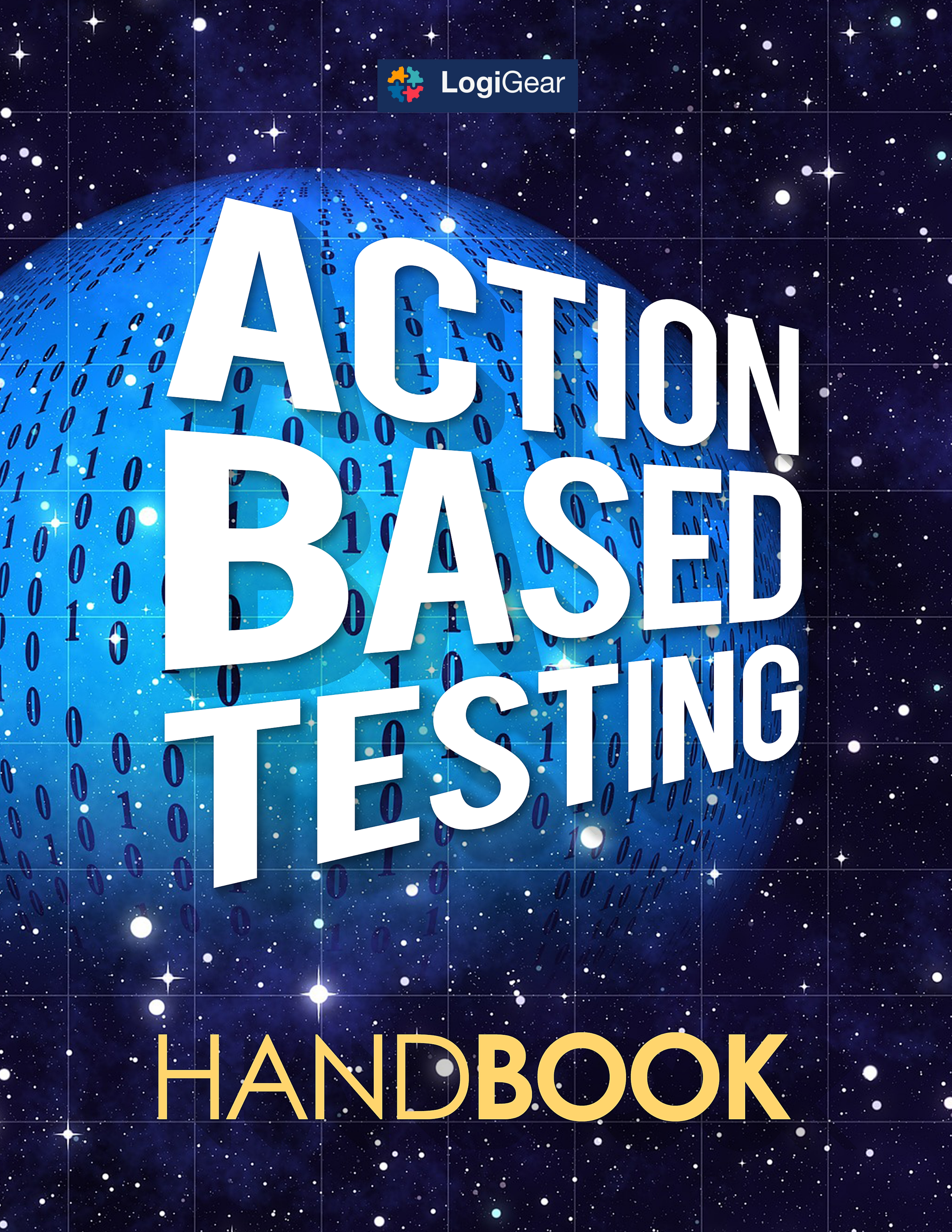
LogiGear

ACTION BASED TESTING HANDBOOK



ABT Handbook
June, 2016

1. Introduction

This handbook provides a set of guidelines for the use of Action Based Testing (ABT) and the TestArchitect (TA) product suite. It contains principles, examples and recommendations for best practices. It is intended to be a dynamic growth document that is updated and expanded over time. Feedback is welcome.

In addition to this handbook there are a number of more general introductory articles on Action Based Testing.

1.1 Organizing and designing the tests

Making automated testing successful (scalable and long-term maintainable) is often regarded as a technical challenge, while in fact it is much more a test design challenge:

- Technical problems can take precedence sometimes, but once resolved they normally go off the critical path
- On the other hand poorly organized tests can be a continuous burden for their automation.

A main focus in ABT and TA therefore is on test design as the driver to automation success. The main goals for test design are:

- Simplicity, in particular also for non-technical people
- Efficiency, reach a maximum effect with a minimal amount of tests and actions
- Manageability, to be able to "see the forest through the trees"
- Maintainability, to be able to adapt to changing applications quickly
- Depth and coverage, in other words, being good tests

A good organization of tests can benefit project in the following ways:

- Better tests: make tests more comprehensive and deeper
- Better automation: know and address impact of changes in the application under test

1.2 Test Modules

The base concept in ABT test design is the "test module". It contains test objectives and test cases to test them, all defined within a single scope. Based on the test module concept, there are roughly 3 phases to organize and design the tests:

1. Establish an overall test design: outline the folders and test modules within them. The discussion in paragraph 2.2 and the template in the appendix can be starting points. Even though in agile projects test development may not be top-down, an overall test design can work as a framework to help keep tests organized, which benefits their maintainability and manageability

2. For each test module:

    a. Do an intake, make sure to understand what needs to be tested and who needs to be involved
    b. Plan the module, in particular the test objectives and test cases
    c. Testing techniques like decision tables and state transition diagrams can sometimes help in this stage to identify test cases

3. Create the test cases using actions. What actions to use depends on the scope of the test module

When the test modules will actually be developed is a project decision, based on priorities, and on how projects are organized. In the case of an agile method like Scrum teams play a large role in deciding what test modules to develop. However, giving the long lasting value potential of tests are products, an organization can decide to plan their production explicitly as outcomes of sprints, in addition to the software products the teams will produce. It is vision on tests:

- Are tests supporting relatively straightforward artefacts just meant to help build and maintain the main product, or
- Are tests products with business value, that reflect a deep understanding of the functions and workings of the applications under test, and the business it supports

1.3 What about Test Cases?

Unlike what is common elsewhere, the core products in ABT are test modules, not the test cases. In ABT, test cases are the outcome of the test creation process, not the input. Once a test module is defined the number of test cases created to achieve that goal can vary with the creative design process of the person creating the test module.

It is good practice to add cases to a module after its initial completion if new ideas for test situations come to mind. It is also not a problem to reorganize the test cases to improve the test module.

From a test design perspective test cases in an ABT test module do not have to be independent from each other. A test designer can define a flow, where each test case uses the status that was left behind by the previous test case. In contrast it is good practice in ABT to keep test modules independent from each other.

1.4 Existing tests

In many automation projects, there are already existing test cases, often in substantial numbers, and developed with much efforts over time. A good approach for this is to:

- Create a high level test design as if it was a new test project, see the previous sections
- Once the test design (module list) is agreed up, migrate the existing test collection:

    - Split cases that have multiple levels of tests in them
    - If test cases have a high level of detail, re-use that to create actions (using the "action definition" features in TA or TA4VS)
    - Don’t hesitate to eliminate redundant materials

2. High Level Test Design (break down of a test in test modules)

2.1 Criteria

Here are some criteria to use to do the breakdown, split into "straightforward criteria" that usually apply in a project, and "additional criteria" that depend more in the specific situation a project is in, and are less common:

Straightforward Criteria

- o Functionality (customers, finances, management information, UI, ...)
- o Architecture of the system under test (client, server, protocol, sub systems, components, modules, ...)
- o Kind of test (navigation flow, negative tests, response time, ...)

