Ranorex®

BEST PRACTICES

10 best practices in test automation

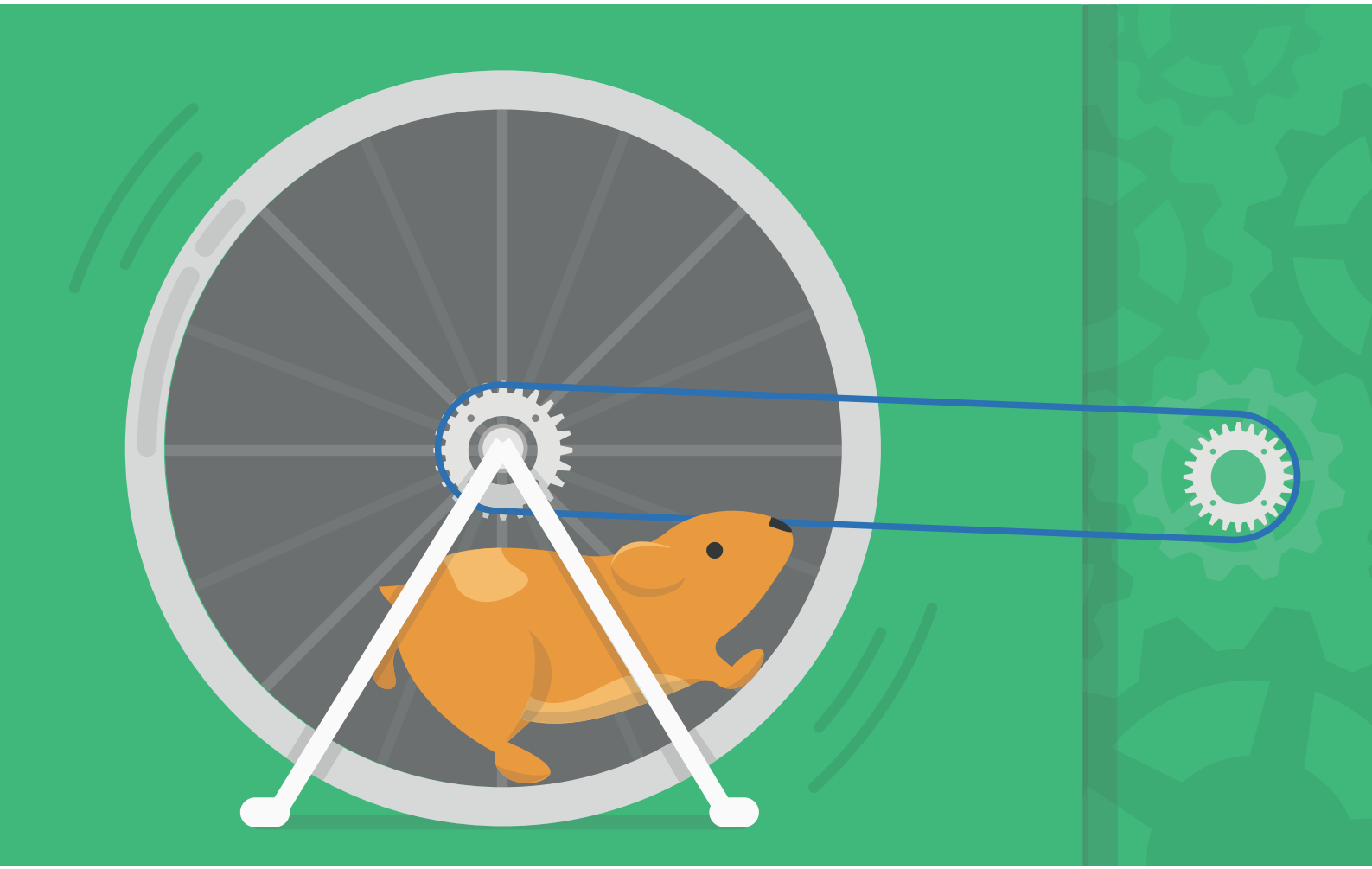
Ranorex®

10 best practices in test automation

Introduction

Software testing is a dynamic field, with the pace of change being driven not just by the continual evolution of development platforms, but also by advances in the automation tools available to testers. This ebook describes 10 best practices that can help ensure automated testing delivers fast, reliable results with a minimum of overhead.

Contents



1. Know what to automate

Because testing resources are limited, one of the first considerations in launching a test automation project is where to focus your efforts. Which test cases will give you the highest return on the time and effort invested? This section provides recommendations for three types of test cases: those to automate, those that will be challenging to automate, and those that shouldn't be automated at all.

What should be automated

In theory, any software test can be automated. The question is whether a particular test will cost more to develop and maintain than it will save in testing. To get the best return on your effort, focus your automation strategy on test cases that meet one or more of the following criteria:

Tests for stable features

Automating tests for unstable features may end up costing significant maintenance effort. To avoid this, test a feature manually as long as it is actively undergoing development.

Regression tests

A regression test is one that the system passed in a previous development cycle. Re-running your regression tests in subsequent release cycles helps to ensure that a new release doesn't reintroduce an old defect or introduce a new one. Because regression tests are executed often, they should be at the top of your priority list for automation. To learn more about regression testing, refer to the Ranorex Regression Testing Guide.

High-risk features

Use risk analysis to determine which features carry the highest cost of failure; then focus on automating those tests. Then, add those tests to your regression suite. For more information on how to prioritize test cases based on risk, see the section on risk assessment in the *Ranorex GUI Testing Guide*.

Smoke tests

Depending on the size of your regression suite, it may not make sense to execute the entire suite for each new build of the system. Smoke tests are a subset of your regression tests which check that you have a good build prior to spending time and effort on further testing. Smoke testing typically includes checks that the application will open, allow login, and perform other key functions. Include smoke tests in your Continuous Integration (CI) process and trigger them automatically with each new build of the system.

Data-driven tests

Any tests that will be repeated are good candidates for test automation, and chief among these are data-driven tests. Instead of manually entering multiple combinations of username and password, or email address and payment type to validate your entry fields, let an automated test do that for you. Best practices for data-driven tests are discussed in section 4.

Load tests

Load tests are simply a variation on data-driven testing, where the goal is to test the response of the system to a simulated demand. Combine a data-driven test case with a tool that can execute the test in parallel or distribute it on a grid to simulate the desired load.

Cross-browser tests

Cross-browser tests help ensure that a web application performs consistently regardless of the version of the web browser used to access it. It is generally not necessary to execute your entire test suite against every combination of device and browser, but instead to focus on the high-risk features and most popular browser versions currently in use. Currently, Google Chrome is the leading browser on both desktop and mobile, and the second-largest on tablets behind Safari. So, it would make sense to run your entire test suite against Chrome, and then your high-risk test cases against Safari, Firefox, Internet Explorer, and Microsoft Edge.

Cross-device tests

Mobile apps must be able to perform well across a wide range of sizes, screen resolutions, and O/S versions. According to Software Testing News, in 2018, a new manual testing lab would need almost 50 devices just to provide 80% coverage of the possible combinations. Automating cross-device tests can reduce testing costs and save significant time.

What is difficult to automate

The following types of test cases are more difficult to automate. That doesn't mean that they shouldn't be automated – only that these test cases will have a higher cost in terms of time and effort to automate. Whether a particular test case will be challenging to automate varies depending on the technology basis for the AUT. If you are evaluating an automation tool or doing a Proof of Concept, be sure that you understand how the tool can help you overcome these difficult-to-automate scenarios.

Mixed-technology tests

Some automated tests require a mix of technologies, such as a hybrid mobile app or a web app with backend database services. To make automating end-to-end tests in this type of environment easier, the ideal solution is to implement an automation framework that supports all of the technologies in your stack. To see whether Ranorex Studio is a good fit for your stack, visit our Supported Technologies page.

Dynamic content

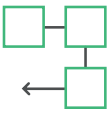
There are many types of dynamic content, such as web pages built based on stored user preferences, PDF documents, or rows in a database. Testing this type of content is particularly challenging given that the state of the content is not always known at the time the test runs. To learn more, refer to the Ranorex blog article Automated Testing and Dynamic IDs.

Waiting for events

Automated tests can fail when an expected response is not received. It's important to handle waits so that a test doesn't fail just because the system is responding slower than normal. However, you must also ensure that a test does fail in a reasonable period of time so that the entire test suite is not stuck waiting for an event that will never happen. To learn how to configure waits in Ranorex automated tests, refer to the list of available actions in the Ranorex Studio User Guide.

Handling alerts/popups

Similar to waiting for events, automated tests can fail due to unexpected alerts or pop-ups. To make them more stable, be sure to include logic in your test to handle these types of events. Ranorex Studio includes an automation helper that makes it easy to handle alerts and pop-ups.



Complex workflows

There are several challenges to automating a workflow. Typically, a workflow test will consist of a set of test cases that each check steps in the workflow. If one step fails, subsequent test steps will not run. Because the steps must be performed in order, they can't be split across multiple endpoints to run in parallel. Another challenge is that automating a workflow involves choosing one particular path through the application, possibly missing defects that occur if a user chooses a different path in production. To minimize these types of issues, make your test cases as modular and independent of each other as possible, and then manage the workflow with a keyword-driven framework.

Certain aspects of web applications

There are aspects of web applications that present unique challenges to automation. One of the primary issues is recognizing UI elements with dynamic IDs. Ranorex provides "weight rules" to tweak the RanoreXPath for specific types of elements, which helps ensure robust object recognition even on dynamic IDs. Other challenges in automating web applications include switching between multiple windows and automating iframes — especially those with cross-domain content. Ranorex Studio can detect and automate objects inside cross-domain iframes, even when web security is enabled.

Certain aspects of mobile applications

Mobile apps can also be challenging to automate. For example, you must ensure that your application responds appropriately to interruptions such as the phone ringing or a low battery message. You must also ensure that your tests provide adequate device coverage, which is a particular challenge for Android apps due to the wide variety of screen sizes, resolutions, and O/S versions found in the installed base. Finally, due to differences between iOS and Android, tests that are automated for a native app on one platform will likely require adaptation to perform as expected on the other platform. As with other difficult-to-automate tests, it's essential to have a testing framework that supports the full technology stack for your application under test.

What shouldn't be automated

There are some types of tests where automation may not be possible or advised. This includes any test where the time and effort required to automate the test exceeds the potential savings. Plan to perform these types of tests manually.

Single-use tests

It may take longer to automate a single-use test than to execute it manually once. Note that the definition of "single-use tests" does not include tests that will become part of a regression suite or that are data-driven.

Tests with unpredictable results

Automate a test when the result is objective and can be easily measured. For example, a login process is a good choice for automation because it is clear what should happen when a valid username and password are entered, or when an invalid username or password are entered. If your test case doesn't have clear pass/fail criteria, it would be better have a tester perform it manually.

Features that resist automation

Some features are designed to resist automation, such as CAPTCHAs on web forms. Rather than attempting to automate the CAPTCHA, it would be better to disable the CAPTCHA in your test environment or have the developers create an entry into the application that bypasses CAPTCHA for testing purposes. If that isn't possible, another solution is to have a tester manually complete the CAPTCHA and then execute the automated test after passing the CAPTCHA. Just include logic in the test that pauses until the tester is able to complete the CAPTCHA, and then resumes the test once login success is returned.

Unstable features

It is best to test unstable features manually. Invest the effort in automation once the feature has reached a stable point in development.

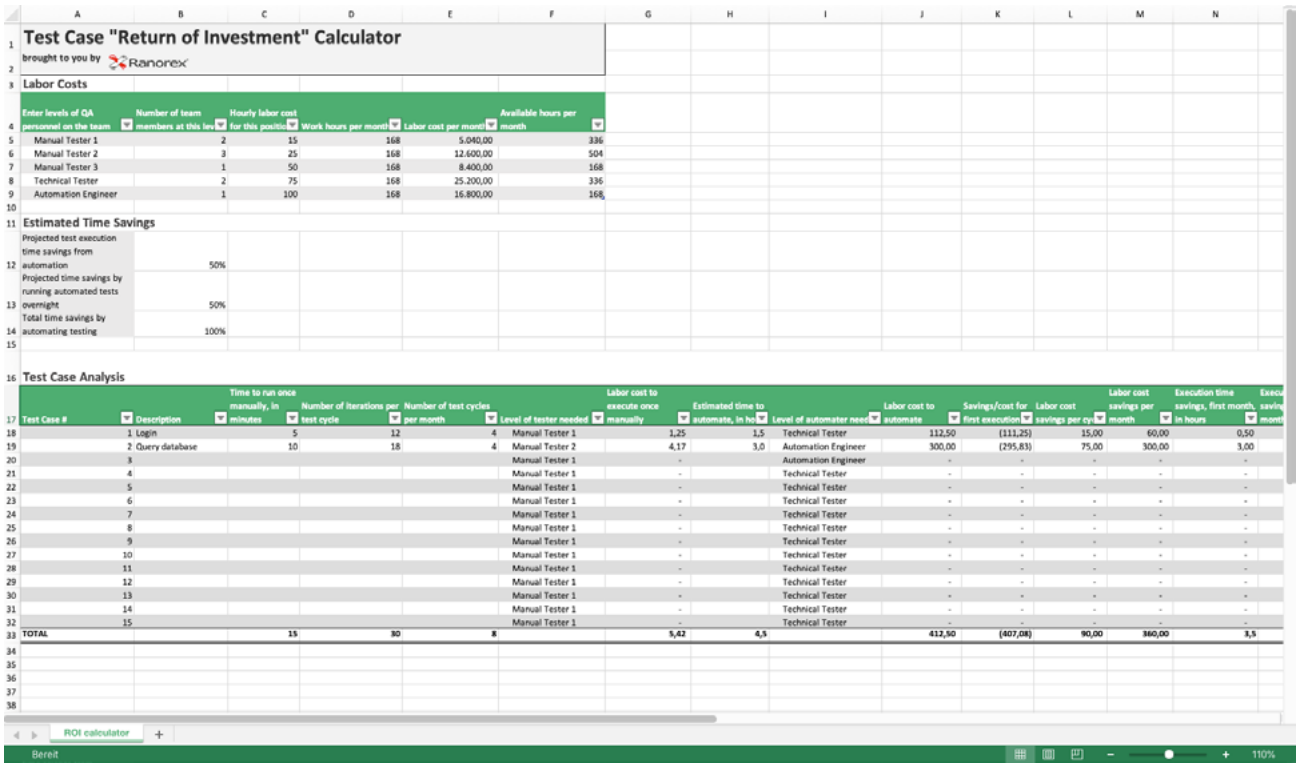
Native O/S features on mobile devices

Particularly on Apple iOS, non-instrumented native system apps are difficult or impossible to automate due to built-in security.

