CHANGE-CENTRIC QUALITY ASSURANCE

Changes as first-class citizens during software development

Quality software has long been synonymous with software "without bugs". Today, however, quality software has come to mean "easy to adapt" because of the constant pressure to change. Software teams seek for a delicate balance between two opposing forces: striving for reliability and striving for agility. In the former, teams optimize for perfection; in the latter they optimize for ease of change.

The **ANSYMO** (University of Antwerp) and **SOFT** (University of Brussels) research groups are investigating ways to reduce this tension between reliability and agility. Together, we seek to make changes the primary unit of analysis during quality assurance and as such we expect to speed up the release process without sacrificing the safety net of quality assurance.

http://soft.vub.ac.be/chaq/

Universiteit Antwerpen

Vrije Universiteit Brussel

iwt

The following are examples of the problems that we address:

• *Monitoring the test process.* Determine the impact of changes on both the test and production code, to persuade team members to increase test activities. Demonstrate that the test process itself meets quality guidelines (e.g., every bug fix is covered by a regression test).

• *Deciding what to re-test.* Instead of running all tests for a given release, run only those tests that are potentially affected by a given change. This allows for instant feedback on the changes that cause tests to fail, saving valuable time in identifying the precise location of a bug.

• *Monitoring the bug database.* Verify whether anomalies occur in the bug database (e.g., wrong severity, assigned to wrong product or component). Assure that all severe bugs have been fixed before a release.

• *Deciding bug assignment.* Once bugs have been reported, determine who is the best person in the team to handle the request. Use historical information to reliably estimate the time it will take to fix the bug.

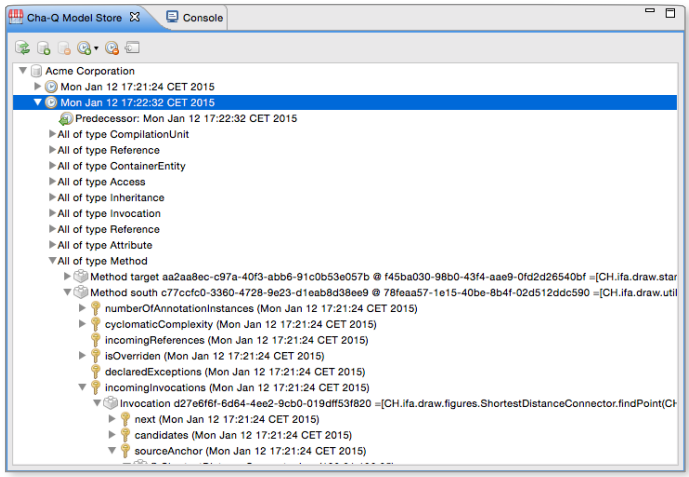
• *Monitoring code changes.* Monitor changes as they are made in the editor or as they are committed to the version repository. Use documented traceability links and past co-change information to recommend related code that should be changed accordingly (e.g., XML configuration files).

• *Automating code changes.* Release a new API version with patches that automatically update all existing client code, reducing the number of API versions in the field. Replay code changes that were successful for a given branch on a variant branch, reducing manual branch synchronization.

Research on Analyzing Changes

Representing software and its changes

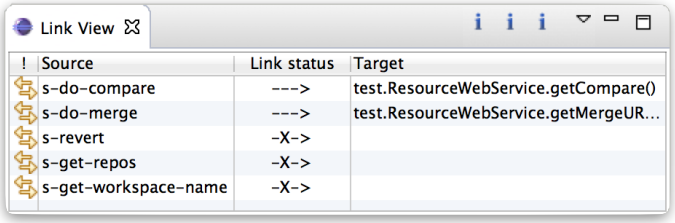
The quality assurance tools investigated by Cha-Q share a common representation of various software entities (e.g., source code, tests, bug records). This *change-centric representation* is the first to offer information about a) their state in a particular snapshot of the software, b) the entire history of their past states, and c) the changes made in between any two successive states. Individual changes to an entity can be analyzed, repeated and reverted —rendering them first-class. We offer a *change distiller* to import existing Java projects that have been versioned in a Git repository, and a complementary *change logger* to continuously update the resulting representation as developers make changes in the IDE. The screenshot depicts our Eclipse plugin for navigating snapshots in a change-centric representation.



