AUTOMATED SOFTWARE TESTING PROJECT

IN GES32 DEVELOPMENT

by

Lixiu Huang

A thesis submitted to the Faculty and the Board of Trustees of the Colorado School of Mines in partial fulfillment of the requirements for the degree of Master of Science (Mathematical and Computer Sciences).

Golden, Colorado

Date 01/9/1999

Signed: Lixiu Huang
Lixiu Huang

Approved: Steven Pruess
Dr. Steven Pruess
Thesis Advisor

Golden, Colorado

Date 1/9/99

Graeme Fairweather
Dr. Graeme Fairweather

Graeme Fairweather
Dr. Graeme Fairweather
Professor and Head,
Department of Mathematical and
Computer Sciences

ABSTRACT

During the short history of software engineering, there has been a dramatic increase in the complexity of software. The importance of software testing is now recognized.

Software testing is to ensure the delivery of high quality software on time within budget. Testing, that includes both verification testing and validation testing, has a life cycle that parallels the software development cycle. Testing processes should, as far as possible, be integrated at the most effective points in the development life cycle.

In practice, experience from the software industry has shown that it is impossible to produce and release software without a single error. Two important components of modern software testing are automated testing and defect tracking.

In this thesis project, I joined a software development team and became involved in a real testing project of a software develop company, GeoGraphix, to help deliver a high quality product, GES32, on time. I developed two automated testing suites for GES32 acceptance testing and GES32 mainstream usage testing using Rational Visual Test 4.0 to help test GES32. The use of automated testing can save testing time, avoid human errors and create reproducible steps.

It is not enough to find a failure. A problem must be reported clearly to ensure that the problem is fixed. A defect tracking system PowerTrack has been used in the GES32 project to ensure that defects are fixed.

TABLE OF CONTENTS

LIST OF TABLES AND FIGURES

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Dr. Steven Pruess, for his enthusiasm, encouragement, and invaluable advice. Moreover, I must thank him for his tireless and thorough efforts in proof-reading drafts of this thesis. I also wish to thank the other members of my thesis committee, Dr. Willy Hereman and Dr. Hugh King, for their careful reading and most appreciated suggestions. Many thanks go to the faculty members of the Department of Mathematical and Computer Sciences, Colorado School of Mines, for their excellent courses which are invaluable to me now and forever.

I greatly appreciate Mr. Art Gust, Manager of the Quality Assurance, GeoGraphix, Incorporation, for providing me the opportunity to join his software testing team and allowing me to use the GES32 testing project as my thesis project.

With great pleasure, I acknowledge the vital role my family has played for their continual love, encouragement, and support. My parents, my brothers, my sisters have helped more than they can.

Finally, I would like to extend my deepest love to my lovely husband Wuning and son Michael, the constant sources of support and encouragement.

Chapter 1

INTRODUCTION

The success of software development depends on quickly and efficiently removing bugs and improving poor software performance. Software testing is to ensure the quality of software products, efficiency in producing software and the ability to accurately predict the cost and schedule of software delivery. Testing software has become increasingly complex and important.

In today's competitive software development market, software testing has become a critical hurdle that must be overcome before bringing products to market. With the limitation of testing time and resources, we never have enough time to test everything completely. We must implement the most cost-effective testing that will ensure that the product is reliable enough, safe enough and meets the user/customer's requirement. This is where automated testing can help. Although automation testing can not replace the human aspect of testing, automated tests are helpful in performing testing quickly and precisely.

The software industry has established that the longer defects remain in the software in the software process, the more expensive it is to remove them [2]. So we should start testing early. We can focus on detecting errors closer to the phase when they are introduced, in order to prevent the costly migration of errors downstream.

How well a bug is reported directly affects how likely the programmer is to fix it. A problem tracking system will provide efficiency in communication among the software development team.

1.1 Overview of Software Testing

In the early days of software development, testing was regarded as "debugging", or fixing the known bugs in software, and was usually performed by developers themselves. There were rarely any dedicated resources for testing. This period is called the *debugging-oriented period* (-1956).

In 1957 software testing was distinguished from debugging and became regarded as detecting bugs in software. The testing was an after-development activity, with the attitude of "to show ourselves that the product actually works". The underlying objective was to be convinced that it works and then ship it. The universities did not talk much about testing either. Computer science courses dealt with numerical methods and

algorithm development, but not with software engineering or testing. This period is *the demonstration-oriented period.*

By 1970's "software engineering" as a term was used more often. The first formal conference on testing was held at University of North Carolina in 1972, and a series of publications followed [6,7,8]. It was Myers who saw that the self-fulfilling nature of human goals had a major impact on testing work. He defined testing as "the process of executing a program with the intent of finding errors". He pointed out that if our goal is to show the absence of errors, we will discover very few of them. If our goal is to show the presence of errors, we will discover a large number of them. Establishing the proper goal and mind-set has a profound effect on testing success.

The 1970s were a major step in the development of a satisfactory testing process, but in the real world of industry, testing continued to be at the top of the list of things that got dumped when schedules and budgets became tight. Testing was started too late in the software development process, and there was not enough time to do it properly when it did start. This period is called the *destruction-oriented period.*

In the 1980s "quality" became a battle cry. Software developers and testers started to get together to talk about software engineering. A standard testing process was established. Testing involved operation of a system or application under controlled conditions and evaluating the results (e.g., 'if the user is in interface A of the application while using hardware B, and does C, then D should happen'). The controlled conditions

should include both normal and abnormal conditions. Testing should intentionally attempt to make things go wrong to determine if things happen when they shouldn't or things don't happen when they should. It is oriented to 'detection'. This period is called the *evaluation-oriented period* (1983 - 1987)

In the 1990s, software testing involves the entire software development PROCESS - monitoring and improving the process, making sure that any agreed-upon standards and procedures are followed, and ensuring that problems are found and dealt with.Testing tools are available to help with the testing process. It is oriented to 'prevention', consequently, this period is called the *prevention-oriented period.*

The evolution of a software testing model is a result of the software development evolution. (Table 1.1)

Table 1.1 Evolution of software development and testing

	Before 1970s	1970s to 1980s	1990s
Software size	small	moderate to large	very large
Degree of software complexity	low	Medium	high
Development methods and standards	ad hoc	Moderate	sophisticated
Recognition of testing's importance	little	Some	significant
Test methods and standards	ad hoc	Primitive	emerging

1.2 Tester's Objective

Traditionally, testing is described as a process of verifying that the system does what it should and doesn't do what it shouldn't. The tester's objective is to show that the system works correctly. This description is not complete. In practice, we can't test a program completely. The domain of possible inputs is too large to test. There are too many possible paths through the program to test all of them in a reasonable amount of time. We can't test the system thoroughly enough to verify that it works correctly except for trivial systems. It also fosters an ineffective attitude, if a tester insists on showing that the program works correctly, he will be more likely to miss problems than if he wants and expects a program to fail. This goal may be easy to achieve but the result is that errors remain in the system.

A better testing definition is to discover errors in the system and verify that the system satisfies the specified requirement, and try to show the system does what it shouldn't or doesn't do what it should. The tester's objective is to find every conceivable fault or weakness in the software working system. A tester should find as many problems as possible, the more serious the problem found, the better. If a tester is thinking his task is to find problems, he will look harder for them than if he thinks his task is to verify that the system works correctly. The result is fewer errors left in the system.

With the "test to destroy" attitude, testers are destructive, but creative too. Testers add value to a product by discovering errors and getting them out in the open as soon as

possible, this saves developers from building products based on error-ridden sources, and ensures a quality product.

1.3 Testing Terminology

In order to discuss the general principles of software testing and to introduce software testing technologies, the following definitions of important commonly used testing terms will be used in the thesis.

- *Volume testing* is to determine whether the program can handle the required volumes of data, requests, etc.
- *Security testing* is to check if the program's security requirements can be subverted.
- *Usability (human factors) testing* is to identify those operations that will be difficult or inconvenient for users. Publications, facilities, and manual procedures need to be tested.
- *Performance usage testing* is to determine whether the program meets its performance requirements.
- *Resource usage testing* is to determine whether the program uses resources (memory, disk space, etc.) at levels that exceed requirements.
- *Configuration testing* is to determine whether the program operates properly when the software or hardware is configured in a required manner.

- *Compatibility/conversion testing* is to determine whether the compatibility objective of the program has been met and whether the conversion procedures work.
- *Installability testing* is to identify the ways in which the installation procedures lead to an incorrect result.
- *Recovery testing* is to determine whether the system or program meets its requirements for recovery after failure.
- *Reliability/availability testing* is to determine whether the system meets its reliability and availability requirements.
- *Freeform testing* refers to unplanned tests.
- *Accessibility testing* is to determine whether users can enter, navigate, and exit the program with relative case.
- *Comprehensibility testing* is to determine whether users understand the product structure, its help system, and documentation.
- *Validation* is the process of evaluating a system or component during or at the end of the development process to determine whether it satisfies the specified requirements. Validation testing is to determine if software meets the requirements as specified at all requirements definition phases, i.e., phases that define system requirements, software requirements, design requirements, and coding.
- *Verification* is a process of evaluating the products of a given software development activity to determine correctness and consistency with respect to the products and

standards provided as input to that activity. It is a testing activity that occurs at all life cycle phases using the inputs of a phase to evaluate the products of that phase.

- *Testing case* refers to a set of test inputs, execution conditions, and expected results developed for a particular objective. It is the smallest entity that is always executed as a unit, from beginning to end. A test case may perform any number of test scenarios (or subtests).
- *Test scenario/test* is a specific behavior of a feature of a program.
- *A test suite* is a group of test cases that test a product. On simple projects there may be only one test suite. On large projects there can be any number.
- *Unit testing* is a process of testing individual program modules in isolation. The developer often performs this.
- *Integration testing* refers to combining modules together so that a larger group of integrated software and hardware modules can be tested.
- *System testing* examines an integrated hardware and software system to verify that the system meets its specified requirements.
- *Final testing* is generally a selected subset of system test cases that formally demonstrates key functionality for final approval and is usually performed after the system is installed at the user's site.

- *Bottom-up testing* starts testing the modules at the bottom of the invocation hierarchy independently using test drivers. Then modules at the next higher level that call these modules are integrated and tested, and so on.
- *Top-down testing* starts at the highest-level module, with stubs replacing the modules it invokes. The next lower-level modules then replace these stubs, with new stubs being provided for the modules that these call, and so on.
- *Benchmark testing* checks the speed and performance of an application.
- *Acceptance testing* tests the broad areas of the product. It's a test that doesn't go too deeply but verifies key areas of concern that, if found to have problems, would cause the build to be rejected. It is especially used on a product that is in its early, unstable stages.
- *Black box testing* tests the program against the end user, external specifications. It is done without any internal knowledge of the product. The program is treated as a black box. You can't see into it.
- *Glass box/White box testing* is done by the programmer. The programmer uses her understanding and access to the source code to test the program. It is distinguished from the *black box testing*.

1.4 GeoGraphix and GES32

In this thesis project, I implemented and maintained automated testing on a 32 bit GeoGraphix Exploration System (GES32), a commercial product, developed by GeoGraphix, Inc.

GeoGraphix was founded in 1984 to deliver cost effective desktop computer systems to the oil and gas industry. Today GeoGraphix works with thousands of customers in 38 countries worldwide. Many of the employees at GeoGraphix are geoscientists, petrophysicists, landmen, and engineers. By working together with customers, GeoGraphix provides the finest quality software to lower finding costs, maximize recoverable reserves, and redefine the techniques used in exploration and production throughout the world.

The GES32 is an integrated database management, mapping, and analysis system on Microsoft Windows NT. It is tailored to the needs and requirements of today's multi-disciplined exploration and development companies in the petroleum industry.

The GES32 incorporates data integration, surface modeling, interpretation, and visual presentation components into a sophisticated and comprehensive desktop technical mapping system. The GES32 facilitates the management, analysis, mapping, and interpretation of many types of data including:

- Cartographic information.

- Subsurface well data.
- Geophysical data.
- Well logs.
- Production data.

A user can enter this data from a keyboard, digitizer, or import it from ASCII files. Additionally, graphic editing facilities permit a user to interactively add a new object to the map and develop cross sectional presentations. Flexibility in data import supports many file formats.

The core of the system is a relational database manager that allows all of this information to be integrated with a cartographic display. Additionally, a user can explicitly link the data to geographic entities that appear on a map. For example, when a user points at an object on the screen, she will immediately access all of the data that relates to that object.

The GES32 is composed of a number of discrete tools: LandNet, LeaseMap, WellBase, GeoAtlas, QueryBuilder, Xsection, etc. Each of them may be operated independently. When used in conjunction with one another, they provide a coherent and complete data management, mapping, and interpretation environment.

GES32 includes about 4000 source code files, totaling 30 MB. There are 15 programmers and 6 testers in the development team.

1.5 Project Overview

The objectives of the thesis project are to understand the software testing process in a software development company. I will build my 'test to break' attitude, an ability to take the point of view of the customer, a strong desire for quality, and an attention to detail. Maintaining a cooperative relationship with developers, and an ability to communicate with both technical (developers) and non-technical (customers, management) people, delivering a high quality product on time. I joined a software development team as a software testing engineer at GeoGraphix, a software development company, involved in a real software testing project, GES32. I created and maintained two automated testing suites. I run them every day to test the daily build of the GES32, analyze the test results and report any defect via the defect tracking system PowerTrack.

The structure of this thesis is as follows. Chapter 2 briefly discusses the software development life cycle and software testing in the development stages. Chapter 3 begins the discussion of testing tasks in the software testing. Chapter 4 discusses the roles of automated testing in the software testing procedure, and two automated testing suites, written in Visual Test 4.0, in the GES32 project. Chapter 5 demonstrates the use of PowerTrack, a defect tracking system, in the GES32 project. Chapter 6 concludes the thesis and points out possible areas that automated testing can help in the GES32 project. In the appendices, I attached a sample-testing plan, sample-testing cases, sample automated testing codes, and automated testing logs.

Chapter 2

TESTING IN SOFTWARE DEVELOPMENT CYCLE

All errors in software are costly. Undetected errors, as well as errors detected late in the development process, are the most expensive of all. The sooner a bug is found and fixed the cheaper (Figure 2.1). In other words, we especially need techniques to improve our defect detection capabilities during the early software life cycle phases. Testing should be started early, and be a continuous activity throughout the development cycle.

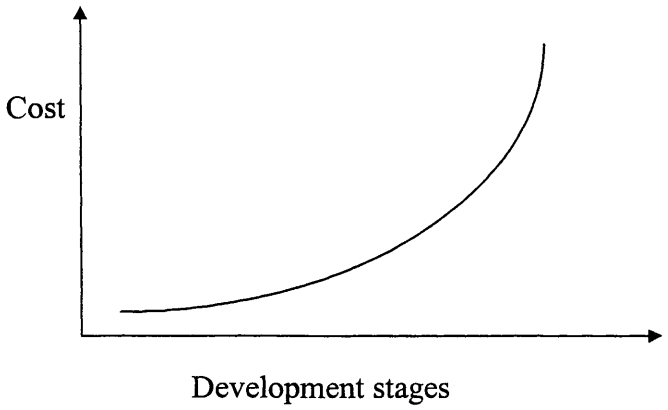


Figure 2.1 Cost of finding and fixing software errors

In this chapter, we discuss an approach to improving software testing that employs the testing in each stage of the development cycle. This approach shows key steps and activities that increase the probability of success in adopting software testing techniques and tools to ultimately improve software quality and overall development productivity. First we review the software development life cycle.

