SIEBEL 7
eBusiness

TESTING SIEBEL *e*BUSINESS APPLICATIONS

VERSION 7.5

AUGUST 2003

12-FRKIM5

Siebel Systems, Inc., 2207 Bridgepointe Parkway, San Mateo, CA 94404

Contents

Chapter 3. Plan Testing Strategy

Chapter 4. Develop Tests

Chapter 5. Execute Siebel Functional Tests

Chapter 6. Execute Integration and Acceptance Tests

Chapter 7. Execute Performance Tests

Chapter 8. Improving the Testing Process

Index

Introduction

This guide introduces and describes processes and concepts of testing Siebel eBusiness Applications. It is intended to be a best practices guide for Siebel Systems customers currently deploying or planning to deploy Siebel 7. It does not describe specific features of the Siebel eBusiness Application 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 customer 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.
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 applications, possibly adding new functionality.
Siebel System Administrators	Administrators responsible for the whole system, including installing, maintaining, and upgrading Siebel applications.

Product Modules and Options

This Siebel Bookshelf contains descriptions of modules that are optional and for which you may not have purchased a license. Siebel's Sample Database also includes data related to these optional modules. As a result, your software implementation may differ from descriptions in this Bookshelf. To find out more about the modules your organization has purchased, see your corporate purchasing agent or your Siebel sales representative.

How This Guide Is Organized

This book describes the seven high-level processes for planning and executing testing activities for Siebel eBusiness Applications. These processes are based on best practices and proven test methodologies. This book should be used as a guide to identify what tests should be run, when to run tests, and who should be involved in the quality assurance process.

The first two chapters of this book provide an introduction to testing and the test processes. All readers are encouraged to read Chapter 2, "Testing Process," which describes the relationships between the seven high-level processes. The chapters that follow each describe a specific process in detail. In each of these chapters, a process diagram is presented to help readers 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 the book in its entirety to understand all testing processes.
- **Functional Testing.** A functional tester should focus on Chapters 3 through 7. These chapters discuss the process of defining, developing, and executing test cases.
- **Performance Testing.** A performance tester should focus on Chapters 3, 4, and 8. 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
- British Computer Society Specialist Interest Group in Software Testing
  http://www.bcs.org.uk
- 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/
- Internet/Software Quality Hotlist
  http://www.soft.com/Institute/HotList/index.html
- Mercury Interactive
  http://www-heva.mercuryinteractive.com/service_support/library/
- Software Testing Institute
  http://www.softwaretestinginstitute.com/index.html
- StickyMinds
  http://www.stickyminds.com/testing.asp
- STQE Magazine
  http://www.stqemagazine.com

Revision History

Testing Siebel eBusiness Applications Version 7.5

Overview of Testing Siebel Applications 1

This section provides an overview of the reasons for implementing testing in software development projects, and introduces a methodology for testing Siebel eBusiness applications with descriptions of the processes and types of testing that are used in this methodology.

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 impact 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 total cost of ownership.

Application Software Testing Methodology

The processes described in this book are based on common application software test definitions and the Siebel eRoadmap implementation methodology. These definitions and methodologies have been proven in customer engagement and remind us 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:

- **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 larger system.
- **Integration Testing.** Validates that all programs and interfaces external to the Siebel application function correctly. Sometimes adding a new module, application, or interface may negatively affect the functionality of another module.
- **Regression Testing.** Code additions or changes may unintentionally introduce unexpected errors or regressions that do not exist previously. Regression tests are executed when a new build or release is available to make sure existing and new features function correctly.
- **Interoperability Testing.** Applications that support multiple platforms or devices need to be tested to verify that every combination of device and platform works properly.
- **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.
- **System Testing.** System testing is a complete system test in a controlled environment. Both users and IT organization are involved to assess the system's readiness for general release.
- **User Acceptance Test (UAT).** Users test the complete, end-to-end business processes. Functional and performance requirements are verified to make sure there are no user task failures and no prohibitive system response times.
- **Performance Testing.** This test is usually performed using an automation tool to simulate user load while measuring system resources used. Client and server response times are both measured.
- **Stress Testing.** This test identifies the maximum load a given hardware configuration can handle. Test scenarios usually simulate expected peak loads.
- **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.

- **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.
- **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, and by displaying the appropriate warning messages, that the unit is operating correctly.

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 project. The Siebel implementation is comprised of activities logically grouped into eight distinct eRoadmap 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 eBusiness system.

