Chapter 1

Profiling Applications

Since the beginning of software engineering, programmers have faced issues related to application performance. Although there has been a great improvement on the programming environment to support better and faster development process, addressing performance issues when programming still requires quite some dexterity.

In principle, optimizing an application is not particularly difficult. The general idea is to make slow and frequently called methods either faster or less frequently called. Note that optimizing an application usually complexifies the application. It is therefore recommended to optimize an application only when the requirements for it are well understood and addressed. In other term, you should optimize your application only when you are sure of what it is supposed to do. As Kent Beck famously formulated: *1 - Make It Work, 2 - Make It Right, 3 - Make It Fast.*

1.1 What does profiling mean?

Profiling an application is a term commonly employed that refers to obtaining dynamic information from a controlled program execution. The obtained information is intended to provide important hints on how to improve the program execution. These hints are usually numerical measurements, easily comparable from one program execution to another.

In this chapter, we will consider measurement related to method execution time and memory consumption. Note that other kind of information may be extracted from a program execution, in particular the method call graph.

It is interesting to observe that a program execution usually follows the

universal 80-20 rule: only a few amount of the total amount of methods (let's say 20%) consume the largest part of the available resources (80% of memory and CPU consumption). Optimizing an application is essentially a matter of tradeoff therefore. In this chapter we will see how to use the available tools to quickly identify these 20% of methods and how to measure the progress coming along the program enhancements we bring.

Experience shows that having unit tests is essential to ensure that we do not break the program semantics when optimizing it. When replacing an algorithm by another, we ought to make sure that the program still do what it is supposed to do.

1.2 A simple example

Consider the method Collection>>select:thenCollect:. For a given collection, this method selects elements using a predicate. It then applies a block function on each selected element. At the first sight, this behavior implies two runs over the collections: the one provided by the user of select:thenCollect: then an intermediate one that contains the selected elements. However, this intermediate collection is not indispensable, since the selection and the function application can be performed with only one run.

The method timeToRun. Profiling one program execution is usually not enough to fully identify and understand what has to be optimized. Comparing at least two different profiled executions is definitely more fruitful. The message timeToRun may be sent to a bloc to obtain the time in milliseconds that it took to evaluate the block. To have a meaningful and representative measurement, we need to "amplify" the profiling with a loop.

Here are some results:

```
| coll |
coll := #(1 2 3 4 5 6 7 8) asOrderedCollection.
[ 100000 timesRepeat: [ (coll select: [:each | each > 5]) collect: [:i |i * i]]] timeToRun
"Calling select:, then collect: --> ~ 570 - 600 ms"

| coll |
coll := #(1 2 3 4 5 6 7 8) asOrderedCollection.
[ 100000 timesRepeat: [ coll select: [:each | each > 5] thenCollect:[:i |i * i]]] timeToRun
"Calling select:thenCollect: --> ~ 397 - 415 ms"
```

Although the difference between these two executions is only about few hundred of milliseconds, opting for one method instead of the other could significantly slow your application!

Let's scrutinize the definition of select:thenCollect:. A naive and non-optimized implementation is found in Collection. (Remember that Collection is the root class of the Pharo collection library). A more efficient implementation is defined in OrderedCollection, which takes into account the structure of an ordered collection to efficiently perform this operation.

```
Collection>>select: selectBlock thenCollect: collectBlock
   "Utility method to improve readability."

   ^ (self select: selectBlock) collect: collectBlock
```

```
OrderedCollection>>select: selectBlock thenCollect: collectBlock
   " Utility method to improve readability.
   Do not create the intermediate collection. "

   | newCollection |
   newCollection := self copyEmpty.
   firstIndex to: lastIndex do: [:index |
      | element |
      element := array at: index.
      (selectBlock value: element)
         ifTrue: [ newCollection addLast: (collectBlock value: element) ]].
   ^ newCollection
```

As you have probably guessed already, other collections such as Set and Dictionary do not benefit from an optimized version. We leave as an exercise an efficient implementation for other abstract data types. As part of the community effort, do not forget to submit your contribution to Pharo if you come up with an optimized and better version of select:thenCollect: or other methods. The Pharo team really value such effort.

