium* (1999), pp. 19–38.

[2] BÄUMER, D., GAMMA, E., AND KIEŻUN, A. Integrating refactoring support into a Java development tool. In *OOPSLA'01 Companion* (October 2001).

[3] BECK, K. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.

[4] BRACHA, G., COHEN, N., KEMPER, C., ODERSKY, M., STOUTAMIRE, D., THORUP, K., AND WADLER, P. Adding generics to the Java programming language: Public draft specification, version 2.0. Tech. rep., Java Community Process JSR-000014, June 23 2003.

[5] CHAN, P., LEE, R., AND KRAMER, D. *The Java Class Libraries Second Edition, Volume 1*. Addison-Wesley, 1998.

[6] CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. Escape analysis for Java. In *Proc. OOPSLA'99* (1999), ACM Press, pp. 1–19.

[7] DEAN, J., GROVE, D., AND CHAMBERS, C. Optimization of object-oriented programs using static class hierarchy analysis. In *Proc. ECOOP'95* (Aarhus, Denmark, Aug. 1995), W. Olthoff, Ed., Springer-Verlag, pp. 77–101.

[8] DIG, D., AND JOHNSON, R. The role of refactorings in API evolution. Tech. Rep. UIUCDCS-R-2005-2555, University of Illinois at Urbana-Champaign, 2005.

[9] DMITRIEV, M. The first experience of class evolution support in PJama. In *Proceedings of the 8th International Workshop on Persistent Object Systems (POS8)* (Tiburon, CA, 1998), pp. 279–296.

[10] DMITRIEV, M. Language-specific make technology for the Java programming language. In *Proc. of OOPSLA'02* (Seattle, WA, 2002), pp. 373–385.

[11] DONOVAN, A., KIEŻUN, A., TSCHANTZ, M. S., AND ERNST, M. D. Converting Java programs to use generic libraries. In *Proc. OOPSLA'04* (Vancouver, BC, Canada, October 26–28, 2004), pp. 15–34.

[12] EIDORFF, P. H., HENGLEIN, F., MOSSIN, C., NISS, H., SØRENSEN, M. H., AND TOFTE, M. AnnoDomini: From type theory to year 2000 conversion tool. In *Proc. POPL'99* (Austin, TX, 1999), pp. 1–14.

[13] FOWLER, M. *Refactoring. Improving the Design of Existing Code*. Addison-Wesley, 1999.

[14] FUHRER, R., TIP, F., KIEZUN, A., DOLBY, J., AND KELLER, M. Efficiently refactoring Java applications to use generic libraries. In *Proc. ECOOP'05* (Glasgow, Scotland, 2005). To appear.

[15] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[16] GRISWOLD, W. G. *Program Restructuring as an Aid to Software Maintenance*. PhD thesis, University of Washington, 1991. Technical Report 91-08-04.

[17] GRISWOLD, W. G., AND NOTKIN, D. Automated assistance for program restructuring. *ACM Trans. Softw. Eng. Methodol. 2*, 3 (1993), 228–269.

[18] HENKEL, J., AND DIWAN, A. CatchUp! Capturing and replaying refactorings to support API evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE'05)* (St. Louis, MO, May 2005).

[19] HIND, M., AND PIOLI, A. Evaluating the effectiveness of pointer alias analyses. *Science of Comp. Programming 39*, 1 (2001), 31–55.

[20] KERIEVSKY, J. *Refactoring to Patterns*. Addison-Wesley, 2004.

[21] KONTOGIANNIS, K., MARTIN, J., WONG, K., GREGORY, R., MÜLLER, H., AND MYLOPOULOS, J. Code migration through transformations: an experience report. In *Proc. CASCON'98* (1998), IBM Press, pp. 1–12.

[22] MENS, T., AND TOURWÉ, T. A survey of software refactoring. *IEEE Trans. on Software Engineering 30*, 2 (February 2004), 126–139.

[23] Opdyke, W. F. *Refactoring Object-Oriented Frameworks*. PhD thesis, University Of Illinois at Urbana-Champaign, 1992.

[24] Opdyke, W. F., and Johnson, R. E. Creating abstract superclasses by refactoring. In *The ACM 1993 Computer Science Conf. (CSC'93)* (February 1993), pp. 66–73.

[25] Palsberg, J., and Schwartzbach, M. *Object-Oriented Type Systems*. John Wiley & Sons, 1993.

[26] Roberts, D., Brant, J., and Johnson, R. E. A refactoring tool for Smalltalk. *Theory and Practice of Object Systems 3*, 4 (1997), 253–263.

