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 optimise for perfection; in the latter they optimise 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

AGENTSCHAP INNOVEREN & ONDERNEMEN

The Cha-Q project addresses a range of challenges, including:

• *Monitoring code changes.* Monitor changes as they are made in an IDE or as they are committed to the version repository. When making changes to one artefact, receive notifications of which related artefacts should be changed as well. *(see p. 2 for more information)*

• *Automating code changes.* Reduce the effort to perform systematic edits, such as adjusting code to a newer API, migrating to another library or framework, fixing multiple instances of a bug, or applying custom refactorings. Replay code changes that were successful for a given branch on another branch, reducing manual branch synchronisation. *(p. 3)*

• *Monitoring test suite quality.* Automatically measure a test suite's ability to detect bugs in the system, based on previous bugs found in the project's history. If bugs are not caught by a test suite, the developer is assisted with improving the test code's strength. *(p. 4)*

• *Estimating task effort.* Based on the issue tracking history of a software project, automatically and reliably estimate the amount of effort that is required to address an issue or a task. *(p. 5)*

• *Mining repositories for systematic edits.* Discover occurrences of repeated, systematic edits in a project's history, and use this information to either minimise the need for such repeated edits, or generate a custom refactoring that automates them. *(p. 5)*

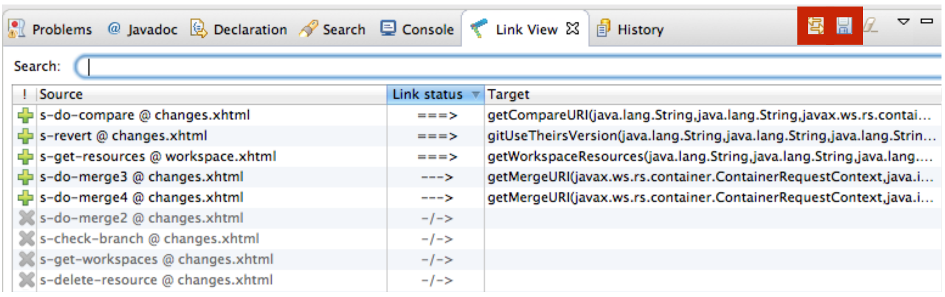
• *Monitoring the test process.* Determine the impact of changes on both the test and production code to persuade team members to increase testing activities. Demonstrate that it is possible to maintain the test code synchronised with the GUI. *(p. 6)*

Maintaining Traceability Links at Inventive Designers

powered by Cha-Q

Non-trivial software systems consist of many different types of artefacts that are interlinked. For instance, there are often links from XML files to specific classes in source code files. Changes to a single artefact can have an unanticipated impact on other related artefacts. These changes form a common source of errors in software projects that use multiple artefact types. While development environments support many different types of artefacts, they often do not consider the so-called traceability links between them.

These errors, while trivial in nature, are costly to find. To remedy this, we have developed a tool called MaTraca. This tool keeps track of the traceability links between artefacts and ensures that developers are aware of them: if one artefact is changed, MaTraca users are shown which parts of other artefacts must change as well. We have used MaTraca at Inventive Designers on their core product, Scriptura Engage. Scriptura Engage contains 200 plugin projects, and spans over 2 millions lines of code. Apart from source code, there are several other types of artefacts such as web pages and XML configuration files, where MaTraca serves to maintain the links between all of these artefacts.



!	Source	Link status	Target
✚	s-do-compare @ changes.xhtml	===>	getCompareURI(java.lang.String,java.lang.String,javax.ws.rs.contai...
✚	s-revert @ changes.xhtml	===>	gitUseTheirsVersion(java.lang.String,java.lang.String,java.lang.Strin...
✚	s-get-resources @ workspace.xhtml	===>	getWorkspaceResources(java.lang.String,java.lang.String,java.lang....
✚	s-do-merge3 @ changes.xhtml	--->	getMergeURI(javax.ws.rs.container.ContainerRequestContext,java.i...
✚	s-do-merge4 @ changes.xhtml	--->	getMergeURI(javax.ws.rs.container.ContainerRequestContext,java.i...
✖	s-do-merge2 @ changes.xhtml	-/->	
✖	s-check-branch @ changes.xhtml	-/->	
✖	s-get-workspaces @ changes.xhtml	-/->	
✖	s-delete-resource @ changes.xhtml	-/->	

The screenshot on the right depicts a view of the tool that, in this example, keeps track of links between web pages and their corresponding web service implementations. The tool notifies users about which links still are satisfied, and which links have become broken due to an artefact change.

