Lust durch Last

# Last- und Performance-Tests komplexer Java-Swing-Applikationen

**Klaus P. Berg**

*Bei Kraftwerksautomatisierungssoftware wird aus verständlichen Gründen großer Wert sowohl auf die funktionale Korrektheit des Systems als auch die Erfüllung nicht-funktionaler Anforderungen wie Performance und Robustheit gelegt. Dieser Artikel schildert ein mögliches Vorgehen bei der Planung und Umsetzung eines umfangreichen Performance- und Lasttest-Szenarios für die komplexe Swing-basierte Oberfläche eines Leittechniksystems. Ein Schwerpunkt liegt dabei auf der Darstellung der Entscheidungsfindung bei der Auswahl eines geeigneten GUI-Testwerkzeugs, das durch gezielte Erweiterungen zum Performance-/Lasttestwerkzeug auf Endbenutzer-Ebene „hochgerüstet" wird. Am konkreten Beispiel mit dem ausgewählten Testwerkzeug „QF-Test" wird aufgezeigt, wie man automatisierte Dialogabläufe mithilfe eines Testroboters für Antwortzeitmessungen verwenden kann, und wie dabei wichtige Randbindungen, wie möglichst belastungsfreies Messen, weitestgehend umgesetzt werden können.*

Abb. 1: Leitstand eines modernen Kraftwerkes mit SPPA-T3000

## Performance als eine Eigenschaft von Softwarequalität

Die ISO-Norm 9126 [ISO9126] definiert für die Softwareproduktqualität eine Reihe von messbaren Eigenschaften, die im Englischen gern auch wegen ihrer Endungen als sogenannte „ilities" bezeichnet werden (Functionality, Reliability, Usability, Efficiency, Maintainability, Portability). Für uns interessant ist in diesem Zusammenhang besonders das Thema Effizienz („Efficiency"). *Effizienz* beschreibt eine Menge von Attributen, die sich mit *Software-Performance* und den dafür verwendeten *Ressourcen* befassen. Diese Attribute enthalten:

- Antwortzeit
- Ausführungsgeschwindigkeit
- Durchsatz (Anzahl erledigter Anfragen/Aufgaben je Zeiteinheit)
- Ressourcenverbrauch (RAM-Verbrauch, Plattenplatz, I/O, Netzwerkverhalten, Bandbreite, Paging-Aktivitäten, Java-Garbage-Collection-Zeiten usw.)

Am meisten interessiert uns beim Thema Performance die *Endbenutzer-Sicht*, und das heißt in unserem Fall, im Wesentlichen die *Antwortzeit des Systems (Systemreaktionszeit) auf bestimmte Benutzereingaben.*

## Das „System Under Test" – eine komplexe Java-Swing-Applikation

Das zu testende System (System Under Test, SUT) ist Teil einer Applikation aus dem industriellen Umfeld der Kraftwerksautomatisierung: Siemens Power Plant Automation SPPA-T3000 [SPPA]. SPPA-T3000 ist eine Client-Server-Lösung, die auf der Client-Seite eine komplexe Java-Swing-Oberfläche – in Form von Java Web Start oder als Browser-Applet – anbietet.

Dass hierbei „komplex" nicht nur ein schmückendes Beiwort ist, verdeutlicht der Leitstand eines modernen Kraftwerks, das diese Software nutzt (s. Abb. 1). Die Vielfalt der GUIs gibt dem Leser genauso wie dem Tester einen guten Eindruck, welche Anforderungen an geeignete Werkzeugunterstützung für eine Performance-Messung in diesem Umfeld zu stellen sind. Wir beschränken uns dabei hier auf die Ermittlung der Eigenschaft „Antwortzeit" mit der Hypothese von maximal 20 gleichzeitig mit demselben Server arbeitenden „Operators" (d. h. 20 Client-PCs oder 20 „virtuelle Benutzer"). Die zu testende grafische Oberfläche besteht aus Standard-Java-Swing-Komponenten, die derzeit unter Java 6 laufen, aus eigen-entwickelten GUI-Klassen, basierend auf Swing, sowie auf einer Vielzahl von Klassen der kommerziellen ILOG-JViews-Bibliothek [JViews].

