2007

Automated Refactoring of Legacy Java Software to Enumerated Types

Raffi T. Khatchadourian
CUNY Hunter College

Jason Sawin
University of St. Thomas

Atanas Rountev
Ohio State University - Main Campus

Automated Refactoring of Legacy Java Software to Enumerated Types[*]

Raffi Khatchadourian   Jason Sawin   Atanas Rountev

Ohio State University

{khatchad,sawin,rountev}@cse.ohio-state.edu

Abstract

Java 1.5 introduces several new features that offer significant improvements over older Java technology. In this paper we consider the new enum *construct, which provides language support for enumerated types. Prior to Java 1.5, programmers needed to employ various patterns (e.g., the weak enum pattern) to compensate for the absence of enumerated types in Java. Unfortunately, these compensation patterns lack several highly-desirable properties of the* enum *construct, most notably, type safety. We present a novel fully-automated approach for transforming legacy Java code to use the new enumeration construct. This semantics-preserving approach increases type safety, produces code that is easier to comprehend, removes unnecessary complexity, and eliminates brittleness problems due to separate compilation. At the core of the proposed approach is an interprocedural type inferencing algorithm which tracks the flow of enumerated values. The algorithm was implemented as an Eclipse plug-in and evaluated experimentally on 17 large Java benchmarks. Our results indicate that analysis cost is practical and the algorithm can successfully refactor a substantial number of fields to enumerated types. This work is a significant step towards providing automated tool support for migrating legacy Java software to modern Java technologies.*

1 Introduction

Java 1.5 introduces a rich set of new features and enhancements such as generics, metadata annotations, boxing/unboxing, and type-safe enumerations [23]. These constructs can ease software development and maintenance and can result in more efficient and robust applications. Even though Java 1.5 has backward compatibility with code from previous releases, there are numerous advantages in migrating such legacy code to these new features.

Code migration can be a laborious and expensive task both for code modification and for regression testing. The costs and dangers of migration can be reduced greatly through the use of automated refactoring tools. This paper presents a fully-automated semantics-preserving approach for migrating legacy Java code to take advantage of the new type-safe enumeration construct in Java 1.5.

An *enumerated (enum) type* [19] is a data type whose legal values consist of a fixed, closely related set of items known at compile time [2]. Typically, the exact values of the items are not programmatically important: what is significant is that the values are distinct from one another and perhaps ordered in a certain way, hence the term "enumeration." Clearly this is a desirable construct, and since it was not included in the Java language until version 1.5, developers were forced to use various *compensation patterns* to represent enum types. These patterns produce solutions with varying degrees of uniformity, type safety, expressiveness, and functionality. Of these patterns, the most popular and proclaimed "standard way" [23] to represent an enumerated type in legacy (≤ 1.4) Java is the *weak enum pattern* [2], also known as *type codes* [9, 14]. This pattern uses declared constants ("codes") defined with relatively small, manually enumerated values. These constants are typically declared as `static final` fields. As discussed in Section 2, there are great advantages to migrating compensation patterns in legacy code to proper enum types.

In this paper we propose a novel semantics-preserving approach for identifying instances of the weak enum pattern in legacy code and migrating them to the new enum construct. At the core of our approach is an interprocedural type inferencing algorithm which tracks the flow of enumerated values. Given a set of static final fields, the algorithm computes an *enumerization grouping* containing fields, methods, and local variables (including formal parameters) whose types can safely be refactored to use an enum type. The algorithm identifies the fields that are being utilized as enumerated values and all other program entities that are transitively dependent upon these values.

The refactoring approach has been implemented as an Eclipse plug-in. The experimental evaluation used a set of 17 Java programs with a total of 899 thousand lines of code. Our study indicates that (1) the analysis cost is practi-

---

cal, with average running time of 2.48 seconds per thousand lines of code, (2) the weak enum pattern is commonly used in legacy Java software, and (3) the proposed algorithm successfully refactors a large number of static final fields into enumerated types.

This work makes the following specific contributions:

- *Algorithm design.* We present a novel automated refactoring approach for migration to Java 1.5 enum types. The approach infers which fields are being used as enumerations and identifies all code changes that need to be made in order to introduce the inferred enum types.

- *Implementation and experimental evaluation.* The approach was implemented as an Eclipse plug-in to ensure real-world applicability. A study on 17 Java programs indicates that the proposed techniques are effective and practical. These results advance the state of the art in automated tool support for the evolution of legacy Java code to modern Java technologies.

2   Motivation and Example

An *enumerated type* has values from a fixed set of constants [2]. Java has historically provided no language mechanisms for defining enumerated types, leading to the emergence of various compensation patterns. However, the compiler depends on the internal representation (typically `int`) of the symbolically named constants, and type checking can not distinguish between values of the enum type and those of the type internally representing those values.

Figure 1(a) shows an example in which named constants are used to encode values of enumerated types.[1] For example, field `color` declared at line 6 represents the color which the traffic signal is currently displaying. The values of this field come from the three static final fields `RED`, `YELLOW`, and `GREEN`, which map symbolic names to their associated integer representations. The compile-time values of these constants are manually enumerated so that each color can be unambiguously distinguished. Of course, the integer values have no real relationship to the colors they represent. Similarly, field `currentAction` declared at line 17 could take its values from the integer constants in static final fields `IDLE`, `INCREASE_SPEED`, `DECREASE_SPEED`, and `STOP`.

Field `MAX_SPEED` (line 15) defines the maximum speed of the automobile. This field differs from the remaining static final fields: unlike their integer values, which are used only to encode enumerated values, the value of `MAX_SPEED` has a very significant meaning. This key distinction illustrates the difference between fields that are *named constants* (e.g., `MAX_SPEED`) from those participating in the *int enum*

pattern [2].[2] In this paper we consider a more general version of this pattern which applies to all primitive types[3]; we will refer to it as the *weak enum pattern*. The term "weak" is used to denote the lack of type safety and other features inherent to the pattern.

