TE WHARE WĀNANGA O TE ŪPOKO O TE IKA A MĀUI
VICTORIA
UNIVERSITY OF WELLINGTON

SWEN 301 : Scalable Software Development

Automated Program Analysis

Jens Dietrich **(jens.dietrich@vuw.ac.nz)**

Overview

- software quality assurance
- models
- how to build models: static vs dynamic
- static analysis
  - pattern-based analysis
  - control flow analysis
  - data flow analysis
- dynamic analysis
  - testing
  - test generation -- quickcheck and fuzzing
  - instrumentation
  - monitoring: logging, JMX, heap and stack snapshots
  - profiling

Automated Program Analysis in Year 4

- dedicated course in year 4
- offered as special topic SWEN438 in 2024
- focus on static techniques, includes elements of compiler construction (parsing, source code and byte code analysis – from discontinued SWEN430)

The Objective of Program Analysis

- build a model from a program (UML, graph-based, Relational DBs, metrics)
- then reason about / query this model
- find issues: bug, design flaws, vulnerabilities, style guide violations, poor performance
- if possible, define regressions: how does this compare to model from previous version ?
- automate all of this ! integrate into builds / CI

Desirable Properties

- **satisfactory precision** (low false positive rate): find (most) real issues
- **satisfactory recall** (low false negative rate): find all issues
- **sound** = full recall (100% found, but possibly some false positives)
- **integration** - analysis should be easy to integrate into builds
- **scalability** - analysis should be able to be performed on large programs with resources (hardware, time) available
- See: https://cacm.acm.org/magazines/2018/4/226371-lessons-from-building-static-analysis-tools-at-google/fulltext

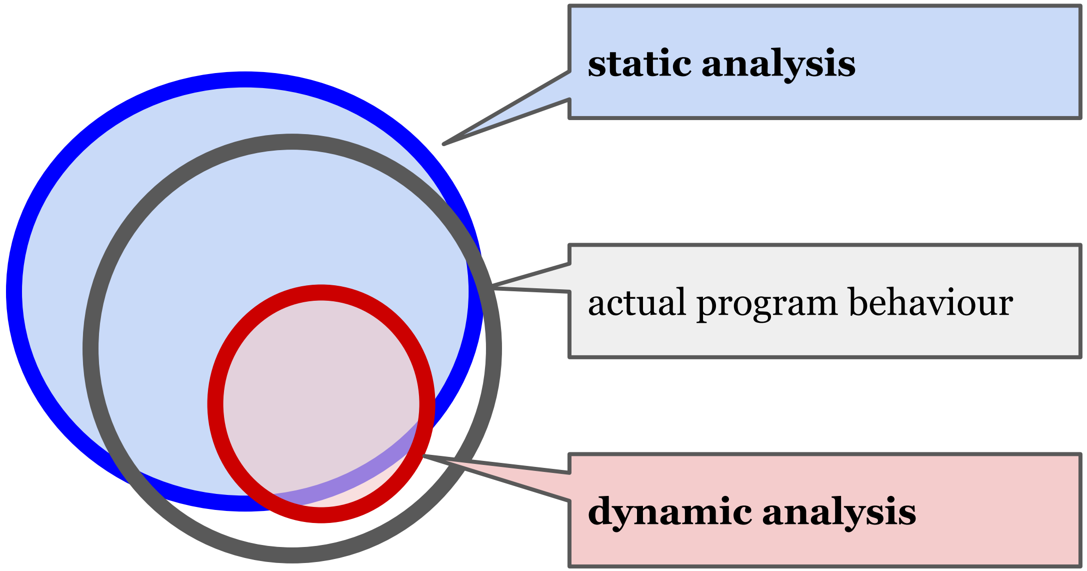
Static vs Dynamic Analysis

- static: take the program, analyse it (source or byte code) and build a model to reason about
- dynamic: take a program, execute it, observe its behaviour and reason about these observations

Static vs Dynamic Analysis (ctd)

- static analysis builds a model from the **entire program** to reason about, it does not miss things, i.e. has no false negatives -- it is (in principle) **sound**
- in practice, it is often **not sound either** as parts of the program (e.g. reflection) are difficult to analyse
- as a model is abstract, it may return false positives -- it is **not precise**
- dynamic analysis only reports actual program behaviour, so it is precise
- but it misses possible program behaviour not exposed by the particular execution observed - so it is **unsound**

Static vs Dynamic Analysis (ctd)



Limits

- Rice's theorem: all non-trivial questions about the behavior of programs are undecidable.
- Approximations (compromising on precision and/or recall) are necessary !

Rice, H. G. (1953), "Classes of recursively enumerable sets and their decision problems", Transactions of the American Mathematical Society, 74 (2): 358–366, doi:10.1090/s0002-9947-1953-0053041-6, JSTOR 1990888

Static Analysis Examples

- the Java compiler
- JVM verification
- dependency analysis
- pattern-/ rule- based code analysis
- call graph analysis

Static Analysis Models

- graph based - call graph, points-to, dependencies
- tree-based - source code (AST)
- metric based -- associate artifacts with numerical values
- other (UML, ..)

Java Compiler and Verifier Analysis

- the java compiler checks source code for correct structure
- enforced semantics rules (examples):
    - type safety
    - correct use of extends / implements, including loop-free checks
    - encapsulation (method, field and class visibility)
- when a program is executed and classes are loaded, the verifier performs similar checks to make sure that binaries cannot circumvent the rules

Analysing Package Dependencies

- analyse source code (look for used classes)
  - tooling: best done by analysis parse tree (AST): https://javaparser.org/
  - challenge: must correctly resolve imports (what is the package of a class?)
- analyse byte code
  - class names are resolved
  - also works for non Java code compiled into Java bytecode: scala, kotlin, ..
  - good tool support available: https://asm.ow2.io/ , bcel, jassist, bytebuddy

Building the Model

- look for use of (any class) from package B in (any class) of package A
- good approximation: removing package B makes it impossible to **compile** A
- represent model as directed graph (digraph)
  - vertices represent packages
  - edges represent dependencies

Reasoning about the Model

- look for loops
- loops are considered antipatterns [Parnas 79]
- scales: can detect loops with O(n) algorithm (Tarjan’s algorithm to compute strongly connected components)
- compute metrics (distance to main sequence) to model abstractness, independence and responsibility -- how change is propagated, relate this to maintainability
- values over threshold represent violations of design principles
- can be automated and tested

Parnas, David Lorge. "Designing software for ease of extension and contraction." *IEEE Transactions on Software Engineering* 2 (1979): 128-138.

Recall and Precision

- false positives (FP): loops of packages with their “subpackages” may actually be acceptable
- false negatives (FN): static analysis may miss reflective reference
- widely used in frameworks: spring, jdbc, jee, jackson, ..
- similar features in all major PLs (example: JS **`eval()`**)

```
package a;
class Foo {
     ..
     String name = // read “b.Bar” from config
     I object = (I)Class.forName(name);
     ..
}
```

```
package b
class Bar implements I {
     ..
}
```

L Sui, J Dietrich, A Tahir, G Fourtounis: On the Recall of Static Call Graph Construction in Practice. ICSE'20

Recall and Precision (ctd)

tp - true positives (actual issues found)
fp - false positives (false alerts)
tn - true negatives (non-issues, correctly ignored)
fn - false negatives (actual issues missed)

$$\text{Recall} = \frac{tp}{tp + fn}$$

how many actual issues are missed, low if only a small number of issues are actually being detected.

$$\text{Precision} = \frac{tp}{tp + fp}$$

how many false positives are detected, low if a lot of false alerts are created

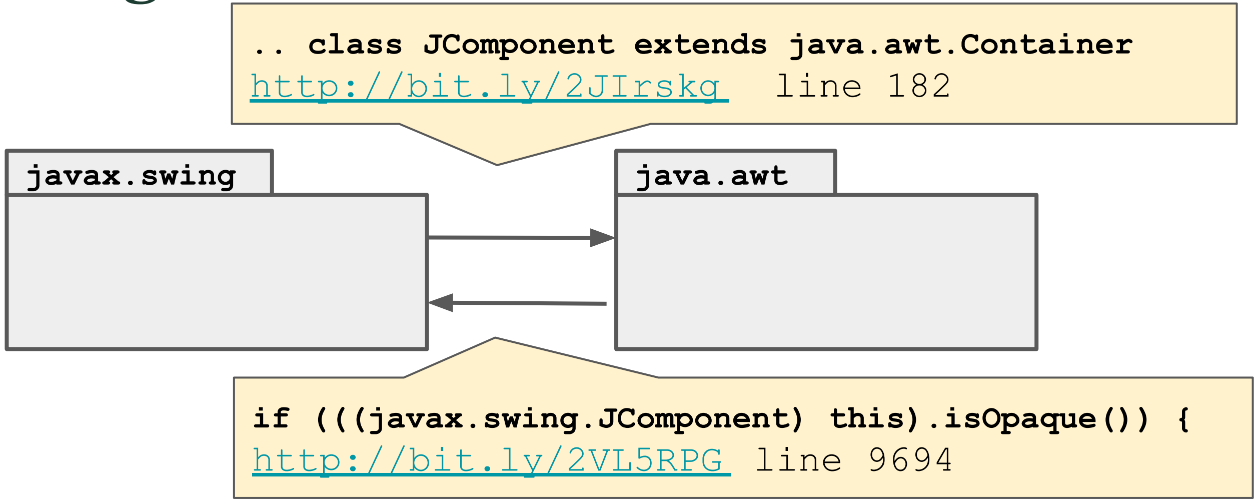
$$\text{Accuracy} = \frac{tp + tn}{tp + tn + fp + fn}$$

combination of precision and recall.