Know what to automate: summary

To ensure that you achieve your automation goals, focus your automation efforts on the right test cases. And be sure to build in time for exploratory testing and UX/usability testing – by their nature, these types of tests can't and shouldn't be automated. To help determine whether or not to automate a particular test case, you can download the Test Case ROI Calculator spreadsheet from the Resources section of the Ranorex website.

As shown in the image below, this simple spreadsheet compares the estimated time and costs to automate a test case vs. the time and costs to execute the same test case manually; it is not designed to determine the ROI of a test automation project.



Test Case "Return of Investment" Calculator

brought to you by Ranorex

Labor Costs

Enter levels of QA personnel on the team	Number of team members at this lev	Hourly labor cost for this positio	Work hours per month	Labor cost per month	Available hours per month
Manual Tester 1	2	15	168	5.040,00	336
Manual Tester 2	3	25	168	12.600,00	504
Manual Tester 3	1	50	168	8.400,00	168
Technical Tester	2	75	168	25.200,00	336
Automation Engineer	1	100	168	16.800,00	168

Estimated Time Savings

Projected test execution time savings from automation	50%
Projected time savings by running automated tests overnight	50%
Total time savings by automating testing	100%

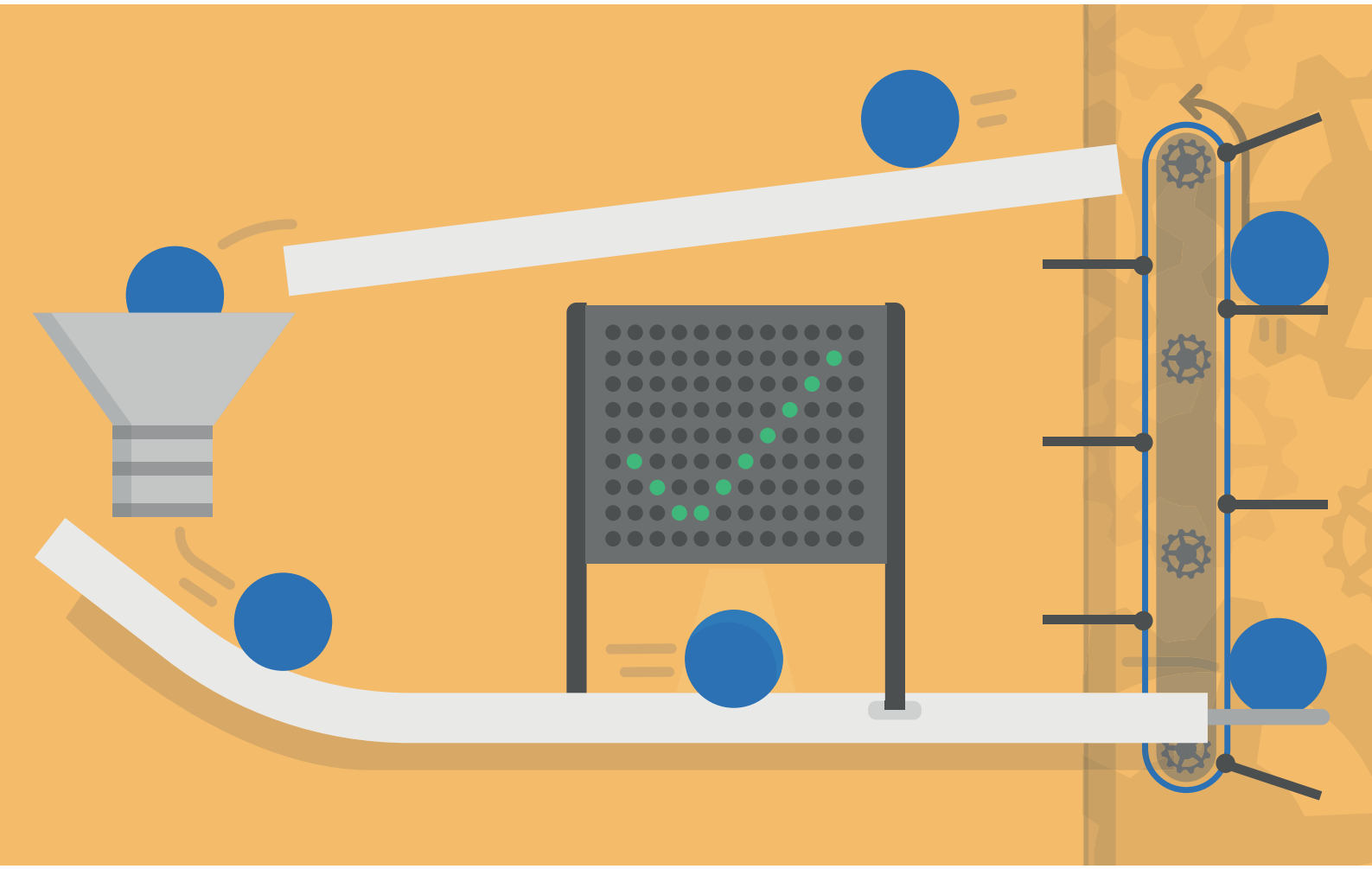
Test Case Analysis

Test Case #	Description	Time to run once manually, in minutes	Number of iterations per test cycle	Number of test cycles per month	Level of tester needed	Labor cost to execute once manually	Estimated time to automate, in ho	Level of automater need	Labor cost to automate	Savings/cost for first execution	Labor cost savings per cy	Labor cost savings per month	Execution time savings, first month, in hours	Execu saving mont
1	Login	5	12	4	Manual Tester 1	1,25	1,5	Technical Tester	112,50	(111,25)	15,00	60,00	0,50	
2	Query database	10	18	4	Manual Tester 2	4,17	3,0	Automation Engineer	300,00	(295,83)	75,00	300,00	3,00	
3					Manual Tester 1	-		Automation Engineer	-	-	-	-	-	
4					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
5					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
6					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
7					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
8					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
9					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
10					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
11					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
12					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
13					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
14					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
15					Manual Tester 1	-		Technical Tester	-	-	-	-	-	
TOTAL		15	30	8		5,42	4,5		412,50	(407,08)	90,00	360,00	3,5	

ROI calculator

Bereit

110%



2. Start with regression tests

In an application release cycle, QA personnel typically perform the following three types of tests:

- **Release-specific** tests, to verify new features of the application under test (AUT).
- **Defect-fix verification** tests, to ensure that a reported defect has been resolved.
- **Regression** tests, which are test cases that the AUT passed in a previous release cycle. Regression tests check that new defects have not been introduced, and old ones have not re-occurred. This includes functional regressions, or failures of the system to perform as expected, and visual regressions, which are unanticipated changes in the appearance of an application.

Best practice #1 lists the key characteristics of test cases that have the best return on your automation investment. These include test cases that will be executed repeatedly and test cases for features that are relatively stable. Regression tests meet both of these considerations, so they should be among the first test cases that you automate. Following are some of the best practices in automating regression test cases.

Prioritize high-value regression test cases

Not all regression test cases have equal importance. Focus your automation efforts on the following types of regression tests:

Smoke tests

These tests ensure that basic functionality is working after a new build of the AUT. Smoke tests check that the application opens and performs tasks such login, display the welcome screen, fetch new data, and so forth. As a best practice, automate your smoke tests and trigger them automatically for each new build of the application so that you know you have a good/“green” build before investing resources in further testing.

Sanity tests

These tests deeply test the most critical functions of your application. If your AUT is a web shopping application, a sanity test would ensure that a user could log on, search for an item, add the item to a cart, and check out. Plan to include all high-priority functions, any functions or modules that have changed, and highly-trafficked workflows in your sanity tests.

Successful test cases

A successful test case is one that has uncovered a large number of defects in past release cycles. Include these in your regression suite to increase your odds of uncovering a new defect – or an old one that has been re-introduced.

Make your regression tests maintainable

Certain practices will make your automated tests more maintainable. These practices apply to all types of automated tests, not just regression tests. Key among these is to set up your automated tests to be as modular and independent of each other as possible. Don't copy-and-paste code between test cases, as this will multiply your maintenance requirements. Instead, if you are going to reuse code, such as a login procedure, create it as an independent module and then re-use it. If your login procedure changes, you will only have one module to update rather than dozens or more.

Another best practice is to keep the definition of your UI elements separate from the action steps in each test case. Also, don't hard-code your test data, but instead maintain it in a spreadsheet or database. Read more about the topic of maintainability in section 3.

Run a full regression suite only when necessary

It is not always necessary to execute your full regression suite for each new build. For a minor release, it may make more sense to run just your smoke tests, plus regression tests for any modules that have changed. To make this easier, organize your regression test cases according to the module of the AUT covered by each test. For example, if a release includes a change to the payment types accepted for an online store, it may only be necessary to run your regression tests for the payment process, but exclude regression tests for other features such as searching for items and placing them in the cart. On the other hand, it may make sense to run the complete regression suite when a release cycle includes changes to many areas of the application, such as localization for a new language. To learn more about prioritizing your regression test cases for a particular release cycle, refer to the Ranorex Regression Testing Guide.

Leverage your development toolchain

An automated regression test is code, so treat it like code, and take advantage of your existing development environment.

Use source control

The purpose of a source control system like Git or Subversion is to allow multiple developers to work on the same application at the same time without overwriting each other's changes. Source control also makes it possible to maintain the correct version of a given regression test with the corresponding version of the application.

Integrate with a CI process

If your development team is following a continuous integration process, integrate your regression tests with your CI server. This will allow you to automatically trigger your regression tests for each system build.

