



Alazar Seyoum Haile

Automation of Test Cases for Web Applications

Automation of CRM Test Cases

Helsinki Metropolia University of Applied Sciences
Bachelor of Engineering
Information Technology
Thesis
14 September, 2011

Author(s) Title	Alazar Haile Automation of CRM test cases
Number of Pages Date	38 pages + 2 appendices 14 September 2012
Degree	Bachelor of Engineering
Degree Programme	Information Technology
Specialisation option	Software Engineering
Instructor(s)	Toimi Toivonen, Product Testing Team Manager Olli Hämäläinen, Principal Lecturer
<p>The main theme of this project was to design a test automation framework for automating web related test cases. Automating test cases designed for testing a web interface provide a means of improving a software development process by shortening the testing phase in the software development life cycle.</p> <p>In this project an existing AutoTester framework and iMacros test automation tools were used. CRM Test Agent was developed to integrate AutoTester to iMacros and to enable the AutoTester, a desktop application testing tool developed in-house, extend its capability to automate also the web related test cases aimed for regression testing.</p> <p>By designing the CRM Test Agent framework, it was possible to make the automation process a viable experience. Being able to shorten the time it takes to execute a few sample test runs shows that the automation process helps improve the product testing carried out by enabling software testing engineers finish the regression testing in a short time easily. In the future, the CRM Test Agent as well as the AutoTester application could further be merged together as a single tool which would simplify the work of the test automation process.</p>	
Keywords	CRM, iMacros, AutoTester, CRM Test Agent

Contents

1 Introduction.....	1
2 Theoretical Background	4
2.1 Software Testing.....	4
2.2 Software Testing Process.....	5
2.3 Understanding the System under Test.....	11
2.3.1 GENERIS System	11
2.3.1 CSWeb Application.....	14
2.4 Test Plan.....	15
3 Types of Software Testing.....	20
3.1 Unit Testing.....	20
3.2 System Testing	21
3.3 Integration Testing	21
3.4 User Acceptance Testing (UAT)	24
3.5 Factory Acceptance Testing (FAT):	25
3.6 Regression Testing	25
4 Design of Test Automation Framework	27
4.1 Tools Used	27
4.2 CRM TestAgent.....	28
4.2.1 Use Case Diagram of CRM Test Agent.....	29
4.2.2 Class Diagram of CRM Test Agent	31
5 Results.....	35

6 Conclusion.....	38
References.....	39
Appendices.....	41
Appendix 1. Sample code.....	41
Appendix 2. Screenshots.....	46

Abbreviations and Terms

ANSI	American National Standards Institute
CRM	Customer Relationship Management
CSWeb	Customer Services Web
DB	Database
FAT	Factory Acceptance Testing
GENERIS	Energy Management Software Licensed by Process Vision Oy
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secured
IEEE	Institute of Electrical and Electronics Engineers
MBT	Model Based Testing
RDBMS	Relational Database Management System
RUP	Rational Unified Process
SAT	System Acceptance Testing
TDD	Test Driven Development

1 Introduction

Software testing is a process of verifying and validating that a software application or program meets the business and technical requirements that guided its design and development and that the application or program works as expected. The software testing process also identifies defects, flaws, or errors in the application code that must be fixed. During test planning, possible sources of defects are identified by reviewing the requirements. A defect is an issue that from the customer's perspective affects the usability or functionality of the application.

Software testing has three main purposes: *verification*, *validation*, and *defect finding*. The verification process confirms that the software meets its technical specifications. A specification is a description of a function in terms of a measurable output value given a specific input value under specific preconditions. [1,26.] The validation process confirms that the software meets the business requirements. Other requirements provide details on how the data will be summarized, formatted, and displayed. A defect is a deviance of the result from the expected value. The ultimate source of the defect may be traced to a fault introduced in the specification, design, or development (coding) phases.

As testing is an integral part of software development life cycle, it is even more crucial when the size of the software being developed becomes complex and of bigger scale. The other significant issues with the process of testing are deciding what functionalities to be tested and how much resource to be allocated to test that part or functionality. Optimal testing practices for different applications could be very different, and different applications require different amounts of testing.

The software industry where sophisticated and scalable software products are vended to the market is required to implement rigorous testing procedures and well defined testing practices. In agile software development process each part of the life-cycle such as gathering customer requirements, requirement analysis, elaboration, and the software design and implementation involves testing to verify that all the tasks meet the required output.

It is very difficult to summarize the classification of software products, but in general, software can be classified based on the area of technology used to develop the product. In this regard, software can generally be classified in to desktop applications and web applications.

Desktop applications are software applications that are installed on desktop computers or server machines and are executed providing their own user interface where the user can directly interact. The main characteristics of this software category, from a software testing perspective, is that each control such as a button, text box, or grid view provides its unique control ID with which other software can easily interact and simulate the user actions without the need of having a human being in close interaction.

Web applications fall in the category of software applications that run on web servers and provide data through the use of HTTP and HTTPS protocols. Major parts of the web applications' functionality deal with improved presentation for human usability. This is inherently hard to test automatically because the design is focused on human's accessibility as opposed to machines or providing other software interaction. [14,26]

Microsoft Dynamics CRM is a Customer Relationship Management web application developed by Microsoft Corporation to enable customers to customize and use the framework in order to reduce the time spent in customer services and data management tasks. [15,8] It provides an easy to use platform for handling customer data, business processes, and financial transactions. It also provides an interactive interface for generating different kinds of reports for businesses. One of the products that are the area of interest in this thesis is discussed in detail in section 2.3.1.

This thesis was written during my industrial work placement in Process Vision Ltd. Process Vision is an IT-company that is specialized in developing information systems and applications for energy business. The company was founded in 1993. It currently employs more than 150 IT and energy management professionals in Finland (Helsinki, Jyväskylä, Kuopio), Sweden (Stockholm) and in Netherlands (Bussum).

Process Vision Ltd focuses on developing versatile solutions for the deregulated energy market targeted for distribution companies, energy retailers, balance coordinators and system operators. These solutions consist of wide measurement data warehouses, systems for balance settlement and balance management, different standard messaging and data transmission functions and systems for contract portfolio management. [19]

During my work placement at Process Vision, the need for automating parts of the product testing process came into existence. My project was carried out to improve the testing process and to shorten the product testing cycle by automating the web-related test cases.

The aim of this thesis is to demonstrate that automating test cases for CSWeb applications plays an important role in the overall application development process by making production and delivery time short by reducing the time spent on regression testing. The thesis focuses on three sub-objectives in order to achieve the general objective.

The first sub-objective is to see if the test plan includes sufficient coverage of the features to be tested and to decide how well the test cases can be built to achieve the maximum test coverage. This helps to decide if a test case is well suited to be automated or to see if it is more advantageous to test manually than by developing an automated test framework. These settings should have clear measurement criteria which would allow gathering empirical data about both development processes and hence benchmarking them against each other.

The second sub-objective is to design and build an automated testing framework which would supposedly improve the testing process and shorten the overall software development life cycle. The framework must also enable the end to end testing of CSWeb applications and other web based applications so that it could also be used to automate other kinds of web products.

The third sub-objective is to measure the time taken by the sample automated test runs and compare it against the time taken by the manual testing process. Finally, a recommendation is made for the future improvements in the test automation design and implementation.

2 Theoretical Background

2.1 Software Testing

The main methods in software testing are:

- Check the code for consistency with design: The areas to check include modular structure, module interfaces, data structures, functions, algorithms, and I/O handling.
- Perform the testing process in an organized and systematic manner with test runs dated, annotated and saved. A plan or schedule can be used as a checklist to help the programmer organize testing efforts. If errors are found and changes made to the program, all tests involving the erroneous segment (including those which resulted in success previously) must be rerun and recorded.
- Ask a colleague for assistance: Some independent party, other than the programmer of the specific part of the code, should analyze the development product at each phase. The programmer should explain the product to the party who will then question the logic and search for errors with a checklist to guide the search. This is needed to locate errors the programmer has overlooked.
- Use available tools: The programmer should be familiar with various compilers and interpreters available on the system for the implementation language being used because they differ in their error analysis and code generation capabilities.
- Apply stress testing to the program: Testing should exercise and stress the program structure, the data structures, the internal functions and the externally visible functions or functionality. Both valid and invalid data should be included in the test set.
- Test one at a time: Pieces of code, individual modules and small collections of modules should be exercised separately before they are integrated into the total program, one by one. Errors are easier to isolate when the number of potential interactions are kept

small. Instrumentation-insertion of some code into the program solely to measure various program characteristics can be useful here. A tester should perform an array of bound checks, check loop control variables, determine whether key data values are within permissible ranges, trace program execution, and count the number of times a group of statements is executed.

- Measure testing coverage: If errors are still found every time the program is executed, testing should continue. Because errors tend to cluster, modules appearing particularly error-prone require special scrutiny. The metrics used to measure testing thoroughness include statement testing (whether each statement in the program has been executed at least once), branch testing (whether each exit from each branch has been executed at least once) and path testing (whether all logical paths, which may involve repeated execution of various segments, have been executed at least once). Statement testing is the coverage metric most frequently used as it is relatively simple to implement. The amount of testing depends upon the cost of an error. Critical programs or functions require more thorough testing than the less significant functions. [2, 2]

2.2 Software Testing Process

Software testing requires the use of a model to guide such efforts as test selection and test verification. Often, such models are implicit, only consisting of the testers' intuitive decisions and applying test inputs in an ad-hoc fashion. The mental model testers build encapsulates application behavior, allowing testers to understand the application's capabilities and more effectively test its range of possible behavior. When these mental models are written down, they become sharable and reusable testing artifacts. In this case, testers are performing what has become to be known as model-based testing. Model-based testing has recently gained attention with the popularization of models (including UML) in software design and development. There are a number of models of software in use today, a few of which make good models for testing. [12,1-3]

This section of the thesis introduces the V-Model testing and the Rational Unified Process (RUP) model which are widely used models in software engineering. Both approaches are viable to the

software development processes but RUP is the most recent and refined model which is practiced in Process Vision.

The V-Model

One of the popular models describing the overall software development and testing processes is the V-Model of software testing (cf. figure 1).

