exactpro

EXITUS ACTA PROBAT

Test Automation Principles

VERSION 4.0 – DECEMBER 2009

Contents

1 Introduction

Test automation usually reduces the overall testing effort by decreasing the time which QA analysts have to spend on repeatable tasks. Quite often automation also adds more depth to the testing process without reducing the actual effort.

This document sets out the principles and recommendations to be considered when starting a test automation project. It is based on our experience with testing matching engines, smart order routers, order management and execution management systems, straight through processing, risk management and position keeping systems and front- middle- and back-office platforms. We describe items we believe will have a significant impact on the effort involved in creating test harnesses and comprehensive libraries of automated tests.

1.1 Targets of Test Automation

It is critically important to identify correctly only those test scenarios which are suitable for automation. Usually, the automation team uses manual test scenarios as the basis for automated test scripts, which can be a simple process when using a UI screen manipulation framework, such as HP Quick Test Pro or Borland Silk Test. However, manual tests are designed and optimised for human execution; on the other hand, they mimic business user manipulation with specific UI screens. It is difficult to automate testing of the UI itself because, in real life, UIs are frequently changed and often have advanced features that are hard to automate.

When working on the automated regression library one has to consider not only UI interactions but also business processes that happen in the trading back-end. Before starting test automation, spend time on the design of the test library and the efficient splitting of the business processes into maintainable atomic actions, such as orders and executions.

Keep up to date any documentation that describes the test environment configuration, the settings and the maintenance procedures.

1.2 Protocol Level Test Harnesses

Most trading systems have a distributed server back-end, connected to a set of other systems such as execution venues, market data delivery systems, clearing and settlement environments, risk management/position keeping systems, etc. In order to accomplish end-to-end testing that covers all major scenarios, all inbound and outbound flows around the trading environment under test must be emulated. This requires building a test harness capable of simulating connected systems. Identifying the right tool for this task is critically important. The tool should support the test scripts in a form that can be understood and interpreted by businesses analysts and QA engineers outside the test automation team. Test tools aspects are covered in more detail further in this document.

1.3 Common Test Automation Problems

It is vitally important to understand and be prepared to resolve problems and obstacles that may obstruct the work of the test automation team. The main problems are the instability of interfaces through which automated scripts interact with the system under test, the lack of the control when working in the test environment and the lack of commitment to support automated tests. These and other potential issues are covered in Chapter 4 of this document.

1.4 Test Automation Teams

Test automation is a combination of development and testing processes. A special approach in terms of roles and staffing is required to accomplish this. Usually test automation is a task for young IT people who have roles somewhere between QA and software development. It is perfectly acceptable to use such people for test automation, but a senior software engineer (or architect) and QA analysts must be involved in order to manage and lead the automation process in the right direction.

It is critically important to ensure that prepared test scripts will not require much effort to update them for newer versions of the trading system under test. It is also necessary to make sure that the scripts actually do the correct actions and verifications from the QA analyst's perspective. Another important point is to ensure that scripts' output allows researching issues quickly and determining whether an identified issue is due to a defect in the test script or in the application under test.

More details on teams, processes and resource estimation are provided in Chapter 5 and Chapter 6 below.

2 Test Automation Overview

Test automation is usually considered as the means of reducing the costs and/or increasing the scope of regression testing (i.e. testing in order to ensure that changes introduced as the result of coding new features did not break old functionality). Thus, the same tests have to be executed against the same functionality for each new release, and it makes sense to prepare test scripts that will do this work automatically.

This works best if the regression library can be broken down into a large number of relatively simple isolated test scenarios. Each scenario should consist of several steps, i.e. actions against the system under test and verifications of responses as well as the resulting system state. For example, if you want to test a search engine within a web site, you will have a scenario consisting of the following steps:

- Enter search criteria and click “Search” button;
- Read and verify the results list; e.g., we may check the number or the results or specific pages that must be returned;
- Iterate through obtained links and verify that they work.

After one has implemented such a scenario, automated testing of thousands of test cases becomes possible by just selecting different search criteria.

For test automation, it is critical that the environment is completely controlled by the automated test scenarios. If it is not, automated scripts will not be stable enough to achieve the benefits of automation as compared to manual testing. Ideally, all system components and data should be controlled by automated tests.

Test automation can also be useful for functional testing of newly developed components. This is usually done in three cases:

1. When automation is undertaken for the same reason as regression testing, i.e. if it is necessary to run multiple test cases, multiple times, against multiple versions of the component throughout its release process. The effort to automate this work and maintain it later costs much less than manual testing;
2. When you have to iterate many similar tests varying just a few input parameters, which is similar to the search engine test described above;
3. When testing a system exposed via an API or a network protocol that cannot be interacted with manually, in which case the benefits of automation are obvious.

Test automation ***is*** recommended:

- When testing is focused around business logic or business rules;
- If one controls the test environment, including all of its external feeds, connections and data storages;
- When the system is exposed via an API or a network connection with a specific protocol;
- When it is possible to develop a few simple scenarios with many variations of input parameters to cover the main system functionality.

Test automation is ***not*** recommended when the testing is concentrated on the UI.