Additional Criteria

- o Stakeholders (like "Accounting", "Compliance", "HR", ...)
- o Complexity of the test (put complex tests in separate modules)
- o Technical aspects of execution (picture checks, special hardware, multi-station, ...)
- o Overall project planning (availability of information, timelines, sprints, ...)
- o Risks involved (extra test modules for high risk areas)
- o Ambition level (smoke test, regression, aggressive, ...)

2.2 Template for a global test design

The appendix contains an example template that you can use as a starting point for organizing your tests ("high level test design"). However, various projects and teams may make different decisions.

Some of the main categories in this template are:

- Business Objects
- Business Flows
- Features
- Interoperability
- Components
- Graphics, multi-media
- Technologies
- Data
- Administration
- Customizability
- Concurrency, race criteria
- Non-functional tests

Here are the main considerations that went into defining the template.

2.2.1 Business versus interaction

The most common phenomenon is having all tests include the detailed steps to execute them. A tester often does this to either direct manual execution or help the automation. However, tests that are specified in more detailed needed for the scope of the test will be hard to automate efficiently, even for an experienced automation engineer who is versed in refactoring techniques. Therefore as a guideline one should distinguish between "business tests" and "interaction tests".

The business under test, like "if a customer buys a pound of sugar, it will cost $1.50". When you see a test like this there is no information on the UI (or API) of an application. You don't even know if the application is in Java/Swing, WPF or web based. The test talks in business terms, not functions of the application. Keeping interaction details out of a test like this means the test is very stable, even if the system undergoes major maintenance. It is also the kind of test a domain expert will be able to understand and contribute to.

The interaction with the application. To let an application support a business one needs to interact with this, usually through a UI, but it can also be via an API (like a web service, an exposed RCP service, or a command line). In particular UI's are usually extensive, detailed and complex, and they need a lot of testing. Therefore tests of the interaction will be necessary and plentiful. In such tests it is commonly not needed to vary the data much, it doesn't matter much whether it is sugar or salt, as long as the drop down box with the products shows the right list of them, and in the right order.

There is no rule that says that business tests are more important or vice versa, but it is usually a good idea to keep the two kinds of tests in separate test modules.

For API interaction can also be a target for testing. For example if an application component communicates with other components using a protocol, tests can be created to make sure all components handle valid and invalid messages well.

2.2.2 Differentiating interaction tests

Though less essential as distinguishing them from business tests, it can help to differentiate the interaction tests further. This is true in particular if the UI of an application is large and complex, and has many features to work with the new and availability. Three categories for this can be UI behavior, input handling, and screen flows. However, be prepared that the distinction can be subtle, and not always unambiguous.

UI's (both traditional and web) have come a long way from the simple forms with text fields from the past. Selections in one place in a UI can (should) trigger effects on other parts. These can be of great importance, in need of detailed tests. A simple example is that a button may become enabled if a value is entered in a field, and should be disables again if the value of such field is removed, with a delete key or with a cut to clipboard operation. Other examples are changing views when tree items are selected, or even pictures that can be dragged, rotated and zoomed in.

Input handling also warrants detailed tests, but the focus is on values and how the application handles them:

- validation, if values are invalid, is there a message
- do fields have correct default values
- are there dependencies between values, like a total field of a column with numbers

The actions for input handling can be system level actions like "enter" and "click", but also user defined like "check message", that hides whether the message is a pop-up, or might appear in red on a status bar.

Screen flow can need attention as well. Entering values in an application may go through more than one screen, and a user may use different routes, or cancel a transaction altogether, which in use cases is known as "alternate" paths.

You should also consider the numerous keyboard operations and short-cuts. These can be function keys, special keys or accelerator keys. A good test is to verify if the tab key goes through the input fields in the right order, and skips controls that are not relevant for data entry (like labels). Obviously the back-tab key should work too.

2.2.3 Business objects and business flows

Virtually every application is meant to support some form of "business". This can be renting cars, exploring oil fields, or playing games. Understanding the business is an important step in writing good tests and organizing them well.

When starting an overall test design it is good to ask two questions about the business:

- First identify what *business objects* the system works with, like cars, orders, customers and invoices, also trying any sub-categories, like trucks and regular cars. For each object find out what data operations are possible, applying lifecycle and other other data operations, this will mostly likely contain many tests. Examples are create, update, delete/close, but also move, copy, rename, import, export, statistics, reporting, graphical representations, etc
- Then identify *business flows*: in an average business situation there can be many transactions that involve multiple objects; like a customer orders a service, the service is delivered, the customer is invoiced, and a payment is received. One service order can result in multiple invoices, and one invoice can combine data from multiple orders. In most systems there will be many flows and variations (both primary and alternate). From a testing perspective combinations of multiple flows can warrant attention, like multiple events (orders, price changes, re-stockings, etc) that influence each other and happen concurrently.

A particular case of testing a business flow is from a user perspective. For example, a customer may order a number of products from a web site, enter a discount code, specify a credit card, and expectan order confirmation.

For each of the objects and flows there will typically be business oriented tests and interaction oriented tests. Try to avoid detailed steps in tests if interaction is not the target of the test.

2.2.4 Global and detailed organization

A common question is how to group the test modules into folders. One could imagine to group all interaction tests together, further divided in categories, and put the business functional tests in another grouping. However, this tends to give little structure, and it can be difficult to handle the impact of a change in the handling of a business object or a business flow when the tests of it are in different locations in the tree.

The template in the appendix takes a different approach. It puts the business objects and business flows at the top level in the tree, and groups business and interaction tests below them. This way all test involving let's say a car are in one easy to find place.

2.2.5 Special tests

Apart from what in most situations are likely to be the two main categories of tests, business objects and business flows, there will be various other categories, many of which are identified in the template as well.

3. Developing the test modules

3.1 Creating a Test Module

Before starting development of a test module, make sure to do a proper "intake". Find out what needs to be tested, and what the best strategy to do is.

If possible, discuss your approach with at least one other person, and write down your design decisions, for example in the "notes" field of the test module.

Items to think about:

- Do we understand the scope and subject matter of this test module ("what is this about?"). If not, can we find out?
- What "situations" need to be part of this test to assure we have met the scope
- What base data do we need, like customers with certain profiles, and how do we obtain it:
    - Create it in the test module
    - Run a "zero module", that enters the base data
    - Can the base data be fixed, or does the test module have to change it (can't it be rolled back if it does)
    - Can we use virtual machines and their snap-shots to quickly set up a suitable base line
- How do we design this test module, see also the notes further on in this chapter

Note: try to create your tests directly into (TA or TA4VS) test modules. Avoid detailed documents or verbose step lists prior to creating the test modules. Well-designed test modules should be self-documenting, with only occasional comments and notes.

The important thing to understand about a test module is its scope: what is the test aiming at. A dialog test looking at controls and tab orders is something else than testing if an interest rate is calculated correctly. The scope determines many things, like:

- What the test objectives are
- Which test cases to expect
- What level of actions to use (individual controls, or more generically, using action definitions)
- What the checks are about and which events should generate a warning or error (if a "lower" functionality is wrong)

First set-up a list of test objectives. This will help all involved to understand what you are going to test and whether it is enough, or too much. The easiest way is to set them up at the beginning of the module, by typing "test objective" for each objective. Then set up the test cases. Relate the test objectives to them. Note that test objectives and test cases only show up in the tree after a check-in of the test module.

Use "section" or "step" to clarify long test cases, but don’t use it too often.

Keep the size of modules below a reasonable maximum, typically about 500 lines.

Number test cases TC 001 etc, test objectives TO 001 etc. Make sure to relate the test objectives to the test cases that test them, using TA's features to do so.

As a general rule test modules are typically testing "business" or "interaction", not both. For a description see paragraph 2.2.1.

3.2 Defining Test Objectives

Making test objectives is part science and part art. It is possible to make and follow some guidelines like those listed in this section, but you will have to analyze underlying requirements and specifications well, and try to design test requirements that are both easy to assess and easy to test.

