AMAP: Automatically Mining Abbreviation Expansions in Programs to Enhance Software Maintenance Tools

Emily Hill, Zachary P. Fry, Haley Boyd, Giri Prasad Sridhara, Yana Novikova, Lori Pollock and K. Vijay-Shanker

Computer & Information Sciences
University of Delaware
It's no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as 'I' for 'Locale'.

However, cases where the long form is nowhere near the short form, such as UI, Attr, J, or VK—these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It’s no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It’s no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK — these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It’s no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is nowhere near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It's no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
Abbreviations in Code

It’s no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It’s no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It’s no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It’s no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It’s no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
It's no secret that developers use abbreviations when writing code. In fact, abbreviations are used more often than you might realize.

Consider, for example, this Java code snippet.

Abbreviations with long forms nearby in the code are common, such as ‘l’ for ‘Locale’.

However, cases where the long form is no where near short form, such as UI, Attr, J, or VK -- these are the interesting and more difficult cases.

In fact, sometimes short forms occur more frequently than the long form!
But why do we care? Let me give you a concrete example for concern location. In this example we’re looking for the “delete auction” concern in an auction sniping program.

Although there are many methods relevant to this concern, let me draw your attention to two. The first is ‘delEntry’, which contains the abbreviation ‘del’ for delete in the method name. A simple lexical search will return this method because it contains the word ‘delete’ both in the comments and other identifiers in the method. However, the other relevant method ‘refilterAll’ only refers to ‘delEntry’, and will be missed by a simple lexical search.
Concrete Example: Concern Location

• Looking for code related to “delete auction”

But why do we care? Let me give you a concrete example for concern location. In this example we’re looking for the “delete auction” concern in an auction sniping program.

Although there are many methods relevant to this concern, let me draw your attention to two. The first is ‘delEntry’, which contains the abbreviation ‘del’ for delete in the method name. A simple lexical search will return this method because it contains the word ‘delete’ both in the comments and other identifiers in the method. However, the other relevant method ‘refilterAll’ only refers to ‘delEntry’, and will be missed by a simple lexical search.
But why do we care? Let me give you a concrete example for concern location. In this example we’re looking for the “delete auction” concern in an auction sniping program.

Although there are many methods relevant to this concern, let me draw your attention to two. The first is ‘delEntry’, which contains the abbreviation ‘del’ for delete in the method name. A simple lexical search will return this method because it contains the word ‘delete’ both in the comments and other identifiers in the method. However, the other relevant method ‘refilterAll’ only refers to ‘delEntry’, and will be missed by a simple lexical search.
But why do we care? Let me give you a concrete example for concern location. In this example we’re looking for the “delete auction” concern in an auction sniping program.

Although there are many methods relevant to this concern, let me draw your attention to two. The first is `delEntry`, which contains the abbreviation ‘del’ for delete in the method name. A simple lexical search will return this method because it contains the word ‘delete’ both in the comments and other identifiers in the method. However, the other relevant method `refilterAll` only refers to `delEntry`, and will be missed by a simple lexical search.
Why do we care?

• Improve effectiveness of language-based software tools
• Concern location
• Documentation to source code traceability
• Analysis of software artifacts, e.g., defect reports
• Program Comprehension

Again, why do we care? If we had access to accurate automatic abbreviation expansion techniques, we could improve the effectiveness of natural language-based software tools---tools that use the lexical information in comments and identifiers.

The most obvious application is in program comprehension. When a developer comes across an unfamiliar abbreviation in code, the automatic expansion technique can present likely long forms, instead of the developer having to waste time looking through code for the expansion.
Why do we care?

- Improve effectiveness of language-based software tools
- Concern location
- Documentation to source code traceability
- Analysis of software artifacts, e.g., defect reports
- Program Comprehension

Automatically expanding abbreviations can give tools access to words and associated meanings that were previously meaningless sequences of characters.

Again, why do we care? If we had access to accurate automatic abbreviation expansion techniques, we could improve the effectiveness of natural language-based software tools---tools that use the lexical information in comments and identifiers.

