JAVA INHERITANCE USAGE – A REPLICATION STUDY

By **Cigdem Aytekin**

CWI & Master Student @ UvA

Thesis Supervisor: **Tijs van der Storm**, CWI

- We replicate the study "What programmers do with inheritance in Java?"
- Our goal is to validate the results
- We analyze the source code instead of the byte code
- Our preliminary results are close to original results, except for one research question (down-call).

OUR STUDY

Ewan Tempero

Hong Yul Yang

James Noble

ORIGINAL STUDY - WHAT PROGRAMMERS DO WITH INHERITANCE IN JAVA?

- Concentrates on the **usage** of the inheritance relationships in a project.

- Article is published in ECOOP – European Conference in Object Oriented Programming – proceedings in 2013, pp 577 – 601.

WHAT PROGRAMMERS DO WITH INHERITANCE IN JAVA?

```java
public class P {
    void p() {
    }
    void c() {
    }
}
public class C extends P {
    void c() {
    }
}
```

```java
public class N {
    void run() {
        C aC = new C();
        aC.p();  // reuse
    }
}
```

DEFINITION VS. USAGE

ORIGINAL STUDY - GOALS

- Propose **a model** for inheritance usage,
- Analyze a corpus of **open source Java systems** – Qualitas Corpus* with this model at hand,
- Make the study **replicable**
  - Qualitas Corpus is available
  - Analysis results are reported in detail per project.

* E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton and J. Noble 'Qualitas Corpus: A Curated Collection of Java Code for Empirical Studies' *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, pp336–345, December 2010.

- Subtype usage
- Reuse (external and internal)
- Downcall
- Other uses of inheritance

INHERITANCE MODEL - CONCEPTS

```java
public class N {
    P aPMethod(P aP) {
        return new C();      // return statement
    }
    void run() {
        P aP = new C();      // assignment
        aPMethod(new C());  // parameter passing
        aP = (P)(new C());  // cast
    }
}
```

DEFINITIONS (1) – SUBTYPE

```java
public class P {
    void p() {
    }
    void c() {
    }
}
public class C extends P {
    void c() {
    }
     void t() {
        p();        // Internal
    }
}
```

```java
public class N {
    void run() {
        C aC = new C();
        aC.p();  // External
    }
}
```

DEFINITIONS (2) - REUSE

DOWNCALL : Late-bound self-reference

```java
public class P {
    void p() {
        c();
    }
    void c() {
    }
}
public class C extends P {
    void c() {
    }
}
```

```java
public class N {
    void run() {
        C aC = new C();
        aC.p();  // Downcall
    }
}
```

- To what extent is **late-bound self-reference** relied on in the designs of Java systems?
- To what extent is inheritance used in Java in order to express a **subtype** relationship?
- To what extent can inheritance be replaced by composition – how often do we see **reuse**?
- What **other inheritance idioms** are in common use in Java systems?

ORIGINAL STUDY - RESEARCH QUESTIONS

```java
public class P {
    void p() { }
}

public class C extends P {
}
```