For a reader of a test (like an auditor) there are two main benefits from using test objectives:

- Quickly understand what a test (module) is about
- Verifying the completeness of the tests. From well-defined test objectives (and test modules) gaps are easier to spot than from test cases

The following guidelines may be helpful for the test objective process:

- Make sure you have a clear picture of what test modules to create, see what was written about this in chapter 2. The main purpose of objectives is to detail out the scope of a test module
- Test objectives are an "analysis" product, in contrast to test cases, which are "design" products. The objectives describe your understanding of what needs to be tested, not how it is tested. Make sure you understand functionality well before writing the test objectives
- As a guideline for the text of test objectives:
    - Make cause and effect clear, and mention cause first ("clicking submit empties all fields")
    - Make condition and effect clear, and mention condition first ("if all fields are populated, ok is enabled")

- Split complex sentences into small, atomic, statements
    - It is okay tough to combine two or more functionalities if this is not adding to complexity (like "ok becomes enabled if both first name and last name are specified")

- Keep test objectives short. Leave out as many words as you can without losing the essential meaning
- Try to keep test objectives independent from each other. Exceptions are possible though. For example a specific situation in the system under test might need a group of test objectives to describe it. In such a case try to use something like:

- "In the case of a sports car: the screen specifies seconds to reach 60MPH"
- "In the case of a sports car: no lease price is available" , etc.

Within a test module, group the test objectives as logically as possible. Based on the organized and well-formulated test objectives, try to identify gaps and either add test objectives if they are obvious, or provide feedback to the customer in case of questions.

Note that the objectives are not test cases. They should focus on *what* to test, not *how* to test. Usually test objectives are quite easy to create, for example directly relating them to business rules or system requirements (in TA they have a "source" field to do so).

Also to keep track of the testing status of an application test objectives can be an easy starting point. With their atomic and straightforward nature well-defined objectives will not likely change much over time, while a test developer may want to change test cases, to reflect increasing insight (new ideas) or relevant application changes. Therefore consider to relate requirements, business rules, bugs and/or other ALM items to test objectives rather than test cases. In turn relate the objectives to the test cases that test them.

3.3 Writing tests

The heart of the test development process is of course writing the actual tests.

Like with many other items, the scope of the test module determines what the tests should look like. If the scope is business oriented, like about renting cars and checking invoices, the tests should tell business stories with business level actions that evaluate the AUT within that scope.

If the scope is interaction the tests should show that as well, and will typically have lower system level actions, which can include built-in actions like "click" and "enter".

Try to be original and interesting in your test design—creativity is key. If bugs were easy to find, then the bugs probably wouldn't be there. As a tester you need to think out of the box, devising scenarios that a developer might not have thought about. Books like "Testing Computer Software" can give you pointer on how to do this in more detail. See also the article on Soap Opera Testing in Better Software Magazine.

Avoid "over-checking", adding checks that do not fit the scope of a test. They will distort your metrics, raising the percentage of passed checks without merit, and will make your tests harder to read and more sensitive to changes in the AUT.

Verify if the actions that you need to describe your tests exist already. If not, define new ones: choose a name and arguments. Choosing a good name in a uniform way will help with the re-useability of actions: you're more likely to find an action that matches what you need if it exists already. TA's auto-complete features will help you find existing actions, and knows what existing arguments are available. Actions can also be entered by dragging them from the explorer tree into the test editor. A dialog will show that allows you to choose what arguments you want to use (the others will have default values). See 5.4 for more details on using and naming actions.

You can lay-out tests test cases in your test module in three different styles:

- Make each test case independent, with its own set-up and tear-down actions. This gives the tests are relative simple "unit test" character, which can fit well when an application is still under development, like in a Scrum sprint
- Let the test cases share the "Initial" and "Final" sections of the test module, which will typically start and close the application under test, and define values for configuration variables. TA can make sure the Initial and Final parts are executed for each time you run a test case individually
- A more elaborate use of test modules is to let the test cases tell a story, in which each test case sets the pre-condition for the next one. This means a test module is always executed as a whole, and it allows for longer and more realistic tests

3.4 Checks, errors, warnings

Make sure to understand the difference between checks and warnings/errors. A check is done to see if an outcome is as expected. The tester anticipates that this might fail, it is the object of the test. A warning or error is given when an event happens that the tester didn't expect, like a window is not opening (and opening the window was not the object of that test).

Don't create "on the fly" checks that do not fit into the scope of the test module. If you have an action "create project" that you use as part of a workflow test, don't check whether the project was created successfully "just to make sure". You should have another test module that, thoroughly, tests creation of projects, and that test module should pass without issues before you run the higher level test module that tests the workflow.

The advantages:

- Tests are more readable
- Your statistics are not polluted with (probably passed) checks that don't belong
- Your automation will be much less cumbersome if you can trust detailed functions before addressing higher level functionalities
- Changes in the system under test will likely have less impact on the test set, and the impact will be more localized, focused on the tests where they matter

Make sure checks are visible. Don't hide checks that are relevant for test modules and test objectives or turn them into high-level actions. A person should be able to understand what you're testing just by looking at the test module, without a need to look into action definitions.

3.5 Commenting, documentation

Use comments that explain what your intentions are, and how you achieve those attentions. However, to ensure easy readability use them with care:

- Don't use comments for obvious steps, like "log into the system" above a line with a "log in" action. A good way to check for this is to leave out the comment and see if the test is still understandable
- Make sure that you do comment when a "trick" is used, an unusual and/or complex construction is used to solve a testing problem (Tip: avoid such constructions altogether, and consult with senior leads before using them)

When using TestArchitect, make sure to populate the fields in the "information" tab of test modules, actions, test cases, and other items.

3.6 Variables and expressions

The values that a test needs to work with are not always clear-cut. They might be different from test run to test run, or might be dependent on a context in ways that are not easy to predict when you design a test.

Instead of hard values you can use "expressions" as values of arguments, like this:

	car	result		
get number of cars	Buick	>> buicks		
	first name	last name	car	number of cars
rent car	John	Smith	Buick	1
	car	currently available cars		
check number of cars	Buick	# buicks - 1		

Consider using variables if you have values that have a potential to change across version changes or environment factors (like a sales-tax percentage). If possible don't use variables for test values, this avoids tests looking like "source code", and makes them less readable for non-technical users.

If you use variables, consider using names like "John" and "Mary", not "user 1" and "user 2".

3.7 Data sets

You should only use data sets if having a large quantity of test values is a meaningful contribution to the test design that cannot be made easily in another way. Using data sets, or other externalized data, can make it hard to read and maintain tests. In many cases identifying equivalence classes (organize values into categories, and have only one value for each of those) can help to keep the needed variety limited, after which they can be specified with a series of high level actions.

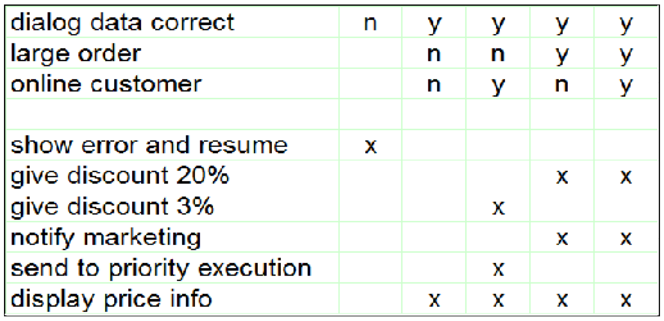
3.8 Naming

For items like actions, arguments, interface entities and interface elements use simple lower case names with spaces. Also try to avoid prefixes, underlines, capitals, numbers and "CamelCase" e.g. so_this_IsNOTAGreatNamE123. Your project is about tests, the actions and interface elements should not dominate.

Try to remove words like "test if", "it should be possible to", "successfully" etc in titles of test modules, test objectives and test cases. You and the reader already know that these items are about testing things that should work well. So rather than "test if an administrator can block a folder successfully", simply say "an administrator can block a folder".