Wie schon andeutet, soll die Qualitätskontrolle, in unserem Fall die Performance-Messung, ergebnisorientiert und nachprüfbar sein. Ergebnisorientiert heißt, es müssen klare Anforderungen für das Endprodukt, also das auszuliefernde System, als qualitativ und quantitativ erfasste funktionale und nichtfunktionale Requirements schriftlich formuliert vorliegen (also mit konkreten Werten wie Ausführungszeit, CPU-Auslastung, RAM-Verbrauch usw.). Nachprüfbar heißt, die Tests müssen als Regressionstest jederzeit mit der damals und heute aktuellen Test- und Zielumgebung wieder durchgeführt werden können und zu konkreten Aussagen wie „Test bestanden" (grün) oder „Test nicht bestanden" (rot) führen.

Wie diese Anforderungen in diesem konkreten Umfeld zu erfüllen sind, zeigen die nachfolgende Beschreibung des verwendeten GUI-Testwerkzeugs und die Methodik, um an die interessierenden „Ereignisse" im SUT „heranzukommen".

## Auf der Suche nach dem „Tool-Gral"

Internet-Recherchen nach Testwerkzeugen für Java-Swing-Anwendungen liefern zwar eine Vielzahl von Treffern, doch sind die Ergebnisse auch brauchbar? Mit „brauchbar" meine ich: genau auf die gestellte Aufgabe der *Performance-Messung aus „Endbenutzer-Sicht"* zugeschnitten und auf (komplexe) *Java-Swing-Anwendungen* spezialisiert. Was man auf jeden Fall findet, sind „Allrounder"-Werkzeuge für den *funktionalen* Test. Damit ist gemeint: Sie decken Java Swing, Eclipse SWT und Web-GUIs ab, bieten aber keinen oder nur wenig Support für

das Thema „oberflächenbasierte Performance-Messung“. Die berühmte „Gretchenfrage“ lautet also: *Mit welchem GUI-Testwerkzeug lässt sich unsere Aufgabenstellung am effizientesten lösen?* Die Antwort ist zwar philosophisch einfach:

*„In Zweifelsfällen entscheide man sich für das Richtige.“*
Karl Kraus
(Österreichischer Schriftsteller, Herausgeber und Zeitkritiker)

Doch wie kommt man zum „richtigen“ Werkzeug? Am meisten hilft die Aufstellung eines eigenen, konsequent problemspezifischen, harten *Kriterienkatalogs* (Templates dazu findet man im Internet), der dann mit dem konkreten SUT mit Tool-Evaluierungslizenzen (oder mit Open Source) abgearbeitet werden muss. So gern und oft ich Open-Source-Frameworks und Werkzeuge einsetze: Bei diesem Projekt stießen sie eindeutig an ihre Grenzen. Das gilt sowohl für den funktionalen Test (z. B. Marathon, Abbot/Costello [Abbot]), aber umso mehr für den Performance-Test (vgl. [OSPTT]).

Wichtig ist auch das Thema „Umgang mit Änderungen im GUI-Layout“, d. h., man braucht insbesondere eine flexible, aber dennoch intelligente und zuverlässige *Komponentenerkennung*. Daneben kommt es auch darauf an, ob man lieber vorwiegend auf der Basis von Skriptsprachen testet, wie bei [Squish] (generiert Skriptcode in Python, JavaScript, Perl oder Tcl), konsequent schlüsselwortbasiert ohne Testroboter (Capture-Replay), wie bei [GUIDancer], oder mit einem Mix aus „(GUI-)Events in Baumstruktur“ plus Skriptsprachen als Ergänzung (Jython/Groovy, wie bei „QF-Test“ [QF]). Das sind nur drei Vertreter mit unterschiedlichen Ansätzen und Schwerpunkten; mehr findet sich – wie gesagt – durch geeignete Internetrecherche.

