Detecting non-local violations of API contracts in large software systems
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }

}
One month later...
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }

}
static void copyAll(Reader r, Writer w) {
    int buffer = 0;
    try {
        do {
            buffer = r.read();
            w.write(buffer);
        } while (buffer != -1);
    } finally {
        r.close();
        w.close();
    }
}
One hour later...
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }
}
java.io

Class PrintWriter

java.lang.Object
   java.io.Writer
      java.io.PrintWriter

public class PrintWriter
   extends Writer

Print formatted representations of objects to a text-output stream. This class implements all of the print methods found in PrintStream. It does not contain methods for writing raw bytes, for which a program should use unencoded byte streams.

Unlike the PrintStream class, if automatic flushing is enabled it will be done only when one of the println() methods is invoked, rather than whenever a newline character happens to be output. The println() methods use the platform's own notion of line separator rather than the newline character.

Methods in this class never throw I/O exceptions. The client may inquire as to whether any errors have occurred by invoking checkError().

Since:
   JDK1.1
public class Logging {

    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }
}

Possible API violation on Writer w

w implements type Writer, whose contract is violated.
At this line, w may be in state closed. – Why?
The operation write(String) is not allowed in that state.

I’m feeling lucky  Show API contract  Ignore
public class Logging {

private static Writer w;

public static void init(OutputStream os) {
    w = new PrintWriter(os);
}

public static void log(String s) {
    w.write(s);
}

public static void logAll(Reader r) {
    StreamUtil.copyAll(r, w);
}
}
Detecting non-local violations of API contracts in large software systems
Pure runtime monitoring

Current approach...
public class Logging {
    private static Writer w;

    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }

    public static void log(String s) {
        w.write(s);
    }

    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }
}

“no write after close”

abc compiler
Current approach: Tracematches

```java
tracematch(Writer w) {
    sym close after returning:
    call(* Writer.close()) && target(w);
    sym write before:
    call(* Writer.write(..)) && target(w);

    close write
    {
        System.out.println(
            "Writer "+w+" was closed!");  
    }
}
```
```
{  
  w1.close();
  System.out.println(
    w1.write("foo");
    "Writer \+\+ was 'closed!'");
}
  w1.write("foo");
```
Problem 1: How to capture an API contract precisely and concisely?
Problem 2: Potentially large runtime overhead
Problem 3: No static guarantees
novel static API specification language(s)

public class Logging {
    private static Writer w;
    
    public static void init(OutputStream os) {
        w = new PrintWriter(os);
    }
    
    public static void log(String s) {
        w.write(s);
    }
    
    public static void logAll(Reader r) {
        StreamUtil.copyAll(r, w);
    }
}

4 novel static program analyses
Current approach: Tracematches

tracematch(Writer w) {
    sym close after returning:
        call(* Writer.close()) && target(w);
    sym write before:
        call(* Writer.write(..)) && target(w);

    close write
    {
        System.out.println(
            "Writer "+w+" was closed!");
    }
}
Good:

Can reason about single objects.
Even better:

Can reason about combinations of objects.
Good:

Semantics of regular expression matching well understood.
Bad:

Regular expressions bad at stating **required** events.

“always close after open”

“open ∆ close”
Bad:

Little reuse.

“no write after close”
and
“always close after open”

requires two tracematches!
Goal:
Make API specifications precise and concise
Plan

Investigate a number of actual programs
Plan

Investigate a number of actual programs

What APIs do programs use?
Plan

Investigate a number of actual programs

What mistakes do people make?
Plan

Investigate a number of actual programs

What specifications would have avoided those mistakes?
Goal

find
reoccurring
patterns
fall-back solution

90 / 10

concise specifications
4 novel static program analyses
First static analysis:

“Quick Check”
public class Logging {
    private static Writer w;
    public static void init(InputStream cs) {
        w = new PrintWriter(cs);
    }
    public static void log(String s) {
        w.println(s);
    }
    public static void copy(Reader r, Writer w) {
        StreamUtil.copy(r, w);
    }
}
Second static analysis:

“Consistent-shadows analysis”
[close, write]
new PrintWriter();

\{c_1\} \times \{w_1, w_2\}

\{c_1, w_1\} \checkmark

\{c_1, w_2\} \times

[close, write]
Third static analysis:

“Flow-sensitive unnecessary shadows analysis”
true  \[\text{w} = \text{i(false)}-s_1\]

true  \[\text{w} \neq \text{i(true)}-s_2\]

\[\text{w} = \text{i(w1)}-s_1\]

\[\text{w1.close();} \quad // s_1\]

\[\text{w1.write("foo");} \quad // s_2\]

\[\text{w1.write("foo");} \quad // s_3\]
4th static analysis: “run-once loop optimization”
Results

102 program/tracematch combinations
static guarantees in 77 cases
Found 5 programs with bugs or questionable code.
less than 10% overhead in 9 cases
private final void FillBuff() {
    ...
    try {
        try {
            if ((i = inputStream.read(...)) == -1) {
                inputStream.close();
                throw new java.io.IOException();
            }
        }
        else
            maxNextCharInd += i;
        return;
    }
    ...
}
static String getline(BufferedReader reader, int line) {
  if (reader == null)
    return "";
  try {
    String text = null;
    for (int i = 0; i < line; i++) {
      text = reader.readLine();
    }
    return text;
  } catch (IOException ioe) {
    return null;
  }
}
Jython / hasNext (delegate)

```java
public Iterator iterator() {
    return new Iterator() {
        Iterator i = list.iterator();
        public void remove() {
            throw new UnsupportedOperationException();
        }
        public boolean hasNext() {
            return i.hasNext();
        }
        public Object next() {
            return i.next();
        }
    };
}
```
private List markUsages(IDataFlowNode inode) {
    
    for (Iterator k = ((List)entry.getValue()).iterator(); k.hasNext()) {
        addAccess(k, inode);
    }

    
}

private void addAccess(Iterator k, IDataFlowNode inode) {
    NameOccurrence occurrence = (NameOccurrence) k.next();

}
private List markUsages(IDataFlowNode inode) {
    ...
    for (NameOccurrence occurrence: entry.getValue()) {
        addAccess(occurrence, inode);
    }
    ...
}

private void addAccess(NameOccurrence occurrence, IDataFlowNode inode) {
    ...
}
## Benchmark programs

<table>
<thead>
<tr>
<th>antlr</th>
<th>hsqldb</th>
</tr>
</thead>
<tbody>
<tr>
<td>bloat</td>
<td>jython</td>
</tr>
<tr>
<td>chart</td>
<td>lucene</td>
</tr>
<tr>
<td>eclipse</td>
<td>pmd</td>
</tr>
<tr>
<td>fop</td>
<td>xalan</td>
</tr>
<tr>
<td>pentaho</td>
<td>scimark</td>
</tr>
</tbody>
</table>
## Tracematches

<table>
<thead>
<tr>
<th>ASyncIterator</th>
<th>HasNextElem</th>
</tr>
</thead>
<tbody>
<tr>
<td>FailSafeEnum</td>
<td>LeakingSync</td>
</tr>
<tr>
<td>FailSafeIter</td>
<td>Reader</td>
</tr>
<tr>
<td>HashMap</td>
<td>Writer</td>
</tr>
<tr>
<td>HasNext</td>
<td>...</td>
</tr>
</tbody>
</table>

+ program specific tracematches