Writing Quality Code with NetBeans

Learn about tools and best practices that help increase the quality of your code

Osvaldo Pinali Doederlein
Modern IDEs like NetBeans have great source code editors, debuggers, profilers, visual builders and other tools that help producing application that are complete, correct and efficient. But this is not enough: source code should also be well organized and well structured, easy to read and maintain, and compliant with a myriad of “best practices” that help to deliver these qualities and avoid problems. In other words: your code should be above suspicion.

Unit tests and code reviews help make sure your code is well written. But you can also get help from several tools that check code formatting, detect bugs, improve OO style, automate tests – and generally relieve you from at least part of the manual work of code quality assurance.

In this article we look at several tools supported by NetBeans, either out of the box (JUnit, refactoring and the NetBeans Profiler) or through plug-in modules (Checkstyle, PMD and FindBugs). We can only provide an introduction to each tool, but the take-home message is that you can improve significantly your Java software construction process without a huge investment in effort.

Learning to use these tools well – configuring, combining and matching their capabilities under the integrated platform of NetBeans – will let you add important items to your programming discipline without large costs in additional development time. In fact, the proper use of good tools has the potential to reduce total effort, by saving you from hunting difficult bugs, or by making your source code better structured, and easier to read and evolve.

Defensive Programming

When I started programming in the late 80’s, any coding mistake would result in a cryptic error code or a system crash. But few veterans should miss compilers from the eighties as much as they do for Atari 2600 games. Modern compilers like javac produce clear diagnostics for dozens of errors that are statically verifiable. In Java SE 5, new language features (remarkably generic types) expand the number of errors that can be caught at compile time, and this is a Good Thing.

However, we can always use more of a good thing. Even though runtime technologies like exception handling and a safe memory model in the JVM, or ACID transactions in a DBMS, handle runtime errors gracefully and prevent disastrous results, the right time to “catch” any bug is development time. Techniques like unit testing help, but the more static diagnostics, the better.

There are of course tools that go beyond javac, and attempt to detect code anti-patterns: snippets of code that despite not being forbidden by the Java Language Specification (JLS), are fingerprints of bad code. Here “bad” might mean buggy, slow, unstructured, or just difficult to understand and maintain. PMD and FindBugs are perhaps the most popular of the open source offerings, and NetBeans plug-in modules make their use a breeze in the IDE: you will hardly notice that you are reviewing code for dozens of potential programming issues.

Using PMD

We will start by looking at PMD. Its NetBeans plug-in can be fetched by the Update Center (from www.nbextras.org). Once installed, Figure 1 shows the configuration dialog for the PMD plug-in module, and one of PMD’s over 180 rules. After everything is set up (to get started, you can use the plug-in’s installation defaults), select a project’s Source Packages folder and run Tools|Run PMD.
The ReturnFromFinallyBlock rule illustrated in the figure is a good example of the kind of diagnostics PMD performs: it says it’s a bad thing to have a return statement inside a finally block. The Example pane illustrates the problem, and the Information pane explains it. Such returns shortcut any exception thrown or leaked by the try/catch structure, discarding exceptions that were supposed to be delivered to the caller (or by an outer try/catch). This is a bug pattern, because it’s not a very intuitive or useful programming style. In fact many programmers will ignore that the Java language will behave that way. In addition, exceptions are of course supposed to be handled by a catch block — so we can condemn as bad style even the rare code written with the “return inside finally” idiom on purpose.

Validating the validator

Not all PMD rules are so crystal-clear to pick, however. Some are even documented as controversial. For example, the UnusedModifier rule flags redundant modifiers, such as a public method in an interface (since all interface members are public). I like this rule; it makes declarations simpler and not redundant. But others may prefer their source code to be as explicit as possible.

There are other reasons to disable rules, like false positives in some rules with less than perfect detection, or rules that are relevant for some projects but not for others. A good example of both is the rule NonThreadSafeSingleton, which finds Singletons with nonsynchronized getter/initializer methods. This rule may mistake some regular methods for a Singleton getter, or be irrelevant for single-threaded apps. However, most of the time this rule will catch real bugs, and when it does so, you will be very glad to have enabled it.

In short, you have to decide which rules should be enforced or not. This is a difficult balance. Having more rules activated increases the chances that PMD catches important problems in your code, with no effort from you. But too many rules may produce a smorgasbord of warnings that require too much work to review, ultimately making people stop using the tool.