Figure 1(a) illustrates the use of the weak enum pattern. Clearly, the meaning of `int` depends on the context of where values are used. The programmer is left with the responsibility of *manually* inferring which `int` entities are intended to represent traffic light colors, which are automobile actions, and which are integers. In effect, the programmer would be required to investigate transitive relationships of these program entities to other program entities/operations. Although the weak enum pattern provides a mechanism to make programmer intent more explicit, it suffers from several significant weaknesses which have been well documented [2, 23].

The most glaring weakness is the lack of type safety. For example, there is no mechanism to enforce the constraint that `color` gets its values only from the three color fields: any integer value would be acceptable at compile time. Such problems would not be detected until run time, when an exception would be thrown. Perhaps worse, the execution will seem to be normal while behaving in a way not originally intended by the programmer. Problems could also arise from the allowed operations: for example, it would be possible to perform arbitrary integer operations, such as addition or multiplication, upon the color values.

The weak enum pattern creates ambiguities at various levels. For example, there are fundamental semantic differences between the constants for automobile actions (beginning on line 11) and `MAX_SPEED` (line 15). Despite these differences, both entities have essentially identical declarations. The programmer depends on documentation and/or extensive interprocedural usage investigation to determine the true intent of the fields. This is also an issue for *multiple* sets of enum constants. For example, methods `getColor` (line 8) and `react` (line 21) declare the same `int` return type, even though the returned entities have very different meaning and context. In essence, the program is less self-documented with respect to the enumerated types, which could have negative effect on software maintenance tasks.

Verbosity and added complexity arises in several areas. First, there is no easy way to print the enumerated values in a meaningful way. Additional code is typically required to produce desirable results, e.g. as in

```
if (this.color == RED) System.out.println("RED");
```

Second, there is no convenient way to iterate over all values of the enumerated type [2], which requires the developer

---

[1]This example was inspired by one of the authors' work at the Center for Automotive Research at the Ohio State University.

[2]A similar pattern called *Type Codes* is described in [9] and [14].

[3]We exclude boolean from this list for several reasons: (i) The type has only two values, `true` and `false`, thus any transformed enum type can only have two members and (ii) our algorithm becomes simpler due to this exclusion.

```
1   class TrafficSignal {
2     public static final int RED = 0;
3     public static final int YELLOW = 1;
4     public static final int GREEN = 2;
5     /* Current color of the traffic signal, initially red by default */
6     private int color = RED;
7     /* Accessor for the light's current color */
8     public int getColor() {return this.color;}}
9
10  class Automobile {
11    private static final int IDLE = 0;
12    private static final int INCREASE_SPEED = 1;
13    private static final int DECREASE_SPEED = 2;
14    private static final int STOP = 3;
15    private static final int MAX_SPEED = 140;
16    /* The action this automobile is currently performing, idle by default */
17    private int currentAction = IDLE;
18    /* The current speed of the automobile, initially 5 mph. */
19    private int currentSpeed = 5;
20
21    private int react(TrafficSignal signal) {
22      switch(signal.getColor()) {
23        case TrafficSignal.RED: return STOP;
24        case TrafficSignal.YELLOW:
25          // decide whether to stop or go
26          if (this.shouldGo())
27            return INCREASE_SPEED;
28          else return STOP;
29        case TrafficSignal.GREEN: // no change
30          return this.currentAction;
31        default: throw new IllegalArgumentException
32          ("Invalid traffic color");}} // required
33
34    public void drive() {
35      TrafficSignal aSignal = ... ;
36      int reaction = this.react(aSignal);
37      if (reaction != this.currentAction &&
38          (reaction != INCREASE_SPEED ||
39           this.currentSpeed <= MAX_SPEED))
40            this.performAction(reaction);}
41
42    private void performAction(int action) {...}}
```

(a) Using integer constants for enumerated types.

```
1   class TrafficSignal {
2     public enum Color {RED,
3       YELLOW,
4       GREEN};
5     /* Current color of the traffic signal, initially red by default */
6     private Color color = Color.RED;
7     /* Accessor for the light's current color */
8     public Color getColor() {return this.color;}}
9
10  class Automobile {
11    private enum Action {IDLE,
12      INCREASE_SPEED,
13      DECREASE_SPEED,
14      STOP};
15    private static final int MAX_SPEED = 140;
16    /* The action this automobile is currently performing, idle by default */
17    private Action currentAction = Action.IDLE;
18    /* The current speed of the automobile, initially 5 mph. */
19    private int currentSpeed = 5;
20
21    private Action react(TrafficSignal signal) {
22      switch(signal.getColor()) {
23        case TrafficSignal.RED: return Action.STOP;
24        case TrafficSignal.YELLOW:
25          // decide whether to stop or go
26          if (this.shouldGo())
27            return Action.INCREASE_SPEED;
28          else return Action.STOP;
29        case TrafficSignal.GREEN: // no change
30          return this.currentAction;
31        default: throw new IllegalArgumentException
32          ("Invalid traffic color");}} // required
33
34    public void drive() {
35      TrafficSignal aSignal = ... ;
36      Action reaction = this.react(aSignal);
37      if (reaction != this.currentAction &&
38          (reaction != Action.INCREASE_SPEED ||
39           this.currentSpeed <= MAX_SPEED))
40            this.performAction(reaction);}
41
42    private void performAction(Action action){...}}
```

(b) Improvements after our refactoring is applied.

Figure 1. Running example: a hypothetical drive-by-wire application.

to manually create such machinery. Third, the weak enum pattern requires the programmer to manually enumerate the values of the constants, which increases the likelihood of errors. For example, different enum constants may be unintentionally assigned the same internal value. Finally, the resulting types are brittle [23]: since the values are compile-time constants, at compile time they are inlined into clients. Therefore, if new constants are added in between existing ones, or if the internal representation of the constants changes, clients must be recompiled.

