Automatic Documentation Inference for Exceptions
MISSION: EXCEPTIONAL
Exceptions: Why?
Handling exceptions
A look at existing practice in 10 popular Java programs
Hypothesis:
  - We can automatically generate documentation describing when exceptions are thrown that is, on average, better than human-written documentation
Evaluation
Usage Considerations and Conclusions
Language construct for transferring control to a place where an event can be handled

2 General Cases

- Legitimate environmental events
  - e.g., the disk is full

- Checking invariants or preconditions
  - e.g., argument must not be null
Context
- Modules lead us to generic (reusable) code
- In general, error handling can’t be generic
CONTEXT

More Context → Less Context → … → Detect Event → Exception!

@slide 7
More Context -> Less Context -> ... -> Exception!

- Handle
- Gather Context
- Detect Event
In real life we can “think up” solutions on-the-fly

In software, we have to anticipate everything

We have to understand the conditions that can cause exceptions
- Mishandling or Not handling can lead to...
  - Security vulnerabilities
    - May disclose sensitive implementation details
  - Breaches of API encapsulation
    - Might want to change exceptions later
  - Any number of minor to serious system failures
- **Solution 1**: No exceptions. Total functions only.
- **Solution 2**: Pretend exceptions don’t happen.
- **Solution 3**: Keep track of all exceptions and handle them appropriately.
/**
 * Moves this unit to america.
 * @exception IllegalStateException If the move is illegal.
 */

public void moveToAmerica() {
    if (!(getLocation() instanceof Europe)) {
        throw new IllegalStateException(
            "A unit can only be " +
            "moved to america from europe.");
    }
    setState(TO_AMERICA);
    // Clear the alreadyOnHighSea flag:
    alreadyOnHighSea = false;
    // Clean up:
}

/**
 * Moves this unit to america.
 *
 * @exception IllegalStateException If the move is illegal.
 */

public void moveToAmerica() {

    if (!(getLocation() instanceof Europe)) {
        throw new IllegalStateException("A unit can only be " + "moved to america from europe.");
    }

    setState(TO_AMERICA);

    // Clear the alreadyOnHighSea flag:
    alreadyOnHighSea = false;
}
/**
 * Moves this unit to america.
 *
 * @exception IllegalStateException If the move is illegal.
 */

public void moveToAmerica() {

    if (!(getLocation() instanceof Europe)) {
        throw new IllegalStateException("A unit can only be " + "moved to america from europe.");
    }

    setState(TO_AMERICA);

    // Clear the alreadyOnHighSea flag:
    alreadyOnHighSea = false;
}
/**
 * Moves this unit to america.
 * @exception IllegalStateException If the move is illegal.
 */

public void moveToAmerica() {

    if (!(getLocation() instanceof Europe)) {
        throw new IllegalStateException("A unit can only be " + "moved to america from europe.");
    }

    setState(TO_AMERICA);

    // Clear the alreadyOnHighSea flag:
    alreadyOnHighSea = false;
}
Need to check all the methods that are *reachable*

With subtyping and dynamic dispatch there could be *many implementations* of a method

And what happens as the system evolves?
For Developers
- Easier to keep track of what’s going on

For Maintenance
- 90% of the total cost of a typical software project
- 40% - 60% of maintenance is spent studying existing software

For Users
- Easier to integrate existing software libraries
<table>
<thead>
<tr>
<th>Program Name</th>
<th>Application Domain</th>
<th>kLOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Azureus</td>
<td>Internet File Sharing</td>
<td>470</td>
</tr>
<tr>
<td>DrJava</td>
<td>Development</td>
<td>131</td>
</tr>
<tr>
<td>FindBugs</td>
<td>Program Analysis</td>
<td>142</td>
</tr>
<tr>
<td>FreeCol</td>
<td>Game</td>
<td>103</td>
</tr>
<tr>
<td>hsqldb</td>
<td>Database</td>
<td>154</td>
</tr>
<tr>
<td>jEdit</td>
<td>Text Editor</td>
<td>138</td>
</tr>
<tr>
<td>jFreeChart</td>
<td>Data Presentation</td>
<td>181</td>
</tr>
<tr>
<td>Risk</td>
<td>Game</td>
<td>34</td>
</tr>
<tr>
<td>tvBrowser</td>
<td>TV guide</td>
<td>155</td>
</tr>
<tr>
<td>Weka</td>
<td>Machine Learning</td>
<td>436</td>
</tr>
<tr>
<td>Total</td>
<td></td>
<td>1944</td>
</tr>
</tbody>
</table>
Exception Instance
- An Exception type and a method that can propagate it
- Each exception instance is an opportunity for a documentation

Depth of an Exception Instance
- Minimum number of dynamic method invocations between the Exception Instance and a throw statement of its type
- Intuitively, greater depth implies harder to figure out
DOCUMENTATION: WHEN DOES IT HAPPEN?

