WELCOME TO THE TUTORIAL!

Tutorial Team

Kadiray Karakaya

Stefan Schott

Jonas Klauke

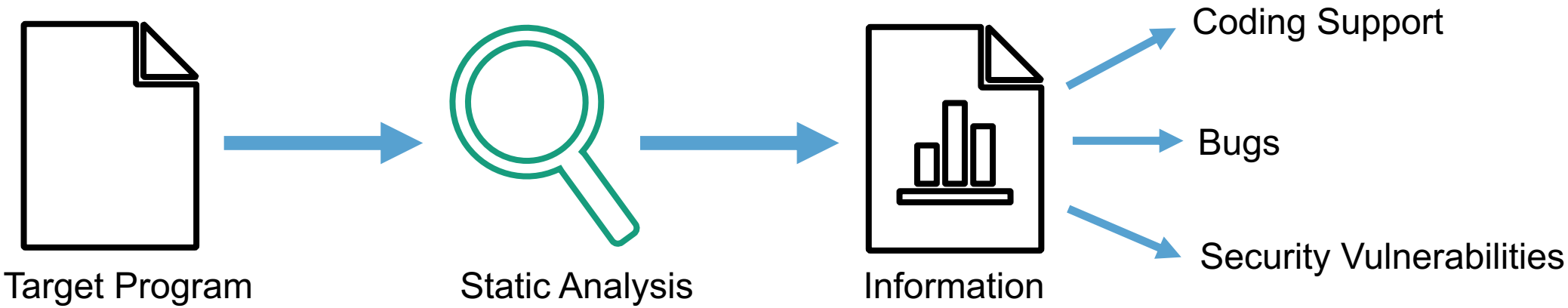
Eric Bodden

Markus Schmidt

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

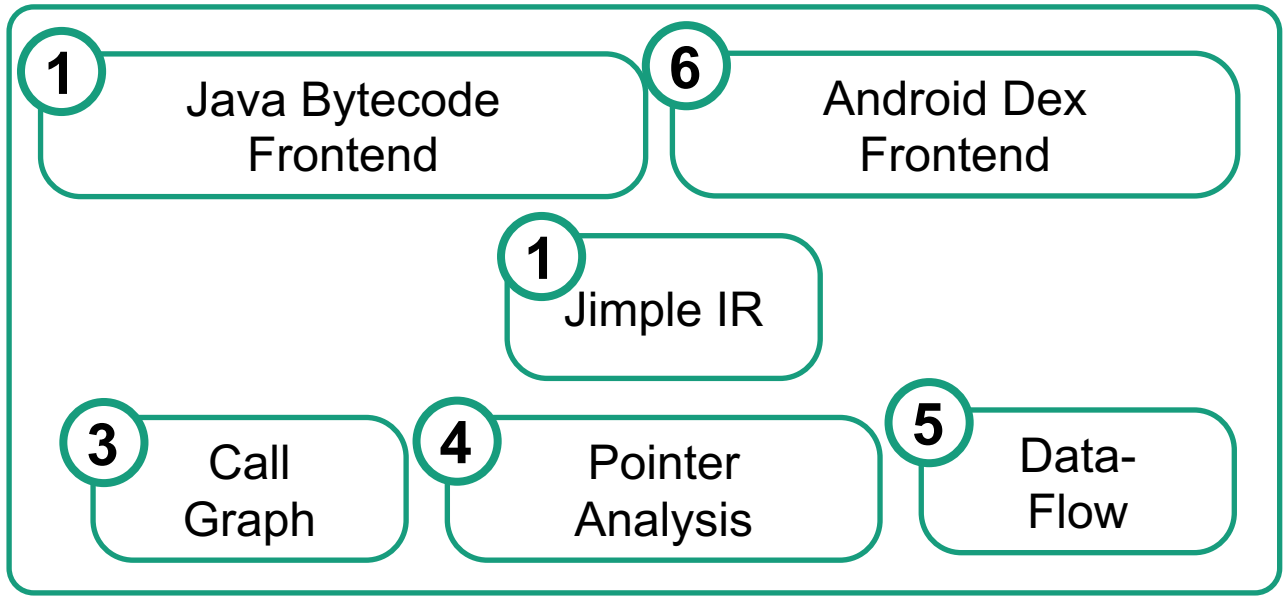
Fraunhofer
IEM

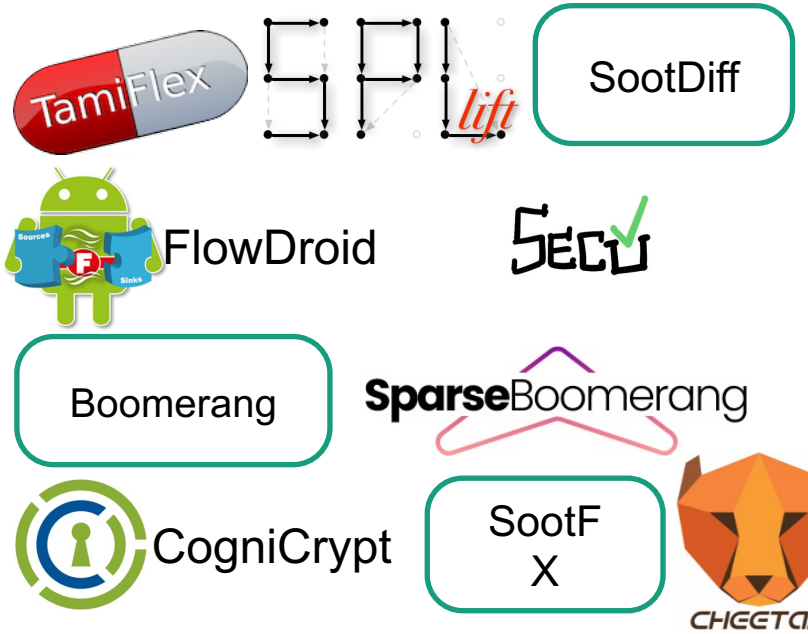
Context: Static Program Analysis

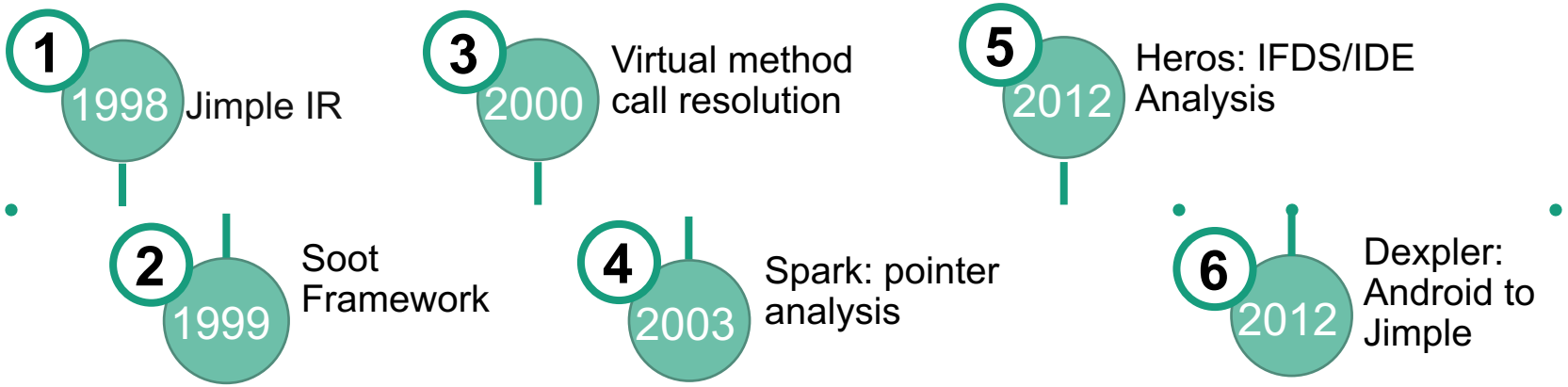


SootUp is a redesign of the **Soot** static analysis framework.

Soot: A Brief History



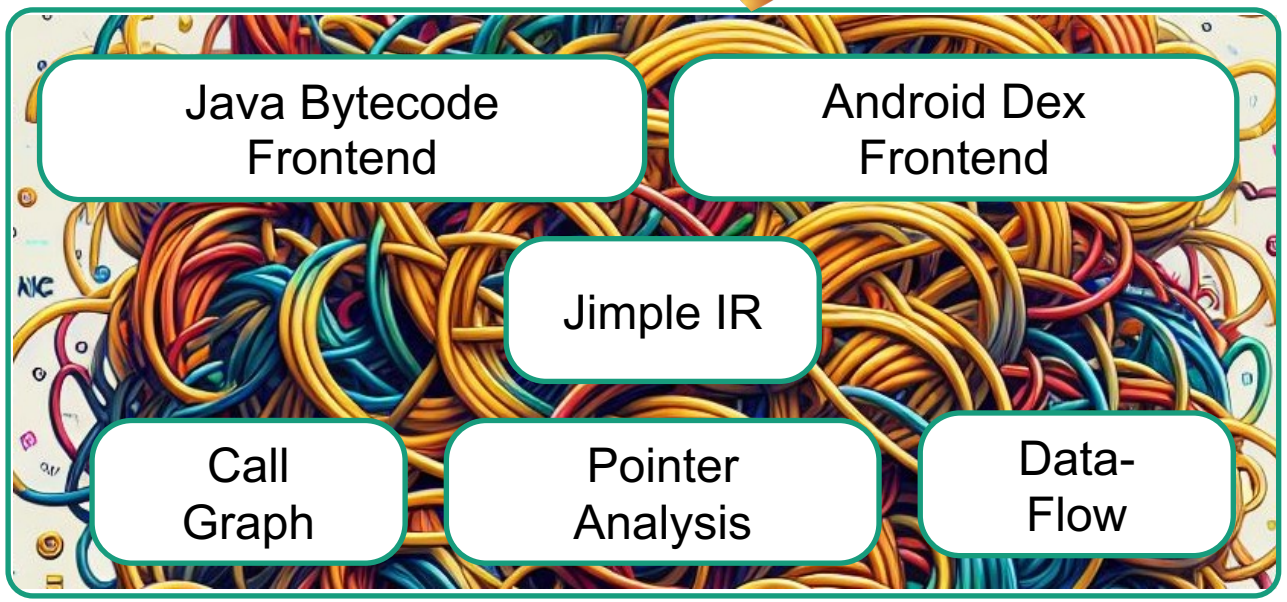




HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Fraunhofer IEM

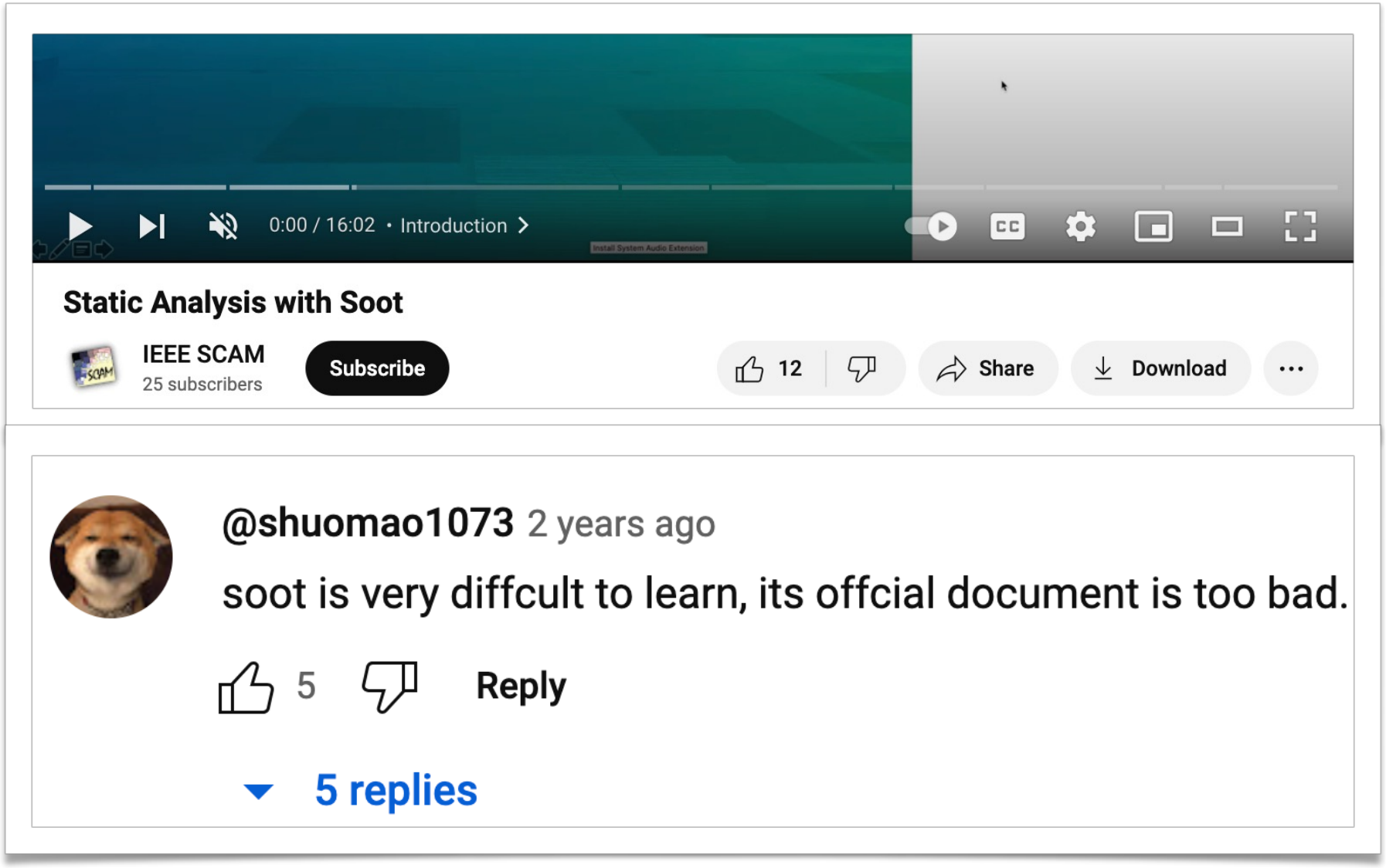
Soot: Strengths & Drawbacks



- Jimple IR
  - Local vars instead of stack in bytecode
  - No nested scopes, explicit control-flow
- Call Graph algorithms
  - CHA, RTA, DTA, VTA
- SPARK pointer analysis framework
  - Whole program
  - Context- and flow-insens.
- Heros Data-Flow Analysis framework
  - IFDS/IDE

