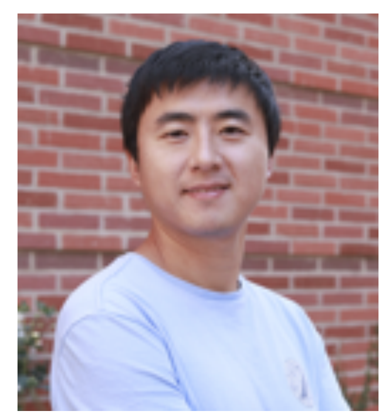
JSHRINK: IN-DEPTH INVESTIGATION INTO DEBLOATING MODERN JAVA APPLICATIONS

Paper at: https://www.bobbybruce.net/assets/pdfs/publications/bruce-2020-jshrink.pdf

---





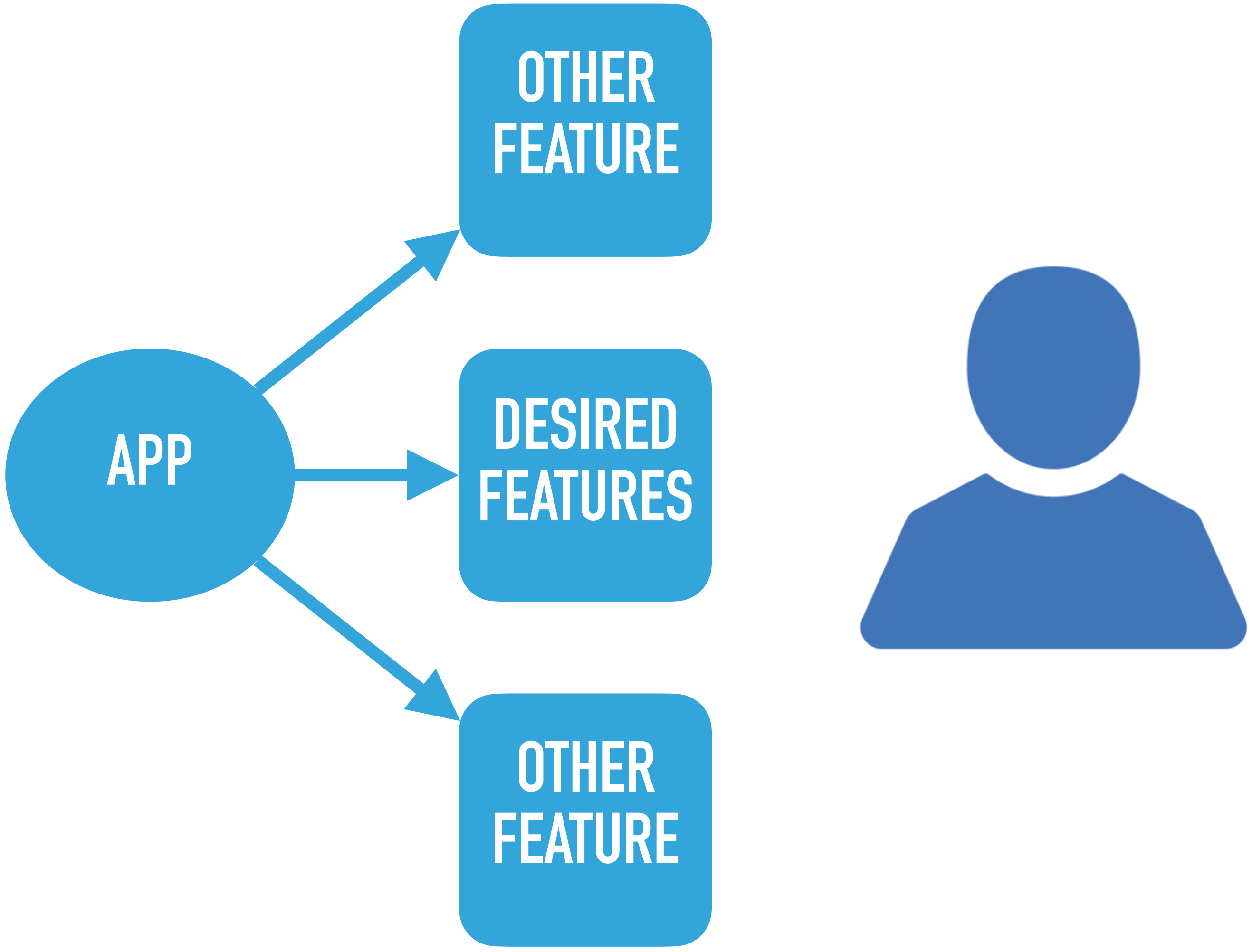

BOBBY R. BRUCE, TIANYI ZHANG, JASPREET ARORA, GUOQING HARRY XU, AND MIRYUNG KIM

PRESENTED BY BOBBY R. BRUCE

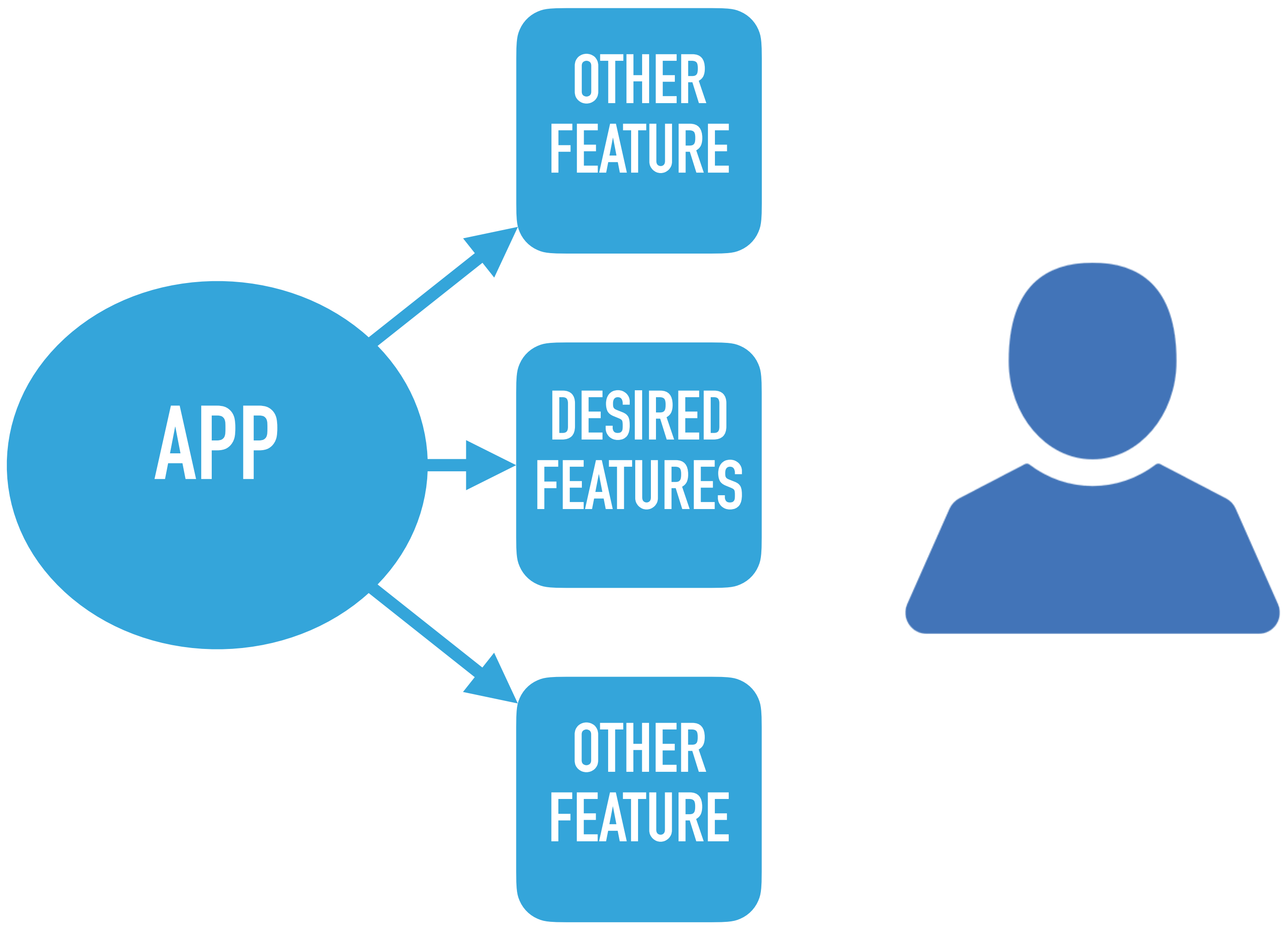
MOTIVATION: MODERN SOFTWARE IS BLOATED

MOTIVATION: MODERN SOFTWARE IS BLOATED

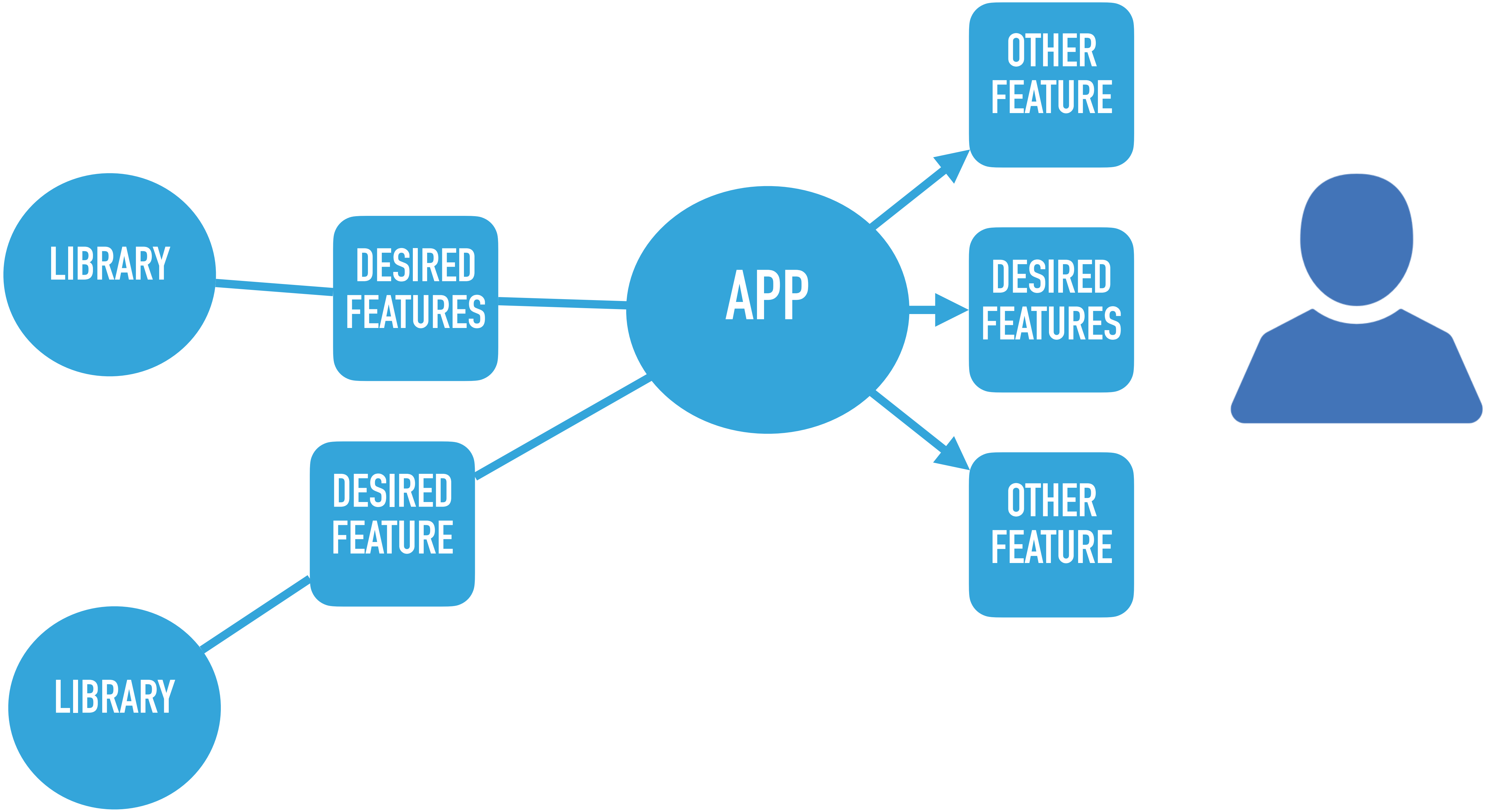
MOTIVATION: MODERN SOFTWARE IS BLOATED



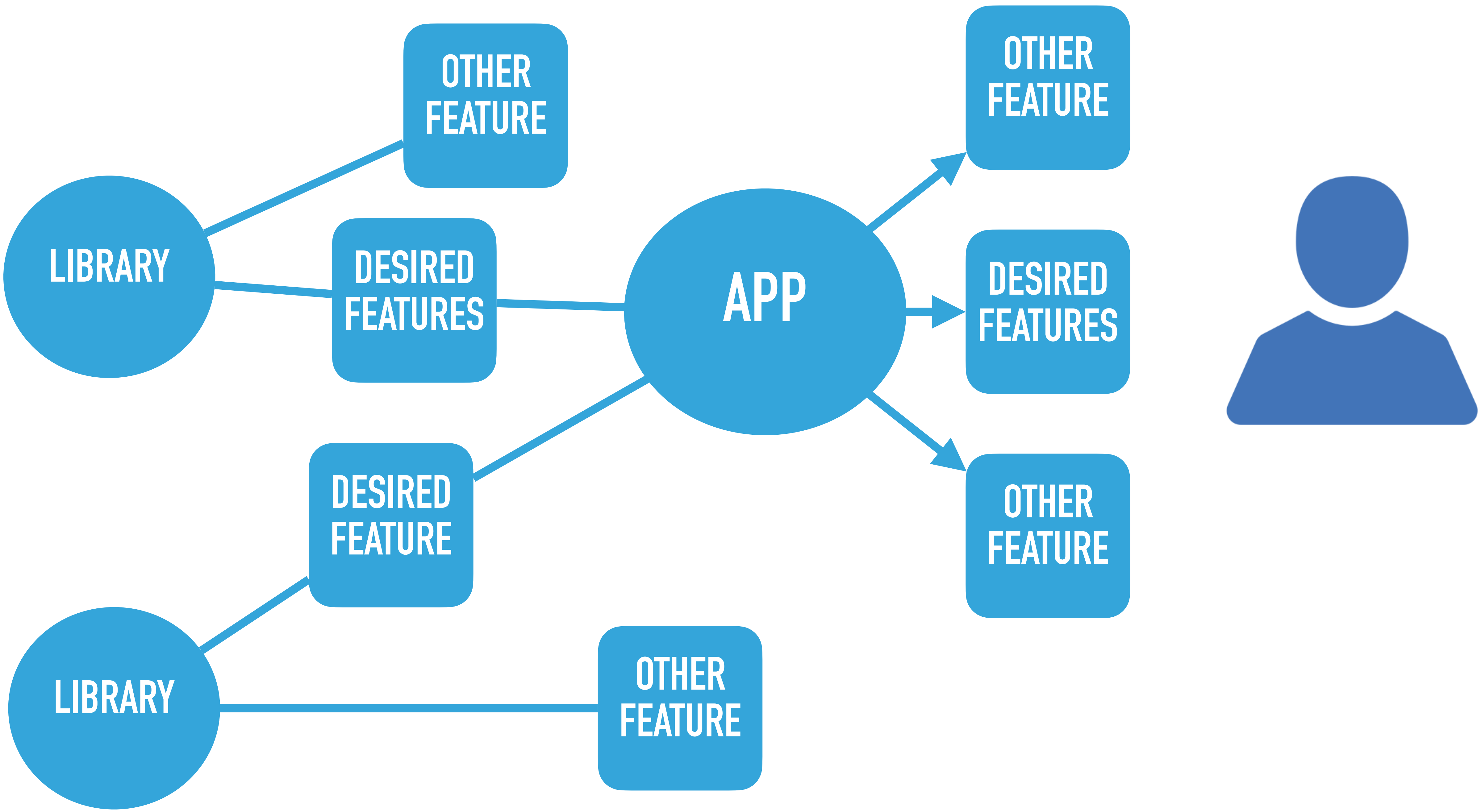
MOTIVATION: MODERN SOFTWARE IS BLOATED



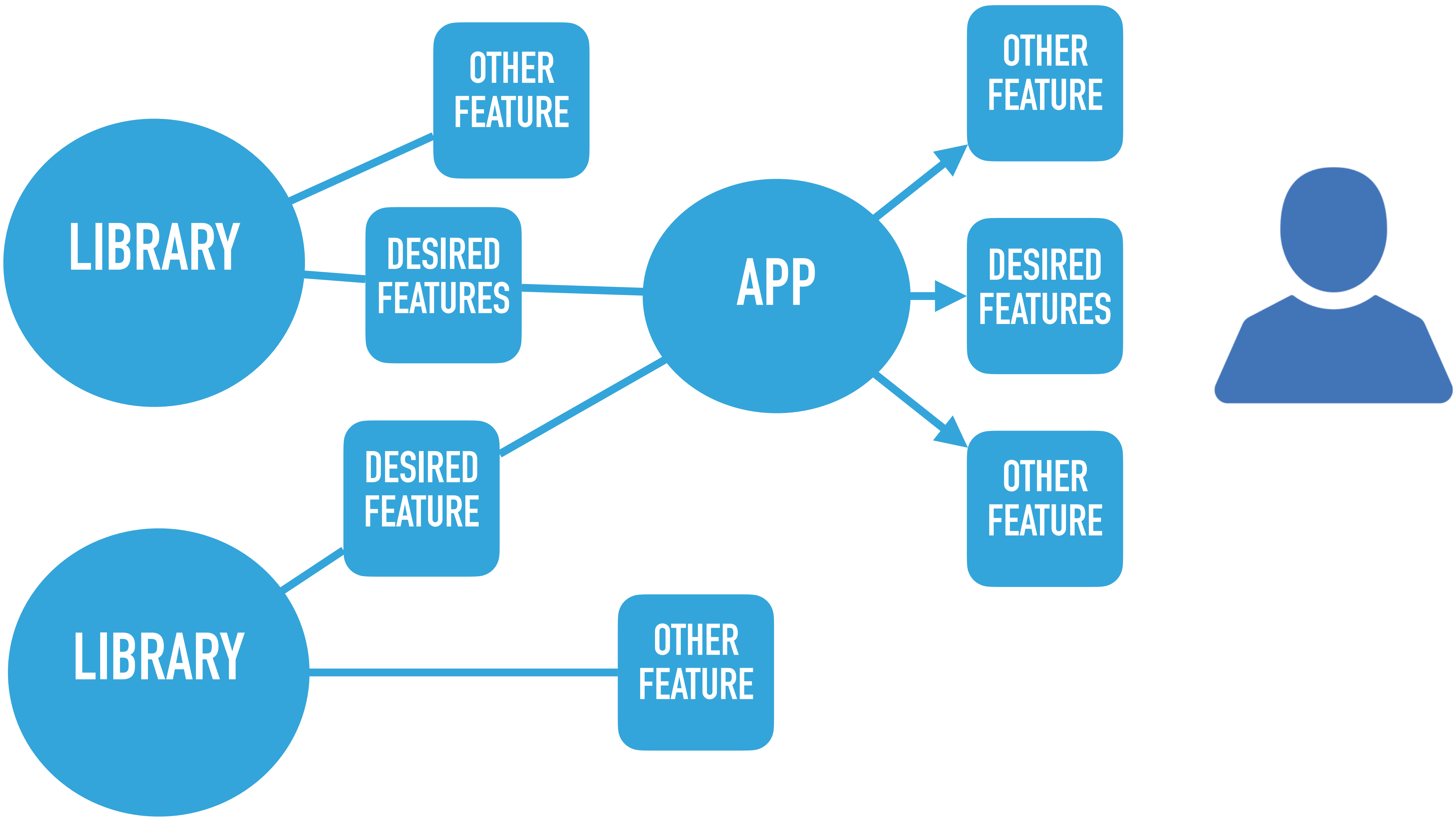
MOTIVATION: MODERN SOFTWARE IS BLOATED



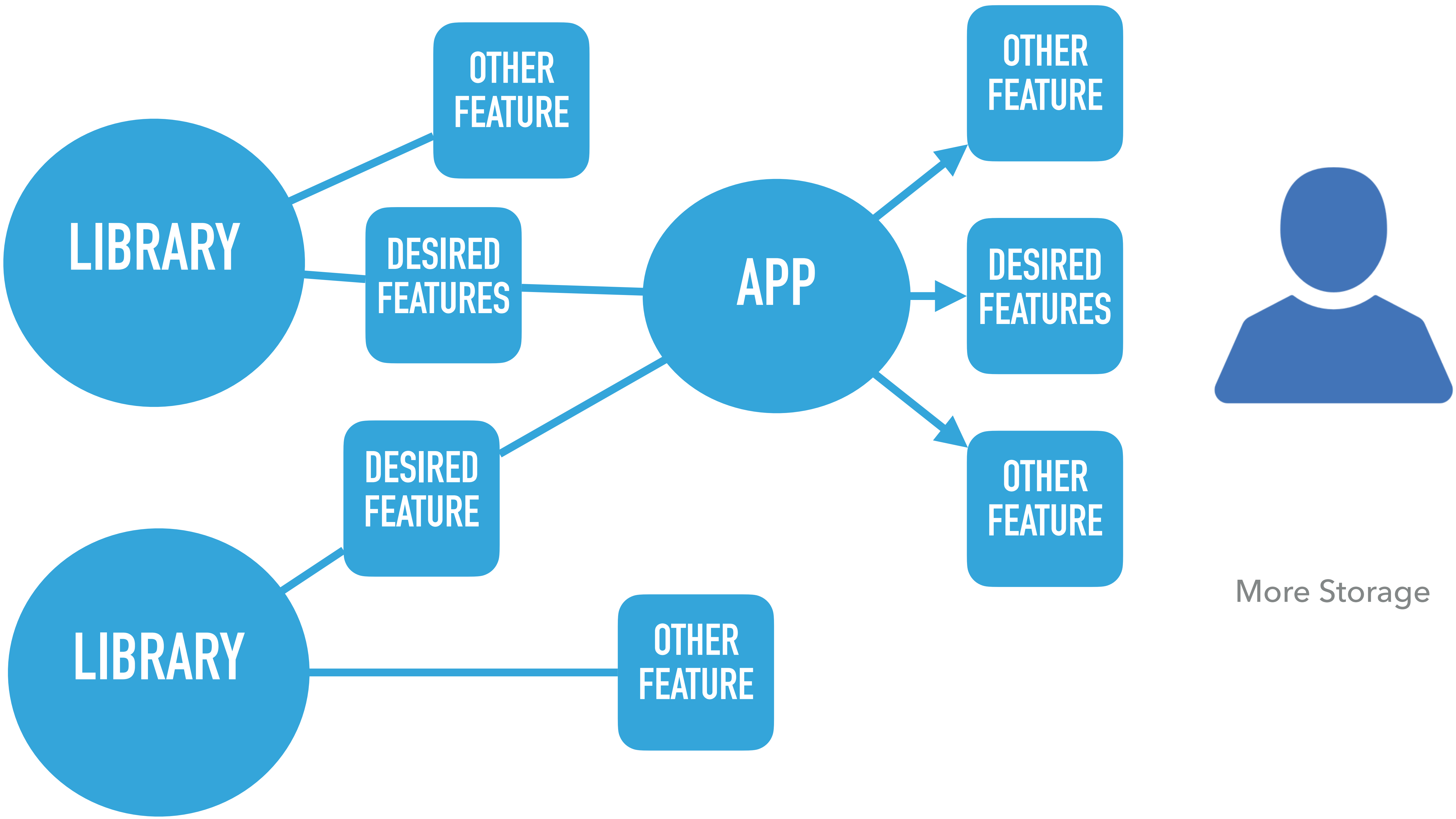
MOTIVATION: MODERN SOFTWARE IS BLOATED



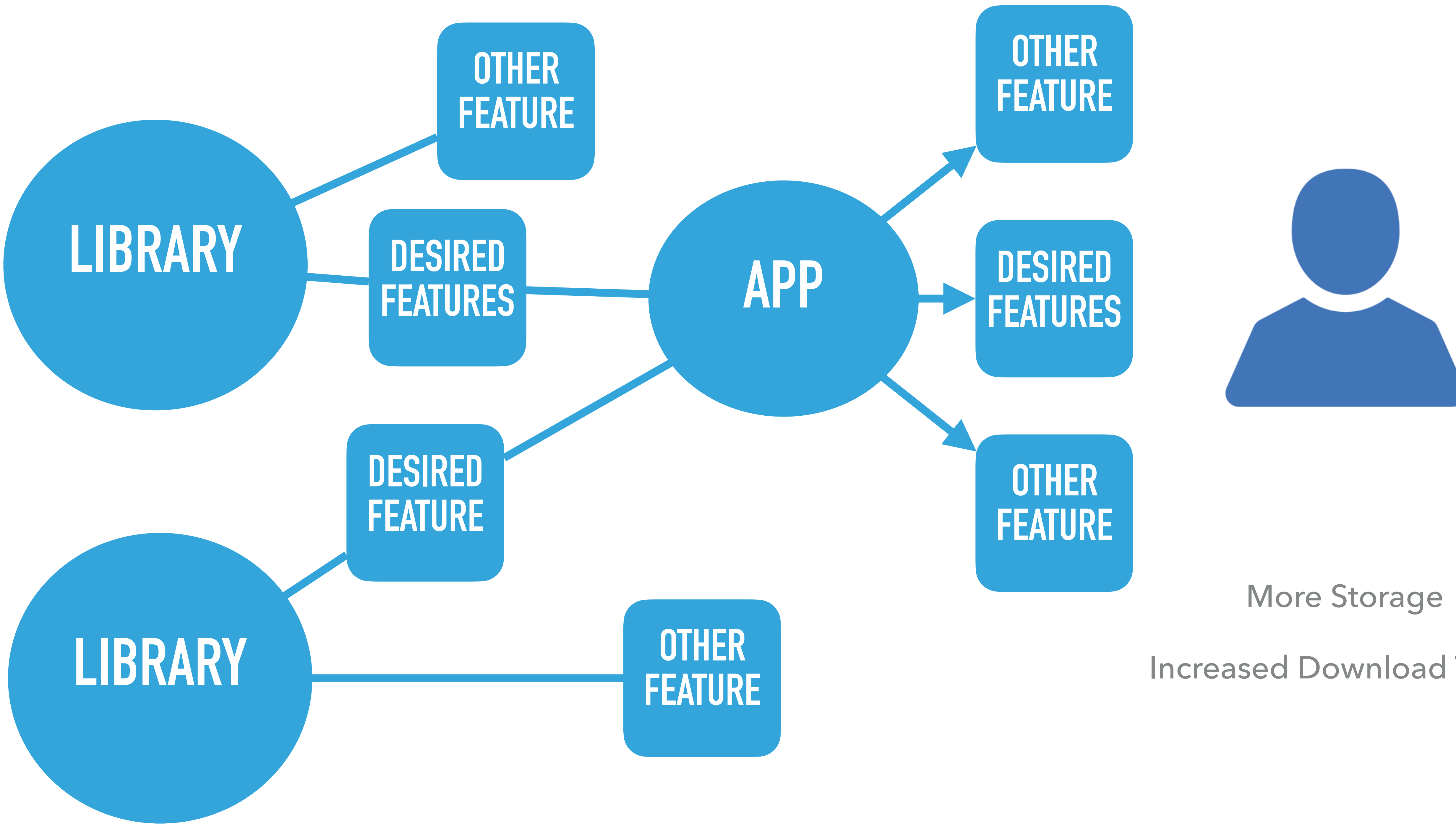
MOTIVATION: MODERN SOFTWARE IS BLOATED



MOTIVATION: MODERN SOFTWARE IS BLOATED



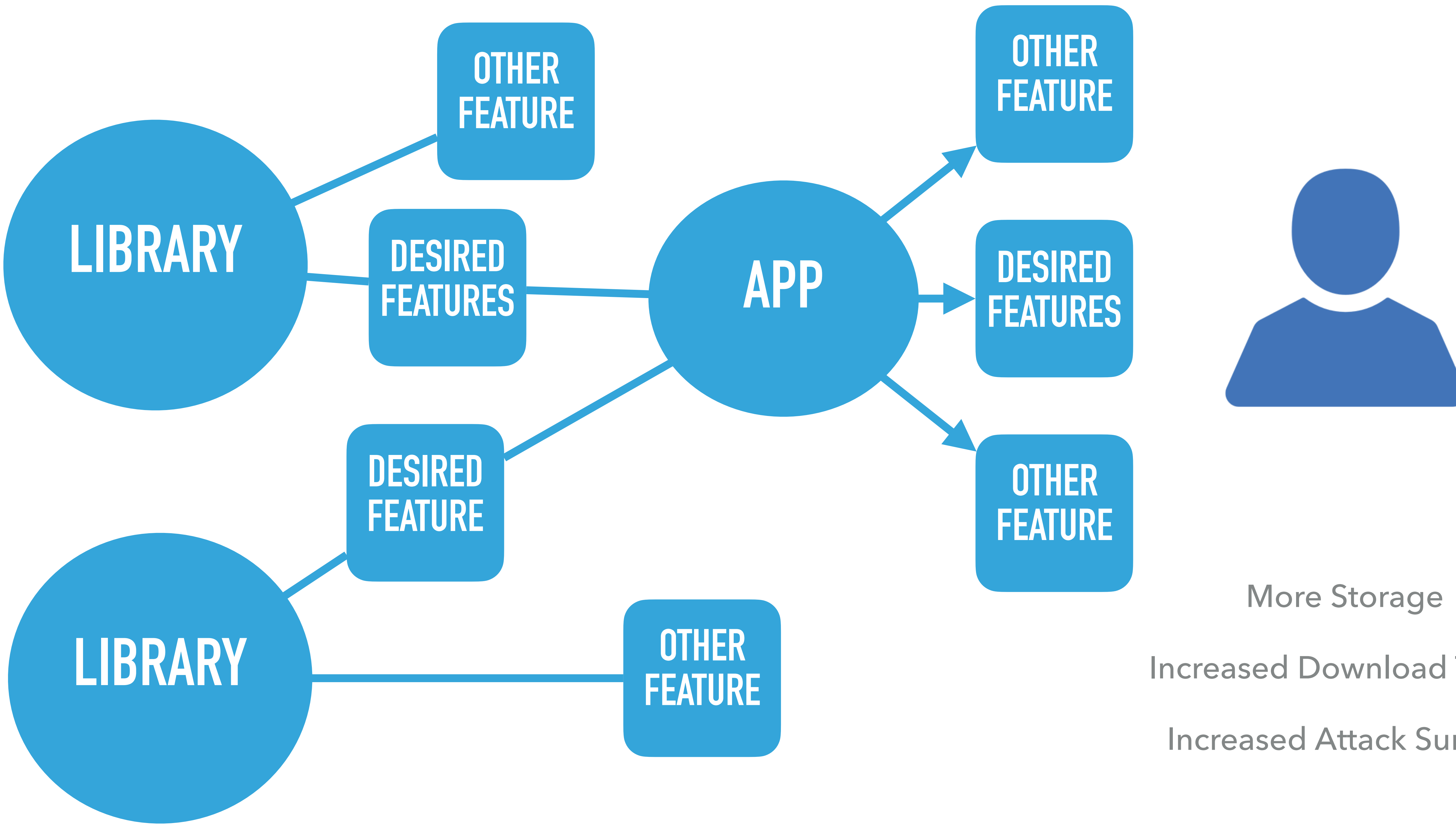
MOTIVATION: MODERN SOFTWARE IS BLOATED



More Storage

Increased Download Times

MOTIVATION: MODERN SOFTWARE IS BLOATED

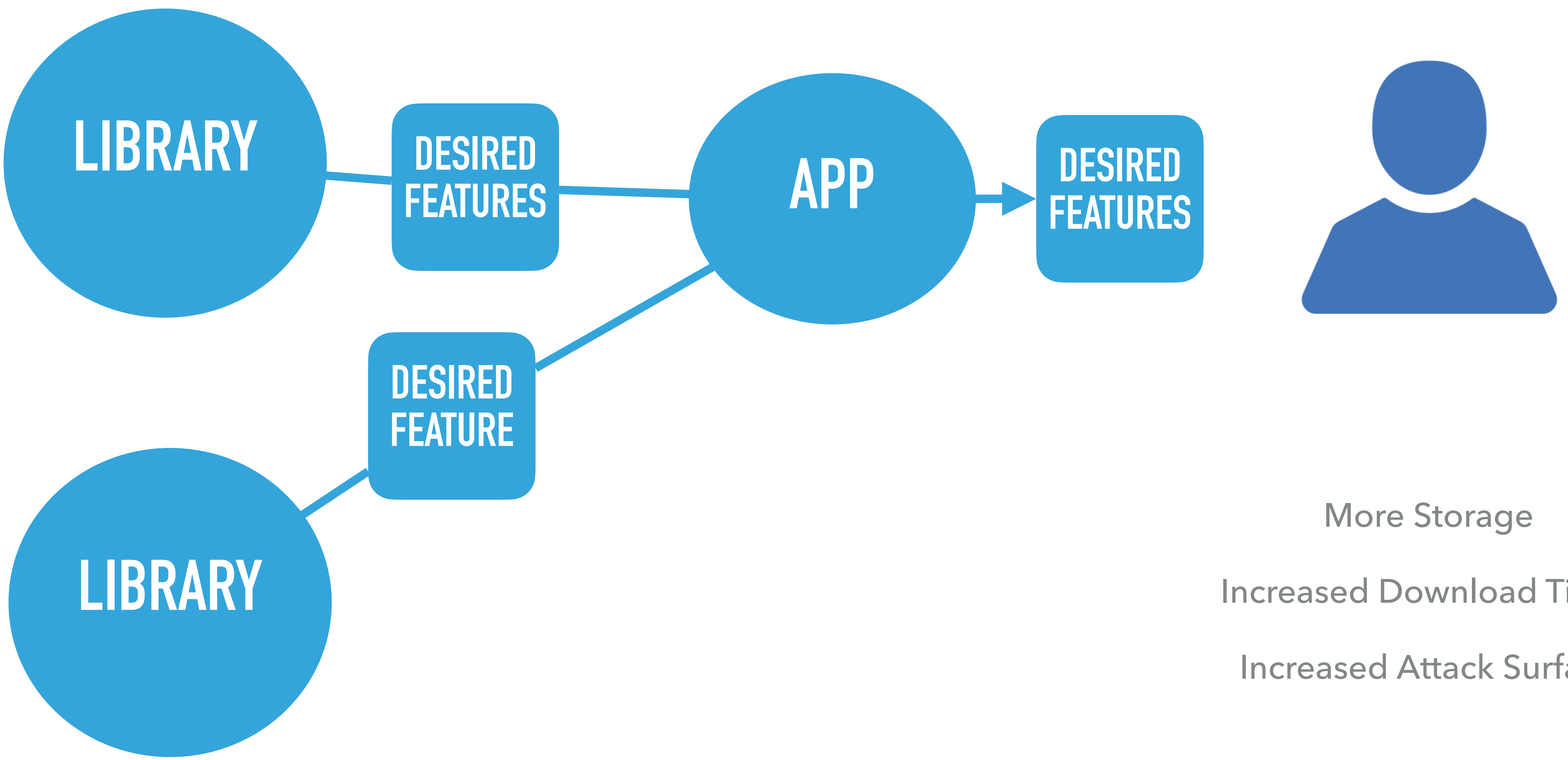


More Storage

Increased Download Times

Increased Attack Surface

MOTIVATION: MODERN SOFTWARE IS BLOATED

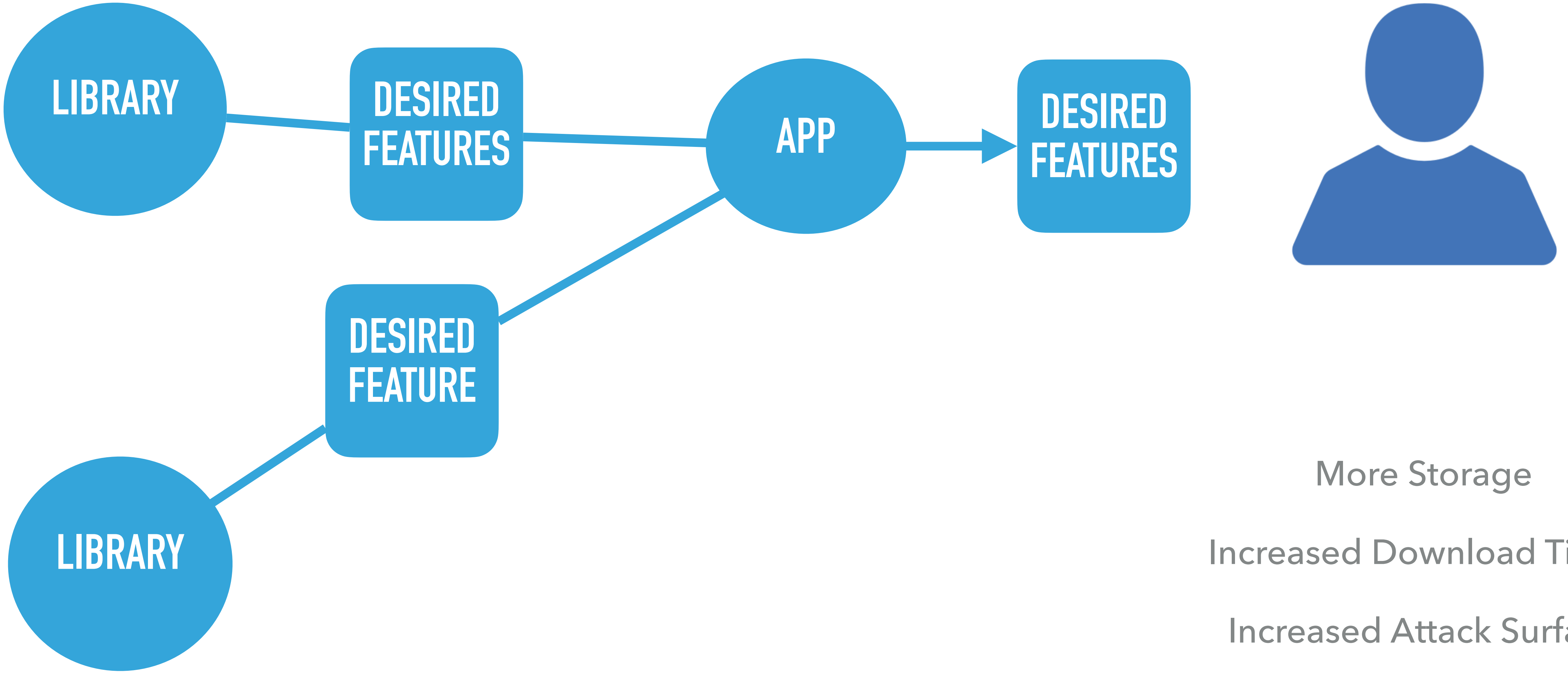


More Storage

Increased Download Times

Increased Attack Surface

MOTIVATION: MODERN SOFTWARE IS BLOATED



More Storage

Increased Download Times

Increased Attack Surface

PREVIOUS WORK

PREVIOUS WORK

Lots of good work on debloating C/C++; but Java has been largely ignored.

PREVIOUS WORK

Lots of good work on debloating C/C++; but Java has been largely ignored.

Practical Experience with an Application Extractor for Java

Frank Tip   Chris Laffra   Peter F. Sweeney
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
{tip,laffra,pfs}@watson.ibm.com

David Streeter
IBM Toronto Laboratory
1150 Eglinton Ave. East
Toronto, Ontario, Canada
daves@ca.ibm.com

Abstract

Java programs are routinely transmitted over low-bandwidth network connections as compressed class file archives (i.e., zip files and jar files). Since archive size is directly proportional to download time, it is desirable for applications to be as small as possible. This paper is concerned with the use of program transformations such as removal of dead methods and fields, inlining of method calls, and simplification of the class hierarchy for reducing application size. Such "extraction" techniques are generally believed to be especially useful for applications that use class libraries, since typically only a small fraction of a library's functionality is used. By "pruning away" unused library functionality, application size can be reduced dramatically. We implemented a number of application extraction techniques in Jax, an application extractor for Java, and evaluate their effectiveness on a set of realistic benchmarks ranging from 27 to 2,332 classes (with archives ranging from 56,796 to 3,810,120 bytes). We report archive size reductions ranging from 13.4% to 90.2% (48.7% on average).

1 Introduction

Java[1] [10] programs are routinely transmitted over the internet as compressed class file archives (i.e., zip files and jar files). A typical example of this situation consists of downloading a web page that contains one or more embedded Java applets. The downloading of class file archives is increasingly often the distribution mechanism of choice for stand-alone Java applications as well (especially for "network computers"). Since the time required to download an application is proportional to the size of the archive, it is desirable for the archive to be as small as possible.

In this paper we evaluate the effectiveness of a number of compiler-optimization and program transformation techniques for extracting Java applications[2]. These transformations include:

- removal of redundant methods and fields,
- devirtualization and inlining of method calls,
- transformation of the class hierarchy, and
- renaming of packages, classes, methods and fields,

and have the effect of reducing application size. Application extraction is generally believed to be especially useful when an application is shipped with a (proprietary) class library, because typically only a small fraction of the library's functionality is used. In such cases, "pruning away" unused library functionality can dramatically reduce application size.

We implemented a number of application extraction techniques in the context of *Jax* (short for Jikes Application eXtractor). *Jax* reads in the class files [15] that constitute a Java application, and performs a whole-program analysis to determine the components (e.g., classes, methods, and fields) of the application that must be retained in order to preserve program behavior. *Jax* removes the unnecessary components, performs several size-reducing transformations to the application, and writes out a class file archive containing the extracted application. *Jax* relies on user input to specify the components of the application that are accessed using Java's reflection mechanism [3], but the extraction process is fully automatic otherwise. *Jax* has been available on IBM's alphaWorks web site[3] since June 1998 and has been downloaded over 10,000 times since then. We are planning to ship *Jax* as a Technology Preview with an IBM product (IBM VisualAge Java 3.0, Enterprise Edition) later this year.

We evaluate the performance of *Jax* on a set of real-life benchmarks ranging from 27 to 2,332 classes (the corresponding archives range from 56,796 to 3,810,120 bytes), and measure a reduction in archive size ranging from 13.4% to 90.2% (48.7% on average[4]). Measurements over modem and LAN connections confirm that download times are reduced proportionally.

[1] Java is a trademark of Sun Microsystems.

[2] In what follows, the word 'application' will be used to refer to applications as well as applets, unless otherwise stated.

[3] www.alphaWorks.ibm.com/tech/JAX

[4] All average percentages reported in this paper are computed using the geometric mean.

OOPSLA '99 11/99 Denver, CO, USA

Exception: "Practical Experience with an Application Extractor for Java", by Tip et al. OOPSLA 1999

