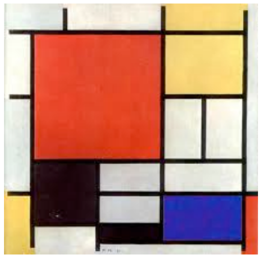
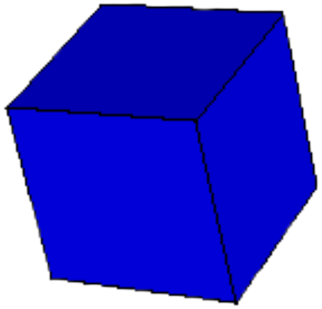
Profiler Zoo

Alexandre Bergel
abergel@dcc.uchile.cl
Pleiad lab, UChile, Chile

moose



Pharo



★☆☆☆

Test coverage

Problem:

Traditional code coverage tools have a binary view of the world

Why the problem is important:

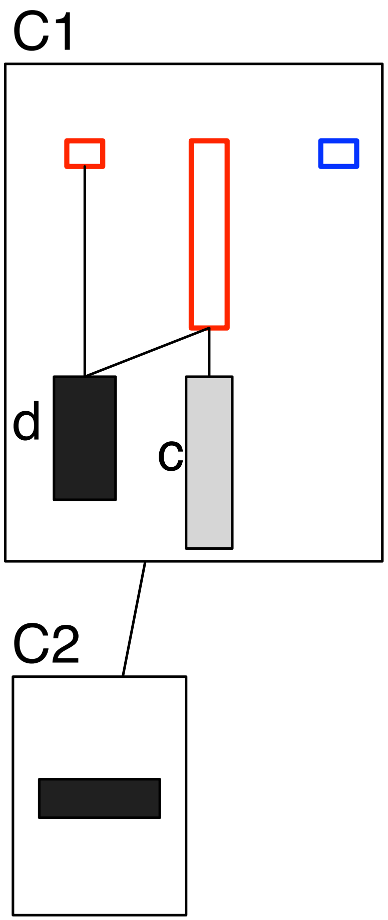
Which method should you test first in order to increase the coverage?

Is my code *well* covered or not?

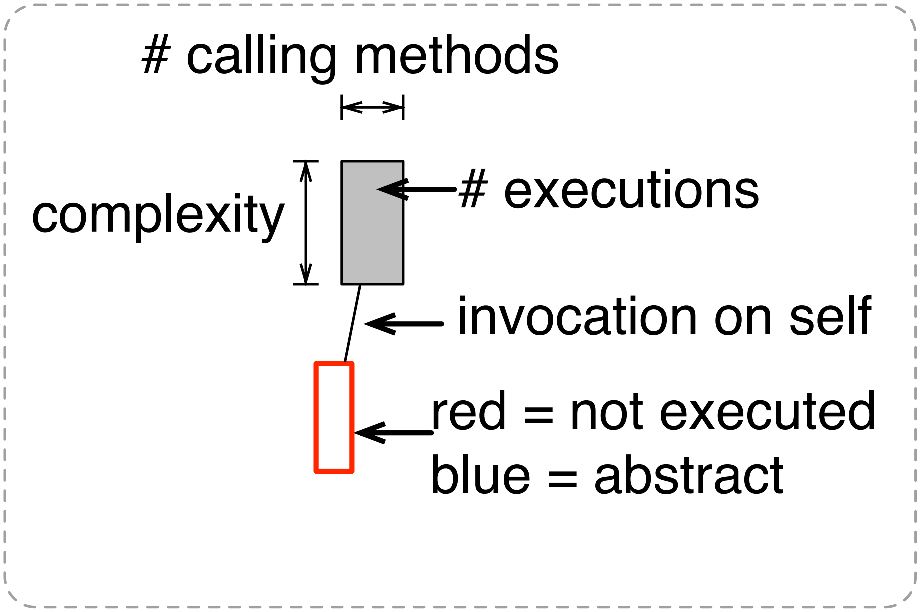
Solution:

An intuitive visual representation of a qualitative assessment of the coverage

Test blueprint



Legend for methods (inner boxes)


