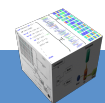




# Performance Debugging Study



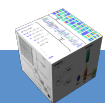
# Performance Debugging Study

---

The goal of this study is to find out **how software developers understand and fix performance bugs** in an unknown project with the help of a profiling tool and sketching. The study is carried out in three phases:

1. **Introduction** to our profiling tool and the underlying sampling approach
2. **Pair programming debugging session** with four "real world" performance bugs
3. **Questionnaire**

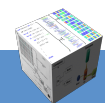
The whole study will be video and audio recorded. During the first two phases, we encourage the participants to **think aloud** and to **sketch** while understanding and fixing the bugs.



# Thinking Aloud

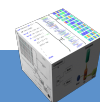
---

## Video Introduction



# Introduction to the Sampling Approach

---

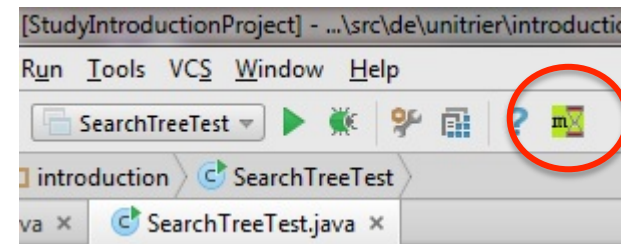
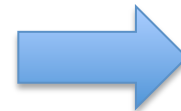
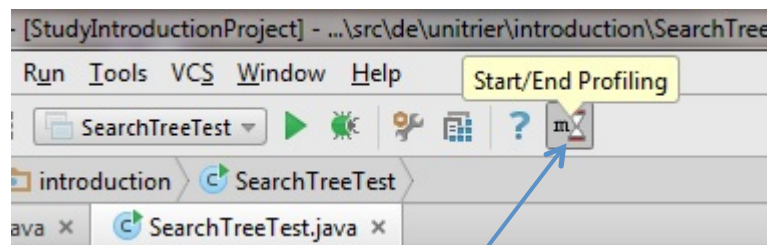


# **Video Introduction**

# IDE Integration

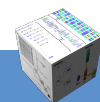
Our tool **integrates the sampling results in-situ in the source code editor** of the IntelliJ Java IDE. The following slides explain the visualizations and how to use the tool.

To enable the profiling for the next run of the program, just click on the corresponding button in the toolbar:

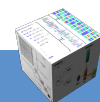
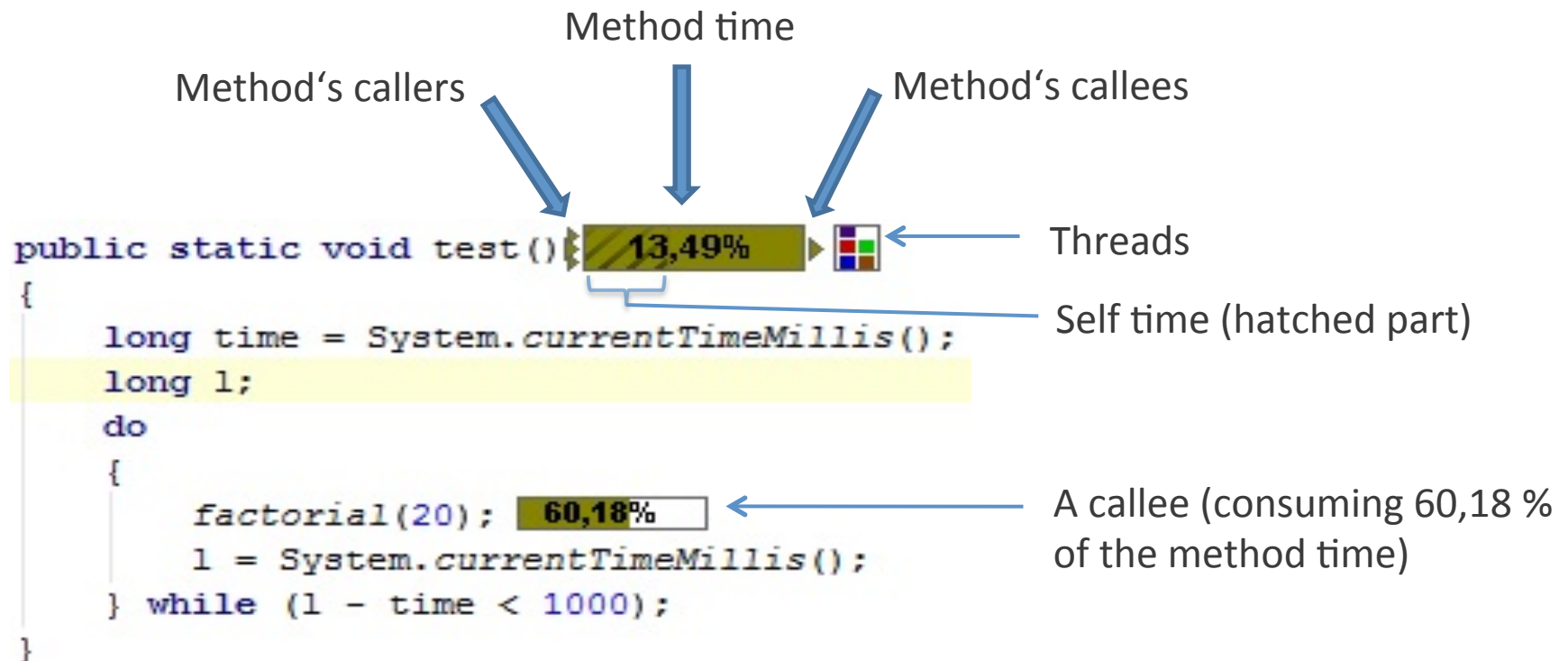