The code-review process must be light enough so you can do it often and quickly. You shouldn’t let one week of construction pass without re-checking all the new-or-updated code (if not the entire project), at least for the most important smoking guns.

Here’s a tip to help adoption of PMD (and similar tools). Start with a “pet project” whose source code is of high quality and small volume. Then activate all of PMD’s rules, run it on the project’s sources, and check the resulting warnings. You will have a clear idea of which rules are inadequate for your personal programming habits. Disable these rules, and run PMD with the same configuration on a larger project — one written by many developers with different skills, with lots of legacy and hacked bug fixes — and check if the number of warnings is reasonable enough so you can start enforcing those rules in the project.

\[1\] Not that it makes much sense to write any non-thread-safe code today, with multi-core CPUs becoming commodity.
Remember that this is mostly a one-time effort for legacy projects. Once all programmers are disciplined to run PMD regularly, and code is written since Revision-1 with a chosen ruleset in mind, the project tends to keep a small or empty set of warnings as it grows. The reviewing process consequently becomes very easy and incremental.

**Showing off PMD**

**Figure 2** shows the NetBeans source editor with a Java class analyzed by PMD. This class was never checked before and all rules are active, so the PMD Output window shows quite a large number of warnings. This output includes some warnings many wouldn’t agree with and disable (excessively long variable names?), as well as one that is hugely important, and which is further commented in the sidebar “The final modifier and refactoring: towards functional-style Java”.

Here is a very small sample of interesting rules supported by PMD:

- **SwitchDensity:** Finds switch statements that have too much code per case, in average. You should consider refactoring large case blocks into separate methods.

- **InsufficientStringBufferDeclaration:**
  
  Even if you build complex strings with StringBuffer (or StringBuilder in JSE 5), your code is not optimal if it uses these classes’ default constructors, which initialize the buffer with a small size, forcing reallocation as data is appended. This rule even recommends a minimum initial size for the buffer, looking at the sizes of all literal strings appended by your code.

- **ForLoopShouldBeWhileLoop:** Detects for loops that could be rewritten as simpler while loops.

- **PositionLiteralsFirstInComparisons:** Prefer “literal”.equals(str) to str.equals(“literal”), because the former will never throw a NullPointerException.

- **AbstractClassWithoutAbstractMethod:** Often indicative of weak OO design. An abstract class with no methods should rather be an interface. And if it contains only concrete methods, perhaps it should not be abstract. Great abstract classes are those that implement design patterns like GoF’s Template Method, with a combination of abstract methods and concrete methods that depend on subclasses implementing the former.

- **UnsynchronizedStatic DateFormatter:** A SimpleDateFormat object that is often used with the same format cannot be initialized once, be stored in a static variable, and then be reused by multiple method invocations that use the same format. The problem is that this API is not thread-safe, so concurrent invocations will break it.

**Using FindBugs**

FindBugs does essentially the same as PMD, so why use two very similar tools? Because each tool has different strengths. PMD’s scope is broader and it is easier to extend. You can create your own rules with relatively little effort (see the sidebar mentioned above), which is also the reason that PMD supports more rules (twice as many as FindBugs) out of the box.

**Figure 2**

PMD in action, warning about violations of the selected rules.
On the other hand, FindBugs has a more advanced architecture, enabling more sophisticated and precise detections. Instead of pattern matching, its “detectors” are implemented on top of bytecode scanning and dataflow analysis. This enables FindBugs to locate such problems as infinite loops, as well as many subtle null-pointer bugs, and even security issues like JDBC code allowing SQL injection attacks. All that with a very small number of false positives. (As a trade-off, you would face a steep learning curve to create detectors for your own rules.)

Showing off FindBugs

FindBugs' plug-in can be downloaded from the same Update Manager site as PMD. Figure 3 shows that FindBugs’ plug-in for NetBeans is prettier, with a custom view that helps to review warnings. The selected warning is for the bug rule “FE”, or “Test for floating point equality”. In Java (and any language using floating-point types), you should avoid comparing floating point values (float, double) with [in]equality operators (== and !=), like explained by the command pane of the plug-in.

In another example, FindBugs sees two violations of DLS (Dead Local Store): redundant assignments to variables that are never read. The flagged variables are exception arguments of catch blocks, so FindBugs actually found that I'm ignoring these exceptions silently, which is of course a bad practice (at least the exception should be logged somewhere).