Frequently extended to support new features → Suboptimal Design

“Soot is very difficult to learn”



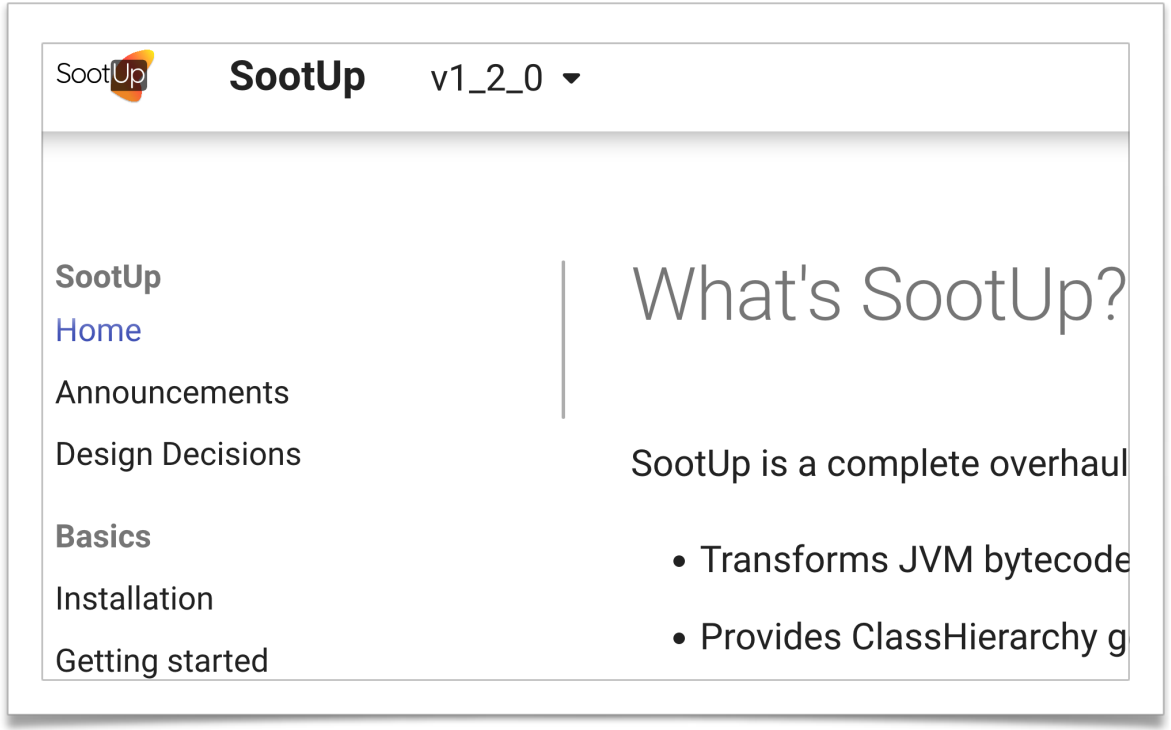
DFG Deutsche Forschungsgemeinschaft
German Research Foundation

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

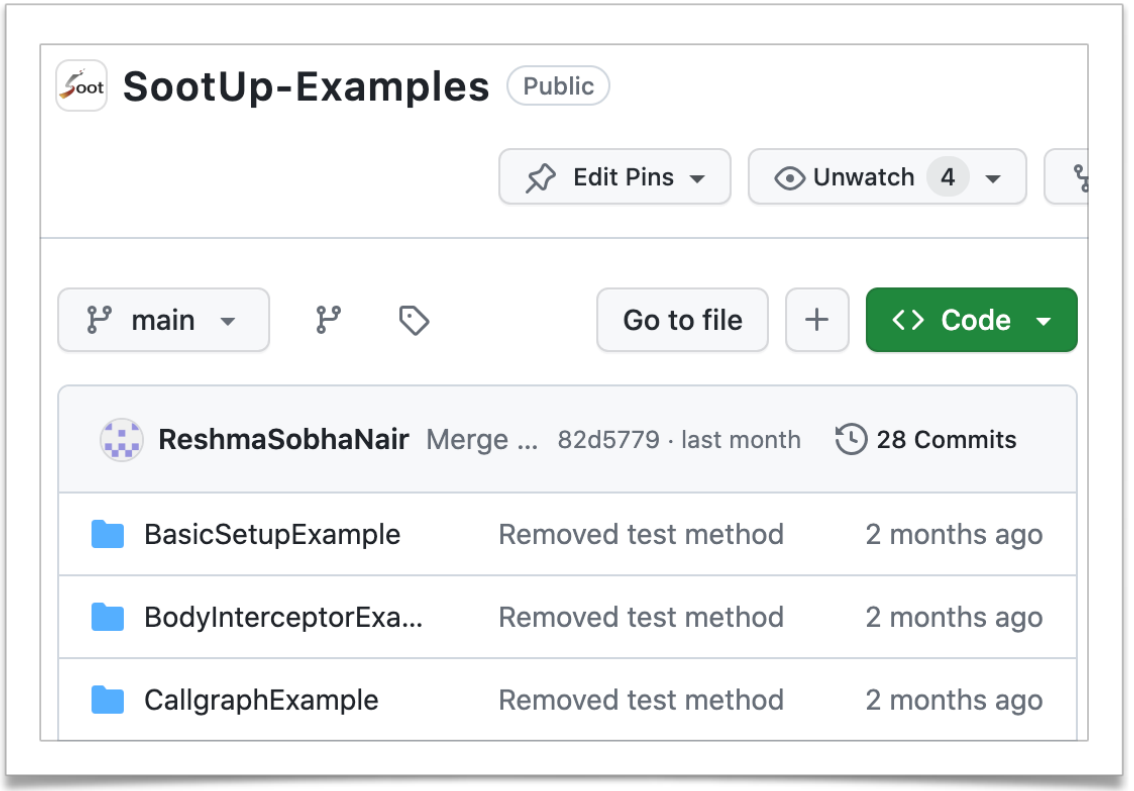
Fraunhofer
IEM

SootUp aims to be beginner friendly

- Dedicated web page for:
  - Up-to-date documentation
  - Tutorials



- Dedicated repository with running examples



HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Fraunhofer IEM

Development Process

- Actively maintained at Github
- No breaking changes through CI 

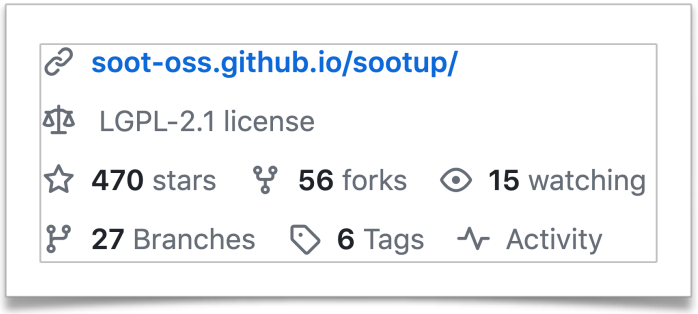
- High code coverage through systematic testing 

- Up-to-date documentation & tutorials 

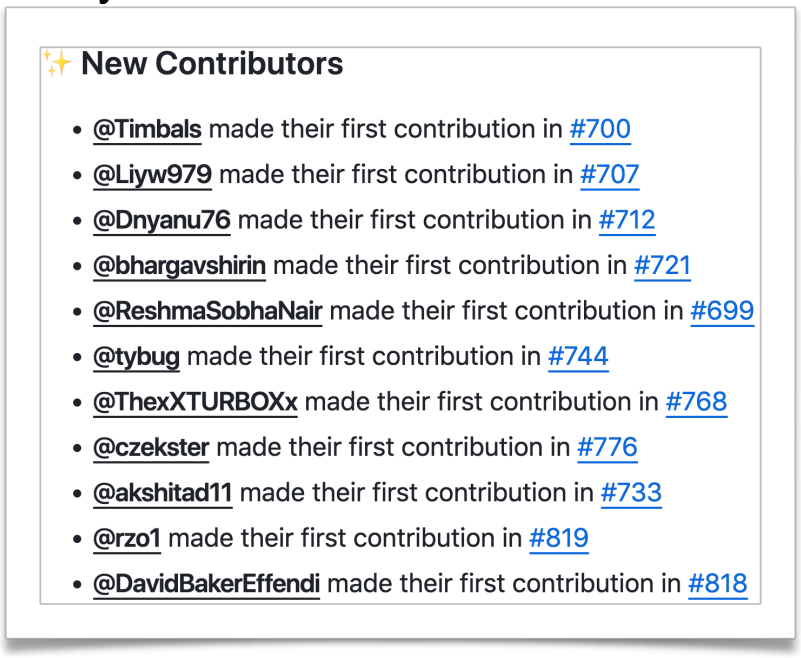
- Released at Maven Central 

Building a User Base

- Already an active user base



- Many outside contributors



- In Labs:
  - SootUp is used in our static analysis course at Paderborn University

- Hackathons (CTF):
  - 1st SootUp Hackathon, at UPB in May 2023
  - 2nd SootUp Hackathon, at UPB in June 2024

- Today: SootUp Tutorial!

Tutorial Program

What's next?

- Technical Introduction
- Exercise 1: Get to know the API
- Live Demo: Call Graph Construction
- Exercise 2: Taint analysis with SootUp and Heros
- Closing

TECHNICAL INTRODUCTION

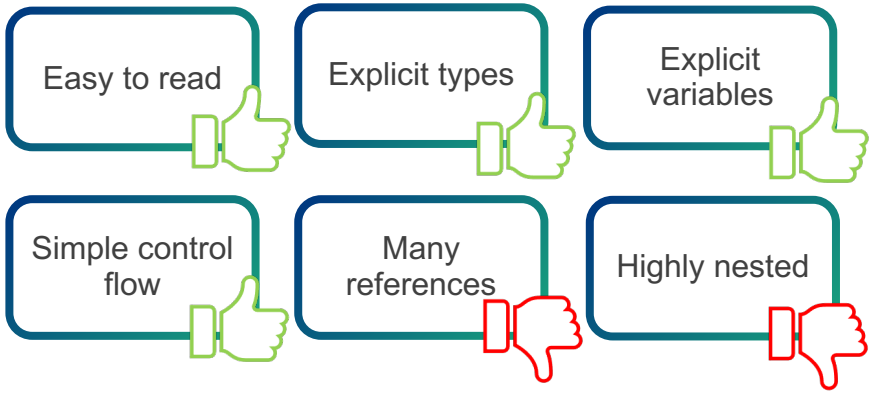
Source Code vs Bytecode vs Jimple

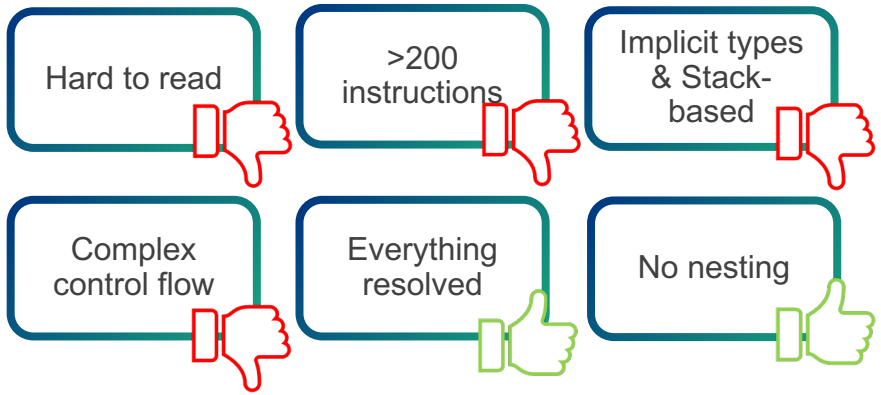
```java
public static void sampleMethod(int x) {
    if (x  % 2 == 0) {
        System.out.println("Even");
    } else {
        System.out.println("Odd");
    }
}
```

Java Source Code

```
1  public static sampleMethod(I)V
2    L0
3      ILOAD 0
4      ICONST_2
5      IREM
6      IFNE L1
7    L2
8      GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
9      LDC "Even"
10     INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
11     GOTO L3
12   L1
13     FRAME SAME
14     GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
15     LDC "Odd"
16     INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
17   L3
18     FRAME SAME
19     RETURN
20   L4
21     LOCALVARIABLE x I L0 L4 0
22     MAXSTACK = 2
23     MAXLOCALS = 1
24 }
```