3.9 Testing Techniques

Many excellent books have been written about testing techniques, which are not repeated here (for an example of a book close to home see: *Testing Computer Software*, by Cem Kaner, Hung Q Nguyen and Jack Falk). Test techniques can often help to identify potential test cases. Here is an example of one test technique: a decision table. It lists a number of conditions and the expected output/behavior the system should have based on their values.



dialog data correct	n	y	y	y	y
large order		n	n	y	y
online customer		n	y	n	y
show error and resume	x				
give discount 20%				x	x
give discount 3%			x		
notify marketing				x	x
send to priority execution			x		
display price info		x	x	x	x

Typically you would create a test case for each column, in particular if the expectations for the system include different behaviors other than only different output values (in which case a data driven approach could be considered).

Some other techniques to consider are state transition diagrams, boundary analysis, and error guessing. For a more unconventional technique look at the Better Software Magazine article on "Soap Opera Testing".

3.10 Data driven testing

Although most test cases will work well with scenarios with fixed data, it can also be useful to let them execute more than variation of the data (input and expected outcomes). You can define the data rows in a "data set", and TestArchitect can repeat the test for each row. So-called "filters" allow you to fine-tune the rows that will be used for the test.

Use data sets if having a large quantity of test values is a meaningful contribution to the test design. It should also fit the scope of the test module. Using data sets, or other externalized data, can make it hard to read and maintain tests. See if you can identify "equivalence classes" that organize values into categories, like "minor", "major", "retirement" for ages, and have only one value for each of those in the data set.

4. Actions

Actions carry the navigation details that you do not want to be visible in the test modules. Usually they are automated, but they can just as well be executed manually.

Each action consists of an action keyword and arguments.

4.1 System level, application level, navigation, utility

For larger projects, it is recommended to organize actions in at least three levels, which we often nickname "low", "middle" and "high".

At the highest level are the business oriented application actions, like "create customer" or "rent a car". They hide navigation details like which screens to go through and which buttons to push.

The lowest level of actions are system/platform oriented, and relate to the user and non-user interfaces of the application. The actions at this level are generic: not depending on a particular application. An action like "select menu item" is usable across many applications and platforms. Many low level actions are built in into the TestArchitect products. The lower level actions are typically used:

- To create interaction tests that need a low level of detail
- As building blocks to create higher level actions (in the TA "action definitions")

In addition to the higher and lower level actions, it is recommend to have some intermediate ("mid lev-el") actions. These actions deal with compound pieces of navigation and data entry that can be used and shared between high level actions, for easier maintenance. For a UI there are typically two cases:

- Enter all fields for a screen (like "enter address fields"), with an argument (and default value) for each field in the screen (dialog, page, form, etc).
- Notes:
    - Since these actions have to deal with the often complex details of dialogs, they tend to be long and detailed
    - They will also have many arguments, try to assign meaningful default values (including if applicable "<ignore>")

- Navigate to a particular screen or location (like "navigate to order screen")

Whenever you do something complicated in the implementation for an action, in particular common with navigation actions, make sure to document what you do well with comment lines.

4.2 Built-in, user defined, scripted

In TestArchitect actions can be implemented in 3 ways:

- Built-in
- Harness
- Action definitions

Built-in actions: close to 400 actions are built in into TestArchitect. Make sure to "shop" in the built-in actions repository first before you create your own.

The "harness" can be used to program ("script") actions in a programming language: C#, Java or Python. It is suitable for non-UI actions that need special access, and for actions that have complex implementa-tions.

Action definitions can be used to create new actions out of existing ones. However, a spreadsheet is not a programming tool. For more complex actions, (e.g. they have loops in them), consider using scripted actions in the harness. See also the notes in 5.1.

4.3 Recommendations for actions

Another important item of attention is the design of the actions. Even though actions should be defined "on the fly", while developing the tests, their design should be done with care. Some items to consider in action design:

- The level of the actions used in a test module should follow the scope of that module. In particular low level actions, like "click" and "enter", are typically found in UI tests, while higher level actions, like "enter customer", are better suited for functional and business.
- Generic, re-usable, actions are usually better than specific ones
- However, each action should have a clear function, don't let one action have multiple unrelated functions
- Always define suitable default values for arguments, so the tester can omit them
- Keep argument names short and simple

We recommend that action definitions do not become "debugged". The lines making up the action definition should first appear in a low level test module. Once that works they can be re-used in the action definition.

Use names that are meaningful and short, as short as possible without losing meaning. Avoid using upper case letters and underlines in names. Some guidelines for name standardization:

- Use a verb followed by an object, like "check picture"
- Standardize on both the verbs and the object names: so always "check" not "verify" (or reverse), and always "customer" and not "client" (or vice versa). That way you are quite likely to come up with the same action names in similar situations

Keep names of action, argument, interface definitions etc lowercase, since they shouldn't dominate the contents. Don't use special characters, tests are not C++ source code. So use "enter order", not "EnterOrder" or "enter_order".

Try to avoid actions with many arguments. If arguments are needed, give them sensible default values. If still many arguments remain, split the line up over multiple rows, using the continuation symbol ">>>". The test line editor in TA can help you with that.

4.3.1 Using a high level action

An important choice is when to introduce a high level action to do part of a test. Here is an example based on an actual project. The system under test is about project management and the objective of these two test lines is to check whether an item is in a project. Notice that the lines will do the job, but that they show details of the navigation, in particular how those projects are organized there—note that they utilize the tree branch formation. There is also a list that will show tasks for a selected project.

	window	tree	tree item path
click tree item	main	projects	/Projects/Drill Assembly
	window	list	item
check list item exists	main	tasks	Plan of Approach

If the goal of the test module is to verify the working of the GUI, this level of detail makes sense.

But if as a tester you're only interested in projects and tasks, a business level action might be a better choice, making the test shorter, more readable, and easier to keep current when new versions of the application under test are made.

	project	task
check task in project	Drill Assembly	Plan of Approach

In particular when done at a larger scale a test becomes much more compact and easier to read for our user (and ourselves). But the most important advantage is that the automation is easier to maintain: once the action is stable, it will always work and when there are changes in the system under test, they will only impact one action, not all the tests.

You should not always hide details. If details matter to the test, they should be visible. In this example the action used is too high level, one can't see what is tested without looking into the automation of the action.

	short name	full name	old pw	short pw
check invalid userid pwd	johnson	John B. Johnson	abcdef99	abc

4.3.2 Hiding navigation

"Navigation" is what you do to go from screen to screen in an application under test. In the navigation you typically let the automation select menu items, type function keys, click buttons or links, etc. Most of our automation efforts dealing with navigation, including timing issues like waiting for windows and controls to appear.

Navigation is important and not always easy. However, it is seldom the objective of a test, so we don't want to see it in too much detail on most of the test modules (only the ones the explicitly test the navigation in a system under test).

Here is an example of what to avoid in a typical test module:

// go to the deposit window	
	key
type	F2
type	1
type	5

The three "type" lines explicitly spell out how to get to a window. This use has the following problems:

- The test becomes hard to read, with lots of extra lines that don't really matter for the scope of the test
- If the organization of menus in this system under test ever changes (like F3 instead of F2) you will have to update every test module

Therefore it is better to do something like this instead:

go to deposit

This is what we call a mid-level action: it hides navigation details, but will not typically be used in a test. Rather it will be part of higher-level actions.

4.3.3 Over-Checking

In ABT the scope of the test module defines what needs to be tested. Items outside of that scope should ideally not be checked, such checks should be left to other test modules. We sometimes refer to the practice of adding checks that are not relevant to the scope "over-checking".
Here are two of the most common examples:

- "On the fly" checking
- "Expected result" fields in test case steps

4.3.3.1 "On The Fly" checking

A common occurrence in tests is the "on the fly" testing: a check is done because the navigation happens to be in a certain window. Here is a typical example. This check on the "breadcrumb" was added to a test module on transaction processing.

Section	**process main transaction**		
go to deposit			
	text		
check breadcrumb	Transactions > Deposits > CD's > Cash amounts		
	window	control	value
Enter	deposit	cash in amount	100

The "breadcrumb" is a little text on top of a dialog showing where you are at the moment. In itself the breadcrumb is a useful thing to test, but it should not be done in this test module.