2.1 Software Development

In business, software development is usually done by a group of people working together. This group is called the development team. The team includes a program manager, who orchestrates the development process by managing Quality Assurance (QA), Development, and Technical Writing Leads. This is illustrated in (Figure 2.2):

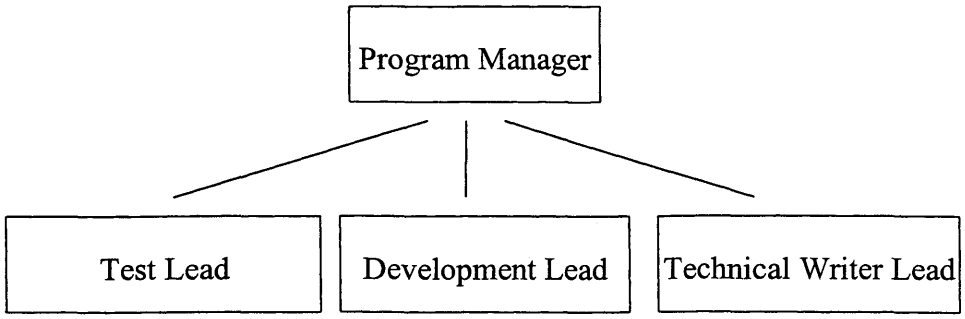


Figure 2.2 Software development team

To build a product, the process starts with the engineers telling the Product Manager what features they can get into the product based on the suggested timeline, and put together a functional specification that outlines the features and milestones of the product. Then, the quality assurance engineers come on the scene and begin designing the test plan for the features of the product. From there, the testing process parallels the entire development process. When programmers complete each feature, the test engineers will step through the test plan. The documentation team will also come about the same time as the quality assurance team and begin designing its approach to documenting the product.

2.2 Software Development Stages

Development models for software vary from company to company and depend on how long a firm has been in the software development business. These models are discussed at length in the literature [2]. Each model offers a different balance among the development tradeoffs; and this subject is beyond our interest. In general, software development goes through a cycle of five basic stages (Figure 2.3):

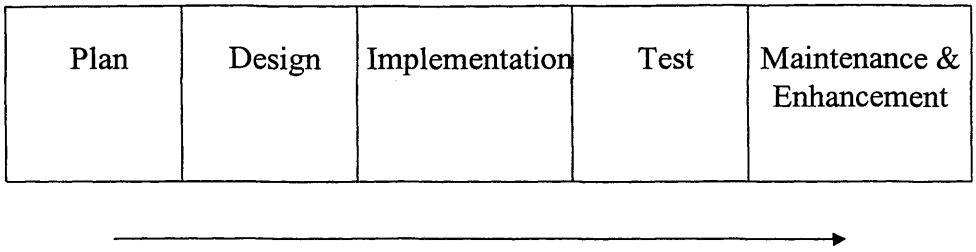


Figure 2.3 Typical software development stages in the life cycle

- Plan: Planners start by describing their vision of a product; it includes what it should do and why. The planners produce one or a few documents to guide future development. These documents may not be very detailed or specific. They may tentatively describe the user interface and goals for reliability or performance. They will probably state cost objectives. The planners cast most requirements in functional terms, and leave design and implementation details to the developer. A requirement is an objective that must be met.
- Design: Designers figure out how to provide the planned capabilities of the product in a requirements document. Designers specify a complete description of the user interface. It includes all screens and other output formats, the commands available, the command syntax. This is called the external design. It focuses on the user's point of view of the product. There is another type of design, internal design. It focuses on

the internal works of the program. The internal design will describe the structure of the product, data types and structures the code will work with. Prototyping is done to explore and evaluate the functionality and user interface of the system when needed.

- Implementation: Programmers write the programs according to the design. Programmers spend more time doing Glass box testing and debugging of a problem than they do implementing the program.
- Testing and Fixing: In this stage, coding is finished, and the program goes to the testing group for further testing. Testers find and report errors and get a new version for testing. The testing and fixing stage is named primarily for historical reasons. In practice, testing and fixing can be done at any stage in the development cycle. However, the cost of finding and fixing errors increases dramatically as development progresses. For example:
    - ⇒ At the design stage, finding a requirements document defect during its first review is inexpensive. It will cost more if the requirements change after code has been written: the code must be rewritten.
    - ⇒ At the coding stage, bug fixes are much cheaper when programmers find their own errors. There is no communication cost, and the error doesn't block or corrupt anyone else's work.
    - ⇒ Fixing an error before a product release is much cheaper than sending new disks, or even a technician, to each customer's site to fix it later.

- Post-Release Maintenance and Enhancement: Activities are performed on the product subsequent to its release for use. These activities include correcting errors in software, adapting the software to changing requirements and processing environments, enhancing system efficiency or functionality, and improving maintainability or reliability of software.

Table 2.1 Integration of the development and testing activities

Development Stage	Testing Activities
Plan	Requirements verification
Design	Functional design verification Internal design review External design review Competitors' product review
Code	Unit testing Initial informal testing Write test plan
Testing	Review and modify test plan Get approval of final test plan Acceptance testing User interface testing Mainstream testing Free form testing Regression tests on many software versions Pre-final testing Final testing Check manufactured masters
Maintenance	Reproduce custom reported defect Regression tests on developer fixed defect

In each development stage, testing activities can be involved. Table 2.1 is an illustration of testing activities in the development stages.

We see leading organizations establish software test life cycles that distinguish between test development, (setting test requirements, designing tests, and implementing or building test cases and procedures or scripts), test execution, and test evaluation [4].

2.3 Testing in the Planning Stage

In the planning stage, we do not test code. We test ideas. The "testers" include marketers, product managers, and senior designers. Members of the Testing group are rarely involved at this stage.

The testers read drafts of the planning documents. Then they gather data, using comparative product evaluations, focus groups, or task analysis. These are commonly described as planning and design tools, but they are also testing procedures: each can lead to a major overhaul of existing plans.

Kaner, Falk & Nguyen (1993) described that the "testers" should evaluate the requirements document (and the functional definition based on it) in terms of at least six issues [3]:

- Are these the "right" requirements? Is this the product that should be built?

- Are they complete? Does Release 1 need more functions? Can some of the listed requirements be dropped?
- Are they compatible? Requirements can be logically incompatible (i.e., contradictory) or psychologically incompatible. Some features spring from a different conceptualization of the product that if the user understands one of them, she probably won't understand the other(s).
- Are they achievable? Do they assume that the hardware works more quickly than it does? Do they require too much memory, too many I/O devices, and too fine a resolution of I/O devices?
- Are they reasonable? There are tradeoffs between development speed, development cost, product performance, reliability, and memory usage. Are these recognized?
- Are they testable? How easy will it be to tell whether the design documents match the requirements?

2.4 Testing in the Design Stage

In this stage, no code has been written. We are still testing ideas. These ideas are more formally expressed and more detailed than in the planning stage. The "testers" in this stage will be the same as the planning stage plus some people (such as a test group manager, technical supporter, etc.) who did not work on the design. The testing team

should find time to get familiar with the design, it will be valuable for writing testing planning later.

Examining the design documents, the testers should have a clear picture of how the system will work if it is built according to the design. In this stage, the testers should explore the following issues:

- Is the design possible? Are the hardware and software available to achieve the design?
- Is the design good? Will it lead to an efficient, compact, testable, maintainable, product?
- Does the design meet the requirements? Are the planning documents informal, changeable, and ambiguous?
- Is the design complete? Does it specify all relationships between modules, how they pass data, what happens in exceptional circumstances, what starting state should be assumed for each module, and how that state will be guaranteed?
- How well does the design cover error handling?

Three common types of review meetings, walk-through, inspection, and technical, are held to identify problems with the design.

In the walk-through meeting, the designer simulates the program step by step, shows what the program reaction is with test data supplied by the reviewers. This simulation will expose awkwardness, redundancy, and many missed details in design.

In each inspection meeting, designers will inspect every line of the design focused on a defined area, such as error handling, conformity with standard.

In the technical meeting, reviewers try to find out things that are ambiguous or confusing.

2.5 Testing in the Coding Stage

During the coding stage, the programmer who wrote the code does the initial testing, because this is a kind of testing that the programmer is especially well equipped to do during coding. It is called glass box testing (or white box testing).

In the glass box testing, the programmers use their understanding and access to the source code to develop test cases. This provides benefits:

- Focused testing: The programmer can test the program in pieces. She can write special test code that feeds interesting values to an isolated module, and report intermediate results obtained from the module. It is much easier to give an individual suspect module a thorough workout in glass box testing than in black box testing.
- Testing coverage: The programmer can also find out which parts of the program are exercised by any test. She can find out which lines of code, which branches, or which paths haven't yet been tested, and she can add tests that she knows will cover the areas not yet touched.

- Control flow: The programmers know what the program is supposed to do next, as a function of its current state. They can modify the program so that it constantly reports what it is doing, or use a special program called a debugger to run the program and track the order in which lines of code are executed.
- Data integrity: The programmer knows which parts of the program modify (or should modify) any item of data. By tracking a data item through the system, they can spot data manipulation by inappropriate modules. They might also write special code that calculates the value that a test variable should have at a given point in program, and compares this with the value that the variable actually has and reports an error.
- Internal boundaries: The programmer can see internal boundaries in the code that are completely invisible to the outside user.
- Algorithm-specific testing: Use a standard algorithm to predict the results. Then compare to the result from the program itself.

Another part of glass testing is performance testing. A programmer can use profilers to study the time the program spends on specific modules, along a specific path, or working with specific types of data. After the modules that execute most often or use the most computer time are identified, those modules are re-examined and re-coded to enhance the system.

A system is developed as a collection of processes and modules. An incremental testing strategy is used. Each piece is first tested separately. The testing of individual

pieces of a process or program is called module testing, unit testing, or element testing. Once the individual parts work, a few modules are tested together. This testing of combinations of pieces of a product is called integrated testing. As integrated testing proceeds, groups of modules are tested with other units, eventually, all of a process's modules are tested together.

It is easy to pin down the cause of an error by using incremental testing. Another benefit of incremental testing is that the programmer focuses on each module individually, which probably yields a better test coverage.

2.6 Testing in the Testing and Fixing Stage

When a programmer finishes coding and initial testing, the program goes to the Testing Group for further testing. Each time, testers receive a new version of the program, to check whether it is stable enough to be tested. This testing is called acceptance or qualification testing. The acceptance test is short with very limited functions.

After the system passes acceptance testing, the tester will test the program according to the test plan, trying to find and report errors and get a new version for testing. The tester feeds program input data, observes output data. The program is treated as a black box by the tester. This test is called black box testing.

The test planning is started as soon as the requirements document is circulated.

Regression testing is used during the testing and fixing stage to make sure that a fix works and doesn't disturb anything else. User interface testing and mainstream functions testing with mainstream data are tested in this stage. If some tasks are not possible to test at a certain time, a free form test is used. We will discuss it in more detail in next chapter.

2.7 Testing in the Maintenance Stage

In the maintenance stage, after the program is released, we may fix missed or pending errors, and make changes or enhancements requested by users. One of the tests we will do during the maintenance phase is to reproduce user reported defects. Most of the other testing we will do should be similar to what we did during function and system testing. Ideally, we already have a battery of regression testing series, so we can run them every time a program is changed.

Another type of testing in the maintenance stage is port testing. This happens when a program is modified to run on another operating system or computer. Here is a strategy for port testing:

- Overall functionality, running the regression testing series that is used during the testing stages.

- Keyboard handling, two computers with proprietary keyboards probably use them slightly differently, especially for the functional or combination keys.
- Terminal handling and interface style, the terminal of the new system may have incompatible characters with the terminal of the original computer. The graphical environment may be different too.
- Compatibility, if we use a third party's program in our system, make sure the third party's program is portable to the new computer and our ported system is compatible with the third party's program.

If we want to make a newer version of the system, we may go over all the production cycle again. At the same time we have to repeat all the testing procedures.

2.8 GES32 Automated Testing Motivation

In this thesis project, the GES32 automated testing, we will focus on the testing and fixing stage. Two automated testing suites, acceptance and mainstream usage, are created. The acceptance testing suite is used to check if a new build or version of the program is stable enough to be tested. The mainstream testing suite is used to test the main features of the program. We will discuss them in more detail in the section 4.3 and section 4.4.

Chapter 3

TESTING TASKS

Software test life cycles have three test phases: test plan (setting test requirements, designing tests, and implementing or building test cases and procedures or scripts), test execution, and test evaluation (defect report). There are generally five phases to testing a software program (Figure 3.1). Over the course of testing a product, we enter each of these phases. Although there is a general order (from top to bottom) in testing steps, we often go back to the earlier step to modify test cases or re-execute tests on a new version of program .

Effective testing is planned to ensure the most critical items are tested and not to waste limited testing resources and time on less important items.

3.1 Test Planning

One testing consultant states it as follows: "The act of designing tests is one of the most effective error prevention mechanisms known. The thought processes that must take

place to create useful tests can discover and eliminate problems at every stage of development." [12]

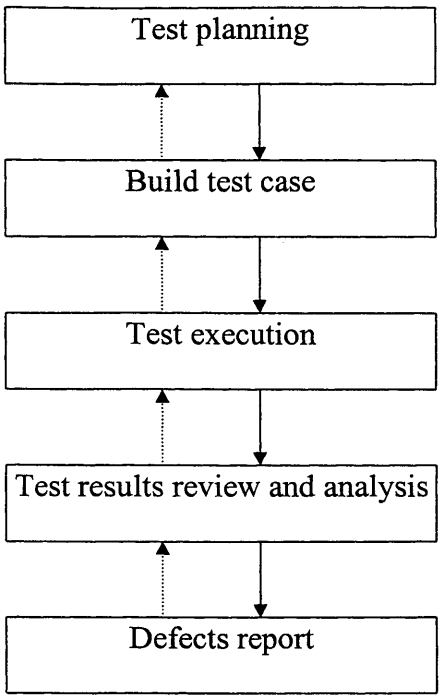


Figure 3.1 Testing workflow

The test plan is a valuable tool that can help managing a testing project and finding bugs. Good test documentation provides three major benefits:

- Facilitates the technical tasks of testing.
- Improves communication about testing tasks and process.
- Provides a structure for organizing, scheduling, and managing the test project.

Test planning is an on-going process. The evolutionary approach is used to develop a test plan. The test plan is small and in "draft" form, as it is produced during the design phase of the software development. It will be updated by inserting new sections or increasing detail in some area during the development phases and will be issued as a final version at the end of project.

When we begin to write a test plan for software, we should define the features or operational characteristics that make testing necessary. Test requirements are defined based upon a specific functional characteristic within the functional specification document of program product development. The test requirements include the functions and logic to be tested, program constraints, software states, input and output data conditions, and usage scenarios [7].

In the testing plan, there are key items:

- Purpose and Scope of the testing plan. This describes the purpose and scope of application testing. It also describes the approaches and goals for application testing and the strategy for fulfilling these.
- Applicable Documents for references, such as a project plan, the functional specification of the project.
- Resources required: Describing the necessary manpower, hardware, software, testing tools, tester education if needed.

- Test Management: Responsibilities for test team and application testing activities. Responsibilities of other groups. Error tracking system and testing result evaluation and review procedures. The criteria for beginning testing (Acceptance), and criteria for completion of testing activities.
- Acceptance tests: The criteria for beginning testing (Acceptance) by the testing group. This brief test will cover only the essential behavior of the program and to make sure the program is reasonably stable before the tester group spends resources on it. It is often a candidate for automated testing.
- Control flow.
- Mainstream usage tests. These include test cases that reflect the program under normal usage.
- Stress tests
- User interface tests
- Regression tests. A series of testing cases that should be run each testing cycle. We will discuss it in more detail in the section 3.4.
- Performance tests
- Release tests
- Schedules: List all milestones with dates, and when all resources will needed. This identifies Test Plans, deliverables, and Due Dates. It also identifies Dependencies, if any.

