How Blaise 5 is Tested: *"Ready! Test! Go!"*

Pre-Conference Session, IBUC 2018

Statistics Netherlands

This presentation is not about testing Blaise5 instruments, but about testing the product Blaise 5 which can be used to create Blaise 5 instruments.

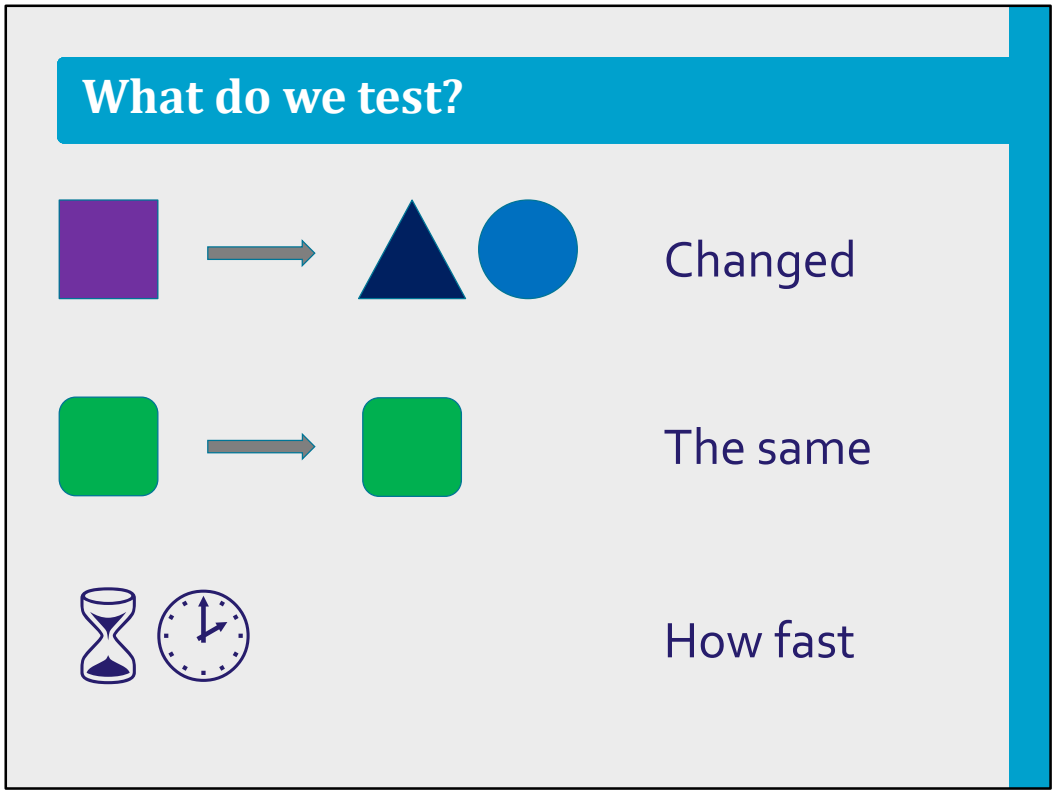
Testing a product like Blaise 5 is different from testing an ordinary application.

When designing, building and testing an ordinary application or information system, the functionality of the system can be fully described. There is a limited number of paths that a user can choose, there are only so many buttons on which he can click and the actions behind those buttons can be fully defined. Even then there are too many possible combinations to test them all. But at least we can assure a certain test coverage.

Blaise 5, however, is quite different. It is a tool for creating applications. A wide variety of applications can be created with it. The one questionnaire is a completely different application than the other questionnaire. And a Manipula program is a completely different application than a questionnaire.

So if we want to test a product like Blaise 5, it is not enough to test the applications that make up the product Blaise 5, such as the Control Centre, the Data Viewer and the Server Manager. We also need to test whether the applications that can be created with Blaise 5 work correctly. However, the variety of applications that can be created with Blaise is virtually infinite and we cannot test everything. So we have to make choices

In this session, I want to give you some insight in what we test and how we test this.



What do we test with each new Blaise build?
There are three things that we test with each new Blaise build, namely:

1. What should have been changed, has been changed correctly

What is it that should have been changed? For example Resolved Bugs. We check whether the problem that is supposedly fixed indeed does not occur anymore. What should have been changed includes also Improvements and New Features. We check whether they actually work correctly. This is all tested manually.

2. What should not have been changed, has remained the same

We check this in our Regression Test. Regression is the phenomenon that features that worked well before, do not work properly anymore after a code change. So we perform regression tests to verify whether features that were previously developed and tested, and should not be changed, still function the same after the software has been altered. This Regression test is a comprehensive set of tests that we are still expanding and that runs automatically every time a new build of the Blaise 5 software is created.

