

# Testen von nebenläufigen Programmen

Stefan Finkenzeller - Philip Frank - 4.7.08

# Was wird überhaupt getestet?

- Testen der Korrektheit
  - Finden von Programmierfehlern  
Invarianten werden eingehalten, keine Deadlocks entstehen, ...
- Testen der Performance
  - Optimierung der Laufzeit  
Datendurchsatz, Reaktionszeit

# Versuchskaninchen: BoundedBuffer

```
public class BoundedBuffer<E> {  
    [...]  
    public boolean isEmpty();  
    public boolean isFull();  
    public void put(E x);  
    public E take();  
}
```

- Implementiert blockierende Warteschlange
- Benutzt Semaphore
- Listing 12.1

# Korrektheit

- Tests mit nur einem Thread um Fehlerquellen auszuschließen

```
void singleThreadedTest() {
    BoundedBuffer<Double> bb = new BoundedBuffer<Double>(10);
    double sum1 = 0.0;
    for (int i = 0; i < 10; i++) {
        double item = Math.random();
        sum1 += item;
        bb.put(item);
    }
    double sum2 = 0.0;
    for (int i = 0; i < 10; i++) {
        sum2 += bb.take();
    }
    assertEquals(sum1, sum2);
}
```

# Korrektheit

- Invarianten
- Daten im Speicher
  - Variablen, Listen, Bäume, ...

Testoperation muss atomar sein

Oder an festgelegten "sicheren" Punkten erfolgen

# Korrektheit

- Blocking

```
void testTakeBlocksWhenEmpty() {
    final BoundedBuffer<Integer> bb = new BoundedBuffer<Integer>(10);
    Thread taker = new Thread() {
        public void run() {
            try {
                int unused = bb.take();
                fail(); // if we get here, it's an error
            } catch (InterruptedException success) { }
        }
    };
    try {
        taker.start();
        Thread.sleep(LOCKUP_DETECT_TIMEOUT);
        taker.interrupt();
        taker.join(LOCKUP_DETECT_TIMEOUT);
        assertFalse(taker.isAlive());
    } catch (Exception unexpected) { fail(); }
}
```

# Korrektheit

- Safety
  - „Nichts Schlimmes / Falsches passiert“
- Wichtig: Testdaten sollen vom Compiler nicht erraten werden können
  - Schlechtes Beispiel: Fortlaufende Zahlen
  - Gutes Beispiel: `System.nanoTime()`

- Listing 12.5 - PutTakeTest
  - Gleich viele Produzenten- und Konsumenten-Threads werden gestartet
  - Produzent füllt den Buffer mit (pseudo-) zufälligen Integern
  - Die Summe der produzierten und konsumierten Zahlen wird auf Gleichheit überprüft

```

class Producer implements Runnable {
    public void run() {
        try {
            int seed = (this.hashCode() ^ (int)System.nanoTime());
            int sum = 0;
            barrier.await();
            for (int i = nTrials; i > 0; --i) {
                bb.put(seed);
                sum += seed;
                seed = xorShift(seed);
            }
            putSum.getAndAdd(sum);
            barrier.await();
        }
        catch (Exception e) { throw new RuntimeException(e); }
    }
}

```

```

static int xorShift(int y) {
    y ^= (y << 6);
    y ^= (y >>> 21);
    y ^= (y << 7);
    return y;
}

```

```

class Consumer implements Runnable {
    public void run() {
        try {
            barrier.await();
            int sum = 0;
            for (int i = nTrials; i > 0; --i) {
                sum += bb.take();
            }
            takeSum.getAndAdd(sum);
            barrier.await();
        }
        catch (Exception e) { throw new RuntimeException(e); }
    }
}

```

```

[...] assertEquals(putSum.get(), takeSum.get()); [...]

