Using a Profiler Efficiently

Strategies that Help you to Find Performance Problems and Memory Leaks

Tim Wellhausen

kontakt@tim-wellhausen.de
http://www.tim-wellhausen.de

May 24, 2009

Proceedings of the 13th European Conference on Pattern Languages of Programs (EuroPLoP 2008), edited by Till Schümmer and Allan Kelly, ISSN 1613-0073.

Abstract: Sooner than later most software development projects suffer from severe runtime problems. When features are given top priority, caring for non-functional requirements such as performance or stability is most often postponed during the initial development phase. Once a system is in production, however, performance problems and memory leaks quickly catch more attention. A Profiler is a very useful development tool to find the causes of these problems. Using a Profiler is not that easy; you need good strategies to detect the actual causes. This paper gives you advice how to use a Profiler efficiently.

Introduction

Performance problems and memory leaks are encountered in many software development projects. Unfortunately, they often have subtle causes that are not apparent by introspecting the code. In particular, complex, multi-layered software systems are hard to debug to find these causes.

Just as a Debugger is the tool at hand to track down problems that affect the correctness of a software system, a Profiler tool can be very useful to trace performance problems and memory leaks. Only a Profiler gives you an accurate view on what's happening inside the system either over a period of time or at a specific point of time.

A Profiler supports the analysis of a software system at runtime: (1) by finding the causes of real or perceived slowness of a system, i.e. those parts of the system that consume more time to fulfill a functionality than they should take or (2) by finding the causes of memory leaks that exhaust the available memory until the application runs very slow or stops running at all.

This paper presents usage patterns that cover both aspects of using a Profiler. The patterns are independent of a specific Profiler product. However, there are some products that support all usage patterns, whereas other products only support some of them. This paper neither gives you an overview of the available products nor explains their completeness regarding the patterns.

Note that this paper assumes that you are already familiar with the Profiler tool of your choice, i.e. that you know how to start a Profiler session and how to take a memory snapshot, for example. Also note that this paper only addresses Profiler products with a rich graphical user interface; Profilers that only record data in text form are out of scope.

Although the patterns are not dependent on specific technologies, they are based on experiences in profiling object-oriented, single- or multi-layered software systems that are developed on a technological platform that involves garbage collection at runtime (for example Java and .NET). It has not been deeply analyzed yet how valid the patterns are if applied to other programming languages and platforms, in particular to the area of embedded software.

The patterns are presented one by one. Each patterns has a short problem and a short solution statement, written in bold font. To get an overview of the pattern, you may first just read these statements for each pattern. Then, you may read the patterns one after another or you may start by reading the first pattern, THINK ABOUT IT FIRST, and then follow the recommendations as given in the patterns' descriptions.

At the end of the paper, you can find an Examples section that shows how the patterns can be applied in sequence, illustrated by real world applications. After that, you can find references to other resources about profiling.

Think About it First

You are responsible to analyze a software system to detect the cause of its severe performance problems or memory leaks. Maybe you always have a good gut feeling of possible causes for such problems; but so far, you don't know the causes yet.

How do you start solving the performance or memory problems of a software system?

As a software developer you are accustomed to find solutions for given problems. If you don't know the exact reason for a problem, you might, for example, be tempted to start developing a solution that incorporates well-known design patterns to improve the performance of your software system in general, such as an object cache or a resource pool.

But without knowing the actual causes, you cannot be sure that any changes you perform on the software system actually improve the performance or remove the memory leaks. Whatever you are doing might simply add complexity to your software system but may not improve it. Or even worse: may introduce new bugs to previously running code.

Therefore:

Don't guess what the causes of the problems might be because quite often you're going to be wrong. Instead, use a Profiler to analyze your software system!

What sounds like a mundane advice already is the single most important advice this paper has to offer. If you don't know the actual reason for a performance problem, don't be induced to prematurely start coding a solution, even if you believe that this solution might solve the problem. More often than not, the real troublemaker is more subtle than you might think on first sight.

By using a Profiler you can double-check whether your assumptions are right. If are are right, go on and develop the solution you had in mind. If you are wrong, however, be relieved that you have spared yourself from unnecessary work and that you have saved the system from unnecessarily adding complexity.

If you are not the developer who has originally written the software system, chances are that you do not know the complete functionality of the system. In this case, there are hopefully test documents that contain step-by-step instructions for each essential use case or process of the system. Having such instructions at hand, you can more easily profile the system to understand both its functionality and runtime behavior.

Naturally, there are cases when your gut feeling is right. If you never trust your guesses but always check first, you might lose time fixing your software system. So there is a trade-off to make when time is the most limited resource at hand. In this case, it may help to time box any efforts following your gut feeling. Additionally, learning and applying tools is much effort that you need to afford in your project.

If the system runs slow, you should begin the profiling session by Finding Performance Anomalies. If you guess that the system suffers from memory leaks, Check for Memory that the software system consumes at runtime.

Find Performance Anomalies

Your software system runs slower than you think it should. The normal workflow seems to be okay but some deviations happen. You don't know yet why.

How do you start tracking down performance problems?