The most obvious application is in program comprehension. When a developer comes across an unfamiliar abbreviation in code, the automatic expansion technique can present likely long forms, instead of the developer having to waste time looking through code for the expansion.
Automatic Abbreviation Expansion

- Given a code segment, identify character sequences that are short forms and determine long form.

To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
## Automatic Abbreviation Expansion

- Given a code segment, identify character sequences that are short forms and determine long form

1. **Split Identifiers:**

```java
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condition, int pressed) {
        if (e.getKeyCode() == KeyEvent.VK_ESCAPE) {
            // We are not editing, forward to filechooser.
            chooser.dispatchEvent(e);
            return true;
        }
        return super.processKeyBinding(ks, e, condition, pressed);
    }

    public Component prepareRenderer(TableCellRenderer renderer, int row, int column) {
        Component comp = super.prepareRenderer(renderer, row, column);
        if (comp instanceof JLabel) {
            // Numbers are right-adjusted, regardless of component orientation
            ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
            // Handle other cases here
        }
    }
}
```

To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
Automatic Abbreviation Expansion

- Given a code segment, identify character sequences that are short forms and determine long form

To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
Automatic Abbreviation Expansion

- Given a code segment, identify character sequences that are short forms and determine long form

1. Split Identifiers:
   - Punctuation
   - Camel case
   - No boundary

```java
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyPressEvent(KeyStroke ks, KeyEvent e, int condition) {
        if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
            chooser.dispatchEvent(e);
            return true;
        }
        return super.processKeyPressEvent(ks, e, condition, pressed);
    }

    public Component prepareRenderer(TableCellRenderer renderer, int row, int column) {
        Component comp = super.prepareRenderer(renderer, row, column);
        if (comp instanceof JLabel) {
            if (convertColumn) {
                // Numbers are right-adjusted in our column orientation
                ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
            } else {
```

To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
Automatic Abbreviation Expansion

- Given a code segment, identify character sequences that are short forms and determine long form

1. Split Identifiers:
   - Punctuation
   - Camel case
   - No boundary
   - e.g., strlen

```java
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condition) {
        if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
            chooser.dispatchEvent(e);
            return true;
        }
        return super.processKeyBinding(ks, e, condition, pressed);
    }

global Component prepareRenderer(TableCellRenderer renderer, int row, int column) {
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
        if (convertColumn) {
            // Numbers are right-adjusted for a right-oriented cell
            ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
            // no boundary
        }
    }
}
```

To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
Automatic Abbreviation Expansion

- Given a code segment, identify character sequences that are short forms and determine long form.

To identify character sequences, or tokens, in code boils down to splitting the identifiers. The hardest case is no boundary cases. If not properly split, abbreviations will be missed, such as in string length.
The simplest approach is a manually created dictionary of common short forms in code. However, this has a couple drawbacks.
Dictionary Approach

• **Simplest solution:** manually create a lookup table of common short forms in code

The simplest approach is a manually created dictionary of common short forms in code. However, this has a couple drawbacks.
Dictionary Approach

- **Simplest solution:** manually create a lookup table of common short forms in code
- Vocabulary evolves over time, must maintain table

The simplest approach is a manually created dictionary of common short forms in code. However, this has a couple drawbacks.
The simplest approach is a manually created dictionary of common short forms in code. However, this has a couple drawbacks.
The simplest approach is a manually created dictionary of common short forms in code. However, this has a couple drawbacks.
The simplest approach is a manually created dictionary of common short forms in code. However, this has a couple drawbacks.
Importance of Context
Importance of Context

• Same abbreviation can have different expansions within the same program
Importance of Context

• Same abbreviation can have different expansions within the same program

• inst $\rightarrow$ instance, instruction, instantiate, install?
Importance of Context

• Same abbreviation can have different expansions *within the same program*

• inst → instance, instruction, instantiate, install?