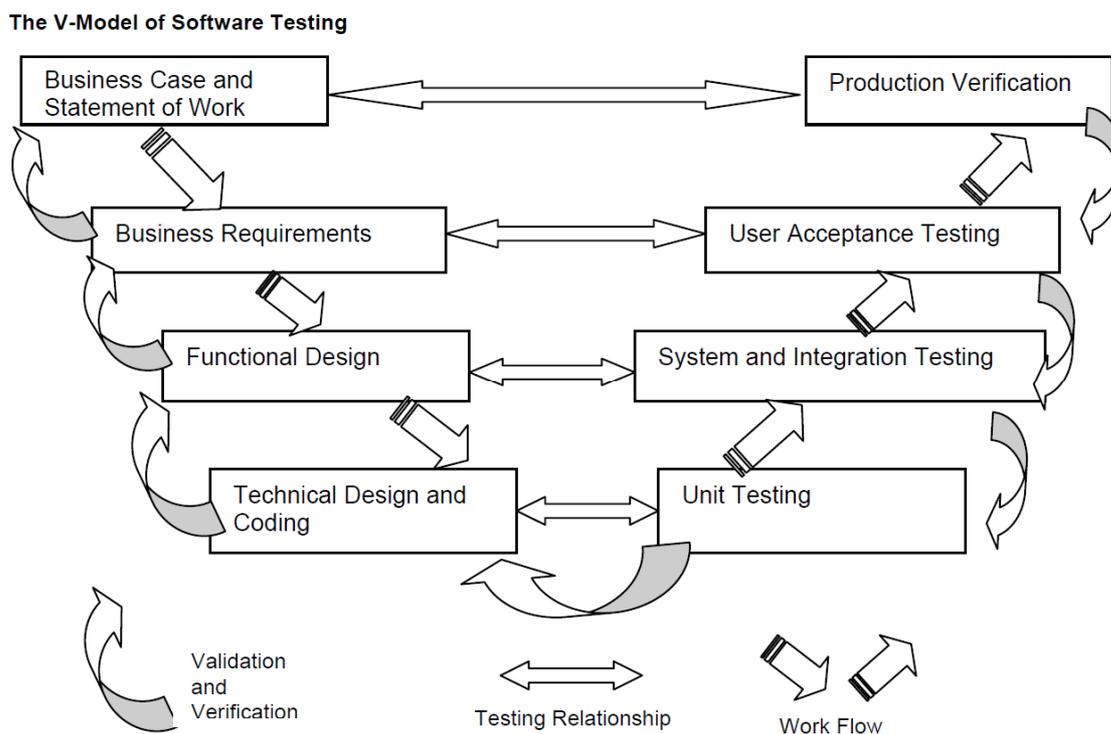


Figure 1 The V-Model of a software testing process [4,4]

Figure 1 shows the V-Model of the software testing process. The V-model was originally developed from the waterfall software process model. The four main process phases – requirements, specification, design and implementation – have a corresponding verification and validation testing phase. Implementation of modules is tested by unit testing. System design is tested by integration testing. System specifications are tested by system testing, and finally acceptance testing verifies the requirements. [4,2] The V-model gets its name from the timing

of the phases. Starting from the requirements, the system is developed one phase at a time until the lowest phase, the implementation phase, is finished. At this stage testing begins, starting from unit testing and moving up one test level at a time until the acceptance testing phase is completed. [4, 2-4]

Rational Unified Process (RUP)

Process Vision uses RUP which provides a disciplined approach to assigning tasks and responsibilities within a development organization. Its goal is to ensure the production of high-quality software that meets the needs of its end-users, within a predictable schedule and budget.

The RUP is a process that enhances team productivity by providing every team member with easy access to a knowledge base with guidelines, templates and tools and also mentors for all critical development activities. Making it possible for all team members access the same knowledge base, no matter if the work is being done on the requirements analysis, design, test, project management, or configuration management, it is ensured that all team members share a common language, process and view on how to develop software. [5,13;6,63-64]

As GENERIS Browser is large scale software with a multi-layered architecture, the RUP approach is the right process model that fits to all sectors of software development teams sharing the fundamental skills and knowledge base. It is also of the most significance to the mainline testing team, as it makes sure that the team members acquire all the basic knowledge of integration and operational skills to ensure that the required functionality is tested and to maintain the whole test coverage.

The RUP involves a quantified set of best practices to effectively deploy software with high quality standard. The RUP process framework includes expanded process content (via plug-ins) in areas or solutions such as service-oriented architecture, packaged administration (commercial off-the-shelf), portfolio management, program management, systems engineering, and many others. The six best practices in RUP are as follows:

1. Develop software iteratively: With a highly scalable software system such as GENERIS Browser, it is not possible to first define the whole requirement at a time, design the entire solution, build and then test the software at the end. An iterative approach is required that allows an increasing understanding of the problem through successive refinements and incrementally develops an effective solution over multiple iterations. The Rational Unified Process supports an iterative approach to development that addresses the highest risk items at every stage in the lifecycle, significantly reducing a project's risk profile. This iterative approach helps attack the risks through demonstrable progress of frequent, executable releases that enable continuous end user involvement and feedback. Because each iteration ends with an executable release, the development team stays focused on producing results, and frequent status checks help to ensure that the project stays on schedule. An iterative approach also makes it easier to accommodate tactical changes in requirements, features or schedule. [7,69-72; 8,14-16]

The RUP model of Process Vision is presented in figure 2.

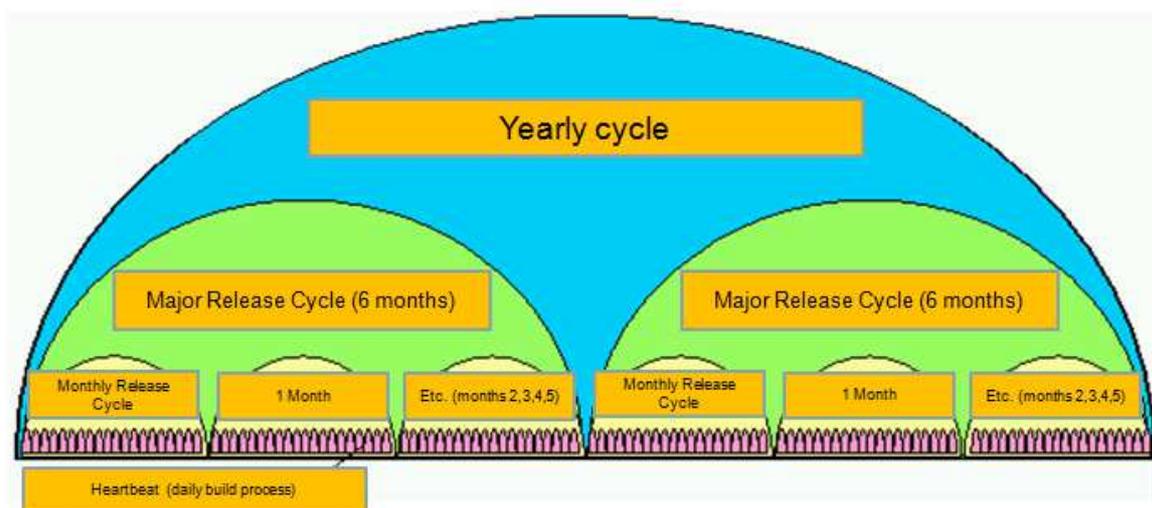


Figure 2 Generis build and release rhythm [19]

Taking Process Vision's RUP as an example, figure 2 illustrates that the product cycle consists of ten releases per year of which two releases are called major releases. The remaining monthly

releases are called branches. The mainline testing team tests the mainline releases which come out every month before they are handed over to projects.

2. Manage requirements: The Rational Unified Process describes how to elicit, organize, and document required functionality and constraints, track and document tradeoffs and decisions, and easily capture and communicate business requirements. The notions of use case and scenarios prescribed in the process have proven to be an excellent way to capture functional requirements and to ensure that these drive the design, implementation and testing of software, making it more likely that the final system fulfills the end user needs. They provide coherent and traceable threads through both the development and the delivered system. [8,82]

3. Use component based architecture: The process focuses on early development and base lining of a robust executable architecture, prior to committing resources for full-scale development. It describes how to design a resilient architecture that is flexible, accommodates change, is intuitively understandable, and promotes more effective software reuse. The Rational Unified Process supports component-based software development. Components are non-trivial modules or subsystems that fulfill a clear function. The Rational Unified Process provides a systematic approach for defining an architecture using new and existing components. These are assembled in a well-defined architecture, either as an ad hoc or as in a component infrastructure such as the Internet and COM, for which the industry of reusable components are emerging. [10,140.]

4. Visually model software: The process shows you how to visually model software to capture the structure and behavior of architectures and components. This allows hiding the details and writing code using graphical building blocks. Visual abstractions help to communicate different aspects of the software, to see how the elements of the system fit together, to make sure that the building blocks are consistent with the code, to maintain consistency between a design and its implementation, and to promote unambiguous communication. The industry standard Unified Modeling Language (UML), created by Rational Software, is the foundation for successful visual modeling. [11,38]

5. Control changes of software: The ability to manage change is making certain that each change is acceptable, and being able to track changes is essential in an environment in which change is inevitable. The process describes how to control, track and monitor changes to enable successful iterative development. It also guides how to establish secure workspaces for each developer by providing isolation from changes made in other workspaces and by controlling changes of all software artifacts (such as models, code, and documents). It brings a team together to work as a single unit by describing how to automate integration and build management. [10,71-72]

6. Verify quality of software: Poor application performance and poor reliability are common factors which dramatically inhibit the acceptability of today's software applications. Hence, quality should be reviewed with respect to the requirements based on reliability, functionality, application performance and system performance. The RUP assists you in the planning, design, implementation and evaluation of these test types. Quality assessment is built into the process, in all activities, involving all participants, using objective measurements and criteria, instead of treating it as an afterthought or a separate activity. [10,71-72]

The focus of quality assurance in the mainline testing team is to maintain the test case development and to make the actual testing for every test cycle and also during the product delivery (Factory Acceptance Testing /SAT). Even though regression testing is a time consuming process, it plays an important role in improving the product quality and performance. Test case automation helps to bridge the gap of product improvement and delay in product delivery by making the testing process faster. [10,68]

2.3 Understanding the System under Test

2.3.1 GENERIS System

Generis System is a large energy information management platform that provides a solution to a broad set of parties involved in the energy marketing sector ranging from medium-sized local utility companies to system operators and energy exchanges. The GENERIS platform is used for meter data management, balance settlement and management, trade and risk management, meter asset management, contract and portfolio management, and also for billing (cf. figure 3).



Figure 3 GENERIS business applications [15,6]

GENERIS Meter Data Management (MDM) is a solution for vendor-independent management of measurement data for different commodities. Data from multiple sources and in different formats are collected, validated and converted to a normalized format. The data are also made available to other system applications and reporting. Generis workflow control integrates Meter Data Management into business processes for balance management and settlement, with support for customer relocation and meter changes. [19]

GENERIS Energy Data Management (EDM) provides business logic for all energy market roles and for all the data required for balance settlement and physical balance management including business-to-business (B2B) market communication. Calculation modules handle data aggregation and profile calculations for allocation and reconciliation in electricity and gas markets. Forecasting tools in GENERIS EDM provide additional information for business planning.

Trade and risk management is handled by GENERIS Trade Management System (TMS). It integrates trading functionality with portfolio management and scheduling. TMS aggregates trades into price and energy time series providing a business overview for physical and financial balance management and also for business planning.