Some performance problems may materialize at places and at times that are not obviously connected to their real causes. If you delve down into details at once, you may miss the actual causes and get lost in too much information that does not lead you anywhere. Getting lost in such a way often is frustrating and may make you stop profiling and start guessing what the reasons might be.

If you don't know the software system very well, you might not easily distinguish between acceptable and unacceptable performance. Some parts of the system behave better than others. How do you know which behavior is good enough and which is not acceptable any more?

Randomly trying to perform some actions to get an impression of the system performance is like looking for a needle in a haystack. You can easily spend a lot of time without getting any hints where to look closer.

Therefore:

First get a general impression of how fast typical operations execute. Then try to find anomalies by comparing more specific actions with these numbers.

Only seldom do all parts of a software system suffer equally from performance. Skimming over many parts of a system should give you a good impression of the overall characteristics you might expect. You could, for example, set the fastest non-trivial operation as a benchmark for all other operations. Those operations that differ considerably from this benchmark are the best candidates for closer inspection.

The more often you try to find performance anomalies, the better you get to know the performance characteristics of your system. By doing this on a regular basis, chances are better to more quickly spot and improve performance problems.

Always observe the size of the data set that your system operates on. For a realistic comparison, the data set size of several operations should be roughly equal.

Getting a good general impression may be difficult if the software system behaves inconsistently, i.e. differently at different times, in particular if there are active background threads running. Also, if the system contains several performance problems at the same time, it may be very complicated to judge how fast typical operations should execute.

By comparing performance characteristics you should be able to identify at least some parts of the software system that you need to inspect more closely. As next step, you should Isolate Actions.

In case the system does not reveal performance problems under normal load: Stress it. Because using a Profiler slows down the machine on which the Profiler and maybe also the system itself are running, you should try to Minimize the Profiler's Overhead of using the Profiler.

If it is difficult to reproduce performance problems on a local workstation, Profile the Real Thing; if you are not allowed to profile in a production environment, Clone Production. In both cases, try again to Find Performance Anomalies.

Check for Memory

Your software system behaves unreliably. You don't know yet why.

How do you find out whether the software system suffers from memory leaks?

There might be many reasons why a software system does not behave as it should. Some reasons are internal to the system (e.g. a memory leak), some are external (e.g. hardware failure). Before you make any changes to your system you need to be sure that the problem is caused by an internal error.

A memory leak may manifest itself by an obvious system event like an `OutOfMemoryException` in case of Java. But in particular if the system has a lot of memory available, it may take a while until the memory has run up and such obvious messages are shown.

If your system is interactive, use the software application as a user would do. If the system executes batch jobs, manually trigger jobs as they would normally run. In both cases, closely watch the system's memory consumption.

If the system actually has a memory leak, you should notice that the memory consumption of the system increases over time. Be aware that a Profiler typically shows the memory consumption as the size of all objects that are currently alive. Some of these objects may not be in use any more. Therefore, you should watch the activity of the garbage collector and, if necessary, regularly trigger a garbage collector run to watch the size of objects in use over time.

Even if the memory consumption increases over time, this must not necessarily indicate a memory leak. Other causes might be data caches that fill up over time, additional code that is loaded at runtime, data that is stored in a session as long as a user is logged in, or data sets whose size increase.

You therefore need to have a good general knowledge about the software system to be able to distinguish between acceptable and unacceptable memory increases.

Once you know that the system has a memory leak, you need to Isolate Actions that are responsible for the memory leak. If the memory leak causes a proliferation of objects, chances are that the Profiler puts your local system under heavy load while keeping track of what's happening inside your system. In that case, Minimize the Profiler's Overhead.

Minimize the Profiler's Overhead

You intensively use a Profiler on a local workstation to track down performance or memory problems.

How can you avoid that the Profiler itself negatively affects the application?

In a real-world application, many operations are executed in a short period of time. Within a couple of seconds, millions of objects may be created and disposed, and thousands of operations may be called. Because the Profiler cannot guess the reasons of the problems at hand, it has to collect all available information to present to you the most accurate view of the internals of the software system.

To monitor a software system in every detail, the Profiler itself consumes many resources, i.e. it needs a lot of memory and takes a considerable amount of time. Using a Profiler may therefore consume so many system resources that the software system to analyze is negatively affected. Connections may time out, network packets may be dropped, or the graphical user interface may become too slow to use.

Therefore:

Before you start a profiling session, reduce the amount of information gathered by the Profiler to the absolute minimum.

Most Profiler products provide a multitude of options to control the behavior of the Profiler itself. As most Profiler products support both the analysis of memory consumption and the analysis of runtime performance, they also support selectively turning these features on and off.

Some more sophisticated Profilers give you detailed control of the granularity with which the Profiler acts. This means, for example, if you need only a general impression of the memory usage, it may not be necessary for the Profiler to record every object creation but to take snapshots of the memory consumption every now and then.

Quite a few Profiler products provide the possibility to start and stop the collection of data at runtime. This means, you are able to initially turn off most options to let the software system start without interference by the Profiler. As soon as the system is ready for profiling, you can selectively turn on the collection of required information.