• cmp → component or compare?
Importance of Context

• Same abbreviation can have different expansions within the same program
  • inst → instance, instruction, instantiate, install?
  • cmp → component or compare?

```java
public static IdentifiableFactory makeJavaSerializationComponentFactory() {
    return new EncapsulationFactoryBase(
        ORBConstants.TAG_JAVA_SERIALIZATION_ID) {
            public Identifiable readContents(InputStream in) {
                byte version = in.read_octet();
                Identifiable cmp = new JavaSerializationComponent(version);
                return cmp;
            }
        }
    }
```
Importance of Context

• Same abbreviation can have different expansions within the same program
  • inst → instance, instruction, instantiate, install?
  • cmp → component or compare?

```java
public static IdentifiableFactory makeJavaSerializationComponentFactory() {
    return new EncapsulationFactoryBase(
        ORBConstants.TAG_JAVA_SERIALIZATION_ID)
            {
        public Identifiable readContents(InputStream in) {
            byte version = in.read_octet();
            Identifiable cmp = new JavaSerializationComponent(version);
            return cmp;
        }
    }
}
```

/* Continue through string-match values until we find one that is either greater than the current key, or equal to it. In the latter case, remove the key. */
```
int cmp = +1;
while ((cmp = nextString.compare薰a(key)) < 0) {
    if (stringIterator.hasNext())
        nextString = (String) stringIterator.next();
    else
        nextString = sentinelKey;
}
if (cmp == 0) {
    keyIterator.remove();
    continue keys;
}
```
Importance of Context

• Same abbreviation can have different expansions within the same program
  • inst $\rightarrow$ instance, instruction, instantiate, install?
  • cmp $\rightarrow$ component or compare?

```
public static IdentifiableFactory makeJavaSerializationComponentFactory()
    return new EncapsulationFactoryBase(
        ORBConstants.TAG_JAVA.Serialization_ID)
    
public Identifiable readContents(InputStream in)
    byte version = in.read_octet();
    Identifiable cmp = new JavaSerializationComponent(version);
    return cmp;

/* Continue through string-match values until we find one that is either greater than the current key, or equal to it. In the latter case, remove the key. */
int cmp = +1;
while ((cmp = nextString.compare(key)) < 0) {
    if (stringIterator.hasNext())
        nextString = (String) stringIterator.next();
    return sentinelKey;
}
```
In this approach we are doing text mining, as opposed to the more general data mining.
In this approach we are doing text mining, as opposed to the more general data mining.
Mining Abbreviation Expansions

- **Key Insight:** Mine long forms from source code
- **Given a token in a code segment (i.e., method),**

In this approach we are doing text mining, as opposed to the more general data mining.
Mining Abbreviation Expansions

- **Key Insight:** Mine long forms from source code
- Given a token in a code segment (i.e., method),
  1. Identify if token is non-dictionary word (i.e., short form candidate)
    - Use an English ispell-based dictionary
    - For details, see paper

In this approach we are doing text mining, as opposed to the more general data mining.
Mining Abbreviation Expansions

- **Key Insight:** Mine long forms from source code
- Given a token in a code segment (i.e., method),
  1. Identify if token is non-dictionary word (i.e., short form candidate)
     - Use an English ispell-based dictionary
     - For details, see paper
  2. Identify potential long forms

In this approach we are doing text mining, as opposed to the more general data mining.
Mining Abbreviation Expansions

**Key Insight:** Mine long forms from source code

Given a token in a code segment (i.e., method),

1. Identify if token is non-dictionary word (i.e., short form candidate)
   - Use an English ispell-based dictionary
   - For details, see paper

2. Identify potential long forms

3. Select most appropriate long form (see paper)

In this approach we are doing text mining, as opposed to the more general data mining.
Key Insight: Mine long forms from source code

Given a token in a code segment (i.e., method),
1. Identify if token is non-dictionary word (i.e., short form candidate)
   • Use an English ispell-based dictionary
   • For details, see paper
2. Identify potential long forms
3. Select most appropriate long form (see paper)

In this approach we are doing text mining, as opposed to the more general data mining.
We looked at hundreds of example abbreviations, and observed a number of types of dictionary words.