Integrate with a defect-tracking application

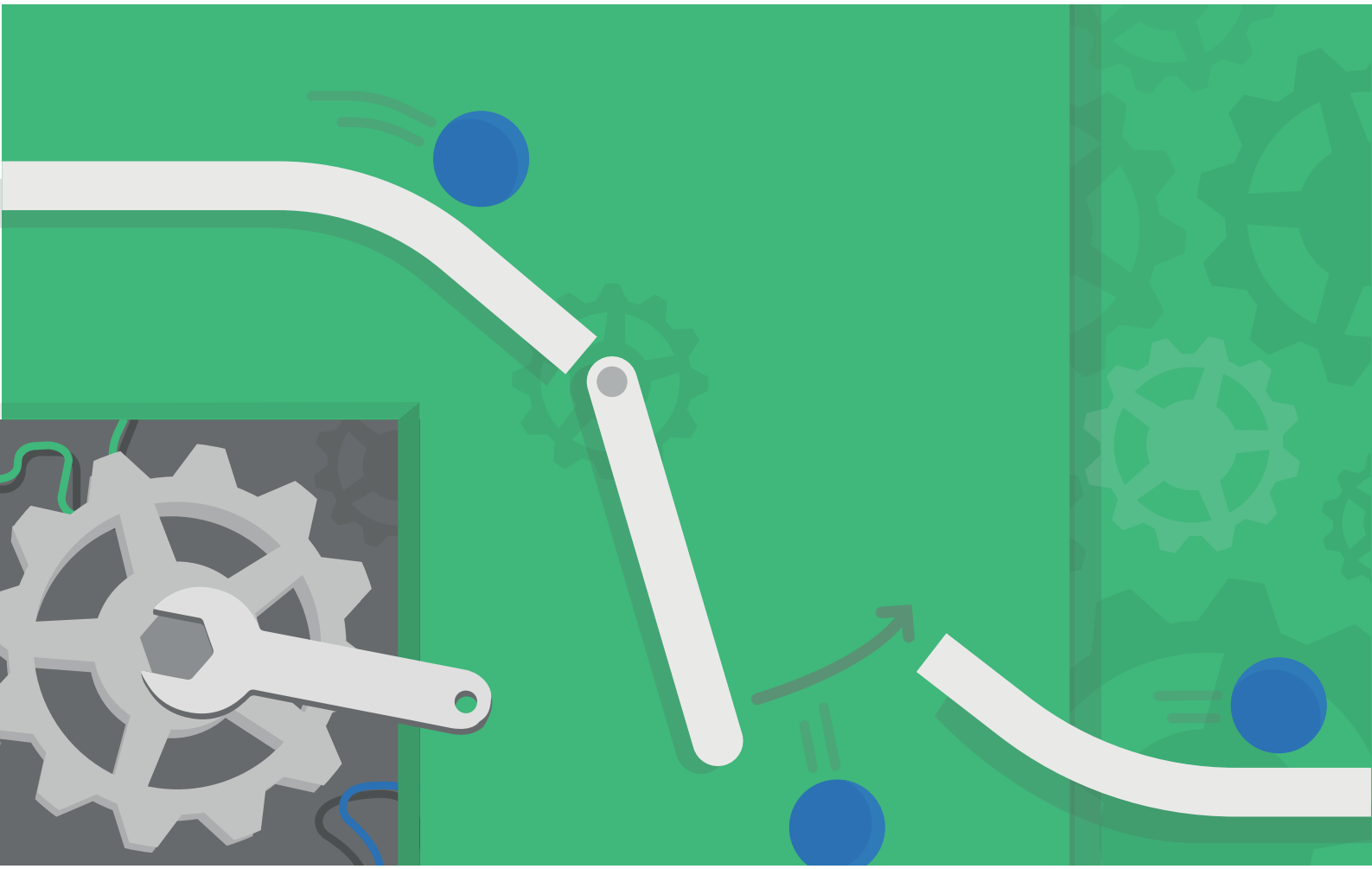
Defect-tracking tools like JIRA and Bugzilla are essential for reporting defects and tracking them through to resolution. Configure your automated regression tests to report defects automatically. Also, use the defect-tracking process to document each defect found in manual testing and the steps to reproduce it. This documentation will provide candidates for new regression test cases in the next development cycle.

Manage the size of your regression suite

With each release cycle, you will likely add a number of new regression test cases. Over time, this can cause the regression suite to become large and require a lot of resources to execute. Keeping an increasing number of regression tests up-to-date with the changing application could become a burden. To prevent this, keep the size of your regression suite manageable. Each release cycle, remove test cases that don't provide value for the testing process, such as tests for obsolete features, or low-priority tests that the AUT consistently passes. Carefully review new test cases for their ability to add value to the testing process, such as ones that were successful in uncovering defects in the previous release cycle, or that test critical new functionality.

Be aware of the limits of regression testing

Like a well-traveled path through a minefield, your regression suite may execute perfectly but miss new defects. Remember that the purpose of regression testing is to ensure that code changes haven't reintroduced old defects or caused new defects in previously-working code. The fact that your AUT has passed all of its regression tests tells you nothing about the quality of new functionality. One of the key benefits of automating your regression tests is that manual testers can have more time to focus on exploratory testing of new features and ensuring a great user experience.



3. Build maintainable tests

One of the top challenges in test automation is maintaining existing automated tests due to changes in the UI. Another challenge is identifying and improving flaky tests – those tests that work sometimes, and fail other times for reasons unrelated to the AUT. This section describes approaches to test case design, coding, and execution that help manage these challenges and reduce time spent in test maintenance.

Design tips to minimize maintenance

Decide what to test before you decide how to test it

A well-designed test case is less likely to need future maintenance. Begin by identifying each function to be tested, and then break down each test for that function into a sequence of simple steps with a clear definition of the expected results. Only after this is complete you should decide which tests will be done manually, and which will benefit from automation. Refer to section 1 for suggestions on the best types of test cases to automate.

Document your test cases

Documentation can help ensure that each test case has been well-designed with preconditions for the test, steps to execute, and expected results. It can be very helpful to use a test case template or a tool such as TestRail to organize your test case documentation.

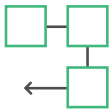
Keep it simple

Ideally, each test case should check a single function and should fail for only one reason. Complex test cases are more likely to be flaky. If you find that a test case requires many steps, consider dividing it into two or more test cases.

Use naming standards

The names of UI elements and test objects should be self-explanatory. If you find that comments are necessary to document a given test case or test step, consider whether the test case is too complex and needs to be simplified.

Coding tips to minimize maintenance



Use a modular structure

A given test case should be able to run independently of other test cases. As much as possible, a test case should not be dependent on the outcome of an earlier test. For example, a test case that verifies payment processing for a web store should not be dependent on a previous test case that puts items in a user's shopping cart. Keeping your test cases independent will not only make your tests easier to maintain, but will also allow you to take advantage of parallel or distributed execution. However, if you do find that a test case dependency is unavoidable, then use the available features of your test automation framework to ensure that the dependent tests are executed in the proper order.

Create automated tests that are resistant to UI changes

To keep your automated tests working even when there are changes in the user interface, don't rely on location coordinates to find a UI object. In addition, if you are testing a web application, avoid relying on the HTML structure of the page or dynamic IDs to locate UI elements. Ideally, the developers of your AUT will include unique IDs for the UI elements in your application. But even if they don't, the Ranorex Spy tool automatically applies best practices in UI element identification to help make your tests more stable.

Group tests by functional area

Group your test cases according to the functional area of the application covered. This will make it easier to update related test cases when a functional area is modified, and also allow you to execute a partial regression suite for that functional area. If you are using Ranorex Studio, you can create reusable user code methods and store them in a central library. Then, you can organize the user code methods for a functional area into a collection. Both user code methods and collections can have a description that appears in the user code library, to help testers select the right user code method for a test.

Don‘t copy-and-paste test code

Instead of repeating the same steps in multiple tests, create reusable modules. For example, you should only have one module that launches your application. Reuse that module in your other test cases. Then, if the process to launch the application changes, you will only need to update that one module. With Ranorex Studio’s support for keyword-driven testing, local and global parameters, and conditional test execution, you can easily build sophisticated test cases from your individual test modules.

Separate test steps from test data

Avoid hard-coding data values into your automated tests. Instead, store data values for your tests in an external file and pass it to your tests using variables or parameters. Read more about data-driven testing in the Ranorex Studio User Guide.

Use source control

Developers use source control tools such as Git, Subversion and Microsoft TFS to collaborate on application code and to revert to earlier versions of the application code when necessary. If possible, you should use the same source control tool that manages your application code to manage the code for your automated tests.

Execution tips to minimize maintenance

Ensure that your test environment is stable

Unreliable servers or network connections can cause otherwise stable tests to fail. Consider using a mock server to eliminate potential points of failure that are not related to the AUT itself.

Use setup and teardown processes

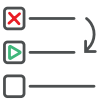
Use setup processes to handle preconditions and ensure that the AUT is in the correct state for the tests to run. A setup process will typically handle launching the application, logging in, loading test data, and any other preparation necessary for the test. Use teardown processes to return the AUT to the proper state after the test run completes, including cleaning up any test data.

Fail fast

Another key principle of efficient test design is to “fail fast.” If there is a serious issue with the application that should stop testing, identify and report that issue immediately rather than allowing the test run to continue. Set reasonable timeout values to limit the time that your test spends searching for UI elements.

Fail only when necessary

Allow your entire test run to fail only when necessary. Stopping a test run after a single error potentially wastes time, and leaves you with no way of knowing whether the other test cases in the run would have succeeded. So, in addition to giving you the ability to stop a test run after an error, Ranorex Studio offers three options for continuing after an error: continue with iteration, continue with sibling, and continue with parent. Read more about these options in the Ranorex Studio User Guide.



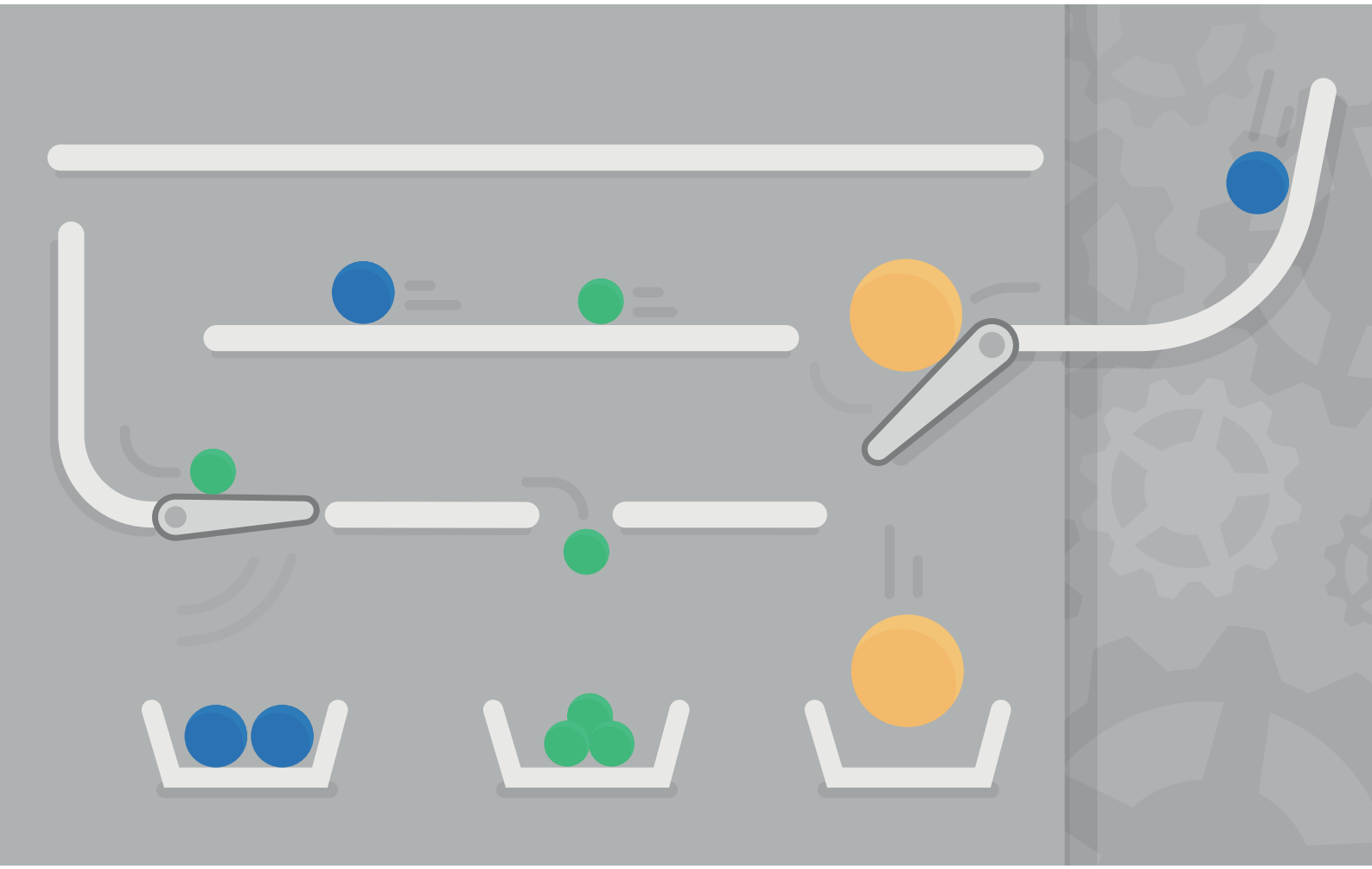
Isolate expected failures

Execute only the automated tests that you expect to succeed. If you have tests for a defect that hasn’t been resolved, remove those test cases from your main test run and execute them separately. This will make it easier to determine if there are real issues in the main test run. Likewise, remove any flaky tests from the main test run, and perform manual testing to cover that functionality.

Take screenshots

Configure your automated tests to capture screenshots and use your reporting mechanism to provide detailed information that will assist in troubleshooting a failed test. Ranorex Studio includes a maintenance mode that allows you to pause a test run so that you can diagnose and resolve errors directly during the test run. To see this in action, watch our screencast Maintenance Mode.

Following these tips will help you build maintainable test cases, so that you only need to modify the minimum possible number of existing test cases when the application changes. Building maintainable test cases also increases stability and makes debugging easier.