The method bench. When sent to a block, the bench message estimates how many times this block is evaluated per second. For example, the expression [1000 factorial] bench says that 1000 factorial may be executed approximately 350 times per second.

1.3 Code profiling in Pharo

The timeToRun method is useful to tell how long an expression takes to be executed. But it is not really adequate to understand how the execution time is distributed over the computation triggered by evaluating the expression. Pharo comes with MessageTally, a code profiler to precisely analyze the time distribution over a computation.

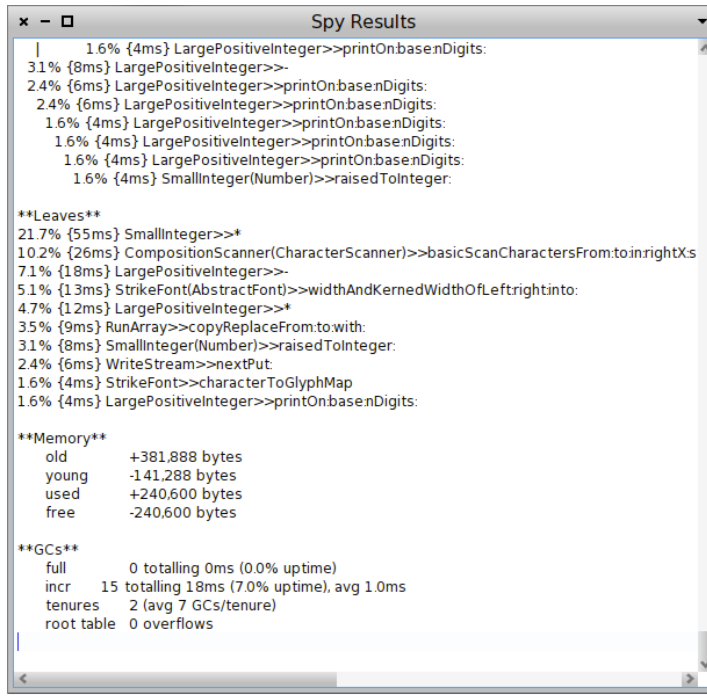

Figure 1.1: MessageTally in action.

MessageTally

MessageTally is implemented as a unique class having the same name. Using it is quite simple. A message spyOn: needs to be sent to MessageTally with a block expression as argument to obtained a detailed execution analysis. Evaluating MessageTally spyOn: [*"your expression here"*] opens a window that contains the following information:

1. a hierarchy list showing the methods executed with their associated execution time during the expression execution.

2. leaf methods of the execution. A leaf method is a method that does not invoke other methods (*e.g.*, primitive, accessors).

3. statistic about the memory consumption and garbage collector involvement.

Each of these points will be described later on.

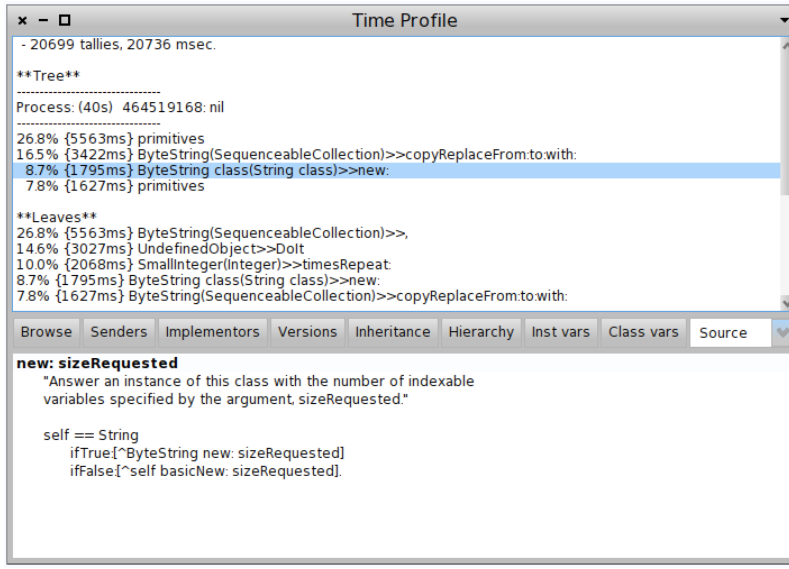

Figure 1.2: TimeProfiler uses MessageTally and navigates in executed methods.