GENERIS Meter Asset Management (MAM) consists of tools and applications for management of metering-related devices. It manages the entire metering device lifecycle from purchase, storage and installation up to the phase-out. GENERIS MAM includes functionality for stock, workflow and work order management.

Contract and portfolio management are the responsibilities of GENERIS Contract Management System (CMS). CMS integrates commercial energy business data with physical metering data. It is used for managing and analyzing sales, purchase contracts and their portfolios. This can be done by using forecast, actual interval measurements or consumption profiles. With GENERIS CMS, users can make sales forecasts and analyze the profitability of specific customer groups. Pricing tools help to create optimal offers or campaigns.

GENERIS Billing Data Management (BDM) provides the billing engine for the generation of billing data for commodities and customers. It also handles business-to-business billing based on up-to-date market prices. Integration of GENERIS BDM with the CMS and TMS applications makes GENERIS a complete end-to-end system solution from measurement data to invoicing.

The GENERIS platform can be used with the GENERIS Browser. The browser is an application running on the Microsoft Windows operating system. It looks and feels like other Windows

applications. The basic layout of the browser consists of a couple of different components: menu bar, toolbar, Business Process bar, tree view, list view and status bar. The main screen of the GENERIS browser can be seen in figure 4.

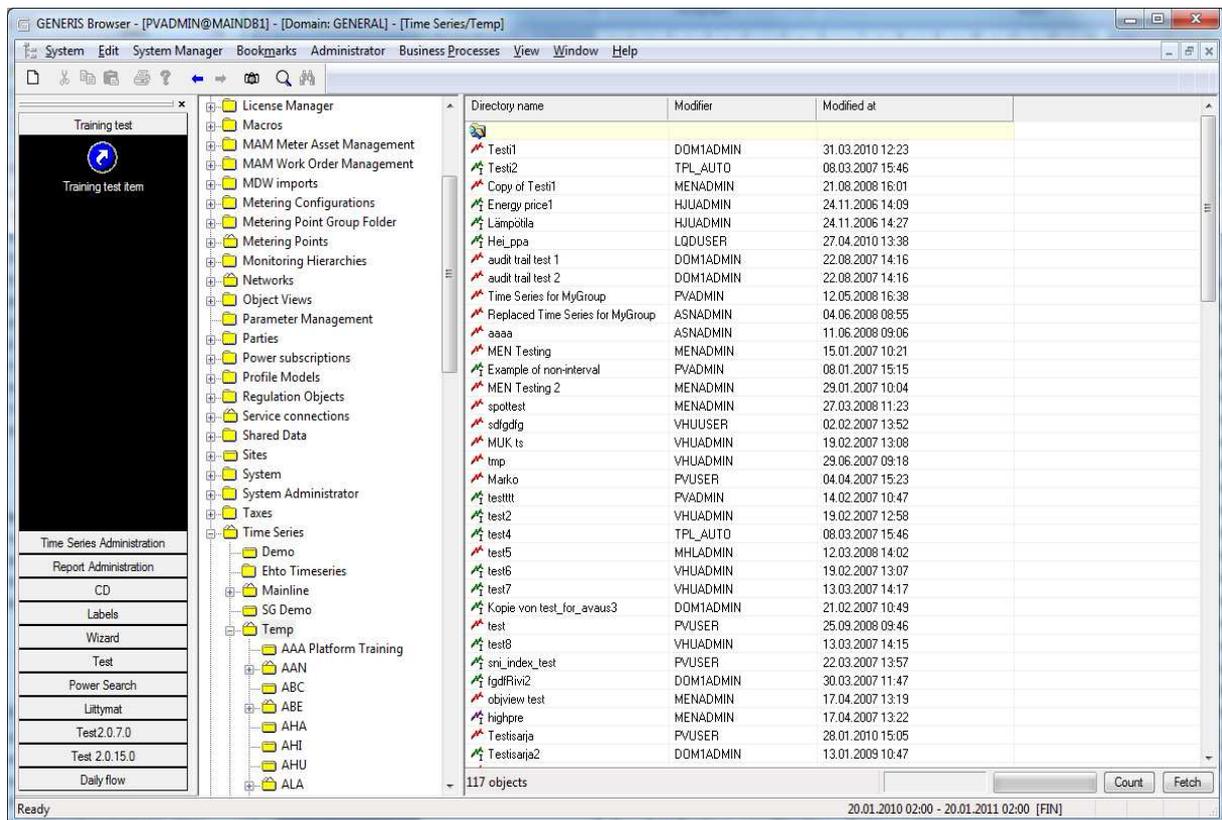


Figure 4. GENERIS browser

The tree view in the GENERIS browser presents the main hierarchy of the system, which is split into different folders. The different GENERIS business applications add different folders to the hierarchy, so the layout of the tree depends on the applications that the customer has purchased. Every folder usually contains subfolders and objects. The objects in one folder are usually of the same type. For example, the Time Series folder contains only Time Series objects.

2.3.1 CSWeb Application

Since the GENERIS Browser incorporates a multitude of applications, it is sometimes difficult for customers to easily create a contract when receiving calls from customers. For the GENERIS Browser to function properly, all related objects must be created and linked to each other. This requires in depth technical knowledge about the Generis Browser, and creating the objects manually requires a significant amount of time.

In order to make the process of customer handling easier, a Microsoft web application which runs over the IIS server is created. This platform is called Energy Vertical Web (EvWeb). EvWeb communicates with the GENERIS system using the Windows Communication Foundation (WCF) services. The WCF services provide an easy way to execute binaries over system boundaries.

Figure 5 shows how the Ev Web application interacts with GENERIS by accessing the Generis Browser binaries through the IIS web service.

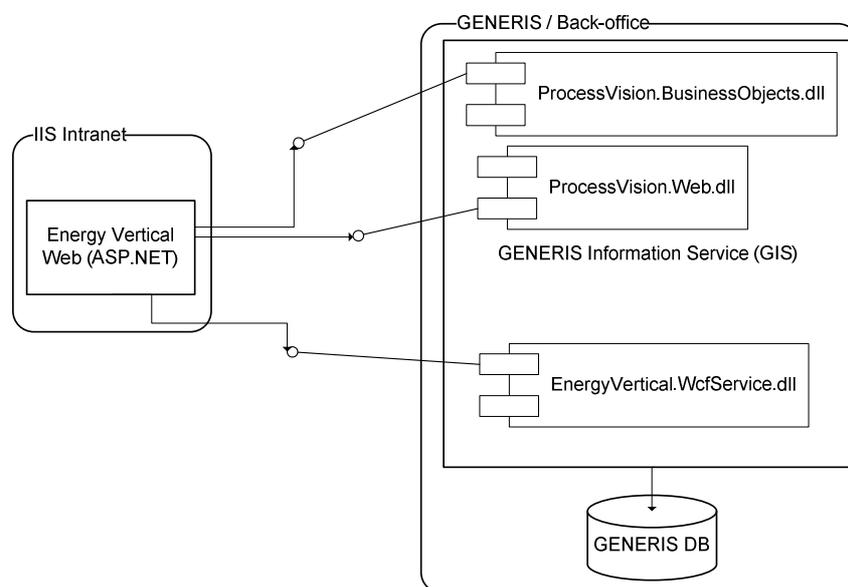


Figure 5 Energy vertical architecture [19]

One of the important applications of the Energy Vertical platform is CSWeb. It provides an easy and interactive web interface for creating, editing, deleting and searching, customers, contracts, metering points and performing other related tasks. [19]

The requirement common to most styles of testing is a well-developed understanding of what the software accomplishes, and the Model Based Testing (MBT) is not different. Forming a mental representation of the system's functionality is a prerequisite to building models. This is a nontrivial task as most systems today typically have convoluted interfaces and complex functionality. Moreover, software is deployed within gigantic operating systems among a clutter of other applications, dynamically linked libraries, and file systems all interacting with and/or affecting it in some manner. [12, 3]

To develop an understanding of an application, testers need to learn about both the software and the environment. By applying some exploratory techniques [12, 3-4] and reviewing available documents, model-based testers can gather enough information to build adequate models.

2.4 Test plan

The test plan is a mandatory document laid out at the beginning of the testing process. For simple and straight-forward projects the plan does not have to be elaborate but it must address certain items. As identified by the "ANSI and IEEE Standard 829/1983 for Software Test Documentation", a software test plan should cover the key components as shown in table 1.

The release of a new application or an upgrade inherently carries a certain amount of risk that it will fail to do what it is supposed to do. A good test plan goes a long way towards reducing this risk. By identifying areas that are riskier than others, software testers can concentrate the testing efforts only on the areas where it is highly vulnerable to errors or newly added features. These areas include not only the must-have features but also areas in which the technical staff is less experienced, perhaps such as the real-time loading of a web form's contents into a database using complex business logic. Because riskier areas require high level of certainty that they work properly, failing to identify these risky areas correctly leads to a misallocated testing effort.

Table 1 Items covered by a test plan [3,4]

Component	Description	Purpose
Responsibilities	Specific people who are assigned the tasks and their assignments	To assign responsibilities and to keep everyone on track and focused
Assumptions	Code and systems status and availability	To avoid misunderstandings about schedules
Test	Testing scope, schedule, duration, and prioritization	To outline the entire process and to map specific tests
Communication	Communication plan: who what, when, how	To identify the information, to select the target people to give the information and to plan the information channel
Risk Analysis	Critical items that will be tested	To provide focus by identifying areas that are critical for success
Defect Reporting	How defects will be logged and documented	To tell how to document a defect so that it can be reproduced, fixed, and retested
Environment	The technical environment, data, work area, and interfaces used in testing	To reduce or eliminate misunderstandings and sources of potential delay

As indicated in table 1, the test plan is a mandatory document. Testing should not be done without a corresponding test plan. For simple, straightforward projects the plan does not have to be elaborate but it must address certain items. As identified by the ANSI, the components specified in table 1 should be covered in a software test plan.

It is possible to identify risk areas by asking people who use the software for their opinion and by gathering information from developers, sales and marketing staff, technical writers, customer support people, and any users who are available to comment on the product. Historical data, bugs and testing reports on similar products or previous releases will identify areas to explore. Furthermore, as bug reports from customers are important, it is also crucial to look at bugs reported by the developers themselves. These will provide insight into the technical areas having trouble. [2,21]

When problems are inevitably found, it is important that both the IT side and the business users have previously agreed on how to respond. This includes having a method for rating the importance of defects so that repair work effort can be focused on the most important problems. Prioritization is very common to use a set of rating categories that represent decreasing relative severity in terms of business/commercial impact. Reported bugs can be rated with terms such as Blocker, Critical, Major, Minor and Cosmetic in a system based on the level of criticality. The priorities can also be rated using terms as Extra Hot, Hot and Normal. The combination of prioritization and criticality defines which issues need to be given precedence over the other.