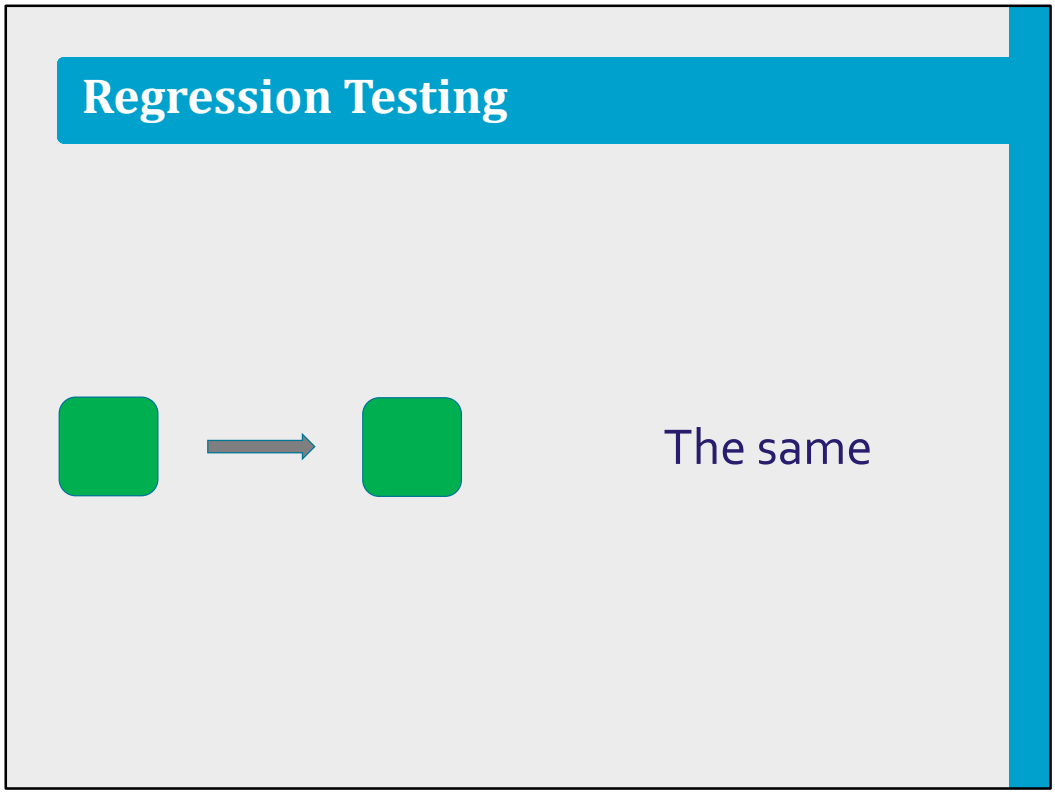
The things that should be changed, i.e. the Resolved Bugs, Improvements, and New Features, we first test manually. If we have found that they are working properly, they should not change again in the next Blaise build. So from then on they belong to

the set of things that should remain the same. Therefore, tests cases for these items should be added to the regression test.

3. Performance

We run Performance tests to examine how well Blaise responds under a specific workload. This way we can estimate which server capacity is needed to handle a certain amount of concurrent users and verify that the response time of the data entry clients is not too high. These Performance tests are also automated tests.

In the remainder of this session, I want to tell you something about this automated testing we do, i.e. our regression testing and our performance testing.

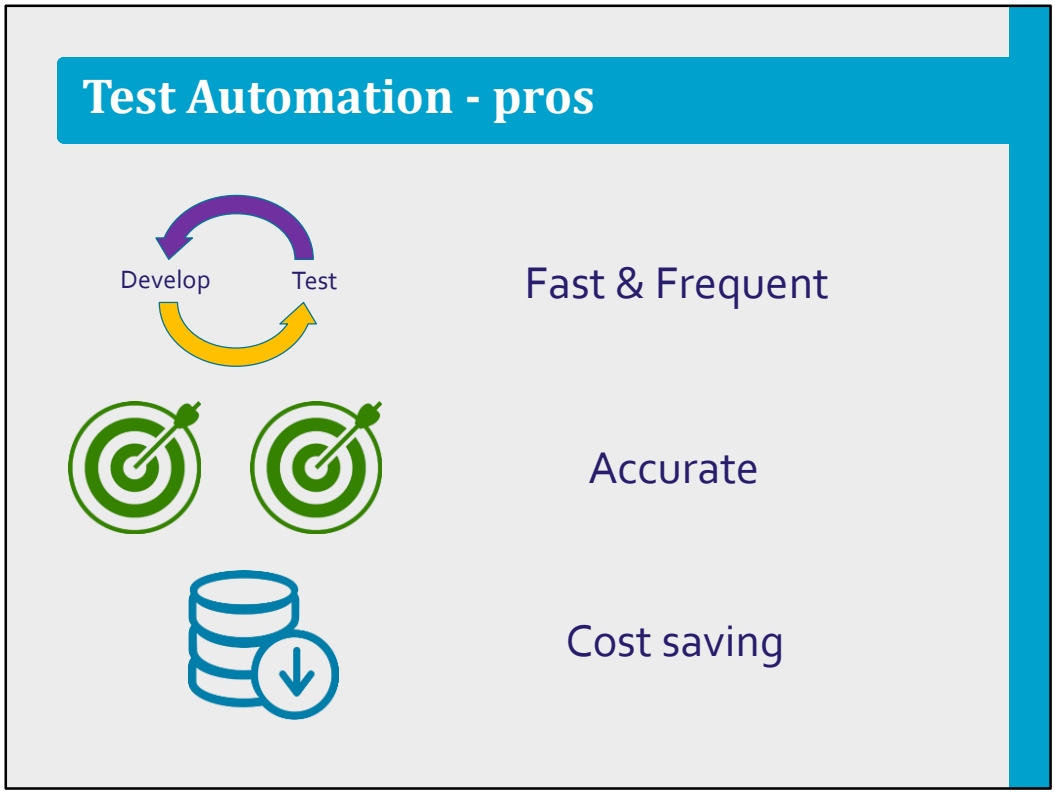


Regression is the phenomenon that features that worked well before, do not work properly anymore after the software has been altered.