Enumerations in Java 1.5. The new enum construct supports powerful enumerated types that are completely and conveniently type safe, comparable, and serializable; saving the programmer from creating and maintaining verbose custom classes. Enum types increase self-documentation (e.g., a getColor method has a return type of Color), enable compile-time type checking, allow meaningful printed values, avoid name conflicts, and support separate compilation.

Figure 1(b) shows an *enumerized* version of the running example, in which the static final fields have been replaced by language enumerated types TrafficSignal.Color and Automobile.Action. The legal values and operations of these new enumerated types are now enforced through compile-time checking. There is a clear distinction between the named constant MAX_SPEED and the enumerated values. It is also clear that the result of a call to react is an Action, which distinguishes it from the return type of getColor and makes the API more informative. Programmers are no longer required to enumerate values by hand, or to write extra "pretty printing" code.

After enumerization, the brittleness of the overall system is reduced. For example, suppose we wanted to make TrafficSignal compatible with Poland's system, where a yellow and red combination is shown directly after red to alert drivers that a change to green is imminent. After RED in Figure 1(a), one could add a new field RED_YELLOW with value of 1; the remaining fields' values would have to be incremented. Even if we did not care to modify Automobile to accommodate the new color, we would still have to recompile it, since upon the original compilation the constant values for the colors where inlined. In Figure 1(b) addi-

tional values can be added easily, and only the enum or the class containing the enum would require recompilation.

3 Enumerization Approach

A refactoring tool which modifies legacy Java code employing the weak enum pattern to utilize the Java 1.5 `enum` construct faces two major challenges: *inferring enumerated types* and *resolving dependencies*. Inferring enumerated types requires distinguishing between weak enum constants and named constants. Figure 1(a) illustrates this issue through fields `STOP` and `MAX_SPEED`. Although their declarations are very similar, they are conceptually very different: while the value of named constant `MAX_SPEED` is meaningful in integer contexts (e.g., for the integer comparison at line 39), the only requirement on the value of enumerated constant `STOP` is that it should be different from the other integer values representing actions. In general, the uses of the enumerated values are limited to assignments, parameter passing, method return values, and equality comparisons. Named constants are used in a much wider context, including mathematical calculations (e.g., dividing by `java.lang.Math.PI`), various value comparisons (as in line 39), and so on. Determining the category to which a constant field belongs requires investigation of every context in which that field's value is used.

Constant fields are not the only program entities that need to be refactored for enumerization. In Figure 1(a), once it has been inferred that `STOP` is an enumerated constant, we must identify all program entities that also require refactoring due to transitive dependencies on `STOP`. We say a entity A is *type dependent* on entity B if changing the type of B requires changing the type of A. An example of such a dependency is method `react`: since it returns the integer form of `STOP`, in the refactored version it must return the enum type containing `STOP`. Furthermore, due to the dependence on the return value of `react`, local integer variable `reaction` in `drive` (line 36 in Figure 1(b)) must also be transformed to be of type `Action`.

The next section describes an interprocedural refactoring algorithm which addresses these challenges through careful categorization of the contexts in which migration from the weak enum pattern to the new enum construct is valid. The algorithm identifies all type dependent entities in those contexts, including fields, local variables, method return types, and formal parameters. After all affected entities are identified, they are classified into groups that must share the same enum type. At the end, all automatically transformed code is semantically equivalent to the original.

4 Algorithm

Assumptions. Our algorithm works on a *closed-world assumption*, meaning that we assume full access to all source code that could possibly affect or be affected by the

$\mathcal{P}$	original program
$\phi(\mathcal{P})$	$\{f \mid f \text{ is a static final field of primitive type in } \mathcal{P}\}$
$\mu(\mathcal{P})$	$\{m \mid m \text{ is a method in } \mathcal{P}\}$
$\upsilon(\mathcal{P})$	$\{l \mid l \text{ is a variable in } \mathcal{P}\}$
α	variable, field, method
α_{ctxt}	*context* in which α may occur

Figure 2. Formalism notation.

procedure $Enumerize(F, \mathcal{P})$
1: $R \leftarrow Enumerizable(F)$
2: $R \leftarrow Unique(R) \cap Distinct(R) \cap Consistent(R)$
3: **for all** $T \in R$ **do**
4: $\quad Transform(T)$
5: **end for**

Figure 3. Top-level enumerization algorithm.

refactoring. We also assume that we are able to statically identify all references to candidate fields and transitively dependent program entities. This assumption could be invalidated through the use of reflection and custom class loaders. We also assume that the original source code successfully compiles under a Java 1.5 compiler.

Top-level processing. Procedure $Enumerize$, shown in Figure 3, is the top-level driver of our approach. It takes as input the source code of the original program $\mathcal{P}$, as well as a set $F \subseteq \phi(\mathcal{P})$ of fields (see the notation in Figure 2; parts of this notation were inspired by [10, 16, 21]). In this paper we consider refactoring the "standard" compensation pattern in pre-Java 1.5 as described in [2, 9, 14, 23]. As such, $Enumerize$ analyzes only static final fields of primitive types since they may potentially be participating in the weak enum pattern. Function $Enumerizable$ (called at line 1) infers which candidate fields are being used as enumerated values and groups them into their corresponding inferred enum types. At line 2, certain semantics-preserving constraints are enforced (further discussed later in this section). Finally, $Transform$ (line 4) performs the actual code refactoring for each inferred enum type T, thus altering the type declarations of each corresponding program entity. The primitive constant declarations are replaced with the new `enum` type declarations and are ordered by their original primitive values to enforce a natural ordering, thereby preserving comparability semantics.