A sample of a test plan, GES32 QA Acceptance Testing, can be found in the Appendix A. It was created by QA manager, a tester coordinator, and a product manager from a GES32 test plan template and according to the functional specification and design documents. In the template, a lot of sections are common for all test plans, such as the testing management, defect fixing and management processes, and defect tracking processes. The tester coordinator creates a first draft of the plan. After the QA manager and product manager review the plan, a modified version of the plan is created.

3.2 Test Items Selection

When planning test procedures, we normally write a specification document outlining the specific command or menu options to select, data to enter, reactions and results we will get from the test procedure.

If we have time, we could develop billions or even trillions of different test cases . Clearly, time and resources will limit the testing we can do. We can only do a small portion of cases, a few hundred or a few thousand; consequently we must choose well.

An excellent test case satisfies the following criteria:

- It has a reasonable probability of catching an error. Keep in mind, our testing objective is to find errors. When searching for ideas for testing cases, try

working backward from an idea of how the program might fail. If the program could fail in this way, how could you catch it?

- It is not redundant. If two cases look for the same error, why run both and waste time?. If you expect the same result from two tests, you consider them equivalent. A group of tests forms an equivalence class if you believe that:
    - They all test the same thing.
    - If one test catches a bug, the others probably will too.
    - If one test doesn't catch a bug, the others won't either.

  If more than one test case involves the same input variables, or they result in similar operations in a program, or they will get the same output variables, you have reason to believe that they belong to the same class.

  In a group having similar test cases, one can be more effective than the others. You need to choose the best of the breed, the one most likely to find errors.
- It is neither too simple nor too complex. You can save testing time by combining two or more test cases into one test case. But don't create a monster that is too complicated to execute and understand or that takes too much time to create. It is often more efficient to run several simpler test cases.
- It makes program failures obvious. This is a major consideration. Testers miss many failures because they don't recognize a program that is staring them in the face. It is necessary to write down the expected output or result of each test

case as you create it. Refer to these notes while testing. Make any printout or files that you will have to inspect as short as possible. Don't let failures hide in a mass of boring print.

3.3 Test Preparation

Before a program is released to the testing group for testing, the testing group needs to develop/review test scenarios and test cases, request hardware, software, and testing tools that are needed for doing testing, and prepare the regression test library.

For automated testing, we design and prepare automated testing cases and suites and write initial testing scripts according to the design.

3.4 Test Execution

After the testing group receives the program from the development group, the testing cycle starts.

The first action is to run the acceptance tests according to the test plan.

After the program passes the acceptance testing, all or selected cases in the test plan will be executed.

During the first cycle or two, the program may be unstable. The testers should focus on the mainstream usage testing cases. Then try to go through as many of the test cases as possible.

A test execution includes the following tasks:

- Test case selection
- Pre-run setup
- Execution
- Recording activities, incidents, and results
- Post-run analysis
- Recreate problem
- Defect report

The key test execution deliverables are test logs, and a test defect (i.e. bug, problem) report.

Four test execution stages are commonly recognized: **unit testing, integration testing, system testing**, and **final testing**.

3.4.1 Usability Testing

Usability testing is to ensure that the software adapts to the user's actual work style. It focuses on accessibility, responsiveness, efficiency, comprehensibility. Usability testing is considered as a validation activity rather than a verification activity, because it

requires a real user to interact with the end product by executing it in either a simulated or a real form.

3.4.2 System Testing

System testing is the process of attempting to demonstrate that a program or system does not meet its original requirements and objectives as stated in the requirement specification. System testing is performed by the testing group before the product is made available to customers. It can begin whenever the product has sufficient functionality to execute some of the test cases or after unit and integration testing are completed. It can be conducted in parallel with function testing.

To run automated testing, the tests may not be completed in the first one or two testing cycles. We need to play them back several times on the same build to verify that they work as intended. Test scripts should be modified as needed. After all test cases are passed by the playing back of scripts, this establishes a baseline of expected behavior for the product. The baseline will be used to test any future build.

3.4.3 Regression Testing

Regression testing detects faults introduced during the modification of program. It verifies that modifications have not caused unintended effects and also verifies that a modified program meets its specified requirements.

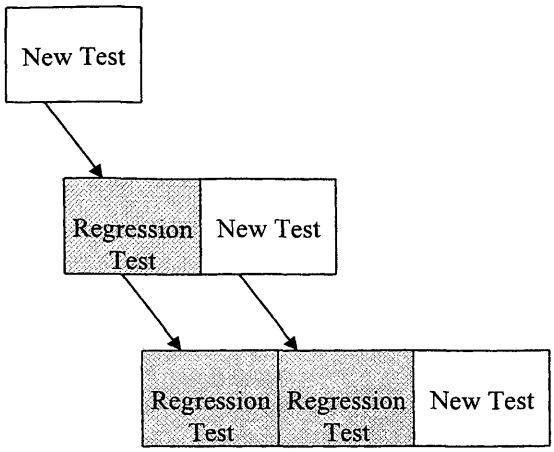


Figure 3.2 Incremental testing

Regression testing is a very important procedure. It is used in two places in testing. The first one is checking whether a bug fix works. When a tester reports a problem, she tells programmers what exactly she did to find it. Then programmers examine the code thoroughly, find the cause of the problem, and fix it. The fix may address only the symptoms the tester reported but does not solve the underlying problem. The fix may fix the problem the tester reported, but break other places in the code. Even worse, the programmers may give back code with the same bug plus whatever they broke when making their "fix". When a tester tests a bug fix, there are three objectives:

- Check that the bug was actually addressed. In order to test it, run exactly the same test process that found the bug.

- Try to find related bugs. Try to reach this bug in different ways, or try to find similar errors in other part of the code.
- Check the rest of the program. Try to find unwanted consequences or anything else.

The second place is when a new version of the program is released for testing, the tester needs to repeat every test case in a regression test library. The library might include all of the best-designed tests, such as the tests that found bugs But if it includes too many test cases, the tester will not have time to run any new test for the new version. The newest tests are the most likely to find new errors. Here are some tactics to manage the regression testing library.

- Dropping tests that are virtual repetitions of another in the library.
- Reducing the concentration of tests on a bug that has now been fixed. Once the bug is gone for several cycles, only keep a few of that mass of tests, get the rest out of the library. Add the tests to the library if a bug and variants of it persist across many cycles of testing.
- Combining test cases. Try to combine some test cases that the program is expected to pass into one big test case.
- Using an automated test for some test cases if it is possible. It will pay to spend time to automate the test if the testing scripts are used for five or more testing cycles.
- Designating some tests for periodic testing. Run some subsets of the tests every second or third cycle. Only try to run all of them during a potential final testing cycle.

There is a great amount of research focused on regression testing issues, such as test suite management, test environments, automation of the regression testing , and the selective retest problem [13,14,15,16,17].

3.5 Testing Result Analysis

When a testing procedure is finished, the testing results or testing log will be reviewed. When a test case fails, we evaluate the results and determine if it is an actual failure in the product or a deliberate change to a new version. If the failure is caused by the changes to the products, modify the test procedure or testing scripts as needed. If the failure is an actual defect of the product, try to repeat the failure and report the defect.

3.6 Problem Reporting and Tracking

The point of writing problem reports is to get bugs fixed. How well a bug is reported directly affects how likely the programmer is to fix it. If the bug reports are not clear and understandable, bugs won't get fixed. A problem tracking system is used to report bugs, file them, and write summary report about them. The key benefits of a good bug tracking system are improvements in communication and accountability that get more bugs fixed.

A fully effective defect report should:

- Explain how to reproduce the problem. Programmers will dismiss reports of the problem that they can't see for themselves. If they can't see it, they can't fix it. Hence write clear problem reports immediately at the time a problem is found.
- Describe the problem in a minimum number of steps. If the report contains unnecessary steps that may make the problem unclear and confuse the readers.
- Make sure the problem report is complete, easy to understand, and non-antagonistic.

The type of information requested on a problem report is:

- A unique problem report number.
- Module or function in which the problem occurred.
- Version identification that identifies the version of the code under test.
- Report type that describes the type of the problem found. Usually, the following report types are listed:
    - Coding errors that report unintended behavior.
    - Design issue that includes the program works as intended, but the reporter disagrees with the design.
    - Suggestions from reporters which include how to improve the program.

    - Documentation that concludes that the program doesn't behave as described in manual or online help, or has features that are not described anywhere.
    - Hardware which reports the program fails on certain kinds of hardware or machine models.
    - Others include any problem that does not fall in any of the above types.
- Severity is used to indicate the rating of the seriousness of problem.
- Problem summary, this one or two lines give symptoms about the problem which helps everyone to quickly review the outstanding problems.
- Problem description discusses problems and explains them if needed. It includes the step by step procedure to reproduce the problem.
- Attachments might be used to support the bug report. This can include a screen capture, a data set that can be used to reproduce problem, or an output file.

While teamwork offers many benefits to software development, it also introduces new obstacles to the development process. The increasing role of team development in the software development environment requires improvement in project management.

A defect tracking system improves efficiency of the project and streamlines the development process. It improves communication between team members, such as

testers, developers, and project managers. It also tracks progress on issues by providing a detailed history log.

3.7 Final Testing

The product is finished. It goes through one last round of release testing. An attempt is made to complete one full round of the test plan on the last version and keep looking for serious problems. Circulate a final deferred bug list. Give management one last chance to evaluate them and reconsider the quality of the product before releasing it. If time is available, retest old bug fixes.

Once the program passes the final testing, it is time to make the disk or CD master for release. We need to check the master disk or CD by installing the program to a fresh machine and running a very simple test of the program. Do the same check to the first few sets of the duplicated disks or CDs that came back from manufacturing.

Between the product release to manufacturing and actual shipment to customers, we have a few days. If there is any doubt, by all means, keep testing the product. If there is anything really nasty, this may justify a recall. It will be expensive, but much cheaper than a recall after the product is shipped.

3.8 Example Testing Tasks in GES32 Acceptance Testing

The acceptance testing suite includes two test cases, one is a stand alone installation, another one launches all modules in the application. The purpose of acceptance testing is to ensure that the QA group receives a good build from developers for testing.

The test requires hardware (a PC with processor speed 90 mhz or higher, 32 MB RAM, 200 MB free disk space, a monitor with resolution 800 by 600 or higher, Security Key, software (Windows NT 4.0 with Service Pack 3, Sentinel driver 5.3), and testing tools (like the PowerTrack System, Microsoft Schedule, network connection).

The test is run every day when the QA group receives a new build from the development team. The GES32 Acceptance testing is done by running automated testing scripts in unattended mode nightly. The automated test procedure is discussed in Section 4.6. The test log will be generated every test run.

The following is a test log with a "failure to display the Select Components dialog box" detected by the running of the automated installation scripts. The testing scripts written in Visual Test 4.0 are in Appendix C.

```
Error! Expect Select Components
Install took:      55.430  seconds.


  [Start Scenario]
  [Name]        Integrated Install
  [Test Location] C:\MSDEV\Projects\GES32\Install\IntegratedInstall.mst (75)
  [Start Time]   11/18/1998 04:07:05
  [Detail 0]     Failed at: C:\MSDEV\Projects\ GES32\Install\IntegratedInstall.mst (111)
  [Detail 0]     Trapped at: C:\MSDEV\Projects\ GES32\Install\IntegratedInstall.mst (111)
  [Result]       FAIL
  [Reason]       Active window is not Select Components
  [Elapsed Time]  55.430
  [End Scenario]
```

When we receive this log in the morning, we double-check the installation. Since the problem still exists, the build is rejected. We generate a defect report in the PowerTrack system and let the developer know the problem. This report includes how the defect is reproduced, version or build numbers, defect type, and defect severity.

Issue ID: G326429 **Title:** Select Component Dialog box does not appear during install
Type: DR (Defect Report)
Severity: 2 - Major problem
Priority: 2 – High
Module: Installation

Submitted By: Lixiu Huang
Date Submitted: 11/18/1998 8:18 AM

In Build560, when install the GES32 to a stand-alone machine, the select components dialog box does not appear. The build is rejected for the QA testing.

After receiving the defect issue from the tester, the developer fixed the problem and rebuilt the program. Then the developer assigned the defect issue back to the QA group for the acceptance testing again.

Issue ID: G326429 **Title:** Select Component Dialog box does not appear during install
Type: DR (Defect Report)
Severity: 2 - Major problem
Priority: 2 – High
Module: Installation

Assigned To: [Leader (Name has been removed)] (for Fixing)
Date Assigned: 11/18/1998 8:20 AM **Fixed**
Should be fixed. The Build560 has been rebuilt.

Submitted By: Lixiu Huang
Date Submitted: 11/18/1998 8:18 AM
In Build560, when install the GES32 to a stand-alone machine, the select components dialog box does not appear. The build is rejected for the QA testing.

We run the automated installation script again. After the program is successfully installed, the automated testing generated a log with the success-installed messages. The program is accepted for further testing in the QA group.

[Start Scenario]
[Name] Integrated Install
[Test Location] C:\MSDEV\Projects\GES32\Install\IntegratedInstall.mst (75)
[Start Time] 11/18/1998 9:05:04
[Result] PASS
[Elapsed Time] 446.434
[End Scenario]

The tester closes the defect issue in PowerTrack.

Issue ID: **G326429** **Title:** **Select Component Dialog box does not appear during install**
Type: **DR (Defect Report)**
Severity: **2 - Major problem**
Priority: **2 – High**
Module: **Installation**

Current Owner: **Lixiu Huang (Closed)**
Date Assigned: **11/18/1998 8:50 PM** **Date Closed:** **11/18/98 10:19 PM** **Close Status: Component Fixed**
Fixed.

Assigned To: [Leader (Name has been removed)] (for Fixing)
Date Assigned: **11/18/1998 8:20 AM** **Fixed**
Should be fixed. The Build560 has been rebuilt.

Submitted By: **Lixiu Huang**
Date Submitted: **11/18/1998 8:18 AM**
In Build560, when install the GES32 to a stand-alone machine, the select components dialog box does not appear. The build is rejected for the QA testing.

Chapter 4

AUTOMATING TESTS

Automated tests execute a sequence of actions without human intervention. This approach helps eliminate human error, and provides faster results, speeds up test turnaround time, and cuts costs during regression testing. Although automating the testing process initially requires an added investment of time and thought, that up-front effort will be recovered by the time product ships. Simply put, automation emulates user actions such as clicking on buttons, typing, dragging and dropping, and selecting menus. Automated tests can do the same steps in the same way over and over again.

According to the Quality Assurance Institute, as much as 80 percent of all software is tested manually [9]. Manual testing is more expensive, boring, and inefficient than automated testing.

The key reasons to use automated testing:

- Save time
- Reproducible steps
- For compatibility testing

- Accurate benchmarking
- Scripts re-use
- Avoid human error

Unattended test operations can be achieved 24 hours a day provided by automation.

4.1 Automation Tool Visual Test 4.0

Visual Test 4.0 is one of many tools on the market today for automating testing on Microsoft Windows-based software. The Visual Test 4.0 originally was released to the market in 1992 as Microsoft Test 1.0 by Microsoft Incorporation. Visual Test has come a long way and now has a new owner, Rational Software, who purchased the product from Microsoft in October 1996.