# calling methods

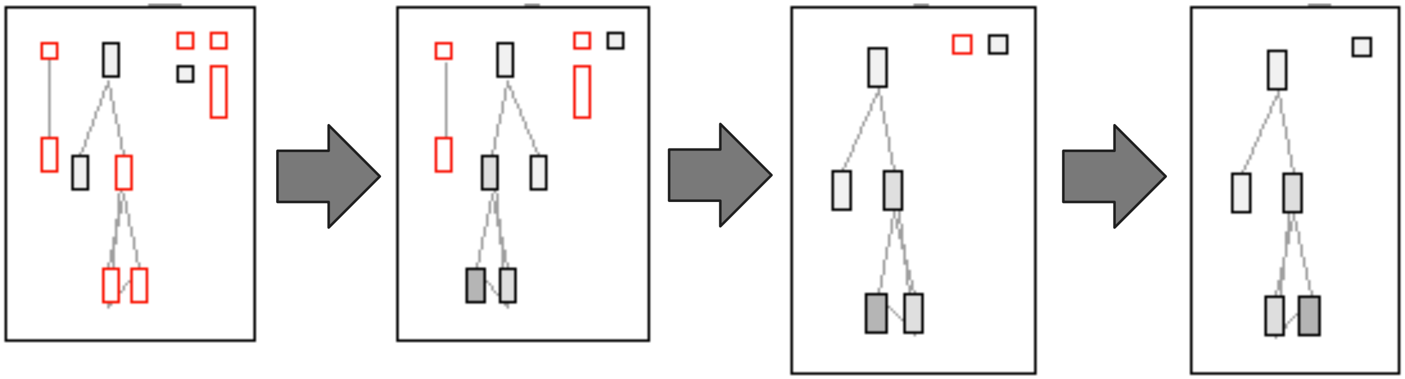
complexity

executions

invocation on self

red = not executed
blue = abstract

Successive improvement



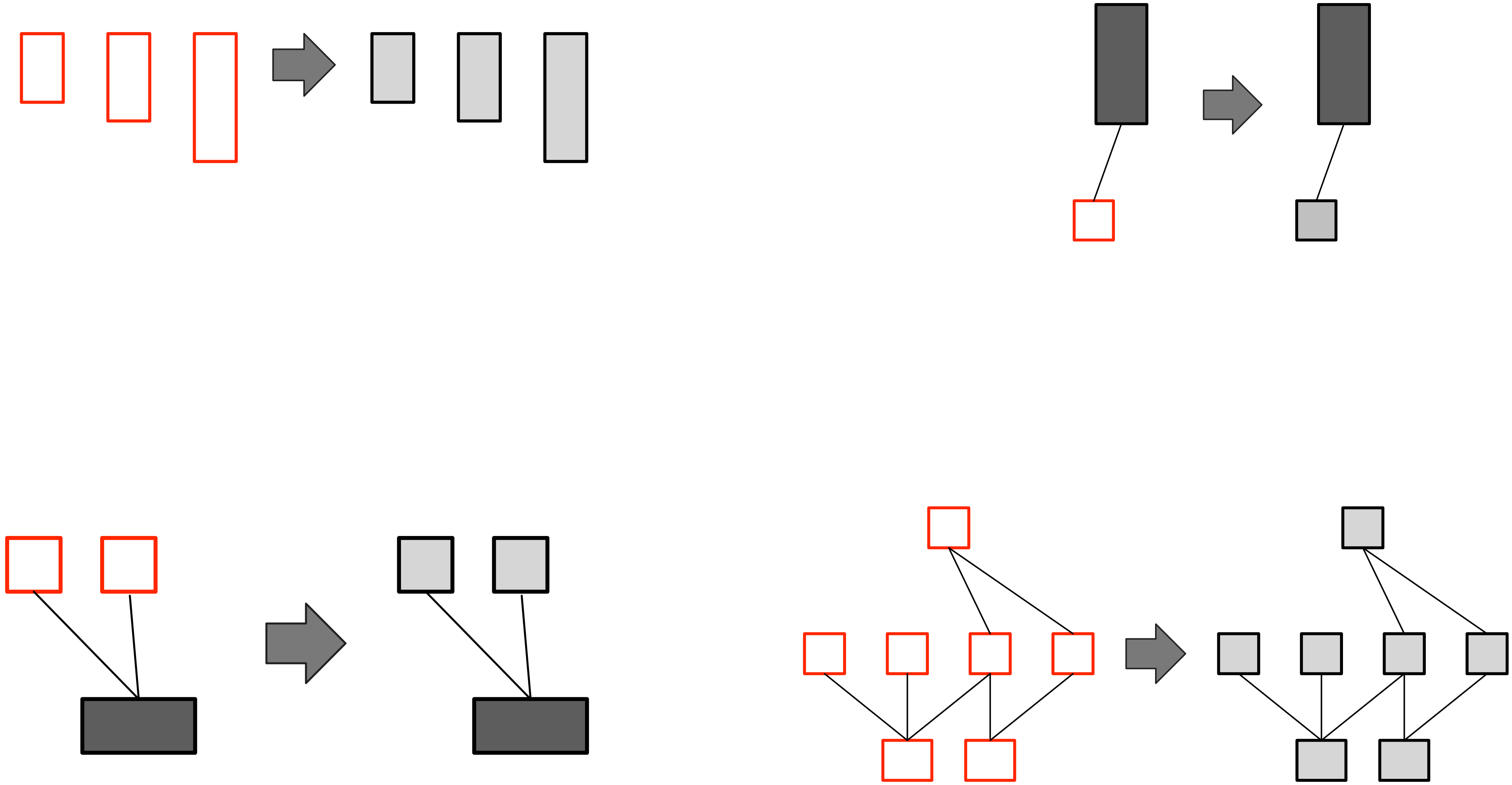
Version 2.2
27.27%

Version 2.3
54.54%

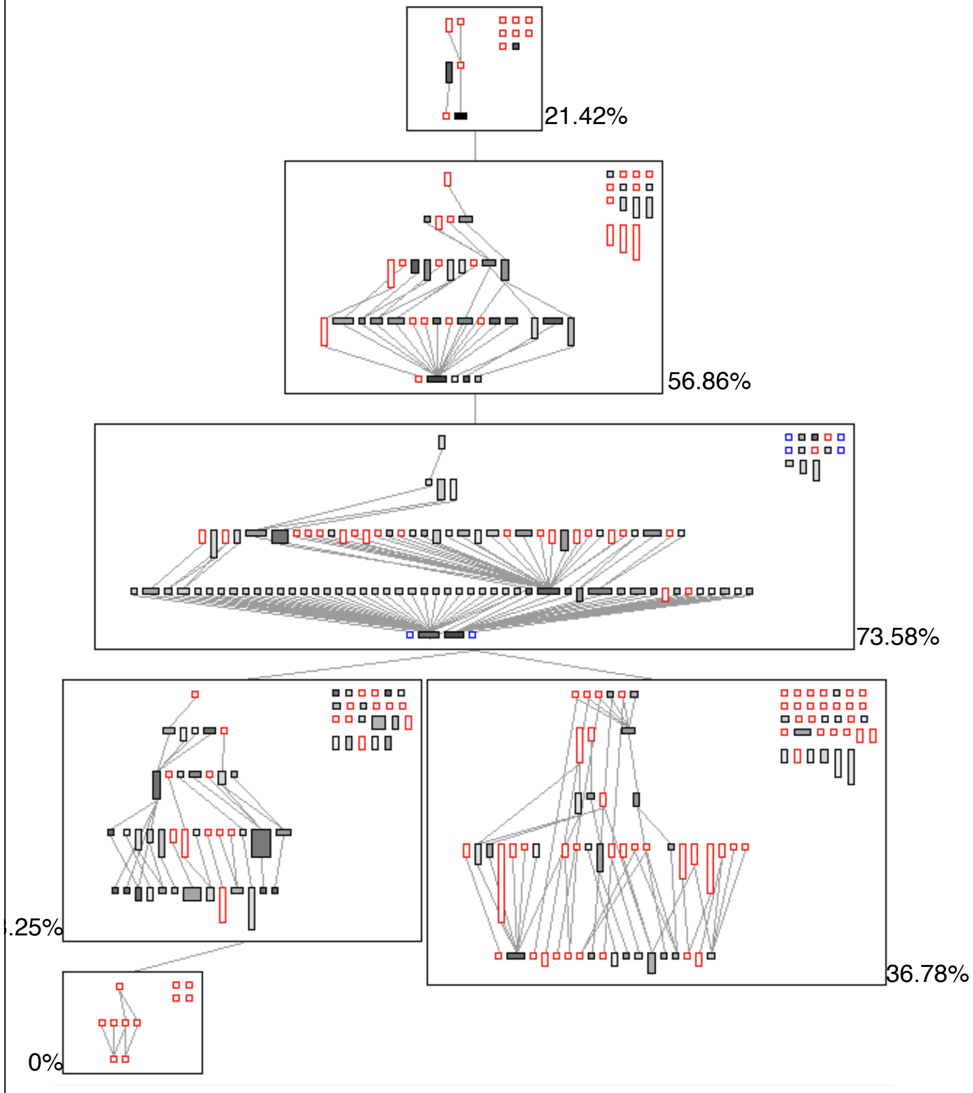
Version 2.4
87.71%

Version 2.5
100%

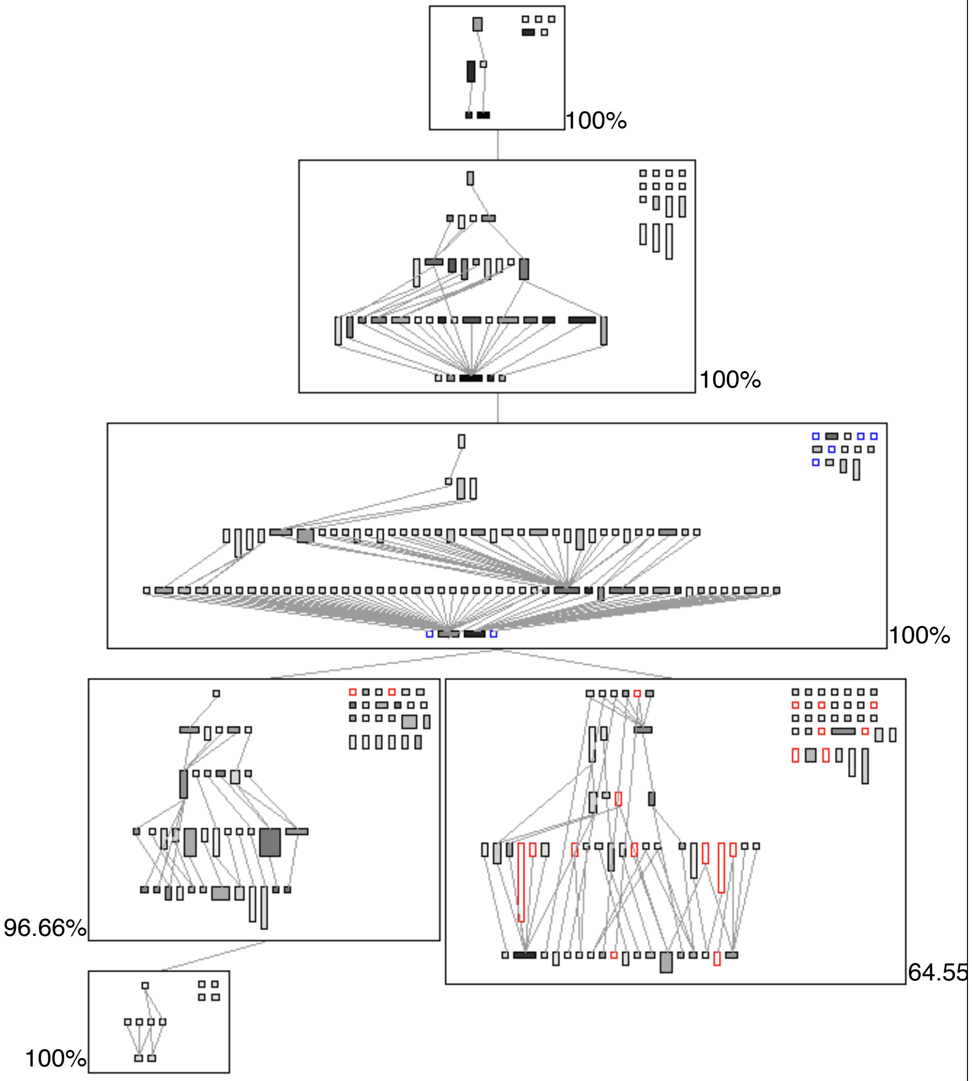
4 patterns



Moose-Test-Core.13
Moose-Core.313

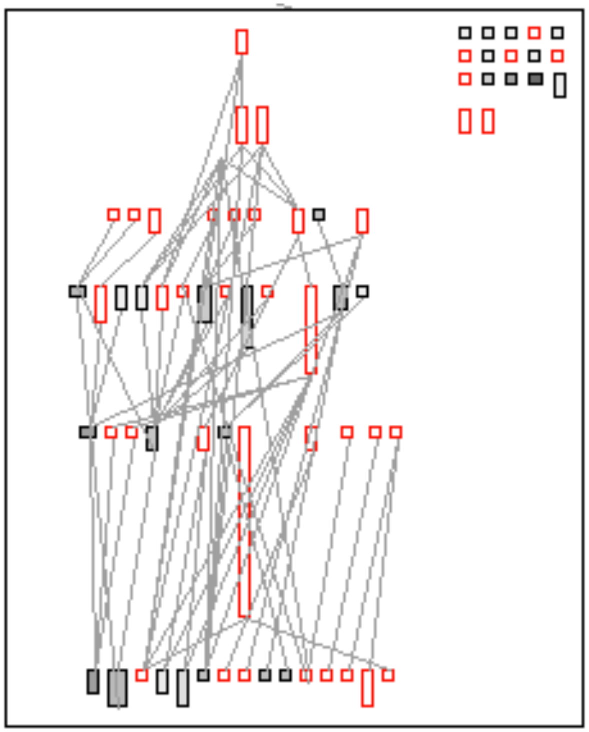


Moose-Test-Core.48
Moose-Core.326

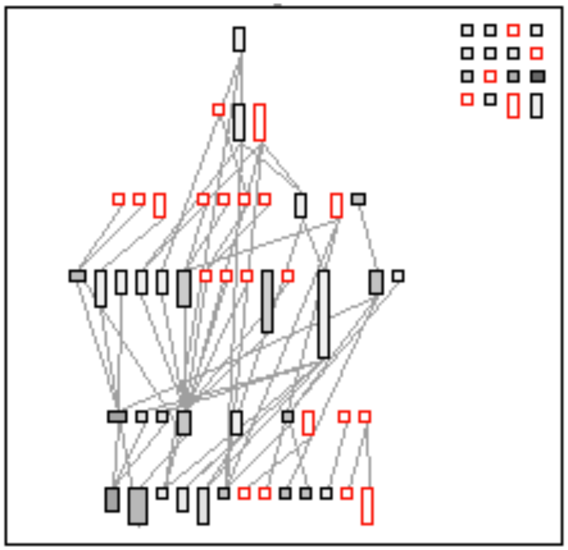


Reducing code complexity

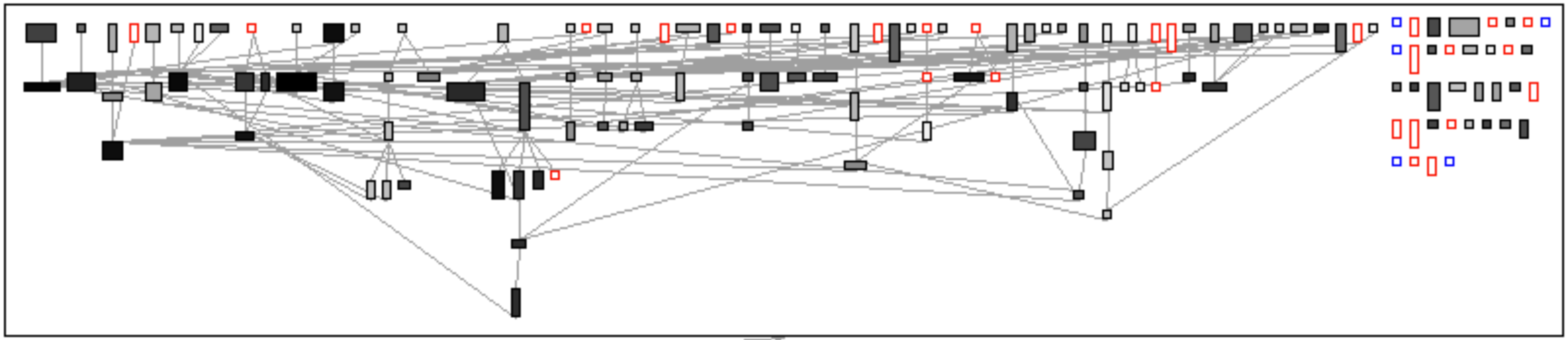