In Figure 4 we can see FindBugs' settings. In addition to a full list of individual rules in the “Configure Detectors” page, you can set up FindBugs more easily with the Feature-oriented page. In my experience, setting the Level option to Medium (excluding only the rules with Experimental status and Low severity) is enough to keep the number of violations small enough to start using the tool, even in large projects that were never before massaged by a code validation tool.

Here is a list of particularly valuable detectors of FindBugs:

- **BC (Impossible Cast)**
  Finds code that if executed will always fail with a ClassCastException.
- **BIT (Incompatible bit masks)**
  Finds code like “if (A & B == C)” that will always result false, because either B and C, or A and C, have no 1 bits in common.
- **DMI: Code contains a hard coded reference to an absolute pathname**
  This goes in the category “I did that while prototyping/debugging, but forgot to clean up later…”.
- **EC: Invocation of equals() on an array, which is equivalent to ==**
  Unfortunately, the Java platform does not define proper behavior of equals() for primitive arrays (like int[]), but programmers often forget this.
- **IC: Initialization circularity**
  This is a very insidious bug, when class A has static initialization code that depends on class B, and B's initialization likewise depends on A’s initialization.

A real-world case study

- **NP: Load of known null value**
  This is a very useful detector, and it just found an important but elusive bug in the project I'm currently working on. Look at this real-world code:
private Map headers = new HashMap();
public void putHeader (String id, String value) {
  if (value == null)
    headers.remove(value);
  else
    headers.put(id, value);
}

Where is the null-pointer bug? This code
will never throw NullPointerException; the
problem is more subtle. My intention was
that putHeader(id, null) would remove the
header id; but I mistyped the code as
headers.remove(value) – it should have
been headers.remove(id).

FindBugs finds this bug because it doesn't
make sense to use a variable whose content
is known to be null, in lieu of the literal null
value. If I really wanted to remove the null
value from my HashMap I should instead
write headers.remove(null).

A bug like this is very hard to find
because nothing looks wrong on a cursory
examination. No exception will ever be
thrown; and no obvious functional problem
will happen (in my app, the only effect of
keeping unwanted data in this Map would be a brief memory leak).

What is even more vicious is that the method looks so simple –
being a thin wrapper over a Map – that I didn’t bother to write a unit
test for it.

Fortunately, tools like FindBugs or PMD don’t fail in discipline. They
will review your code without missing even the “too easy to need
testing” code.

Programming with Style

Besides fixing bugs, most programmers also care about
keeping their source code well indented and formatted,
making it easier to read and modify. Opinions differ about
many style rules, but most people will agree that any style is
better than no style.

Checkstyle can check all sorts of style rules, like indentation,
spacing, naming conventions, modifier usage, curly braces and so
on. It can also find violations of several programming best practices
and potential bugs, so there’s some intersection of functionality
with PMD and FindBugs, which can also check some style-related
problems. In general, Checkstyle is much better on style rules, and
the latter tools are more powerful otherwise. My recommendation is
to enable all tools, but configure their respective configurations to
not have overlap (otherwise your reviewing effort will increase with
the duplicated warnings).

Checkstyle’s NetBeans plug-in must be downloaded from the
project site, and installed from the local NBM file. The plug-in doesn’t
currently offer a GUI for choosing rules. You must first configure a
XML file that specifies style rules, like these:  

```xml
<module name="LeftCurly"
  option="nl"/>
```

You don’t actually have to write such a file, as Checkstyle
comes with standard configurations, the most popular being
sun_checks.xml (conventions from Sun Microsystems). You can
start with such a default configuration and tweak a few rules for a
perfect match with your preferences (detailed documentation for
each rule is provided by Checkstyle’s docs).

Figure 5 shows the output from Checkstyle for a test class that
purposefully violates many of my personal style rules: no javadocs
for public methods and no package.xml file; equals() without
Unit testing

Unit tests (of course) are runtime tests that cover fine-grained code artifacts: classes and individual methods. These tests are sometimes classified as “white-box tests”, because writing them requires intimate knowledge of the code: you should invoke all important methods directly, and know which parameters are required and what results are expected for every call.

In the Java platform, JUnit is the de facto unit testing framework, and most of its success should be attributed to simplicity: writing test cases require using a minimalist API and following very simple conventions, remarkably that a test case is a method with signature like “public void testName()”. But the integrated JUnit support in NetBeans makes it even simpler for beginners.

