Testing Siebel Business Applications

Version 8.0

December 2006

ORACLE®

Contents

1 What's New in This Release

What's New in Testing Siebel Business Applications, Version 8.0

Table 1 lists changes described in this version of the documentation to support release 8.0 of the software.

Table 1. New Product Features in Testing Siebel Business Applications, Version 8.0

Topic	Description
"Hand-Scripting Functional Tests" on page 63	Added description of the Test Optimizer.
"SiebInkData Object" on page 102	Added new test automation object.
"SiebTask Object" on page 124 and "SiebTask Methods" on page 124	Added new test automation object and methods.
"SiebTaskLink Object" on page 126	Added new test automation object.
"SiebTaskStep Object" on page 127 and "SiebTaskStep Methods" on page 128	Added new test automation object and methods.
"SiebTaskUIPane Object" on page 128 and "SiebTaskUIPane Methods" on page 130	Added new test automation object and methods.

2 Overview of Testing Siebel Applications

This section provides an overview of the reasons for implementing testing in software development projects, and introduces a methodology for testing Oracle's Siebel Business Applications with descriptions of the processes and types of testing that are used in this methodology. This section includes the following topics:

About Testing Siebel Business Applications

This guide introduces and describes the processes and concepts of testing Siebel business applications. It is intended to be a guide for best practices for Oracle customers currently deploying or planning to deploy Siebel business applications, version 7.7 or later. It does not describe specific features of the Siebel Business Applications product suite.

Although job titles and duties at your company may differ from those listed in the following table, the audience for this guide consists primarily of employees in these categories:

Application Testers	Testers responsible for developing and executing tests. Functional testers focus on testing application functionality, while performance testers focus on system performance.
Business Analysts	Analysts responsible for defining business requirements and delivering relevant business functionality. Business analysts serve as the advocate for the business user community during application deployment.
Business Users	Actual users of the application. Business users are the customers of the application development team.
Database Administrators	Administrators who administer the database system, including data loading, system monitoring, backup and recovery, space allocation and sizing, and user account management.
Functional Test Engineers	Testers with the responsibility of developing and executing manual and automated testing. Functional test engineers create test cases and automate test scripts, maintain regression test library and report issues and defects.
Performance Test Engineers	Testers with the responsibility of developing and executing automated performance testing. Performance test engineers create automated test scripts, maintain regression test scripts and report issues and defects.
Project Managers	Manager or management team responsible for planning, executing, and delivering application functionality. Project managers are responsible for project scope, schedule, and resource allocation.
Siebel Application Developers	Developers who plan, implement, and configure Siebel business applications, possibly adding new functionality.
Siebel System Administrators	Administrators responsible for the whole system, including installing, maintaining, and upgrading Siebel business applications.
Test Architect	Working with the Test Manager, an architect designs and builds the test strategy and test plan.
Test Manager	Manages the day-to-day activities, testing resources, and test execution. Manages the reporting of test results and the defect management process. The Test Manager is the single point of contact (POC) for all testing activities.

NOTE: On simple projects, the Test Architect and Test Manager are normally combined into a single role.

How This Guide Is Organized

This book describes the processes for planning and executing testing activities for Siebel business applications. These processes are based on best practices and proven test methodologies. You use this book as a guide to identify what tests to run, when to run tests, and who to involve in the quality assurance process.

The first two chapters of this book provide an introduction to testing and the test processes. You are encouraged to read the remainder of this chapter "Overview of Testing Siebel Applications," which describes the relationships between the seven high-level processes. The chapters that follow describe a specific process in detail. In each of these chapters, a process diagram is presented to help you to understand the important high-level steps. You are encouraged to modify the processes to suit your specific situation.

Depending on your role, experience, and current project phases you will use the information in this book differently. Here are some suggestions about where you might want to focus your reading:

- **Test manager.** At the beginning of the project, review Chapters 2 through 8 to understand testing processes.
- **Functional testing.** If you are a functional tester focus on Chapters 3 through 7 and 9. These chapters discuss the process of defining, developing, and executing functional test cases.
- **Performance testing.** If you are a performance tester focus on Chapters 3, 4, 7, and 10. These chapters describe the planning, development, and execution of performance tests.

At certain points in this book, you will see information presented as a best practice. These tips are intended to highlight practices proven to improve the testing process.

Additional Resources

- American Society of Quality
  http://www.asq.org/pub/sqp
- Bitpipe
  http://www.bitpipe.com/rlist/term/Testing.html
- Economic Impact of Inadequate Infrastructure for Software Testing
  http://www.nist.gov/director/prog-ofc/report02-3.pdf
- Empirix
  http://www.empirix.com/Empirix/Corporate/Resources/
- International Federation for Information Processing
  http://www.ifip.or.at/
  (click on the "Search IFIP" link)
- Making Software Development High Performance
  http://www.swforum.com/
  (click on the "Search" hyperlink)
- Internet/Software Quality Hotlist
  http://www.soft.com/Institute/HotList/index.html
- Mercury Interactive
  http://download.mercury.com/cgi-bin/portal/download/index.jsp
- Software Testing Institute
  http://www.softwaretestinginstitute.com/index.html
- StickyMinds
  http://www.stickyminds.com/testing.asp

Introduction to Application Software Testing

Testing is a key component of any application deployment project. The testing process determines the readiness of the application. Therefore, it must be designed to adequately inform deployment decisions. Without well-planned testing, project teams may be forced to make under-informed decisions and expose the business to undue risk. Conversely, well-planned and executed testing can deliver significant benefit to a project including:

- **Reduced deployment cost.** Identifying defects early in the project is a critical factor in reducing the total cost of ownership. Research shows that the cost of resolving a defect increases dramatically in later deployment phases. A defect discovered in the requirements definition phase as a requirement gap can be a hundred times less expensive to address than if it is discovered after the application has been deployed. Once in production, a serious defect can result in lost business and undermine the success of the project.
- **Higher user acceptance.** User perception of quality is extremely important to the success of a deployment. Functional testing, usability testing, system testing, and performance testing can provide insights into deficiencies from the users' perspective early enough so that these deficiencies can be corrected before releasing the application to the larger user community.
- **Improved deployment quality.** Hardware and software components of the system must also meet a high level of quality. The ability of the system to perform reliably is critical in delivering consistent service to the users or customers. A system outage caused by inadequate system resources can result in lost business. Performance, reliability, and stress testing can provide an early assessment of the system to handle the production load and allow IT organizations to plan accordingly.

Inserting testing early and often is a key component to lowering the total cost of ownership. Software projects that attempt to save time and money by lowering their initial investment in testing find that the cost of not testing is much greater. Insufficient investment in testing may result in higher deployment costs, lower user adoption, and failure to achieve business returns.

Best Practice

Test early and often. The cost of resolving a defect when detected early is much less then resolving the same defect in later project stages. Testing early and often is the key to identifying defects as early as possible and reducing the total cost of ownership.

Application Software Testing Methodology

The processes described in this book are based on common test definitions for application software, and the Siebel eRoadmap implementation methodology. These definitions and methodologies have been proven in customer engagement, and demonstrate that testing must occur throughout the project lifecycle. For more information about the Siebel eRoadmap implementation methodology, see *Planning a Successful Siebel Implementation*.

Common Test Definitions

There are several common terms used to describe specific aspects of software testing. These testing classifications are used to break down the problem of testing into manageable pieces. Here are some of the common terms that are used throughout this book:

- **Business process testing.** Validates the functionality of two or more components that are strung together to create a valid business process.
- **Data conversion testing.** The testing of converted data used within the Siebel application. This is normally performed before system integration testing.
- **Functional testing.** Testing that focuses on the functionality of an application that validates the output based on selected input that consists of Unit, Module and Business Process testing.
- **Interoperability testing.** Applications that support multiple platforms or devices need to be tested to verify that every combination of device and platform works properly.
- **Negative testing.** Validates that the software fails appropriately by inputting a value known to be incorrect to verify that the action fails as expected. This allows you to understand and identify failures. By displaying the appropriate warning messages, you verify that the unit is operating correctly.
- **Performance testing.** This test is usually performed using an automation tool to simulate user load while measuring the system resources used. Client and server response times are both measured.
- **Positive testing.** Verifies that the software functions correctly by inputting a value known to be correct to verify that the expected data or view is returned appropriately.
- **Regression testing.** Code additions or changes may unintentionally introduce unexpected errors or regressions that did not exist previously. Regression tests are executed when a new build or release is available to make sure existing and new features function correctly.
- **Reliability testing.** Reliability tests are performed over an extended period of time to determine the durability of an application as well as to capture any defects that become visible over time.
- **Scalability testing.** Validates that the application meets the key performance indicators with a predefined number of concurrent users.
- **Stress testing.** This test identifies the maximum load a given hardware configuration can handle. Test scenarios usually simulate expected peak loads.
- **System integration testing.** This is a complete system test in a controlled environment to validate the end-to-end functionality of the application and all other interface systems (for example, databases and third-party systems). Sometimes adding a new module, application, or interface may negatively affect the functionality of another module.
- **Test case.** A test case contains the detailed steps and criteria (such as roles and data) for completing a test.
- **Test script.** A test script is an automated test case.
- **Unit testing.** Developers test their code against predefined design specifications. A unit test is an isolated test that is often the first feature test that developers perform in their own environment before checking changes into the configuration repository. Unit testing prevents introducing unstable components (or units) into the test environment.

- **Usability testing.** User interaction with the graphical user interface (GUI) is tested to observe the effectiveness of the GUI when test users attempt to complete common tasks.
- **User acceptance test (UAT).** Users test the complete, end-to-end business processes, verifying functional requirements (business requirements).

Siebel eRoadmap Implementation Methodology

The Siebel eRoadmap implementation methodology accelerates project implementations by focusing on the key strategic and tactical areas that must be addressed to maximize the customer's return on investment, while minimizing their business risk to promote a successful completion of a Siebel business application project. The Siebel eRoadmap implementation is comprised of activities logically grouped into distinct stages to make sure proper project management and control techniques are used during the life cycle of a project. These stages (illustrated in Figure 1) are iterative in nature, allowing customers to realize the benefits of their new business system. The test process maps to this iterative approach by aligning testing to the early stages of eRoadmap, to support early and ongoing testing and communication.

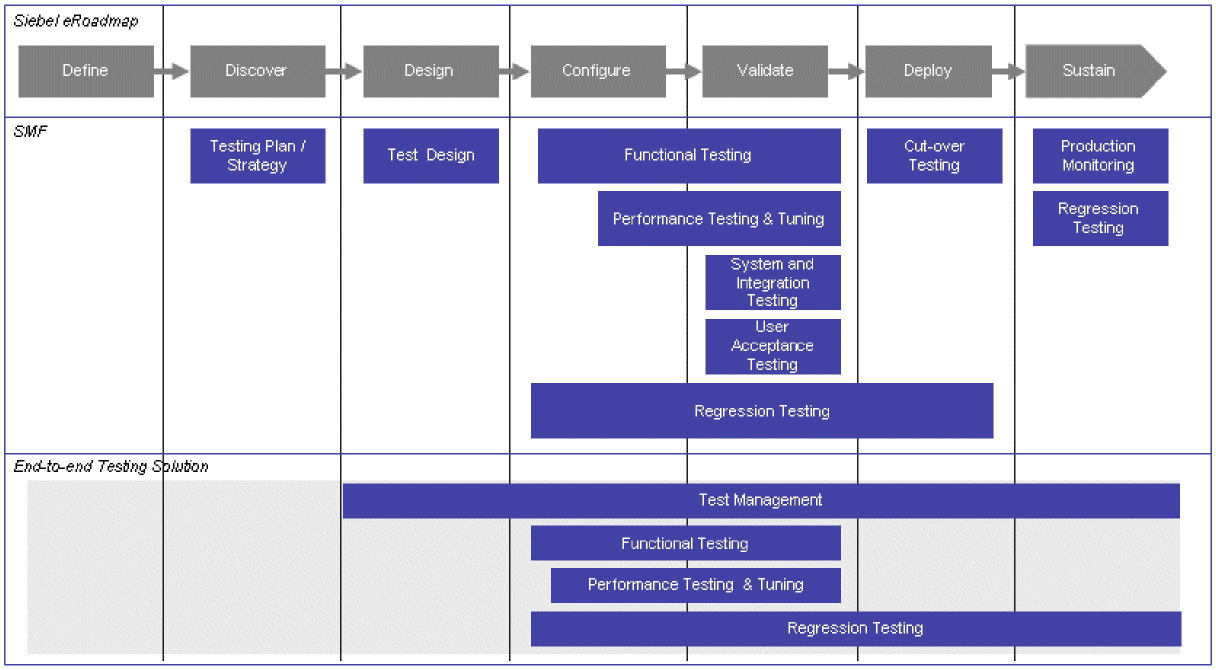


Figure 1. Siebel eRoadmap Implementation Methodology

Testing is an end-to-end process that begins when you begin to define the requirements for your Siebel application. The first stage is the development of a testing strategy by the testing team that establishes priorities and defines the testing approach, resources, hardware and software requirements, and the appropriate test cases. The output of this stage is a comprehensive test plan.

After the test plan is created and published, the test team begins to create detailed test cases. Common inputs to this stage are use cases, business processes, design documentation, and business requirements.

Functional testing begins after prototyping begins and continues throughout the configuration of the Siebel application as developers test each unit they configure. Tested units are then moved into the testing environment, where the appropriate units are combined to form a corresponding module. The test team then verifies whether or not the module functions correctly (for example, returns the correct value), has the correct layout (for example, drop-down menus and text fields), and the interface is correct. After validating a module, functional testing continues by combining two or more modules to create a valid business process or scenario to verify whether or not all modules work together as required.

Performance testing begins after you have a business process configured. The testing begins using a bottom-up approach that normally uses three servers to begin testing the performance of a single user. This stage provides an assessment of whether or not the application satisfies the key performance indicators (KPIs).

After all of the designated business processes are tested individually, the testing continues in a production-like test environment. This phase requires an image of the full database and all interfaces with the Siebel application (such as CTI, middleware, and email). The first step is to establish a benchmark of performance using the completion of a performance test by running all of the defined business processes. Next, perform a scalability test by adding users until you reach the number of users expected to use the system over the life of the application. Finally, execute a reliability test over an extended period of time to determine durability of the application and capture any defects that become visible over time.

After completing the functional testing, the next stage is the data conversion testing to validate that the converted data is correct. System and integration testing starts after the data conversion testing is completed, and the data is loaded into the test environment.

System and integration testing validates that the Siebel application operates with other systems and interfaces. Test the Siebel application in a test environment that allows the Siebel application to interoperate with the other required systems (such as CTI, load balancer, and middleware).

User acceptance testing (UAT) consists of testing the Siebel application using the appropriate business owners and end users. When performing UAT, make sure that you have users who are familiar with the existing business processes.

For more information about the Siebel eRoadmap implementation methodology, see *Planning a Successful Siebel Implementation*.

Modular and Iterative Methodology

An IT project best practice that applies to both testing and development is to use a modular and incremental approach to develop and test applications to detect potential defects earlier rather than later. This approach provides component-based test design, test script construction (automation), execution and analysis. It brings the defect management stage to the forefront, promoting communication between the test team and the development team. Beginning the testing process early in the development cycle helps reduce the cost to fix defects.

This process begins with the test team working closely with the development team to develop a schedule for the delivery of functionality (a drop schedule). The test team uses this schedule to plan resources and tests. In the earlier stages, testing is commonly confined to unit and module testing. After one or more drops, there is enough functionality to begin to string the modules together to test a business process.

After the development team completes the defined functionality, they compile and transfer the Siebel application into the test environment. The immediate functional testing by the test team allows for early feedback to the development team regarding possible defects. The development team can then schedule and repair the defects, drop a new build of the Siebel application, and provide the opportunity for another functional test session after the test team updates the test scripts as necessary.

Best Practice

Iterative development introduces functionality to a release in incremental builds. This approach reduces risk by prompting early communication and allowing testing to occur more frequently and with fewer changes to all parts of the application.

Continuous Application Lifecycle

One deployment best practice is the continuous application lifecycle. In this approach, application features and enhancements are delivered in small packages on a continuous delivery schedule. New features are considered and scheduled according to a fixed release schedule (for example, once every quarter). This model of phased delivery provides an opportunity to evaluate the effectiveness of prebuilt application functionality, minimizes risk, and allows you to adapt the application to changing business requirements.

Continuous application lifecycle incorporates changing business requirements into the application on a regular timeline, so the business customers do not have a situation where they become locked into functionality that does not quite meet their needs. Because there is always another delivery date on the horizon, features do not have to be rushed into production. This approach also allows an organization to separate multiple, possibly conflicting change activities. For example, the upgrade (repository merge) of an application can be separated from the addition of new configuration.

Best Practice

The continuous application lifecycle approach to deployment allows organizations to reduce the complexity and risk on any single release and provides regular opportunities to enhance application functionality.

Testing and Deployment Readiness

The testing processes provide crucial inputs for determining deployment readiness. Determining whether or not an application is ready to deploy is an important decision that requires clear input from testing.

Part of the challenge in making a good decision is the lack of well-planned testing and the availability of testing data to gauge release readiness. To address this, it is important to plan and track testing activity for the purpose of informing the deployment decision. In general, you can measure testing coverage and defect trends, which provide a good indicator of quality. The following are some suggested analyses to compile:

- For each test plan, the number and percentage of test cases passed, in progress, failed, and blocked. This data illustrates the test objectives that have been met, versus those that are in progress or at risk.
- Trend analysis of open defects targeted at the current release for each priority level.
- Trend analysis of defects discovered, defects fixed, and test cases executed. Application convergence (point A in Figure 2) is demonstrated by a slowing of defect discovery and fix rates, while maintaining even levels of test case activity.

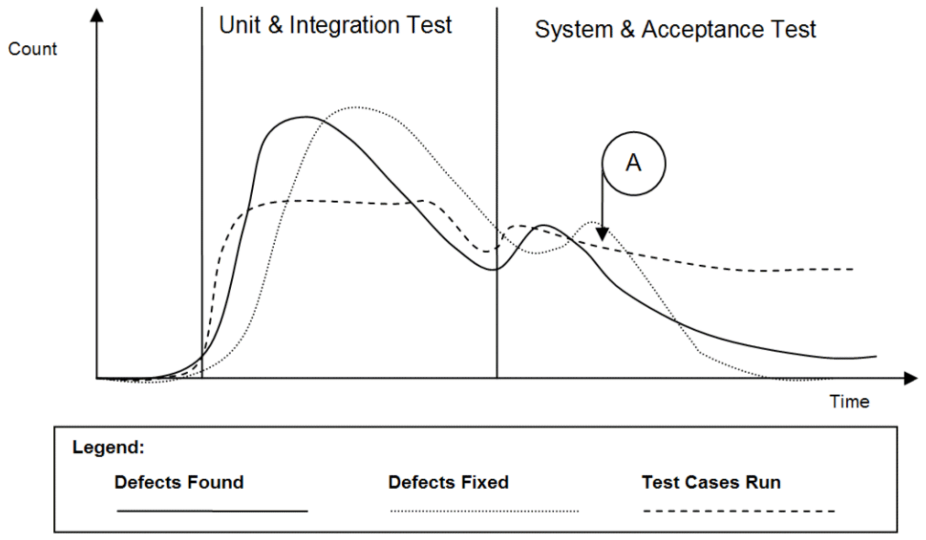


Figure 2. Trend Analysis of Testing and Defect Resolution

Testing is a key input to the deployment readiness decision. However it is not the only input to be considered. You must consider testing metrics with business conditions and organizational readiness.

Overview of the Siebel Testing Process

Testing processes occur throughout the implementation lifecycle, and are closely linked to other configuration, deployment, and operations processes. Figure 3 presents a high-level map of testing processes.

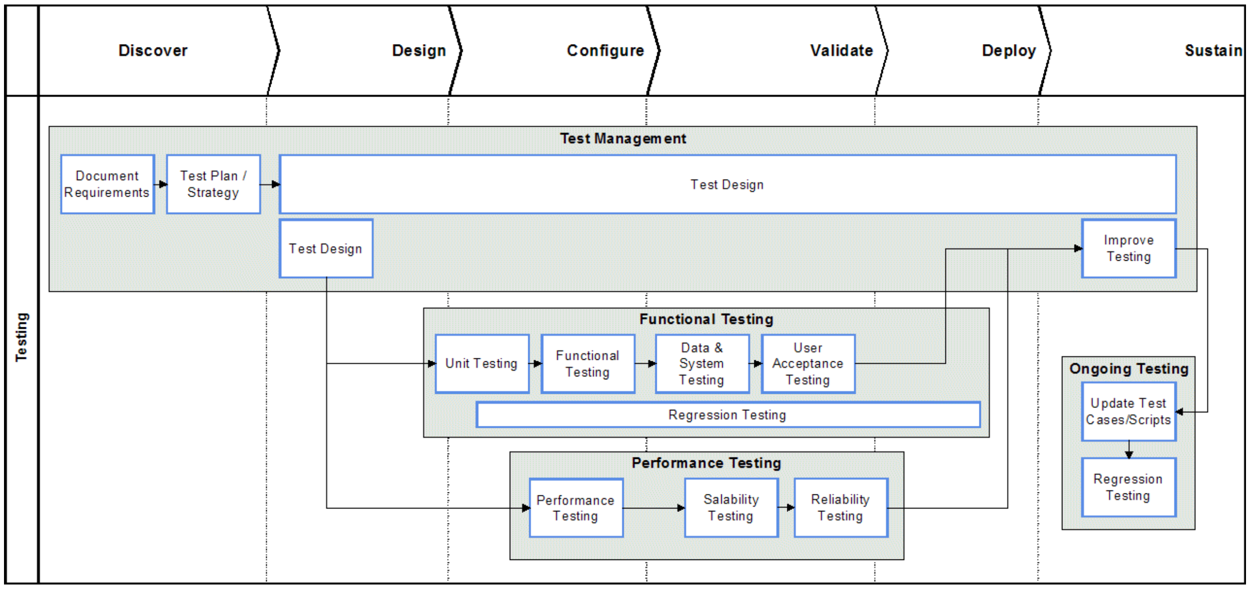


Figure 3. High-Level Testing Process Map

Each of the seven testing processes described in this book are highlighted in bold in the diagram and are outlined briefly in the following topics:

- Plan Testing Strategy on page 22
- Design and Develop Tests on page 23
- Execute Siebel Functional Tests on page 23
- Execute System Integration Tests on page 23
- Execute Acceptance Tests on page 23
- Execute Performance Tests on page 24
- Improve and Continue Testing on page 24

Plan Testing Strategy

The test planning process makes sure that the testing performed is able to inform the deployment decision process, minimize risk, and provide a structure for tracking progress. Without proper planning many customers may perform either too much or too little testing. The process is designed to identify key project objectives and develop plans based on those objectives.

It is important to develop a testing strategy early, and to use effective communications to coordinate among all stakeholders of the project.

Design and Develop Tests

In the test design process, the high-level test cases identified during the planning process are developed in detail (step-by-step). Developers and testers finalize the test cases based on approved technical designs. The written test cases can also serve as blueprints for developing automated test scripts. Test cases should be developed with strong participation from the business analyst to understand the details of usage, and less-common use cases.

Design evaluation is the first form of testing, and often the most effective. Unfortunately, this process is often neglected. In this process, business analysts and developers verify that the design meets the business unit requirements. Development work should not start in earnest until there is agreement that the designed solution meets requirements. The business analyst who defines the requirements should approve the design.

Preventing design defects or omissions at this stage is more cost effective than addressing them later in the project. If a design is flawed from the beginning, the cost to redesign after implementation can be high.

Execute Siebel Functional Tests

Functional testing is focused on validating the Siebel business application components of the system. Functional tests are performed progressively on components (units), modules, and business processes in order to verify that the Siebel application functions correctly. Test execution and defect resolution are the focus of this process. The development team is fully engaged in implementing features, and the defect-tracking process is used to manage quality.

Execute System Integration Tests

System integration testing verifies that the Siebel application validated earlier, integrates with other applications and infrastructure in your system. Integration with various back-end, middleware, and third-party systems are verified. Integration testing occurs on the system as a whole to make sure that the Siebel application functions properly when connected to related systems, and when running along side system-infrastructure components.

Execute Acceptance Tests

Acceptance testing is performed on the complete system and is focused on validating support for business processes, as well as verifying acceptability to the user community from both the lines of business and the IT organization. This is typically a very busy time in the project, when people, process, and technology are all preparing for the rollout.

Execute Performance Tests

Performance testing validates that the system can meet specified key performance indicators (KPIs) and service levels for performance, scalability, and reliability. In this process, tests are run on the complete system simulating expected loads and verifying system performance.

Improve and Continue Testing

Testing is not complete when the application is rolled out. After the initial deployment, regular configuration changes are delivered in new releases. In addition, Oracle delivers regular maintenance and major software releases that may need to be applied. Both configuration changes and new software releases require regression testing to verify that the quality of the system is sustained.

The testing process should be evaluated after deployment to identify opportunities for improvement. The testing strategy and its objectives should be reviewed to identify any inadequacies in planning. Test plans and test cases should be reviewed to determine their effectiveness. Test cases should be updated to include testing scenarios that were discovered during testing and were not previously identified, to reflect all change requests, and to support software releases.

3 Plan Testing Strategy

This section describes the process of planning your tests. It includes the following topics:

- Overview of Test Planning
- Test Objectives on page 26
- Test Plans on page 27
- Test Environments on page 32

Overview of Test Planning

The objective of the test planning process is to create the strategy and tactics that provide the proper level of test coverage for your project. The test planning process is illustrated in Figure 4 on page 26.