Turn on profiling with a click on the hourglass

If profiling is activated, the background of the hourglass turns green

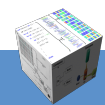
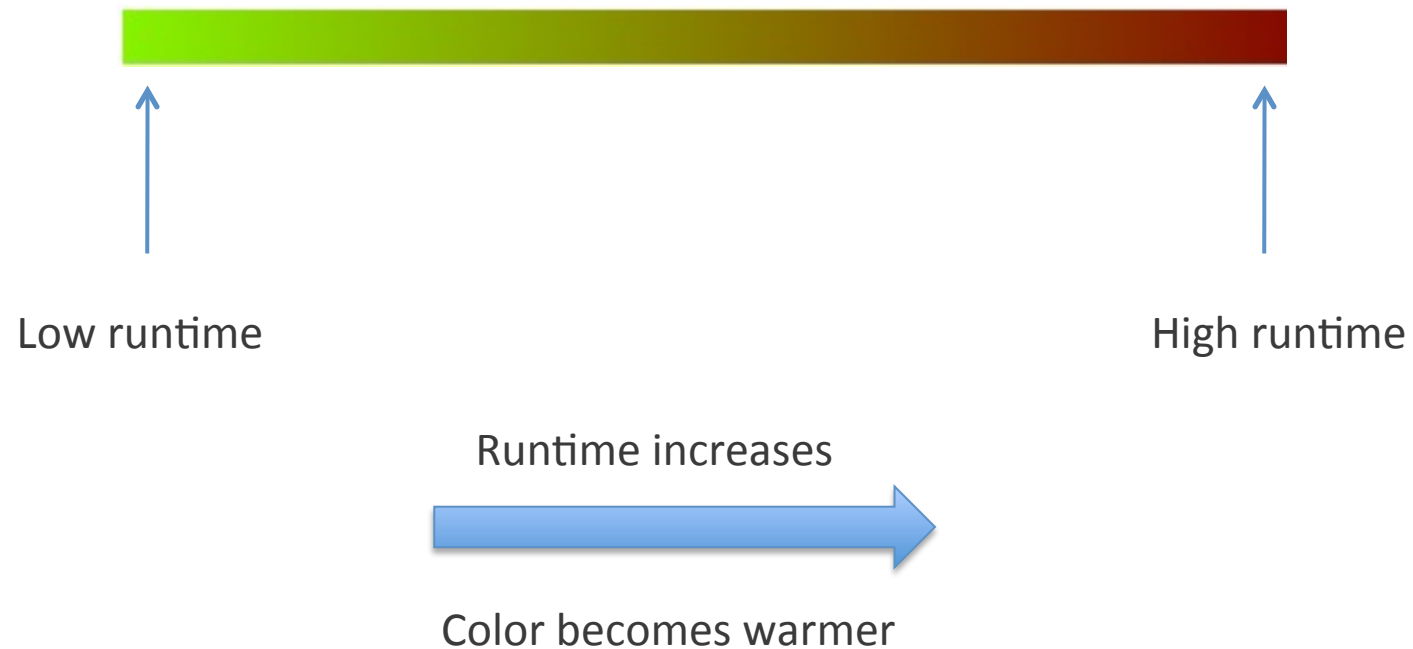


