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).
ORIGINAL STUDY - WHAT PROGRAMMERS DO WITH INHERITANCE IN JAVA?
Concentrates on the **usage** of the inheritance relationships in a project.

public class P {
    void p() {
    }
    void c() {
    }
}

public class C extends P {
    void c() {
    }
}

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

---

INHERITANCE MODEL - CONCEPTS

- Subtype usage
- Reuse (external and internal)
- Downcall
- Other uses of inheritance
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
    }
}
public class P {
    void p() {
    }
    void c() {
    }
}

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

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

DEFINITIONS (2) - REUSE
PUBLIC CLASS P {
  void p() {
    c();
  }
  void c() {
  }
}
PUBLIC CLASS C EXTENDS P {
  void c() {
  }
}
PUBLIC CLASS N {
  void run() {
    C aC = new C();
    aC.p();  // Downcall
  }
}

DEFINITIONS (3) - 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?
public class P {
    void p() {
    }
}

public class C extends P {
}

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.
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
DIFFERENCE 1:
SOURCE CODE VS. BYTE CODE
DIFFERENCE 2: SET OF TYPES ANALYZED

Set of classes and interfaces in project byte code

Set of classes and interfaces in project source code
REPLICATION STUDY - IMPLEMENTATION
- 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 );
}
PRELIMINARY RESULTS .
We expect **less** downcall cases than the original study.
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
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.
We expect approximately *same* results,
May be a bit less than original study – again, *calls that are inserted by compiler* may cause this.
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.
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?
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/
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....
<table>
<thead>
<tr>
<th>CC, CI or II</th>
<th>Stands for:</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CC</td>
<td>Class – Class</td>
<td>Child and parent are both classes.</td>
</tr>
<tr>
<td>CI</td>
<td>Class - Interface</td>
<td>Child is a class and parent is an interface</td>
</tr>
<tr>
<td>II</td>
<td>Interface – Interface</td>
<td>Child and parent are both interfaces.</td>
</tr>
</tbody>
</table>
```java
public class P {
    void p() {
    }
}

public class C extends P {
}

public class G extends C {
}

public class N {
    void run() {
        C aC = new C();
        aC.p(); // direct
        G aG = new G();
        aG.p(); // indirect
    }
}
```
The relationship btw. C and P is **explicit**.

The relationship btw. G and P is **implicit**.
Category
Constants
Framework
Generic
Marker
Super
public class ConstantParent {
    static final int anI = 0;
    static final double aD = 2.9d;
    static final String anS = "333";
}

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

public interface AMarkerParent
{
    // an empty interface
}

public class AnImplementor
    implements AMarkerParent
{
    // .......
    // .......
}
- 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.
public class P {
    void p() {
        c();
    }
    void c() {
    }
}
public class C extends P {
    void c() {
    }
}
public class N {
    void run() {
        C aC = new C();
        aC.p(); // Downcall
    }
}
public class P {
    void p() {
    }
}

public class C extends P {
}

public class G extends C {
}

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