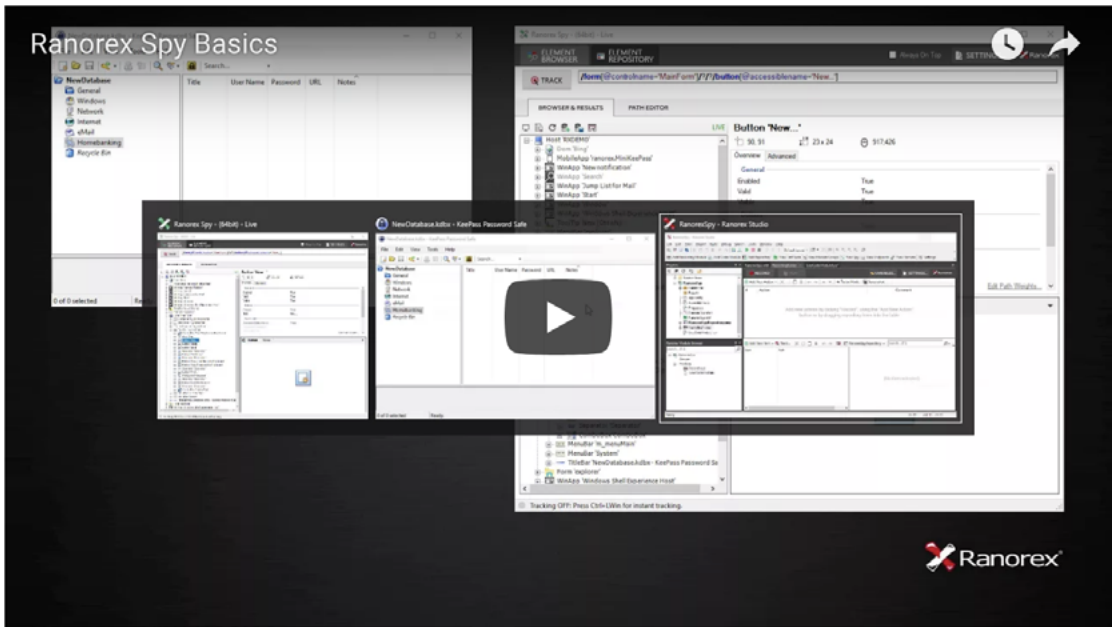
4. Use reliable locators

If not properly designed, user interface (UI) tests can be slow and prone to failure, or “fragile”. But your tests can be stable, even when the UI changes. One of the most important factors in designing a UI test for stability is the method used to identify UI elements including text, form fields, buttons, scrollbars, and other controls.

Understanding the basics

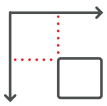
A typical UI (whether for desktop, mobile, or web application) is organized into series of containers nested one inside the other. To locate a given UI object, an automated test uses attributes such as the object ID, class, name, and the path to locating it within its container. Some locators are inherently more stable than others.

Watch the video linked below for a demonstration of identifying UI objects in Ranorex Spy. As you watch, notice that Ranorex Spy includes wildcards (*/*) in the path to an object, which increases the reliability of the selector. If there is a change in the hierarchy of the application, the automated test will still be able to find the UI element.



Building stable locators

The following principles help ensure that your automated tests can find your UI objects reliably.



Avoid coordinate-based recognition

Coordinate-based recognition locates a UI object based on its (X,Y) grid coordinates plus a length value. For this to work, the object must always be at the same spot on the screen. Given that any change to the layout of the UI would break coordinate-based recognition, this approach was fragile even on legacy applications designed for fixed-size screens. It's even more unstable in today's environment of varied screen sizes, resolutions and orientations.

Avoid image-based recognition

In image-based recognition, an automated test does a pixel-by-pixel comparison of a stored image to the screen image. If the image of the desired UI element is found, then the coordinate value of its location is returned to the automated test. Using image-based recognition to locate a specific UI element is subject to the same inherent weakness as coordinate-based recognition and should be avoided, if possible. On the other hand, you can certainly do image-based comparison in an automated regression test, because in that case, you are using the image itself as a UI object rather than using it to find another object.

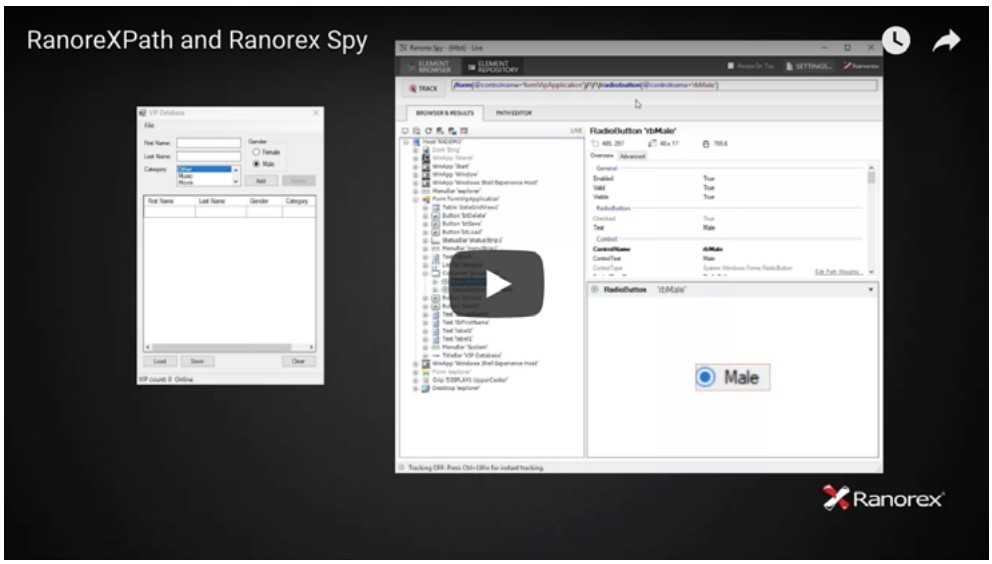
Avoid using dynamic IDs

A dynamic ID is one that changes each time the application or web page is loaded. Dynamic IDs are often associated with dynamic content, such as a web page that displays the current date and time. The ideal approach is to create a permanent, unique ID for each UI element in your application and use that to locate objects. If that's not possible, use another unique attribute such as the object name (unless that is also dynamic), link, CSS or XPath. Ranorex Studio supports the use of weight rules to filter out an object's dynamic ID and use another property for object identification, along with conditions to control when the weight rule should apply. To learn more, refer to the Ranorex blog article Automated Testing and Dynamic IDs.

Use the shortest path.

Keep the path for object recognition as short as possible, balancing the twin goals of fast and stable object recognition. For example, if you are creating an automated test for a web page, use the closest relative object such as the immediate parent or child of the UI element. The exception to this rule is if the parent or child object does not have a unique ID, but the grandparent object does, then use the grandparent object instead for increased stability.

The video linked below demonstrates object identification using the RanoreXpath. The RanoreXpath is based on the XPath query language for selecting nodes from an XML document, but includes capabilities not available in the XPath WC3 standard.



Additional best practices

The following practices will help make your UI tests more resilient and reduce maintenance.

Use an object repository

Using a repository to manage your UI objects allows you to separate the definition of a UI element from your test automation code. Then, if you must change the selector for a UI element, you only need to make the change once for every test case that references the object. A shareable object repository can also enhance collaboration between testers and developers.

Get your developers involved.

As mentioned earlier, having a unique ID for each UI element makes test automation much simpler. In addition, developers can assist the testing effort by creating log files to record application behavior, as well as other indications of the application's status such as meaningful error messages, status bars and breadcrumbs.

Troubleshooting object recognition

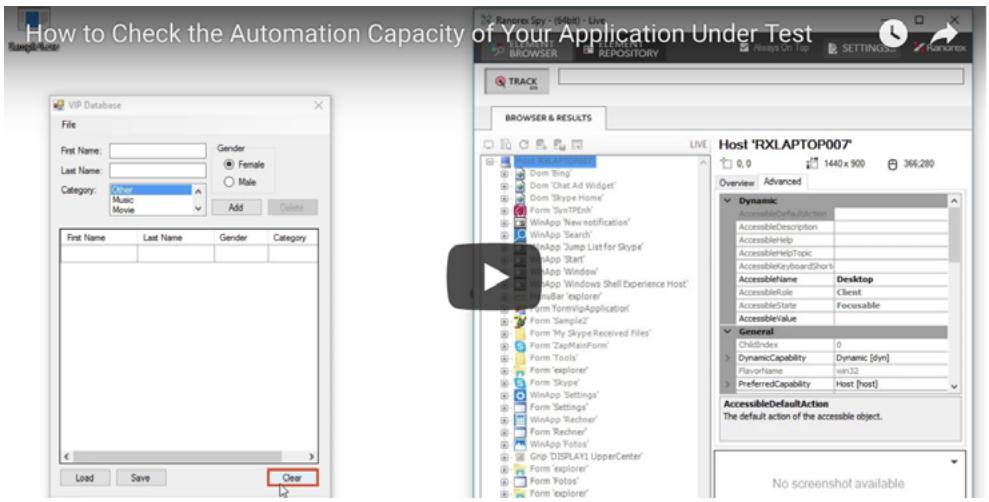
Even when using stable locators, object recognition can sometimes fail. Below are tips to help you resolve issues with object recognition.

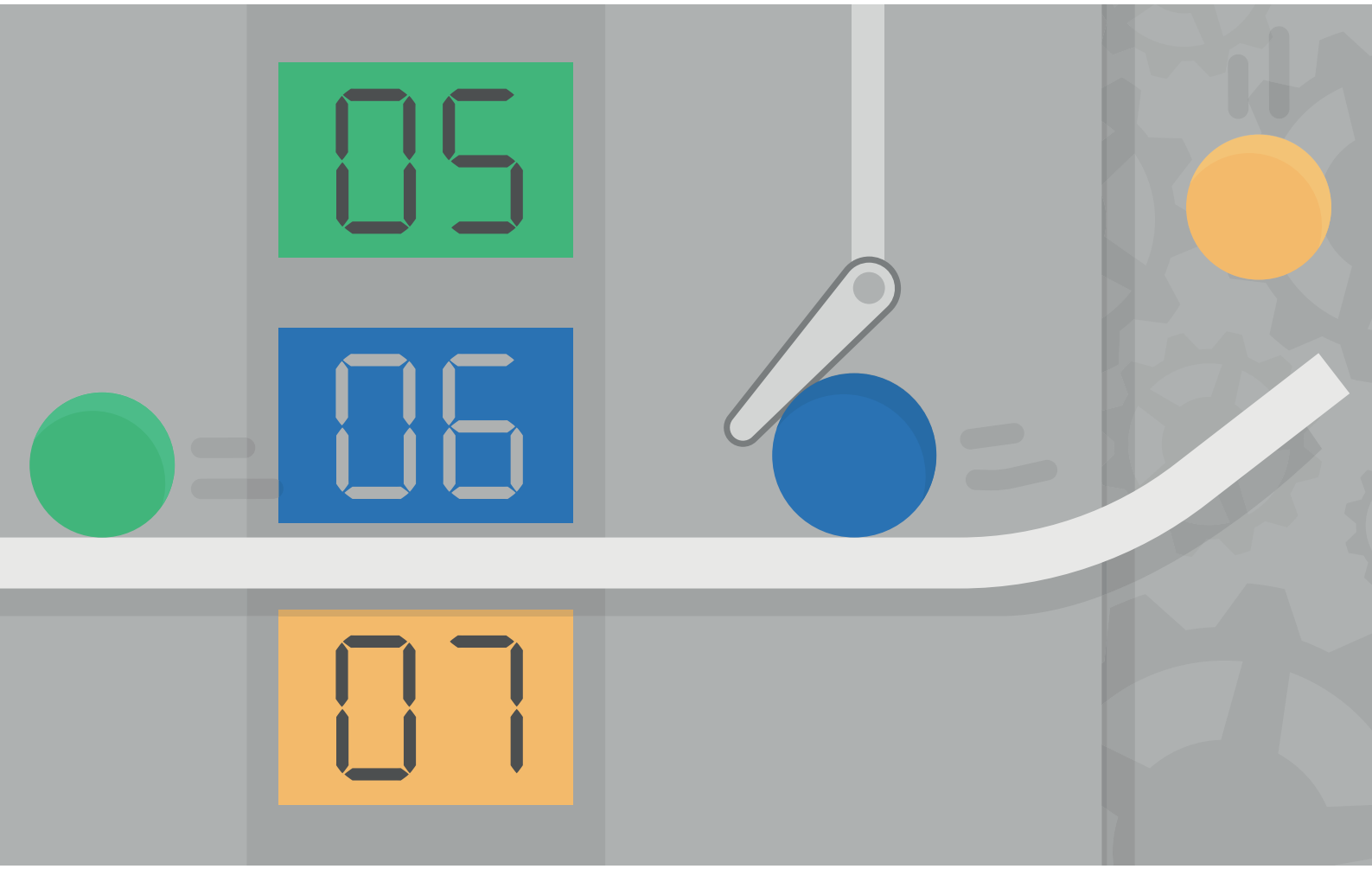
Invisible UI elements

Elements such as drop-down menus, pop-up windows and combo boxes may become visible only after a mouse-click, and disappear when the application loses focus. To handle this, make sure your automated test includes the steps that a user would normally perform to bring focus to the desired UI element. It may also be necessary to include a wait time in your automated test, and delay execution until the desired element becomes visible.

Snapshots

If you are an existing Ranorex customer, or are evaluating Ranorex, you can generate snapshots to share with our support team for assistance in identifying UI objects. The video linked below demonstrates how to check the ability of Ranorex Studio to identify the UI elements in your application





5. Conduct data-driven testing

The ability to do data-driven testing is one of the key benefits of test automation. In data-driven testing, an automated test case retrieves input values from a data source such as an Excel spreadsheet or a database file. The test case is repeated automatically for each row of data in the data source. So, instead of 10 testers having to manually execute test cases for 10 different data values, and determine whether or not each test case succeeded, an automated test can execute 100 test cases in a single test run.