The inputs to this process are the business requirements and the project scope. The outputs, or deliverables, of this process include:

- **Test objectives.** The high-level objectives for a quality release. The test objectives are used to measure project progress and deployment readiness. Each test objective has a corresponding business or design requirement.
- **Test plans.** The test plan is an end-to-end test strategy and approach for testing the Siebel application. A typical test plan contains the following sections:
  - **Strategy, milestones, and responsibilities.** Set the expectation for how to perform testing, how to measure success, and who is responsible for each task
  - **Test objectives.** Define and validate the test goals, objectives, and scope
  - **Approach.** Outlines how and when to perform testing
  - **Entrance and exit criteria.** Define inputs required to perform a test and success criteria for passing a test
  - **Results reporting.** Outlines the type and schedule of reporting
- **Test cases.** A test plan contains a set of test cases. Test cases are detailed, step-by-step instructions about how to perform a test. The instructions should be specific and repeatable by anyone who typically performs the tasks being tested. In the planning process, you identify the number and type of test cases to be performed.

- **Definition of test environments.** The number, type, and configuration for test environments should also be defined. Clear entry and exit criteria for each environment should be defined.

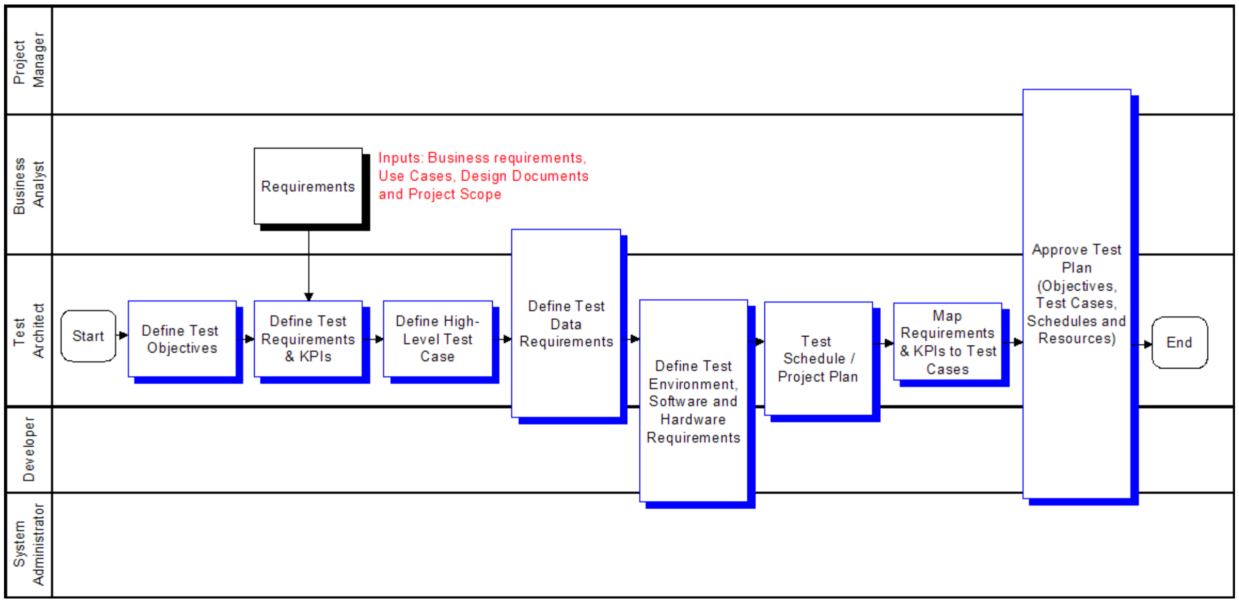

Project Manager
Business Analyst
Test Architect
Developer
System Administrator
Requirements
Inputs: Business requirements, Use Cases, Design Documents and Project Scope
Start
Define Test Objectives
Define Test Requirements & KPIs
Define High-Level Test Case
Define Test Data Requirements
Define Test Environment, Software and Hardware Requirements
Test Schedule / Project Plan
Map Requirements & KPIs to Test Cases
Approve Test Plan (Objectives, Test Cases, Schedules and Resources)
End

Figure 4. Plan Testing Strategy Process

Test Objectives

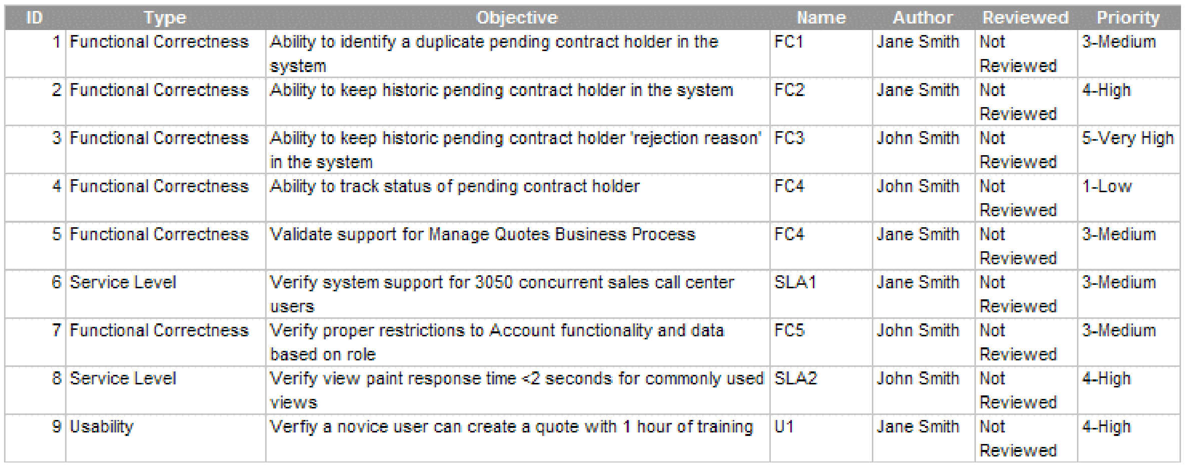
The first step in the test planning process is to document the high-level test objectives. The test objectives provide a prioritized list of verification or validation objectives for the project. You use this list of objectives to measure testing progress, and verify that testing activity is consistent with project objectives.

Test objectives can typically be grouped into the following categories:

- **Functional correctness.** Validation that the application correctly supports required business processes and transactions. List all of the business processes that the application is required to support. Also list any standards for which there is required compliance.
- **Authorization.** Verification that actions and data are available only to those users with correct authorization. List any key authorization requirements that must be satisfied, including access to functionality and data.
- **Service level.** Verification that the system will support the required service levels of the business. This includes system availability, load, and responsiveness. List any key performance indicators (KPIs) for service level, and the level of operational effort required to meet KPIs.
- **Usability.** Validation that the application meets required levels of usability. List the required training level and user KPIs required.

The testing team, development team, and the business unit agree upon the list of test objectives and their priority. Figure 5 shows a sample Test Objectives document.

A test case covers one or more test objective, and has the specific steps that testers follow to verify or validate the stated objectives. The details of the test plan are described in "Test Plans" on page 27.



ID	Type	Objective	Name	Author	Reviewed	Priority
1	Functional Correctness	Ability to identify a duplicate pending contract holder in the system	FC1	Jane Smith	Not Reviewed	3-Medium
2	Functional Correctness	Ability to keep historic pending contract holder in the system	FC2	Jane Smith	Not Reviewed	4-High
3	Functional Correctness	Ability to keep historic pending contract holder 'rejection reason' in the system	FC3	John Smith	Not Reviewed	5-Very High
4	Functional Correctness	Ability to track status of pending contract holder	FC4	John Smith	Not Reviewed	1-Low
5	Functional Correctness	Validate support for Manage Quotes Business Process	FC4	Jane Smith	Not Reviewed	3-Medium
6	Service Level	Verify system support for 3050 concurrent sales call center users	SLA1	Jane Smith	Not Reviewed	3-Medium
7	Functional Correctness	Verify proper restrictions to Account functionality and data based on role	FC5	John Smith	Not Reviewed	3-Medium
8	Service Level	Verify view paint response time <2 seconds for commonly used views	SLA2	John Smith	Not Reviewed	4-High
9	Usability	Verfiy a novice user can create a quote with 1 hour of training	U1	Jane Smith	Not Reviewed	4-High

Figure 5. Sample Test Objectives

Test Plans

The purpose of the test plan is to define a comprehensive approach to testing. This includes a detailed plan for verifying the stated objectives, identifying any issues, and communicating schedules towards the stated objectives. The test plan has the following components:

- **Project scope.** Outlines the goals and what is included in the testing project.
- **Test cases.** Detail level test scenarios. Each test plan is made up of a list of test cases, their relevant test phases (schedule), and relationship to requirements (traceability matrix).
- **Business process script inventory and risk assessment.** A list of components (business process scripts) that require testing. Also describes related components and identifies high-risk components or scenarios that may require additional test coverage.
- **Test schedule.** A schedule that describes when test cases will be executed.
- **Test environment.** Outlines the recommendations for the various test environments (for example, Functional, System Integration, and Performance). This includes both software and hardware.
- **Test data.** Identifies the data required to support testing.

Business process testing is an important best practice. Business process testing drives the test case definition from the definition of the business process. In business process testing, coverage is measured based on the percentage of validated process steps.

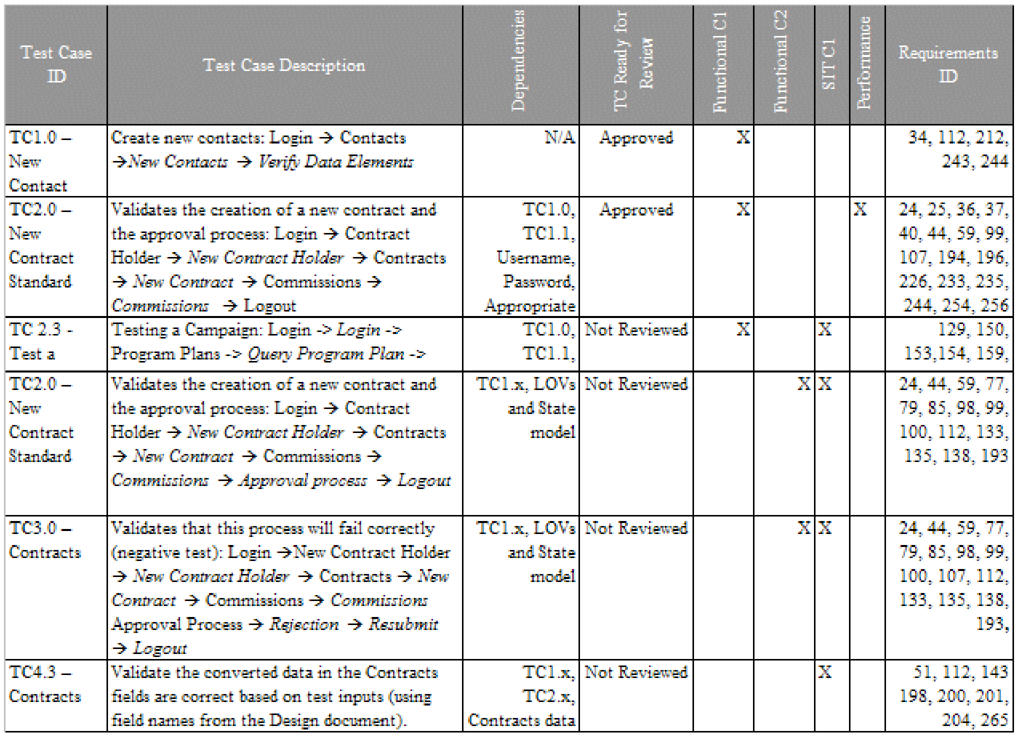
Best Practice

Functional testing based on a required business process definition provides a structured way to design test cases, and a meaningful way to measure test coverage based on business process steps.

Business process testing is described in more detail in the sections that follow.

Test Cases

A test case represents an application behavior that needs to be verified. For each component, application, and business process you can identify one or more test cases that need verification. Figure 6 shows a sample test case list. Each test plan contains multiple test cases.



Test Case ID	Test Case Description	Dependencies	TC Ready for Review	Functional C1	Functional C2	SIT C1	Performance	Requirements ID
TC1.0 – New Contact	Create new contacts: Login → Contacts →*New Contacts* → *Verify Data Elements*	N/A	Approved	X				34, 112, 212, 243, 244
TC2.0 – New Contract Standard	Validates the creation of a new contract and the approval process: Login → Contract Holder → *New Contract Holder* → Contracts → *New Contract* → Commissions → *Commissions* → Logout	TC1.0, TC1.1, Username, Password, Appropriate	Approved	X			X	24, 25, 36, 37, 40, 44, 59, 99, 107, 194, 196, 226, 233, 235, 244, 254, 256
TC 2.3 - Test a	Testing a Campaign: Login -> *Login* -> Program Plans -> *Query Program Plan* ->	TC1.0, TC1.1,	Not Reviewed	X		X		129, 150, 153,154, 159,
TC2.0 – New Contract Standard	Validates the creation of a new contract and the approval process: Login → Contract Holder → *New Contract Holder* → Contracts → *New Contract* → Commissions → *Commissions* → *Approval process* → *Logout*	TC1.x, LOVs and State model	Not Reviewed		X	X		24, 44, 59, 77, 79, 85, 98, 99, 100, 112, 133, 135, 138, 193
TC3.0 – Contracts	Validates that this process will fail correctly (negative test): Login →New Contract Holder → *New Contract Holder* → Contracts → *New Contract* → Commissions → *Commissions* Approval Process → *Rejection* → *Resubmit* → *Logout*	TC1.x, LOVs and State model	Not Reviewed		X	X		24, 44, 59, 77, 79, 85, 98, 99, 100, 107, 112, 133, 135, 138, 193,
TC4.3 – Contracts	Validate the converted data in the Contracts fields are correct based on test inputs (using field names from the Design document).	TC1.x, TC2.x, Contracts data	Not Reviewed			X		51, 112, 143 198, 200, 201, 204, 265

Figure 6. Sample Test Plan: Test Case List

This example uses the following numbering schema for the Test Case ID:

TC1.x – New records and seed data required to support other test cases

TC2.x – Positive test cases

TC3.x – Negative test cases

TC4.x – Data Conversion testing

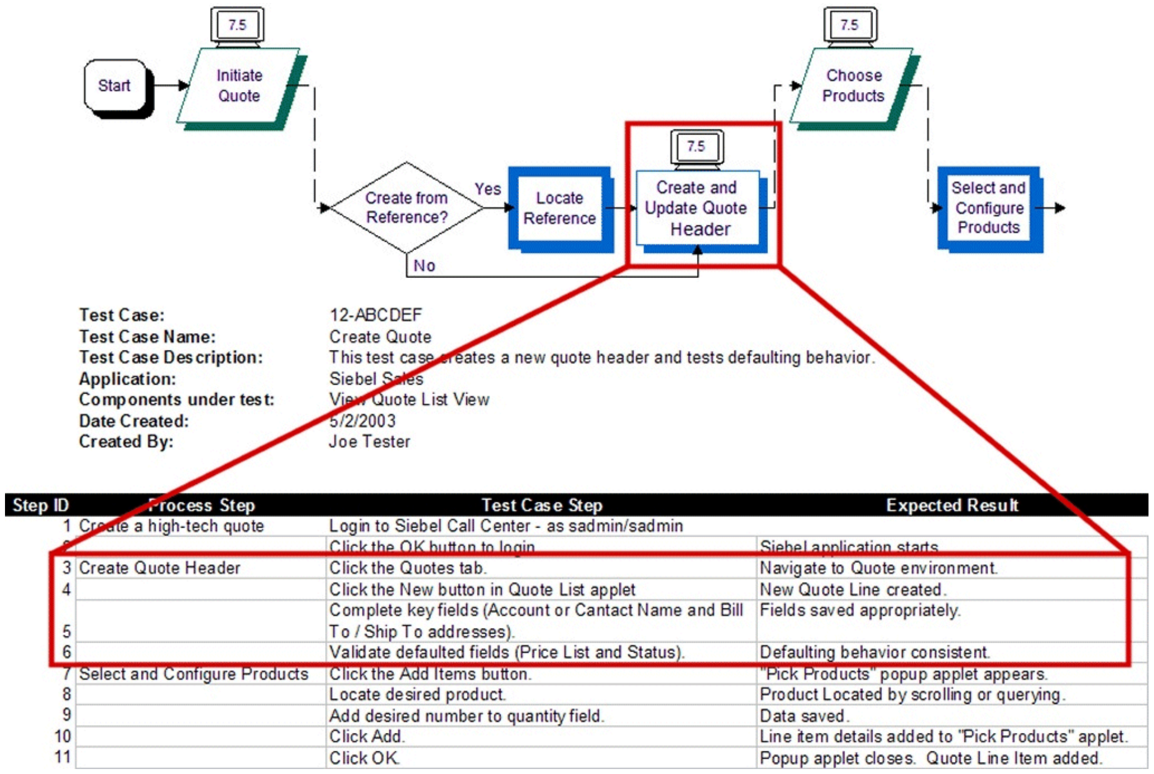
TC5.x – System integration testing

Note how the test schedule is included in Figure 6. For example, TC1.0 – New Contact is performed during Functional Cycle 1 (Functional C1) of the functional testing. Whereas, TC3.0 – Contracts occurs during Functional Cycle 2 (Functional C2) and during system integration testing.

During the Design phase of the test plan, there are a number of test types that you must define:

- **Functional test cases.** Functional test cases are designed to validate that the application performs a specified business function. The majority of these test cases take the form of user or business scenarios that resemble common transactions. Testers and business users should work together to compile a list of scenarios. Following the business process testing practice, functional test cases should be derived directly from the business process, where each step of the business process is clearly represented in the test case.

  For example, if the test plan objective is to validate support for the Manage Quotes Business Process, then there should be test cases specified based on the process definition. Typically, this means that each process or subprocess has one or more defined test cases, and each step in the process is specified within the test case definition. Figure 7 illustrates the concept of a process-driven test case. Considerations must also be given for negative test cases that test behaviors when unexpected actions are taken (for example, creation of a quote with a create date before the current date).



Test Case:	12-ABCDEF
Test Case Name:	Create Quote
Test Case Description:	This test case creates a new quote header and tests defaulting behavior.
Application:	Siebel Sales
Components under test:	View Quote List View
Date Created:	5/2/2003
Created By:	Joe Tester

Step ID	Process Step	Test Case Step	Expected Result
1	Create a high-tech quote	Login to Siebel Call Center - as sadmin/sadmin	
2		Click the OK button to login.	Siebel application starts.
3	Create Quote Header	Click the Quotes tab.	Navigate to Quote environment.
4		Click the New button in Quote List applet	New Quote Line created.
5		Complete key fields (Account or Cantact Name and Bill To / Ship To addresses).	Fields saved appropriately.
6		Validate defaulted fields (Price List and Status).	Defaulting behavior consistent.
7	Select and Configure Products	Click the Add Items button.	"Pick Products" popup applet appears.
8		Locate desired product.	Product Located by scrolling or querying.
9		Add desired number to quantity field.	Data saved.
10		Click Add.	Line item details added to "Pick Products" applet.
11		Click OK.	Popup applet closes. Quote Line Item added.

Figure 7. Business Process-Driven Test Case with its Corresponding Process Diagram

- **Structural test cases.** Structural test cases are designed to verify that the application structure is correct. They differ from functional cases in that structural test cases are based on the structure of the application, not on a scenario. Typically, each component has an associated structural test case that verifies that the component has the correct layout and definition (for example, verify that a view contains all the specified applets and controls).

- **Performance test cases.** Performance test cases are designed to verify the performance of the system or a transaction. There are three categories of performance test cases commonly used:
  - **Response time or throughput.** Verifies the time for a set of specified actions. For example, tests the time for a view to paint or a process to run. Response time tests are often called *performance tests*.
  - **Scalability.** Verifies the capacity of a specified system or component. For example, test the number of users that the system can support. Scalability tests are often called *load* or *stress* tests.
  - **Reliability.** Verifies the duration for which a system or component can be run without the need for restarting. For example, test the number of days that a particular process can run without failing.

Test Phase

Each test case should have a primary testing phase identified. You can run a given test case several times in multiple testing phases, but typically the first phase in which you run it is considered the primary phase. The following describes how standard testing phases typically apply to Siebel business application deployments:

- **Unit test.** The objective of the unit test is to verify that a unit (also called a component) functions as designed. The definition of a unit is discussed in "Component Inventory" on page 31. In this phase of testing, in-depth verification of a single component is functionally and structurally tested.

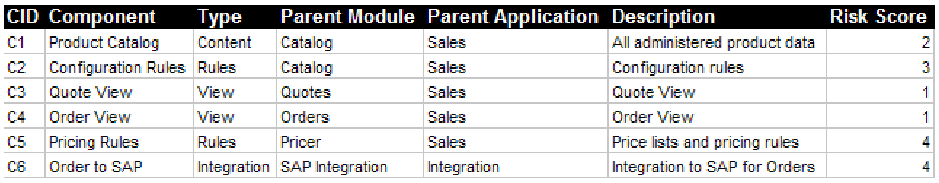
  For example, during the unit test the developer of a newly configured view verifies that the view structure meets specification and validates that common user scenarios, within the view, are supported.

- **Module test.** The objective of the module test is to validate that related components fit together to meet specified application design criteria. In this phase of testing, functional scenarios are primarily used. For example, testers will test common navigation paths through a set of related views. The objective of this phase of testing is to verify that related Siebel components function correctly as a module.
- **Process test.** The objective of the process test is to validate that business process are supported by the Siebel application. During the process test, the previously-tested modules are strung together to validate an end-to-end business process. Functional test cases, based on the defined business processes are used in this phase.
- **Data conversion test.** The objective of the data conversion test is to validate that the data is properly configured and meets all requirements. This should be performed before the integration test phase.
- **Integration test.** In the integration test phase, the integration of the Siebel business application with other back-end, middleware, or third-party components are tested. This phase includes functional test cases and system test cases specific to integration logic. For example, in this phase the integration of Siebel Orders with an ERP Order Processing system is tested.
- **Acceptance test.** The objective of the acceptance test is to validate that the system is able to meet user requirements. This phase consists primarily of formal and ad-hoc functional tests.

- **Performance test.** The objective of the performance test is to validate that the system will support specified performance KPIs, maintenance, and reliability requirements. This phase consists of performance test cases.

Component Inventory

The Component Inventory is a comprehensive list of the applications, modules, and components in the current project. Typically, the component inventory is done at the project level, and is not a testing-specific activity. There are two ways projects typically identify components. The first is to base component definition on the work that needs to be done (for example, specific configuration activities). The second method is to base the components on the functionality to be supported. In many cases, these two approaches produce similar results. A combination of the two methods is most effective in making sure that the test plan is complete and straightforward to execute. The worksheet shown in Figure 8 is an example of a component inventory.



CID	Component	Type	Parent Module	Parent Application	Description	Risk Score
C1	Product Catalog	Content	Catalog	Sales	All administered product data	2
C2	Configuration Rules	Rules	Catalog	Sales	Configuration rules	3
C3	Quote View	View	Quotes	Sales	Quote View	1
C4	Order View	View	Orders	Sales	Order View	1
C5	Pricing Rules	Rules	Pricer	Sales	Price lists and pricing rules	4
C6	Order to SAP	Integration	SAP Integration	Integration	Integration to SAP for Orders	4

Figure 8. Sample Component Inventory Document

Risk Assessment

A risk assessment is used to identify those components that carry higher risk and may require enhanced levels of testing. The following characteristics increase component risk:

- **High business impact.** The component supports high business-impact business logic (for example, complex financial calculation).
- **Integration.** This component integrates the Siebel application to an external or third-party system.
- **Scripting.** This component includes the coding of browser script, eScript, or VB script.
- **Ambiguous or incomplete design.** This component design is either ambiguous (for example, multiple implementation options described) or the design is not fully specified.
- **Availability of data.** Performance testing requires production-like data (a data set that has the same shape and size as that of the production environment). This task requires planning, and the appropriate resources to stage the testing environment.
- **Downstream dependencies.** This component is required by several downstream components.

As shown in Figure 8 on page 31, one column of the component inventory provides a risk score to each component based on the guidelines above. In this example one risk point is given to a component for each of the criteria met. The scoring system should be defined to correctly represent the relative risk between components. Performing a risk assessment is important for completing a test plan, because the risk assessment provides guidance on the sequence and amount of testing required.

Best Practice

Performing a risk assessment during the planning process allows you to design your test plan in a way that minimizes overall project risk.

Test Plan Schedule

For each test plan, a schedule of test case execution should be specified. The schedule is built using four different inputs:

- **Overall project schedule.** The execution of all test plans must be consistent with the overall project schedule.
- **Component development schedule.** The completion of component configuration is a key input to the testing schedule.
- **Environment availability.** The availability of the required test environment needs to be considered in constructing schedules.
- **Test case risk.** The risk associated with components under test is another important consideration in the overall schedule. Components with higher risk should be tested as early as possible.

Test Environments

The specified test objectives influence the test environment requirements. For example, service level test objectives (such as system availability, load, and responsiveness) often require an isolated environment to verify. In addition, controlled test environments can help:

- **Provide integrity of the application under test.** During a project, at any given time there are multiple versions of a module or system configuration. Maintaining controlled environments can make sure that tests are being executed on the appropriate versions. Significant time can be wasted executing tests on incorrect versions of a module or debugging environment configuration without these controls.
- **Control and mange risk as a project nears rollout.** There is always a risk associated with introducing configuration changes during the lifecycle of the project. For example, changing the configuration just before the rollout carries a significant amount of risk. Using controlled environments allows a team to isolate late-stage and risky changes.

It is typical to have established Development, Functional Testing, System Testing, User Acceptance Testing, Performance Testing, and Production environments to support testing. More complex projects often include more environments, or parallel environments to support parallel development. Many customers use standard code control systems to facilitate the management of code across environments.

The environment management approach includes the following components:

- **Named environments and migration process.** A set of named test environments, a specific purpose (for example, integration test environment), and a clear set of environment entry and exit criteria. Typically, the movement of components from one environment to the next requires that each component pass a predefined set of test cases, and is done with the appropriate level of controls (for example, code control and approvals).
- **Environment audit.** A checklist of system components and configuration for each environment. Audits are performed prior to any significant test activity. The Environment Verification Tool can be used to facilitate the audit of test environments. For more information about the Environment Verification Tool, see Oracle's Siebel SupportWeb at http://supportweb.siebel.com/.
- **Environment schedule.** A schedule that outlines the dates when test cases will be executed in a given environment.