The definition of each term for criticality is given below:

1. Blocker: It is impossible to continue testing because of the severity of the defect.
2. Critical: Testing can continue but the application cannot be released into production until this defect is fixed.
3. Major: Testing can continue but this defect will result in a severe departure from the business requirements, if released for production.
4. Medium: Testing can continue and the defect will cause only minimal departure from the business requirements when in production.
5. Minor: Testing can continue and the defect will not affect the release in to the production. The defect should be corrected but few or no changes to business requirements are envisaged.

6. Cosmetic: Minor cosmetic issues like colors, fonts, and pitch do not affect testing or production release. If, however, these features are important business requirements, they will receive a higher severity level.

Planning a test is generally important because it results in avoiding bias towards functional testing during which each feature is tested alone in a unit test, and the systems integration test is just a series of unit tests strung together. Such an unstructured testing approach causes the misconception that each feature unit tested does not reflect the integrity of the whole application when operating all together. [2,21]

On the other hand, testing every combination of keystrokes or commands (this is where unstructured testing comes in) is difficult at best and may also be impossible for cases such as huge applications with which specific target areas only need to be tested. It is important to remember that features do not function in isolation from each other.

Users have a task orientation which means they have the need to find defects that are important to them. In finding the defects that are important to them, they need to have a well formed test plan. An incomplete test plan can result in a failure to check how the application works on different hardware and operating systems or when combined with different third-party software. The test plan should include some fundamental information on what kind of platform the customers use and what operating system and machine performance they have.

There may be more than a few possible system combinations that need to be tested and that can require utilization of a possibly expensive computer laboratory and spending much time setting up tests. Configuration testing is not a cheap process. Performing this test is advantageous in avoiding the product from behaving differently when it runs on the customers' computers.

A crucial test is to see how the application behaves when it is under a normal load or under stress. Stress can be derived from the business requirements, but for web enabled applications, stress can be considered as a spike in the number of transactions. Stress testing is applied by performing very large transactions at the same time or by carrying out a large number of

almost identical simultaneous transactions. The objective of stress testing is to see what happens when the application is pushed to take care of more transactions than the basic requirements.

Stress testing is often put off until the end of testing, after everything else that is going to be fixed has been completed. Unfortunately this leaves little time for repairs when the failure threshold of the stress test is closer to the requirement.

There are two common omissions in many test plans - the installation procedures and the documentation are ignored. An installation instruction that is missing a key step creates an overload to the testing process and puts customers in dissatisfaction while causing a considerable amount of work time to be wasted. Moreover, although the documents may have been written by a professional technical writer, they might not have been tested by a real user. Bad installation instructions immediately cause lowered expectations regarding the product, and poorly organized or written documentation certainly does not help a confused or irritated customer to feel better. Hence, testing installation instruction procedures and documentation is a good way to avoid making a bad first impression or making a bad situation worse. [3,4]

3 Types of Software Testing

In agile software development much of the functional testing is done after the project implementation is completed. There is a special testing team dedicated for making quality assurance (QA) tasks and a comprehensive testing plan. [13,72] If the test plan is executed and all the errors are found and corrected, then the project stakeholders should be convinced that the software is ready for deployment. These kinds of testing where rigorous investigation of bugs to check if the integral part of the software behaves as expected and achieves the required goal falls under the category of Integration Testing, Functional Testing, Factory Acceptance Testing and System Acceptance Testing. [13,72-81]

3.1 Unit Testing

A series of stand-alone tests are conducted during unit testing. Each test examines an individual component that is new or has been modified. A unit test is also called a module test because it tests the individual units of a code that comprises the application.

Each test validates a single module that, based on the technical design documents, was built to perform a certain task with the expectation that it will behave in a specific way or produce specific results. Unit tests focus on functionality and reliability, and the entry and exit criteria can be the same for each module or specific to a particular module. Unit testing is done in a test environment prior to system integration. If a defect is discovered during a unit test, the severity of the defect will dictate whether or not it will be fixed before the module is approved. Most of the bugs which are common errors in programming logic are revealed by this kind of testing which is defined by test driven development.

Test Driven Development (TDD) is a practice which prescribes that a test code should be written before the application code. TDD is very strict about how the development should be done. It says that all the coding should be done in the following faces. The first step is to write about the test method and to execute it. The test should fail or not compiled as there is no application code yet. [16,126]

3.2 System Testing

System testing tests all components and modules that are new, changed, affected by a change, or needed to form the complete application. The system test may require involvement of other systems but this should be minimized as much as possible to reduce the risk of externally-induced problems. Testing the interaction with other parts of the complete system comes in integration testing. The emphasis in system testing is validating and verifying the functional design specification and seeing how all the modules work together.

The first system test is often a smoke test. This is an informal quick test through which the application's major functions are tested without bothering with details. The term comes from the hardware testing practice of turning on a new piece of equipment for the first time and considering it a success if it does not start smoking or burst into flames.

System testing requires many test runs because it entails a feature by feature validation of the software behavior using a wide range of both normal and erroneous test inputs and data. The test plan is critical here because it contains descriptions of the test cases, the sequence in which the tests must be executed, and the documentation needed to be collected in each run.

When an error or defect is discovered, previously executed system tests must be rerun after the repair is made to make sure that the modifications have not caused other problems. This will be covered in more detail in the section on regression testing.

3.3 Integration Testing

Integration testing examines all the components and modules that are new, changed, affected by a change, or needed to form a complete system. Where system testing tries to minimize outside factors, integration testing requires involvement of other systems and interfaces with

other applications, including those owned by an outside vendor, external partners, or the customer. For example, integration testing for the CSWeb interface that handles customer input for addition to a database must include the Generis application even if the database is hosted by a vendor. Furthermore, the complete system must be tested end-to-end. In this example, integration testing does not stop with the database load; the test reads must verify that it was correctly loaded.

Integration testing also differs from system testing in that when a defect is discovered, not all previously executed tests have to be rerun after the repair is made. Only those tests with a connection to the defect must be rerun, but retesting must start at the point of repair if it is before the point of failure. For example, the retest of a failed FTP process may use an existing data file instead of recreating it if up to that point everything else was well. [4]

Integration testing has a number of sub-types of tests that may or may not be used, depending on the application being tested or expected usage patterns. [2,32]

- **Compatibility testing** – Compatibility tests insure that the application works with differently configured systems based on what the users have or may have. When testing a web interface, compatibility with different browsers and connection speeds are tested.
- **Performance testing** – Performance tests are used to evaluate and understand the application's scalability when, for example, more users are added or the volume of data increases. This is particularly important for identifying bottlenecks in high usage applications. The basic approach is to collect timings of the critical business processes while the test system is under a very low load (a 'quiet box' condition). Subsequently, the same timings are collected with progressively higher loads until the maximum required load is reached. For a data retrieval application, reviewing the performance pattern may show that a change needs to be made in a stored SQL procedure or that an index should be added to the database design.

- **Stress testing** – Stress testing is performance testing conducted with simulated loads which are bigger than normal operational load. Stressing runs the system or application beyond the limits of its specified requirements to determine the load under which it fails and how it fails. A gradual performance slow-down leading to a non-catastrophic system halt is the desired result, but if the system will suddenly crash and burn, it is important to know the point where that will happen. Catastrophic failure in production means beepers going off, people coming in after hours, system restarts, frayed tempers, and possible financial losses. This test is arguably the most important test for mission-critical systems.
- **Load testing** – Load tests are different from stress tests. They test the capability of the application to function properly under expected normal production conditions and measure the response times for critical transactions or processes to determine if they are within the limits specified in the business requirements and design documents or that they meet service level agreements. For database applications, load testing must be executed on a current production-size database. If some database tables are forecast to grow much larger in the foreseeable future, then serious consideration should be given to testing against a database of the projected size.

Performance, stress, and load testing are all major undertakings and will require substantial input from the business sponsors and IT staff in setting up a test environment and designing test cases that can be accurately executed. Because of this, these tests are sometimes delayed and made part of the User Acceptance Testing phase. Especially load tests must be documented in detail so that the tests are repeatable in case they need to be executed several times to ensure that new releases or changes in database size do not push response times beyond prescribed requirements and service level agreements.

3.4 User Acceptance Testing (UAT)

User acceptance testing (UAT) is also called beta testing, application testing, and end-user testing. It refers to testing which moves from the hands of the IT department to those of the business users. Software vendors often make extensive use of beta testing, some more formally than others, because they can get users to do it for free. [2]

By the time UAT is ready to start, the IT staff has resolved in one way or another all the defects they identified. Regardless of their best efforts, though, they probably do not find all the flaws in the application. A general rule of thumb is that no matter how bulletproof an application seems when it goes into UAT, a user somewhere can still find a sequence of commands that will produce an error.

To be of real use, UAT cannot be performed by random users playing with the application. A mix of business users with varying degrees of experience and subject matter expertise need to actively participate in a controlled environment. Representatives from the group work with testing coordinators to design and conduct tests that reflect activities and conditions seen in normal business usage. Business users also participate in evaluating the results. This insures that the application is tested in real-world situations and that the tests cover the full range of business usage. The goal of UAT is to simulate realistic business activity and processes in the test environment.

A phase of UAT called "unstructured testing" will be conducted whether or not it is in the test plan. Also known as guerilla testing, this means that business users type anything to find the weakest parts of the application. In effect, they try to break it. Although the test is a free-form test, it is important that users who participate understand that they have to be able to reproduce the steps that led to any errors they find. Otherwise, it is of no use.

A common occurrence in UAT is that once the business users start working with the application they find that it does not do exactly what they want it to do or that it does something that, although correct, is not quite optimal. Investigation shows that the root cause is in the business requirements, so the users ask for a change.

3.5 Factory Acceptance Testing (FAT)

Factory acceptance testing is sometimes called production verification testing. This is a final opportunity to determine if the software is ready for release. Its purpose is to simulate the production cutover as closely as possible and for a period of time to simulate real business activity. As a sort of full dress rehearsal, it should identify anomalies or unexpected changes to existing processes introduced by the new application. For mission critical applications the importance of this testing cannot be overstated.

The application should be completely removed from the test environment and then completely reinstalled exactly as it will be in the production implementation. Then mock production runs will verify that the existing business process flows, interfaces, and batch processes continue to run correctly. Unlike parallel testing in which the old and new systems are run side-by-side, mock processing may not provide accurate data handling results due to limitations of the testing database or the source data.

3.6 Regression Testing

Regression testing is also known as validation testing and it provides a consistent, repeatable validation of each change to an application under development or being modified. Each time a defect is fixed, the potential exists to inadvertently introduce new errors, problems, and defects. An element of uncertainty is introduced about the ability of the application to repeat everything that went right up to the point of failure.

Regression testing refers to selective retesting of an application or system that has been modified to insure that no previously working components, functions, or features fail as a result of the repairs. Regression testing is conducted in parallel with other tests and can be viewed as a quality control tool to ensure that the newly modified code still complies with its specified requirements and that the unmodified code has not been affected by the change. It is important to understand that regression testing does not test that a specific defect has been fixed. Regression testing tests that the rest of the application up to the point or repair was not adversely affected by the fix.