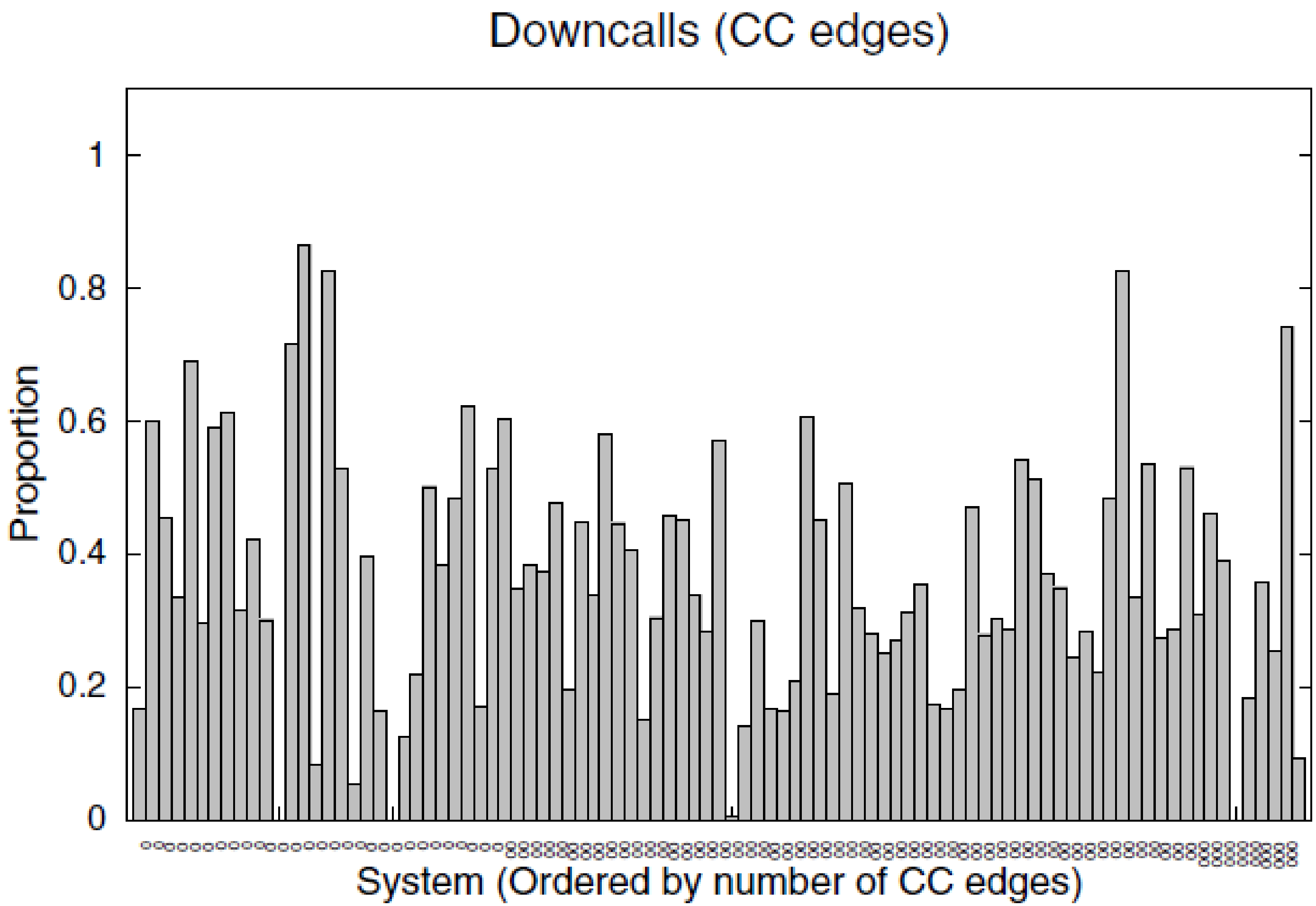
```java
public class N {
    void run() {
        C aC = new C();
        aC.p();  // External
    }
}
```

The number of **child- parent pairs** that show some type of usage (pair <C,P> is counted).