The Visual Test 4.0 is integrated with Microsoft Development Studio. The Developer Studio is an editor, debugger, and jumping-off point for launching other related utilities associated with the development process. Visual Test 4.0 shares this editor with Microsoft Visual C++, Microsoft Visual J++, Microsoft Fortran PowerStation, and Microsoft Development Library. Programmers need to learn only one programming environment, that is, one editor and one debugger.

Visual Test 4.0 is primarily used as an automation tool for Windows applications. It can truly simulate user action on a graphical user interface (GUI). The Visual Test 4.0 includes a set of powerful features to create, manage, and run tests.

- Test Language

The Visual Test programming language is similar to Visual Basic (VB), and is a procedural language. Keeping each individual test script modular and later combining small scripts into a large test case or suite will make the task of organization and debugging the test scripts quick and easy.

Visual Test offers many high-level functions that make it easy to manipulate, control, retrieve data from, and send data to standard Windows-based controls.

Functions to manipulate objects are limited to standard Windows objects. But custom controls such as grids are not standard and therefore can't be directly manipulated. Visual Test 4.0 language provides a PLAY command to perform simulated keyboard inputs and mouse actions as if they had been entered by a user.

The user can set the default waiting time in a Visual Test 4.0 project for normal Windows actions. However, often during software automated testing, time is a factor that cannot be controlled. Visual Test 4.0 provides a function sleep(time) to specify a time period to when the next action will occur.

- Scenario Recorder

Scenario Recorder is a very powerful tool that a Visual Test 4.0 user can use to record keyboard, mouse, and control events, and then generate a test case that simulates these user actions. While recording these user actions, a user can use the Scenario Recorder to add verifications and comments to the test case. After creating a test case file with the Scenario Recorder, the user can use Visual Test to run the test case file and verify that the application being tested responds as expected to the simulated user actions.

The user can use the Scenario Recorder to get an idea about the commands that can be used to perform certain actions.

- Execute Visual Test case

There are several ways to run Visual Test case.

The first one is that a user can run a test case by setting the order of the MSTs file in the project. The second one is using the Suite Manager tool which comes withVisual Test 4.0 to create Suites by selecting a list of MSTs to run from a test project and designate their order.

A user can also run the test case on a remote computer. The third method is to distribute compiled p-code versions of tests to remote computers and run them using the run-time engine of Visual Test, MTRUN.EXE. The last way to run an auto-test case is to use Test language network distribution procedures to use one computer as a

host to distribute and monitor tests running on other computers (stations). The remote computers don't have to install a full version of Visual Test.

A suite of testing cases can be played back in unattended mode. A batch file can do playback grouped testing procedures in sequence.

- Windows Information Utility

Windows Information utility is available in the Visual Test 4.0 for the user to determine the type of a control, or the class name of the control of a window. It will help the user select the right function call to manipulate user action.

Visual Test 4.0 was chosen for the GES32 automated testing project because:

- Visual Test 4.0 uses Microsoft Development Studio as its development environment. It is the same as GES32.
- It is easy to manipulate the GUI (Graphics User Interface) controls.
- Record and Play make a very powerful tool for the programming.
- No extra license is needed for the testing running on a remote computer.

4.2 The Role of Test Automation

Since most products require tests to be run many times, automated testing generally leads to significant labor cost savings over time. It is helpful in performing tests

quickly and precisely. Successes in automation testing require careful planning and design work.

Traditional manual testing approaches are no longer effective, because the testing phase is usually kept to a minimum and often squeezed into the end of the development process. Testing has tended to be reactive and delayed to the latter stages of the development process. Development companies have traditionally applied people rather than automation to improve their testing process. This strategy has dramatically increased engineering costs. Automated testing will improve test engineer productivity and remove the human error inherent in manual test execution. Because the automated testing scripts can be built in parallel with development, they have the greatest impact on resources and time-to-market especially in the latter part of the testing stage.

Automated testing is not a universal solution. Automated testing should not be considered as a replacement for hand testing, but rather as an enhancement. The human tester, who has to run the tests, checks if testing is proceeding satisfactorily and thinks about what other items need to be checked to ensure quality.

Automated testing takes a long time to create. It may take as long as ten times to run the test only once by hand. Once an automated testing suite has been created, it is a big investment. Any user interface or workflow changes in the program will affect the automated testing script.

Automated testing should be started as early as possible otherwise the cost of automated testing will not be recouped. Automated testing should be started early because automated testing later will lower testing productivity during the peak bug finding period. Automated testing should be started early to create a standard acceptance test, because it will be run so many times and running it manually wastes time.

Automated testing should not be started early because that reduces available testing time during early testing. This delays discovery and correction of many errors. Automated testing should not be started early because the program will change too much. The program is unstable and may be subject to a significant design change. Automated testing should not be started early; it will not pay in for ten testing cycles if creating an automated testing needs as long as ten times to do manually. Some project managers insist that their bug free wonder needs only two or three testing cycles.

There is a difficult tradeoff in whether to use automated testing early. You will have to rely on your own good judgment on when to start the automated testing. For small projects, the time needed to learn and implement automated testing may not be worth it. For larger projects, or on-going long-term projects it can be valuable. Typically a company will pass the break-even point for labor costs after just five or six runs of an automated test.

We don't have to fully automate our tests. We can run some tests manually and others by machine. Automate the easiest ones first, or the ones we know will run many

times, or the ones that are hardest or the most annoying to key in by hand. We can partially automate individual test cases. For instance, it often pays to capture and replay keystrokes, and to analyze the program's output visually. This partial automated test can save the time and bother of retyping test data each time, but doesn't force us to maintain comparison files for the output.

Many times during software development, changes to a product can affect the compatibility of a test procedure or scripts with the test project. The test procedure or scripts need to be modified.

4.3 Automated GES32 Acceptance Testing Cases

In the GES32 acceptance testing, there are two testing cases. One is installing a stand-alone version of the GES32 to a computer. The second case is trying to launch applications of the GES32 product. The detailed description of the test cases can be found in the appendix B.

One of the best places to use automation testing is for the acceptance test. They are the tests run most often. In the GeoGraphix GES32 project, the development group submits an updated version of the program to the testing group every week, plus a few correction versions when particularly bad versions bounce out of testing. In addition, the development group builds a nightly build every day to ensure all changes are in sync

among all developers. Automated acceptance testing is a very quick way to verify the stability of new builds.

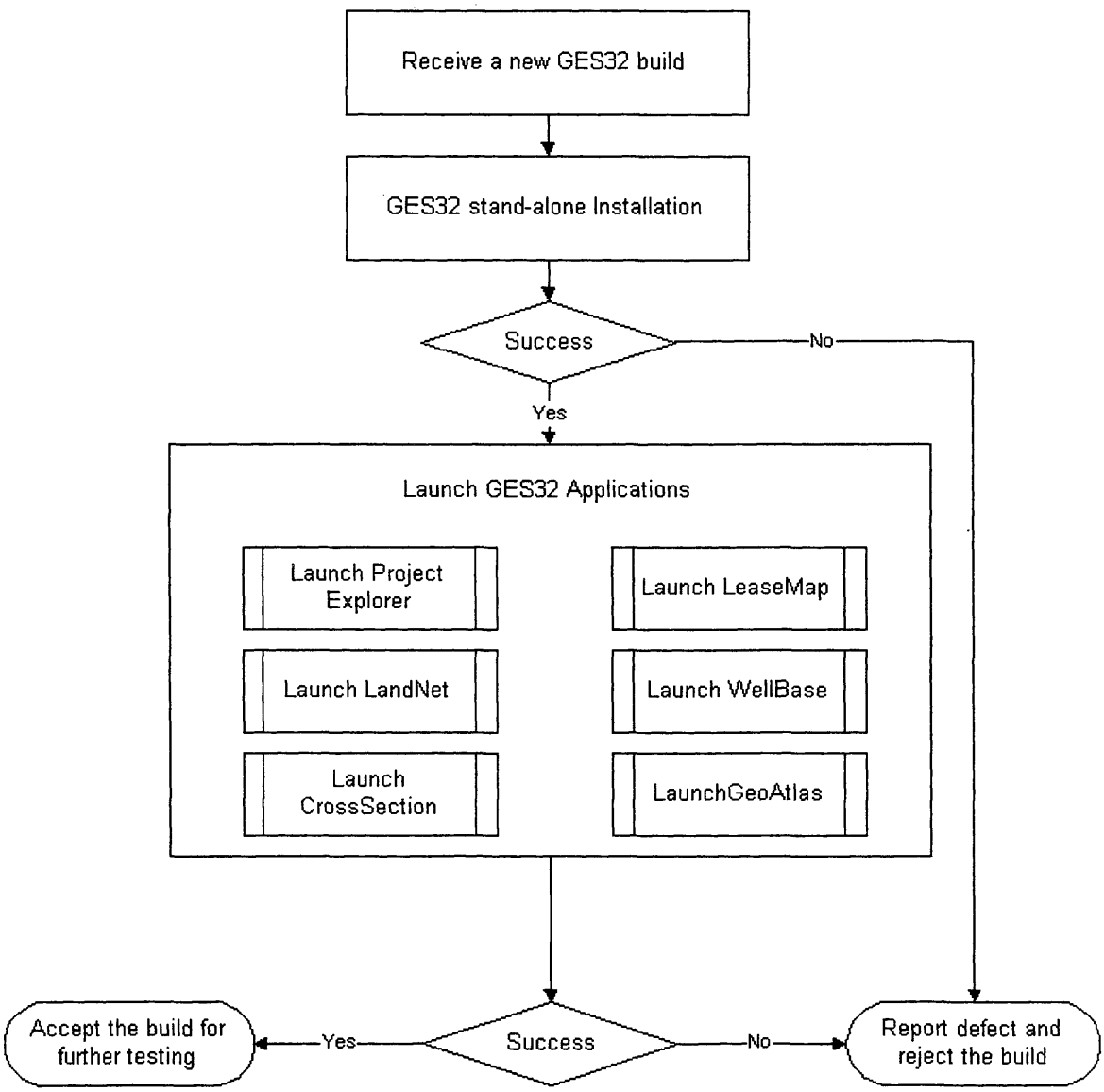


Figure 4.1 GES32 acceptance testing

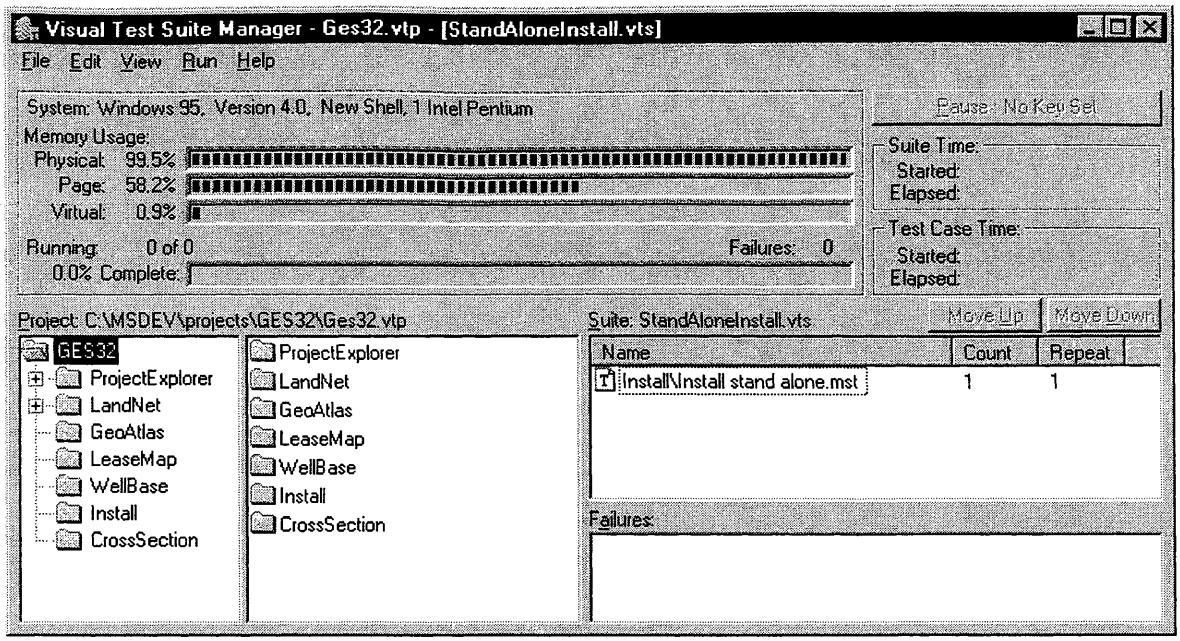


Figure 4.2 Automated GES32 stand alone installation suite.

Another good reason to use automated testing in acceptance testing is that repeating the same tests so many times tempts every tester to skip most of the acceptance test or not to pay careful attention to the results.

The third reason to automate acceptance testing is that there aren't many tests on any individual area of the program. The test skips quickly through the program. Nothing is covered in detail in acceptance testing. Hence if one area of the program changes, usually it will not affect the acceptance testing case. Therefore no testing script needs to be changed.

In the GES32 automated testing project, acceptance testing includes two testing suites (Figure 4.1). One is the stand-alone installation suite (Figure 4.2). It will install a stand-alone version of the GES32 to a machine. Another one is the launchApplication suite (Figure 4.3), which includes six test cases, Launch ProjectExplorer, Launch LeaseMap, Launch LandNet, Launch GeoAtlas, Launch WellBase and Launch CrossSection. The suites run every day to check if a new build of the program satisfies the acceptance criteria.

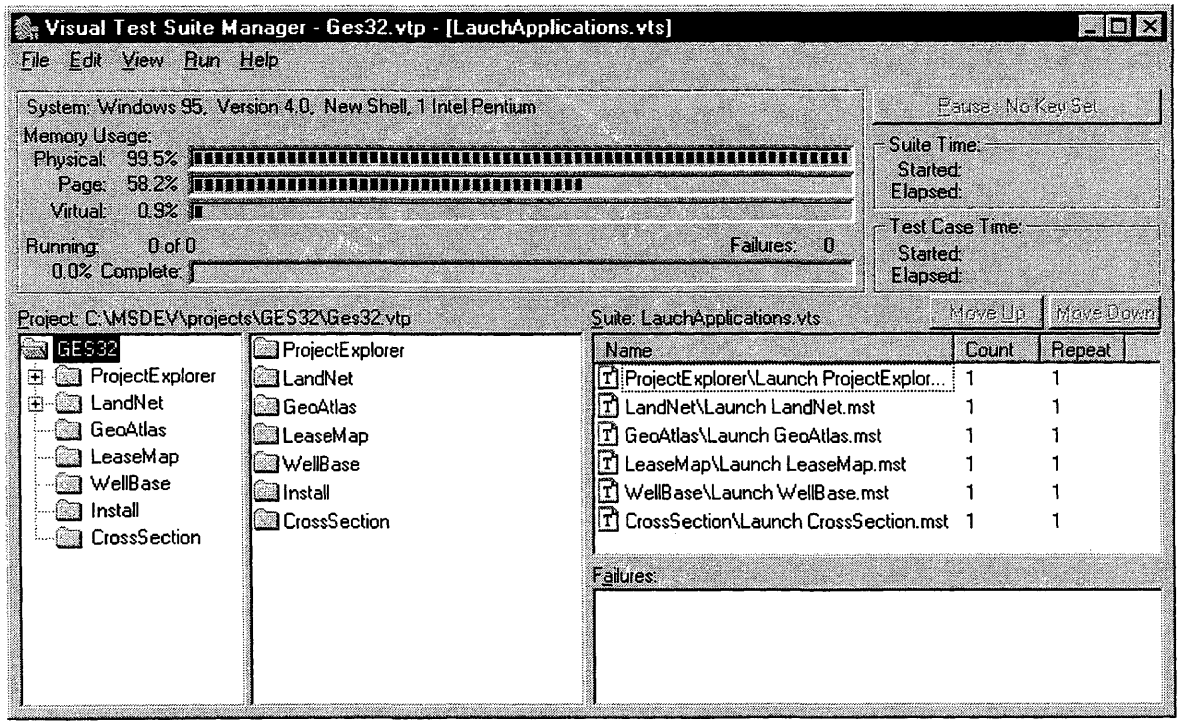


Figure 4.3 Automated GES32 launch application suite.