Wenn man oberflächenbasiert Last und Performance für Java-Swing-GUIs messen will, sind Werkzeuge, die die funktionalen Tests eines solchen SUTs beherrschen, schon mal kein schlechter Ausgangspunkt. Spielen neben Standard-Swing-Komponenten auch eigen-entwickelte oder Third-Party-GUI-Komponenten noch eine große Rolle, dann braucht Ihr Werkzeug unbedingt die Fähigkeit, entweder den „inneren Aufbau“ solcher sogenannter „Custom Controls“ erlernen zu können (durch „Extension Plug-in“-Mechanismen oder spezielle „Resolver-Interfaces“) oder das Scripting muss es erlauben, in „Eigenregie“ – statt ausschließlich mit den Werkzeugmöglichkeiten – mit solchen Komponenten weiterzuarbeiten. Haben Sie möglicherweise auch schon Ziele für ein *Testautomatisierungs-Framework* (vgl. [SWQL08]) formuliert, dann sind auch diese mit den Tool-Kandidaten Ihrer engeren Wahl abzugleichen.

Zu guter Letzt noch ein Hinweis zum Thema *Tool-Support*: Wenn man wirklich mal nicht mit Handbuch und Tutorien weiterkommt und knifflige technische Fragen hat, braucht es kompetentes Personal am anderen Ende der Hotline und möglichst kurze „Turn-Around-Zeiten“. Will heißen: Wenn ich hier in Deutschland oder einer vergleichbaren Zeitzone anrufe oder eine „Brauche-dringend-Hilfe-Mail“ hinschicke, nutzt mir eine Verbindung nur zum Verkaufsapparat oder eine Mannschaft im Ausland, die vielleicht nach einer Woche erst verstanden hat, was mein eigentliches Problem ist, recht wenig …

Am Ende dieser langen, aber eminent wichtigen Kette von Fragen und Antworten zum Auswahlprozess stand bei uns das Werkzeug *QF-Test* [QF] der deutschen Firma QFS. Da wir zu einem früheren Zeitpunkt für dasselbe SUT schon zahlreiche oberflächenbasierte, funktionale Testwerkzeuge evaluiert und einen „Sieger“ gefunden hatten, lag es nahe, dessen Eignung auch für Last- und Performance-Tests zu prüfen. Die Fragestellung ist also hier: *Bietet das Werkzeug Features für solche nicht-funktionalen Tests „schon von Haus aus“ an oder lassen sie sich wenigsten mit moderatem Aufwand „oben drauf“ bauen?* In der Beschreibung von QF-Test heißt es: „QF-Test ist ein Werkzeug zur Erstellung, Ausführung und Verwaltung von automatisierten System- und *Lasttests* für Java- und Web-Anwendungen mit grafischer Benutzeroberfläche (GUI).“ Was bedeutet das nun genau?

## GUI-basierte Last- und Performance-Tests: Eine Herausforderung für Tool und Tester

Neben funktionalen und Systemtests eignet sich das von uns ausgewählte Werkzeug, QF-Test, wie oben bereits angedeutet, auch zur Durchführung von Lasttests von geringer bis mittlerer Größe, Stresstests oder Performance-Tests. Dabei wird die Performance einer Server-Anwendung getestet, indem eine Anzahl von Clients gleichzeitig ausgeführt wird. Wichtig ist dabei die generelle Möglichkeit, mehrere SUT-Clients gleichzeitig auf einem Desktop (in parallelen Threads) zu betreiben, ohne dass diese sich gegenseitig behindern (vgl. [QF]). Davon machen wir aktuell zwar nicht Gebrauch, da unser SUT für diesen Betriebszustand nicht spezifiziert ist, aber rein technisch bestünde auch diese Möglichkeit.

Mit GUI-basierten Last/Performance-Tests werden sogenannte „End-to-End“-Zeiten ermittelt, d. h. die Zeit von einer Aktion des Anwenders bis zur Anzeige des Ergebnisses an der Oberfläche. Protokollbasierte Tests messen dagegen nur die Zeit für die Anfrage an den Server. Beides kann sinnvoll sein, je nach Situation (vgl. [QF]). Übrigens: Auch wenn man QF-Test nicht „multi-threaded“, sondern einmal pro Client-PC betreibt, sind so viele Runtime-Lizenzen notwendig, wie man virtuelle Benutzer simulieren möchte. Solch ein Vorgehen bedeutet damit mehr Hardware (oder den Einsatz von „virtuellen PCs“) und mehr Budget für die Lizenzgebühren. Doch einerseits steigen die Preise für neue Runtime-Lizenzen nicht linear an (Rabatt), andererseits kann man bei hohem Bedarf auch von der Option eines zeitbegrenzten Lizenzleasings Gebrauch machen.