PREVIOUS WORK

Lots of good work on debloating C/C++; but Java has been largely ignored.

Practical Experience with an Application Extractor for Java

Frank Tip Chris Laffra Peter F. Sweeney
IBM T.J. Watson Research Center
P.O. Box 704, Yorktown Heights, NY 10598, USA
{tip,laffra,pfs}@watson.ibm.com

David Streeter
IBM Toronto Laboratory
1150 Eglinton Ave. East
Toronto, Ontario, Canada
daves@ca.ibm.com

Abstract

Java programs are routinely transmitted over low-bandwidth network connections as compressed class file archives (i.e., zip files and jar files). Since archive size is directly proportional to download time, it is desirable for applications to be as small as possible. This paper is concerned with the use of program transformations such as removal of dead methods and fields, inlining of method calls, and simplification of the class hierarchy for reducing application size. Such "extraction" techniques are generally believed to be especially useful for applications that use class libraries, since typically only a small fraction of a library's functionality is used. By "pruning away" unused library functionality, application size can be reduced dramatically. We implemented a number of application extraction techniques in Jax, an application extractor for Java, and evaluate their effectiveness on a set of realistic benchmarks ranging from 27 to 2,332 classes (with archives ranging from 56,796 to 3,810,120 bytes). We report archive size reductions ranging from 13.4% to 90.2% (48.7% on average).

1 Introduction

Java[1] [10] programs are routinely transmitted over the internet as compressed class file archives (i.e., zip files and jar files). A typical example of this situation consists of downloading a web page that contains one or more embedded Java applets. The downloading of class file archives is increasingly often the distribution mechanism of choice for stand-alone Java applications as well (especially for "network computers"). Since the time required to download an application is proportional to the size of the archive, it is desirable for the archive to be as small as possible.

In this paper we evaluate the effectiveness of a number of compiler-optimization and program transformation techniques for extracting Java applications[2]. These transformations include:

- removal of redundant methods and fields,
- devirtualization and inlining of method calls,
- transformation of the class hierarchy, and
- renaming of packages, classes, methods and fields,

and have the effect of reducing application size. Application extraction is generally believed to be especially useful when an application is shipped with a (proprietary) class library, because typically only a small fraction of the library's functionality is used. In such cases, "pruning away" unused library functionality can dramatically reduce application size.

We implemented a number of application extraction techniques in the context of *Jax* (short for Jikes Application eXtractor). *Jax* reads in the class files [15] that constitute a Java application, and performs a whole-program analysis to determine the components (e.g., classes, methods, and fields) of the application that must be retained in order to preserve program behavior. *Jax* removes the unnecessary components, performs several size-reducing transformations to the application, and writes out a class file archive containing the extracted application. *Jax* relies on user input to specify the components of the application that are accessed using Java's reflection mechanism [3], but the extraction process is fully automatic otherwise. *Jax* has been available on IBM's alphaWorks web site[3] since June 1998 and has been downloaded over 10,000 times since then. We are planning to ship *Jax* as a Technology Preview with an IBM product (IBM VisualAge Java 3.0, Enterprise Edition) later this year.

We evaluate the performance of *Jax* on a set of real-life benchmarks ranging from 27 to 2,332 classes (the corresponding archives range from 56,796 to 3,810,120 bytes), and measure a reduction in archive size ranging from 13.4% to 90.2% (48.7% on average[4]). Measurements over modem and LAN connections confirm that download times are reduced proportionally.

[1] Java is a trademark of Sun Microsystems.

[2] In what follows, the word 'application' will be used to refer to applications as well as applets, unless otherwise stated.

[3] www.alphaWorks.ibm.com/tech/JAX

[4] All average percentages reported in this paper are computed using the geometric mean.

Exception: "Practical Experience with an Application Extractor for Java", by Tip et al. OOPSLA 1999

- Proposed and evaluated a set of Java bytecode transformations.
- These transformations have been utilized heavily by other researchers in this area (including us!).
- **The effectiveness of these transformations at preserving program behavior has not been evaluated thoroughly in a modern context.**

LIMITATIONS OF THE PREVIOUS WORK

LIMITATIONS OF THE PREVIOUS WORK

Previous works relied on purely **static** analysis.

LIMITATIONS OF THE PREVIOUS WORK

Previous works relied on purely **static** analysis.

Java contains **dynamic** features.

LIMITATIONS OF THE PREVIOUS WORK

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod(args[0]);
    method1.invoke(appObj);
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```

Previous works relied on purely **static** analysis.

Java contains **dynamic** features.

LIMITATIONS OF THE PREVIOUS WORK

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod(args[0]);
    method1.invoke(appObj);
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```

Previous works relied on purely **static** analysis.

Java contains **dynamic** features.

Question: What methods are reachable in this program?

LIMITATIONS OF THE PREVIOUS WORK

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod(args[0]);
    method1.invoke(appObj);
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```

Previous works relied on purely **static** analysis.

Java contains **dynamic** features.

Question: What methods are reachable in this program?

Answer: Technically, they all are.