4.4 Automated GES32 Mainstream Usage Testing Cases

GES32 mainstream usage testing reflects the way that customers will use the product. It includes a series of procedures in consecutive order. The testing emphasis is on detecting errors that are most likely to occur in normal use. The testing is customer oriented and is meant to maximize customer satisfaction.

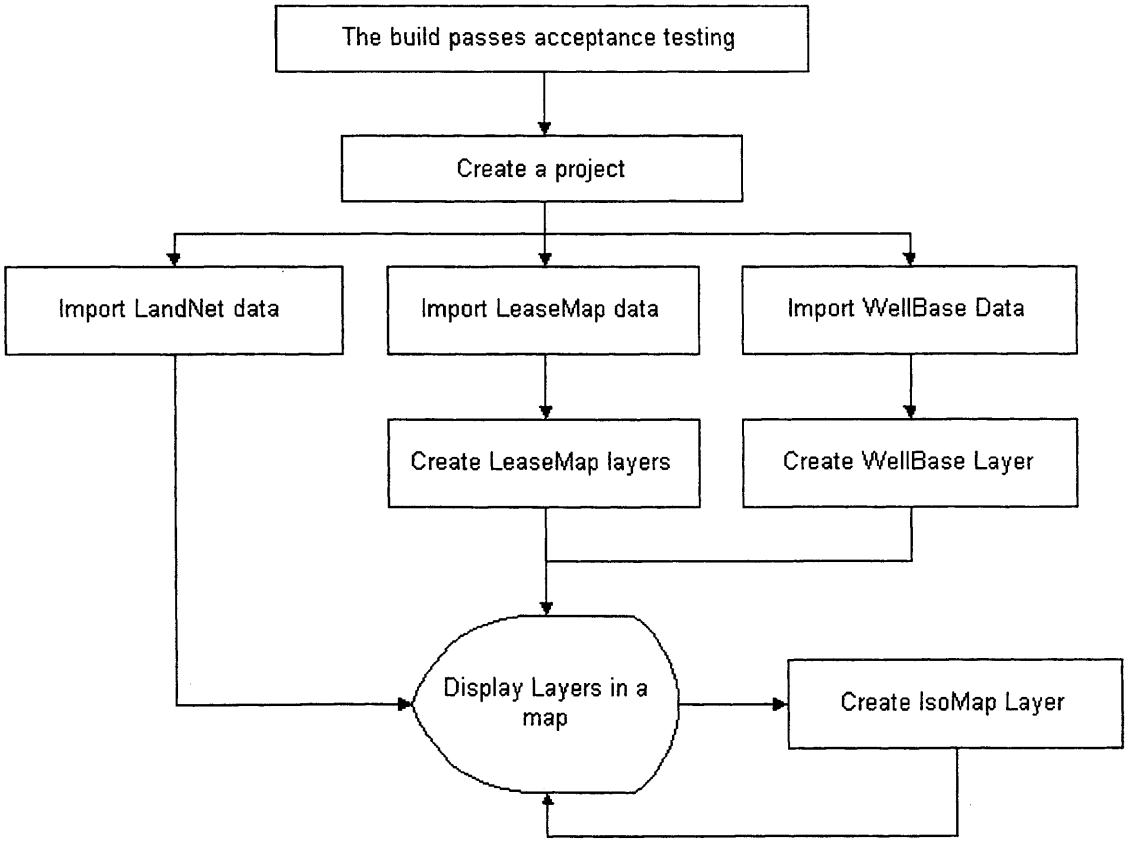


Figure 4.4 GES32 mainstream usage testing

Mainstream usage testing is performed after unit and integration testing (by the programmer), when substantial functionality is available and the program is stable. It is suitable for automated testing.

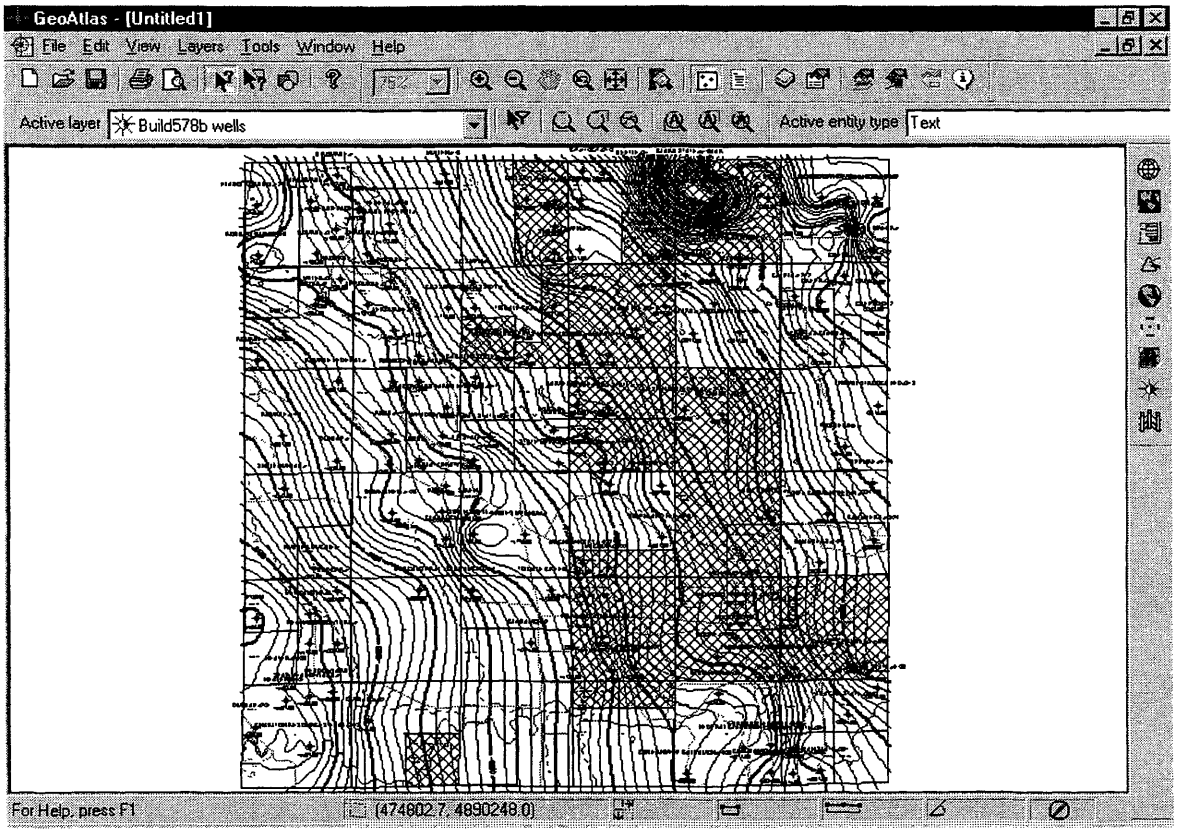


Figure 4.5 GES32 layers shown in GeoAtlas

The GES32 automated mainstream testing (Figure 4.4) includes creating a project using ProjectExplorer through New Project Wizard, importing LandNet CDF format data, LeaseMap LDF format data, WellBase ASSCII2 format well Data, creating LeaseMap, WellBase and IsoMap Layers, and showing all created layers in the GeoAtlas (Figure 4.5). The automated GES32 mainstream testing suite is shown in (Figure 4.6).

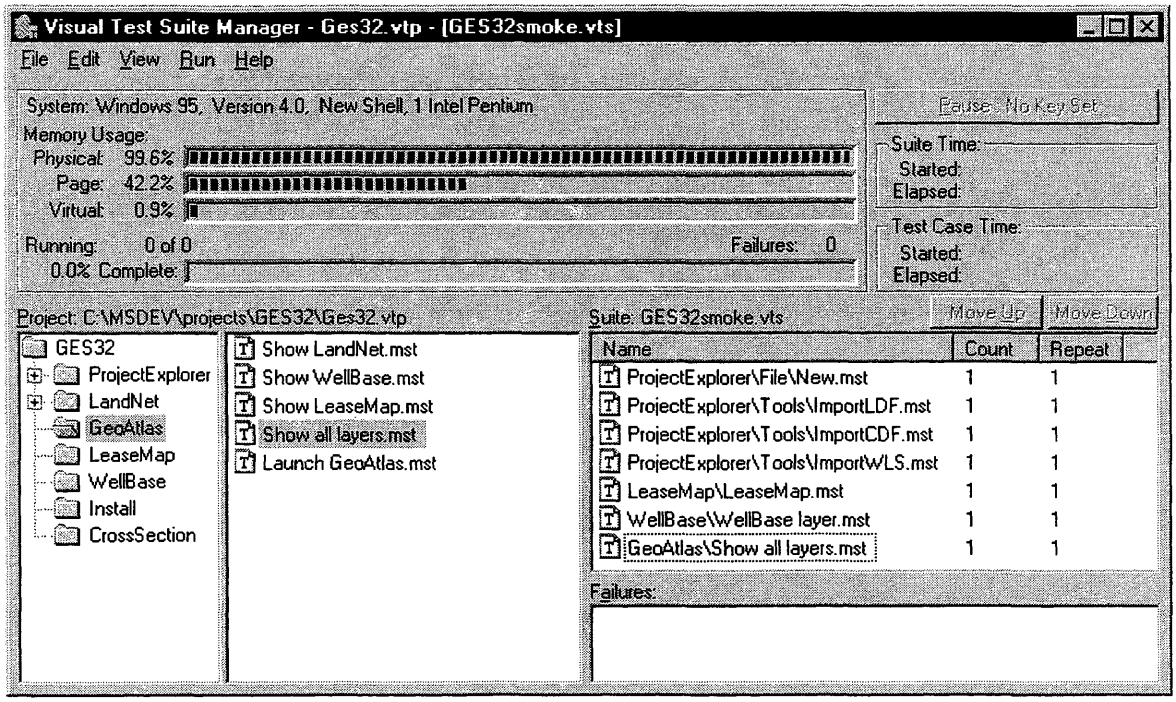


Figure 4.6 Automated GES32 mainstream testing suite

4.5 Automated Regression Testing

Another good place to use automation testing is regression testing. Regression testing is an expensive procedure performed on a modified program to provide confidence that the modified code behaves correctly, and that modifications have not adversely affected existing functionality. Regression testing is performed many times over the life of the product testing. In addition, regression testing is often performed in crisis situations, under greater time constraints.

The more quickly you can automate the regression test, the better. Most programs need many more cycles of testing than anyone expects. Stop automating only when you are certain that the last cycle of testing is near.

4.6 Automated Testing Procedure in GES32

Currently, two automated testing suites, an acceptance testing suite and a mainstream usage testing suite, are developed on the automated test engineer's machine. Then the suites are compiled into p-code versions and are distributed to a network computer. They are played back daily in the GeoGraphix Test Lab in sequence (Figure 4.7). The playbacks are scheduled at early morning in unattended mode after the developers have performed a daily build successfully at mid-night.

The GES32 testing team has six testers. Right now, only one tester is devoted to the automated testing. It includes developing and maintaining automated testing scripts, overseeing the running of automated testing suites, analyzing testing logs and reporting program defects if needed. The GES32 automated testing suites are doing about 25 per cent of the testing job. They are the most frequent tests.

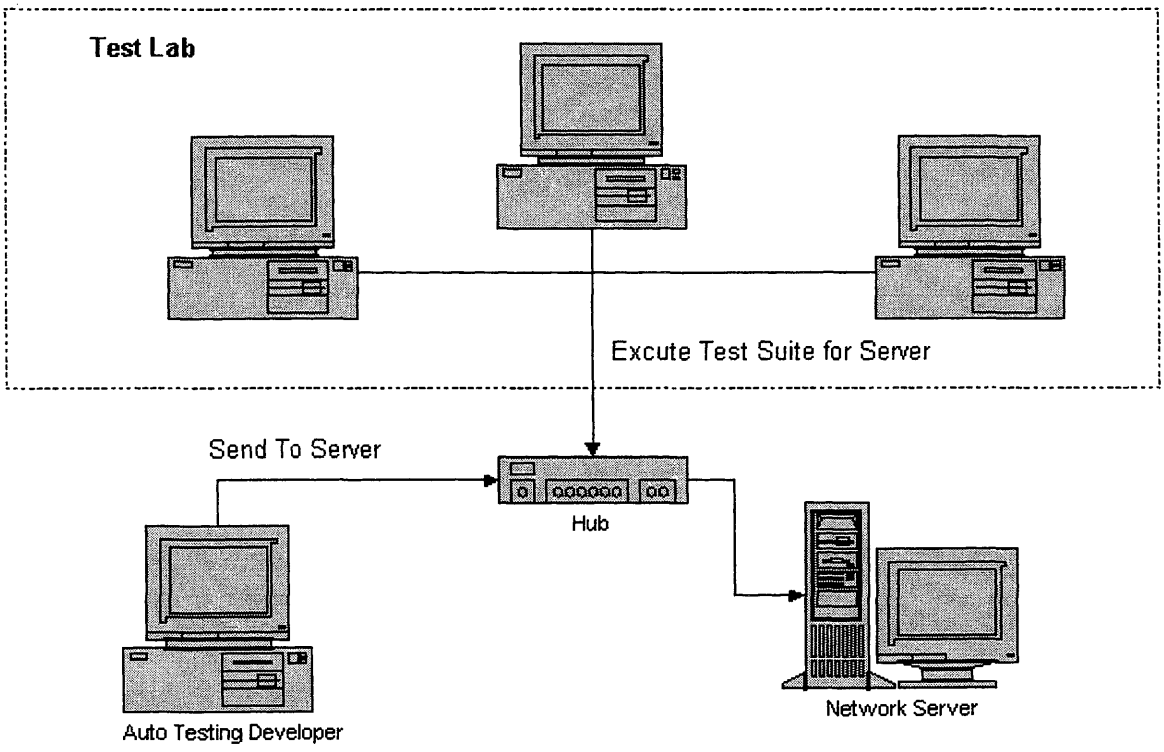


Figure 4.7 Automated testing processing

Chapter 5

GES32 DEFECTS TRACKING SYSTEM

Testers, developers, and technical supporters working on different areas of the product report program defects. During the course of software testing, a large number of defects are reported. Problem reports are a tester's primary work product. The problem tracking system and procedures will have more impact on testers reports' effectiveness than any other systems or procedures. The prime objective of the problem tracking system is to fix the bugs that should be fixed.

PowerTrack, a multi-user, network problem tracking system (Figure 5.1), is used as a defect tracking system in the GES32 project. All testers, programmers, technical writers, technical supporters and project managers have access to it. In this system, all defects can be recorded, retrieved, summarized, and reported. The PowerTrack e-mail auto-notification feature can be used to send notice about any issue to the current owner of the issue. It provides good communication about bugs in the track system. It ensures that a person, who needs to know about the problem, learns it soon after it is reported. In managing the defects, we need to review reported defects, set a priority and status, and

assign them to people who are responsible for taking the next action, such as fixing, reviewing, researching, testing or closing the defects.