Performance Test Environment

In general, the more closely the performance test environment reflects the production environment, the more applicable the test results will be. It is important that the performance test environment includes all of the relevant components to test all aspects of the system, including integration and third-party components. Often it is not feasible to build a full duplicate of the production configuration for testing purposes. In that case, the following scaled-down strategy should be employed for each tier:

- **Web Servers and Siebel Servers.** To scale down the Web and application server tiers, the individual servers should be maintained in the production configuration and the number of servers scaled down proportionately. The overall performance of a server depends on a number of factors besides the number of CPUs, CPU speed, and memory size. So, it is generally not accurate to try to map the capacity of one server to another even within a single vendor's product line.

  The primary tier of interest from an application scalability perspective is the application server tier. Scalability issues are very rarely found on the Web server tier. If further scale-down is required it is reasonable to maintain a single Web server and continue to scale the application server tier down to a single server. The application server should still be of the same configuration as those used in the production environment, so that the tuning activity can be accurately reflected in the system test and production environments.
- **Database server.** If you want to scale down a database server, there is generally little alternative but to use a system as close as possible to the production architecture, but with CPU, memory, and I/O resources scaled down as appropriate.
- **Network.** The network configuration is one area in which it is particularly difficult to replicate the same topology and performance characteristics that exist in the production environment. It is important that the system test includes any active network devices such as proxy servers and firewalls. The nature of these devices can impact not only the performance of the system, but also the functionality, because in some cases these devices manipulate the content that passes through them. The performance of the network can often be simulated using bandwidth and latency simulation tools, which are generally available from third-party vendors.

4 Design and Develop Tests

This section describes the process of developing the tests that you should perform during the development of your project. It includes the following topics:

- Overview of Test Development
- Design Evaluation on page 36
- Test Case Authoring on page 37
- Test Case Automation on page 42

Overview of Test Development

It is important that you develop test cases in close cooperation between the tester, the business analyst, and the business user. The process illustrated in Figure 9 illustrates some of the activities that should take place in the test development process.

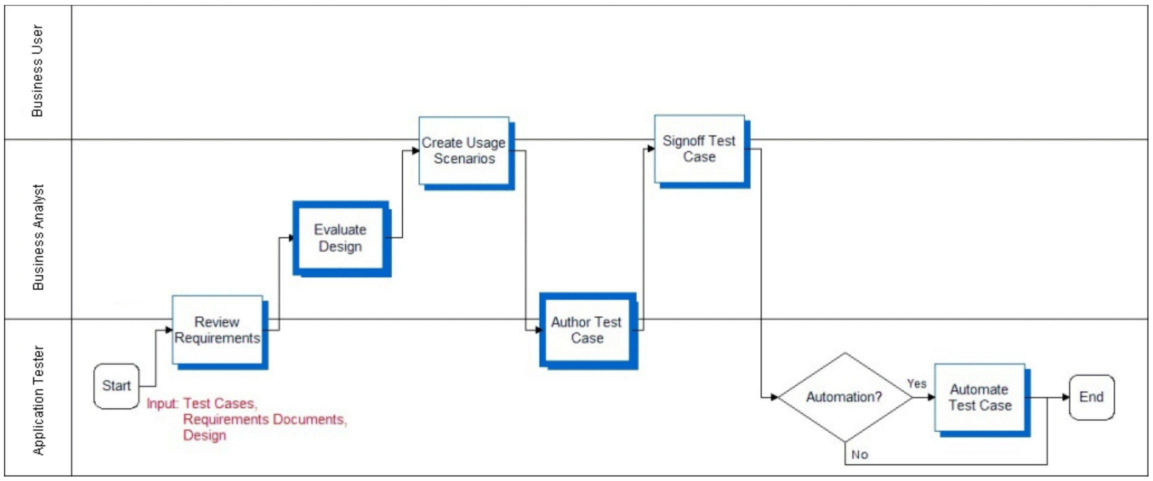


Figure 9. Develop Tests Process

To generate valid and complete test cases, they must be written with full understanding of the requirements, specifications, and usage scenarios.

The deliverables of the test development process include:

- **Requirement gaps.** As a part of the design review process, the business analyst should identify business requirements that have incomplete or missing designs. This can be a simple list of gaps tracked in a spreadsheet. Gaps must be prioritized and critical issues scoped and reflected in the updated design. Lower priority gaps enter the change management process.

- **Approved technical design.** This is an important document that the development team produces (not a testing-specific activity) that outlines the approach to solving a business problem. It should provide detailed process-flow diagrams, UI mock-ups, pseudo-code, and integration dependencies. The technical design should be reviewed by both business analysts and the testing team, and approved by business analysts.
- **Detailed test cases.** Step-by-step instructions for how testers execute a test.
- **Test automation scripts.** If test automation is a part of the testing strategy, the test cases need to be recorded as actions in the automation tool. The testing team develops the functional test automation scripts, while the IT team typically develops the performance test scripts.

Design Evaluation

The earliest form of testing is design evaluation. Testing during this stage of the implementation is often neglected. Development work should not start until requirements are well understood, and the design can fully address the requirements. All stakeholders should be involved in reviewing the design. Both the business analyst and business user, who defined the requirements, should approve the design. The design evaluation process is illustrated in Figure 10.

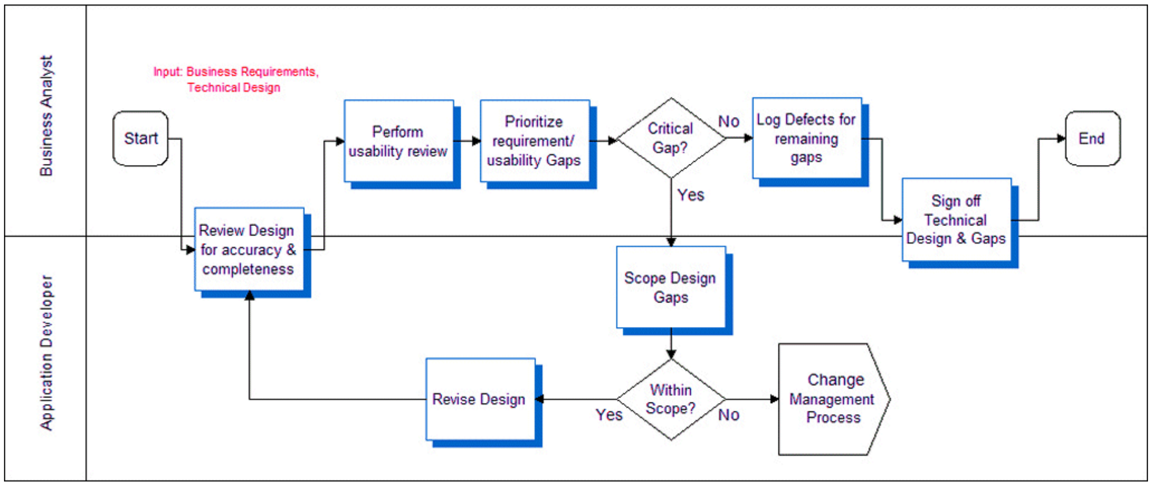

Business Analyst
Application Developer
Input: Business Requirements, Technical Design
Start
Review Design for accuracy & completeness
Perform usability review
Prioritize requirement/ usability Gaps
Critical Gap?
No
Yes
Log Defects for remaining gaps
Sign off Technical Design & Gaps
End
Scope Design Gaps
Within Scope?
Yes
No
Revise Design
Change Management Process

Figure 10. Evaluate Design Process

Reviewing Design and Usability

Two tools for identifying issues or defects are the Design Review and Usability Review. These early stage reviews serve two purposes. First, they provide a way for development to describe the components to the requirement solution. Second, they allow the team to identify missing or incomplete requirements early in the project. Many critical issues are often introduced by incomplete or incorrect design. These reviews can be as formal or informal as deemed appropriate. Many customers have used design documents, white board sessions, and paper-based user interface mock-ups for these reviews.

Once the design is available, the business analyst should review it to make sure that the business objectives can be achieved with the system design. This review identifies functional gaps or inaccuracies. Usability reviews determine design effectiveness with the UI mock-ups, and help identify design inadequacies.

Task-based usability tests are the most effective. In this type of usability testing, the tester gives a user a task to complete (for example, create an activity), and using the user interface prototype or mock-up, the user describes the steps that he or she would perform to complete the task. Let the user continue without any prompting, and then measure the task completion rate. This UI testing approach allows you to quantify the usability of specific UI designs.

The development team is responsible for completing the designs for all business requirements. Having a rigorous design and review process can help avoid costly oversights.

Test Case Authoring

Based on the test case objective, requirements, design, and usage scenarios, the process of authoring test cases can begin. Typically this activity is performed with close cooperation between the testing team and business analysts. Figure 11 illustrates the process for authoring a test case.

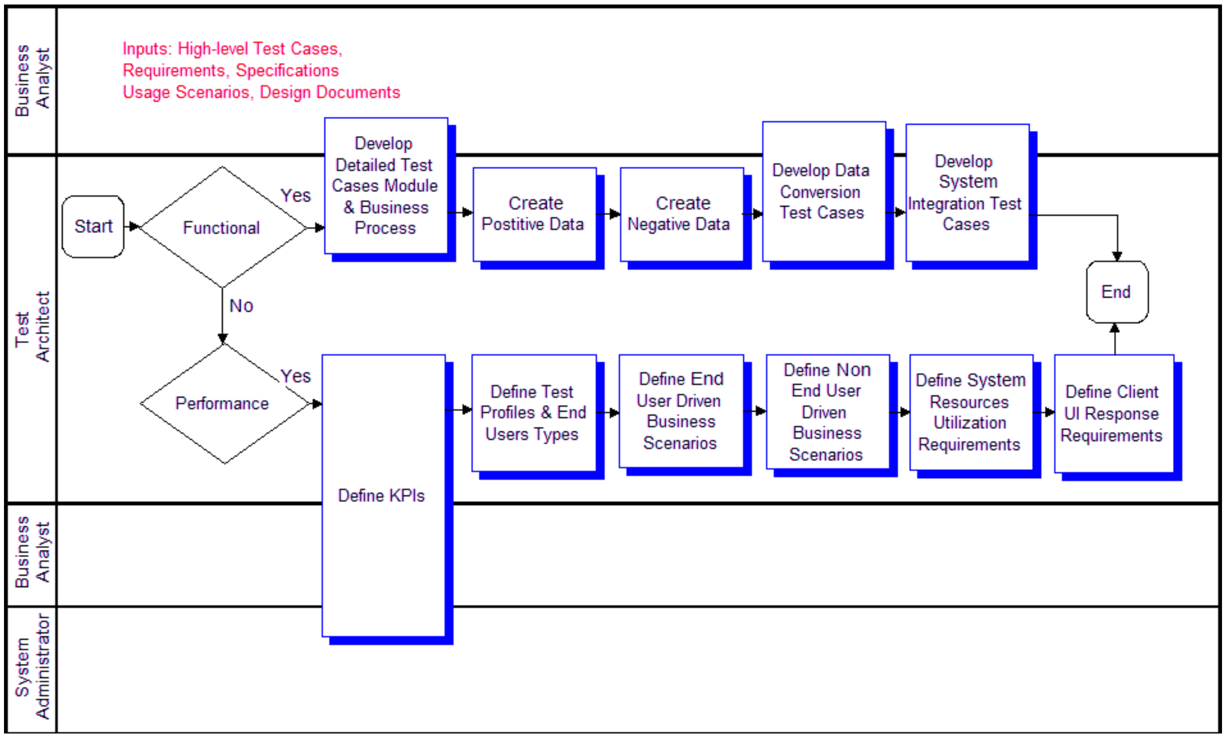

Business Analyst
Inputs: High-level Test Cases, Requirements, Specifications Usage Scenarios, Design Documents
Test Architect
Start
Functional
Yes
No
Develop Detailed Test Cases Module & Business Process
Create Postitive Data
Create Negative Data
Develop Data Conversion Test Cases
Develop System Integration Test Cases
End
Performance
Yes
Define KPIs
Define Test Profiles & End Users Types
Define End User Driven Business Scenarios
Define Non End User Driven Business Scenarios
Define System Resources Utilization Requirements
Define Client UI Response Requirements
Business Analyst
System Administrator

Figure 11. Test Authoring Process

As you can see from the process, functional and performance test cases have different structures based on their nature.

Functional Test Cases

Functional test cases test a common business operation or scenario. Table 2 shows some examples of functional test cases.

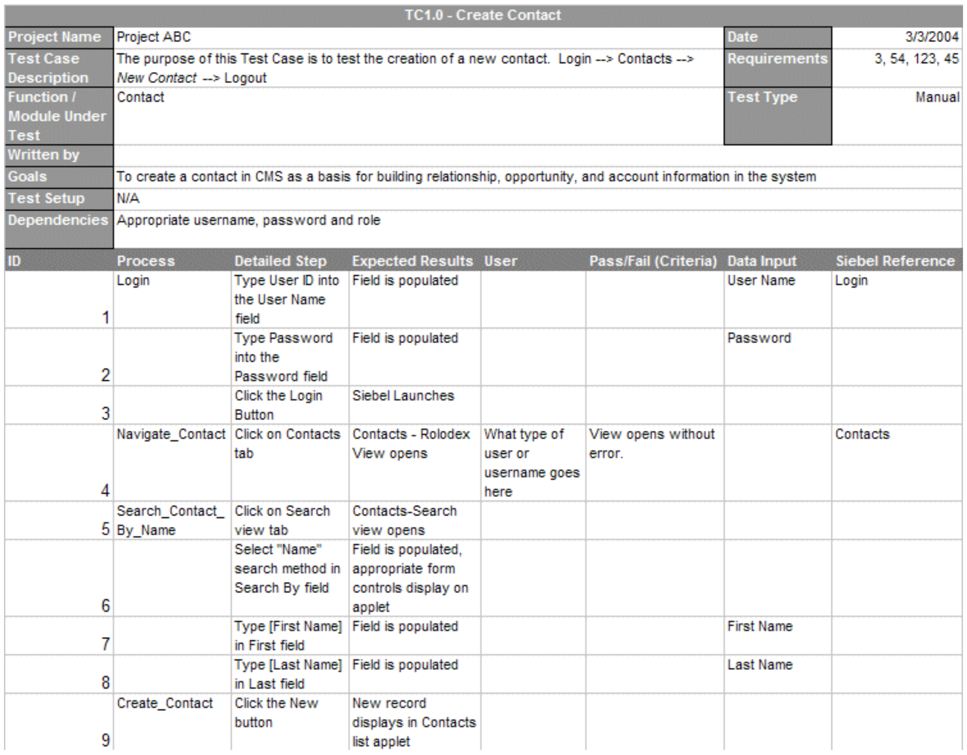
Table 2. Common Functional Test Cases

Test Phase	Example
Unit test	■ Test common control-level navigation through a view. Test any field validation or default logic. ■ Invoke methods on an applet.
Module test	■ Test common module-level user scenarios (for example, create an account and add an activity). ■ Verify correct interaction between two related Siebel components (for example, Workflow Process and Business Service).
Process test	■ Test proper support for a business process.
User interface	■ Verify that a view has all specified applets, and each applet has specified controls with correct type and layout.
Data entity	■ Verify that a data object or control has the specified data fields with correct data types.

A functional test case may verify common control navigation paths through a view. Functional test cases typically have two components, test paths and test data.

Test Case

A test case describes the actions and objects that you want to test. A case is presented as a list of steps and the expected behavior at the completion of a step. Figure 12 shows an example of a test case. Notice that in the Detailed Step column, there are no data values in the step. Instead you see a parameter name in brackets as a place holder. This parameterization approach is a common technique used with automation tools, and is helpful for creating reusable test cases.



TC1.0 - Create Contact

Project Name	Project ABC	Date	3/3/2004
Test Case Description	The purpose of this Test Case is to test the creation of a new contact. Login --> Contacts --> *New Contact* --> Logout	Requirements	3, 54, 123, 45
Function / Module Under Test	Contact	Test Type	Manual
Written by			
Goals	To create a contact in CMS as a basis for building relationship, opportunity, and account information in the system		
Test Setup	N/A		
Dependencies	Appropriate username, password and role		

ID	Process	Detailed Step	Expected Results	User	Pass/Fail (Criteria)	Data Input	Siebel Reference
1	Login	Type User ID into the User Name field	Field is populated			User Name	Login
2		Type Password into the Password field	Field is populated			Password	
3		Click the Login Button	Siebel Launches				
4	Navigate_Contact	Click on Contacts tab	Contacts - Rolodex View opens	What type of user or username goes here	View opens without error.		Contacts
5	Search_Contact_By_Name	Click on Search view tab	Contacts-Search view opens				
6		Select "Name" search method in Search By field	Field is populated, appropriate form controls display on applet				
7		Type [First Name] in First field	Field is populated			First Name	
8		Type [Last Name] in Last field	Field is populated			Last Name	
9	Create_Contact	Click the New button	New record displays in Contacts list applet				

Figure 12. Sample Test Case

Test Data

Frequently, you can use a single path to test many scenarios by simply changing the data that is used. For example, you can test the processing of both high-value and low-value opportunities by changing the opportunity data entered, or you can test the same path on two different language versions of the application. For this reason, it can be helpful to define the test path separately from the test data.

System Test Cases

System test cases are typically used in the system integration test phase to make sure that a component or module is built to specification. Where functional tests focus on validating support for a scenario, system tests make sure that the structure of the application is correct. Table 3 shows some examples of typical system tests.

Table 3. Common System Test Cases

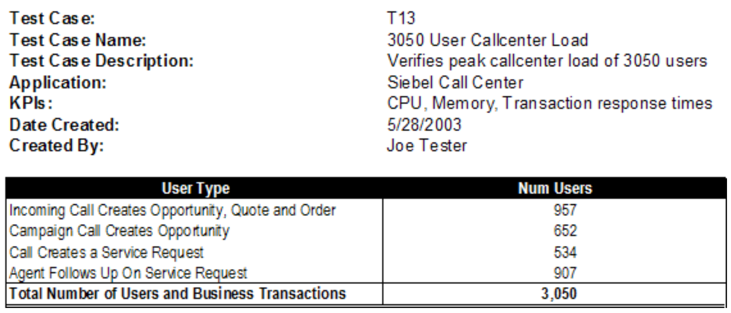
Object Type	Example
Interface	Verify that an interface data structure has the correct data elements and correct data types.
Business Rule	Verify that a business rule (for example, assignment rule) handles all inputs and outputs correctly.

Performance Test Cases

You accomplish performance testing by simulating system activity using automated testing tools. Oracle has several software partners who provide load testing tools that have been validated to integrate with Siebel business applications. Automated load-testing tools are important because they allow you to accurately control the load level, and correlate observed behavior with system tuning. This section describes the process of authoring test cases using an automation framework.

When you are authoring a performance test case, first document the key performance indicators (KPIs) that you want to measure. The KPIs can drive the structure of the performance test and also provide direction for tuning activities. Typical KPIs include resource utilization (CPU, memory) of any server component, uptime, response time, and transaction throughput.

The performance test case describes the types of users and number of users of each type that will be simulated in a performance test. Figure 13 presents a typical test profile for a performance test.



Test Case:	T13
Test Case Name:	3050 User Callcenter Load
Test Case Description:	Verifies peak callcenter load of 3050 users
Application:	Siebel Call Center
KPIs:	CPU, Memory, Transaction response times
Date Created:	5/28/2003
Created By:	Joe Tester

User Type	Num Users
Incoming Call Creates Opportunity, Quote and Order	957
Campaign Call Creates Opportunity	652
Call Creates a Service Request	534
Agent Follows Up On Service Request	907
Total Number of Users and Business Transactions	**3,050**

Figure 13. Performance Test Profile

Test cases should be created to mirror various states of your system usage, including:

- **Response time or throughput.** Simulate the expected typical usage level of the system to measure system performance at a typical load. This allows evaluation against response time and throughput KPIs.
- **Scalability.** Simulate loads at peak times (for example, end of quarter or early morning) to verify system scalability. Scalability (stress test) scenarios allow evaluation of system sizing and scalability KPIs.
- **Reliability.** Determine the duration for which the application can be run without the need to restart or recycle components. Run the system at the expected load level for a long period of time and monitor system failures.

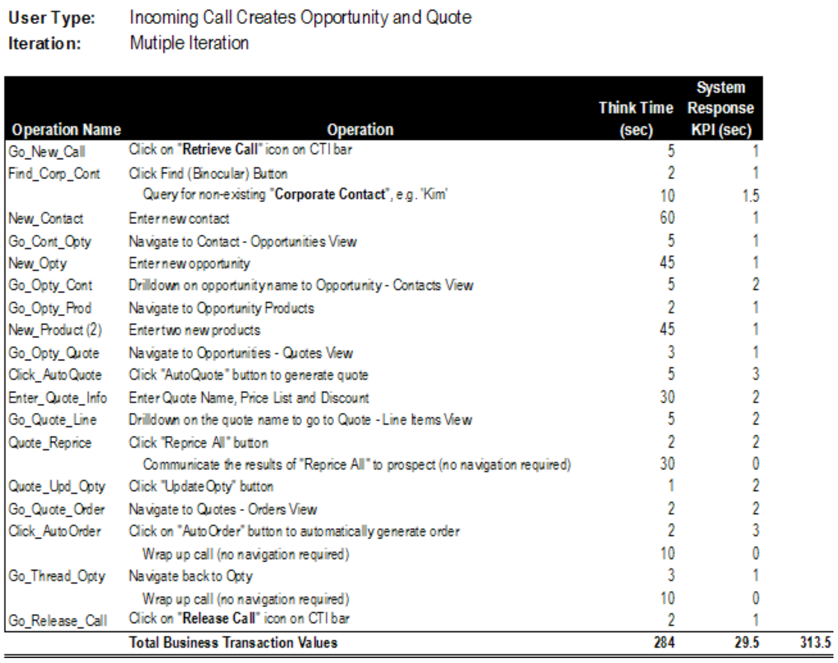
User Scenarios

The user scenario defines the type of user, as well as the actions that the user performs. The first step to authoring performance test cases is to identify the user types that are involved. A user type is a category of typical business user. You arrive at a list of user types by categorizing all users based on the transactions they perform. For example, you may have call center users who respond to service requests, and call center users who make outbound sales calls. For each user type, define a typical scenario. It is important that scenarios accurately reflect the typical set of actions taken by a typical user, because scenarios that are too simple, or too complex skew the test results. There is a trade-off that must be balanced between the effort to create and maintain a complex scenario, and accurately simulating a typical user. Complex scenarios require more time-consuming scripting, while scenarios that are too simple may result in excessive database contention because all the simulated users attempt simultaneous access to the small number of tables that support a few operations.

Most user types fall into one of two usage patterns:

- Multiple-iteration users tend to log in once, and then cycle through a business process multiple times (for example, call center representatives). The Siebel application has a number of optimizations that take advantage of persistent application state during a user session, and it is important to accurately simulate this behavior to obtain representative scalability results. The scenario should show the user logging in, iterating over a set of transactions, and then logging out.

- Single-iteration scenarios emulate the behavior of occasional users such as e-commerce buyers, partners at a partner portal, or employees accessing ERM functions such as employee locator. These users typically execute an operation once and then leave the Siebel environment, and so do not take advantage of the persistent state optimizations for multiple-iteration users. The scenario should show the user logging in, performing a single transaction, and then logging out.



User Type:	Incoming Call Creates Opportunity and Quote
Iteration:	Mutiple Iteration

Operation Name	Operation	Think Time (sec)	System Response KPI (sec)	
Go_New_Call	Click on "**Retrieve Call**" icon on CTI bar	5	1	
Find_Corp_Cont	Click Find (Binocular) Button	2	1	
	Query for non-existing "**Corporate Contact**", e.g. 'Kim'	10	1.5	
New_Contact	Enter new contact	60	1	
Go_Cont_Opty	Navigate to Contact - Opportunities View	5	1	
New_Opty	Enter new opportunity	45	1	
Go_Opty_Cont	Drilldown on opportunity name to Opportunity - Contacts View	5	2	
Go_Opty_Prod	Navigate to Opportunity Products	2	1	
New_Product (2)	Enter two new products	45	1	
Go_Opty_Quote	Navigate to Opportunities - Quotes View	3	1	
Click_AutoQuote	Click "AutoQuote" button to generate quote	5	3	
Enter_Quote_Info	Enter Quote Name, Price List and Discount	30	2	
Go_Quote_Line	Drilldown on the quote name to go to Quote - Line Items View	5	2	
Quote_Reprice	Click "Reprice All" button	2	2	
	Communicate the results of "Reprice All" to prospect (no navigation required)	30	0	
Quote_Upd_Opty	Click "Update Opty" button	1	2	
Go_Quote_Order	Navigate to Quotes - Orders View	2	2	
Click_AutoOrder	Click on "AutoOrder" button to automatically generate order	2	3	
	Wrap up call (no navigation required)	10	0	
Go_Thread_Opty	Navigate back to Opty	3	1	
	Wrap up call (no navigation required)	10	0	
Go_Release_Call	Click on "**Release Call**" icon on CTI bar	2	1	
	Total Business Transaction Values	**284**	**29.5**	**313.5**

Figure 14. Sample Test Case Excerpt with Wait Time

As shown in Figure 14, the user wait times are specified in the scenario. It is important that wait times be distributed throughout the scenario, and reflect the times that an actual user takes to perform the tasks.

Data Sets

The data in the database and used in the performance scenarios can impact test results, because this data impacts the performance of the database. It is important to define the data shape to be similar to what is expected in the production system. Many customers find it easiest to use a snapshot of true production data sets to do this.

Test Case Automation

Oracle partners with the leading test automation tool vendors, who provide validated integrations with Siebel business applications. Automation tools can be a very effective way to execute tests. In the case of performance testing, automation tools are critical to provide controlled, accurate test execution. When you have defined test cases, you can automate them using third-party tools.

Functional Automation

Using automation tools for functional or system testing can cost less than performing manual test execution. You should consider which tests to automate because there is a cost in creating and maintaining functional test scripts. Acceptance regression tests benefit the most from functional test automation technology.