# Basic In-situ Visualization for Methods





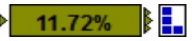

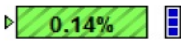
# Color Scale

The color of the visualization depends on the methods runtime

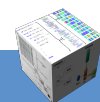




# Examples

method	comment
<code>main()</code> 	no callers, one callee, no considerable self time
<code>paint()</code> 	one caller, one callee, some self time
<code>lighting()</code> 	multiple callees, four threads of the same type
<code>hitObject()</code> 	high method time, multiple callers and callees, high self time
<code>cross()</code> 	low method time, no callees, only self time

Beck et al., „In situ understanding performance bottlenecks through visually augmented code, “  
In ICPC '13, May 2013, pp. 63-72



# Popup Window – Runtime Tab



Click on the visualization to open a popup window

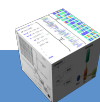
Determines the width of the hatched part

```
public static void test() {
    long time = System.currentTimeMillis();
    long l;
    do {
        factorial(20);
        l = System.currentTimeMillis();
    } while (l - time < 1000);
}
// MyThread extends Thread
```

Callers	Callees
34,54% de.unitrier.TestClass.runTest()	60,18% de.unitrier.TestClass.factorial(int)
34,10% de.unitrier.Container.doStuff()	
7,90% de.unitrier.MyThread4.run()	
7,90% de.unitrier.MyThread2.run()	
7,90% de.unitrier.MyThread.run()	
7,65% de.unitrier.MyThread3.run()	

The runtime tab shows a list of callers and callees

(doesn't have to be all callers and callees one can see in the code since it shows the result of a sampling)




# Popup Window – Runtime Tab – Navigation

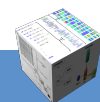
```
public static void test() {
    long time = System.cu
    long l;
    do
    {
        factorial(20);
        l = System.curren
    } while (l - time < 1
}

ss MyThread extends Thread
```

Callers	Callees
34,54% de.unitrier.TestClass.runTest()	60,18% de.unitrier.TestClass.factorial(int)
34,10% de.unitrier.Container.doStuff()	
7,90% de.unitrier.MyThread4.run()	
7,90% de.unitrier.MyThread2.run()	
7,90% de.unitrier.MyThread.run()	
7,65% de.unitrier.MyThread3.run()	

If you just navigated to a method through a click on a callee you can **go back** where you came from with a click on the corresponding caller within the Runtime tab of the current method. Or you use the IntelliJ build in navigation tool. 

**Navigate** to a caller/callee method with a click on its name.



# Popup Window – Threads Tab

The threads tab shows **all threads executing this method**, grouped by type (color)

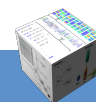
```
public static Set<Long> test() {
    Set<Long> set = new HashSe
    long time = System.current
    long l;
    do
    {
        set.add(factorial(20))
        l = System.currentTime
    } while (l - time < 500);
    return set;
}
```

test(): method time: 29,40% - self time: 15,09% (51,31% of method time)


Runtime Threads

- 0,42% Thread-9:166: de.unitrier.MyThread2
- 0,59% Thread-7:158: de.unitrier.MyThread2
- 0,47% Thread-5:160: de.unitrier.MyThread2
- 0,83% Thread-10:165: de.unitrier.MyThread2
- 0,55% Thread-12:168: de.unitrier.MyThread2
- 0,42% Thread-4:167: de.unitrier.MyThread2
- 0,54% Thread-3:161: de.unitrier.MyThread2
- 0,51% Thread-11:164: de.unitrier.MyThread2
- 0,55% Thread-6:159: de.unitrier.MyThread2
- 0,51% Thread-8:163: de.unitrier.MyThread2
- 10,41% Thread-0:147: java.lang.Thread
- 69,74% main:1: java.lang.Thread
- 4,47% Thread-16:177: de.unitrier.MyThread4
- 4,72% Thread-15:175: de.unitrier.MyThread4
- 1,49% Thread-13:170: de.unitrier.MyThread3
- 2,87% Thread-14:173: de.unitrier.MyThread3
- 0,47% Thread-2:149: de.unitrier.MyThread
- 0,47% Thread-1:150: de.unitrier.MyThread

Total number of threads: 18



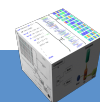
# Visualization for Classes

```
public class TestClass { 27,17% 
```

A class can have runtime as well. The **class runtime** tells in what percentage of the whole runtime arbitrary methods of that class were active. Its self time determines in what percentage methods of that class were executing instructions themselves (analogous to method self time).

