# Profiling, debugging, testing for the next century

Alexandre Bergel

`http://bergel.eu`
Pleiad Lab, Department of Computer Science (DCC), University of Chile

This paper presents the research line carried out by the author and his collaborators on programming environments. Most of the experiences and case studies summarized below have been carried out in Pharo[1] – an object-oriented and dynamically typed programming language.

***Programming as a modern activity.*** When I was in college, I learned programming with C and Pascal using a textual and command-line programming environment. At that time, about 15 years ago, Emacs was popular for its sophisticated text editing capacities. The `gdb` debugger allows one to manipulate the control flow including the step-into, step-over, and restart operations. The `gprof` code execution profiler indicates the share of execution time for each function, in addition to the control flow between each method.

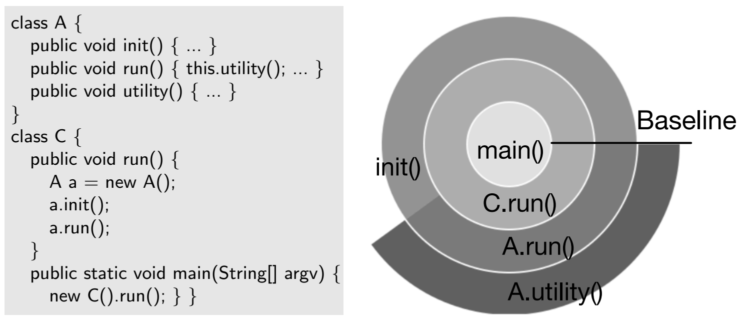
Nowadays, object-orientation is compulsory in university curricula and mandatory for most software engineering positions. Eclipse is a popular programming environment that greatly simplifies the programming activity in Java. Eclipse supports sophisticated options to search and navigate among textual files. Debugging object-oriented programs is still focused on the step-into, step-over and restart options. Profiling still focuses on the method call stack: the JProfiler[2] and YourKit[3] profilers, the most popular and widely spread profilers in the Java World, output resource distributions along a tree of methods.

*Sending messages* is a major improvement over *executing functions*, which is the key to polymorphism. Whereas programming languages have significantly evolved over the last two decades, most of the improvements on programming environments do not appear to be a breakthrough. Navigating among software entities often means searching text portions in text files. Profiling is still based on methods executions, largely discarding the notion of objects. Debugging still comes with its primitive operations based on stack manipulation; again ignoring objects. Naturally, some attempts have been made to improve the situation: Eclipse offers several navigation options; popular (and expensive) code profilers may rely on code instrumentation to find out more about the underlying objects; debuggers are beginning to interact with objects [1,2]. However, these attempts remain largely marginal.

***Profiling.*** Great strides have been made by the software performance community to make profilers more accurate (*i.e.,* reducing the gap between the actual applica-

[1] `http://pharo.org`
[2] `http://www.ej-technologies.com/products/jprofiler/overview.html`
[3] `http://www.yourkit.com`

tion execution and the profiler report). Advanced techniques have been proposed such as variable sampling time [3] and proxies for time execution [4,5]. However, much less attention has been paid to the visual output of a profile. Consider JProfiler and YourKit, two popular code profilers for the Java programming language: profile crawling is largely supported by text searches. We address this limitation with *Inti*.



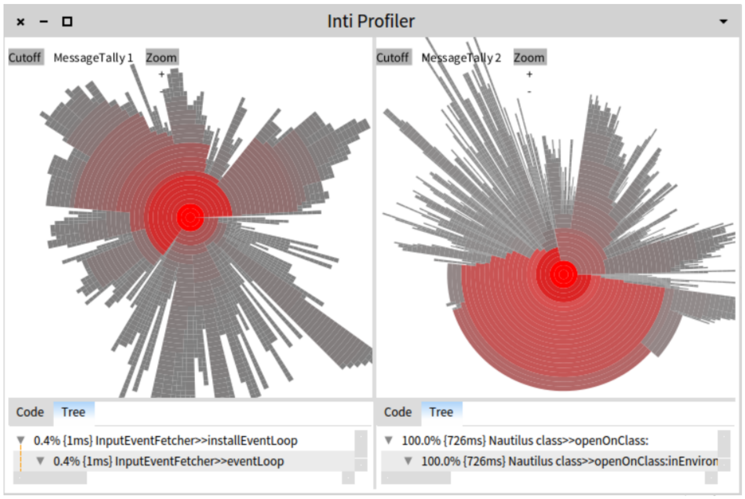
```
class A {
   public void init() { ... }
   public void run() { this.utility(); ... }
   public void utility() { ... }
}
class C {
   public void run() {
      A a = new A();
      a.init();
      a.run();
   }
   public static void main(String[] argv) {
      new C().run(); } }
```