Figure 1.1 shows the result of the expression `MessageTally spyOn: [20 timesRepeat: [Transcript show: 1000 factorial printString]]`. The message `spyOn:` executes the provided block in a new process. The analysis focuses on one process, only, the one that executes the block to profile. The message `spyAllOn:` profiles all the processes that are active during the execution. This is useful to analyze the distribution of the computation over several processes.

A tool a bit less crude than `MessageTally` is `TimeProfileBrowser`. It shows the implementation of the executed method in addition (Figure 1.2). `TimeProfileBrowser` understand the message `spyOn:`. It means that in the below source code, MessageTally can be replaced with TimeProfileBrowser to obtain the better user interface.

Integration in the programming environment

As shown previously, the profiler may be directly invoked by sending `spyOn:` and `spyAllOn:` to the `MessageTally` class. It may be accessed through a number of additional ways.

Via the World menu. The World menu (obtained by clicking outside any Pharo window) offers some profiling facilities under the `System` submenu

(Figure 1.3). Start profiling all Processes creates a block from a text selection and invokes spyAllOn:. The entry Start profiling UI profiles the user interface process. This is quite handy when debugging a user interface!

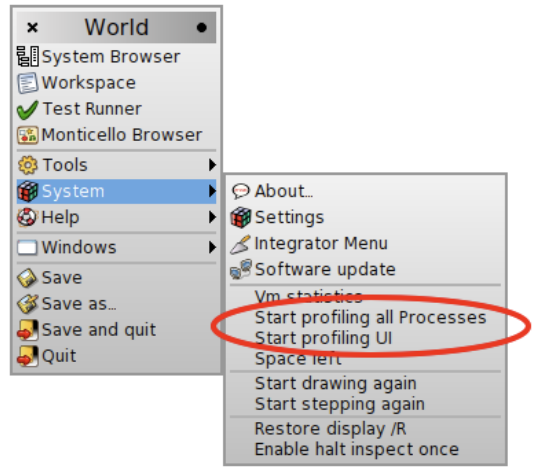

Figure 1.3: Access by the menu.

Via the Test Runner. As the size of an application grows, unit tests are usually becoming a good candidate for code profiling. Running tests often is rather tedious when the time to run them is getting too long. The Test Runner in Pharo offers a button Run Profiled (Figure 1.4).

Pressing this button runs the selected unit tests and generates a message tally report.

1.4 Read and interpret the results

The message tally profiler essentially provides two kinds of information:

- execution time is represented using a tree representing the profiled code execution (**Tree**. Each node of this tree is annotated with the time spent in each leaf method (**Leaves**).
- memory activity contains the memory consumption (**Memory** and the garbage collector usage (**GC**).

For illustration purpose, let us consider the following scenario: the string character 'A' is cumulatively appended 9 000 times to an initial empty string.

