

Ray PL Buse

# Automatic Documentation Inference for Exceptions



@author Ray PL Buse

# MISSION: EXCEPTIONAL

[ARRESTED  
COMPUTING]

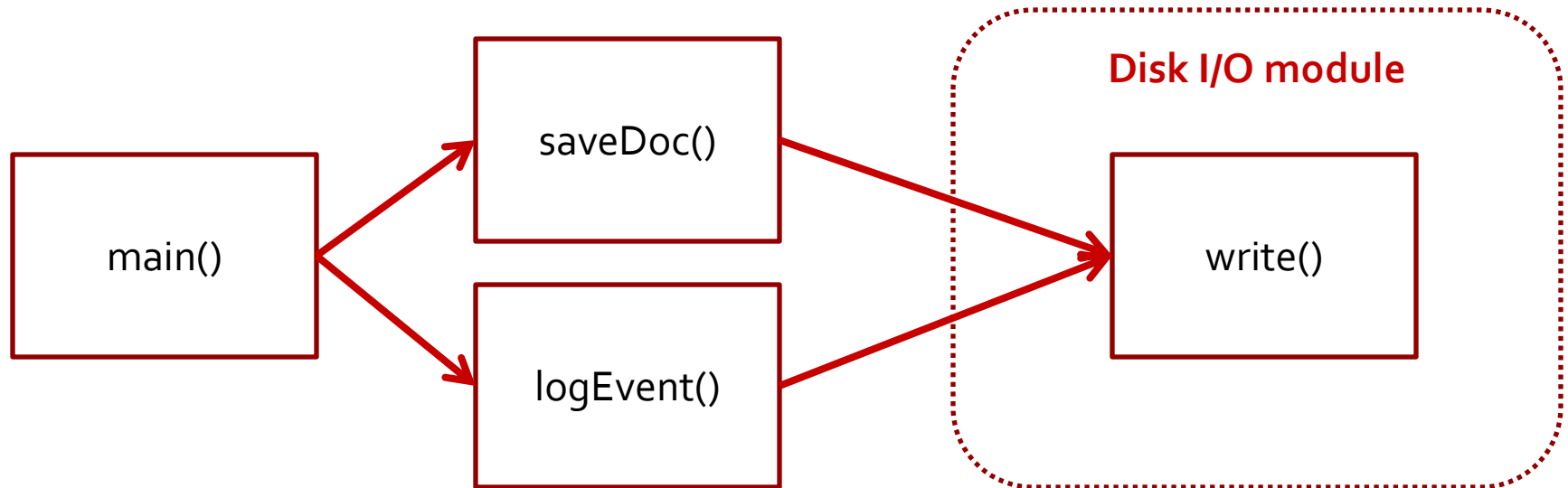
- Exceptions: Why?
- Handling exceptions
- A look at existing practice in 10 popular Java programs
- Hypothesis:
  - We can automatically generate documentation describing when exceptions are thrown that is, on average, better than human-written documentation
- Evaluation
- Usage Considerations and Conclusions

- 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

# EXCEPTIONS: WHERE DO THEY COME FROM?

@slide 5

- Context
  - Modules lead us to generic (reusable) code
  - In general, error handling can't be generic