Fig. 1: Sunburst-like visualization

*Inti* is a *sunburst*-like visualization dedicated to visualizing CPU time distribution. Consider the code and arc-based visualization given in Figure 1. The code indicates a distribution of the computation along five different methods. Each method frame is presented as an arc. The baseline represents the starting time of the profile. The angle of each arc represents the time distribution taken by the method. In this example, `C.main` and `C.run` have an angle of 360 degrees, meaning that these two methods consume 100% of the CPU time. Methods `A.init` consumes 60% and `A.run` 40% (these are illustrative numbers). The distance between an arc and the center of the visualization indicates the depth of the frame in the method call stack. A nested method call is represented as a stacked arc.

Inti exploits the sunburst representation in which colors are allocated to particular portion of the sunburst. For example, colors are picked to designate particular classes, methods or packages in the computation: the red color indicate classes belonging to a particular package (*e.g.,* Figure 2).

The benefits of Inti are numerous. Inti is very compact. Considering the number of physical spaces taken by the visualization, Inti largely outperforms the classical tree representation: Inti shows a larger part of the control flow and CPU time distribution in much less space. Details about each stack frame are accessible via tooltip. Hovering the mouse cursor over an arc triggers a popup window that indicates the CPU time consumption and the method's source code.



```
▼ 0.4% {1ms} InputEventFetcher>>installEventLoop
  ▼ 0.4% {1ms} InputEventFetcher>>eventLoop
```

```
▼ 100.0% {726ms} Nautilus class>>openOnClass:
  ▼ 100.0% {726ms} Nautilus class>>openOnClass:inEnviron
```

Fig. 2: Sunburst-like profile

***Delta profiling.*** Understanding the root of a performance drop or improvement requires analyzing different program executions at a fine grain level. Such an analysis involves dedicated profiling and representation techniques. JProfiler and YourKit both fail at providing adequate metrics and visual representations, conveying a false sense of the root cause of the performance variation.

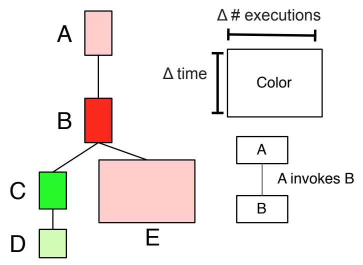


Fig. 3: Performance evolution blueprint

We have proposed *performance evolution blueprint*, a visual tool to precisely compare multiple software executions [6]. The performance evolution blueprint is summarized in Figure 3. A blueprint is obtained after running two executions. Each box is a method. Edges are invocations between methods (a calling method is above the called methods). The height of a method is the difference of execution time between the two executions. If the difference is positive (*i.e.,* the method is slower), then the method is shaded in red; if the difference is negative (*i.e.,* the method is faster), then the method is green. The width of a method is the absolute difference in the number of executions, thus always positive. Light red / pink color means the method is slower, but its source code has not changed between the two executions. If red, the method is slower and the source code has changed. Light green indicates a faster non-modified method. Green indicates a faster modified method.

Our blueprint accurately indicates roots of performance improvement or degradation: Figure 3 indicates that method B is likely to be responsible for the slowdown since the method is slower and has been modified. We developed Rizel, a code profiler to efficiently explore performance of a set of benchmarks against multiple software revisions.

***Testing.*** Testing is an essential activity when developing software. It is widely acknowledged that a test coverage above 70% is associated with a decrease in reported failures. After running the unit tests, classical coverage tools output the list of classes and methods that are not executed. Simply tagging a software element as covered may convey an incorrect sense of necessity: executing a long and complex method just once is potentially enough to be reported as 100% test-covered. As a consequence, a developer may receive an incorrect judgement as to where to focus testing efforts.