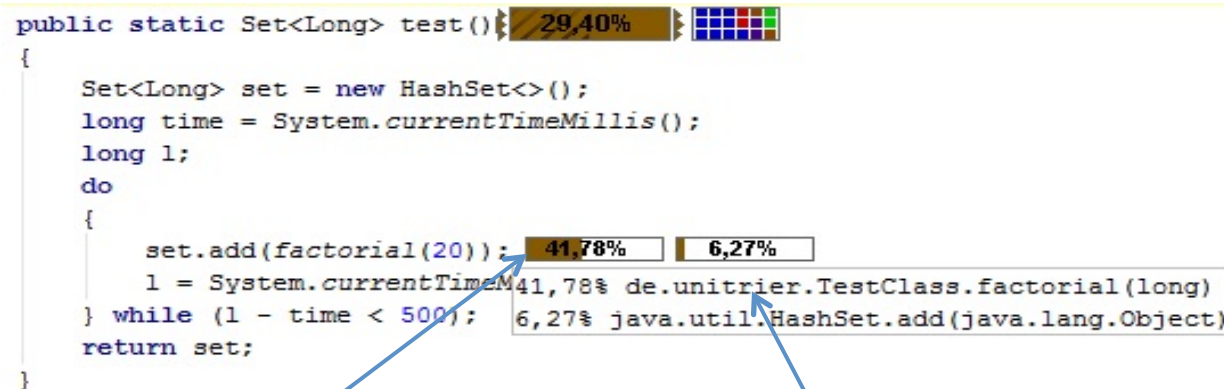
The **callers and callees of a class** are also classes. Thus the callers of a class are all classes of which a method called a method from the current class. And the callees of a class are all classes of which a method was called from a method of the current class.

The class visualization works analogous to a method visualization.



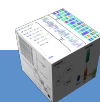
# The Callee Tooltip

```
public static Set<Long> test() { 29,40% }
{
    Set<Long> set = new HashSet<>();
    long time = System.currentTimeMillis();
    long l;
    do
    {
        set.add(factorial(20)); 41,78% 6,27%
        l = System.currentTimeMillis();
    } while (l - time < 500);
    return set;
}
```



With a click on the callee visualization, a text occurs showing the **name of the callee(s)** to which the visualization belongs.

**Navigate** to a called method with a click on its name within the callee tooltip.



# Overview

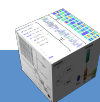
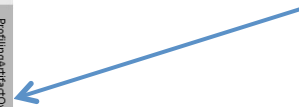
The screenshot displays the IntelliJ IDEA IDE interface. The main editor shows a Java file named 'TestClass.java' with the following code:

```
6 public class TestClass {
7     // ...
8     public static void main(String[] args) {
9         // ...
10        runTest();
11        B.runOnA();
12    }
13
14    static class C {
15        public static void runTest() {
16            // /MismatchedQueryAndUpdateOfCollection/
17            ArrayList<A> objects = new ArrayList<>();
18            for (int i = 0; i < 5; i++) {
19                {
20                    A a = new B();
21                    new C().runOnC();
22                    objects.add(a);
23                }
24            }
25            new Thread(new Runnable() {
26                @Override
27                public void run() {
28                    B.runOnA();
29                }
30            }).start();
31        }
32    }
33}
```

The right-hand side of the IDE shows the 'ProfilingArtifactOverview' window. It contains a table of artifacts and their runtime percentages:

Runtime	Artifact's name
26.79%	de.unitrier.TestClass.test ()
26.63%	java.lang.Thread.run ()
23.39%	com.intellij.rt.execution.application.Applet.run ()
23.28%	java.net.ServerSocketImpl.accept (java.net.ServerSocketImpl)
23.28%	java.net.ServerSocket.accept ()
23.22%	java.net.PlainSocketImpl.accept (java.net.PlainSocketImpl)
23.22%	java.net.DualStackPlainSocketImpl.socketAccept ()
23.22%	java.net.AbstractPlainSocketImpl.accept (java.net.PlainSocketImpl)
23.22%	java.net.DualStackPlainSocketImpl.accept0 ()
23.11%	com.intellij.rt.execution.application.Applet.run ()
22.88%	java.lang.reflect.Method.invoke (java.lang.reflect.Method)
22.88%	sun.reflect.DelegatingMethodAccessorImpl.invoke ()
22.88%	sun.reflect.NativeMethodAccessorImpl.invoke ()
22.88%	de.unitrier.TestClass.main (java.lang.String[])
22.88%	sun.reflect.NativeMethodAccessorImpl.invoke ()
19.67%	de.unitrier.TestClass.runTest ()
19.15%	de.unitrier.TestClass.C.runOnC ()

Open the overview with a click on the the corresponding tool button you can find on the right border of the IntelliJ window.