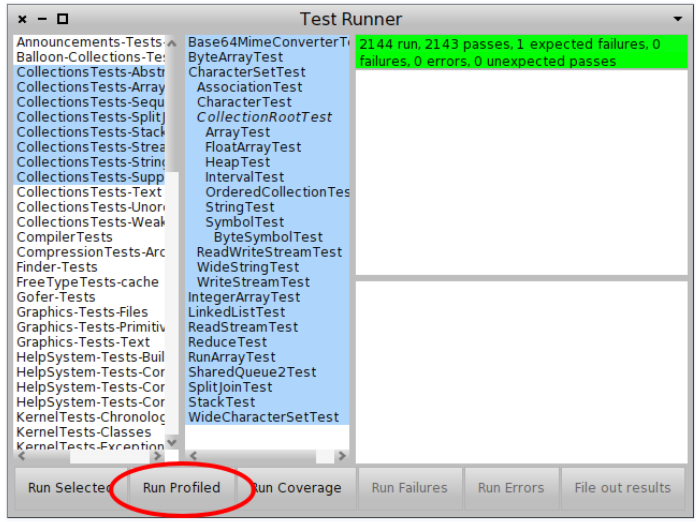

Figure 1.4: Button to generate a message Tally in the TestRunner.

```
MessageTally spyOn:
    [ 500 timesRepeat: [
                | str |
                str := ''.
                9000 timesRepeat: [ str := str, 'A' ]]].
```

The complete result is:

```
 - 24038 tallies, 24081 msec.

**Tree**
--------------------------------
Process: (40s)  535298048: nil
--------------------------------
29.7% {7152ms} primitives
11.5% {2759ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
  5.9% {1410ms} primitives
  5.6% {1349ms} ByteString class(String class)>>new:

**Leaves**
29.7% {7152ms} ByteString(SequenceableCollection)>>,
9.2% {2226ms} SmallInteger(Integer)>>timesRepeat:
5.9% {1410ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.6% {1349ms} ByteString class(String class)>>new:
4.4% {1052ms} UndefinedObject>>DoIt
```

```
**Memory**
   old        +0 bytes
   young      +9,536 bytes
   used       +9,536 bytes
   free       −9,536 bytes

**GCs**
   full          0 totalling 0ms (0.0% uptime)
   incr       9707 totalling 7,985ms (16.0% uptime), avg 1.0ms
   tenures    0
   root table  0 overflows
```

The first line gives the overall execution time and the number of samplings (also called *tallies*, we will come back on sampling at the end of the chapter).

Tree: Cumulative information

The **Tree** section represents the execution tree per processes. The tree tells the time the Pharo interpreter has spent in each method. It also tells the different invocation using a call graph. Different execution flows are kept separated according to the process in which they have been executed. The process priority is also displayed, this helps distinguishing between different processes. The example tells:

```
**Tree**
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
Process: (40s)  535298048: nil
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
29.7% {7152ms} primitives
11.5% {2759ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
 5.9% {1410ms} primitives
 5.6% {1349ms} ByteString class(String class)>>new:
```

This tree shows that the interpreter spent 29.7% of its time by executing primitives. 11.5% of the total execution time is spent in the method SequenceableCollection>>copyReplaceFrom:to:with:. This method is called when concatenating character strings using the message comma (,), itself indirectly invoking new: and some virtual machine primitives.

The execution takes 11.5% of the execution time, this means that the interpreter effort is shared with other processes. The invocation chain from the code to the primitives is relatively short. Reaching hundreds of nested calls is no exception for most of applications. We will optimize this example later on.

Two primitives are listed as tree leaves. These correspond to different primitives. Unfortunately, MessageTally does not tell exactly which primitive has been invoked.

Leaves: leaf methods

The ** Leaves** part lists *leaf methods* for the code block that has been profiled. A leaf method is a method that does not call other methods. More exactly, it is a method m for which no method invoked by m have been "detected". This is the case for variable accessors (*e.g.*, Point»x), primitive methods and methods that are very quickly executed. For the previous example, we have:

```
**Leaves**
29.7% {7152ms} ByteString(SequenceableCollection)>>,
9.2% {2226ms} SmallInteger(Integer)>>timesRepeat:
5.9% {1410ms} ByteString(SequenceableCollection)>>copyReplaceFrom:to:with:
5.6% {1349ms} ByteString class(String class)>>new:
4.4% {1052ms} UndefinedObject>>DoIt
```

Memory

The statistical part on memory consumption tells the observed changes on the quantity of memory allocated and the garbage collector usage. To fully understand this information, one needs to keep in mind that Pharo's garbage collector (GC) is a scavenging GC, relying on the principle that an old object has greater change to live even longer. It is designed following the fact that an old object will probably be kept referenced in the future. On the contrary, a young object has greater change to be quickly dereferenced.