Version 1.58.1
Coverage: 40.57%



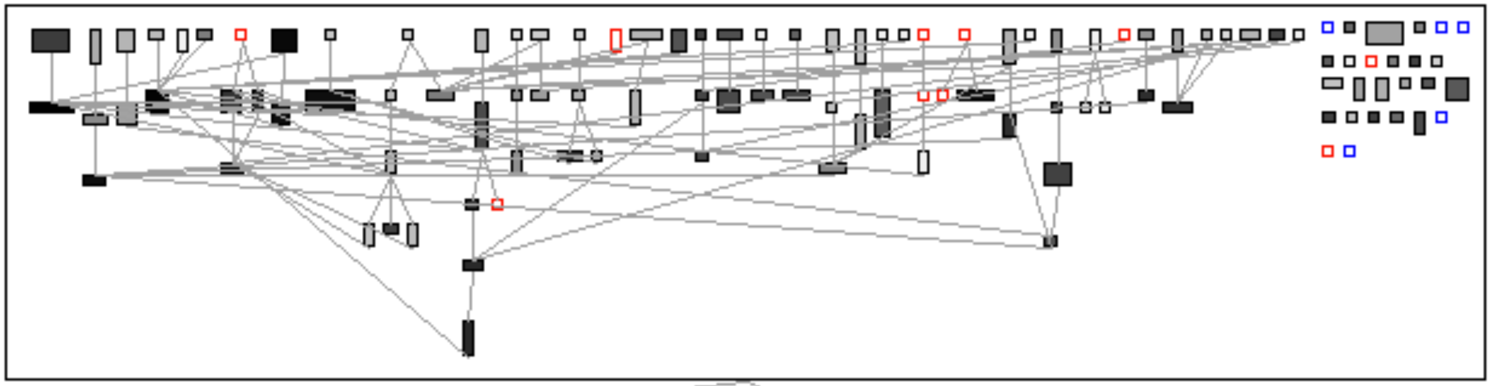
Version 1.58.9
Coverage: 60.60%



Reducing code complexity



Version 2.10



Version 2.17

★★☆☆

Execution profiling

Problem:

Traditional code profilers are driven by the method stack, discarding the notion of sending messages

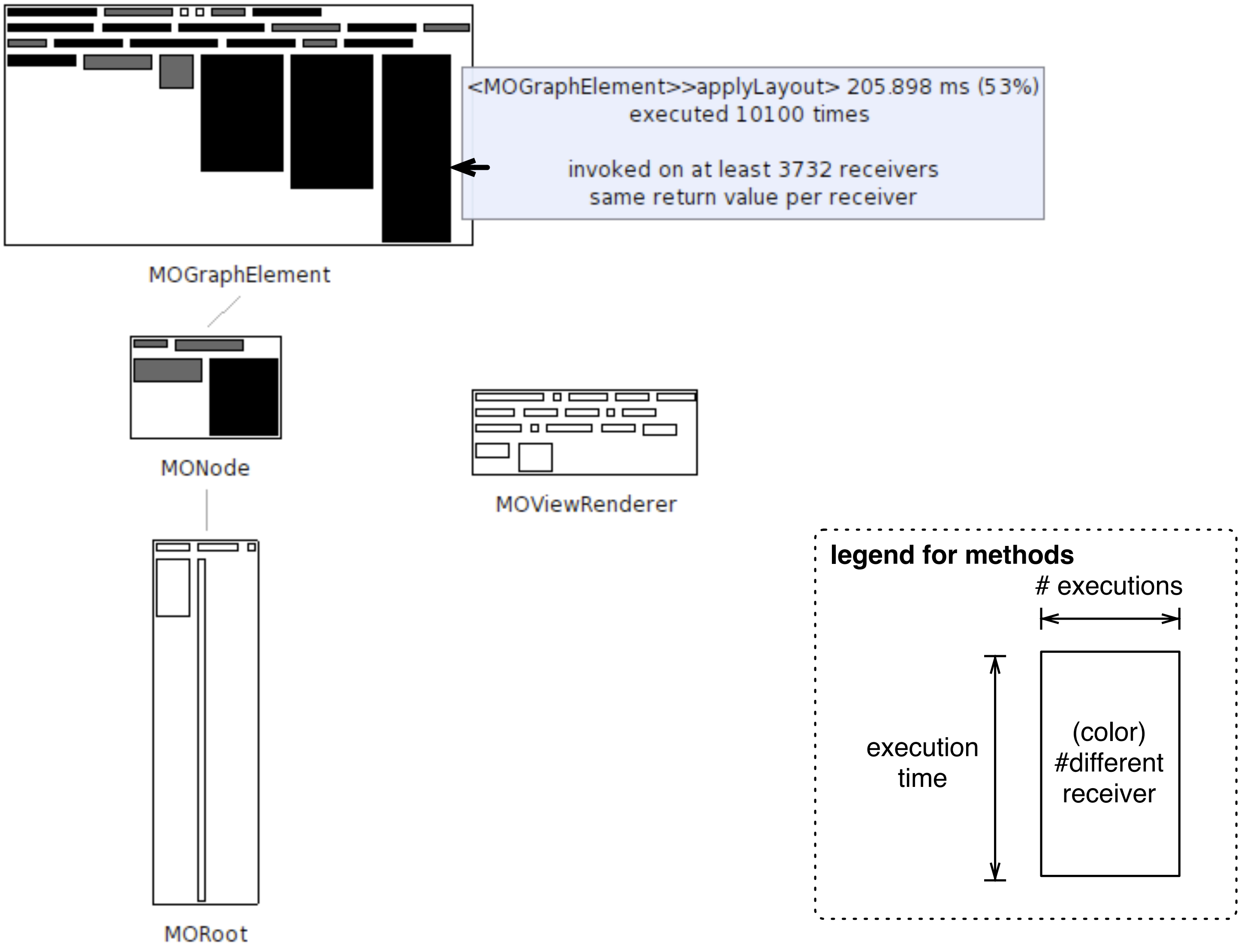
Why the problem is important:

How to answer to “Is there a slow function that is called too often?”

Solution:

An intuitive visual representation of the execution that visually compare the time spent and the number of executions