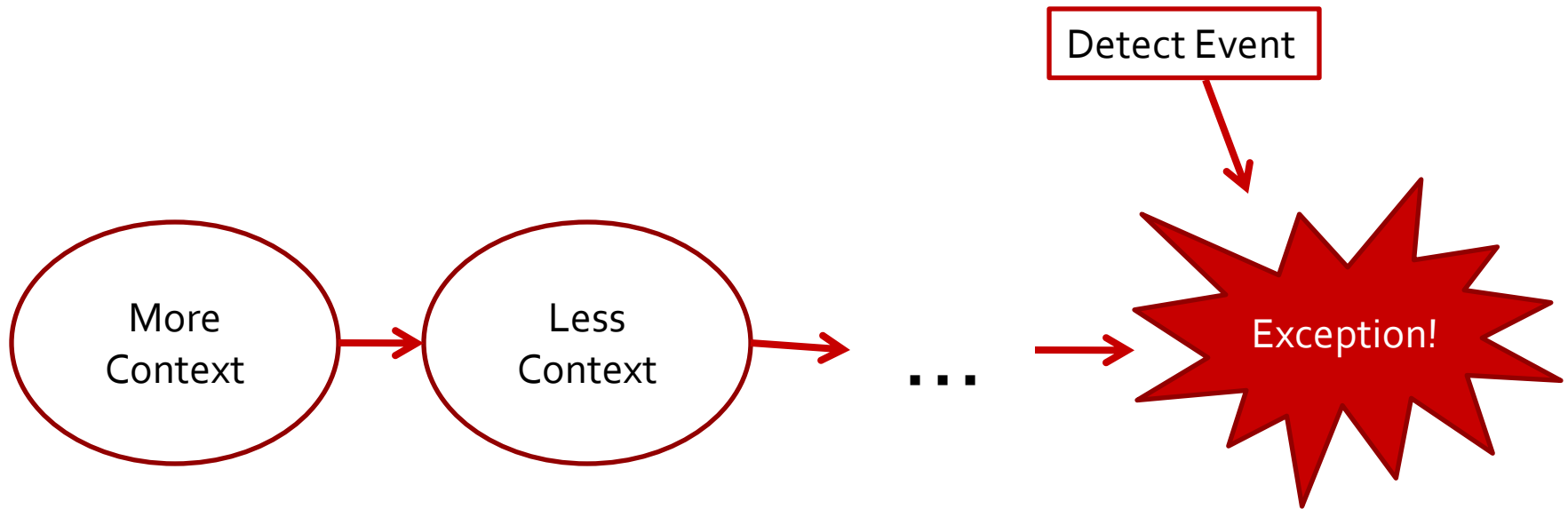
**TELEFON**

TELEFON

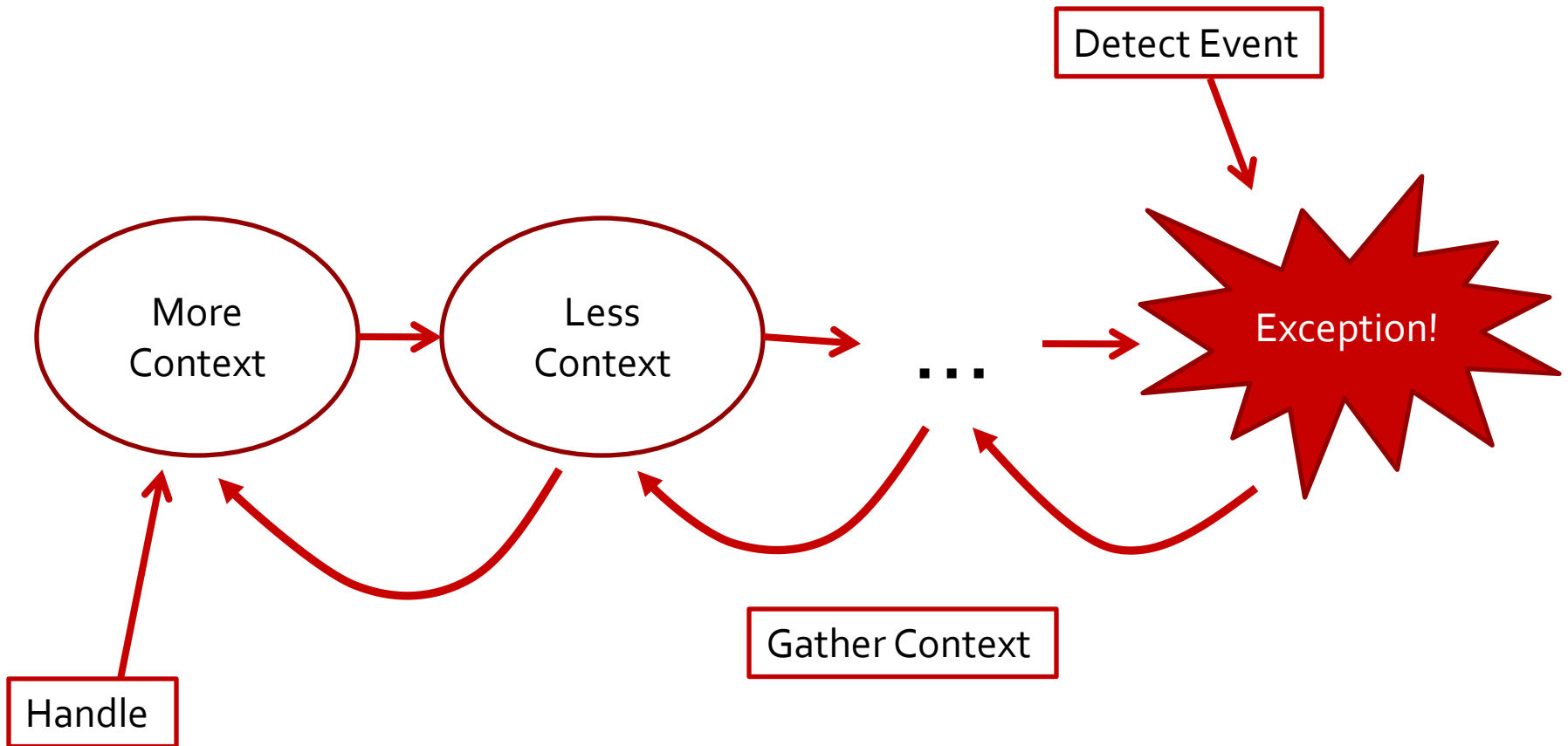


# CONTEXT CONTEXT

@slide 7



# CONTEXT





# THE REALITY DILEMMA

@slide 9

- 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.

# AN EXAMPLE

@slide 12

```
/**
 * Moves this unit to america.
 *
 * @exception IllegalArgumentException if the destination is illegal.
 */
public void moveToAmerica() {
    if (!(getDestination().equals("america")))
        throw new IllegalArgumentException("Destination can only be \"america\"");
    setState("moving");
    // Clear the destination field if the unit is already on the map.
}
```

A screenshot of a game map showing a unit moving to America. The map is a 3D isometric view of a world with various terrain types like mountains, forests, and water. A unit is shown moving across the map. The interface includes a mini-map in the bottom left and a unit information panel in the bottom right.

# AN EXAMPLE


@slide 13

```
/**
 * 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;
}
```



# AN EXAMPLE

@slide 14


```
/**
 * 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;
}
```

When does this throw  
an exception?



Here's one spot



# AN EXAMPLE

@slide 15

```
/**
 * 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;
}
```

When does this throw  
an exception?

Must check here

and here

# THE PROBLEM: It's hard

@slide 16

- 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?