Several memory zones are considered and the migration of a young object to the space dedicated for old object is qualified as tenured. (Following the metaphor of American academic scientists, when a permanent position is obtained.)

An example of the memory analyze realized by MessageTally:

```
**Memory**
   old       +0 bytes
   young     +9,536 bytes
   used      +9,536 bytes
   free      -9,536 bytes
```

MessageTally describes the memory usage using four values:

1. the old value is about the grow of the memory space dedicated to old objects. An object is qualified as "old" when its physical memory location is in the "old memory space". This happens when a full garbage

collector is triggered, or when there are too many object survivors (according to some threshold specified in the virtual machine). This memory space is cleaned by a full garbage collection only. (An incremental GC does not reduce its size therefore).

An increase of the old memory space is likely to be due to a *memory leak*: the virtual machine is unable to release memory, promoting young objects as old.

2. the young value tells about the increase of the memory space dedicated to young objects. When an object is created, it is physically located in this memory space. The size of this memory space changes frequently.
3. the used value is the total amount of used memory.
4. the free value is the remaining amount of memory available.

In our example, none of the objects created during the execution have been promoted as old. 9 536 bytes are used by the current process, located in the young memory space. The amount of available memory has been reduced accordingly.

GCs

The **GCs** provides statistics about the garbage collector. An example of a garbage collector report is:

```
**GCs**
  full          0 totalling 0ms (0.0% uptime)
  incr       9707 totalling 7,985ms (16.0% uptime), avg 1.0ms
  tenures       1 (avg 9707 GCs/tenure)
  root table  0 overflows
```

Four values are available.

1. The full value totals the amount of full garbage collections and the amount of time it took. Full garbage collection are not that frequent. They results from often allocating large memory chunks.
2. The incr is about incremental garbage collections. Incremental GCs are usually frequent (several times per second) and quickly performed (about 1 or 2 ms). It is wise to keep the amount of time spent in incremental GCs below 10%.
3. The number of tenures tells the amount of objects that migrated to the old memory space. This migration happens when the size of the young memory space is above a given threshold. This typically happens

when launching an application, when all the necessary objects haven't been created and referenced.

4. The root table overflows is the amount of root objects used by the garbage collector to navigate the image. This navigation happens when the system is running short on memory and need to collect all the objects relevant for the future program execution. The overflow value identifies the rare situation when the number of roots used by the incremental GC is greater than the its internal table. This situation forces the GC to promote some objects as tenured.

In the example, we see that only the incremental GC is used. As we will subsequently see, the amount of created objects is quite relevant when one wants to optimize performances.

1.5 Illustrative analysis

Understanding the result obtained when profiling is the very first step when one wants to optimize an application. However, as you probably started to feel, understanding why a computation is costly is not trivial. Based on a number of examples, we will see how comparing different profiling results greatly helps to identify costly message calls.

The method "," is known to be slow since it creates a new character string and copy both the receiver and the argument into it. Using a Stream is a significant faster approach to concatenate character strings. However, nextPut: and nextPutAll: must be carefully employed!

Using a Stream for string concatenation. At the first glance, one could think that creating a stream is costly since it is frequently used with relatively slow inputs and outputs (*e.g.*, network socket, disk accesses, Transcript). But replacing the string concatenation employed in the previous example by a stream operation is almost 10 times faster! This is easily understandable since concatenating 9000 times a character strings creates 8999 intermediately objects, each being filled with the content of another. Using a stream, we simply have to append a character at each iteration.

```
MessageTally spyOn:
   [ 500 timesRepeat: [
                | str |
                str := WriteStream on: (String new).
                9000 timesRepeat: [ str nextPut: $A ]]].
```

```
 - 807 tallies, 807 msec.

**Tree**
--------------------------------
Process: (40s)  535298048: nil
--------------------------------

**Leaves**
33.0% {266ms} SmallInteger(Integer)>>timesRepeat:
21.2% {171ms} UndefinedObject>>DoIt

**Memory**
   old        +0 bytes
   young      -18,272 bytes
   used       -18,272 bytes
   free       +18,272 bytes

**GCs**
   full          0 totalling 0ms (0.0% uptime)
   incr          5 totalling 7ms (3.0% uptime), avg 1.0ms
   tenures       0
   root table    0 overflows
```