In a test case for a user registration process, the data source might contain the columns and rows like those shown below:

First Name	Last Name	Desired User Name	Desired Password	Expected Result
John	Jones	jjones14	Mypass123	Pass
Jody	Jones	jjones14	Mypass456	Fail
Julia	Jones	jjones15	mypass	Fail

If the external source also contains validation values, then the data-driven test can compare the results of the test to the validation value to determine whether the test case passes. For example, for a test of the “multiply” function in a calculator application, the data table might look something like the following:

Input 1	Input 2	Expected Result
12	12	144
15	50	750
-1	18	-18

In both examples, the actual result of the test can be compared to the expected result to determine whether or not the test case succeeded.

Benefits of data-driven testing

Reduced execution time

One of the most obvious benefits of data-driven testing is that automation makes it possible to rapidly execute a large volume of test cases, especially repetitive ones that cover positive and negative test cases, or corner, edge and boundary cases.

Increased accuracy

Even the most careful tester can make errors when manually entering large amounts of data. With data-driven testing, you can be certain that the exact data values specified in the Excel spreadsheet or database are used to execute the test case.

Improved use of system and human resources

Automated data-driven test cases can execute at night, when test servers would otherwise be idle. With manual testers freed from entering repetitive test data, they can focus on more challenging exploratory and user experience testing.

Reduced test case maintenance

Separation of test data from test code reduces maintenance in several ways. First, you can easily add or remove test cases by changing the test data without affecting the test code itself. Well-designed data driven tests with error handling and conditional execution can also reduce the need for redundant test cases. For example, instead of having separate test cases to check different types of valid and invalid passwords for a welcome screen, with data-driven testing you have just a single test case to maintain.

Better test data storage

Data-driven testing allows you to store test data in a central repository, whether that's an Excel spreadsheet, CSV file or database file. This makes the data easier to share, re-use, backup, and maintain.

Supports more than just testing

In addition to validating that your application works as expected, data-driven tests can also be used to simulate data entry for load and performance testing. It's also possible to use a data-driven test case to populate your production database.

Best practices for data-driven testing

Separate test data from the test code whenever possible

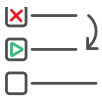
Use data tables to provide input and validation values for your automated tests rather than storing data values directly in your test cases. Hard-coding values makes your test cases more difficult to maintain and also more difficult for another tester to read. For the same reason, you should use external data tables to supply values for test environment settings such as system variables. Read more about how to use parameters to set system variables for Ranorex Studio test runs in the Ranorex Studio User Guide.

Use realistic data

Ensure that the data in your data table provides adequate coverage of your test scenarios. Include both positive data values that should succeed, and negative data values that should return an error. Boundary value analysis can be helpful in producing data to ensure coverage of both positive and negative test cases. For example, if a number field accepts values only between 1 and 100, your positive test cases would include 1, 2, 99 and 100. It's not necessary to test all of the numbers in between if the test case succeeds for these numbers. Negative test cases would include -1, 0, 101 and 102. Another approach to data-driven testing is to execute test cases against a subset of your production data, which helps ensure that your tests cover the types of data that the application actually processes.

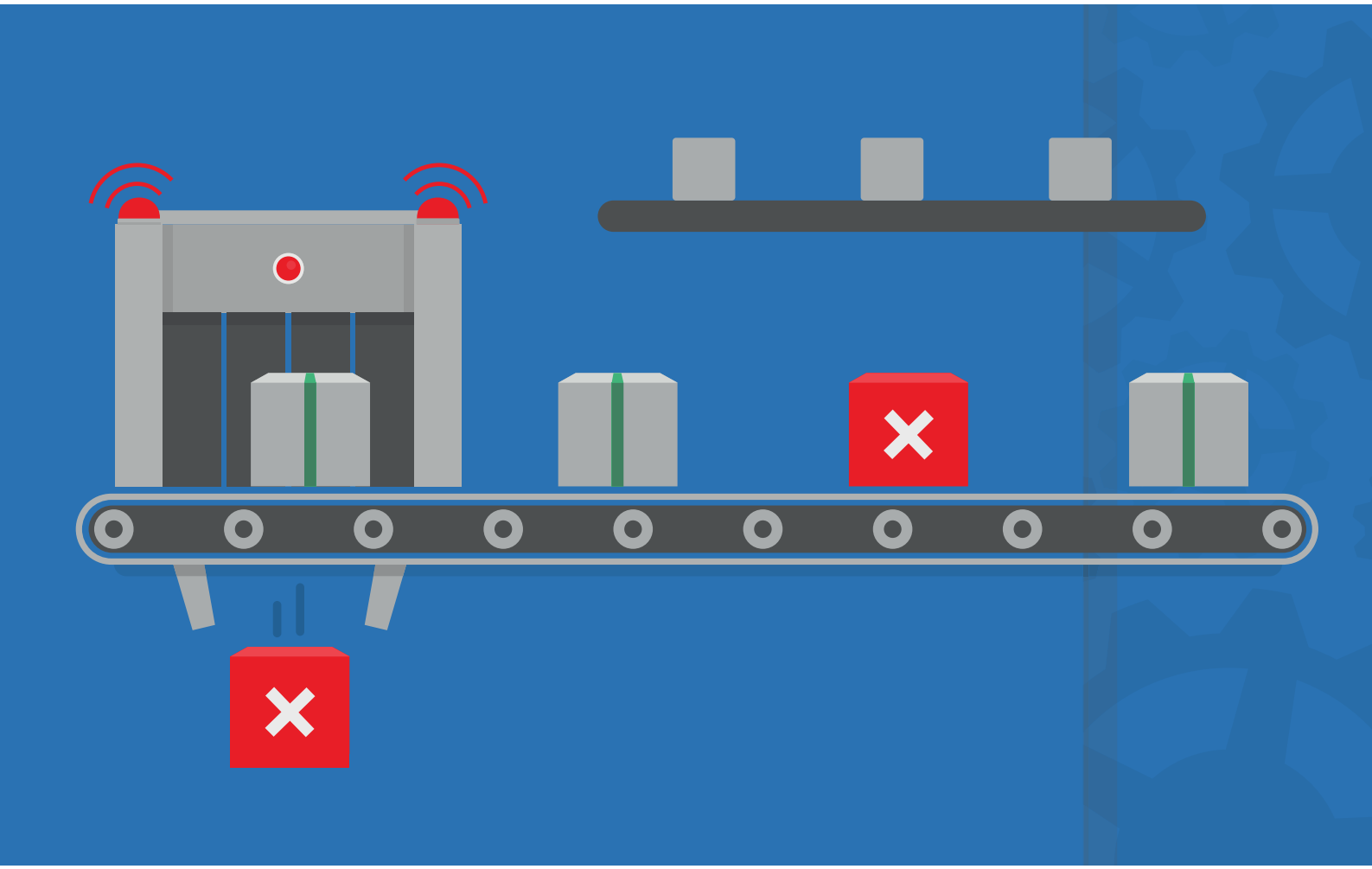
Use setup/teardown modules

Each test case should configure the test environment that it needs, including test data, and clean up afterward. So, if your test case reads a number of rows from an Excel spreadsheet and inserts them into your application, the test case should include a teardown step to delete the records that it created. This practice keeps your test cases independent of each other and increases the chances that the entire test run will succeed.



Configure error handling

Include logic in your data-driven test to determine what should happen if one test case fails. For example, Ranorex supports the following options for handling a validation error: “continue with iteration” which will continue the test case with the next row of test data, “continue with sibling” which will end the current test case and continue with the next one at the same level, “continue with parent” which will end the current test case and continue with the next parent test case, and “stop” to abort the test run entirely.



6. Resolve failing test cases

Even when test cases have been carefully designed to be stable and maintainable, test failures can happen. There are several possible uses of the term “test failure,” so let’s distinguish between them:

A negative test case

This is a test case that you expect to return an error from the application under test (AUT), such as an invalid password. This type of test case succeeds when it returns the expected error message.

A test case that uncovers a defect in the application

This is actually a successful test case because identifying defects is one of the main goals of software testing. Consider this type of test case for inclusion in your regression test set.

A test case that fails for a reason unrelated to the functionality of the application

This is the meaning of the term “failed test case” as used in this section.

When a test case fails, the first job is to decide whether situation #2 or #3 applies: Did the test case fail due to a defect in the AUT, or is the problem with the test case itself, such as missing or invalid test data, problems with the test environment, or changes in the AUT that are not defects? If it is not immediately clear, you may need to troubleshoot the test case itself before reporting a defect in the application.

It may be tempting to simply re-run a failed test case to see if it passes. But a test case that passes sometimes and fails on other occasions for no discernable reason is a “flaky,” unreliable test case. It’s important to resolve the issue that caused it to fail so that you can have confidence in the results of your automated testing.

Configure test runs to assist debugging

Recommendation #3, “Build Maintainable Tests,” describes best practices for designing test cases that make them more stable and less likely to fail. These included eliminating dependencies between test cases as much as possible, ensuring that your test environment is stable, and removing tests that you expect to fail (such as ones for unresolved defects) from the test run. It is also helpful to configure your test cases to take a screenshot when a failure occurs.

In addition to these recommendations, be sure to configure the test run to handle failures appropriately. Only allow a failing test to stop the entire test run if that makes sense for the situation – for example, if the application fails to launch, or smoke tests fail. Ranorex Studio’s modular approach to test case design includes several options for continuing after a test case returns an error, including “continue with iteration,” “continue with sibling,” and “continue with parent.” You can also automatically retry a failed test case. To learn more, read the Ranorex Studio User Guide chapter on the Test Suite.

It’s also important to manage the size of test run reports by focusing only on true errors and failures. For example, Ranorex Studio supports multiple pre-defined report levels, including “debug,” “information,” “warning,” and “success.” In a large test run, reporting information at this level may result in an excessive amount of data. Consider reporting results only for the “error” and “failure” levels to make it easier to spot true problems that need to be resolved.

Isolate the problem

If many test cases are failing, look for a problem with the environment, test framework, or the AUT.

Environment

Issues with the environment can include required services not running, or not running in administrative mode if required.

Test Framework

Look for issues with the test framework, such as a licensing error, or a remote agent not configured properly.

Application Under Test

Verify that the AUT is prepared correctly. This can include issues such as location-specific system settings, the wrong browser version, or even a different system language. Or, there could be a pending O/S update that blocks the user interface.

If most test cases in your test run have succeeded, then suspect issues with the individual failing test case(s). There may be an error message that points to the cause. If not, don't just assume that the test case failed "accidentally" and re-run it. All test failures happen for a reason. **A test case that appears to succeed or fail for no discernable reason is a "flaky" test.** To get to the root of the problem, refer to the probable-cause checklist below.

Troubleshoot failed test cases

Work through a probable-cause checklist to troubleshoot each failed test case, asking questions such as the following:

- Is the test case up-to-date with the AUT? For example, has the test case been updated with any/all changes in UI elements?
- Is the input data correct and available to the test?
- Are all parameters set correctly?
- Are the expected results valid? Does the test case expect a single valid result, but the application returns multiple valid results?
- Does the test case have any dependencies on earlier test cases that might have caused the problem? To avoid this situation, make test cases as modular and independent of each other.
- Did the teardown of the most recent test run work correctly? Is the AUT in the correct state, for example, with all browser windows closed? Has all the data entered during the last test run been deleted or reset?
- Is there a timing issue? A study of flaky tests done by the University of Illinois at Urbana-Champaign found that flaky tests are often caused by asynchronous waits: the test fails because the AUT doesn't return the expected result fast enough. In this case, it may be

necessary to add a wait time to the test case step so that it doesn't fail unnecessarily. For more information on how this works in Ranorex Studio, refer to the description of the **Wait For** action in the Ranorex Studio User Guide.

Use your debugging tools

Make use of the tools available to you that may help resolve failing test cases. For example, Ranorex Studio provides several tools to assist in troubleshooting failed test cases, including the following:

Debugger

This tool allows you to set breakpoints and step through a failed test case, examining the value of variables and expressions for each statement.