The more settings the Profiler product provides the better you may thus minimize the overhead of monitoring the software system. But this also mandates that you are aware of what exactly you need to know about the software system. If you turn off some options to gather information, maybe you miss exactly those information that would help you to understand the cause of the problems. In particular, if changing these settings dynamically is not possible, it may be cumbersome to restart the Profiler to change the settings and try again.

Another way to minimize the impact of the Profiler itself may be to Isolate Actions and to turn on the profiling only for performing the actions that you would like to analyze isolated from all other actions.

Isolate Actions

Your application has performance problems or memory leaks and you've got a good general impression of the runtime behavior of the whole software system.

How can you track down the reasons for the problems at hand after you got first indications of what they are?

Because a Profiler can give you a wealth of information, you may easily get lost. In particular, if you more or less randomly execute some functions of the software system, you will have a hard time to isolate the problems.

Therefore:

Follow a Divide and Conquer strategy, i.e. perform distinct actions, preferably small steps at a time, and check the outcome of the Profiler after each action or step.

You first need to come up with a sequence of actions during which you assume that the problem takes place, i.e. during which the memory consumption increases significantly or the execution time of the operations is far too long. After each step, check the data the Profiler presents to verify that the step performs as it should be.

By pursuing a divide and conquer strategy, you may start with more coarse grained steps until you find a peculiar step. Then split this step into several smaller steps and repeat these steps until you find that step that is responsible for the problem.

Be aware that caches and background threads may alter the results of successively performing the same operations. If that is possible, disable these caches and background operations while trying to isolate the actions that cause the actual problems.

This pattern can only be applied for actions that can easily be executed repeatedly. In particular if some problems only appear during the system's startup or in the last seconds before the system crashes, it is very difficult to isolate them.

You should also always be aware that you may drill down into the software system at the wrong location. If you realize that you analyze the wrong part of the system, track back and start over. If you do this several times in a row, stop the Profiler and think over your assumptions. It may help to either Find Performance Anomalies or Check for Memory again.

If you are tracking down a performance problem, it may help to Repeat Actions to get more significant numbers. If you need to analyze a memory leak, try to Come Full Circle to more accurately compare memory snapshots.

Repeat Actions

The software system to profile has subtle performance problems. You have some candidate actions in mind that probably cause the problem.

How can you clarify the cause of performance problems?

Even if you are able to isolate suspicious actions, those operations that are actually too slow may not be obvious. It could be, for example, that some operations have a big up-front initialization overhead that tamper the results. It could also be that some operation have a higher intrinsic complexity than others, which does not stick out clearly.

Some software systems do not behave the same every time the same operation is executed. Depending on a lot of different factors, a software system may be slowed down temporarily. The reasons may be worker threads in the background such as the garbage collector, event processing, etc.

Therefore:

Execute the candidate actions multiple times in a row to suppress side effects and to magnify the actions' effects on the system's performance.

If you execute the same action several times in a row, side effects such as background work or initialization effort diminish against the actual complexity of an operation. The more often you repeat an action, the less weight statistical mavericks have.

Repeating an action may be achieved by manually starting the same action over again or by letting the action be executed automatically. If a software system needs to evaluate lines of a text file, for example, you could provide a bigger input file. In particular repeating an algorithmic operation often gives you a good view on its intrinsic complexity.

This pattern can easily be applied if the software system has a graphical user interface with which you may start and restart an operation without significantly changing the state of the system. If you need to perform heavy-weight actions to reset the system and execute again the action, this may already tamper the performance evaluation too much to get meaningful results. This pattern can also not be applied successfully if the cause of the performance problem is part of the initialization work or if the problem cannot be reproduced because it relies on side-effects.

If you repeat actions but still cannot clearly identify the reasons for the performance problems of the software system, you could Stress it or try again to Find Performance Anomalies. It may also help to drill down into the actions that you have repeatedly executed and try to Isolate Actions again.

Come Full Circle

The software system to profile has a memory leak and you have been able to isolate the action that causes the leak.

How can you identify the actual objects that cause a memory leak?

A memory leak appears when some memory has been reserved but not set free after its usage. This means, every memory leak is caused by an operation that reserves memory, holds it, and does not set it free when it is no longer needed. You must therefore find this operation.

Every operation that the software system executes may change the memory consumption. By analyzing an arbitrary snap shot of the memory consumption, it is hard to tell which objects are in regular use and which objects should have been given free earlier.

Therefore:

Find a sequence of actions that includes the offending action and that leaves the software system in the same state as before. Then compare the number of living objects of the same classes and identify those that have been increased but should not have.

If the software system has a graphical user interface you may be able to open a dialog, execute an operation and close the dialog again to free all resources needed by the dialog. If the memory leak appears in a part of the software system for which no graphical user interface exists, you may trigger system jobs that execute the actions and leave the system in the same state as before.

In each case, you need to mark the state of memory consumption of the software system as reference before you start to execute the sequence of actions. After the sequence of actions is executed, you may compare the state of the memory consumption to that reference. Many Profiler tools provide a function to set a marker against which the memory consumption is permanently compared.

To be successful, you need a close understanding of the objects involved in the actions because you need to find those objects that do still exist at the end of the sequence but should not exist any more. Depending on the platform, the programming language, and the settings of the Profiler, you may need to explicitly trigger a garbage collector run to remove all unused objects before you can analyze the memory consumption in detail.

