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

Emily Hill, Zachary P. Fry, Haley Boyd, Giriprasad Sridhara,
Yana Novikova, Lori Pollock and K. Vijay-Shanker
Computer & Information Sciences
University of Delaware

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!

Abbreviations in Code

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

short form

long form

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

short form

long form

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

short form

long form

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

short form

long form

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

short form

long form

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

short form

long form

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

short form

long form

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.LEADING);
        }
    }
    return comp;
```


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

short form

long form

```
public JPanel createDetailsView() {
final JFileChooser chooser = getFileChooser();

    Locale l = chooser.getLocale();
fileNameHeaderText = UIManager.getString("FileChooser.fileNameHeaderText",l
fileSizeHeaderText = UIManager.getString("FileChooser.fileSizeHeaderText",l
fileTypeHeaderText = UIManager.getString("FileChooser.fileTypeHeaderText",l
fileDateHeaderText = UIManager.getString("FileChooser.fileDateHeaderText",l
fileAttrHeaderText = UIManager.getString("FileChooser.fileAttrHeaderText",l

JPanel p = new JPanel(new BorderLayout());

DetailsTableModel detailsTableModel = new DetailsTableModel(chooser);

final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // We are not editing, forward to filechooser.
        chooser.dispatchEvent(e);
    }
    }

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

Abbreviations can occur more frequently than long form

In Java 2: `num` (5,226) more frequent than `number` (4,314)

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!

Concrete Example: Concern Location

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.

Concrete Example: Concern Location

- Looking for code related to “delete auction”

```
/**
 * Delete an auction entry, using that auction entry to match against.
 * This also tells the auction entry to unregister itself!
 *
 * @param inEntry - The auction entry to delete.
 */
public void delEntry(AuctionEntry inEntry) {
  boolean removedAny = _tSort.delete(inEntry);

  if(removedAny) {
    AuctionServerManager.getInstance().delete_entry(inEntry);
  }
}
```


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"

```
/**
 * Delete an auction entry, using that auction entry to match against.
 * This also tells the auction entry to unregister itself!
 *
 * @param inEntry - The auction entry to delete.
 */
public void delEntry(AuctionEntry inEntry) {
  boolean removedAny = _tSort.delete(inEntry);

  if(removedAny) {
    AuctionServerManager.getInstance().delete_entry(inE
  }
}
```

```
public void refilterAll(boolean clearCurrent) {
  for(int i=_tSort.getRowCount()-1; i>=0; i--) {
    AuctionEntry ae = (AuctionEntry) _tSort.getValueAt(i, -1);
    if (clearCurrent) {
      delEntry(ae);
    } else {
      ae.setCategory(null);
      FilterManager.getInstance().refilterAuction(ae, false);
    }
  }
}
```


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.

Automatic Abbreviation Expansion

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

```
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component 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

```
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
```

1. Split Identifiers:

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

```
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
```

1. Split Identifiers:
   - Punctuation

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

```
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnIndexToModel(column) == COLUMN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
```

1. Split Identifiers:
   - Punctuation
   - Camel case

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

```
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnI                    MN_FILESIZE) {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
```

no boundary

1. Split Identifiers:
   - Punctuation
   - Camel case
   - 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.

Automatic Abbreviation Expansion

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