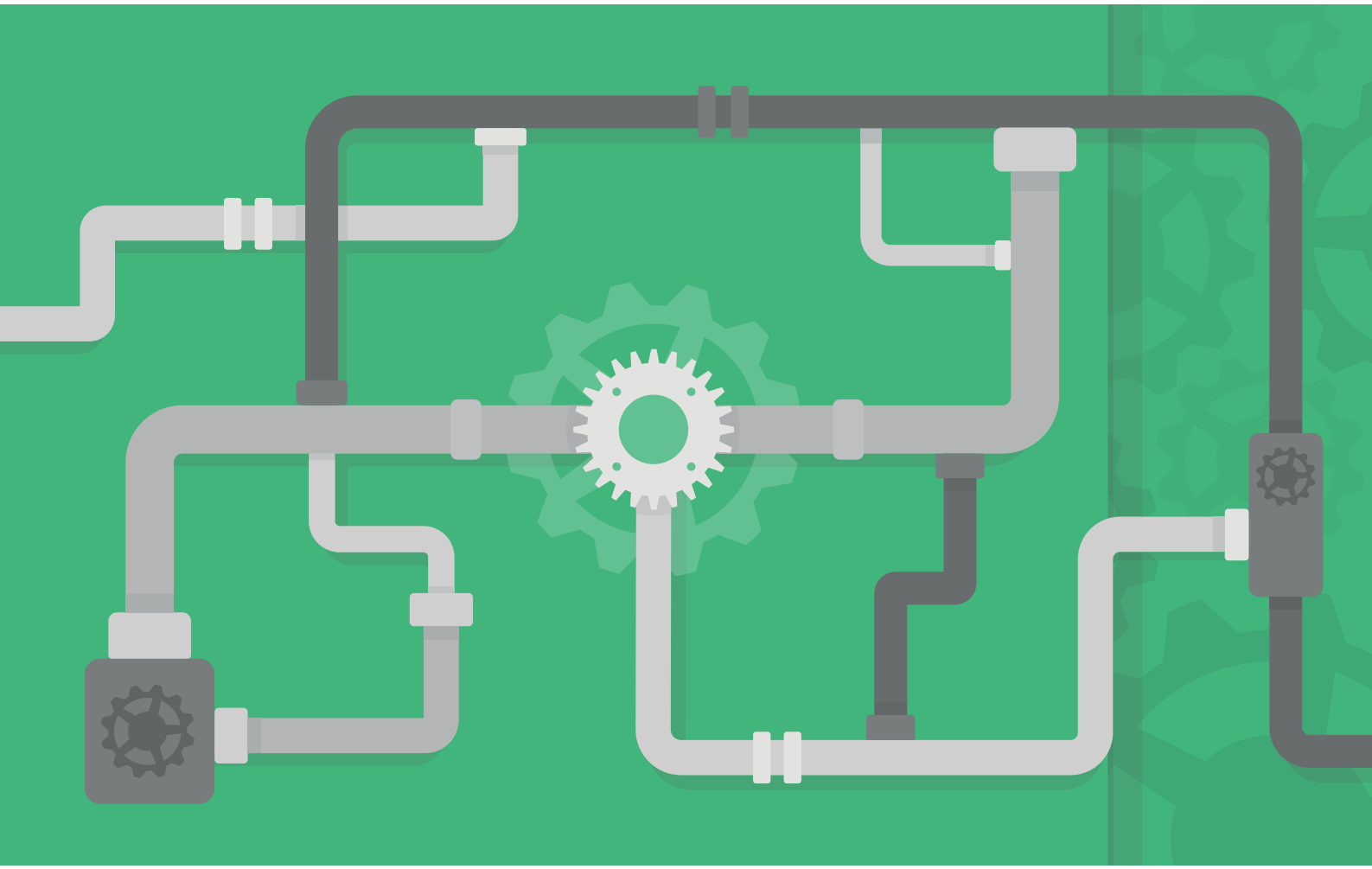
Maintenance Mode

This tool allows you to identify and repair failing test cases directly from the test run report. Learn more in the following Ranorex blog article: Maintenance mode.

Ranorex Remote

This is a great tool for troubleshooting test failures that occur on virtual machines. Use the Ranorex Remote Agent to update a run configuration to perform only the steps necessary to reach the point just before the failure occurred, so that the AUT is in the correct state. Then, connect to the virtual machine and troubleshoot the failed test case, as described in the blog article How to Reconstruct Failed Test Cases in CI Systems.

Taking the time to resolve your failed test cases, and to learn from the failures, will help make your entire test suite more reliable.

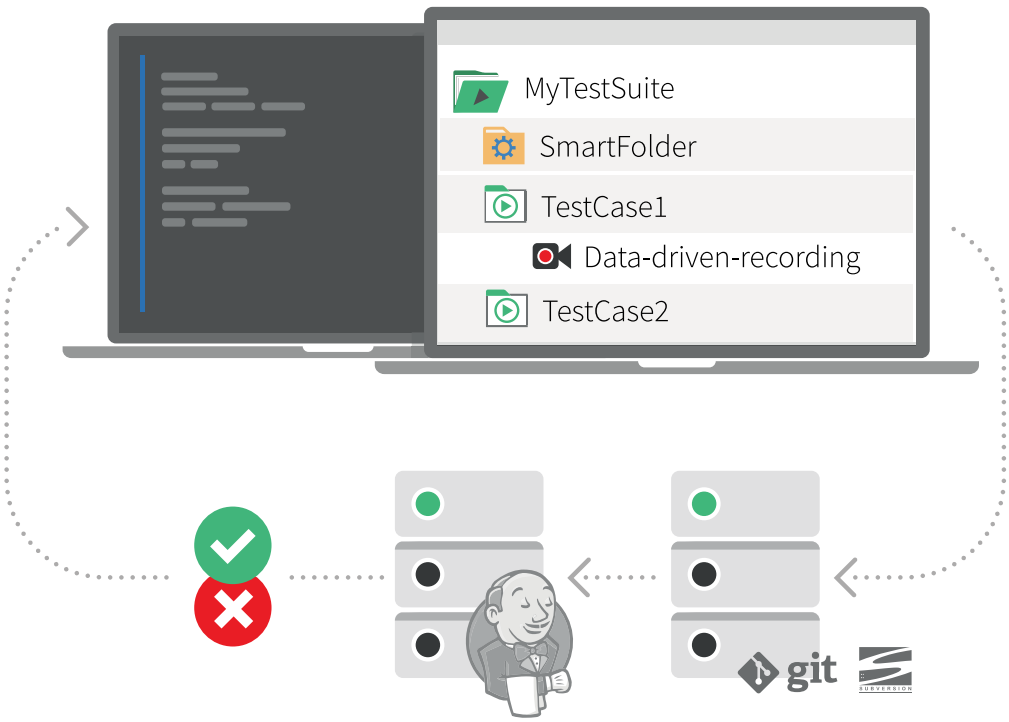


7. Integrate with a CI pipeline

Rapid application development practices such as Continuous Integration (CI), Continuous Deployment (CD), and DevOps have a common goal: small, frequent releases of high-quality, "working" software. Whether your development cycle is measured in weeks or days, integrated automated tests are essential to maintaining the pace of development.

Automated tests in a CI pipeline

The image below shows a typical CI pipeline. A developer checks out code from the shared repository in the version control system, such as Git, TFS or Subversion. Once code changes are complete, the developer commits the change back to the version control system, triggering a CI job. The CI server builds the application under test and triggers automated tests to verify whether the new code results in a good, "green" build. The results of testing are reported back to the entire team for a decision regarding deployment of the application. In a CD environment, the application is deployed automatically to the production environment.



Continuous integration with automated testing offers several benefits to organizations, including the following:

- Developers get fast feedback on the quality, functionality, or system-wide impact of their code changes, when defects are easier and less expensive to fix.
- Frequent integration of small changes reduces the volume of merge conflicts that can occur when several developers are working on the same application code, and makes merge conflicts easier to resolve when they do happen.
- Everyone on the team has a clear understanding of the status of the build.
- A current “good build” of the application is always available for testing, demonstration, or release.

Recommendations for automated testing in a CI pipeline

The recommendations below focus on **test automation** in a CI pipeline, some of which overlap the best practices for the CI process itself.

Don‘t rely solely on unit tests

Unit testing in an individual developer’s local environment doesn’t tell you enough about how that code will work once it is introduced to the production application. Integration of new or revised code may cause a build to fail for several reasons. For example, changes made by another developer may conflict with the new code, or there may be differences between the developer’s local environment and the production environment. Therefore, it’s important to run integration tests, regression tests and high-priority functional UI tests as part of the build verification process.

Make your build self-testing

Component testing checks individual units of code. Component testing is often called unit testing, but may also be called **module testing** or **program testing**. Developers write and execute unit tests to find and fix defects in their code as

early as possible in the development process. This is critical in agile development environments, where short release cycles require fast test feedback. Unit tests are white-box tests because they are written with a knowledge of the code being checked.

Refactor your automated tests

A build that takes a long time to complete disrupts the CI process. To keep your testing efficient, approach your automated testing code as you would the application code itself. Regularly look for redundancies that can be eliminated, such as multiple tests that cover the same feature or data-driven tests that use repetitive data values. Techniques such as boundary value analysis and equivalence partitioning can help reduce your data-driven testing to just the essential cases.

Keep your build fast

It's essential that build tests complete as quickly as possible so that developers aren't discouraged from committing frequently. To keep the process fast, trigger the minimum automated tests required to validate your build. Due to their more complex nature, integration tests are usually slower than unit tests. Run your smoke and sanity tests first to rapidly identify a broken build before spending time on additional tests. If your team merges frequently, it may be more efficient to run integration tests only for daily builds rather than every merge. Run a full regression only when necessary, such as in preparation for deployment to the production system. For example, Ranorex Studio supports the use of run configurations to run a subset of tests, such as smoke tests or tests just for features or modules of the application that have changed. Exclude from the build test set any low-priority regression test cases that haven't found a defect in several test cycles.

Use source control for your automated tests

Maintain your automated tests in the same repository as your code. This will make it easier to match the correct version of a test to the version of the source code. Ranorex Studio integrates with popular solutions for source control including Git, Microsoft Team Foundation Server, and Subversion.

Test in the right environment

To minimize the chance of test failures due to issues such as incorrect O/S version or missing services, test in an environment that is stable. Ideally, you will have an isolated test platform that is dedicated solely to testing. Your test environment should also be as identical as possible to the production environment, but this can be challenging. Realistically, it may be necessary to virtualize certain dependencies such as third-party applications. In complex environments, a virtualization platform or solution such as Docker containers may be an efficient approach to replicating the production environment.

Test in parallel

Speed is essential in a CI/CD environment. Save time by distributing your automated tests on a Selenium grid or running them in parallel on multiple physical or virtual servers. As mentioned earlier in this series, keep your automated tests as modular and independent of each other as possible so that you can test in parallel.

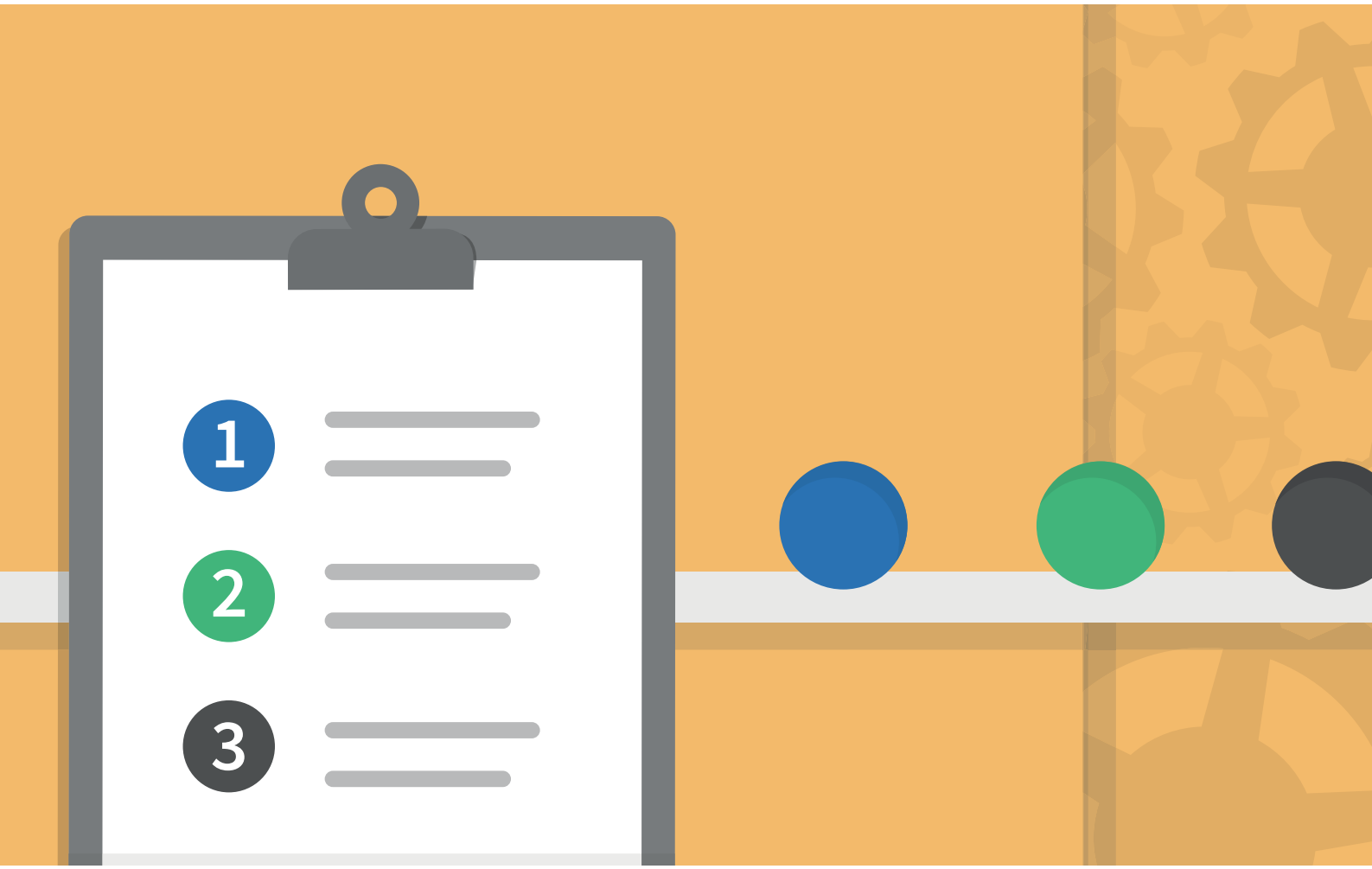
Include functional UI and exploratory testing

It takes a combination of automated testing approaches to confirm that your application is ready for deployment to the production environment. In addition to your automated unit and integration tests, include automated user interface tests to verify core functionality, check common user paths through the application end-to-end and validate complex workflows. Exploratory testing can uncover defects that automated tests miss.