Applying this technique is very difficult if the software system changes its internal state and therefore changes its internal memory consumption during the execution of actions that come full circle. You then need to check very closely which objects may still exist and which objects may not. It might also be necessary to stop any work done in parallel on background threads or by asynchronously started jobs.

If you have found the objects that actually cause the memory leak, you may still not know the reason for their existence. Try to Trace the Roots to find the reason why they have not been removed from memory. If you're stuck because your assumptions about the memory leak have mislead you, try again to Check for Memory.

Trace the Roots

You have found a memory leak, i.e. identified objects that still exist in memory when they should not exist any more.

How do you find the reason why offending objects are still in memory?

It may be straightforward to find objects that are still alive but should not. But the pure existence of these objects does not explain why they still exist. There may be many places in the source code where these objects have been created, they may have been passed as parameters to many methods, and they may be referenced from many other objects.

Even if you have identified the objects by coming full circle, maybe many steps were necessary to return the system to the same state as in the beginning. Manually introspecting the source code that has been executed by all of these steps may just not be feasible.

Therefore:

Create a snapshot that includes all living objects and pick a single object that should not exist any more. From this object on follow the incoming object references until you reach the root object that is responsible to hold the whole chain of objects.

To apply this pattern, you need a Profiler tool that has a graphical view on the network of interconnected objects of a memory snapshot. This means, you should be able to choose any living object and expand all of its incoming object references graphically. You need to recursively check the incoming references of all referring objects until you find an object that is valid to exist.

If your technological platform incorporates a garbage collector there are always garbage collector roots, i.e. static objects that may never be removed. All objects that can be reached by object reference chains from these roots are also never removed from memory. You therefore first need to identify the chain of object references from the offending object backwards to a garbage collector root. Then you need to analyze this chain from the garbage collector root on to find the first object reference that should not exist any more. For example, on the object reference chain, there may be a list that should be empty but still contains object references. The actual cause of the memory leak in this case is the code that has not properly emptied or disposed the list.

Manually searching for garbage collector roots may be a very challenging task. Some Profilers provide an option that performs the search for the garbage collector roots automatically. Using such an option, the Profiler tool presents the chains of object references you are looking for. Note that quite often there is not only one such chain but several such chains. You should therefore always first search for a single chain, analyze it, discover the actual bug, fix the bug, and restart both the application and the Profiler to determine whether there were multiple causes that need to be fixed separately.

The fewer connections the software system has at runtime, the easier it is to use this technique. Having a clearly structured system with defined dependencies between internal layers helps a lot to cut down on the number of interconnected objects and therefore significantly reduces the effort to find garbage collector roots. Some software systems, in particular rich or fat client applications typically have a huge number of object references. Manually searching for garbage collector roots is very difficult in these cases.

Stress it

Your software system behaves well when you test it but loses performance under high load.

How do you find performance problems that do not appear when your software system runs in normal operation?

Some performance problems are caused by ill-designed algorithms. These problems can typically be found more or less easily by Repeating Actions. Once you have picked these low-hanging fruits you need to address performance problems that do not always manifest because they may be the results of many interconnected causes that only appear under high load.

You could try to Profile the Real Thing to find the performance problems in the real production system. But quite often this is not feasible: The system may not be ready for production yet, or it is too critical to risk profiling it in the production environment. But without real users, the system may still behave nicely.

Therefore:

Employ a tool that artificially creates a high load on your software system by simulating multiple simultaneous user actions while you profile the system.

There are many tools available to stress test a software system, may it be a rich-client or a web-client application or a system without a graphical user interface. The common denominator of all of these tools is that they are able to simulate the behavior of typical users and that they provide the option to easily scale the number of simultaneous user requests on your system.

Using such a tool, you must first try to identify the typical behavior of the users of your software system. Analyzing the logs of the system or just asking some users may give you an impression of their typical usage. Then you need to simulate and automate the user actions so that they can be replayed by the tool. By slowly increasing the number of simulated simultaneous users you may find the threshold from which on the system does not behave as desired any more.

This technique may reveal performance problems that only manifest in the production system otherwise. However, it cannot reproduce all problems. Some problems are caused not only by a high system load from many users in parallel but by specific properties of the production environment such as the operation system or the server hardware. In these cases, you should Profile the Real Thing or Clone Production and stress test the software system running in a production environment.

If you simulate too many distinct user actions at once, it may be difficult to track down the causes of the problems under high load. In that case you could Isolate Actions and Stress It again, now executing fewer actions at the same time.

Profile the Real Thing

Your software system suffers from performance problems in the production environment.

How do you find performance problems that you cannot reproduce on a local developer's machine?

Setting up a Profiler on a local developer's machine is easy and straightforward in most cases. Still, most often a local machine is configured differently from a production environment, i.e. on the production environment a different operation system may be installed, operation system settings may differ, or there may be more memory or disk space available.

Furthermore, some problems may be caused by the environment in which the production machine runs, in particular causing slow network connections, latency problems, or blocked reverse DNS lookups. All of these differences may be the reason for performance problems that do not manifest elsewhere.

