Software engineering and programmer productivity tools

Programming languages
Automated testing
Debuggers
IDEs
Type systems
Verification
Security
Preventing Errors Before They Happen

http://CheckerFramework.org/
Twitter: @CheckerFrmwrk
Live demo: http://eisop.uwaterloo.ca/live

Werner Dietl, University of Waterloo
Michael Ernst, University of Washington
Motivation

java.lang.NullPointerException
Cost of software failures

$312 billion per year global cost of software bugs (2013)

$300 billion dealing with the Y2K problem

$440 million loss by Knight Capital Group Inc. in 30 minutes in August 2012

$650 million loss by NASA Mars missions in 1999; unit conversion bug

$500 million Ariane 5 maiden flight in 1996; 64 bit to 16 bit conversion bug
Software bugs can cost lives

1997: 225 deaths: jet crash caused by radar software
1991: 28 deaths: Patriot missile guidance system
2003: 11 deaths: blackout
1985-2000: >8 deaths: Radiation therapy

2011: Software caused 25% of all medical device recalls
Outline

- Solution: Pluggable type-checking
- Tool: Checker Framework
- How to use it
Java's type system is too weak

Type checking prevents many errors

```java
int i = "hello";
```

Type checking doesn't prevent enough errors

```java
System.console().readLine();
Collections.emptyList().add("one");
```
Java's type system is too weak

Type checking prevents many errors

```java
int i = "hello";
```

Type checking doesn't prevent enough errors

```java
NullPointerException
System.console().readLine();
Collections.emptyList().add("one");
```
Java's type system is too weak

Type checking prevents many errors

```java
int i = "hello";
```

Type checking doesn't prevent enough errors

```java
System.console().readLine();
```

```java
Collections.emptyList().add("one");
```

UnsupportedOperationException
Some errors are silent

```java
Date date = new Date();
myMap.put(date, "now");
date.setSeconds(0); // round to minute
myMap.get(date);
```
Some errors are silent

Date date = new Date();
myMap.put(date, "now");
date.setSeconds(0);  // round to minute
myMap.get(date);

Corrupted map
Some errors are silent

dbStatement.executeQuery(userInput);
Some errors are silent

dbStatement.executeQuery(userInput);

SQL injection attack

Initialization, data formatting, equality tests, …
Solution: Pluggable Type Checking

1. Design a type system to solve a specific problem
2. Write type qualifiers in code (or, use type inference)
   ```java
   @Immutable Date date = new Date();
   date.setSeconds(0);  // compile-time error
   ```
3. Type checker warns about violations (bugs)
   ```bash
   % javac -processor NullnessChecker MyFile.java
   MyFile.java:149: dereference of possibly-null reference bb2
   allVars = bb2.vars;
   ^
   ```
Nullness and encryption demo

- Detect errors
- Guarantee the absence of errors
- Verify the correctness of optimizations

https://xkcd.com/327/
Type Checking

Source \rightarrow Compiler \rightarrow No errors \rightarrow Executable

- Fix bugs
- Change types

Errors
Optional Type Checking

- Source → Compiler → Executable
- No errors → Executable
- Errors → Optional Type Checker → Warnings
- Optional Type Checker
- Guaranteed behavior
- Fix bugs
- Change types
- Fix bugs
- Add/change annotations
Optional Type Checking

Source → Compiler → Executable

Errors
- Fix bugs
- Change types

Warnings
- Fix bugs
- Add/change annotations

Optional Type Checker
- Guaranteed behavior

No errors
Prevent null pointer exceptions

Type system that statically guarantees that:
the program only dereferences known non-null references

Types of data:
@NonNull reference is never null
@Nullable reference may be null
Null pointer exception

```java
String op(Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);
```
String op(Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);
Null pointer exception

Where is the defect?

String op(Data in) {
    return "transform: " + in.getF();
}

...

String s = op(null);
Null pointer exception

Where is the defect?