```

# Liveness Tests

- „Schließlich passiert das Richtige.“
- Das Programm bleibt an keiner Stelle „stecken“
  - Deadlocks, Endlosschleifen
  - Endloses Wiederholen bei Fehlschlägen

# Resource Management

- Verweise auf nicht mehr benötigte Objekte müssen aufgegeben werden, damit die Garbage Collection Speicher und andere Ressourcen freigeben kann
  - keine „Leaks“
- Möglichkeiten den Speicherverbrauch zu messen:
  - `Runtime.freeMemory()`
  - **Garbage Collection:** `System.gc()`,  
`System.runFinalization()`

# Callbacks benutzen

- Hilfsklassen der zu testenden Klasse erweitern und entsprechende Methoden überschreiben um die Aufrufe („Callbacks“) zu überwachen
- Beispiel: ThreadFactory wird erweitert um die von einem ThreadPool erzeugten Threads zu zählen (Listing 12.8 & 12.9)

# Verschränkungen erzeugen

→ Viele Kontextwechsel

- `Thread.sleep()`
- `Thread.yield()`

# Performance Tests

## → Messen der Laufzeit

- Datendurchsatz
- Reaktionszeit
- Scalability

## → Ziel: Finden des optimalen Algorithmus, anpassen der Einstellungen (z.B. Threadpoolgröße)

# BarrierTimer

```
this.timer = new BarrierTimer();

this.barrier = new CyclicBarrier(npairs * 2 + 1, timer);

public class BarrierTimer implements Runnable {

    private boolean started;
    private long startTime, endTime;

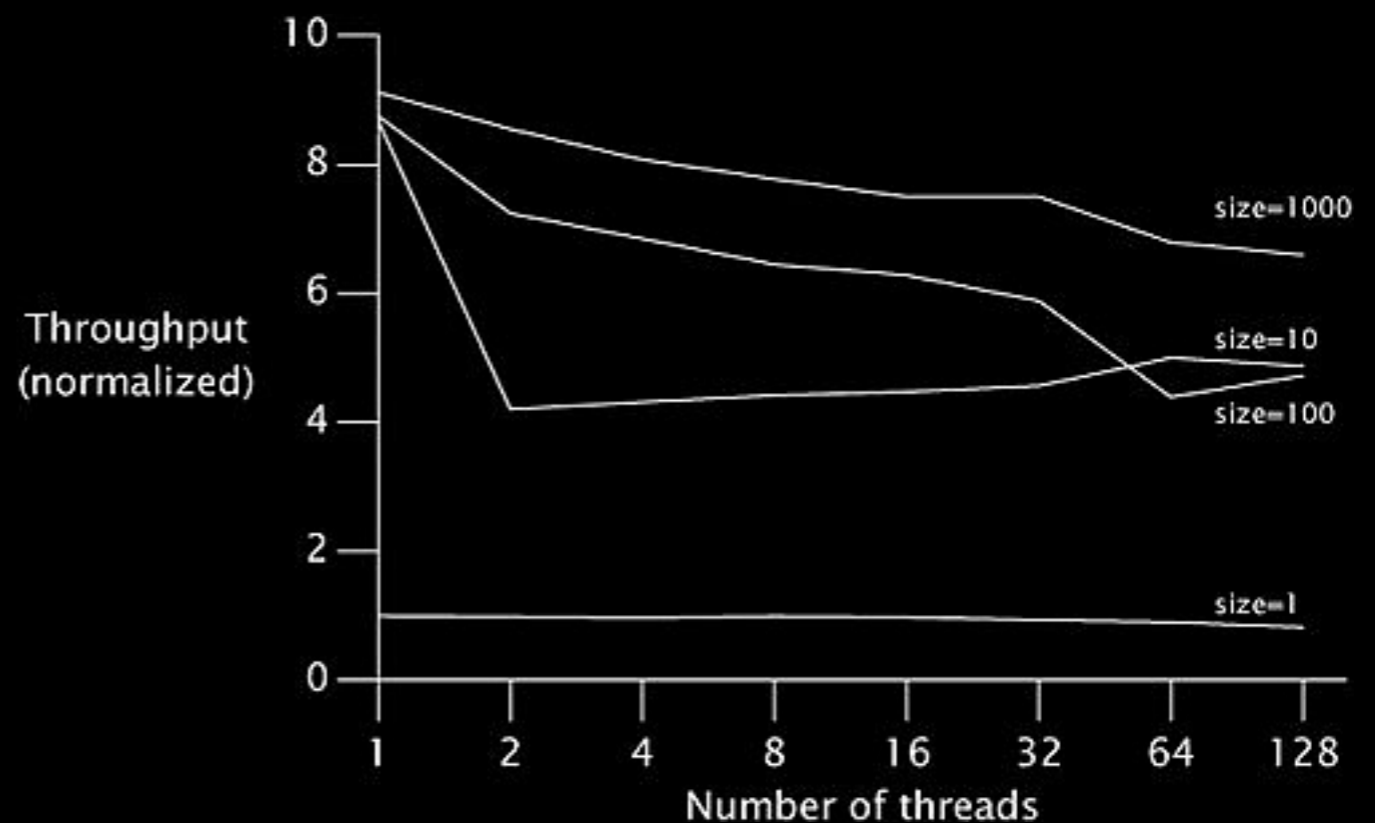
    public synchronized void run() {
        long t = System.nanoTime();
        if (!started) {
            started = true;
            startTime = t;
        }
        else
            endTime = t;
    }

    public synchronized void clear() {
        started = false;
    }

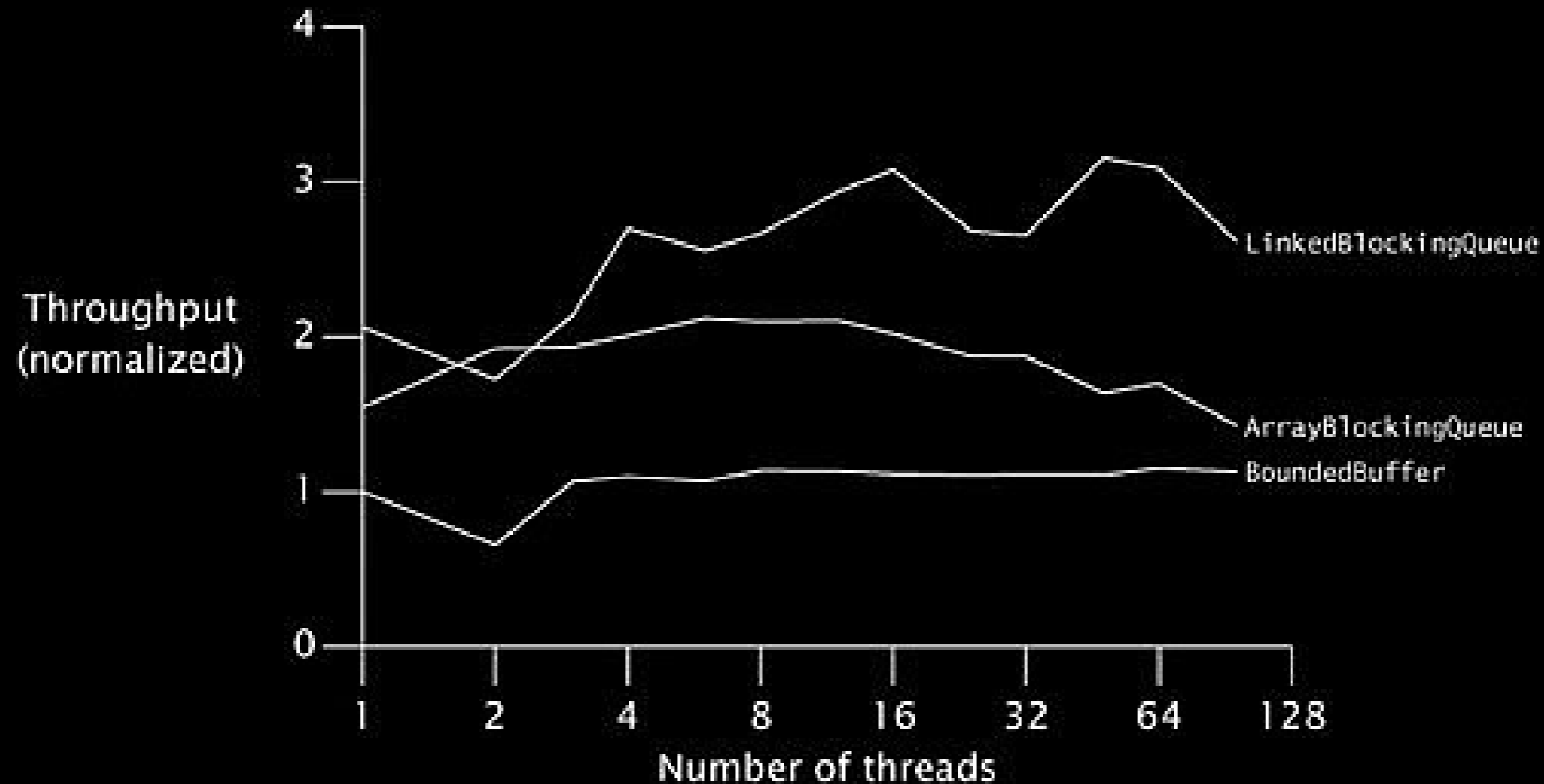
    public synchronized long getTime() {
        return endTime - startTime;
    }
}
```