```
final JTable detailsTable = new JTable(detailsTableModel) {
    // Handle Escape key events here
    protected boolean processKeyBinding(KeyStroke ks, KeyEvent e, int condi
    if (e.getKeyCode() == KeyEvent.VK_ESCAPE && getCellEditor() == null) {
        // 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, i
    Component comp = super.prepareRenderer(renderer, row, column);
    if (comp instanceof JLabel) {
                if (convertColumnI                    MN_FILESIZE {
        // Numbers are right-adjusted, regardless of component orientation
        ((JLabel)comp).setHorizontalAlignment(SwingConstants.RIGHT);
        } else {
```

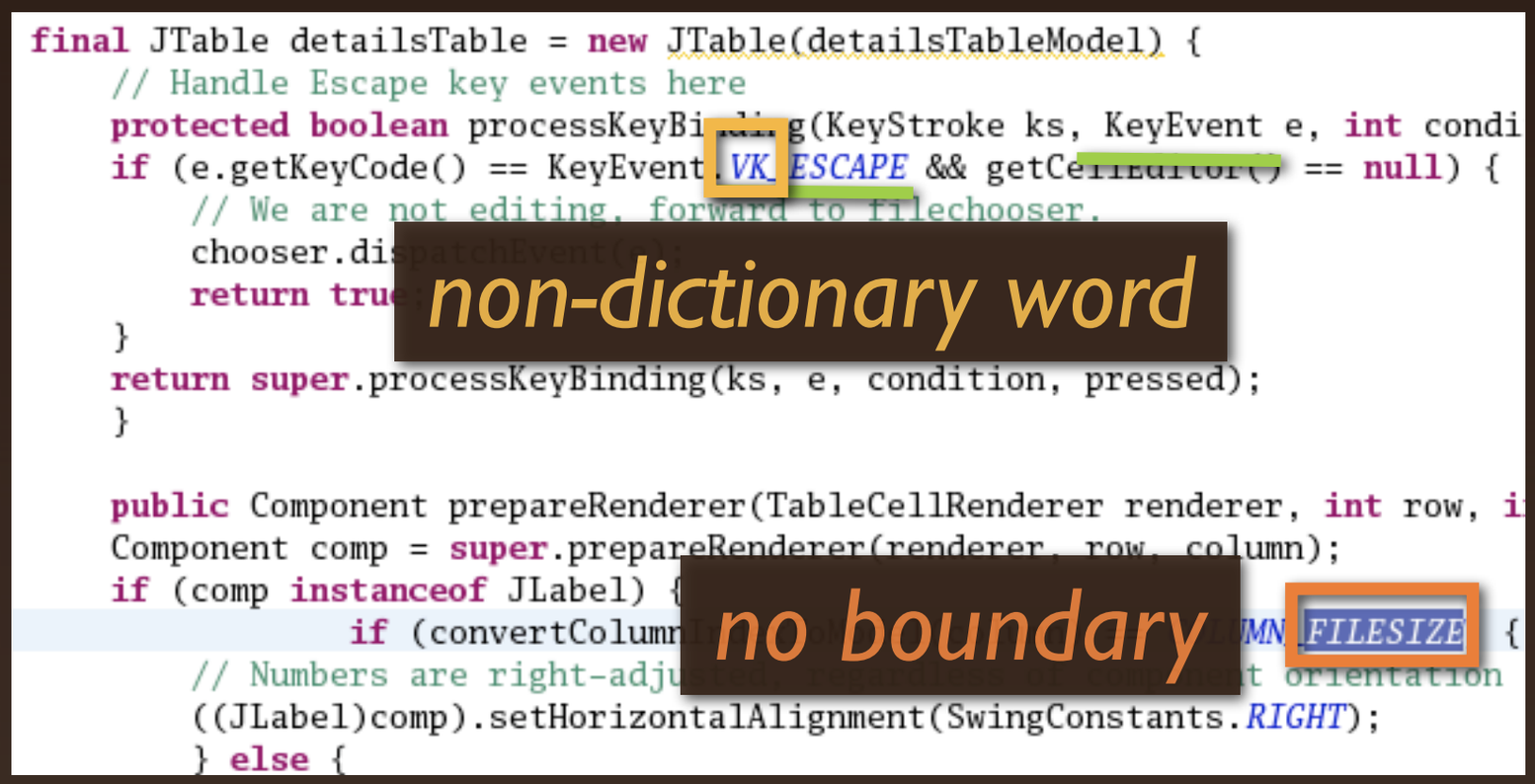
no boundary

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

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

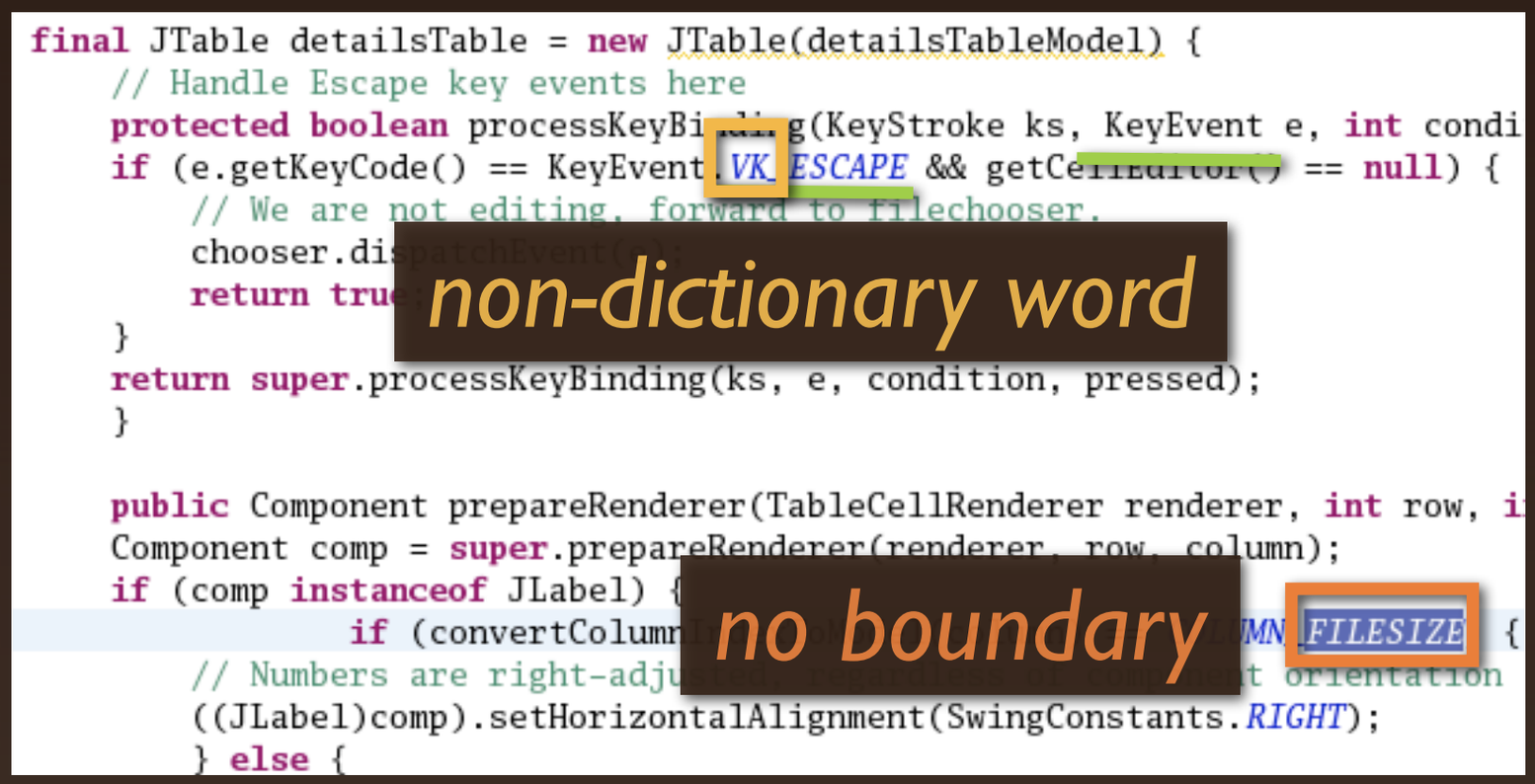


1. Split Identifiers:
   - Punctuation
   - Camel case
   - No boundary
     - e.g., `strlen`
2. Identify non-dictionary words