To summarize, apart from the final testing phase, developers usually make sure that the code they write is error free and gives the expected results by doing unit tests. Most of the application development frameworks provide a platform for creating automated test units for unit testing. This report focuses on devising a mechanism to shorten the time invested in regression testing where a number of testers, usually more than three, perform repeated testing of the same test case per each test cycle. Reducing the time spent on the testing process is attained by automating the test cases. Hence, the test framework can run all the test cases unattended.

4 Design of Test Automation Framework

4.1 Tools Used

The test automation process involves several tools that interact with a C# application called CRMTTest Agent. iMacros is a third party software that provides an easy platform for recording and replaying web use cases during the test case automation process. The software programmatically interacts with the target websites and enters pre-defined static data. It can also fill out forms and automate the download and upload of text, images, files and web pages. It is possible to import or export data to and from web applications using CSV and XML files, databases, and other sources. iMacros is a vital tool for automating regression testing.

The CRMTTest Agent application acts as a broker between an existing in-house test automation framework called AutoTester. AutoTester is developed by the Process Vision software development team to automate the testing of the Generis browser. This software lacks the capability of automating web related test cases.

The AutoTester has a platform for interfacing with other executables. It provides a mechanism of executing other binaries from the script written by AutoTester. Hence, the CRMTTest Agent is invoked easily from the existing AutoTester framework by passing the basic test case and test run parameters. In doing so it provides a uniform automation platform for all the test cases in regression testing.

The interaction of CRMTTest Agent with other software components is presented in figure 7. As shown in the diagram, the CRMTTest Agent is invoked from the AutoTester framework. The CRMTTest Agent runs the macro in the iMacros browser and monitors the execution until the iMacro completes its execution and returns its status to the AutoTester framework. After the status is returned to CRMTTest Agent, the pass and fail options are decided and logged to a log file in xml format.

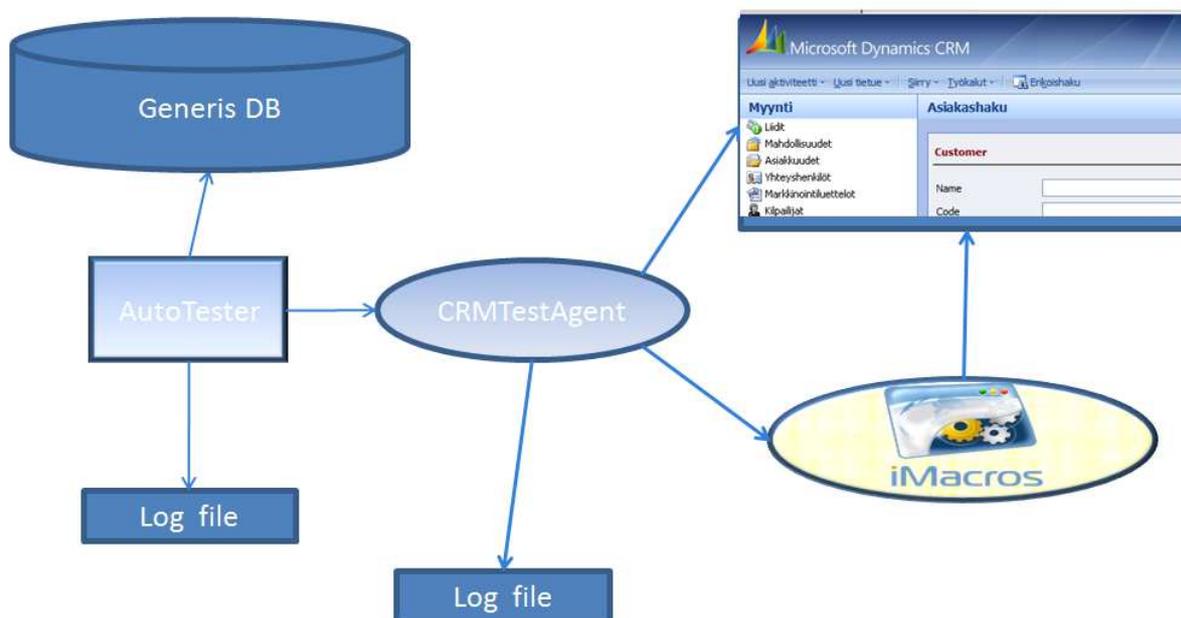


Figure 7 Interaction of CRM Test Agent, iMacros and CSweb

As shown in figure 7 the test agent is the core controlling interface where the software tester directly interacts with the appropriate macro. It starts the iMacros tool which runs the recorded user actions on the CSWeb. In addition to controlling the macro, the test agent also monitors the status of the CSWeb to see if any error has occurred and records each status to a log file. Finally, it compares the expected test values against the results and makes the pass and fail decisions.

4.2 CRM TestAgent

The CRM Test Agent was developed in Microsoft c#. It is a simple interface that bridges the existing AutoTester application to the iMacros. It accepts the basic test case information from AutoTester, in other words the calling application, and executes iMacros. The executable test agent then monitors whether the macro is executed successfully or not. If there is any error, it collects the error code and matches the error code with the corresponding error description. Finally, the CRM Test Agent gives an XML log as an output providing all the necessary test case information and test run results.

4.2.1 Use Case Diagram of CRM Test Agent

The AutoTester and the iMacros interact with CRM Test Agent to execute each test run. By using the help of the iMacros scripting engine, it is possible to call the execute method from the CRM Test agent and also to check the status of the re-executed macro during the testing process. After the execution of the test run, the script in CRM Test Agent collects the results and logs the information on the test case and test run in to an XML file.

The use case diagram in figure 8 shows the basic interaction of AutoTester to the CRM TestAgent. It also indicates the direct link between the CRM TestAgent and the iMacros.

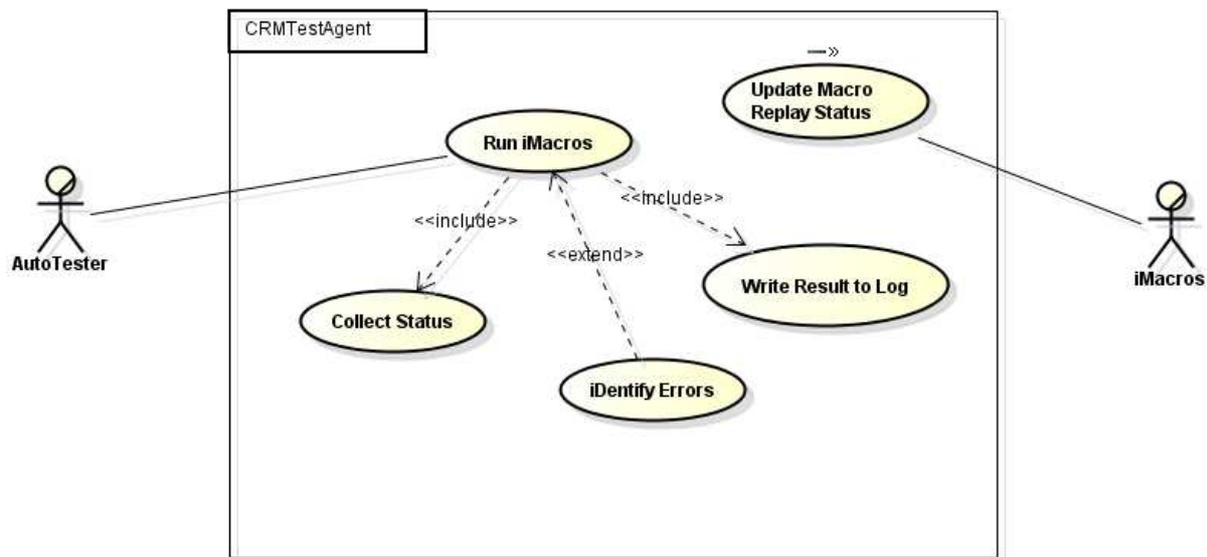


Figure 8 Use case diagram of CRM TestAgent

As shown in figure 8 illustrating the use case, the Auto Tester runs the iMacros.App engine via the CRM Test Agent running a macro interface, and then the iMacros engine writes the results of the test to an XML log file.

The list below presents the description of each use case in figure 8. The first use case describes how the AutoTester interacts with iMacros. The second use case description explains how the error code is propagated from iMacros to the CRM Test Agent. The third use case description presents the process of invoking iMacros from CRM Test Agent and monitoring the status of the replayed macro.

Use Case: **Run iMacros**

Description: AutoTester runs the iMacros.App engine

Assumption: The AutoTester script is written correctly and runs successfully

Actors: AutoTester.exe system interacts with CRM Test Agent

Steps:

1. AutoTester runs the iMacros.App to execute the macro
2. Replaying macro completes
3. TestAgent collects the status of the macro
4. TestAgent logs the test result

Variations: The system detects the result and determines the pass or fail test run

Use Case: **Update Macro Replay status**

Description: The iMacros.App engine returns the status of the replay

Assumptions: iMacros.App engine is invoked successfully

Actors: iMacros.App engine

Steps:

1. iMacros.App starts replay
2. If macro completes with error
 - Status is returned to CRM Test Agent with error code and description
 - else
 - Status is returned with the string 'Passed' and error code '0'

Variations: With both pass or fail statuses the iMacros.App returns the correct status and error code to CRM Test Agent

Use Case: **Log Result**

Description: iMacros.App logs the result of the macro

Assumption: The particular macro exists and starts replay

Actors: iMacros.App engine invoked by the CRM Test Agent performs the macro run and gives the status feedback

Steps:

1. iMacros.App is invoked by the CRM Test Agent
2. iMacros.App starts running the macro
3. iMacros.App returns the status of the macro replay to CRM Test Agent

Variations: iMacros.App handles various reasons for a replay failure including interruptions of macro by user

4.2.2 Class Diagram of CRM Test Agent

The class diagram of CRM Test Agent describes the different classes involved in the system which are used to integrate the AutoTester executable to the iMacros script running engine. It depicts the operations and behaviors of the classes. The test case class maintains the basic information of the test case to which the test automation script is written for. The list of variables in the class consists of value holders which are later to be passed to the CRM Test Agent class as an object. Once the iMacros engine replays the existing macro for the particular test case, the results are collected and the test status is used to instantiate the test result object. An object from this class will then contain the error code and log path to the results of the replay of the iMacros.

The major classes that constitute the CRM Test Agent are shown in figure 9. In the class diagram the basic properties and attributes of the classes are also shown. The diagram is only a partial listing of classes and their association. More information about the class definitions and usages could be found from the appendix section of this thesis.