Java Bytecode





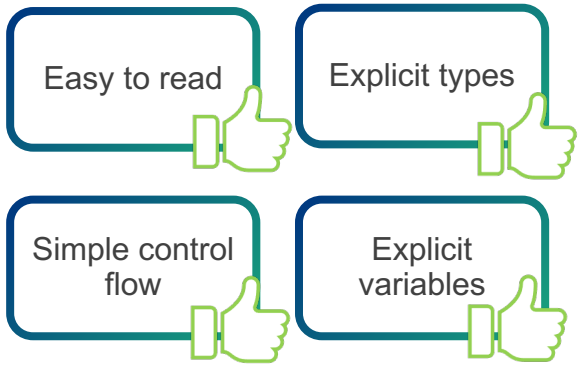
HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Fraunhofer IEM

Source Code vs Bytecode vs Jimple

```java
public static void sampleMethod(int x) {
    if (x  % 2 == 0) {
        System.out.println("Even");
    } else {
        System.out.println("Odd");
    }
}
```

Java Source Code



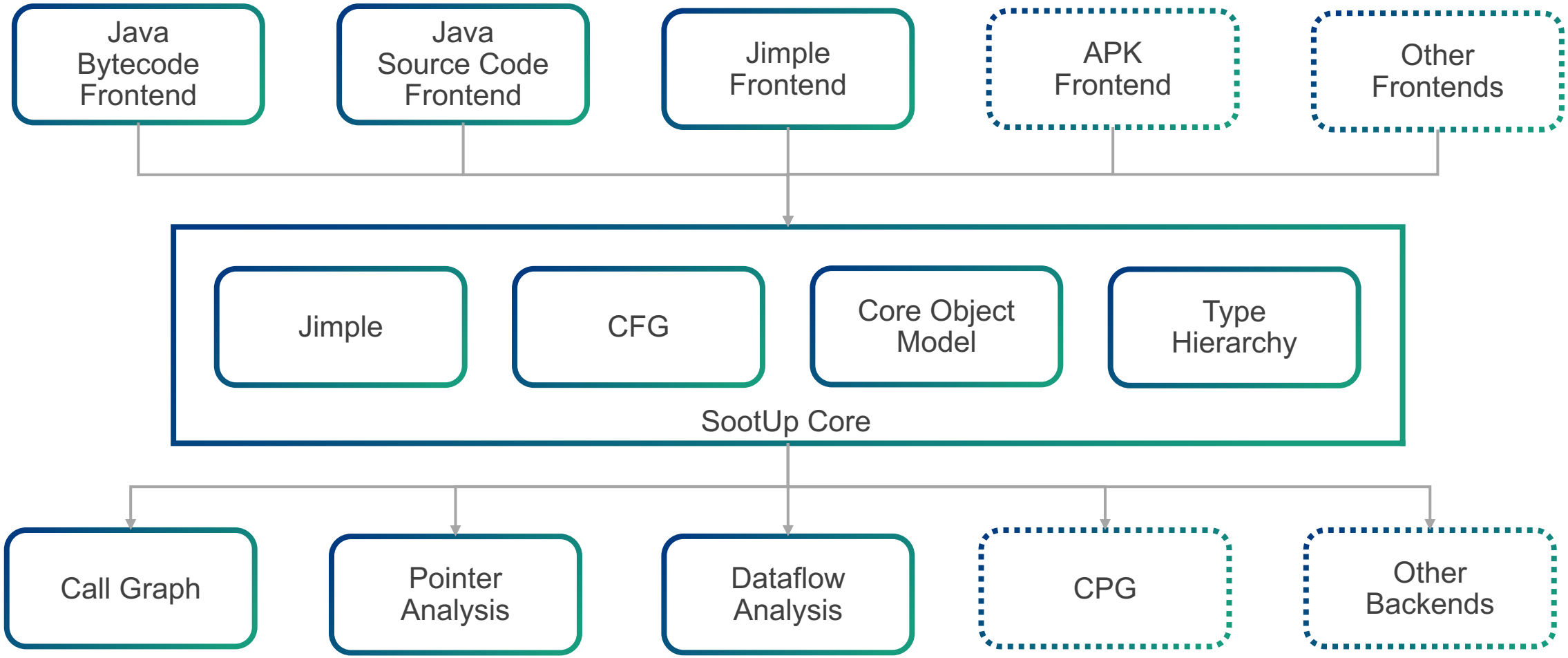

```
1 public static void sampleMethod(int)
2 {
3   int x, $stack1;
4   java.io.PrintStream $stack2, $stack3;
5
6   x := @parameter0: int;
7
8   $stack1 = x % 2;
9   if $stack1 ≠ 0 goto label1;
10
11  $stack3 = <java.lang.System: java.io.PrintStream out>;
12  virtualinvoke $stack3.<java.io.PrintStream:
13    void println(java.lang.String)>("Even");
14  goto label2;
15
16  label1:
17    $stack2 = <java.lang.System: java.io.PrintStream out>;
18    virtualinvoke $stack2.<java.io.PrintStream:
19      void println(java.lang.String)>("Odd");
20
21  label2:
22    return;
23 }
```

Jimple

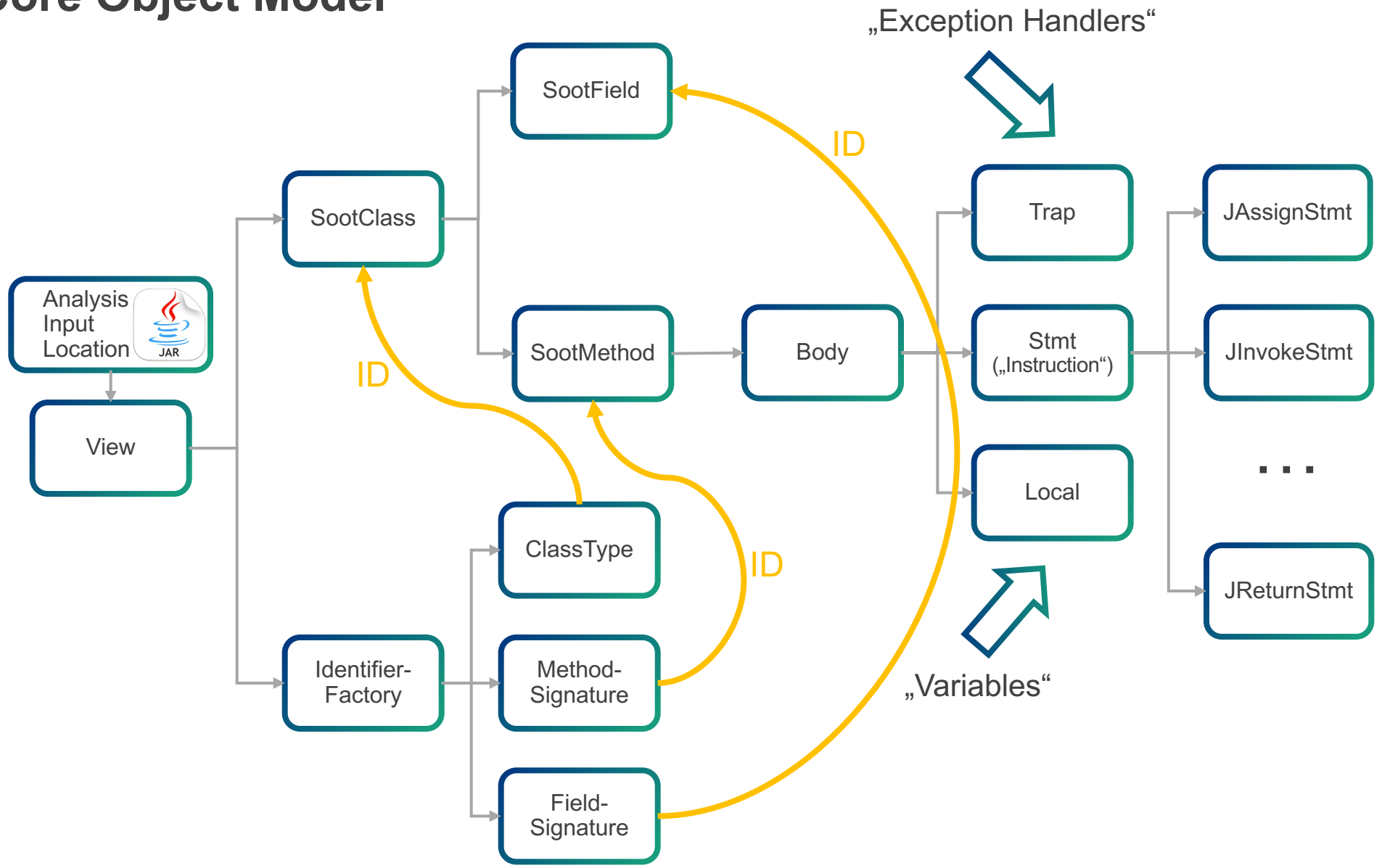
```
1 public static sampleMethod(I)V
2   L0
3     ILOAD 0
4     ICONST_2
5     IREM
6     IFNE L1
7   L2
8     GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
9     LDC "Even"
10    INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
11    GOTO L3
12  L1
13    FRAME SAME
14    GETSTATIC java/lang/System.out : Ljava/io/PrintStream;
15    LDC "Odd"
16    INVOKEVIRTUAL java/io/PrintStream.println (Ljava/lang/String;)V
17  L3
18    FRAME SAME
19    RETURN
20  L4
21    LOCALVARIABLE x I L0 L4 0
22    MAXSTACK = 2
23    MAXLOCALS = 1
24 }
```

Java Bytecode

Architecture and Modules



Core Object Model

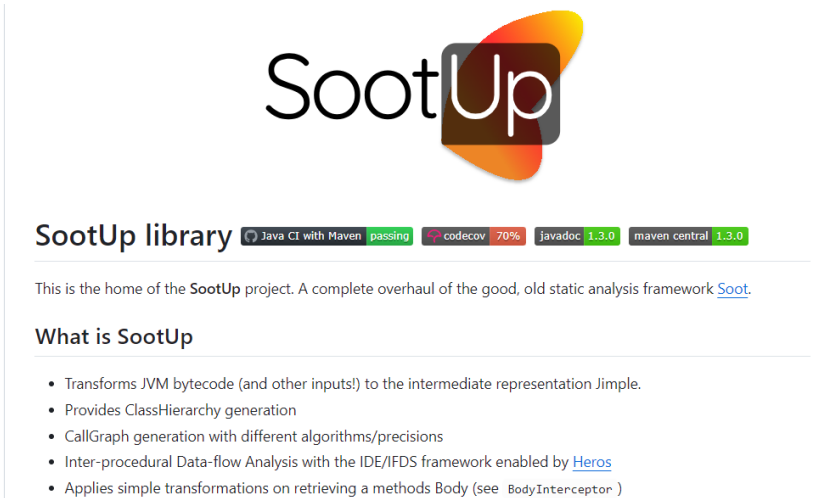


Basic API Usage

1.
2.
3.
4.

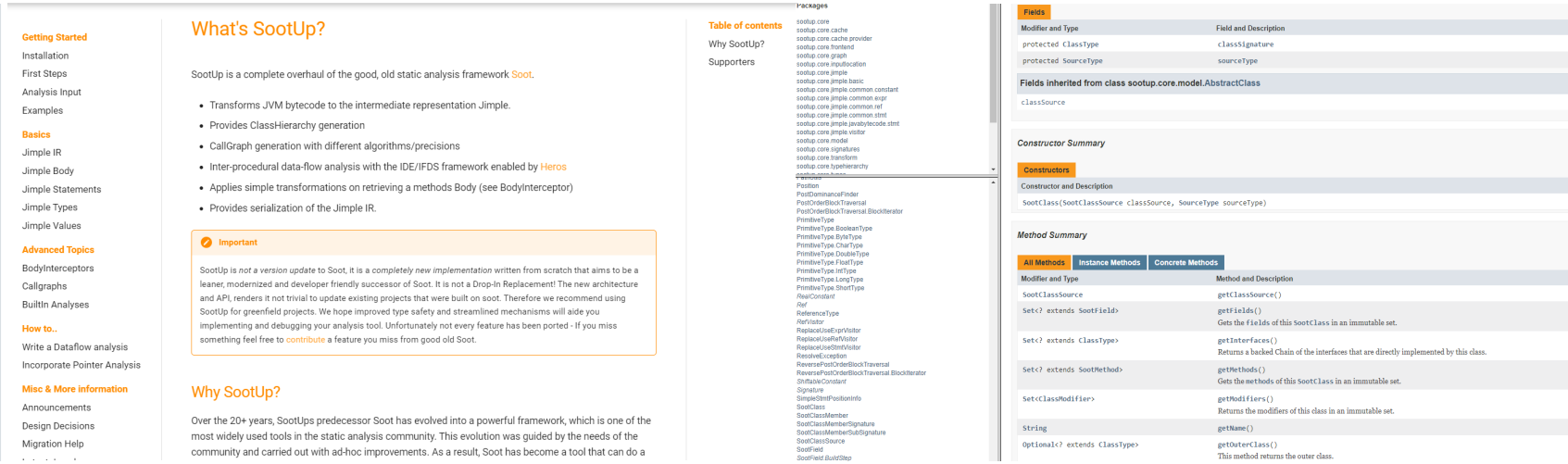
```
1  AnalysisInputLocation inputLocation =
2          new JavaClassPathAnalysisInputLocation("path2Binary");
3
4  JavaView view = new JavaView(inputLocation);
5
6  JavaClassType classType =
7          view.getIdentifierFactory().getClassType("example.HelloWorld");
8
9  Optional<JavaSootClass> sootClassOpt = view.getClass(classType);
10 JavaSootClass sootClass = sootClassOpt.get();
```