Hat man nun das „richtige“ Werkzeug gefunden, gilt dennoch zu bedenken: „Der Einsatz eines Lasttestwerkzeugs ist nicht ausreichend für die erfolgreiche Durchführung von Performance-Tests“ [Franke10]. Es braucht auch einen spezialisierten Testprozess, der in [Franke10] recht gut beschrieben ist. Wir hatten für unser SUT, die SPPA-T3000-Applikation, schon zahlreiche „Reaktionszeit-Anforderungen“ bei unterschiedlicher Grundlast des Systems in Requirements-Spezifikationen vorliegen. Daher lag es nahe, als eine Art „Proof of Concept“, sich exemplarisch anspruchsvolle Aufgabenstellungen aus diesen Dokumenten herauszupicken und mit dem ausgewählten Werkzeug erste Tests aufzubauen und durchzuführen. Wie gesagt, ging es zunächst nur einmal auf das Erfassen und Protokollieren von Reaktionszeiten (inklusive der Aussage „Test bestanden/nicht bestanden“). Der „Merge“ mit den zugehörigen Zahlen aus dem Monitoring des (Server-) Ressourcenverbrauchs ist dann ein Thema für die Nutzung des Testautomatisierungs-Frameworks und derzeit noch nicht implementiert.

Wir zeichnen also Test-Szenarien aus „Operator-Sicht“ (= Endbenutzer-Sicht) auf, vergeben geeignete „Think-Times“, um die Ausführung realitätsnah zu gestalten, und fügen dann an den interessierenden Stellen sogenannte *Transaktionen* in die Skripte ein, die die Basis für die Zeitmessungen bilden. Lassen Sie uns dazu zunächst ein Beispiel einer komplexen SPPA-T3000-GUI-Komponente anschauen. Abbildung 2 zeigt ein Display mit 64 Piktogrammen, die aus 12 unterschiedlichen Objekttypen (wie Tankfüllanzeige, Motor usw. bestehen). Das

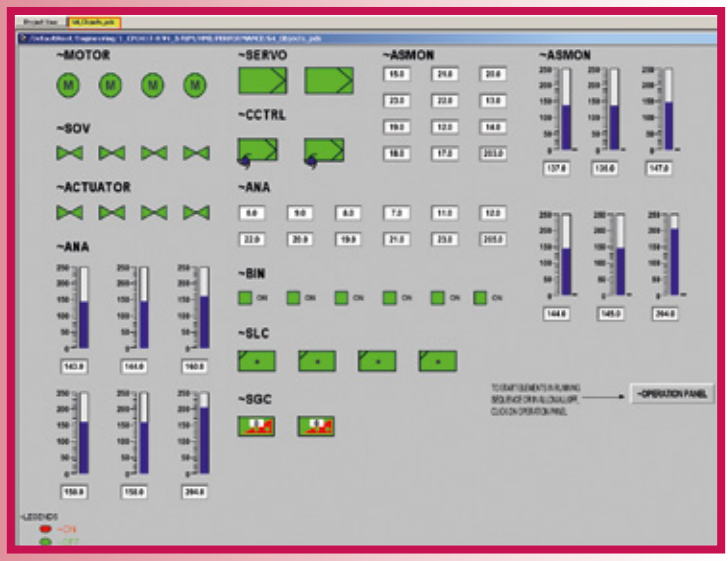


Abb. 2: Testszenario „Performance-Tests für 64 Objekte - Aufschaltzeit“

Testszenario, das hiermit gemessen werden soll, ist die maximale Zeit, bis die Ausgänge aller 64 Komponenten einen vom Server übermittelten Wert (Float oder binär) anzeigen („Aufschaltzeit“). Wie geht man dazu vor?

Mit QF-Test lassen sich prinzipiell Tests erstellen, ohne auch nur eine einzige Zeile Code zu schreiben. Die Kontrollstrukturen, die festlegen, was in welcher Reihenfolge in welchen zeitlichen Abständen zu tun ist, und die notwendigen Testdaten werden bei QF-Test in der Testsuite (einer hierarchischen Baumstruktur) vereint. Abbildung 3 zeigt die Performance-Testumsetzung der Aufgabenstellung aus Abbildung 2 mittels QF-Test.