LIMITATIONS OF THE PREVIOUS WORK

```
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod(args[0]);
    method1.invoke(appObj);
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```

Previous works relied on purely **static** analysis.

Java contains **dynamic** features.

Question: What methods are reachable in this program?

Answer: Technically, they all are.

Though true, this answer is unhelpful.

LIMITATIONS OF THE PREVIOUS WORK

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod(args[0]);
    method1.invoke(appObj);
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```

Previous works relied on purely **static** analysis.

Java contains **dynamic** features.

Question: What methods are reachable in this program?

Answer: Technically, they all are.

Though true, this answer is unhelpful.

In Java, practical reachability analysis **requires a dynamic component**

LIMITATIONS OF THE PREVIOUS WORK

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod(args[0]);
    method1.invoke(appObj);
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```

Previous works relied on purely **static** analysis.

Java contains **dynamic** features.

Question: What methods are reachable in this program?

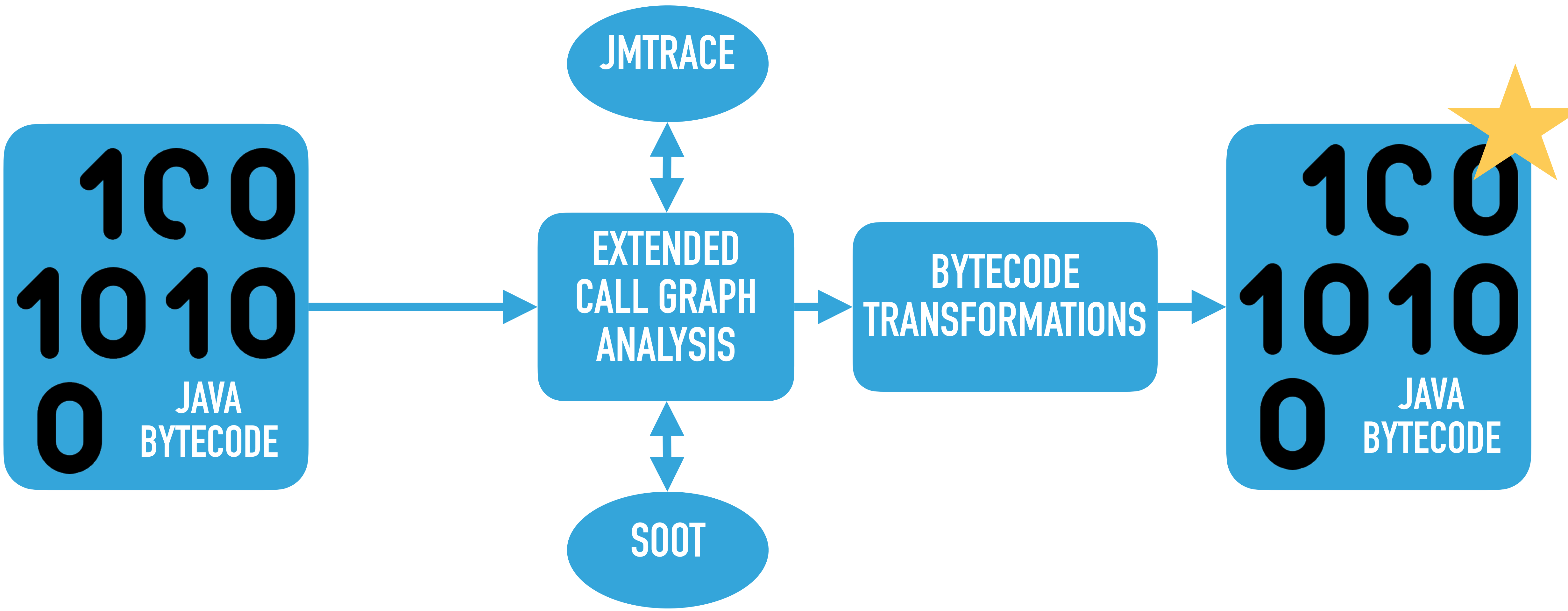
Answer: Technically, they all are.