MaTraca enables developers to mark links of interest, locate the specific artefact parts on both sides of each link, and evaluate compliance of these links. That is, whether a link still is correct, broken, or only partially correct because the parameters on both ends of the link no longer match. For example, a method parameter in a Java file may need to correspond with an attribute value in an XML configuration file. One of MaTraca's strengths is its configurability: it can be set up to monitor various types of links that are specific to one company or to a particular domain, such as links from functional requirements to tests in safety-critical domains.

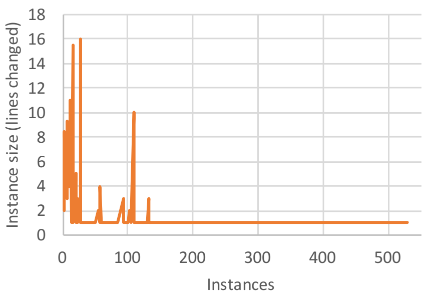
We have put MaTraca to the test at Inventive Designers with a pilot study, in which a group of junior and senior developers performed several development tasks, stemming from real modification requests, and involving different types of artefacts. For each task, we compare how developers who were allowed to use MaTraca fared against the developers who could only rely on their usual development environment. Our study confirms that MaTraca is beneficial for the identification and maintenance of links between artefacts, that it detects mistakes or omissions early, and that the tool gives the developers confidence that their modifications are correct.

Managing Traceability Links With MaTraca
Angela Lozano, Carlos Noguera, and Viviane Jonckers
Proceedings of the 23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering (SANER 2016)

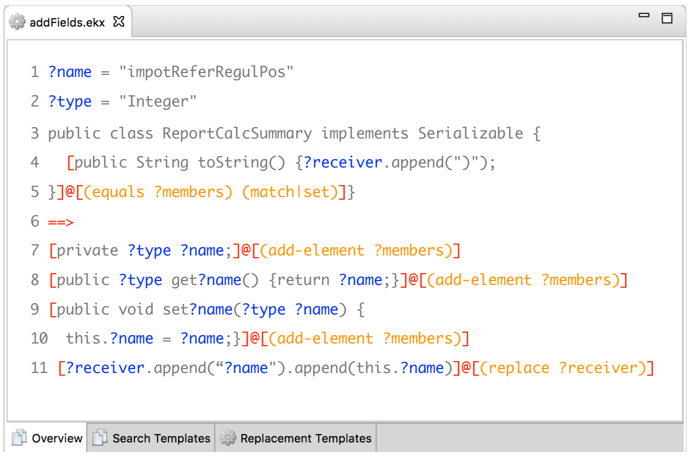
Automating Systematic Edits at FOD Financiën

Developers often need to perform similar, but non-identical, changes to different files. For instance, updating all clients of a library to a newer version. Manually performing such systematic edits is laborious and error-prone. To better understand the prevalence of systematic edits in practice, we analysed a year of development activity at FOD Financiën. We studied Tax-on-web's widely used tax simulation service (274.000 lines of code, 751 commits in a year). Producing correct results is crucial for this service, which is why FOD Financiën employs continuous integration to frequently test their code. However, the code often needs to be updated under time pressure to accommodate tax law changes.

As our study found that 16% of all commits contain systematic edits (each affecting 101.6 lines of code on average), tool support for performing systematic edits faster —and with fewer mistakes— is in order. Other researchers even found numbers as high as 75% for older projects that primarily receive maintenance work. The chart on the right provides a closer look into the systematic edits we found. Some involve one-line changes that are repeated in hundreds of different locations, whereas others represent larger changes that are only repeated tens of times. Both kinds of edits however represent a substantial manual effort, and a simple search-and-replace does not suffice for context-dependent modifications spread across hundreds of files.