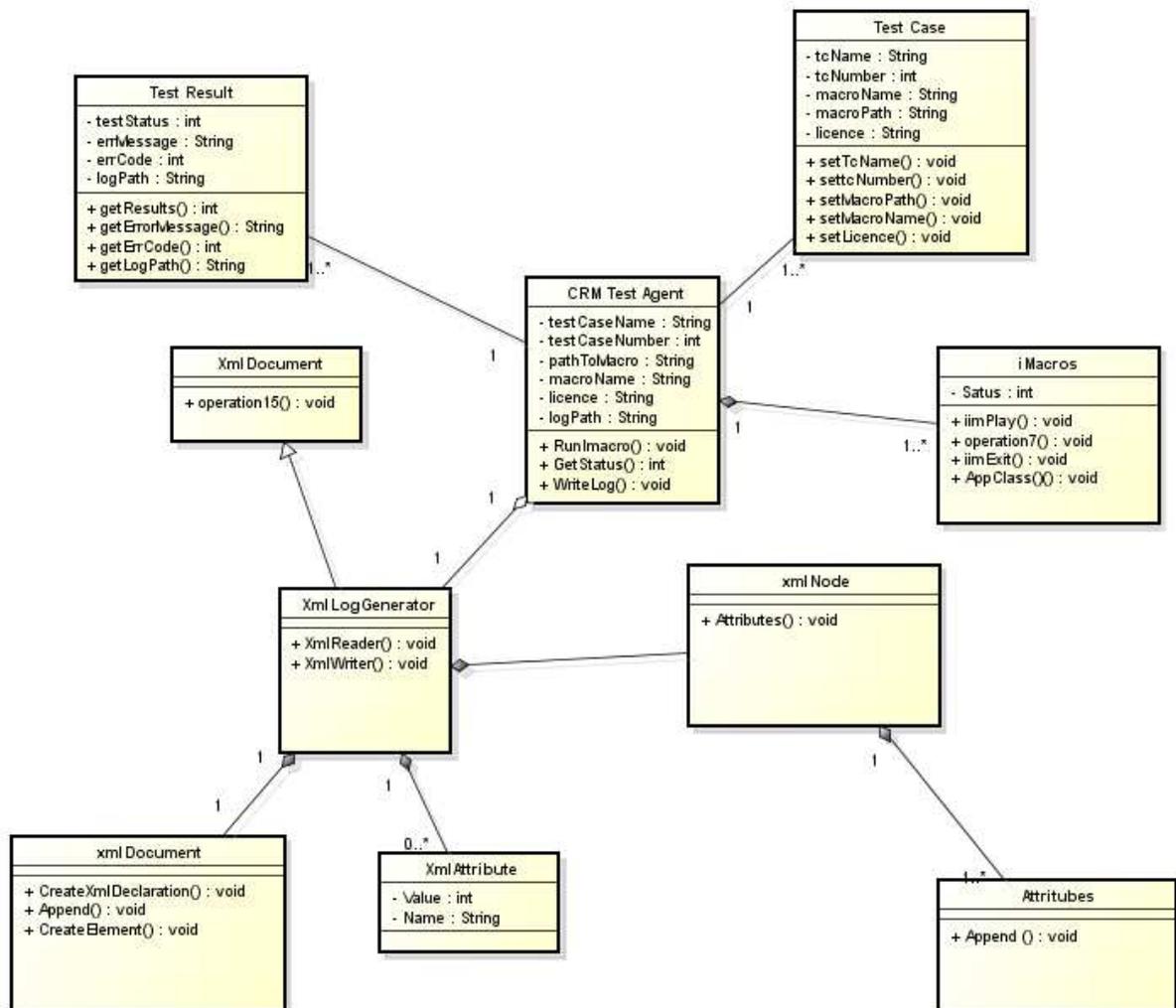


Figure 9 Class diagram of CRM Test Agent

The XmlLogGenerator class writes the test case data and the result of the iMacros execution to an XML log file. All the other classes such as XmlDocument, XmlNode, and XmlAttribute are classes that are provided by the Microsoft .Net framework which provides built-in classes to easily create an XML document and to read, write and edit xml nodes to the xml document.

The iMacros.App scripting engine provides an interface to start the macro and to return the status of the replayed macro. The result of iMacros.App is collected via its attribute called status. This status is casted to a string type in the CRM Test Agent, and it is checked if it is a pass or

fail. Then using the result of the status, the CRM Test Agent identifies the error and logs the result to an xml file format.

To explain the process of the overall CRM Test Agent interactions further, the sequence diagram in figure 10 presents the overall method invocation and message sending amongst the different modules of the framework.

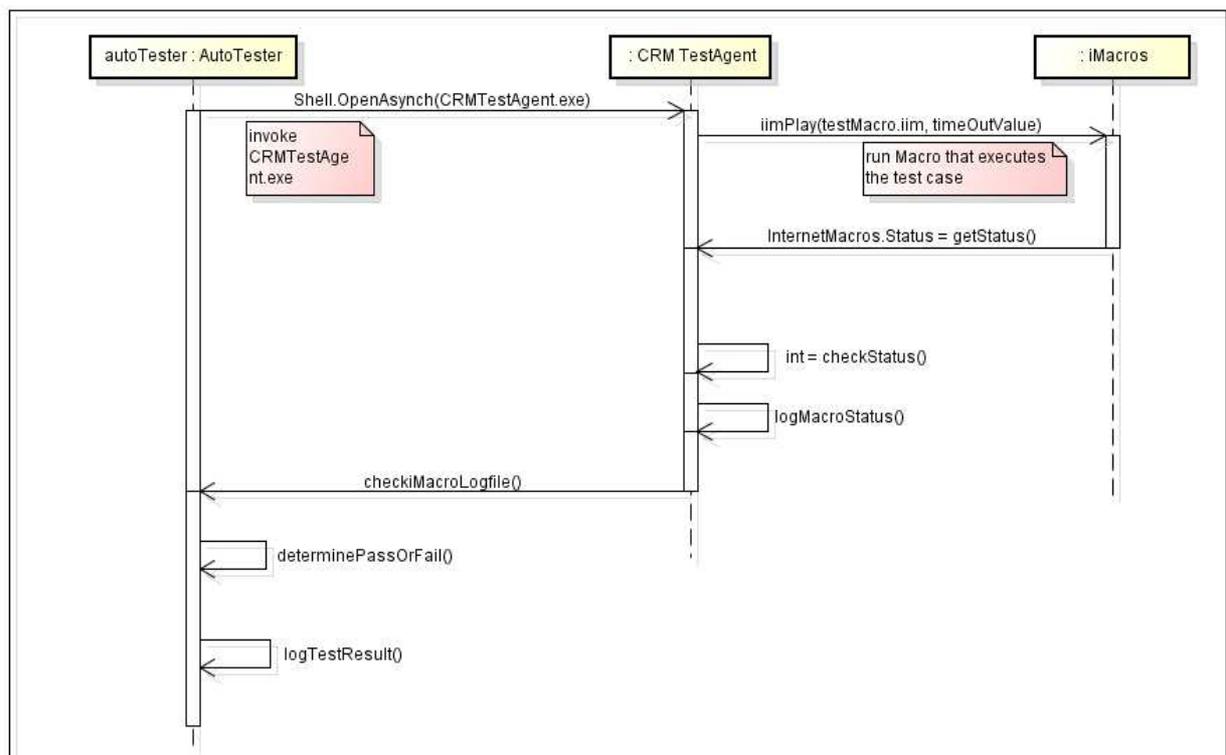


Figure 10 Sequence diagram of CSWeb test automation framework

In the sequence diagram, figure 10, it is shown that `AutoTester` invokes the `CRMTestAgent` executable by running the `mode.Shell.OpenAsynch()` method asynchronously. It is the only function implemented in the `AutoTester` framework to run executables asynchronously. Once the `CRM Test Agent` is started, it runs the macro using its `iimPlay()` function that executes the macro saved for the target test case. After the `iimPlay()` function call has been completed either due to successful termination or aborting due to error, the `CRMTestAgent` collects the status and logs the results in an xml log file in a predefined folder location. The `AutoTester` then determines whether the test macro completed the macro execution successfully or not by

comparing the AutoTester's log file against a pre-saved reference log file in the corresponding test case folder. Then AutoTester executes the remaining autotester script for tests that could be done in the Generis Browser or in the Generis Database, logs the overall test result in a test result log file, and then provides an appropriate color code for each test in its interface (i.e Red for failed test runs and Green for passed test runs [cf. Appendix 2.4])

5 Results

The first hypothesis was that automated tests reduce the testing time as compared to manual testing time. If this hypothesis proves to be valid, it would be positive considering the aim of this report. The second hypothesis was that writing a test automation code takes a considerable amount of time as compared to executing manual tests.

In order to check these hypotheses, ten test cases were randomly selected. The automated test run times were measured and compared against the time taken by manual testing to the same set of test cases. A full description of the data is presented in table 2.

Table 2 presents sample use cases that were taken from the test cases used to test the CRM customer services web application. It also shows the sample use cases that are randomly selected to compare and contrast the amount of time taken to test the use cases manually and by test automation.

Relative complexity in the fourth column of the table is based on how many windows and branches need to be tested and the length of steps that should be gone through during the test process. According to complexity, the test cases under consideration are classified into High, Medium and Low complexity test cases. If a test case involves opening more than three different windows or as many as five functionalities on average, it is regarded as a high complexity test case. However, if the complexity level is such that the test case needs to test less than three functionalities or opens not more than two windows, it is referred to as a medium category. Low complexity test cases are those test cases which interact with only one window and are simple enough to test minimum functionality, such as checking whether a hyperlink works or not.

As manual testing consumes a great deal of time in both the process of software development and during the software application testing, the integrated CRM test case automated framework has provided a faster option to complete test runs. It has also improved the efficiency of black

box testing where the testers do not have to be required the complex knowledge of the CSWeb application.

Table 2 Time measurement of executing manual and automated test runs

Use case	Time taken by manual testing	Time taken by automated testing	Relative complexity
Creating and deleting customer	17 minutes	2.3 minutes	MEDIUM
Creating bank account for a customer	12 minutes	2.7 minutes	MEDIUM
Adding notes to a metering point	16 minutes	3.5 minutes	MEDIUM
Creating and removing a metering point and changing work order	22 minutes	2.9	HIGH
Opening a district heating metering point by the use of Generis Server.	5 minutes	30 Seconds	LOW

As shown in table 2, the manual testing process takes a significant amount of time as compared to automated testing. In the experiment, automated testing took only 16 percent of the total time allocated for manual testing on the average. It was also noted that the automated testing process is not affected by the complexity of the test whereas the time taken to do the manual testing is directly proportional to the extent of complexity of the functionality to be tested.

The automation of the CSWeb test cases eliminated the possibility of human error when the same sequence of complex actions is repeated again and again in regression testing. The clean-up SQL queries run for each test run when using automated test runs. On the other hand, in manual testing mostly the cleanup work is forgotten and the testers have a tendency of moving

to the next test run without doing test cleanups once the test passes. There were also several instances in which test data objects were deleted mistakenly by software testers from the test servers during manual testing. Automated testing avoids the possibility of losing test case objects by only removing the intended cleanup data only.