# Overview

Switch to this tab to show the runtime list aggregated for classes

ProfilingArtifactOverview

Filter artifacts (separate with comma)

include

exclude

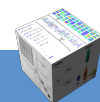
Apply

Methods Classes

Runtime	Artifact's name
27,39%	de.unitrier.TestClass.test()
26,27%	java.lang.Thread.run()
23,16%	com.intellij.rt.execution.application.AppMain\$1.run()
22,96%	java.net.PlainSocketImpl.accept(java.net.SocketImpl)
22,96%	java.net.ServerSocket.implAccept(java.net.Socket)
22,96%	java.net.ServerSocket.accept()
22,96%	java.net.DualStackPlainSocketImpl.socketAccept(java.net.SocketImpl)
22,96%	java.net.AbstractPlainSocketImpl.accept(java.net.SocketImpl)
22,96%	java.net.DualStackPlainSocketImpl.accept0(int, java.net.InetSocket)
22,32%	com.intellij.rt.execution.application.AppMain.main(java.lang.String[])
22,25%	java.lang.reflect.Method.invoke(java.lang.Object, java.lang.Object[])
22,25%	sun.reflect.DelegatingMethodAccessorImpl.invoke(java.lang.Object, java.lang.Object[])
22,25%	sun.reflect.NativeMethodAccessorImpl.invoke(java.lang.Object, java.lang.Object[])
22,25%	de.unitrier.TestClass.main(java.lang.String[])
22,25%	sun.reflect.NativeMethodAccessorImpl.invoke0(java.lang.reflect.Method, java.lang.Object, java.lang.Object[])
19,15%	de.unitrier.TestClass.runTest()
18,77%	de.unitrier.TestClass\$C.runOnC()
11,68%	de.unitrier.TestClass.factorial(long)
6,19%	de.unitrier.A.runOnA()
3,11%	de.unitrier.TestClass\$1.run()
1,65%	de.unitrier.MyThread4.run()
0,58%	de.unitrier.MyThread2.run()
0,52%	java.lang.ClassLoader.loadClass(java.lang.String)
0,52%	java.lang.ClassLoader.loadClass(java.lang.String, boolean)
0,52%	sun.misc.Launcher\$AppClassLoader.loadClass(java.lang.String, boolean)
0,42%	java.net.URLClassLoader\$1.run()
0,42%	java.net.URLClassLoader.findClass(java.lang.String)
0,42%	java.security.AccessController.doPrivileged(java.security.PrivilegedAction)
0,34%	sun.launcher.LauncherHelper.checkAndLoadMain(boolean, int, java.lang.String)
0,32%	de.unitrier.A.A()

Apply filters

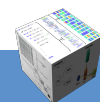
Navigate to a method or class through a click on its name in the overview.





# Hands-on Example

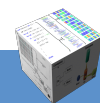
---



# Introductory Tasks

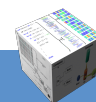
## Open project "Study Introduction Project" in IntelliJ:

1. Open the file `SearchTreePerformanceTest.java`
2. Activate the profiling tool and run the project using run configuration "**SearchTreePerformanceTest**".
3. Briefly tell what the performance test actually tests and describe in what context it will run (e.g. data structure, variables, input, methods, measurements, comments, ...)
4. Which two methods consume the most runtime within the `main` method of the class `SearchTreePerformanceTest`?
5. Try to follow the callees of the methods found in **Task 4** till you end up in a method which has no callee. What observations did you make?
6. Why is the self time of the methods you ended up in **Task 5** so high in the context of the performance test? You may sketch while finding the answer.
7. Explain how the method `buildTree` works with the help of the profiling visualization. Why is this method faster than just inserting every single node?
8. Try to explain the runtime difference for the performance test at hand.



# Debugging Session

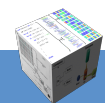
---



# Performance Bug 1

Open project "Apache Commons Collections" in IntelliJ and open file "PerformanceTest\_01.java" of the package "performancetests".  
To use the profiling tool, please select run configuration "PerformanceTest\_01".