Suppose you have written a class like that shown in Listing 1. This class is part of a library of several implementations of a Date-parsing interface (for a fixed input format), with successive optimization refinements.

Now, optimization is a risky business, so I will sleep better if these methods are tested. With NetBeans, you can select the class and call Tools>Create JUnit tests. This wizard will create a class (in the project’s Test Packages) with several methods like the one shown in Listing 2.

You must only fill in the blanks providing some real input and output data, like in Listing 3 (where I also trimmed the code a little). Then you only have to run the unit tests with Run>Test “project name”, and check the outcome in the JUnit Test Results view (Figure 6). Writing and performing tests was never easier!
Unit Profiling

Notice that JUnit reports not only whether each test has passed, but also its execution time. This suggests we can reuse unit tests for performance testing. Unfortunately, a single execution of a simple method like my parse()s may be too fast for the precision of JUnit’s timer (see the “0.0s” result for most tests in Figure 6). Not to mention other difficulties with Java microbenchmarks, like the need of warm-up time to let dynamic JIT compilers work. But you can fix this by instrumenting the test code, adding loops to run long enough to allow performance measurement, like in Listing 4.

Notice that because we’re repeating each tested method 100,000 times, the result timing printed by JUnit should be interpreted as hundredths of milliseconds. Also you should run your performance tests with the same JVM options that are expected to be used in production; server-side programs will typically use at least the -server option to enable the HotSpot Server JVM (edit this in the project’s Properties>Run>VM options).

You can see the result of this performance test in Figure 7. Now the difference in performance between the various parsing algorithms is very clear (the SDF algorithm is the standard invocation of SimpleDateFormat.parse(); the others are increasingly optimized).

Being able to reuse JUnit tests for performance benchmarking is cool: you don’t have to write timing code with System.currentTimeMillis() (or nanoTime() in JSE 5) before and after runs; neither print formatted...
Of all PMD rules, some of my top favorites are LocalVariableCouldBeFinal, MethodArgumentCouldBeFinal and ImmutableField. The rules suggest declaring local variables, parameters and fields as final whenever possible. Many programmers follow this recommendation solely for fields that are initialized only by constructors. The final tag prevents bugs where a method would inadvertently update the value of a field that should be constant over an object's lifecycle, such as a person's date of birth.

But what's the deal with final parameters and locals? Check this code:

```
// Before:
Employee lookupByPhone (String phone) {
    // Normalizes the phone number
    phone = removeNonDigits(phone);
    Employee employee = findEmployeByPhone(phoneNorm);
    logger.debug("lookupByPhone(" + phone + ") = " + employee);
    return employee;
}

// After:
Employee lookupByPhone (final String phone) {
    final String phoneNorm = removeNonDigits(phone);
    final Employee employee = findEmployeByPhone(phoneNorm);
    logger.debug("lookupByPhone(" + phone + ") = " + employee);
    return employee;
}
```

The method lookupByPhone() shows the value of final modifiers. First off, they explicitly segregate three semantically distinct entities, which happen to be implemented by a single "local variable" construct in the Java language: Parameters, Local Variables, and Local Constants.

Using final where appropriate delivers three main benefits:

1. **Avoids misusing a parameter as a local variable** (e.g., as a for loop counter).

   This is confusing, especially for debugging; if you break into the middle of a method's execution, you won't see the original value of parameters that are later assigned to. Even clicking back in the call stack may not reveal parameter values easily, because these may originate from expressions that weren't stored in variables by the caller (e.g., f(g() + h())). With final parameters, the inputs for a method or constructor are always preserved.

2. Shows clearly which identifiers are constants (names for fixed values or shared sub-expressions), and which are real variables (data whose values vary with time).

   The ability to see constants is important because the fixed binding of a name/value pair is an invariant that you can rely on when analyzing the behavior of complex code.

3. **Results in clearer, self-documenting code.**

   In the example, the original code was modifying the phone parameter; to make this parameter final, we had to introduce a new local variable, phoneNorm. The good thing is that we can encode the meaning of this change in the new variable's name: the phone number without non-digit characters is a normalized phone number (i.e., compatible with PKs in the database, keys in Maps, etc.). But we don't have to write a comment like "Normalizes the phone number", because the new identifier conveys this information – not only in its declaration, but anywhere else it appears.

