Seminar 5 - Optimization

PV260 Software Quality

March 24, 2015

Context

We have working application with clean code following SOLID principles - the Regex Fractals generator as we left it at the end of the refactoring seminar. It can be used to generate images as large as 1024*1024 when using the options described below. However we want it to run on arbitrarily large image size (say as least 16384x16384)
For the purposes of this exercise we limit the resources the JVM can use. When running the program we supply these flags:

- `-Xmx512m` maximum memory available to JVM, in megabytes
- `-Xss104k` thread stack size, in kilobytes
- `-Xverify:none` workaround for a profiler bug in NetBeans

In NetBeans: right click project → Properties → Run → VM Options

Instrumentation

> *Instrumentation is the addition of byte-codes to methods for the purpose of gathering data to be utilized by tools. Since the changes are purely additive, these tools do not modify application state or behavior. Examples of such benign tools include monitoring agents, profilers, coverage analyzers, and event loggers.*

Extra info for the curious:

- `http://docs.oracle.com/javase/7/docs/api/java/lang/instrument/Instrumentation.html`
- `http://www.javamex.com/tutorials/memory/instrumentation.shtml`

Classmexer agent

http://www.javamex.com/classmexer/

> *Classmexer is a simple Java instrumentation agent that provides some convenience calls for measuring the memory usage of Java objects from within an application.*

Installation:

- Download the agent from `http://www.javamex.com/classmexer/classmexer-0_03.zip`
- Add the agent jar to project classpath
- Add the JVM flag `-javaagent:path/to/classmexer.jar`

Usage:

- `MemoryUtil.memoryUsageOf(object)` size of the object itself, without taking into account sizes of its children
- `MemoryUtil.deepMemoryUsageOf(object)` size of the object and whole child hierarchy under it (note that when the object is GCd, some of its children might not be GCd if they are also referenced from elsewhere, so not all deep memory is released)

Profiler

Profiling techniques:

- `Samplig:`
  Periodic queries to the VM, provides less accurate aproximation but causes less overhead.
- `Instrumentation:`
  Inserts own code into compiled bytecode of the profiled application, which means performance can be significantly affected by the profiling itself (Observer effect). However for situations where exact counts (method calls, memory used . . .) are required, this technique is preferable.

Profiling targets:

- `CPU:`
  Measures how much time is spent in parts of the program.
- `Memory:`
  Measures how much space objects take on the heap, how many instances of a class there are etc.

Overhead of Recursion

revisions 1, 2

- When generating the signatures grid the data we care about are the Grid containing computed Signatures
- However when the computation is in progress there is the overhead of:
  - the signatures are held in the Fragments themselves before being copyed into the final Grid
  - the extra space taken by every Frame that is not the signature (position, collection of children etc.)
  - because we have the whole recursive structure from the root to the leaves, we dont have size*size Fragments but rather $\sim$ 1.33*size*size Fragments
- That means if we want to compute 2048*2048 signatures, the String signatures themselves take $\sim$ 256MB, our algorithm however takes at least 2x that, for reasons mentioned above however it would be MUCH more as the signatures in the Fragment structure would take $\sim$ 256MB*1.33 and the overhead of all the Fragments themselves is huge, about as much space as the signatures themselves.

Overhead of Recursion

revisions 1, 2

- We need algorithm that doesnt duplicate all the data during the generation and has much lower overhead
- Possible solution is to transform the recursive algorithm to iterative, so that it:
  - avoids all the object creation, only the signatures are kept
  - doesnt create the signatures in some intermediate structure from which it then copyes them into the grid (requiring at least 2x the size of the signatures in memory), instead it writes directly into the grid
- Such implementation is the IterativePermutationsFractalGrid
- Notice how it doesnt follow the way we would solve the task intuitively as well as the recursive algorithm does

Intermezzo - Sizes of Objects

byte, boolean	byte
short, char	2 bytes
int, float	4 bytes
long, double	8 bytes

reference	4 bytes
plaint Object	16 bytes
Byte, Integer ...	16 bytes
Long, Double	24 bytes
empty String	40 bytes

- Every Object has $\sim$ 12 bytes of metadata (implementation specific)
- All sizes are aligned to next multiple of 8 (so that if you use 17 bytes, your object takes 24 bytes of heap space)

Extra info for the curious:

- `http://www.javamex.com/tutorials/memory/string_memory_usage.shtml`
- `http://java-performance.info/overview-of-memory-saving-techniques-java/`

Minimizing Space Used by Signatures

revision 3

Lets consider the case of $2048 * 2048$ image.