The Implementation of the Cha-Q Meta-Model: A Comprehensive, Change-Centric Software Representation
Coen De Roover, Christophe Scholliers, Viviane Jonckers, Javier Pérez, Alessandro Murgia, Serge Demeyer
Electronic Communication of the European Association of Software Science and Technology, Volume 65 (2014)

Towards monitoring changes and recommending related changes to be made

Non-trivial software systems are composed of a myriad of interlinked artefacts (classes, XML files, requirements documents). Changes to a single artefact can have an unanticipated impact on the rest of the system. Our *change-centric software representation* and its accompanying *change logger* enable tool support for maintaining these links, by monitoring changes to their source and destination. The screenshot shown here depicts an Eclipse plugin that warns about changes that would invalidate references from XML configuration files to web service implementations and vice versa. Its strength lies in that it can be configured to monitor other links that are specific to one company or to a particular domain (e.g., links from functional requirements to tests in safety-critical domains).

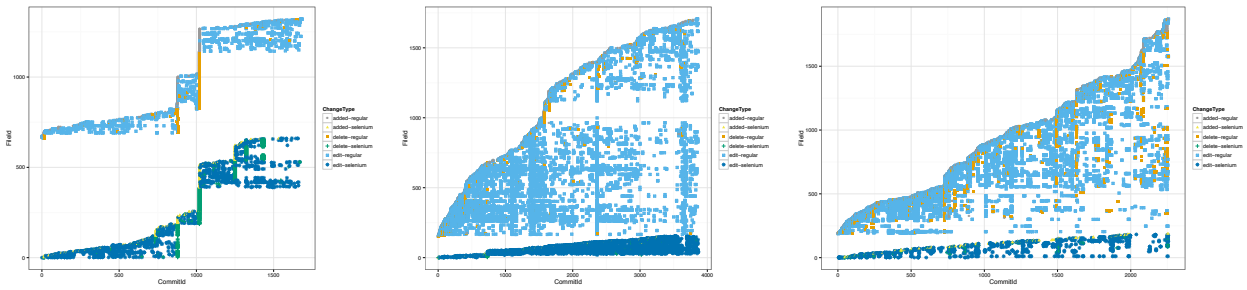


However, configuring our tool still requires explicit knowledge about which links need to be maintained. Often, this knowledge remains implicit due to missing or outdated documentation. Techniques have been proposed to reconstruct this knowledge from a snapshot of the system, but also from the system's commit history. In a study on two large open source projects, we found that the best results stem from the latter —provided that the commit history is sufficiently large, and that individual commits carry meaningful messages.

Explaining why Methods Change Together
Angela Lozano, Carlos Noguera, Viviane Jonckers
Proceedings of the 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM14)

Example change analysis: maintenance of Selenium tests for web applications

Our change-centric software representation enables monitoring existing quality assurance practices and processes. We investigated how developers maintain web applications and their Selenium-based functional tests. Selenium is a popular solution for automating functional UI tests through scripts that interact with a web browser and assert properties about the page it is rendering. However, small changes in the web application can easily break existing test scripts. We confirmed this by studying the development history of 6 large open source projects.



Depicted above are visualizations of the Git repositories of the XWiki, OpenLMIS and Atlas projects. The X-axis corresponds to individual commits from a repository. The Y-axis depicts the files that are changed in each commit. Commits to test scripts reside at the bottom of each visualization. The visualizations show that developers do maintain a relatively small number of test scripts, but not necessarily synchronously to the application under test. In fact, we found that it takes on average about 11.23 non-test commits before a test script is changed. We also found that test scripts survive at most three commits on average before being deleted or changed beyond recognition. This indicates that they are changed drastically, possibly to keep up with user interface changes.