Regression testing is the re-running of tests to verify whether the features that were previously developed and tested, still function correctly after a software change.

It is not possible to include everything in a regression test, because it is impossible to test every possible combination. So it is always possible that regression slips through unnoticed. The more extensive and the more detailed the regression test, the less likely that regression slips through unnoticed. So we always keep expanding the regression test.

Because the same tests have to be executed over and over again with each new Blaise build, it is a good thing to automate them.



Automating the regression tests has several advantages.

Regression tests that have been automated can be executed fast and frequently, preferably each time that a new build of the software is generated. This way the developers get early feedback on their work.

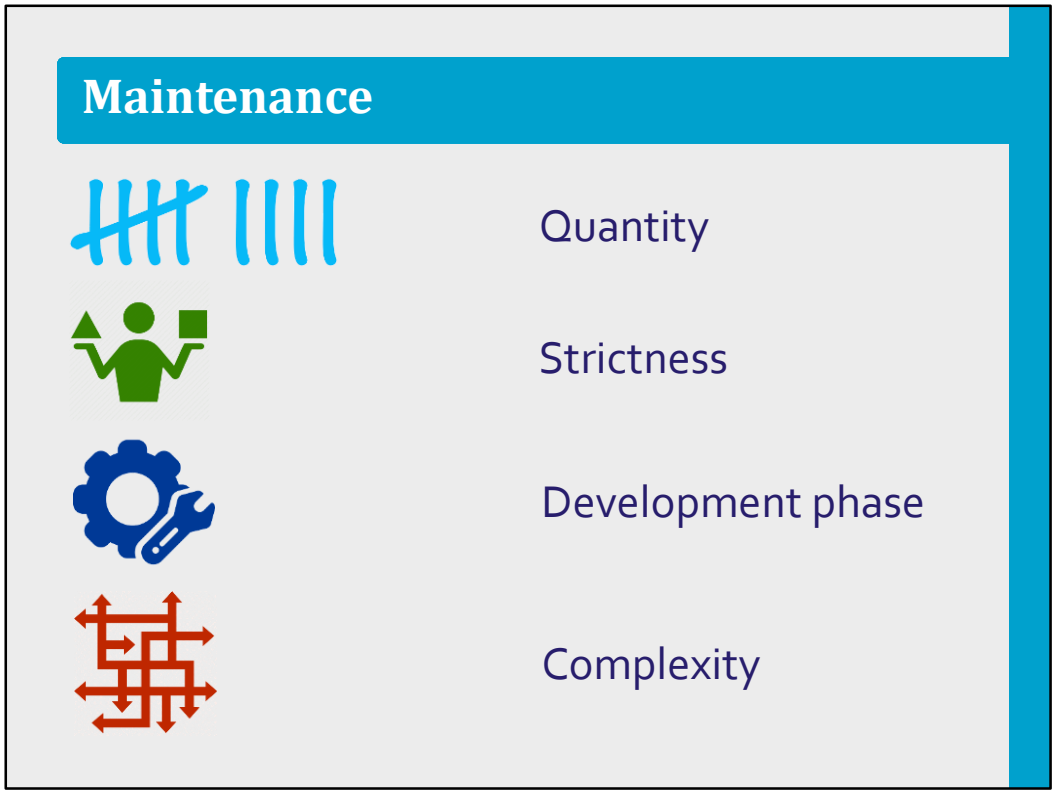
Another advantage of test automation is the certainty that the tests are executed in exactly the same way as they were executed in previous test runs and that no human mistakes are made.

The third advantage is that automating regression tests is cost saving in the long run. To automate tests is more expensive than to execute them manually once. But when test execution is repeated often, it is more efficient to automate the tests.

Test Automation - cons

Maintenance

But there is also a drawback. The automated test cases have to be maintained. And the more extensive and the more detailed the regression test is, the more maintenance it needs.



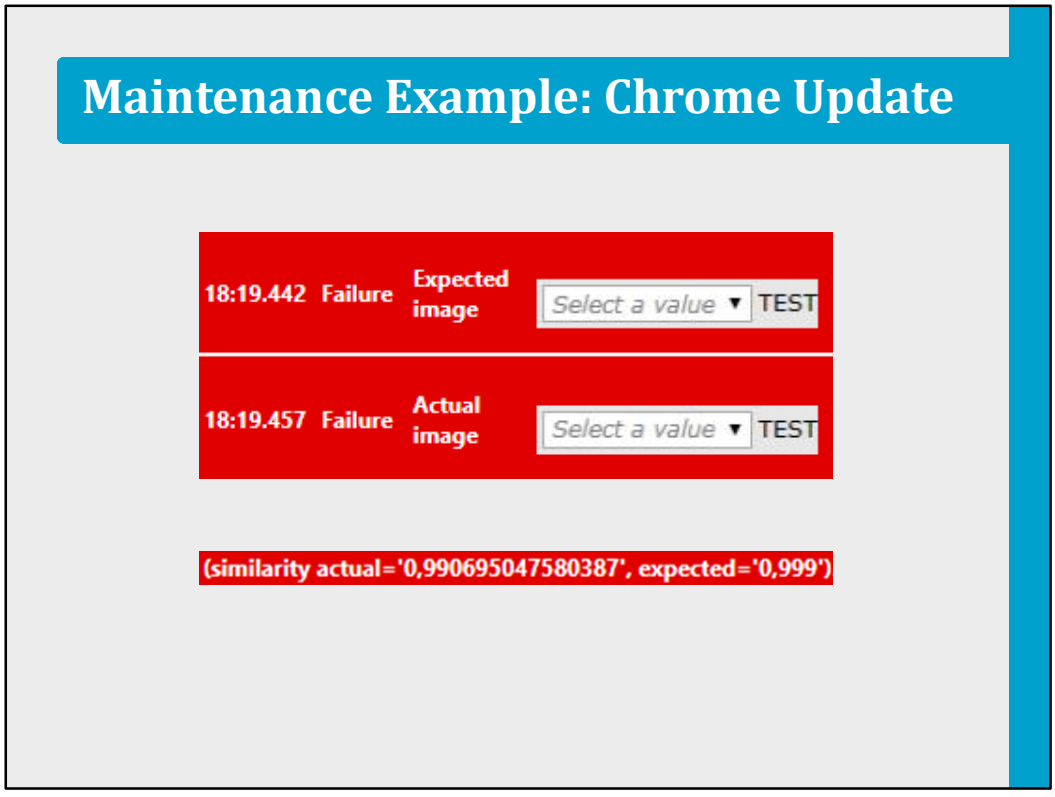
What determines the amount of maintenance needed?
Amount of tests
Strictness of comparison
Development phase of features
Complexity of test cases