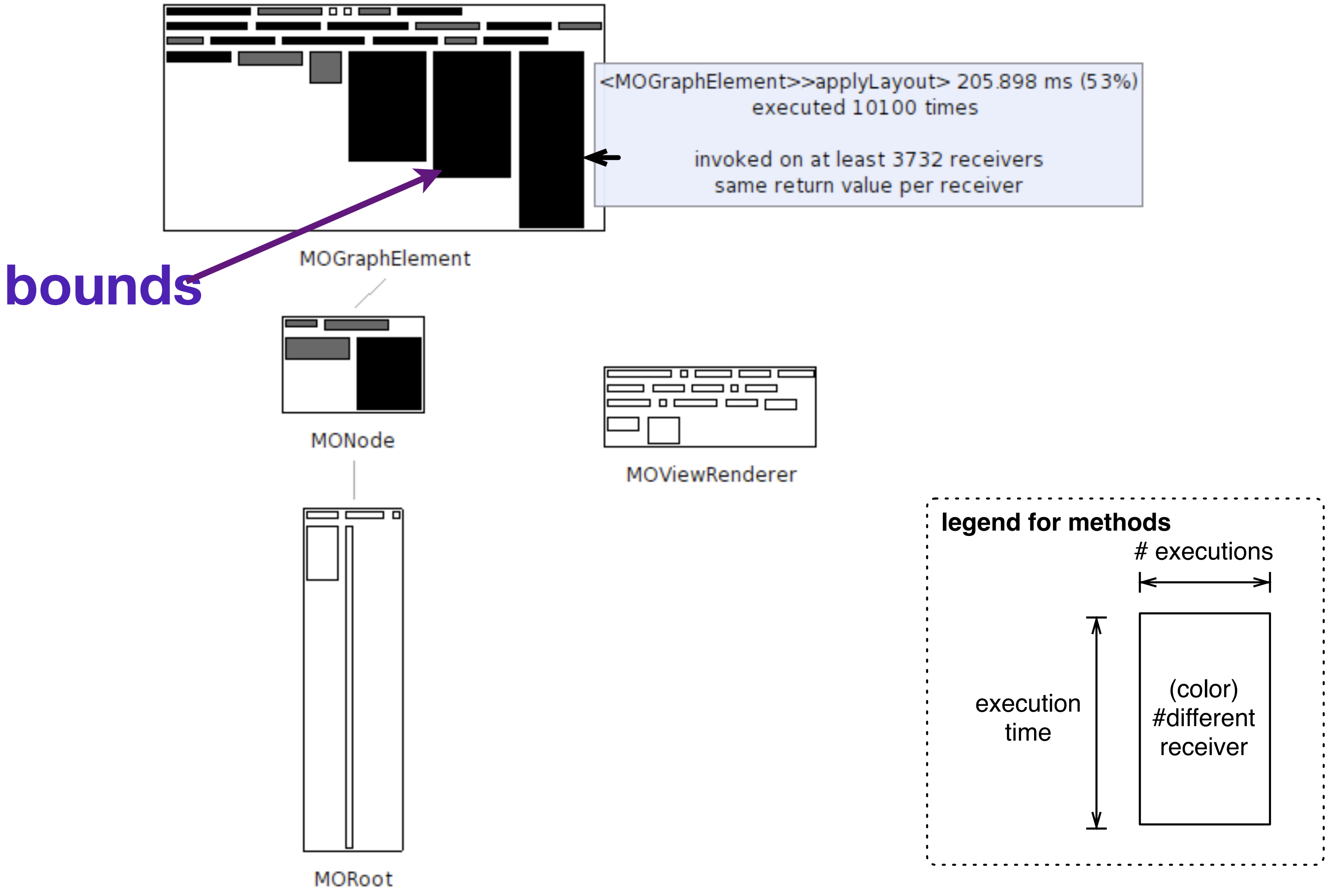
Structural profiling blueprint



<MOGraphElement>>applyLayout> 205.898 ms (53%)
executed 10100 times

invoked on at least 3732 receivers
same return value per receiver

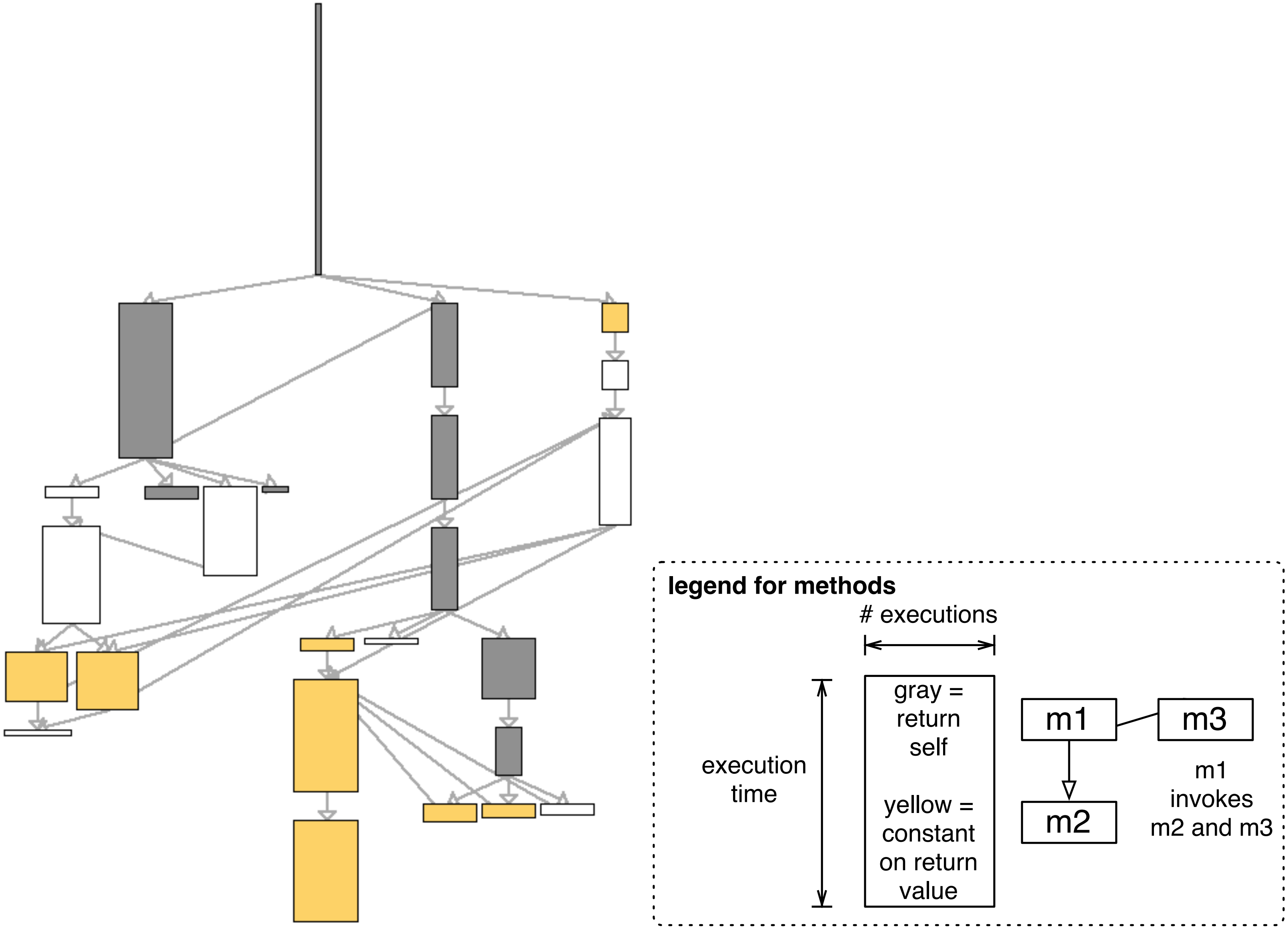
Structural profiling blueprint



<MOGraphElement>>applyLayout> 205.898 ms (53%)
executed 10100 times

invoked on at least 3732 receivers
same return value per receiver

Behavioral profiling blueprint



legend for methods

# executions

execution time

gray = return self

yellow = constant on return value

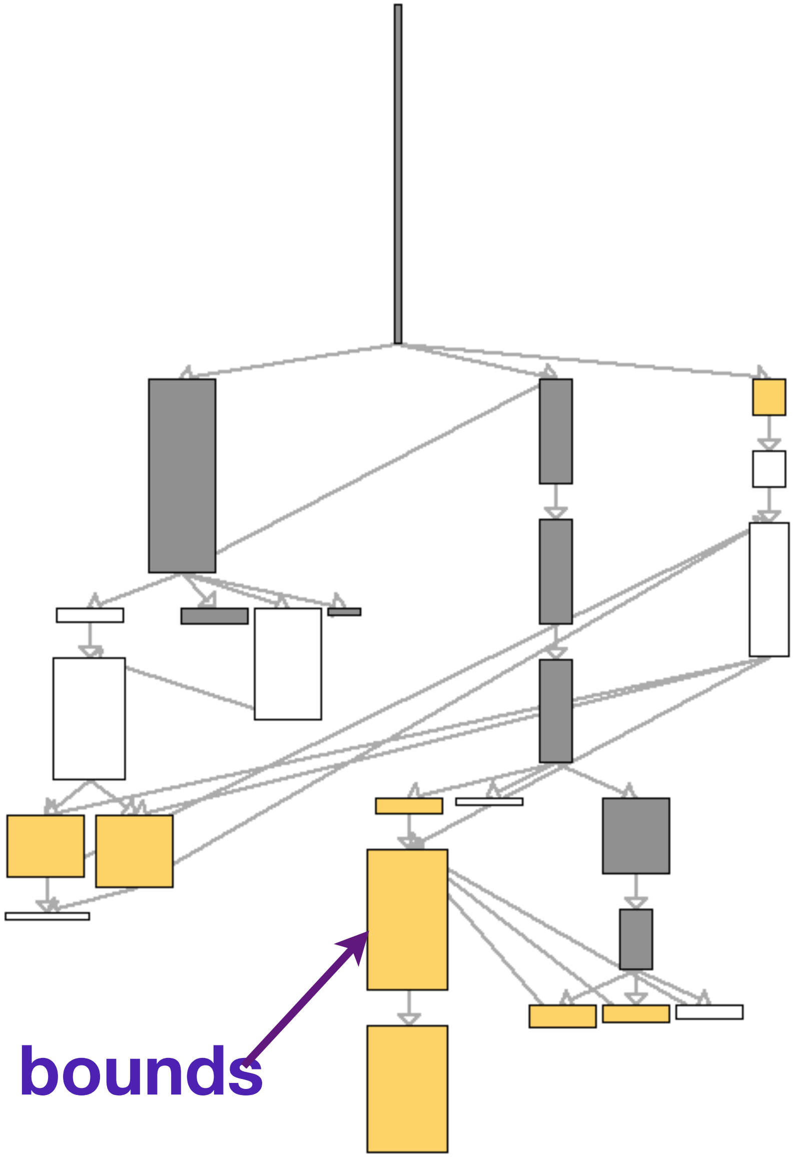
m1

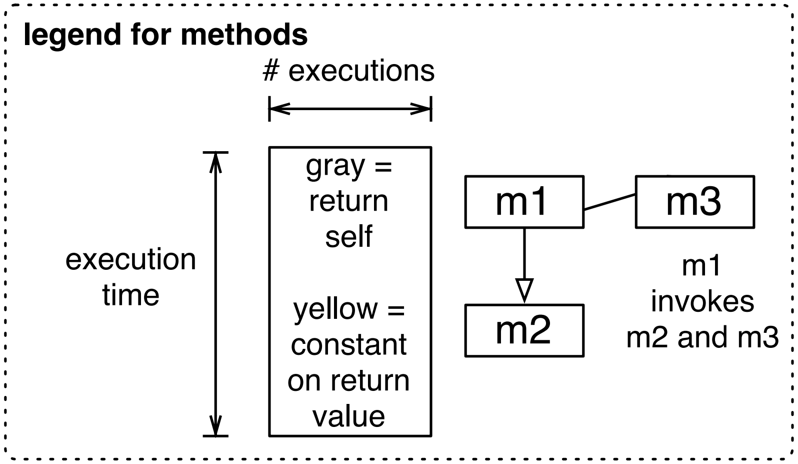
m2

m3

m1 invokes m2 and m3

Behavioral profiling blueprint



