# Experience Talk
*“QA Infrastructure – meeting commercial robustness criteria”*

Marcus Lagergren **(marcus@bea.com)**
BEA Systems

## QA infrastructure

- QA infrastructure is harder and probably even more important than development infrastructure.
- The most valuable lesson we have learned is that it must be developed parallel to the VM and significant effort must be spent on it.
- It is at least as important as the VM itself.

## Build System

- Build system, test system and source control are parts of the same distributed system.
- Mobility - Build anything anywhere, locally or globally (distributed). ”A global cross compiler”
- Build system should be *self contained* & *part of source control.*
  - Do a sync from source control, have all the details.
  - We chose to put buildtools there as well to produce deterministic bits and provide self sufficience

bea Think liquid.™

## Test System

- Local and remote test runs possible.
  - Submit jobs ”crunch through these tests”
  - ”Submit if passes tests”.
- Test machines in the distributed system
  - Performance test machines (dedicated)
  - Functionality test machines (not necessarily dedicated)
  - Any machine can volounteer CPU cycles for functional testing.
  - Easy to add and remove machines.

## Continuous Automatic Testing

- Need continuous automatic testing.
  - Bit rot sets in immediately when code is removed from automated testing.
- Release version may break debug version and vice versa.
- Linux version may break Windows version and vice versa.
- Use fascist compiler flags.

## Building Blocks – Tests

- Many tests, especially regression tests, for a JVM needn't be more than a main class with a return value.
- *Claim:* if it's simple enough to write and submit a test, > 50% of the bugs can get regression tests submitted as part of the original bugfix.
  - I will address the other 50% later.
- Easy-to-write tests make it possible for the test suite to grow naturally.
  - If 10 minutes of spare time can lead to a new test being written, checked in and enabled as part of the global test suite, we have succeeded.

Think liquid.

## Testing Functionality & Performance

- Functionality
  - Simple tests, ”yes/no”
  - ”Terror harnesses” that attack the cross sections between modules. (AllocAndRun <program>, RedefineClasses <program> ExceptionsInClinits <program>, ...)
  - Complex tests/Large apps that run for a long time
- Performance
  - Anything and EVERYTHING affects performance.
  - Automatic regression tests with warnings, database of deviations, baselines and invariants
  - It should be easy to add more benchmarks

## Building Blocks – Result Database

- Result database
  - Sensible layout.
  - Easy to maintain and backup.
- Should be easy to query from local machines about historical test results.
  - ”When exactly did this performance regression appear?”
  - ”List all benchmark scores on this machine for this benchmark since January 1”
  - ”Has this functional test failed before? What were the bugfixes?”

## Testing – The Hard Part

- Simple Java reproducers aren't enough for all kinds of bugs.
- How do we test for a specific optimization bug in the code generator?
- How do we test for a strange boundary case that crashes the GC, that happens after two weeks in production?
- Key observation: We need to be able to export and import a *state*.

Think liquid.

## Testing – The Hard Part

- Examples:
  - *Create a very special heap with a few objects in nasty places. Load it and trigger a garbage collection. Save it and compare to reference.*
  - *Serialize an IR from just before an offending optimization. Load it and trigger the optimization. Save the resulting IR and compare it to reference.*
  - *Compare would be more of an ”equals” than a ”memcmp”*
- We need a level of modularization that’s good enough for this.
- The collection of tests should grow naturally, but the VM design should allow the ways of testing the VM to grow naturally as well.

# Testing – The Hard Part

- But of course it's not as simple as that.
- What about multi threaded apps? Race conditions?
  - Plenty of threads operate on the same memory – e.g. Multi threaded GC. How can we make test cases?
  - Synchronization points.
  - Randomized input, randomized sleeps. Try to cover the malicious side effects of parallelism.

bea Think liquid.

## Testing – The Hard Part

- Sometimes we just need to crunch a lot of code for a long, long time.
- Nothing else suffices to reproduce a problem or the framework that would make it possible doesn't exist.
- At least make the dumps comprehensible.
- ”Phonehome”
  - Suprisingly effective if you have enough beta testers.

## Testing – How can we get a framework going?

- Learn from history
- For example, go over 500 bug parade entries for HotSpot or 500 JRockit CR:s.
  - How many can be tested by small deterministic reproducers?
  - What about the rest - brainstorm what functionality the VM would need  if we had to write a simple reproducer for each problem.

## Development – The platform matrix

- Try to keep the amount of platform independent VM code as large as possible.
- It is always a choice between platform specific features and test matrix growth.
- Initially, our performance critical code was native. As our JIT got better, we would write more and more in Java. Native overhead today is much worse
- Augmented Java – intrinsics, *”pd_addr”,* preprocessed Java files, annotations

bea Think liquid.

## Development – The platform matrix

- Other seemlingly platform dependent things can be made platform independent.
  - *Example: Native stubs. The bulk of the work is parameter marshalling, the register allocator can do that already.*
- Implementation language: Debugging is an issue
  - Powerful C/C++ debuggers exist. Meta-debugging is usually harder.

## Thank you

Questions