Once the automation script is written well, testers can use the test suite that would help in testing each and application. This means that the chance of missing out key parts of testing is unlikely to occur. In other words, a clear well-written script with complete test coverage makes regression testing easily cover what the test case is designed for. Automated testing avoids the subjective decisions made by testers missing out features what they would think would logically work.

Implementing the test automation scripts in the AutoTester framework was not an easy task. The time taken to develop the CRMTTestAgent and integrating it to the existing AutoTester framework took much less than recording the macros and writing automation scripts in autoTester for 12 test cases. This shows that developing autotester scripts in verifying the Generis Browser software and database is not quite a straightforward process. Moreover, the existing Autotester provides support for Oracle databases only. AutoTester needs future improvements to accommodate executing SQL statements for Microsoft Sql Server databases.

Test case automation requires a continuous script review as there are some changes in requirements and implementations in the software development process. Some test scripts fail in almost every test cycle and test scripts need to maintain in each case. The mainline testing team usually fixes failed scripts when the monthly test cycle is over. This sometimes creates an overhead to the test case development process when the focus of attention is given to fixing the test case automation scripts.

6 Conclusion

This project was performed to provide an understanding of the need for automating test cases designed for regression testing. Regression testing is repeated for each test cycle making the manual testing an overhead to the entire software development life cycle.

This thesis has also demonstrated the design of the test case automation framework for the CSWeb application used in PV and analyzes the improvement of the software testing process and its impact on the efficiency of the entire software development life-cycle. In this regard, the test case automation has reduced the time taken by the manual testing to a significant level.

The other objective was to design and build an automated testing framework which would improve the testing process of the CSWeb application and shorten the overall software development life cycle by improving the software testing process relatively fast. The test framework enabled the automation of CSWeb testing and improved the performance of regression testing process. It also avoids the human errors that could be caused by manual testing.

Then finally taking sample CSWeb test cases, the overall time taken by the manual testing and automated testing process were measured. Then a comparison was made to demonstrate the advantages of test case automation over manual testing. In doing so, it was concluded that test case automation provided a fast way of running test cases independent of the test case complexity and avoided possible human error.

The overall automation development has been a slow process due to studying the existing automation tool and the third party product iMacros. In addition, some of the CSWeb test cases could not be automated because of having a separate database on the SQL server and the existing AutoTester frame work does not support such databases. The development of this tool can further be extended to support such databases.

References

- 1 Kent Beck. Test driven development by example. USA: Pearson Education; 2003.
- 2 Bentley John. Software testing fundamentals and concepts, roles, and terminology. Charlotte, NC; 2006.
- 3 Maric Brian. The test manager at the project status meeting. Software research; San Francisco, CA; 1997.
- 4 Forsberg Kevin, Mooz Harold. The Relationship of system engineering to the project cycle. Oslo, Norway; 1991.
- 5 Jacobson Ivar, Booch Grady, Rumbaugh Jim. Unified software development process. Boston, Massachusetts: Addison-Wesley; 1999.
- 6 Boehm Barry. A spiral model of software development and enhancement, IEEE Computer; 1988, 1: 64 -70.
- 7 Boehm Barry. Anchoring the software process. IEEE Software 1996, 13(4): 73-82.
- 8 Kruchten Philippe. A rational development process. Boston, Massachusetts: Addison-Wesley; 1996.
- 9 Jacobson Ivar, Christerson Magnus, Jonsson Patrik, Övergaard Gunnar. Object-oriented software engineering - A use case driven approach. Wokingham, England: Addison-Wesley; 1992.
- 10 Brown Alan. Component-based software engineering. full stop IEEE Computer Society: Los Alamitos, CA; 1996.
- 11 Booch Grady, Jacobson Ivar, Rumbaugh James. Unified modeling language 1.3. Reading, Massachusetts: Addison Wesley; 1998.

- 12 El-Far Ibrahim K., Whittaker James A.. Model-based software testing – To appear in J.J. Marciniak, editor. Encyclopedia of Software Engineering; 2001. 3: 6 -14.
- 13 Bran Alian, Moore James W. Guide to software engineering body of knowledge – Chapter 5 [online]. Los Alamitos, California: IEEE Computer Society; March 2004. URL:http://swebok.org/ironman/pdf/Swebok_Ironman_June_23_%202004.pdf. Accessed 2 May 2011.
- 14 Bruke Erik M.. Java and XSLT. USA: O'Reilly & Associates; 2001.
- 15 Seppänen Tommi. Enhancing user tasks in energy market software with user interface design patterns. Helsinki: Aalto University School of Science; 2011.
- 16 Steger Jim, Snyder Mike. Programming Microsoft Dynamics® CRM 4.0. USA: Microsoft Press; 2009.
- 17 Beck Kent. Test Driven Development By Example. Boston, MA, USA: Pearson Education; 2003.
- 18 Zallar Kerry [online]. Testing and test resources. Wolverhampton, England. URL: <http://www.testing.org>. Accessed: 2 March 2011.
- 19 Process Vison internal document. Component-Based Software Engineering: IEEE Computer Society: Los Alamitos, CA; 1996.

Appendices

Appendix 1. Sample code

CRM Test Agent

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ProcessVision.TestAgent.CRMTestAgent
{
    using System;
    using System.Diagnostics;
    using iMacros;
    using XMLLog;
    using Model;
    class CRMTestAgent
    {
        public static void RunImacros(string path, string macros_name)
        {
            Process startMacro = new Process();
            startMacro.StartInfo.FileName = path + macros_name;
            startMacro.Start();
        }

        public static void result_Display(ref TestCase testCase)
        {
            Console.WriteLine("TC Name: " + testCase.TestCaseName);
            Console.WriteLine("TC Number: " + testCase.TestCaseNumber);
            Console.WriteLine("Macros Name: " + testCase.MacrosName);
        }

        public static void LogStatus(iMacros.Status stat)
        {
            Console.WriteLine("Status: " + stat);
        }
        public static void LogStatus(iMacros.Status stat, String message)
        {
            Console.WriteLine("Message: " + message);
            Console.WriteLine("Status: " + stat);
        }

        public static void Main(string[] args)
        {
            int m_timeout = 10000;
            iMacros.App mapp;
            mapp = new iMacros.AppClass();
            iMacros.Status status;
            status = mapp.iimInit("", true, null, null, null, m_timeout);
            TestResult testReusltCompose = new TestResult();

            /* 1) For each Test case, it is needed to pass the following parameters to the TestAgent
            * a) Identify the Test case name and test case number
            * b) Identify the test Case number
            * c) Specify the path to the iim macro
            * d) Specify the macro file name (specify the full file name with file extension)
            * e) Specify the Licence Group of the test case
            */
        }
    }
}

```

```

* 2) Build this C# code for each test case
* 3) Find the exe file of the project and call it from your atscript file
*/

/*****
This is an example of the parameters required by the AutoTest Agent
string testCaseName = "CSweb(Metering point): Open district heating metering point by use of Generis Server";
string testCaseNumber = "CSWebXXX";
string pathToMacros = @"\\mainlinex64ifm\Mainline testing\Test Case Data\CS_WEB\CSWebXXX\";
string macroName = "CSWebXXX.iim";
string licenceGroup = "GENERIS Energy Vertical Licences";
string log_path = @"\\mainlinex64ifm\Mainline testing\Test Case Data\CS_WEB\CSWebXXX\";

*****/

try
{
string testCaseName = args[0];
string testCaseNumber = args[1];
string pathToMacros = args[2];
string macroName = args[3];
string licenceGroup = args[4];
string logPath = args[5];

LogStatus(status, "Initialising Macros App");

TestCase tcaseDemo = new TestCase(testCaseName, testCaseNumber, pathToMacros, macroName, licenceGroup);
testReusltCompose.LogPath = logPath + tcaseDemo.TestCaseNumber + ".xml";

Console.WriteLine("\n\n*****The parameters passed to the CRMTTestAgent*****");
Console.WriteLine("TcName: " + testCaseName);
Console.WriteLine("TcNumber: " + testCaseNumber);
Console.WriteLine("PathToMacros:" + pathToMacros);
Console.WriteLine("MacroName: " + macroName);
Console.WriteLine("LicenceGroup: " + licenceGroup);
Console.WriteLine("Log_path: " + logPath);
Console.WriteLine("Log" + testReusltCompose.LogPath);

status = mapp.iimPlay(tcaseDemo.MacrosPath + tcaseDemo.MacrosName, m_timeout);

testReusltCompose.ErrorCode = status.ToString();
testReusltCompose.ErrorMessage = mapp.iimGetLastError().ToString();

if (status > 0)
{
testReusltCompose.TestStatus = "Passed!";
LogStatus(status, mapp.iimGetLastError().ToString());
}
else
{
testReusltCompose.TestStatus = "Failed!";
testReusltCompose.ErrorMessage = mapp.iimGetLastError().ToString();
LogStatus(status);
}

status = mapp.iimExit(m_timeout);
XMLLogGenerator testLog = new XMLLogGenerator();
testLog.generateLogFile(tcaseDemo, testReusltCompose);
}
catch (IndexOutOfRangeException e)
{
{
Console.WriteLine("Invalid number of arguments passed to CRMTTestAgent. Check the correct number and sequence of
arguments when calling CRMTTestAgent");
}

Console.WriteLine(e.ToString());
Console.ReadLine();
}

```



```

TAG POS=14 TYPE=INPUT:TEXT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=01/08/2011

TAG POS=11 TYPE=INPUT:TEXT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT={{!VAR1}}

'SET !VAR4 {{!NOW:dd}}
'SET !VAR5 {{!NOW:mm}}
'SET !VAR6 {{!NOW:yyyy}}
'ADD !VAR4 {{!VAR5}} {{!VAR6}}

ONDIALOG POS=1 BUTTON=YES
ONDIALOG POS=2 BUTTON=YES
TAG POS=1 TYPE=INPUT:RADIO FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_rblReasonCode_1&&VALUE:SellerChange CONTENT=YES
TAG POS=1 TYPE=INPUT:RADIO FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_rblReasonCode_1&&VALUE:SellerChange CONTENT=YES
TAG POS=4 TYPE=SELECT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=%3678182
TAG POS=4 TYPE=SELECT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=%1153623
TAG POS=6 TYPE=SELECT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=%3677292
TAG POS=6 TYPE=SELECT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=%1359749
TAG POS=5 TYPE=SELECT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=%2
TAG POS=1 TYPE=A FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl*
TAG POS=7 TYPE=SELECT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=%3957546
TAG POS=20 TYPE=SELECT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=%0_0
TAG POS=26 TYPE=INPUT:TEXT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=250000
TAG POS=27 TYPE=INPUT:TEXT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl* CONTENT=140000
TAG POS=1 TYPE=INPUT:SUBMIT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl*&&VALUE:Save<SP>reading
TAG POS=1 TYPE=INPUT:SUBMIT FORM=ID:aspnetForm
ATTR=ID:ctl00_cphWorkspaceContent_dlnNewContractWorkflows_ctl00_ucStep2NewContractWorkfl*&&VALUE:Update<SP>forecas
t
TAG POS=1 TYPE=INPUT:SUBMIT FORM=ID:aspnetForm ATTR=ID:ctl00_btnNext&&VALUE:Next
'Confirm page
TAG POS=1 TYPE=INPUT:SUBMIT FORM=ID:aspnetForm ATTR=ID:ctl00_btnNext&&VALUE:Confirm

```

Sample AutoTester script