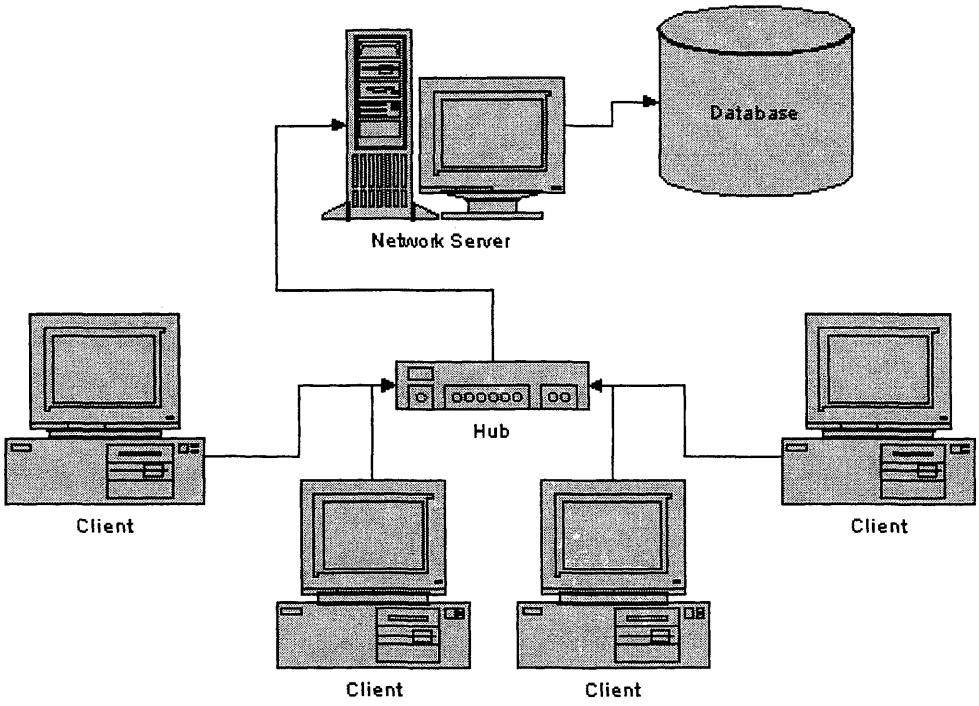


Figure 5.1 GES32 network PowerTrack system

Defects reports are extremely useful in determining the current status of the testing effort. Defect Distribution reports show specified characteristics of defects across

a variety of attributes. Defect age reports show how long defects have had their current status. Defect Trend reports shows the progression of defects from one status to another.

Defect tracking is an important part of the software testing effort. The PowerTrack system provides a structured environment for storing and retrieving defect information. Through it the defect information can be tracked from initial finding through resolution.

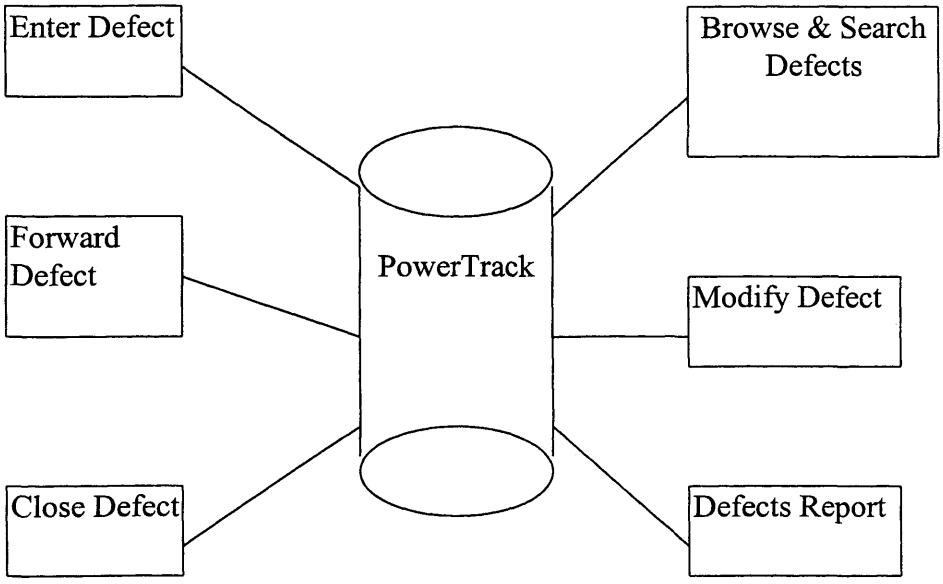


Figure 5.2 Defect tracking system

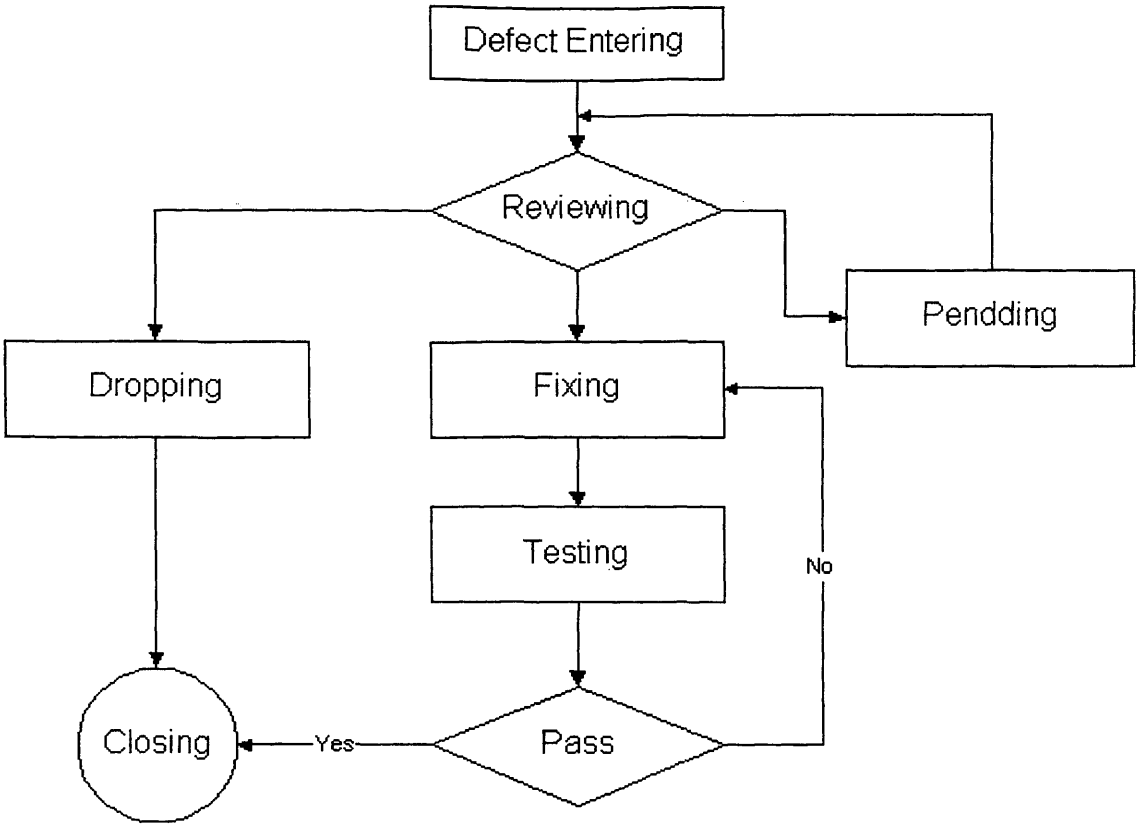


Figure 5.3 Defect flow in the defect tracking system

5.1 Generating a Defect to PowerTrack

When a problem is found, we investigate it in detail and write a clear report. We create an issue about the defect in the PowerTrack system. It includes information such as a descriptive title for the issue, type of issue, to whom such an issue is assigned,

description of the symptom of the issue. Set priority, severity, environment, and test data according to a different defect if applicable.

When entering defect information into PowerTrack, we can specify as much or as little information as is available at the time. As soon as additional information becomes available, we can update the defect information and status as necessary.

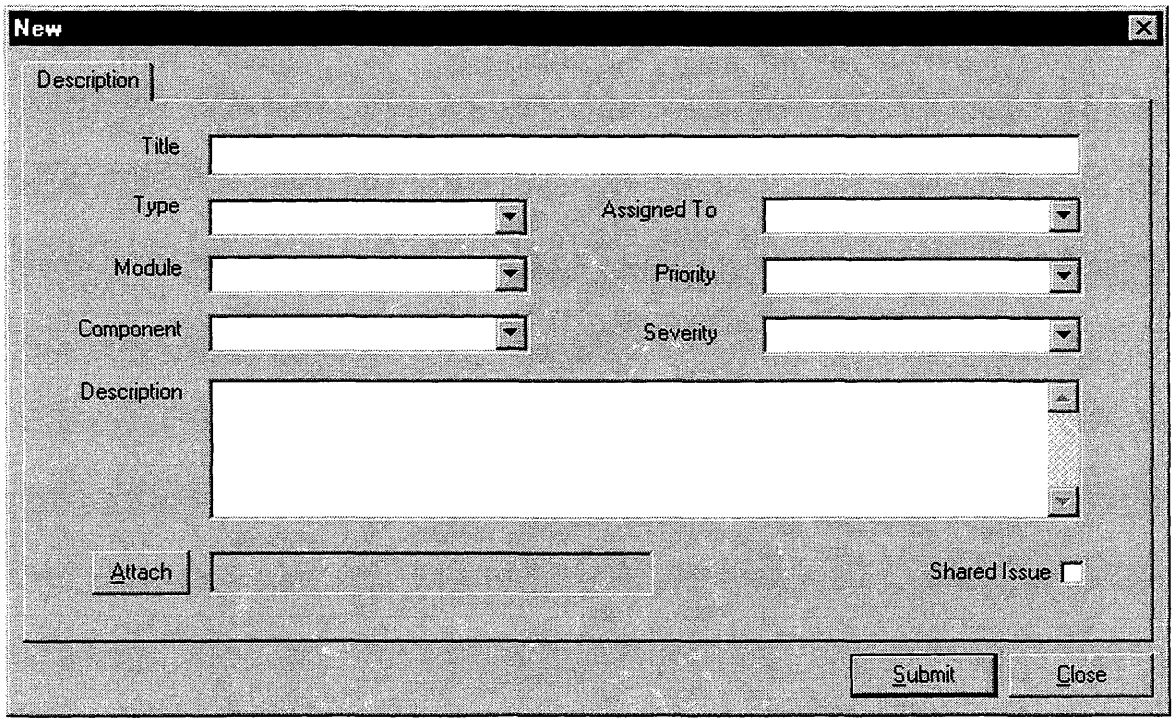


Figure 5.4 New defect entry dialog box

In the dialog box New of PowerTrack (Figure 5.4), we give a brief Title description for a new defect issue, select a type of the defect from CR (Change Request),

DR (Defect Request), FE (Feature Enhancement), MI (Minor Improvement), and NF (New Feature), set priority of the defect from 1- Highest to 4 - Lowest, select a component of the defect from the list of Application Linkage, Attributes, Business Association, CAD Function, Calculated Value, Completion, etc., select a component of the defect from the list of 3D Viewer, Coordinate System, Database, GeoAtlas, install, LandNet, LeaseMap, WellBase, etc., define a severity of the defect as one of System crash, Major problem, Minor problem, Trivial, Enhancement. A description of the defect symptom includes how to reproduce a problem. Attach attachments, such as testing data, screen capture, or other files, to the defect if necessary..

5.2 Browse or Search Defects

There are two filters in the main view (Figure 5.5): Team Member Filter and the Status Filter. Only the issues that belong to a certain team member and status that have been selected appear in the Issue List. Selecting an issue from the Issue List, the detail Pages display all information relevant to the Issue. Browse through the pages by clicking on the tabs, and the information of the Description, History, Current Status, Notes and Links can be seen.

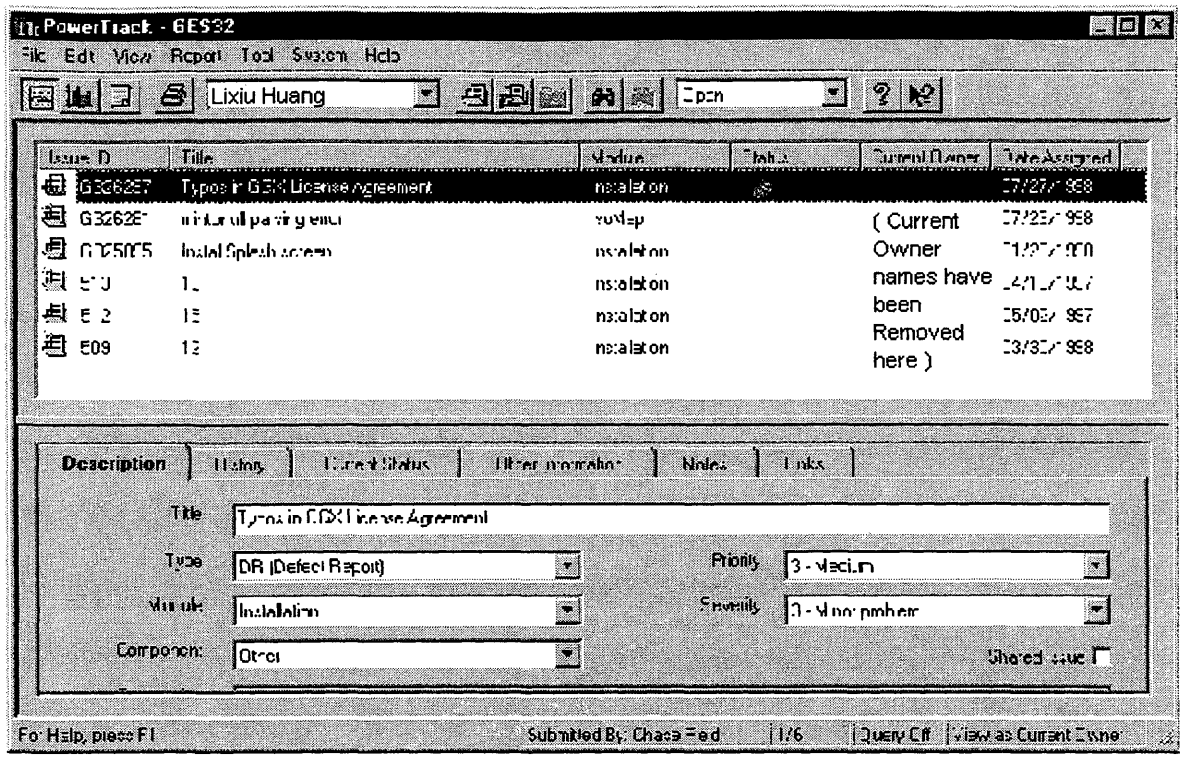


Figure 5.5 PowerTrack main view

PowerTrack features powerful and intelligent search operations. It can handle searches for issue sets based on multiple parameters. In the Search dialog box (Figure 5.6), there are separate file pages: Type, Component, Priority, Progress Status, and Other. All detailed search conditions are defined within these pages. PowerTrack quickly searches out all issues satisfying the detailed search conditions set in the Search dialog box. The Issue List refreshes and displays the results of the search.