Tooling and Integration

- JDepend
- integration in build tools (Maven, ANT, ..) and IDEs

Findings



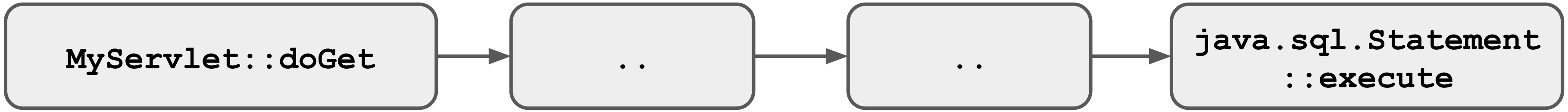
```
.. class JComponent extends java.awt.Container
```
http://bit.ly/2JIrskg line 182

```
if (((javax.swing.JComponent) this).isOpaque()) {
```
http://bit.ly/2VL5RPG line 9694

- unnecessary framework dependency
- unnecessary: Apache Harmony JDK implementation did not have this dependency
- monolithic JDK: large footprint (download, startup, memory)
- issue since 97, only started to address this in Java 9 (jigsaw)

Callgraph Analysis

- model method invocations
- use case: find out whether servlet application entry points can reach method calls that could be used for SQL injection attacks
- model is again a digraph
- once model has been computed, it can be queried for ***reachability***
- can also try to track data through method calls: can we pass data controlled by client (example: header, request parameter) to Statement::execute ? This is called ***taint analysis*** .



Challenges

- **`java.sql.Statement`** is an interface and **`execute`** is abstract, so we are looking actually not for **`Statement.execute`**, but for concrete implementations of it
- could looks for all possible methods implementing this, but many might not be reachable, will lead to many false positives (false alerts)
- developers will not accept tool that has high false positive rate
- can remove many false positives by tracking objects (points-to analysis), but analysis cost goes up ($O(n^3)$ -- aka “the cubic bottleneck”)
- active area of research, details beyond scope of this course
- tools: doop, soot, wala, several commercial tools (codeql)

Rule-Based Code Analysis

- looks for suspicious pattern that can be described by **rules** within code
- if source code is analysed, this is often called linting
- issues found:
  - violations of conventions (e.g. getter / setter conventions, coding style guides
  - classes / methods exceeding a certain size
  - potential issues with casts and null pointers
  - potential violations of equals/hashcode contract, use of == instead of equals
  - unused variables and imports, “dead code”
  - ..
- often rule-based, can be extended

Objectives

- by enforcing style / conventions / size constraints -- programs become more comprehensible and therefore more maintainable
- this may also impact functionality (e.g., convention-based tools like automated data binding for XML and JSON)
- many bugs found are potential runtime problems that can now be detected at build time instead of runtime, fixing is cheaper, and they are not detected by clients

Rule-Based Code Analysis Tools

- checkstyle - focus on syntax
- pmd - former DARPA project, patterns can be written as queries on source code model (abstract-syntax tree AST), includes clone detection, support for multiple languages and formats (including Maven poms !)
- findbugs (now spotbugs) - deeper analysis (null, casts, equals/hashcode) .
- errorprone -- similar to findbugs, developed and maintained by google
- all offer Maven (and other build tools and IDE) integrations

Examples: Bugs found by SpotBugs

```
public class Person {
    private String name = "";
    private String firstName = "";
    private int age = 0;
    // setters and getters generated by IDE (IntelliJ)
    public boolean equals(Object o) {
        // code generated by generated by IDE (IntelliJ)
    }
}
```

https://github.com/jensdietrich/se-teaching/tree/main/spotbugs

Examples: Bugs found by SpotBugs

- **`hashCode`** is not implemented, but **`equals`** is
- can lead to situations where two instances are equal, but have different **`hashCode`**
- broken contract, leads to unexpected behaviour
- example: **`HashMap`** and **`HashSet`** lookups (**`contains()`**, **`get()`**) incorrectly return **`false/null`**
- limitations: incorrect implementations are not detected (by the version used), see **`nz.ac.vuw.jenz.spotbugs.Student`**

Examples: Bugs found by SpotBugs

```
public class TurtlesAllTheWayDown {
    public String someField = null;
    public static void main(String[] args) {
        doSomethingLoopy();
        System.out.println("This is fun, lets try this again");
        main(args);
    }
    public static void doSomethingLoopy() {
        int i = 0;
        while (i<42) {
            System.out.println("I feel loopy ");
        }
    }
}
```

unused field reported

infinite recursive loop reported

infinite loop reported as i is never changed inside loop

https://github.com/jensdietrich/se-teaching/tree/main/spotbugs

Examples: Style/Convention Checks with SpotBugs

```
public class whatAnOddClass {
    private void Main() {}
}
```

class names should start with uppercase characters

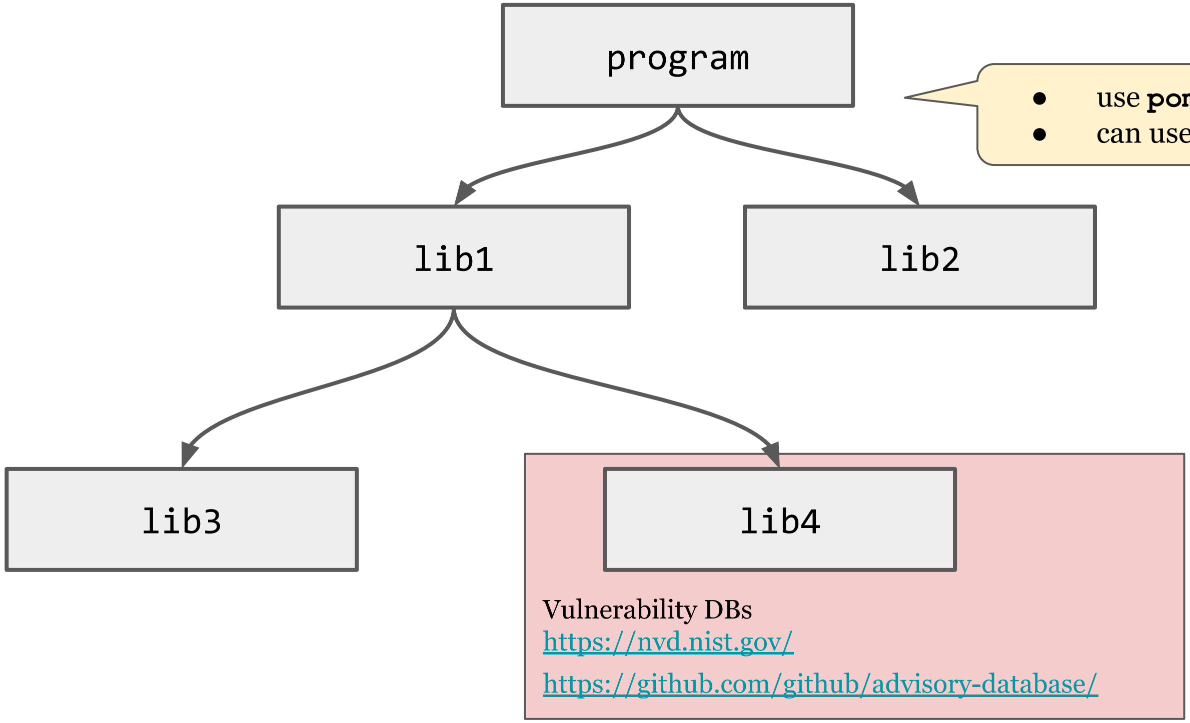
method names should start with lowercase characters

https://github.com/jensdietrich/se-teaching/tree/main/spotbugs

Repository-Based QA

- emerging class of tools that analyse software ecosystems
- example: dependabot, snyk -- integrated in GitHub, docker etc
- does analyse dependencies declared in build files, not in code

Software Composition Analysis (SCA)
dependabot, snyk, owasp dependency check, npm audit, ..