Therefore:

Take on the effort to install and set up a Profiler tool in the production environment and run your tests there.

There are two options to perform profiling tests on the production environment. The more intrusive option is to locally start a Profiler tool and to let it remotely connect to the productive software system. This allows you to deeply analyze the behavior of the software system while it is running.

The less intrusive option is to install a Profiler tool on the production machine that runs there locally, measures the productive software system, and stores information about the runtime behavior in the local file system, for example in flat files. This means, the profiling information is not evaluated at runtime. Instead, you need to periodically get the profiling information and start the graphical user interface of your Profiler tool on your local machine to analyze the collected information. While this option is easier to sell to operators, it prevents you from selectively performing actions and analyzing the systems' behavior.

This advice probably is the most difficult to follow because in many companies there is a strict separation between developing and operating a software system. You may need to convince managers from other teams to allow you to profile your software system in their production environment and you may need to convince the operators whose support you need to actually run any tests. Both tasks may be impossible to achieve. Still, only in the production environment you may be able to analyze problems that appear exclusively there.

After successfully setting up the Profiler tool in the production environment, you should Find Performance Anomalies to get an impression of the runtime behavior of your software system on the production environment. As alternative to profiling the software system in the production environment, in particular if you are not allowed to install a Profiler tool there, you may try to Clone Production. If the performance problems are caused by high load rather than by specific settings of the production environment, stay with profiling a development system and Stress it.

Clone Production

Your software system suffers from performance problems in the production environment.

How can you profile the software system when you must not install a Profiler tool in the production system?

Some problems only appear in the production environment and cannot be reproduced on a local developer's machine. Profiling the software system locally therefore does not help, profiling in the production environment, on the other hand, is not allowed.

You could try to use profiling techniques that do not depend on a Profiler tool, for example gathering as many information as possible in log files. Although it is generally a good idea to write as many relevant information as possible into log files, you cannot retrieve everything of interest from inside the application itself. Besides, changing existing code that has been tested to gather profiling information may not be a good idea as you may have to test again the complete software system. Also, you may not be able to deploy changed code at will but only according to a release plan.

Therefore:

Set up a dedicated test environment that is a clone of the production environment, i.e. that runs on the same hardware and uses the same operation system settings, and profile the software system there.

To set up a test environment, you probably need management support. Typically, the test environment cannot be set up by the development team and setting up a clone of the production environment is expensive, in particular if exactly the same hardware should be used as in the production environment.

To reduce the costs of cloning the production environment, you could try to scale down the test environment without changing the overall characteristics of the system. You could achieve this, for example, by using fewer processors (but still having more than one) or by reducing the available memory.

A clone of the production environment has many benefits other than easier profiling. Most development projects already have distinct environments for development, test, and production. In that case, you should employ the existing test environment for profiling. You probably need to coordinate any profiling tests efforts with functional test efforts carried out by the test team.

A common problem in cloning the production system is the availability of production data. As this data must often be protected from public access, it might be necessary to create an anonymous clone of production data that obfuscates the original context.

The biggest disadvantage of cloning the production system is the effort of keeping the clone in sync. After the initial effort to set up the cloned environment, you need to reflect all changes applied to the production environment. If not, you may not be able rely on the results from testing on the cloned system any more.

After successfully setting up the Profiler tool in the cloned environment, you should Find Performance Anomalies. Cloning the production system alone may not reveal the problems of the software system if these problems do only appear under high load. In that case, Stress it.

Unfinished Patterns

There are more patterns on how to efficiently profile a software system than have been presented so far in this paper. This section gives an overview of patterns that have not yet been elaborated in detail.

Reduce Moving Parts

How do you reliable measure the current performance of your software system?	Stop all background threads and schedulers that might start new threads.

Design for Profiling

How do you facilitate profiling your software system?	Design the system in such a way that profiling parts of it independently becomes possible, for example by employing a layered and component-based architecture and by making it possible to stop background work.

Have a Mental Map

How do you improve your ability to quickly find performance problems and memory leaks?	Try to always have a mental map of the complete system and compare the results of all profiling operations with your expected outcome.

Let the System Warm Up

How do you avoid side-effects when measuring the system performance?	Let the system warm up before you start any measurements so that, for example, all caches are filled with data.

Act as a User Would Do

How can you increase the chance to detect performance problems and memory?	Operate the system as a real user would do, i.e. execute complete use cases without following shortcuts.

Bring Real Users in

How can you increase the chance to detect performance problems and memory leaks if you don't know what the users are doing?	Bring in real users and let them operate on your test system while you closely watch the system's behavior.

Strangle the System

How can you stress your system if it still works quite well under high load or if you cannot produce a very high load?	Limit the resources that are available to your software system, for example, by removing memory or CPUs.

Examples

To relate the given patterns to the real world, this section gives some examples of how the patterns can be applied. The first example shows how to track down memory leaks, the second how to find performance bottlenecks.

Both examples are based on real, open-source applications. Please note that the bugs that are going to be found in these applications have been artificially introduced for the purpose of this section. They have never existed in the original distributions.