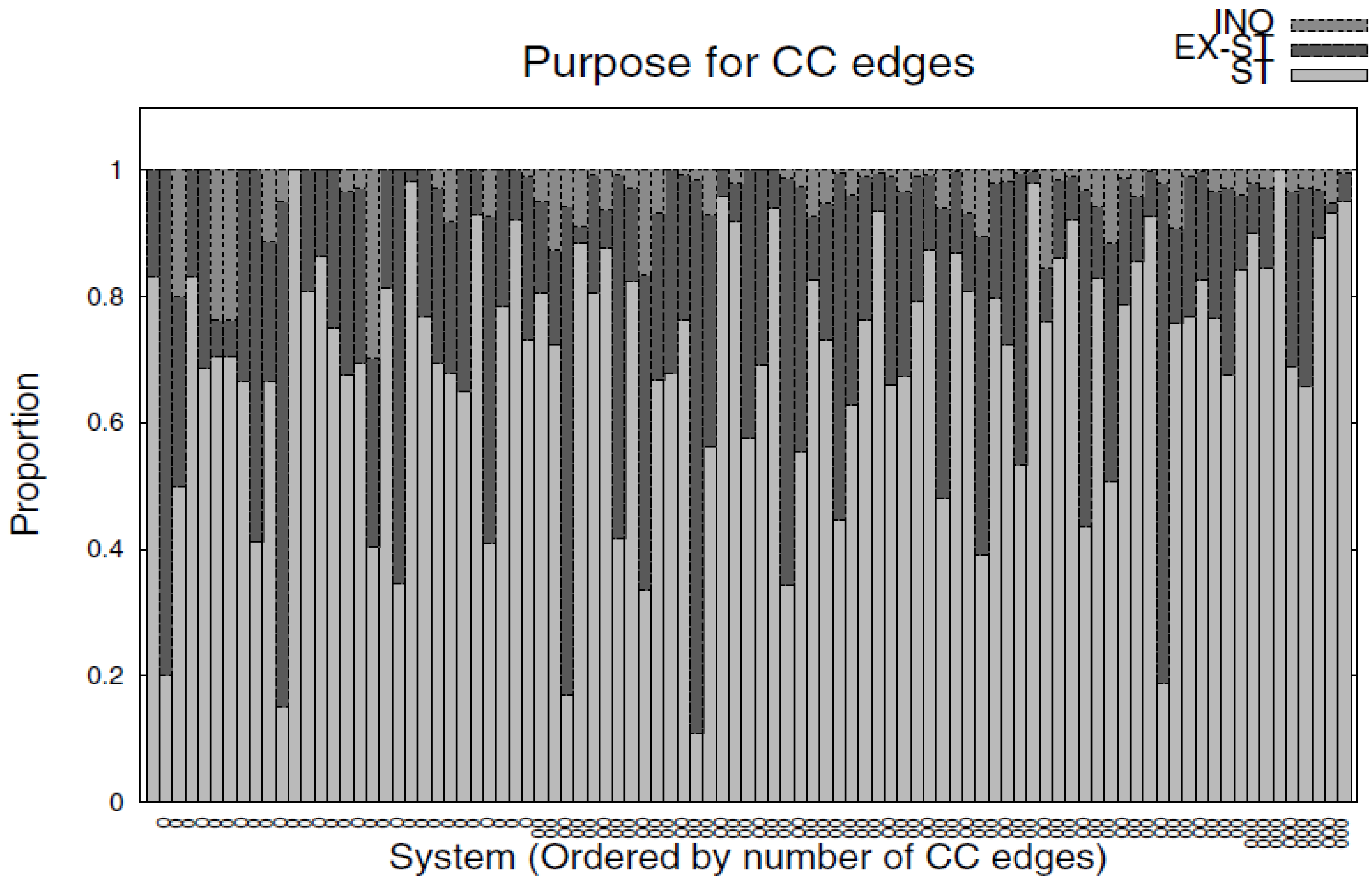
And **not the number of occurrences** (aC.p()). It does not matter if usage occurs once or many times.