- use **`pom.xml`** to extract dependencies
- can use **`mvn dependency:tree`**

Vulnerability DBs
https://nvd.nist.gov/
https://github.com/github/advisory-database/

Vulnerability Databases - NVD

- NVD - National Vulnerability Database
- U.S. government repository
- CVEs act as identifiers – assigned by CVEs are assigned by a CVE Numbering Authorities (CNAs)
- CVAs include Mitre Corporation (non-profit) and Vendors
- maps CVEs to code (Official Common Platform Enumeration (CPEs))
- assigns metadata (kind of vulnerability, severity) (Common Weakness Enumeration CWE)
- example: https://nvd.nist.gov/vuln/detail/CVE-2022-25845

Vulnerability Databases - GHSA

- part of github
- users can make changes via pull requests
    - example: https://github.com/github/advisory-database/pull/2444
- often has more precise mapping to code (e.g. maven artifact coordinates) than NVD
- example: https://github.com/advisories/GHSA-pv7h-hx5h-mgfj

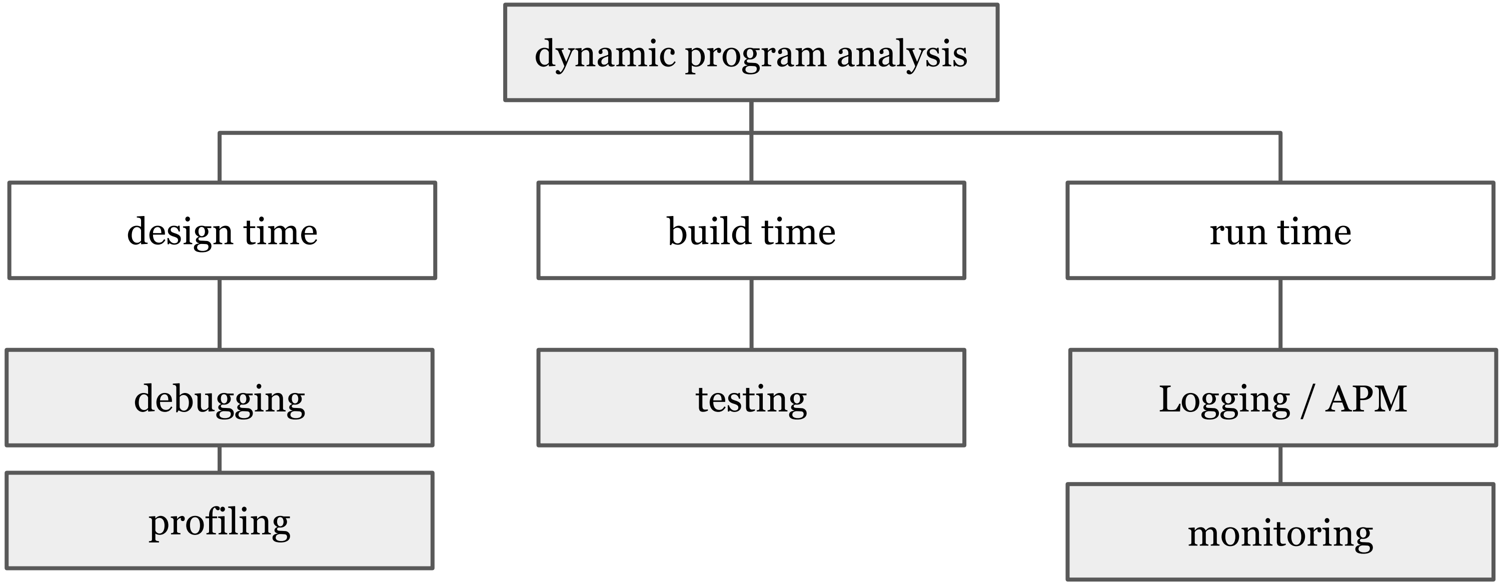
Other Databases

- vendors maintain additional databases
- sometimes have additional info, such as patch commits, and proof-of-concept (exploit) code
- databases are usually synced after some time
- example: https://security.snyk.io/vuln/SNYK-JAVA-COMALIBABA-2859222

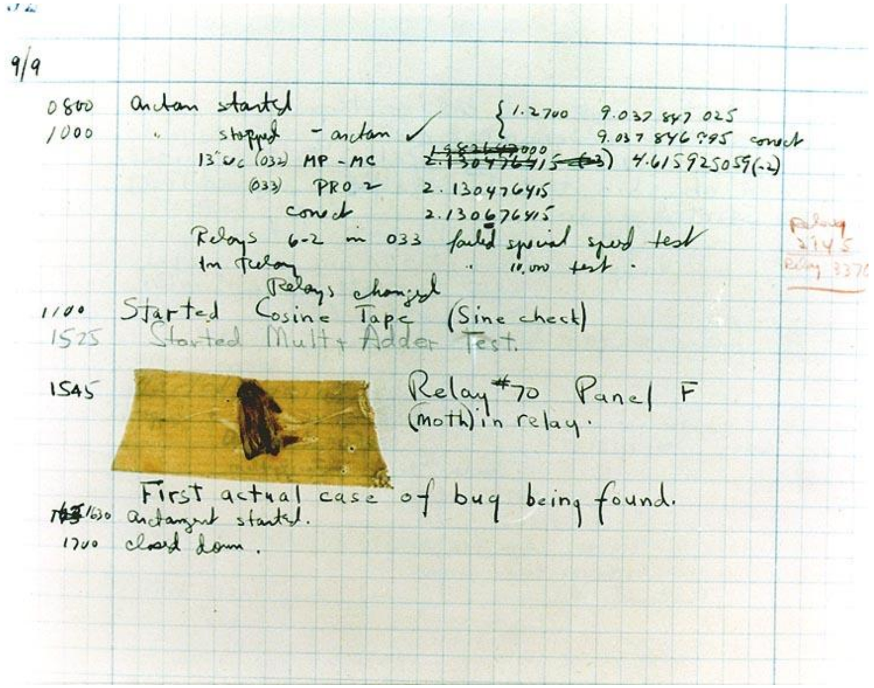
Dynamic Program Analysis

- actual program behaviour is observed and often recorded
- behaviour is triggered by “driver” - way to run the program (like main)
- checks are made against the recording

Dynamic Program Analysis



Debugging



9/9

0800 Arctan started

1000 " stopped - arctan ✓ {1.2700 9.037 847 025

9.037 846 995 conect

13" uc (032) MP - MC ~~1.982147000~~ ~~2.130476415~~ (~~-3~~) 4.615925059(-2)

(033) PRO 2 2.130476415

conect 2.130676415

Relays 6-2 in 033 failed special speed test

in relay " 11.000 test .

Relays changed

1100 Started Cosine Tape (Sine check)

1525 Started Mult + Adder Test.

1545 Relay #70 Panel F

(moth) in relay.

First actual case of bug being found.

~~1630~~ 1630 Arctangent started.

1700 closed down.

