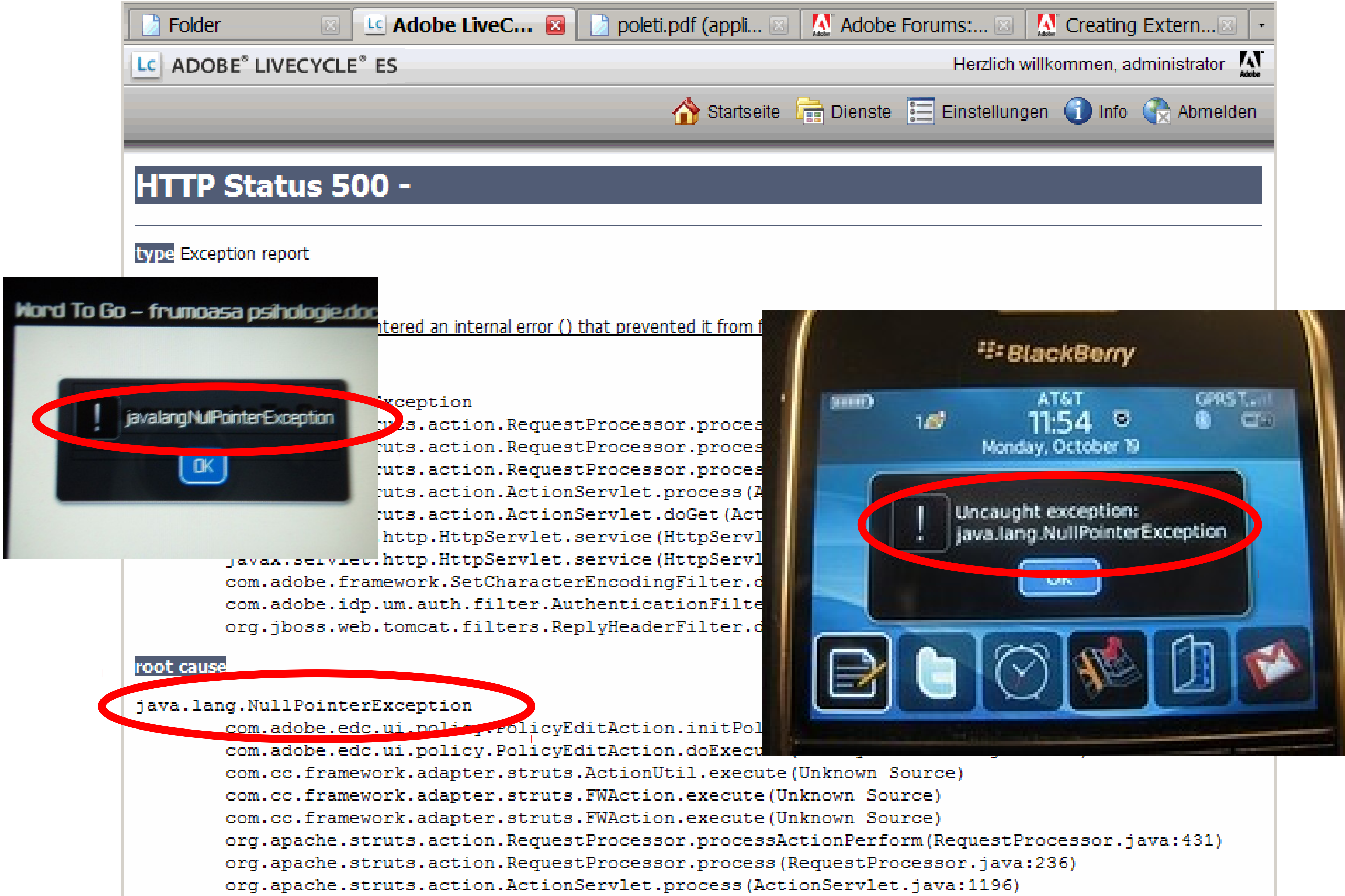
Building and Using Pluggable Type-Checkers