Amount of tests
The more tests we have, the more tests we have to maintain.
Alternative: Less tests
Drawback: Less coverage

Strictness of comparison
This applies especially to the layout tests. If we create tests in which we check whether a survey page looks exactly the same as in a previous build, so not one pixel can be different, then a minor change, which will not even be noticed with the naked eye and which is harmless or even by design or maybe the result of a browser update, will cause these tests to fail. And we would have to repair the tests by replacing all reference images. So in our layout tests we might not check if the images exactly match the reference images, but for example if they match 99.9%. However, we cannot be too lenient, because then harmful unintended layout changes might slip through.
Alternative: Less strict comparison
Drawback: Unintended layout changes might slip through

Development phase of features
It is usually a bad idea to automate the testing of features that are still under development, especially when a user interface is involved. Because then one has to continuously adjust the test cases to the changed functionality and the changed user interface. But on the other hand, when changes are made to a certain feature, that is when you most need a regression test. Because then the chance of accidentally adjusting things that should not be adjusted is greatest. We therefore have to make a trade-off between more maintenance because a feature is still under development and a greater chance of regression because a feature is still under development.
Alternative: Creating regression tests only after development of the feature has finished
Drawback: There is no regression test when it is needed the most

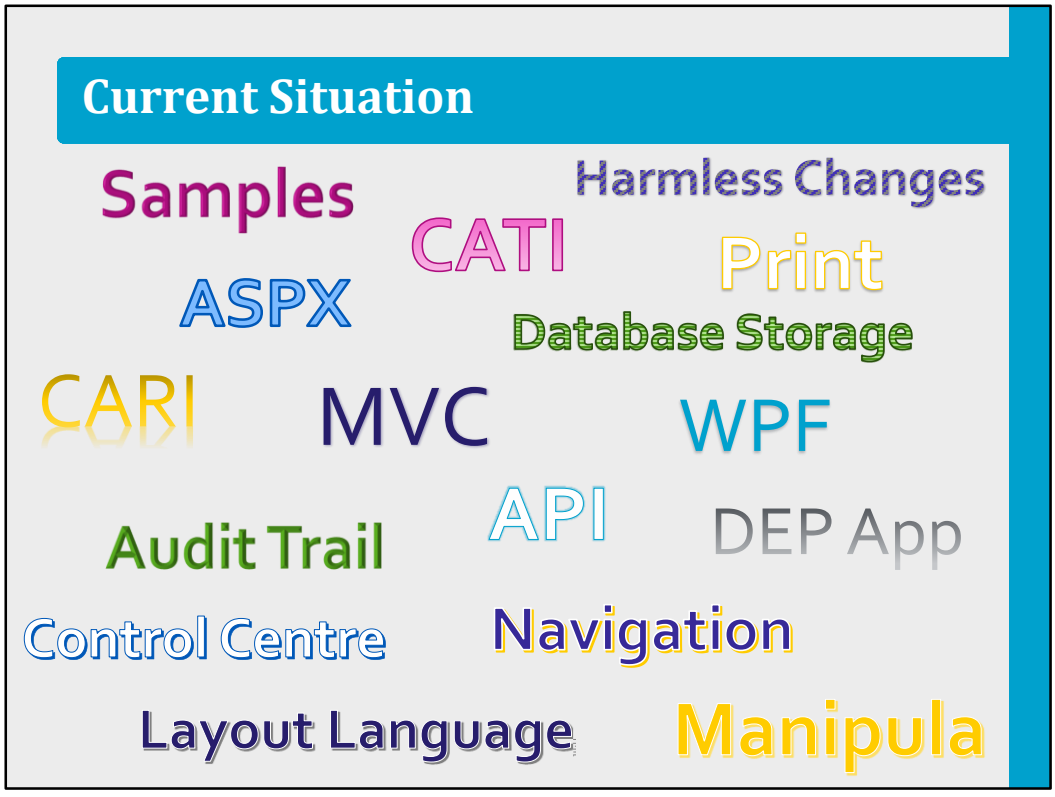
Complexity of test cases
If we create simple and short questionnaires to test only one aspect of Blaise at a time, then our tests only need maintenance if there has been a change which concerns that aspect. And if a test case fails it is easy to see why the test case fails and what has to be modified, because the test case is so nice and clear. But if we only have simple test cases, we would not detect bugs that only occur when several aspects are combined in complex questionnaires. However, with complex test cases it is not always clear right away what goes wrong and how it can be solved. It takes more effort to analyze that.
Alternative: Adding only simple test cases
Drawback: Regression that only occurs in certain more complex situations is not detected



This is an outtake from a test report of an MVC client layout test that was executed after a Chrome update.