String preallocation. Using OrderedCollection without a preallocation of the collection is well known for being costly. Each time the collection is full, its content has to be copied into a larger collection. Carefully choosing a preallocation has an impact of using ordered collections. The message new: aNumber could be used instead of new.

```
MessageTally spyOn:
   [ 500 timesRepeat: [
                | str |
                str := WriteStream on: (String new: 9000).
                9000 timesRepeat: [ str nextPut: $A ]]].
```

For this example, it is possible to improve the script by using the method atAllPut:. The script below takes only a couple of milliseconds.

```
MessageTally spyOn:
   [ 500 timesRepeat: [
      | str |
      str :=String new: 9000.
      str atAllPut: $A ]].
```

An experiment. Doing benchmarks shines when different executions are compared. In the previous piece of code, replacing the value 9000 by 500 is

valuable. The time taken with 9000 iterations is 2.7 times slower than with 500. Using the string concatenation (*i.e.*, using the , method) instead of a stream widens the gap with a factor 10. This experiment clearly illustrates the importance of using appropriate tools to concatenate strings.

The time of the profiled execution is also an important quality factor for the result. MessageTally employs a sampling technique to profile code. Per default, MessageTally samples the current executing thread each millisecond per default. It is therefore necessary that all the methods involved in the computation are executed a "fair" amount of time to appear in the result report. If the application to profile is very short (few milliseconds only), then executing it a number of times help improving the accuracy of the report.

1.6 Counting messages

The profiling we have realized so far is focused on method execution time. The advantage of method call stack sampling is that it has a relatively low impact on the execution. The disadvantage is the relatively imprecision of the result. Even though the obtained results are sufficient in most of the case, they are always an approximation of the real execution.

MessageTally allows for a profiling based on program interpretation. The idea is to use a bytecode interpreter instead of execution sampler. The main advantage is the exactness of the result. The information obtained with the message tallySends: indicates the amount of time each method involved in a computation has been executed. Figure 1.5 gives the result obtained by executing

```
MessageTally tallySends:[ 1000 timesRepeat:  [3.14159 printString]].
```

The downside of tallySend: is the time taken to execute the provided block. The block to profile is executed by an interpreter written in Pharo, which is slower then the one of the virtual machine. A piece of code profiled by tallySends: is about 200 times slower. The interpreter is available from the method ContextPart»runSimulated: aBlock contextAtEachStep: block2.

1.7 Memorized Fibonacci

As a small application of the techniques we have seen, consider the Fibonacci function ($fib(n) = fib(n-1) + fib(n-2)$ with $fib(0) = 0, fib(1) = 1$). We will study two versions of it: a recursive version and a memorized version. Memoizing consists in introducing a cache to avoid redundant computation.

Consider the following definition, close to the mathematical definition:

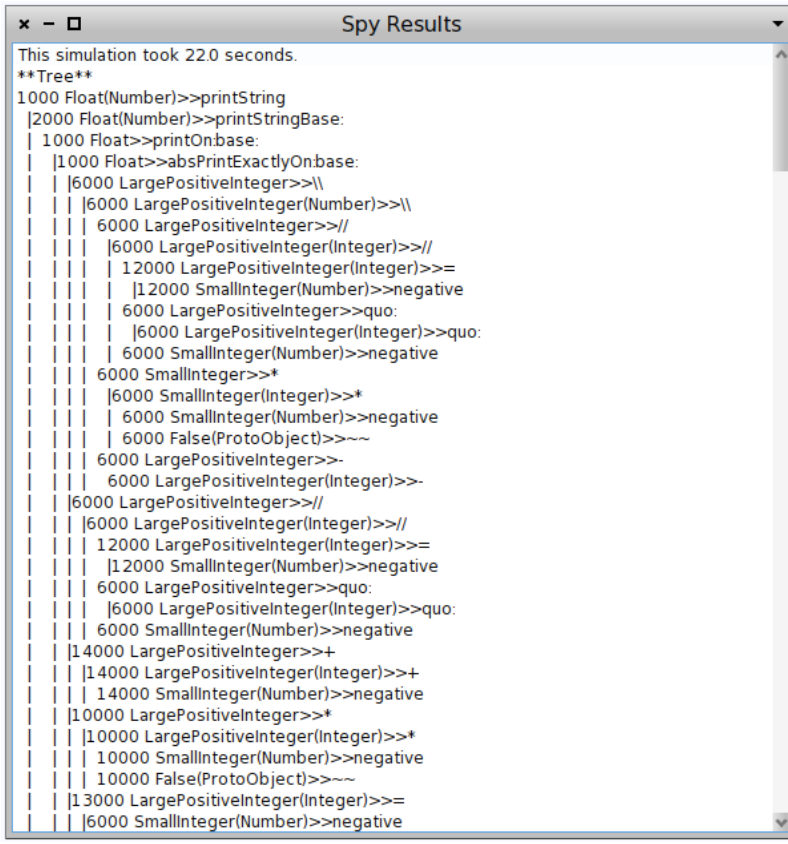

Figure 1.5: All executed messages during an execution.

```
Integer»fibSlow
   self assert: self >= 0.
   (self <= 1) ifTrue: [ ^ self].
   ^ (self − 1) fibSlow + (self − 2) fibSlow
```

The method fibSlow is relatively inefficient. Each recursion implies a duplication of the computation. The same result is computed twice, by each branch of the recursion.

A more efficient (but also slightly more complicated) version is obtained by using a cache that keeps intermediary computed values. The advantage is to not duplicate computations since each value is computed once. This classical way of optimizing program is called memoizing.

```
Integer»fib
```

```
    ^ self fibWithCache: (Array new: self)

Integer»fibLookup: cache
    | res |
    res := cache at: (self + 1).
    ^ res ifNil: [ cache at: (self + 1) put:  (self fibWithCache: cache  ) ]

Integer»fibWithCache:  cache
    (self <= 1) ifTrue: [ ^ self].
    ^ ((self − 1) fibLookup: cache) + ((self − 2) fibLookup: cache)
```

As an exercise, profile 35 fibSlow and 35 fib to be convinced of the gain of memoizing.

1.8 SpaceTally for memory consumption per Class

It is often important to know the amount of instances and the memory consumption of a given class. The class SpaceTally offers this functionality.

The expression SpaceTally new printSpaceAnalysis runs over all the classes of the system and gathers for each of them its code size, the amount of instances and the total memory space taken by the instances. The result is sorted along the total memory space taken by instances and is stored in a file named STspace.text, located next to the Pharo image.

It is not surprising to see that strings, compiled methods and bitmaps represents the largest part of the Pharo memory. The proportion of the compiled code, string and bitmap may be found in other platforms for diverse applications.

SpaceTally's output is structured as follows:

Class	code space	# instances	inst space	percent	inst average size
ByteString	2053	109613	9133154	31.20	83.32
Bitmap	3653	379	6122156	20.90	16153.45
CompiledMethod	20067	51579	3307151	11.30	64.12
Array	2535	85560	3071680	10.50	35.90
ByteSymbol	1086	35746	914367	3.10	25.58
...					

Each line represents the memory analysis of a Pharo class. Classes are ordered along the space they occupy. The class ByteString describes strings. It is frequent to have strings to consume one third of the memory. Code space gives the amount of bytes used by the class and its metaclass. It does not include the space used by class variables. The value is given by the method Behavior>>spaceUsed.