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`
2. Identify non-dictionary words
3. 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.

Dictionary Approach

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.

Dictionary Approach

- Simplest solution: manually create a lookup table of common short forms in code
  - Vocabulary evolves over time, must maintain table
  - Same abbreviation can have different expansions depending on domain:

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

Dictionary Approach

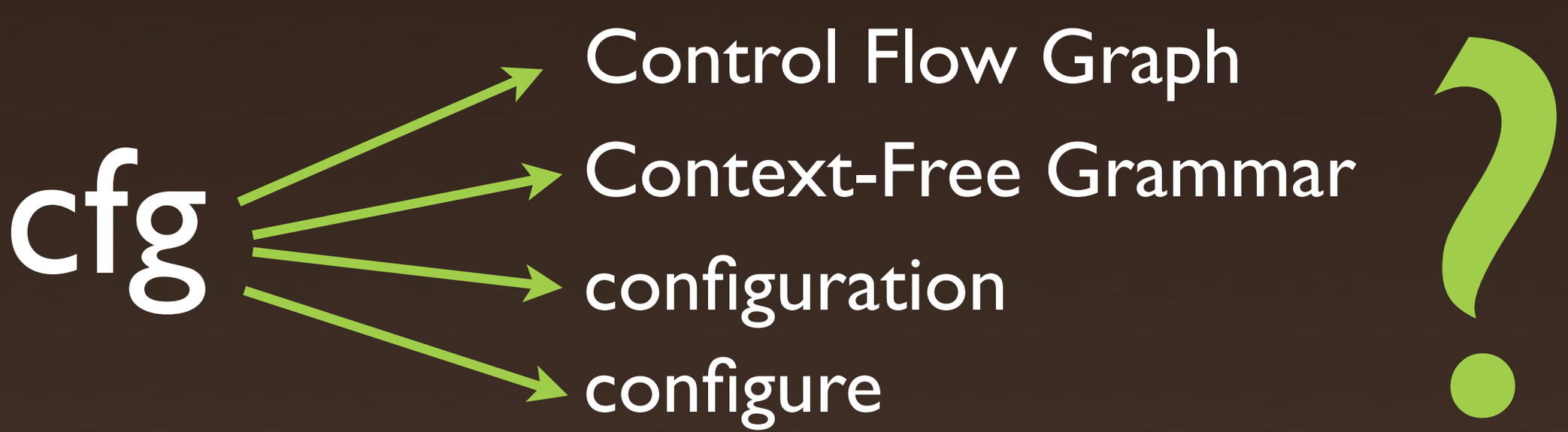
- Simplest solution: manually create a lookup table of common short forms in code
  - Vocabulary evolves over time, must maintain table
  - Same abbreviation can have different expansions depending on domain:

cfg ?

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
- Same abbreviation can have different expansions depending on domain:



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 → 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?