3 The Components of Test Automation Solution

Automated testing suites for trading systems consist of a variety of components, including the following:

3.1 Test Script Development Environment

Usually testers use tools to develop and execute automated test scripts. One of the standard tools is the HP Quick Test Pro Utility, which allows the development of test scenarios using VBScript language. It enables testers to capture UI application lay-outs and to write the test code which uses this recorded layout to manipulate windows. The tool also has built-in capabilities to script test scenarios which produce pass/fail reports as an outcome.

Another example would be FIX/API testing tools. These tools allow developing automated tests using the FIX protocol or a custom API. They also allow preparing test scenarios that involve the invocation of the web services, responses validations and generic java actions. Most of them also have built-in capabilities of scripting pass/fail criteria.

In general, an automated test tool is a program which allows testers developing test scenarios to use a programming language or pseudo code. Usually these tools allow the following:

- Record human actions when he or she is working with the application;
- Write scripts which simulate human actions and capture screens or application responses;
- Run and debug such scripts;
- Generate test execution reports;
- Use external Excel spreadsheets to store test data.

3.2 Test Scenarios

Test scripts contain the logic of tests. They can be either programs or spreadsheets defining the steps of the testing process. Test scripts should be kept in a human readable format, because QA or business analysts should be able to read and understand them.

3.3 Prepared Test Input Data

Test data is a crucial part of all automated test suites and comprises the following:

- Different securities and their parameters;
- Different trader accounts and their settings;
- Market Data all securities featuring in the test scenario;
- Other thing which may affect trading business logic.

Automated tests are heavily dependent on test data. Part of it can be stored in Excel spreadsheets within the test script. The test script developer is responsible for maintaining accurate and consistent test data. Another part of test data is stored within databases; the person running the test script is responsible for maintaining this data, including appropriate user accounts, etc.

3.4 Action Based Test Engine

Test cases for trading systems consist of a small number of clearly defined transactions, i.e. orders and executions. It is useful to present test scenarios as sets of order management requests, market data manipulations, venues responses and system responses verifications. It is also recommended that the test script be split into two major parts: (1) a library with generic actions (place, order, cancel, amend or verify the results); and (2) actual test scenarios defined in spreadsheets.

Requirements for test automation frameworks, which are used to run the test cases described as a set of Excel spreadsheets, normally include the following:

- Each line in the spreadsheet is a single action within the test case;
- Usually the spreadsheet contains multiple test cases;
- Each action is function with a set of input parameters which returns a set of outcomes;
- Consequent actions may re-use previous actions, parameters or outcomes. For this, the tester should have the ability to place in one of the cells an expression that instructs the extraction of values (either inputs or outcomes) from one of the previous actions;
- The input data values for each test case may be put into a separate spreadsheet(s). Each step may contain a link to a row within the input data table within the data spreadsheet(s);
- During the test, everything should be logged into a results table within the reporting database or a log file. The tester should be able to inspect the results at any time during or after test execution.

3.5 Emulation of external systems

Distributed trading environments normally have multiple interfaces to various external subsystems, such as execution venues, market data delivery and back office services. Often, such external subsystems are not available during testing hours and mostly they do not offer enough features and control either, which testers require in order to conduct meaningful end-to-end testing. Due to that, one of many test automation tasks would be to recreate the functionality of such external systems. This is normally achieved by building emulators allowing the testers to have full control over venue settings, market-maker responses for market venues or current prices for market data.

On the back-end side of the trading environment, one needs to emulate market data, exchanges (market makers) responses and operations/accounts events.

Emulation of Order Execution Connections

The Order Execution Emulator mimics exchange or trading venue responses. It can simulate either responses on incoming requests, such as acknowledgements or unsolicited messages (fills or cancels). The tester should have control either by specifying the response modes or by sending messages on behalf of the exchange. The emulator also needs to sustain order states in order to test different scenarios, including multiple manipulations with the same orders, such as amendments or cancellations of a partially filled order.

Emulation of Market Data

This should be a stand-alone tool which emulates prices coming into the trading system and connects to the Market Data Dissemination System (Ticker Plant) for the prices which come from the real markets.

Emulation of Operations or Account Events

This tool should emulate different events that may be issued by Operations, account changes or events coming from Back Office applications. For example:

- Funds in/ Funds out;
- Position in/out;
- Account attributes updates including:
    - Margin feature;
    - Option approval level;
    - PDT flag;
    - Restrictions;
- Margin Calls; …etc.

4 Common Problems with Test Automation

This section is our attempt to summarise the most common problems and risks that may be encountered when performing test automation projects and discusses possible solutions.

4.1 UI Controllers are Unstable

UI test automation tools interact with windows controls within a front-end application. There are two major problems with this approach. Firstly, scripts developed with this technology are tied in to the current version of the trading desktop's UI and can easily be broken when the latter is upgraded, e.g. due to the order of the controls on the screen. Secondly, there are many different UI frameworks (such as ActiveX, .NET forms), against which UI controllers often freeze or crash. Therefore the scripts developed on such tools may require significant maintenance and execution effort. Sometimes constant involvement of the test engineer is required to watch the script execution and to restart it if a problem occurred.

The instability of UI controllers can normally be resolved as follows:

- Most trading back-ends can be handled via API or some network connection, so the use of UI controllers can be avoided whenever possible;
- Experienced automation engineers usually know how to make scripts more stable, so it's advisable to hire someone experienced in using a UI controller that corresponds to your specific UI technology;
- Use different recovery scenarios. If QTP or similar tools hang, just kill them automatically and restart the script.