[27] Ruf, E. Effective synchronization removal for Java. In *Proc. PLDI 2000* (2000), ACM Press, pp. 208–218.

[28] Ryder, B. G. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proc. CC 2003* (Warsaw, Poland, April 2003), pp. 126–137.

[29] Shivers, O. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, CMU, May 1991. CMU–CS–91–145.

[30] Sutter, B. D., Tip, F., and Dolby, J. Customization of Java library classes using type constraints and profile information. In *Proc. ECOOP'04* (Oslo, Norway, June 2004), pp. 585–609.

[31] Tip, F., Kieżun, A., and Bäumer, D. Refactoring for generalization using type constraints. In *Proc. OOPSLA'03* (Anaheim, CA, October 2003), pp. 13–26.

[32] Tip, F., Sweeney, P. F., Laffra, C., Eisma, A., and Streeter, D. Practical extraction techniques for Java. *ACM Trans. on Programming Languages and Systems 24*, 6 (2002), 625–666.

[33] von Dincklage, D., and Diwan, A. Converting Java classes to use generics. In *Proc. of OOPSLA'04* (Vancouver, BC, Canada, 2004), pp. 1–14.

# APPENDIX

## A. SPECIFICATION LANGUAGE

The language for specifying migrations is a language of rewrite rules over templates of Java expressions. A partial grammar is given in Figure 14. The left side of a rewrite rule consists of a Java allocation or method call expression, with additional annotations (e.g., return types) and *template variables*. A template variable is used for matching the left side of a rule against fragments of the program AST. The right side of a rule consists of an arbitrary Java expression template – an expression with annotations and template variables. Here, the occurrence of a variable denotes a substitution with an appropriate fragment from the program AST, as matched with the left side.

A sample specification for the synchronization-less migration from class `Vector` to class `ArrayList` is given in Figure 15. Figure 16 specifies an auxiliary migration, from `Enumeration` to `Iterator`. Figure 17 presents the auxiliary class used in Figure 15 (a partial version of this class was presented earlier in Figure 3).

| | |
|---|---|
| *Migration*: | *Rule* \| *Rule Migration* |
| *Rule*: | *Allocation* → *Expression* \| |
| | *MethodCall* → *Expression* |
| *Allocation*: | new *class-name*(*Args*), unsynchronized \| |
| | new *class-name*(*Args*), synchronized |
| *MethodCall*: | *Type class-name.id*(*Args*) \| |
| | *Type Var.id*(*Args*) |
| *Args*: | $\varepsilon$ \| *Arg* \| *Arg Args* |
| *Arg*: | *Var* \| *Expression* |
| *Var*: | *Type* : *id* \| *id* |
| *Type*: | *prim-type* \| *array-type* \| *class-name* |

**Figure 14: Migration Specification Language Grammar. The nonterminals *prim-type* and *array-type* denote any string representing a Java primitive or array type, respectively. *Expression* reduces to any construct that, after variable substitution, is a valid Java expression. $\varepsilon$ denotes the empty string.**

```
public class Util {
  public static void copyInto(List v, Object[] target) {
    if (target == null) throw new NullPointerException();
    for (int i = v.size() - 1; i >= 0; i--)
      target[i] = v.get(i);
  }
  public static void setSize(List v, int newSize) {
    while (v.size() < newSize) v.add(null);
    while (v.size() > newSize) v.remove(newSize);
  }
  public static int indexOf(List v, Object elem, int index) {
    if (elem == null) {
      for (int i = index; i < v.size(); i++)
        if (v.get(i) == null) return i;
    } else {
      for (int i = index; i < v.size(); i++)
        if (elem.equals(v.get(i))) return i;
    }
    return -1;
  }
  public static int lastIndexOf(List v, Object e, int indx) {
    if (indx >= v.size())
      throw new IndexOutOfBoundsException();
    if (e == null) {
      for (int i = indx; i >= 0; i--)
        if (v.get(i) == null) return i;
    } else {
      for (int i = indx; i >= 0; i--)
        if (e.equals(v.get(i))) return i;
    }
    return -1;
  }
}
```