WHAT COUNTS…

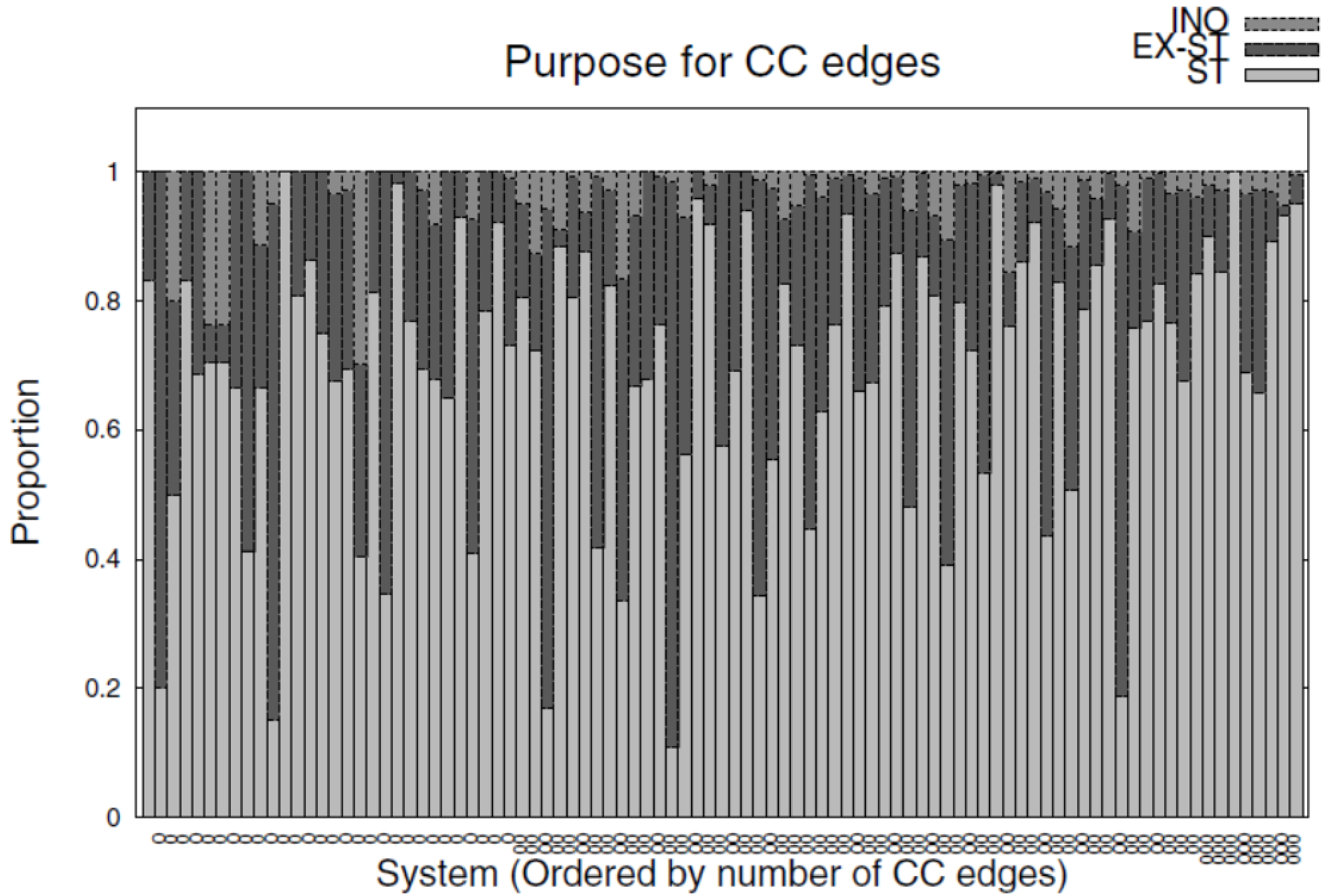
RESULTS OF THE ORIGINAL STUDY…



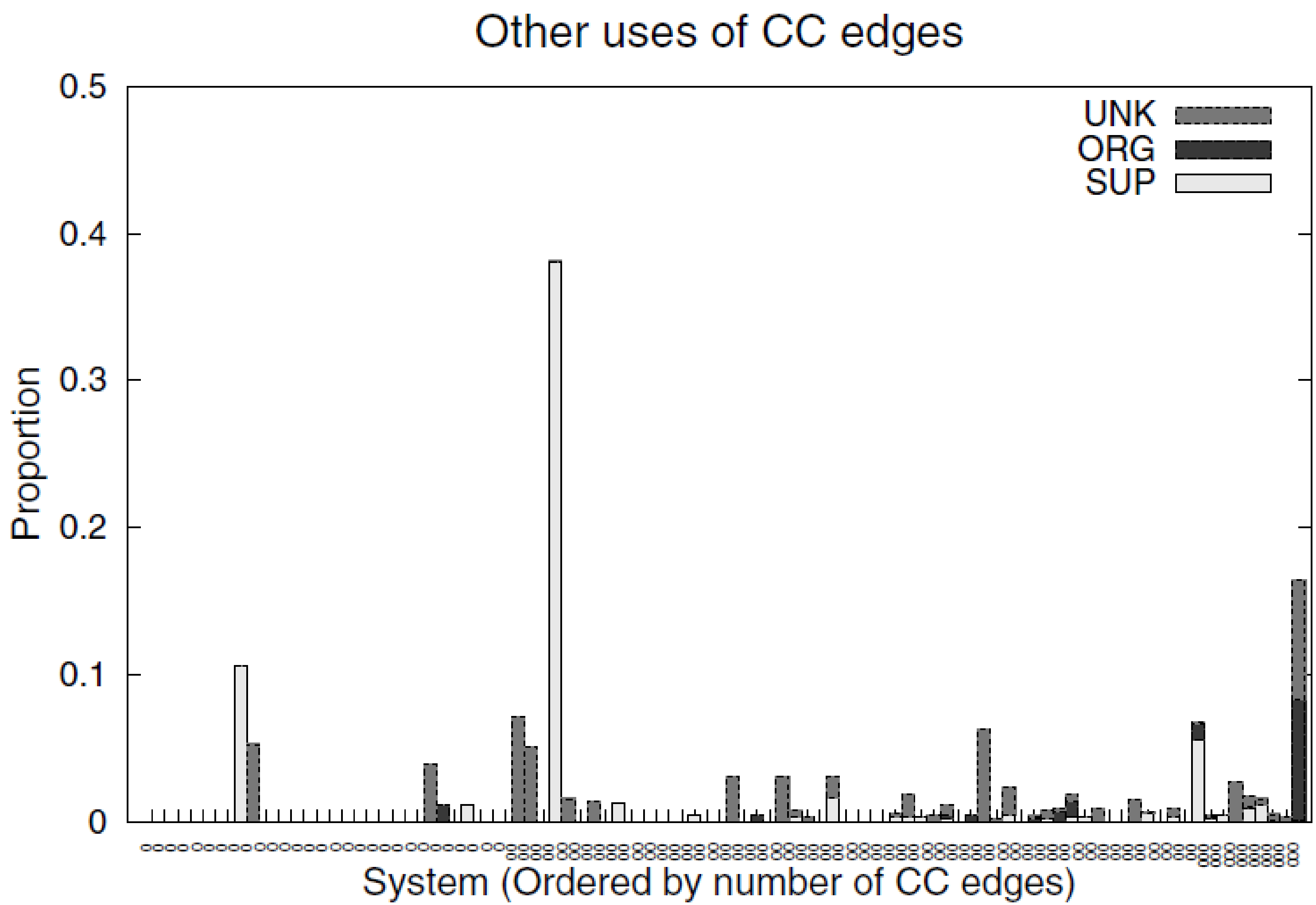
RESULTS – DOWNCALL – 33 %



RESULTS – SUBTYPE – 66 %



RESULTS – EXTERNAL REUSE – 22 %



RESULTS - OTHER INHERITANCE USES

REPLICATION STUDY

Thesis for Master Software Engineering in UvA.

Planning to finish up at the end of August 2014.

To contribute to **the validation** of the original study results,

From a **different perspective**: Java source code instead of byte code.