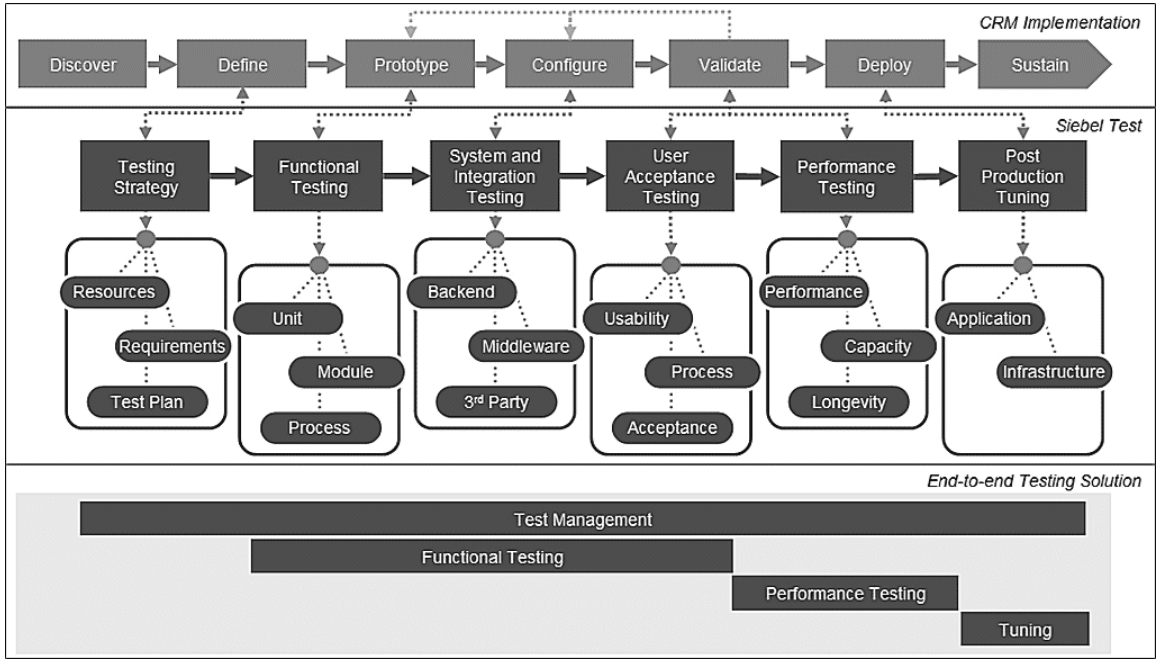


Figure 1. Siebel eRoadmap Implementation Methodology

Testing is an end-to-end process that begins when you begin to configure the Siebel application. The first stage is the development of a testing strategy by the testing team to define environmental simulation requirements and testing approaches, establish priorities, and define and create proper functional and load test scripts. The output of this is stage is a comprehensive Test Plan.

As mentioned earlier, functional testing begins once 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 using business processes and scenarios to verify whether or not all modules work together as required.

The next stage of functional testing, System and Integration testing, is validation 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.

Performance testing provides an assessment of whether or not an infrastructure will perform and scale to your requirements. 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 through the completion of a performance test. Next, complete a capacity test by adding users until you reach the number of users expected to use the system over the life of the application. Finally, execute the test over an extended period of time (longevity test) to determine durability of an application as well as capture any defects that become visible over time.

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

Phased Delivery

A phased delivery strategy which implements complex features over time in successive projects is preferred. This approach manages project risks by gradually adding functionality and users. It also provides an early opportunity to validate requirements and improve the deployment process. Project phasing is a general IT project concept, and not testing specific. The test plan, however, must consider the concepts of phased delivery.

Continuous Application Lifecycle

One deployment best practice is 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 per 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. Since 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

Continuous Application Lifecycle. 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.

Iterative Development

Another best practice is Iterative Development, in which functionality is delivered to the testing team based on a fixed build schedule. Functionality is scheduled to be delivered in specific builds and testing is scheduled according to the build strategy.

Iterative Development provides benefits similar to those of Continuous Application Lifecycle, by allowing a project to carefully manage risk and introduce new functionality in a controlled manner.

Best Practice

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

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, testing coverage and defect trends can be measured and 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 to 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.

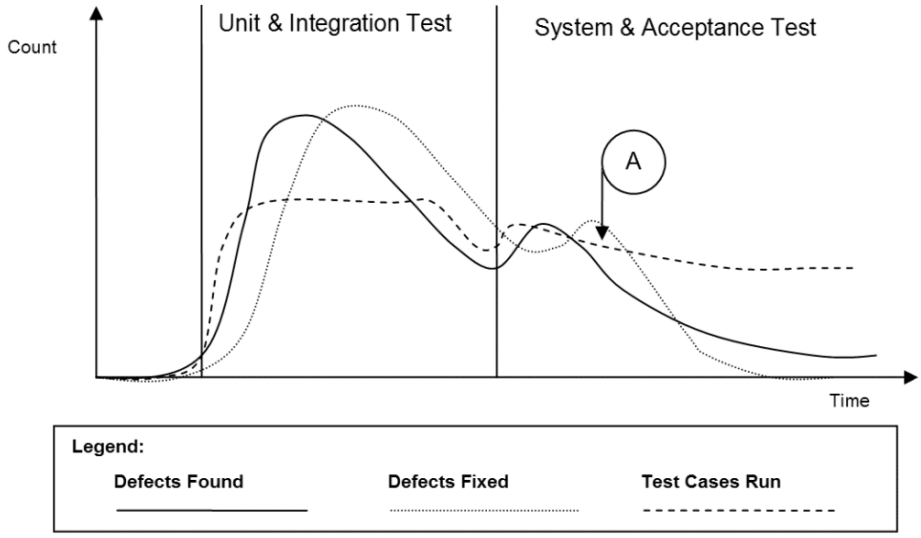


Figure 2. Trend Analysis of Testing and Defect Resolution

Testing is a key input to the deployment readiness decision, however it is not the only input to be considered. Testing metrics must be considered in conjunction with business conditions and organizational readiness.

Testing Process 2

This section describes the series of phases that are involved in the testing process. Figure 3 presents a high-level map of testing processes.

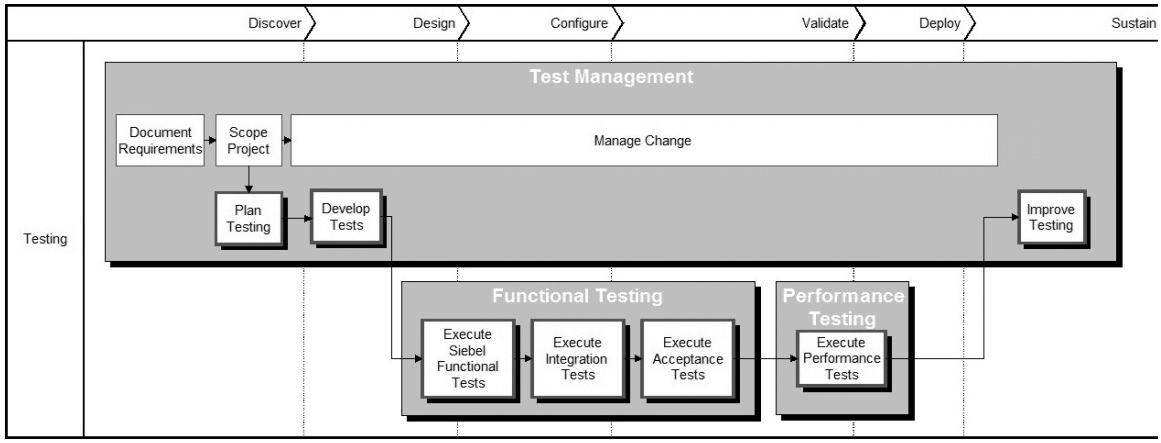


Figure 3. High-level Testing Process Map

As you can see, testing processes occur throughout the implementation lifecycle, and are closely linked to other configuration, deployment, and operations processes. Each of the seven testing processes described in this book are highlighted in bold in the diagram and are outlined briefly in the next section, "Overview of the Siebel Testing Process."