The problems when doing a check this way however are multiple:

- The test becomes longer and harder to understand--checks like this are distracting
- You have to maintain it in all your modules whenever the text of "breadcrumb" changes
- The statistics, how many checks, how many passed, etc, become cluttered with irrelevant checks

It is better have a separate category of test modules that focuses on UI related checks, that includes menus, breadcrumbs, short-cut keys, protected fields, etc. Keep it out of any other module.

A related use of "checks on the fly" is to help the automation: to make sure that after a certain navigation action we are really on the screen we're supposed to be. Such verifications that double check the automation should never be done in the form of checks, since that will clutter up the statistics. In case it goes wrong it will also show up as a "fail", while in this case a warning or error would be better: it is not the test that found a bug, but the navigation that went wrong.

Another way of saying this: when you run a business oriented test module like a financial transaction, the "navigation" (UI, menu's, etc) in a system under test should already work. To test this separate UI oriented test modules should have run successfully first, so the UI wrinkles are out before going for the business stuff.

Complex actions
The action below occurred in one of our project several years ago. It verifies whether a red marker, called a "flower", occurs next to an input field, to indicate that the field is mandatory.

// check that "deposit" has a flower, and that "account" has not

	window	field1	status1	field2	status2
check flower status	deposit	deposit	yes	account	no

The action takes a variable amount of arguments, with each argument specifying a field and whether that field should have a flower.

Although the action does the job, it is a bit complex and hard to read. An easier version would be:

// check that "deposit" has a flower, and that "account" has not

	window	field	flower
check flower	deposit	deposit	yes
check flower	deposit	account	no

Since this is about fields being mandatory the following would also be an option. It checks whether the fields are marked as mandatory, without specifying what that mark looks like. This puts the focus more on the purpose of the test, and does not have to change if the marking changes:

// check that "deposit" is mandatory, and that "account" is not

	window	field	flower
check mandatory	deposit	deposit	yes
check mandatory	deposit	account	no

In addition one could have tests that actually leave the field blank and verify that an error message is given.

In addition you could have a test that checks, for a known mandatory field, whether there is a "flower" symbol (and not if not mandatory).

4.3.3.2 Step results

Probably the most common source of checks that are not related to the scope of a test are the "steps" that are used in most test management tools to detail out a test case.

Number	**Description**	**Expected result**
1	login as "john"	the welcome screen is visible
2	etc	

This is typically translated into action lines like these:

	user	password
login	john	secret
	window	
check windows exists	welcome	

In most tests the log-in step is only necessary to get into the application under test: verifying whether the log-in works however is usually not part of the scope of the test. This check should therefore be omitted.

In larger test sets with many steps for each test case these "expect result" checks are a problem. They obscure the actual testing and are often wrongfully counted in statistics as successful checks.

A way to deal with them can be to scrutinize each expected result field and assess whether:

- A verification of this expected result is a meaningful addition to the test module, fitting in the intended scope of the test
- If not, is there another (lower-level) test module covering this verification. If not consider creating one. This approach is a bottom-up alternative to identifying test modules in an existing project

5. Test life-cycle management

5.1 Project organization and management

In the vision of ABT there are three related but distinguishable product life-cycles, each with its own deliverables:

- System development (main product: software)
- Test development (main product: test modules)
- Automation development (main product: working actions)

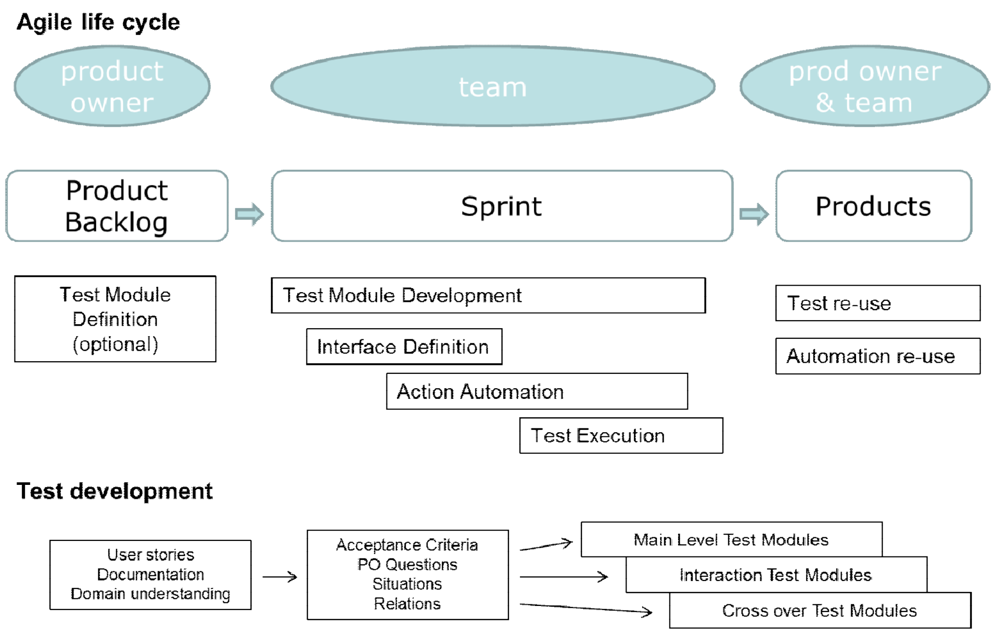
It is for example possible to re-use business level test modules even if the application is completely re-engineered towards another platform, like from desktop to web-based. For instance how a mortgage works is largely driven by business principles, and it doesn't matter what platform a mortgage application is implemented on, or what its UI's are.

In most projects it is a good idea to define two ownership arenas: test design and automation. The test design responsibility is to coordinate the test designs and action use. The automation ownership is about making actions work, mapping interfaces, writing scripted actions, and solving problems like timing and access. We often call this "automation engineering". The testers and automation engineers will often be in the same team, but may have different skillsets: designing a good test is different from creating stable automation.

For complex projects, pay attention to the early "intake" phase. Create a test design that identifies test modules. Start small: create some example test modules that are discussed with the test owners and best practice specialists. Once agreement is reached such examples can be used as guidelines for the rest of the project.

5.2 Agile

The successful arrival of agile principles has had an effect on the vision on organization of test design and automation. Testing activities are now much closer to the development ones, and in a method like scrum, responsibility is shared in sprint teams. Action Based Testing fits well in an agile project, and often allows for getting a good deal of test development and automation done in the same sprint as the application. It also allows for easy outsourcing of remaining testing and automation work.



Start a sprint with higher level test modules that cover the work items taken up by that sprint. The actions and checks in those modules should be at a similar level as the user stories and acceptance criteria.

Later in the sprint, when typically more final UI designs and implementations become available, add some test modules to test interaction and UI.

Also identify any relations between the work items in the sprint and the remainder of the application(s), and develop tests for them as well. We refer to this as "cross over test modules".

Aim for a close cooperation between the members of the team. There should be agreement on what approach to use for test development and automation, and who can do what to help.

In particular developers should help by providing "testability" features. See for this also the next chapter on automation. In particular the developers should decide early on automation ID's for UI entities and element. Interface definition can then be created early on without even the need of a viewer, and they will remain largely current throughout the sprint.

Stick to the three layer structure of actions: low level, mid-level, high level. A mid-level action that enters the fields for a single dialog can easily be updated if a UI changes. High level actions should mainly use mid-level actions for their implementation.

To make the test automation in a sprint flow well, an organization should first have any technology problems resolved. In most organizations, the automation experts will have their own process to identify and resolve technology challenges (for example using a Kanban approach), and share their time between sprint teams. If a technical hurdle occurs (like an inability to do an operation on a specific control), treat them as impediments, and ask the automation expert to address it, as well asking for their assistance in creating a temporary workaround so the sprint can continue.