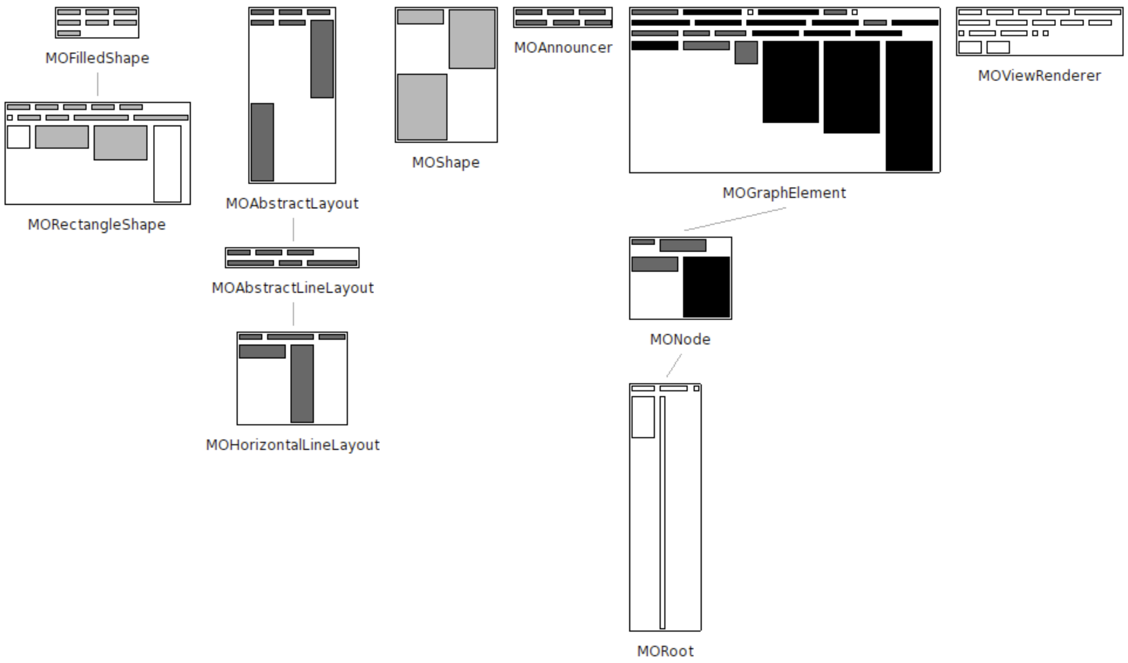


Code of the bounds method

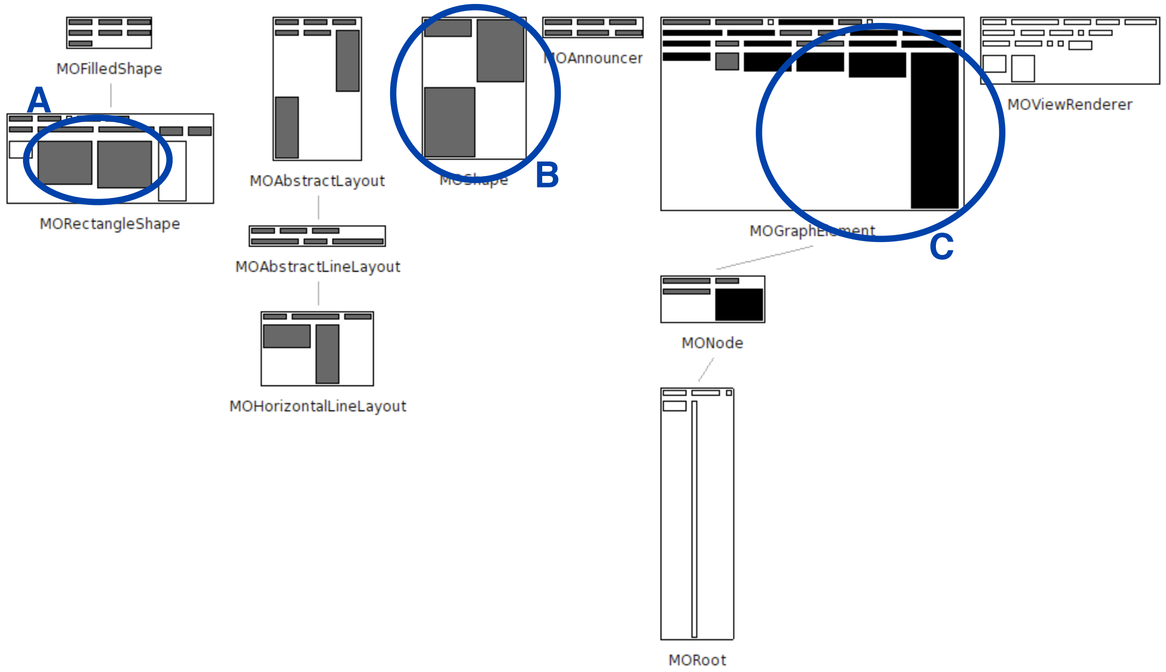
```
MOGraphElement>>bounds
 "Answer the bounds of the receiver."

 | basicBounds |

 self shapeBoundsAt: self shape ifPresent: [ :b | ^ b ].

 basicBounds := shape computeBoundsFor: self.
 self shapeBoundsAt: self shape put: basicBounds.

 ^ basicBounds
```

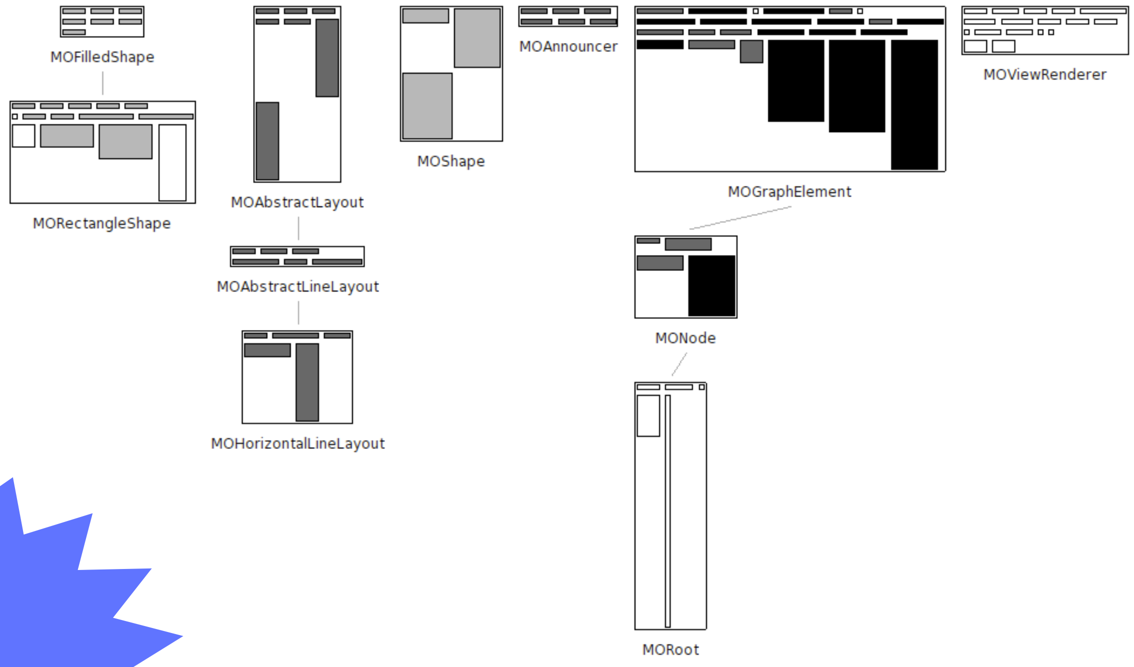
Memoizing

```
MOGraphElement>>bounds
 "Answer the bounds of the receiver."

 | basicBounds |
 boundsCache ifNotNil: [ ^ boundsCache ].
 self shapeBoundsAt: self shape ifPresent: [ :b | ^ b ].

 basicBounds := shape computeBoundsFor: self.
 self shapeBoundsAt: self shape put: basicBounds.

 ^ boundsCache := basicBounds
```