   Notice that phoneNorm is also declared final, because it's the single transformation we have to do to the phone data. Indeed, most of the time we can replace a variable by multiple constants. This in turn leads to a "functional-style" of Java programming, one that uses as few destructive assignments* as possible.
A second example illustrates better the benefits of such a programming style:

```java
final Employee employeeToEvaluate =
    company.getDepartment(depID).getEmployees().get(empName);
if (employeeToEvaluate.calcProductivity() < 7.5)
    company.fire(employeeToEvaluate);
```

Notice that the `employee` variable is redundant: it exists with the sole purpose of avoiding the duplication of a long expression. This variable is really a simple alias to a navigation path in our object graph (company → department[depID] → employees (a Map) → get(empName)). Coding this expression into a `final` variable, i.e. a constant, implicitly documents this fact.

Another interesting aspect of `final` variables is their interaction with refactorings. If you have duplicated expressions in a method, you can use NetBeans’ `Introduce Variable` refactoring to remove this duplication. But Figure B1 shows the right way to use this refactoring: checking the option “Declare Final”. It’s there for a good reason.

If you buy my idea of functional-style Java, there are other tricks that allow even more variables to be eliminated or replaced by `finals`. For example, what to do with a piece of code that uses multiple assignments to a variable, due to different paths of execution?

```java
double raise = 1.0; // default
if (employee instanceof Manager)
    raise = 1.1;
else if (employee instanceof CEO)
    raise = 1.25;
else if (employee instanceof JavaProgrammer)
    raise = 1.6;
employee.setSalary(employee.getSalary() * raise);
```

The answer is easy: use NetBeans’ `Extract Method` refactoring to move the if/then/else structure to a new method, say `calcRaise(Employee)`. (A good OO programmer will further promote this to a polymorphic method of Employee, overridden by each subtype, instead of using `instanceof` tests.) The resulting code is much simpler:

```java
employee.setSalary(employee.getSalary() * calcRaise(employee));
```

```java
double calcRaise(Employee emp) {
    if (employee instanceof Manager)
        return 1.1;
    else if (employee instanceof CEO)
        return 1.25;
    else if (employee instanceof JavaProgrammer)
        return 1.6;
    else
        return 1.0; // default
}
```

Notice that the Extract Method refactoring will not do the transformation of assignments to the `raise` variable to `return` statements; you have to do that yourself to obtain the code above. We also invoked this new method directly from the `employee.setSalary(…)` expression instead of using a `final` variable to hold that temporary value. We can not only make a variable `final`, but eliminate it completely! Notice we don’t lose any code clarity, because the same semantic information encoded in the name of the `raise` variable is now encoded in the name of the `calcRaise()` method. Notice also that `calcRaise()` is a “purely functional” method: it contains no destructive assignments and no side effects, and exhibits the monotonic property (if invoked repeatedly with the same input values, it will deliver the same results, and not affect anything else in the system).

These properties are very interesting for a number of reasons. Now, I won’t sell you the full Functional Programming paradigm (I don’t practice it either), as it includes harder deals, like forcing the replacement of all iteration by recursion. My point is just snatching into Java an easy part of that paradigm that delivers many of the benefits.

---

* Assignments that overwrite a previous value, bound to the same identifier by a past assignment.
The transformation we just demonstrated shows more interesting benefits:

- **Code is more modular.**
  - Having lots of small methods is usually better than a few big ones. Even if the new methods are one-trick ponies that don’t lead to any reuse, they help making your code self-documenting. Picking good identifiers for these methods and their parameters will embody information that is not always present in equivalent code inside a monolithic method.
  - Of course, these “extracted methods” will usually be private. A class with a large number of public methods is ugly, but the number of private methods is not important. If the class as a whole is too big, the excess complexity is independent of its internal structure. And the fix for that complexity is breaking the class into more pieces (e.g. via inheritance or delegation) – not faking less complexity by packing code into fewer, tighter methods.

- **Unit testing is much easier.**
  - It’s hard to argue with this: simpler methods that do less things are much easier to test. In the new version of our code, we can write a set of JUnit tests that target `calcRaise()` and exercise all possibilities of its algorithm: employees of multiple types and states, including the null parameter. It’s much harder to do that if your interface with the code is just a higher-level method that sets the raises (the employee may not be an immediate parameter of that method). Indeed, the refactored code allows you to black-box test a class (i.e. test it knowing only a minimal interface). You don’t have to dig inside the source code of each method to analyze every possible execution path and laboriously derive the input data that will force execution of each path.