Relay 2145

Relay 3370

the first bug ever found was .. a bug !

http://en.wikipedia.org/wiki/Grace_Hopper#Anecdotes

The Psychology of Debugging

- debugging is problem solving
- focus on finding the bug, not blaming people
- "that's impossible" is not possible
- use a debugger (not **`System.out ..`**)
- read error messages
- switch into debug mode: set log level to DEBUG, enable assertions, enable compiler warnings
- try to reproduce the bug in a test case

Debuggability as Design Goal

- it is important to keep **debuggability** in mind when developing software
- using **logging** supports debugging
- since logging is expensive, log levels must be used
- exceptions should be **chained** using causes
- assertions , precondition checks (runtime exceptions) and test cases can be used to localise faults
- in general, modular designs are easier to debug

Reading Exceptions

- many bugs are manifested as exceptions or errors
- the general term covering both exceptions and errors is **throwables**
- i.e., in Java, **`Throwable`** is the superclass of both **`Exception`** and **`Error`**
- throwables have a message and a **stack trace**
- the stack trace shows the method invocations leading to the throwable
- these invocations have line numbers
- IDEs often turn them into hyperlinks to source code

Printing Stack Traces

- console:
  `exception.printStackTrace()`
  (default out is **`System.err`**)
  `exception.printStackTrace(PrintStream out)`
- logging (log4j):
  `logger.error("a message", exception)`
  `logger.warn("a message", exception)`
  .. (equivalent methods for other log levels)

Exception Chaining

- often, an exception is caused by another exception
- with exception chaining, a reference to the **causing exception** can be retained
- when printing the stack trace, the stack trace of the parent exception should be printed as well
- this supports better diagnosis
- the “try with resource” syntax provides a more concise syntax for some nested exception patterns

Debuggers

- debuggers are tools to debug code
- they support the step-by-step execution of programs, and the suspension of (partial, i.e. thread only) program execution at breakpoints
- debuggers support the **inspection of program state** (objects within the scope of the execution)
- **expressions can be evaluated** (including complex stream/lambda expressions to find elements in large collections !)
- **watches** are saved expression that are being evaluated
- some debuggers also support the direct manipulation of program state

Delta Debugging

- technique to deal with failures in large and complex programs
- try to reduce dataset to isolate failure causes, obtain minimal set
- basic algorithm: as long as the failure can be reproduced, randomly reduce data set
- can be completely automated, highly effective

Breakpoints

- breakpoints are added to lines of source code
- the execution of a program will be interrupted at a breakpoint
- from breakpoint, developer can step-by-step “walk” through execution
- often, the program execution should only stop if additional conditions are satisfied

Conditional Breakpoints

- **boolean conditions** - the program only stops if the condition evaluates to true
- **hit count** - if set to N, the program will only stop when the breakpoint is encountered for the *N*th time
- **condition change** - the program will stop if the value of an expression changes

Debugging Scope

- debugging requires source code
- this means that it is not possible to debug into code from external libraries
- however, often this is required to establish where a problem originate from
- possible solutions:
  - associate libraries with sources
  - decompile libraries on-the-fly (IntelliJ)
  - Maven facilitates this: sources must be released along binaries, so IDEs can locate them if libraries are specified as Maven dependencies

Interference

- the debugger may itself interact with the code -- in the debugger, objects are displayed as strings generated using **`toString()`**
- **`toString()`** may have side effects !
- **Heisenbugs** - the observation in the debugger is state changing
- named after Werner Heisenberg's observer effect in quantum mechanics
- https://github.com/jensdietrich/se-teaching/tree/main/heisenbug

Profiling

- another dynamic program analysis: run and observe
- measure use of resources, such as
  - memory (utilisation, space used by objects of a certain type)
  - cpu (utilisation, time consumed by function, GC activity)
  - threads (utilisation, state)
  - custom resource (via JMX)
- generate views and reports to help the software engineer to understand and optimise program performance
- related: memory dumps for post mortem static analysis

Java Profilers (selection)

- VisualVM is part of the JDK (until version 8) and includes a profiler, see https://visualvm.github.io/
- JConsole is a simar older tool
- some IDE includes a profiler
- JProfiler is an excellent commercial profiler
- the Eclipse memory analyser (MAT) is a Eclipse plugin to visualise heap dumps
- IntelliJ now has a profile on-board

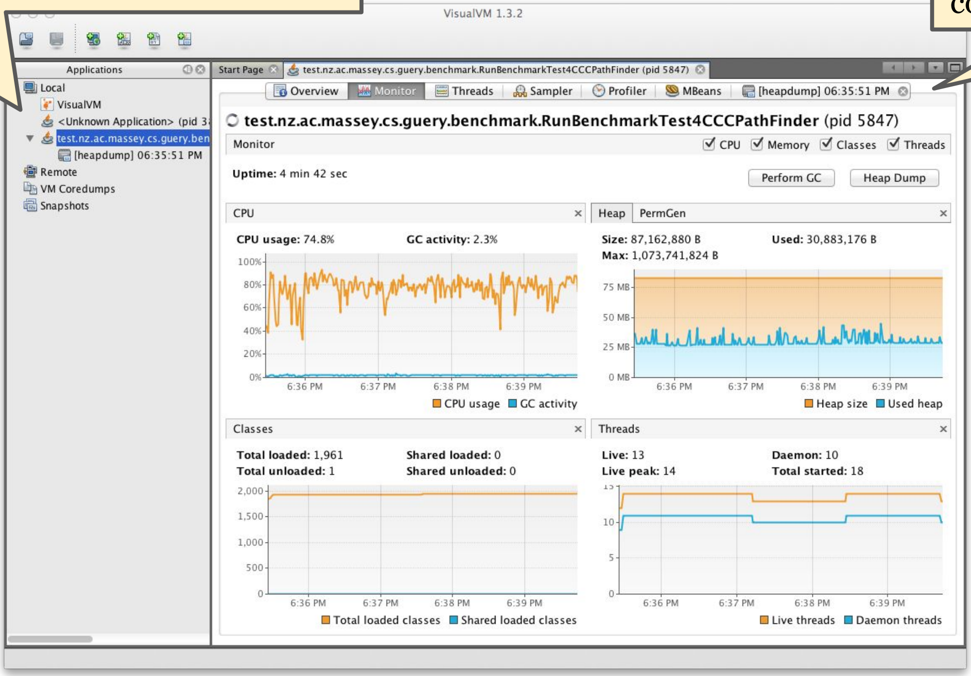
VisualVM

- is extensible through plugins (e.g. MBeans plugin)
- can interact with JMX objects
- note that VisualVM only connects to running applications - it does not start applications
- easy solution: add a "warmup loop" to application to delay startup (e.g., **`Thread.sleep(10_000)`**)
- other profilers (JProfiler) can directly start applications

VisualVM Monitor

running local and remote JVMs are listed here, to connect VisualVM simply select and click one

overview: basic information about CPU, memory usage, loaded classes and threads, and actions to perform garbage collection and to take a heap dump



test.nz.ac.massey.cs.guery.benchmark.RunBenchmarkTest4CCCPathFinder (pid 5847)

Uptime: 4 min 42 sec

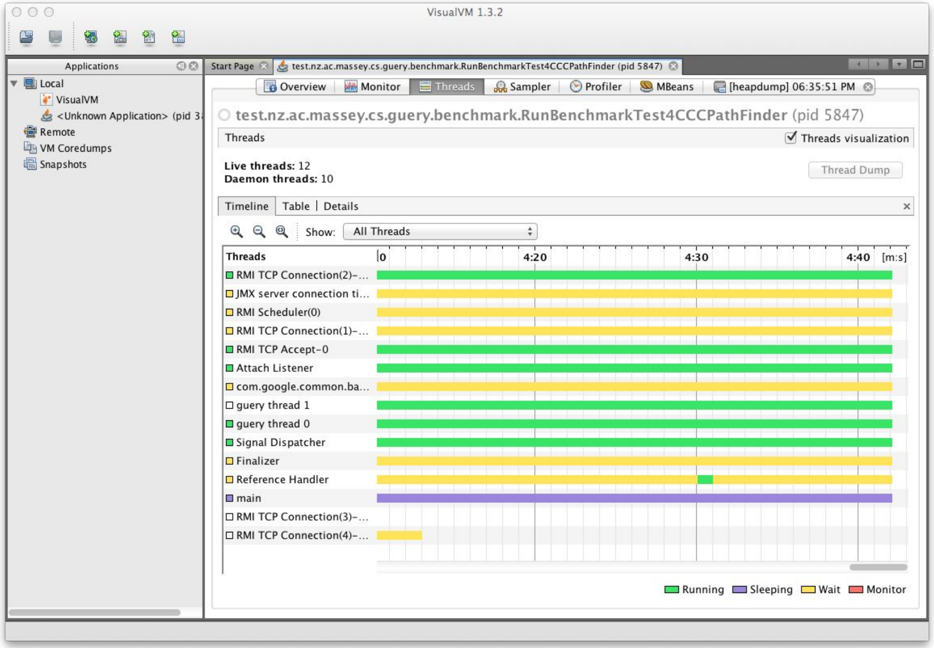
CPU usage: 74.8% **GC activity:** 2.3%

Size: 87,162,880 B **Used:** 30,883,176 B
Max: 1,073,741,824 B

Total loaded: 1,961 **Shared loaded:** 0
Total unloaded: 1 **Shared unloaded:** 0

Live: 13 **Daemon:** 10
Live peak: 14 **Total started:** 18

Thread Info



test.nz.ac.massey.cs.guery.benchmark.RunBenchmarkTest4CCCPathFinder (pid 5847)