The difference between combination and no boundary is that in combination, one or more of the concatenated tokens is an abbreviation.

It should be noted that our technique handles both combination and no boundary cases, as well as misspellings that are instances of dropped letter.
Step 2: Search for potential long forms

Types of Non-Dictionary Words

- Single-Word
- Prefix (attr, obj, param, i)
- Dropped Letter (src, evt, msg)

We looked at hundreds of example abbreviations, and observed a number of types of dictionary words.

The difference between combination and no boundary is that in combination, one or more of the concatenated tokens is an abbreviation.

It should be noted that our technique handles both combination and no boundary cases, as well as misspellings that are instances of dropped letter.
Types of Non-Dictionary Words

• Single-Word
  • Prefix (attr, obj, param, i)
  • Dropped Letter (src, evt, msg)

• Multi-Word
  • Acronyms (ftp, xml, [type names])
  • Combination (println, doctype)

We looked at hundreds of example abbreviations, and observed a number of types of dictionary words.

The difference between combination and no boundary is that in combination, one or more of the concatenated tokens is an abbreviation.

It should be noted that our technique handles both combination and no boundary cases, as well as misspellings that are instances of dropped letter.
Types of Non-Dictionary Words

- **Single-Word**
  - *Prefix* (attr, obj, param, i)
  - *Dropped Letter* (src, evt, msg)

- **Multi-Word**
  - *Acronyms* (ftp, xml, [type names])
  - *Combination* (println, doctype)

- **Others**
  - *No boundary* (saveas, filesize)
  - *Misspelling* (instanciation, zzzcatzzzzdogzzz)

We looked at hundreds of example abbreviations, and observed a number of types of dictionary words.

The difference between combination and no boundary is that in combination, one or more of the concatenated tokens is an abbreviation.

It should be noted that our technique handles both combination and no boundary cases, as well as misspellings that are instances of dropped letter.
For each of these abbreviation types, we search for long forms in the code by using regular expressions. For the actual patterns, please see the paper.

Notice that dropped letter and combination can match many more meaningless sequences.
Step 2: Search for potential long forms

Long Form Search Patterns

Given short form \texttt{arg}, we search for regular expressions matching long forms in code:

For each of these abbreviation types, we search for long forms in the code by using regular expressions. For the actual patterns, please see the paper.

Notice that dropped letter and combination can match many more meaningless sequences.
Step 2: Search for potential long forms

Long Form Search Patterns

Given short form arg, we search for regular expressions matching long forms in code:

- **Single-Word**
- **Prefix argument**
- **Dropped letter average**

For each of these abbreviation types, we search for long forms in the code by using regular expressions. For the actual patterns, please see the paper.

Notice that dropped letter and combination can match many more meaningless sequences.
Step 2: Search for potential long forms

Long Form Search Patterns

Given short form \texttt{arg}, we search for regular expressions matching long forms in code:

- **Single-Word**
  - Prefix \texttt{argument}
  - Dropped letter \texttt{average}

- **Multi-Word**
  - Acronym \texttt{attribute random group}
  - Combination \texttt{access rights}

For each of these abbreviation types, we search for long forms in the code by using regular expressions. For the actual patterns, please see the paper.

Notice that dropped letter and combination can match many more meaningless sequences.
Now that we know these patterns, how do we determine what abbreviation type a given short form is?

We were pretty sure conservative acronym and prefix should go first, followed by greedier dropped letter and combination -- but in what order?

After looking at hundreds of examples, we determined the following order: acronym → prefix → dropped letter → combination
Now that we know these patterns, how do we determine what abbreviation type a given short form is?

We were pretty sure conservative acronym and prefix should go first, followed by greedier dropped letter and combination -- but in what order?

After looking at hundreds of examples, we determined the following order: acronym → prefix → dropped letter → combination
Now that we know these patterns, how do we determine what abbreviation type a given short form is?