```
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?

```
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(vers
    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.compareTo(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 → instance, instruction, instantiate, install?
  - cmp → 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(vers
    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.compareTo(key)) < 0) {
if (stringIterator.hasNext())
    nextString = (String) stringIterator.next();
ntinelKey;
}
emove();
ntinue keys;
}
```

Need to use abbreviation context to correctly identify long form

Mining Abbreviation Expansions

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

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.

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.

Step 2: Search for potential long forms

Types of Non-Dictionary Words

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.

Step 2: Search for potential long forms

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.

Step 2: Search for potential long forms

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, zzzcatzzzdogzzz)

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

Long Form Search Patterns

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:

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 *arg*ument
  - Dropped letter *a*ve*r*a*g*e

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 *arg*ument
  - Dropped letter *a*ve*r*a*g*e
- Multi-Word
  - Acronym *a*ttribute *r*andom *g*roup
  - Combination *a*ccess *r*i*g*hts

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

Search Pattern Order

- Search by abbreviation 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

Step 2: Search for potential long forms

Search Pattern Order

- Search by abbreviation 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

Step 2: Search for potential long forms

Search Pattern Order

- Search by abbreviation type:

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

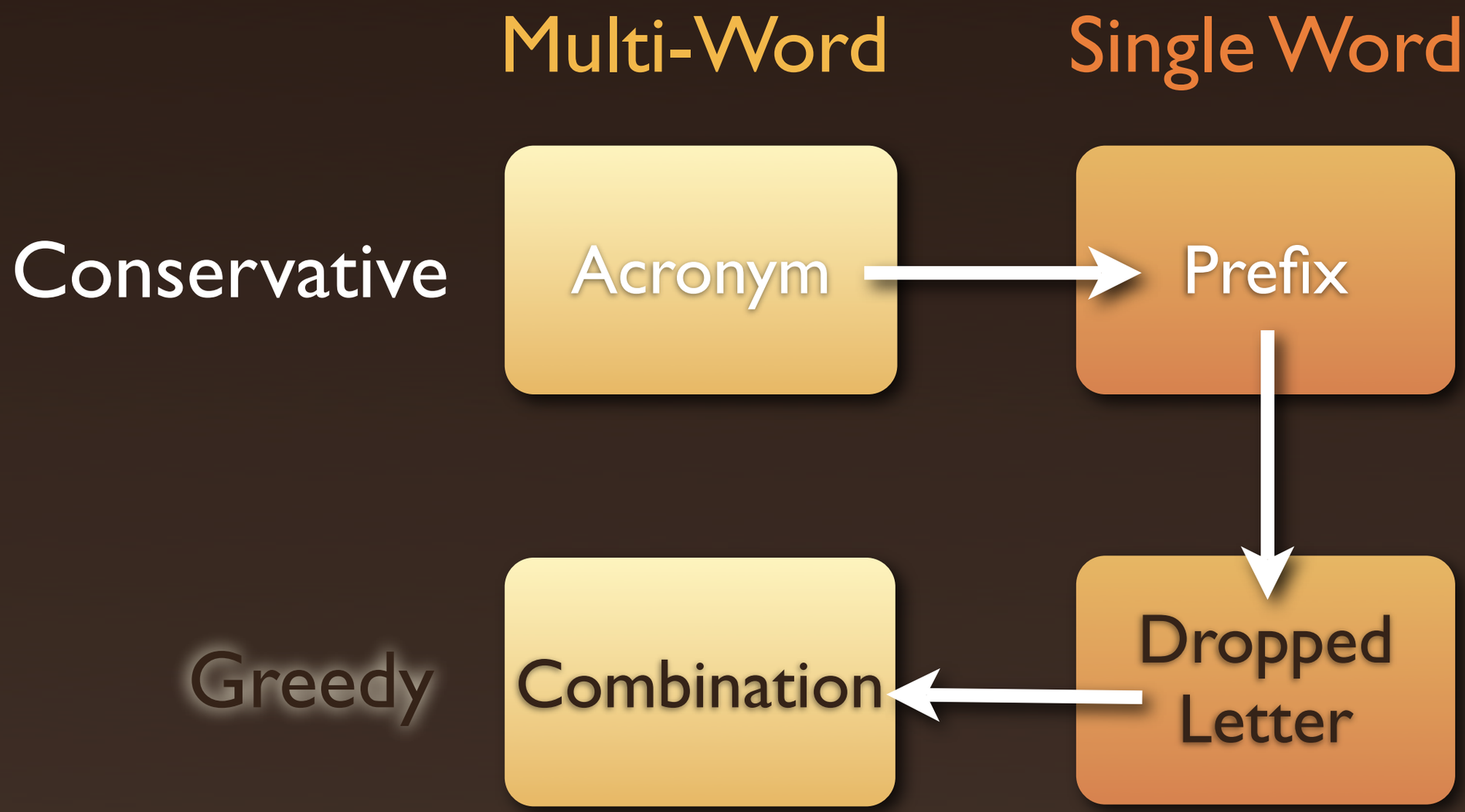
We were pretty sure conservative acronym and prefix should go first, followed by greedier dropped letter and combination -- but in what order?

After looking at hundreds of examples, we determined the following order:
acronym → prefix → dropped letter → combination

Step 2: Search for potential long forms

Search Pattern Order

- Search by abbreviation type:



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

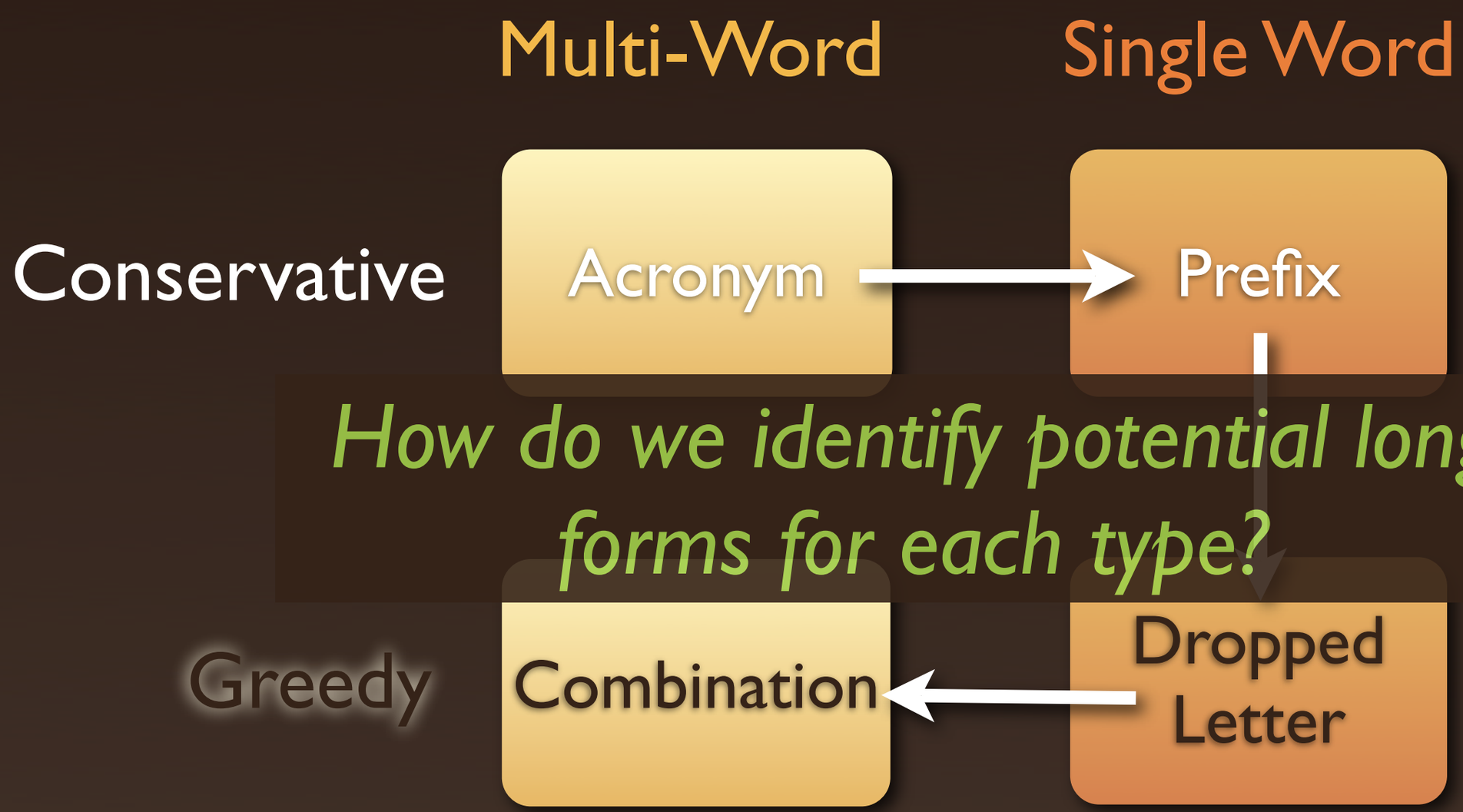
We were pretty sure conservative acronym and prefix should go first, followed by greedier dropped letter and combination -- but in what order?