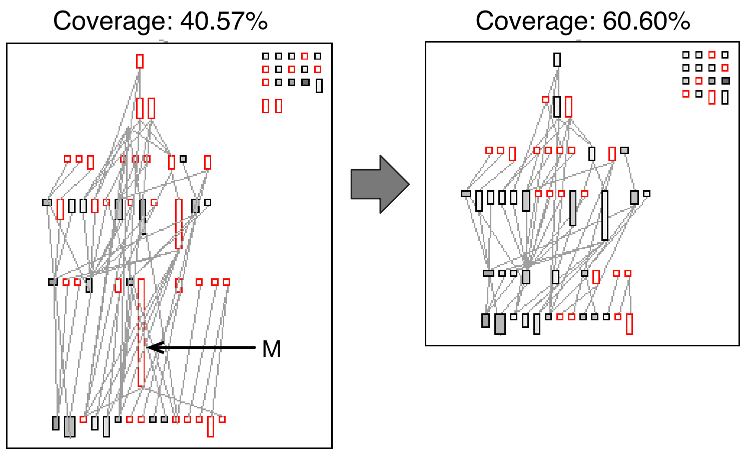


Fig. 4: Test blueprint

By relating execution and complexity metrics, we have identified essential patterns to characterize the test coverage of a group of methods [7]. Each pattern has an associated action to increase the test coverage, and these actions differ in their effectiveness. We empirically determined the optimal sequence of actions to obtain the highest coverage with a minimum number of tests. We present *test blueprint*, a visual tool to help practitioners assess and increase test coverage by graphically relating execution and complexity metrics. Figure 4 is an example of a test blueprint, obtained as the result of the test execution. Consider Method `M`: the definition of this method is relatively complex, which is indicated by the height of the box representing it. `M` is shared in red, meaning it has not been covered by the unit test execution. Covering this method and reducing its complexity is therefore a natural action to consider.

Two versions of the same class are represented. Inner small boxes represent methods. The size of a method indicates its cyclomatic complexity. The taller a method is, the more complex it is. Edges are invocations between methods, statically determined. Red color indicates uncovered methods. The figure shows an evolution of a class in which complex uncovered methods have been broken down into simpler methods.

***Debugging.*** During the process of developing and maintaining a complex software system, developers pose detailed questions about the runtime behavior of the system. Source code views offer strictly limited insights, so developers often turn to tools like debuggers to inspect and interact with the running system. Traditional debuggers focus on the runtime stack as the key abstraction to support debugging operations, though the questions developers pose often have more to do with objects and their interactions [8].

We have proposed *object-centric debugging* as an alternative approach to interacting with a running software system [9]. By focusing on objects as the key abstraction, we show how natural debugging operations can be defined to answer developer questions related to runtime behavior. We have presented a running prototype of an object-centric debugger, and demonstrated, with the help of a series of examples, how object-centric debugging offers more effective support for many typical developer tasks than a traditional stack-oriented debugger.

***Visual programming environment.*** Visualizing software-related data is often key in software developments and reengineering activities. As illustrated above in our tools, interactive visualizations play an important intermediary layer between the software engineer and the programming environment. General purpose libraries (e.g., D3, Raphaël) are commonly used to address the need for visualization and data analytics related to software. Unfortunately, such libraries offer low-level graphic primitives, making the specialization of a visualization difficult to carry out.

Roassal is a platform for software and data visualization. Roassal offers facilities to easily build domain-specific languages to meet specific requirements. Adaptable and reusable visualizations are then expressed in the Pharo language. Figure 5 illustrates two visualizations of a software system's dependencies. Each class is represented as a circle. On the left side, gray edges are inheritance (the top superclass is at the center) and blue lines are dependencies between classes. Each color indicates a component. On the right side, edges are dependencies between classes, whereas class size and color indicate the size of the class. Roassal