4.2 Trading Systems are Connected to Exchange Test Links

Trading systems are normally tested using test order management links provided by exchanges or execution venues. These links are shared between all exchange participants and thus it is difficult to predict the behaviour of the instrument order book when using such links. This makes test scripts unstable and hard to execute.

Possible solutions include:

- Developing an emulator instead of using the client link. This is not a good idea if one needs to test a gateway to exchange, because it is better to test this against the software provided by the exchange. If that is not the case, one can develop a simple emulator of exchange connection in order to isolate the automated tests;
- One may try picking a rarely traded instrument, hoping that nobody else will use it for testing and build a special algorithm in one's automated script that prepares an order book for each test case.

4.3 Test Environment is Unstable

Sometimes it is difficult to run automated tests, because the test environment is unstable. Tests fail for no obvious reason due to environment problems.

Possible solutions include:

- Try to delegate the control of the test environment to your automation engineers, so that environmental issues can be investigated and fixed quicker;

- Develop a specific environment monitoring script (or a health check script), which will proactively monitor the environment and notify the automation engineers if a failure happens or will even restart the automated script and the test environment automatically.

4.4 Test Execution is Too Slow

End-to-end order management test libraries contain thousands of test cases. If you use UI controller tools (such as QTP), it may take a few minutes to execute each test case, and only one script per workstation can be run. As a result, to execute the whole regression library, a significant amount of time and/or hundreds of workstations may be required.

Possible solutions include:

- Try a non-UI approach, i.e. injecting orders via an API or network connection;
- Allocate enough hardware and prepare dozens of virtual boxes to execute test scripts;
- Prioritise your test scenarios and run only those that have top priority.

4.5 New Version of Software Under Test Breaks Test Scripts

It is not unusual for automated scripts to break due to changes introduced in a new build of software under test. If this happens often, and the effort required to fix the scripts is substantial, one may end up spending more time on fixing the scripts than actually executing the tests.

The solutions include restructuring the testing framework in order to isolate pieces that change quite often. The best practice is to have the business logic defined within simple spreadsheets containing the list of actions. If a modification of the software under test breaks one or more actions previously defined in such a list, one is only going to have to fix the localised issue with specific action(s), while all the other code of the automated test scripts can be left untouched.

5 Teams & Processes

This section describes the processes related to test automation activities. It covers automation framework development, automated test scenarios design, scripting and execution.

5.1 General Description

The following activities are involved in test automation:

- **Test Scenarios Business and System Analysis:** this is where it's decided if it is feasible to automate specific scenarios and come up with a technical design;
- **Development of Test Engine:** it is necessary to ensure that the Automation Engine is capable of supporting the test logic and all required actions and validations;
- **Development of Test Cases:** i.e. the preparation of the test cases, including the test matrices and input data;
- **Trial Run of Test Cases :** a dry run of the test cases in a test environment is needed in order to ensure that they work; issues with the test cases are investigated and fixed;
- **Execution of Test Cases:** this is an ongoing task of executing the existing test scenarios cases and analysing the results in order to determine if the functionality under test is working correctly;
- **Ongoing Support of Test Cases** (if a software upgrade occurs): major software upgrades usually break or invalidate a lot of previously developed test cases; the purpose of the ongoing support process is to fix such test scenarios as early as possible.

The following automation teams can work on the activities described above:

- **Test Engine Team:** the team designs the tests, develops the Test Framework and modifies the framework or scenarios when major software upgrades happen. This should be a solid, stable team of automation and QA engineers who are responsible for the overall process of test automation;
- **Test Cases Preparation Team:** this team implements and verifies the test cases. Its size and composition may vary depending on specific needs. These are more or less junior and temporary engineers and testers who are responsible for preparing new tests. This team is also a "boot camp" for specialists who are going to continue pursuing development and test automation as career paths;
- **Test Cases Execution Team:** this team is responsible for ongoing test execution, issue research and reporting. This team consists of regular QA persons who are capable of executing test scripts and analysing the output.

Overall, the recommended lifecycle of automated test scenarios is as follows:

- Business Analysts or QA come up with the functionality to be covered by the automated tests. This can be either a functional specification or an existing manual test script. This task goes to the Test Engine Team;
- The Test Engine Team performs the initial analysis and comes up with the following recommendations:
  - Whether it is feasible to automate the functionality;
  - What effort in terms of updating the test framework will be required;

- If the effort estimate is confirmed by QA, the Test Engine Team starts working on design and core improvements. The outcome is as follows:
    - Automated Test Cases Technical Specifications;
    - Automated Test Cases Coversheet;
    - Updated Automated Test Engine (or new actions);
- Next, the Test Cases Preparation Team starts working on the test scripts. They are responsible for coding and performing the trial run;
- During the implementation or the trial run, the Test Engine Team assists the Test Cases Preparation Team with researching issues encountered when running the test scripts (is it the framework or the system?). In addition, the Test Engine Team verifies that the test scripts are implemented as designed and that they generate relevant test results and technical logs;
- When the script implementation and the trial run is finished, the test script is added to the automated regression test library and the Test Cases Preparation Team reports any defects they find into a bug tracking system;
- During subsequent regression cycles, the Test Cases Execution Team runs the scripts and the Test Engine Team helps resolve problems with the test scripts, if any. For major software upgrades, the automated tests may need to be updated accordingly. The following tasks are the joint responsibility of the Test Engine and Test Cases Preparation Teams:
    - Analyse possible impact software upgrades have on the automated scripts;
    - Fix actions or framework accordingly;
    - Re-run tests or report issues;
- During test runs, the following metrics regarding the test script are collected:
    - The number of defects found in the script;
    - The average effort (person/hours) needed to execute the script;
    - The time needed for test execution and the number of test cases that can be executed concurrently;
    - How often the script breaks due to code updates in the system under test;
    - How much effort was spent to update / maintain the script.