Helpful Resources



- Repository

https://github.com/soot-oss/SootUp



- Documentation

https://soot-oss.github.io/SootUp

- Javadoc

https://javadoc.io/doc/org.soot-oss/sootup.core

EXERCISE 1: GET TO KNOW THE API

Exercise 1: Get to know the API

- Task: Extract the following properties from the given Jar:
  1. Number of **classes**
  2. Number of **private methods**
  3. Number of **static fields**
  4. Number of **methods** where the **return type** is the same as its first **parameter type** (if exists)
  5. Number of **assignment statements** in method *„boolean test(java.lang.String)“* in class *„org.reflections.util.FilterBuilder“*

Steps to access the exercise:

1. git clone https://github.com/soot-oss/SootUp-Tutorial.git
2. git checkout 1_Exercise-1

OR

Download from: https://sootup-tutorial.surge.sh

Exercise 1: Helpful APIs

AnalysisInputLocation	
	new **JavaClassPathAnalysisInputLocation**(String path2jar)

JavaView	
	new **JavaView**(AnalysisInputLocation inputLocation)
IdentifierFactory	**getIdentifierFactory**()
Collection<JavaSootClass>	**getClasses**()
Optional<JavaSootMethod>	**getMethod**(MethodSignature sig)

IdentifierFactory	
ClassType	**getClassType**(String fullyQualifiedClassName)
MethodSignature	**getMethodSignature**(ClassType ct, String methodName, String returnType, List<String> parameterTypes)

JavaSootClass	
Optional<JavaSootMethod>	**getMethod**(String name, Iterable<Type> parameterTypes)
Set<JavaSootMethod>	**getMethods**()
Set<JavaSootField>	**getFields**()

JavaSootField	
boolean	**isStatic**()

JavaSootMethod	
boolean	**isPrivate**()
int	**getParameterCount**()
Type	**getReturnType**()
Type	**getParameterType**(int index)
boolean	**hasBody**()
Body	**getBody**()

Body	
List<Stmt>	**getStmts**()

Stmt
Subtypes: **JAssignStmt**, **JInvokeStmt**, **JReturnStmt**, **JIfStmt**, … (instanceof)

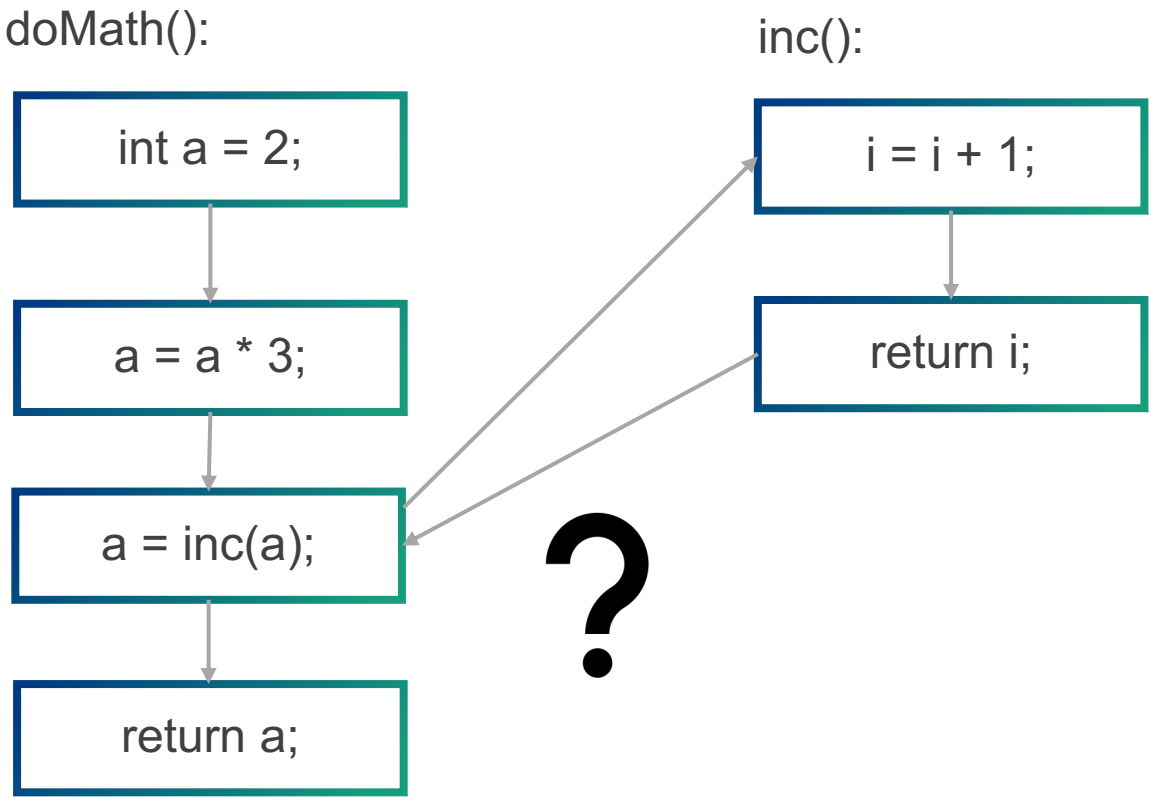
For more help check:

- Documentation: soot-oss.github.io/SootUp
- Javadoc: soot-oss.github.io/SootUp/apidocs

LIVE DEMO: CALL GRAPHS

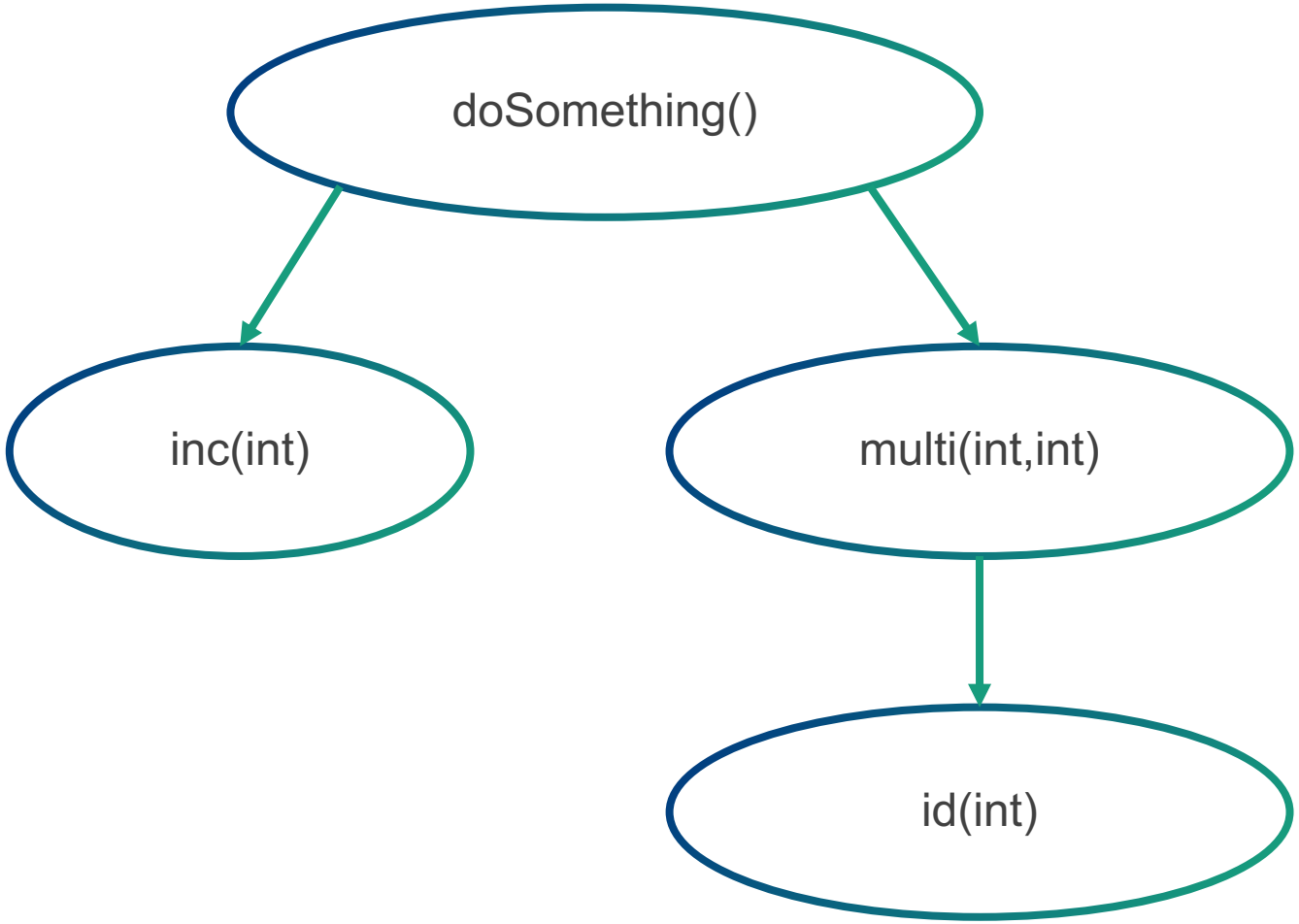
Interprocedural Analysis

```
public int doMath(){
  int a = 2;
  a= a*3
  a = inc(a);
  return a
}

public int inc(int i){
  return I + 1;
}
```



Call Graph

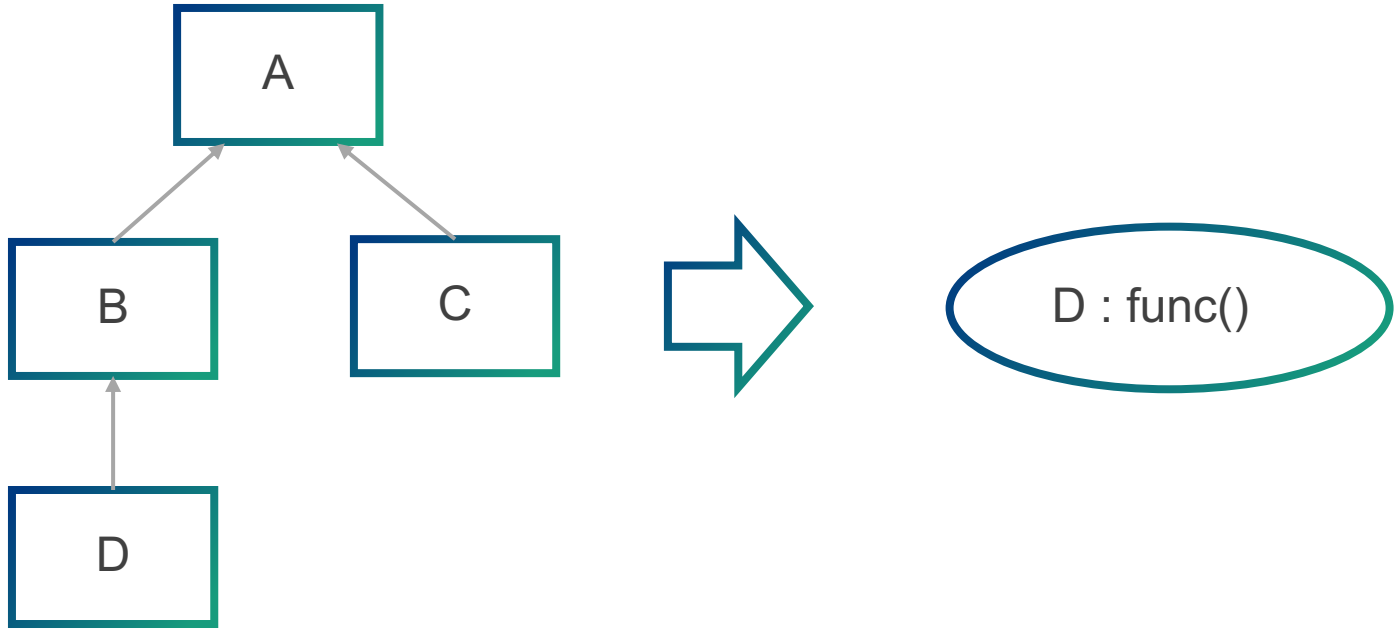
```
public int doSomething(){
  int a = 2;
  a = inc(a);
  a = inc(a);
  return multi(a,2);
}

public int inc(int i){
  return I + 1;
}

public int multi(int i, int d){
 return id(i) * d;
}
public int id(int i){
 return i;
}
```