Werner M. Dietl

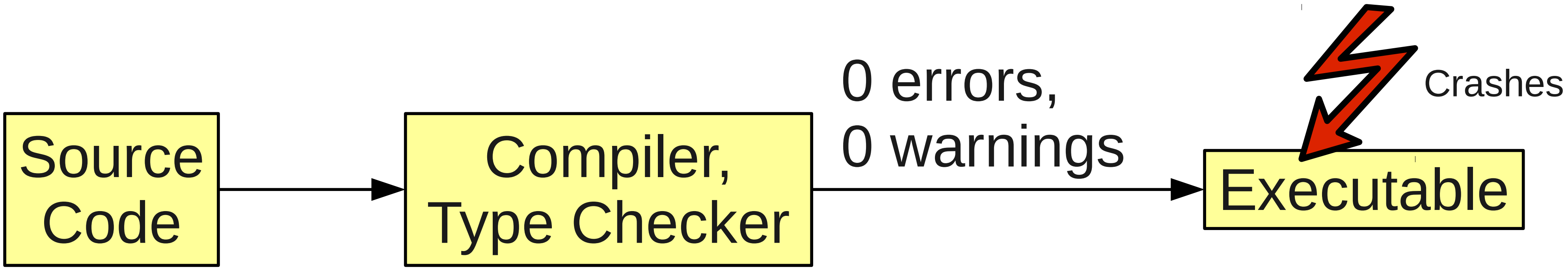
Joint work with:
Stephanie Dietzel, Michael D. Ernst,
Kıvanç Muşlu, and Todd W. Schiller