If the test script shows poor performance according to one of these metrics, the Test Engine Team makes a recommendation if the test script needs to be re-designed.

5.2 Test Scenarios Business and System Analysis

Normally, a small team (2-3 persons) of QA automation analysts handles this task. These analysts understand the technical aspects of the automated scripts as well as the business logic of the test scenarios and cases that need to be automated.

This team receives either the test scenarios or functional specifications that need to be covered by automated tests. It analyses these requirements and designs the automated tests. When this work is performed, the following questions are answered:

- Is it beneficial to automate the test?
- Do we need to enhance the test framework or add new actions/validations?
- How exactly will these tests be automated?
- What input data is needed to perform the test? How will this data be generated?
- What will the test reports look like? How will QA determine whether issues found are caused by script defects or defects of the system under test?

- What are the limitations of the automated scripts? What kind of issues will the script be able to detect? What kind of issues will the automated script not detect?

As the result of this work, two documents are produced:

1. **Automated Test Cases Coversheet** describing the test scenario, outlining how it is automated and defining what information will be present in test execution reports. QA Analysts and Business Analysts review this document in order to validate the expected benefits from automation and to understand the coverage of the automated test script;
2. **Technical Specification** describing the implementation of the test script and effort estimates in terms of person/days and duration.

5.3 Development of Test Engine

The second main task for the team mentioned in the previous sub-section is to provide overall support of the development and execution of the automated test library, which implies:

- Development and bug fixes of the Test Automation Engine;
- Development and bug fixes of low level functionality (actions) interacting with entry and validation points;
- Keeping existing scripts in the adequate state and ensuring that existing scripts are operational and provide valid results;
- Help with analysing issues found by the test scripts.

5.4 Test Case Preparation

A separate team of automation engineers is responsible for preparing the test scenarios and trial runs. Their main tasks include:

- Acceptance of the Automated Test Coversheet and design specifications from the Core Team;
- Implementation of the test scenarios;
- Running the test scenarios in the test environment;
- Analysis of failed actions; repairing the test scripts; reporting issues found by the test script;
- Certification with QA and the Core Team that the script does what it is expected to do and that it generates informative output, both in terms of business result and technical logs;
- Hand over the scripts to the Core and Transition Teams.

5.5 Test Case Execution

A team of QA engineers, trained in using automation framework, is responsible for:

- Gaining knowledge of and understanding the existing scripts;
- Understanding the business logic behind each test scenario;
- Executing the test scenario within the timeframe allocated for the test run;
- Analysing the outcome and reporting defects in the software under test;
- Performing test script or automation framework troubleshooting, if possible.

5.6 Ongoing Support of Test Cases

Automated test scripts often break when a new version of the system / software is released for testing. This is caused by UI, protocol, API or architecture changes needed to support new functionality. Preventive maintenance of the automation library is recommended in order to achieve efficiency of the test cycles. This is a task for the Core Team and it means the following:

- Understanding the scope of upcoming releases;
- Understanding what in these releases could break the scripts;
- Understanding what amount work needs to be done to fix the automation framework;
- Fixing corresponding actions and verifying them in some development / staging environment;
- Supporting the Execution team during the test run.

6 Test Automation Effort Estimation

Each test automation project is unique and requires detailed investigation prior to providing any meaningful estimates. The section outlines a few basic principles for estimating the effort required to automate testing of the trading system(s).

The first step is to review the system or application under test and carefully design the required test framework and test scripts. Normally, such a task takes 3 man-weeks of a Senior Systems Analyst working on-site.

6.1 Test Tool Costs

Depending on the test design, it may be necessary to purchase certain test tools. The cost of UI-based test automation tools would be about US $ 3,000 - 7,000 per seat and 3 - 8 seats might be required depending on the scope of work. Other test tools are FIX testing frameworks, such as Aegis or VeriFIX. They cost about US $ 30,000 - 60,000 for a site license, depending on which protocols and exchanges are supported.

Open source solutions, such as jUnit enhanced with QuickFix library, need development effort to customise them. The final cost depends on the system and specific testing needs. Generally, the effort required to build a test harness based on open source solutions is 8 to 12 developer-months.

6.2 Test Framework Costs

Test framework development requires a separate effort. Many tools (such as QTP) contain a test framework allowing scripts to be organised into test cases and the preparation of data driven test scenarios. However, such tools need to be enhanced in order obtain the action-based framework described in the earlier sections. This effort is usually about 3 - 4 developer-months.