Der Wurzelknoten in Abbildung 3 repräsentiert dabei die komplette Testsuite und kann mehrere Ebenen von Knoten enthalten. Auf diese Weise lassen sich Test-Sets, Test-Cases, Test-Steps, Sequenzen, Prozeduren, aber auch Skripte in die Suite einbauen, um so den gesamten Ablauf zu strukturieren. Sogar Testablaufsteuerung mit Prozeduraufrufen, LOOPs, IF-THEN-ELSE-, TRY-CATCH-Strukturen usw. ist möglich. Bei Skripten wurde Jython von Anfang an schon angeboten (das merkt man dem Werkzeug an einigen Stellen auch immer noch an), Groovy erst seit der Version 3 (derzeit ist in Version 3.4.0 Groovy 1.7.6 integriert, was dem erfahrenen Java-Entwickler möglicherweise eher liegen wird).

Wichtig ist der Unterschied zwischen Server-Skripten, die von einem in QF-Test eingebetteten Interpreter (Jython oder Groovy) ausgeführt werden und das SUT-GUI nicht direkt belasten, und SUT-Skripten, die clientseitig im SUT ausgeführt werden. Für den Performance-Test kommen bei uns beide Skriptarten zum Einsatz, schwerpunktmäßig jedoch die SUT-Skripte (in Groovy), da nur mit ihnen direkt auf die GUI-Komponenten des SUT zugegriffen werden kann und diese damit auch veränderbar sind. Die wesentlichen Schritte sind:

- *„Performance-Testaufrüstung“ des Werkzeugs*: Beim Start des Tests wird analog zum bekannten JUnit-Setup zunächst eine Prozedur (hier: ein Groovy-Server-Skript) aufgerufen, die einen Listener instanziiert, mit dem jeder durchlaufene Testbaumknoten analysiert wird. Da die Knoten auch beliebige Kommentare enthalten dürfen, wird für den Performance-Test nach Kommentaren der Art `@transaction START <tid>` bzw. `@transaction END <tid> <maximale Reaktionszeit in ms>` gescannt. Durch die von QF-Test an die Prozedur übergebenen Zeitstempel können dann die Laufzeiten für solche „Performance-Transaktionen“ bestimmt werden. Das Verfahren ist sehr flexibel, weil beliebige „GUI-Events“, d. h. Baumknoten, auf diese Weise geklammert werden können.
- *Vorbereitung der Messung:* Um möglichst wenig Analysezeit während der eigentlichen Performance-Messung zu „verbraten“, werden die auf dem ILOG-Display vorhandenen Piktogramme nach Anzahl und Typ analysiert und abgespeichert. Auf ILOG-Komponenten lässt sich sowohl mit einem sogenannten *Custom-Checker* (siehe nächster Punkt) zugreifen als auch in einem „normalen“ SUT-Skript, z. B. mittels des von QF-Test übergebenen Runtime-Kontexts und der gewünschten Komponenten-ID: `def component = rc.getComponent(qf-test-component-id)`. QF-Test ist aber auch in der Lage, mit GUI-Unterelementen zu arbeiten, die selbst keine GUI-Elemente sind, z. B. die Zellen einer Tabelle oder Knoten in einem Baum. Nicht-Standard-Unterelemente wie Zeichnungen auf einem Canvas oder ILOG-Komponentenstrukturen könnten auch mithilfe des ItemResolver-Mechanismus (ein Interface von QF-Test) implementiert werden.
- *Zeitmessung in einer LOOP starten:* Zustandsgesteuerte Messung mit einer speziellen QF-Test-Erweiterung, dem *Custom Checker*.