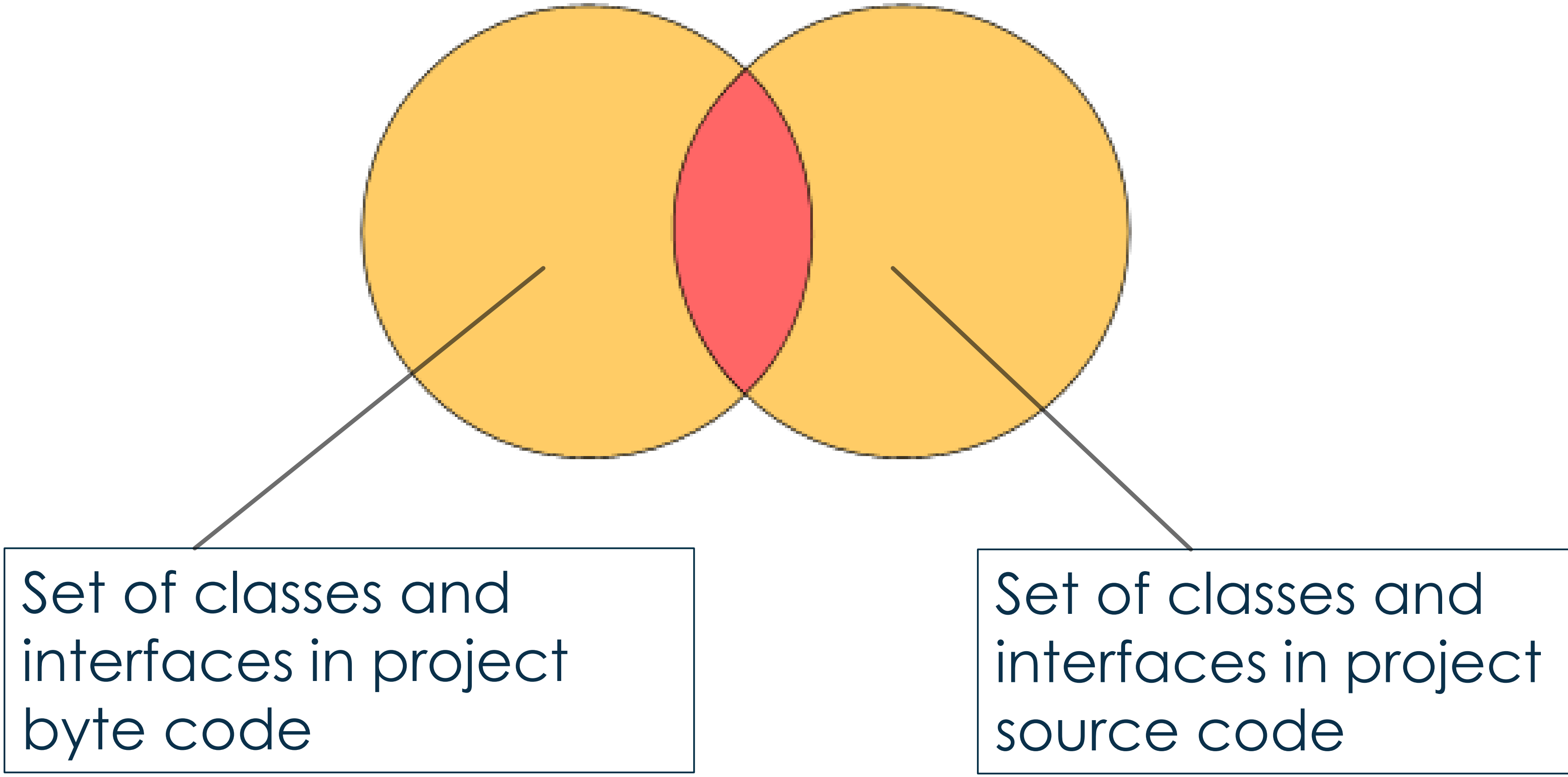
GOALS OF OUR STUDY

```java
public static class GTToken
                extends Token
  { int realKind =
     JavaParser15Constants.GT;
  }
```

```
public ....Token$GTToken();

Code:
     stack=2, locals=1, args_size=1
        0: aload_0
        1: invokespecial #10   // Method net/sourceforge/cobertura/javancss/parser/java15/Token."<init>":()V
        4: aload_0
        5: bipush        126
        7: putfield      #12     // Field realKind:I
       10: return
```