Overview of the Siebel Testing Process

This section provides a brief overview of each of the seven testing processes.

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.

Develop Tests

In the test development process, the test cases identified during the planning process are developed. 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 corner 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 eBusiness 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 Integration Tests

Integration testing verifies that the Siebel application, validated earlier, integrates with other applications and infrastructure in your system. Integration with various backend, 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 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 roll out.

Execute Performance Tests

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

Improve 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, Siebel Systems delivers regular maintenance and major software releases that may need to be applied. Both configuration changes and new software releases should be tested 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.

Plan Testing Strategy 3

This section describes the process of planning your tests.

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 test plan that is designed to achieve the objectives.
- **Test Plans.** Describes the tasks necessary to verify a particular test objective. For example, a performance test plan may have an objective of verifying that a given server configuration can support five hundred users. The test plan must document how this will be verified, covering what hardware and software are needed, how often to execute the test, who will develop the test case and execute it, and what data points or analysis output are required.
- **Test Cases.** A test plan is made up of 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 would typically perform 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.

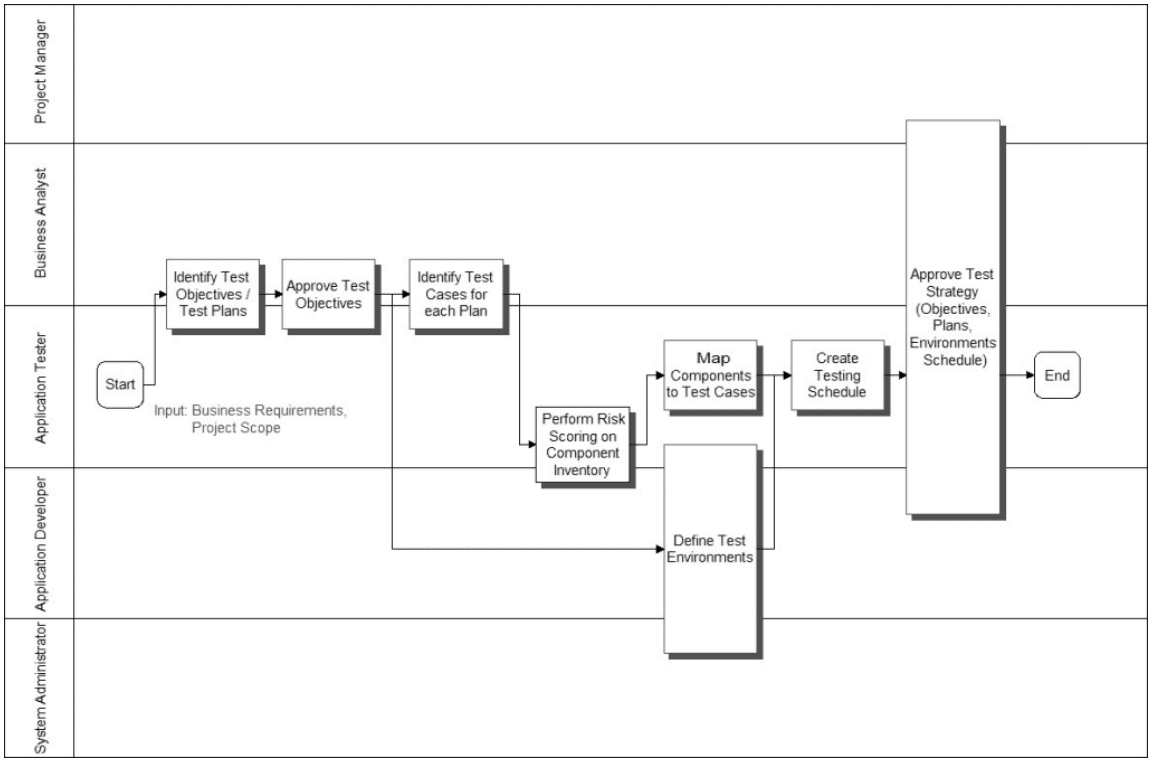


Figure 4. Diagram of the Plan Testing Strategy Process

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. This list of objectives is used 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 only available 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 KPI required.

The list of test objectives and their priority should be agreed to by the testing team, development team, and the business unit. Figure 5 shows a sample Test Objectives document.

For each test objective, there is a specified test plan that testers follow to verify or validate the stated objective. The details of the test plan are described in "Test Plans" on page 28.

OID	Priority	Type	Objective	Test Plan
O1	1	Functional Correctness	Validate support for Manage Quotes Business Process	Manage Quote
O2	1	Functional Correctness	Validate support for Manage Orders Business Process	Manage Order
O4	1	Service Level	Verify system support for 3050 concurrent sales callcenter users	System Load
O5	1	Functional Correctness	Validate that created orders are accepted in SAP	SAP Order Integration
O6	1	Functional Correctness	Validate product configuration and pricing rules	Product Configuration
O7	1	Authorization	Verify proper restrictions to application funcationality based on role	Responsibility Verification
O8	2	Service Level	Verify view paint response time < 2 seconds for commonly used views	System Response Time
O9	2	Usability	Verify novice user can create quote with 1 hour of training	Quote Usability
O10	2	Functional Correctness	Validate that customer information complies with Corporate Privacy policy	Privacy Compliance
O11	2	Service Level	Verify that batch processes can complete in 2 hour maintenance window	Batch Throughput
O12	2	Service Level	Verify 5 day component uptime	System Reliability
O13	2	Service Level	Verify quote validation < 5 seconds	Quote Performance
O14	2	Service Level	Verify 2 min outbound scripted call time	Call Script Performance
O15	3	Service Level	Verify that updates/patches can be applied in 1 day/quarter maintenace window	System Maintenance

Figure 5. Sample Test Objectives

Test Plans

The purpose of the test plan is three-fold. First, it provides a detailed plan for verifying a stated objective. Second, it anticipates issues associated with high-risk components. Third, it communicates schedule and progress toward a stated objective. The test plan has the following components:

- **Test Cases.** Detail level test scenarios. Each test plan is made up of a list of test cases, their relevant test phases, and relationship to components.
- **Component Inventory / Risk Assessment.** A list of components that need to be tested. 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.