A good option for fitting ABT testing in a scrum project is outsourcing. This can help the team keep up with the application, with the benefit of having ample tests and automation available to be re-used in future sprints. There are roughly three models:

- Fully outsourced: there is a complete team that carries out the system, test and/or automation development tasks
- Fully integrated: the outsourced team-members participate in the sprint. If in a different time-zone they can continue ongoing efforts and have them available the next day
- "Second unit": one or more members in the onshore team directly send work-items to the outsourced team, thus speeding up the process of test and automation development

*** discuss agile facility

5.3 When to develop and when to run tests

Test modules can be developed at any time in the life cycle. Typically it is good to start a test module only when the underlying system knowledge is available. For higher level tests with business functions this can be fairly early, but UI specific tests have to wait until the UI design is in a more or less in a stable stage. Usually this doesn’t occur until the final stage.

Modules should only be executed when the system under test is "ready" for them. For a higher level modules, like functional tests, this means that the lower level tests should have passed, in other words the UI should be stable. If errors or warnings appear that certain dialogs or controls were not available, consider it as a sign that lower level tests should have been (developed and) run successfully first.

The organization of test modules in their folders, like described in the template in the appendix, is the result of a design effort and more or less static over time. However, it does generally not define which tests to run when, and in what order. For this another concept "test suites" is available. The suites can be a predefined series of test modules (or test cases within them), or they can be "query based", meaning all tests are executed that fit a specified criterion. In ABT, the test development is based on the test modules. More traditional concepts, like "smoke test", "regression tests", "build acceptance tests", and "functional acceptance tests" generally don't play a big role in the test designs, but they often figure as suites.

6. Miscellaneous Techniques

TestArchitect has a number of features that can help the efficiency and manageability of your projects.

6.1 Variations

TestArchitect offers an extensive set of features to handle differences that are caused by different versions or an application or different configurations in which it needs to operate.

Examples are:

- Different languages or localizations
- Versions of the applications or its components, like 1.1, 1.2, 1.3 beta 2, etc
- Operating systems, browsers, databases, etc

The variations are mostly used to help actions and interface mappings to handle the differences without impacting the tests. However, they can also distinguish different versions of tests and data sets that they use. For more detailed explanation of variations, refer to the TestArchitect documentation.

6.2 Project Subscription

In TestArchitect, a project can subscribe to another project, which we then call the "supplier project". The subscribing project can use the actions and interface definitions from the supplier project, and recursively from any projects the supplier project itself might subscribe to. Projects may also subscribe to each other.

Project subscription allows for re-use among projects, and it helps organize which team is responsible for which items. The most common uses are:

- Multiple systems that use a common core system or library
- Certain actions that are used and maintained as "utilities" for multiple projects
- Multiple applications or components that fit in in a common UI framework (as "plug-ins")

For more information about the subscription feature see the TestArchitect documentation.

6.3 Regular Expressions

In several places TestArchitect will accept regular expressions that can help make values more generic.

Regular expressions are enclosed in braces {..}. For example "{customer.*}} will cover "customer a", "customer b", "customer 123" etc. A dot means an arbitrary character and the asterisk means "zero or more times", so ".*" means a number of arbitrary characters. For a more detailed explanation on regular expression, see the TestArchitect documentation.

You can use a regular expression in quite a few places, like:

- In menu items
- List items
- Interface definitions (for example "{.* - Notepad}" as a window title instead of "Untitled - Notepad"

6.4 Graphics and multi-media

In our projects we frequently test graphics. These can be

- Pictures, like photos, logos, icons or other symbols
- Other media like video clips and sound fragments

- Charts like pie or line charts, or histograms

For pictures and media items most tests come down to:

- Should the correct item displayed or played?
- Is the item itself correct?

For charts it is usually about:

- Is the data that is presented correct?
- Is the presentation of that data correct?

Technically accessing pixels of graphics displayed on a screen is actually fairly straightforward. The complexity however lies in the test design. It is easy to detect for a test that a pixel is certain level of green, but much harder for a tester to figure out that it indeed should be green.

6.4.1 Check picture

The "check picture" action in TestArchitect allows you to verify a picture only once. The next time the test encounters the same picture that was approved before, it will automatically approve it again. See the TestArchitect documentation for more details.

When using picture checks, keep in mind that various factors can "spoil" the picture check: making the picture slightly different from a previous run. If that happens:

- Find out why it is different? could it be avoided, for example by stricture definition of test data or environment, or by eliminating random rendering effects by the application under test
- Is it possible to define comparison criteria to allow for differences? Note that this depends highly on the specific test situations. For example a tolerance value for the maximum percentage of differences can work well for a photo, but not for a line chart (even if a line is thoroughly different, you may only see a small amount of pixels being different)

For a chart (graphical representation of data) try to separate the following in different test modules:

- The underlying data. Ideally there should be white box functions to test these independently from the display. Ask your developer for these, rather than trying to reverse engineer what is displayed
- The representation. Given certain data, does the system show the correct graphics? Here also developers can help, by providing function to explicitly set test data
- The manipulation functions for the graphics, like zooming, rotating, slicing, changing color etc. To help TA to match the resulting images the next time you run the test, see if you can use the keyboard emulation to drive the manipulations, rather than the mouse emulation, which on most systems is less deterministic

TestArchitect offers two ways to store a picture:

- as an "absolute" check, in which case the name starts with"/"
- as a "relative" check, the name does not start with "/"

Absolute checks are stored in the "Picture Checks" folder in the project. All test modules (and actions) can refer to them. An example is an icon in the TestArchitect explorer tree: if a test module is checked out the tree should show a checked out test module icon. The design of that icon only has to be checked once, all other tests can just verify that that icon is displayed when the tree node is about a checked out test module.

Relative checks are stored in the "Picture Checks" tab in the test module. They are typically describing charts that depend on the data being used to generate them, and are therefore dependent on the test module verifying them.

6.4.2 Media items

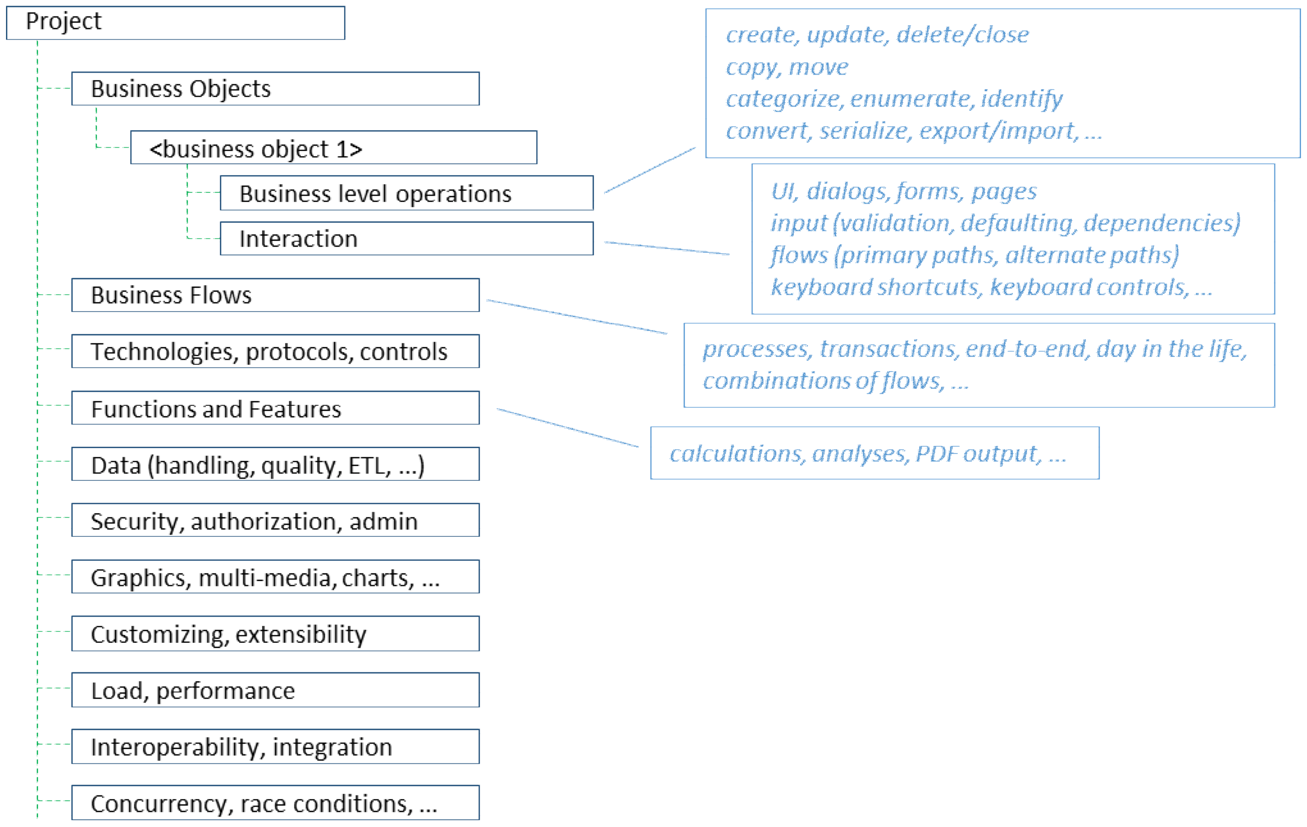
Web pages will show various media items like pictures or audio and video clips. Many of these are file based, accessed by their URL. For these kind of situations a strategy similar to check pictures. However, rather than capturing a picture from a control (or screen), consider looking at the file:

- is the correct file shown or played
- is the file correct

Let's say a web application is showing a speech of the company's president, does it (1) link to the right file, like "speech.mp4", and (2) does that file indeed contain that speech, and not for example some-body's vacation video. Section 6.4 discusses a way to deal with graphics. A similar technique also works for example of the president speech, where the checksum of the "speech.mp4" can be used to know whether the file has not changed since it was verified the last time. A test can then just verify that there is a link to the correct file.

Appendix A - High Level Test Design Template

In chapter 2 some criteria were described on how to identify the test modules. This appendix gives a possible organization, mainly meant as an example.


Project
Business Objects
<business object 1>
Business level operations
Interaction
Business Flows
Technologies, protocols, controls
Functions and Features
Data (handling, quality, ETL, ...)
Security, authorization, admin
Graphics, multi-media, charts, ...
Customizing, extensibility
Load, performance
Interoperability, integration
Concurrency, race conditions, ...
create, update, delete/close
copy, move
categorize, enumerate, identify
convert, serialize, export/import, ...
UI, dialogs, forms, pages
input (validation, defaulting, dependencies)
flows (primary paths, alternate paths)
keyboard shortcuts, keyboard controls, ...
processes, transactions, end-to-end, day in the life, combinations of flows, ...
calculations, analyses, PDF output, ...

A 1 Business Objects

Many tests can be assigned to "business objects", the items an application keeps data about, and form the ingredients of its operations.

Examples are customers and cars in a car sales system, but also orders for cars, or invoices about cars.

A.1.1 Business level operations

Business objects have a lifecycle: they're being created, updated or removed. A common concept is the "CRUD": "Create, Read, Update, Delete":

- "Delete" often is more like "marked inactive" or something similar.
- "Read" can translate into "use"

Business object undergo other operations as well:

- Related to other objects (a car is reserved for a customer),
- Put in and out of sets or containers (a car is put in a tree with car brands and models)

- Copied, moved, exported/imported, converted, enumerated, serialized, etc

Objects also have multiple kinds, like "cars" can be consumer cars, trucks, buses, etc. A useful testing technique that can help addressing these is "equivalence partitioning".

A.1.2 Interaction tests

Many of the tests in a typical project handle the UI and other interaction aspects of working with business objects.

Depending on how many tests you think you, you may want to consider splitting the interaction tests further they meet any of these criteria:

- If there is more than one dialog (form, page, ...) to create a new object form, consider testing each form individually
- Input handling of values, like validation, defaulting and dependencies
- Specific UI behavior, like enabling a button when fields have been populated
- Flows (primary paths, alternate paths)
- Keyboard shortcuts and controls, for example <ctrl> C to copy, or the way the tab key moves through a dialog

A 2 Business Flows

Most systems process events going through multiple steps involving more than one business object. The data is passed along, sometimes splitting, and/or recombining, and as a result this usually changes the data in the database.

In particular when transactions influence how other data is processed one can expect issues and how tests can be an aid. . The kinds of tests in this category would be known as end-to-end, integration, end-user, day in the life, etc. They will mostly be written as business level stories with application level actions.

As an example take a company that renders services and sells products. A customer might order some products, and services for a project that runs for several months. This will create an order (business object). Later on, the same customer might order some additional work resulting in another order. The customer gets invoices (also business objects) that may combine one-time product and services costs from multiple lines. Reversely one order may be billed, over time, on multiple invoices. The customer then, hopefully, pays one or more of the invoices, and these payments (business objects) will be processed in the financial administration. Managers and other stakeholders can view management information regarding orders, customers, and the financial situation, that need to reflect the business events that have happened so far.

Another example where business flows influence each other   at a car rental company. There might be transactions for people renting cars, while at the same time there the inventory changes, due to new cars being acquired and existing cars needing services or being retired. These inventory changes will influence the rental ordering, and rental orders influence whether a car can be serviced or not.

A specific example of a flow can be a session of an internet user. The user may browse some pages, search for an article, add some personal information, place an order, pay be credit card, check back on the order status, etc.

A 3 Features

Systems often have functionalities that involve little or no data changes, but that are available to users or processes and need testing. Examples are inquiries, calculations, printing or export features, etc.

An example would be a mortgage calculator, where a user can enter some fields, and get a preliminary estimates of monthly payments. Or a system may have a reporting feature that also includes PDF output.

A 4 Interoperability

Many systems do not work in isolation. They exchange data and events with other systems or components. While cooperation of components within a system might be covered in the business flow tests this category would cover the collaboration with other systems.

Tests in this category might be using the other system "live", or might emulate its contribution. This emulation could be expressed with actions. For example, a financial system could interact with a credit agency. The tester could write an action line to send a credit score request to the agency, but let the application send this to an emulation process. Then the next action could verify whether the request was received by this emulator, and follow it with another action to tell the emulator what to respond, after which a check can be made whether the application reacted to this response correctly.

A 5 Components

Modern day systems are typically built up of many classes, libraries, components and tiers that have functions, methods or services that can be invoked individually. Though many tests will go through the UI, interacting with these internal items can be useful to assess their individual correctness, and to make actions that are faster and more maintainable than the ones going through the UI.

However, the UI is where all parts of a system come together, and testing through the UI will most likely retain a prominent part of the total testware.

A 6 Graphics, multi-media

Many systems have non-textual elements like graphics and multi-media. Given their complexity, it is usually a good idea to separate tests targeting those in separate test modules.

Section 6.4 discusses some strategies to deal with graphics and other media items.

A 7 Technologies

Systems may implement and use generic technologies to support their functions. Examples are standards, protocols or custom controls. When you test the system it will use these technologies. However, you may also consider testing these technologies in isolation.

For a protocol for example you may want to know if the system generates correct messages in given situations, and can also handle incoming invalid messages without breaking.

Another example could be an in-house created spreadsheet-like table control class that is used in various places in an application. It may have functions like sorting, filtering, inserting rows, modifying cells, resizing columns etc. It makes sense to create a separate set of tests to test many of the features of this control class in-depth, instead of embedding too many of the tests in every location where the control class is used. You can achieve better coverage, and most, or all changes in the control class can be accommodated by these tests alone.

A 8 Data

Data in itself is less often a target of tests, but if it is a target, it can be a category to distinguish.

Most databases will have a set of integrity rules that describe what data is valid. It makes sense to run a number of events and after that test where certain rules, or any other criteria that a tester can think of like the following: does every US address have a state in the Union or do the zip codes match the addresses they accompany?