The upper picture is what we expected, i.e. what was visible before the browser update. The bottom picture shows what was actually visible in the browser after the browser update. In the lower picture the text 'Select a value' is placed a bit higher in the dropdown box. Our test automation tool detects that the images are only the same for about 99%. We have defined in our test that the images should be at least 99.9% the same for the test to pass. That is why the test fails.

This time the difference between the two images was caused by a browser update and not by a change in Blaise. But we cannot be too lenient and only demand a 99% similarity. Because then a similar change in the position of a text as a result of a Blaise bug would remain undetected. So we have to keep checking for a 99.9% similarity. This means that we have to replace all the reference images of the tests that are affected by the browser update.



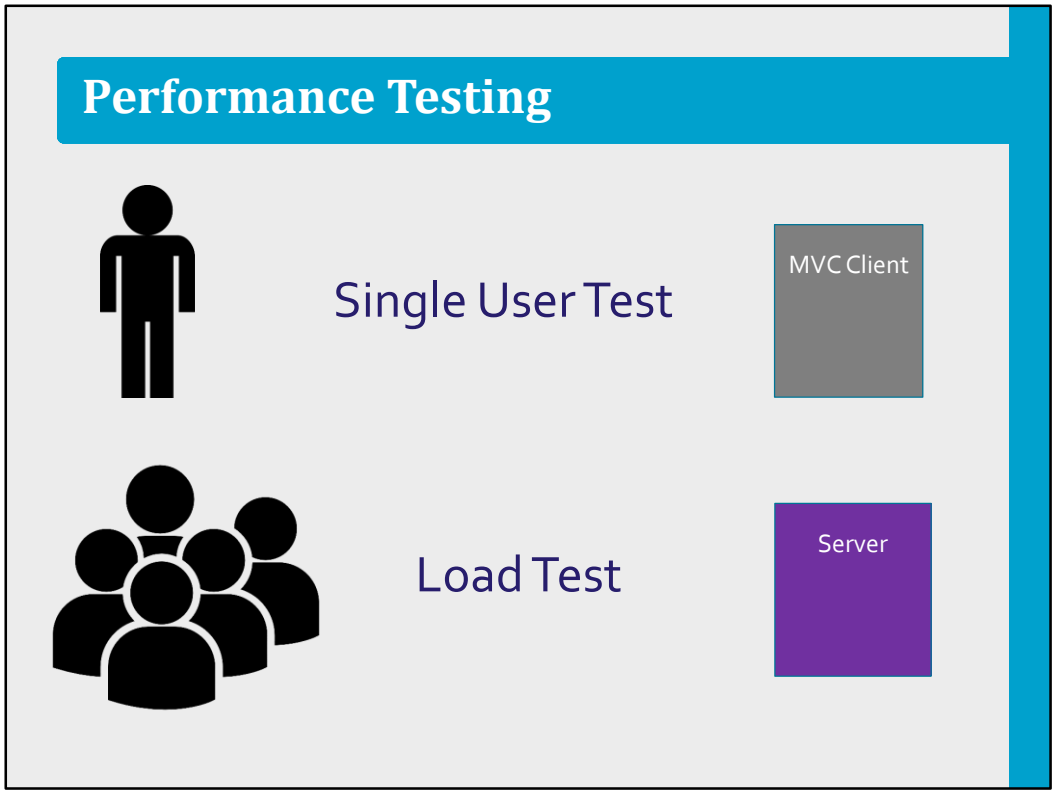
We started building our automated regression test in July 2015 and are still expanding it.

Currently, automated tests have been implemented for these components.

Test Automation - tooling

Ranorex
The tool we use for test automation is Ranorex. Ranorex is well suited for testing Windows applications and web applications on desktops.

CrossBrowserTesting
We are currently in the process of expanding our automated regression testing to mobile devices. We are doing a pilot for the automated regression testing of our MVC web client on mobile devices using SmartBear's CrossBrowserTesting.com. CrossBrowserTesting.com is a web service for running web tests on real mobile and desktop devices in the cloud.