Type inferencing. Function $Enumerizable$, shown in Figure 4, is at the heart of the proposed approach. This type inferencing algorithm is based on a family of type inferencing approaches from [18] and has two goals:

(i) infer fields that are being used as part of enumerated types (i.e., participating in the weak enum pattern)

(ii) construct minimal sets such that members of the same set must share the same enum type after refactoring

The output of the algorithm is a set of *enumerization sets* containing fields, method declarations, and local variables (including formal parameters) and their minimal groupings that are enumerizable with respect to the input constants.

```
function Enumerizable(C)
 1:  W ← C /* seed the worklist with the input constants */
 2:  N ← ∅ /* the non-enumerizable set list, initially empty */
 3:  for all c ∈ C do
 4:     MakeSet(c) /* init the union-find data structure */
 5:  end for
 6:  while W ≠ ∅ do
 7:     /* remove an element from the worklist */
 8:     α ← e | e ∈ W
 9:     W ← W \ {α}
10:     for all α_ctxt ∈ Contexts(α, P) do
11:        if ¬isEnumerizableContext(α, α_ctxt) then
12:           /* add to the non-enumerizable list */
13:           N ← N ∪ {α}
14:           break
15:        end if
16:        /* extract entities to be enumerized due to α */
17:        for all α̂ ∈ Extract(α, α_ctxt) do
18:           if Find(α̂) = ∅ then
19:              MakeSet(α̂)
20:              W ← W ∪ {α̂}
21:           end if
22:           Union(Find(α), Find(α̂))
23:        end for
24:     end for
25:  end while
26:  F ← AllSets() /* the sets to be returned */
27:  for all α' ∈ N do
28:     F ← F \ Find(α') /* remove nonenum sets */
29:  end for
30:  return  F /* all sets minus the non-enumerizable sets */
```

Figure 4. Building enumerization sets.

The algorithm uses a worklist W which is initialized with all given constant fields, as well as a set N of entities that are not amenable to enumeration. A union-find data structure maintains sets of related entities; initially, each input constant field belongs to a separate singleton set. Each worklist element α is a program entity whose type may have to be changed to an enum type. A helper function $Contexts$ identifies all contexts (explained next) in which α and its related entities α' appear in $\mathcal{P}$ such that each context α_{ctxt} needs to be examined later in the algorithm.

$Contexts(\alpha, \mathcal{P})$ includes all *inner-most* (i.e., identifier terminals in the grammar) expressions corresponding to α[4]. Furthermore, if α is a method, this set of contexts also includes $Contexts(\alpha', \mathcal{P})$ for every method α' which overrides α or is overriden by α. Similarly, if α is a formal parameter, the set of contexts includes $Contexts(\alpha', \mathcal{P})$ for every corresponding formal parameter α' in an overriding or overridden method. Entities α' need to be considered due to polymorphism. For example, if the return type of a method m is changed from int to an enum type, this change must be propagated to all methods overriding m or being overrid-

---

[4]Excludes those appearing in initializations of constant fields.

den by m. Similar propagation is necessary when m's formal parameters are changed (otherwise, method overriding would incorrectly be transformed to method overloading).

Function *isEnumerizableContext* examines a context α_{ctxt} to determine if it is amenable to enumeration with respect to α by using two helper functions *EnumerizableAscender* and *EnumerizableDescender*. Upon application, these helper functions examine the context sent to *isEnumerizableContext* by traversing, in disparate directions, the syntax tree of the input expression. The intent of these functions are loosely analogous to that of synthesized and inherited attributes of attribute grammars [17], respectively. Function *Extract* is responsible for determining further transitive relationships due to the enumerization of α. *Extract* also has two helper functions *ExtractionAscender* and *ExtractionDescender* which are similar in flavor to the aforementioned helper functions. For conciseness, in the following discussion we will use the abbreviations EC, EA, ED, and EX to refer to these functions. EC has two parameters: the entity α whose enumerizability is under question and a context α_{ctxt} which is type dependent on α. EX, on the other hand, has one parameter α_{ctxt} whose constituent, type dependent program entities must be examined for enumerizaton.

Function EC immediately calls EA passing it α_{ctxt}, the context to be examined and α, the entity whose enumerization is under question. Figure 5 portrays several of the rules of EA which are inductively defined in the grammar. EA begins at α_{ctxt} (e.g., ID) and *climbs* (or ascends) its way up the grammar until it reaches a *significant ancestor* of α. We say that a statement or expression is a significant ancestor of α if the value of α can be exploited at that point. The ascent is performed via the $Parent$ function which returns the parent expression above α_{ctxt} in the syntax tree. The function $contains$ helps determine which expression EA ascended from.

On the way to the significant ancestor, EA may find expressions that are not amenable to enumerizaton. In that case, EA will return *false* and EC, in turn, will return the result of EA. Such a situation is depicted in the rule for array access/creation in Figure 5. On the other hand, once EA successfully reaches the significant ancestor, it will then call ED in order to commence a descent down the *pivotal* expression(s); that is, an expression that is consequently type dependent. Several of the rules of ED are given in Figure 6. As shown, ED completes its descent at the leaf nodes of the syntax tree, returning *true* for terminal IDs and *false* for contexts which are not amenable to enumerizaation (e.g., literals). EA will then, in turn, return the result of ED.