For functional testing, automation provides the greatest benefit when testing relatively stable functionality. Typically, automating a test case takes approximately five to seven times as long as manually executing it once. Therefore, if a test case is not expected to be run more than seven times, the cost of automating it may not be justified.

Performance Automation

Automation is necessary to conduct a successful performance test. Performance testing tools virtualize real users, allowing you to simulate thousands of users. In addition, these virtual users are less expensive, more precise, and more tolerant than actual users. The process of performance testing and tuning is iterative, so it is expected that a test case will be run multiple times to first identify performance issues, and then verify that any tuning changes have corrected observed performance issues.

Performance testing tools virtualize real users by simulating the HTTP requests made by the client for the given scenario. The Siebel Smart Web Client Architecture separates the client-to-server communication into two channels, one for layout and one for data. The protocol for the data channel communication is highly specialized; therefore Oracle has worked closely with leading test tool vendors to provide their support for Siebel business applications. Because the communication protocol is highly specialized and subject to change, it is strongly recommended that you use a validated tool.

At a high level, the process of developing automated test scripts for performance testing has four steps. Please refer to the instructions provided by your selected tool vendor for details:

- **Record scripts for each of the defined user types.** Use the automation tool's recording capability to record the scenario documented in the test case for each user. Keep in mind the multiiteration versus single iteration distinction between user types. Many tools automatically record user wait times. Modify these values, if necessary, to make sure that the recorded values accurately reflect what was defined in the user type scenario.
- **Insert parameterization.** Typically, the recorded script must be modified for parameterization. Parameterization allows you to pass in data values for each running instance of the script. Because each virtual user runs in parallel, this is important for segmenting data and avoiding uniqueness constraint violations.
- **Insert dynamic variables.** Dynamic variables are generated based on data returned in a prior response. Dynamic variables allow your script to intelligently build requests that accurately reflect the server state. For example, if you execute a query, your next request should be based on a record returned in the query result set. Examples of dynamic variables in Siebel business applications include session ids, row ids, and timestamps. All validated load test tool vendors provide details on how dynamic variables can be used in their product.

- **Script verification.** After you have recorded and enhanced your scripts, run each script with a single user to verify that it functions as expected.

Oracle offers testing services that can help you design, build, and execute performance tests if you need assistance.

Best Practice

Using test automation tools can reduce the effort required to execute tests, and allows a project team to achieve greater test coverage. Test Automation is critical for Performance testing, because it provides an accurate way to simulate large numbers of users.

5 Execute Siebel Functional Tests

This section describes the process of executing Siebel functional tests. It includes the following topics:

- Overview of Executing Siebel Functional Tests
- Track Defects Subprocess on page 47

Overview of Executing Siebel Functional Tests

The process of executing Siebel functional tests is designed to provide for delivery of a functionally validated Siebel application into the system environment. For many customers the Siebel application is one component of the overall system, which may include other back-end applications, integration infrastructure, and network infrastructure. Therefore, the objective of the Execute Siebel Functional Tests process is to verify that the Siebel application functions properly before inserting it into the larger system environment. This process is illustrated in Figure 15.

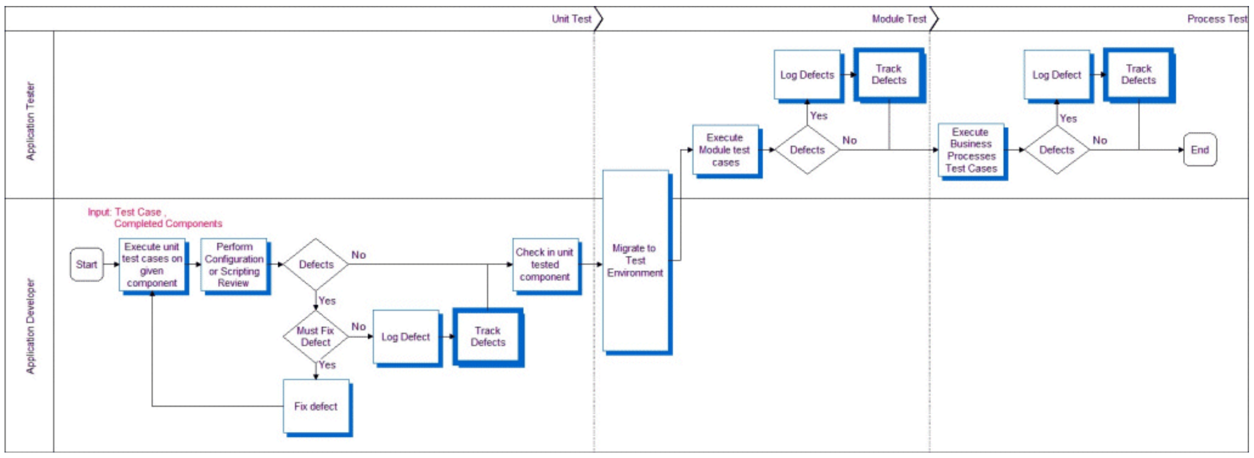

Unit Test
Module Test
Process Test
Application Tester
Application Developer
Input: Test Case , Completed Components
Start
Execute unit test cases on given component
Perform Configuration or Scripting Review
Defects
No
Yes
Must Fix Defect
No
Log Defect
Track Defects
Yes
Fix defect
Check in unit tested component
Migrate to Test Environment
Execute Module test cases
Defects
Yes
No
Log Defects
Track Defects
Execute Business Processes Test Cases
Defects
Yes
No
Log Defect
Track Defects
End

Figure 15. Execute Siebel Functional Tests Process

There are three phases in this process:

- **Unit test.** The unit test validates the functionality of a single component (for example, an applet or a business service).
- **Module test.** The module test validates the functionality of a set of related components that make up a module (for example, Contacts or Activities).

- **Process test.** The process test validates that multiple modules can work together to enable a business process (for example, Opportunity Management or Quote to Order).

Application developers test their individual components for functional correctness and completeness before checking component code into the repository. The unit test cases should have been designed to test the low-level details of the component (for example, control behavior, layout, data handling).

Typical unit tests include structural tests of components, negative tests, boundary tests, and component-level scenarios. The unit test phase allows developers to fast track fixes for obvious defects before checking in. A developer must demonstrate successful completion of all unit test cases before checking in their component. In some cases, unit testing identifies a defect that is not critical for the given component; these defects are logged into the defect tracking system for prioritization.

Once unit testing has been completed on a component, that component is moved into a controlled test environment, where the component can be tested along side others at the module and process level.

Reviews

There are two types of reviews done in conjunction with functional testing, configuration review and scripting code review:

- **Configuration review.** This is a review of the Siebel application configuration using Siebel Tools. Configuration best practices should be followed. Some common recommendations include using optimized, built-in functionalities rather than developing custom scripts, and using primary joins to improve MVG performance.
- **Scripting code review.** Custom scripting is the source of many potential defects. These defects are the result of poor design or inefficient code that can lead to severe performance problems. A code review can identify design flaws and recommend code optimization to improve performance.

Checking in a component allows the testing team to exercise that component along side related components in an integration test environment. Once in this environment, the testing team executes the integration test cases based on the available list of components. Integration tests are typically modeled as actual usage scenarios, which allow testers to validate that a user can perform common tasks. In contrast to unit test cases, these tests are not concerned with specific details of any one component, but rather the way that logic is handled when working across multiple components.

Track Defects Subprocess

The Track Defects subprocess is designed to collect the data required to measure and monitor the quality of the application, and also to control project risk and scope. The process, illustrated in Figure 16, is designed so that those with the best understanding of the customer priorities are in control of defect prioritization. The business analyst monitors a list of newly discovered issues using a defect tracking system like the Siebel Quality module. These users monitor, prioritize, and target defects with regular frequency. This is typically done daily in the early stages of a project, and perhaps several times a day in later stages.

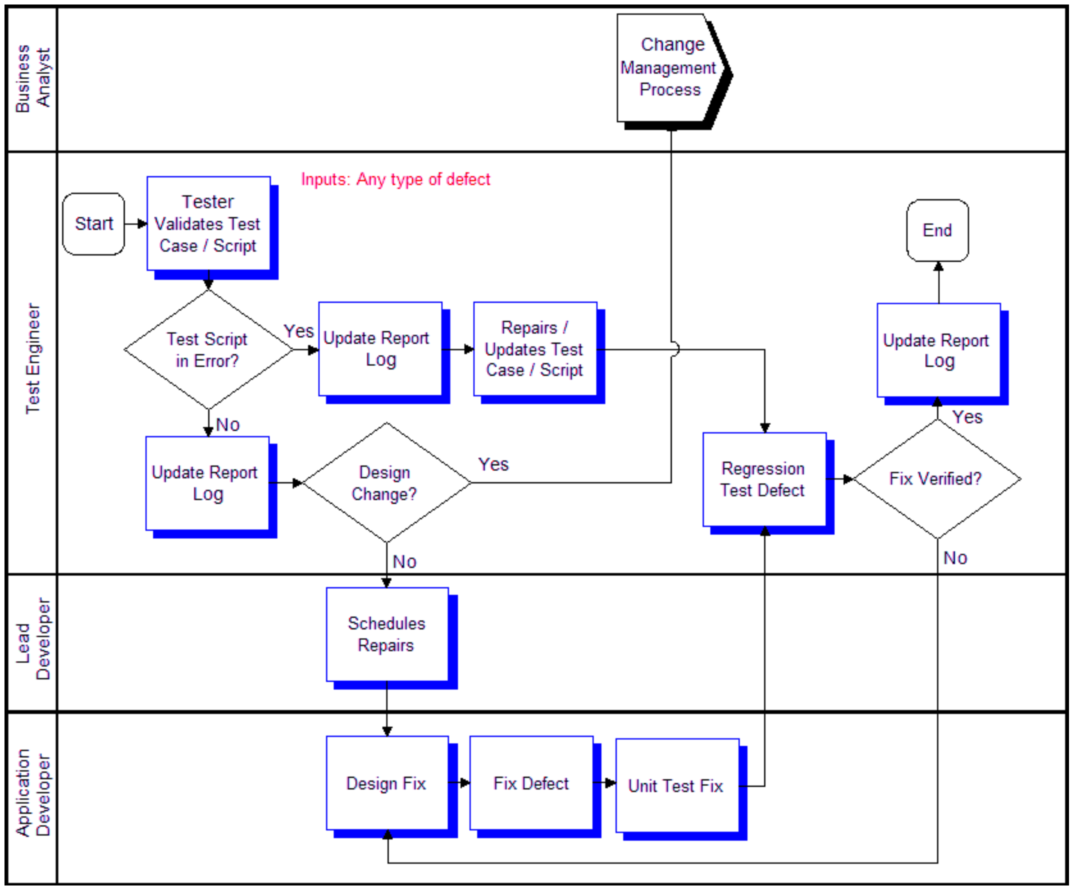


Figure 16. Track Defects Subprocess

The level of scrutiny is escalated for defects discovered after the project freeze date. A very careful measurement of the impact to the business of a defect versus the risk associated with introducing a late change must be made at the project level. Commonly, projects that do not have appropriate levels of change management in place have difficulty reaching a level of system stability adequate for deployment. Each change introduced carries with it some amount of regression risk. Late in a project, it is the responsibility of the entire project team, including the business unit, to carefully manage the amount of change introduced.

Once a defect has been approved to be fixed, it is assigned to development and a fix is designed, implemented, unit tested, and checked in. The testing team must then verify the fix by bringing the affected components back to the same testing phase where the defect was discovered. This requires regression testing (reexecution of test cases from earlier phases). The defect is finally closed and verified when the component or module successfully passes the test cases in which it was discovered. The process of validating a fix can often require the reexecution of past test cases, so this is one activity where automated testing tools can provide significant savings. One best practice is to define regression suites of test cases that allow the team to reexecute a relevant, comprehensive set of test cases when a fix is checked in.

Tracking defects also collects the data required to measure and monitor system quality. Important data inputs to the deployment readiness decision include the number of open defects and defect discovery rate. Also, it is important for the business customer to understand and approve the known open defects prior to system deployment.

Best Practice

The use of a defect tracking system allows a project team to understand the current quality of the application, prioritize defect fixes based on business impact, schedule resources, and carefully control risk associated with configuration changes late in the project.

6 Execute System Integration and Acceptance Tests

This section describes the process of executing integration and acceptance tests. It includes the following topics:

Overview of Executing Integration and Acceptance Tests

The processes of executing integration and acceptance tests are designed to verify that the Siebel application can properly communicate with other applications or components in the system, support end-to-end business processes, and will be accepted by the user community. This is a very busy and exciting phase of any project, because it marks a point where the system is nearing deployment.

The three major pieces involved in executing integration and acceptance tests processes are as follows:

- **Testing integrations with the Siebel application.** In most customer deployments, the Siebel application integrates with several other applications or components. Integration testing focuses on these touch points with third-party applications, network infrastructure, and integration middleware.
- **Functional testing of business processes.** Required business processes must be tested end-to-end to verify that transactions are handled appropriately across component, application, and integration logic. It is important to push a representative set of transaction data through the system and follow all branches of required business processes.
- **Testing system acceptance with users.** User acceptance testing allows system users to use the system to perform simulated work. This phase of testing makes sure that users will be able to use the system effectively once it is live.

Execute Integration Tests

Completion of the Siebel Functional Testing process verifies that the Siebel application functions correctly as a unit. In Integration Testing you verify that this unit functions correctly when inserted into the complete, larger system. In this process, your test cases should be defined to test the integration points between the Siebel application and other applications or components. Typical components include back office applications, integration middleware, network infrastructure components, and security infrastructure. Tests in this process should focus on exercising integration logic, and validating end-to-end business processes that span multiple systems. Figure 17 illustrates this process.

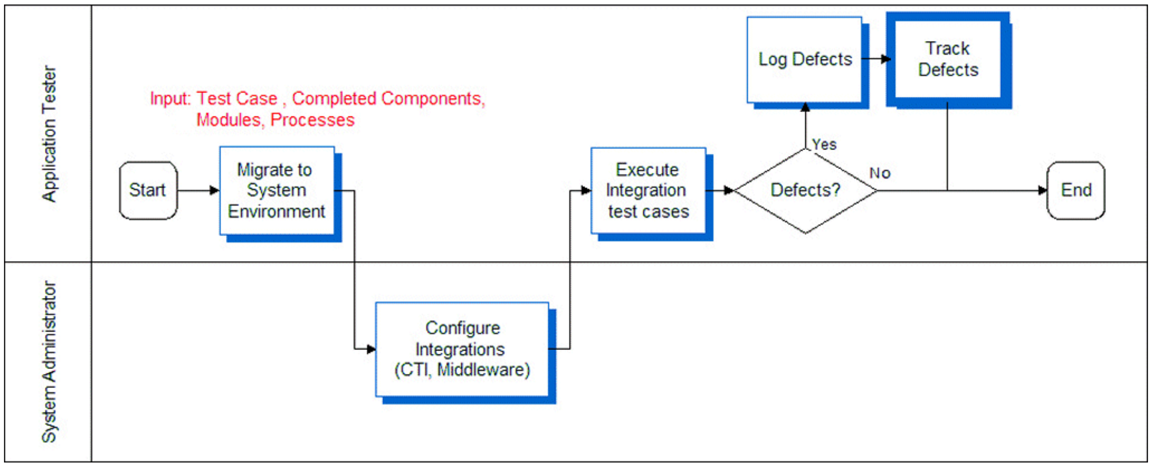

Application Tester
System Administrator
Input: Test Case , Completed Components, Modules, Processes
Start
Migrate to System Environment
Configure Integrations (CTI, Middleware)
Execute Integration test cases
Defects?
Yes
No
Log Defects
Track Defects
End

Figure 17. Execute Integration Tests Process

Execute Acceptance Tests

Once the system as a whole has been validated, you must make sure that the functionality provided is acceptable to the business users. Hopefully, the business user has been engaged all along, approving at each phase of the project to make sure that there are no surprises. In the User Acceptance testing process, open the system up to a larger community of trained users and ask them to simulate running their business on the system. User Acceptance testing should be designed to simulate live business as closely as possible. Complete this process by having the user community representative (business user) approve the acceptance test results. Figure 18 illustrates this process.

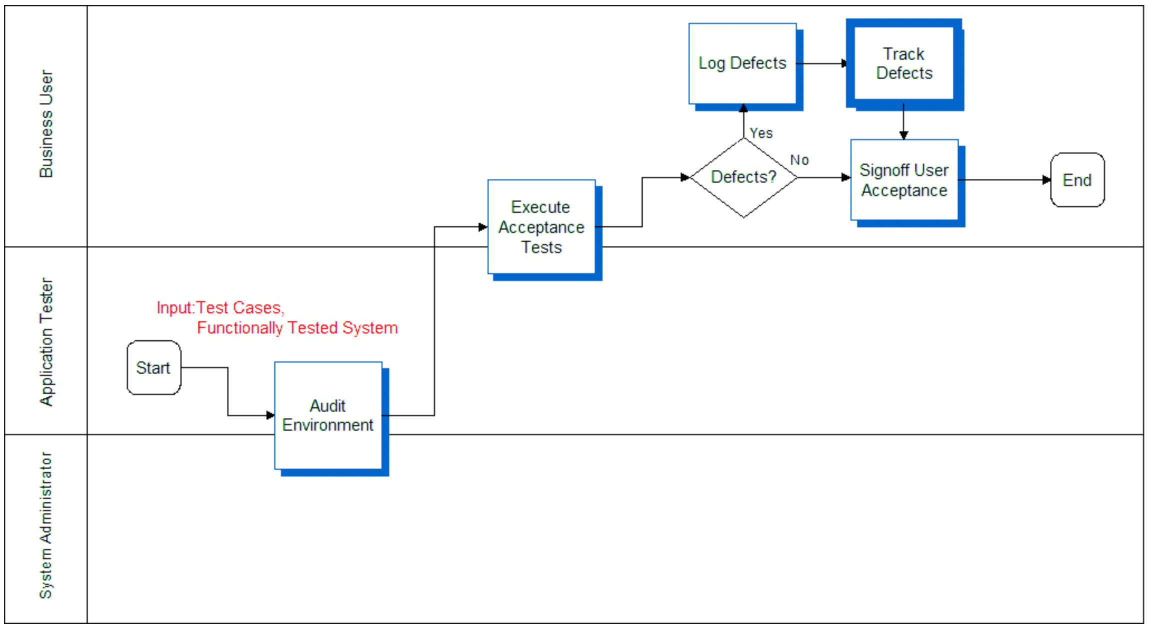


Figure 18. Execute Acceptance Tests Process

7 Execute Performance Tests

This section describes the process of executing performance tests. It includes the following topics:

- Overview of Executing Performance Tests
- Executing Tests on page 54
- Performing an SQL Trace on page 54
- Measuring System Metrics on page 55
- Monitoring Failed Transactions on page 55

Overview of Executing Performance Tests

As described earlier, there are three types of performance test cases that are typically executed: response time, stress, and reliability testing. It is important to differentiate between the three because they are intended to measure different KPIs (key performance indicators). Specialized members of the testing and system administration organizations, who have ownership of the system architecture and infrastructure, typically manage performance tests.

Figure 19 illustrates the process for performance test execution. The first step involves validating recorded user-type scripts in the system test environment.

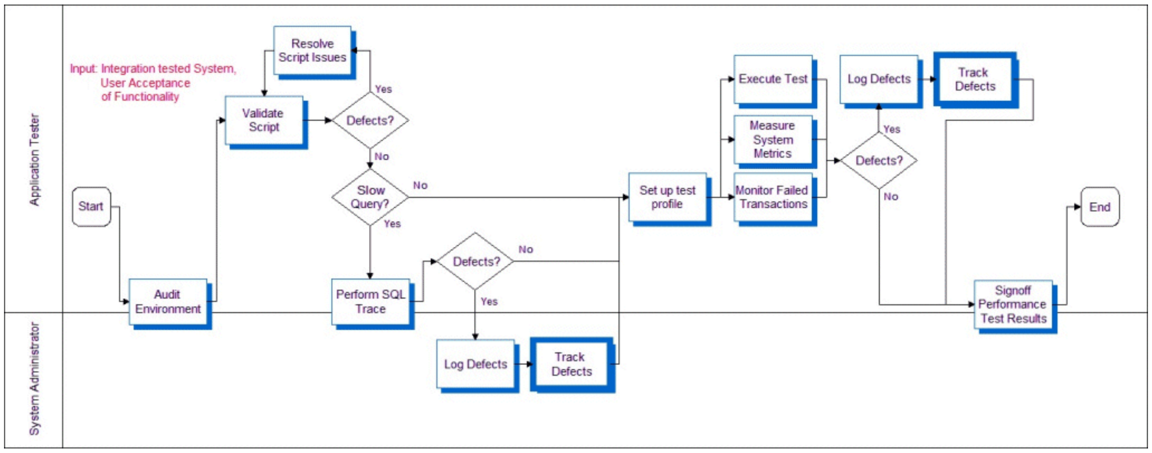


Figure 19. Execute Performance Tests Process

Executing Tests

Execute each script for a single user to validate the health of the environment. A low user-load baseline should be obtained before attempting the target user load. This baseline allows you to measure system scalability by comparing results between the baseline and target loads.

Users must be started at a controlled rate to prevent excessive resource utilization due to large numbers of simultaneous logins. This rate depends on the total configured capacity of the system. For every 1000 users of configured system capacity, you add one user every three seconds. For example, if the system is configured for 5000 users, you add five users every three seconds.

Excessive login rate causes the application server tier to consume 100% CPU, and logins begin to fail. Wait times should be randomized during load testing to prevent inaccuracies due to simulated users executing transactions simultaneously. Randomization ranges should be set based on determining the relative wait times of expert and new users when compared to the average wait times in the script.

Performing an SQL Trace

Because poorly formed SQL or suboptimal database-tuning causes many performance issues, the first step to improve performance is to perform an SQL trace. An SQL trace creates a log file that records the statements generated in the Siebel object manager and executed on the database. The time required to execute and fetch on an SQL statement has a significant impact on both the response time seen by end users of a system, and on the system's resource utilization on the database tier. It is important to discover slow SQL statements and root cause, and fix issues before attempting scalability or load tests, as excessive resource utilization on the database server will invalidate the results of the test or cause it to fail.

To obtain an SQL trace

1. Set a breakpoint in the script at the end of each action and execute the script for two iterations.
2. Enable EvtLogLvl (ObjMgrSqlLog=5) to obtain SQL traces for the component on the application server that has this user session or task running.
3. Continue executing the script for the third iteration and wait for the breakpoint at the end of action.
4. Turn off SQL tracing on the application server (reset it to its original value, or 1).
5. Complete the script execution.

The log file resulting from this procedure has current SQL traces for this business scenario. Typically, any SQL statement longer than 0.1 seconds is considered suspect and must be investigated, either by optimizing the execution of the query (typically by creating an index on the database) or by altering the application to change the query.

Measuring System Metrics

Results collection should occur during a measurement period while the system is at a steady state, simulating ongoing operation in the course of a business day. Steady state is achieved once all users are logged in and caches (including simulated client-side caches) are primed. The measurement interval should commence after the last user has logged in and completed the first iteration of the business scenario.

The measurement interval should last at least one hour, during which system statistics should be collected across all server tiers. We recommend that you measure the following statistics:

- CPU
- Memory
- System calls
- Context switches
- Paging rates
- I/O waits (on the database server)
- Transaction response times as reported by the load testing tool

  NOTE: Response times will be shorter than true end-user response times due to additional processing on the client, which is not included in the measured time.

The analysis of the statistics starts by identifying transactions with unacceptable response times, and then correlating them to observed numbers for CPU, memory, I/O, and so on. This analysis provides insight into the performance bottleneck.

Monitoring Failed Transactions

Less than 1% of transactions should fail during the measurement interval. A failure rate greater than 1% indicates a problem with the scripts or the environment.

Typically, transactions fail for one of the following three reasons:

- **Timeout.** A transaction may fail after waiting for a response for a specified timeout interval. A resource issue at a server tier, or a long-running query or script in the application can cause a timeout.

  If a long-running query or script is applicable to all users of a business scenario, it should be caught in the SQL tracing step. If SQL tracing has been performed, and the problem is only seen during loaded testing, it is often caused by data specific to a particular user or item in the test database. For example, a calendar view might be slow for a particular user because prior load testing might have created thousands of activities for that user on a specific day. This would only show as a slow query and a failed transaction during load testing when that user picks that day as part of their usage scenario.

  Long-running transactions under load can also be caused by consumption of all available resources on some server tier. In this case, transaction response times typically stay reasonable until utilization of the critical resource closely approaches 100%. As utilization approaches 100%, response times begin to increase sharply and transactions start to fail. Most often, this consumption of resources is due to the CPU or memory on the Web server, application server, or database server, I/O bandwidth on the database server, or network bandwidth. Resource utilization across the server tiers should be closely monitored during testing, primarily for data gathering purposes, but also for diagnosing the resource consumption problem.

  Very often, a long-running query or script can cause consumption of all available resources at the database server or application server tier, which then causes response times to increase and transactions to time out. While a timeout problem may initially appear to be resource starvation, it is possible that the root cause of the starvation is a long-running query or script.

- **Data issues.** In the same way that an issue specific to a particular data item may cause a timeout due to a long-running query or script, a data issue may also cause a transaction to fail. For example, a script that randomly picks a quote associated with an opportunity will fail for opportunities that do not have any associated quotes. You must fix data if error rates are significant, but a small number of failures do not generally affect results significantly.

- **Script issues.** Defects in scripts can cause transaction failures. Common pitfalls in script recording include the following:

  - Inability to parse Web server responses due to special characters (quotes, control characters, and so on) embedded in data fields for specific records.

  - Required fields not being parameterized or handled dynamically.

  - Strings in data fields that are interpreted by script error-checking code as indicating a failed transaction. For example, it is common for a technical support database to contain problem descriptions that include the string, The server is down or experiencing problems.