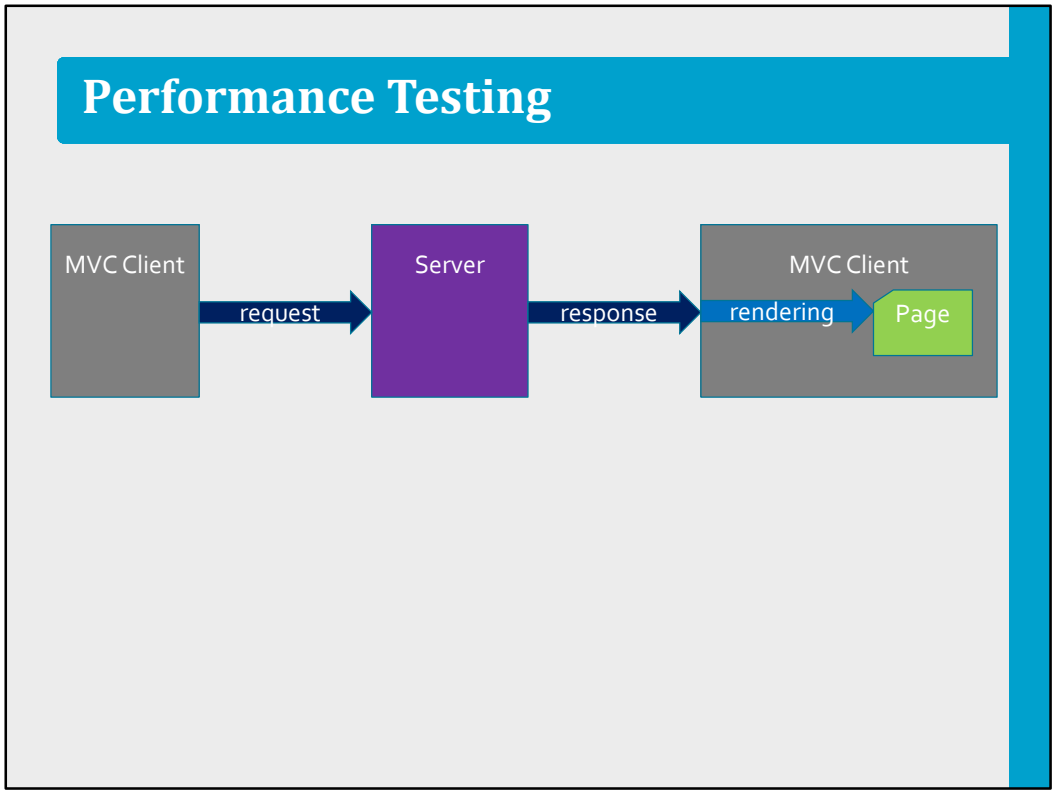


A performance test is a test that examines how fast a system responds under specific conditions.

We have implemented two kinds of performance tests for Blaise: a Single User Test and a Load Test.

Single User Test
In the Single User Test we measure how long it takes for the MVC client (i.e. the Blaise web client) to process a certain action. We test this on several devices with different operating systems and different browsers. This way we get an idea of how fast the MVC client performs on all those device / operating system / browser combinations.

Load Test
In the Load Test we measure how long it takes for the server to respond to a request of the client. We measure this time while the server handles an increasing number of requests simultaneously.

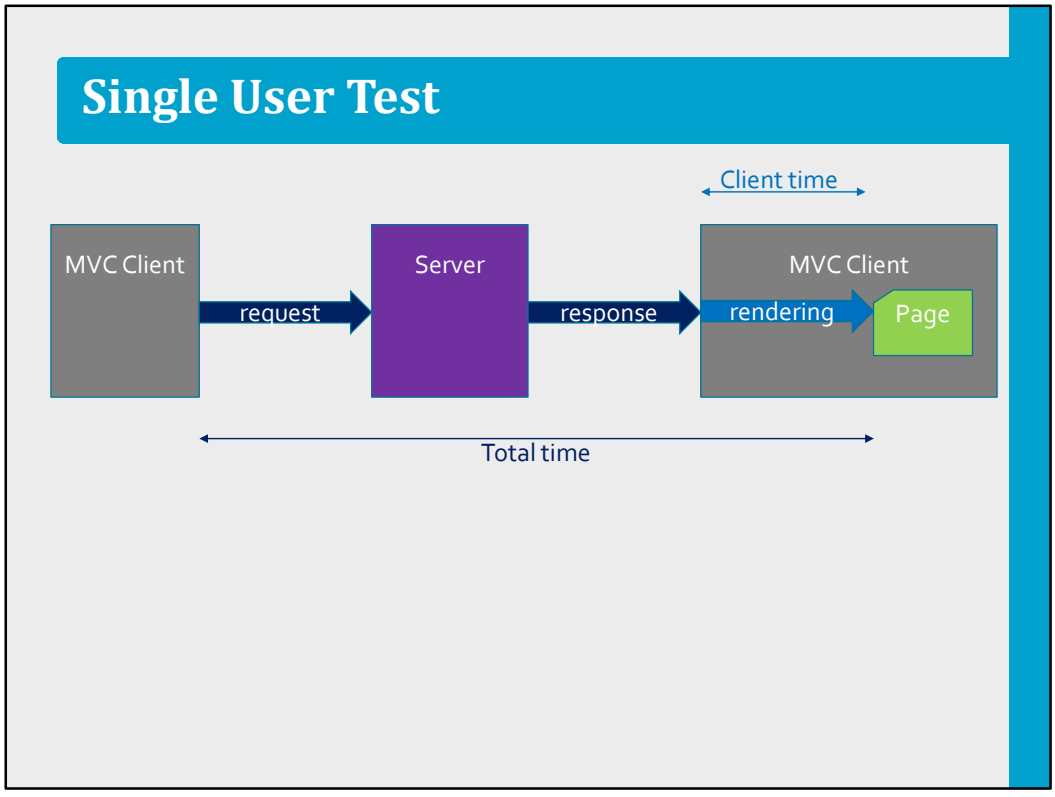


The setup we use for these performance tests is as follows.

We have an MVC client in which a user performs a certain action, for example clicking the Next button. A request is then sent to the server. The server processes the request and sends the new page instructions back to the client. The client receives the response and renders the new page.

The two clients in the picture are one and the same. I have only drawn the client a second time to give a chronological picture.

We run the performance tests on a client server environment that is separate from our common test environment to prevent other processes from influencing the measurements.