After looking at hundreds of examples, we determined the following order:
acronym → prefix → dropped letter → combination

Step 2: Search for potential long forms

Search Pattern Order

- Search by abbreviation 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

Step 2: Search for potential long forms

Our Scoped Approach

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:

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 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)

```
/**
 * 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 v
```

```
     * @param env Environment attributes.
     * @param mbs The MBeanServer for which the connector server provides
     * remote access.
     *
     */
    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 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();
```


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.

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.

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)

```
public void setBarCodeImg(int type, String text, boolean showText, boolean checkSum){

    StringBuffer bcCall = new StringBuffer("it.businesslogic.ireport.barcode.BcImage.g
    //boolean isFormula = text.trim().startsWith("$");
    bcCall.append(type);
```


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.

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)

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


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)

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.

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. Referenced identifiers and string literals
6. Method comments
7. Class comments (prefix & acronym only)

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

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

What if no long form found?

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

```
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

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

```
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.

Experimental Evaluation:
Design

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

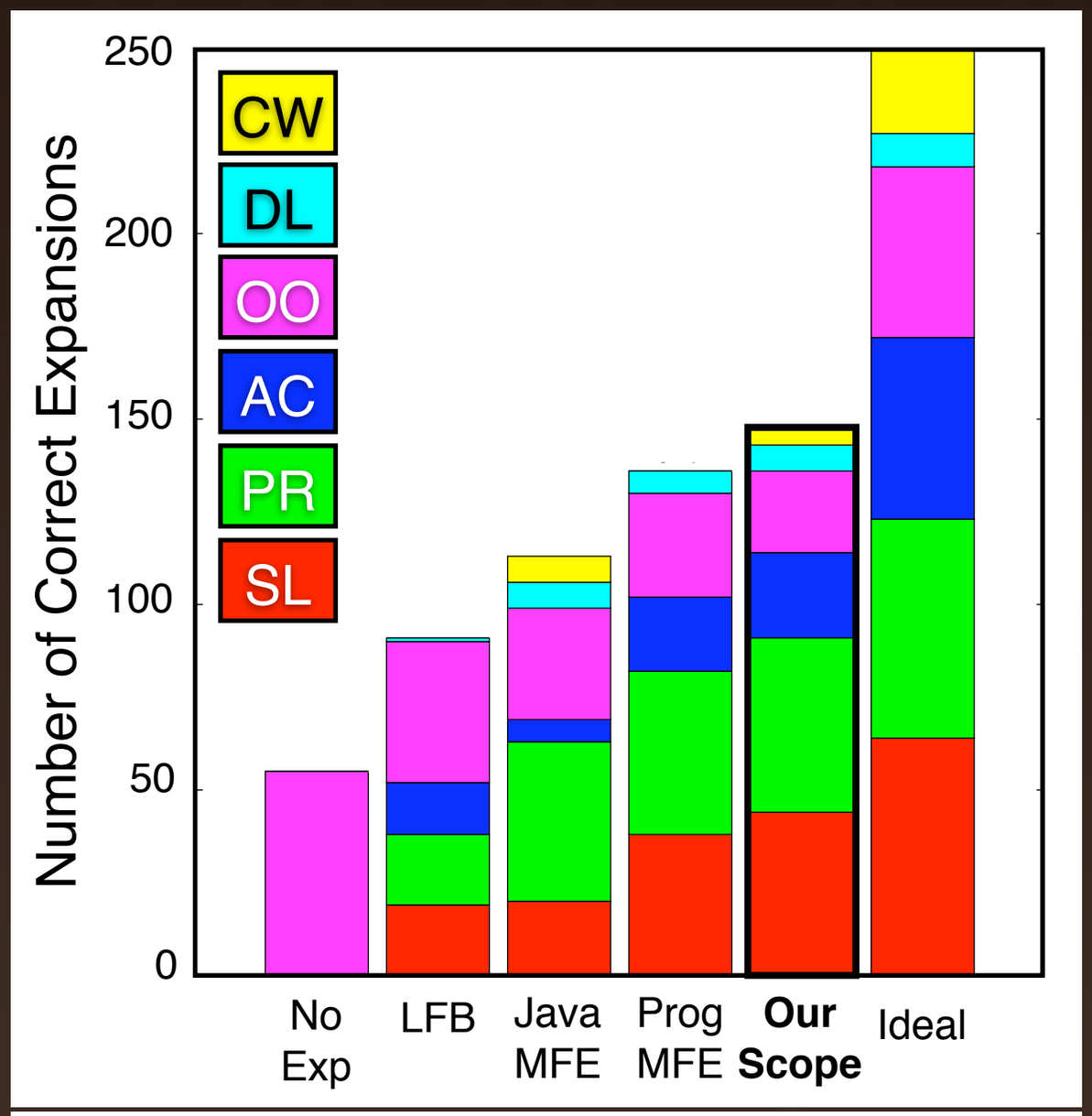
- Gold Set:
  - 250 randomly selected non-dictionary words from 5 large, open source Java programs (300-400K NCLOC)
  - 2 programmers investigated non-dictionary words in context and determined long forms