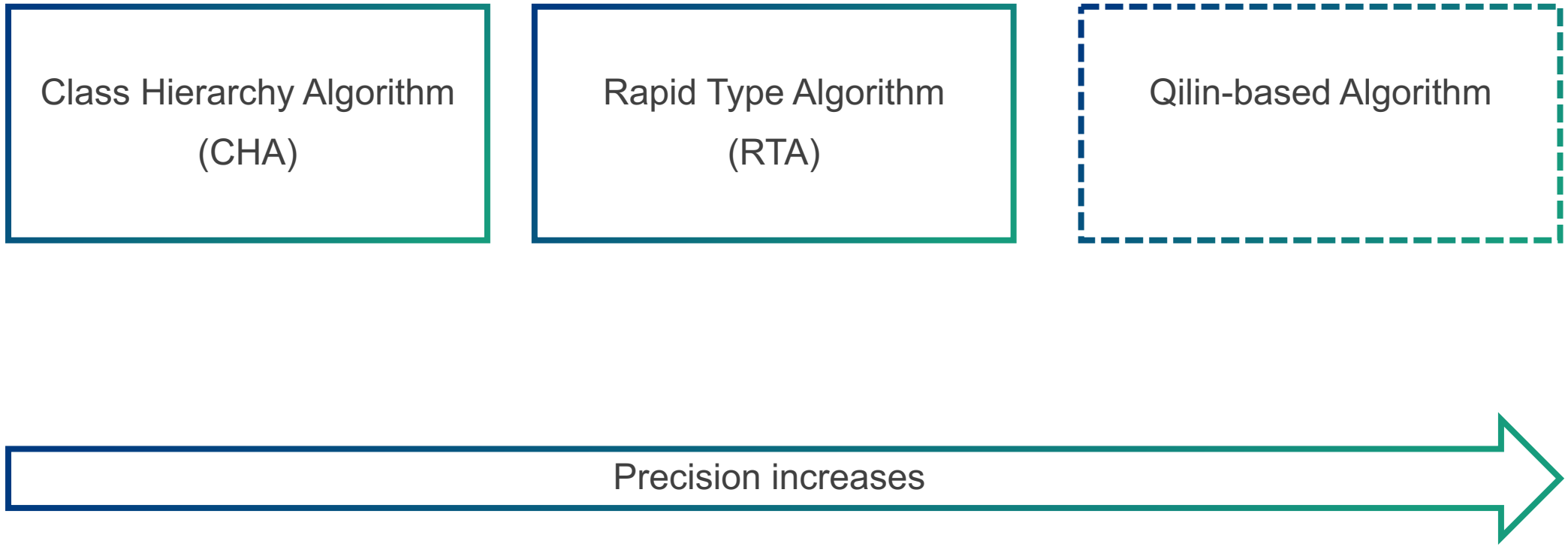
Polymorphism in Java

Virtual invokes

```
public void start (){
   C c = new C();
   A a = new D();
   a.func();
}
```



Supported Call Graph Algorithm

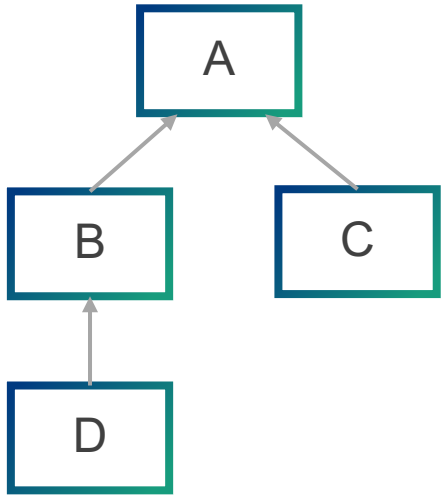


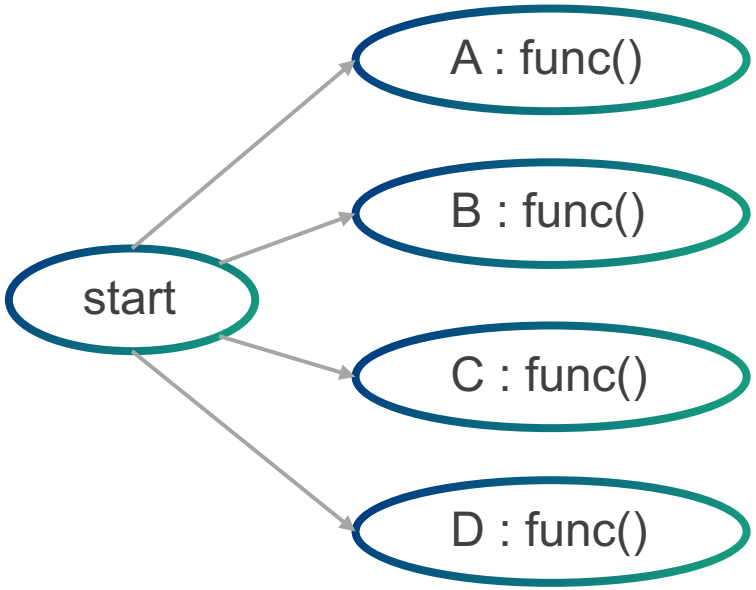
Supported Call Graph Algorithm

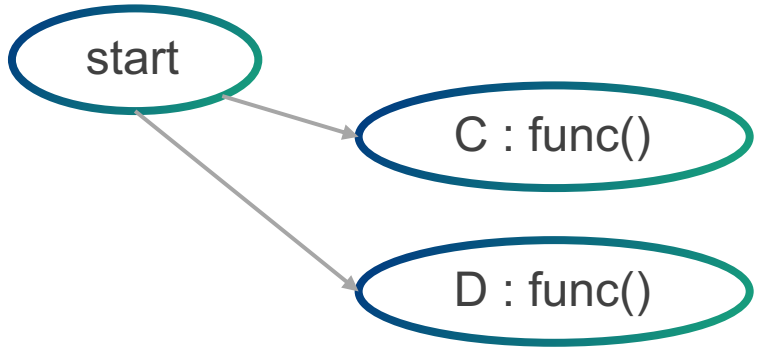
```
public void start (){
   C c = new C();
   A a = new D();
   a.func();
}
```

Class Hierarchy Algorithm (CHA)

Rapid Type Algorithm (RTA)



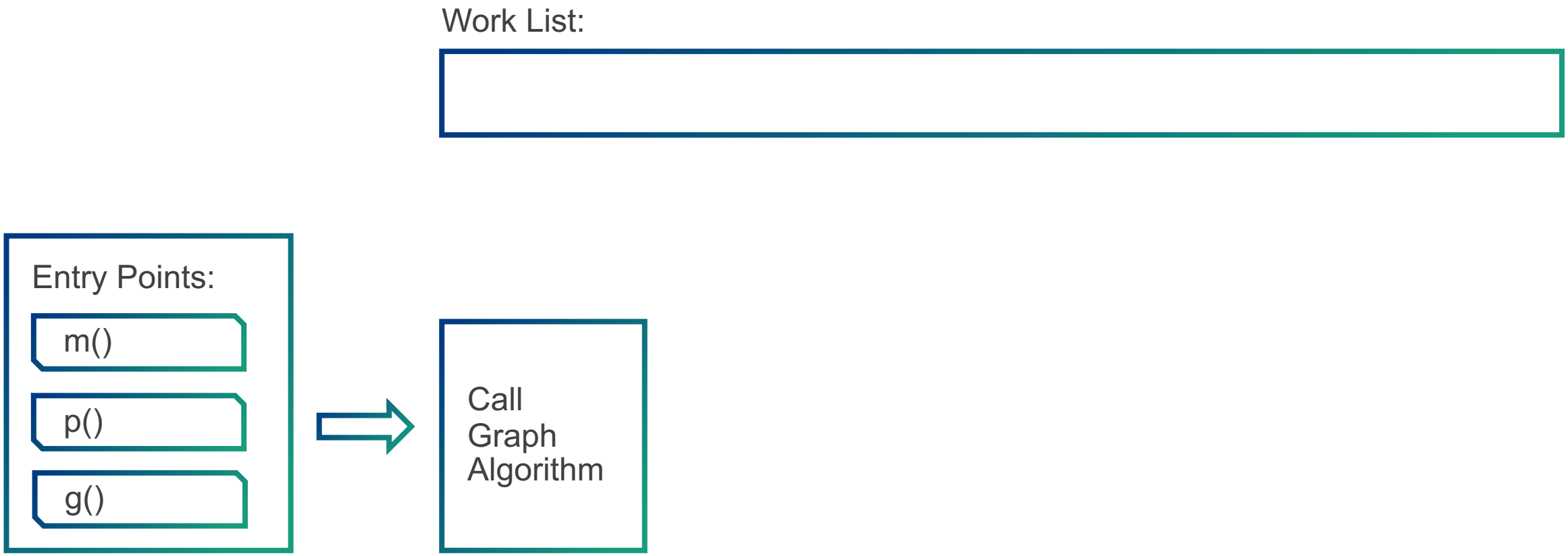




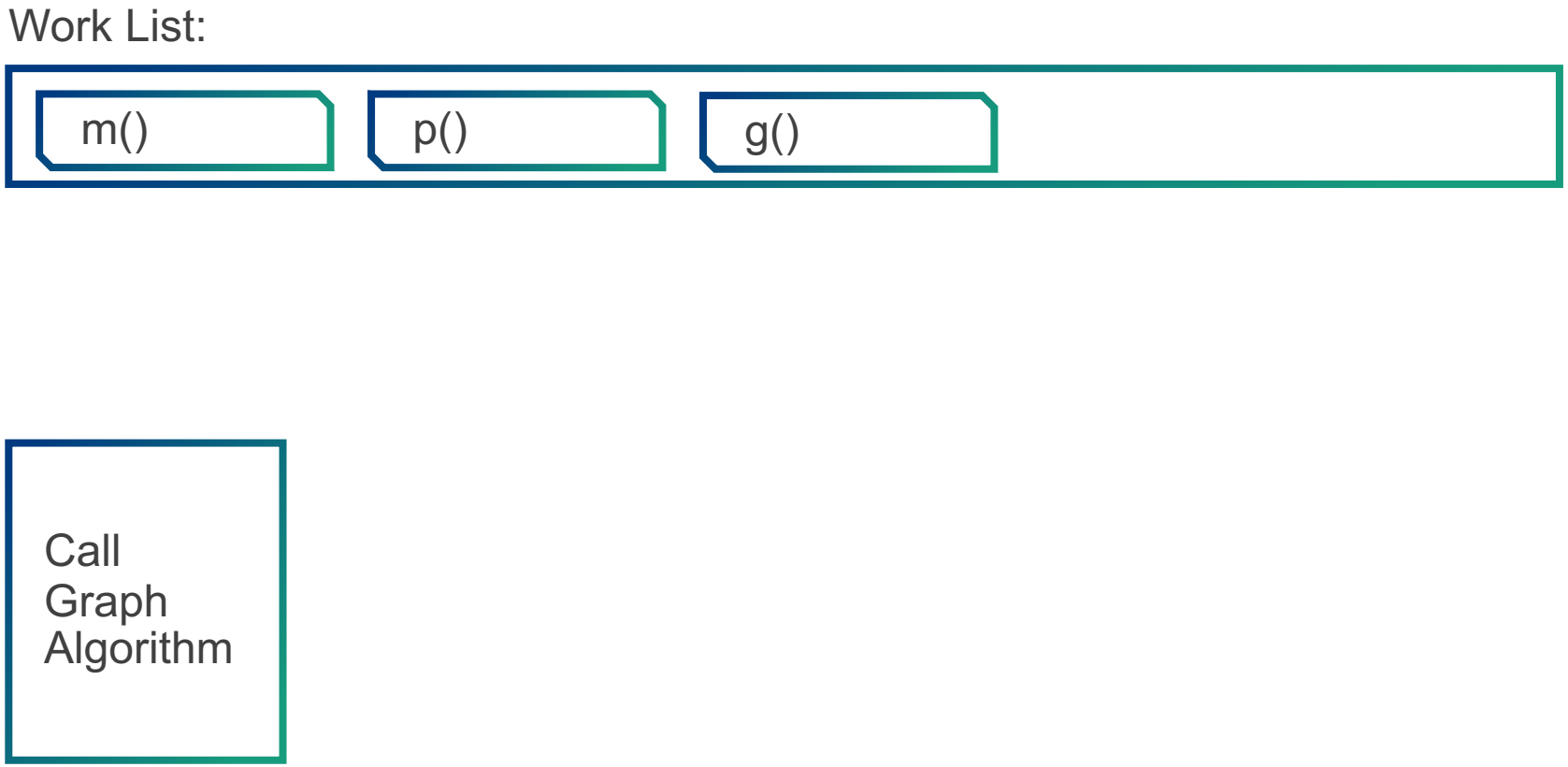
HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Fraunhofer IEM