A particular area of interest for testing is "big data" in data warehouses that is extracted from heterogeneous sources, often external to the system under test. This data is usually added using a complex high volume process called ETL ("Extract, Transform, Load"), and testing can address both the ETL steps themselves (with test values), and do spot checks on the validity of the data that was stored.

A 9 Administration

Virtually every system has administrative functions. Even though bugs in such functions can have a large impact, they often do not get the testing attention that they warrant.

Among the key administrative functions are:

- Users and groups, which are in fact also business objects with lifecycles and UI's
- Authorization, that can be created, listed, updated and revoked. They often often have complex structures and business rules. Sub-categories are:
-
    - Authorizations life cycles
    - Effect of authorizations

- Specific security tests, e.g. when a user logs out of a website and then tries to continue working using the browser back button
- System management, like stopping, starting, backup/restore (you want to make sure to test restore, bugs here will hurt), installing, uninstalling, updating, etc

A 10 Customizability

Many systems allows various forms of customization by their users or administrators. For each customization possibility tests would be needed to:

- Make sure the customization can be made, updated and removed properly
- Verify that the customizations have the intended effect

Some examples of customizations:

- Settings, options, preference, configurations
- Fields to add to items in the system
- UI changes and adjustments, like moving or adding panels, toolbar buttons, etc
- Extensibility, to add functionality to a system, like plug-ins in a browser or IDE

A 11 Concurrency, race criteria

Applications that are used by more than one user, or that have processes that can run concurrently with users or each other, may show unwanted behavior if operations interfere with each other. For example one user might be working on a file while another user deletes the folder in which it is residing.

A particular form of concurrency problems is the "race criterion", where one user can accidentally overwrite a change made be another user at the same time. For example two users might have opened a form. User A might change a field, and store that change. However, user B still has the now outdated version of the form open, change another field and store the form, thus undoing the change made by user A.

TestArchitect supports a feature called "lead deputy" that allows testers to design tests that emulate two or more users (and/or processes) working concurrently. Since such tests will then require two machines (physical or virtual), it is best to dedicate separate test modules to concurrency testing.

A 12 Non-functional tests

Systems have aspects that can be objects of testing, but do not necessarily leant themselves to functional value based tests.

A.12.1 Load and performance

Actions (and TestArchitect) are not commonly used for load and performance testing. A possible use is to use a load generation tool to generate a specified load, and then let an action-based test execute the test to see if the application under test still works well during the load and has an acceptable response time.

For example you can have two actions: "get time" and "check response time", and use them like this:

start clock count		
	window	item
select menu item	notepad	Help->About Notepad
	window	
wait for window	about notepad	
	maximum	
check response time	8	

In this example the "check response time" would fail if since the time elapsed since the latest "start clock count" (available in TA) is more than 8 seconds.

A.12.2 Environment disruptions

An application depends on many environmental conditions, like hardware, networks, setting files, libraries, storage devices, other systems, etc. Problems or disruptions in any of these can have an impact, and the application should handle that gracefully.

Tests that emulate disruptions and assess the application responses will require deep understanding and thinking on the part of the testers, often in cooperation with developers. Once devised they are not necessarily hard to write down with actions, but the implementation of the actions may take a significant technical effort. Virtualization may help create conditions like network disruptions or resource denials.

A.12.3 Documentation

Documentation needs testing too. Language and layout needs to be correct and consistent and application behavior and functionality need to match what is described about them in documentation. The documentation must also be complete: cover all the functions the application is offering.

The correctness of documentation is not commonly a target of automated tests. However, mismatches in documentation can be painful and make a bad impression on users. One approach to consider would be linking documentation items to test modules. If the test module changes the related documentation should be updated, and if the documentation changes (functionally), the corresponding test modules should be reviewed and executed.

Help

For Help functionalities automated tests can be designed to make sure the correct help screen or page shows up when the user presses a help key or clicks a help link. Automated tests are particularly well suited for this, since verifying help functions can be tedious and can easily be overlooked.

Appendix B - Anti-patterns of test design

Note: this text largely comes from an article written for Techwell. Contributions by practitioners and others are added over time.

In our “Action Based Testing” method we organize tests into easy to manage “test modules” that look like spreadsheets, which we edit and manage in our tool, TestArchitect. The test modules contain test cases that are written as sequences of “actions,”—spreadsheet lines with an action keyword and zero or more action arguments.

Since test design can have a big impact on automation, several people have asked if there might be “anti- patterns,” situations to look for in a test that potentially could be harmful for their maintainability and scalability.

Wikipedia describes an anti-pattern as “a common response to a recurring problem that is usually ineffective and risks being highly counterproductive.” The term was coined by Andrew Koenig, based on the well-known notion of “design patterns.”

Just like with design patterns, anti-patterns can benefit from a short and catchy name to make them easy to remember and talk about. Below is a list of typical situations I have seen in tests that I think can harm automation These anti-patterns often occur in combination and can alsohave some overlap as well.

Enter Enter Click Click:

Problem: Test steps are too detailed.

Many tests, even automated ones, have been designed on a detailed step-by-step basis. This makes it difficult to manage and maintain those tests. For example, it will be hard to factor the impact of changes in the application under test into the tests. Identifying common sequences of steps and putting them in actions or functions will relieve this problem.

Interaction Heavy:

Problem: Not having many business tests.

A main distinction I recommend that testers make is between "business tests" that focus on business objects, rules and processes, and "interaction tests" that focus on the interaction with the application. However, I've seen in many projects where testers focus only on the interaction. This makes the tests shallow and misses potential business level problems.

Lifeless:

Problem: Missing life cycle steps of business objects.

Most applications work on "business objects," like orders, invoices, products, customers, etc. These objects have their life-cycles in the application, like creation, update, retrieval and closure, and also operations like copy, move, export, import, etc. However, the tests for such basic operations in applications are often scattered and as a result, hard to find and incomplete.

Lame:

Problem: No depth or variety, no testing techniques used.

Time pressure and other factors often result in shallow test cases that don't challenge the application much. Try to think as a tester, somebody who wants to break things. Applying testing techniques and interaction with various stakeholders can help you in this process.

Clueless:

Problem: No clear scope for the tests.

A very common situation is lack of scope for tests. The tests then are hard to find and assess, and may do work that is already done in other tests.

Over-Checking:

Problem: Checks not relevant for the scope.

Since test designers often follow an approach of steps with an expected result for each step, tests do many checks that do not fit the scope of such tests. Such checks are unnecessary and probably overlapping similar checks elsewhere. They then clutter up result statistics (e.g. they create too many "passes"), and aggravate the impact of changes in the application under test.

Cocktail:

Problem: Interaction tests are mixed with business tests. Even if tests are testing business functionalities, like business object life cycles and business rules, calculations and processes, they are often mixed with tests that deal with interaction details, resulting in a convoluted and hard to maintain mix. A common example is to describe a log-in process in detail in all tests that start with a log in.

Sneaky Checking:

Problem: Checks hidden in actions.

Even though it is good to have business level actions that hide unneeded details for many of the tests, try to avoid hiding too much. In particular, checks should be explicit and visible in the main test (the test modules), at the appropriate level of detail. An outsider should be able to understand what is being tested by just looking at the test module, without a need to inspect how actions are implemented.

Action Explosion:

Problem: Many actions, with little re-use.

Some testers may have interpreted a statement like "actions are good" too literally. In tests one then sees actions for every little step in a test. The result is many (thousands) of actions, and even though actions are meant to ease maintenance, they themselves become hard to manage.

Mystery Actions:

Problem: Actions should have clear names, representing their function.

In some projects one can find actions like "verify transaction compliance", and without clarification they can't be sure of what that actually means.

Techno:

Problem: Actions and tests that look like code, using camel case or underlines, and are therefore often _NOts0EasY_2REad, in particular for non-technical users.

Endless:

Problem: Long winding test cases with many steps.

In some projects the amount of test lines in a single test case number in the thousands.

Swiss Army Knife:

Problem: an action that has more than one clear function or does more than one thing.

Consider splitting it up in multiple single-purpose actions.