# Performance von BoundedBuffer

- Performance nimmt bei hoher Threadzahl eher ab
- Grund: Viel Zeit wird mit dem blocken und entblocken von Threads verbraucht.



# Vergleich Verschiedener Algorithmen



→ Libraries benutzen



# Falls tricke beim Testen

- Eigenheiten des Compilers und der Virtual Machine, die beim Testen beachtet werden sollten

# Garbage Collection

- Tritt quasi unvorhersehbar auf.
- Lässt sich nicht ausschalten.
- `Per -verbose:gc` Aktivität sichtbar
- `System.gc ();`

# Dynamische Kompilierung

- JVM benutzt teilweise vorkompilierten Bytecode und teilweise dynamisch bei der Ausführung kompilierten Code
- Lösung: Zeit einräumen
  - Zeitmessungen nicht gleich beim Programmstart beginnen
  - Programm „warmlaufen“ lassen
  - Problem: Konstante Rekompilierung und Optimierung
- Grundsätzlich: Tests immer mehrmals laufen lassen, auch in der selben Instanz eines Programms

# Unrealistische Programmabläufe

- Compiler optimiert einzelne Methoden in Abstimmung auf das gesamte Programm
  - Unterschiede vom Testprogramm zum wirklichen Einsatzgebiet können Unterschiede in der Performance verursachen
- Tests möglich nah an der Wirklichkeit realisieren

# Unrealistische Arbeitsaufwände

- Relation von Berechnungen, Zugriffen auf synchronisierte Daten und andere Operationen sollte möglichst realitätsnah sein.
- Schlechtes Beispiel: `PutTakeTest`, entspricht u. U. nicht der Wirklichkeit, da keine aufwendigen Berechnungen durchgeführt werden, sondern nur der Zugriff auf die Liste getestet wird.

# Dead Code Elimination

- Code des Tests wird u. U. vom Compiler „wegoptimiert“, da er auf das Ergebnis des Programms scheinbar keinen Einfluss hat
  - Für den Compiler unvorhersehbare Ergebnisse erzeugen und auch verwenden/ausgeben
  - Statische Eingabedaten vermeiden

# Tipps , Tricks und Tools

# Codereview

- Durchsicht des Codes durch einen anderen Programmierer
- Häufig: Programmierer und Reviewer besprechen den Code gemeinsam
- Entdecken von Fehlern / Verbesserungsmöglichkeiten
- Höhere Qualität der Kommentare
- Erforderlich: Taktgefühl beim Reviewer, Kritikfähigkeit beim Programmierer

# FindBugs

<http://findbugs.sourceforge.net/>

- Statisches Analysetool
- Untersucht Java Code auf sehr viele häufige Fehler, beispielsweise bei
  - Komplizierten Sprachfeatures
  - Missverständlichen API-Definitionen
  - Tippfehlern

# FindBugs

<http://findbugs.sourceforge.net/>

## FindBugs (1.2.1-dev-20070506) Analysis for jdk1.7.0-b12

Bug Summary

Analysis Information

List bugs by bug category

List bugs by package

AA P1 AA P2 AA P3 AA Exp.