DIFFERENCE 1: SOURCE CODE VS. BYTE CODE



DIFFERENCE 2: SET OF TYPES ANALYZED

Rascal

REPLICATION STUDY - IMPLEMENTATION

RASCAL...

- A programming language from CWI - SWAT group
- Used for meta-programming (software analysis, transformation, DSL implementations, ...)
- Integrated in Eclipse
- Open source
- Extensive online documentation and interactive tutorial
- Syntax similar to Java

```
public void run() {
    set[Declaration] projectASTs =
        createAstsFromEclipseProject(|project://cobertura-1.9.4.1|,
true);
    map [loc, num] methodsMap = ();
    for (anAST <- projectASTs) {
      visit (anAST) {
            case m1:\methodCall(_, _, _, _) : {
                if (m1@decl in methodsMap) {
                     [m1@decl] = methodsMap[m1@decl] + 1;
                }
                else {methodsMap += (m1@decl : 1) ; }
            }
            case m2:\methodCall(_,_,_) : {
                if (m2@decl in methodsMap) {
                     [m2@decl] = methodsMap[m2@decl] + 1;
                }
                else {methodsMap += (m2@decl : 1); };
            }
        }
    }
map [loc, num] frequentlyCalledMethods =
(aMethod : methodsMap[aMethod] | aMethod <- methodsMap,
methodsMap[aMethod] > 400 );
}
```

Create Abstract Syntax Trees

Visit the nodes of the Abstract Syntax Tree

Pattern match – pick the method calls only

Map comprehension – filter result

PRELIMINARY RESULTS .

DOWNCALL – LESS

We expect **less** downcall cases than the original study

```java
public class P {
    void p() {
        P aP = new P();
        aP.c();   // receiver
    }
    void c() {
    }
}
public class C extends P {
    void c() {
    }
}
```

```java
public class N {
    void run() {
        C aC = new C();
        aC.p();  // Downcall
    }
}
```

```java
public static class GTToken
                extends Token
  { int realKind =
     JavaParser15Constants.GT;
  }
```

```
public ....Token$GTToken();

Code:
     stack=2, locals=1, args_size=1
        0: aload_0
        1: invokespecial #10   // Method net/sourceforge/cobertura/javancss/parser/java15/Token."<init>":()V
        4: aload_0
        5: bipush        126
        7: putfield      #12     // Field realKind:I
       10: return
```

COMPILER INSERTS A METHOD CALL…

- We expect **approximately same** percentage of **subtype** cases – perhaps a little bit less
- Reason: **our analysis limitation** in parameter passing to the methods of third party types.