The Cha-Q project has therefore developed a powerful, but user-friendly, program transformation tool. It won a best paper award at the *International Conference on Software Analysis, Evolution and Re-engineering*. Using this tool, we are able to fully automate about 62.6% of the systematic edits to the tax simulation service. The remaining systematic edits can be automated partially; i.e., they require some manual work afterwards.



```
1 ?name = "impotReferRegulPos"
2 ?type = "Integer"
3 public class ReportCalcSummary implements Serializable {
4   [public String toString() {?receiver.append(")");
5 }]@[(equals ?members) (match|set)]}
6 ==>
7 [private ?type ?name;]@[(add-element ?members)]
8 [public ?type get?name() {return ?name;}]@[(add-element ?members)]
9 [public void set?name(?type ?name) {
10  this.?name = ?name;}]@[(add-element ?members)]
11 [?receiver.append("?name").append(this.?name)]@[(replace ?receiver)]
```

One of the transformations used to automate the systematic edits is shown to the right. The effect of the transformation is shown below it. Whenever a field is added to the `ReportCalcSummary` class, its `toString()` method needs to be updated, and methods to get/set the field should be added. Lines 1-2 specify which field to add. Lines 3-5 specify which parts of the class need to change. Lines 7-11, after the ==> arrow, specify how these parts should change. Placeholder variables such as ?receiver substitute for code. Directives such as equals, match|set or add-element, exert fine-grained control over which code has to be changed and how.

```
public class ReportCalcSummary implements Serializable {
+ private Long impotReferRegulPos;
+ public Long getImpotReferRegulPos() {
+   return impotReferRegulPos;}
+ public void setImpotReferRegulPos(Long impotReferRegulPos) {
+   this.impotReferRegulPos = impotReferRegulPos;}
...
+ .append("impotReferRegulPos = ") .append(this.impotReferRegulPos)
```

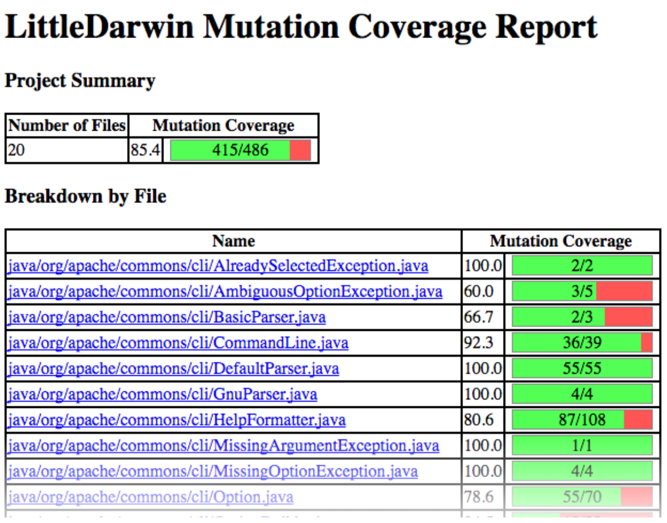
Our program transformation tool can be used to automate different scenarios involving systematic edits: repairing multiple instances of a bug, adjusting the code to API changes, migrating to alternative libraries or frameworks, creating custom refactorings, etc.

Search-Based Generalization and Refinement of Code Templates
Tim Molderez and Coen De Roover
Proceedings of the 8th Symposium of Search Based Software Engineering (SSBSE 2016)

Strengthening the Regression Test Suite at HealthConnect

HealthConnect is a Belgian company with deep expertise in integration architectures, software development, eHealth integration and project management in the health sector. One of HealthConnect's products is CareConnect: the first Belgian cloud-based Electronic Medical Record software for general practitioners. CareConnect engages itself to be the first to integrate the latest eHealth services into its software. To deal with the rapid changes, CareConnect has a fully automated regression test suite to ensure that the new features do not break any existing functionality. The reputation of CareConnect largely depends on their ability to prevent regression bugs, and HealthConnect was interested to assess potential weaknesses in their regression test suite.

The Cha-Q team made such an assessment by means of mutation testing, a well-studied method of strengthening a test suite. It consists of two phases: first, "mutants" are generated. A mutant is a buggy version of the code that is created by automatically injecting a single bug into the code. Second, the test suite is executed on this buggy version of the code to verify whether the test suite is able to detect the bug. If the bug is not caught by the test suite, this mutant has uncovered a weakness in the test suite. Because it is known which bug was injected, the test developer also has an indication of how the test suite should be improved.