Though true, this answer is unhelpful.

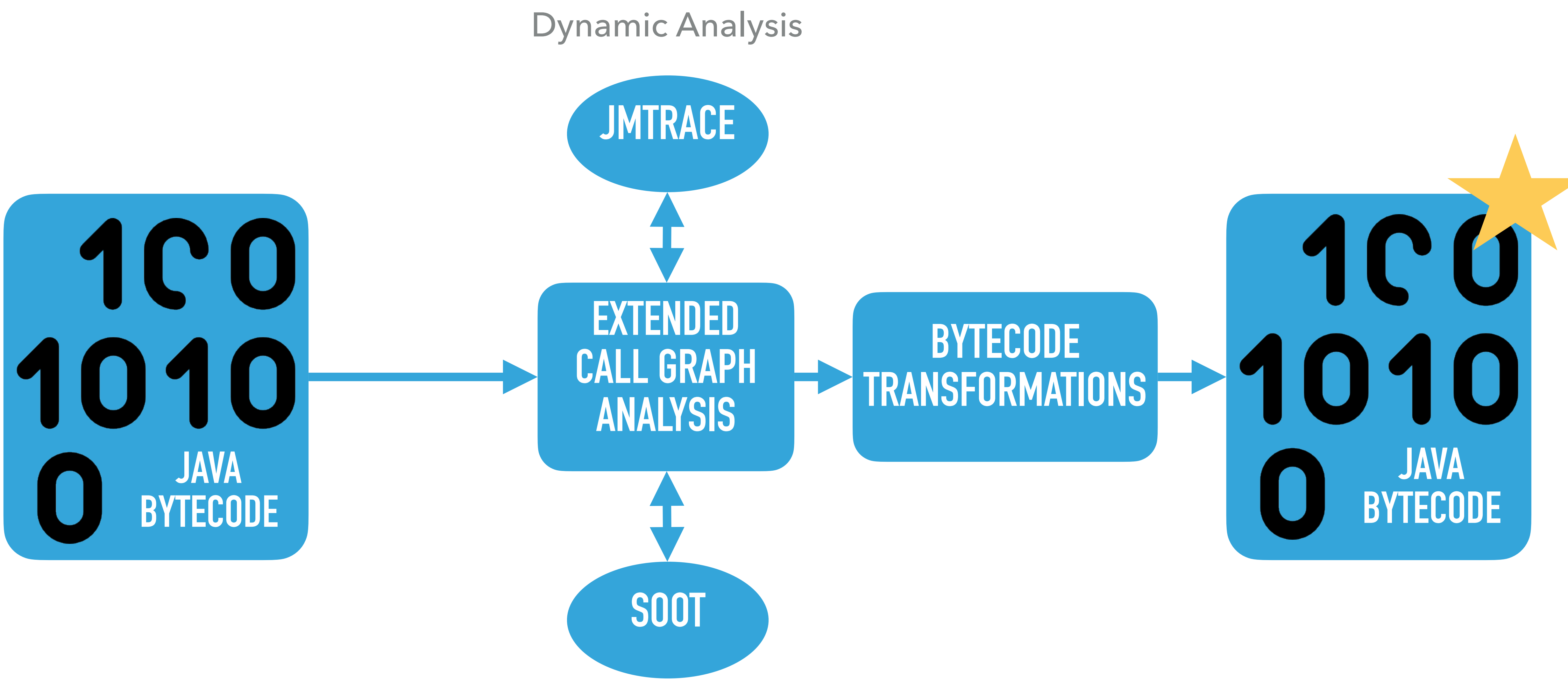
In Java, practical reachability analysis **requires a dynamic component**

Without decent reachability analysis, debloating is impossible.

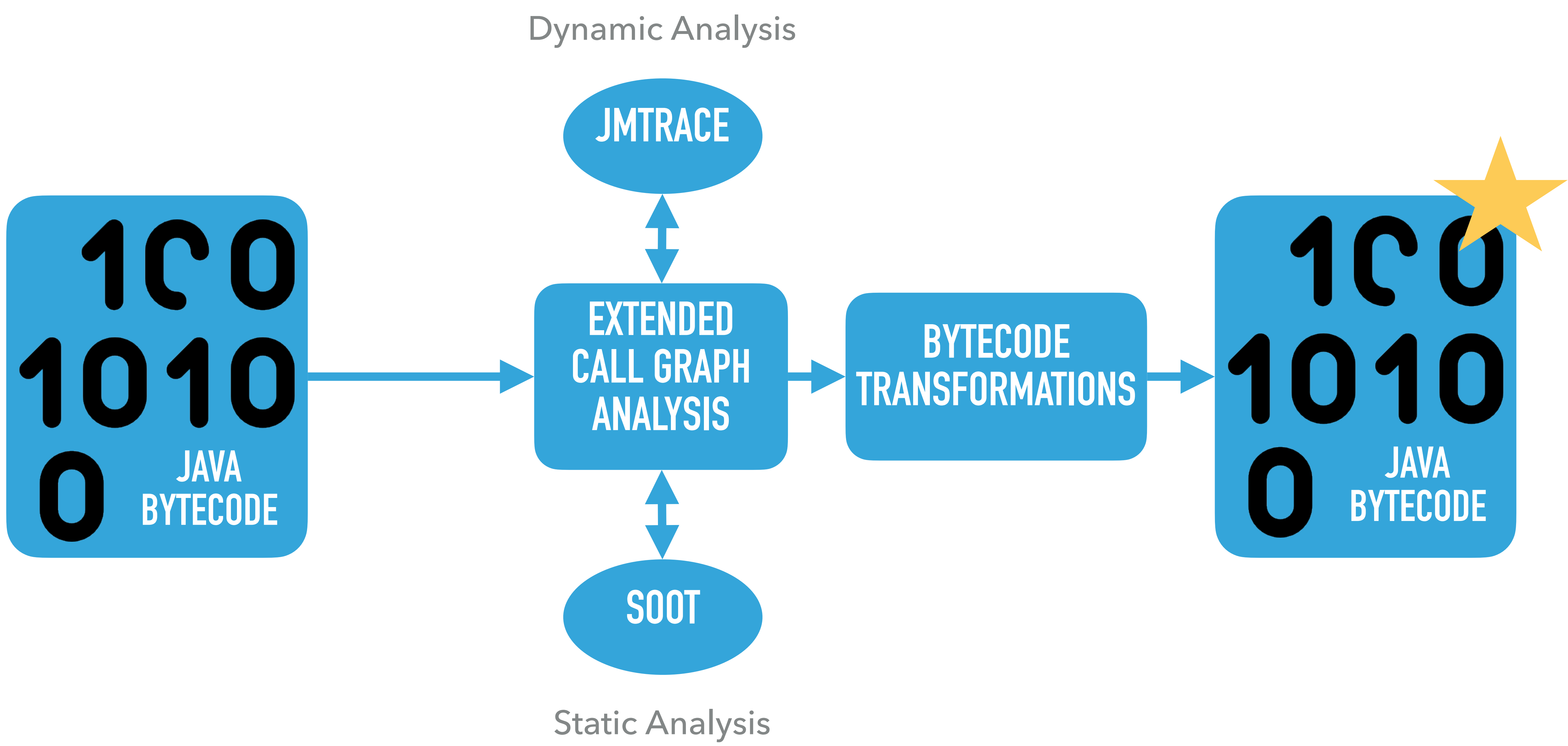
JSHRINK: A HIGH LEVEL VIEW



JSHRINK: A HIGH LEVEL VIEW



JSHRINK: A HIGH LEVEL VIEW



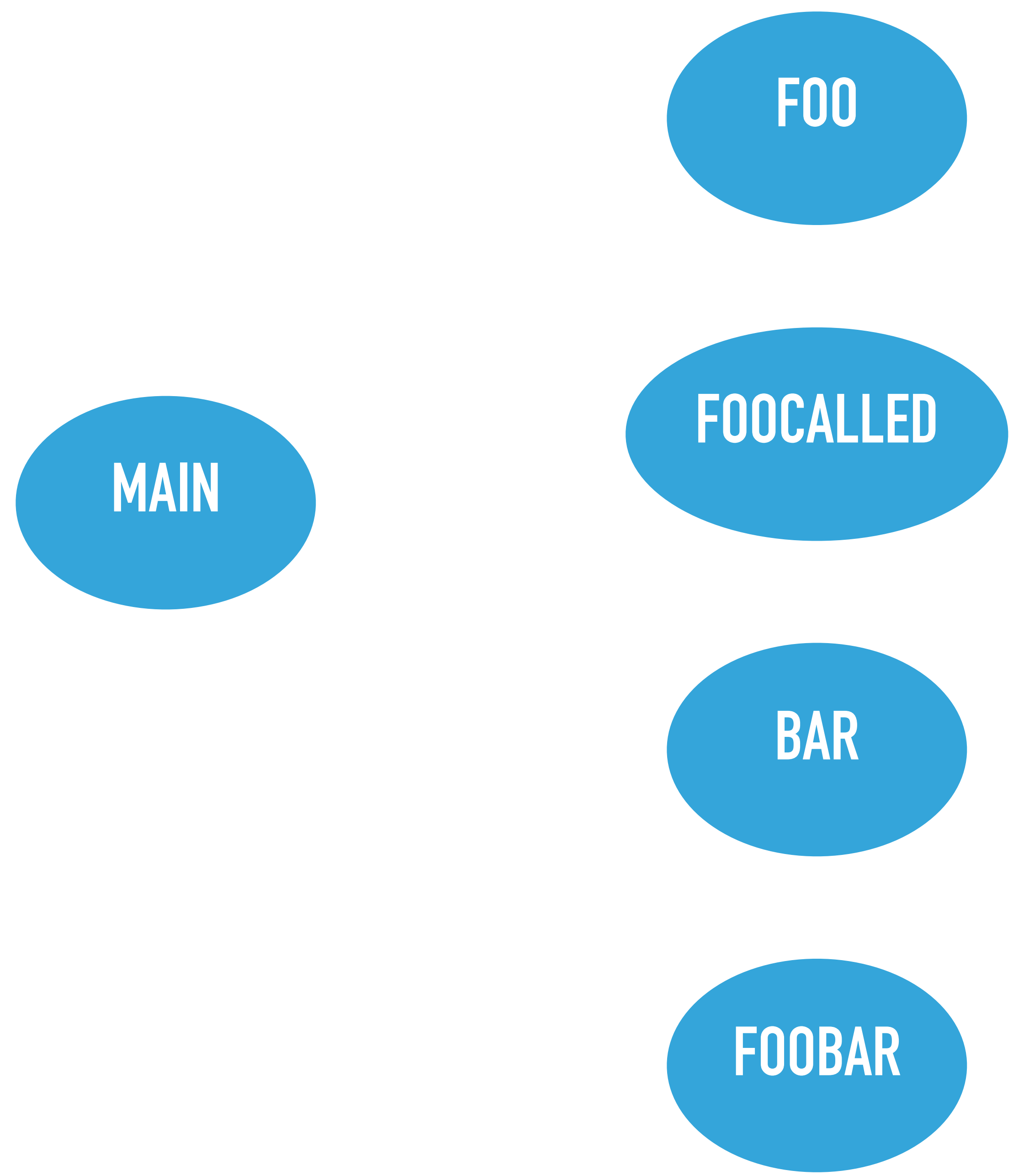
THE EXTENDED CALL GRAPH

THE EXTENDED CALL GRAPH

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```

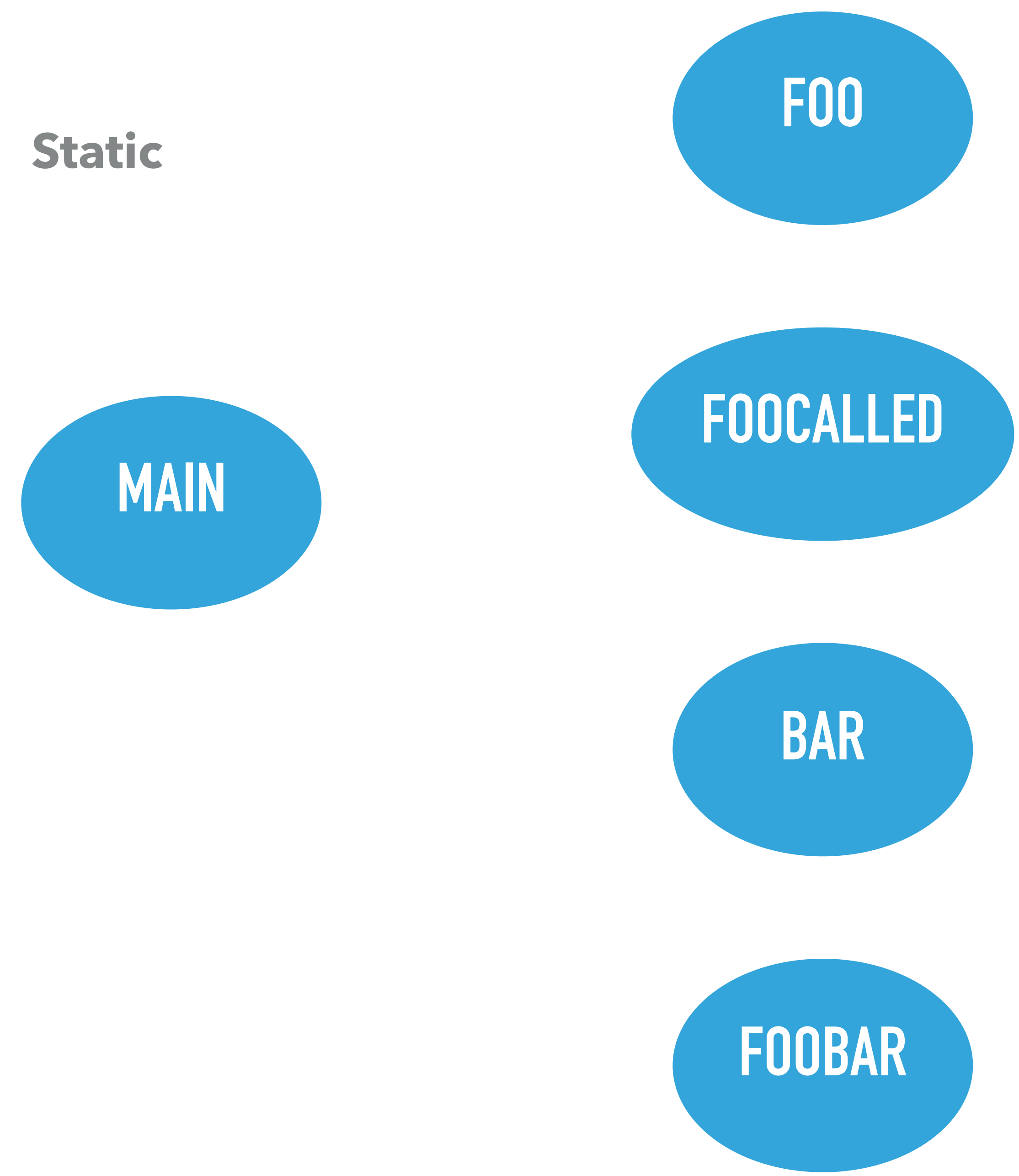
THE EXTENDED CALL GRAPH

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```



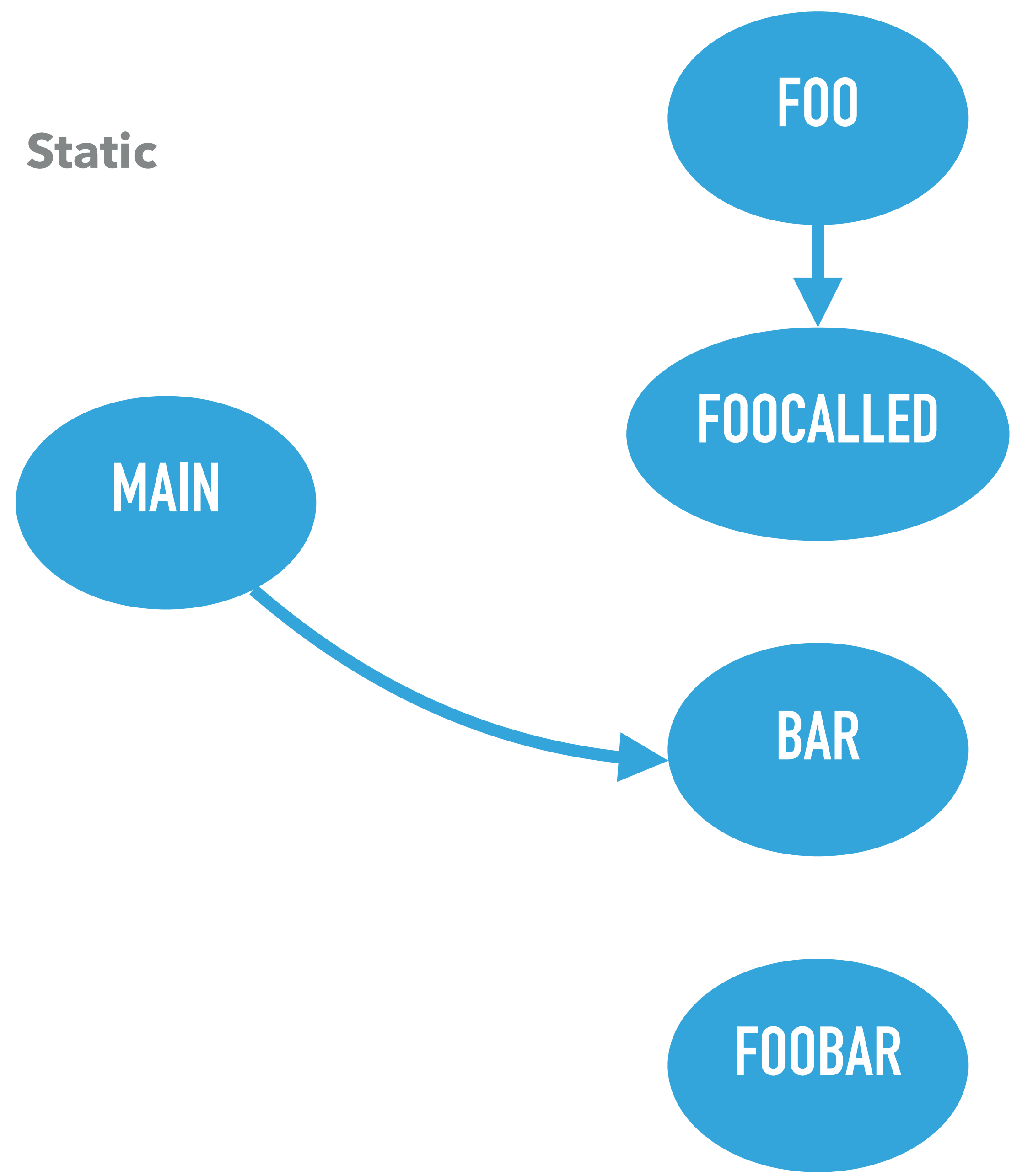
THE EXTENDED CALL GRAPH

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```



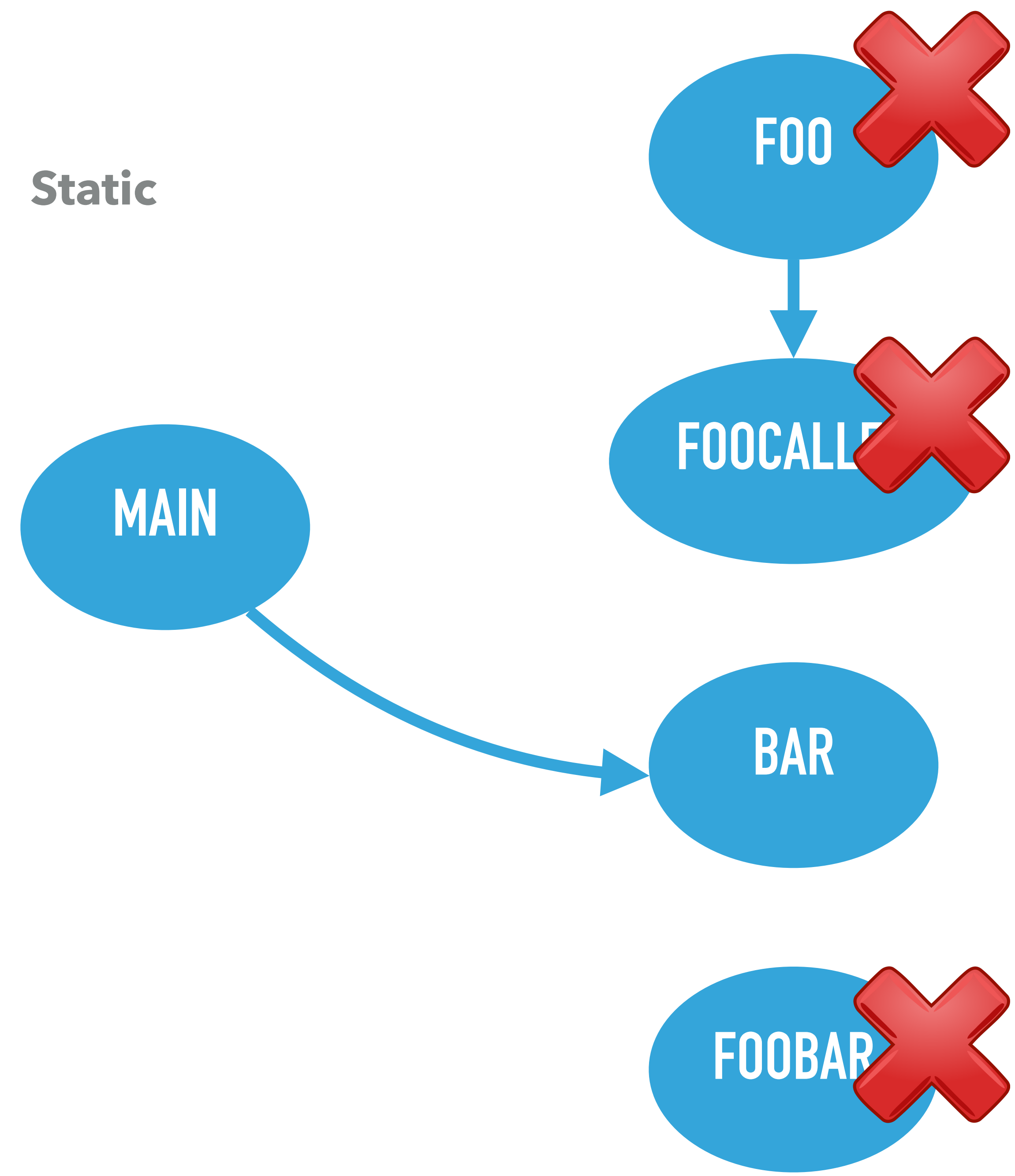
THE EXTENDED CALL GRAPH

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```



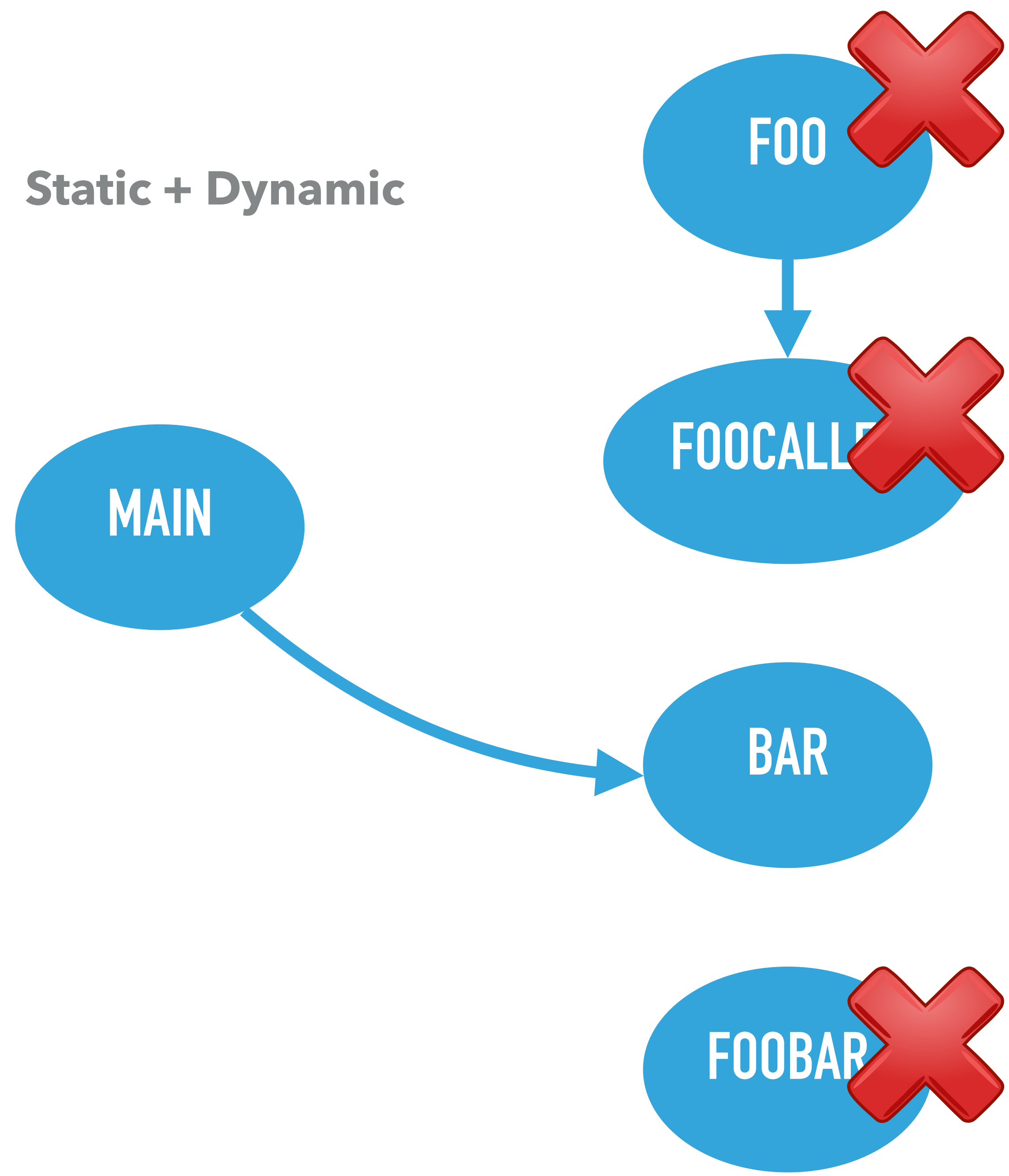
THE EXTENDED CALL GRAPH

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```



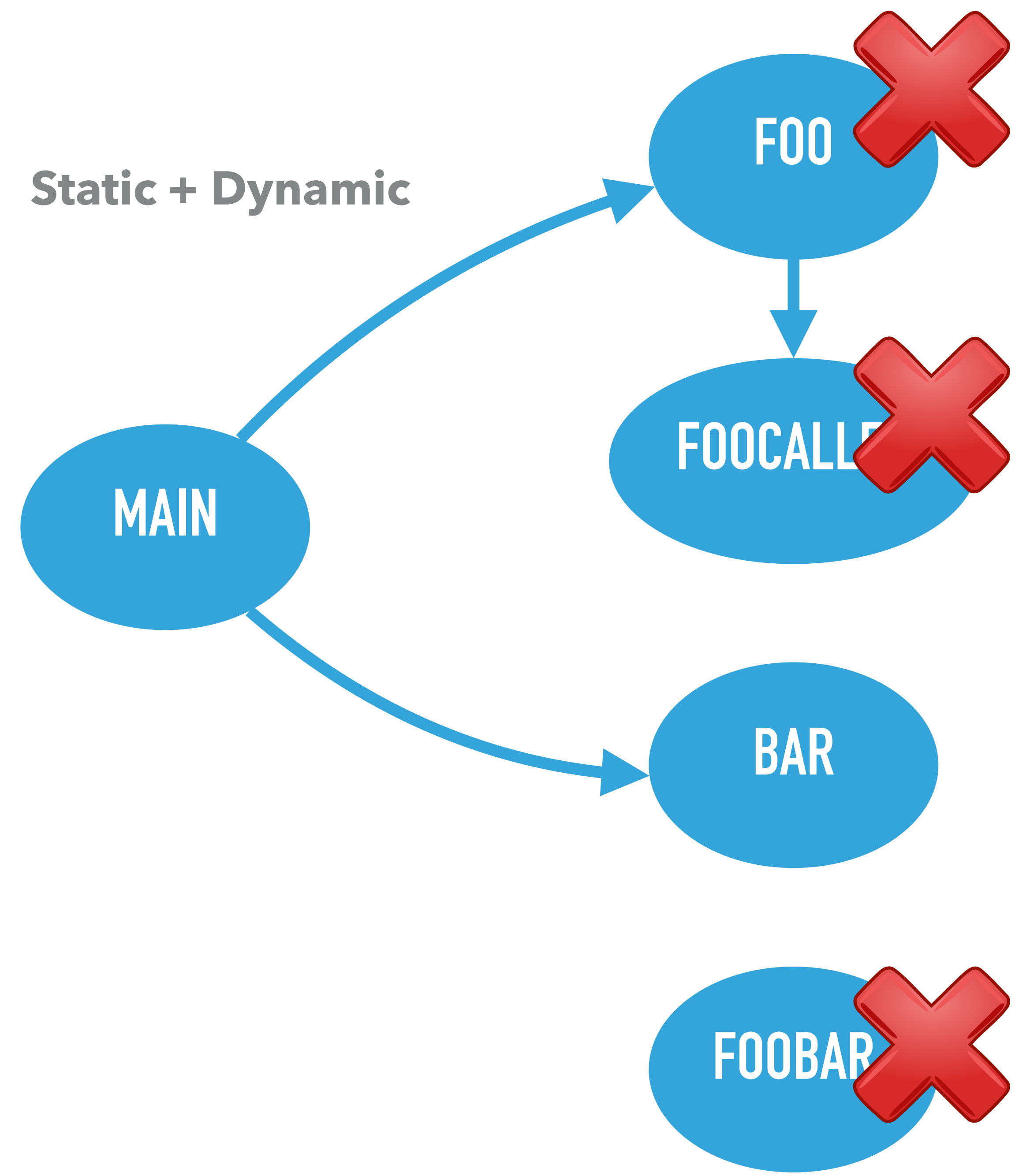
THE EXTENDED CALL GRAPH

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```



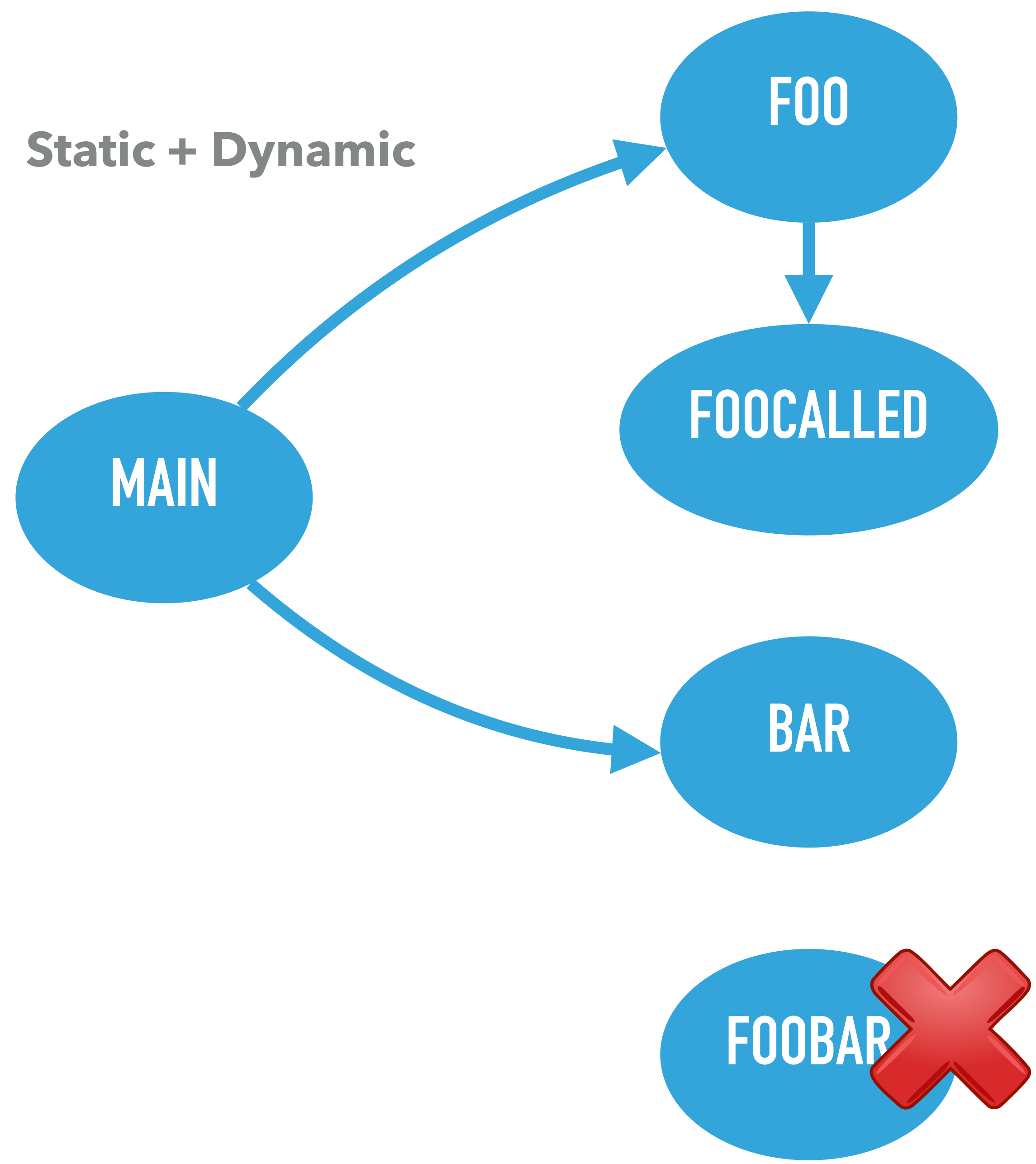
THE EXTENDED CALL GRAPH

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```



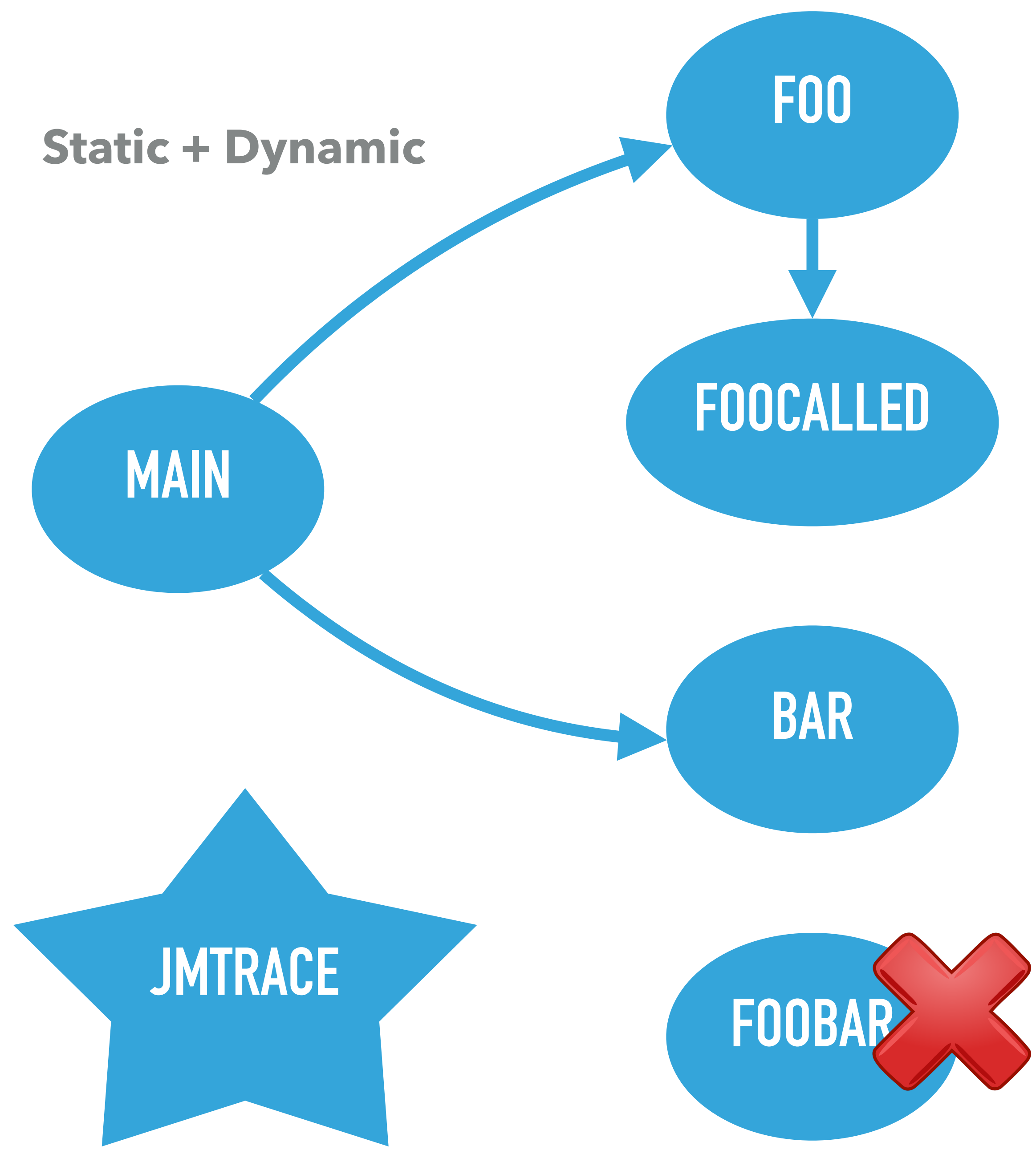
THE EXTENDED CALL GRAPH