```

#const MacroName = "CSWeb1071.iim"
#const LicenceGroup = "GENERIS Energy Vertical Licences"
#const Log_path = "\\mainlinex64ifm\Mainline testing\Test Case Data\CS_WEB\CSWeb1071\"

#const STR_COMPARISON_LOG_FOLDER = "\\mainlinex64ifm\Mainline testing\Test Case
Data\CS_WEB\CSWeb1071\referenceLog"
#const STR_REF_LOG_FILE_NAME = "CSWeb1071_reference.xml"
#const STR_LOG_FOLDER = "\\mainlinex64ifm\Mainline testing\Test Case Data\CS_WEB\CSWeb1071\"
#const STR_LOG_FILE_NAME = "CSWeb1071.xml"

#const STR_USERNAME = "EDMSNETADMIN"
#const STR_PASSWORD = "EDMSNETP"
#const STR_DATASOURCE = "MAINDB"

'check the two deliveries are created
#const SQL_VERIFY_DELIVERIES = "select count(*) from g_delivery where scode like 'OSS000_609375M_%'"

'Check that there is no ending bill (count = 0)
#const SQL_VERIFY_ENDING_BILL = "select count(*) FROM G_BILL_PERIOD WHERE LID IN (SELECT BP.LID
FROM G_BILL_PERIOD BP, G_CONTRACT C, G_CONTRACT_BILLINFO_LINK CBL WHERE CBL.LCONTRACTID =
C.LOBJID AND BP.LBILLINFOID = CBL.LBILLINFOID AND C.SCODE = 'OSS000_609375M') AND bendingdelivery = 1"

'clean the new delivery and open the old delivery validity

#const SQL_RESET_1 = "UPDATE G_DELIVERY SET TSTOP = to_date('01.01.2081 00:00','dd.mm.yyyy hh24:mi')
WHERE SCODE = 'OSS000_609375M_1'"

#const SQL_RESET_2 = "UPDATE G_OBJECT_MAIN SET TSTOP = to_date('01.01.2081 00:00','dd.mm.yyyy hh24:mi')
where LOBJID = (SELECT LOBJID FROM G_DELIVERY WHERE SCODE = 'OSS000_609375M_1')"

#const SQL_RESET_3 = "delete from G_object_main where lobjid = (select Lobjid from g_delivery where scode =
'OSS000_609375M_2)'"

'Map Mainlinex64ifm\mainline testing\test case data\ to Z: drive
Shell.Open ("Z:\CS_WEB\CSWeb1071\CSWeb1071Starter", "", 450000)
Script.Sleep(3000)
'Window.WaitFor("iMacros v7.10.1044", 5000)

File.VerifyIdenticalTwoFiles (STR_COMPARISON_LOG_FOLDER, STR_REF_LOG_FILE_NAME, STR_LOG_FOLDER,
STR_LOG_FILE_NAME)
Script.Sleep(1000)

'Check ending delivery dates are correctly entered
Sql.Oracle.VerifyScalar (STR_USERNAME,STR_PASSWORD ,STR_DATASOURCE,SQL_VERIFY_DELIVERIES ,2)

'Check ending bill entry from the billing plan'
Sql.Oracle.VerifyScalar (STR_USERNAME,STR_PASSWORD,STR_DATASOURCE,SQL_VERIFY_ENDING_BILL ,0)

```

Appendix 2. Screenshots

CRM Test Agent console

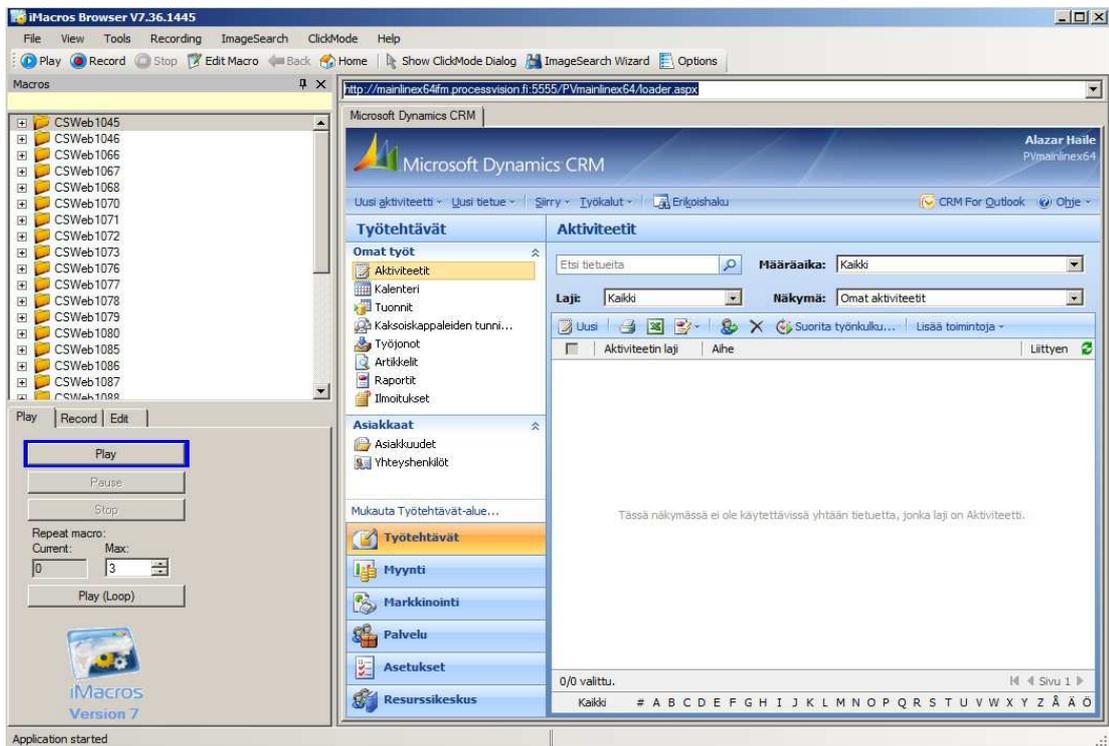
```

C:\Windows\system32\cmd.exe

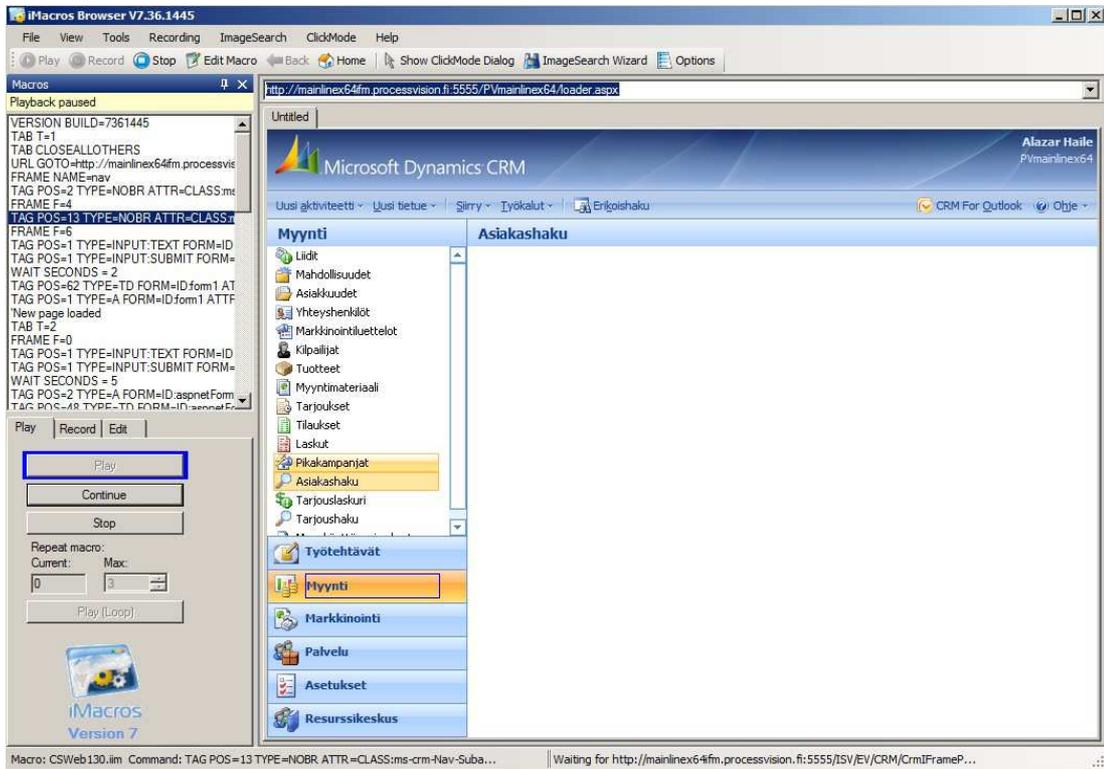
C:\Program Files (x86)\Process Vision\AutoTester\Bin>Z:\CS_WEB\CRMTestAgent "CSWeb(Custome):Create and remove a new customer" "CSWeb126" "\\mainline64ifm\Mainline Testing\Test Case Data\CS_WEB\CSWeb126\\" "CSWeb126.iim" "GENERIS Energy Vertical Licences" "\\mainline64ifm\Mainline Testing\Test Case Data\CS_WEB\CSWeb126\\"
Message: Initialising Macros App
Status: sOk

*****The parameters passed to the CRMTestAgent*****
ToName: CSWeb(Custome):Create and remove a new customer
ToNumber: CSWeb126
PathToMacros: \\mainline64ifm\Mainline Testing\Test Case Data\CS_WEB\CSWeb126\
MacroName: CSWeb126.iim
LicenceGroup: GENERIS Energy Vertical Licences
Log_path: \\mainline64ifm\Mainline Testing\Test Case Data\CS_WEB\CSWeb126\
Log\mainline64ifm\Mainline Testing\Test Case Data\CS_WEB\CSWeb126\CSWeb126.xml
  
```

iMacros browser opening a CSWeb application



iMacros replaying a macro on CSWeb application



AutoTester Batch runner showing an autotester script