8 Improve and Continue the Testing Process

This section describes the steps you can take to make iterative improvements to the testing process, as illustrated in Figure 20 on page 58. It includes the following topic, Improve and Continue Testing.

Improve and Continue Testing

After the initial deployment, regular configuration changes are delivered in new releases. In addition, Oracle delivers regular maintenance and major software releases. Configuration changes and new software releases must be tested to verify that the quality of the system is sustained. This is a continuous effort using a phased deployment approach, as discussed in "Modular and Iterative Methodology" on page 19.

This ongoing lifecycle of the application is an opportunity for continuous improvement in testing. First, a strategy for testing functionality across the life of the application is built by identifying a regression test suite. This test suite provides an abbreviated set of test cases that can be run with each delivery to identify any regression defects that may be introduced. The use of automation is particularly helpful for executing regression tests. By streamlining the regression test process, organizations are able to incorporate change into their applications at a much lower cost.

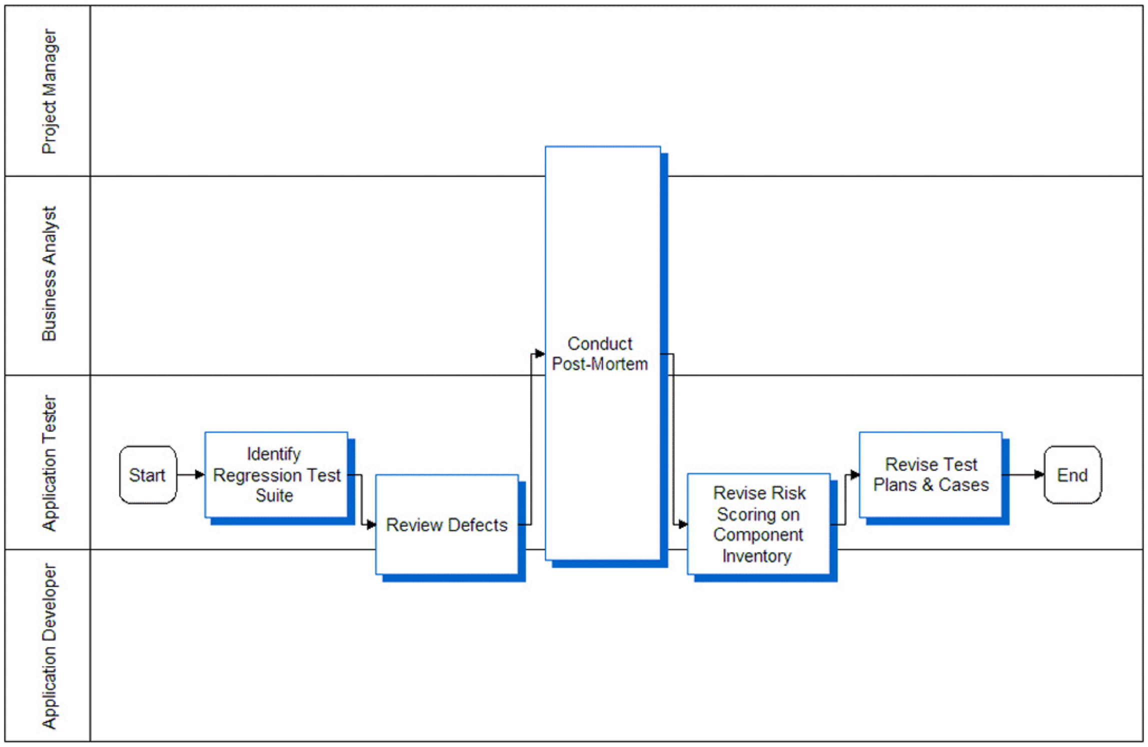


Figure 20. Improve Testing Process

The testing strategy and its objectives should be reviewed to identify any inadequacies in planning. A full review of the logged defects (both open and closed) can help calibrate the risk assessment performed earlier. This review provides an opportunity to measure the observed risk of specific components (for example, which component introduced the largest number of defects). A project-level final review meeting (also called a post-mortem) provides an opportunity to have a discussion about what went well, and what could have gone better with respect to testing. Test plans and test cases should be reviewed to determine their effectiveness. Update test cases to include testing scenarios exposed during testing that were not previously identified.

9 Automating Functional Tests

The topics in this section discuss the concepts and benefits of automating functional testing, and provide a brief explanation of how it works with Siebel Test Automation:

Benefits of Functional Test Automation

Functional testing provides validation of the functional processes of your Siebel application. Without test automation, this can be accomplished by having users log on and manually perform the tasks in a business process. Although this can be effective, it can become quite costly.

By using a test automation tool, you can prepare a test script that exercises the functionality of a particular module or performs the tasks in a comprehensive business process. Then you can share and use this script repeatedly. This approach leads to better application configurations and increased user acceptance, because it allows you to perform additional software testing with little additional cost. Test automation eliminates the need for multiple human test passes, and reduces the risk of human errors in the testing process.

Key Features of Functional Test Tools

Tools for functional test automation (such as QuickTest Pro from Mercury Interactive or e-Tester from Empirix) provide features that allow you create and run functional test scripts. These features include:

- **Test script recording.** Rather than composing a script in a text editor, you start a recorder and perform a series of steps in the application. The test tool records the session in a script, which you can replay later to execute the test.
- **Automated test execution.** Rather than manually performing the tasks in the test script each time, you start the test using a test tool and let it run on its own.
- **Data value parameterization.** You can use variables in your test script that pull in data values from an external table during test execution. This allows you to avoid hard-coding data values into a test script.
- **Reusable tests.** Multiple testers can share tests across multiple cycles of application testing.

- **Result tracking.** The test tool records important statistics that allow you to identify functional errors.

Not all functional test automation tools have all of these features, and some have more. For more information on available features, refer to the documentation for your testing tool.

Architectural Overview of Functional Testing

A Siebel application consists of a set of Web pages accessed through a Web browser, and generated dynamically by the Siebel Web Engine (SWE) on the Siebel server. Each Web page in a Siebel application is constructed with components such as applets, views, and controls. Siebel Test Automation provides objects that run on the client to access corresponding user interface components in the Siebel application. (For more information on the infrastructure of the Siebel Web Engine, see *Configuring Siebel Business Applications*.)

When test automation is activated on the Siebel server and requested in the URL, the SWE generates additional information about each user interface component in the application when constructing the Web page. The test tool uses this information when recording and running test scripts against the application. Figure 21 shows the relationships between the components of the functional test automation architecture during the recording of a test.

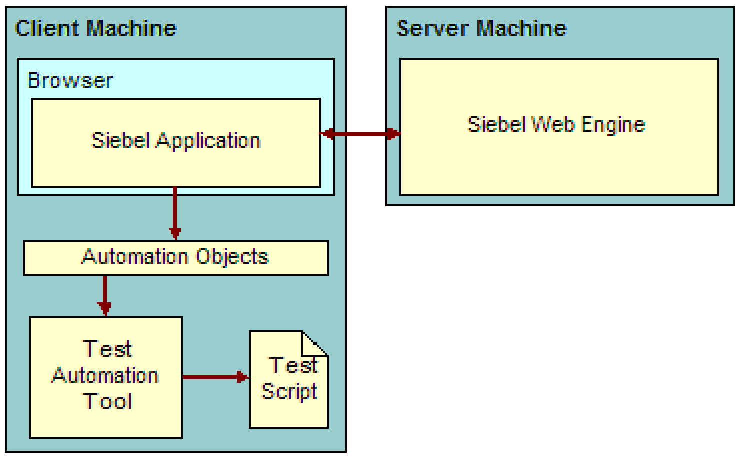


Figure 21. Functional Test Components During Test Recording

Figure 22 shows the relationships between the components of the functional test automation architecture during the replay of a recorded test. For more information on activating test automation, see "Setting Up Your Functional Testing Environment" on page 62.

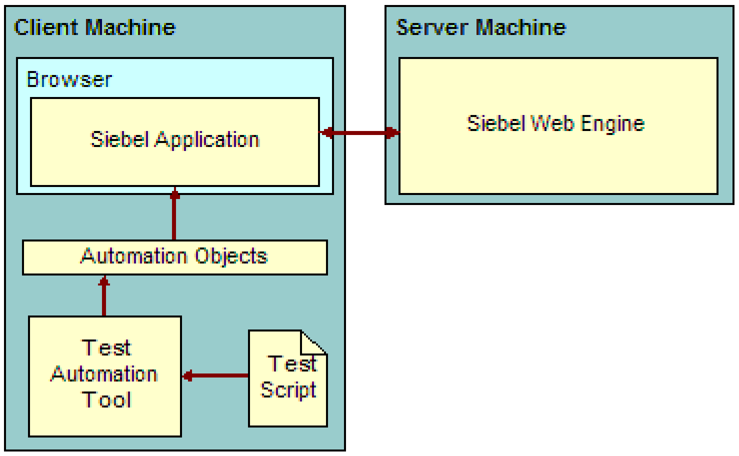


Figure 22. Functional Test Components During Test Replay

Test Object Information in High-Interactivity Applications

High-interactivity Siebel applications use specialized user interface components that run in the Web browser on the client machine. Each component has a specific set of properties, events, and methods that provide functionality for the application. The functional test automation objects for high-interactivity Siebel applications map to these user interface components, and allow you to manipulate the Siebel application from a test script. See "Functional Test Automation Objects for High Interactivity Siebel Applications" on page 81 for a description of each of these test automation objects.

Test Object Information in Standard-Interactivity Applications

Standard-interactivity Siebel applications are rendered in standard HTML that does not provide the rich set of usability features available with the high-interactivity components. Therefore, the test automation information generated by the SWE is quite different.

Each user interface component in a standard-interactivity Siebel application is identified in the Web page by three special attributes that support creating automated test scripts:

- **RN** (Repository Name) indicates the name of the object as it is stored in the repository.
- **UN** (UI Name) indicates the name of the object as it appears in the user interface.
- **OT** (Object Type) indicates the type of object that the HTML element represents (for example, SiebWebTextArea). For a complete list of the object types for standard-interactivity applications, see "Standard Interactivity Functional Test Objects" on page 150.

Setting Up Your Functional Testing Environment

You must perform the following steps to set up your Siebel Test Automation environment:

- Install your functional test tool and any necessary add-ins.

  For more information, see the documentation that accompanies your test tool.

- Activate test automation on your Siebel Server.

  To activate test automation on the server, set the EnableAutomation and AllowAnonUsers configuration parameters to TRUE. For information about setting configuration parameters, see the *Siebel System Administration Guide*.

  NOTE: For the Mobile Web Client, these configuration parameters are in the [InfraUIFramework] section of the application’s .CFG file.

  After you have updated the configuration parameters, you must restart the Siebel Server. This prepares the server for test automation.

- Request test automation information in the URL.

  To perform functional test automation on your Siebel application, you must tell the SWE to generate test automation information using an SWE command. To do so, append the SWECmd=AutoOn token to the URL. For example:

  ```
  http://hostname/callcenter/start.swe?SWECmd=AutoOn
  ```

  This tells the Siebel Web Engine (SWE) to generate test automation information for Siebel applications.

Secure Access to the Siebel Test Automation Framework

The Siebel Test Automation framework allows you to implement secure access to test automation, in which the SWE requires a password to generate test automation information. This is useful for real-time testing in a production environment, and can be integrated with system monitoring tools, such as Topaz from Mercury Interactive.

In addition to the steps described above, you must also perform the following steps to activate secure access to Siebel Test Automation:

1. Define a password on your Siebel Server.

   To define a password for test automation on the server, set the AutoToken configuration parameter to the test automation password. For information about setting configuration parameters, see the *Siebel System Administration Guide*.

   NOTE: For the Mobile Web Client, these configuration parameters are in the [InfraUIFramework] section of the application’s .CFG file.

   The password is case-sensitive and can contain only alphanumeric characters. After you have updated the configuration parameters, you must restart the Siebel Server. This prepares the server for secure access to test automation.

2. Send the password in the URL.

   When you have defined a password on your Siebel Server for secure access to test automation, you must indicate that password in the URL. The URL token is in the format AutoToken=*password*. For example:

   ```
   http://hostname/callcenter/start.swe?SWECmd=AutoOn&AutoToken=mYtestPassword
   ```

   If the AutoToken configuration parameter is defined for the application, and you do not indicate the password in the URL, the SWE does not generate test automation information.

Using Siebel Test Automation for Functional Testing

Siebel Test Automation objects support the automation of functional tests on the following types of Siebel applications:

- Standard Web Client
- Mobile Web Client
- Siebel Developer Web Client

You can use a third-party test automation tool to write, record, and run test scripts on these types of applications. When using a third-party tool, refer to the documentation that accompanies the test tool for information about the specific features and functionality that it provides.

NOTE: The Siebel Test Automation framework does not provide generalized automation services; it is only supported for test automation.

Before attempting to use Siebel Test Automation to automate functional tests, make sure your environment is set up as described in "Setting Up Your Functional Testing Environment" on page 62.

Hand-Scripting Functional Tests

One method of test script development involves using a third-party test automation tool to record the actions that you perform in the Siebel application. However, you can also write scripts by hand (without the use of a recorder).

There are two approaches you can use to do this:

- Add the test automation objects to the object repository. See "Using the Test Optimizer to Populate the Object Repository" on page 64 for more information.
- Add special arguments in the script to call the test automation object directly. See "Hand-Scripting Components That Are Not in the Test Object Repository" on page 64 for more information.

See Appendix A, "Functional Test Object Reference" for more information on object types and repository names. For more tips on preparing test scripts, see "Best Practices for Functional Test Script Development" on page 67.

Using the Test Optimizer to Populate the Object Repository

The Test Optimizer is an API that helps you to use third-party test automation tools to do functional and UI testing. The Test Optimizer API allows automation tools (such as Mercury QuickTest Professional) to query for metadata directly from the Siebel repository. This approach allows you to create scripts offline, without recording against a live client application.

For more information about the Test Optimizer and how to install it and use it to populate the test object repository, see the Technical Note for Test Optimizer on Oracle's Siebel SupportWeb (http://supportweb.siebel.com).

Hand-Scripting Components That Are Not in the Test Object Repository

If you write your test scripts manually and the necessary test automation objects do not exist in the test tool's object repository, you must insert special arguments to call the test automation object directly. For example:

```
SiebText("micclass:=SiebText", "repositoryname:=lastname")
```

The value for micclass is the object type from the automation object model (in this case, “SiebText”), and the value for repositoryname is the name of the object in the Siebel repository. For automation objects that do not have a matching representation in the Siebel repository, use the automation object type for both micclass and repositoryname. For example:

```
SiebPDQ("micclass:=SiebPDQ", "repositoryname:=SiebPDQ")
```

Using these special arguments, you can reference container objects and objects that execute test actions. This allows you to author test scripts without having to record test steps manually.

Best Practices for Functional Test Automation

This section presents best practices that help you prepare and execute automated functional tests. The best practices are organized into the following categories:

- Best Practices for Functional Test Design on page 64
- Best Practices for Functional Test Script Development on page 67
- Best Practices for Functional Test Environment and Execution on page 70

Best Practices for Functional Test Design

The following best practices are provided to assist in your design of functional tests:

- Review Test Plans to Identify Candidates for Automation on page 65
- Define a Common Structure and Templates for Creating Tests on page 65
- Define Test Script Naming Conventions on page 65

- Design Flexible Scripts with Defined Purpose on page 65
- Design Modular Scripts on page 66
- Design Reusable Scripts on page 66
- Design Multilingual Tests for Internationalized Applications on page 66
- Make Test Scripts Independent of the Operating Environment on page 67
- Make Test Scripts Independent of Test Data on page 67

Review Test Plans to Identify Candidates for Automation

Use predefined criteria (like the following list) to identify test plans that are candidates for automation:

- Can you automate the entire test plan using preconfigured automation functionality?
- Do you need to rearrange the test plan to better suit automation?
- Can you reuse existing scripts, or modify scripts from sample libraries?

Define a Common Structure and Templates for Creating Tests

Before testing begins, define templates, standards, and naming conventions for test plan documents and automation scripts. This will make it easier in the long-run to correlate test plans to test scripts, to follow the logic of test steps, and to maintain test instructions.

Define Test Script Naming Conventions

When creating a standard template for test plan documents and test scripts, define naming conventions that reflect the test procedures. Make sure that the names are meaningful and illustrate the logical operations being performed. The use of naming conventions makes it easier to identify scripts, read the script logic, and understand how scripts are organized during the maintenance phase.

For script modules, names should be logically expressive of the purpose of the module and test steps. Additionally, the name can be correlated to the title of the corresponding test plan document. For script variables, follow standard naming guidelines. For example, use the g_ prefix for global variables and the str_ prefix for strings.

Design Flexible Scripts with Defined Purpose

Define the purpose and summarize the validation conditions for each test script in advance. The intent of the test is a key input into script design that determines how much flexibility you need to write into the script.

For a test that is intended to validate a business process, create a script that maps closely to the process, and divide the script into modules that correspond to process steps. This allows you to test each pass through the business process with minimal branching logic, and to modify the order of the script if the business process changes.

Additionally, you might need to add fail-safe logic to the script that ignores minor discrepancies in the UI configuration and proceeds with executing the business process. Consider, for example, a script that sets a value in a particular control as part of a business process. Your test should validate that the control is present and that it functions as expected; but it need not validate the font size and color of the control's text. You might also need to add logic so that the script proceeds even when the control is not present in the UI.

For a test that is intended to validate specific attributes of a UI component (rather than a business process), create a script that checks UI properties explicitly. Avoid adding fail-safe logic to this type of test, because the script should detect when the UI has changed and fail accordingly.

Design Modular Scripts

A module is an independent reusable test component comprised of inter-related entities. Conceptual script modules may be defined by native functionality of a particular test tool, or by a script developer who writes reusable test routines. Examples of modules include a login or query, or the creation of a new Contact or Service Request.

Each automation script should consist of small logical modules rather than having one continuous unstructured sequence of test steps. This approach makes scripts easier to maintain in the long-run, and allows rapid creation of new scripts from a library of proven modules. Other script developers can use well-designed modules with minimal or no adaptations. Modules also can increase the independence of a script from the test environment.

You can categorize modules broadly as RequiredToPass, Critical, and Test Case. Typical RequiredToPass modules are setup scripts and scripts that launch the application. Failure of RequiredToPass modules should result in the entire script being aborted. Critical modules include navigating to the client area, inserting test data, and so on. Failure of Critical modules may affect certain test cases. Test Case modules are where you perform the test cases. The test cases should be able to fail without affecting the execution of the whole script. You should be able to rerecord or insert a test case without changing other modules in the script. In addition, Test Case modules can have a specified number of iterations. For example, a test case module to add a new account can be constructed to execute an indefinite number of times without changing any script.

Module interdependency should be clearly established. Execution status for RequiredToPass and Critical modules should be stored in global variables or environment variables so that other modules can use this information for purposes such as error handling. Some test tools will skip an entire module when a failure occurs.

Design Reusable Scripts

Reusability is necessary for building a library of test cases that can be shared between developers and reused for subsequent test cycles. You can improve reusability using a variety of strategies including script modularization, parameterization, and external definition of global functions and constants.

Design Multilingual Tests for Internationalized Applications

Parameterize all hard-coded data contained in internationalized test scripts, and then create script logic to switch the data table at runtime based on the current language setting. You can obtain the language setting for the Siebel application from a suffix on the URL (for example, callcenter_enu).

Parameterization is especially important for picklist values, dates, currencies, and numbers because their formats differ across languages. As a general rule, parameterize all test data and references to configurable UI components.

NOTE: The column names and structure of the external data table must be consistent across all languages for the script to access the data successfully.

Use regular expressions to perform basic pattern matching and greater than and less than comparisons. This is especially important for inserting data validation conditions into the script. Do not directly compare string representations of picklist values, dates, currencies, and numbers.

Scripts that perform only navigation without getting or setting data do not need to be modified to run on multiple languages.

Make Test Scripts Independent of the Operating Environment

Develop and use strategies to create environment-independent test scripts. Design your test scripts so that they are capable of running on disparate hardware configurations, operating systems, language locales, database platforms, and browser versions.

Make Test Scripts Independent of Test Data

When authoring a test script, do not leave any hard-coded data values in the script. Instead, replace the hard-coded data with variables that reference an external data source. This procedure is generally called parameterization. Parameterizing your test scripts makes them independent of the data structure of the application being tested. Without parameterization, scripts can stop running due to database conflicts and dependencies.

Parameterization also allows you to switch data tables dynamically, if necessary. Store the data used by test scripts in external data tables. Then use the data table import function within the test script to import data. This feature can be useful for multilingual testing, allowing you to switch the data table based on the current language environment.

NOTE: The column names and structure of the external data table must match the variable names used in the script.

Best Practices for Functional Test Script Development

The following best practices are provided to assist in your development of functional test scripts:

- Comment Your Test Scripts on page 68
- Scope Variables and Data Tables for Script Modules on page 68
- Include Multiple Validation Conditions in the Script on page 68
- Handle Error Conditions on page 69
- Define Data Values with Structured Format on page 69
- Use Variables and Expressions When Working with Calculated Fields on page 69
- Import Generalized Functions and Subroutines on page 69

- Run a Query Before Adding a Record or Accessing Data on page 69
- Run a Query Before Accessing Application Data on page 70
- Manage Test Data from Within the Test Script on page 70
- Remove New Records Created During a Test on page 70
- Exercise UI Components with Basic Mouse Clicks on page 70

Comment Your Test Scripts

Use commented text in your scripts to describe the test steps. A test script that is thoroughly annotated is much easier to maintain. When sharing test scripts among a large team of testers and developers, it is often helpful to document conventions and guidelines for providing comments in test scripts.

Scope Variables and Data Tables for Script Modules

Make sure to scope script variables and data tables appropriately. The scope, either modular or global, determines when a variable or data table is available. By scoping them appropriately, you can be sure that the global variables are available across multiple scripts, and modular variables maintain their state within a specific script module.

The test tool typically allows you to store data values in tables and variables, but often these tables and variables have a scope that is defined by the test tool. You might need to override the scoping that is predefined by the test tool. Identify when variables are used within the script, and construct your test so that variables and data values are available when needed.

Include Multiple Validation Conditions in the Script

Use verification procedures to perform the function of the visual monitoring that the tester does during manual testing:

- **Checkpoints.** Include object checkpoints to verify that properties of an object are correct. Properties that can be included in checkpoints are described in Appendix A, “Functional Test Object Reference.”
- **Verification routines.** Include routines that verify that the actions requested were performed, expected values were displayed, and known states were reached. Make sure that these routines have appropriate comments and log tracing.
- **Negative and boundary testing.** Include routines that perform negative tests. For example, attempt to insert and commit invalid characters and numbers into a date field that should only accept date values. Also include routines that perform boundary testing in fields that should accept only a specific range of values. Test values above, below, and within the specified limits. For example, attempt to insert and commit the values of 0, 11, and 999 in a field that should accept values only between 1 and 31. Boundary tests also include field length testing. For example a field that accepts 10 characters should be tested with string lengths that are greater than, less than, and equal to 10.

Handle Error Conditions

For critical scripts that validate key application functionality, insert validation conditions and error event handling to decide whether to proceed with the script or abort the script. If error events are not available in the test tool, you can write script logic to address error conditions. Set a global or environment variable on or off at the end of the script module, and then use a separate script module to check the variable before proceeding. Construct your test scripts so that for every significant defect in the product, only one test will fail. This is commonly called the one bug, one fail approach.

When an error condition is encountered, the script should report errors to the test tool's error logging system. When a script aborts, the error routine should clean up test data in the application before exiting.

Construct your test scripts so that individual test modules use global setup modules to initialize the testing environment. This allows you to design tests that can restart the application being tested and continue with script execution (for example, in the event of a crash).

Define Data Values with Structured Format

Some fields in Siebel applications require data values that have a defined format. Therefore, you must use data values that are formatted as the fields are configured in the Siebel application. For example, a date field that requires a value in the format 4/28/2003 02:00:00 PM causes an error if the data value supplied by the test script is 28 Apr 2003 2:00 PM. Test automation checkpoints should also use data that has been formatted correctly, or use regular expressions to do pattern matching.

Use Variables and Expressions When Working with Calculated Fields

Some fields in Siebel applications are calculated automatically and are not directly modifiable by the user (for example, Today's Date). Construct your test scripts so that they remember calculated values in a local variable, or in an output value if the calculated value needs to be used later in the script. For example, you might need to use a calculated value to run a Siebel query.

When you set a checkpoint for a calculated value, you might not know the value ahead of time. Use a regular expression in your checkpoint such as an asterisk (*) to verify that the field is not blank. When you are using a tabular checkpoint, you might want to omit the calculated field from the checkpoint.

Import Generalized Functions and Subroutines

Store generalized script functions and routines in a separate file. This allows you to maintain these pieces of script separately from specialized test code. In your test tool, use the import functionality (if available) to access the generalized scripts stored in the external file.

NOTE: When developing and debugging generalized functions, keep them in the specialized test script until they are ready to be extracted. This is because you might not be able to debug external files due to test tool limitations.

Run a Query Before Adding a Record or Accessing Data

Before creating data that could potentially cause a conflict, run a query to verify that no record with the same information already exists. If a matching record is found, the script should delete it, rename it, or otherwise modify the record to mitigate the conflict condition.

Run a Query Before Accessing Application Data

Before accessing an existing record or record set, run a query to narrow the records that are available. Do not assume that the desired record is in the same place in a list applet because the test database can change over time.

You also should query for data ahead of time when you are in the process of developing test scripts. Check for data that should not be in the database, or that was left in the database by a previous test pass, and delete it before proceeding.

Manage Test Data from Within the Test Script

Create all test data necessary for running a script within the script itself. Avoid creating scripts that are dependent on preexisting data in a shared test database. Manage test data using setup scripts and script data tables, rather than database snapshots.

Remove New Records Created During a Test

Remove all records created by the test at the end of the script. This should be done at the beginning also, in case a previous test failed to complete the clean-up process.

You can implement the clean-up process as a reusable script module. For each module in the test, you can create a corresponding clean-up module and run it before and after the test module.