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

Business Process Testing. 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 one can identify one or more test cases that need verification. Figure 6 shows a sample test case list for a few test plans. Each test plan contains one or more test cases, which are categorized by type.

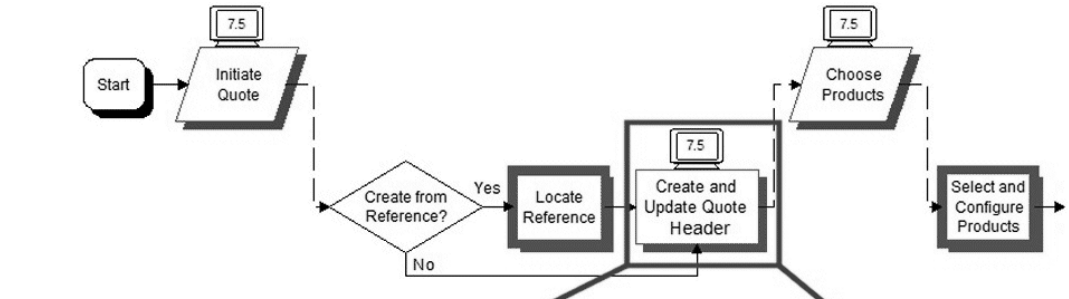
TID	Test Plan	Test Case	Type	Primary Phase	Description
T1	Manage Quote	Create Quote Header	Functional	Module	Create Quote header information
T2		Select Products	Functional	Module	Select single, multiple, complex products
T3		Configure Complex Services	Functional	Unit	Configure complex services
T4		Configure Complex Products	Functional	Unit	Configure complex products
T5		Invalid Product Configurations	Negative	Unit	Test handling of invalid product configurations
T6		Request Shipping	Functional	Integration	Test applciation of shipping information
T7		Price Products	Functional	Unit	Test pricing logic including discounting
T8		Quote Views	Structural	Unit	Verify the behaviors of the Quote views
T9		Process Quote	Functional	System	Manage quote business process test
T10	Manage Order	Create Order Header	Functional	Module	Create order header information
T11		Order Views	Structural	Unit	Verify the behaviors of the Order views
T12		Send Order	Functional	Integration	Test integration with SAP
T13		Process Order	Functional	Process	Manage order business process test
T13	System Load	3050 User Callcenter Load	Performance	Performance	Verify that the system can support 3050 concurrent call center users.

Figure 6. Sample Test Plan: Test Case List

There are three major types of test cases that may be specified: Functional, Structural, and Performance.

- **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. 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 can be supported by the system. 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. A given test case may be run several times in multiple testing phases, but typically the first phase in which it is run is considered the primary phase. The following describes how standard testing phases typically apply to Siebel eBusiness Application deployments.

- **Unit Test.** The objective of 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 32. In this phase of testing, in-depth verification of a single component is functionally and structurally tested.

  For example, during 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 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 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.

- **Integration Test.** In the Integration Test phase, the integration of the Siebel eBusiness Application with other backend, middleware, or third-party components are tested. This phase includes functional test cases and structural 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 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 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.** Component supports high business-impact business logic (for example, complex financial calculation).
- **Integration.** Component that integrates the Siebel application to an external or third-party system.
- **Scripting.** Component includes the coding of browser script, eScript, or VB script.
- **Ambiguous or Incomplete Design.** Component design is either ambiguous (for example, multiple implementation options described) or design is not fully specified.
- **Downstream Dependencies.** Component is required by several downstream components.

As shown in Figure 8 on page 32, 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, since the risk assessment provides guidance on the sequence and amount of testing required.

Best Practice

Risk Assessment. 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 three 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 roll out.** There is always risk associated with introducing configuration changes during the lifecycle of the project. For example, changing configuration just before 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, 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 and 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, please refer to Siebel SupportWeb at http://ebusiness.siebel.com/supportweb/.
- **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 system test environment reflects the production configuration, the more applicable the test results will be. It is important that the system test environment include 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 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 tuning activity can be accurately reflected in the system test and production environments.

- **Database Server.** If a database server needs to be scaled down, 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 system test include 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, since 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 generally available from third-party vendors.

Develop Tests 4

This section describes the process of developing the tests that you should perform during the development of your project.

Overview of Test Development

It is important for the development of test cases to be performed 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.

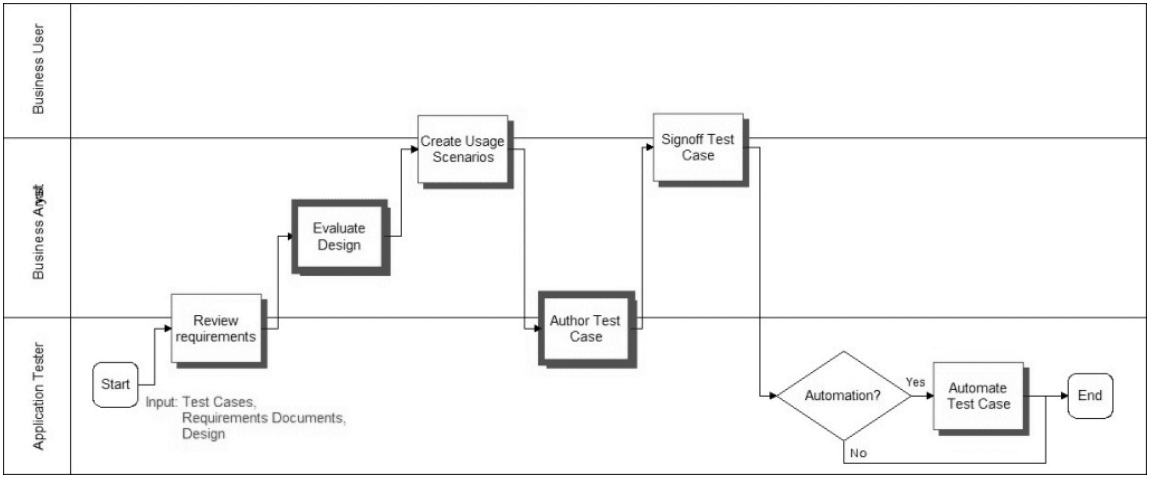


Figure 9. Diagram of the Develop Tests Process

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 to document its 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 and approved by both business analysts and the testing team.
- **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.