Turning our attention to the actual changes that test scripts undergo, we used our *change distiller* to compute all changes between successive versions of each test script. We then categorized these changes according to the parts of a test script that they affect: expressions used to locate DOM elements on a web page such as buttons (*Locator*), statements used to simulate interactions with a DOM element such as clicking or entering text (*Command*), annotations used for demarcating individual tests (*Demarcator*), and assertions about the properties of a DOM element (*Assert*). The above table provides insights into which parts of a test script are most prone to changes: *Locators* and *Asserts*. A closer inspection reveals that these often contain "magic constants" such as the identifier of a DOM element or the expected value of one of its properties. Our recommendations are therefore clear: magic constants should not only be avoided in the application under test, but also in its test scripts.

Project	Total	Locator	Command	Demarcator	Asserts
Atlas	8068	90	3	104	3282
XWiki	68665	115	154	24	1490
Tama	31821	95	89	43	36
Zanata	12959	497	119	0	1
EEG/ERP	248	3	0	0	6
OpenLMIS	69792	2550	401	8	3454

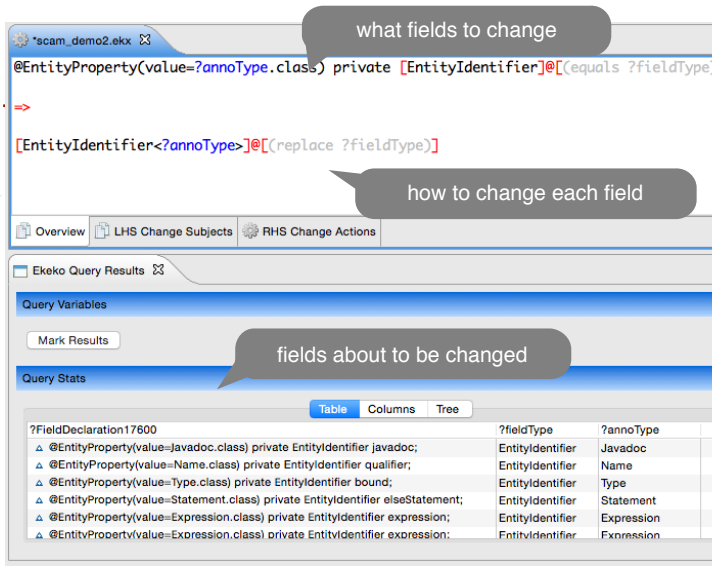
Prevalence and Maintenance of Automated Functional Tests for Web Applications
Laurent Christophe, Reinout Stevens, Coen De Roover and Wolfgang De Meuter
Proceedings of the 30th International Conference on Software Maintenance and Evolution (ICSM'14)

Research on Automating Changes

Advanced search-and-replace

powered by Cha-Q

Developers often perform repetitive changes to source code. For instance, to repair duplicate occurrences of a bug or to update all clients of a library to a newer version. Manually performing such changes is laborious and error-prone. Structural search-and-replace, as seen in IntelliJ, is the state of the art in tool support. It lets developers automate changes using search and replacement templates of code. Unfortunately, code that deviates the littlest from the search templates will not be found and hence not be replaced. The Cha-Q project is taking this idea to the next level: a powerful, but user-friendly program transformation tool that is decidedly template-driven.

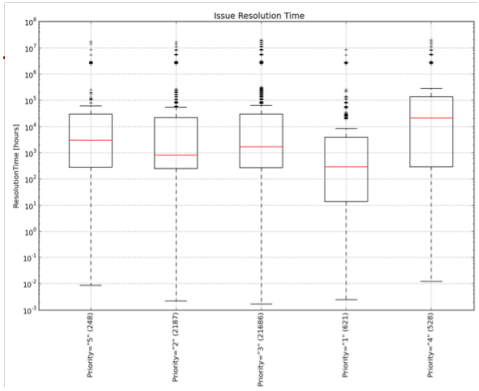



```
//Before changes:
@EntityProperty(value = SimpleName.class)
private EntityIdentifier label;
//After changes:
@EntityProperty(value = SimpleName.class)
private EntityIdentifier<SimpleName> label;
```