SUBTYPE – THE SAME

- We expect approximately **same** results,
- May be a bit less than original study – again, **calls that are inserted by compiler** may cause this.

EXTERNAL REUSE – THE SAME

- We expect more or less the same results about other uses
- Mainly that subtype and reuse explain most of the cases,
- Some minor differences can be expected.

OTHER USES – THE SAME

- Carry out the study from a different perspective: Java source code analysis,
- Verify the original study results for subtype, reuse and other uses of inheritance,
- Bring up a question about down-call – why the source code analysis deliver less down-call cases?

SUMMARY - OUR CONTRIBUTION

USEFUL LINKS

Original article: http://link.springer.com/chapter/10.1007/978-3-642-39038-8_24

Original study results: https://www.cs.auckland.ac.nz/~ewan/qualitas/studies/inheritance/

Qualitas Corpus website: http://qualitascorpus.com/

Qualitas.compiled Corpus: http://java.labsoft.dcc.ufmg.br/qualitas.class/index.html

Rascal homepage: http://www.rascal-mpl.org/

NICE CHALLENGE... JAVA GENERICS

```
public class GParent <T> {
}

public class GChild <T>
         extends GParent <T>
{
}

public class GRunner {
  GParent <P> aP= new Gchild
                   <P> ();
}
```

In the byte code, the type erasure is already applied for Java Generics,

In the source code, we have to find the correct mapping.

Challenging for subtype analysis during parameter passing.

We are working on it...

Not finished yet...

We will be ready soon....

RESULTS OF THE REPLICATION STUDY

CC, CI or II	Stands for:	
CC	Class – Class	Child and parent are both classes.
CI	Class - Interface	Child is a class and parent is an interface
II	Interface – Interface	Child and parent are both interfaces.

DEFINITIONS (4) - CC, CI AND II

```java
public class P {
    void p() {
    }
}

public class C extends P {
}


public class G extends C {
}
```

```java
public class N {
    void run() {
        C aC = new C();
        aC.p(); // direct
        G aG = new G();
        aG.p(); // indirect
    }
}
```

DEFINITIONS (6) - DIRECT VS. INDIRECT (REUSE AND SUBTYPE ONLY)

```java
public class P {
}

public class C extends P {
}

public class G extends C {
}
```

The relationship btw. C and P is **explicit.**

The relationship btw. G and P is **implicit**.

DEFINITIONS (5) – EXPLICIT VS. IMPLICIT

DEFINITIONS (7) – OTHER USES OF INHERITANCE

- Category
- Constants
- Framework
- Generic
- Marker
- Super

```java
public class ConstantParent {
 static final int anI = 0;
 static final double aD = 2.9d;
 static final String anS =
"333";
}

public class ConstantChild
         extends ConstantParent
{
     // .......
}
```

```java
public interface AMarkerParent
{
    // an empty interface
}

public class AnImplementor
      implements AMarkerParent
{
    // .......
}
```

DEFINITIONS (8) - CONSTANTS, MARKER

- Only classes and interfaces (no enums, exceptions, annotations)
- No types from third party libraries analyzed
- Heuristics are used for defining framework and generic relations
- Static analysis – downcall results may overstate the reality.

LIMITATIONS – ORIGINAL STUDY

```java
public class P {
    void p() {
        c();
    }
    void c() {
    }
}
public class C extends P {
    void c() {
    }
}
```

Original study: They do **not** look for
an explicit method call on an object of child type...

```java
public class N {
    void run() {
        C aC = new C();
        aC.p();  // Downcall
    }
}
```

IMPORTANT DETAIL - DOWNCALL

```java
public class P {
    void p() {
    }
}

public class C extends P {
}

public class G extends C {
}
```

> aG is reusing a method of P. But this is counted as external reuse between G and C...

```java
public class N {
    void run() {
        G aG = new G();
        aG.p();      // indirect reuse.
    }
}
```

IMPORTANT DETAIL: INDIRECT CALLS ARE COUNTED