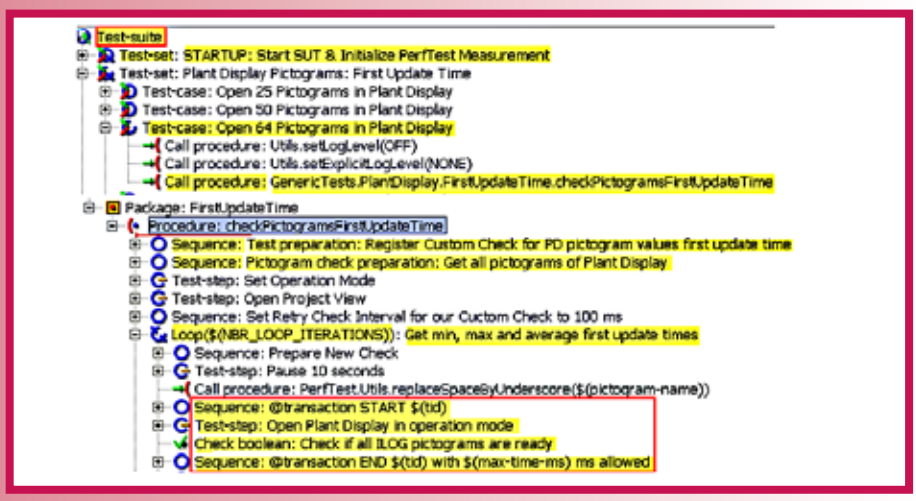


Abb. 3: QF-Test-Baumstruktur für „Performance-Tests für 64 Objekte – Aufschaltzeit“

- *Custom-Check durchführen:* Der *Custom-Checker* ist in unserem Fall ein Groovy-SUT(!)-Skript, das einerseits ein spezielles QF-Test-Interface implementiert, andererseits auch als eine Art „Zustandsmaschine“ arbeitet, um z. B. *vor der eigentlichen Messung* zunächst einmal die im Display enthaltenen Komponenten und ihren Typ zu bestimmen. Mit dem Custom-Checker wird das von QF-Test selbst als Knotenmenge angebotene Standard-Check-Set „GUI-problemspezifisch“ erweitert. Das meinte ich z. B. zu Beginn dieses Artikels, als ich von durchaus „anspruchsvollen“ Testaufgaben sprach. Der Checker wird in einer eigenen, QF-Test-internen Schleife (Aufrufzeit parametrisierbar) aufgerufen und gibt den Boole-

schen Wert „true“ zurück, sobald der Test erfüllt ist.
- *Zeitmessung stoppen*.
- *Gemessene Zeiten protokollieren*, d.h. sowohl in eine CSV-Datei als auch ins QF-Test-Runlog die Ergebnisse (inkl. „passed/failed“) schreiben.

## Der Wunsch nach belastungsfreiem Messen

Bei Performance-Tests ist es eminent wichtig, durch die eigene Messarbeit das zu testende SUT-GUI so wenig wie möglich zu belasten (Wunschziel wären sogenannte „Non-intrusive Tests“). Man muss sich nämlich immer bewusst sein, dass sowohl das SUT-GUI als auch die QF-Test-SUT-Skripte, wie der o.g. Custom-Checker, auf dem AWT Event Dispatch Thread (EDT) ablaufen. Swing ist nun mal „single-threaded“. Daher liegt es im QF-Test-Verständnis und im Können des Implementierers, einerseits die Interaktionen (und damit das „Ping-Pong-Spiel“) zwischen QF-Test-Knotenabarbeitung und SUT-Skript minimal zu halten und andererseits im SUT-Skript selbst die auf dem EDT ablaufenden Anweisungen so schnell und so knapp wie möglich zu gestalten. Nur so können die gemessenen Zeiten „realitätsnah“ sein – und von allen Stakeholdern akzeptiert werden.

## „Allein ist nicht genug“

Frei nach dem gleichnamigen Gedicht von Peter Rühmkorf (*1929, †2008), reicht auch beim professionellen Performance-/Lasttest ein gutes GUI-Testwerkzeug alleine nicht aus. Testautomatisierungs-Frameworks [SWQL08] als Ergänzung unterstützten optimalerweise den gesamten Testzyklus, von der Erstellung der Testfälle bzw. der Einbindung vorhandener Suiten über die automatisierte Ausführung auf verschiedenen PCs bis hin zur Analyse der Testergebnisse mit Reporterstellung. Damit können auch notwendige Batch-Prozesse für Vor- oder Nacharbeiten zu den einzelnen Tests eingebunden und angestoßen werden. Auch Ressourcenmessungen, die auf den Clients/Servern anfallen, lassen sich optimalerweise mit den gemessenen Ausführungszeiten auf einer gemeinsamen Zeitachse darstellen. Gerade bei Java potenziell anfallende längere Garbage-Collection-Zeiten könnten so auch z. B. möglicherweise Aufschluss über ungewöhnliche Reaktionszeiten geben.