Tracking down a Memory Leak

The first example is based on the application *FreeMind* (freemind.sf.net), which is a free and open-source mind mapping tool. FreeMind is a fat client application, developed in Java.

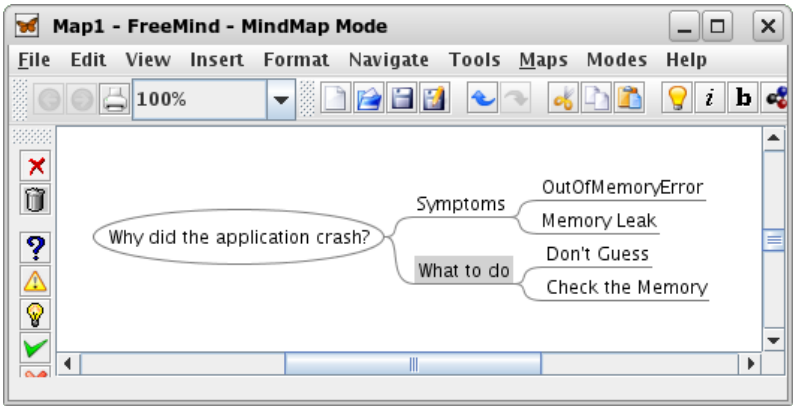
Assume that you are the developer of FreeMind and that you've just got a bug report from a user. The user complains that the application crashes after editing a mind map for some time. This seems to be a serious problem, so you decide to track down its cause.

THINK ABOUT IT FIRST

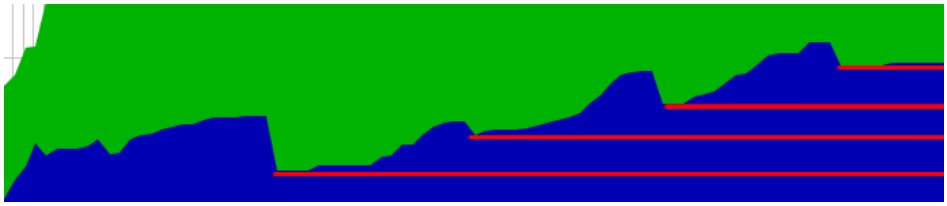
The user also sent you a stack trace of the application that shows an `OutOfMemoryError`, which is an indication that the application probably suffers from a memory leak. You know that there are some cases where you have not properly cleaned up objects before disposing them. But instead of guessing, you decide to start the Profiler to have a closer look to avoid introspecting and maybe changing code that is not responsible for the problem at hand.

CHECK FOR MEMORY

As the user did not report a specific action that he or she performed just before the crash, you decide to first get a general impression of the runtime behavior of the application. Started under the control of the Profiler, the application comes up and you begin to draw a mind map.



You haven't noticed any problems so far, the application runs smoothly; the problem does not seem to appear immediately. However, a look at the memory consumption reveals that something went wrong.



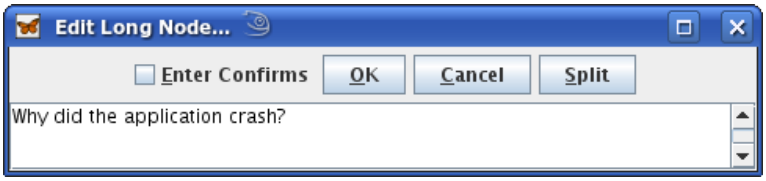
For a reason you don't know yet, the memory consumption grew steadily. The garbage collector ran several times, freeing resources that were not needed any more, but after each run, the used memory increased. To get any further, you need to understand better which action causes the leak.

ISOLATE ACTION

Therefore, you start to selectively perform actions to edit the mind map and closely check the memory consumption after each step. To check the actual memory consumption at a time, you trigger the garbage collector after each step. After a short time you suspect that editing an existing node of the mind map might cause the problem. But you need to verify this assumption.

COME FULL CIRCLE

You assume that the dialog to edit a node might be the cause of the problem. To reliably check the memory consumption of editing a node, you decide to measure the difference before opening and after closing that dialog.



In order to compare the memory consumption, you first trigger another garbage collection run and then mark the current values. After that you open the dialog, edit some text and close it. After triggering another garbage collection run, you have a close look at which objects do now exist that did not exist earlier on.

Among many other objects that don't look suspicious you discover that there are still references to the dialog class that you've just used to edit the node.

sun.awt.X11.XDialogPeer	13	+1 3.744 by...
java.lang.Object	1.353	+1 10.824 b...
javax.swing.JViewport$ViewListener	17	+1 272 bytes
javax.swing.JDialog	13	+1 4.888 by...
sun.awt.X11.PropMwmHints	15	+1 360 bytes
sun.awt.X11.XInputMethod	15	+1 960 bytes
freemind.view.mindmapview.EditNodeDialog$1	13	+1 312 bytes
freemind.view.mindmapview.EditNodeDialog$5	13	+1 416 bytes
sun.awt.X11.XContentWindow	15	+1 1.920 by...
javax.swing.JToggleButton$ToggleButtonModel	21	+1 840 bytes
freemind.view.mindmapview.EditNodeDialog$6	13	+1 208 bytes

In particular, there is now one object more of the class than before. Now you know the reason for the memory leak: a dialog object has not been removed from memory after the dialog window has been closed. Nevertheless, you don't know yet why this happens.