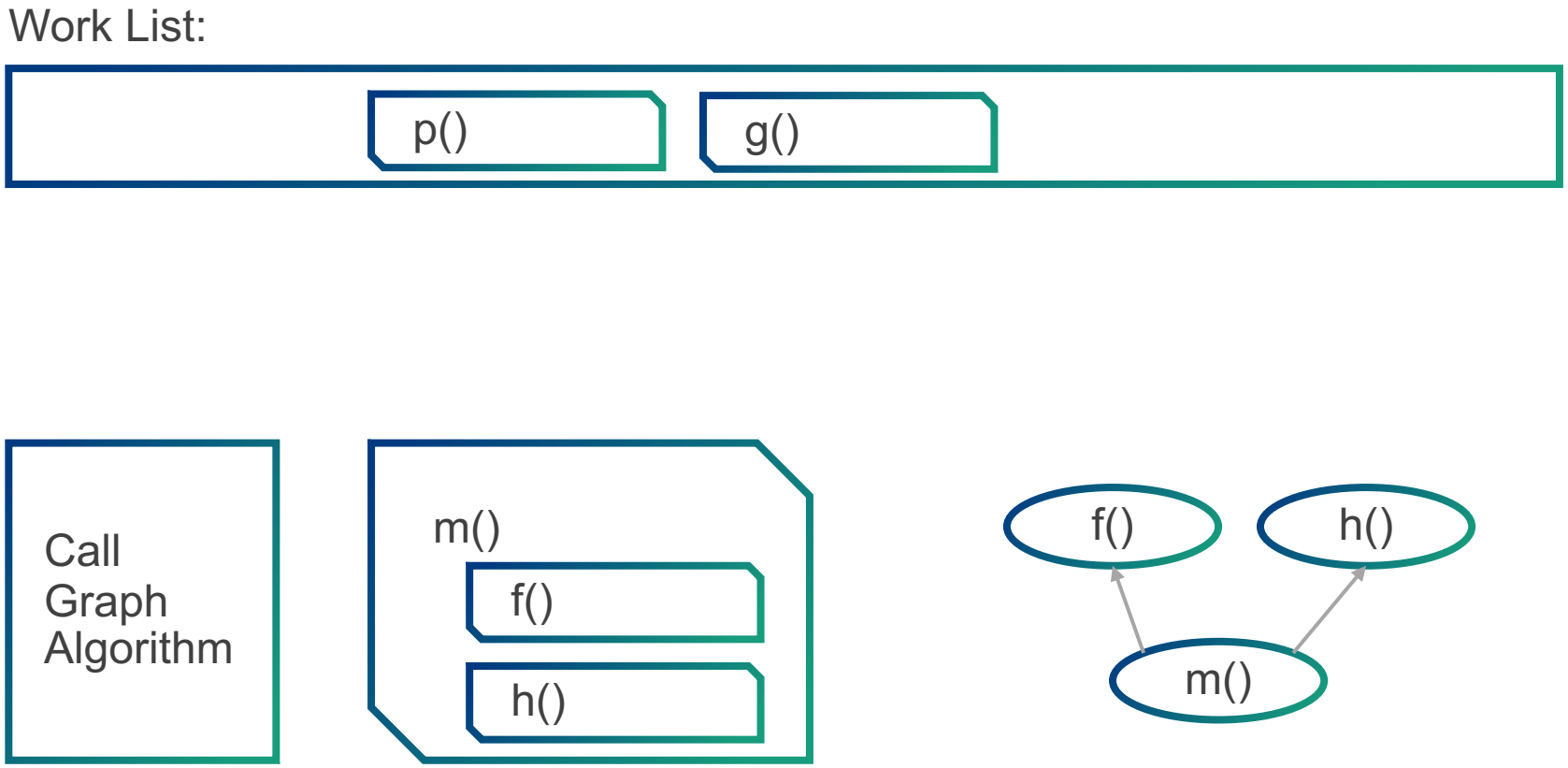
How to Create a Call Grap



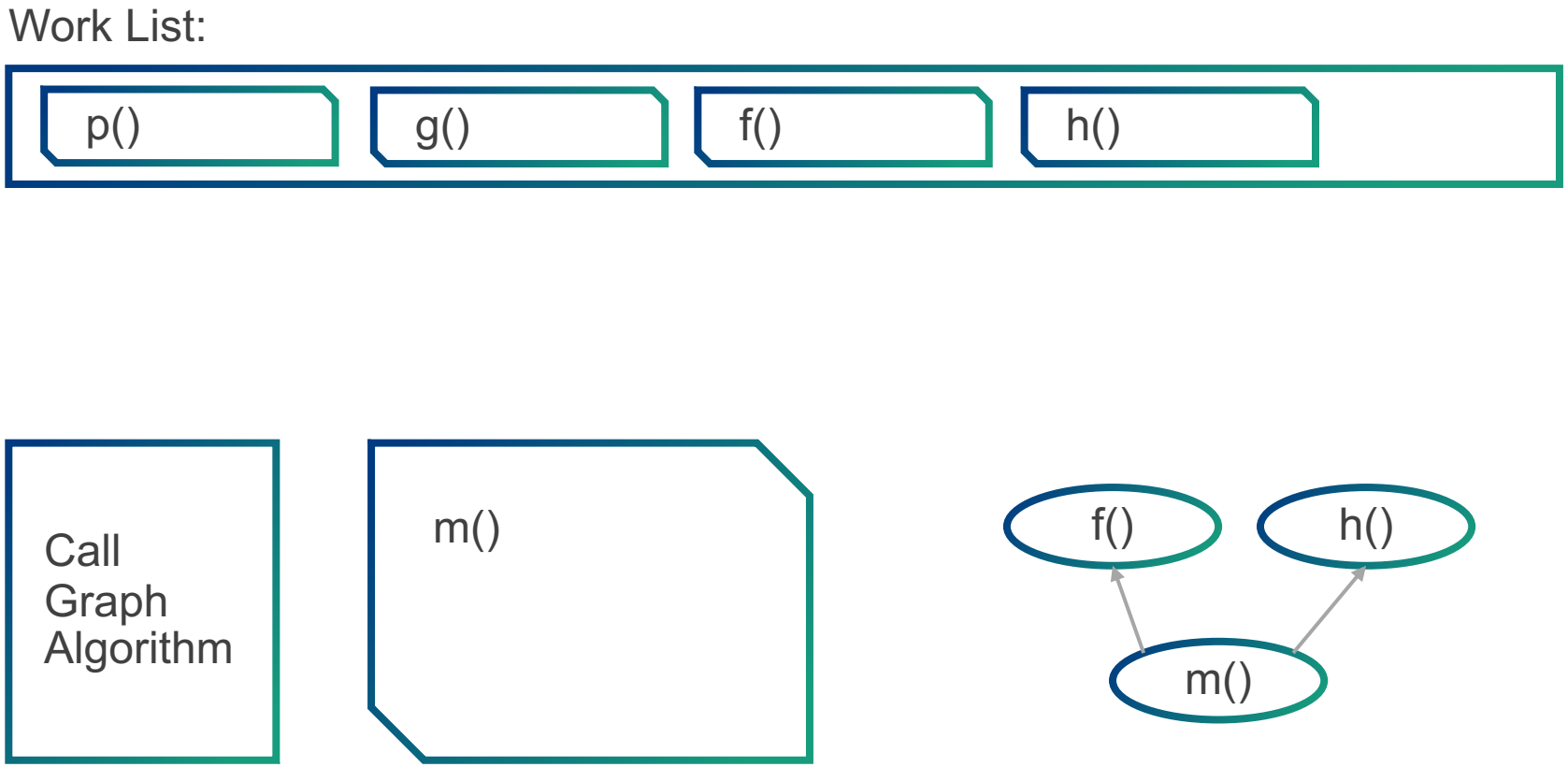
How to Create a Call Graph



How to Create a Call Graph



How to Create a Call Graph



Implicit Calls

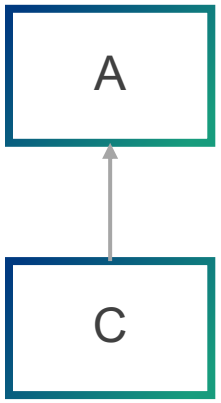
Static Initializer

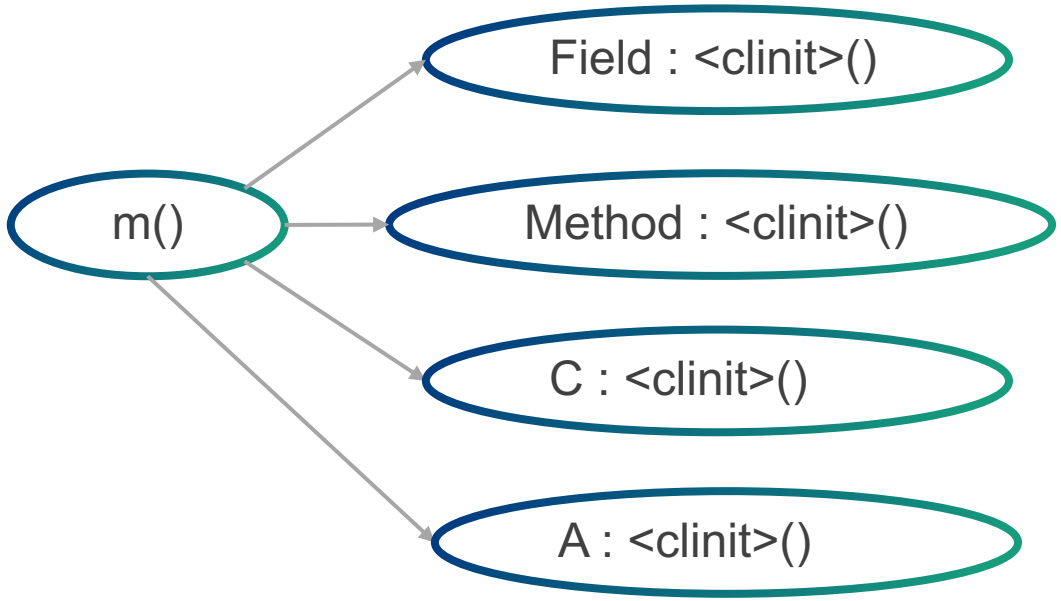
```
public class Field{
static int f = doMath();
}
```

```
public void m (){
int a = Field.f;
Method.m();
A a = new C();
}
```

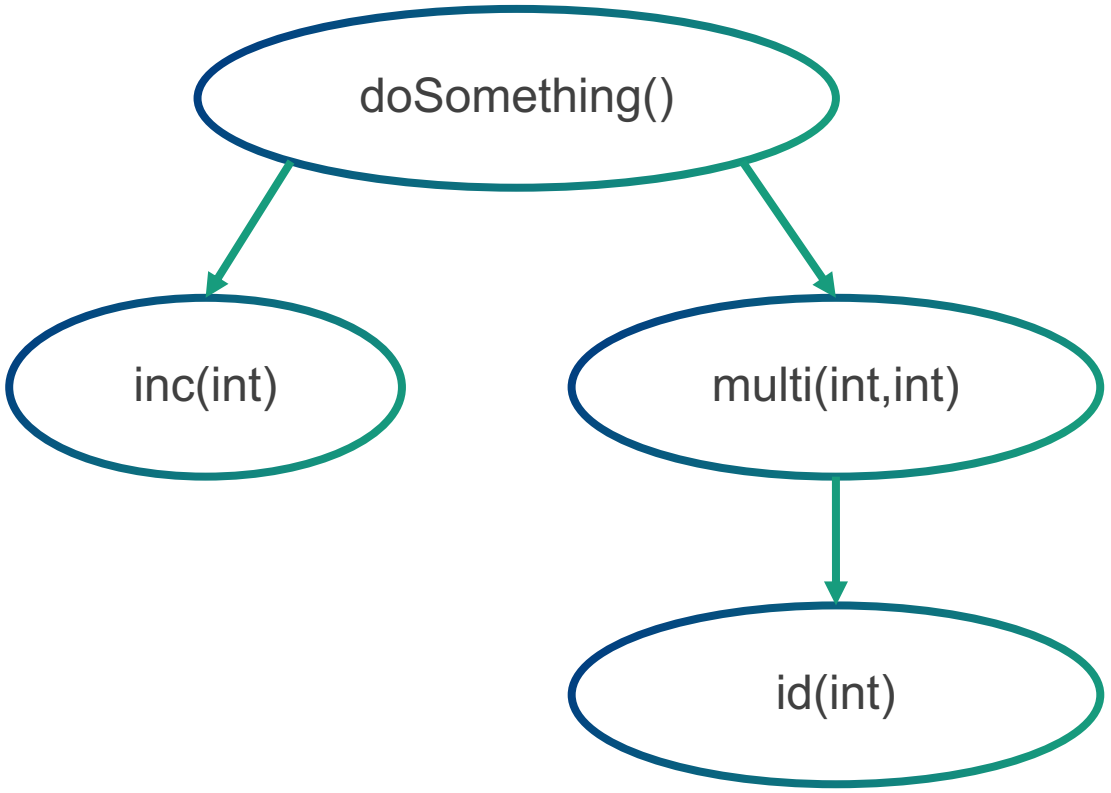
Bytecode:

```
public class Field{
  static int f;
  public void <clinit> (){
    f = doMath();
  }
}
```