# DOCUMENTATION: WHY?

@slide 17

- 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

# BENCHMARKS

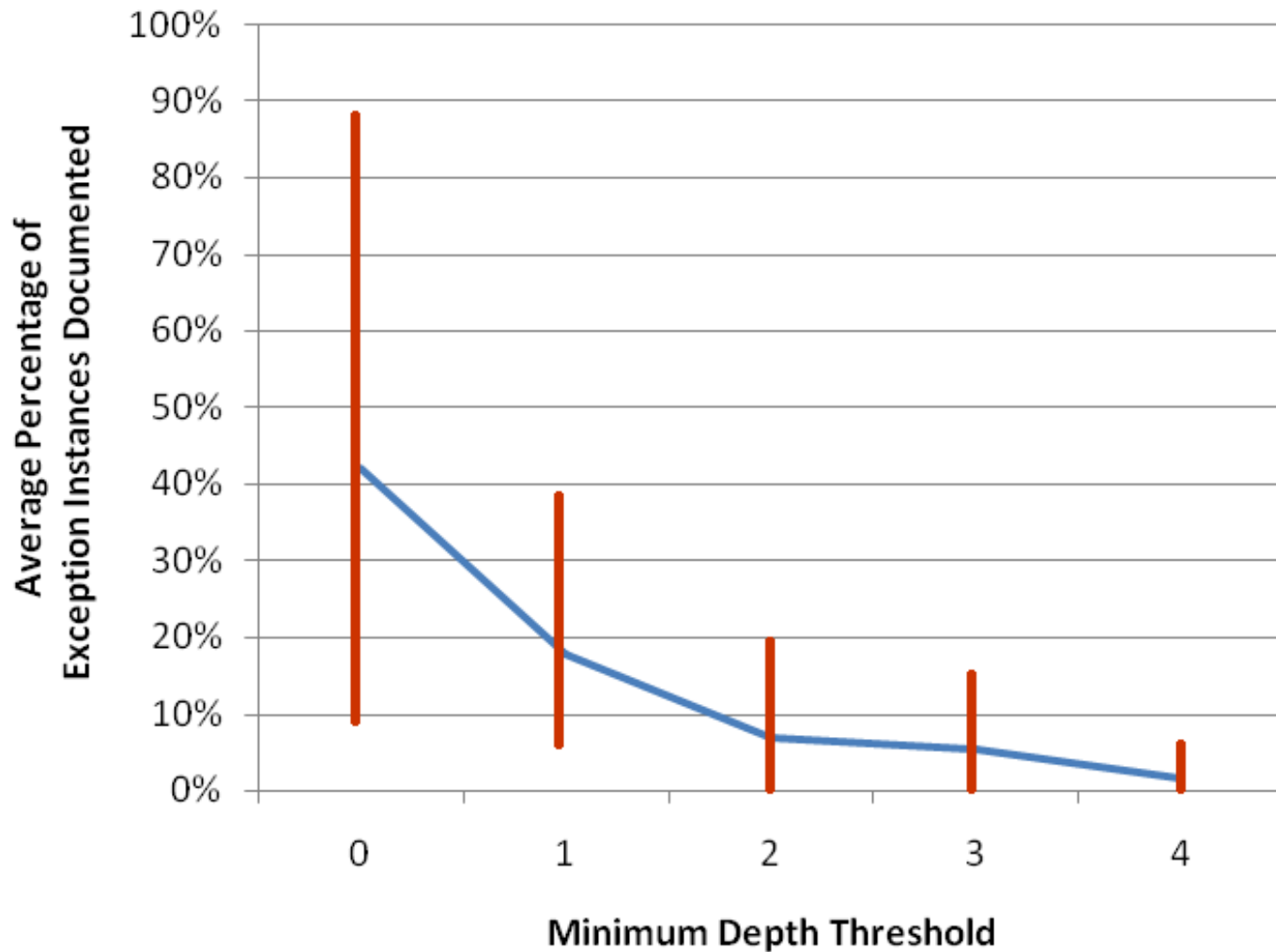
@slide 18

Program Name	Application Domain	kLOC
Azureus	Internet File Sharing	470
DrJava	Development	131
FindBugs	Program Analysis	142
FreeCol	Game	103
hsqldb	Database	154
jEdit	Text Editor	138
jFreeChart	Data Presentation	181
Risk	Game	34
tvBrowser	TV guide	155
Weka	Machine Learning	436
Total		1944