We were pretty sure conservative acronym and prefix should go first, followed by greedier dropped letter and combination -- but in what order?

After looking at hundreds of examples, we determined the following order: acronym $\rightarrow$ prefix $\rightarrow$ dropped letter $\rightarrow$ combination
Now that we know these patterns, how do we determine what abbreviation type a given short form is?

We were pretty sure conservative acronym and prefix should go first, followed by greedier dropped letter and combination -- but in what order?

After looking at hundreds of examples, we determined the following order: acronym → prefix → dropped letter → combination
Step 2: Search for potential long forms

Search Pattern Order

- Search by abbreviation type:

<table>
<thead>
<tr>
<th>Multi-Word</th>
<th>Single Word</th>
</tr>
</thead>
<tbody>
<tr>
<td>Conservative</td>
<td>Prefix</td>
</tr>
<tr>
<td>Acronym</td>
<td>Dropped Letter</td>
</tr>
<tr>
<td>Combination</td>
<td></td>
</tr>
</tbody>
</table>

How do we identify potential long forms for each type?

Now that we know these patterns, how do we determine what abbreviation type a given short form is?

We were pretty sure conservative acronym and prefix should go first, followed by greedier dropped letter and combination -- but in what order?

After looking at hundreds of examples, we determined the following order: acronym → prefix → dropped letter → combination
Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)

```
/**
 * Copies characters from this string into the destination character array.
 * @param srcBegin index of the first character in the string to copy.
 * @param srcEnd index after the last character in the string to copy.
 * @param dst the destination array.
 * @param dstBegin the start offset in the destination array.
 * @exception NullPointerException if <code>dst</code> is <code>null</code>
 */
public abstract void getChars(int srcBegin, int srcEnd, char dst[],
                               int dstBegin);
```

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)

```java
/**
 * Copies characters from this string into the destination character array.
 * @param srcBegin index of the first character in the string to copy.
 * @param srcEnd index after the last character in the string to copy.
 * @param dst the destination array.
 * @param dstBegin the start offset in the destination array.
 * @exception NullPointerException if <code>dst</code> is <code>null</code>
 */
public abstract void copy(String src, int srcBegin, int srcEnd, int dstBegin, char[] dst);

public static ClassLoader resolveServerClassLoader(Map env, MBeanServer mbs)
```

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form "scopes" until clear winner:

1. JavaDoc (parameter name=sf)
Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)

```
private void circulationPump(ControlFlowGraph cfg, InstructionContext start,
    final Random random = new Random();
    InstructionContextQueue icq = new InstructionContextQueue();

    Object source = event.getSource();
    if (source instanceof Component) {
        Component comp = (Component)source;
        comp.dispatchEvent(event);
    } else if (source instanceof MenuComponent) {
```

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)
3. Method Name (If only)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)
3. Method Name (If only)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)
3. Method Name (If only)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)
3. Method Name (lf only)
4. Statements (sf & lf)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)
3. Method Name (lf only)
4. Statements (sf & lf)

```
final int nConstructors = constructors.size();
final int nArgs = _arguments.size();
final Vector argsType = typeCheckArgs(stable);
```
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)
3. Method Name (lf only)
4. Statements (sf & lf)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Step 2: Search for potential long forms

Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)
3. Method Name (lf only)
4. Statements (sf & lf)
5. Referenced identifiers and string literals
6. Method comments
7. Class comments (prefix & acronym only)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
Our Scoped Approach

Inspired by static scoping, start from method containing short form and search increasingly broader long form “scopes” until clear winner:

1. JavaDoc (parameter name=sf)
2. Type Names of declared variables (name=sf)

What if no long form found in any scope for any abbreviation type?

3. Referenced identifiers and string literals
4. Statements (sf & lf)
5. Method comments
6. Class comments (prefix & acronym only)

Our approach is inspired by, but doesn’t follow, static variable scoping. We start with locations in the code where we have high confidence in the long forms, and increase the scopes to less reliable locations in a best effort to find long forms.
In the second example, the long form ‘arguments’ didn’t appear anywhere in the class file.