The general approach is to have the clean-up script perform a query for the records in a list applet and iterate through them until all of the associated test records are deleted or renamed. When the records need to be renamed, the initial query should be repeated after each record is renamed, until the row count is 0.

Exercise UI Components with Basic Mouse Clicks

When recording a test script, perform all actions using the visual components as if you were a beginning user. This requires clicking on the UI components rather than using keyboard accelerators and other shortcuts.

Most shortcuts in Siebel applications are supported for test automation. However, the Tab key shortcut is not supported—pressing the Tab key typically moves the focus from one control to another based on a preconfigured tab order. Click the mouse to move focus rather than using the Tab key.

Best Practices for Functional Test Environment and Execution

The following best practices are provided to assist you executing functional test scripts:

- Test with One Browser Window and One Siebel Application at a Time on page 71
- Avoid Navigating Between Web Applications During a Test on page 71
- Use Test Tool APIs to Aggregate Result Reports on page 71
- Launch Siebel Developer and Mobile Web Client Applications for Testing from the Command Line on page 71

Test with One Browser Window and One Siebel Application at a Time

When developing and executing tests, make sure there is only a single browser window open (the one that contains the Siebel application).

A test script may click on a link within the Siebel application to open a separate browser under certain circumstances, but that additional browser must be a pure standard-interactivity Siebel application or a nonSiebel application.

NOTE: Some test management tools require you to have two browsers open at a time. The first browser runs the test management tool, and the second is for developing and executing tests.

Avoid Navigating Between Web Applications During a Test

Avoid switching to another Siebel application or to any other application from the browser address bar during a test. Switching between applications in the same browser window can work in some instances, but it is not the recommended approach.

When testing high-interactivity applications, you can test only one Siebel application instance at a time. Do not use the browser address bar to select another page from the history list during a test. Navigating using the browser history list is not supported.

Use Test Tool APIs to Aggregate Result Reports

Some test tools provide a programming interface that allows you to aggregate the results of multiple test passes. If such an API exists for your test tool, use it to aggregate the results of several test passes into a single file. Otherwise, you must analyze individual test results manually, which can be a cumbersome process for large sets of tests that run in unattended mode.

Launch Siebel Developer and Mobile Web Client Applications for Testing from the Command Line

When recording test scripts on Mobile Web Client and Siebel Developer Web Client applications, you can launch the Siebel application using the following methods:

- Start recording and enter the full command line in the Windows Run dialog box (available from the Windows Start menu). The test tool records this operation as a native line of script.
- Start recording and launch the Mobile Web Client from an existing Start menu shortcut. This operation is recorded by the test tool as a native line of script.
- Configure the test tool to launch the Mobile Web Client when you begin recording a script. You must save the full command line in a persistent setting in the test tool to launch the Mobile Web Client.

NOTE: Make sure the command line contains the /u and /p switches to log in with a username and password when the application launches. You cannot record or replay the login page using the Mobile Web Client or the Siebel Developer Web Client.

For the Mobile Web Client and Siebel Developer Web Client, you do not need to include any special switches (such as, SWECmd=AutoOn and AutoToken=*password*) in the URL, because you are launching the application from a command line rather than a URL. However, you must update the .cfg file for the application that you plan to test.

10 Automating Load Tests

The topics in this section discuss the concepts and benefits of automating load testing, and provides a brief explanation of how it works with Siebel Test Automation:

Benefits of Load Test Automation

Load testing measures key performance indicators (KPIs), such as response time and reliability, of your Siebel application while it is under a load of multiple users. Without load test automation, you can accomplish this by having many users log on, and run through a business process during a specified period of time. Although this can be effective, it can become quite costly.

By using a test automation tool, you can simulate the load and achieve more comprehensive and precise results without the cost associated with multiple human testers. In addition, a load testing automation tool reduces the risk associated with user errors.

Key features of Load Test Tools

Tools for load test automation (such as LoadRunner from Mercury Interactive or e-Load from Empirix) provide features that allow you to simulate a multiple-user environment. These features include:

- **Test script recording.** Rather than composing a script in a text editor, you start a recorder and perform a series of steps in the application. The test tool records the session, which you can use to perform the test.
- **Virtual users.** A single test script can be executed simultaneously by many virtual users from a single machine.
- **Controlled test execution.** The test tool allows you to specify parameters that indicate how the load test is run, such as how quickly to add users, how many users to add, and which scripts to run.
- **Result tracking.** The test tool records important statistics, such as memory usage and response times, that allow you to track KPIs and isolate bottlenecks.

Not all load testing tools have all of these features, and some have more. For more information, refer to the documentation for your load testing tool.

Architectural Overview of Load Testing

The Siebel Correlation Library allows you to use a third-party tool to automate load testing on Siebel applications. This is a dynamically linked library (DLL) that provides services necessary that help the testing tool generate and execute test scripts against your Siebel application.

When the Correlation Library is installed and your test environment is properly set up, the Correlation Library helps the test tool translate the recorded test session into a script that can be executed to test your Siebel application. Then during test execution, the Correlation Library provides similar services as described below. (For more information on the environment set up, see "Setting Up Your Load Testing Environment" on page 75.)

The Correlation Library essentially acts as an interpreter for the test tool. The test tool records the HTTP traffic from the Siebel application. (This HTTP traffic equates to a Web page.) Then the test tool sends the Web page to the Correlation Library. The Correlation Library parses the Web page and returns to the test tool the appropriate correlation information for the Web page. The correlation information provides for parameterization of data values.

During recording, the correlation information consists of the following and is sent for each record value:

- The data value of the field
- The associated row ID
- A unique name
- The data type of the field (which can be bool, currency, datetime, date, id, integer, mltext, number, note, phone, text, time, and utcdatetime)
- The input name of the associated control (spanning prefix)
- The display name of the associated control (as displayed in the user interface)

During playback, the test tool needs only enough information to execute the test. Therefore, the Correlation Library sends only a subset of the correlation information to save memory and CPU on the load test machine. The information sent by the Correlation Library during playback consists of the following:

- The data value of the field
- The associated row ID

Parameterizing Transaction Data

In addition to the application data entities that are parameterized by the Correlation Library, you must also parameterize the transaction data entities. Application data entities are those that are associated with the function of the application, such as Row ID, SWE Timestamp, and SWE Count. Transaction data entities are those that contain record data entered into the application by users, such as Contact Name and Account Name. Transaction data entities must be parameterized manually, because they are not handled by the Correlation Library.

Setting Up Your Load Testing Environment

To set up your environment for automated load testing using Siebel Test Automation, you must perform the following steps:

- Install your load testing tool.
- Copy the Correlation Library (ssdtcorr.dll) to a location that is accessible to the load testing tool (for example, the BIN directory of the load testing tool). You can find the ssdtcorr.dll file in your Siebel Server installation (siebsrvr\bin on Windows, siebsrvr/lib on UNIX).

In addition, there are steps that you must perform to set up your test tool so that it will communicate properly with the Correlation Library. For more information, refer to the documentation from your test tool vendor.

Best Practices for Load Testing

This section outlines some best practices that you should consider when developing and executing load tests on your Siebel applications. The following best practices are provided:

- Preserve Environment During Recording and Running Load Tests on page 75
- Preserve List States During Recording and Running Load Tests on page 76
- Make Scenarios and Transaction Definitions Granular on page 76
- Eliminate Message Bar Traffic from Transactions on page 76
- Preserve a Base State for Iterative Actions on page 76
- Make Iterative Actions Self-Sufficient on page 76
- Reset the SWE Count for Iterative Actions on page 76
- Avoid Concurrency Errors in Update Operations on page 76
- Stabilize Response Time Before Terminating Test on page 76
- Parameterize User Key Fields, Dates, and Time Zones on page 77
- Make Data Value Parameters User-Specific on page 77
- Make Sure User Role and Position Are Compatible on page 77

In addition to these tips, you should also read and understand Chapter 7, "Execute Performance Tests."

Preserve Environment During Recording and Running Load Tests

Make sure that the environment when running a load test is in the same state as it was when the test was recorded. Changes to the operating environment might require tests to be rerecorded.

Preserve List States During Recording and Running Load Tests

A Siebel application that contains a list applet can return one or more records in response to a query. It can also return zero records. Both of these can be expected outcomes. When conducting load tests on a Siebel application that contains a list applet, make sure that the expected state of the list (either with records or without records) is the same during the recording session as it will be during the execution of the test. Failure to do so will result in a test error. If necessary, you can add a record before recording the test.

Make Scenarios and Transaction Definitions Granular

Where possible, break scenarios into several smaller scenarios to focus the tests. Make sure transaction definitions are granular enough to be able to pinpoint performance issues to specific GUI actions.

Eliminate Message Bar Traffic from Transactions

Eliminate HTTP traffic related to the message bar from transactions. Put message bar requests into your script outside of transaction blocks. Put one message bar request into your test script after every 120 seconds of wait time.

Preserve a Base State for Iterative Actions

When setting up repeatable or iterative actions, make sure to leave the application in a base state from where the next iteration can pick up and complete successfully. Home page is a good example of such a base state.

It is also important to maintain the number of rows displayed in a list applet, or the number of controls displayed in a form applet. If Show More is clicked during the test, then Show Less must also be clicked before the end of the test.

Make Iterative Actions Self-Sufficient

An iterative action has to be self-sufficient in that all the correlations have to originate and end within the action. Values for correlated variables should not come from a previous action (which may or may not be iterative).

Reset the SWE Count for Iterative Actions

At the beginning of each iterative action, make sure that you reset the SWE Count to the SWEC value of the first request in the iteration.

Avoid Concurrency Errors in Update Operations

In tests that involve update operations, make sure you are using different virtual users or make sure the same virtual user accesses different records to avoid concurrency errors.

Stabilize Response Time Before Terminating Test

When running a load test, wait for the response time to stabilize before stopping the load test.

Parameterize User Key Fields, Dates, and Time Zones

Make sure that user key fields are parameterized. Parameterize dates, especially variable dates (such as today's date or tomorrow's date). Parameterize time zones. If you do not parameterize time zone values, your script will fail for virtual users who are not set up to use the time zone in which the test was recorded.

Make Data Value Parameters User-Specific

Set up your data value parameters to be specific for each virtual user (for example, User1 logs in and searches for Contact1 while User2 logs in and searches for Contact2). Each virtual user should have its own subset of the data in the database.

Make Sure User Role and Position Are Compatible

Siebel applications restrict a user's ability to perform actions based on the roles and positions associated with the user. Make sure that the roles and positions of the users are compatible and allow the user to perform the required actions.

Troubleshooting Load Testing Issues

This section provides tips for resolving common issues associated with recording and running automated load test scripts.

NOTE: The causes and fixes described for these common issues are those that are most often encountered. However, they might not be the only such causes or fixes.

The following issues are addressed:

- Back or Refresh Error
- No Content HTTP Response on page 78
- Same Values Error on page 78
- Restoring the Context Error on page 79
- Cannot Locate Record Error on page 79
- End of File Error on page 79

Back or Refresh Error

The Siebel application displays an error applet indicating the following:

```
We detected an Error which may have occurred for on or more of the following reasons:
We are unable to process your request. This is most likely because you used the
browser BACK or REFRESH button to get to this point.
```

Cause

This issue can be caused by the following conditions:

- The SWETS was not parameterized for the current request.

- The SWEC was not correlated correctly for the current request.
- The request was submitted twice to the Siebel server without the SWEC being updated.
- The frame was not created on the server, possibly because the SWEMethod has changed since the script was recorded. A previous request should have set up a frame for the browser to download.

Fix

To resolve this issue, try the following fixes:

- Make sure that SWETS is parameterized.
- Make sure that you reset the SWE Count to the SWEC value of the first request in the iteration at the beginning of the iteration.

If these fixes do not resolve the issue, rerecord the script.

No Content HTTP Response

The server returns an error like the following:

```
HTTP/1.1 204 No Content
Server: Microsoft-IIS/5.0
Date: Fri, 31 Jan 2003 21:52:30 GMT
Content-Language: en
Cache-Control: no-cache
```

Cause

The row ID is not properly correlated.

Fix

Manually correlate the row ID.

Same Values Error

The server returns an error like the following:

```
@0`0`3`3``0`UC`1`Status`Error`SWEC`10`0`1`Errors`0`2`0`Level0`0`ErrMsg`The same
values for 'Name' already exist.  If you would like to enter a new record, please
ensure that the field values are unique.`ErrCode`28591`
```

Cause

One of the values in this request (in the previous code example, it is the value for the Name field) is the same as a value in another row in the database table.

Fix

You must replace this value with a unique value for each iteration for each user. The recommended solution is to use the row ID parameter for the value; this causes the value to be unique.

Restoring the Context Error

The server returns an error like the following:

```
@0`0`3`3``0`UC`1`Status`Error`SWEC`9`0`1`Errors`0`2`0`Level0`0`ErrMsg`An error
happened during restoring the context for requested location`ErrCode`27631`
```

Cause

The row ID is not properly correlated.

Fix

Manually correlate the row ID.

Cannot Locate Record Error

The server returns an error like the following:

```
@0`0`3`3``0`UC`1`Status`Error`SWEC`23`0`2`Errors`0`2`0`Level0`0`ErrMsg`Cannot
locate record within view: Contact Detail - Opportunities View applet: Opportunity
List Applet.`ErrCode`27573`
```

Cause

The input name SWERowId does not contain a row ID for a record on the Web page. This input name should have been parameterized. The parameter's source value may have changed its location.

Fix

Manually correlate the row ID.

End of File Error

The server returns an error like the following:

```
@0`0`3`3``0`UC`1`Status`Error`SWEC`28`0`1`Errors`0`2`0`Level0`0`ErrMsg`An end of
file error has occurred.  Please continue or ask your systems administrator to check
your application configuration if the problem persists.`ErrCode`28601`
```

Cause

The input name SWERowId does not contain a row ID for a record on the Web page. This input name should have been parameterized. The parameter's source value may have changed its location.

Fix

Manually correlate the row ID.

A Functional Test Object Reference

The test objects described in this section represent the API for Oracle's Siebel Test Automation. You can use these objects to compose functional test scripts that you execute using your automated testing tool. For more information on functional testing in Siebel applications, see Chapter 5, "Execute Siebel Functional Tests" and Chapter 9, "Automating Functional Tests." For information about writing and running test scripts, see the documentation that accompanies your test tool.

The following topics are included in this section:

- Functional Test Automation Objects for High Interactivity Siebel Applications
- Descriptions of each of the functional test automation objects, beginning with the SiebApplet Object on page 83
- Common Test Automation Object Properties on page 144
- Common Test Automation Object Methods on page 144
- Standard Interactivity Functional Test Objects on page 150

Functional Test Automation Objects for High Interactivity Siebel Applications

You can use the functional test automation objects described in this section to write test scripts in your test tool that test Siebel high-interactivity applications. Each of these objects corresponds to a functional component in the user interface of a Siebel application.

The following objects are categorized by the *logical type* of object they represent, and the topics that describe the objects are organized alphabetically in this reference. Additionally, each object is one of the following *functional* types:

- **Container objects.** Container objects such as SiebApplication are objects that contain child objects or controls.
- **Collection objects.** Collection objects such as SiebMenu are objects that represent a collection of repository objects.
- **Multivalue objects.** Multivalue objects such as SiebPickList are objects that contain a group of data values.
- **Singleton objects.** Singleton objects such as SiebThreadbar are objects for which only one instance can be represented in a given context.

The description for each object in this reference indicates the functional type of the object. Some objects can be more than one type. For example, SiebThreadbar is both a multivalue object and a singleton object, because it contains a group of data values and can have only one instance in a given context.

Each object provides events, properties, and methods that allow you to manipulate the object in a test automation environment. Events represent user actions that can be recorded with a functional test automation tool. Properties are object-specific settings that indicate the state of the object. Methods are object behaviors that can be used in test scripts, but cannot be recorded using a test tool.

Application Hierarchy Objects

These objects represent the layers of the application hierarchy in Siebel applications:

- SiebApplet Object on page 83
- SiebApplication Object on page 86
- SiebScreen Object on page 121
- SiebView Object on page 140

System Objects

These objects represent system-level controls that appear in Siebel applications:

- SiebMenu Object on page 112
- SiebPDQ Object on page 116
- SiebToolbar Object on page 135

Navigation Objects

These objects represent navigational elements that appear in Siebel applications:

- SiebPageTabs Object on page 114
- SiebScreenViews Object on page 122
- SiebTask Object on page 124
- SiebTaskStep Object on page 127
- SiebTaskUIPane Object on page 128
- SiebThreadbar Object on page 133
- SiebViewApplets Object on page 142

Core Control Objects

These objects represent simple controls that appear in Siebel applications:

- SiebButton Object on page 91
- SiebCheckbox Object on page 96
- SiebPicklist Object on page 118
- SiebTaskLink Object on page 126

- SiebText Object on page 131
- SiebTextArea Object on page 132

Complex Control Objects

These objects represent complex controls that appear in Siebel applications:

- SiebCalculator Object on page 92
- SiebCalendar Object on page 94
- SiebCurrency Object on page 100
- SiebInkData Object on page 102
- SiebList Object on page 103
- SiebRichText Object on page 120
- SiebTree Object on page 138

Custom Control Objects

These objects represent custom controls that appear in Siebel applications:

- SiebCommunicationsToolbar Object on page 98
- SiebTaskAssistant Object on page 125

SiebApplet Object

The SiebApplet object provides methods and properties that allow you to manipulate an applet in a test automation environment.

Parent

The SiebApplet object is a child of the SiebView Object.

Type

The SiebApplet object is a container object that is one of the Application Hierarchy Objects.

Events

There are no events associated with the SiebApplet object.

Methods

The following methods are available from the SiebApplet object:

- GetActiveControlName Method
- GetClassCount Method

- GetRepositoryName Method
- GetRepositoryNameByIndex Method
- IsControlExists Method
- SetActiveControl Method

For a description of these methods, see "SiebApplet Methods" on page 84.

Properties

The SiebApplet object has the following properties.

Property Name	Type	Description
ClassName = "SiebApplet"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsPopupApplet	Boolean	A Boolean value indicating whether or not the current applet represents a popup applet.
RecordCounter	String	Indicates the visible text of the record counter string (for example, "1 - 7 of 7+").
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebApplet Methods

This section provides descriptions of the methods available from the SiebApplet Object.

GetActiveControlName Method

The GetActiveControlName method can be called from the applet to get the name of the control that is currently active.

Available from

SiebApplet Object

Syntax

GetActiveControlName ()

Returns

A String containing the RepositoryName of the active control.

GetClassCount Method

For a description of the GetClassCount Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

IsControlExists Method

The IsControlExists method returns a Boolean value indicating whether or not the specified control exists.

Available from

SiebApplet Object

Syntax

IsControlExists (*RepName*)

Argument	Description
RepName	A String indicating the RepositoryName of the control.

Returns

A Boolean value indicating whether or not the specified control exists.

SetActiveControl Method

The SetActiveControl method can be called from the applet to set the focus to the specified control.

Available from

SiebApplet Object

Syntax

SetActiveControl (*ControlRepName*)

Argument	Description
ControlRepName	A String indicating the RepositoryName of the control.

Returns

Void

SiebApplication Object

The SiebApplication object provides methods and properties that allow you to manipulate an application in a test automation environment.

Parent

Not applicable. The SiebApplication object is the top-level object in the Siebel Test Automation hierarchy.

Type

The SiebApplication object is a container object that is one of the Application Hierarchy Objects.

Events

The SiebApplication object has the following events.

Event Name	Description
ProcessKeyboardAccelerator (*AccelKeys*)	Executes keyboard accelerators. *AccelKeys* is a String that specifies the accelerator keys to execute (for example, F9 or Ctrl-Shift-K).

Methods

The following methods are available from the SiebApplication object:

- GetBusyTime Method
- GetClassCount Method
- GetLastErrorCode Method
- GetLastErrorMessage Method
- GetLastOpId Method
- GetLastOpTime Method
- GetRepositoryName Method
- GetRepositoryNameByIndex Method

- GetSessionId Method
- SetTimeOut Method

For a description of these methods, see “SiebApplication Methods” on page 87.

Properties

The SiebApplication object has the following properties.

Property Name	Type	Description
ClassName = "SiebApplication"	Const String	See “Common Test Automation Object Properties” on page 144 for a description of this property.
RepositoryName	String	See “Common Test Automation Object Properties” on page 144 for a description of this property.
UIName	String	See “Common Test Automation Object Properties” on page 144 for a description of this property.

SiebApplication Methods

This section provides descriptions of the methods available from the SiebApplication Object.

GetBusyTime Method

The GetBusyTime method returns the time (in milliseconds) it took to execute the previous operation.

Available from

SiebApplication Object

Syntax

GetBusyTime()

Returns

An Integer indicating the number of milliseconds required to execute the previous operation.

Usage

The GetBusyState method is executed *asynchronously*; that is, it is executed immediately after the previous statement, without waiting for the previous statement to complete. Therefore, an accurate response time is not returned by the GetBusyState method until the previous operation and the GetBusyState method have completed execution. (See also, the description of the GetLastOpTime Method on page 89.)

The GetBusyState method is useful for measuring operations performed against standard-interactivity components within high-interactivity applications, such as Dashboard, SearchCenter, and Sitemap navigation.

GetClassCount Method

For a description of the GetClassCount Method, see "Common Test Automation Object Methods" on page 144.

GetLastErrorCode Method

The GetLastErrorCode method returns the last error code that was issued.

Available from

SiebApplication Object

Syntax

GetLastErrorCode()

Returns

An Integer indicating the last error code that was issued.

GetLastErrorMessage Method

The GetLastErrorMessage method returns the last error message that was issued.

Available from

SiebApplication Object

Syntax

GetLastErrorMessage()

Returns

A String containing the text of the last error message that was issued.

GetLastOpId Method

The GetLastOpId method returns the identification number of the previous operation.

Available from

SiebApplication Object

Syntax

GetLastOpId()

Returns

An Integer indicating the identification number of the previous operation.

Usage

This method returns the operation id from the Siebel ARM (Siebel Application Response Measurement) log file. You can use the operation id to examine timing and performance indicators contained in the log, and then map them to the lines of automation script that executed the corresponding operations. This is a manual process in which you must write a GetLastOpId method call into the automation script, and then use the returned operation id to find the corresponding entries in the Siebel ARM log file.

NOTE: The Siebel ARM feature captures timing data useful for monitoring the performance of the Siebel application. For more information on Siebel ARM, see the *Siebel Performance Tuning Guide*.

GetLastOpTime Method

The GetLastOpTime method returns the time (in milliseconds) it took to execute the previous operation.

Available from

SiebApplication Object

Syntax

GetLastOpTime()

Returns

An Integer indicating the number of milliseconds required to execute the previous operation.

Usage

The GetLastOpTime method is executed *synchronously*. That is, it is executed after the previous statement has completed execution. (See also, the description of the GetBusyTime Method on page 87.)

The GetLastOpTime method is useful for measuring operations such as navigating to a screen, raising a popup applet, and selecting a menu item.

GetSessionId Method

The GetSessionId method returns the Session id of the current Siebel client.

Available from

SiebApplication Object

Syntax

GetSessionId()

Returns

A String that indicates the id of the current Siebel client session.

Usage

The returned string can be used to correlate the server-side Siebel ARM (Siebel Application Response Measurement) log files with the client operation id.

NOTE: The Siebel ARM feature captures timing data useful for monitoring the performance of the Siebel application. For more information on Siebel ARM, see the *Siebel Performance Tuning Guide*.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

SetTimeOut Method

The SetTimeOut method specifies the length of time to wait for the application to return from an operation before timing out.

Available from

SiebApplication Object

Syntax

SetTimeOut (*TimeOutSeconds*)

Argument	Description
TimeOutSeconds	An Integer that indicates the number of seconds to wait before timing out.

Returns

An Integer indicating the previous timeout value (in seconds).

Usage

When the timeout is exceeded, the test tool stops execution of the script, and displays an error dialog box. The new timeout value is valid in the current test script until the next SetTimeOut is executed or the application restarts. When the SetTimeOut method is not used or it goes out of scope, the default value of 60 seconds is used.

SiebButton Object

The SiebButton object provides events and properties that allow you to manipulate a button control in a test automation environment.

Parent

The SiebButton object is a child of the SiebApplet Object.

Type

The SiebButton object is one of the Core Control Objects.

Events

The SiebButton object has the following event.

Event Name	Description
Click	Clicks the button.

Methods

There are no methods available from the SiebButton object.

Properties

The SiebButton object has the following properties.

Property Name	Type	Description
ClassName = "SiebButton"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsEnabled	Boolean	Indicates whether or not the button object is enabled.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebCalculator Object

The SiebCalculator object provides methods and properties that allow you to manipulate a calculator control in a test automation environment.

Parent

The SiebCalculator object is a child of the SiebApplet Object, SiebList Object, and SiebCurrency Object.

Type

The SiebCalculator object is one of the Complex Control Objects.

Events

The SiebCalculator object has the following events.

Event Name	Description
CancelPopup	Closes the calculator popup applet without saving changes (for example, by clicking the Cancel button).
ClickKey (*KeyValue*)	Clicks a key in the calculator popup applet. *KeyValue* specifies the key to click. This event is only valid when the calculator popup applet is open.
OpenPopup	Opens the calculator popup applet.
ProcessKey (*KeyName*)	Invokes the specified key inside the control. *KeyName* is a String that specifies the key to invoke. The only *KeyName* accepted by the ProcessKey event is "Enter".
SetText (*TextValue*)	Enters text in the text box. *TextValue* specifies the text to enter.

Methods

The following method is available from the SiebCalculator object:

- ClickKeys Method

For a description of this method, see "SiebCalculator Methods" on page 93.

Properties

The SiebCalculator object has the following properties.

Property Name	Type	Description
ClassName = "SiebCalculator"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsEnabled	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsOpen	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsRequired	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Text	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebCalculator Methods

This section provides descriptions of the methods available from the SiebCalculator Object.

ClickKeys Method

The ClickKeys method clicks keys in the open calculator popup applet.