Enumerizable contexts. EC returns *false* if the given context α_{ctxt} is definitively not enumerizable with respect to α (e.g., α being used as an array index). Otherwise, EC returns *true* if α_{ctxt} is *promising* with respect to α — that

<u>Identifiers.</u>
function $EA(\alpha, \text{ID})$

  1: **return** $EA(\alpha, Parent(\text{ID}))$

Equality expressions
function $EA(\alpha, \text{EXP}_1 == \text{EXP}_2)$

  1: **return** $ED(\text{EXP}_1) \wedge ED(\text{EXP}_2)$

Array access/creation expressions
function $EA(\alpha, \text{EXP}_1[\text{EXP}_2])$

  1: **if** $contains(\text{EXP}_2, \alpha)$ **then**

  2:   **return false**

  3: **else**

  4:   **return** $EA(\alpha, Parent(\text{EXP}_1))$

  5: **end if**

Conditional expressions
function $EA(\alpha, \text{EXP}_1 ? \text{EXP}_2 : \text{EXP}_3)$

  1: **if** $contains(\text{EXP}_2, \alpha) \vee contains(\text{EXP}_3, \alpha)$ **then**

  2:   **return** $EA(\alpha, Parent(\text{EXP}_1 ? \text{EXP}_2 : \text{EXP}_3))$

  3: **else**

  4:   **return true**

  5: **end if**

Figure 5. Enumerizable ascender.

<u>Integer literals</u>
function $ED(\text{IL})$

  1: **return false**

<u>Identifiers</u>
function $ED(\text{ID})$

  1: **return true**

Parenthesized expressions
function $ED((\text{EXP}))$

  1: **return** $ED(\text{EXP})$

Assignment expressions
function $ED(\text{EXP}_1 = \text{EXP}_2)$

  1: **return** $ED(\text{EXP}_1) \wedge ED(\text{EXP}_2)$

Infix addition expressions
function $ED(\text{EXP}_1 + \text{EXP}_2)$

  1: **return false**

Figure 6. Enumerizable descender.

is, enumerizing α does not adversely affect the relation between α and the enclosing expressions of α_{ctxt}. We say that such a situation is "promising" as opposed to "definite" because there may exist other program entities $\hat{\alpha}$ that are type dependent on α and we cannot yet ensure that every context $\hat{\alpha}_{ctxt}$ in which $\hat{\alpha}$ appears is enumerizable. This additional checking for $\hat{\alpha}$ is performed by EX, which extracts the type dependent entities that require further investigation to determine if they are enumerizable with respect to a particular α (see [15] for further details). These extracted entities will be put on the worklist and eventually checked by EC.

To illustrate the type checking component mechanics we show the application of the EC function at each significant ancestor discovered during the evaluation the assignment `color=RED` from line 6 of our motivating example depicted in Figure 1(a). The terminal expression `RED` within the as-

signment expression `color=RED` would have been returned by *Contexts* when α is `RED`. Applying EC for this context we have:

$$EC(\text{RED}, \text{RED}) \equiv$$
$$EA(\text{RED}, \text{RED}) \equiv$$
$$EA(\text{RED}, \text{color} = \text{RED}) \equiv$$
$$ED(\text{color}) \wedge ED(\text{RED}) \equiv$$
$$true \wedge true \equiv true$$

As a result, this expression is considered "promising". The subsequent application of EX would extract the program entity `color` so that all of its contexts may be checked.

Consider a hypothetical assignment `color=5` when α is `color`; here `color` is type dependent on the integer literal `5`. Using the rules in Figures 5 and 6, $EC(\text{color}, \text{color})$ is determined to be *false*. Because the type of the integer literal cannot be altered to an enum type, `color` also cannot be altered and should be included in set N (line 13 in Figure 4).

There are other situations where type dependencies prevent a program entity from being enumerized. For example, consider the following statement where α is again `RED`: `if(color==arr[RED]) color=GREEN;`. The derivation using our rules would consist of the following:

$$EC(\text{RED}, \text{RED}) \equiv$$
$$EA(\text{RED}, \text{RED}) \equiv$$
$$EA(\text{RED}, \text{arr}[\text{RED}]) \equiv$$
$$false$$

In this case, EC returns *false* since it would be impossible to alter the type of `RED` because the index to an array access must be an integral type [12]. Note that the *then* portion of the if statement is not evaluated as it is not type dependent on α. Although EX is not called when EC returns *false*, EX would nevertheless return $\emptyset$ upon these arguments.

In general, the enumerizability of particular α may depend on its occurrences within comparison expressions (see the rules for equality/inequality expressions in Figure 5). For comparison expressions with `==` and `!=`, as long as both operand expressions are enumerizable both will be included in the same inferred enum type, and the integer equality/inequality in the original code will be transformed to reference equality/inequality. For `<`, `<=`, `>`, and `>=`, the refactored code can use the methods from interface `java.lang.Comparable`, which is implemented by all enum types in Java 1.5, to preserve comparability semantics amongst the inferred type's members. This holds true so as long as the inferred enum type declarations are in the order given by their original primitive representations.

An interesting case is contexts in which polymorphic behavior may occur. In these cases, we need to consider entire hierarchies of program entities. Much of the polymorphic behavior enforcement is implemented with the help of function *Contexts* described earlier, however, additional checks

are needed within *isEnumerizableContext* and *Extract* in order to ensure the preservation of program semantics. In particular, the formal parameter expressions and the method invocation expressions require additional investigation of program entities in $\mathcal{P}$. For example, in the case of formal parameters EX must be certain to extract the program entities embedded in the corresponding actual argument expressions for each method invocation in the method hierarchy. Due to space constraints, we invite the reader to examine our companion report [15] for further details.

Transitive dependencies. In function *Enumerizable*, if either a context which is not amenable to enumerization is encountered, or one that can not be transformed, we mark the set containing the α in question as a "non-enumerizable" set (line 13 in Figure 4). If this is not the case, the algorithm proceeds to extract other program entities that are required to undergo enumerization consideration due to the enumerization of α (line 17). For each of these program entities $\hat{\alpha}$ the following steps are taken. If $\hat{\alpha}$ is not currently contained in an existing set (line 18), which implies that it has not previously been seen, then a new singleton in the union-find data structure is created and consequently added to the worklist (lines 19 and 20). The two sets, the set containing α and the set containing $\hat{\alpha}$, are then merged on line 22 thereby capturing the transitive dependencies between each program entity. Once the computation is complete, i.e., the worklist has emptied, the sets defined implicitly by the union-find data structure are returned minus the non-enumerizable sets (line 30).