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)

$$\text{accuracy} = \frac{\#\text{ correct expansions}}{250}$$

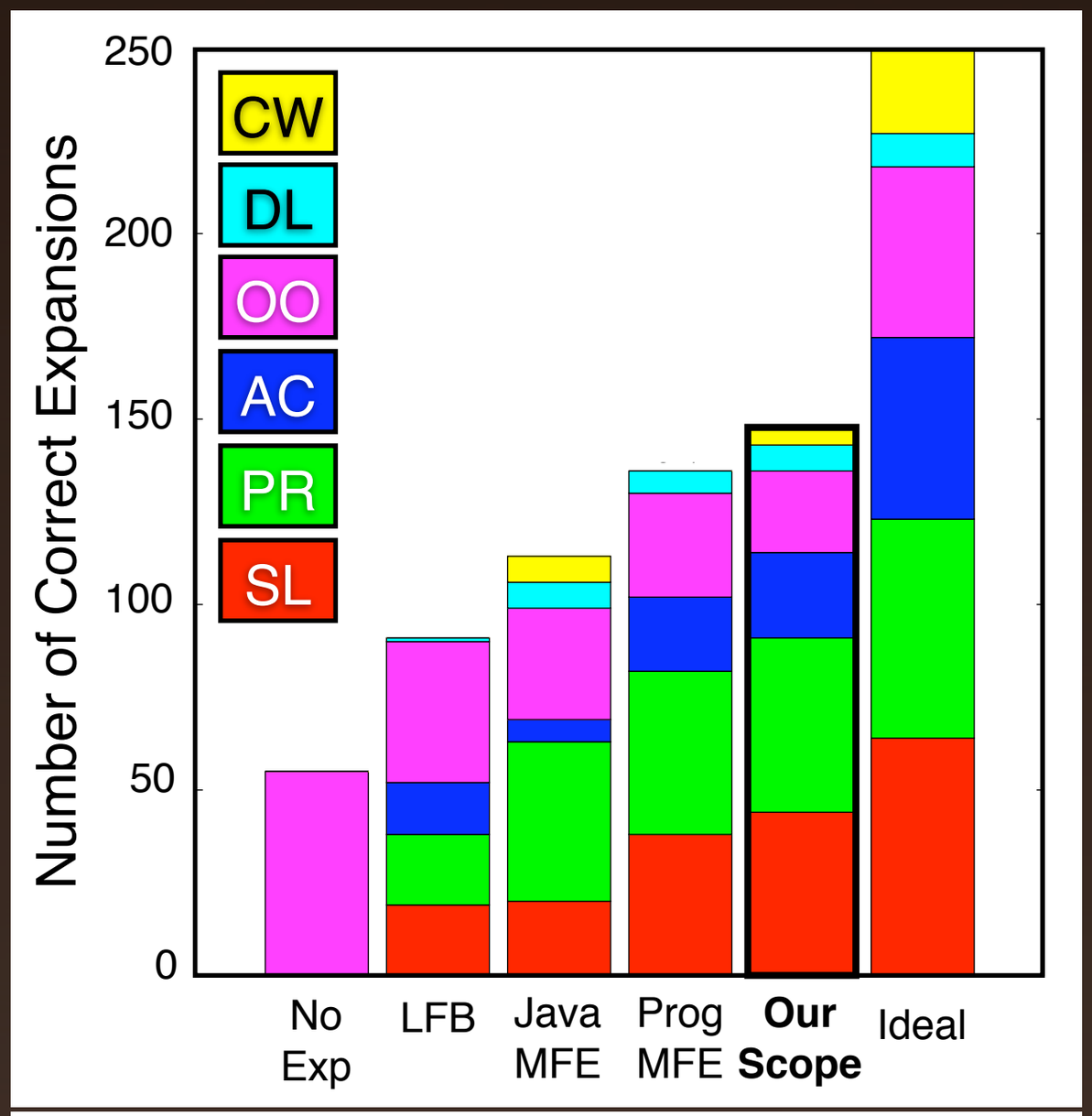
Experimental Evaluation: Results



Accuracy:	22%	***40%***	45%	54%	59%	