Average Percentage of Exception Instances Documented

Minimum Depth Threshold

100%
90%
80%
70%
60%
50%
40%
30%
20%
10%
0%

0 1 2 3 4
DOCUMENTATION: CONSISTENCY

<table>
<thead>
<tr>
<th>Program and Exception Type</th>
<th>Percentage of Exceptions Documented in Javadoc</th>
</tr>
</thead>
<tbody>
<tr>
<td>weka</td>
<td>100%</td>
</tr>
<tr>
<td>azureus</td>
<td>90%</td>
</tr>
<tr>
<td>risk</td>
<td>80%</td>
</tr>
<tr>
<td>findbugs</td>
<td>70%</td>
</tr>
<tr>
<td>drjava</td>
<td>60%</td>
</tr>
<tr>
<td>IllegalStateException</td>
<td>50%</td>
</tr>
<tr>
<td>DataLength</td>
<td>40%</td>
</tr>
<tr>
<td>InvalidCipherText</td>
<td>30%</td>
</tr>
<tr>
<td>Exception</td>
<td>20%</td>
</tr>
<tr>
<td>NullPointerException</td>
<td>10%</td>
</tr>
<tr>
<td>ClassNotFoundException</td>
<td>0%</td>
</tr>
<tr>
<td>DataflowAnalysis</td>
<td></td>
</tr>
<tr>
<td>OperationCanceled</td>
<td></td>
</tr>
<tr>
<td>IllegalArgumentException</td>
<td></td>
</tr>
</tbody>
</table>
Can we do better?

```java
/**
 * Moves this unit to america.
 * @exception IllegalStateException If the move is illegal.
 */

public void moveToAmerica() {

    if (!(getLocation() instanceof Europe)) {
        throw new IllegalStateException("A unit can only be "
             + "moved to america from europe.");
    }

    setState(TO_AMERICA);

    // Clear the alreadyOnHighSea flag:
    alreadyOnHighSea = false;
}
```
/**
 * Moves this unit to america.
 *
 * @exception IllegalArgumentException thrown when
 *         getLocation() is not a Europe.
 */

public void moveToAmerica() {

    if (!(getLocation() instanceof Europe)) {
        throw new IllegalArgumentException("A unit can only be "
                + "moved to america from europe.");
    }

    setState(TO_AMERICA);

    // Clear the alreadyOnHighSea flag:
    alreadyOnHighSea = false;
}
We can create an automatic tool that documents exceptions better than developers have.

- Better?
  - More complete
  - More precise
- A simple example:

```java
main()
{
    if ( x < 0 )
        throw new Exception();
    else
        sub( x );
}

sub( int n )
{
    if ( n == 4 )
        throw new Exception();
}
```
Find the throw statements

```java
main()
{
    if ( x < 0 )
        throw new Exception();
    else
        sub( x );
}

sub( int n )
{
    if ( n == 4 )
        throw new Exception();
}
```
- Link method invocations to possible targets
  - We use an off-the-shelf call graph generator

```java
main()
{
    if ( x < 0 )
        throw new Exception();
    else
        sub( x );
}

sub( int n )
{
    if ( n == 4 )
        throw new Exception();
}
```
Determine which methods can throw which exceptions

- Use a fixpoint worklist to deal with cycles
- Must consider catch and finally blocks
Enumerate control flow paths that can lead to exceptions
- Work backward from exception throwing statements
Symbolically execute paths, record predicates
- Use another fixpoint worklist

```java
main() {Exception}
{
    if (x < 0 )
        throw new Exception();
    else
        sub(x);
}

sub(int n) {Exception}
{
    if (n == 4 )
        throw new Exception();
}
```
Predicates along the path become the documentation

```java
@throws Exception if
    x < 0 OR (x >= 0 AND x==4)
main()
{
    if ( x < 0 )
        throw new Exception();

    else
        sub( x );
}

@throws Exception if
    parameter:n == 4
sub( int n )
{
    if ( n == 4 )
        throw new Exception();
}
Finally, some simplification & readability enhancements

- **TRUE** becomes “always”
- **FALSE OR** x becomes “x”
- x != null becomes “x is not null”
- x instanceof T becomes “x is a T”
- x.hasNext() becomes “x is nonempty”
- x.iterator().next() becomes “x.{some element}”
THE ALGORITHM: SUMMARY

- Generate call graph
- Track all explicitly thrown exceptions by concrete type
- Construct and symbolically execute all (exponentially many) paths that can lead to a `throw`
- Construct predicates and make them more readable
Baseline: Existing *JavaDocs*
- 10 Benchmarks from earlier
- ~950 documentations

Run tool on each program and create pairs
- `<tool doc, existing doc>`

Bin each in: *Worse, Same, Better*
- Sometimes we do **better**:

  Worse: if inappropriate
  (Us) Better: parameter: params not a KeyParameter
  Worse: id == null
  (Us) Better: id is null or id.equals("")

- Sometimes we do about the **same**:

  Same: has an insufficient amount of gold.
  (Us) Same: getPriceForBuilding() > getOwner().getGold()

- Sometimes we do **worse**:

  Better: the queue is empty
  (Us) Worse: private variable m_Head is null
- `throw` statements are relatively rare
- Only have to execute paths that lead to a `throw`
- We don’t follow back edges
  - Some limit needed to guarantee termination
- Whole process takes about 10 min on average
Exceptions that seem possible aren’t really
  - Better call graph

Exceptions contexts are deep and complex
  - Could be a symptom of bad design
  - Might want to ignore certain types or threshold depth

Same exception type stands for many error conditions
  - Increase granularity of exception type hierarchy
- External API
  - System users
- Code Reviews
  - Reading & Inspection
- Verification
  - If we want to be more formal
- Exceptions probably aren’t going away
- Many exception instances remain poorly or not documented in practice
- On average, we do at least as well as humans 83% of the time and are fully automatic
- We can scale to large programs
  - Azureus has 470 kLOC, tool runs in ~25 min
throw new OutOfSlidesException();