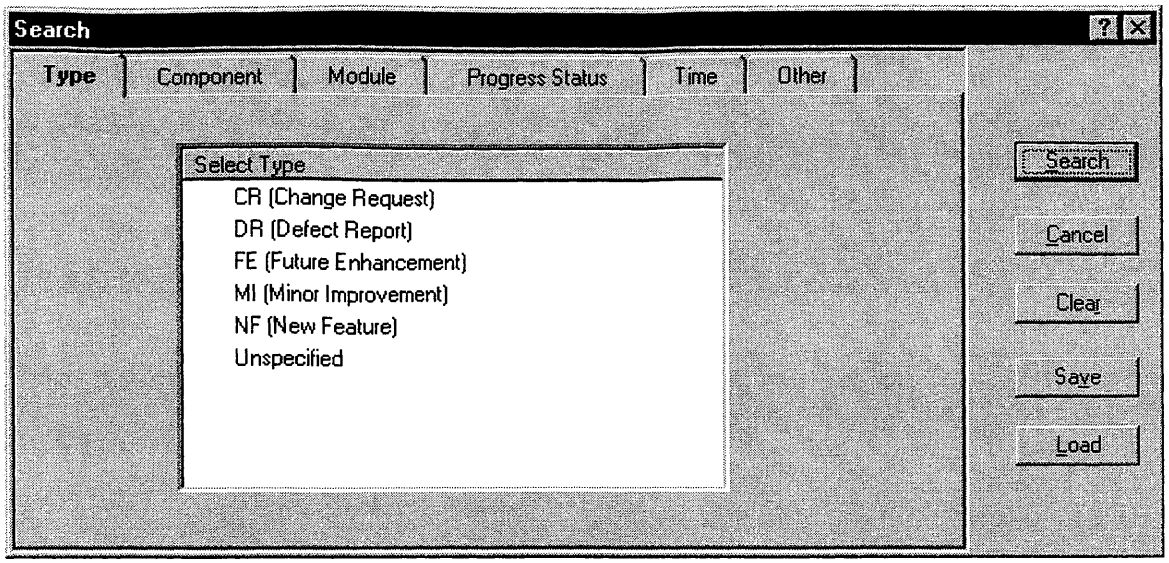


Figure 5.6 PowerTrack search dialog box

5.3 Forward an Issue

The PowerTrack forwarding feature is created to allow existing issues to be forwarded to other members or groups.

After a defect is fixed, the developer often updates the Current Status Page (Figure 5.7) to keep other team members informed of work completed, types a brief entry in the Work Description text box and selects a forwarding Reason in the Forward dialog box (Figure 5.8) and uses this feature to forward the issue to tester for testing and closure.

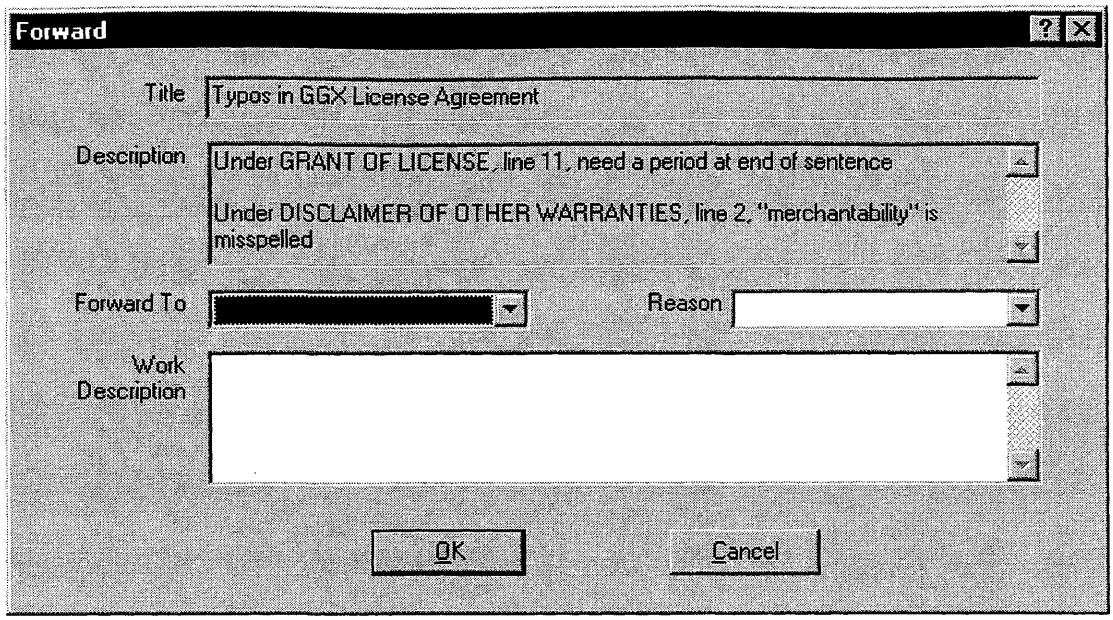


Figure 5.7 PowerTrack defect forward dialog box

When the problem report comes back to the tester, the tester starts to execute the exact test case reported in the record. If the test passes the initial test, try a few tests for obvious side effects. Also try variations of the initial test case. If the program fails the same test that it used to fail, note this on the original Problem Report, change the Current Status Page status back to Pending, and send it back to the programmer. If the program passes the original test, close the original report as Fixed. If a side effect is found, enter a new defect into the PowerTrack system.

5.4 Closing an Issue

Issues are closed when they have been resolved or dropped. Only individuals with privilege can close Issues. In the close issue dialog box, selecting a close status for the drop-down menu, compose a short message in the closing statement text box. The Closing Statement is posted to the History Page of the closed issue.

5.5 A Sample Defect Tracking

Issue ID: G325437 **Title:** ProjectExplorer Exit bug
Type: DR (Defect Report)
Severity: 2 - Major problem
Priority: 2 – High
Module: ProjectExplorer

Current Owner: Lixiu Huang (Closed)
Date Assigned: 04/17/1998 3:19 PM **Date Closed:** 06/22/98 1:49 PM **Close Status:** Component Fixed
fixed

Assigned To: [Leader (Name has been removed)] (for Assistance)
Date Assigned: 04/16/1998 2:43 PM **Fixed**
Should be fixed in build 425

Assigned To: [Leader (Name has been removed)] (for Evaluation)
Date Assigned: 04/10/1998 3:42 PM

Assigned To: [QA Coordinator (Name has been removed)]
Date Assigned: 04/08/1998 3:39 PM

Submitted By: Lixiu Huang
Date Submitted: 04/08/1998 3:39 PM
In ProjectExplorer, after finishing Import LandNet data and trying to exit from ProjectExplorer, Debug Assertion Failed appears. See attachment.

The defect was found on 04/08/1998 by a QA tester. The defect was generated in PowerTrack and reported to the QA coordinator. The QA coordinator assigned the defect to a develop leader for evaluation. The develop leader assigned the defect to a developer for assistance. After the developer fixed the defect, the defect was assigned back to the QA tester for testing and closing. The QA tester tested and closed the defect.

5.6 Reports

PowerTrack provides powerful reporting modules. The reports are grouped into three classifications: Issue List Reports, Distribution Reports and Productivity Reports.

A weekly summary of a new report can be generated to tell everyone what new problems were found this week. The report can be sorted by the severity or by current owner. A daily status report shows the defects reported and grouped by defect types daily. In Figure 5.8 is shown the data on GES32 for the first two weeks of September 1998. The Figure 5.9 is a summary of unresolved problems for a GES32 module. It shows the number of unresolved problems that have changed since the previous week. It is very useful for the project management to reallocate resources for bug fixing. A weekly total number of the fixed defects for a GES32 module summarize the project's progress over time (Figure 5.10). The number of new problem reports per week or per testing cycle usually increases, peaks, and then declines. The Figure 5.11 is a typical case of the

number of defects reported per testing cycle for the GES32 modules. It is unwise to ship the product before reaching a stable, low rate of discovery of new problems. Each of these reports gives us a base of historical data and the current status of the project.

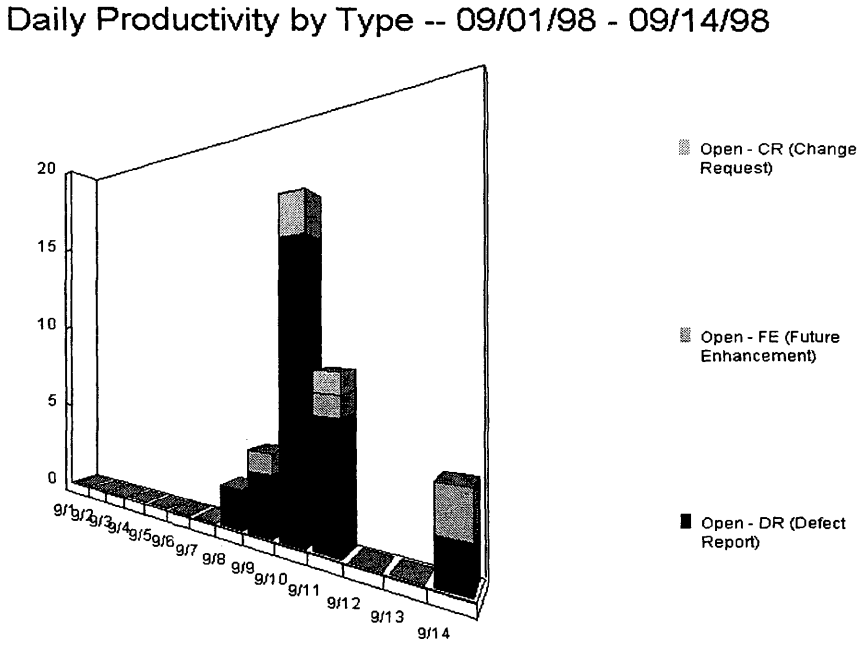


Figure 5.8 Weekly summary of new problem report grouped by defect types

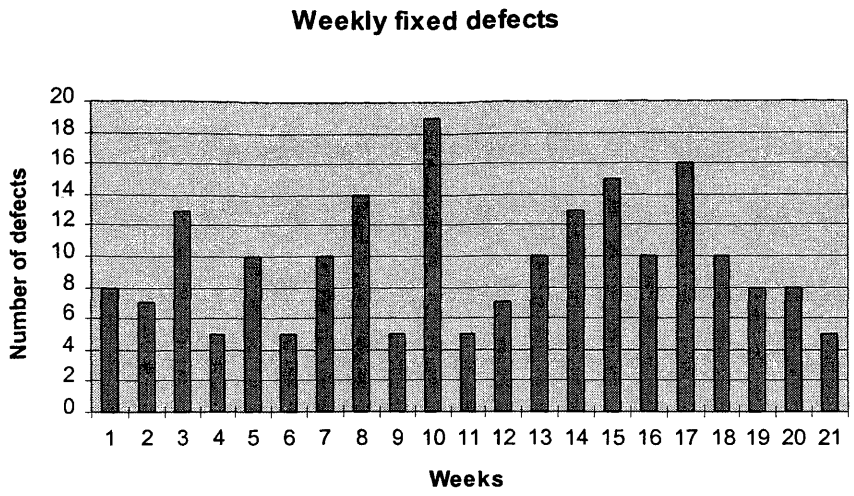


Figure 5.9 Weekly summary of fixed defects

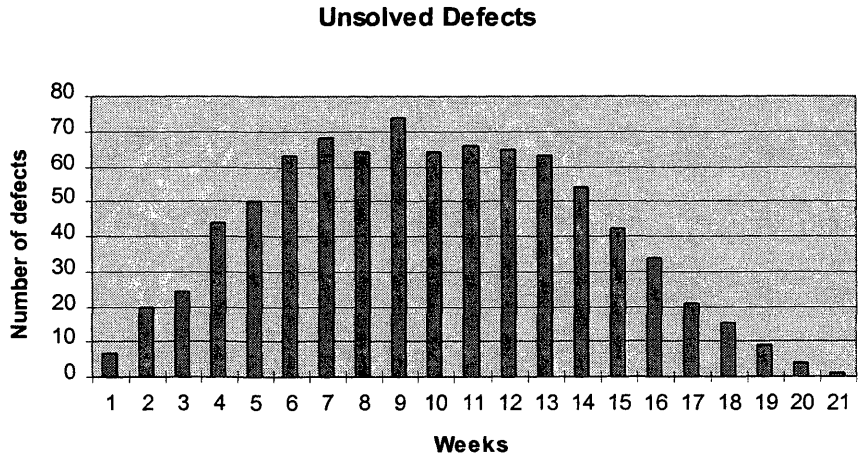


Figure 5.10 Weekly summary of unresolved defects

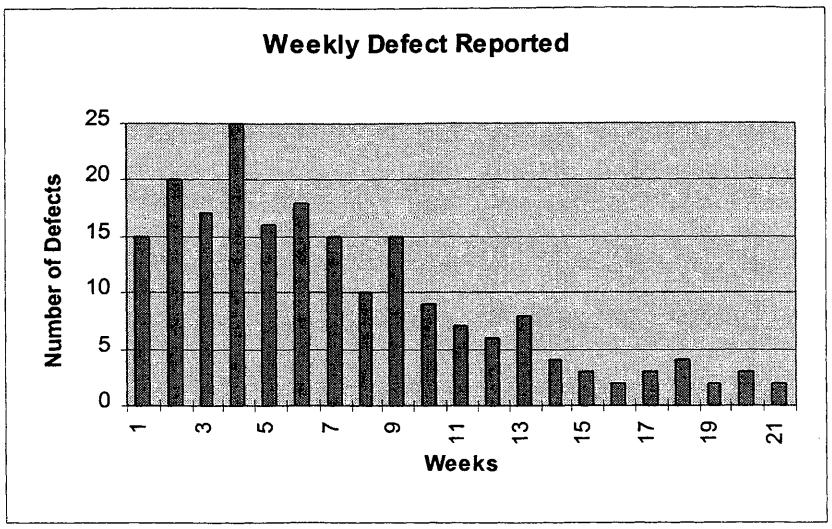


Figure 5.11 Weekly summary of new problems reported

Chapter 6

CONCLUSION

Today, **the** complication and importance of software testing has been recognized. Testing is a risk game that tries to search for the most likely defects since we know that we cannot exhaustively test any system with all possible inputs. A cost-effective testing will ensure that the program is delivered on time and within budget. In this thesis we discussed software testing in the software development life cycle. We should start testing early to detect errors closer to the phase when they are introduced. With the "test to destroy" attitude, testers should find as many problems as possible.

The modern software testing technique, automated testing, can help, performing testing quickly and precisely. As a GES32 test team member, I developed and maintained two automated software testing suites, written in Visual Test 4.0, for the GES32 acceptance testing and mainstream testing. Since the testing suites can be run without an attendant every night, a lot of testing time is saved.

It is possible to develop a new automated testing suite for regression testing. With this suite, we can take advantage of testing automation in the final stage of the development life cycle.

We also demonstrated how a defect tracking system, the PowerTrack, is used in the GES32 testing project. The key benefits of a good bug tracking system are the improvements in communication and accountability that get more bugs fixed.

BIBLIOGRAPHY

[1] Thomas R. Arnold, II Software Testing with Visual Test 4.0. Foster City: IDG Books Worldwide, Inc., 1996.

[2] Edward Kit, Software Testing in the Real World. New York: Addison-Wesley, 1995.

[3] Cem Kaner, Jack Falk, Hung Nguyen, Testing Computer Software, Second Edition. New York: International Thomson Computer Press, 1993.

[4] Rational Software Corporation, Rational Visual Test Tour, Santa Clara: Rational Software Corporation, 1997.

[5] TechExcel, Inc. PowerTrack User's Guide, Lafayette: TechExcel, Inc., 1997.

[6] Hetzel, W. Program Test Methods. Englewood Cliffs, New Jersey: Prentice-Hall, 1973.