UNIVERSITY *of* WASHINGTON
COMPUTER SCIENCE & ENGINEERING

Software still has errors



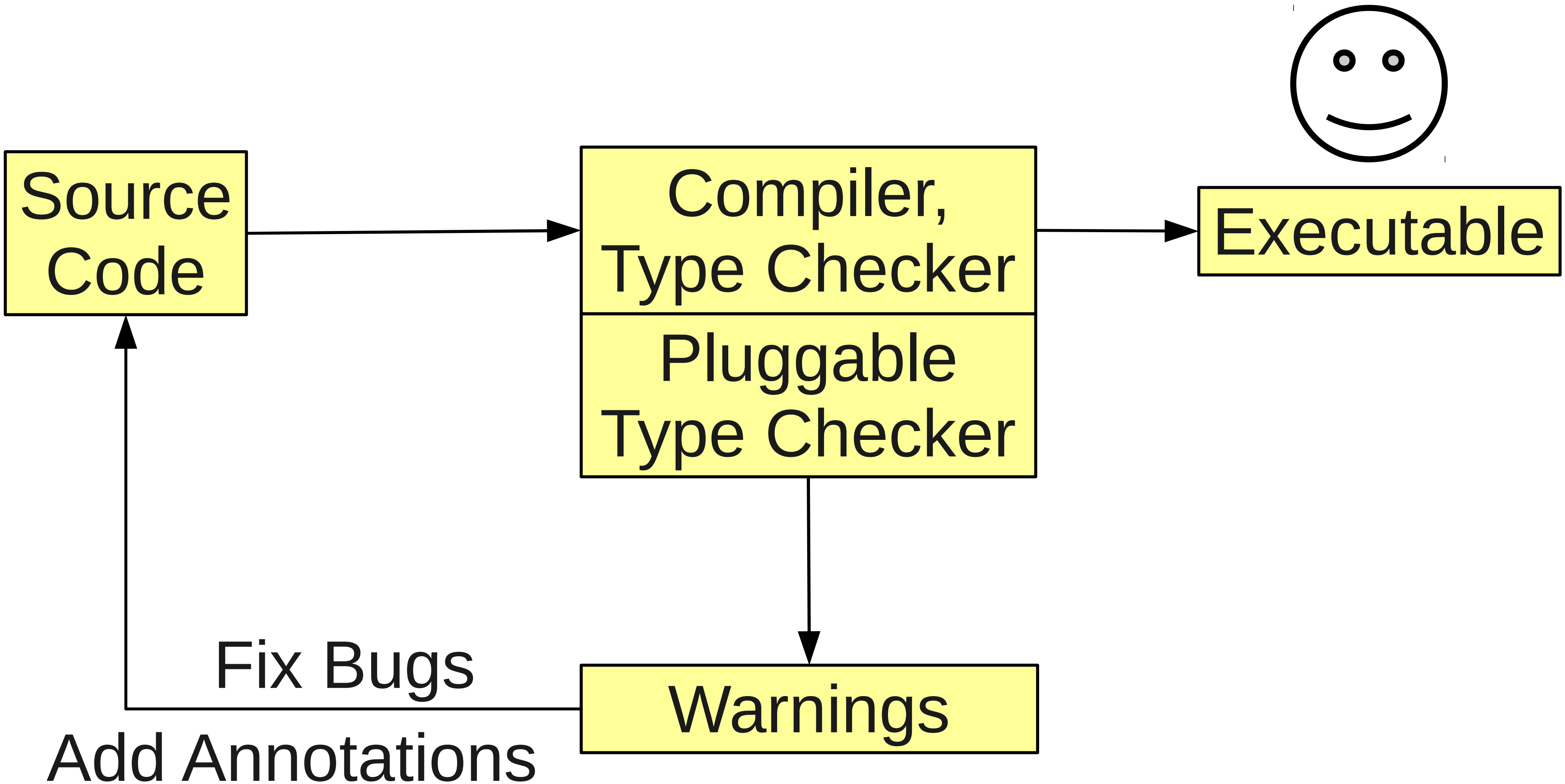
Static type systems



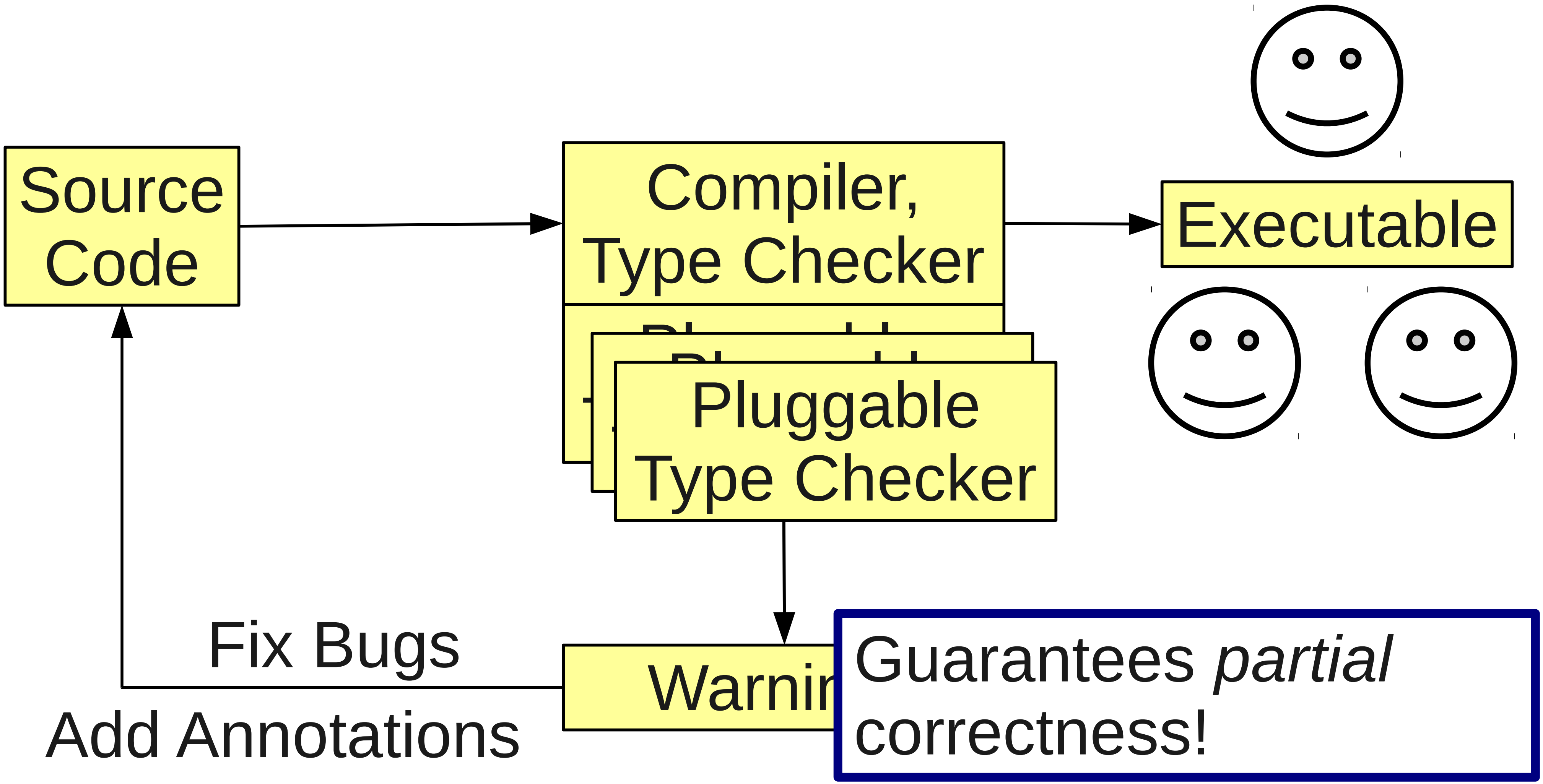
Static type systems

- Java/C# provide limited type systems
- Static type systems could prevent:
  - Null-pointer exceptions [Fähndrich & Leino '03]
  - Unwanted mutations [Tschantz & Ernst '05]
  - Concurrency errors [Boyapati et al. '02, Cunningham et al. '07]
- Express additional facts about a program
- Statically ensure absence of certain errors

Pluggable type checkers



Pluggable type checkers



Pluggable type systems

Example: Ensure encrypted communication

```
void send(@Encrypted String msg) {…}

@Encrypted String msg1 = ...;
send(msg1);   // OK

String msg2 = ....;
send(msg2);   // Warning!
```

The Checker Framework

- A framework for pluggable type checkers
- “Plugs” into the OpenJDK compiler
- Easy to use

```
javac -processor EncryptionChecker …
```

- Eclipse plug-in, Ant and Maven integration

Lack of uptake of pluggable types

Common assumptions:

- **Testing finds** all important **bugs**
- Usage **adds** annotation **clutter**
- **Learning** their usage is **hard**
- **Building** checkers is **difficult**

These were true before the Checker Framework.

Do they still apply?

Our contribution: case studies

- **Checkers reveal** important latent **bugs**
  - Ran on 2 million LOC of real-world code