Live threads: 12
Daemon threads: 10

Threads
- 🟩 RMI TCP Connection(2)-…
- 🟨 JMX server connection ti…
- 🟨 RMI Scheduler(0)
- 🟨 RMI TCP Connection(1)-…
- 🟩 RMI TCP Accept-0
- 🟩 Attach Listener
- 🟨 com.google.common.ba…
- ☐ guery thread 1
- 🟩 guery thread 0
- 🟩 Signal Dispatcher
- 🟨 Finalizer
- 🟨 Reference Handler
- 🟪 main
- ☐ RMI TCP Connection(3)-…
- ☐ RMI TCP Connection(4)-…

0 4:20 4:30 4:40 [m:s]

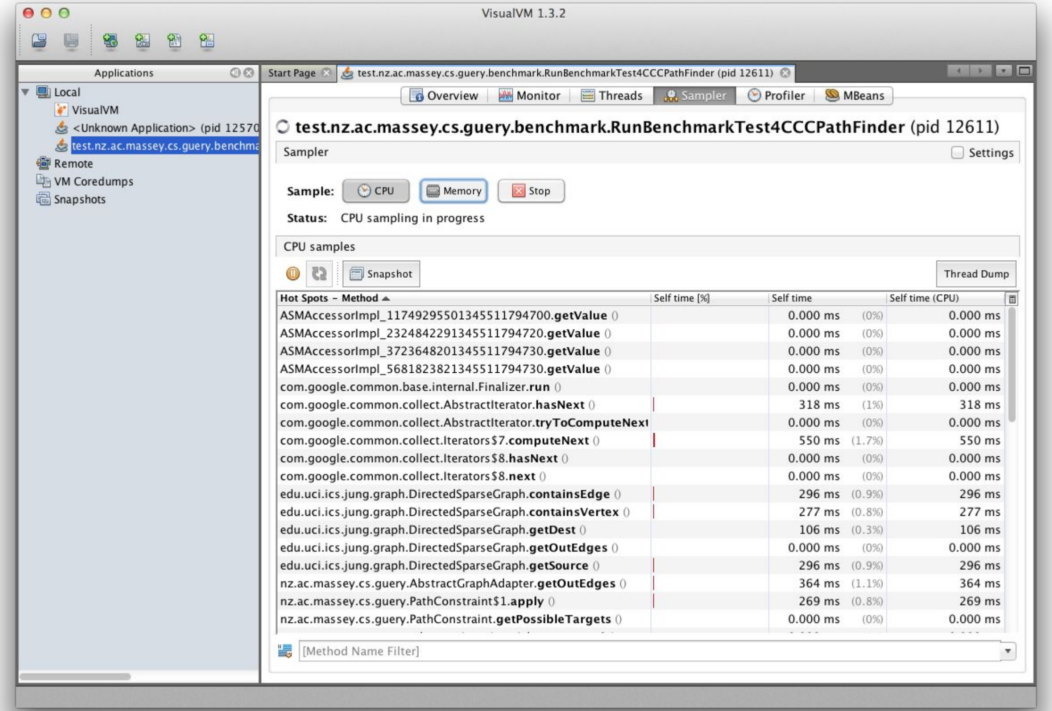
🟩 Running 🟪 Sleeping 🟨 Wait 🟥 Monitor

application threads with colours encoding status (running, sleeping, wait, monitor): can be used to detect deadlocks and other anomalies

Sampling

- non-(less)-invasive technique: the profiler polls the applications periodically by inserting interrupts, and creates reports from these data
- fast but not accurate - generates a statistical approximation
- low overhead, non-invasive - byte code is not changed
- VisualVM supports CPU and memory sampling
- cpu sampling: records how much time was used to execute methods
- memory sampling: records number of instances and bytes used by those instances per type (class, primitive, array)

VisualVM Memory Sampling Example



test.nz.ac.massey.cs.guery.benchmark.RunBenchmarkTest4CCCPathFinder (pid 12611)

Sampler

Sample: CPU Memory Stop

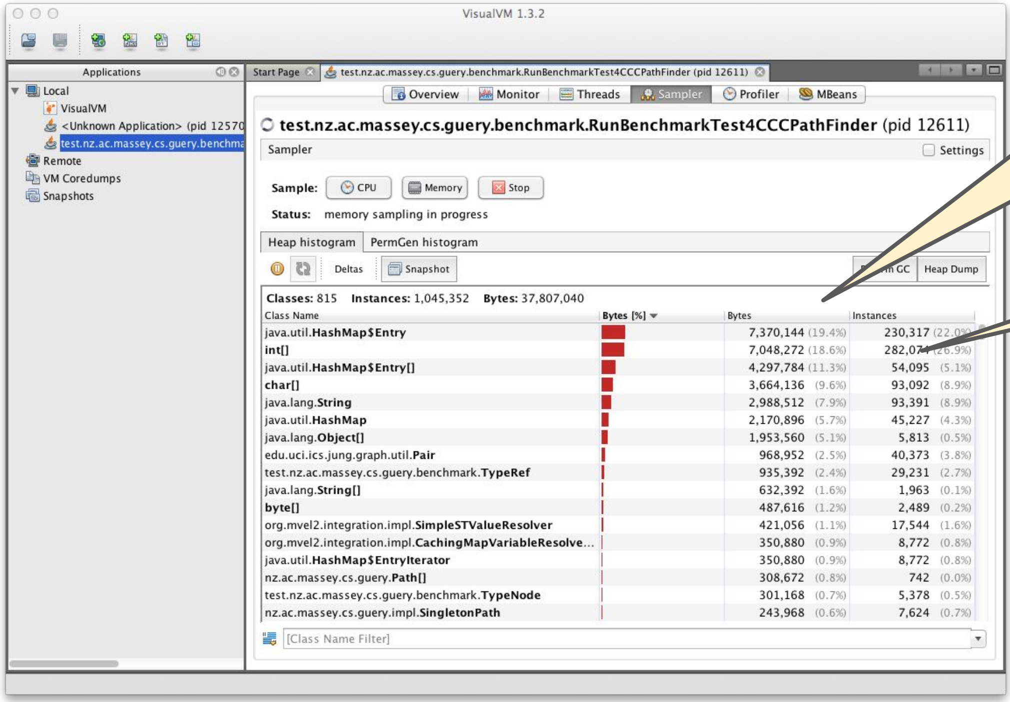
Status: CPU sampling in progress

CPU samples

Hot Spots – Method	Self time [%]	Self time	Self time (CPU)
ASMAccessorImpl_11749295501345511794700.**getValue** ()		0.000 ms (0%)	0.000 ms
ASMAccessorImpl_2324842291345511794720.**getValue** ()		0.000 ms (0%)	0.000 ms
ASMAccessorImpl_3723648201345511794730.**getValue** ()		0.000 ms (0%)	0.000 ms
ASMAccessorImpl_5681823821345511794730.**getValue** ()		0.000 ms (0%)	0.000 ms
com.google.common.base.internal.Finalizer.**run** ()		0.000 ms (0%)	0.000 ms
com.google.common.collect.AbstractIterator.**hasNext** ()		318 ms (1%)	318 ms
com.google.common.collect.AbstractIterator.**tryToComputeNext**		0.000 ms (0%)	0.000 ms
com.google.common.collect.Iterators$7.**computeNext** ()		550 ms (1.7%)	550 ms
com.google.common.collect.Iterators$8.**hasNext** ()		0.000 ms (0%)	0.000 ms
com.google.common.collect.Iterators$8.**next** ()		0.000 ms (0%)	0.000 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.**containsEdge** ()		296 ms (0.9%)	296 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.**containsVertex** ()		277 ms (0.8%)	277 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.**getDest** ()		106 ms (0.3%)	106 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.**getOutEdges** ()		0.000 ms (0%)	0.000 ms
edu.uci.ics.jung.graph.DirectedSparseGraph.**getSource** ()		296 ms (0.9%)	296 ms
nz.ac.massey.cs.guery.AbstractGraphAdapter.**getOutEdges** ()		364 ms (1.1%)	364 ms
nz.ac.massey.cs.guery.PathConstraint$1.**apply** ()		269 ms (0.8%)	269 ms
nz.ac.massey.cs.guery.PathConstraint.**getPossibleTargets** ()		0.000 ms (0%)	0.000 ms

lists absolute (ms) and relative (%) times of time spend on invoking methods

VisualVM CPU Sampling Example



test.nz.ac.massey.cs.guery.benchmark.RunBenchmarkTest4CCCPathFinder (pid 12611)