**Figure 17: Auxiliary class for migrating `Vector` to `ArrayList`**

| | | |
|---|---|---|
| **new** Vector(), *unsynchronized* | → | **new** ArrayList() |
| **new** Vector(Collection:c), *unsynchronized* | → | **new** ArrayList(c) |
| **new** Vector(int:c), *unsynchronized* | → | **new** ArrayList(c) |
| void Vector:receiver.copyInto(Object: array) | → | void Util.copyInto(receiver, array) |
| boolean Vector:receiver.add(Object:v) | → | boolean receiver.add(v) |
| boolean Vector:receiver.add(int:i, Object:v) | → | boolean receiver.add(i, v) |
| boolean Vector:receiver.addAll(Collection:c) | → | boolean receiver.addAll(c) |
| boolean Vector:receiver.addAll(int:i, Collection:c) | → | boolean receiver.addAll(i, c) |
| void Vector:receiver.addElement(Object:v) | → | boolean receiver.add(v) |
| void Vector:receiver.clear() | → | void receiver.clear() |
| Object Vector:receiver.clone() | → | Object receiver.clone() |
| boolean Vector:receiver.contains(Object:o) | → | boolean receiver.contains(o) |
| boolean Vector:receiver.containsAll(Collection:c) | → | boolean receiver.containsAll(c) |
| Object Vector:receiver.elementAt(int:i) | → | Object receiver.get(i) |
| Enumeration Vector:receiver.elements() | → | Iterator receiver.iterator() |
| void Vector:receiver.ensureCapacity(int:c) | → | void receiver.ensureCapacity(c) |
| boolean Vector:receiver.equals(Object:o) | → | boolean receiver.equals(o) |
| Object Vector:receiver.firstElement() | → | Object receiver.get(0) |
| Object Vector:receiver.get(int:i) | → | Object receiver.get(i) |
| int Vector:receiver.hashCode() | → | int receiver.hashCode() |
| int Vector:receiver.indexOf(Object:o) | → | int receiver.indexOf(o) |
| int Vector:receiver.indexOf(Object:o, int:i) | → | int Util.indexOf(receiver, o, i) |
| void Vector:receiver.insertElementAt(Object:o, int:i) | → | int receiver.add(i, o) |
| boolean Vector:receiver.isEmpty() | → | boolean receiver.isEmpty() |
| Iterator Vector:receiver.iterator() | → | Iterator receiver.iterator() |
| Object Vector:receiver.lastElement() | → | Object receiver.get(int receiver.size() - 1) |
| int Vector:receiver.lastIndexOf(Object:o) | → | int receiver.lastIndexOf(o) |
| int Vector:receiver.lastIndexOf(Object:o, int:i) | → | int Util.lastIndexOf(receiver, o, i) |
| ListIterator Vector:receiver.ListIterator() | → | ListIterator receiver.ListIterator() |
| ListIterator Vector:receiver.ListIterator(int:i) | → | ListIterator receiver.ListIterator(i) |
| Object Vector:receiver.remove(int:i) | → | Object receiver.remove(i) |
| boolean Vector:receiver.remove(Object:o) | → | boolean receiver.remove(o) |
| boolean Vector:receiver.removeAll(Collection:c) | → | boolean receiver.removeAll(c) |
| void Vector:receiver.removeAllElements() | → | void receiver.clear() |
| boolean Vector:receiver.removeElement(Object:o) | → | boolean receiver.remove(o) |
| void Vector:receiver.removeElementAt(int:i) | → | boolean receiver.remove(i) |
| boolean Vector:receiver.retainAll(Collection:c) | → | boolean receiver.retainAll(c) |
| Object Vector:receiver.set(int:i, Object:o) | → | Object receiver.set(i, o) |
| void Vector:receiver.setElementAt(Object:o, int:i) | → | Object receiver.set(i, o) |
| void Vector:receiver.setSize(int:s) | → | void Util.setSize(receiver, s) |
| int Vector:receiver.size() | → | int receiver.size() |
| List Vector:receiver.subList(int:f, int:t) | → | List receiver.subList(f, t) |
| Object[] Vector:receiver.toArray() | → | Object[] receiver.toArray() |
| Object[] Vector:receiver.toArray(Object[]:a) | → | Object[] receiver.toArray(Object[]:a) |
| void Vector:receiver.trimToSize() | → | void receiver.trimToSize() |
| String Vector:receiver.toString() | → | String receiver.toString |

**Figure 15: Specification for migrating from `Vector` to `ArrayList`**

| | | |
|---|---|---|
| boolean Enumeration:receiver.hasMoreElements() | → | boolean receiver.hasNext() |
| Object Enumeration:receiver.nextElement() | → | Object receiver.next() |

**Figure 16: Auxiliary migration from `Enumeration` to `Iterator`**