  - Found 40 user-visible bugs, hundreds of mistakes
- **Annotation** overhead is **low**
  - Mean 2.6 annotations per kLOC
- **Learning** their usage is **easy**
  - Used successfully by first-year CS majors
- **Building** checkers is **easy**
  - New users developed 3 new realistic checkers

Kinds of case studies

- 2 existing type checkers
  - Absence of null-pointer exceptions
  - Correct use of object and reference equality
- 3 new type checkers
  - Correct compiler message key substitution
  - Consistent use of integer constants as enums
  - Consistency of Java class name strings
- Classroom study
  - Nullness checker used by first-year CS majors

Case study subject programs

Swing:	610 kLOC
Lucene:	479 kLOC
[illegible]	257 kLOC
[illegible]es):	231 kLOC
[illegible]	222 kLOC
JabRef:	117 kLOC
Google Collections:	78 kLOC
GanttProject:	69 kLOC
ASM:	33 kLOC
Checker Framework:	31 kLOC
Annotation File Utilities:	17 kLOC

We manually annotated each program for one type system until all warnings were eliminated.

Outline

1. Motivation
2. **Checkers reveal** important latent **bugs**
3. **Annotation** overhead is **low**
4. **Learning** the usage is **easy**
5. **Building** checkers is **easy**

1. Checkers reveal important latent bugs

Nullness Checker:

- 9 crashing bugs in Google Collections
  - 45000 tests (2/3 of the LOC)
  - Uses FindBugs @Nullable annotations, no FindBugs warnings
- >90 bugs in Daikon