Tool-Kandidaten, die mit QF-Test schon zusammenarbeiten, sind beispielsweise T-REGS von der Firma Novaobjects, TAPE von C1 SetCon, die Testautomatisierungslösung der Schweizer Firma BISON und Scapa TPP (Test and Performance Platform) der englischen Firma Scapa Technologies. Auch mit dem Open Source „Software Testing Automation Framework (STAF)“ lässt sich schon eine Automatisierungs-Infrastruktur bauen, die allerdings erst mal nur ein „Grundgerüst“ bietet. Hier muss jeder Leser als Tester seinen individuellen Fragebogen abarbeiten und die notwendige Kosten/Nutzen-Bilanz ziehen. Unsere Wahl viel bei diesem Prozess auf T-REGS.

## Fazit

Last- und Performance-Test von *komplexen* Swing-Anwendungen – wie der vorgestellten Siemens SPPA-T3000-Software – sind auch ein *komplexes* Unterfangen. Egal mit welchem GUI-Testwerkzeug-Support man das Thema angeht und betreibt. Dieser Artikel sollte am Beispiel einer Lösung mit dem kommerziellen Werkzeug QF-Test demonstrieren, wie solche Tests dennoch nicht zur (ungeliebten) Last werden müssen. Im Gegenteil: Mit einem flexiblen, erweiterbaren GUI-Testwerkzeug, intelligenter Skript-Programmierung sowie einer guten Testautomatisierungslösung ergeben sich für den ambitionierten Tester ganz neue Herausforderungen, die mit Sicherheit den „Test-Job“ nicht so schnell zur Routine werden lassen.

## Literatur und Links

**[Abbot]** Marathon und Abbot/Costello, Open Source Tools für den funktionalen Test von Java-Swing-Anwendungen, Untersuchungsbericht vom 29.10.2010, `http://home.edvsz.fh-osnabrueck.de/skleuker/CSI/Werkzeuge/abbot.html`

**[Franke10]** R. Franke, Last- und Performancetests, Blickpunkte, 5/2010, Pentasys AG, `http://www.pentasys.de/de-user-Blickpunkte-index.html`

**[GUIDancer]** GUI-Testwerkzeug, `http://www.bredex.de/de/guidancer/first.html`

**[ISO9126]** ISO-Norm 9126 zur Softwarequalität, `http://de.wikipedia.org/wiki/ISO/IEC_9126`

**[JViews]** ILOG JViews: Graphical software components for Jav developers, `http://www-01.ibm.com/software/integration/visualization/java/`

**[OSPTT]** Open Source Performance Testing Tools: Madhu Ramachandran, Performance Testing: Open Source Approach, 12.12.2008, `http://www.qanc.co.kr/4research_0402_download.htm?data_no=215&name=Open%20Source%20Tools%20for%20Performance%20Testing.pdf` und Open source performance test tools (50 found), `http://www.opensourcetesting.org/performance.php`

**[QF]** QF-Test, GUI-Testwerkzeug von Quality First Software, `http://www.qfs.de/de/qftest/index.html`

**[SPPA]** Siemens Power Plant Automation SPPA-T3000 (Kraftwerksautomatisierungssoftware), `http://www.energy.siemens.com/hq/de/automatisierung/stromerzeugung/sppa-t3000.htm`

**[Squish]** GUI-Testwerkzeug von Frologic, `http://www.froglogic.com/products/features.php`

**[SWQL08]** J. Hochrainer, K. Aigner, J. Bergsmann, Testmanagement- & Testautomatisierungswerkzeuge, Studie des Software Quality Lab, 2008, `http://www.software-quality-lab.at/swql/uploads/media/Tool-Studie-2008_V1.00-Einleitung.pdf`

**Klaus P. Berg** arbeitet als Senior Engineer bei der Siemens AG München, Corporate Research and Technology, Abteilung Software, Security & Systems. Seine Schwerpunkte sind Codequalität und der implementierungsnahe System- und Performance-Test von Java-Anwendungen sowie die Tool-Entwicklung in diesem Bereich.
E-Mail: Klaus-Peter.Berg@siemens.com