					63%	

- 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

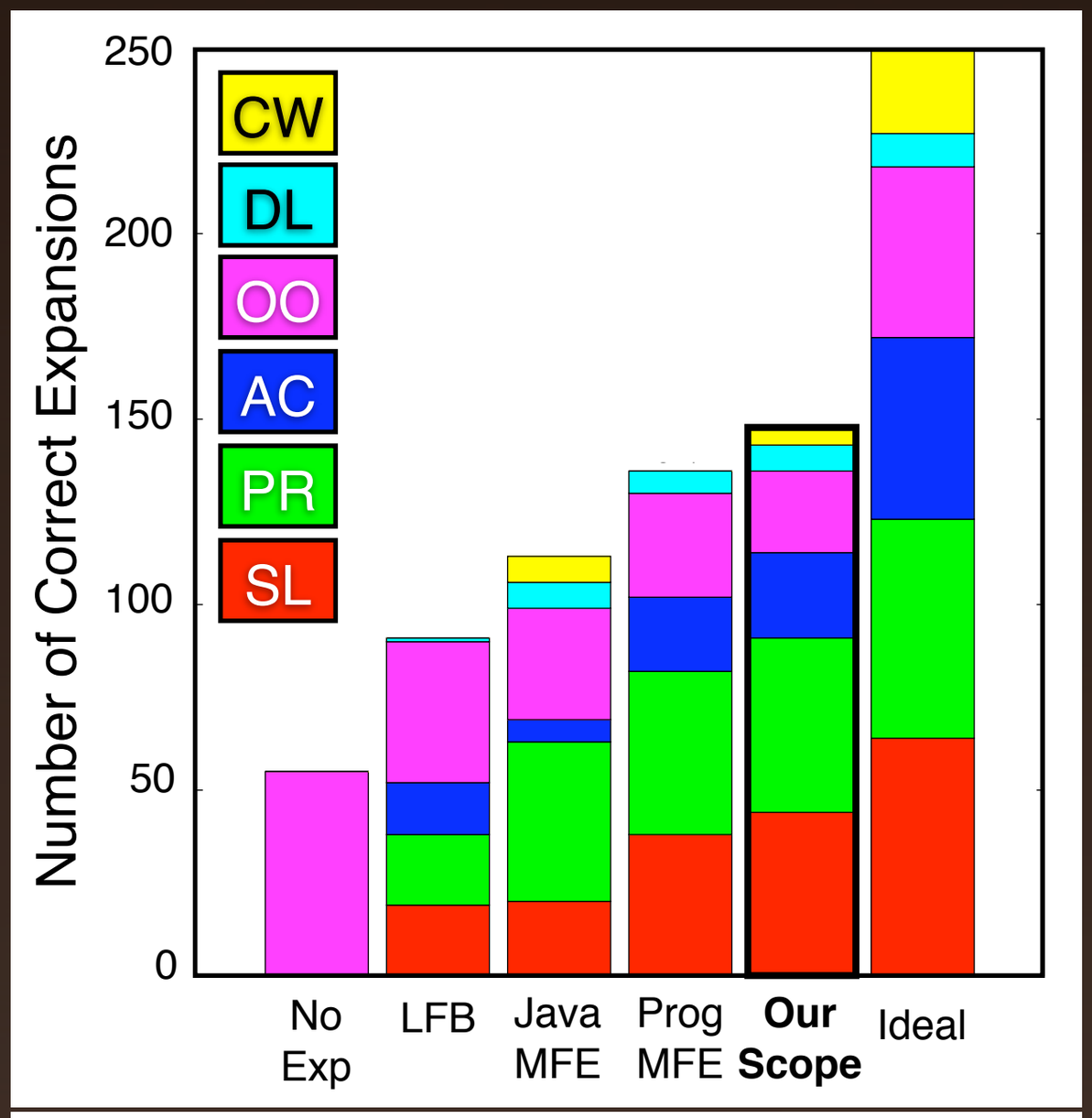


Accuracy:	22%	***40%***	45%	54%	59%	
					63%	

- 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

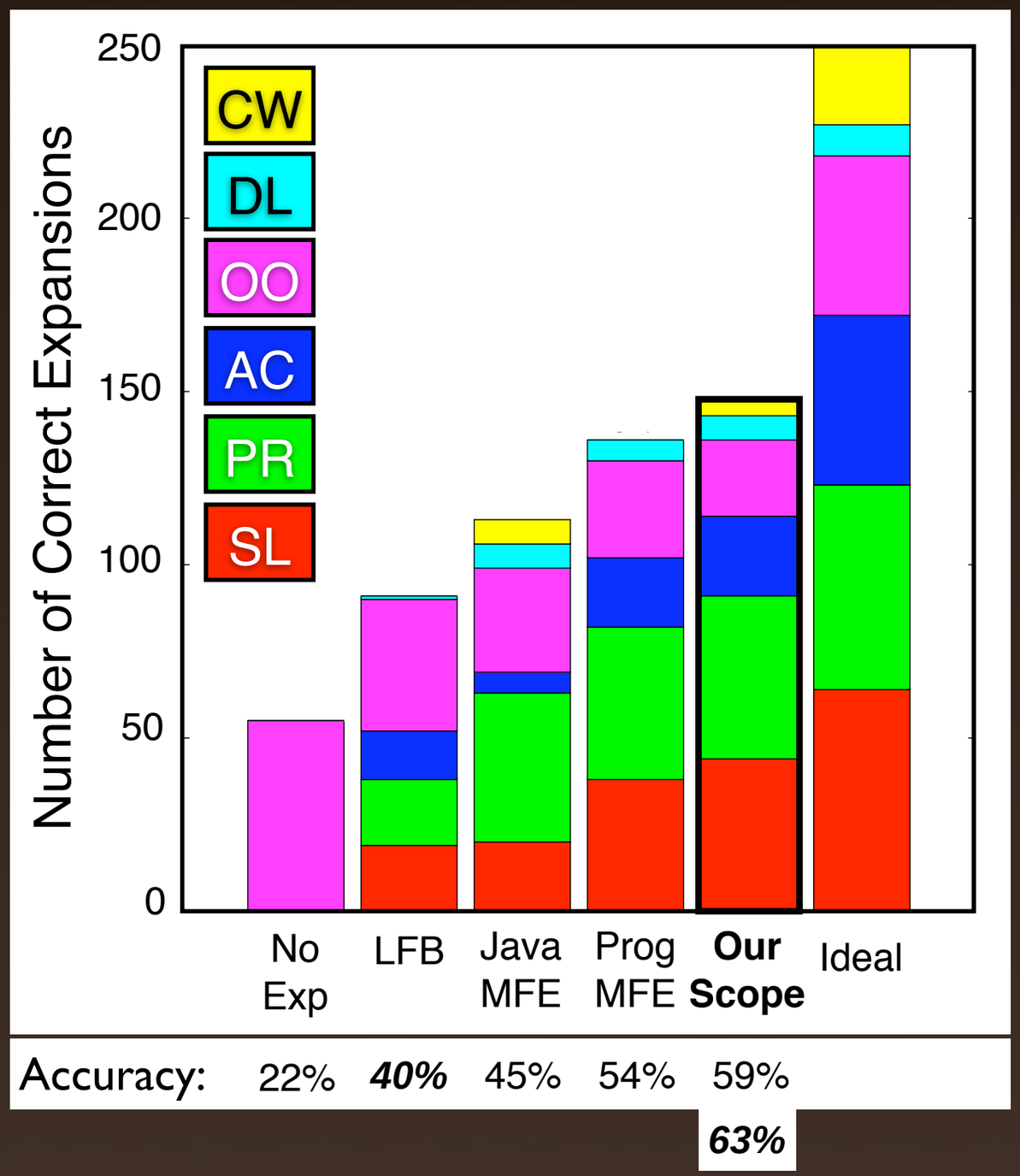