TRACE THE ROOTS

To further understand the problem, you decide to analyze the object graph to find out which objects still hold references to the dialog object. Instead of manually checking all incoming object references to a dialog object, you utilize the Profiler's function to search for garbage collector roots. After a short time, the Profiler shows the first path to such a root.

The graph tells you that the dialog still exists in memory because it is referenced by its native peer object. Because you know the basics about developing a dialog with Java's GUI library Swing, you are sure that somewhere in your code, you forgot to properly dispose the dialog after it is closed.

You switch back to your IDE, open the source code of the dialog class into an editor and find the right spot where you made the dialog invisible instead of properly disposing it.

```
okButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    eventSource.setValue(BUTTON_OK);
    dialog.setVisible(false);
  }
});
```

You are relieved to have found the bug so quickly, immediately create a new version of the application and send it to the user that reported the bug.

Discovering Performance Bottlenecks

The second example is based on the application *blojsom* (blojsom.sf.net), which is a free and open-source blog software. Blojsom is a web application, developed in Java.

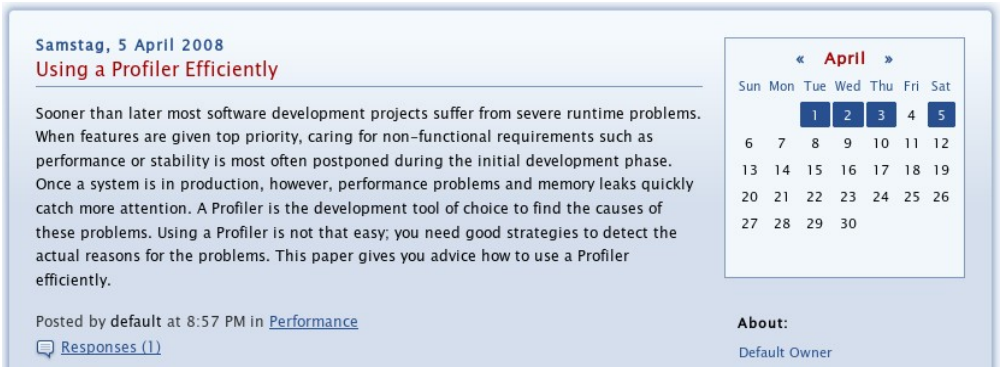
Once again, please assume that you are a developer of blojsom. You have worked hard to finish a new version and are almost ready to publish it. As a last step, you use the software as a normal user would do to find any obvious bugs that have gone unnoticed before.

THINK ABOUT IT FIRST

The application seems to be slower than usual. Maybe some changes you did have deteriorated the performance. Maybe it's just because of the new features that you implemented. One feature in particular could be improved by adding a cache to prevent some unnecessary database calls from happen. But because you want to avoid unnecessary work, you decide to have a closer look before making any changes.

FIND PERFORMANCE ANOMALIES

You set up a new and clean database without any prior blog entries and start up the web server. Still without using the profiler, you create a few categories on the admin pages and then write some blog entries and comments.



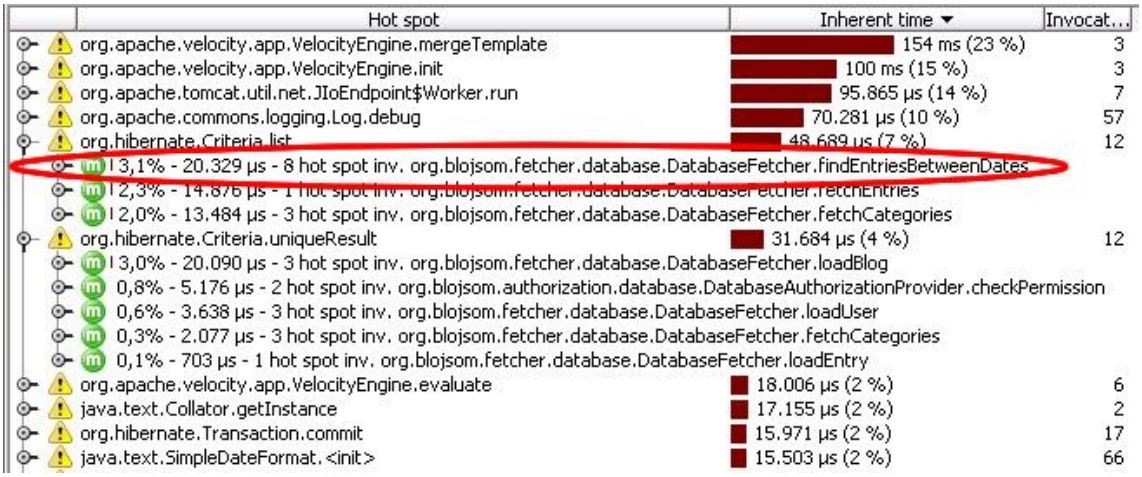
You realize that while the administration pages work as usual, blog pages seem to load slower than before.