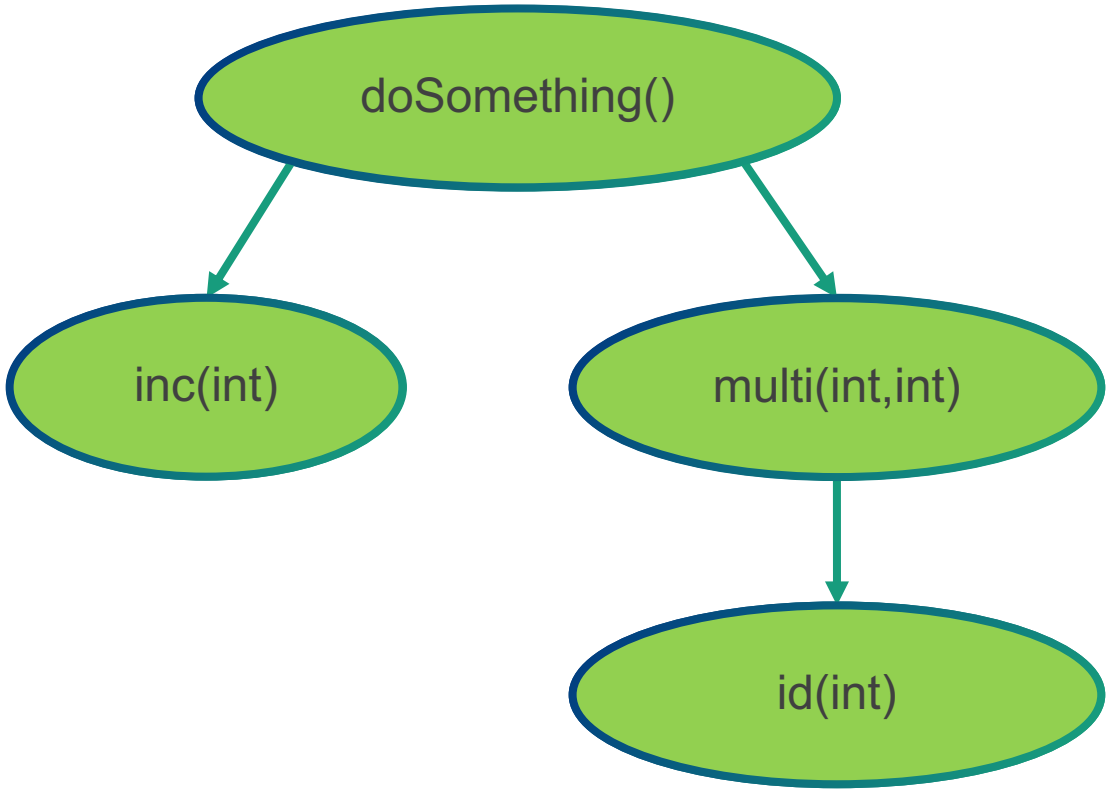




Call Graph API

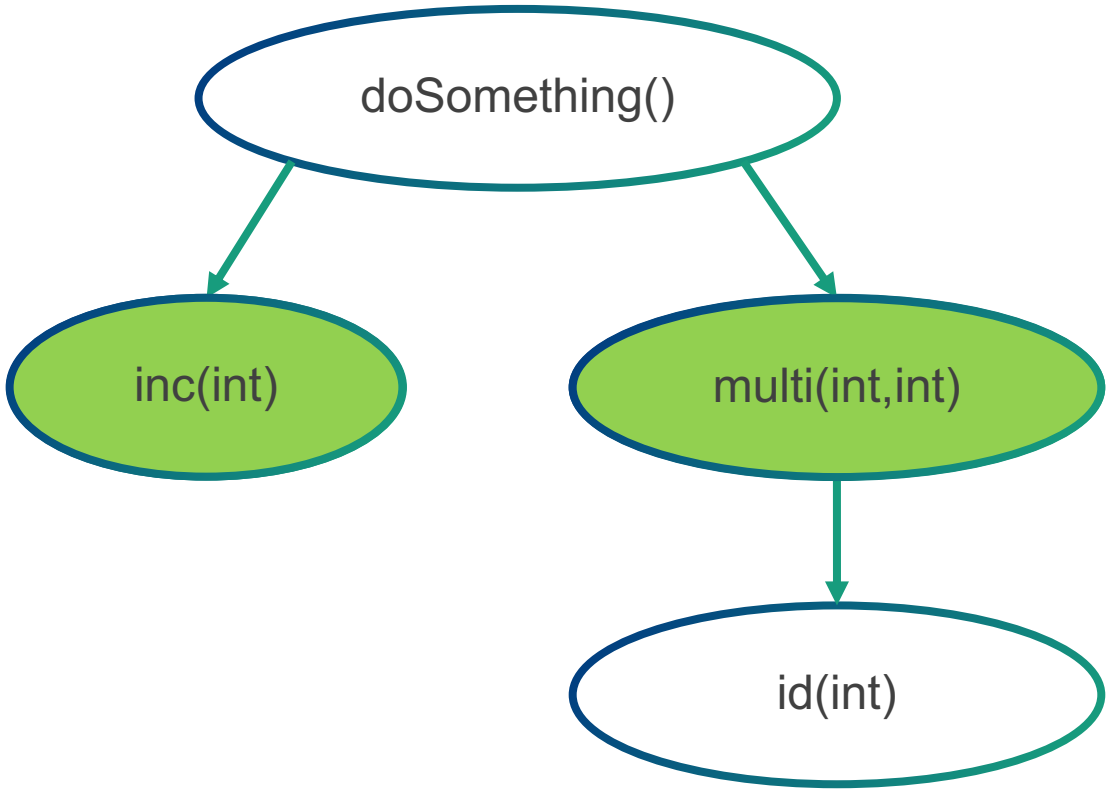


Call Graph API



getMethodSignatures() :

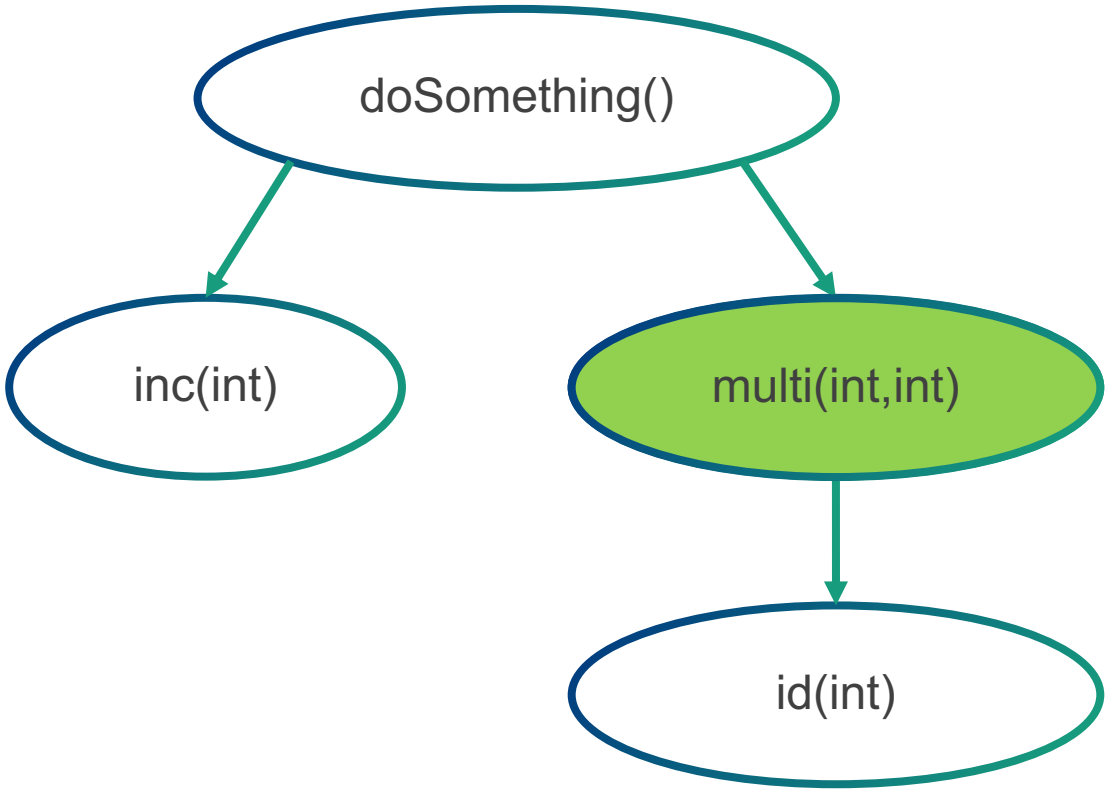
Call Graph API



getMethodSignatures() :

callsFrom(„doSomething()“)

Call Graph API

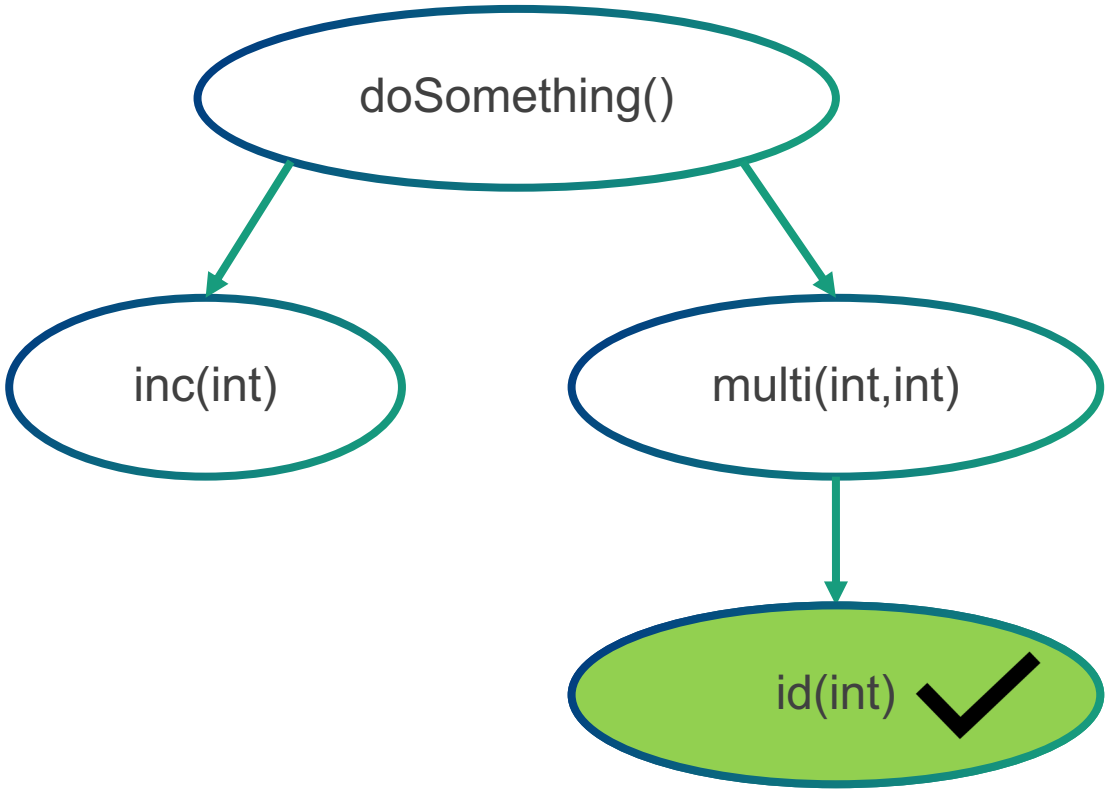


getMethodSignatures() :

callsFrom(„doSomething()“)

callsTo(„id(int)“)

Call Graph API



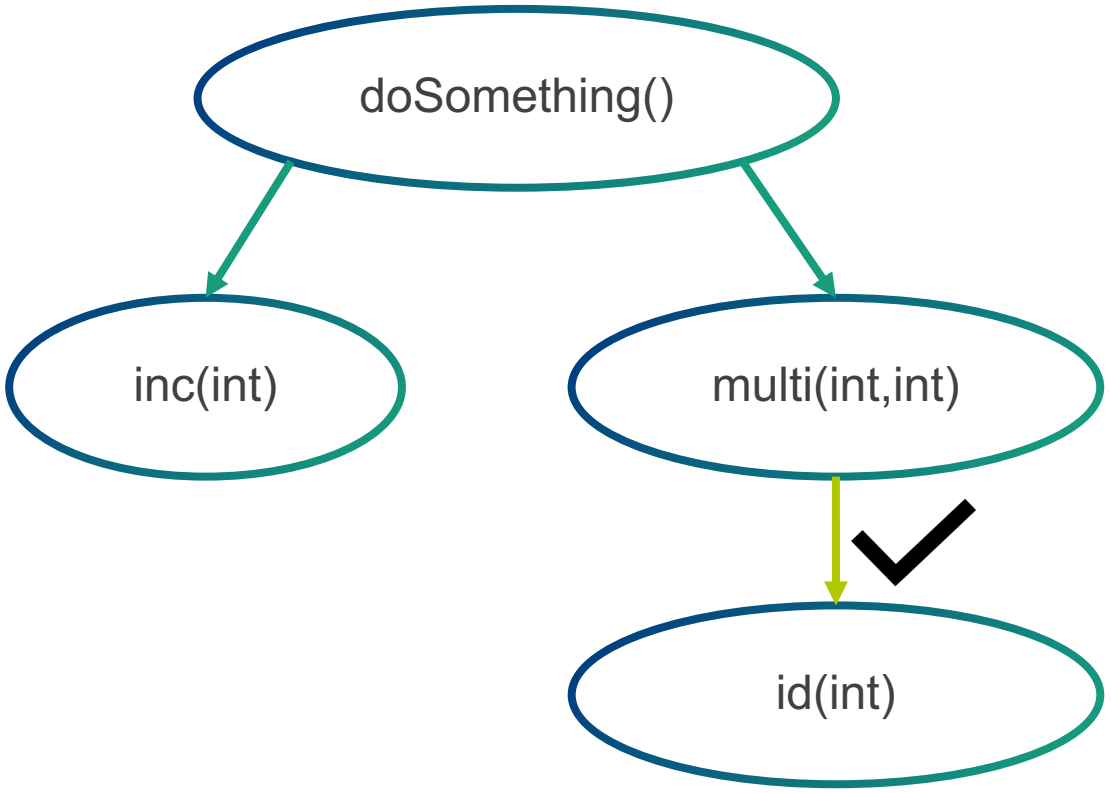
getMethodSignatures() :

callsFrom(„doSomething()“)

callsTo(„id(int)“)

containsMethod(„id(int)“)

Call Graph API



getMethodSignatures() :