The # instances column gives the amount of instances. It is the result of

Behavior>>instanceCount. The inst space column is the amount of bytes consumed by all the instances, including the object header. It is the result of Behavior>>instancesSizeInMemory. The percentage of the memory occupation is given by the column percent and the last column gives the average size of instances.

Running SpaceTally on all classes takes a few minutes. SpaceTally may also be executed on a reduced set of classes to increase the analysis time. Consider:

```
((SpaceTally new spaceTally: (Array with: TextMorph with: Point))
   asSortedCollection: [:a :b | a spaceForInstances > b spaceForInstances])
```

The method SpaceTally>>spaceTally: analyzes the memory consumed by each classes of its argument. It returns a list of instance of SpaceTallyItem.

1.9 Few advices

We have seen a number of strategies to measure and optimize a program. The examples we have used are relatively small. Optimizing a program is not always an easy task. Identifying method candidate for inserting a cache is simple and efficient once (i) you know when to invalidate the cache and (ii) when you are aware of the impact on the overall execution when inserting the code.

In general, it is more valuable to understand the overall algorithm than trying to optimize leaf methods. The way data are structured may also provide opportunities for optimization. For example, using an ordering collection or a linked list may not be appropriated to represent acyclic graphs. Using a set may offer better performance or a dictionary in the case that hash values are reasonably well distributed.

The memory consumption may plays an important role. The overall performance may significantly decreases if the garbage collector is often solicited. Recycling objects and avoiding unnecessary object creations helps reducing the solicitation of the garbage collector.

1.10 How MessageTally is implemented?

MessageTally is a gorgeous example on how to use Pharo's reflecting capabilities. The method spyEvery: millisecs on: aBlock contains the whole profiling logic. This method is indirectly called by spyOn:. The millisecs value is the amount of milliseconds between each sample. It is set at 1 per default. The block to be profiled is aBlock.

The essence of the profiling activity is given by the following code excerpt:

```
observedProcess := Processor activeProcess.
Timer := [
   [ true ] whileTrue: [
      | startTime |
      startTime := Time millisecondClockValue.
      myDelay wait.
      self
         tally: Processor preemptedProcess suspendedContext
         in: (observedProcess == Processor preemptedProcess
               ifTrue: [ observedProcess ] ifFalse: [ nil ])
         by: (Time millisecondClockValue − startTime) // millisecs ].
   nil] newProcess.
Timer priority: Processor timingPriority−1.
```

Timer is a new process, set at a high priority, that is in charge of monitoring aBlock. The process scheduler will therefore favorably active it (timingPriority is the process priority of system processes). It creates an infinite loop that waits for the amount of necessary milliseconds (myDelay) before snapshooting the method call stack. The process to observe is observedProcess. It is the process in which the message spyEvery: millisecs on: aBlock has been sent.

The idea of profiling is to associate to each method context a counter. This association is realized with an instance of the class MessageTally (the class defines the variables class, method and process).

At a regular interval (myDelay), the counter of each stack frame is incremented with the amount of elapsed milliseconds. The stack frame is obtained by sending suspendedContext to the process that has just been preempted.

The method tally: context in: aProcess by: count increments each stack frame by the amount of milliseconds given by count.

The memory statistic are given by differentiating the amount of consumed memory, before and after the profiling. Smalltalk, an instance of the class SmalltalkImage, contains many accessing methods to query the amount of available memory.

1.11 Chapter summary

In this chapter, we see the basic of profiling in Pharo. It has presented the functionalities of MessageTally and introduced a number of principles for resorbing performance bottleneck.

- The method timeToRun and bench offer simple benchmarking and

should be sent to a block.

- MessageTally is a sampling-based code profiler.
- Evaluating `MessageTally spyOn: [ "an expression" ]` executes the provided block and display a report.
- Accuracy is gained by increasing the execution time of the profiled block.
- The Pharo programming environment gives several convenient ways to profile.
- Counting messages is slow but accurate profiling technique.
- Memoization is a common and efficient code pattern to speed up execution.
- SpaceTally reports about the memory consumption.