Available from

SiebCalculator Object

Syntax

ClickKeys (*KeysValue*)

Argument	Description
KeysValue	A String that indicates the keys to be clicked

Returns

Void

Usage

This method is only valid when the calculator popup applet is open.

SiebCalendar Object

The SiebCalendar object provides events and properties that allow you to manipulate a calendar control in a test automation environment.

Parent

The SiebCalendar object is a child of the SiebApplet Object, SiebList Object, and SiebCurrency Object.

Type

The SiebCalendar object is one of the Complex Control Objects.

Events

The SiebCalendar object has the following events.

Event Name	Description
CancelPopup	Closes the calendar popup applet without saving changes (for example, by clicking the Cancel button).
ClosePopup	Closes the calendar popup applet after saving changes (for example, by clicking the Save button).
NextMonth	Changes the displayed month in the calendar popup applet to the next month (for example, from February to March).
OpenPopup	Opens the calendar popup applet.
PrevMonth	Changes the displayed month in the calendar popup applet to the previous month (for example, from April to March).

Event Name	Description
ProcessKey (*KeyName*)	Invokes the specified key inside the control. *KeyName* is a String that specifies the key to invoke. The following values are valid for *KeyName*: ■ "Enter" ■ "LeftArrow" ■ "RightArrow" ■ "UpArrow" ■ "DownArrow"
SelectTimeZone (*TimeZone*)	Sets the Time Zone in the open calendar popup applet. *TimeZone* is a String that specifies the Time Zone value in the format "(GMT-09:00) Alaska".
SetDay (*Day*)	Sets the Day in the open calendar popup applet. *Day* is an Integer that specifies the new day value.
SetMonth (*Month*)	Sets the Month in the open calendar popup applet. *Month* is an Integer that specifies the new month value.
SetText (*TextValue*)	Enters text in the parent control for the calendar. *TextValue* specifies the text to enter.
SetTime (*TimeText*)	Sets the Time in the open calendar popup applet. *TimeText* is a String that specifies the new time value.
SetYear (*Year*)	Sets the Year in the open calendar popup applet. *Year* is an Integer that specifies the new year value.

Methods

There are no methods available from the SiebCalendar object.

Properties

The SiebCalendar object has the following properties.

Property Name	Type	Description
CalendarType	String	Specifies the type of calendar. The value of this property must be either Date, DateTime, or DateTimeZone.
ClassName = "SiebCalendar"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Day	String	Specifies the current day in the open calendar control.
IsEnabled	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.

Property Name	Type	Description
IsOpen	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsRequired	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Month	String	Specifies the current month in the open calendar control.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Text	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Time	String	Specifies the current time in the open calendar control.
TimeZone	String	Specifies the current time zone in the open calendar control.
TimeZoneCount	Integer	Specifies the number of time zones in the time zone picklist for the open calendar control.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Year	String	Specifies the current year in the open calendar control.

SiebCheckbox Object

The SiebCheckbox object provides events and properties that allow you to manipulate a checkbox in a test automation environment.

Parent

The SiebCheckbox object is a child of the SiebApplet Object and the SiebList Object.

Type

The SiebCheckbox object is one of the Core Control Objects.

Events

The SiebCheckbox object has the following events.

Event Name	Description
ProcessKey (*KeyName*)	Invokes the specified key inside the control. *KeyName* is a String that specifies the key to invoke. The only *KeyName* accepted by the ProcessKey event is “Enter”.
SetIndeterminate	Sets the state of the checkbox to intermediate. The indeterminate state is available only in query mode.
SetOff	Sets the state of the checkbox to not checked.
SetOn	Sets the state of the checkbox to checked.

Methods

There are no methods available from the SiebCheckbox object.

Properties

The SiebCheckbox object has the following properties.

Property Name	Type	Description
CheckState	String	Indicates the state of the SiebCheckbox object. The following values are valid for CheckState: ■ “Checked” ■ “Unchecked” ■ “Indeterminate”
ClassName = "SiebCheckbox"	Const String	See “Common Test Automation Object Properties” on page 144 for a description of this property.
IsChecked	Boolean	Indicates whether or not the checkbox is checked. The value is TRUE for the indeterminate state.
IsEnabled	Boolean	See “Common Test Automation Object Properties” on page 144 for a description of this property.
IsRequired	Boolean	See “Common Test Automation Object Properties” on page 144 for a description of this property.
RepositoryName	String	See “Common Test Automation Object Properties” on page 144 for a description of this property.
UIName	String	See “Common Test Automation Object Properties” on page 144 for a description of this property.

SiebCommunicationsToolbar Object

The SiebCommunicationsToolbar object provides methods and properties that allow you to manipulate the communications toolbar in a test automation environment.

Parent

The SiebCommunicationsToolbar object is a child of the SiebApplication Object.

Type

The SiebCommunicationsToolbar object is one of the Custom Control Objects.

Events

The SiebCommunicationsToolbar object has the following events.

Event Name	Description
Click (*ButtonName*)	Clicks a button. *ButtonName* is a String that specifies the RepositoryName of the button.
SelectWorkItem (*ItemName*)	Selects a WorkItem. *ItemName* is a String that specifies the WorkItem to select.
SetText (*Text*)	Enters text in the text box. *Text* is a String that specifies the text to enter.
ShowButtonToolTip (*ButtonName*)	Displays the tooltip for the button. *ButtonName* is a String that specifies the RepositoryName of the button. This event is significant because the CommunicationsToolbar tooltip is dynamic, causing a round-trip to the server.

Methods

The following methods are available from the SiebCommunicationsToolbar object:

- GetButtonState Method
- GetButtonTooltip Method

For a description of these methods, see "SiebCommunicationsToolbar Methods" on page 99.

Properties

The SiebCommunicationsToolbar object has the following properties.

Property Name	Type	Description
ClassName = "SiebCommunicationsToolbar"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
CurrentWorkItem	String	Indicates the currently selected work item.
IsEnabled	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsVisible	Boolean	Indicates whether or not the object is enabled. You can not checkpoint this property because the control is not rendered by the browser when it is not visible. However, you can manually write script code to check the property.
MediaType	String	Indicates the type of media.
Message	String	The communication channel information displayed next to the Siebel menu that indicates where the message comes from (for example, Call Display).
RepositoryName = "SiebCommunicationsToolbar"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Text	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName = "CommunicationsToolbar"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
WorkItemDuration	String	Indicates the duration of the currently selected work item.

SiebCommunicationsToolbar Methods

This section provides descriptions of the methods available from the SiebCommunicationsToolbar Object.

GetButtonState Method

The GetButtonState method returns the state of the specified button.

Available from

SiebCommunicationsToolbar Object

Syntax

GetButtonState (*ButtonName*)

Argument	Description
ButtonName	A String that specifies the RepositoryName of the button

Returns

A String that indicates the logical state of the button (for example, blinking).

GetButtonTooltip Method

The GetButtonTooltip method returns the tooltip for the specified button.

Available from

SiebCommunicationsToolbar Object

Syntax

GetButtonTooltip (*ButtonName*)

Argument	Description
ButtonName	A String that specifies the RepositoryName of the button

Returns

A String containing the text of the tooltip.

SiebCurrency Object

The SiebCurrency object provides methods and properties that allow you to manipulate a currency calculator in a test automation environment.

Parent

The SiebCurrency object is a child of the SiebApplet Object and the SiebList Object.

Type

The SiebCurrency object is a container object that is one of the Complex Control Objects.

Events

The SiebCurrency object has the following events.

Event Name	Description
CancelPopup	Closes the currency calculator popup applet without saving changes (for example, by clicking the Cancel button).
ClosePopup	Saves changes and closes the currency calculator popup applet.
OpenPopup	Opens the currency calculator popup applet.
ProcessKey (*KeyName*)	Invokes the specified key inside the control. *KeyName* is a String that specifies the key to invoke. The only *KeyName* accepted by the ProcessKey event is “Enter”.
SetText (*TextValue*)	Enters text in the currency calculator popup applet. *TextValue* specifies the text to enter.

Methods

The following methods are available from the SiebCurrency object:

- GetClassCount Method
- GetRepositoryName Method
- GetRepositoryNameByIndex Method

For a description of these methods, see “SiebCurrency Methods” on page 102.

Properties

The SiebCurrency object has the following properties.

Property Name	Type	Description
Amount	String	A data value representing the value of the Amount field in the popup applet.
ClassName = "SiebCurrency"	Const String	See “Common Test Automation Object Properties” on page 144 for a description of this property.
CurrencyCode	String	A data value representing the value of the Currency Code field in the popup applet.
ExchangeDate	String	A data value representing the value of the Exchange Date field in the popup applet.
IsEnabled	Boolean	See “Common Test Automation Object Properties” on page 144 for a description of this property.
IsOpen	Boolean	See “Common Test Automation Object Properties” on page 144 for a description of this property.

Property Name	Type	Description
IsRequired	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Text	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebCurrency Methods

This section provides descriptions of the methods available from the SiebCurrency Object.

GetClassCount Method

For a description of the GetClassCount Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

SiebInkData Object

The SiebInkData object provides properties that allow you to manipulate an InkData object in a test automation environment.

Parent

The SiebInkData object is a child of the SiebApplet Object.

Type

The SiebInkData object is one of the Complex Control Objects.

Events

There are no events associated with the SiebInkData object.

Methods

There are no methods available from the SiebInkData object.

Properties

The SiebInkData object has the following properties.

Property Name	Type	Description
ClassName = "SiebInkData"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsEnabled	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebList Object

The SiebList object provides methods and properties that allow you to manipulate a list object in a test automation environment.

Parent

The SiebList object is a child of the SiebApplet Object.

Type

The SiebList object is a singleton container object that is one of the Complex Control Objects. Due to the complex nature of this object, it is also a collection object.

Events

The SiebList object has the following events.

Event Name	Description
ActivateRow (*RowNumber*)	Activates a row in the list. *RowNumber* is an Integer that indicates which row to activate, based on the number of currently visible rows. (The show more or show less button determines the number of currently visible rows.)
AscendSort (*ColumnRepName*)	Sorts the list on the column in ascending order. *ColumnRepName* is a String that indicates the RepositoryName of the column to sort on.
ClickHier	Toggles the active row between expanded and collapsed (for hierarchical lists).
DescendSort (*ColumnRepName*)	Sorts the list on the column in descending order. *ColumnRepName* is a String that indicates the RepositoryName of the column to sort on.
DoubleClick (*RowNumber*, *ColumnRepName*)	Double-clicks a row. *RowNumber* is an Integer that specifies the row to double-click. *ColumnRepName* is a String that specifies the RepositoryName of the column to double-click. The *ColumnRepName* argument is optional. When recording a script, *ColumnRepName* is recorded only when the double-click occurs on a control in the row.
DrillDownColumn (*ColumnRepName*, *RowNumber*)	Clicks a drilldown link. *ColumnRepName* is a String that specifies the RepositoryName of the column that contains the drilldown link. *RowNumber* is an Integer that specifies the row that contains the drilldown link, based on the number of currently visible rows (determined by show more/less).
FirstRowSet	Navigates to the first set of records in the list.
LastRowSet	Navigates to the last set of records in the list.
NextRow	Navigates to the next row in the list.
NextRowSet	Navigates to the next set of records in the list.
PreviousRow	Navigates to the previous row in the list.
PrevRowSet	Navigates to the previous set of records in the list.

Event Name	Description
SelectRow (*RowNumber*, *SelectType*)	Activates a row in conjunction with a multiselect key. *RowNumber* is an Integer that specifies which row to activate, based on the number of currently visible rows (determined by show more and show less). *SelectType* is a String that specifies which multiselect key (either Shift or Control) to use. If *SelectType* is not defined, no multiselect key is used.
ToggleFreezeColumns (*ColumnRepName*)	Freezes or unfreezes the columns to horizontal scrolling. *ColumnRepName* is a String that specifies the RepositoryName of the last column to toggle. This event is generated by a double-click on a column header within a list applet. The freeze begins with the first column and ends with the double-clicked column.

Methods

The following methods are available from the SiebList object:

- GetActiveControl Method
- GetCellText Method
- GetColumnRepositoryName Method
- GetColumnRepositoryNameByIndex Method
- GetColumnSort Method
- GetColumnType Method
- GetColumnUIName Method
- GetTotalsValue Method
- IsColumnDrilldown Method
- IsColumnExists Method
- IsRowExpanded Method
- SetActiveControl Method

For a description of these methods, see "SiebList Methods" on page 106.

Properties

The SiebList object has the following properties.

Property Name	Type	Description
ActiveRow	Integer	Specifies which row is currently active, based on the number of currently visible rows (determined by show more and show less).
ClassName = "SiebList"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
ColumnsCount	Integer	Specifies the number of currently visible columns in the list.
RepositoryName = "SiebList"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RowsCount	Integer	Specifies the number of currently visible rows (determined by show more and show less).
SelectedRows	String	A pipe-delimited string of row numbers that are currently selected.
UIName = "List"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebList Methods

This section provides descriptions of the methods available from the SiebList Object.

GetActiveControl Method

The GetActiveControl method returns the name of the control that is currently active.

Available from

SiebList Object

Syntax

GetActiveControl ()

Returns

A String containing the RepositoryName of the active control.

GetCellText Method

The GetCellText method returns the text of the specified cell.

Available from

SiebList Object

Syntax

GetCellText (*ColumnRepName*, *RowNumber*)

Argument	Description
ColumnRepName	A String that indicates the RepositoryName of the column containing the cell
RowNumber	An Integer that indicates the number of the row containing the cell

Returns

A String containing the text of the specified cell.

GetColumnRepositoryName Method

The GetColumnRepositoryName method returns the RepositoryName of the column that has the UIName specified in the argument.

Available from

SiebList Object

Syntax

GetColumnRepositoryName (*ColumnUIName*)

Argument	Description
ColumnUIName	A String that indicates the UIName of the column

Returns

A String containing the RepositoryName of the specified column.

GetColumnRepositoryNameByIndex Method

The GetColumnRepositoryNameByIndex method returns the RepositoryName of the column that has the index number specified in the argument.

Available from

SiebList Object

Syntax

GetColumnRepositoryNameByIndex (*ColumnIndex*)

Argument	Description
ColumnIndex	An Integer that indicates the index number of the column

Returns

A String containing the RepositoryName of the specified column.

GetColumnSort Method

The GetColumnSort method returns a String (either Ascend or Descend) indicating how the specified column is currently sorted.

Available from

SiebList Object

Syntax

GetColumnSort (*ColumnRepName*)

Argument	Description
ColumnRepName	A String that indicates the RepositoryName of the column

Returns

A String indicating how the specified column is currently sorted (either Ascend or Descend).

GetColumnType Method

The GetColumnType method returns the type of the underlying control (for example, SiebText) for the column that has the RepositoryName specified in the argument.

Available from

SiebList Object

Syntax

GetColumnType (*ColumnRepName*)

Argument	Description
ColumnRepName	A String that indicates the RepositoryName of the column

Returns

A String indicating the type of the underlying control for the specified column.

GetColumnUIName Method

The GetColumnUIName method returns the UIName of the column that has the RepositoryName specified in the argument.

Available from

SiebList Object

Syntax

GetColumnUIName (*ColumnRepName*)

Argument	Description
ColumnRepName	A String that indicates the RepositoryName of the column

Returns

A String indicating the UIName of the specified column.

GetTotalsValue Method

The GetTotalsValue method returns the value in the totals row of a list control for the specified column. If the specified control does not have a totals row, GetTotalsValue returns an empty string.

Available from

SiebList Object

Syntax

GetTotalsValue (*ColumnRepName*)

Argument	Description
ColumnRepName	A String that indicates the RepositoryName of the column

Returns

A String indicating the value in the totals row for the specified column, or an empty String if the control does not have a totals row.

IsColumnDrilldown Method

The IsColumnDrilldown method returns a Boolean value indicating whether or not the specified column is a drilldown column.

Available from

SiebList Object

Syntax

IsColumnDrilldown (*ColumnRepName*)

Argument	Description
ColumnRepName	A String that indicates the RepositoryName of the column

Returns

A Boolean value indicating whether or not the specified column is a drilldown column.

IsColumnExists Method

The IsColumnExists method returns a Boolean value indicating whether or not the specified column exists.

Available from

SiebList Object

Syntax

IsColumnExists (*ColumnRepName*)

Argument	Description
ColumnRepName	A String that indicates the RepositoryName of the column

Returns

A Boolean value indicating whether or not the specified column exists.

IsRowExpanded Method

The IsRowExpanded method returns a Boolean value indicating whether or not the specified row is expanded (for hierarchical lists). For nonhierarchical lists, IsRowExpanded always returns TRUE indicating that the row is expanded.

Available from

SiebList Object

Syntax

IsRowExpanded (*RowNumber*)

Argument	Description
RowNumber	An Integer that indicates the number of the row

Returns

A Boolean value indicating whether the row is expanded (TRUE) or collapsed (FALSE). Always returns TRUE for nonhierarchical lists.

SetActiveControl Method

The SetActiveControl method sets the focus in the list to the specified control.

Available from

SiebList Object

Syntax

SetActiveControl (*ControlRepName*)

Argument	Description
ControlRepName	A String indicating the RepositoryName of the control.

Returns

Void

SiebMenu Object

The SiebMenu object provides methods and properties that allow you to manipulate menus and menu items in a test automation environment.

Parent

The SiebMenu object is a child of the SiebApplet Object and the SiebApplication Object.

Type

The SiebMenu object is a collection object that is one of the System Objects.

Events

The SiebMenu object has the following event.

Event Name	Description
Select (*ItemName*)	Selects a menu item from the Siebel application menu. *ItemName* is a String that specifies the RepositoryName of the Menu Item object.

Methods

The following methods are available from the SiebMenu object:

- GetRepositoryName Method
- GetRepositoryNameByIndex Method
- GetUIName Method
- IsEnabled Method
- IsExists Method

For a description of these methods, see “SiebMenu Methods” on page 113.

Properties

The SiebMenu object has the following properties.

Property Name	Type	Description
ClassName = "SiebMenu"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Count	Integer	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName = "SiebMenu"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName = "Menu"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebMenu Methods

This section provides descriptions of the methods available from the SiebMenu Object.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

GetUIName Method

For a description of the GetUIName Method, see "Common Test Automation Object Methods" on page 144.

IsEnabled Method

The IsEnabled method returns a Boolean value indicating whether or not the specified menu item is enabled.

Available from

SiebMenu Object

Syntax

IsEnabled (*ItemName*)

Argument	Description
ItemName	A String that indicates the RepositoryName of the Menu Item object

Returns

A Boolean value indicating whether the specified menu item is enabled (TRUE) or disabled (FALSE).

IsExists Method

For a description of the IsExists Method, see "Common Test Automation Object Methods" on page 144.

SiebPageTabs Object

The SiebPageTabs object provides methods and properties that allow you to navigate page tabs in a test automation environment.

Parent

The SiebPageTabs object is a child of the SiebApplication Object.

Type

The SiebPageTabs object is a singleton collection object that is one of the Navigation Objects.

Events

The SiebPageTabs object has the following events.

Event Name	Description
GotoScreen (*ScreenName*)	Navigates to a PageTab. *ScreenName* is a String that specifies the RepositoryName of the PageTab object.
GotoView (*ViewName*)	Navigates to a View. *ViewName* is a String that specifies the RepositoryName of the View object, represented by an Aggregate Category link.

Methods

The following methods are available from the SiebPageTabs object:

- GetRepositoryName Method
- GetRepositoryNameByIndex Method

- GetUIName Method
- IsExists Method

For a description of these methods, see "SiebPageTabs Methods" on page 115.

Properties

The SiebPageTabs object has the following properties.

Property Name	Type	Description
ActiveScreen	String	Indicates the RepositoryName of the active PageTab object.
ActiveView	String	Indicates the RepositoryName of the active View, as represented by the active Aggregate Category link.
ClassName = "SiebPageTabs"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName = "SiebPageTabs"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
ScreenCount	Integer	The total count of PageTabs for Screen navigation.
UIName = "PageTabs"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
ViewCount	Integer	The total count of available (using Aggregate Category links) from the active PageTab object.

SiebPageTabs Methods

This section provides descriptions of the methods available from the SiebPageTabs Object.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

GetUIName Method

For a description of the GetUIName Method, see "Common Test Automation Object Methods" on page 144.

IsExists Method

For a description of the IsExists Method, see "Common Test Automation Object Methods" on page 144.

SiebPDQ Object

The SiebPDQ object provides methods and properties that allow you to manipulate a predefined query (PDQ) in a test automation environment.

Parent

The SiebPDQ object is a child of the SiebScreen Object.

Type

The SiebPDQ object is a singleton multivalue object that is one of the System Objects.

Events

The SiebPDQ object has the following event.

Event Name	Description
Select (*ItemName*)	Selects a PDQ. *ItemName* is a String that specifies the visible title of the PDQ.

Methods

The following methods are available from the SiebPDQ object:

- GetPDQByIndex Method
- IsExists Method

For a description of these methods, see "SiebPDQ Methods" on page 117.

Properties

The SiebPDQ object has the following properties.

Property Name	Type	Description
ActivePDQ	String	Indicates the visible title of the active PDQ.
ClassName = "SiebPDQ"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Count	Integer	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName = "SiebPDQ"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName = "PDQ"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebPDQ Methods

This section provides descriptions of the methods available from the SiebPDQ Object.

GetPDQByIndex Method

The GetPDQByIndex method returns the visible title of the specified PDQ.

Syntax

GetPDQByIndex (*Index*)

Argument	Description
Index	An Integer that indicates the index of the PDQ object within the Count property

Returns

A String indicating the visible title of the PDQ object.

IsExists Method

For a description of the IsExists Method, see "Common Test Automation Object Methods" on page 144.

SiebPicklist Object

The SiebPicklist object provides methods and properties that allow you to manipulate a picklist in a test automation environment.

Parent

The SiebPicklist object is a child of the SiebApplet Object and the SiebList Object.

Type

The SiebPicklist object is a multivalue object that is one of the Core Control Objects.

Events

The SiebPicklist object has the following events.

Event Name	Description
ProcessKey (*KeyName*)	Invokes the specified key inside the control. *KeyName* is a String that specifies the key to invoke. The only *KeyName* accepted by the ProcessKey event is “Enter”.
Select (*ItemName*)	Selects an item from the picklist. *ItemName* is a String that specifies the visible text of the picklist item.
SetText (*TextValue*)	Enters text in the picklist control (for example, when composing a query). *TextValue* is a String that specifies the text to enter. The SetText event fails on picklists that are used as applet toggle controls.

Methods

The following methods are available from the SiebPicklist object:

- GetItemByIndex Method
- IsExists Method

For a description of these methods, see “SiebPicklist Methods” on page 119.

Properties

The SiebPicklist object has the following properties.

Property Name	Type	Description
ActiveItem	String	Indicates the visible title of the currently selected Picklist item.
ClassName = "SiebPickList"	Const String	See “Common Test Automation Object Properties” on page 144 for a description of this property.

Property Name	Type	Description
Count	Integer	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsEnabled	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsOpen	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsRequired	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebPicklist Methods

This section provides descriptions of the methods available from the SiebPicklist Object.

GetItemByIndex Method

The GetItemByIndex method returns the visible title of the specified picklist item.

Syntax

GetItemByIndex (*Index*)

Argument	Description
Index	An Integer that specifies the index of the item in the Count property.

Returns

A String indicating the visible text of the picklist item.

IsExists Method

For a description of the IsExists Method, see "Common Test Automation Object Methods" on page 144.

SiebRichText Object

The SiebRichText object provides methods and properties that allow you to manipulate a rich text control in a test automation environment.

Parent

The SiebRichText object is a child of the SiebApplet Object.

Type

The SiebRichText object is one of the Complex Control Objects.

Events

The SiebRichText object has the following event.

Event Name	Description
SetText (*TextValue*)	Enters text in the text area. *TextValue* is a String that specifies the text to enter.

Methods

There are no methods available from the SiebRichText object.

Properties

The SiebRichText object has the following properties.

Property Name	Type	Description
ClassName = "SiebRichText"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsEnabled	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsRequired	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Text	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebScreen Object

The SiebScreen object provides methods and properties that allow you to manipulate a screen object in a test automation environment.

Parent

The SiebScreen object is a child of the SiebApplication Object.

Type

The SiebScreen object is a container object that is one of the Application Hierarchy Objects.

Events

The SiebScreen object has no events associated with it.

Methods

The following methods are available from the SiebScreen object:

- GetClassCount Method
- GetRepositoryName Method
- GetRepositoryNameByIndex Method

For a description of these methods, see "SiebScreen Methods" on page 121.

Properties

The SiebScreen object has the following properties.

Property Name	Type	Description
ClassName = "SiebScreen"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebScreen Methods

This section provides descriptions of the methods available from the SiebScreen Object.

GetClassCount Method

For a description of the GetClassCount Method, see “Common Test Automation Object Methods” on page 144.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see “Common Test Automation Object Methods” on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see “Common Test Automation Object Methods” on page 144.

SiebScreenViews Object

The SiebScreenViews object provides methods and properties that allow you to manipulate a screen view in a test automation environment.

Parent

The SiebScreenViews object is a child of the SiebScreen Object.

Type

The SiebScreenViews object is a singleton collection object that is one of the Navigation Objects.

Events

The SiebScreenViews object has the following event.

Event Name	Description
Goto (*ViewName*, *Level*)	Navigates to a screen view. *ViewName* is a String that specifies the name of the screen view; *Level* is a String that specifies its level (L2, L3, or L4).

Methods

The following methods are available from the SiebScreenViews object:

- GetRepositoryName Method
- GetRepositoryNameByIndex Method
- GetUIName Method

For a description of these methods, see "SiebScreenViews Methods" on page 123.

Properties

The SiebScreenViews object has the following properties.