```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```



THE EXTENDED CALL GRAPH

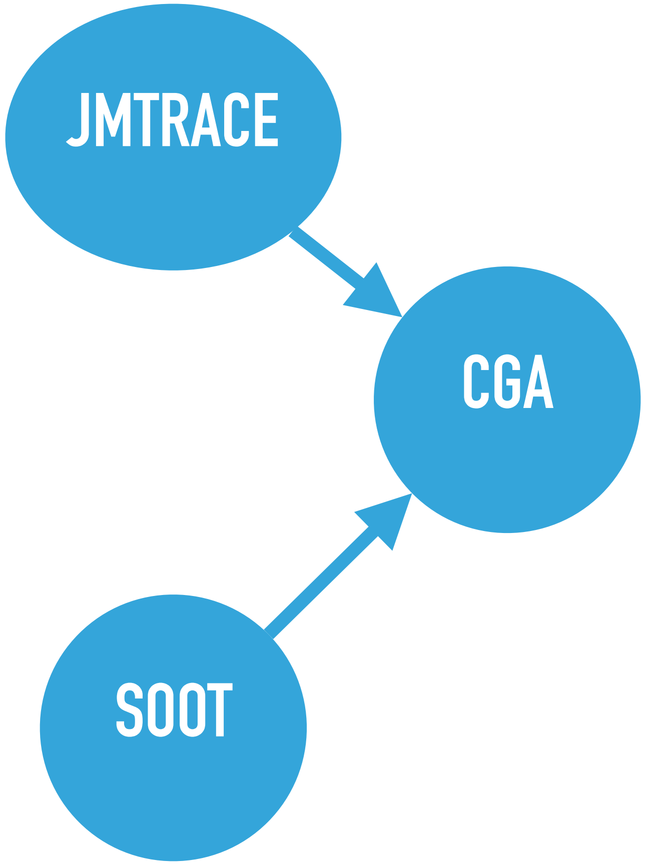
```java
import java.lang.reflect.*;

class Reflection{
  public static void main(String[] args)
    throws IllegalAccessException,
    InvocationTargetException,
    InstantiationException,
    NoSuchMethodException {

    Class appClass = Reflection.class;
    Object appObj = appClass.newInstance();

    Method method1 = appClass
      .getMethod("foo");
    method1.invoke(appObj);

    bar();
  }

  private void foo(){
    System.out.println("Hello " + fooCalled());
  }

  private void fooCalled(){
    System.out.println("world!");
  }

  private void bar(){
    System.out.println("Goodbye world!");
  }

  private void foobar(){
    System.out.println("Hello and goodbye");
  }
}
```