Verify your deployment

After deploying the new build to the production environment, run your smoke tests in the production environment as a quick check to ensure a successful deployment.

To learn more about how to integrate Ranorex Studio tests in your CI pipeline, read our blog article Integrate Automated Testing into Jenkins. While this section focuses on Jenkins, Ranorex tests can be triggered from any CI server process.



8. Write testable requirements

A testable requirement describes a single function or behavior of an application in a way that makes it possible to develop tests to determine whether the requirement has been met. To be testable, a requirement must be clear, measurable, and complete, without any ambiguity.

Principles of Testable Requirements

Assume that you are planning to test a web shopping application. You are presented with the following requirement: "*Easy-to-use search for available inventory*." Testing this requirement as written requires assumptions about what is meant by ambiguous terms such as "easy-to-use" and "available inventory." To make requirements more testable, **clarify ambiguous wording** such as "fast," "intuitive" or "user-friendly."

Requirements shouldn't contain implementation details such as "the search box will be located in the top right corner of the screen," but otherwise should be **measurable and complete**. Consider the following example for a web shopping platform:

> *"When at least one matching item is found, display up to 20 matching inventory items, in a grid or list and using the sort order according to the user preference settings."*

This requirement provides details that lead to the creation of tests for boundary cases, such as no matching items, 1 or 2 matching items, and 19, 20 and 21 matching items. However, this requirement describes more than one function. It would be better practice to separate it into three separate requirements, as shown below:

- *When at least one matching item is found, display up to 20 matching inventory items*
- *Display search results in a grid or list according to the user preference settings*
- *Display search results in the sort order according to the user preference settings*

The principle of **one function per requirement** increases agility. In theory, it would be possible to release the search function itself in one sprint, with the addition of the ability to choose a grid/list display or a sort order in subsequent sprints.

Testable requirements **should not include** the following:

- Text that is irrelevant. Just as you can't judge a book by the number of words, length by itself is not a sign of a testable requirement. Remove anything that doesn't add to your understanding of the requirement.
- A description of the problem rather than the function that solves it.
- Implementation details. For implementation details such as font size, color, and placement, consider creating a set of standards that apply to the entire project rather than repeating the standards in each individual requirement.
- Ambiguity. Specifications should be specific. Avoid subjective terms that can't be measured, such as "usually." Replace these with objective, measurable terms such as "80%."

Approaches to requirements

There are a variety of approaches to writing requirements: from traditional requirements documents to more agile approaches such as user stories, test-driven development (TDD), acceptance test-driven development (ATDD), and behavior-driven development(BDD). These approaches all benefit from following principles for testable requirements.

User stories

A user story is a requirement that is written as a goal, using language that avoids technical jargon and is meaningful to the end-user. User stories are brief and often follow the format: **As a** [user role] **I want/need to** [feature] **so that** [goal]. For example: *"as a customer searching for a product, I want to choose whether to see the list of available products in a list or in a grid so that I can compare the available products."*

As the name implies, writing requirements as user stories puts the focus on the user or customer. By themselves, requirements expressed as user stories don't have enough information to be testable. User stories should include acceptance criteria so that the team can know when the story is "done." Read more about user stories at the Agile Alliance website.

Test-driven development (TDD)

In TDD, requirements are written as unit tests. The unit tests are executed before any coding and should fail because the code they describe doesn't exist yet. Then the code is written or refactored to make the test case pass, the test is executed again to ensure that it does pass, and then any necessary refactoring occurs.

This approach is sometimes called developer testing, both because this testing is performed by developers, but also due to where the testing occurs in the development cycle. However, testers have a valuable role to play in TDD. Testers can work with developers to create better unit tests, applying techniques such as boundary value analysis, equivalence partitioning, and risk analysis; and help ensure that necessary integration and workflow testing occurs. TDD tests are typically written in a tool such as Junit or VBUnit, and form an important part of the documentation of the application. Read more about TDD at the Agile Alliance website.

Acceptance test-driven development (ATDD)

In ATDD, user stories and their accompanying acceptance criteria become the tests that are used to demonstrate to a customer that the application works as intended. Acceptance tests are typically written in collaboration by a team of three, called the "three amigos", that includes a user representative, a developer, and a tester. To make sure the tests are understandable by everyone on the team, they are written in "business domain" terms rather than technical terms.

The workflow in ATDD is similar to TDD: first, the user story is written, followed by the acceptance test. Then the user story is implemented, and the team repeats the acceptance test to confirm that it passes. Finally, any needed refactoring is done. It is possible for a team to practice both TDD and ATDD at the same time. For recommendations on writing good acceptance tests, refer to the article The ABCs of Acceptance Test Design by author Jeff Langr.

Behavior-driven development (BDD)

One way to increase clarity in requirements is to write them as realistic examples rather than using abstract terms. This approach is referred to as specification by example (SBE) or behavior-driven development (BDD). BDD is similar to ATDD but uses a specific syntax called Gherkin. In BDD, user stories are supplemented with examples to create "scenarios." The scenarios for a feature are stored together in a feature file that can serve as an executable specification.

BDD scenarios are written in using the GIVEN-WHEN-THEN syntax, as shown in the example below.

Feature: Search Results Display
As a customer searching for a product
I want to choose whether to see the list of available products in a list or in a grid
So that I can compare the available products.

- Scenario 1
  Given: I perform a search for an inventory item
  And: There are at least two items returned by my search
  When: My preferences are set to list display
  Then: I see a list of the items returned by my search

- Scenario 2
  Given: I perform a search for an inventory item
  And: There are at least two items returned by my search
  When: My preferences are set to grid display
  Then: I see a grid of the items returned by my search

- Scenario 3
  Given: I perform a search for an inventory item
  And: There are no items returned by my search
  When: My preferences are set to list display
  Then: I see a list of suggested alternatives

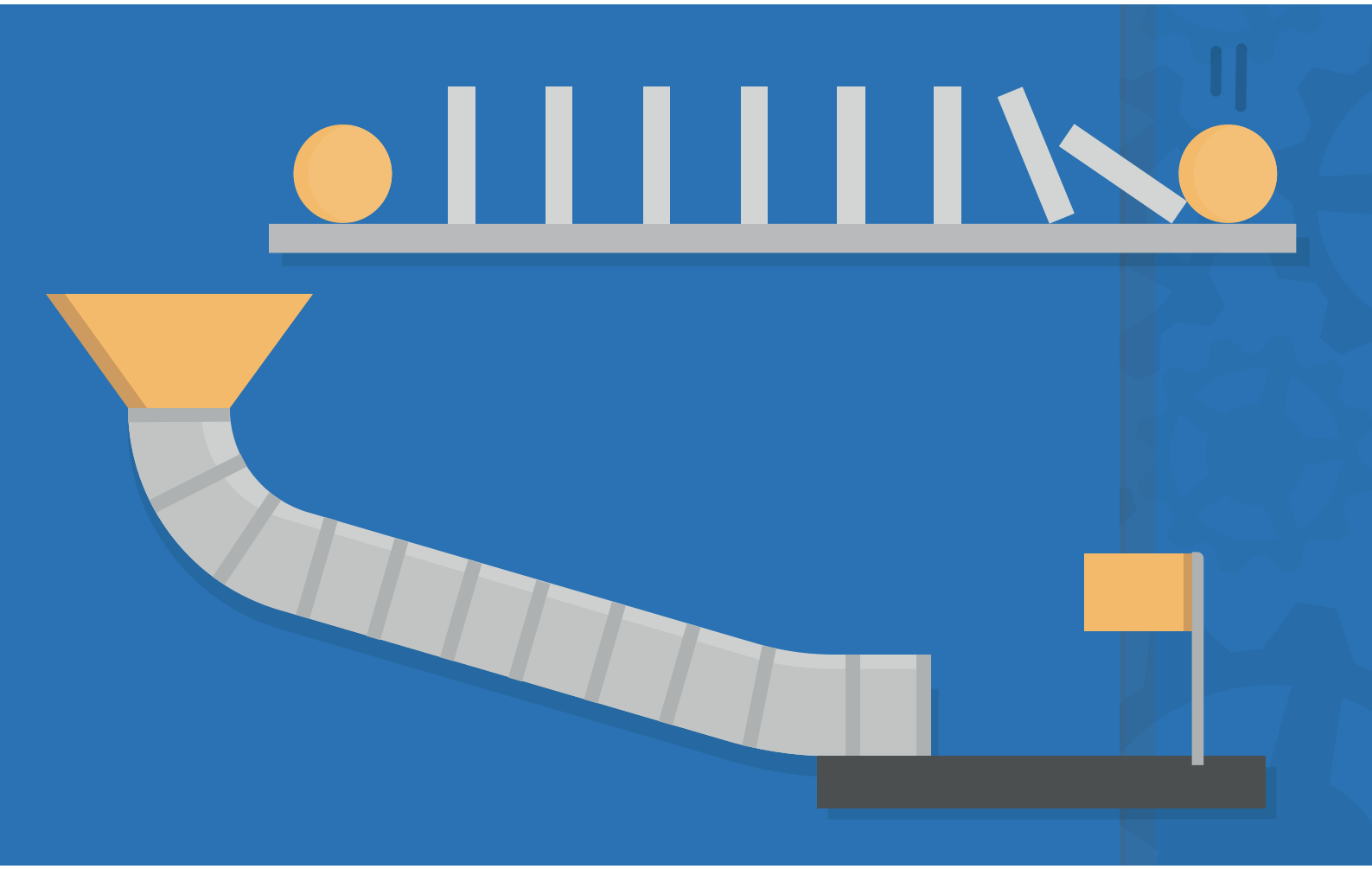
Tools for testable requirements

No special tools are necessary to create testable requirements. They can be documented in word processing files or even note cards. But tools can make the process more efficient. ATDD tests may be captured and automated using a tool such as FitNesse. For BDD there are also a variety of language-specific tools to write requirements in the Gherkin GIVEN-WHEN-THEN syntax and prepare them for automation, including the following:

- Cucumber for Ruby
- Jbehave for Java
- SpecFlow for C#
- Jasmine for JavaScript

Ranorex Studio's robust tools and open API support all testing approaches, including TDD, ATTD, and BDD. To see how Ranorex Studio integrates with SpecFlow to automate, BDD scenarios, read the article How to Use Ranorex Studio in Your BDD Process on the Ranorex blog.

Dozens of books have been published on the topic of writing effective software requirements, and this ebook presents just a brief overview of strategies for ensuring that your requirements are testable. But the most important strategy is to ensure that testers and user representatives are included early in the process of requirements definition. While testable requirements make it easier to automate your tests, the key goal is to ensure that the entire team shares a clear understanding of the requirements.



9. Plan end-to-end testing

End-to-end testing (E2E) examines the real-world scenarios of an application from start to finish, touching as many functional areas and parts of the application's technology stack as possible. Compared to unit tests, which are narrow in scope, E2E tests have a broad scope, and so are sometimes called "**broad stack"** or "full stack" tests. E2E tests focus on validating the workflows of an application from the perspective of the end-user, which makes them highly valued by management and customers. E2E testing is usually performed last in the testing process, following lower-level unit, integration, and system testing. Despite their value, automated E2E tests can be complex to build, fragile, and challenging to maintain. As a result, a common approach is to plan a smaller number of E2E tests than unit and integration tests, as shown in the **test automation pyramid**. E2E testing is conducted in as realistic an environment as possible, including the use of back-end services and external interfaces such as the network, database, and third-party services. Because of this, E2E testing can identify issues such as real-world timing and communication issues that might be missed when units and integrations are tested in isolation.

End-to-end example

Assume that you are testing a web shopping platform that requires a third party to validate payment details. This application might contain E2E tests such as the following:

- User logs on, searches for an item, puts the item in the cart, selects payment and shipping details, and then checks out and logs off.
- Use logs on, searches for an existing order that has been shipped, reviews the tracking information, and receives a detailed response on the delivery of the order, then logs off.
- User logs on, searches for an existing order that has been shipped, requests a return of the order, receives a shipping label to return the item, and logs off.
- User logs on, opens their account information, adds a new payment type, receives verification that the payment type is valid, and logs off.

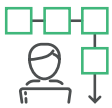
These tests will access third-party services such as payment verification and shipment tracking, as well as one or more databases for customer information, inventory, orders, and more.