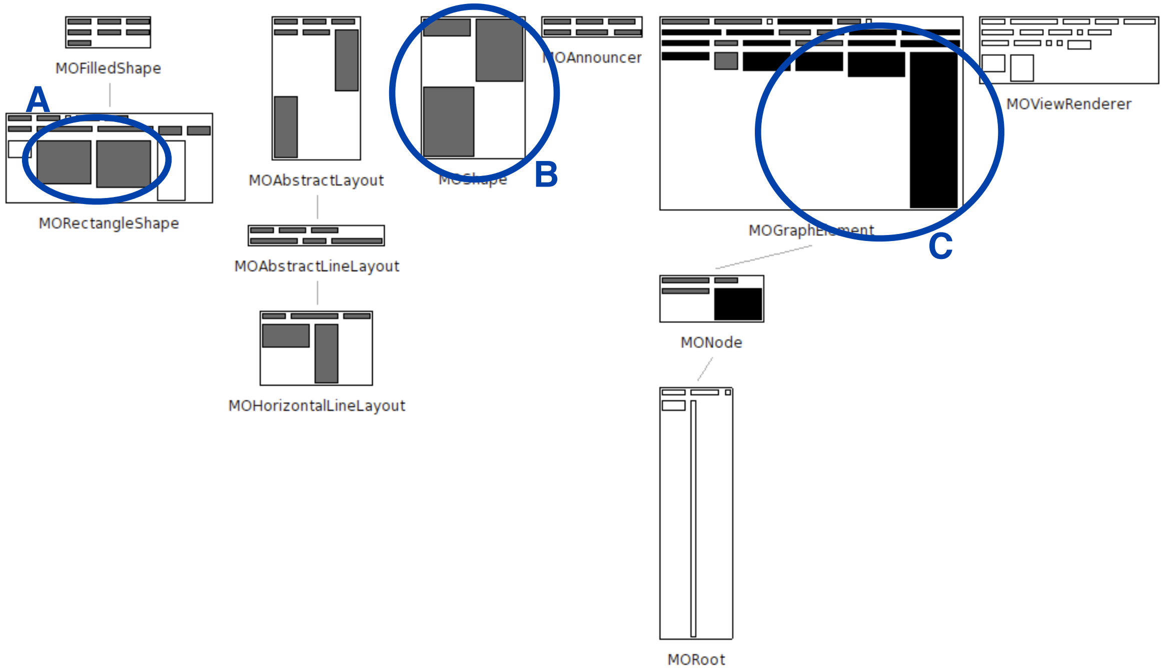
Upgrading
MOGraphElement>>bounds

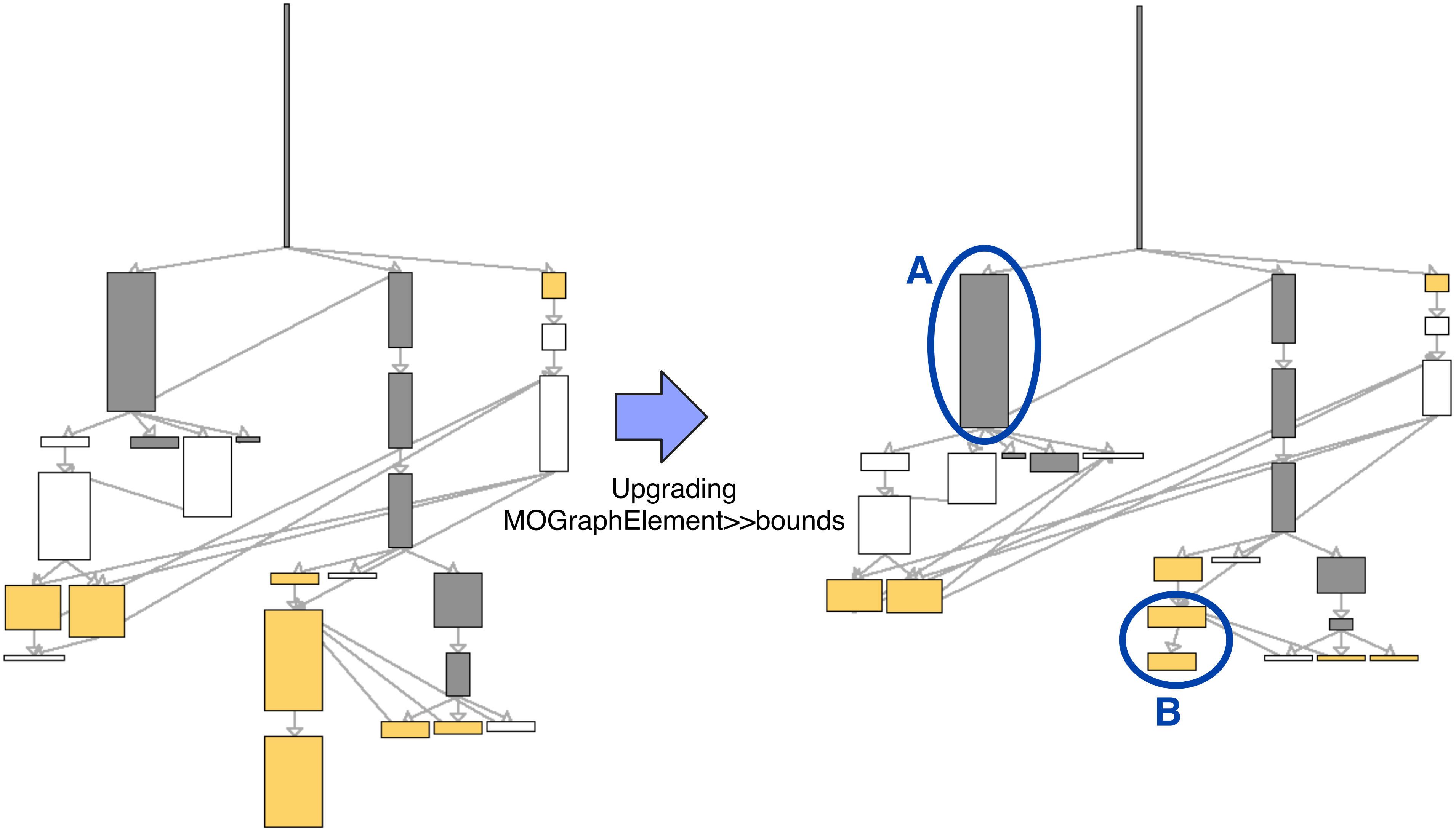




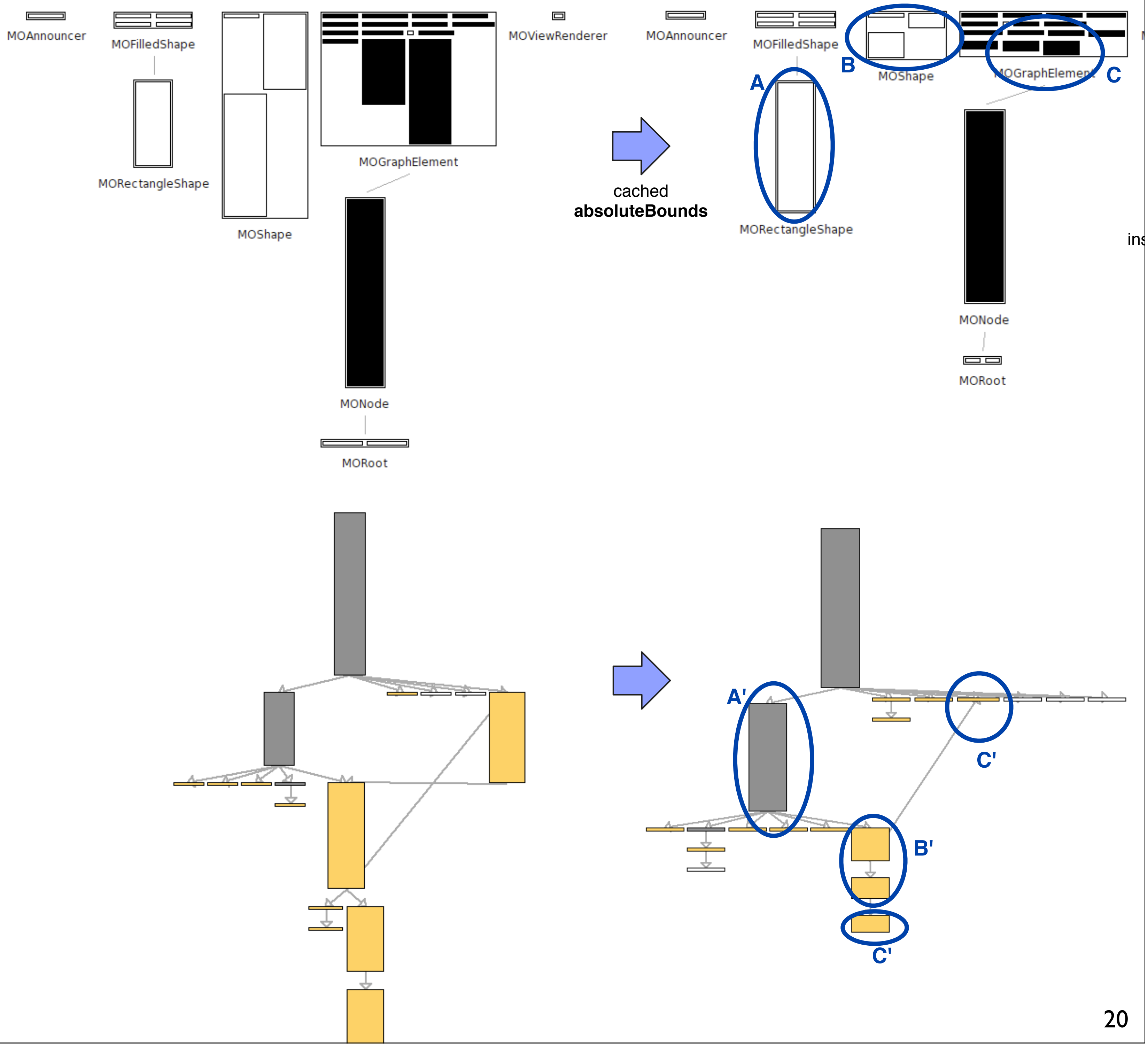
43% speedup

Upgrading MOGraphElement>>bounds





Upgrading
MOGraphElement>>bounds



cached
absoluteBounds

★★★☆

Measuring execution time

Problem:

Traditional code profilers sample program execution at a regular interval. This is inaccurate, non portable and non deterministic

Why the problem is important:

all profiles are meaningless if I get a new laptop or change the virtual machine

cannot profile short execution time

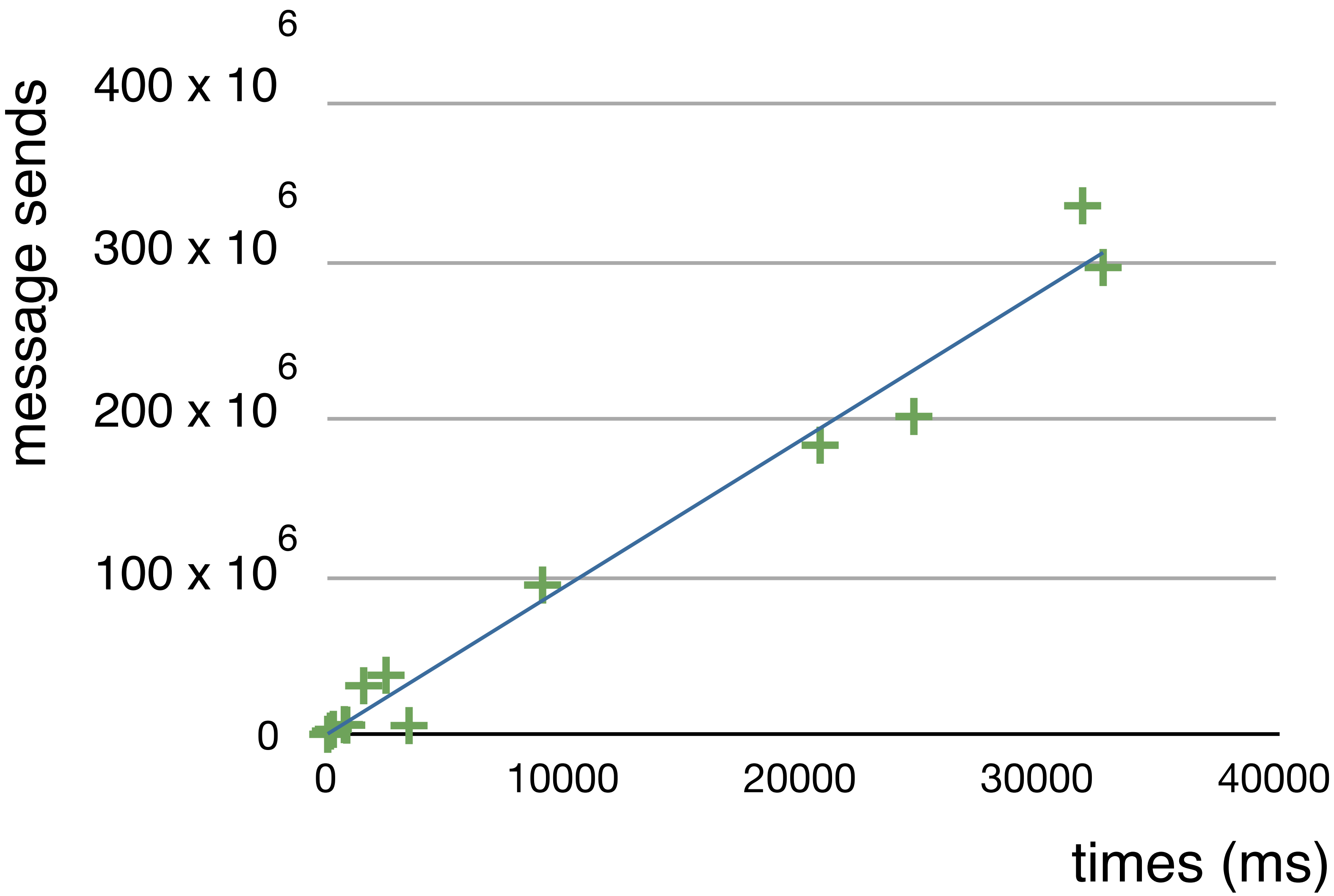
Solution:

counting messages as a proxy for execution time

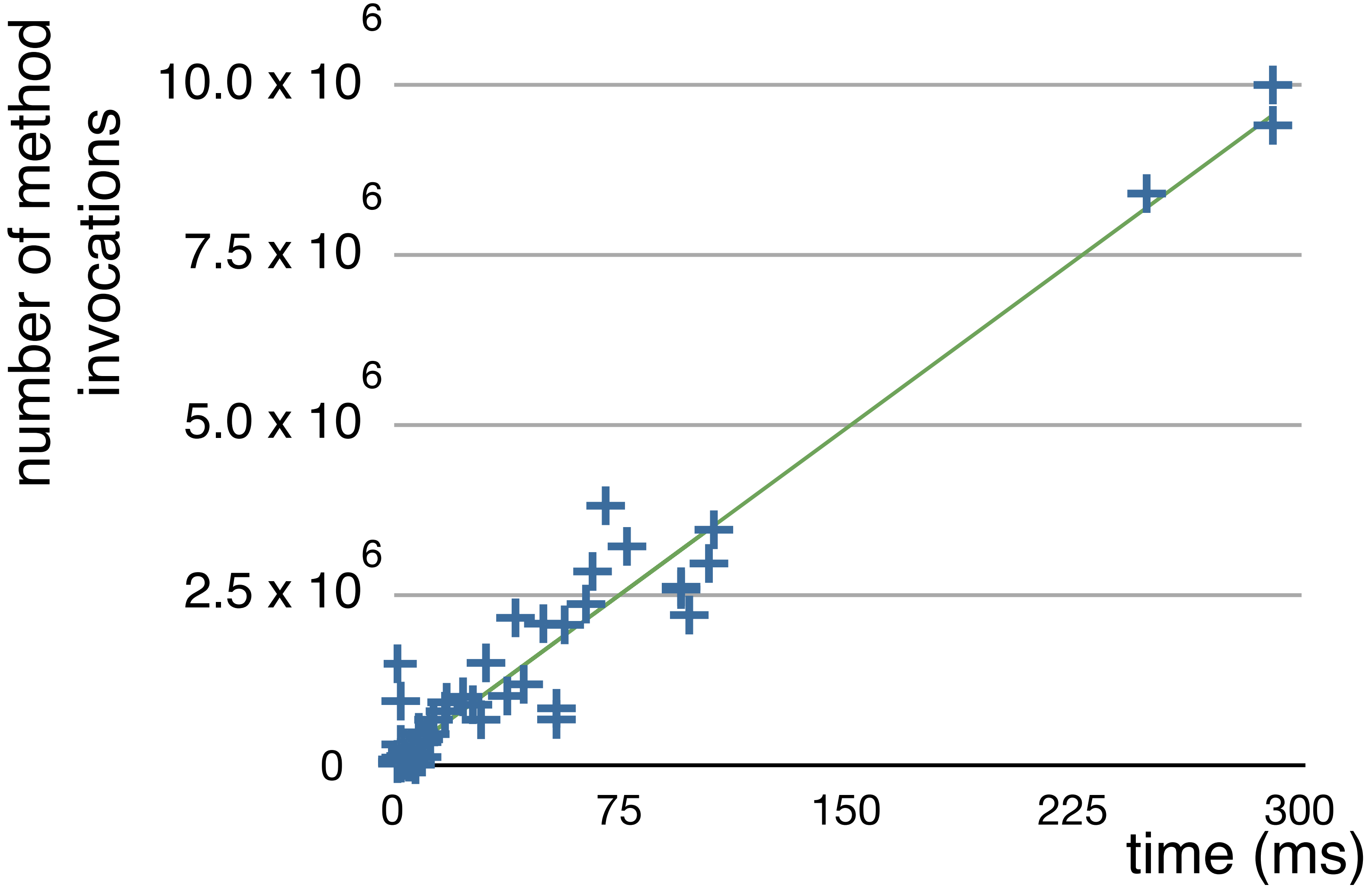
Counting messages

```
Wallet >> increaseByOne
   money := money + 1

Wallet >> increaseBy3
   self
      increaseByOne;
      increaseByOne;
      increaseByOne.


aWallet increaseBy3
=> 6 messages sent
```

Execution time and number of message sends



Counting Messages to Identify Execution Bottlenecks



Contrasting Execution Sampling with Message Counting

No need for sampling

Independent from the execution environment

Stable measurements

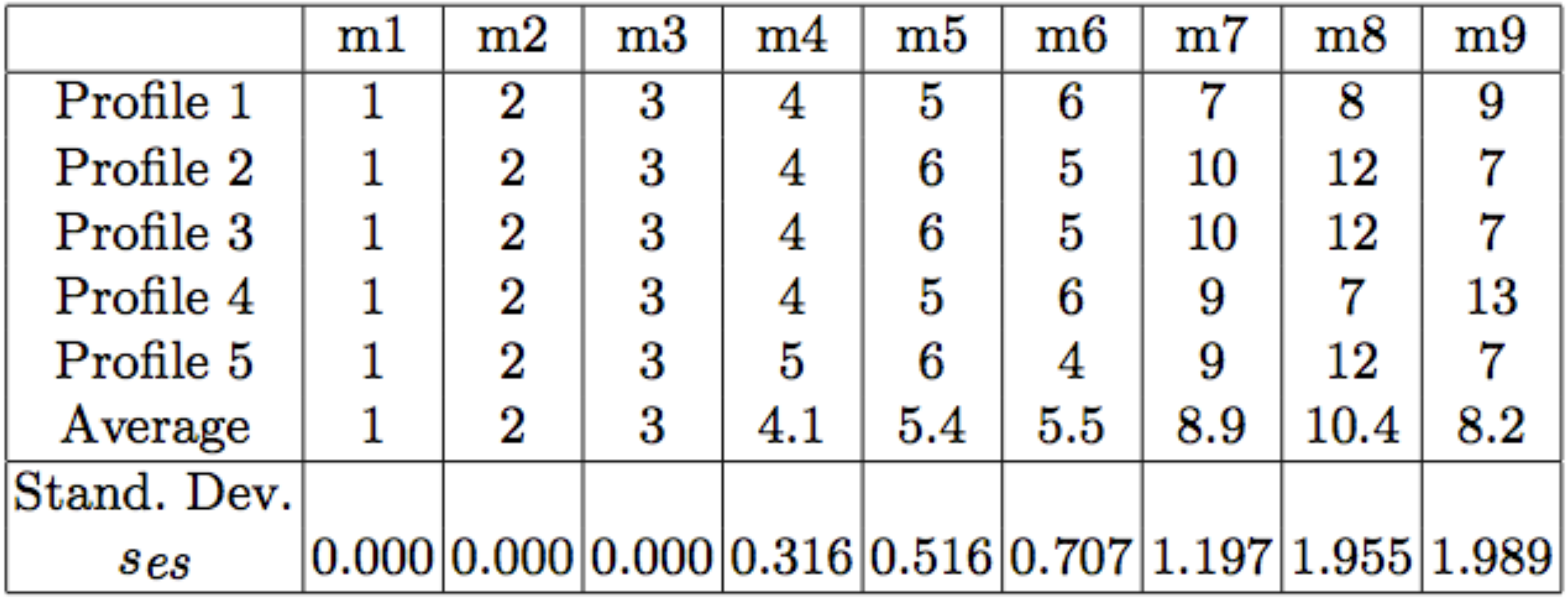
Counting messages in unit testing

```
CollectionTest>>testInsertion
   self
      assert: [ Set new add: 1 ]
      fasterThan: [Set new add: 1; add: 2 ]
```

Counting messages in unit testing

```
MondrianSpeedTest>> testLayout2
  | view1 view2 |
  view1 := MOViewRenderer new.
  view1 nodes: (Collection allSubclasses).
  view1 edgesFrom: #superclass.
  view1 treeLayout.

  view2 := MOViewRenderer new.
  view2 nodes: (Collection withAllSubclasses).
  view2 edgesFrom: #superclass.
  view2 treeLayout.

  self
   assertIs: [ view1 root applyLayout ]
   fasterThan: [ view2 root applyLayout ]
```

Assessing profiler stability



	m1	m2	m3	m4	m5	m6	m7	m8	m9
Profile 1	1	2	3	4	5	6	7	8	9
Profile 2	1	2	3	4	6	5	10	12	7
Profile 3	1	2	3	4	6	5	10	12	7
Profile 4	1	2	3	4	5	6	9	7	13
Profile 5	1	2	3	5	6	4	9	12	7
Average	1	2	3	4.1	5.4	5.5	8.9	10.4	8.2
Stand. Dev. s_{es}	0.000	0.000	0.000	0.316	0.516	0.707	1.197	1.955	1.989

$$\psi^n(P) = \sum_{i=1}^{n} s_{es}(i) * w(n)$$

$$w(n) = 1/ln\ (n+1)$$

★★★★

Identifying redundant computation

Problem:

Traditional code profiler cannot determine whether the same computation is realized twice or more

Why the problem is important:

Redundant computation cannot be identified and removed without heavily involving programmer imagination

Solution:

Finding side-effect-free methods that are executed more than once

Example

```
MOGraphElement>> absoluteBounds
    ^ self shape absoluteBoundsFor: self
```

```
MOGraphElement>> absoluteBounds
    boundsCache ifNotNil: [ ^ boundsCache ].
    ^ boundsCache := self shape absoluteBoundsFor: self

MONode>> translateBy: realStep
    boundsCache := nil.
    ...
```

Example

```
AbstractNautilusUI>>packageIconFor: aPackage
   |t|
   (packageIconCache notNil
     and: [ packageIconCache includesKey: aPackage ])
        ifTrue: [ ^ packageIconCache at: aPackage ].
   packageIconCache
     ifNil: [ packageIconCache := IdentityDictionary
    new ].

   aPackage isDirty ifTrue: [ ^ IconicButton new ].
   t := self iconClass iconNamed: #packageIcon.

   packageIconCache at: aPackage put: t.
   ^t
```

Identifying memoization candidate

A method is candidate for being memoized if

- it is executed more than once on a particular object
- it returns the same value per receiver and arguments
- it does not any “perceptible” side effect
- its execution is sufficiently long