Although after the Java MFE you could add in a hand-tuned list, the focus of this work was to see how well we could do with a purely automated approach.
Step 2: Search for potential long forms

What if no long form found?

• Fall back to Most Frequent Expansion (MFE)

In the second example, the long form ‘arguments’ didn’t appear anywhere in the class file.

Although after the Java MFE you could add in a hand-tuned list, the focus of this work was to see how well we could do with a purely automated approach.
Step 2: Search for potential long forms

What if no long form found?

- Fall back to Most Frequent Expansion (MFE)
- MFE leverages successful local expansions and applies throughout the program

In the second example, the long form ‘arguments’ didn’t appear anywhere in the class file.

Although after the Java MFE you could add in a hand-tuned list, the focus of this work was to see how well we could do with a purely automated approach.
Step 2: Search for potential long forms

What if no long form found?

- Fall back to Most Frequent Expansion (MFE)
- MFE leverages successful local expansions and applies throughout the program

```java
final int nConstructors = constructors.size();
final int nArgs = _arguments.size();
final Vector argsType = typeCheckArgs(stable);
```

In the second example, the long form ‘arguments’ didn’t appear anywhere in the class file.

Although after the Java MFE you could add in a hand-tuned list, the focus of this work was to see how well we could do with a purely automated approach.
Step 2: Search for potential long forms

What if no long form found?

- Fall back to Most Frequent Expansion (MFE)
- MFE leverages successful local expansions and applies throughout the program

```java
final int nConstructors = constructors.size();
final int nArgs = _arguments.size();
final Vector argsType = typeCheckArgs(stable);
```

```java
public boolean hasReferenceArgs() {
    return _left.getType() instanceof ReferenceType ||
    _right.getType() instanceof ReferenceType;
}

public boolean hasNodeArgs() {
    return _left.getType() instanceof NodeType ||
    _right.getType() instanceof NodeType;
}

public boolean hasNodeSetArgs() {
    return _left.getType() instanceof NodeSetType ||
    _right.getType() instanceof NodeSetType;
}
```

In the second example, the long form ‘arguments’ didn’t appear anywhere in the class file.

Although after the Java MFE you could add in a hand-tuned list, the focus of this work was to see how well we could do with a purely automated approach.
Step 2: Search for potential long forms

What if no long form found?

- Fall back to Most Frequent Expansion (MFE)
- MFE leverages successful local expansions and applies throughout the program

1. **Program**: provides domain knowledge

In the second example, the long form ‘arguments’ didn’t appear anywhere in the class file.

Although after the Java MFE you could add in a hand-tuned list, the focus of this work was to see how well we could do with a purely automated approach.
Step 2: Search for potential long forms

What if no long form found?

- Fall back to Most Frequent Expansion (MFE)
- MFE leverages successful local expansions and applies throughout the program

1. **Program**: provides domain knowledge
2. **Java**: more general programming knowledge

In the second example, the long form ‘arguments’ didn’t appear anywhere in the class file.

Although after the Java MFE you could add in a hand-tuned list, the focus of this work was to see how well we could do with a purely automated approach.
Step 2: Search for potential long forms

What if no long form found?

- Fall back to Most Frequent Expansion (MFE)
- MFE leverages successful local expansions and applies throughout the program

1. Program: provides domain knowledge
2. Java: more general programming knowledge
   - Option: hand-tuned common abbreviation list

In the second example, the long form ‘arguments’ didn’t appear anywhere in the class file.