**Conclusions**

The purpose of this discussion was to show that tools like PMD and refactoring are powerful enough even to help enforce sophisticated programming idioms and paradigms. Even if you don’t like the concept of “functional-style Java”, you may have other ideas of recommended practices, and perhaps rely on these tools to realize such ideas.

It’s worth notice that PMD is easy to extend. Many kinds of rules can be programmed in XML ruleset files, with XPath expressions – often one-liners – that match Java abstract syntax tree nodes. Not trivial but much easier than getting acquainted with the complex API of a tool like FindBugs.

Another goal of this discussion was revealing that many recommended rules are worth more than their face value. PMD’s documentation for the `LocalVariableCouldBeFinal` rule is a mere “A local variable assigned only once can be declared final”. But this definitely doesn’t capture its potentially profound implications.

---

*Not possible if `calcRaise()` is a private method, so standard practice for unit-testers is using package-private instead. But even if you don’t want to do relax your member access, the decomposition of code into more methods makes it much easier to analyze it and plan tests that, by entering your class only through public methods, will cover all important code.

results, or create main() methods that invoke all tests of a “suite”, not to mention other features of JUnit. When you’re not interested in testing performance but only in correctness, just set the `LOOPS` control variable to 1 (this could be done more dynamically, by a JVM property).

**Using the NetBeans Profiler**

The section on performance testing couldn’t end here of course, because we are using NetBeans, which includes a very powerful Profiler. This feature requires an independent installation, but it’s a “standard extension” (see details about the tool in the article “Exploring the NetBeans Profiler” in this edition).

Having the Profiler installed, don’t start it with the standard option `Profiler>Profile Main project` – because this will run the application’s main class. Instead, follow
Refactoring and code templates can not only save you typing time, but spare you from bugs caused by inane editing tasks – like fixing all calls to a method with a default value to a new parameter, or typing the thousandth “for (int i = ...)” loop header in a project. In the runtime and building side, NetBeans’ integrated support for Ant is unparalleled, allowing further automation of tasks like compilation, packaging and deployment. (There is a plug-in for Maven, too: Mevenide).

A Version Control System is another must for any self-respecting project, even single-developer ones. NetBeans supports several VCSs, including CVS and Subversion. But don’t miss important extras, like the Local History and CVS/SVN Report (see the special section Plug-in Showcase in this magazine).

Conclusions

The NetBeans IDE contains many powerful, high-level features that go beyond basic editing and debugging capabilities, and even beyond eye-catching GUI tools like wizards and visual builders. Built-in tools include support for refactoring, unit testing and profiling. Extra tools include FindBugs, PMD, Checkstyle and many others, all integrated into the open, extensible architecture of the IDE. It’s a real sin to miss all the productivity and quality benefits of adding these tools to your daily routine. It will save much more time in the long run than the few days you’ll need to invest in learning and customizing the tools, even those that provide many dozens of complex validation rules.

Figure 8 shows the results of a simple profiling session. We can see the performance of each tested method, and also the breakdown of their executions into invoked APIs and subroutines. The HotSpots view shows a global ranking of the higher-cost methods, and the NetBeans Profiler offers many other tools to analyze and visualize the performance of your code (see the article NetBeans Profiler, in this edition, for additional details and features).

This combination of unit testing and profiling saves effort and also gives you freedom to program incrementally, experimenting new ideas, performing fixes, reviews and refactoring, all without fear of breaking some code or making it slower.

Additional Tools

Other facilities of NetBeans are important for writing good code, and keep its quality as projects grow and evolve. Osvaldo Pinali Doederlein (opinali@gmail.com) is a software architect and consultant, working with Java since 1.0beta. He is an independent expert for the JCP, having participated in the JSR-175 (JSE 5) effort. He is also a contributing editor for the Brazilian Java Magazine.

These steps:

- Change the LOOPS counter to a smaller value like 100, because execution is slower under the profiler (if you don’t filter out any code).
- Select the unit test class and execute ‘Profile file’. Select Analyze performance > Entire application, and execute it.
- Wait a few seconds until the process terminates.

Figure 8

Running the JUnit performance benchmark under NetBeans Profiler.