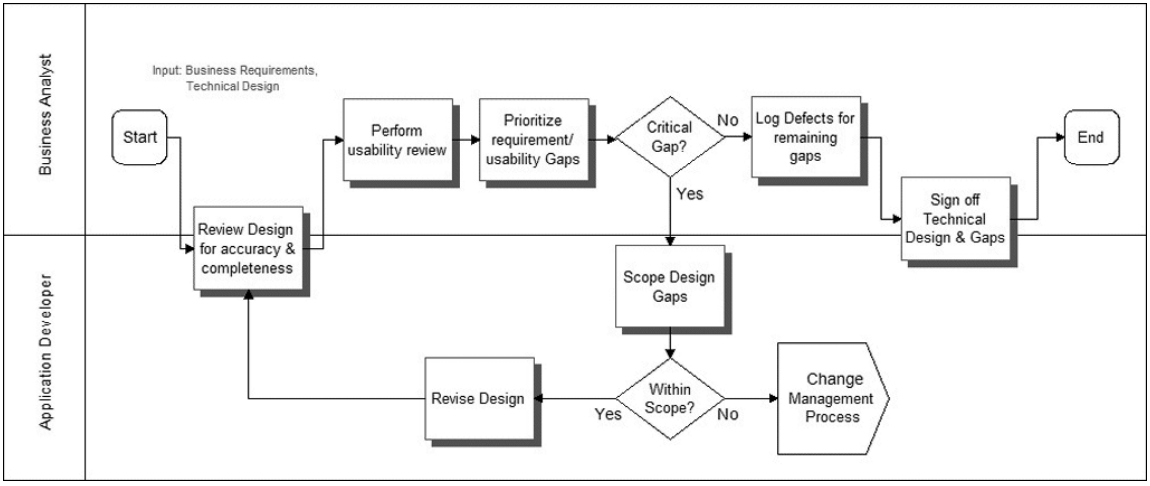


Figure 10. Diagram of the Evaluate Design Process

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.

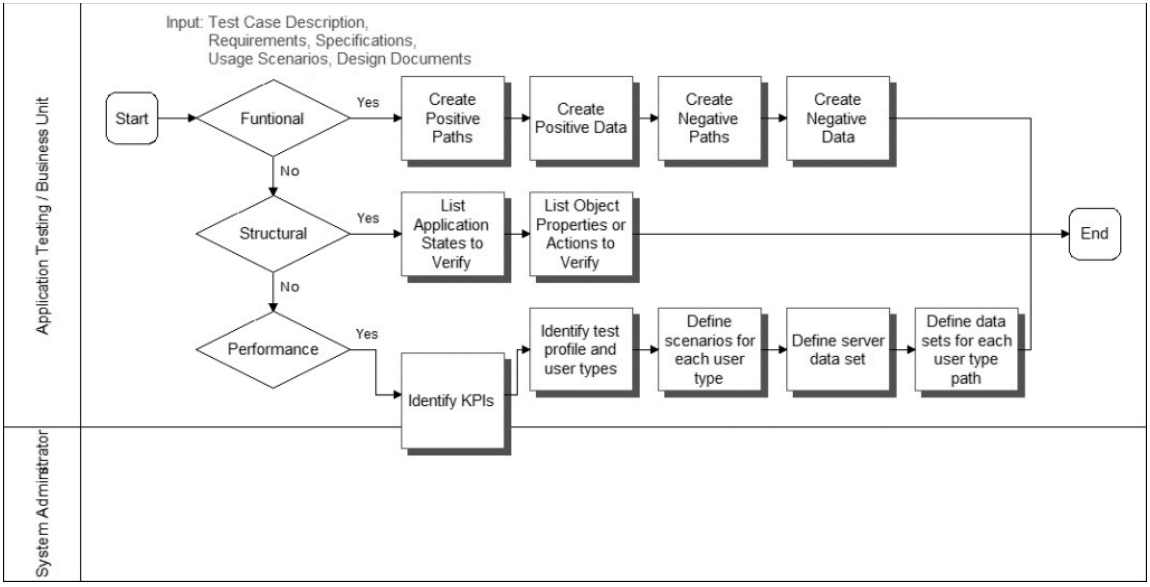


Figure 11. Diagram of the Test Authoring Process

As you can see from the process, functional, structural, 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 1 shows some examples of functional test cases.

Table 1. 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.

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 Path

Test paths describe the actions and objects to be tested. A path 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 path. Notice that in the Test Case Step column, there are no data values in the path; instead you see a parameter name in italics as a place holder. This parameterization approach is a common technique used with automation tools and is helpful for creating reusable test cases.

Test Case:	T1
Test Case Name:	Create Quote Header
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 Quote Header	Navigate to Quotes Screen	My Quotes View displays
2		Click the **New** button in Quote List applet	New Quote Line created
3		Enter *Name* into name field	Name value saved
4		Select Opportunity by searching on the *Account* field in picklist	Validate that Account, Currency, Price List fields are set according to account defaults
4.1			Validate that Effective Date, Status, Active Flag are defaulted to today's date or active.
4.2			Validate that the Effective Through date is set to 30 days in the future.
5		Save record using menu	

Figure 12. Sample Test Path

Test Data

Frequently, a single path can be used 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 separate from the test data.

Structural Test Cases

Structural test cases are typically used in unit or module test phases to make sure that a component or module is built to specification. Where functional tests focus on validating support for a scenario, structural tests make sure that the structure of the application is correct. Table 2 shows some examples of typical structural tests.