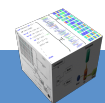
1. Switch driver and navigator.
2. Look at the given performance test and try to understand the difference between the two test cases.
3. Verify the performance bug in the method `retainAll` with the help of the profiling tool and try to understand it.
4. Propose a solution/fix for the bug. Please create a sketch describing the problem and your solution.
5. Implement your bug fix.
6. Verify your fix using the profiling tool.



# Questions - Tool

---

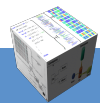
1. What information from the profiling tool or other parts of the IDE was required to understand the performance bug?
2. Do you think that the in-situ visualization of the profiling data was beneficial compared to a list representation?



# Questions - Sketches

---

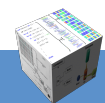
1. What information from the profiling tool or other parts of the IDE were important for creating your sketches?
2. How would you characterize the purpose and value of your sketches made during the debugging session?
3. Do you think that your sketch could help to explain the performance bug to someone else?



# Performance Bug 2

Open project "Apache Commons Collections" in IntelliJ and open file "PerformanceTest\_02.java" of the package "performancetests".  
To use the profiling tool, please select run configuration "PerformanceTest\_02".

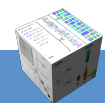
1. Switch driver and navigator.
2. Look at the given performance test and try to understand the difference between the two test cases.
3. Verify the performance bug in the method `retainAll` with the help of the profiling tool and try to understand it.
4. Propose a solution/fix for the bug. Please create a sketch describing the problem and your solution.
5. Implement your bug fix.
6. Verify your fix using the profiling tool.



# Questions - Tool

---

1. What information from the profiling tool or other parts of the IDE was required to understand the performance bug?
2. Do you think that the in-situ visualization of the profiling data was beneficial compared to a list representation?

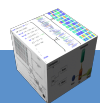




# Questions - Sketches

---

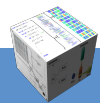
1. What information from the profiling tool or other parts of the IDE were important for creating your sketches?
2. How would you characterize the purpose and value of your sketches made during the debugging session?
3. Do you think that your sketch could help to explain the performance bug to someone else?



# Performance Bug 3

Open project "Apache Commons Collections" in IntelliJ and open file "PerformanceTest\_03.java" of the package "performancetests".  
To use the profiling tool, please select run configuration "PerformanceTest\_03".

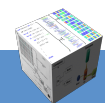
1. Switch driver and navigator.
2. Look at the given performance test and try to understand it.
3. Verify the performance bug in the method `containsAll` with the help of the profiling tool and try to understand it.
4. Propose a solution/fix for the bug. Please create a sketch describing the problem and your solution.
5. Implement your bug fix.
6. Verify your fix using the profiling tool.



# Questions - Tool

---

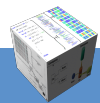
1. What information from the profiling tool or other parts of the IDE was required to understand the performance bug?
2. Do you think that the in-situ visualization of the profiling data was beneficial compared to a list representation?



# Questions - Sketches

---

1. What information from the profiling tool or other parts of the IDE were important for creating your sketches?
2. How would you characterize the purpose and value of your sketches made during the debugging session?
3. Do you think that your sketch could help to explain the performance bug to someone else?

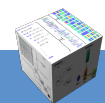


# Performance Bug 4

Open project "Guava Libs" in IntelliJ and open file "PerformanceTest\_04.java" of the package "performancetests".

To use the profiling tool, please select run configuration "PerformanceTest\_04".

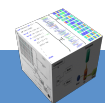
1. Switch driver and navigator.
2. Look at the given performance test and try to understand the difference between the two test cases.
3. Verify the performance bug in the method `contains` which is called on variable `immutableSet` with the help of the profiling tool and try to understand it.
4. Propose a solution/fix for the bug. Please create a sketch describing the problem and your solution.
5. Implement your bug fix.
6. Verify your fix using the profiling tool.



# Questions - Tool

---

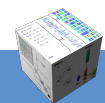
1. What information from the profiling tool or other parts of the IDE was required to understand the performance bug?
2. Do you think that the in-situ visualization of the profiling data was beneficial compared to list representation?



# Questions - Sketches

---

1. What information from the profiling tool or other parts of the IDE were important for creating your sketches?
2. How would you characterize the purpose and value of your sketches made during the debugging session?
3. Do you think that your sketch could help to explain the performance bug to someone else?



# Final Questionnaire

---

(one for each participant)