String op(Data in) {
    return "transform: " + in.getF();
}

String s = op(null);
Specification 1: non-null parameter

```java
String op(@NonNull Data in) {
    return "transform: " + in.getF();
}
...
String s = op(null);
```
Specification 1: non-null parameter

String op(@NonNull Data in) {
    return "transform: " + in.getF();
}

...  
String s = op(null); // error
Specification 2: nullable parameter

```java
String op(@Nullable Data in) {
    return "transform: " + in.getF();
}

... String s = op(null);
```
Specification 2: nullable parameter

String op(@Nullable Data in) {
    return "transform: " + in.getF();
}  // error

... String s = op(null);
Benefits of type systems

- **Find bugs** in programs
  - Guarantee the **absence of errors**
- **Improve documentation**
  - Improve code structure & maintainability
- Aid compilers, optimizers, and analysis tools
  - E.g., could reduce number of run-time checks

- Possible negatives:
  - Must write the types (or use type inference)
  - False positives are possible (can be suppressed)
The Checker Framework

A framework for pluggable type checkers
“Plugs” into the OpenJDK or OracleJDK compiler

```
javac -processor MyChecker ...
```

Standard error format allows tool integration
Eclipse plug-in

```java
public class Test {
    public static void main(String[] args) {
        Console c = System.console();
        c.printf("Test");
    }
}
```
Ant and Maven integration

```xml
<presetdef name="jsr308.javac">
  <javac fork="yes"
        executable="$\{checkerframework\}/checker/bin/$\{cfJavac\}" >
    <!-- JSR-308-related compiler arguments -->
    <compilerarg value="-version"/>
    <compilerarg value="-implicit:class"/>
  </javac>
</presetdef>

<dependencies>
  ... existing <dependency> items ... 
  <!-- annotations from the Checker Framework:
       nullness, interning, locking, ... -->
  <dependency>
    <groupId>org.checkerframework</groupId>
    <artifactId>checker-qual</artifactId>
    <version>1.9.7</version>
  </dependency>
</dependencies>
```
Live demo: http://eisop.uwaterloo.ca/live/

Checker Framework Live Demo

Write Java code here:

```java
import org.checkerframework.checker.nullness.qual.Nullable;

class YourClassNameHere {
    
    void foo(Object nn, @Nullable Object nbl) {
        nn.toString(); // OK
        nbl.toString(); // Error
    }
}
```

Choose a type system: Nullness Checker

Examples:

Nullness: [NullnessExample](#) | [NullnessExampleWithWarnings](#)

MapKey: [MapKeyExampleWithWarnings](#)

 Interning: [InterningExample](#) | [InterningExampleWithWarnings](#)

Lock: [GuardedByExampleWithWarnings](#) | [HoldingExampleWithWarnings](#) | [EnsuresLockHeldExample](#) | [Loc...](#)
Example type systems

Null dereferences (@NonNull)
>200 errors in Google Collections, javac, ...

Equality tests (@Interned)
>200 problems in Xerces, Lucene, ...

Concurrency / locking (@GuardedBy)
>500 errors in BitcoinJ, Derby, Guava, Tomcat, ...

Fake enumerations / typedefs (@Fenum)
problems in Swing, JabRef
String type systems

Regular expression syntax (@Regex)
   56 errors in Apache, etc.; 200 annos required
printf format strings (@Format)
   104 errors, only 107 annotations required
Signature format (@FullyQualified)
   28 errors in OpenJDK, ASM, AFU
Compiler messages (@CompilerMessageKey)
   8 wrong keys in Checker Framework
Security type systems

Command injection vulnerabilities (@OsTrusted)
  5 missing validations in Hadoop
Information flow privacy (@Source)
  SPARTA detected malware in Android apps

You can write your own checker!
Checkers are usable

- Type-checking is familiar to programmers
- Modular: fast, incremental, partial programs
- Annotations are not too verbose
  - @NonNull: 1 per 75 lines
  - @Interned: 124 annotations in 220 KLOC revealed 11 bugs
  - @Format: 107 annotations in 2.8 MLOC revealed 104 bugs
  - Possible to annotate part of program
  - Fewer annotations in new code
- Few false positives
- First-year CS majors preferred using checkers to not
- Practical: in daily use at Google, on Wall Street, etc.
# Comparison: other nullness tools