Although after the Java MFE you could add in a hand-tuned list, the focus of this work was to see how well we could do with a purely automated approach.
To evaluate our scoped approach, we compared our technique with 3 others.
Experimental Evaluation:
Design

- **Variable:** Abbreviation expansion technique
  - Our scope approach (long form per method)
  - Program MFE (long form per program)
  - Java MFE (same long form for all programs)
  - LFB (Lawrie, Field, Binkley from SCAM ’07)

To evaluate our scoped approach, we compared our technique with 3 others.
Experimental Evaluation: Design

- **Variable:** Abbreviation expansion technique
  - Our scope approach (long form per method)
  - Program MFE (long form per program)
  - Java MFE (same long form for all programs)
  - LFB (Lawrie, Field, Binkley from SCAM ’07)
    - Search ispell dictionary, rather than just those occurring in code
    - Search for dropped letter (incl. prefix) and acronym

To evaluate our scoped approach, we compared our technique with 3 others.
Experimental Evaluation:
Design
Experimental Evaluation: Design

- **Gold Set:**
  - 250 randomly selected non-dictionary words from 5 large, open source Java programs (300-400K NCLOC)
  - 2 programmers investigated non-dictionary words in context and determined long forms
Experimental Evaluation: Design

- **Gold Set:**
  - 250 randomly selected non-dictionary words from 5 large, open source Java programs (300-400K NCLOC)
  - 2 programmers investigated non-dictionary words in context and determined long forms

- **Measure:** **accuracy** (% correctly expanded short forms)

\[
\text{accuracy} = \frac{\# \text{ correct expansions}}{250}
\]
Experimental Evaluation:
Results

- Histogram of number of correct expansions for each abbreviation expansion technique, broken down by type.
- For reference, included results of returning no expansion as well as the type breakdown for the ideal set (maximum possible number correct, 250)
- Below each bar is the overall accuracy, the two bold accuracy numbers don’t contain CW to be fair to LFB
Experimental Evaluation: Results

- Scope 57% more accurate than LFB

Histogram of number of correct expansions for each abbreviation expansion technique, broken down by type.
- For reference, included results of returning no expansion as well as the type breakdown for the ideal set (maximum possible number correct, 250)
- Below each bar is the overall accuracy, the two bold accuracy numbers don’t contain CW to be fair to LFB
• Scope 57% more accurate than LFB
• Scope 30% more accurate than Java MFE

Histogram of number of correct expansions for each abbreviation expansion technique, broken down by type.

For reference, included results of returning no expansion as well as the type breakdown for the ideal set (maximum possible number correct, 250)

Below each bar is the overall accuracy, the two bold accuracy numbers don’t contain CW to be fair to LFB
Experimental Evaluation: Results

- Scope 57% more accurate than LFB
- Scope 30% more accurate than Java MFE
- Program MFE acceptable approximation when speed more important than accuracy

- Histogram of number of correct expansions for each abbreviation expansion technique, broken down by type.
- For reference, included results of returning no expansion as well as the type breakdown for the ideal set (maximum possible number correct, 250)
- Below each bar is the overall accuracy, the two bold accuracy numbers don’t contain CW to be fair to LFB
Experimental Evaluation: Results

- Scope 57% more accurate than LFB
- Scope 30% more accurate than Java MFE
- Program MFE acceptable approximation when speed more important than accuracy
- Room for improvement:
  AC: Acronym,
  CW: Combination

- Histogram of number of correct expansions for each abbreviation expansion technique, broken down by type.
- For reference, included results of returning no expansion as well as the type breakdown for the ideal set (maximum possible number correct, 250)
- Below each bar is the overall accuracy, the two bold accuracy numbers don’t contain CW to be fair to LFB
Conclusions & Future Work

- **Scope 57% more accurate than state of art LFB**
- **Mine** even larger set of programs
  (e.g., Java programs from Sourceforge)
- **Evaluate** within a software maintenance tool
- **Extend** to other programming languages & natural languages in addition to English
Conclusions & Future Work

• Scope 57% more accurate than state of art LFB
• Mine even larger set of programs (e.g., Java programs from Sourceforge)
• Evaluate within a software maintenance tool
• Extend to other programming languages & natural languages in addition to English

www.cis.udel.edu/~hill/amap

This work was supported by an NSF Graduate Research Fellowship and Award CCF-0702401.