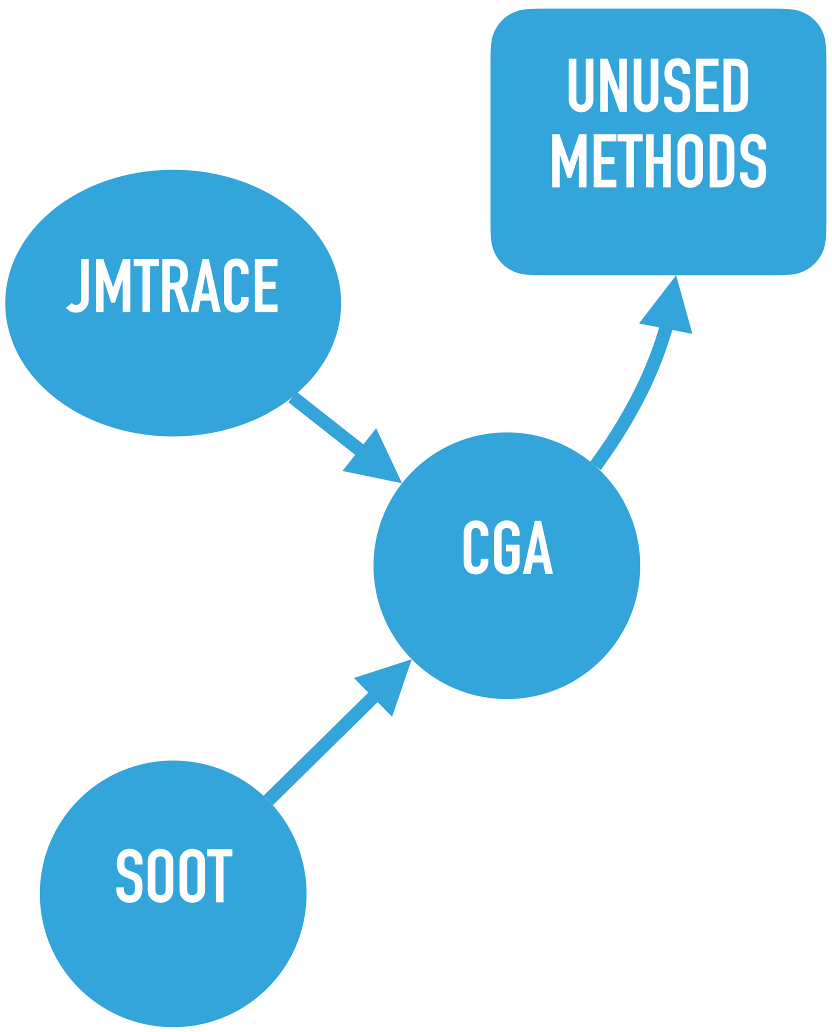
TRANSFORMATIONS

TRANSFORMATIONS

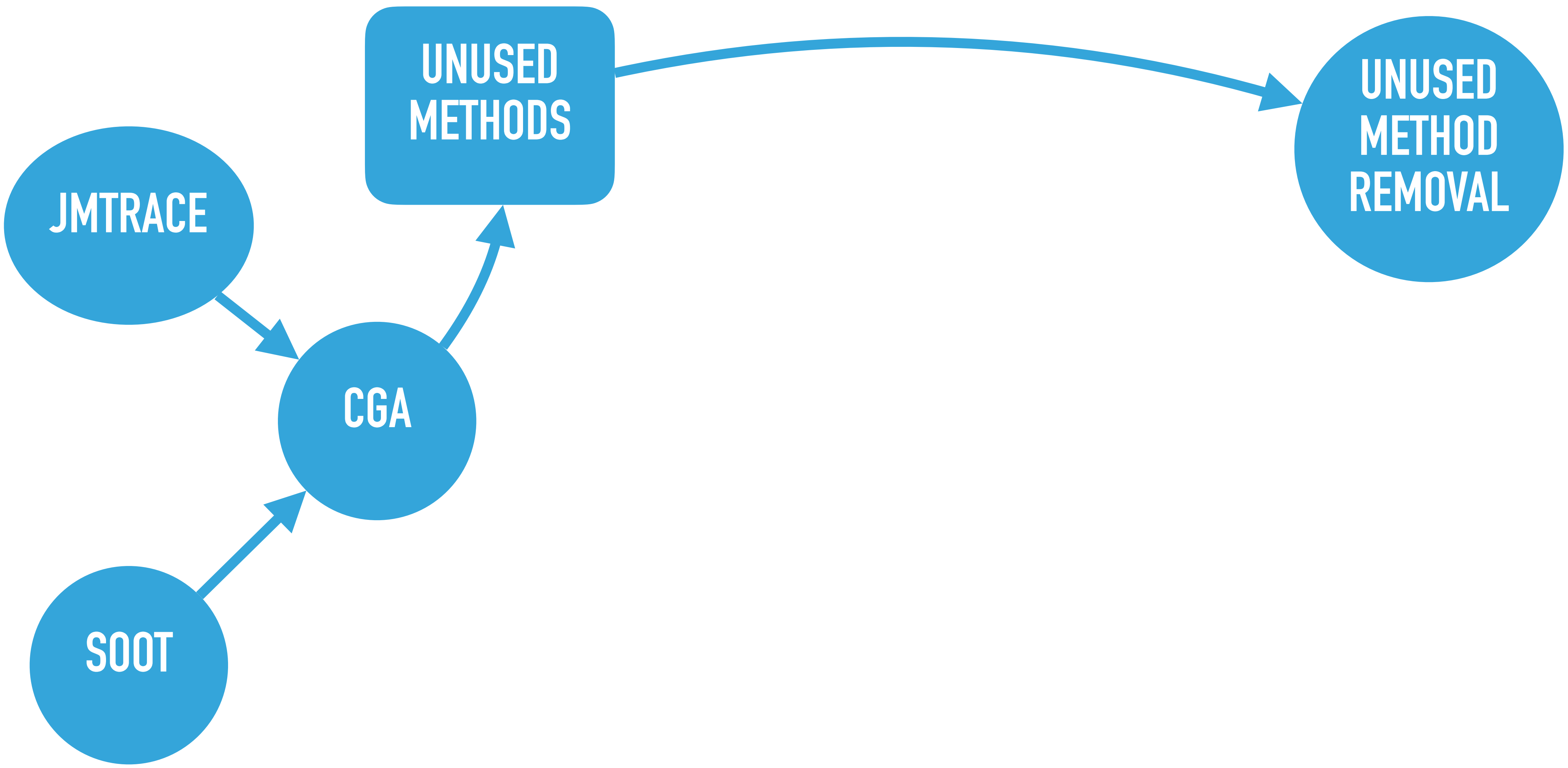
TRANSFORMATIONS



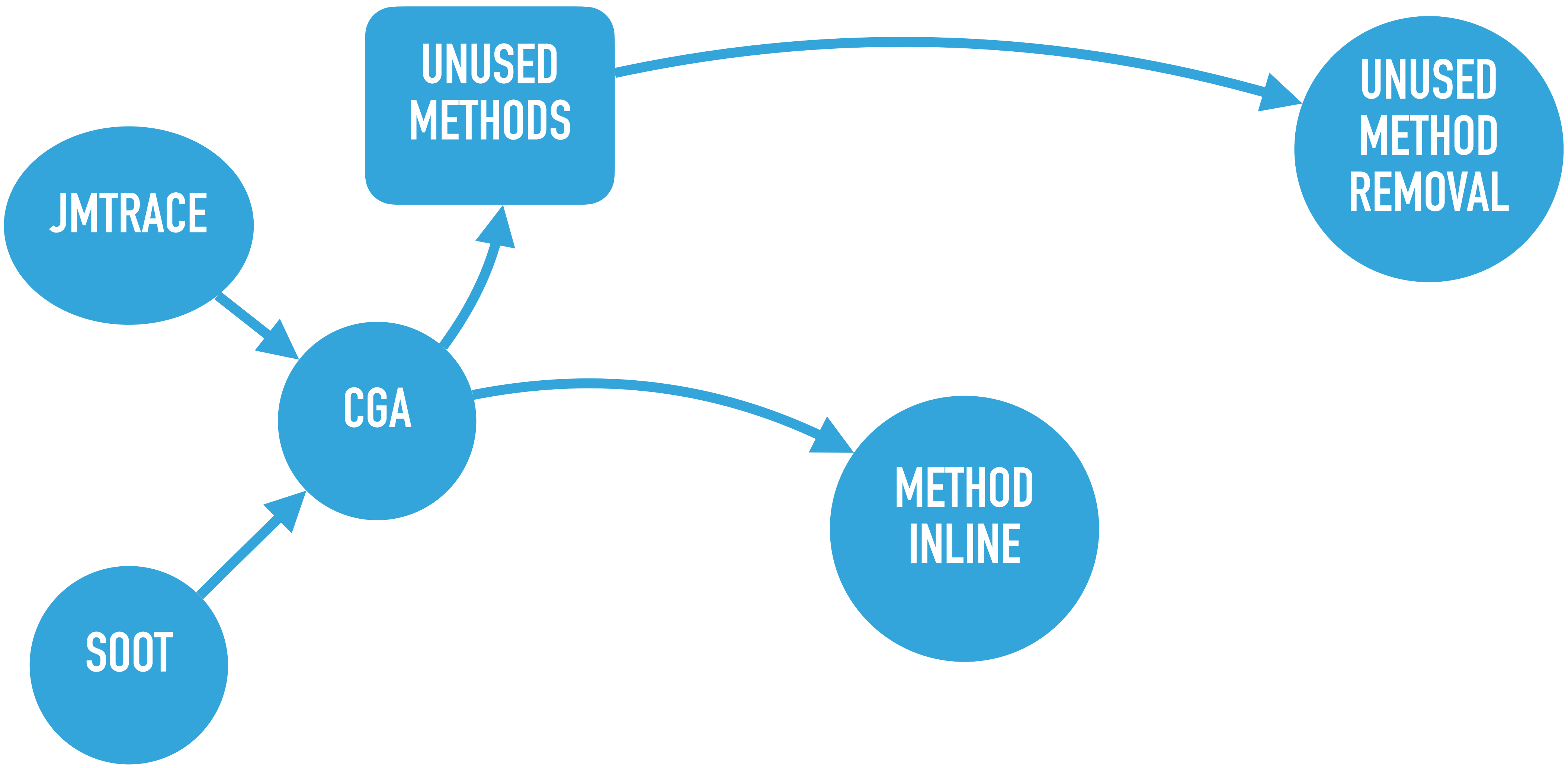
TRANSFORMATIONS



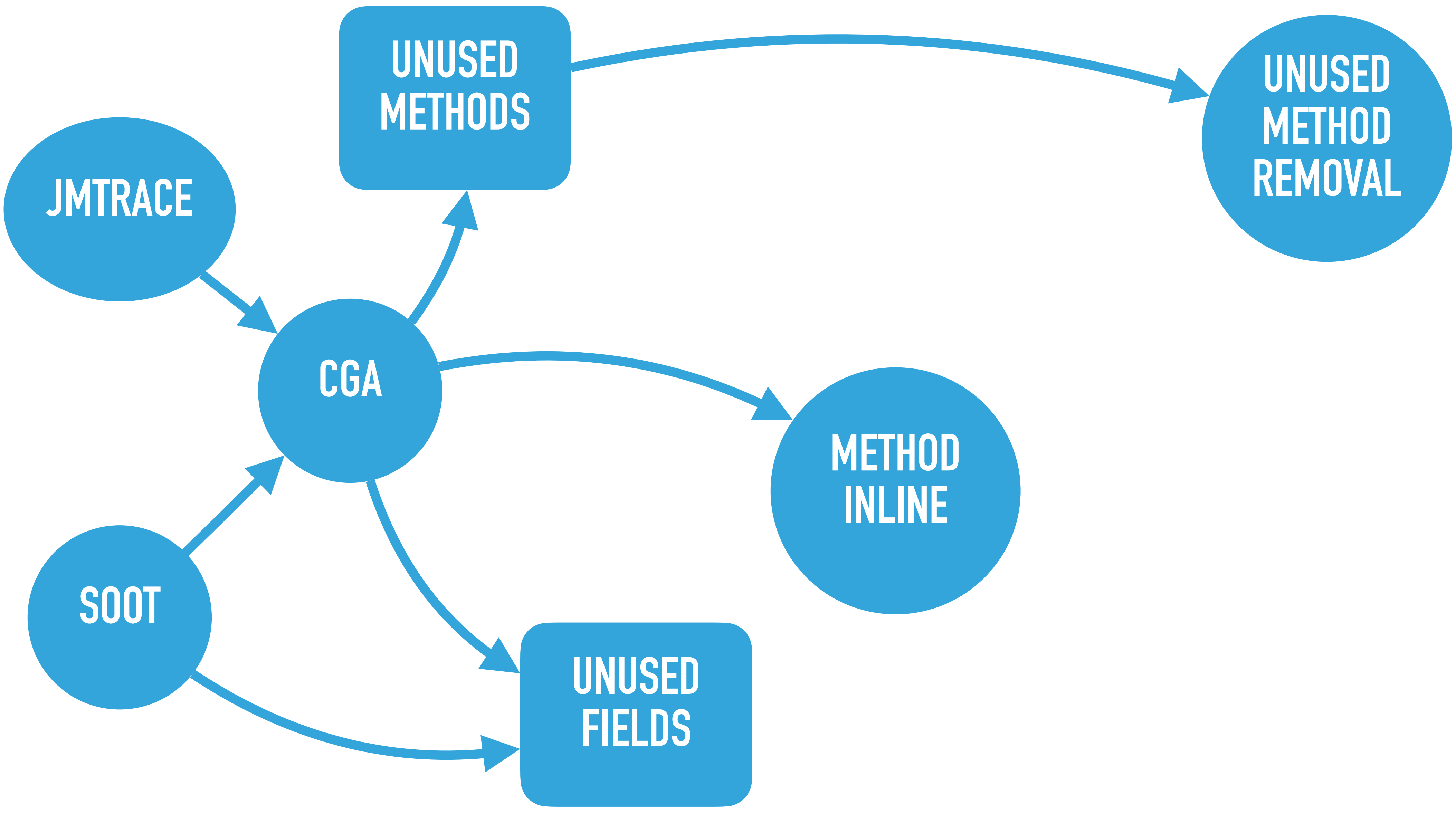
TRANSFORMATIONS



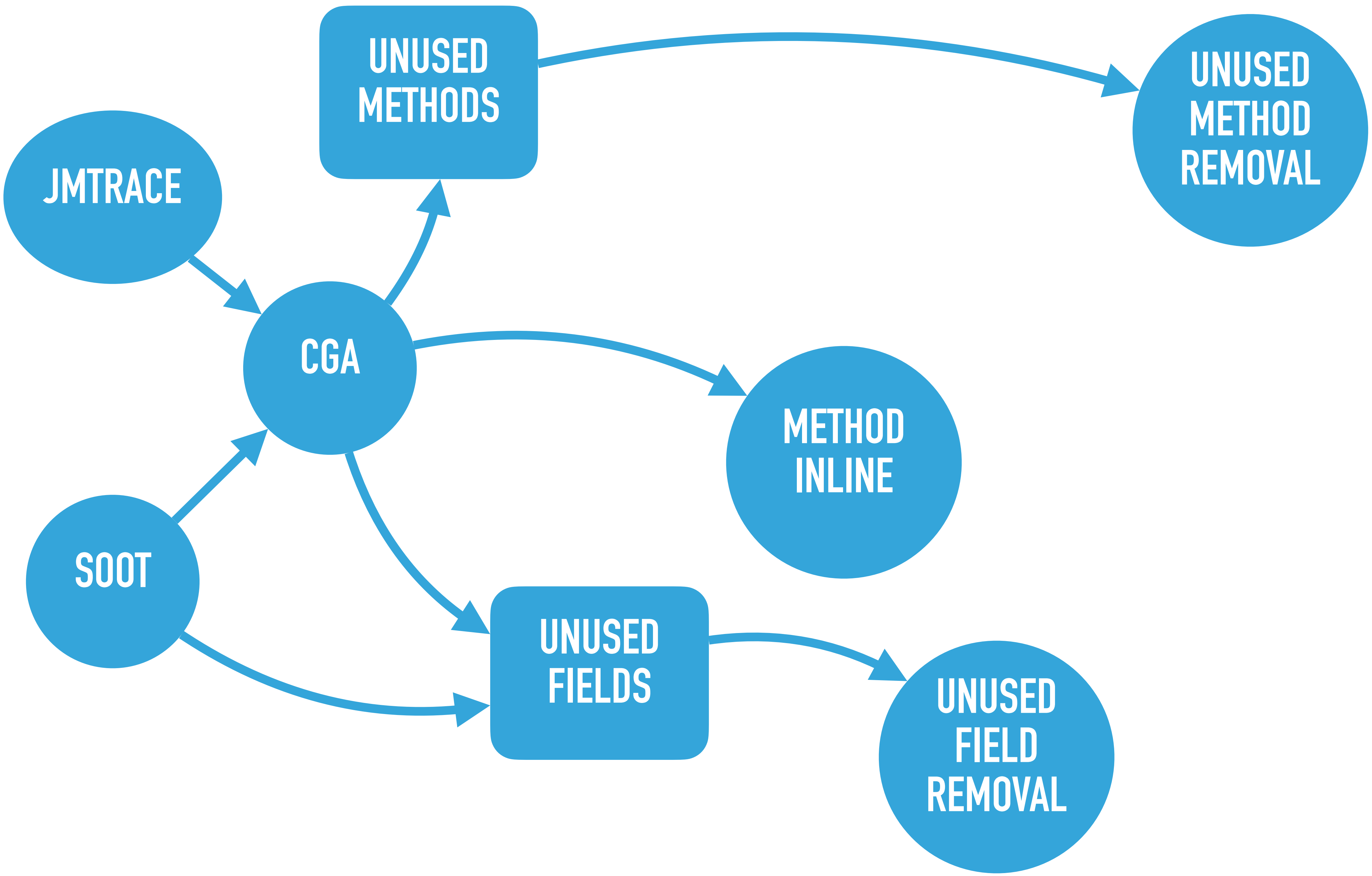
TRANSFORMATIONS



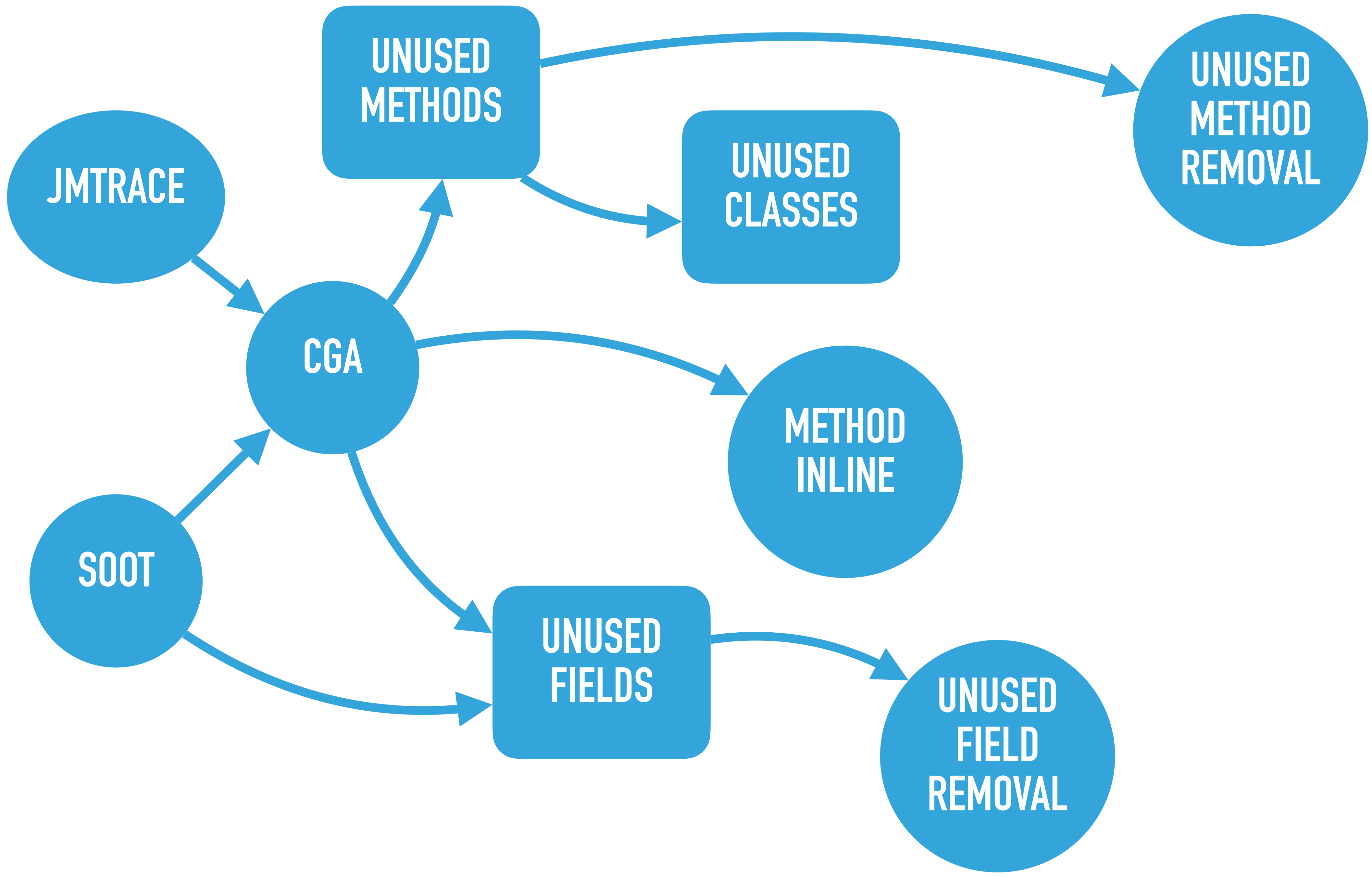
TRANSFORMATIONS



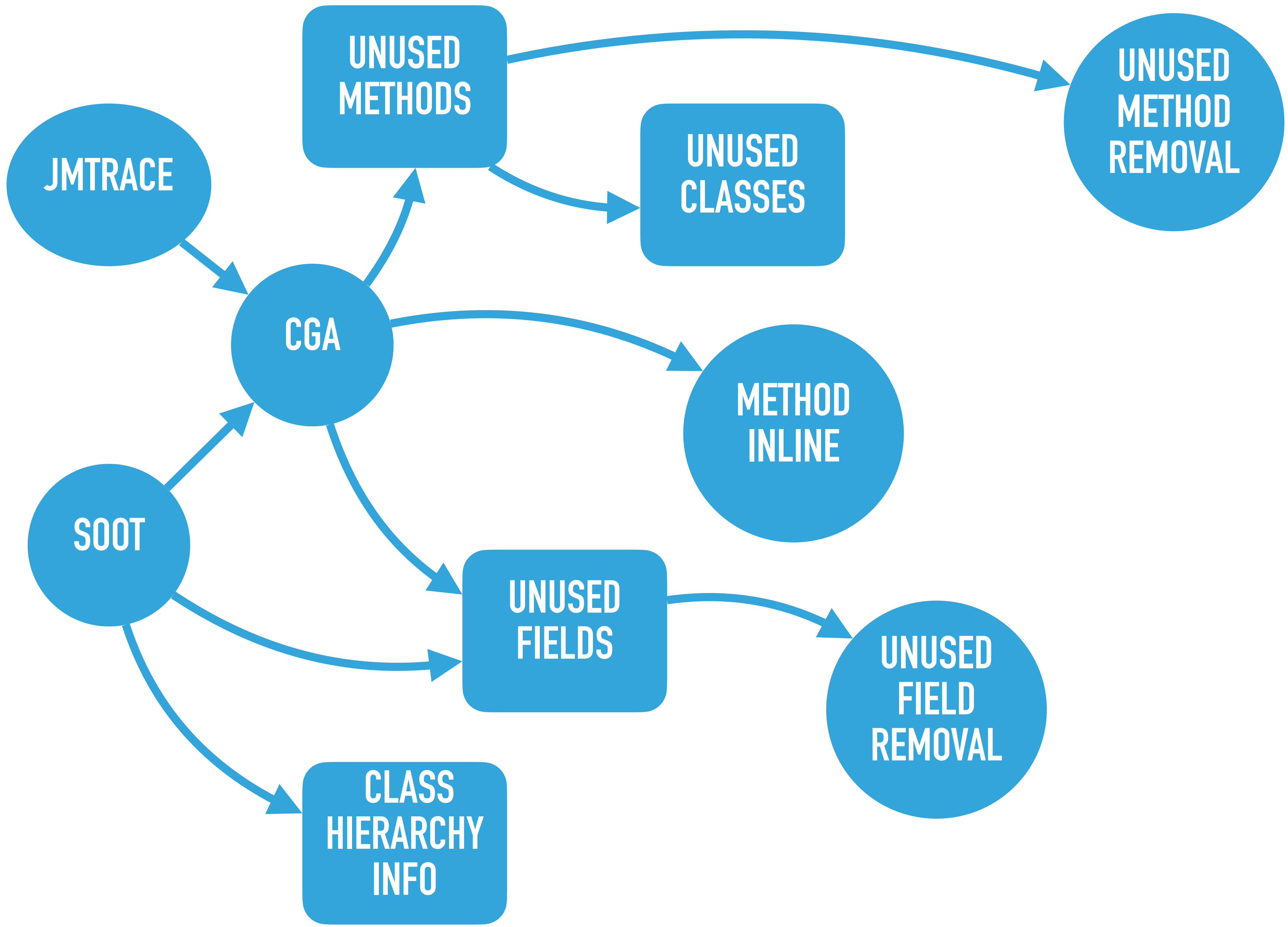
TRANSFORMATIONS



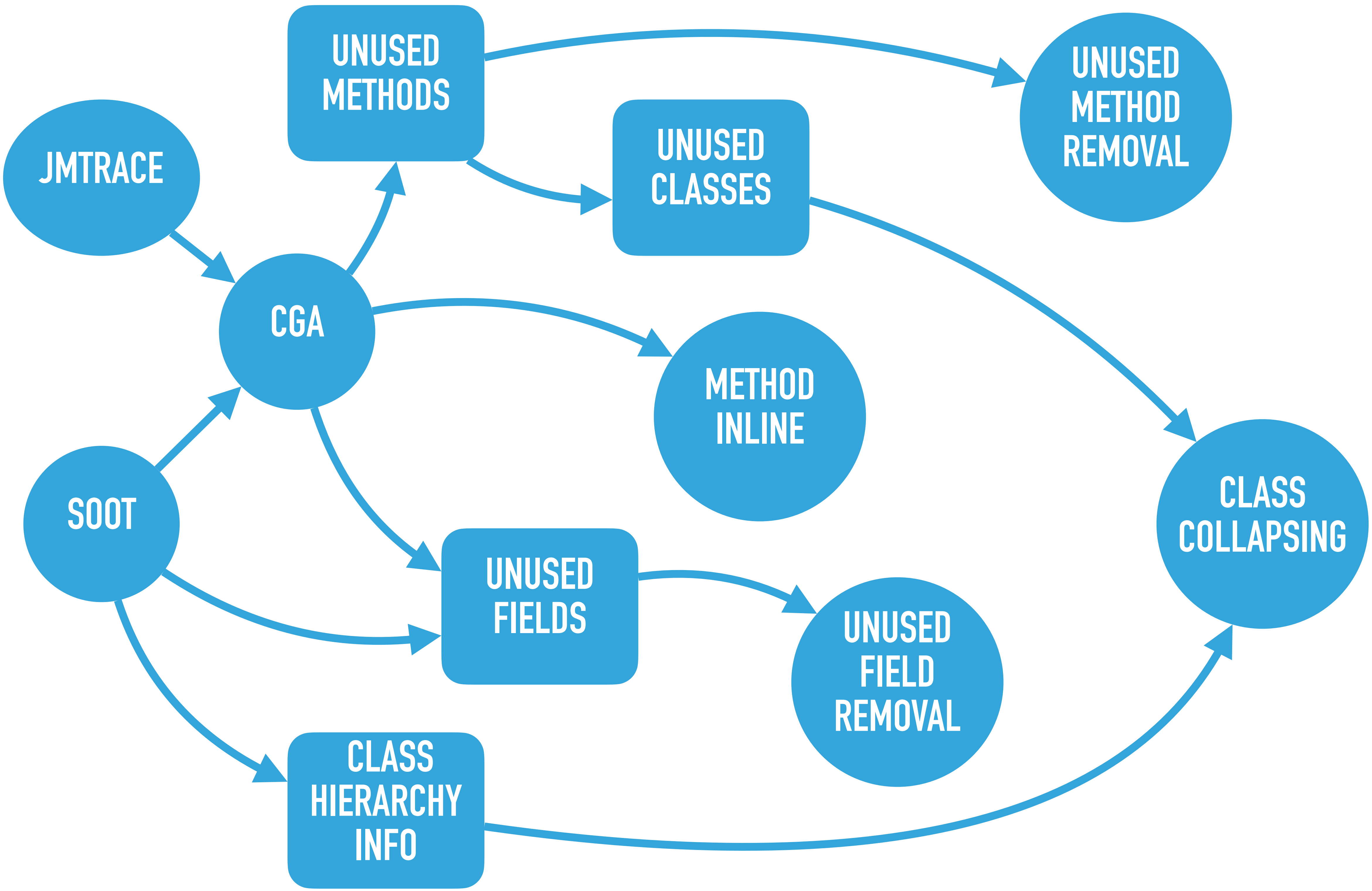
TRANSFORMATIONS



TRANSFORMATIONS

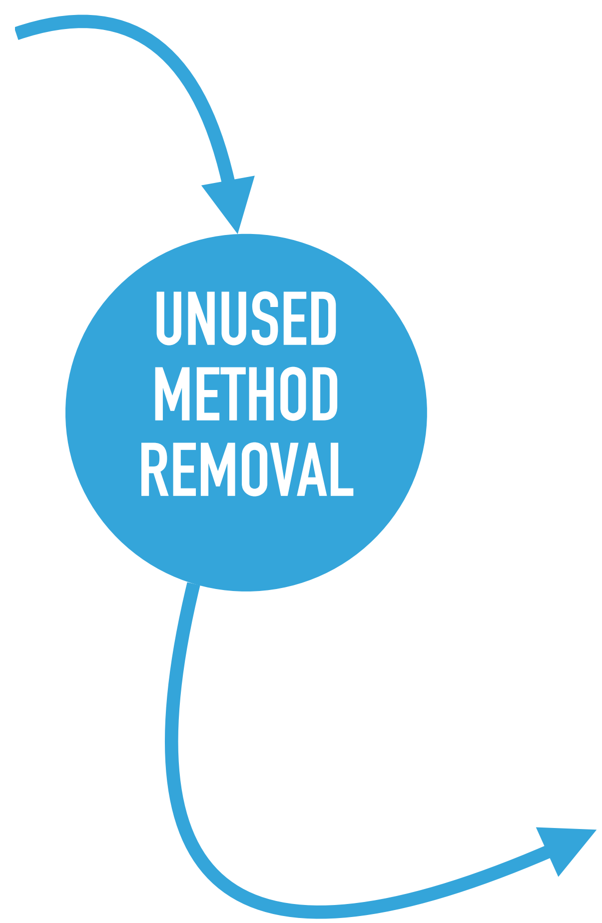


TRANSFORMATIONS



UNUSED METHOD REMOVAL

```
class A{
  public A(){ }

  public String method_1(){
    return "A_String";
  }
}

class B extends A{

  public String foo = "foo";
  public String bar = "bar";

  public static void main(String[] args){
    B b = new B();

    System.out.println(b.method_1());
    System.out.println(b.method_2());
  }

  public B(){
    super();
  }

  public String method_2(){
    return this.foo;
  }

  public String method_3(){
    return this.bar;
  }
}
```




```
class A{
  public A(){ }

  public String method_1(){
    return "A_String";
  }
}

class B extends A{

  public String foo = "foo";
  public String bar = "bar";

  public static void main(String[] args
    B b = new B();

    System.out.println(b.method_1());
    System.out.println(b.method_2());
  }

  public B(){
    super();
  }

  public String method_2(){
    return this.foo;
  }

  public String method_3(){
    return this.bar;
  }
}
```