6.3 Test Library Scope Assessment

When estimating the scope for developing test libraries, it is important to estimate how many “significantly different scenarios” are needed. For example, placing new orders and amending existing orders are significantly different scenarios. Placing two different types of order does not differ from the test logic point of view, unless you are verifying any special conditions associated with an order. When preparing a list of significantly different scenarios, the whole list of scenarios may be obtained by just iterating through different parameters of orders, types or instruments, etc.

The number of significantly different scenarios will be crucial in estimating the scope of work that needs to be performed in order to develop a test library as well as the effort required to develop automated tests and, later, to execute the tests.

6.4 Test Scenarios Preparation Effort

Test preparation effort can be estimated based on the number of significantly different test scenarios. It normally takes about 5 man-days to automate each scenario.

6.5 Test Environment and Test Lab Costs

Costs should be allocated for creating a separate test environment for test script execution, and additional hardware may be required for test execution boxes if UI controlling tools are used.

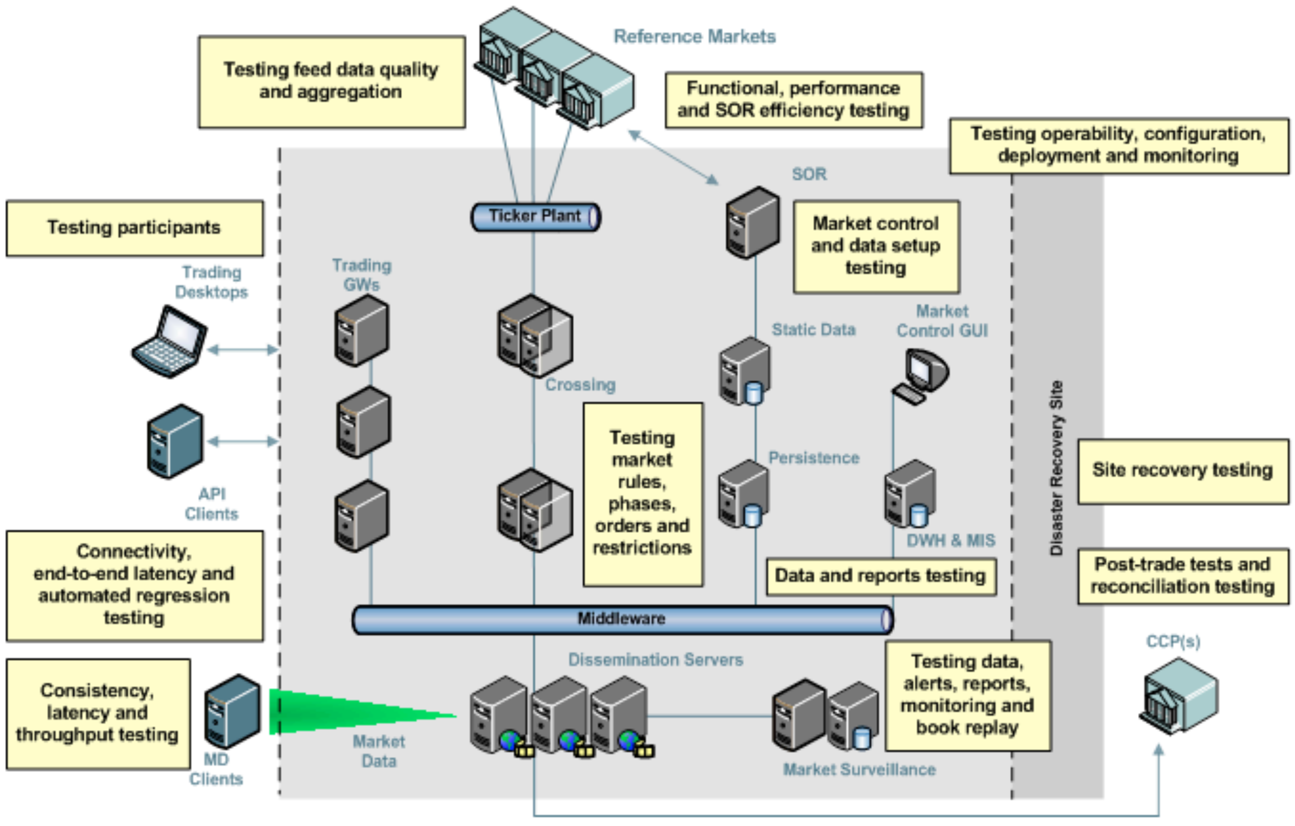
6.6 Test Execution / Maintenance Effort

Test execution requires a special effort, which is highly dependant on script stability and the time needed to execute the tests. During the first few runs, the entire automation team is required in order to execute the test scripts. When the test scripts become stable, it normally becomes possible to reduce the execution team to 25% - 30% of the original test script development team.

7 Typical Projects

7.1 Execution Venues, Dark Pools and MTFs

A typical execution venue, such as an MTF, a Dark Pool or an exchange, consists of many different sub-systems with different functionalities as shown on the following simplified diagram:


Reference Markets
Testing feed data quality and aggregation
Functional, performance and SOR efficiency testing
Testing operability, configuration, deployment and monitoring
SOR
Ticker Plant
Testing participants
Trading Desktops
Trading GWs
Market control and data setup testing
Static Data
Market Control GUI
Crossing
API Clients
Testing market rules, phases, orders and restrictions
Persistence
Disaster Recovery Site
Site recovery testing
DWH & MIS
Connectivity, end-to-end latency and automated regression testing
Data and reports testing
Post-trade tests and reconciliation testing
Middleware
CCP(s)
Dissemination Servers
Testing data, alerts, reports, monitoring and book replay
Consistency, latency and throughput testing
MD Clients
Market Data
Market Surveillance

The testing of some of these components is described in subsequent sections of this chapter. In this sub-section, we only cover the testing of specific functionalities, namely:

- Order entry via trading gateways;
- Market data dissemination;
- Order crossing and routing rules.