Table 2. Common Structural Test Cases

Object Type	Example
User Interface (View)	Verify that a view has all specified applets and each applet has specified controls with correct type and layout.
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.
Data Object	Verify that a data object has the specified data fields with correct data types.

Figure 13 shows a typical structural test case for a view. It is set up to verify that all components of the view (object) are built and function as specified.

Test Case: T8
Test Case Name: Quote Views
Test Case Description: Structural Test of Quote View
Application: Siebel Sales
Components under test: View Quote List View
Date Created: 5/2/2003
Created By: Joe Tester

Verification	Object	State	Property/Action	Expected Behavior
1	Quote List View		Applets	Quote List Applet, Quote Entry Applet
	Quote List Applet	Edit List	Columns	See Specification
			Buttons	New, Query, Browse Catalog, Revise, Get Advise (All Active)
			Menu	Verify (Active), Submit (Active), Update Opportunity (Inactive)
			Edit column fields	Check control type and data handling
		New	Buttons	Browse Catalog (Inactive)
			Menu	Verify (Active), Submit (Active), Update Opportunity (Inactive)
			Default Fields	Create date = today, Status = In Progress, Currency = USD
		Query	Buttons	Go, Cancel (all active)
			Menu	Verify (Inactive), Submit (Active), Update Opportunity (Inactive)
			Execute Query	Returns correct results
	Quote Entry Applet	Base	Fields	See Specification for field, layout, more, less, required
			Buttons	New (active), Query (active)
			Menu	No specialized
		New	Enter Quote Information	Applets takes input, defaults fields accordingly
		Query	Buttons	Go, Cancel (all active)
			Execute Query	Returns correct results

Figure 13. Sample Structural Test Case

Performance Test Cases

Performance testing is accomplished by simulating system activity using automated testing tools. Siebel Systems has several software partners who provide load testing tools that have been validated to integrate with Siebel 7. Automated load-testing tools are important since 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.

The first thing to document when authoring a performance test case are the key performance indicators (KPIs) that will be measured. 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 14 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 14. 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. 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 services 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, as 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 as 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 15. Sample Test Case Excerpt With Think Time

As shown in Figure 15, the user think times are specified in the scenario. It is important that think 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, since 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

Siebel Systems partners with the leading test automation tool vendors, who provide validated integrations to Siebel 7. 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. Once you have defined test cases, they can be automated using third-party tools.

Functional Automation

Using automation tools for functional or structural testing can cost less than performing manual test execution. You should consider which tests to automate since there is a cost to 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 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 7 Smart Web 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 Siebel Systems has worked closely with leading test vendors to provide their support for Siebel 7. Since 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:

1. **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 multi-iteration versus single-iteration distinction between user types. Many tools automatically record user think times; modify these values, if necessary, to make sure that the recorded values accurately reflect what was defined in the user type scenario.

2. **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. Since each virtual user runs in parallel, this is important for segmenting data and avoiding uniqueness constraints.

3. **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 7 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.

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

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

Best Practice

Test Automation. 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, as it provides an accurate way to simulate large numbers of users.

Execute Siebel Functional Tests 5

This section describes the process of executing Siebel functional tests.

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 backend 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 16.

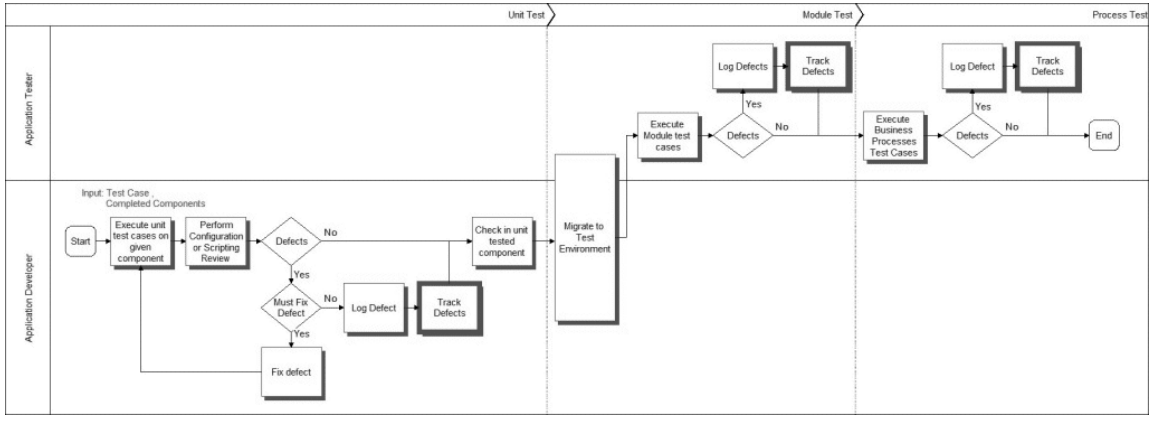


Figure 16. Diagram of the Execute Siebel Functional Tests Process

There are three phases to 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 done in 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 17, 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.