Sampler

Sample:

Status: memory sampling in progress

Classes: 815 **Instances:** 1,045,352 **Bytes:** 37,807,040

Class Name	Bytes [%]	Bytes	Instances
java.util.**HashMap$Entry**		7,370,144 (19.4%)	230,317 (22.0%)
int[]		7,048,272 (18.6%)	282,074 (26.9%)
java.util.**HashMap$Entry[]**		4,297,784 (11.3%)	54,095 (5.1%)
char[]		3,664,136 (9.6%)	93,092 (8.9%)
java.lang.**String**		2,988,512 (7.9%)	93,391 (8.9%)
java.util.**HashMap**		2,170,896 (5.7%)	45,227 (4.3%)
java.lang.**Object[]**		1,953,560 (5.1%)	5,813 (0.5%)
edu.uci.ics.jung.graph.util.**Pair**		968,952 (2.5%)	40,373 (3.8%)
test.nz.ac.massey.cs.guery.benchmark.**TypeRef**		935,392 (2.4%)	29,231 (2.7%)
java.lang.**String[]**		632,392 (1.6%)	1,963 (0.1%)
byte[]		487,616 (1.2%)	2,489 (0.2%)
org.mvel2.integration.impl.**SimpleSTValueResolver**		421,056 (1.1%)	17,544 (1.6%)
org.mvel2.integration.impl.**CachingMapVariableResolve…**		350,880 (0.9%)	8,772 (0.8%)
java.util.**HashMap$EntryIterator**		350,880 (0.9%)	8,772 (0.8%)
nz.ac.massey.cs.guery.**Path[]**		308,672 (0.8%)	742 (0.0%)
test.nz.ac.massey.cs.guery.benchmark.**TypeNode**		301,168 (0.7%)	5,378 (0.5%)
nz.ac.massey.cs.guery.impl.**SingletonPath**		243,968 (0.6%)	7,624 (0.7%)

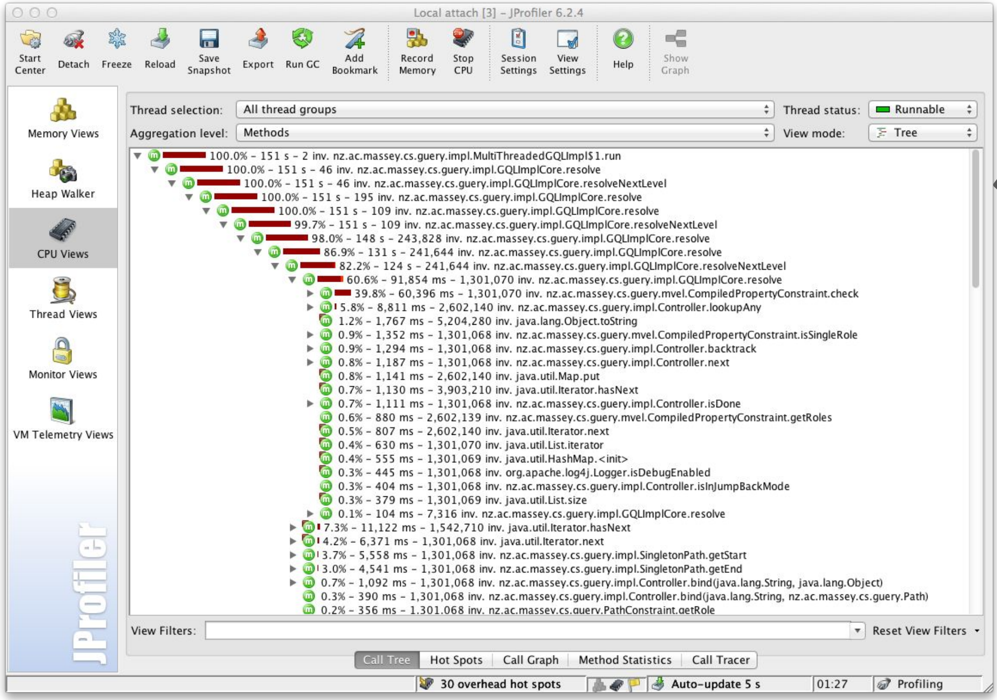
space used (bytes) by instances of the respective classes

instance count for the respective classes

Profiling (vs Sampling)

- invasive technique
- code must be instrumented (probes are inserted) to gather data, i.e. byte code is changed
- this can add significant overhead
- however, profiling can gather more data, and is more precise
- in particular, method invocations can be associated with callees (methods invoking methods), adding more contextual information
- a challenge is to keep profiling transparent - this often fails for large and complex systems, in particular if these systems use instrumentation and custom classloaders (example: application servers)

Profiling View with Call Stack (taken with JProfiler)



Thread selection: All thread groups
Thread status: Runnable
Aggregation level: Methods
View mode: Tree

```
▼ 100.0% – 151 s – 2 inv. nz.ac.massey.cs.guery.impl.MultiThreadedGQLImpl$1.run
  ▼ 100.0% – 151 s – 46 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolve
    ▼ 100.0% – 151 s – 46 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolveNextLevel
      ▼ 100.0% – 151 s – 195 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolve
        ▼ 100.0% – 151 s – 109 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolve
          ▼ 99.7% – 151 s – 109 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolveNextLevel
            ▼ 98.0% – 148 s – 243,828 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolve
              ▼ 86.9% – 131 s – 241,644 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolve
                ▼ 82.2% – 124 s – 241,644 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolveNextLevel
                  ▼ 60.6% – 91,854 ms – 1,301,070 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolve
                    ▶ 39.8% – 60,396 ms – 1,301,070 inv. nz.ac.massey.cs.guery.mvel.CompiledPropertyConstraint.check
                    ▶ 5.8% – 8,811 ms – 2,602,140 inv. nz.ac.massey.cs.guery.impl.Controller.lookupAny
                      1.2% – 1,767 ms – 5,204,280 inv. java.lang.Object.toString
                    ▶ 0.9% – 1,352 ms – 1,301,068 inv. nz.ac.massey.cs.guery.mvel.CompiledPropertyConstraint.isSingleRole
                    ▶ 0.9% – 1,294 ms – 1,301,068 inv. nz.ac.massey.cs.guery.impl.Controller.backtrack
                    ▶ 0.8% – 1,187 ms – 1,301,068 inv. nz.ac.massey.cs.guery.impl.Controller.next
                      0.8% – 1,141 ms – 2,602,140 inv. java.util.Map.put
                      0.7% – 1,130 ms – 3,903,210 inv. java.util.Iterator.hasNext
                    ▶ 0.7% – 1,111 ms – 1,301,068 inv. nz.ac.massey.cs.guery.impl.Controller.isDone
                      0.6% – 880 ms – 2,602,139 inv. nz.ac.massey.cs.guery.mvel.CompiledPropertyConstraint.getRoles
                      0.5% – 807 ms – 2,602,140 inv. java.util.Iterator.next
                      0.4% – 630 ms – 1,301,070 inv. java.util.List.iterator
                      0.4% – 555 ms – 1,301,069 inv. java.util.HashMap.<init>
                      0.3% – 445 ms – 1,301,068 inv. org.apache.log4j.Logger.isDebugEnabled
                      0.3% – 404 ms – 1,301,068 inv. nz.ac.massey.cs.guery.impl.Controller.isInJumpBackMode
                      0.3% – 379 ms – 1,301,069 inv. java.util.List.size
                    ▶ 0.1% – 104 ms – 7,316 inv. nz.ac.massey.cs.guery.impl.GQLImplCore.resolve
                  ▶ 7.3% – 11,122 ms – 1,542,710 inv. java.util.Iterator.hasNext
                  ▶ 4.2% – 6,371 ms – 1,301,068 inv. java.util.Iterator.next
                  ▶ 3.7% – 5,558 ms – 1,301,068 inv. nz.ac.massey.cs.guery.impl.SingletonPath.getStart
                  ▶ 3.0% – 4,541 ms – 1,301,068 inv. nz.ac.massey.cs.guery.impl.SingletonPath.getEnd
                  ▶ 0.7% – 1,092 ms – 1,301,068 inv. nz.ac.massey.cs.guery.impl.Controller.bind(java.lang.String, java.lang.Object)
                    0.3% – 390 ms – 1,301,068 inv. nz.ac.massey.cs.guery.impl.Controller.bind(java.lang.String, nz.ac.massey.cs.guery.Path)
                    0.2% – 356 ms – 1,301,068 inv. nz.ac.massey.cs.guery.PathConstraint.getRole
```

View Filters:
Reset View Filters

hierarchical breakdown using the call stack: how much time was spent on each invocation of other methods.

invocations have **context** !