For our single user performance test, a questionnaire from the health survey of Statistics Netherlands, called the ‘Gezo’, is used.

Ranorex fills out the Gezo questionnaire automatically.

In the single user test we measure how long it takes for a user's request to be fully processed. We measure the time between the moment the user performs a certain action for which a server request is sent and the moment the request is fully processed, in this case the moment at which the new page is fully rendered. We call this time the 'total time'.

We also measure how long it takes the MVC client to render a page. We measure the time between the moment the MVC client receives the page instructions from the server and the moment that the page is fully rendered in the browser. We call this the 'client time'.

All these measurements are done automatically by the MVC client.

These measurements are done for several combinations of browsers, operating systems and devices and for a variety of page complexities.

Single User Test Results: Client Time

	Blaise version			
Device / OS / Browser	**5.2.0**	**5.2.5**	**5.3**	**5.4**
Desktop - Windows 7 - Chrome	1,28	0,06	0,09	0,10
Desktop - Windows 7 - Firefox	1,18	0,11	0,17	0,16
Desktop - Windows 7 - IE11	2,28	0,16	0,38	0,45
Desktop - Windows 10 - Chrome	1,46	0,06	0,08	0,10
Desktop - Windows 10 - Firefox	1,13	0,13	0,16	0,13
Desktop - Windows 10 - IE11	2,68	0,16	0,35	0,46
Desktop - Windows 10 - Edge	2,86	0,09	0,20	0,22
Samsung Galaxy Tab S2 - Android 6.0 - Chrome	5,80	0,31	0,33	0,51
Samsung Galaxy Tab S2 - Android 6.0 - Internet	5,61	0,29	0,41	0,44
iPad Air2 - iOS 10.0 - Chrome	2,61	0,08	0,14	0,15
iPad Air2 - iOS 10.0 - Safari	2,45	0,09	0,11	0,15
Average Client Time (seconds)	**2,67**	**0,14**	**0,22**	**0,26**

Desktop specs:
Fujitsu ESPRIMO_D556
CPU : Intel Core i7-6700 @ 3.40GHz (4 cores, 8 logical processors)
Memory : 16 GB
Hard drive 256 : GB SSD

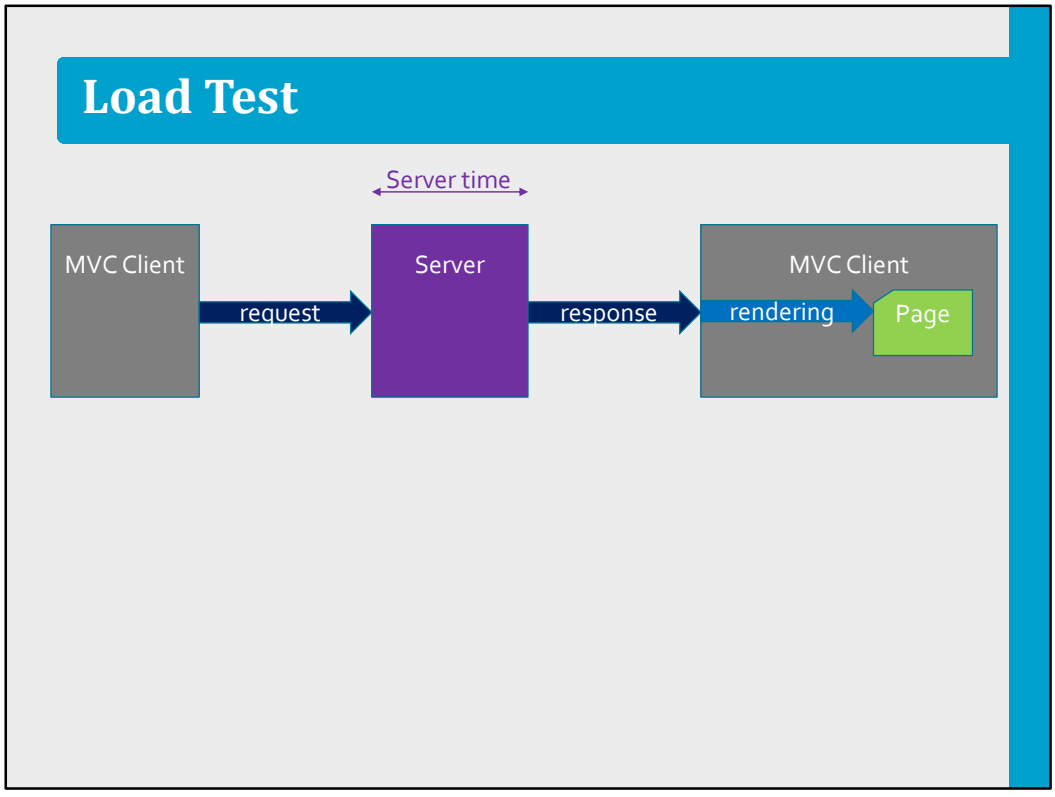
The performance tests have been executed with the following Blaise builds:
5.2.0 build : 5.2.0.1175
5.2.5 build : 5.2.5.1322
5.3 build : 5.3.0.1493
5.4 build : 5.4.2.1669