Function *Enumerizable* is responsible for type inferencing; that is, it ensures that the proposed transformation is type-correct. Its result is a partitioning of program entities, limited to variables, fields, and methods, that are enumerizable with respect to a given set of static final fields. This essential relationship existing between each member of each enumerizable set is expressed by our first *member constraint*, listed as constraint 1 of Figure 7. This constraint simply expresses that all members of each set are enumerizable with respect to the original input constants, of whom are also in the set. The partitioning captures the *minimal* dependency relationships between these entities; if a transformation of one of the elements occurs, then, in order for preserve type correctness, a transformation of *all* elements in its set must also occur. However, we must make further, more subtle considerations as to which sets can be transformed. We discuss such considerations next.

Semantics-preserving constraints. In addition to analyzing the *usage* of potential enumerated type constants, in order to preserve semantics upon transformation, it is also necessary to analyze their *declarations*. Returning to the *Enumerize* function listed in Figure 3, the functions invoked on line 2 enforce program behavioral preservation by excluding sets containing constants that do not meet the

Let $Enumerizable : \mathscr{P}\left[\phi(\mathcal{P})\right] \longrightarrow \mathscr{P}^{(2)}\left[\phi(\mathcal{P}) \cup \mu(\mathcal{P}) \cup \upsilon(\mathcal{P})\right]$ be a function mapping a set of primitive static final fields to a set of *minimal* program entity sets that are enumerizable in respect to those fields.

$$\forall k \in K[\exists X, Y \mid k \in X \wedge X \in Enumerizable(Y) \wedge K \subseteq Y] \quad (1)$$

Let $\iota : \phi(\mathcal{P}) \to \Sigma^*$ be a function mapping a field to its unqualified identifier.

$$\forall k_i, k_j \in K[i \neq j \Rightarrow \iota(k_i) \neq \iota(k_j)] \quad (2)$$

Let $\mathbb{P}$ be the set of all legal primitive values.
Let $\sigma : \phi(\mathcal{P}) \to \mathbb{P}$ be a function mapping a constant to its primitive value.

$$\forall k_i, k_j \in K[i \neq j \Rightarrow \sigma(k_i) \neq \sigma(k_j)] \quad (3)$$

Let $\mathbb{V} = \{public, protected, private, package\}$ be the set of legal visibilities.
Let $\vartheta : \phi(\mathcal{P}) \to \mathbb{V}$ be a function mapping a constant to its visibility.

$$\forall k_i, k_j \in K[\vartheta(k_i) = \vartheta(k_j)] \quad (4)$$

Figure 7. Member constraints for transforming a group of candidate fields K

remaining member constraints given in Figure 7. Invocation of the function $Unique$ corresponds to the enforcement of constraint 2, $Distinct$ to constraint 3, and $Consistent$ to constraint 4. Essentially, these constraints express that, for each set to be transformed into a corresponding enum type, and for semantics to be preserved, each static final field must be uniquely named (since constants may have originated from different classes), distinctly valued (so that each originally assigned primitive value will correspond to a single memory reference), and consistently visible (since the new enum types are not allowed to have instances with independent visibilities). The resulting intersection of the sets abiding to each of the member constraints is then assigned back to R. At line 4, each set $T \in R$ corresponds to the program entities that will be transformed to the new language enumeration type T and the transformation takes place $\forall T \in R$.

5 Experimental Study

Implementation. We implemented our algorithm as a plug-in in the popular Eclipse IDE. Eclipse ASTs with source symbol bindings were used as an intermediate program representation. The plug-in is built over an existing refactoring framework [1] and is coupled with other refactoring support in Eclipse. To increase applicability to real-world applications, we relaxed the closed-world assumption described in Section 3. For example, if the tool encounters a variable that is transitively dependent upon an element outside of the source code being considered, this variable and all other entities dependent on it are conservatively labeled as non-enumerizable.

Experimental evaluation. To evaluate the effectiveness of our algorithm, we used the 17 open-source Java applica-

benchmark	KLOC	classes	prim	cands	enum	uses	rtypes	time (s)
ArtOfIllusion	75	378	333	77	77	111	46	207
Azureus	272	1894	1255	399	347	635	173	1269
java5	180	1586	1299	557	450	572	363	760
JavaCup	6	41	55	3	3	3	3	19
jdepend	3	28	13	1	1	1	1	1
JFlex	10	46	140	24	19	27	9	75
JFreeChart	71	420	153	36	24	43	12	128
jGap	6	137	25	4	4	5	1	7
jgraph	14	91	25	6	3	6	1	11
JHotDraw	29	496	34	11	11	24	8	14
junit	8	271	7	2	2	3	1	1
jwps	20	155	156	76	64	102	25	60
sablecc	29	237	16	8	8	10	2	9
tomcat6	153	1164	738	344	335	400	255	346
verbos	5	41	10	6	6	15	2	3
VietPad	11	84	36	17	17	22	4	8
Violet	7	73	36	14	13	20	6	9
Total:	899	7142	4331	1585	1384	1999	915	2227

Table 1. Experimental results.

tions and libraries listed in Table 1.[5] The second column in the table shows the number of non-blank, non-comment lines of source code, which range from 3K for `jdepend` to 272K for `Azureus`. The third column shows the number of class files after compilation. For each benchmark, the analysis was executed five times on a 2.6 GHz Pentium4 machine with 1 GB RAM. The average running time, in seconds, is shown in column *time* in Table 1. On average, the analysis time was 2.48 seconds per KLOC, which is practical even for large applications.

Column *prim* shows the number of static final fields of primitive types[6]. We separate these fields into two categories. First, certain fields *definitely* cannot be refactored, because the semantics of the program depends on the specific, actual values of these fields. These fields include those that were either directly or transitively dependent on operations that utilized their exact value or created a transitive dependency on an entity which could not be refactored (a complete list of filtered contexts is described in [15]). We also include in this category the fields which cannot be refactored due to lack of access to source code (e.g., a field passed as a parameter to a method defined in a library whose source code is not available).