Execution venues accept orders via an electronic interface. The industry standard is FIX protocol, but some venues may accept other protocols or provide APIs for order entry. Along with the FIX connection (or API), the venue should provide rules of engagement and / or specs for developers of connectivity modules on the other side. Venues must also conduct certification of the applications to verify their compatibility with the gateway.

It is important that the gateway server works correctly and supports everything described in rules of engagement. The functionality of gateway servers can be covered by a set of automated tests. These tests emulate client applications connecting to the gateway, sending requests (orders, amends, cancels, etc.) and receiving responses. On the other end, orders will come to the crossing engine, so it is important to prepare for the test by placing an order book within the crossing engine.

A similar approach can be applied to market data dissemination testing.

The heart of an execution venue is its crossing/execution engine, which is often enhanced by connections to external reference markets, to which orders can be routed or whose prices must be taken into account during order execution.

Automated tests are required to test such complex functionality and the approach is to prepare a test framework allowing the test automation script developer to code the sequences of the following actions:

- Place/amend/cancel order on behalf of client;
- Emulate specific prices on the reference market;
- Read and verify responses from the gateway;
- Read market data updates coming from the venue, if available;
- Read internal state of the order book, if possible.

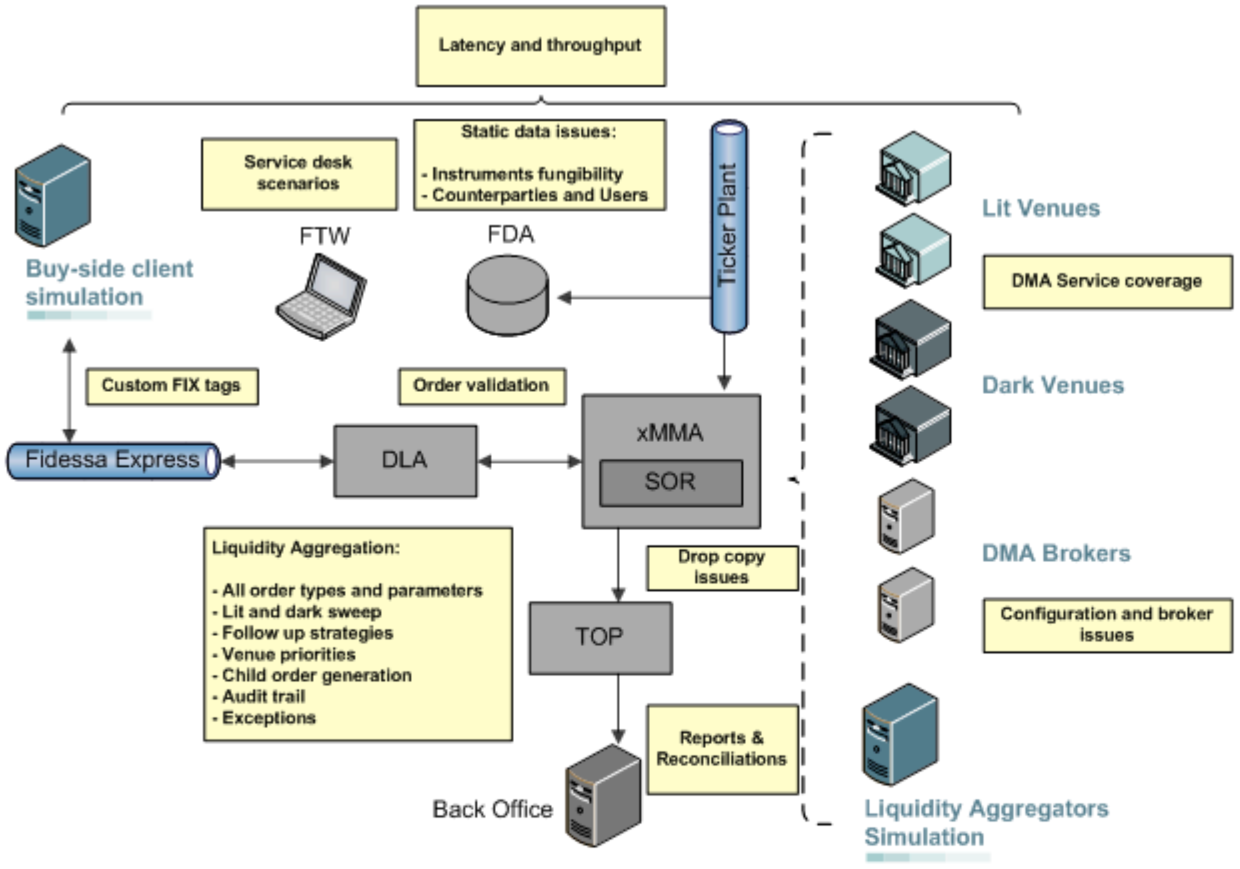
By combining these actions into sequences, we can prepare automated test scripts verifying any possible crossing/routing rules. It is important that the order book is in the expected state before each individual test case; so you must either clean up the order book between executing the test scenarios or make next test scenario rely on result of the previous one.