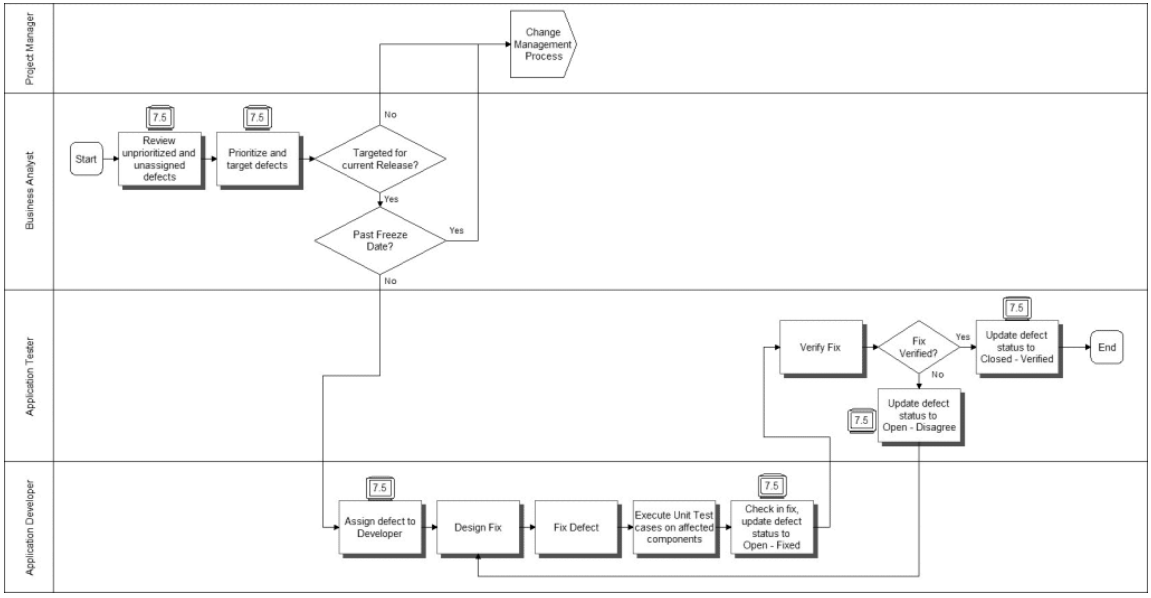


Figure 17. Diagram of the Track Defects subprocess

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 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

Track Defects. The use of a Defect Tracking System allows a project manager to understand the current quality of the application, prioritize defect fixes based on business impact, and carefully control risk associated with configuration changes late in the project.

Execute Integration and Acceptance Tests 6

This section describes the process of executing integration and acceptance tests.

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, since it marks a point where the system is nearing deployment.

The three major pieces to the executing integration and acceptance tests processes include:

- **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 Siebel 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 effectively use the system 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 18 illustrates this process.

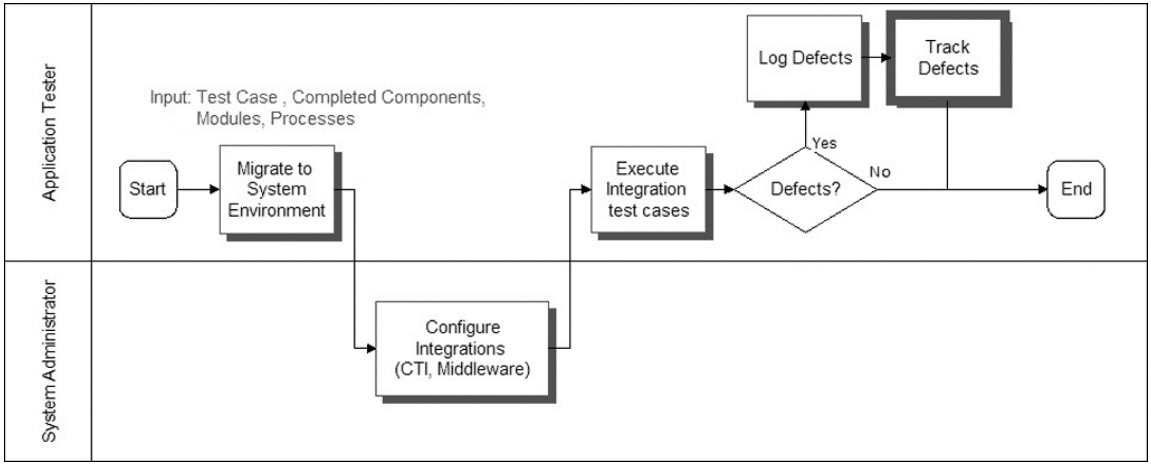


Figure 18. Execute Integration Tests Process

Execute Acceptance Tests

Once the system as a whole has been validated, you need to 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 19 illustrates this process.

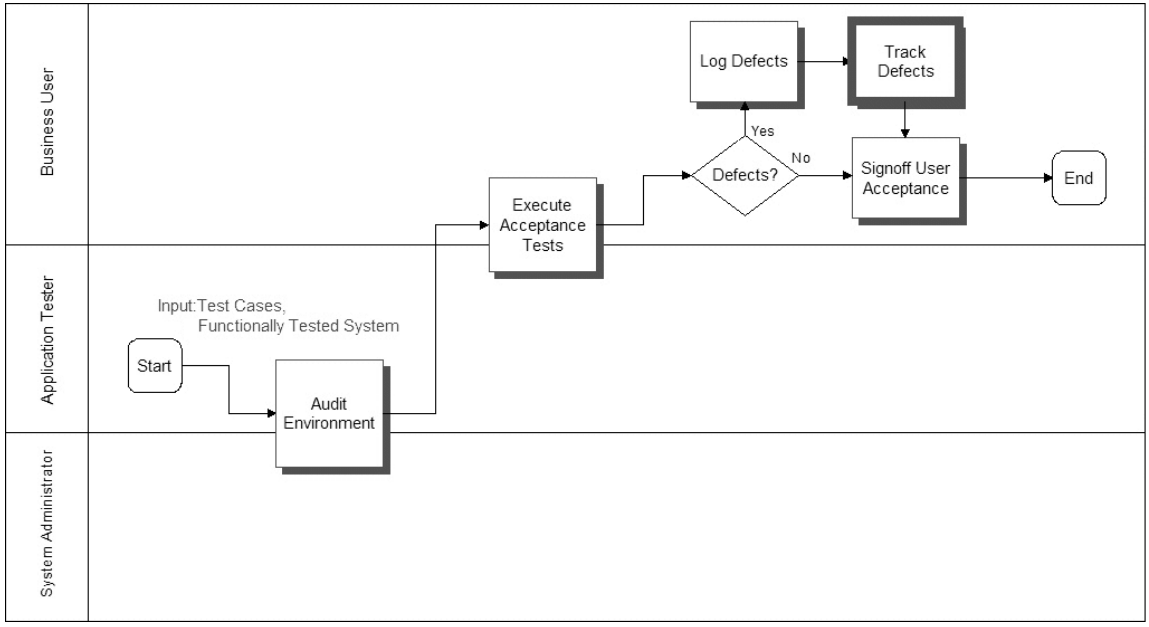


Figure 19. Diagram of the Execute Acceptance Tests Process

Execute Performance Tests 7

This section describes the process of executing performance tests.