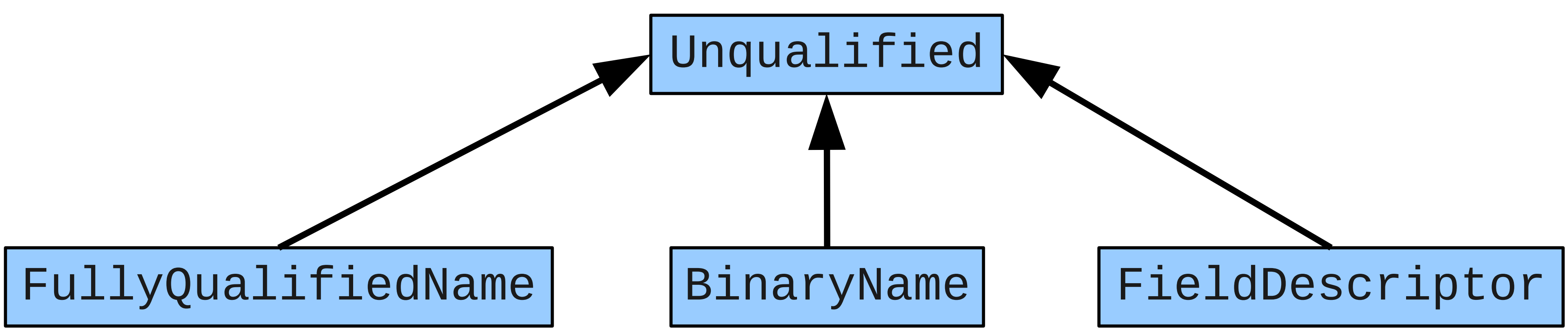
Reveals bugs: null-pointer exceptions

Example from Google Collections:

```
class ForMapWithDefault {
  @Nullable Object defaultValue;
  public int hashCode() {
    return map.hashCode() +
      defaultValue.hashCode();
  }          java.lang.NullPointerException
…
}
```

Reveals bugs: Java signatures

- JDK's String representations of class names:
  - Fully qualified names: `package.Outer.Inner`
  - Binary names: `package.Outer$Inner`
  - Field descriptors: `Lpackage/Outer$Inner;`
- Important to keep them separated



Reveals bugs: Java signatures

Signature Checker:

- 11 crashing bugs in OpenJDK
- 13 in libraries

Reveals bugs: Java signatures

Example from `java.lang.Class`:

```
static Class<?> forName(String className)
```

"Returns the Class object associated with the class or interface with the given string name. ...
Parameters:
 className - the **fully qualified name** of the
  desired class"

`java.lang.ClassNotFoundException`

```
Class.forName("package.Outer.Inner")
```

```
Class.forName("package.Outer$Inner")
```

`OK!`

2. Annotation overhead is low

Nullness:	`13` Ann./kLOC
Signature:	`1.5` Ann./kLOC
Fenum:	`1.1` Ann./kLOC
Interning:	`0.52` Ann./kLOC
Compiler Msgs.:	`0.35` Ann./kLOC

Annotation overhead is low

- Good defaults
  - Non-Null Except Locals reflects common usage
    - Fields, parameters, … are **`@NonNull`**
    - Only local variables are **`@Nullable`**
  - Define defaults using the tree kind, type kind, or regular expressions
- Flow-sensitive local inference

```
@Nullable Object o;
o = new Object();
o.toString(); // OK! o inferred non-null!
```

3. Learning their usage is easy

- 28 first-year CS majors at UW
- Assignment: prove absence of NPE
  - Mean code size: 9 kLOC
- Result: all students fixed unknown bugs!
- Invested time:
  - 2 hours of demos and instructions
  - 5.6 hours spent on assignment on average

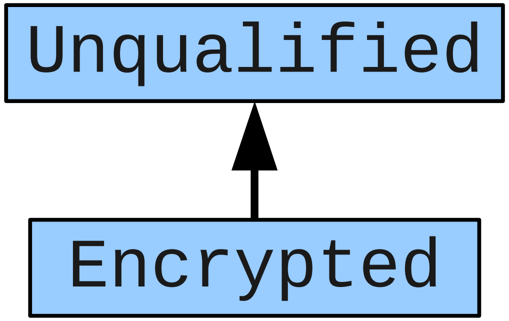
4. Building checkers is easy

Example: Ensure encrypted communication

```
void send(@Encrypted String msg) {…}
```

```
@Encrypted String msg1 = ...;
send(msg1);   // OK
```

```
String msg2 = ....;
send(msg2);   // Warning!
```