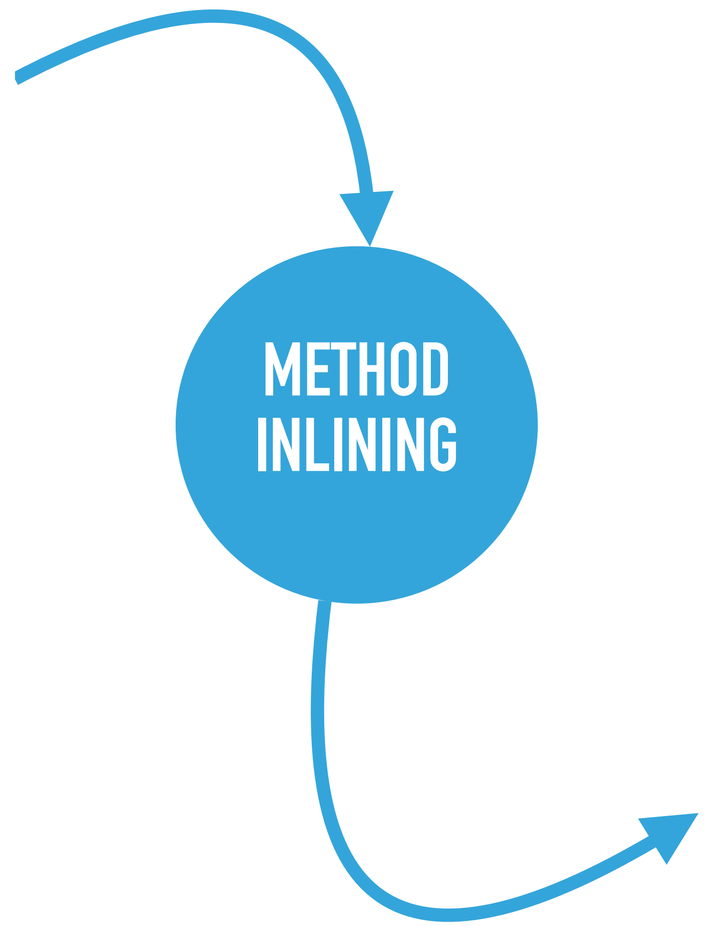
UNUSED FIELD REMOVAL

```
class A{
  public A(){ }

  public String method_1(){
    return "A_String";
  }
}

class B extends A{

  public String foo = "foo";
  public String bar = "bar";

  public static void main(String[] args){
    B b = new B();

    System.out.println(b.method_1());
    System.out.println(b.method_2());
  }

  public B(){
    super();
  }

  public String method_2(){
    return this.foo;
  }

}
```


```
class A{
  public A(){ }

  public String method_1(){
    return "A_String";
  }
}

class B extends A{

  public String foo = "foo";
  public String bar = "bar";

  public static void main(String[] args){
    B b = new B();

    System.out.println(b.method_1());
    System.out.println(b.method_2());
  }

  public B(){
    super();
  }

  public String method_2(){
    return this.foo;
  }

}
```

METHOD INLINING

```
class A{
  public A(){ }

  public String method_1(){
    return "A_String";
  }
}

class B extends A{

  public String foo = "foo";

  public static void main(String[] args){
    B b = new B();

    System.out.println(b.method_1());
    System.out.println(b.method_2());
  }

  public B(){
    super();
  }

  public String method_2(){
    return this.foo;
  }

}

class C extends A{}
```



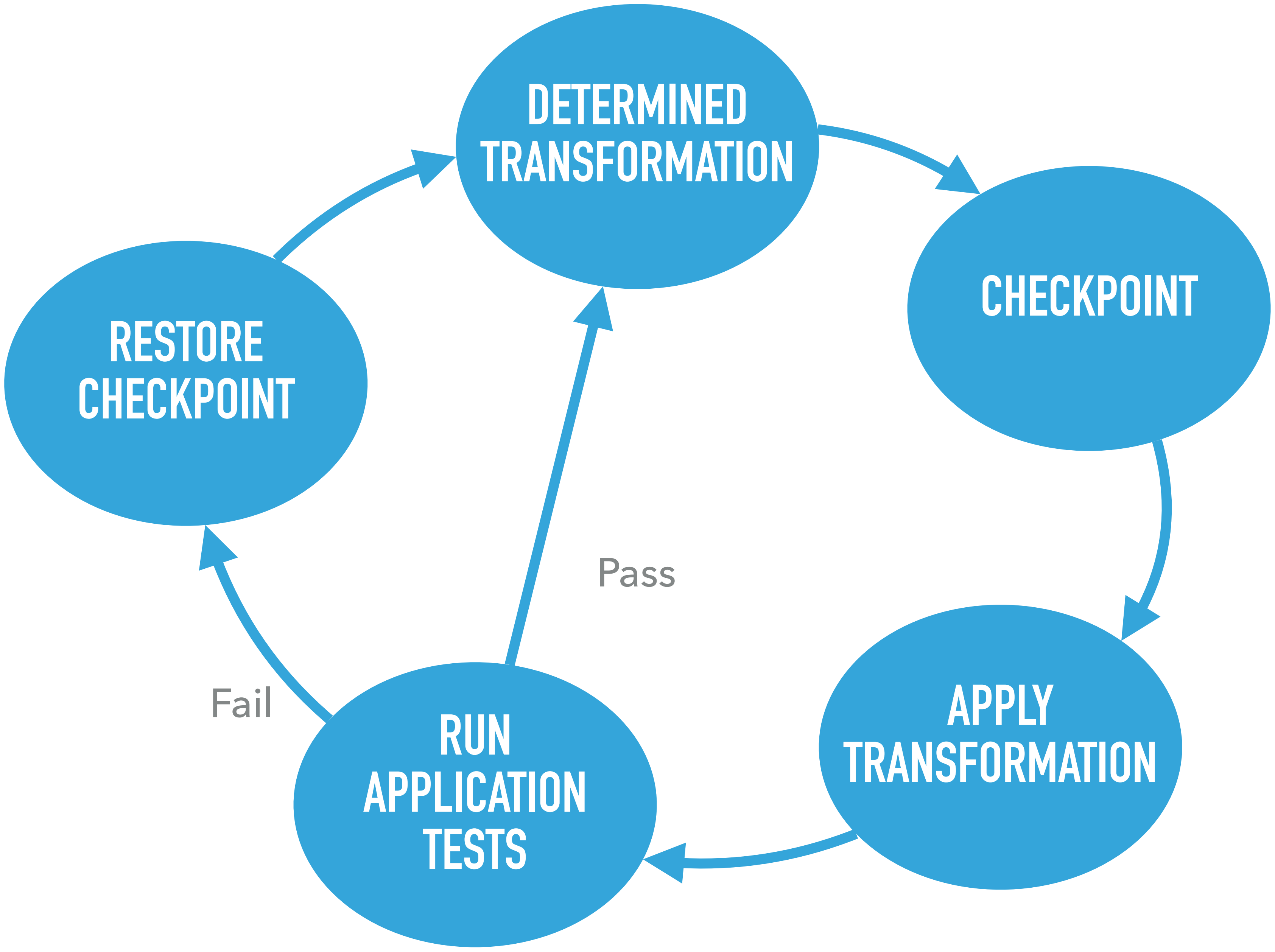

```
class A{
  public A(){ }

  public String method_1(){
    return "A_String";
  }
}

class B extends A{                  "A_String"

  public String foo = "foo";

  public static void main(String[] args){
    B b = new B();

    System.out.println(b.method_1());
    System.out.println(b.method_2());
  }

  public B(){
    super();                          this.foo
  }

  public String method_2(){
    return this.foo;
  }

}

class C extends A{}
```

CLASS COLLAPSING

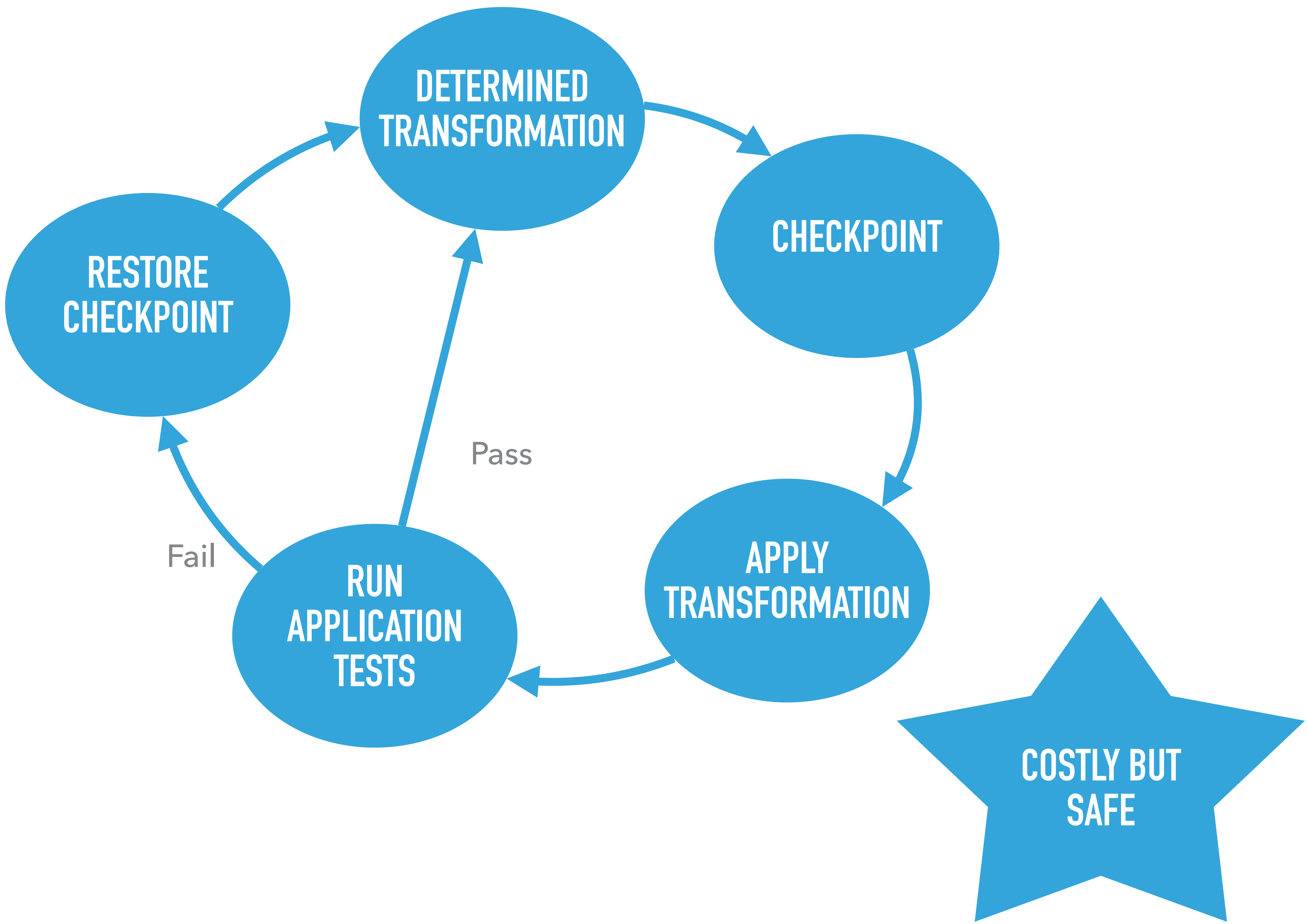
```java
class A{
  public A(){ }

}

class B extends A{

  public String foo = "foo";

  public static void main(String[] args){
    B b = new B();

    System.out.println("A_String");
    System.out.println(b.foo);
  }

  public B(){
    super();
  }

}

class C extends A{}
```


```java
class A{
  public A(){ }

}

class A{

  public String foo = "foo";

  public static void main(String[] args){
    A b = new A();

    System.out.println("A_String");
    System.out.println(b.foo);
  }

  public A(){
    super();
  }

}

class C extends A{}
```

CHECKPOINTING



CHECKPOINTING



RESEARCH AGENDA

RESEARCH AGENDA

Reduction – How much Java byte code reduction is achievable when applying different transformations?

RESEARCH AGENDA

Reduction – How much Java byte code reduction is achievable when applying different transformations?

Preservation and Robustness – To what extent are program semantics preserved when debloating software when using JShrink?

RESEARCH AGENDA

Reduction – How much Java byte code reduction is achievable when applying different transformations?

Preservation and Robustness – To what extent are program semantics preserved when debloating software when using JShrink?

Dynamic Impact – How does program behavior differ when running with dynamic analysis, compared to without?

CANDIDATE PROGRAMS

CANDIDATE PROGRAMS

Selection criteria:

- **Popular:** At least 100 GitHub stars.
- **Compilable:** We only support the Maven build system. This is a technical restriction.
- **Timeout**: The static call graph must be executable in 10 hours.
- **Testable**: We only selected applications with a JUnit test suite.

CANDIDATE PROGRAMS

Selection criteria:

- **Popular:** At least 100 GitHub stars.
- **Compilable:** We only support the Maven build system. This is a technical restriction.
- **Timeout**: The static call graph must be executable in 10 hours.
- **Testable**: We only selected applications with a JUnit test suite.

- jam-tools
- bucket
- qart4j
- dubbokeeper
- frontend-maven-plugin
- gson
- distlrucache
- retrofit1-okhttp3-client
- rxrelay
- rxreplayingshare
- junit4
- http-request
- lanterna
- java-apns
- mybatis-pagehelper
- algorithms
- fragmentargs
- mocha
- to mighty
- zt-zip
- gwt-cal
- Java-Chronicle
- maven-config-processor-plugin
- jboss-logmanager
- autoLoadCache
- profiler

CANDIDATE PROGRAMS

Selection criteria:

- **Popular:** At least 100 GitHub stars.
- **Compilable:** We only support the Maven build system. This is a technical restriction.
- **Timeout**: The static call graph must be executable in 10 hours.
- **Testable**: We only selected applications with a JUnit test suite.

- jam-tools
- bucket
- qart4j
- dubbokeeper
- frontend-maven-plugin
- gson
- distlrucache
- retrofit1-okhttp3-client
- rxrelay
- rxreplayingshare
- junit4
- http-request
- lanterna
- java-apns
- mybatis-pagehelper
- algorithms
- fragmentargs
- mocha
- to mighty
- zt-zip
- gwt-cal
- Java-Chronicle
- maven-config-processor-plugin
- jboss-logmanager
- autoLoadCache
- profiler

Average LOC: 14,729

RQ: REDUCTION

Transformation	Mean Size Reduction*
Method Removal	11.0%
Field Removal	1.0%
Class Collapsing	0.1%
Method Inlining	2.1%
All without checkpointing	14.2%
All with checkpointing	13.3%

*Includes both application and library code

RQ: PRESERVATION AND ROBUSTNESS

RQ: PRESERVATION AND ROBUSTNESS

We held back 20% of the each project’s test, and used the remaining 80% for the dynamic analysis.

RQ: PRESERVATION AND ROBUSTNESS

We held back 20% of the each project's test, and used the remaining 80% for the dynamic analysis.

JShrink, utilizing all transformations, without checkpointing, results in **81 test cases of 5213 failing (1.6%).**

RQ: PRESERVATION AND ROBUSTNESS

We held back 20% of the each project's test, and used the remaining 80% for the dynamic analysis.

JShrink, utilizing all transformations, without checkpointing, results in **81 test cases of 5213 failing (1.6%).**

Most of these tests are due to the class collapse; **75 of the 81 failures.**

RQ: PRESERVATION AND ROBUSTNESS

We held back 20% of the each project's test, and used the remaining 80% for the dynamic analysis.

JShrink, utilizing all transformations, without checkpointing, results in **81 test cases of 5213 failing (1.6%).**

Most of these tests are due to the class collapse; **75 of the 81 failures.**

All known bugs are due to limitations or bugs in Soot static analysis not giving information to account for all cases.

RQ: PRESERVATION AND ROBUSTNESS

We held back 20% of the each project's test, and used the remaining 80% for the dynamic analysis.

JShrink, utilizing all transformations, without checkpointing, results in **81 test cases of 5213 failing (1.6%).**

Most of these tests are due to the class collapse; **75 of the 81 failures.**

All known bugs are due to limitations or bugs in Soot static analysis not giving information to account for all cases.

No Tests fail when Checkpointing is enabled.

RQ: DYNAMIC IMPACT

Transformation	Test Failure Rate
JShrink Static	58.4%
JShrink Static + Dynamic	1.6%
JShrink Static + Dynamic + Checkpointing	0%

Most failures in 'JShrink Static' are due to JVM validation, 'NoClassDefFoundError', and 'ClassNotFoundException' errors.

DISCUSSION POINTS

We evaluated Tip et al.'s transformations in a modern context.

- Method removing is the most effective transformation, though all reduced bloat.
- Class Collapsing is the least effective, the most difficult to engineer, and the most error prone.

Reachability analysis in Java requires dynamic, as well as static, analysis

- Reachability in modern Java cannot be determined purely statically, dynamic analysis is needed.
- Without dynamic analysis, code debloating via reachability analysis can be dangerous.
- **Important Note: Dynamic analysis is as good as the test inputs!**

THANK YOU!

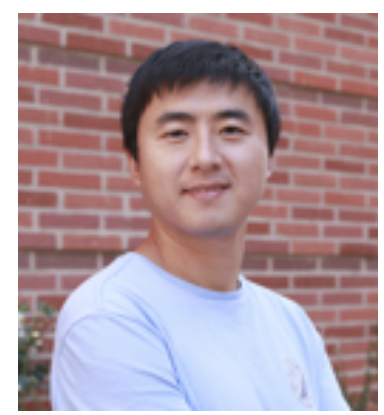
Artifact: https://doi.org/10.6084/m9.figshare.12435542

Research supported, in part by:

- ONR grant N0001418-1-2037
- NSF grants: CCF-1764077, CCF-1527923, CCF-1723773
- Intel CAPA
- Samsung
- The Alexander von Humboldt Foundation

Paper at: https://www.bobbybruce.net/assets/pdfs/publications/bruce-2020-jshrink.pdf






BOBBY R. BRUCE, TIANYI ZHANG, JASPREET ARORA, GUOQING HARRY XU, AND MIRYUNG KIM

PRESENTED BY BOBBY R. BRUCE