Experimental Evaluation: Results



- 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

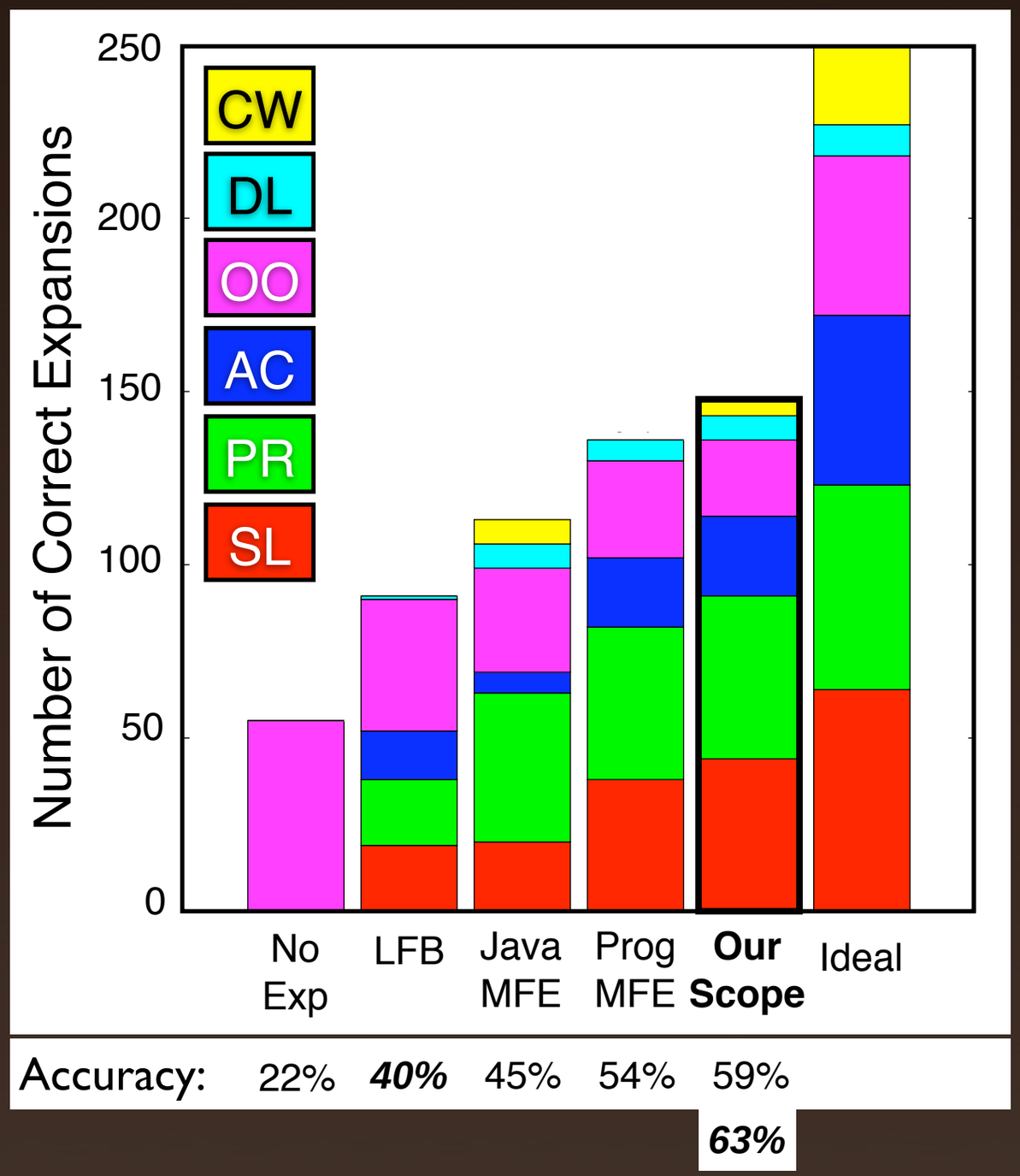
Experimental Evaluation:
Results



- 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.