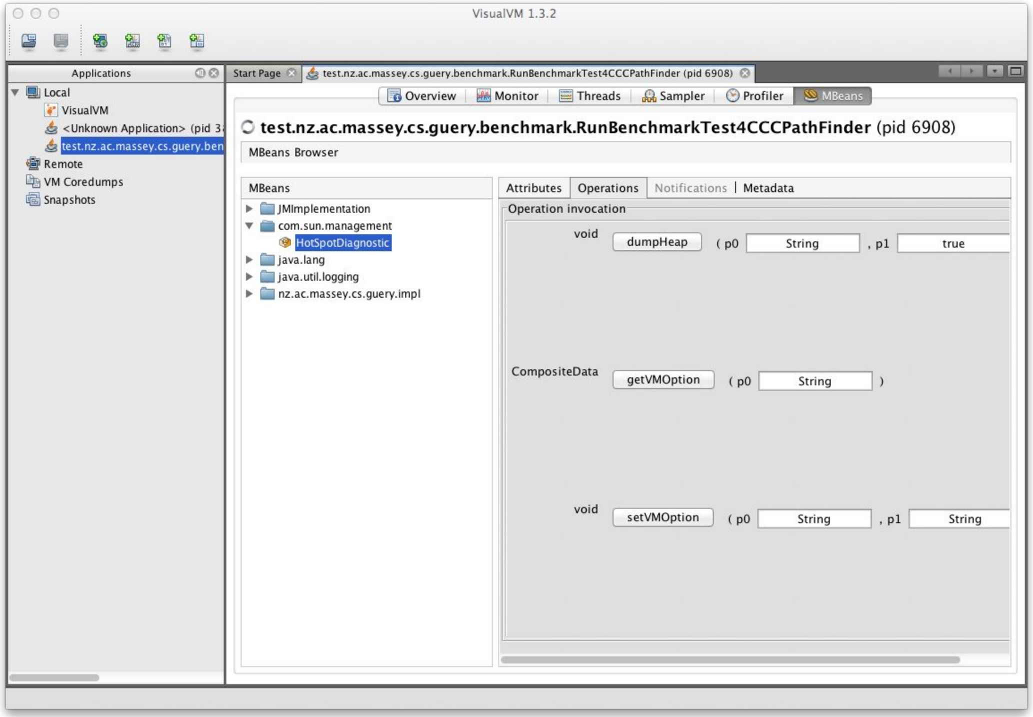
Heap and Thread Dumps

- heap and thread dumps can be taken by VisualVM and other profiling tools from a running application and saved for later static analysis
- heap dumps are also useful for **post-mortem analysis** -- to investigate reasons for application failure, dumps are produce when applications crash
- CLI tools: **`jstack, jmap, jcmd`**
- visualvm can be started to produce a heap dump when it fails with a memory leak
- the stackwalker API can be used within applications to explore the stack (without raising exceptions), e.g. to implement methods that are sensitive to calling context
- see also: https://www.baeldung.com/java-heap-dump-capture

Java Management Extensions (JMX)

- JMX is a technology that allows applications to connect to JVM applications
- with JMX, methods can be invoked
- this can be used to perform actions, or to query the state of the application
- to support JMX from the host application (the application to be monitored), MBeans need to be implemented and registered
- there are multiple standards MBeans in the core class libraries and common libraries

Taking Heap Dumps via JMX


VisualVM 1.3.2
Applications
Local
VisualVM
<Unknown Application> (pid 3
test.nz.ac.massey.cs.guery.ben
Remote
VM Coredumps
Snapshots
Start Page
test.nz.ac.massey.cs.guery.benchmark.RunBenchmarkTest4CCCPathFinder (pid 6908)
Overview
Monitor
Threads
Sampler
Profiler
MBeans
test.nz.ac.massey.cs.guery.benchmark.RunBenchmarkTest4CCCPathFinder (pid 6908)
MBeans Browser
MBeans
JMImplementation
com.sun.management
HotSpotDiagnostic
java.lang
java.util.logging
nz.ac.massey.cs.guery.impl
Attributes
Operations
Notifications
Metadata
Operation invocation
void dumpHeap (p0 String , p1 true
CompositeData getVMOption (p0 String)
void setVMOption (p0 String , p1 String

this form is generated from the signature of a method: the name of the method is **dumpHeap** with two parameters (a string and a boolean)

Implementing a Simple MBean

```
public interface GQLMonitorMBean {
    public int getVertexCount ();
    public int getProcessedVertexCount ();
}
```

- example from GUERY library: https://bitbucket.org/jensdietrich/guery/
- code **here**
- monitor progress of engine that processes queries on graphs
- goal: try to get progress as ratio processed vertex count / vertex count
- MBean is interface, a simple implementation class must also be provided

Registering a Simple MBean

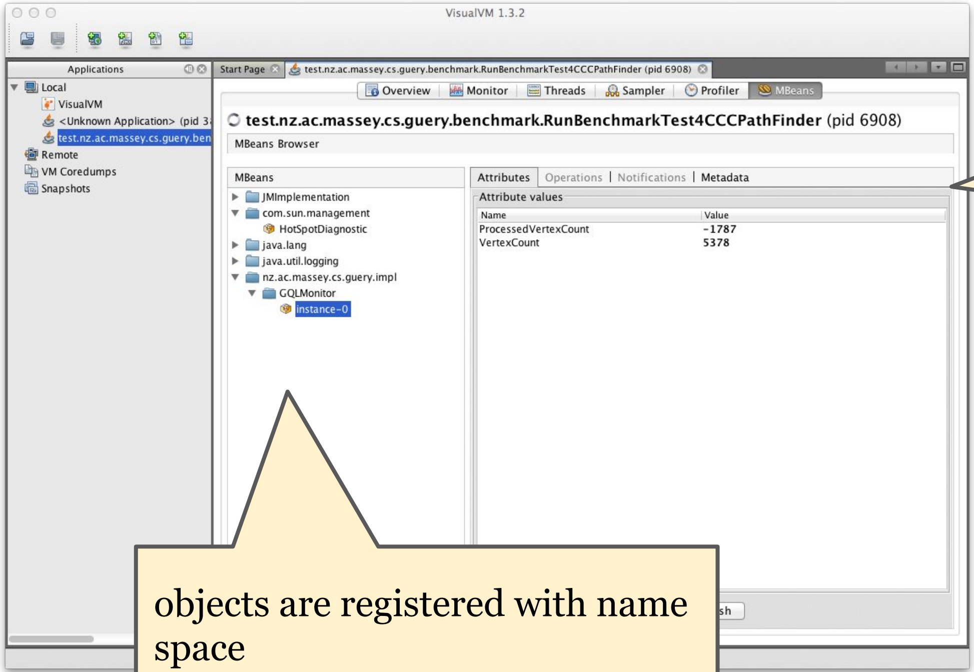
```
final GQLMonitorMBean monitor = ..
MBeanServer mbs = ManagementFactory.getPlatformMBeanServer();
int id = session_counter++;
ObjectName name =
    new ObjectName("nz.ac.massey..:type=GQLMonitor,id=instance-"+id);
mbs.registerMBean(monitor, name);
```

creation

registration

- mbeans are registered upon creation in a global registry by name
- names are not just strings, a certain type **`ObjectName`** is used for this purpose
- (monitoring) clients can discover mbeans at runtime by name

Using the MBean (Example: VisualVM)



Name	Value
ProcessedVertexCount	-1787
VertexCount	5378

properties are **polled**, an explicit refresh is needed to update

objects are registered with name space

JMX-Monitoring vs Logging

- both are used to monitor applications
- JMX works over networks, enables remote monitoring
- this is also possible with logging, but requires special appenders
- JMX **pulls information** on demand **from the application**
- logging **pushes information out of the application**
- with JMX, methods can be exposed and executed from the monitoring application (example: clearing caches, shutting down applications)

Selected Topics in Testing

Automated Regression Testing

- widely used QA technique
- tests are completely automated and executed during builds
- tests passes are often preconditions for release
- tests provide evidence (but not proofs) of correctness