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 since they are intended to measure different KPIs (key performance indicators). Performance tests are typically managed by specialized members of the testing and system administration organizations, who have ownership of the system architecture and infrastructure.

Figure 20 illustrates the process for performance test execution. The first step involves validating recorded user-type scripts in the system test environment.

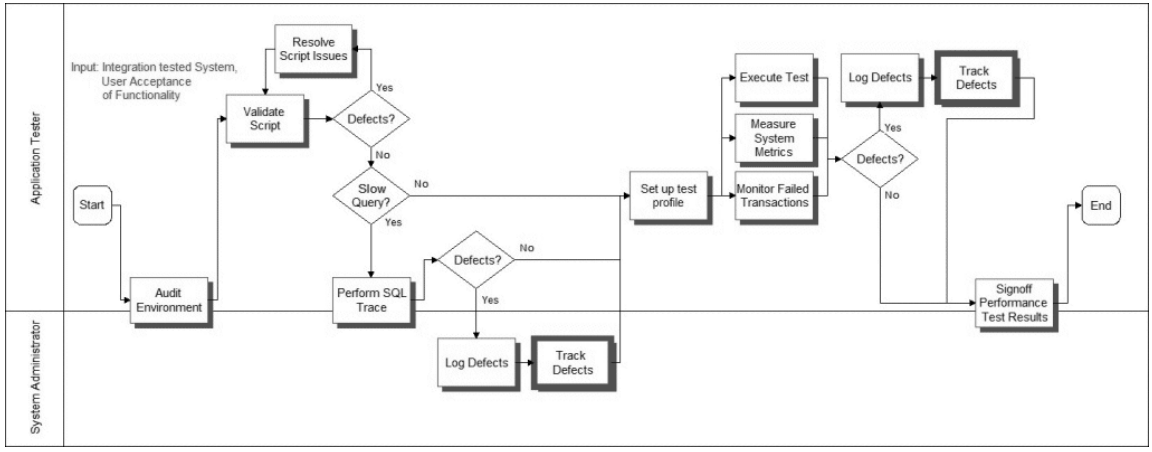


Figure 20. Diagram of the Execute Performance Tests Process

Execute Test

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 need to 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 should 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. Think 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 think times of expert and new users when compared to the average think times in the script.

Performing SQL Trace

Since many performance issues are caused by poorly formed SQL or sub optimal database tuning, the first step to improve performance is to perform SQL trace. SQL trace creates a log file that records the statements generated in the Siebel object manager and that are 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 over 0.1 seconds is considered suspect and needs to 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.

Measure 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 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.

Monitor 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. This can be caused by a resource issue at a server tier or by a long-running query or script in the application.

  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 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. Data should be fixed if error rates are significant, but a small number of failures do not generally affect results significantly.
- **Script Issues.** Transaction failures can also be caused by defects in scripts. 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 Improving the Testing Process

This section describes the steps you can take to make iterative improvements to the testing process, as illustrated in Figure 21 on page 70.

Improve Testing

After the initial deployment, regular configuration changes are delivered in new releases. In addition, Siebel Systems 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 "Phased Delivery" on page 17.

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.

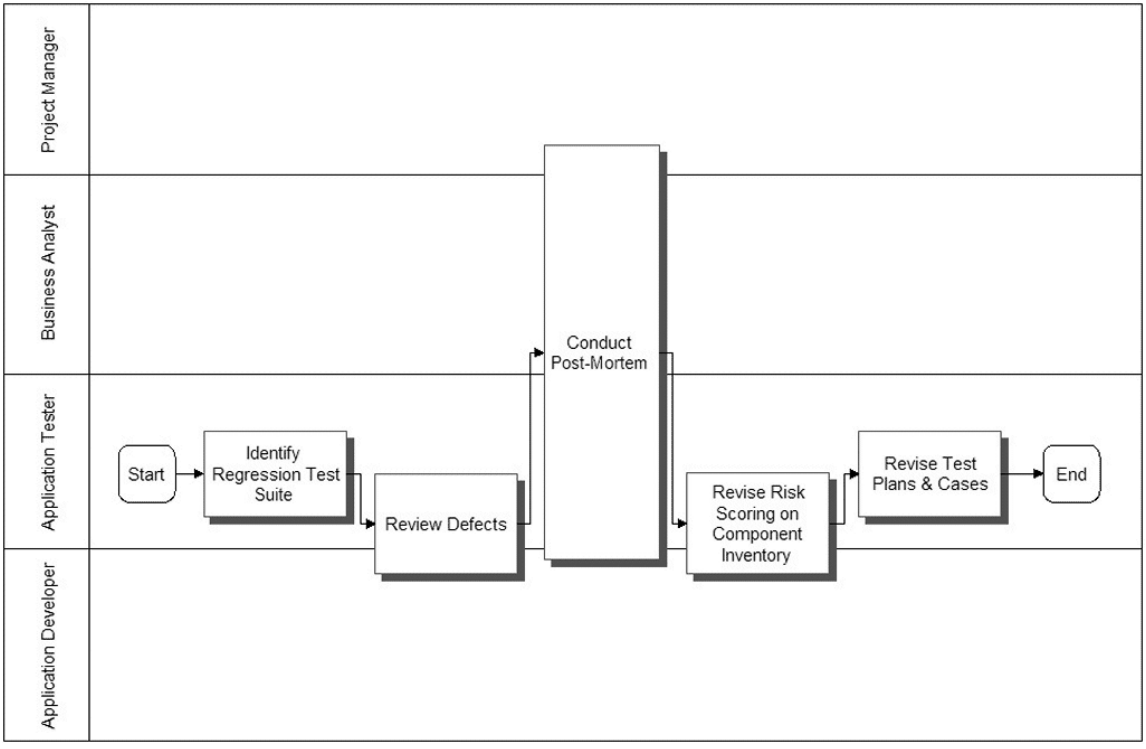


Figure 21. Diagram of the Improve Testing Process

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 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.

Index

No index is available for this guide.