Shown on the right are repetitive changes that stem from our own commit history. All fields of type *EntityIdentifier* carrying an *@EntityProperty* annotation had to receive their annotation's value as a type parameter. The screenshot depicts the search and replacement templates that automate these changes for 266 different fields dispersed throughout 97 files. The replacement template, after the => arrow, is instantiated for each match for the search template before the arrow. Next to wildcards ... and placeholder variables ?annotype and ?fieldtype (substituting for field names, types and annotation values), directives such as equals and replace within each template further control what code has to be changed and how. The prototype already lends itself to API evolution scenarios. For instance, a scenario in which three deprecated method calls need to be replaced by a single call to a new method throughout all client code —while accounting for changes in exception handling and intermediate return values.

The Ekeko/X Program Transformation Tool
Coen De Roover and Katsuro Inoue
Proceedings of 14th IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM14)

Case Study: Predicting Bug Fixing Time



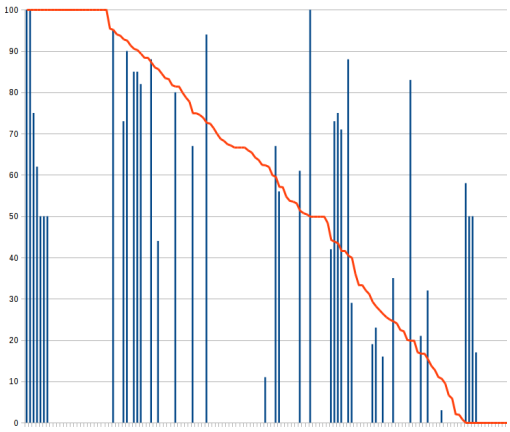
Software teams keep track of issues with systems such as Jira or Bugzilla. However, such systems may play a more pro-active role than merely managing the workflow within the team. In particular, one might estimate the time to resolve an issue based on the resolution time of similar issues in the past. Good estimation on the time to fix is crucial for improving customer satisfaction and project planning.

In the Cha-Q project we have developed a prototype tool for estimating the issue resolution time. Using the historical data reported in the issue tracking system (e.g. component, priority), we estimate the likelihood that an issue will be fixed within a certain amount of time. Moreover, we analyze how these estimations vary depending on the information inside the reported issue. The figure to the right shows a box-plot with the estimates depending on the priority. Priority 1 issues (most urgent) are indeed handled faster than others. This may come in handy for demonstrating that service level agreements are met.

powered by Cha-Q

The data shown here is based on a sample of the bug database of one of the Cha-Q partners. In the coming months we will replicate this experiment on other bug databases to assess the quality of the estimates during SCRUM sprints.

Case Study: Assessing Test Quality



The red line shows the mutation coverage in % for every class sorted from most to least. The vertical blue bars show the branch coverage.

Branch coverage vs. mutation coverage

For adequate testing, software teams need tests which maximize the likelihood of exposing a defect. Traditionally the adequacy of a test suite is assessed using test coverage, revealing which statements in the code base are poorly tested. Monitoring the test coverage is a recommended practice, but only provides an initial approximation. Additional measures are necessary to ensure that the test suite is effective in exposing defects.

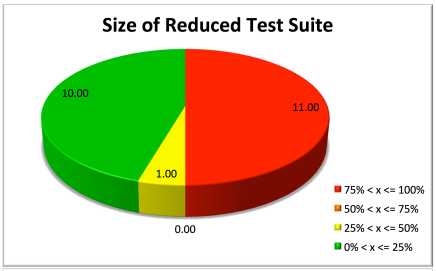
Mutation testing is the next logical step. A mutation test deliberately injects one defect into the base code, creating a so-called mutant. When running the test suite, at least one test should fail, in which case the test suite is said to kill the mutant. Doing this for a series of mutants, the ratio between the number of killed mutants versus the total number of mutants injected is a measure for the adequacy of the test suite.