7.2 Advanced Execution and Smart Order Routers

With smart order routers, we deal with a decision-making system that splits parent orders into several children and sends them to different venues to achieve better execution. This system often relies on a complicated algorithm analysing the market data and exchange responses coming from several different execution venues.

Usually such systems have an API for parent order requests and responses and are connected to multiple venues via different APIs or protocols (e.g. FIX). When the testing of such systems is being automated, different levels of test cases complexity can be applied:

- Automation of incoming/outgoing gateways testing. This is similar to what we described above in the ‘Execution Venues, Dark Pools and MTFs’ sub-section;
- Test automation verifying the functionality of the system on its transactional level. In this case, automated tests are sequences of simple actions executed against the system, such as:
    - Place / amend / cancel the parent order;
    - Emulate specific prices on one or more external venues;
    - Emulate specific response(s) on a child order coming to one of the external venues;
    - Verify the response to client orders.



By combining such actions, one can verify that the SOR algorithm works as designed on a set of relatively simple test scenarios of specific orders hitting specific prices. This is an essential part of system testing, because, before testing how the algorithm behaves, one should verify that it is implemented correctly.

Behavioural testing, or Back-Testing, verifies the algorithm's behaviour under specific market conditions. In this case, we must feed the algorithm with historical market data and simulate real market responses on the child orders under these market conditions. For small orders, it may be assumed that they will be executed if they are within an 'ask' or 'bid' price limit (depending on the order). However, if the orders are substantial, one should implement a market model to simulate the market response.

7.3 Bank or Hedge Funds Proprietary Desk Trading Systems

In this case, the trading environment consists of multiple components supporting all trading activities of the trading desk. Third party or proprietary components may be integrated into a complete trading environment. These components are:

- Trading UI;
- OMS/STP system;
- Order Routing/Exchange Connectivity;
- Market Data Distribution/Ticker Plant;
- Position keeping/Risk Management;

- Back-office/Account Management;
- Algo-Trading Servers;
- DMA components.

Usually these components are integrated via some middleware, such as Tibco RV or similar. The most important task for automated testing is to cover end-to-end scenarios across the whole integrated environment. In order to perform such tests, the following actions must be available in the test automation framework:

- Enter / amend / cancel order via the trading UI;
- Read the response via the trading UI;
- Read open orders, positions, balances, P&L, etc. from the trading UI;
- Emulate market data coming into the system;
- Emulate execution venues or brokers to which the system is connected
- Retrieve trades and corresponding information from the back-office database, either directly or by requesting a report from the corresponding reporting engine;
- Sending request into the DMA gateway, if available;
- Reading responses from DMA gateway, if available.

By combining these actions, one can cover the major functionality of the trading system and perform end-to-end test scenarios related to orders and executions processed by the system. More complex scenarios might be required if the system contains an algo-trading module or a smart order router.

7.4 Algo Trading Systems

Algorithmic trading systems monitor prices on one or multiple markets and submit or manage orders automatically in order to generate profit or minimise losses on large volume trades or to provide optimal execution.

One example of an algo trading system would be Smart Order Routers mentioned earlier in the 'Advanced Execution and Smart Order Routers' sub-section. Therefore all of the test automation ideas related to algorithmic SORs are applicable to algo-trading systems.

We recommend that testing of algo trading systems be performed on the following three levels:

1. Ensure that the infrastructure around the algorithm works correctly, e.g. orders / requests issued by the algorithm are delivered to the correct venue, in the expected format;
2. Ensure that the algorithm itself works as designed and that it has no defects or produces no miscalculations;
3. Execute back-testing (or behavioural testing) against real market data and a simulated exchange, in order to ensure that algorithm itself will generate profit or serve its purpose.

Test automation works well for all of these tasks. For testing on level 1 above, you must be able to emulate requests coming from the algorithm and the execution venues, to which the infrastructure is connected. For testing on level 2 above, you must start the algorithm automatically and emulate specific market conditions. Testing on level 3 is the most complicated task, which, in addition to what is tested on levels 1 and 2, requires that the market data player and the market simulator be enhanced adequately.