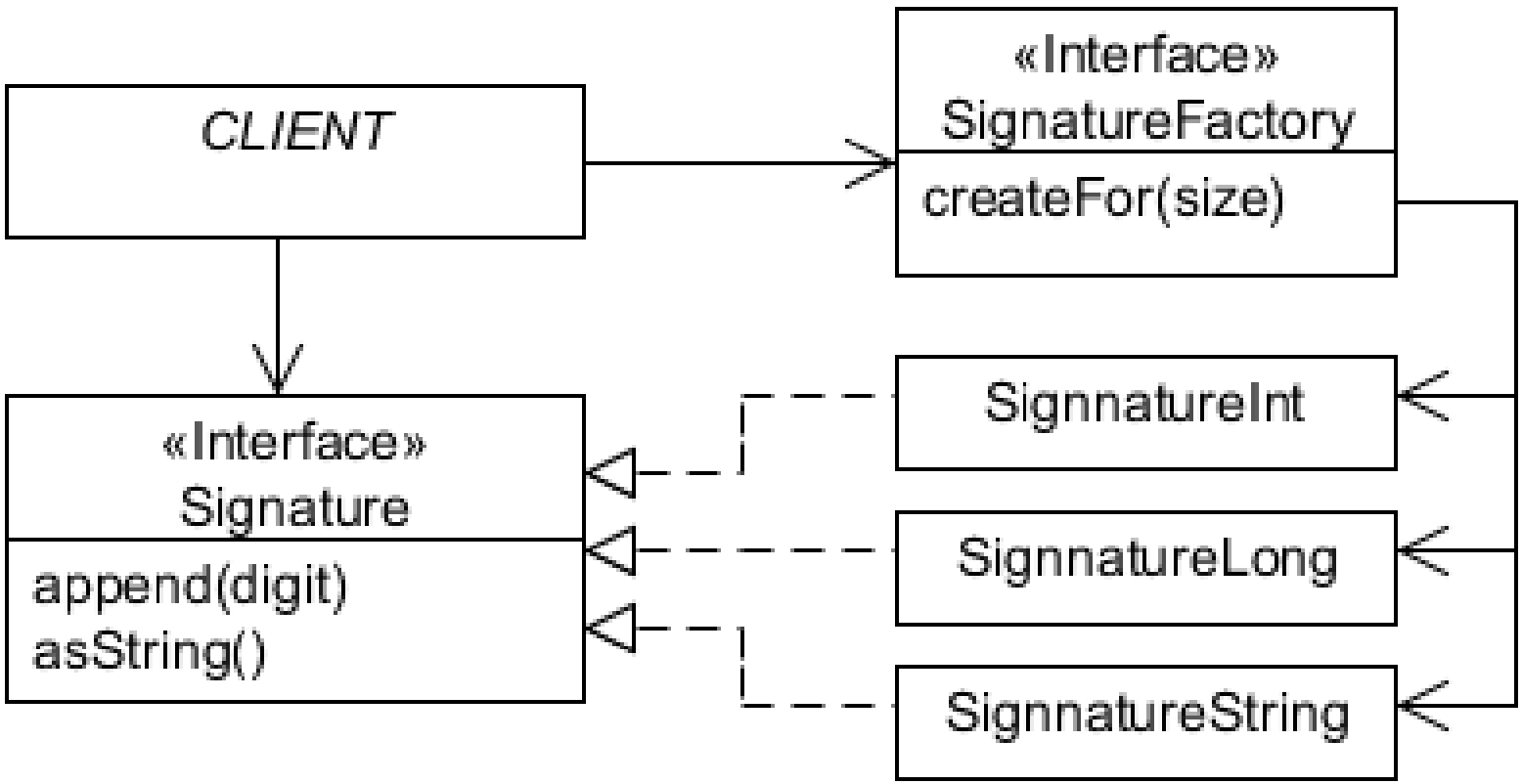
- If we use String:
  2048 is 2^{11} so we need signatures with length 11.
  String with length 11 takes 64 bytes of space:
  12b header, 4B int cached hash, 4B int cached hash32, 4B reference to char[] value, which itself has 12B header, 11*2 chars, aligned to nearest multiple of 8 we get 40 for char[] + 24 String = 64B
  $2048 * 2048 * 64B \sim 260MB$
- If we use Long:
  We know the signature will always be only numbers.
  With this extra information we can use more efficient way of storing the signature value.
  Max possible value of Long is 9223372036854775807, so we can save signature of size up to 19 in it.
  $2048 * 2048 * 24B \sim 96MB$

Minimizing Space Used by Signatures

revision 3

- For signatures of size at most 9 we want to use Integer (16B)
- For signatures of size at most 19 we want to use Long (24B)
- For even larger signatures we can use the String (>72B)

To shield client code from this optimization we use the following:



Minimizing Space Used by Signatures

revision 3

Now the code is still reasonably maintainable while the performance has improved dramatically.