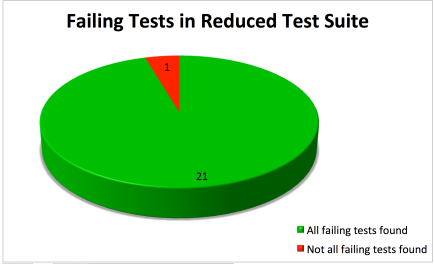
A student internship within Agfa HealthCare NV investigated the effectiveness of mutation testing for unit testing. A case study on a critical component (38K lines of Java code) confirmed that its unit test suite is quite strong. In particular the mutation tests confirmed that the black box test killed all mutants, something which could not be inferred from the branch coverage. Nevertheless (as can be seen in the figure) some classes could benefit from additional unit tests.

Mutation Analysis: An Industrial Experience Report.
Ali Parsai. Promoter: Prof. Serge Demeyer.
Masters thesis; January 2015. Computer Science, University of Antwerp.

Case Study: Deciding What to Retest

Unit testing is an established practice within agile software development. In many projects, roughly half of the entire code base consists of unit tests. Unfortunately, running all the tests easily takes several hours, hence developers are less inclined to run the test suite after each and every change. Thus, software is vulnerable for extended periods of time as the production code evolves, but the test code does not (immediately) follow.





Test-selection addresses this issue, by deducing the subset of (unit) tests that need to be re-executed given a series of changes to the production code. Upon commit, the test selection tool needs only a few minutes to assess whether the changes are safe; the complete test suite is executed during the nightly build.

An experiment with a code base owned by "Agentschap Wegen en Verkeer" demonstrated that test-selection might work in practice. We analyzed the complete history of a small project (56K lines of Java code spanning 14 months of development) retroactively running the complete test-suite for every commit in the version control system. We discovered a few failing test-runs and showed that the test-selection tool would have identified the culprits.

Change-based test selection in the presence of developer tests.
Quinten Soetens, Serge Demeyer, and Andy Zaidman.
Proceedings of 13th European Conference on Software Maintenance and Reengineering (CSMR13)

We need you !

We are halfway into the project; the basic tool infrastructure is in place and has been validated on open-source projects. In the next two years (2015 — 2016) we intend to test the tools under realistic circumstances. That's why we need your help.

We **seek** software teams interested in state-of-the-art tooling. In particular, those teams that

- Adopt agile practices
  (continuous integration, continuous delivery).
- Employ a version control system (SVN, GitHub).
- Track the issues (Bugzilla, JIRA).
- Monitor software quality
  (unit tests, static analysis, security vulnerabilities).
- Release often (at least internally).

We **offer** a seat in our industrial steering board.

- Meetings twice a year (Antwerp or Brussels);
  discuss with likeminded people.
- Roadmap for future software engineering tools.
- Opportunities for student internships
  (e.g., see "Case Study: Assessing Test Quality" on p.5).
- Participation in Experiments
  (e.g., see "Case Study: Deciding What to Retest" on p.5).
- Possibilities for follow-up research projects.

How to join ?

Contact Prof. Serge Demeyer (serge.demeyer@uantwerpen.be – 03/265.39.08) or Prof. Coen De Roover (cderoove@vub.ac.be – 02/629.34.92).

- Participation in the steering board is free of charge.
- You send us a letter of intent, detailing what is most attractive to you.
- You commit to attend one meeting per year.

Current members

BARCO Visibly yours

ctg

TPVISION

MediaGeniX

inventivedesigners the output innovators

Tool demonstration

To learn more about what we achieved over the last two years, come to our tool demonstration:

Tuesday, February 24th – 13:00 till 17:30
Campus Middelheim, Universiteit Antwerpen

Details (program, location, registration) at http://soft.vub.ac.be/chaq/