LittleDarwin Mutation Coverage Report

Project Summary

Number of Files	Mutation Coverage	
20	85.4	415/486

Breakdown by File

Name	Mutation Coverage	
java/org/apache/commons/cli/AlreadySelectedException.java	100.0	2/2
java/org/apache/commons/cli/AmbiguousOptionException.java	60.0	3/5
java/org/apache/commons/cli/BasicParser.java	66.7	2/3
java/org/apache/commons/cli/CommandLine.java	92.3	36/39
java/org/apache/commons/cli/DefaultParser.java	100.0	55/55
java/org/apache/commons/cli/GnuParser.java	100.0	4/4
java/org/apache/commons/cli/HelpFormatter.java	80.6	87/108
java/org/apache/commons/cli/MissingArgumentException.java	100.0	1/1
java/org/apache/commons/cli/MissingOptionException.java	100.0	4/4
java/org/apache/commons/cli/Option.java	78.6	55/70

For this particular feasibility study, we modelled the mutants after a set of difficult bugs. We identified these bugs via a manual inspection of HealthConnect issue tracking system, looking for bugs which almost slipped into the field, would have caused major trouble, yet are very difficult to find. This inspection ultimately resulted in two classes of bugs: (i) collection order bugs, and (ii) null type bugs.

In the former, an order-preserving data structure is changed into an unordered one (e.g. a LinkedHashSet would be replaced by a HashSet). In the latter, null type bugs are introduced at different points in a method (e.g. the input value for a method is replaced by null, or instead of initialising an object, it is replaced by null). We injected these mutants into the common component of the CareConnect software suite (comprising over 50,000 lines of Java code), as it is one of the components suspected of having weaknesses. We discovered that overall the regression test suite does not catch most of these mutants (caught 1259 out of 10245 — 12.3% mutation coverage). The remaining mutants provide a detailed target for the tests that need to be added.

In the near future we will replicate this experiment in other sites. Here as well, we will start from the issue tracking system to identify difficult bugs, model mutants after these bugs, inject them into the system, run the regression test suite, and assess whether the test suite is strong enough.

Adaptable Mutation Testing for Continuous Integration Environments
Ali Parsai, Alessandro Murgia, Serge Demeyer and Coen De Roover
Proceedings of 14th BElgian-NEtherlands software eVOLution seminar (BENEVOL 2015)

Exploiting the Issue Tracker for Project Planning at Inventive Designers

Inventive Designers is a software company providing a multichannel customer communications platform. Internal software development is organised around SCRUM, an agile development process for developing and sustaining complex products. One technique used in the SCRUM process is planning poker. Over the last 2 years, Inventive Designers organises a planning poker session every two weeks and they are quite happy with the result. During this meeting, team members estimate the relative effort of the remaining issues and tasks in the sprint backlog. However, it sometimes is necessary to provide estimations for issues and tasks *before* the team gathers for the planning poker session.

That is where the Cha-Q team can help. We retrieved 699 issue reports stored in the issue tracking system of Inventive Designers (JIRA) for further analysis. We applied a series of text mining techniques on this data and derived a recommendation system to estimate the effort needed for addressing issues. An estimate is produced by comparing a pending issue report against all previous issues. These estimates can then be used before the planning poker session. A retrospective simulation shows that our estimates achieve a "mean magnitude of relative error" of 0.5, which is the same score as the developers' own estimations.

We also verified that the recommendation system can be helpful *during* planning poker sessions, where its estimates can supplement the discussion whenever there is a disagreement in the team. In this case, the system can show a list of keywords that influenced the estimate, which are invaluable for improving the issue reports and to reach agreement. Based on this experiment, we believe the system can help the team to schedule the developers' workload, and help long-term project planning.

Estimating Story Points from Issue Reports
Simone Porru, Alessandro Murgia, Serge Demeyer, Michele Marchesi, and Roberto Tonelli
Proceedings of the 12th Int. Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE 2016)

Mining systematic edits at TP Vision

When developing software, it sometimes is necessary to perform similar edits in multiple source code locations, e.g. when adjusting code to API changes. Such systematic edits may (unconsciously) be repeated several times. To provide insights into the systematic edits that occur throughout a project's history, we created a prototype tool. It makes use of big data processing techniques, to find similar changes that frequently occur together in one commit.