- We deal with abstraction of Signature, new implementations can easily be added. As an example see SignatureBitwiseInt which uses bits inside an int to store signatures up to the size of 14 while taking only 16B of space (2 /3 of how much the Long implementation takes). The bitwise implementation is however slowest of them all (We trade low storage cost for higher computational difficulty of extracting the data we store)
- We could use array of long directly inside the FractalGrid further reducing the space required, HOWEVER:
  - we would impose hardset limit of max possible size of images the program can deal with (max storable signature has length 19)
  - even if we managed to store signatures in as little as 1 byte per signature, with larger images ($32768 * 32768 * 1 \sim 1GB > 512MB$ we have) the problem would persist.

Minimizing Space Used by Signatures

revision 3

If for whatever reason we needed to do this optimization and use raw primitives, it would be preferable to use some collection which encapsulates this behavior rather than dealing with the arrays directly for reasons described in previous seminars. One possibility would be to use the Trove library which provides primitive backed Collections using the native Java Collections interface.

`http://trove.starlight-systems.com/`

Intermezzo - Compile time Optimizations

Which of the following would you use?

```
public String A(String a, String b) {
    return "a=" + a
           + ",b=" + b
           + ", something";
}

public String B(String a, String b) {
    return new StringBuilder()
        .append("a=").append(a)
        .append(",b=").append(b)
        .append(", something").toString();
}
```

Intermezzo - Compile time Optimizations

- They are compiled to identical bytecode, so A is much better as it is more readable and the performance is the same.
- To decompile a `.class` file use the `javap` command with a flag `-c`. So the whole command in cmd would be `javap -c SomeClass.class`.
- StringBuilder is to be used when the concatenation doesnt happen in a single pass, so for example in a loop.
- Loop using '+' creates a new StringBuilder on every pass, appends the result so far and the new part. This can indeed be a performance killer on long concatenations.

Towards Lazy Computations

revision 4

- Our problem is that we store the whole grid in memory.
- This solution is not scalable, only way to keep up with increasing grid sizes is adding more physical memory.
- The requirements grow exponentially, so if we start with X memory required to produce image size 2048, we will need 4X to produce 4096, but at least 16X to produce 8192
- Now imagine that apart from big images we also want to run 10 generations at the same time in parallel. Or 100, 1000...

Towards Lazy Computations

revision 4

- Our program doesnt need to have the whole grid precomputed, we merely need to know what is the signature of [x,y] when asked.
- Solution is to compute this signature based on the [x,y] when asked.
- That is compute it at the latest time possible - lazily
- Using this technique we only ever need to have enough memory for computation of a single Signature.

Towards Lazy Computations

revision 4

See OnDemandFractalGrid for such implementation.
We already know the positives of this implementation, some negatives are:

- The code is far less readable than the original recursive version and it doesn't reflect the domain concept as well.
- With the original solution any subsequent call to signatureOf(x,y) after creation would be O(1) as all the signatures are precomputed, with the OnDemand solution the signature has to be recomputed every time.