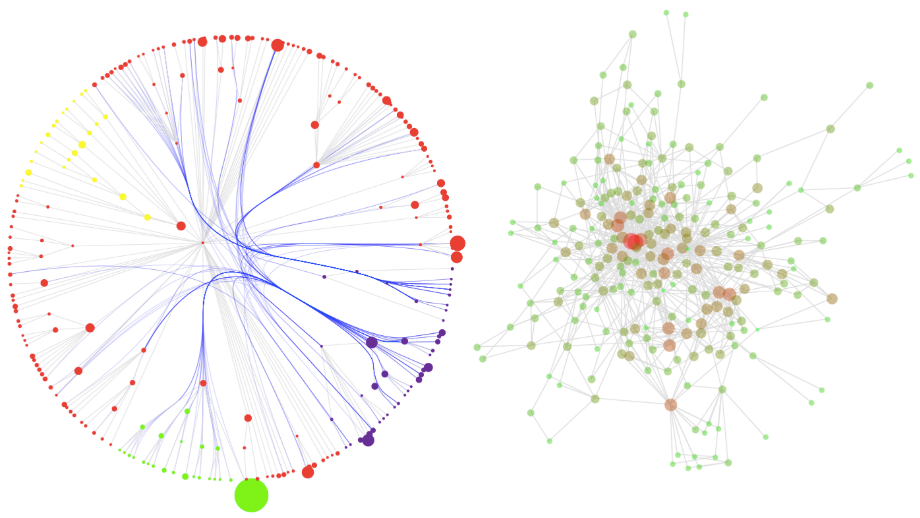


Fig. 5: Visualization of a software system's dependencies

has been successfully employed in over a dozen software visualization projects from several research groups and companies.

***Future work.*** Programming is unfortunately filled with repetitive, manual activities. The work summarized above partially alleviates this situation. Our current and future research line is about making our tools not only object-centric, but domain-centric. We foresee that being domain specific is a way to reduce the cognitive gap between what the tools present to the programmers, and what the programers expect to see from the tools.

## References

1. A. Lienhard, T. Gîrba, O. Nierstrasz, Practical object-oriented back-in-time debugging, in: Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP'08), Vol. 5142 of LNCS, Springer, 2008, pp. 592–615, ECOOP distinguished paper award. `doi:10.1007/978-3-540-70592-5_25`.
URL `http://scg.unibe.ch/archive/papers/Lien08bBackInTimeDebugging.pdf`
2. G. Pothier, E. Tanter, J. Piquer, Scalable omniscient debugging, Proceedings of the 22nd Annual SCM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'07) 42 (10) (2007) 535–552. `doi:10.1145/1297105.1297067`.
3. T. Mytkowicz, A. Diwan, M. Hauswirth, P. F. Sweeney, Evaluating the accuracy of java profilers, in: Proceedings of the 31st conference on Programming language design and implementation, PLDI '10, ACM, New York, NY, USA, 2010, pp. 187–197. `doi:10.1145/1806596.1806618`.
URL `http://doi.acm.org/10.1145/1806596.1806618`

4. A. Camesi, J. Hulaas, W. Binder, Continuous bytecode instruction counting for cpu consumption estimation, in: Proceedings of the 3rd international conference on the Quantitative Evaluation of Systems, IEEE Computer Society, Washington, DC, USA, 2006, pp. 19–30. doi:10.1109/QEST.2006.12.
URL http://portal.acm.org/citation.cfm?id=1173695.1173954
5. A. Bergel, Counting messages as a proxy for average execution time in pharo, in: Proceedings of the 25th European Conference on Object-Oriented Programming (ECOOP'11), LNCS, Springer-Verlag, 2011, pp. 533–557.
URL http://bergel.eu/download/papers/Berg11c-compteur.pdf
6. J. P. S. Alcocer, A. Bergel, S. Ducasse, M. Denker, Performance evolution blueprint: Understanding the impact of software evolution on performance, in: A. Telea, A. Kerren, A. Marcus (Eds.), VISSOFT, IEEE, 2013, pp. 1–9.
7. A. Bergel, V. Peña, Increasing test coverage with hapao, Science of Computer Programming 79 (1) (2012) 86–100. doi:10.1016/j.scico.2012.04.006.
8. J. Sillito, G. C. Murphy, K. De Volder, Questions programmers ask during software evolution tasks, in: Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, SIGSOFT '06/FSE-14, ACM, New York, NY, USA, 2006, pp. 23–34. doi:10.1145/1181775.1181779.
URL http://people.cs.ubc.ca/~murphy/papers/other/asking-answering-fse06.pdf
9. J. Ressia, A. Bergel, O. Nierstrasz, Object-centric debugging, in: Proceeding of the 34rd international conference on Software engineering, ICSE '12, 2012. doi:10.1109/ICSE.2012.6227167.
URL http://scg.unibe.ch/archive/papers/Ress12a-ObjectCentricDebugging.pdf