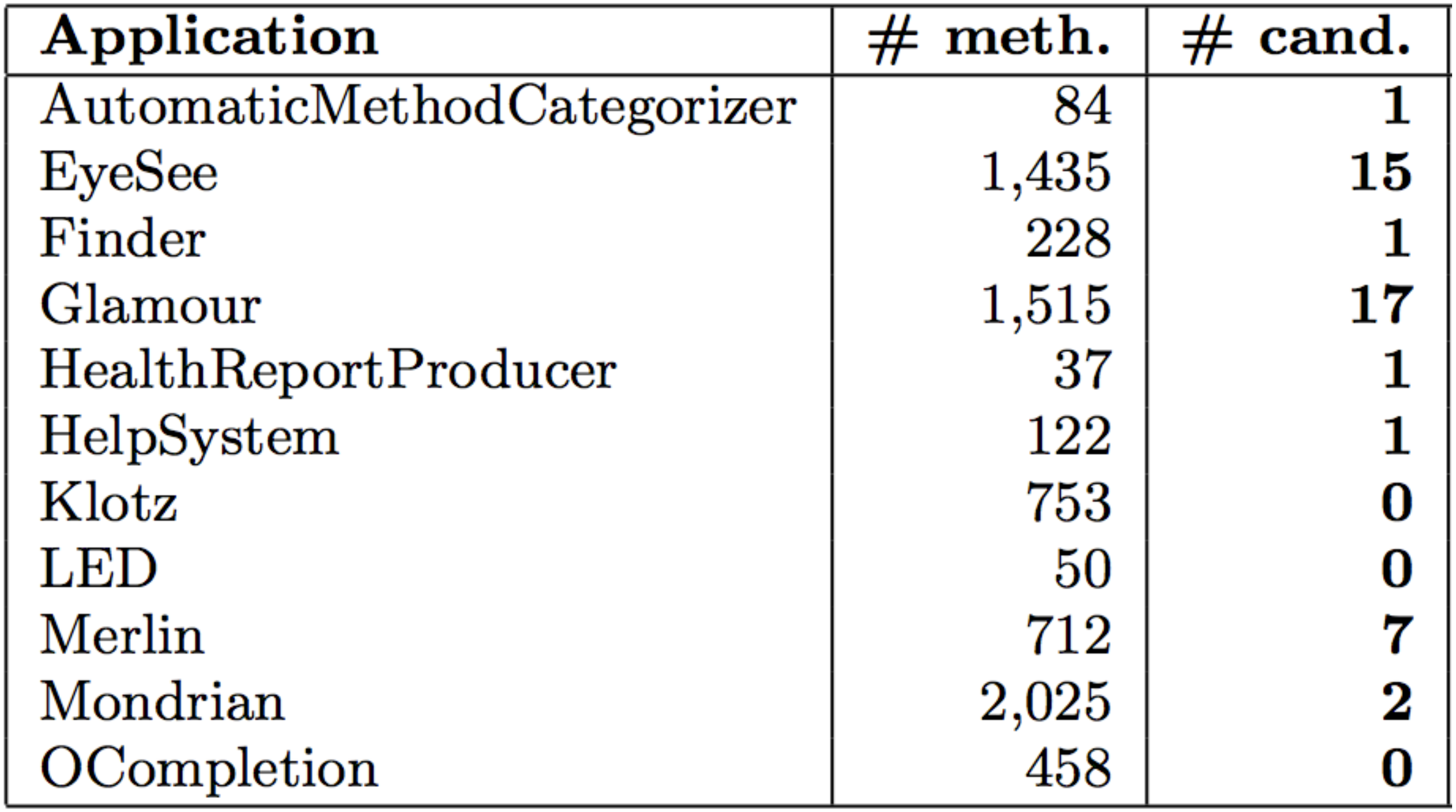
Experiment

We took 11 applications and profiled their unit tests

We identified candidates for each of them

We memoized some of the candidates

The tests are kept green



Application	**# meth.**	**# cand.**
AutomaticMethodCategorizer	84	**1**
EyeSee	1,435	**15**
Finder	228	**1**
Glamour	1,515	**17**
HealthReportProducer	37	**1**
HelpSystem	122	**1**
Klotz	753	**0**
LED	50	**0**
Merlin	712	**7**
Mondrian	2,025	**2**
OCompletion	458	**0**

Execution time

In some case we reduce the execution time by 20%

e.g., Nautilus

In some other case, the execution time increased (!)

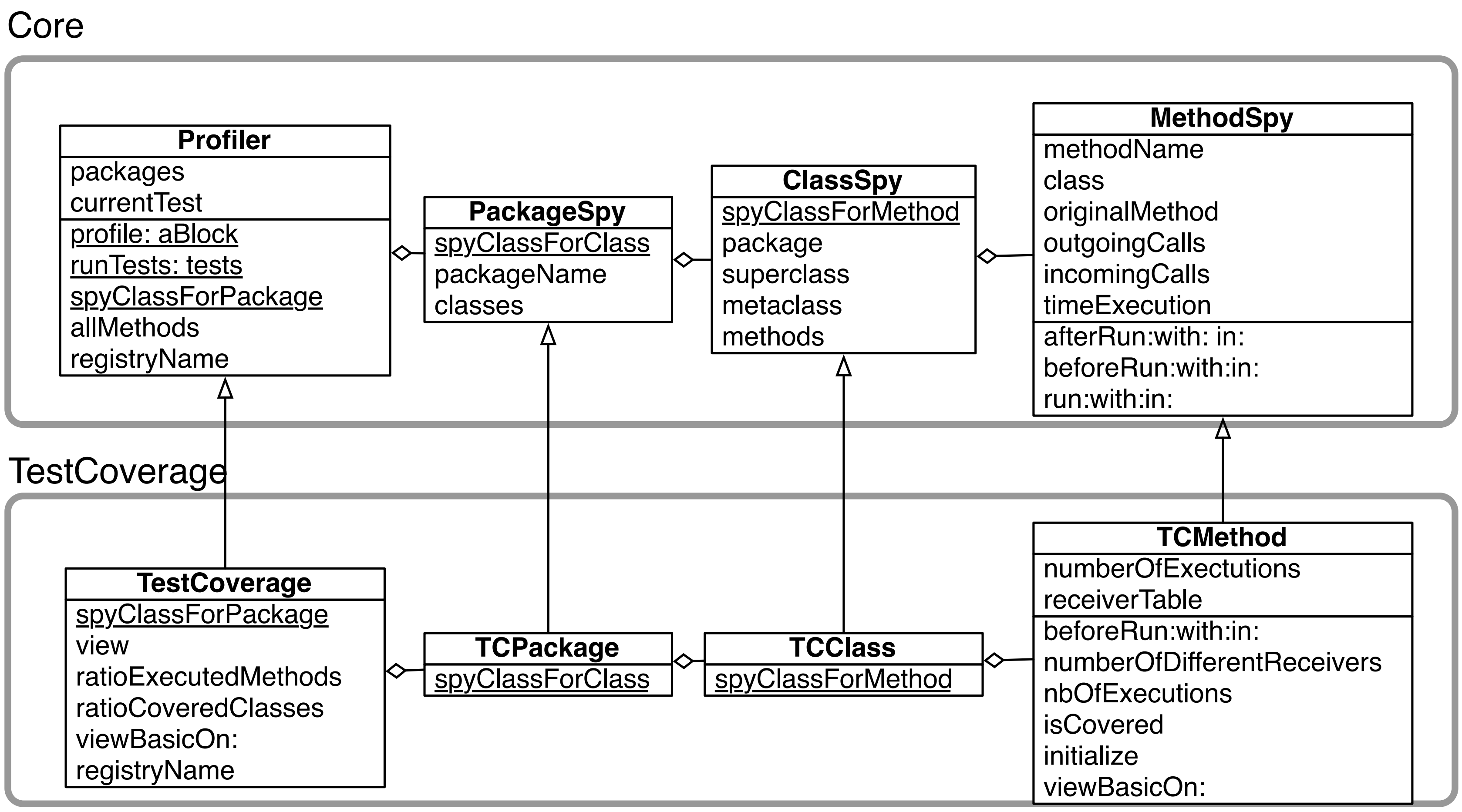
This is the case for very short methods

Research questions

Is there a general way to identify redundant messages by monitoring program execution?

Can redundant messages be removed while preserving the overall program behavior?

The Spy profiling framework



Core

Profiler
packages
currentTest
profile: aBlock
runTests: tests
spyClassForPackage
allMethods
registryName

PackageSpy
spyClassForClass
packageName
classes

ClassSpy
spyClassForMethod
package
superclass
metaclass
methods

MethodSpy
methodName
class
originalMethod
outgoingCalls
incomingCalls
timeExecution
afterRun:with: in:
beforeRun:with:in:
run:with:in:

TestCoverage

TestCoverage
spyClassForPackage
view
ratioExecutedMethods
ratioCoveredClasses
viewBasicOn:
registryName

TCPackage
spyClassForClass

TCClass
spyClassForMethod

TCMethod
numberOfExectutions
receiverTable
beforeRun:with:in:
numberOfDifferentReceivers
nbOfExecutions
isCovered
initialize
viewBasicOn:

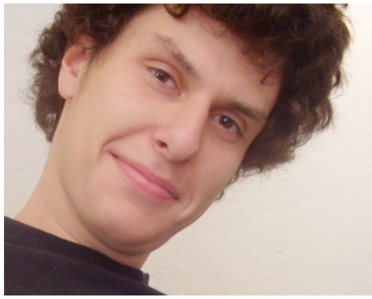
Closing words

Little innovation in the tools we commonly use

- Profilers & debuggers have not significantly evolves

Fantastic opportunities for improvement

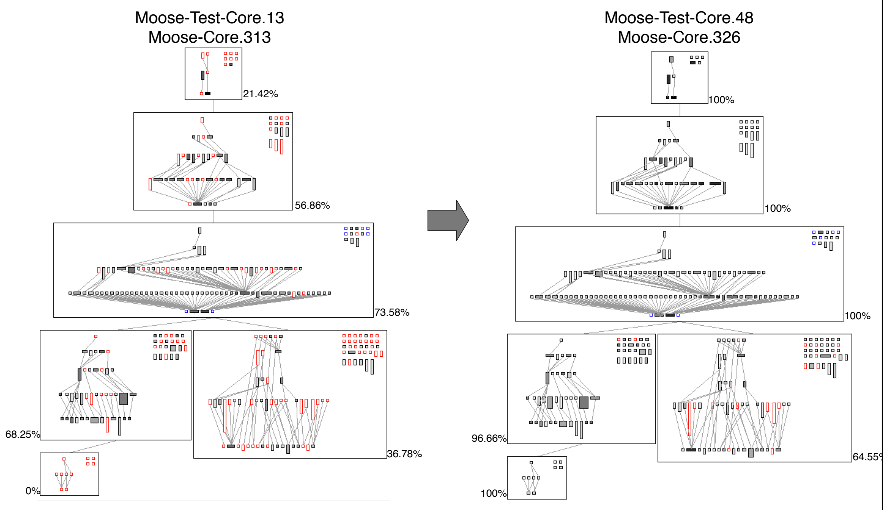
Thanks to



Laurent Laffont, RMoD inria group

all the people who participated in our experiments

Test coverage with Hapao
Profiling blueprints
Proxy for execution time
Identifying redundant computation



Profiler Zoo

http://bergel.eu
http://hapao.dcc.uchile.cl
http://moosetechnology.org/tools/spy