- 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

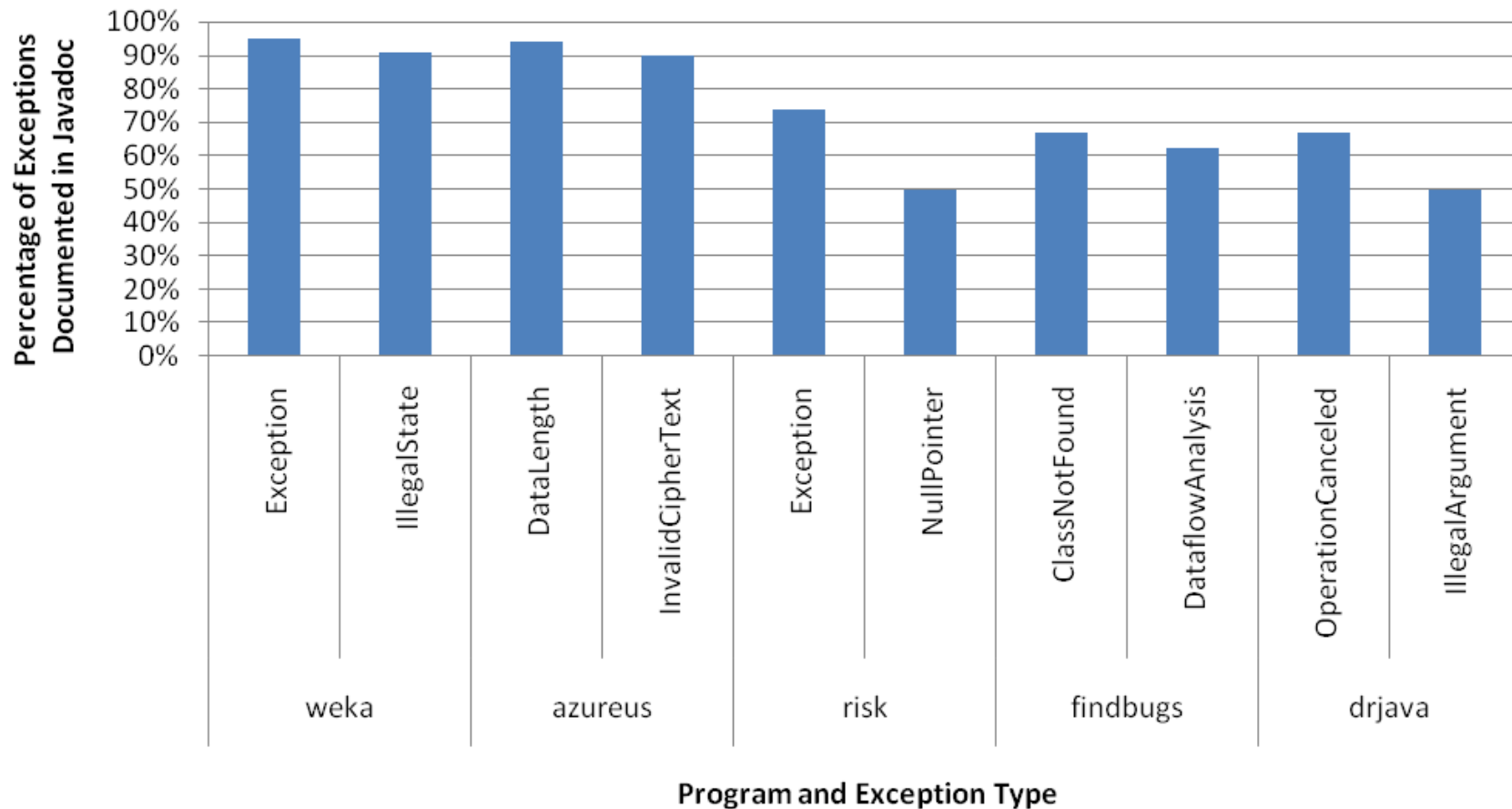
# DOCUMENTATION: WHEN DOES IT HAPPEN?

@slide 20



# DOCUMENTATION: CONSISTENCY

@slide 21



# AN EXAMPLE

@slide 22

```
/**
 * 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;
}
```

Can we do better?



# AN EXAMPLE


@slide 23

```
/**
 * Moves this unit to america.
 *
 * @exception IllegalStateException thrown when
 *         getLocation() is not a Europe.
 */
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;
}
```



- We can create an automatic tool that documents exceptions **better** than developers have
  - Better?
    - More complete
    - More precise



- A simple example:

```
main()
{
    if( x < 0 )
        throw new Exception();

    else
        sub( x );
}

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

- Find the throw statements

```
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

```
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

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

    else
        sub( x );
}

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



- Enumerate control flow paths that can lead to exceptions
  - Work backward from exception throwing statements

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

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

- Symbolically execute paths, record predicates
  - Use another fixpoint worklist

```
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

```
@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();
}
```

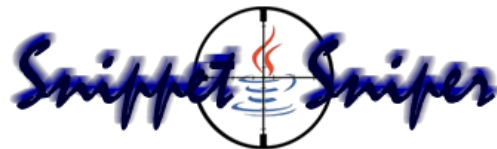
- Finally, some simplification & readability enhancements
  - `TRUE` becomes "always"
  - `FALSE OR x` becomes "x"
  - `x != null` becomes "x is not null"
  - `x instanceof T` becomes "x is a T"
  - `x.hasNext()` becomes "x is nonempty"
  - `x.iterator().next()` becomes "x.{some element}"



# THE ALGORITHM: SUMMARY

@slide 33

- 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**



# EXPERIMENTAL SETUP

@slide 34

- 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**:

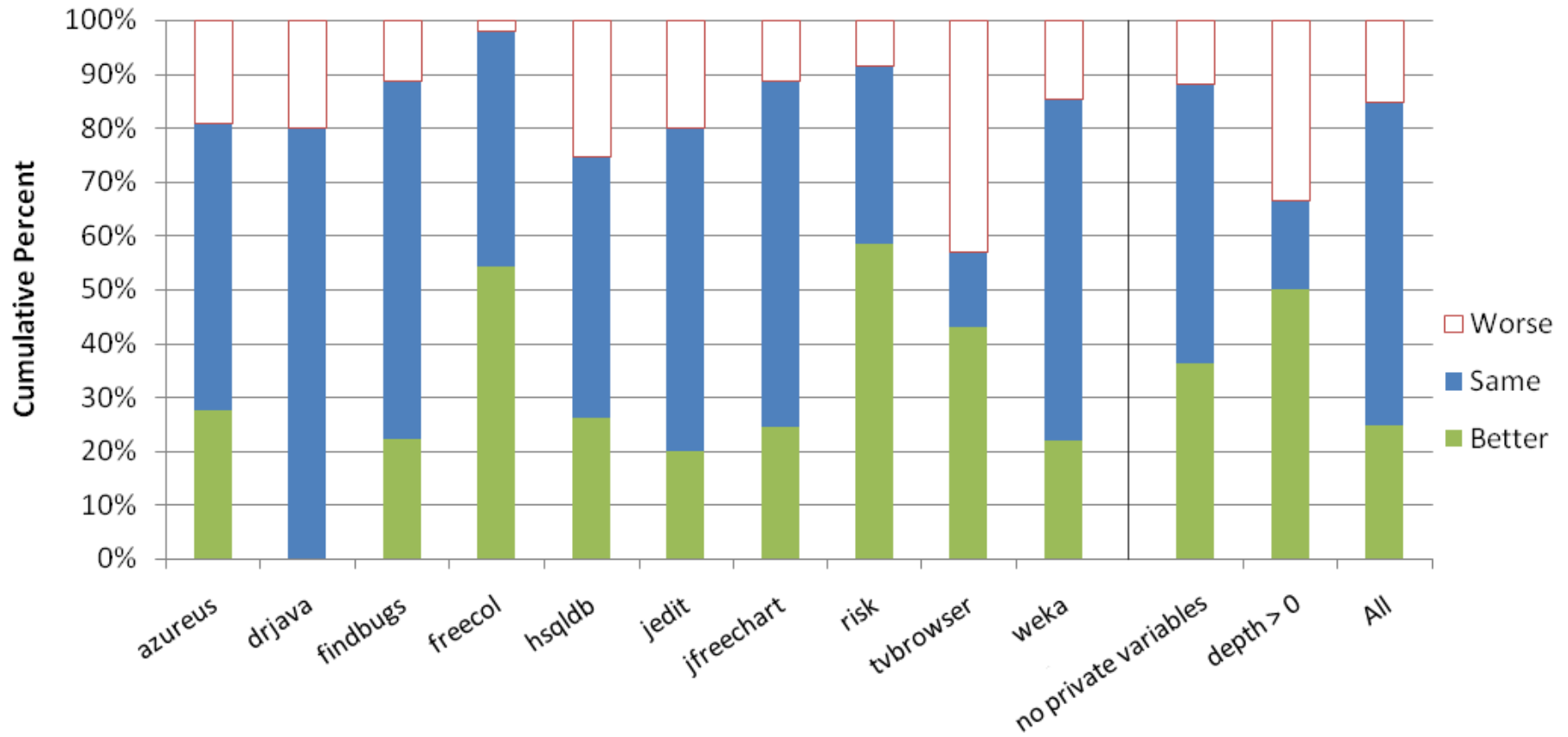
```
Same: has an insufficient amount of gold.
(Us) Same: getPriceForBuilding() > getOwner().getGold()
```

- Sometimes we do **worse**:

```
Better: the queue is empty
(Us) Worse: private variable m_Head is null
```

# RESULTS

@slide 36



- `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

# LIMITATIONS

@slide 38

- 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

# CONCLUSION

@slide 40

- 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();
```