We categorize the remaining fields to be *candidate fields*. The number of candidate fields per benchmark is shown in column *cands*. The fact that the actual values of these fields do not directly affect the semantics of the program provides a strong indication that they are playing the role of enumerations in the weak enum pattern. The set of candidate fields along with their corresponding, transitive entities represent the *minimal* set of elements a programmer would have to investigate for refactoring. Note that although these sets are

minimal, for three of our benchmarks they still contain well over 300 elements, and several others contain over 50 elements.

The number of fields that our plug-in could safely refactor is shown in column *enum*. The results show that our approach was able to refactor 87% of the fields that could possibly be participating in the weak enum pattern. The tool was unable to refactor the remaining 13% of fields because either they or an element of their dependency sets were used in explicit cast expressions. We conservatively choose not to refactor elements used in cast expressions due to the existence of possible side effects on the values of variable through narrowing conversions. For example, consider the following code: `short z = 128; byte x = (byte)z;`. This is a valid cast in Java, but this cast will result in `x` having the value -128 and *not* 128. Clearly, not accounting for such an occurrence prior to refactoring could lead to significant changes in program semantics upon migration. Detecting such changes due to explicit casts is beyond the scope of the work being considered in this paper.

Of course, fields are not the only program entities whose type requires alteration. Column *uses* shows the total number of declaration sites that must be modified to accommodate the enumerization. The numbers motivate the need for automated tools such as ours. In particular, the large applications require hundreds of code modifications (e.g., over 600 for `Azureus`). These code modifications are spread across many classes and packages, and occur in many distinct methods. Attempting to identify the needed modifications by hand would be a labor-intensive and error-prone task.

Column *rtypes* shows the number of resulting enum types produced by our tool. Note that the number of types is relatively close to the number of enum fields. This indi-

---

[5] `java5` denotes the package `java.` included in the Java 1.5 JDK.
[6] Excludes `boolean` types.

cates that there are few actual enumeration values per enum type, on average about 2.2 per type. This number may not reflect the number of weak enum pattern instances intended by the programmer. Our algorithm is conservative in its type creation, only grouping fields that share transitive dependencies. These are the only fields that *must* share the same enum type upon refactoring. However, given the current state of the program source, dependencies may not exist between all enumerations intended to be grouped as one type. For the running example, DECREASE_SPEED should intuitively be grouped with the other vehicle actions. Unfortunately, since it is not currently being referenced by the code, it does not share a dependency with any of the other fields and as a result it is assigned a singleton set. Clearly, in this case no algorithmic method could guarantee the exact grouping intended by the programmer; however, there are various heuristics that may be employed to better approximate the intended types (e.g., heuristics that take into account lexical proximity of field declarations, similar to what is described in [13]).

Summary. Overall, the experimental results indicate that the analysis cost is practical, that the weak enum pattern is commonly used in legacy Java software, and that the proposed algorithm successfully refactors a large number of fields (and their dependent entities) into enumerated types.

6 Related Work

Both Fowler [9] and Kerievsky [14] present the refactoring entitled REPLACE TYPE CODE WITH CLASS. Both detail a series of steps involved in transforming *type codes* (entities subscribing to what we label as the *weak enum pattern* in this paper) into instances of custom, type-safe classes utilizing the Singleton pattern [11]. Bloch [2] presents a similar solution. While the pattern describes an enum class that seems effective in regards to the same criteria we have presented in this paper, the refactoring process is entirely manual and the transformation is not to language enumerated types. Most importantly, the developer is required to posses *a priori* knowledge of exactly which fields are potentially participating in the type code pattern in order to perform the refactoring. Our proposed approach does not require such knowledge and is completely automated.

Tip et al. [26] propose two automated refactorings, EXTRACT INTERFACE and PULL UP MEMBERS, both well integrated into the Eclipse IDE. These refactorings deal with *generalizing* Java software in an effort to make it more reusable. Although this proposal shares similar challenges with our approach in respect to precondition checking and interprocedural dependency analysis, there are several key differences. The generalization approach manipulates the interfaces[7] of reference types along with the means in which

---

[7]The term *interface* is used here in a broad sense.

objects communicate through those interfaces, as our approach entails transforming primitive type entities *to* reference types. Moreover, a method based on *type constraints* [18] is used to resolve dependencies amongst program entities. Sutter et al. [24] also use type constraints in addition to profile information to customize the use of Java library classes. A type constraint approach would have also been conceivable for our work in that similar type constraints may have been formed for primitive types. Nonetheless, a type constraint-based approach for primitive transformation may have proven to be excessive since primitive types do not share many of the same relationship as reference types (e.g., sub-typing). Therefore, we preferred more of a type checking approach as opposed to constraint solving.

Several other approaches [7, 10, 16, 25, 28] exist to migrate legacy Java source to utilize new Java 1.5 language features, in particular generics. Although both generics and language enumeration types serve to improve type safety, the two features are conceptually different and face unique challenges in automated migration, including preserving program erasure [3], sub-typing compatibility, and inferring wildcard types. However, our proposal for inferring *enumerated types*, although not being required to address such issues, requires introducing a *new* type in the original source as opposed to introducing a type parameter or argument for an *existing* type. When refactoring primitives one must consider many additional operations that may be invoked on the primitive entities that are not available to reference types.

Steimann et al. [21, 22] propose an approach to decouple classes with inferred interfaces. Similar to our approach, a new type is introduced in the source (i.e., the inferred interface), and the compile-time types of program entities are altered as a result of the refactoring. Additionally, both approaches do not leverage constraint solving mechanisms, instead, Steimann et al. utilize a static analysis based on [4].