<table>
<thead>
<tr>
<th>Tool</th>
<th>Null pointer errors</th>
<th>False warnings</th>
<th>Annotations written</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Found</td>
<td>Missed</td>
<td></td>
</tr>
<tr>
<td>Checker Framework</td>
<td>8</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>FindBugs</td>
<td>0</td>
<td>8</td>
<td>1</td>
</tr>
<tr>
<td>Jlint</td>
<td>0</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>PMD</td>
<td>0</td>
<td>8</td>
<td>0</td>
</tr>
</tbody>
</table>

Checking the Lookup program for file system searching (4kLOC)
False warnings are suppressed via an annotation or assertion
What a checker guarantees

The program satisfies the type property. There are:

- no bugs (of particular varieties)
- no wrong annotations

- Caveat 1: only for code that is checked
  - Native methods (handles reflection!)
  - Code compiled without the pluggable type checker
  - Suppressed warnings
    - Indicates what code a human should analyze

Checking part of a program is still useful

- Caveat 2: The checker itself might contain an error
Tips

- Start by type-checking part of your code
- Only type-check properties that matter to you
- Use subclasses (not type qualifiers) if possible
- Write the spec first (and think of it as a spec)
- Avoid warning suppressions when possible
- Avoid raw types such as List; use List<String>
Verification

- **Goal:** prove that no bug exists
- **Specifications:** user provides
- **False negatives:** none
- **False positives:** user suppresses warnings
- **Downside:** user burden

Bug-finding

- **Goal:** find some bugs at low cost
- **Specifications:** infer likely specs
- **False negatives:** acceptable
- **False positives:** heuristics focus on most important bugs
- **Downside:** missed bugs

Neither is “better”; each is appropriate in certain circumstances.
Community

Open source project:
https://github.com/typetools/checker-framework
  ● Monthly release cycle
  ● 11,000 commits, 75 authors

Issue tracker:
  ● 110 issues closed in releases June 1 - Sep 16

Mailing lists:
  ● to reach developers
  ● to reach whole community
Pluggable type-checking improves code

Checker Framework for creating type checkers
  ● Featureful, effective, easy to use, scalable
Prevent bugs at compile time
Create custom type-checkers
Improve your code!

http://CheckerFramework.org/
Why get involved in research?

Unique experience available only at college
Deeper understanding of computing
Close collaboration with faculty, grad students
Contribute to science
Fame (4 single-authored papers on undergrad research, at FSE 2016)
Fun!
How to get involved in PLSE research

Take CSE 331 (“software design & implementation”)
Contact faculty in the PLSE group:

- Rastislav Bodik
- Alan Borning
- Luis Ceze
- Alvin Cheung
- Michael Ernst
- Daniel Grossman
- Andrew Ko
- Zachary Tatlock
- Emina Torlak
- Xi Wang
How to get involved in PLSE research

Take CSE 331 ("software design & implementation")
Contact faculty in the PLSE group:
Input Format Validation

Demo: ensure that certain strings contain valid regular expressions.
Regular Expression Example

```java
public static void main(String[] args) {
    String regex = args[0];
    String content = args[1];
    Pattern pat = Pattern.compile(regex);
    Matcher mat = pat.matcher(content);
    if (mat.matches()) {
        System.out.println("Group: " + mat.group(1));
    }
}
```
public static void main(String[] args) {
    String regex = args[0];
    String content = args[1];
    Pattern pat = Pattern.compile(regex);
    Matcher mat = pat.matcher(content);
    if (mat.matches()) {
        System.out.println("Group: " + mat.group(1));
    }
}
Fixing the Errors

Pattern.compile only on valid regex
Matcher.group(i) only if > i groups

... if (!RegexUtil.isRegex(regex, 1)) {
    System.out.println("Invalid: " + regex);
    System.exit(1);
}