Automated Testing of Non-functional Requirements

Kristoffer Dyrkorn
BEKK Consulting, Oslo, Norway
kristoffer.dyrkorn@bekk.no

Frank Wathne
BEKK Consulting, Oslo, Norway
frank.wathne@bekk.no

Abstract

We present an open-source toolkit that enables automated testing of non-functional requirements. Our toolkit offers an alternative to current approaches (e.g. testing suites) by being lightweight, flexible, free and by reusing libraries and executables found on common operating systems and platforms. The toolkit provides developers and project managers with reports about the system under development. We would like to advocate the importance of non-functional testing, especially in the context of agile projects, and we have experienced significant risk reductions in our projects when using the toolkit described here.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging - Testing tools (e.g., data generators, coverage testing); K.6.3 [*Software Engineering*]: Software Management - Software development

General Terms Measurement, Performance, Reliability, Verification

Keywords Automated testing, non-functional requirements, metrics, open source

1. Introduction

Most automated testing tools are limited to the aspect of functional testing. We claim that the testing of non-functional requirements (performance, scalability, robustness, recoverability, cacheability, etc) in server applications should be considered equally relevant and subject to automation. In our opinion, running such tests are highly valuable during the development process. In our session we will demonstrate an implementation that allows for automated testing of non-functional requirements.

OOPSLA'08, October 19–23, 2008, Nashville, Tennessee, USA.
ACM 978-1-60558-220-7/08/10.

2. Background

Agile methods are well-known practices within software development. Along with automated tests, these practices have improved software development processes and the quality and predictability of software releases. Agile practices has increased the focus on writing testable code, leading to widespread adoption of Inversion-of-Control containers and the Dependency Injection pattern.

However, most automated tests do not cover the external – i.e. non-functional – behaviour of the code being developed. As an example, performance and scalability issues will not be detected by the execution of unit tests. In addition, non-deterministic or transient phenomena (e.g. the effects of caching) might not be revealed at the unit test level. Thus the "green lights" commonly in use in automated test tools are only valid in a local context and might give a false impression of what the net effects of code changes are. Automated testing of the non-functional system properties will narrow this risk-inducing gap.

3. Automation at the System Level

Our method of doing system-wide automated testing follows the principle of black-box testing. We run client programs against a server in order to exercise the system as a whole in a realistic manner. Example communication protocols and interfaces include: HTTP (for standard web traffic and REST-based integration), HTTP + SOAP/WSDL (for web services), JDBC/ODBC (for databases) and JMS/MSMQ/MQ (for queue-based systems).

The client programs send pre-generated or self-generated requests to the system, and we collect response messages, response metadata (e.g. headers, response times), application logs, server logs, and statistics from the operating system (CPU usage, memory usage). We also run performance and scalability tests and simulate system interruptions by forcing restarts of server processes under load.

The gathered data is formatted into a project-specific report containing the most relevant system outputs and metrics, e.g. response times, processing rates at increasing load, application logs, validation errors of system responses, and memory and CPU usage. The generated report is then archived for later retrieval.

4. Generating Requests

Valid testing depends on realistic request patterns, the ideal – and utopic – approach being to expose the system to its *future* traffic. Other approaches will imply a sampling of the system behaviour, thus being subject to inaccuracies. One will have to ensure that test data expose the significant system behaviours to a sufficient degree. Some ways to generate requests are:

- Using historical requests from the production system
- Using self-discovering request generators
- Generating random – but valid – requests
- Manually recording requests

The approach to choose will depend on the complexity of requests and responses. We have experienced great benefits from using self-discovering generators – for example, by using a web crawler to traverse a web site. Self-discovery makes the tests less susceptible to changes. By the proper use of WSDL and generated data, self-discovery is also possible for web services. Other protocols might not support self-discovery .

Using historical data from a production system is beneficial if a production system already exists. Test and production environments must be compatible, and security policies, costs and/or operational routines must allow this. As a principle, we strongly believe that the option to easily replay transactions from a production system on a test system should be available in any software project. This feedback loop will simplify the recreation of production errors and give highly relevant test results.

An alternative is to use a fixed set of test data and to observe the relative changes in metrics over time. The need for consistent sampling (using test data over time, to make comparisons possible) will have to be balanced against the need to adapt to system or API changes (updating the test data, but losing the baseline). As usual, accuracy must be traded with effort.

Our experience is, however, that the testing approach described here gives valuable output already at a low initial cost and that accuracy can be improved incrementally. System level testability should be a design goal in a manner similar to what testable code now is – due to agile methods and automated functional testing.

5. Implementation, Usage and Experiences

Our implementation is based on scripts that execute various command-line programs (protocol clients), and the console output forms the basis of the test report. Ant has been chosen as the top-level script container due to easy integration with automated test tools such as CruiseControl. Unix text utilities were chosen for text processing due to simplicity, flexibility and cross-platform availability. We have also written some clients as basic command-line programs. The test report consists of a set of HTML files and graphs.

Our practice has been to run the automated tests every night and to review the reports before the standup meeting the following day. Thus the project team has a continuous overview of the system behaviour, and task priorities can be changed immediately if major issues are identified. Being able to quickly reverse changes or start work on alternative solutions reduces risk effectively.

Overall, we have focused on making important information as easily available as possible. We believe there is a tendency for teams to forget or avoid repetitive and cumbersome tasks, e.g. reading application logs from multiple servers. Gathering a description of the system behaviour in one place is helpful and also simplifies matching error messages in logs with other reported system abnormalities. Also, having an archive of reports makes it possible to track the system behaviour over time.

In one of our projects it became essential to do validation of the system responses. Here our task was to build an information web site with strict accessibility requirements, and we adopted our toolkit to generate reports over HTML validation errors, broken links and character encoding issues.

In conclusion, we believe that by doing automated non-functional testing we have been able to rapidly detect issues that otherwise would be hidden or discovered later. This has lead to better risk management and predictability in our projects, and we have experienced that careful composition and orchestration of basic software components can yield a flexible and powerful tool.

6. Future Improvements

Our implementation does not provide a complete solution to all aspects of non-functional testing. Instead, we think it is better to create a simple and open toolkit that in turn can be adapted to a variety of projects and architectures. Some areas to further explore are: Testing of security (request mutation, cross-site scripting), bandwidth conservation (web caching, response compression) and HTML rendering (visual verification of layout).

Acknowledgments

The authors would like to thank The Research Council of Norway for donating the toolkit to open source.

References

[1] http://en.wikipedia.org/wiki/Test_automation

[2] http://en.wikipedia.org/wiki/Agile_software_development

[3] http://en.wikipedia.org/wiki/Inversion_of_control

[4] http://en.wikipedia.org/wiki/Dependency_injection

[5] http://cruisecontrol.sourceforge.net/