Automated usage analysis and type inferencing techniques similar to ours also exist for other languages. Eidorff et al. [8] demonstrate a Year 2000 conversion tool utilizing type inferencing techniques for correcting problematic date variables in COBOL (a *weakly-typed* programming language) systems. Ramalingam et al. [20] also exploit usage analysis techniques to identify implicit aggregate structure and programmer intent of COBOL program entities not evident from their declarations.

Proposals for identifying enumerated types exist for COBOL and C. Although our work applies to a significantly different source language, methods for identifying enumerated types in these legacy systems share similar challenges. Deursen and Moonen [5, 6] present a general approach utilizing judgements and rules for inferring type information from COBOL programs. Our approach, however, is focused more on the *migration* to a *specific* language enumerated type construct that contains corresponding, preexist-

ing constraints. As a result, our approach must deal with different semantic preservation issues upon transformation. Moreover, refactoring primitives to reference types presents unique challenges as objects in Java cannot share the same memory location. Gravley and Lakhotia [13] tender an approach for identifying enumerated types in C programs that utilize a pre-compiler directive pattern via their declaration characteristics.

7 Conclusions and Future Work

In this paper we have presented a novel, semantic preserving, type inferencing algorithm which migrates legacy Java code employing the weak enum pattern to instead utilize the modern, type-safe *enum* language construct introduced in Java 1.5. We implemented our algorithm as a plug-in for the popular Eclipse IDE and evaluated it on 17 open source applications. Our experiments showed that not only did our tool scale well to large applications but was able to refactor 87% of all fields that could possibly be participating in the weak enum pattern. Prior to being able to publicly distribute our plug-in, we first must address several implementation details including developing a user-friendly interface and exploring potentially faster intermediate representations of code (e.g. Jimple [27]). In the future we also plan to investigate ways of extending our tool to also refactor patterns using constant values of reference types, such as Strings and Dates, as enumeration members.

Acknowledgments

References

[1] D. Bäumer, E. Gamma, and A. Kiezun. Integrating refactoring support into a Java development tool. In *OOPSLA'01 Companion*, October 2001.

[2] J. Bloch. *Effective Java Programming Language Guide*. Prentice Hall PTR, 2001.

[3] G. Bracha and et al. Adding generics to the Java programming language: Public draft specification, version 2.0. Technical Report JSR 014, Java Community Process, June 2003.

[4] J. Dean, D. Grove, and C. Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP*, pages 77–101, 1995.

[5] A. V. Deursen and L. Moonen. Type inference for COBOL systems. In *IEEE Working Conf. on Reverse Engineering*, page 220, 1998.

[6] A. V. Deursen and L. Moonen. Understanding COBOL systems using inferred types. In *IEEE Int. Workshop on Program Comprehension*, page 74, 1999.

[7] A. Donovan, A. Kieżun, M. S. Tschantz, and M. D. Ernst. Converting Java programs to use generic libraries. In *OOPSLA*, pages 15–34, 2004.

[8] P. H. Eidorff and et al. Annodomini: From type theory to year 2000 conversion tool. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, 1999.

[9] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 1999.

[10] R. Fuhrer, F. Tip, A. Kieżun, J. Dolby, and M. Keller. Efficiently refactoring Java applications to use generic libraries. In *ECOOP*, pages 71–96, July 27–29, 2005.

[11] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, Boston, MA, USA, 1995.

[12] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java^{TM} Language Specification (3rd Edition)*. Addison-Wesley, 2005.

[13] J. M. Gravley and A. Lakhotia. Identifying enumeration types modeled with symbolic constants. In *IEEE Working Conf. on Reverse Engineering*, page 227, 1996.

[14] J. Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004.

[15] R. Khatchadourian and et al. Automated refactoring of legacy Java software to enumerated types. Technical Report OSU-CISRC-4/07-TR26, Ohio State University, Apr. 2007.

[16] A. Kiezun, M. D. Ernst, F. Tip, and R. M. Fuhrer. Refactoring for parameterizing Java classes. In *International Conference on Software Engineering*, 2007.

[17] D. E. Knuth. Semantics of context-free languages. *Theory of Computing Systems*, 2(2), 1967.

[18] J. Palsberg and M. I. Schwartzbach. *Object-oriented type systems*. John Wiley and Sons Ltd., Chichester, UK, 1994.

[19] B. C. Pierce. *Types and programming languages*. MIT Press, 2002.

[20] G. Ramalingam, J. Field, and F. Tip. Aggregate structure identification and its application to program analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 119–132, 1999.

[21] F. Steimann, P. Mayer, and A. Meißner. Decoupling classes with inferred interfaces. In *ACM Symposium on Applied Computing*, pages 1404–1408, 2006.

[22] F. Steimann, W. Siberski, and T. Kühne. Towards the systematic use of interfaces in java programming. In *PPPJ*, 2003.

[23] Sun Microsystems. *Java Programming Language: Enhancements in JDK 5*. `java.sun.com/j2se/1.5.0`.

[24] B. D. Sutter, F. Tip, and J. Dolby. Customization of java library classes using type constraints and profile information. In *ECOOP*, pages 585–610, 2004.

[25] F. Tip, R. Fuhrer, J. Dolby, and A. Kieżun. Refactoring techniques for migrating applications to generic Java container classes. Technical Report RC 23238, IBM T.J. Watson Research Center, February 2004.

[26] F. Tip, A. Kieżun, and D. Bäumer. Refactoring for generalization using type constraints. In *OOPSLA*, pages 13–26, Nov 2003.

[27] R. Vallée-Rai and et al. Optimizing Java bytecode using the Soot framework: Is it feasible? In *International Conference on Compiler Construction*, LNCS 1781, pages 18–34, 2000.

[28] D. von Dincklage and A. Diwan. Converting Java classes to use generics. In *OOPSLA*, pages 1–14, 2004.