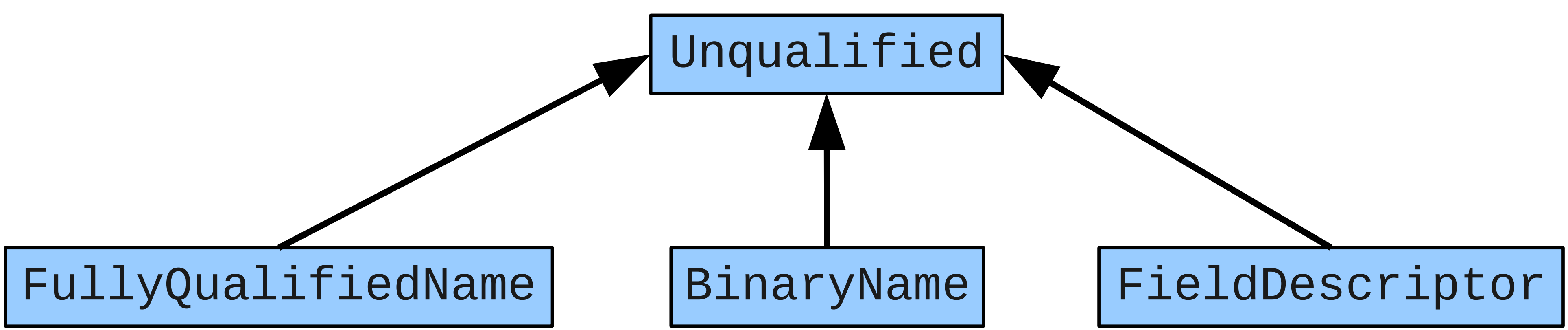


The complete checker:

```
@TypeQualifier
@SubtypeOf(Unqualified.class)
public @interface Encrypted {}
```

Signature String Checker

- JDK's String representations of class names:
  - Fully qualified names: `package.Outer.Inner`
  - Binary names: `package.Outer$Inner`
  - Field descriptors: `Lpackage/Outer$Inner;`
- Important to keep them separated



Signature String Checker

Type Qualifiers

```
@TypeQualifier
@SubtypeOf({Unqualified.class})
@ImplicitFor(stringPatterns="^[A-Za-z_]
  [A-Za-z_0-9]*(\\.[A-Za-z_][A-Za-z_0-9]
  *)*(\\[\\])*$")
public @interface FullyQualifiedName {}

@TypeQualifier
@SubtypeOf({Unqualified.class})
@ImplicitFor(stringPatterns="^[A-Za-z_]
  [A-Za-z_0-9]*(\\.[A-Za-z_][A-Za-z_0-9]
  *)*(\\$[A-Za-z_][A-Za-z_0-9]*)?(\\[\\]
  )*$")
public @interface BinaryName {}

@TypeQualifier
@SubtypeOf({Unqualified.class})
@ImplicitFor(stringPatterns="^\\[*([BCDF
  IJSZ]|L[A-Za-z_][A-Za-z_0-9]*(/[A-Za-z_]
  [A-Za-z_0-9]*)*(\\$[A-Za-z_][A-Za-z_0-9]
  *)?;)$")
public @interface FieldDescriptor {}
```

```
@TypeQualifier
@SubtypeOf({BinaryName.class,
  FullyQualifiedName.class})
public @interface SourceName {}

@TypeQualifier
@SubtypeOf({Unqualified.class})
public @interface MethodDescriptor {}

@TypeQualifier
@SubtypeOf({BinaryName.class,
  FieldDescriptor.class, SourceName.class,
  FullyQualifiedName.class,
MethodDescriptor.class})
@ImplicitFor(trees={Tree.Kind.NULL_LITERAL})
public @interface SignatureBottom {}
```

```
@TypeQualifiers({BinaryName.class,
  FullyQualifiedName.class, SourceName.class,
  FieldDescriptor.class, Unqualified.class,
  MethodDescriptor.class, SignatureBottom.class})
public final class SignatureChecker
  extends BaseTypeChecker {}
```

Type Checker

Signature String Checker

- Written by a first-year graduate student without prior experience with the framework
- Found 11 crashing bugs in OpenJDK, 13 more in libraries
- Example:

```
class Class<T> {
  Class<?> forName(@BinaryName String className);
  @BinaryName String getName();
  @FullyQualifiedName String getCanonicalName();
}
String name = myclass.getCanonicalName();
Class.forName(name); // Warning
```

Building complex checkers is possible

Nullness Checker is actually 3 checkers:

- Correct object initialization
- Nullness itself
- Correct usage of keys in map accesses

Refined defaulting:

- Refined flow-sensitive inference
- Heuristics for `Map.get` behavior

Checker Code Sizes

Nullness Checker:	`4311` LOC
Interning Checker:	`960` LOC
Fake Enumerations Checker:	`489` LOC
Signature Strings Checker:	`167` LOC
Compiler Messages Checker:	`70` LOC

Applicability of type checkers

- Many properties amenable to static checking
  - Concurrency
  - Object encapsulation
  - Energy efficiency
  - Even dependencies on external information
- Look for properties that depend on the static structure and not the behavior of code
- Value sound results over heuristics

Conclusions

1. **Checkers reveal** important latent **bugs**
2. **Annotation** overhead is **low**
3. **Learning** their usage is **easy**
4. **Building** checkers is **easy**

It is easy to improve the quality of your Java code, and you should start today!

`http://checker-framework.googlecode.com/`