Tests Fixtures

- code executed before / after each / all tests
- JUnit4: **`@Before, @After, @BeforeClass, @AfterClass`** annotated methods
- JUnit5: **`@BeforeEach, @AfterEach, @BeforeAll, @AfterAll`** annotated methods
- setup and cleanup predictable environment for tests
- tests should not interfere, and should not require a certain order of execution
- can be difficult to achieve if tests run against shared persistent data, junit offers annotations to customise settings
- for complex environment dependencies, mocking should be used (see service section)

Tests as Rules

if service is available query service and check result **(pseudo code!)**

```
// if: service is available
assume that httpclient.connect(“http:localhost”)

// do: query service
String result = httpclient.get(“http:localhost/foo”);

// then: check expected state
assert that result.contains(42)
```

note: **`assert`** is not the assert statement here, but represents an invocation of one of JUnit’s **`assert*`** methods

Assumptions

- if assumptions fail, tests will be ignored
- test frameworks set test outcome to ***skip***
- some test runners will flag such tests as passed (consistent with the semantic of material implication)

Assertions

- check state after execution of tested code
- basic form: **`assertTrue(..)`** -- check whether state satisfies a certain condition
- common case: compare value against expected state
  `assertEquals(42,value)`
- expected state (42 in example) is called the **test oracle**

Sourcing Test Oracles

- input specific: compute manually ("2+2 should be 4")
- computed by other means (when porting or modernising a program, e.g. oracle provided by a spreadsheet)
- statistical oracle
- generic oracle: invariants that should always be true
  example: code should not throw a runtime exception
- oracles for handling of abnormal executions: expected exceptions

Possible Outcomes of a Test

- **fail** – assertion fails (JUnit internally: AssertionError encountered)
- **skip** – assumption fails
- **error** – some other exception or error has been encountered (*implicit* oracle: test code should not produce such an exception or error)
- **success** - otherwise

Coverage

- given some tests, how much code is executed when tests run ?
- different aggregation levels: branch, method, class, package, ..
- reveals untested code
- can be easily measured (emma/jacoco, clover, ..), integrations in build tools and IDEs
- **high test coverage is desirable, but no guarantee for code code quality**, in particular when assertions are meaningless (**`assertTrue(true)`**);

Inside Code Coverage

```
foo(int i) {
    if (i==0) {
        ..
    }
    else {
        ...
    }
}
```

```
foo(int i) {
    visited("foo");
    if (i==0) {
        visited("foo-branch1");
        ..
    }
    else {
        visited("foo-branch2");
        ...
    }
}
```

A coverage tool ***instruments*** the code (usually byte code) and inserts probes

label	**visited**
foo	true
foo-branch1	true
foo-branch2	false

For each probe, a unique label is created with a flag whether this label was visited during tests. When the probe executed, the flag is flipped to true. The coverage metrics are then computed from this table.

What is a Good Test ?

- coverage + assertions
- even with high coverage, tests may be poor if assertions are too coarse (e.g. if there are no assertion or only **`assertTrue(true)`** assertion, tests only check for unhandled exceptions !)
- good tests reveal bugs (see TDD)

Mutation Testing

- alternative approach to assess the quality of tests
- how can we establish that tests like this are meaningless

```
@Test test() {

    // exercise program under test, high coverage
    ..
    // but test always succeeds !

    assertTrue(true);
}
```

Mutation Testing (ctd)

- tested code is mutated (faults are injected), e.g. conditions are negated
- mutated code is then tested and results are classified as follows:
- killed - test on mutated code fails -- good !
- survived- test on mutated code succeeds -- bad
- **mutation coverage** is then computed as the ratio between killed and killed+survived

Mutation Testing (ctd)

- if tests have meaningless assertions (assertTrue(true)), then many tests are likely to survive (unless changes trigger exceptions) as they are not sensitive to the actual program behaviour -- so mutation coverage will be low
- tool: http://pitest.org/
- example: https://github.com/jensdietrich/se-teaching/tree/main/mutationtesting

Generating Tests

- increasingly popular technique to complement manual testing
- create random tests to check whether program fails
- example: create http traffic (jmeter) to stress test server, create UI events to break mobile application ("monkeys" -- https://developer.android.com/studio/test/monkey), netflix chaos monkey to shutdown VMs etc
- monkeys -- reference to infinite monkey theorem

Generating Tests Challenges

- usually test generation has two aspects:
- generate scripts or data input to exercise a program
- generate assertions, i.e. oracles -- this is much harder as it requires inferring what the programmers intention is what constitutes correct state
- easier case: absence of abnormal behaviour (exceptions, overflows, null pointers, ..)
- the input generation part is often referred to as **fuzzing**

Types of Fuzzing / Input Generation

blackbox -- purely random input generation. Example: jmeter -- generate http traffic, look for response times and server errors

greybox -- input generation driven by feedback from program, e.g. coverage, often based on genetic algorithms, often highly successful in detecting bugs and vulnerabilities in low-level software, e.g. AFL

whitebox -- build sophisticated models of program execution (e.g. using symbolic execution), example: path finder

Example: Quickcheck

- originally developed for Haskell, ported to many languages
- Java port: https://github.com/pholser/junit-quickcheck (also in maven repo, https://mvnrepository.com/artifact/com.pholser)
- generators create random input used for tests
- assertion test for generic properties (invariants)

Quickcheck Example

```
public class Rectangle {
    private int length = 1;
    private int height = 1;
    public Rectangle(int length, int y) {
        this.length = length;
        this.height = y;
    }
    public int getLength() {return length;}
    public int getHeight() {return height;}
    public int size () {
        return length * height;
    }
}
```

simple class, use quickcheck to test size

assumption: because size is the product of length and height, it should be **divisible** by both

https://github.com/jensdietrich/se-teaching/tree/main/quickcheck

Generating Random Input Data for Testing

```
public class LargeRectangleGenerator extends Generator<Rectangle> {
    ..
    public Rectangle generate(SourceOfRandomness sourceOfRandomness, ..) {
        int x = sourceOfRandomness.nextInt(1,Integer.MAX_VALUE);
        int y = sourceOfRandomness.nextInt(1,Integer.MAX_VALUE);
        return new Rectangle(x,y);
    }
}
```

arbitrarily large int
are generated

Testing Invariants

special runner used to execute tests with generated input

generator supplies input data

```
@RunWith(JUnitQuickcheck.class)
public class RectanglesTest {
    @Property(trials = 100)
    public void sizeCanBeDividedByLength(@From(LargeRectangleGenerator.class)
             Rectangle rect) {
        assertTrue(rect.size()%rect.getLength()==0);
    }
    @Property(trials = 100)
    public void sizeCanBeDividedByHeight(@From(LargeRectangleGenerator.class)
             Rectangle rect) {
        assertTrue(rect.size()%rect.getHeight()==0);
    }
}
```

Test Results

```
java.lang.AssertionError: Property named 'sizeCanBeDividedByLength' failed:

With arguments: [Rectangle{length=1033274003, height=408517975}]
```

- since the code does not check for overflows and underflows, the assumption that the size can be divided by length is incorrect
- tests provide counter example

Finding Hash Contract Violations

```
@Override
public boolean equals(Object o) {
     if (this == o) return true;
     if (o == null || getClass() != o.getClass()) return false;
     Rectangle rectangle = (Rectangle) o;
     return height == rectangle.length;
}

@Override
public int hashCode() {
     int result = length;
     result = 31 * result + height;
     return result;
}
```

if height and length are equal, two rectangles can be equal, but have different hash codes

Quickcheck Test

“quadratic blowup” since it is unlikely to find a matching pair , a larger number of trials must be chosen

requires two randomly created instances

```
@Property(trials = 100_000)
public void equalRectanglesShouldHaveSameHashcode(
    @From(SmallRectangleGenerator.class) Rectangle rect1,
    @From(SmallRectangleGenerator.class) Rectangle rect2) {
        assertTrue(
            !rect1.equals(rect2) || rect1.hashCode()==rect2.hashCode()
        );
    }
```

Test Results

```
java.lang.AssertionError: Property named
'equalRectanglesShouldHaveSameHashcode' failed:

With arguments: [Rectangle{length=439, height=2}, Rectangle{length=2,
height=284}]
```

- test provides counter example

SmallRectangleGenerator vs LargeRectangleGenerator

- to detect overflow-related bugs, large numbers have to be used, i.e./ length and height are computed using **`sourceOfRandomness.nextInt(1,Integer.MAX_VALUE)`**
- to detect the hash contract violation, this is not suitable: the probability of creating the same positive number twice is ca $4.6 * 10^{18}$
- solution: a different generator is used to create small rectangles with (length / height) values between 1 and 1000
- illustrates that some pre-analysis if often necessary to write good (quickcheck) tests