Adapting Instead of Copying

revisions 5,6

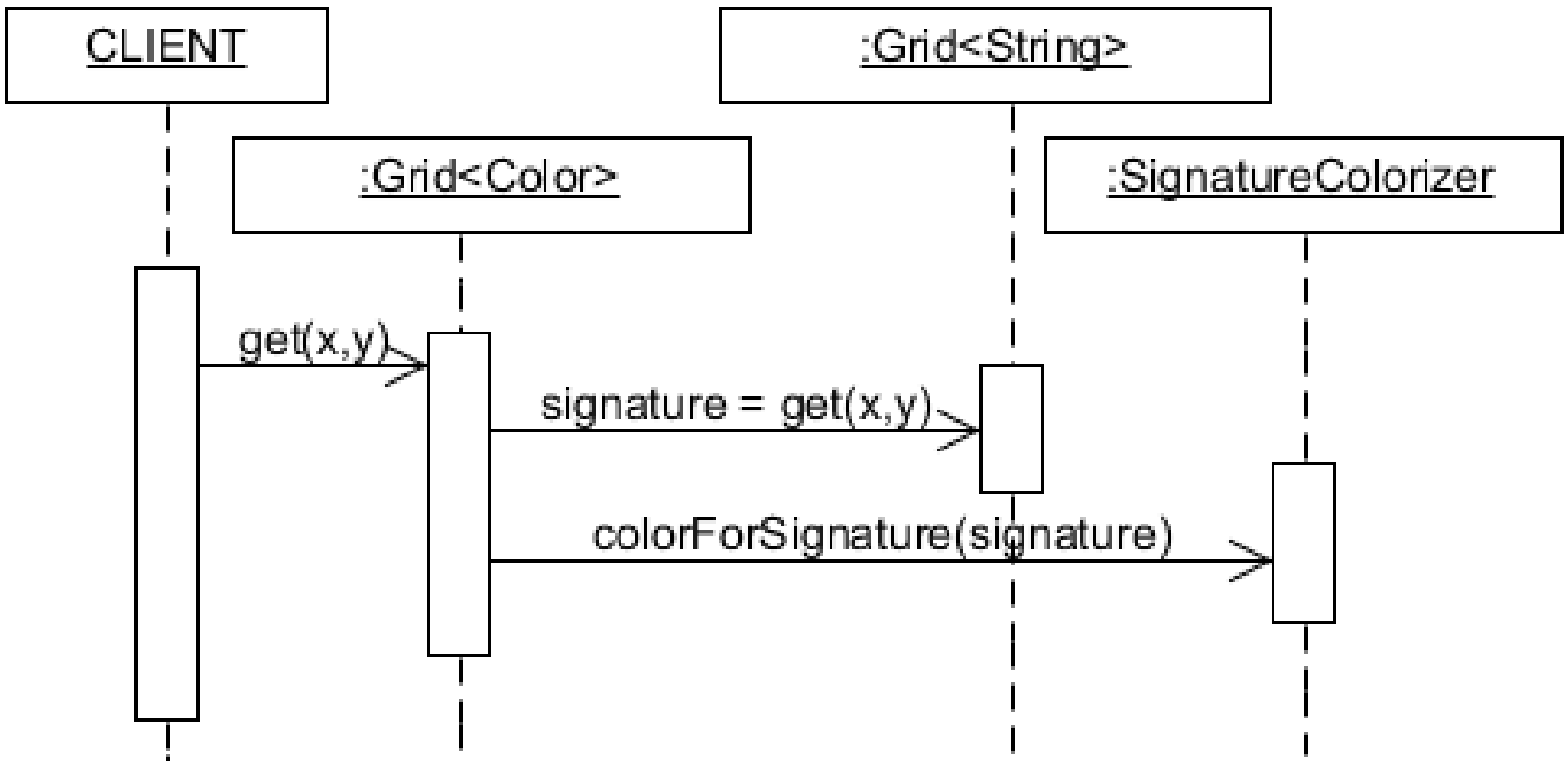
Now that we have the signatures calculated, or rather a way to calculate them when needed, we need to represent them as a Grid<String>as that is what the rest of the program works with.

- One way would be to simply copy the signatures into a new Grid, but that would bring us back to square one as we would need to hold all the signatures in memory
- Working solution is to create an adapter which implements a Grid, has a backing FractalGrid and delegates all get() calls to the FractalGrid's signatureOf()
- This solution breaks the LSP, however at the moment it seems to be the only way to meet the requirements so we have to go with it

Adapting Instead of Copying

revisions 5,6

- The same approach can be used to make the GridColorizer return Grid adapter which behaves as follows:



Optimization at the Cost of Violating SOLID

revision 7

The last bottleneck in outputing ASCII image is again the fact we need to have the whole image stored in memory to use the current classes and methods.

- We need to write directly into a non-memory-based stream, file in our case
- This can't be achieved with the current ImageConverter directly, so we have to break the LSP again to reach our goal
- By implementing ImageConverter<Void>we at least signalize that our converter will always return null
- With the current design it is not possible to both meet our requirements and have perfectly clean code
- Now that we have fulfilled the requirements in the simplest possible way we can refactor the project again to make it clean again

Helicopter View of the Program Flow Change