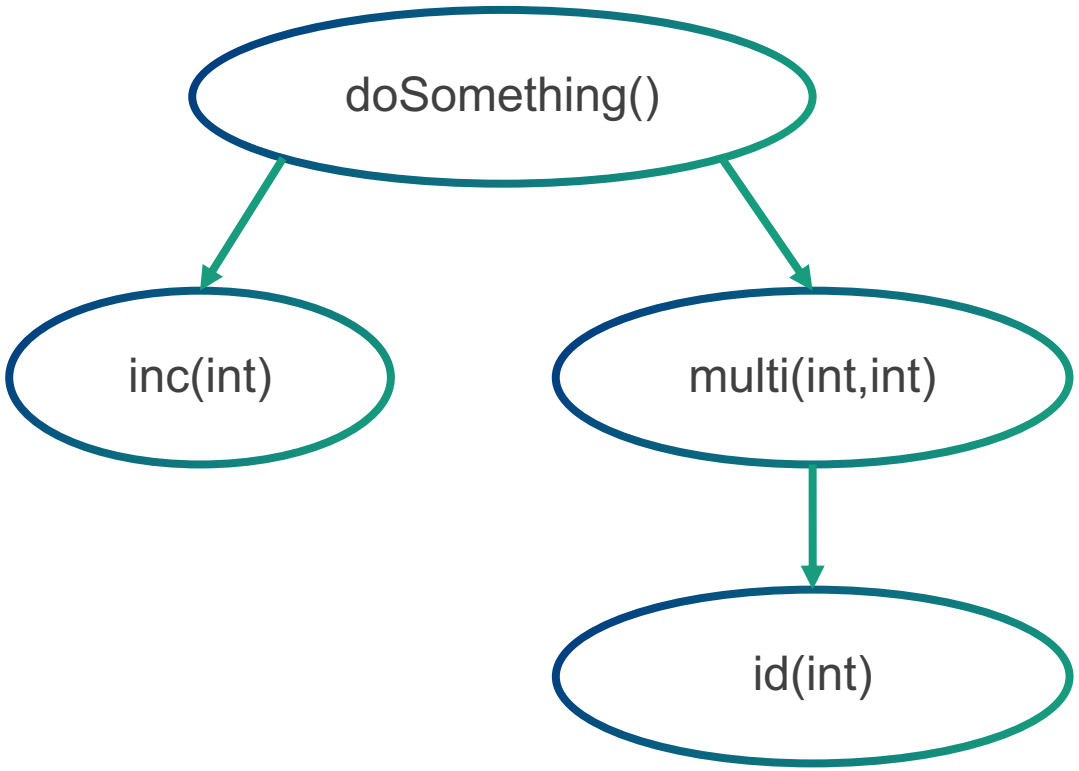
callsFrom(„doSomething()“)

callsTo(„id(int)“)

containsMethod(„id(int)“)

containsCall(„multi(int,int)“,„id(int)“)

Dot Exporter




```
digraph : {
        doSomething() -> inc(int)
        doSomething() -> multi(int,int)
        multi(int,int) -> id(int)
}
```

EXERCISE 2: TAINT ANALYSIS WITH SOOTUP AND HEROS

Taint Analysis

- A taint is a sensitive information that we want to track throughout the application run.
  - Hardcoded credentials that might leak
  - Untrusted input that might reach critical methods

- Terminology
  - Source:
    - x = „SECRET“
  - Sink:
    - sink(x)

```
void entryPoint() {
    String a = "SECRET";

    String b = a;

    String c = id(b);

    SinkClass sc = new SinkClass();

    sc.sink(c);
}
```

Taint Analysis with SootUp and Heros

- Heros is an IFDS solver
- IFDS framework let‘s us define data-flow analysis problems:
  - Interprocedural
    - Across method boundaries
  - Flow- and context-sensitive
    - Maintaining order statements
    - Precisely tracking effects in all method contexts
  - Flow Functions to define effects of each statement

Taint Analysis with SootUp and Heros

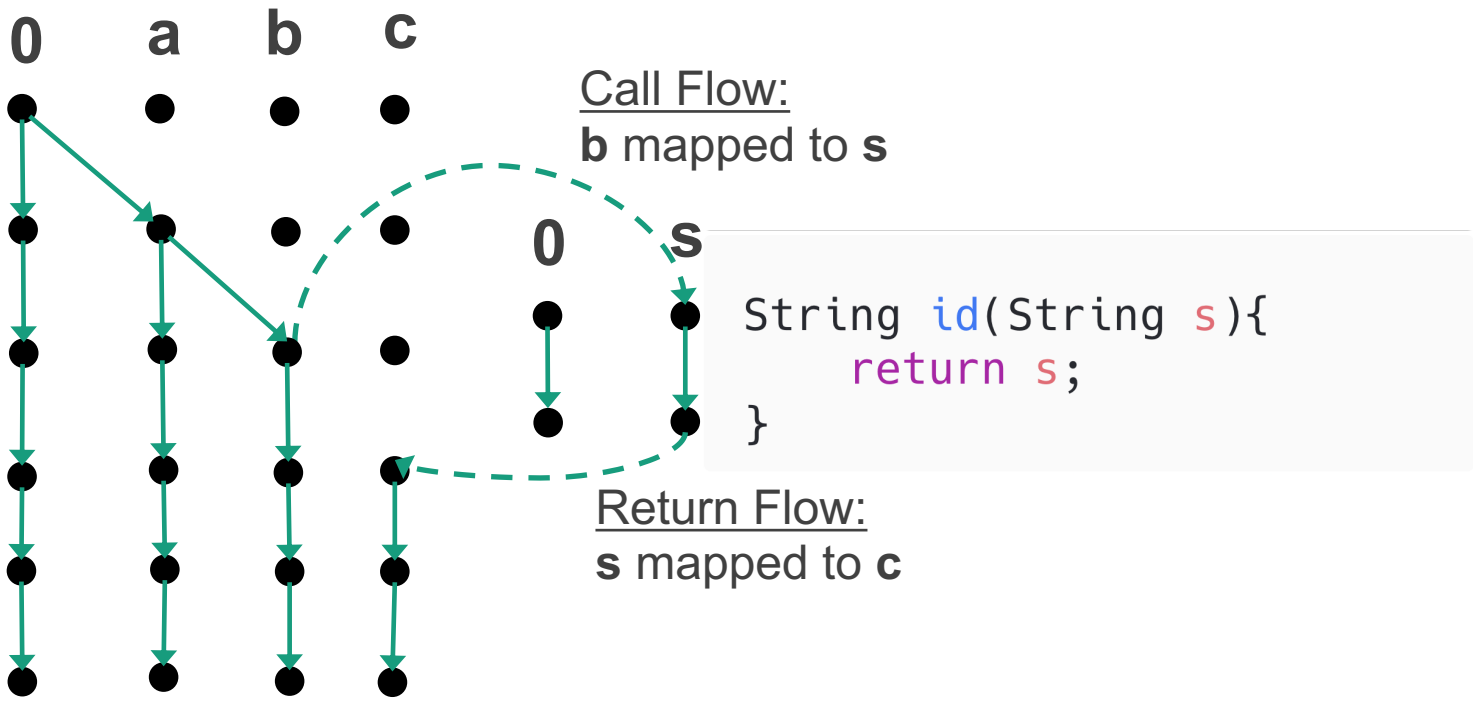
Flow Functions

```
void entryPoint() {
    String a = "SECRET";

    String b = a;

    String c = id(b);

    SinkClass sc = new SinkClass();

    sc.sink(c);
}
```

Normal Flow:
Generate **a**

Propagate **a** to **b**

Call-to-Return Flow



Exercise 2: Implement a Taint Analysis Problem

- Task: Implement the Flow Functions in TaintAnalysisProblem.java
  1. **getNormalFlow**()
     1. Generating a Taint: only for this specific assignment, x = "SECRET"
     2. Propagating (Transferring) a Taint between different variables
  2. **getCallFlow**()
     1. Mapping call arguments to local parameters, foo(x)
  3. **getReturnFlow**()
     1. Mapping returned variable to assigned variable, y = foo(x)

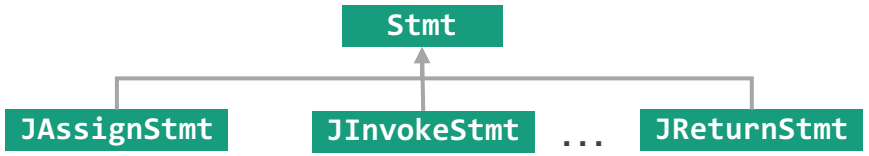
Steps to access the exercise:

1. git clone https://github.com/soot-oss/SootUp-Tutorial.git
2. git checkout 3_Exercise-2

OR

Download from: https://sootup-tutorial.surge.sh

Exercise 2: Helpful APIs



StringConstant	
String	**getValue**()

JAssignStmt	
Value	**getLeftOp**()
Value	**getRightOp**()

JReturnStmt	
Value	**getOp**()

Gen<D>	
	Gen(D genValue, D zeroValue)

Transfer<D>	
	Transfer(D toValue, D fromValue)

Identity<D>	
Identity<D>	**v**()

KillAll<D>	
KillAll<D>	**v**()

For more help check:

- Documentation: soot-oss.github.io/SootUp
- Javadoc: soot-oss.github.io/SootUp/apidocs
- Heros: javadoc.io/doc/de.upb.cs.swt/heros/latest/index.html

CLOSING

Future Work

In Progress

- Android frontend

FlowDroid

- Scalable demand-driven pointer analysis

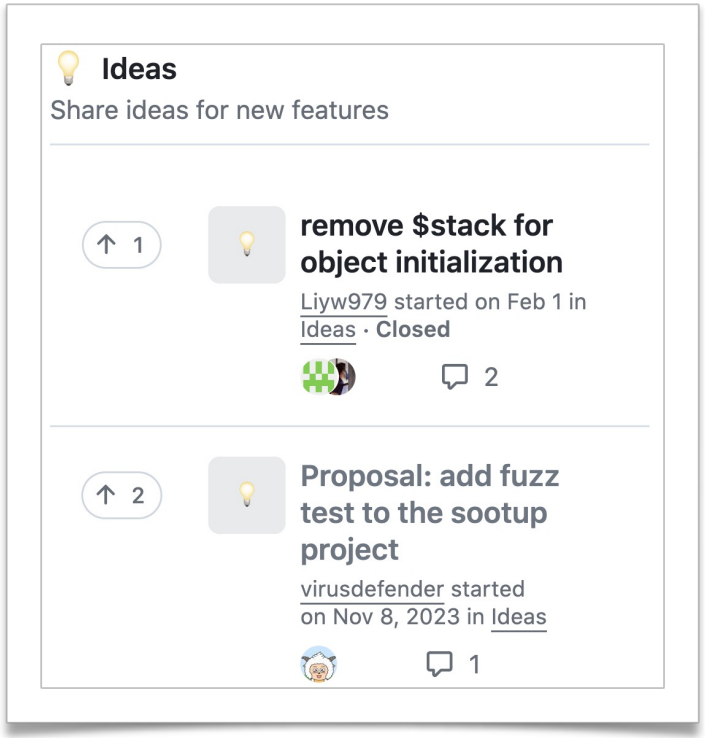
SparseBoomerang

In Near Future

- Frontends for other languages: e.g. Python, Javascript
- Partial code analysis

We are always open to suggestions!

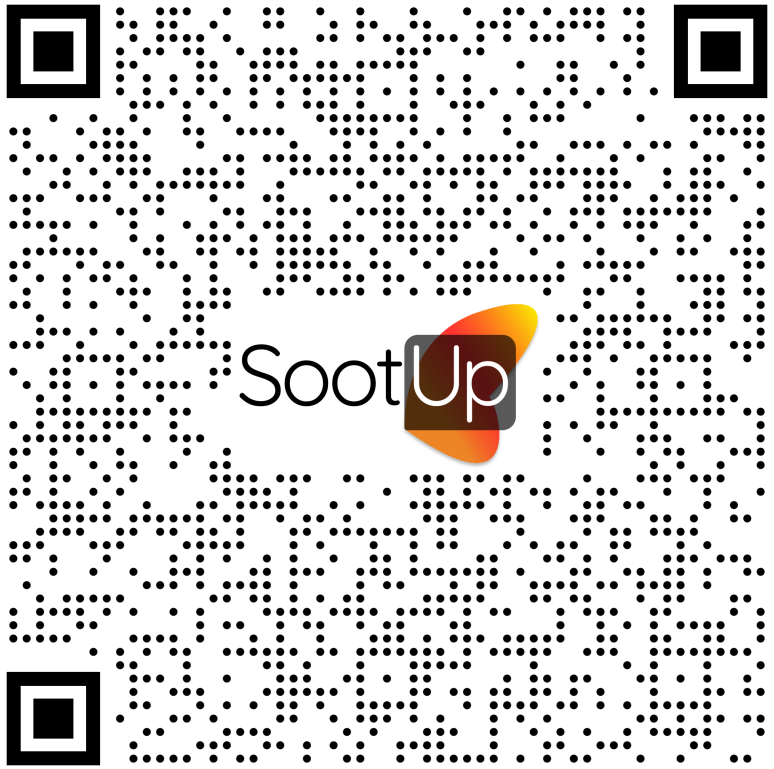
Feel free to propose your ideas in discussions:



HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

Fraunhofer IEM

SUS Questionnaire



Deutsche Forschungsgemeinschaft
German Research Foundation

Federal Ministry of Education and Research

HEINZ NIXDORF INSTITUT
UNIVERSITÄT PADERBORN

aws

We are looking forward to see you be part of the SootUp community!

SootUp

https://github.com/soot-oss/SootUp