Single User Test Results: Total Time

	Blaise version			
Device / OS / Browser	**5.2.0**	**5.2.5**	**5.3**	**5.4**
Desktop - Windows 7 - Chrome	1,38	0,12	0,16	0,16
Desktop - Windows 7 - Firefox	1,28	0,19	0,23	0,22
Desktop - Windows 7 - IE11	2,41	0,25	0,45	0,55
Desktop - Windows 10 - Chrome	1,60	0,12	0,14	0,15
Desktop - Windows 10 - Firefox	1,32	0,21	0,22	0,18
Desktop - Windows 10 - IE11	2,74	0,23	0,41	0,52
Desktop - Windows 10 - Edge	2,97	0,18	0,27	0,29
Samsung Galaxy Tab S2 - Android 6.0 - Chrome	5,90	0,37	0,42	0,58
Samsung Galaxy Tab S2 - Android 6.0 - Internet	5,69	0,36	0,48	0,52
iPad Air2 - iOS 10.0 - Chrome	2,71	0,14	0,21	0,21
iPad Air2 - iOS 10.0 - Safari	2,54	0,18	0,18	0,21
Average Total Time (seconds)	**2,78**	**0,21**	**0,29**	**0,33**

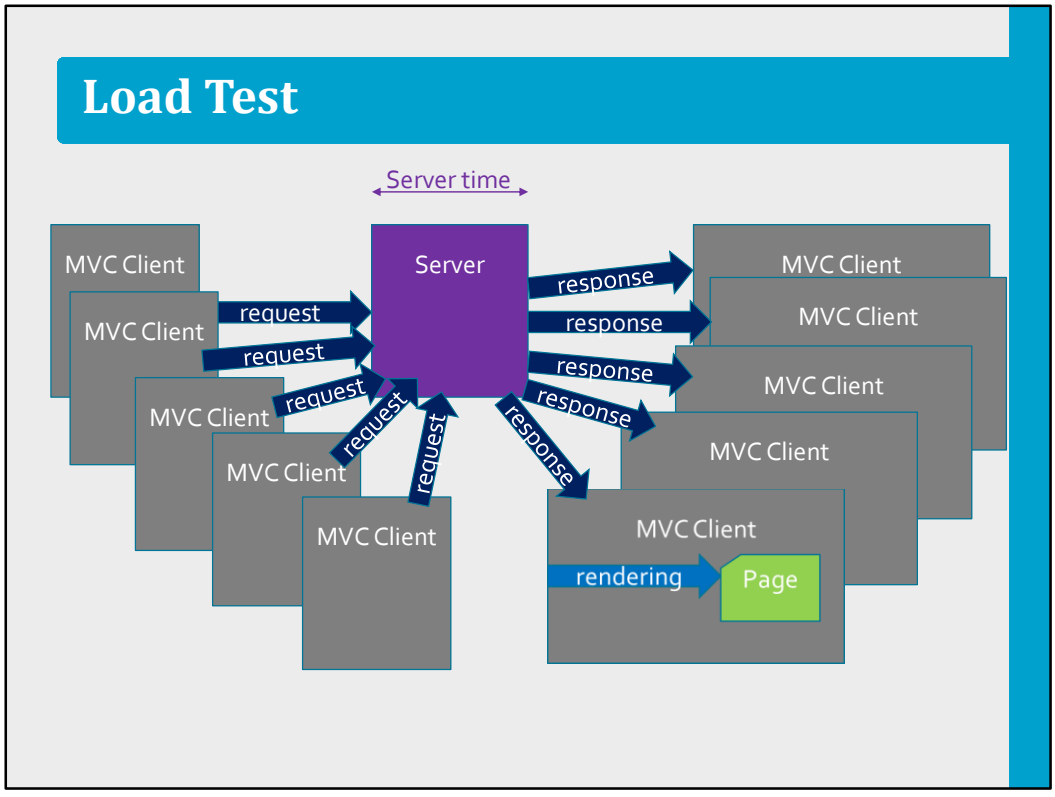
Desktop specs:
Fujitsu ESPRIMO_D556
CPU : Intel Core i7-6700 @ 3.40GHz (4 cores, 8 logical processors)
Memory : 16 GB
Hard drive 256 : GB SSD

The performance tests have been executed with the following Blaise builds:
5.2.0 build : 5.2.0.1175
5.2.5 build : 5.2.5.1322
5.3 build : 5.3.0.1493
5.4 build : 5.4.2.1669



In the Load Test we use the same setup and questionnaire as in the Single User Test.

But now we are not interested in the client time but in the server time. This is the time between the moment the server receives the client's request and the moment the server sends its response.



While we measure this server time, we increase the number of requests the server has to process simultaneously by increasing the amount of concurrent users. We do this with the Load Testing feature of Visual Studio. In this way, we measure for different amounts of concurrent users the average response time of the server.

Load Test Results: Server Time

	Blaise version			
Concurrent Users	5.2.0	5.2.5	5.3	5.4
100	0,10	0,08	0,08	0,06
200	0,25	0,24	0,17	0,18
250	0,41	0,25	0,43	0,48
275	0,53	0,89	0,81	0,62
300	1,77	1,69	1,55	1,40

Server park setup:
Server1 : Web / Data-Entry / Resource
Server2 : Audit / Cati / Data / Session (Management)

Server specs:
Fujitsu ESPRIMO_D556
CPU : Intel Core i7-6700 @ 3.40GHz (4 cores, 8 logical processors)
Memory : 16 GB
Hard drive 256 : GB SSD
OS: Windows 2012 R2

The performance tests have been executed with the following Blaise builds:
5.2.0 build : 5.2.0.1175
5.2.5 build : 5.2.5.1322
5.3 build : 5.3.0.1493
5.4 build : 5.4.2.1669

Questions?