Bad practice (954: 118/836/0/0)

Correctness (249: 81/168/0/0)

Multithreaded correctness (272: 1/271/0/0)

DC: Possible double check of field (76: 0/76/0/0)

IS: Inconsistent synchronization (121: 0/121/0/0)

LI: Unsynchronized Lazy Initialization (1: 0/1/0/0)

ML: Synchronization on updated field (Mutable Lock) (3: 0/3/0/0)

MWN: Mismatched wait() or notify() (1: 0/1/0/0)

NN: Naked notify (9: 0/9/0/0)

RS: Class's readObject() method is synchronized (9: 0/9/0/0)

Ru: Method invokes run() (1: 0/1/0/0)

Invokes run on a thread (did you mean to start it instead?) AA (1: 0/1/0/0)

SC: Constructor invokes Thread.start() (4: 0/4/0/0)

SWL: Sle Ru / RU\_INVOKE\_RUN

TLW: Wa This method explicitly invokes run () on an object. In general, classes implement the Runnable

UG: Unsy interface because they are going to have their run () method invoked in a new thread, in which case

UL: Lock Thread.start () is the right method to call.

UW: Unconditional wait (6: 0/6/0/0)

VO: Use of volatile (2: 0/2/0/0)

Wa: Wait not in loop (8: 0/8/0/0)

Performance (1772: 7/1765/0/0)

Dodgy (654: 52/602/0/0)

# PMD

<http://pmd.sourceforge.net/>

- Statisches Analysetool
- Nicht nur zur Suche von Bugs, findet optional beispielsweise auch:
  - Unbenutzten Code
  - Probleme bei der Migration zu einer bestimmten Version des JDK
- Vorteil: Definition eigener Regeln problemlos möglich, beispielsweise zur Einhaltung eigener Policies
- Lässt sich in unzählige IDEs integrieren

# NetBeans

<http://www.netbeans.org>

- Entwicklungsumgebung ähnlich Eclipse
- Unterstützt neben Java auch C++, Ruby, PHP und viele andere Sprachen
- Windows, Linux, Mac OS X und Solaris
- Nützlich für Performancetests: Profiler
  - Messung von CPU-Zeit
  - Ansicht des Heaps
  - Setzen spezieller Testpunkte im Code

# NetBeans Profiler

**NetBeans IDE 6.1**

File Edit View Navigate Source Refactor Build Run Profile Versioning Tools Window Help

<default config>

**Profiler**

Controls: Take Snapshot, Live Results, Reset Collected Results

Status: Saved Snapshots, View

Basic Telemetry:

- Instrumented: 82 Methods
- Filter: Profile only project classes
- Threads: 9
- Total Memory: 35,442,688 B
- Used Memory: 32,777,168 B
- Time Spent in GC: 0.2%

Call Tree - Method

Method	Time [%]	Time	Invocations
All threads		20690 ms (100%)	1
pool-1-thread-2		9818 ms (100%)	1
marsroviewer.LoadImageTask.doInBackground ()		9818 ms (100%)	1
marsroviewer.LoadImageTask.doInBackground ()		9818 ms (100%)	1
Self time		0.061 ms (0%)	1
pool-1-thread-3		6106 ms (100%)	1
pool-1-thread-1		3463 ms (100%)	1
marsroviewer.LoadImageTask.doInBackground ()		3463 ms (100%)	1
marsroviewer.LoadImageTask.doInBackground ()		3463 ms (100%)	1
marsroviewer.LoadImageTask.loadImage (javax.imageio.ImageReader, javax.imageio.ev...		2015 ms (58.2%)	1
marsroviewer.LoadImageTask.findImageReader (java.net.URL)		1422 ms (41.1%)	1
Self time		25.6 ms (0.7%)	1
marsroviewer.LoadImageTask\$1.<init> (marsroviewer.LoadImageTask)		0.062 ms (0%)	1
Self time		0.138 ms (0%)	1
AWT-EventQueue-0		1302 ms (100%)	1