Property Name	Type	Description
ActiveView	String	Indicates the RepositoryName of the active Screen View.
ClassName = "SiebScreenViews"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
L2Count	Integer	Indicates the number of second-level screen views on the current screen.
L3Count	Integer	Indicates the number of third-level screen views on the current screen.
L4Count	Integer	Indicates the number of fourth-level screen views on the current screen.
RepositoryName = "SiebScreenViews"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName = "ScreenViews"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebScreenViews Methods

This section provides descriptions of the methods available from the SiebScreenViews Object.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

GetUIName Method

For a description of the GetUIName Method, see "Common Test Automation Object Methods" on page 144.

SiebTask Object

The SiebTask object provides methods and properties that allow you to manipulate a task in a test automation environment.

Parent

The SiebTask object is a child of the SiebApplication Object.

Type

The SiebTask object is a container object that is one of the Navigation Objects.

Events

There are no events associated with the SiebTask object.

Methods

The following methods are available from the SiebTask object:

- GetClassCount Method
- GetRepositoryName Method
- GetRepositoryNameByIndex Method

For a description of these methods, see "SiebTask Methods" on page 124.

Properties

The SiebTask object has the following properties.

Property Name	Type	Description
ClassName = "SiebTask"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebTask Methods

This section provides descriptions of the methods available from the SiebTask Object.

GetClassCount Method

For a description of the GetClassCount Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

SiebTaskAssistant Object

The SiebTaskAssistant object provides events and properties that allow you to manipulate the task assistant in a test automation environment.

Parent

The SiebTaskAssistant object is a child of the SiebApplication Object.

Type

The SiebTaskAssistant object is one of the Custom Control Objects.

Events

The SiebTaskAssistant object has the following events.

Event Name	Description
Close	Closes the task list.
Done	Clicks the Return To link.
Next	Clicks the See additional steps link.
Start (*TaskId*)	Clicks the link of the specified task to start a task. *TaskId* is a String that indicates the id of the task to start.
Step (*StepNum*)	Clicks the specified step. *StepNum* is a String that indicates the number of the step.
StepView (*StepNum*)	Clicks the View link for the specified step. *StepNum* is a String that indicates the number of the step.

Methods

There are no methods available from the SiebTaskAssistant object.

Properties

The SiebTaskAssistant object has the following properties.

Property Name	Type	Description
ActiveStep	String	Indicates the number of the current step.
ActiveTask	String	Indicates the id of the current task.
ClassName = "SiebTaskAssistant"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName = "SiebTaskAssistant"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
StepCount	String	Indicates the number of visible steps.
TaskCount	String	Indicates the number of tasks in the current list.
TaskId	String	Indicates the name of the current task.
UIName = "TaskAssistant"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebTaskLink Object

The SiebTaskLink object provides an event and properties that allow you to start a task in a test automation environment.

Parent

The SiebTaskLink object is a child of the SiebTaskUIPane Object.

Type

The SiebTaskLink object is one of the Core Control Objects.

Events

The SiebTaskLink object has the following event.

Event Name	Description
Click	Clicks the task link in the task pane to start the task.

Methods

There are no methods available from the SiebTaskLink object.

Properties

The SiebTaskLink object has the following properties.

Property Name	Type	Description
ClassName = "SiebTaskLink"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebTaskStep Object

The SiebTaskStep object provides methods and properties that allow you to manipulate a task step in a test automation environment.

Parent

The SiebTaskStep object is a child of the SiebTask Object.

Type

The SiebTaskStep object is one of the Navigation Objects.

Events

There are no events associated with the SiebTaskStep object.

Methods

The following methods are available from the SiebTaskStep object:

- GetClassCount Method
- GetRepositoryName Method
- GetRepositoryNameByIndex Method

For a description of these methods, see "SiebTaskStep Methods" on page 128.

Properties

The SiebTaskStep object has the following properties.

Property Name	Type	Description
ActiveApplet	String	Indicates the RepositoryName of the active Applet object.
AppletCount	Integer	Indicates the number of applets present in the current context.
ClassName = "SiebTaskStep"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
TaskStepTitle	String	Indicates the title of the current task step.
TaskViewTitle	String	Indicates the title of the active task view.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebTaskStep Methods

This section provides descriptions of the methods available from the SiebTaskStep Object.

GetClassCount Method

For a description of the GetClassCount Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

SiebTaskUIPane Object

The SiebTaskUIPane object provides methods and properties that allow you to manipulate the task pane in a test automation environment.

Parent

The SiebTaskUIPane object is a child of the SiebApplication Object.

Type

The SiebTaskUIPane object is a container object that is one of the Navigation Objects.

Events

The SiebTaskUIPane object has the following events.

Event Name	Description
Close	Closes the task pane.
GotoInBox	Clicks the Go to Inbox link to navigate to the Inbox.

Methods

The following methods are available from the SiebTaskUIPane object:

- GetClassCount Method
- GetRepositoryNameByIndex Method
- GetStepByIndex Method
- GetTaskByIndex Method
- Start Method

For a description of these methods, see "SiebTaskUIPane Methods" on page 130.

Properties

The SiebTaskUIPane object has the following properties.

Property Name	Type	Description
ActiveTask	Integer	Indicates the id of the current task.
ClassName = "SiebTaskUIPane"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName = "TaskUIPane"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
StepCount	Integer	Indicates the number of steps in the current task in the task pane.
TaskCount	Integer	Indicates the number of tasks in the task pane.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebTaskUIPane Methods

This section provides descriptions of the methods available from the SiebTaskUIPane Object.

GetClassCount Method

For a description of the GetClassCount Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

GetStepByIndex Method

The GetStepByIndex method returns the RepositoryName of the SiebTaskStep object specified by the given index.

Available from

SiebTaskUIPane Object

Syntax

GetStepByIndex (*StepIndex*)

Argument	Description
StepIndex	An integer indicating the index of the step in the StepCount property.

Returns

A String indicating the RepositoryName of the object.

GetTaskByIndex Method

The GetTaskByIndex method returns the RepositoryName of the SiebTask object specified by the given index.

Available from

SiebTaskUIPane Object

Syntax

GetTaskByIndex (*TaskIndex*)

Argument	Description
TaskIndex	An integer indicating the index of the task in the TaskCount property.

Returns

A String indicating the RepositoryName of the object.

Start Method

The Start method starts the specified task by clicking its link in the task pane.

Available from

SiebTaskUIPane Object

Syntax

Start (*TaskName*, *TaskGroup*)

Argument	Description
TaskName	A String indicating the RepositoryName of the task.
TaskGroup	A String indicating the group to which the task belongs.

Returns

Void

SiebText Object

The SiebText object provides methods and properties that allow you to manipulate a text box in a test automation environment.

Parent

The SiebText object is a child of the SiebApplet Object, the SiebCurrency Object, and the SiebList Object.

Type

The SiebText object is one of the Core Control Objects.

Events

The SiebText object has the following events.

Event Name	Description
OpenPopup	Opens the associated popup applet.
ProcessKey (*KeyName*)	Invokes the specified key inside the control. *KeyName* is a String that specifies the key to invoke. The only *KeyName* accepted by the ProcessKey event is Enter.
SetText (*TextValue*)	Enters text into the text box. *TextValue* is a String that specifies the text to enter.

Methods

There are no methods available from the SiebText object.

Properties

The SiebText object has the following properties.

Property Name	Type	Description
ClassName = "SiebText"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsEnabled	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsEncrypted	Boolean	A Boolean value indicating whether or not the text value of the object is masked, such as in a password text box.
IsRequired	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
PopupType	String	Indicates the type of popup associated with the text box (SVPick for a single-value picklist, MVPick for a multiple-value picklist, or Text for no popup).
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Text	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebTextArea Object

The SiebTextArea object provides events and properties that allow you to manipulate a text area in a test automation environment.

Parent

The SiebTextArea object is a child of the SiebApplet Object and the SiebList Object.

Type

The SiebTextArea object is one of the Core Control Objects.

Events

The SiebTextArea object has the following event.

Event Name	Description
SetText (*TextValue*)	Enters text into the text area. *TextValue* is a String that specifies the text to enter.

Methods

There are no methods available from the SiebTextArea object.

Properties

The SiebTextArea object has the following properties.

Property Name	Type	Description
ClassName = "SiebTextArea"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsEnabled	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
IsRequired	Boolean	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Text	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebThreadbar Object

The SiebThreadbar object provides methods and properties that allow you to manipulate a threadbar in a test automation environment.

Parent

The SiebThreadbar object is a child of the SiebScreen Object.

Type

The SiebThreadbar object is a singleton multivalue object that is one of the Navigation Objects.

Events

The SiebThreadbar object has the following event.

Event Name	Description
Goto (*LinkName*)	Clicks a link in the threadbar object. *LinkName* is a String that specifies the name of the link.

Methods

The following methods are available from the SiebThreadbar object:

- GetThreadItemByIndex Method
- IsExists Method

For a description of these methods, see "SiebThreadbar Methods" on page 135.

Properties

The SiebThreadbar object has the following properties.

Property Name	Type	Description
ActiveThreadItem	String	A data value representing the right-most thread link in the threadbar (the link to the page immediately preceding the page that is rendered on the screen).
ClassName = "SiebThreadbar"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Count	Integer	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName = "SiebThreadbar"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
ThreadItems	String	A data value representing the entire threadbar. Multiple items on the threadbar are separated by a | (pipe) character.
UIName = "Threadbar"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebThreadbar Methods

This section provides descriptions of the methods available from the SiebThreadbar Object.

GetThreadItemByIndex Method

The GetThreadItemByIndex method returns the text of the thread item that has the specified index.

Syntax

GetThreadItemByIndex (*Index*)

Argument	Description
Index	An Integer that indicates the index of the thread item in the Count property.

Returns

A String indicating the visible text of the thread item.

IsExists Method

For a description of the IsExists Method, see "Common Test Automation Object Methods" on page 144.

SiebToolbar Object

The SiebToolbar object provides methods and properties that allow you to manipulate a toolbar in a test automation environment.

Parent

The SiebToolbar object is a child of the SiebApplication Object.

Type

The SiebToolbar object is a collection object that is one of the System Objects.

Events

The SiebToolbar object has the following event.

Event Name	Description
Click (*CtrlName*)	Clicks a toolbar item. *CtrlName* is a String that specifies the RepositoryName of the toolbar item.

Methods

The following methods are available from the SiebToolbar object:

- IsControlEnabled Method
- IsControlExists Method
- GetRepositoryName Method
- GetRepositoryNameByIndex Method
- GetUIName Method

For a description of these methods, see "SiebToolbar Methods" on page 136.

Properties

The SiebToolbar object has the following properties.

Property Name	Type	Description
ClassName = "SiebToolbar"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Count	Integer	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property. For the SiebToolbar object, UIName is set to the same value as RepositoryName, because there is no visible display value for a toolbar object.

SiebToolbar Methods

This section provides descriptions of the methods available from the SiebToolbar Object.

IsControlEnabled Method

The IsControlEnabled method returns a Boolean value indicating whether or not the specified control is enabled on the toolbar.

Syntax

IsControlEnabled (*RepName*)

Argument	Description
RepName	A String that indicates the RepositoryName of the object

Returns

A Boolean value indicating whether the specified toolbar control is enabled (TRUE) or disabled (FALSE).

IsControlExists Method

The IsControlExists method returns a Boolean value indicating whether or not the specified control exists on the toolbar.

Syntax

IsControlExists (*RepName*)

Argument	Description
RepName	A String that indicates the RepositoryName of the object

Returns

A Boolean value indicating whether the specified toolbar control exists (TRUE) or does not exist (FALSE).

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

GetUIName Method

For a description of the GetUIName Method, see "Common Test Automation Object Methods" on page 144.

SiebTree Object

The SiebTree object provides methods and properties that allow you to manipulate a tree view object in a test automation environment.

Parent

The SiebTree object is a child of the SiebApplet Object.

Type

The SiebTree object is a singleton multivalue object that is one of the Complex Control Objects.

Events

The SiebTree object has the following events.

Event Name	Description
Collapse (*Position*)	Collapses a node in the tree view. *Position* is a String that specifies the position of the node in the tree.
Expand (*Position*)	Expands a node in the tree view. *Position* is a String that specifies the position of the node in the tree.
NextPage	Scrolls the tree view to the next page.
PreviousPage	Scrolls the tree view to the previous page.
Select (*Position*)	Selects a node in the tree view. *Position* is a String that specifies the position of the node in the tree.

NOTE: The *Position* parameter of the Collapse, Expand, and Select events is a String that indicates the position of the node in the tree. It is in the format

first-level-position.*second-level-position*.*third-level-position*

where each position is in relation to its current context within the preceding level. For example, 1.3.2 represents the second node within the third node of the first root node.

Methods

The following methods are available from the SiebTree object:

- GetChildCount Method
- GetTreeItemName Method
- IsExpanded Method
- IsExists Method

For a description of these methods, see "SiebTree Methods" on page 139.

Properties

The SiebTree object has the following properties.

Property Name	Type	Description
ActiveTreeItem	String	A data value representing the current tree item.
ClassName = "SiebTree"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName = "SiebTree"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName = "Tree"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebTree Methods

This section provides descriptions of the methods available from the SiebTree Object.

GetChildCount Method

The GetChildCount method finds the tree item at the specified position and returns the number of child nodes of the tree item that are displayed on the current page.

Syntax

GetChildCount (*Position*)

Argument	Description
Position	A String that indicates the position of the node in the tree. It is in the format *first-level-position*.*second-level-position*.*third-level-position* where each position is in relation to its current context within the preceding level. For example, 1.3.2 represents the second node within the third node of the first root node.

Returns

An Integer indicating the number of child nodes of the specified tree item that are displayed on the current page. For lists that span multiple pages, the parent node is displayed on each page.

GetTreeItemName Method

The GetTreeItemName method finds the tree item at the specified position and returns its name.

Syntax

GetTreeItemName (*Position*)

Argument	Description
Position	A String that indicates the position of the node in the tree. It is in the format *first-level-position*.*second-level-position*.*third-level-position* where each position is in relation to its current context within the preceding level. For example, 1.3.2 represents the second node within the third node of the first root node.

Returns

A String indicating the visible text of the specified tree item.

IsExpanded Method

The IsExpanded method returns a Boolean value indicating whether or not the specified tree node is expanded.

Syntax

IsExpanded (*Position*)

Argument	Description
Position	A String that indicates the position of the node in the tree. It is in the format *first-level-position*.*second-level-position*.*third-level-position* where each position is in relation to its current context within the preceding level. For example, 1.3.2 represents the second node within the third node of the first root node.

Returns

A Boolean value indicating whether the specified tree node is expanded (TRUE) or collapsed (FALSE).

IsExists Method

For a description of the IsExists Method, see "Common Test Automation Object Methods" on page 144.

SiebView Object

The SiebView object provides methods and properties that allow you to manipulate a view object in a test automation environment.

Parent

The SiebView object is a child of the SiebScreen Object.

Type

The SiebView object is a container object that is one of the Application Hierarchy Objects.

Events

The SiebView object has no associated events.

Methods

The following methods are available from the SiebView object:

- GetClassCount Method
- GetRepositoryName Method
- GetRepositoryNameByIndex Method

For a description of these methods, see "SiebView Methods" on page 141.

Properties

The SiebView object has the following properties.

Property Name	Type	Description
ActiveApplet	String	Indicates the RepositoryName of the active Applet object.
AppletCount	Integer	Indicates the number of child applets of the view.
ClassName = "SiebView"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName	String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebView Methods

This section provides descriptions of the methods available from the SiebView Object.

GetClassCount Method

For a description of the GetClassCount Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

SiebViewApplets Object

The SiebViewApplets object provides methods and properties that allow you to manipulate a view applet in a test automation environment.

Parent

The SiebViewApplets object is a child of the SiebView Object.

Type

The SiebViewApplets object is a singleton collection object that is one of the Navigation Objects.

Events

The SiebViewApplets object has the following event.

Event Name	Description
Select (AppletName)	Selects an applet. AppletName is a String that specifies the RepositoryName of the applet to select.

Methods

The following methods are available from the SiebViewApplets object:

- GetRepositoryName Method
- GetRepositoryNameByIndex Method
- GetUIName Method
- IsExists Method

For a description of these methods, see "SiebViewApplets Methods" on page 143.

Properties

The SiebViewApplets object has the following properties.

Property Name	Type	Description
ActiveApplet	String	Indicates the RepositoryName of the active Applet object.
ClassName = "SiebViewApplets"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
Count	Integer	See "Common Test Automation Object Properties" on page 144 for a description of this property.
RepositoryName = "SiebViewApplets"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.
UIName = "ViewApplets"	Const String	See "Common Test Automation Object Properties" on page 144 for a description of this property.

SiebViewApplets Methods

This section provides descriptions of the methods available from the SiebViewApplets Object.

GetRepositoryName Method

For a description of the GetRepositoryName Method, see "Common Test Automation Object Methods" on page 144.

GetRepositoryNameByIndex Method

For a description of the GetRepositoryNameByIndex Method, see "Common Test Automation Object Methods" on page 144.

GetUIName Method

For a description of the GetUIName Method, see "Common Test Automation Object Methods" on page 144.

IsExists Method

For a description of the IsExists Method, see "Common Test Automation Object Methods" on page 144.

Common Test Automation Object Properties

The following table provides descriptions of common properties available from multiple test automation objects.

Property Name	Type	Description
ClassName	String, Const String	The name of the class of the test automation object.
Count	Integer	The number of objects of a given type that are present in the current context.
IsEnabled	Boolean	A Boolean value that indicates whether or not the object is enabled.
IsOpen	Boolean	A Boolean value that indicates whether or not the object is open. This property is used in objects that have a popup applet.
IsRequired	Boolean	A Boolean value that indicates whether or not the object is required.
RepositoryName	String, Const String	The name of the object as it is stored in the repository of the test automation tool.
Text	String	The text value of the object.
UIName	String, Const String	The name of the object as it is displayed in the user interface.

Common Test Automation Object Methods

This section provides descriptions of common methods available from multiple test automation objects. The methods described in this section include:

- GetClassCount Method
- GetRepositoryName Method
- GetRepositoryNameByIndex Method
- GetUIName Method
- IsExists Method

GetClassCount Method

The GetClassCount method searches the repository for objects of the specified type and returns an Integer indicating the number of such objects.

Available from

SiebApplet Object, SiebApplication Object, SiebCurrency Object, SiebScreen Object, SiebTask Object, SiebTaskStep Object, SiebTaskUIPane Object, SiebView Object

Syntax

GetClassCount (ClassName)

Argument	Description
ClassName	A String that specifies the type of object to be counted

Returns

Integer indicating the number of objects of the specified type.

GetRepositoryName Method

The GetRepositoryName method finds the object with the specified parameters, and returns the object's RepositoryName.

Available from

SiebApplet Object, SiebApplication Object, SiebCurrency Object, SiebMenu Object, SiebPageTabs Object, SiebScreen Object, SiebScreenViews Object, SiebTask Object, SiebTaskStep Object, SiebToolbar Object, SiebView Object, SiebViewApplets Object

Syntax A (SiebApplet Object, SiebApplication Object, SiebCurrency Object, SiebScreen Object, SiebTask Object, SiebTaskStep Object, SiebView Object)

GetRepositoryName (*ClassName*, *UIName*)

Argument	Description
ClassName	A String that indicates the ClassName of the object
UIName	A String that indicates the UIName of the object

Syntax B (SiebMenu Object, SiebToolbar Object, SiebViewApplets Object)

GetRepositoryName (*UIName*)

Argument	Description
UIName	A String that indicates the UIName of the object. For the SiebMenu object, submenu items are delimited by // (two slashes).

Syntax C (SiebPageTabs Object)

GetRepositoryName (*UIName*, [*NavType*])

Argument	Description
UIName	A String that indicates the UIName of the PageTab or View object.
NavType	(Optional) A String that indicates the type (Screen or View) of navigation object. When *NavType* is defined as Screen, then UIName should be specified as the display name for a PageTab object. When *NavType* is defined as View, then UIName should be specified as the display name for a View as represented by an Aggregate Category link. If the *NavType* argument is not defined the default value of Screen is used.

Syntax D (SiebScreenViews Object)

GetRepositoryName (*UIName*, *Level*)

Argument	Description
UIName	A String that indicates the UIName of the object.
Level	A String that indicates the level (L2, L3, or L4) of the object.

Returns

A String indicating the RepositoryName of the object.

GetRepositoryNameByIndex Method

The GetRepositoryNameByIndex method returns the RepositoryName of the specified object.

Available from

SiebApplet Object, SiebApplication Object, SiebCurrency Object, SiebMenu Object, SiebPageTabs Object, SiebScreen Object, SiebScreenViews Object, SiebTask Object, SiebTaskStep Object, SiebTaskUIPane Object, SiebToolbar Object, SiebView Object, SiebViewApplets Object

Syntax A (SiebApplet Object, SiebApplication Object, SiebCurrency Object, SiebScreen Object, SiebTask Object, SiebTaskStep Object, SiebTaskUIPane Object, SiebView Object)

GetRepositoryNameByIndex (*ClassName*, *ClassIndex*)

Argument	Description
ClassName	A String that indicates the ClassName of the object.
ClassIndex	An Integer that indicates the index of the object in the value returned by the GetClassCount method.

Syntax B (SiebMenu Object, SiebToolbar Object, SiebViewApplets Object)

GetRepositoryNameByIndex (*Index*)

Argument	Description
Index	An Integer that indicates the index in the Count property of the object.

Syntax C (SiebPageTabs Object)

GetRepositoryNameByIndex (*Index*, [*NavType*])

Argument	Description
Index	An Integer that indicates the index of the PageTab or View object in the corresponding Count property (ScreenCount or ViewCount).
NavType	(Optional) A String that indicates the type (Screen or View) of navigation object. When *NavType* is defined as Screen, then *Index* should be an index into the ScreenCount property. When *NavType* is defined as View, then *Index* should be an index into the ViewCount property. If the *NavType* argument is not defined the default value of Screen is used.

Syntax D (SiebScreenViews Object)

GetRepositoryNameByIndex (*Index*, *Level*)

Argument	Description
Index	An Integer that indicates the index of the object in the appropriate Count property (L2Count, L3Count, or L4Count).
Level	A String that indicates the level (L2, L3, or L4) of the object.

Returns

A String indicating the RepositoryName of the object.

GetUIName Method

The GetUIName method returns the UIName of the object that has the specified RepositoryName.

Available from

SiebMenu Object, SiebPageTabs Object, SiebScreenViews Object, SiebToolbar Object, SiebViewApplets Object

Syntax A (SiebMenu Object, SiebToolbar Object, SiebViewApplets Object)

GetUIName (*RepName*)

Argument	Description
RepName	A String that indicates the RepositoryName of the object.

Syntax B (SiebPageTabs Object)

GetUIName (*RepName*, [*NavType*])

Argument	Description
RepName	A String that indicates the RepositoryName of the PageTab or View object.
NavType	A String that indicates the type of navigation object (Screen for a PageTab object, or View for a View object).

Syntax C (SiebScreenViews Object)

GetUIName (*RepName*, *Level*)

Argument	Description
RepName	A String that indicates the RepositoryName of the object
Level	A String that indicates the level (L2, L3, or L4) of the object

Returns

A String indicating the UIName (display name in the user interface) of the object. For the SiebMenu object, submenu items are delimited by // (two slashes).

IsExists Method

The IsExists method returns a Boolean value indicating whether or not the specified object exists.

Available from

SiebMenu Object, SiebPageTabs Object, SiebPDQ Object, SiebPicklist Object, SiebThreadbar Object, SiebTree Object, SiebViewApplets Object

Syntax A (SiebMenu Object, SiebViewApplets Object)

IsExists (*RepName*)

Argument	Description
RepName	A String that indicates the RepositoryName of the object

Syntax B (SiebPageTabs Object)

IsExists (*RepName*, [*NavType*])

Argument	Description
RepName	A String that indicates the RepositoryName of the PageTab or View object
NavType	A String that indicates the type of navigation object (Screen for a PageTab object, or View for a View object)

Syntax C (SiebPDQ Object, SiebPicklist Object)

IsExists (*ItemName*)

Argument	Description
ItemName	A String that indicates the visible title of the PDQ or Picklist object

Syntax D (SiebThreadbar Object)

IsExists (*LinkName*)

Argument	Description
LinkName	A String that indicates the visible title of the threadbar item

Syntax E (SiebTree Object)

IsExists (*Position*)

Argument	Description
Position	A String that indicates the position of the node in the tree. It is in the format *first-level-position*.*second-level-position*.*third-level-position* where each position is in relation to its current context within the preceding level. For example, 1.3.2 represents the second node within the third node of the first root node.

Returns

A Boolean value indicating whether the specified object exists (TRUE) or does not exist (FALSE).

Standard Interactivity Functional Test Objects

The following list identifies the object types for standard-interactivity Siebel Business Applications from Oracle.

Object Types for Standard Interactivity Applications

SiebWebButton

SiebWebCalculator

SiebWebCalendar

SiebWebChartControl

SiebWebCheckBox

SiebWebColumnSortAsc

SiebWebColumnSortDes

SiebWebComboBox

SiebWebFile

SiebWebHieraListExpandCollapse

SiebWebHistoryBack

SiebWebHistoryDropdown

SiebWebHistoryForward

SiebWebJavaApplet

SiebWebLink

SiebWebListBox

SiebWebMailto

SiebWebMenu

SiebWebMenu

SiebWebMultiSelect

SiebWebPageTab

SiebWebPageTabScroll

SiebWebPassword

SiebWebPdq

SiebWebPopupButton

SiebWebRadioButton

SiebWebReportDropdown

SiebWebRowSelect

SiebWebScreenSubViewDropdown

SiebWebScreenSubViewTab

SiebWebScreenSubViewTabScroll

SiebWebScreenViewDropdown

SiebWebScreenViewTab

SiebWebScreenViewTabScroll

SiebWebSitemapAnchor

SiebWebSitemapScreen

SiebWebSitemapView

SiebWebText

SiebWebTextArea

SiebWebThreadbar

SiebWebToggleDropdown

SiebWebToggleTab

SiebWebToolbarItem

SiebWebTreeExpandCollapse

SiebWebTreeNode

SiebWebTreeScroll

SiebWebURL

Index