7.5 Risk Management, Position Keeping, Tech Analysis, Decision Support

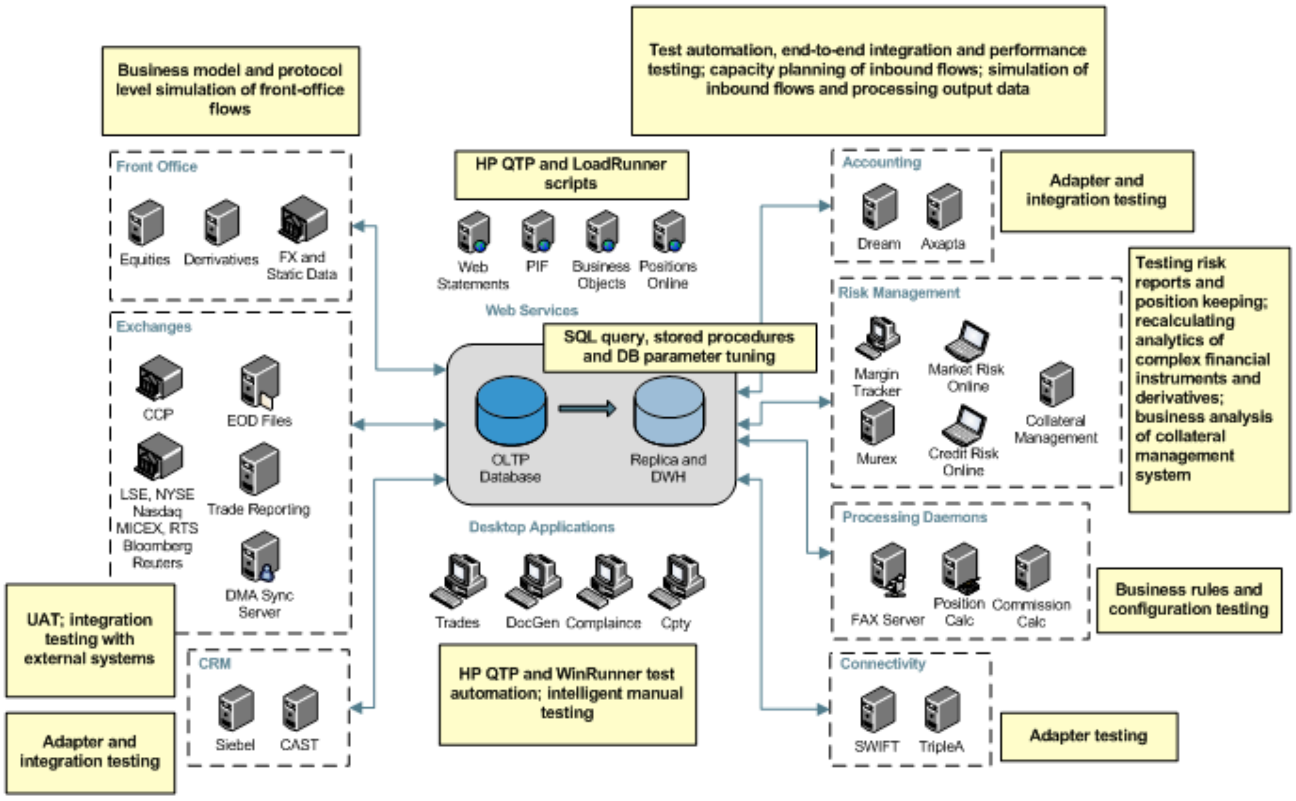
Traders, Trading Support Analysts or Risk Analysts use these systems to monitor trading activity real time and to understand the risks, P&L, 'Greeks' and other metrics associated with trading activity. Technical Analysis, Decision Support or Charting Tools also fall into this category of systems. They usually collect trading activity events or market data and calculate different metrics based on them.

Testing these systems is a complex task, because you have to verify the non-trivial math that sits within the applications. Automated testing helps with testing this math, as testers may implement the corresponding calculations within their automated tests. However, this approach has a disadvantage: in order to test such functionality comprehensively, one must re-implement the functionality of the system under test, and significant effort may be required to develop and maintain such automated tests.

7.6 Back-Office and Data Storage

Back-Office systems are usually the main source of information for a variety of different reports and metrics, which need to be delivered to external counterparties, such as clients, banks, clearing agents, custodians, etc. Back-Office systems are usually complex, as they are designed to support thousands of different reports / client queries. Their complexity grows exponentially as the trading environment becomes connected to multiple markets located in different countries. Testing of such systems may require a considerable effort and different approaches.

At the heart of most of Back-Office systems there is usually a set of databases (such as Trading Database, Book of Trades, Account Settings or CRM, Security Master, etc.) and a set of ETL processes between these databases and the reports generation systems.


Business model and protocol level simulation of front-office flows
Test automation, end-to-end integration and performance testing; capacity planning of inbound flows; simulation of inbound flows and processing output data
Front Office
Equities
Derivatives
FX and Static Data
HP QTP and LoadRunner scripts
Web Statements
PIF
Business Objects
Positions Online
Web Services
Accounting
Dream
Axapta
Adapter and integration testing
Risk Management
Margin Tracker
Market Risk Online
Collateral Management
Murex
Credit Risk Online
Testing risk reports and position keeping; recalculating analytics of complex financial instruments and derivatives; business analysis of collateral management system
Exchanges
CCP
EOD Files
LSE, NYSE Nasdaq MICEX, RTS Bloomberg Reuters
Trade Reporting
DMA Sync Server
SQL query, stored procedures and DB parameter tuning
OLTP Database
Replica and DWH
Desktop Applications
Trades
DocGen
Complaince
Cpty
Processing Daemons
FAX Server
Position Calc
Commission Calc
Business rules and configuration testing
UAT; integration testing with external systems
CRM
Siebel
CAST
Adapter and integration testing
HP QTP and WinRunner test automation; intelligent manual testing
Connectivity
SWIFT
TripleA
Adapter testing

The functionality of back-office systems can best be tested by performing data reconciliation between different databases as well as between the databases and outbound data flows / reports. In order to prepare such tests, help from SQL developers can be invaluable, as they can develop reconciliation SQL queries to compare different data sources and to compare data sources with canned reports.