Call Tree | Hot Spots | Combined | Info

**VM Telemetry Overview**

16:19:10: Heap Size, Used Heap

16:19:15: Living Generations, Relative Time Spent

16:19:15: Threads, Loaded Classes

**Profiling Points**

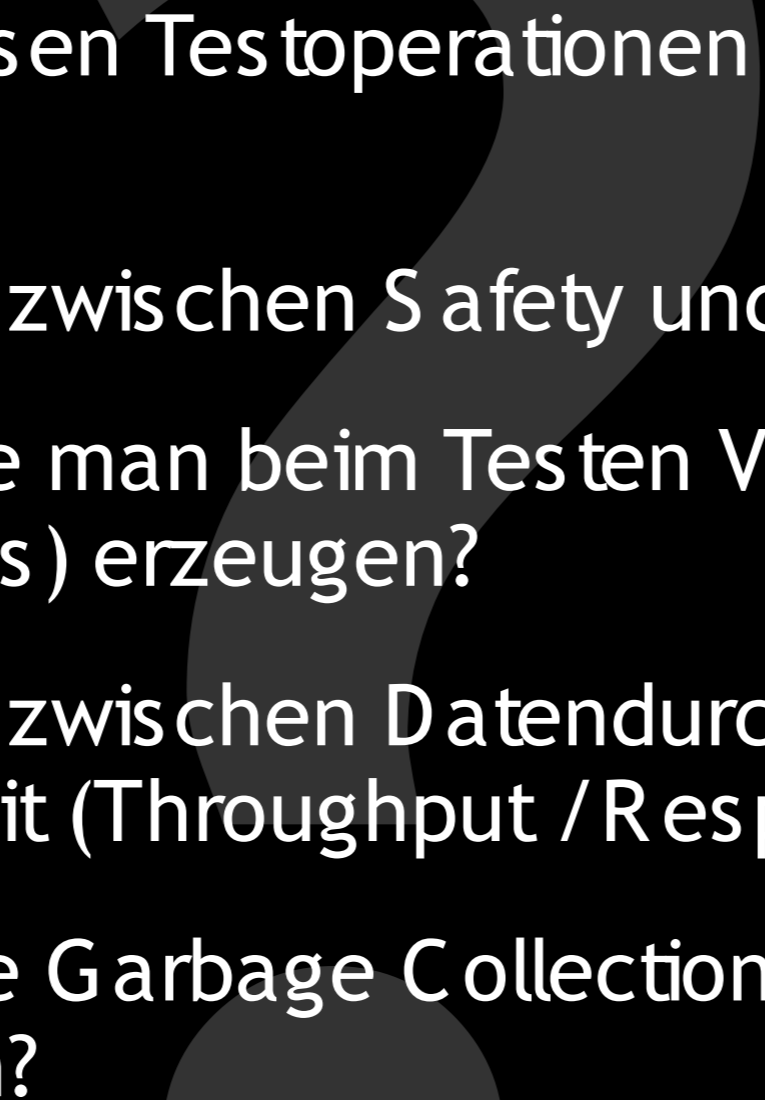
Project: All Projects

Scope	Project	Profiling Point	Results
	MarsRoverViewer	Take Snapshot at MarsRoverViewerA...	No results available

MarsRoverViewer (profile) running...

# Was noch zu sagen bleibt...

- Testen  $\neq$  alle Bugs finden
- Testen = Vertrauen steigern
- „Code a little, test a little“
- Kommentare & Annotations
- Tools, Tools, Tools

- 
- Warum müssen Testoperationen meistens atomar sein?
  - Unterschied zwischen Safety und Liveness?
  - Warum sollte man beim Testen Verschränkungen (Interleavings) erzeugen?
  - Unterschied zwischen Datendurchsatz und Reaktionszeit (Throughput / Responsiveness)?
  - Wie kann die Garbage Collection Testergebnisse beeinflussen?