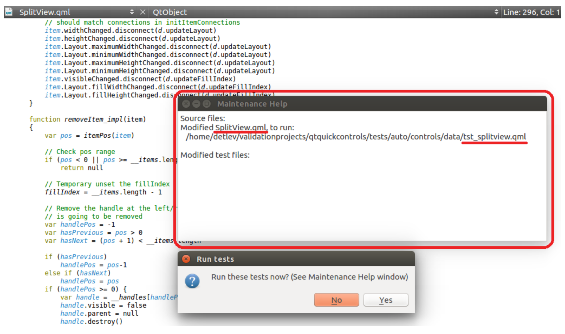
# Sys. edits	# Instances
467	3
221	4
98	5
88	6
49	7
41	8
29	9
15	10
10	11
5	12

To test our tool in the field, we did a study at TP Vision, which is engaged in producing a wide range of television sets. The software for each set shares many commonalities, but also has subtle differences. Because of this, it is important to know about the systematic edits that are present, and to design the system to keep these systematic edits at a minimum. In our study, we applied our tool to one of TP Vision's source code repositories (27.648 lines of code, 1654 commits). Our findings in the table to the right indicate that systematic edits do occur frequently. For example, the third row states that in 98 different systematic edits, a similar edit was performed in 5 different locations. The output of our tool can be used to motivate a restructuring of the code to minimise systematic edits. We can also automate a given systematic edit and generate a custom refactoring for it, to save time and effort whenever the systematic edit needs to be applied to an additional location.

Mining Change Histories for Unknown Change Patterns.
Arvid de Meyer
Master's thesis, August 2015, Vrije Universiteit Brussel

TestTraceabilityManager: Keeping the GUI and its Tests in Sync

Test maintenance is an important part of software development, especially in case of GUI tests. When code is added or modified, test maintenance is necessary to verify that the production code is still tested properly. Keeping track of the relationships between these tests and the code reduces the complexity of this task.



```
SplitView.qml    QtObject    Line: 296, Col: 1
        // should match connections in initItemConnections
        item.widthChanged.disconnect(d.updateLayout)
        item.heightChanged.disconnect(d.updateLayout)
        item.Layout.maximumWidthChanged.disconnect(d.updateLayout)
        item.Layout.minimumWidthChanged.disconnect(d.updateLayout)
        item.Layout.maximumHeightChanged.disconnect(d.updateLayout)
        item.Layout.minimumHeightChanged.disconnect(d.updateLayout)
        item.visibleChanged.disconnect(d.updateFillIndex)
        item.Layout.fillWidthChanged.disconnect(d.updateFillIndex)
        item.Layout.fillHeightChanged.dis
    }

    function removeItem_impl(item)
    {
        var pos = itemPos(item)

        // Check pos range
        if (pos < 0 || pos >= __items.len
            return null

        // Temporary unset the fillIndex
        fillIndex = __items.length - 1

        // Remove the handle at the left/
        // is going to be removed
        var handlePos = -1
        var hasPrevious = pos > 0
        var hasNext = (pos + 1) < __items

        if (hasPrevious)
            handlePos = pos-1
        else if (hasNext)
            handlePos = pos
        if (handlePos >= 0) {
            var handle = __handles[handleP
            handle.visible = false
            handle.parent = null
            handle.destroy()
```

```
Maintenance Help
Source files:
Modified SplitView.qml, to run:
 /home/detlev/validationprojects/qtquickcontrols/tests/auto/controls/data/tst_splitview.qml

Modified test files:
```

```
Run tests
Run these tests now? (See Maintenance Help window)
```

To do this, we created “TestTraceabilityManager”, a tool for automatically creating and maintaining traceability links between QML software artefacts and QML tests within the QtCreator IDE. This is the first tool that brings traceability links to the QML environment. The tool is also able to automatically maintain and adapt traceability links during development without needing a full re-analysis of the project. It tracks the changes made to the software project since the last analysis in a non-intrusive way. This approach can save valuable time for the maintainers of these tests. By analysing an open source system, we validated the correctness of the results against a manually created reference set of traceability links. Our tool correctly retrieved 97% of all links.

PDF **Automated traceability links between test and production code in QML interfaces**
Detlev Van Looy
Master’s thesis, January 2016, Universiteit Antwerpen

Cha-Q members

BARCO Visibly yours
ctg
TPVISION
MediaGenix
jamo solutions
inventivedesigners the output innovators
Federale Overheidsdienst FINANCIEN
health CONNECT ICT EXPERTISE THAT CARES
CM OMPARTNERS

Interested? - Contact us

Cha-Q is reaching its conclusion, but this is only the beginning of research on change-centric quality assurance, and we need *your* help!

If you are interested in launching a follow-up research project with us, 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).

Tool demonstration event

To learn more about what we achieved within the Cha-Q project, we invite you to join our tool demonstration event:

Monday, December 5th – 13:00 till 17:30
U-Residence, Vrije Universiteit Brussel

Program, location, and registration at http://soft.vub.ac.be/chaq/