Minimize the Profiler's Overhead

Because the Profiler tool may affect the runtime performance of the application, you want to minimize the side effects of using the Profiler tool. You start the tool and change settings so that the Profiler only records performance data and nothing else. You are quite sure that the problems are not caused by the initialization of the application; therefore you start the server with no profiling at all and wait until the application is properly initialized.

Isolate Action

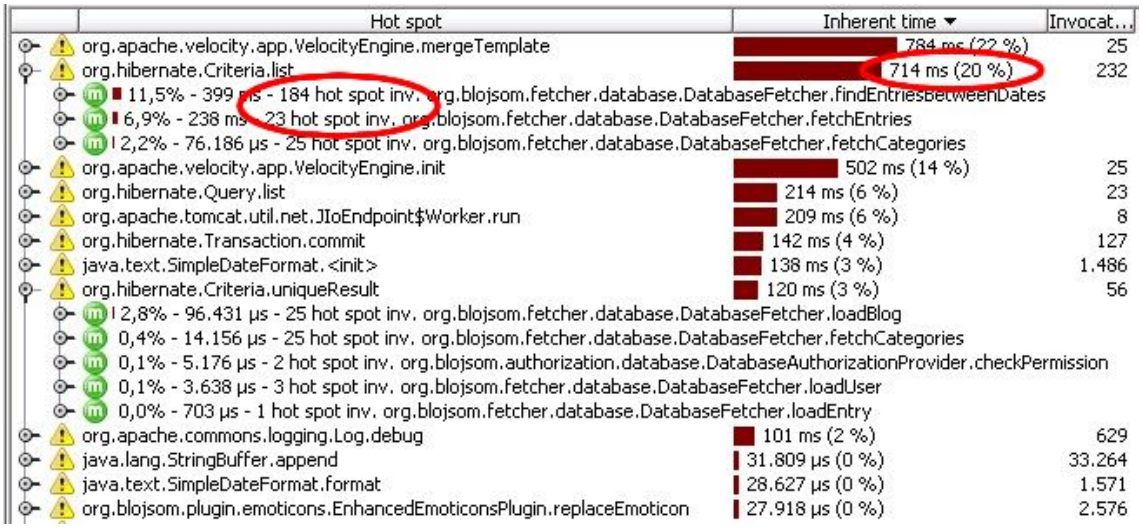
You don't want to get lost in too many details too soon and therefore decide to perform several distinct actions in a row and to then look at the information the Profiler has collected meanwhile: you create a new blog entry, look at the entry, and write a comment. After these operations, you open a view in which the Profiler tool shows those methods that took the most execution time.



On the first look, everything seems right. You are wondering however why the database calls (`Criteria.list`) from the method `findEntriesBetweenDates` took to long. You know that this method is called from the calendar view on the right hand side of the blog page.

Repeat Action

To verify your assumption, you reload the main blog page a couple of times by constantly pressing F5 in the browser. The rendering of a blog page is not affected by any data caching so that each call should cause the same number of database calls. Then, you have another look at the same view of the Profiler tool.



You notice that the amount of time spend in `Criteria.list` has relatively increased a lot. You also notice that for each page request, this method is called once from `fetchEntries` but much more often from `findEntriesBetweenDates`. This does not seem to be right.

You start your IDE, open the respective source file, navigate to the method `findEntriesBetweenDates`, and immediately find the following code:

```
List entryList = entryCriteria.list();

DatabaseEntry[] entries = (DatabaseEntry[]) entryList.toArray(new DatabaseEntry[entryList.size()]);
for (int i = 0; i < entries.length; i++) {
  Criteria categoryCriteria = session.createCriteria(Category.class);
  categoryCriteria.add(Restrictions.eq("name", entries[i].getCategory()));
  List categoryList = categoryCriteria.list();
  // TODO: Visiting categories probably not needed
}
```

Suddenly you realize that you began to implement a feature for which you needed to visit the category objects of all blog entries. You did not finish developing this function but the code that you left causes another database round trip for each blog entry. Besides the fact that this code is written very inefficiently, it does not even make any sense right now.

So you remove the code and take another look at the performance measures from your Profiler tool. Now that everything seems right you publish the new version of the application.

Acknowledgements

I would like to thank Sachin Bammi who gave important feedback as my shepherd for the EuroPLoP 2008 conference. I'd also like the thank the participants of the EuroPLoP 2008 workshop for their well thought-out and constructive comments and suggestions. In particular the section with unfinished patterns is based on their input. I hope to write a follow-up with the new material in more detail soon.

Resources

As far as the author is aware of there is no related work in pattern form that explains how to profile a software system. This section therefore presents references to other resources that explain the usage of Profilers in a more general form.

[1] List of profiling tools for Java:
http://www.javaperformancetuning.com/resources.shtml

[2] Jim Patrack, Handling memory leaks in Java programs:
http://www.ibm.com/developerworks/java/library/j-leaks/

[3] Brian Goetz, Java theory and Practice: Plugging memory leaks with weak references,
http://www.ibm.com/developerworks/java/library/j-jtp11225/index.html

[4] Tess Ferrandez: If broken it is, fix it you should. A blog about debugging and profiling .NET applications, http://blogs.msdn.com/tess/default.aspx