Best practices for end-to-end testing

A typical E2E test can be complex, with multiple steps that are time-consuming to do manually. This complexity can also make E2E tests difficult to automate and slow to execute. The following practices will help manage the costs of automated E2E testing while maintaining the benefits.

Keep an end-user perspective

Design E2E test cases from the perspective of an end user, with a focus on the features of the application rather than its implementation. If available, use documents such as user stories, acceptance tests, and BDD scenarios to capture the user's perspective.

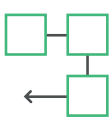


Limit exception testing

Focus E2E tests on high-use "happy path" or "golden path" cases that capture typical use scenarios like the ones listed for the web shopping platform described above. Use lower-level unit and integration tests to check bad path/sad path exception cases, such as a user attempting to order more of an item than is currently in inventory, or returning an item past the allowable return date.

Apply risk analysis

Given the relative expense of performing E2E tests manually or automating them, concentrate on your application's high-risk features. To determine a high-risk feature, consider both how likely a failure is to happen, and the potential impact that it would have on end users. A risk assessment matrix is a useful tool in identifying risk. Read more about risk analysis in the Ranorex Testing Wiki.



Test in the right order

When a single unit test fails, it's relatively easy to figure out where the defect occurred. As tests grow in complexity and touch more components of an application, the increase in potential points of failure make it harder to debug them when a failure occurs. Running your unit and integration tests first allows

you to catch errors when they are relatively easy to resolve. Then, during E2E testing, complete your critical smoke tests first, followed by sanity checks and other high-risk test cases.

Manage your test environment

Make your environment setup process as efficient and consistent as possible. Document the requirements for the test environment and communicate them to system administrators and anyone else involved in the setup of the environment. Include in your documentation how you will handle updates to the operating system, browsers, and other components of the test environment to keep it as similar as possible to the production environment. One solution may be to use an image backup of the production environment for testing purposes.

Separate test logic from your UI element definitions

To make your automated E2E tests more stable, separate the logic of your tests from the UI element definitions. Use an object repository or a page object pattern to avoid having your test logic interact directly with the user interface. This makes your tests less likely to fail due to changes in the structure of the UI. Ranorex Studio includes an object repository for separating automated tests from UI element definitions, or you can use the page object pattern when testing web applications with Selenium.

Handle waits for UI elements appropriately

Don’t allow an E2E test to fail unnecessarily when waiting for a page to load or a UI element to appear on the screen. In Ranorex Studio, you can add an action that waits for a UI element to exist, for a specified period. If that period is exceeded, then the test fails. The wait time should be at least as long as the normal time that it should take for the UI element to appear. But don’t make it too long. Excessively long wait times may indicate a problem with the application, interfaces or environment, and are annoying to end-users. In addition, allowing long wait times in your automated tests can also affect the overall execution of your E2E testing. In general, set a wait time that is just little longer than the normal time it takes for a UI element to appear.

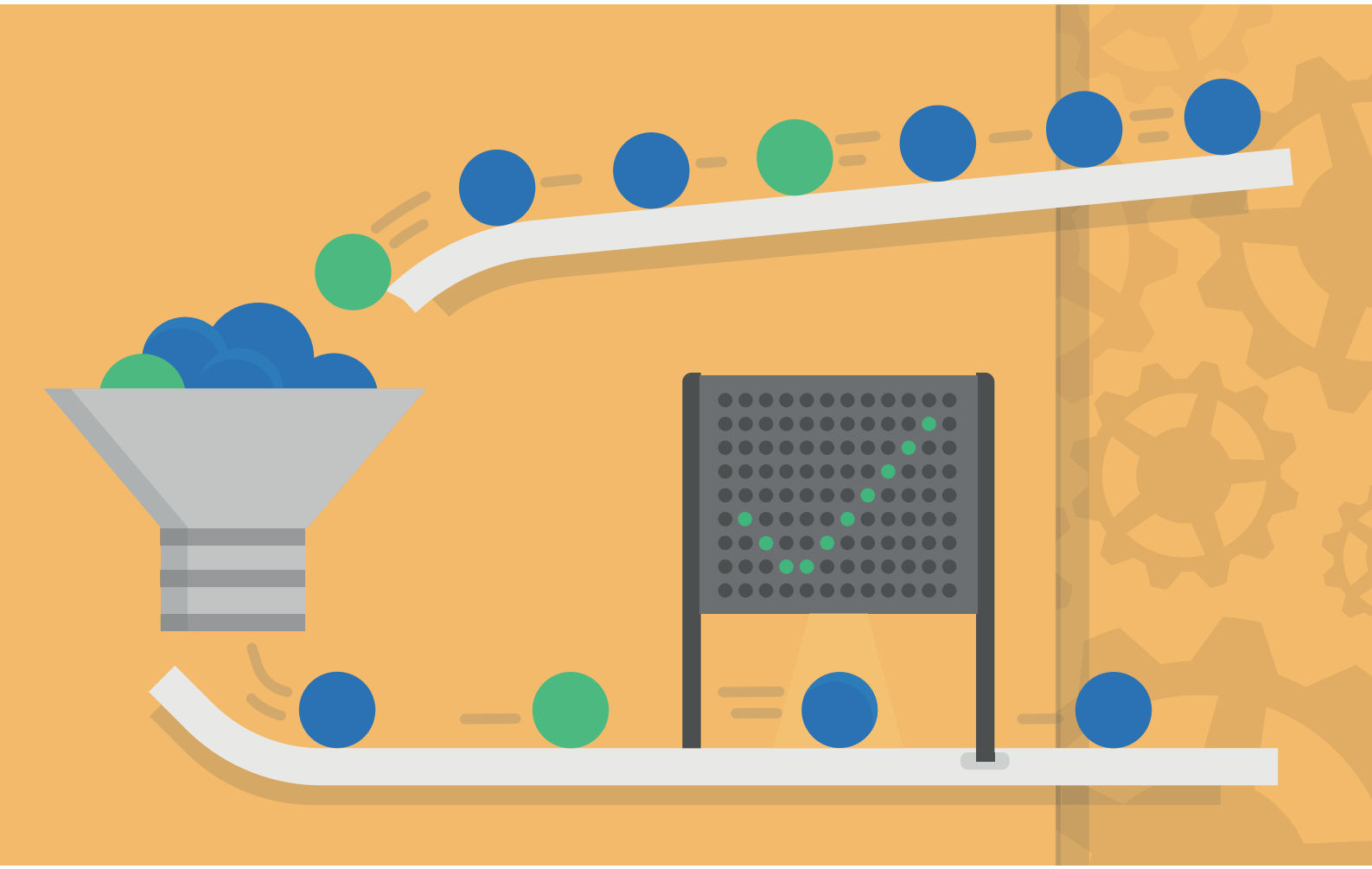
Choose the right devices

For mobile devices, concentrate physical testing on the most popular versions of iOS and Android, and then use simulators/emulators to provide coverage for less popular versions. Test on both WiFi and cell carrier connections at various speeds from 3G to LTE.

Optimize your setup/teardown process

Use setup/teardown processes so that your test environment is always ready to go. Create your test data using automated scripts, and clean it up afterward so that the test environment is restored to its original status and is ready for the next round of testing.

It's important to plan manual and exploratory testing as part of your E2E testing, to address difficult-to-automate aspects such as usability and user experience. And, to ensure that you have a complete and well-balanced set of tests, include automated performance and load testing, which is covered in the next section.



10. Combine functional and load testing

The goal of software development is to deliver a great user experience. This includes not just the functionality and usability of an application, but also its performance. Mobile applications that load slowly are likely to be deleted. Web pages that take a long time to refresh may be abandoned, leading to lost traffic. Combine functional testing with load testing to confirm that an application's features work as expected with reliable performance even during peak use. A typical QA process might have the following phases:

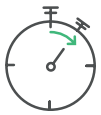
- The application passes unit and integration testing.
- Automated functional testing validates that new functions of a website or application work as described in the user stories/BDD scenarios and that no regressions have been introduced.
- The QA team conducts exploratory testing to uncover hidden defects. These tests are performed under a lower load than would be expected in production.
- Finally, a performance test verifies the system's behavior in simulated usage conditions – varying traffic, multiple users, increased network load, etc.

The drawback of this approach is that the first three phases provide feedback on how an application performs in ideal conditions. It is not until the very end of the testing process that the team discovers how the system works under conditions that simulate real life. In contrast, combined functional and load testing process might have the following steps:

- The application passes unit and integration testing.
- Automated functional testing is performed in combination with load testing to confirm that the application will perform as expected under realistic conditions.
- The QA team conducts exploratory testing to uncover hidden defects. Initially, these tests are performed under a lower load than would be expected in production but are followed by additional exploratory testing with a simulated load. This provides important insights into how end-users will experience the application in production.

This approach to testing enables you to verify key functionality under realistic usage conditions earlier in the testing process when it is easier to identify and resolve them.

Key concepts



Performance testing

Performance testing generates benchmarks for evaluating a system, which consists of an application or website along with its database files, servers, network hardware, etc. Performance testing benchmarks can include the numbers of concurrent users that an application can support comfortably, and the complete system's responsiveness, throughput, resource usage, latency, and more. Generally, latency refers to the time required for an application to respond to a user input. In networking, the term latency is used more specifically to describe the amount of time it takes a data packet to travel one-way between nodes of the network, or for a data packet to make a full round-trip back to its origin point.

Load testing

Load testing is a subset of performance testing. Load testing examines the performance of an application during defined periods of "normal" load and "high" load. Load testing can include activities such as simulating a target number of concurrent user transactions as well as a target volume of transactions. The goal of load testing is to determine whether the system performs well during periods of normal usage as well as high usage.

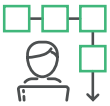
Stress testing

Like load testing, stress testing is a subset of performance testing. Stress testing identifies the point at which system response degrades significantly or even fails, called the "breakpoint." Stress testing is done by increasing the load beyond the maximum anticipated volume. Stress testing can help identify how the application or website will respond to extreme conditions. For example, if an application is overwhelmed by user transactions and crashes, how long does it take to recover? How much data, if any, will be lost?

Best practices

Define measurable metrics

Set performance goals that are measurable, such as response time, throughput, number of requests processed per second, and resource utilization. Always measure performance from the user's perspective. For example, it doesn't matter if response time from the server is fast if the client application is slow.



Use realistic scenarios

Create realistic tests that simulate the most common workflows. Prioritize your user's "happy path" or "golden path" scenarios. For more information on prioritizing functional tests, refer to the end-to-end testing best practices in section 9.

Configure a clean environment

Your test environment should be isolated from other applications and traffic that may affect performance. For web testing, this means clearing your browser cache and cookies; if the browser uses cached data and cookies to process client requests, it can cause unreliable load and performance testing results.

Verify high-risk scenarios first

Conduct smoke and sanity tests before proceeding to your other tests. If there is a significant problem with the application under test, you want to know that right away before spending time on other types of testing.

Start small

Verify that your functional tests complete successfully for a limited number of users before simulating multiple users. Scale up gradually to determine where bottlenecks may occur.

Tools

Ranorex Studio provides a full set of tools to automate functional UI testing for desktop, web, and mobile applications. Within Ranorex Studio, you can simulate transaction loads in several ways:

- Use data-driven testing to generate a volume of data and execute CRUD actions (create, update, and delete) against a database
- Set the number of iterations for a given test action to simulate actions such as queries and page loads
- Test in parallel to generate a load from a variety of devices

For more comprehensive load testing, Ranorex integrates with **NeoLoad** from Neotys. NeoLoad provides advanced tools for designing and analyzing load tests, including component testing for APIs, web services, and microservices. To provide the most realistic analysis of your application or website's performance, Neoload supports using physical machines or virtual servers, as well as cloud-based providers in multiple geographical locations. NeoLoad allows you to identify performance bottlenecks with real-time monitoring of an application's infrastructure. To learn more about the Ranorex integration with NeoLoad, refer to one of the resources below:

- Download the NeoLoad plugin for Ranorex.
  NeoLoad Integration for Ranorex at GitHub. In addition to the necessary files, this link includes step-by-step instructions to configure and use the NeoLoad plugin for Ranorex
- Watch the on-demand webinar
  Combining Automated Functional and Load Testing. Learn the benefits of combining automated functional and load tests with Ranorex Studio and NeoLoad
- Read the blog article
  Ranorex-NeoLoad Webinar – Questions Answered. Get answers to questions asked during the Ranorex-NeoLoad webinar to fully understand how to set up the Ranorex-NeoLoad integration.

Conclusion

The Ranorex team hopes that this ebook has been informative and helpful to you. For additi-onal articles and screencasts on the topic of test automation, visit the Resources page of the Ranorex website. Or, download our free ebook "Strategies for a Successful Test Automation Project." To learn more about how Ranorex Studio can meet your test automation needs, download a free trial using the link below. For free assistance in setting up and using Ranorex Studio in your environment, contact our sales team. You can also register to attend one of our live test automation webinars, with sessions for desktop, web, and mobile applications.

To explore the features of Ranorex Studio risk-free download a free trial today, no credit card required.

Download Free Trial