[7] Myers, G.J. Software Reliability: Principle and Practices. John Wiley, 1976.

[8] Myers, Glenford J. The Art of Software Testing. New York: John Wiley and Sons, 1979.

[9] Baksaas, Birger Computer-Aided Software Testing. In Dr. Dobb's Journal, February (1994):36-38.

[10] Kepple, Laurence R. The Black Art of GUI Testing. In Dr. Dobb's Journal, February (1994):40-46.

[11] Bradley, N. Scott Software Testing Cycles. In Dr. Dobb's Journal, February (1994): 95-128.

[12] Beizer, Boris, "Software Testing Techniques," Van Nostrand Reinhold, 1990.

[13] A.B. Taha, S.M. Thebaut, and S.S. Liu. An approach to software fault localization and revalidation based on incremental data flow analysis. In Proceedings of the 13th Annual International Computer Software and Applications Conference, pages 527-534, September 1989.

[14] R. Lewis, D.W. Beck, and J. Hartmann. Assay - a tool to support regression testing. In ESEC'89. 2nd European Software Engineering Conference Proceedings, pages 487-496, September 1989.

[15] H.K.N. Leung and L. White. Insights Into Regression Testing. In Proceedings of the Conference on Software Maintenace-1989, pages 60-69, October 1989.

[16] J. Ziegler, J. M. Grasso, and L.G. Burgermeister. An Ada based real-time closed \- loop integration and regression test tool. In Proceedings of the Conference on Software Maintenance-1989, pages 81-90, October 1989.

[17] Gregg Rothermel. Efficient, Effective Regression Testing Using Safe Test Selection Techniques, Ph. D. Dissertation, the Graduate School of Clemson University, May 1996.

[18] Friedman, Michael and Voas, Jeffrey M. Software Assessment: Reliability, Safety, Testability. New York: Wiley Press, 1995.

[19] Hetzel, Bill. The Complete Guide to Software Testing, Second Edition. Wellesley, MA: QED Information Sciences, 1988.

[20] Ince, Darrel. An Introduction to Software Quality Assurance and Its Implementation. London, New York: McGraw-Hill, 1994.

[21] Marick, Brian. The Craft of Software Testing. Englewood Cliffs, NJ: Prentice Hall, 1995.

[22] Royer, T.C. Software Testing Management - Life on the Critical Path. Englewood Cliffs, NJ: Prentice Hall, 1993.

[23] Tallon, Helga; Hopkinson, John; Wallmuller, Ernest. Software Quality Assurance: A Practical Approach. Englewood Cliffs, New Jersey: Prentice Hall, 1994.

[24] Jeffrey Voas, Frank Charron, etc. Predicting How Badly "Good" Software Can Behave, IEEE Software, Vol. 14, No. 4, pages 73-83, July 1997.

[25] Jeffrey Voas, Keith Miller, Software Testability: The New Verification, IEEE Software, Vol. 12, No. 3, pages 17-28, May 1995.

Appendix A

GES32 ACCEPTANCE TESTING PLAN

The following is the GES32 Acceptance testing plan. Some sensitive or commercial confident data have been removed.

GES32 QA Acceptance Testing

Development Team: [Put Names Here]

Technical Writer: [Put Name Here]

Test Coordinator: [Put Names Here]

Testers: [Put Names Here]

Test Plan Overview

[The Test Coordinator for specific application or module will complete this section with input from the Development Lead.]

Tester will develop a Rational Visual Test 4.0 suite to automate GES32 acceptance testing. The acceptance testing suite includes two test cases, one is stand alone installation, another one is launch all modules in the application.

This test plan discusses the following:

1. System Configurations
2. Schedule and Resources
3. Risk Areas
4. Testing Management
5. Testing Requirements

System Configurations

[The Test Coordinator for specific application or module will complete this section.]

Operation System: Windows NT 4.0 with Service Pack 3

Drivers:	Sentinel 5.3
Protocols:	TCP/IP
Processor Speed:	P90 or higher
RAM:	32 or more
Screen Resolution:	800X600 or higher

Schedule and Resources

Testing Schedule

[This section will be complete by the QA Manager with input from the Program Manager.]

Task	Start Date	End Date	Duration	Resource Allocation
Development Testing				
RTT				
Certification Testing				
Synchronized Testing				
RTM				

Risk Areas

[The Test Coordinator for specific application or module will complete this section, with input from the Development Lead.]

This is the first Visual Test 4.0 automated testing project. It may take more time than we thought.

Test Management

[This section describes the testing phases and the "rules" for each. The rules are the agreed upon expectations between developers and QA during testing.]

Development

- Developers are responsible to launching all dialog boxes and running simple functions to ensure the feature will be stable to test.
- Known defects which will cause confusion or concern to the end user will be noted in Release Notes.
- Developers will verify fixes before submitting it back to QA
- Developers will receive defect via PowerTrack as opened issues
- The Application or Module Program Manager and Product Manager will be the focal points for questions, concerns, and problems. If the questions, concerns, and problems are such that they may impact the Application or Module schedule, it will be raised to the Application or Module SDD team.

Testing

- QA will create this testing plan and maintain it throughout the testing cycle, a current copy will reside on [Network Path]

- QA will submit all defects to PowerTrack as Open.
- The Test Coordinator and QA Manager will be the focal points for questions, concerns, and problems. If the questions, concerns, and problems are such that they may impact the Application or Module schedule, it will be raised to the Application or Module SDD team.

Defect Tracking

- Unless directed otherwise by the Program Manager, the Test Coordinator will make the determination of defect assignment and prioritization.
- Defects discoveries, subsequent reports on defects, and defect resolution will be documented in PowerTrack. Testers will initially describe defects, provide location of test data and context files to reproduce the defects, and assign Severity and Priority. The Test Coordinator will assign version, and build number module, component.
- Will differentiate between defects and enhancement.

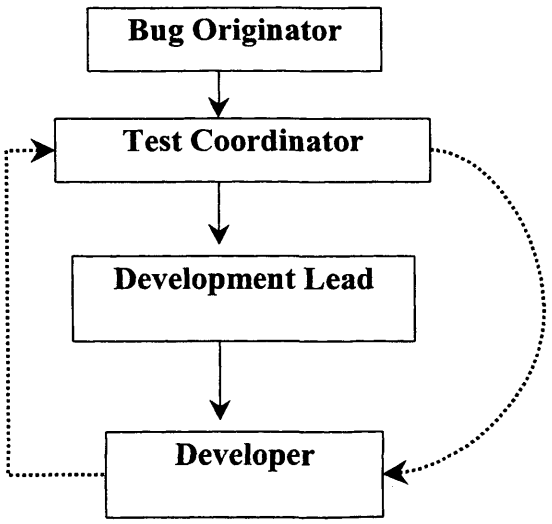
Defect Fixes and Management Process

Definitions

Defect Originator:	The person who found the defect
Test Coordinator:	The person in charge of testing of the project
Development Lead:	The lead programmer on the project.
Developer:	Programmer responsible for the coding of a some portion of the project.
Defect:	A bug that has been validated by the test coordinator.

Defect Tracking Process

In order to ensure that bugs get reported to the correct programmer, the following process should be used.



The test coordinator may send the defect directly to the developer if it is clear who wrote the code -- if the development lead and the test coordinator have agreed that this is OK

Once the developer fixes the defect, it should be re-summited to the test coordinator for testing. The test coordinator will close the fixed defect or send it back to the developer if it has not been fixed.

All bugs MUST be submitted through the test coordinator. It is the test coordinator's job to determine if the "bug" is an enhancement or validate the bug. A validated bug is elevated to a defect. Further, by submitting all bugs through the test coordinator, the chance of getting duplicate defects entered into the error tracking system is reduced. The development leader (if one exists) and the test coordinator should discuss how they want to handle assigning defects to developers. The development lead may request to have all defects submitted to him/her or may chose to have the test coordinator send the defects directly to the developer in those cases where it is clear which developer caused the bug.

Setting Defect Severity and Priority

The test coordinator will be responsible for setting the defect severity and will set an initial defect priority. The defect priority should be reset by the development lead/developer in conjunction with the program manager and/or the SDD team for the project. When defects are of such a nature that parties outside the project are impacted, a panel of those impacted parties may be set up to determine the priority of the defects.

Severity

- Level 1 (System Crashes) = Severe problem rendering the application unusable with system hang or crash, feature hang or crash, or loss of data.
- Level 2 (Major Problem) = serious problem hindering the use of the application as intended with inaccurate results, slow response times, problems with no workarounds.
- Level 3 (Minor Problem) = problem that detract from the work at hand, inconsistencies, spelling errors, or problems with workarounds.
- Level 4 (Trivial) = problems that present themselves in a unique situation of which require a cost-prohibitive fix. (Will not be fixed)

Priority Defect

- 1 (Highest) = Implement in current release. This code is usually associated with a Severity 1 Problem. It may also indicate defects of lower severity that are relatively easy to fix and test.
- 2 (High) = Implement in next incremental release. This code is usually associated with a Severity 2 problem.
- 3 (Medium) = Implement in next major release. This code is usually associated with a Severity 3 or 4 problem.
- 4 (Low) = No commitment or schedule to fix at this time.

Enhancements

- 1 (Highest) = enhancement to add value to the application by making it more usable across a wide client base
- 2 (High) = enhancement for limited marked of marginal economic feasibility.
- 3 (Medium) = enhancement not attainable with present application (Will not be done).

Management

If there are any disputes between QA and a developer that can't get resolved through normal channels, the Program Manager and QA Manager will work together to resolve the dispute.

Testing Requirements

Acceptance Criteria

1. Descriptions of new features/enhancements and defects which have been fixed.
2. The GES32 is able to install on a stand-alone machine (See Attachment).
3. All modules in the GES32 will be launched. It includes:
   - GeoAtlas
   - LandNet
   - LeaseMap
   - ProjectExplorer
   - QueryBuilder
   - WellBase
   - XSection

Tasks

1. Create a Rational Visual Test 4.0 automated testing suite GES32 Acceptance.
2. Maintain the GES32 Acceptance suite as needed.
3. Execute the suite on the test lab daily without an attendant.
4. Analysis the auto testing log.
5. Report defect if any.

Areas which will not be tested

[The Test Coordinator for specific application or module will complete this section.]

1. GES32 server installation (Windows NT, NOVELL)
2. GES32 workstation installation

Appendix B

SAMPLE GES32 TEST CASES

B.1 Stand-alone Installation Testing Case

Purpose of the test:

- Make sure the GES32 is installable on a clean machine.

System requirements:

- Hardware: Pentium processor, CD-ROM or remote access to CD-ROM, 64 MB of system RAM (128 MB recommended), 200 MB of free hard disk space. 15-inch monitor and video graphics card with SVGA resolution.
- Software: Microsoft Windows NT 4.0 with Service Pack 3, TCP/IP or IPX/SPX network protocol, Sentinel Driver.

Others:

- User must have administrative right to the computer to install the GES32.
- GeoGraphix FLEXlm license file and matched security key for GES32 product.

- Insert the GES32 CD-ROM into the CD-ROM driver or have the GES32 installation media available and accessible in the network

Testing actions, expected system reactions and results:

- Start GES32 installation by running setup.exe in the appropriate CD-ROM driver or network driver. The GeoGraphix Products setup page appears.
- Click the Next button, the GeoGraphix Software License Agreement page appears.
- Click the Yes button to accept the term of the License Agreement, the GeoGraphix User Information page appears.
- Type in name and company in the User Information page, then click the Next button, the Select Install Type page appears.
- Click the Stand-Alone button to install a stand-alone version of the GES32, the Choose Destination Location page for GeoGraphix products appears with a default Destination Folder:

  C:\Program Files\Geographix.

- Accept the default GeoGraphix product location, click the Next button, the Choose Destination Location page for GeoGraphix products projects appears with a default Destination Folder:

  C:\GeoGraphix\Projects

- Accept the default project location, click the Next button, the Select Components page appears. The default settings are for installation of all four software components, plus sample data sets and demo files.
- Deselect all components except the GES and GES Sample Data components, Click the Next button, the Select Program Folder page appears with default Program Folder GeoGraphix.
- Accept the default setting and click the Next button, the System Template Units page appears with a default English unit.
- Accept the default English unit, click the Next button, the GeoGaphix License File page appears with a default I have license file to install selected.
- Accept the default: I have a license file to install. Click the Next button, the License File Source Location page appears.
- Insert the license file diskette in the diskette driver A. Click the OK button to continue the installation, the Copying program files progress bar appears. Upon completion of the loading progress, the ReadMe File page appears with the default Yes, I want to view the ReadMe file now checked.
- Unchecked the check box in the ReadMe File page, click Next button, the Setup Complete page appears with the default Yes, I want to restart my computer now selected.
- Accept the default, click the Finish button to restart the machine.

B.2 Launch Applications Testing Case

Purpose of the test:

- Make sure that all modules of the GES32 are good for the further testing.

System requirements:

- Hardware: Pentium processor, CD-ROM or remote access to CD-ROM, 64 MB of system RAM (128 MB recommended), 200 MB of free hard disk space. 15-inch monitor and video graphics card with SVGA resolution.
- Software: Microsoft Windows NT 4.0 with Service Pack 3, TCP/IP or IPX/SPX network protocol, Sentinel Driver.

Others:

- GeoGraphix FLEXlm license file and matched security key for GES32 product.
- GES32 product is installed successfully.

Testing actions, expected system reactions and results:

- Launch Project Explorer in start menu by choosing

  Start>>Programs>>GeoGraphix>>Project Explorer

  The Project Explorer window opens. In the left pane, the default active project Sample appears.

- Launch LandNet in start menu by choosing

  Start>>Programs>>GeoGraphix>>LandNet

The LandNet window opens.

- Launch LeaseMap in start menu by choosing

  Start>>Programs>>GeoGraphix>>LeaseMap

  The LeaseMap window opens.

- Launch WellBase in start menu by choosing

  Start>>Programs>>GeoGraphix>>WellBase

  The WellBase window opens.

- Launch CrossSection in start menu by choosing

  Start>>Programs>>GeoGraphix>>CrossSection

  The CrossSection window opens.

- Launch GeoAtlas in start menu by choosing

  Start>>Programs>>GeoGraphix>>GeoAtlas

  The GeoAtlas window opens.

B.3 GES32 Mainstream Usage Testing Cases

Purpose of the tests:

- Make sure that the user can create, modify, and use the map in the GES32.

System requirements:

- Hardware: Pentium processor, CD-ROM or remote access to CD-ROM, 64 MB of system RAM (128 MB recommended), 200 MB of free hard disk space. 15-inch monitor and video graphics card with SVGA resolution.
- Software: Microsoft Windows NT 4.0 with Service Pack 3, TCP/IP or IPX/SPX network protocol, Sentinel Driver.

Others:

- GeoGraphix FLEXlm license file and matched security key for GES32 product.
- GES32 product passes acceptance testing.
- Sample data within the GES32 product.

Testing actions, expected system reactions and results:

Create new GES32 project New Sample.

- Launch Project Explorer in start menu by choosing

  Start>>Programs>>GeoGraphix>>Project Explorer

  The Project Explorer window opens. In the left pane, the default active project Sample appears.

- To create a project, Choose File>>New Project menu, the New Project Wizard page appears.

- In the Project Name box, type project name New Sample. Click the Next button. The Database Coordinates System page appears.
- In the Database Coordinates page, adjust the database coordinates settings to the following:

Region:	United States
System:	Geographic Latitude/Longitude
Datum:	NAD 1927 - North American Datum of 1927 (Mean)
Area:	Mainland United States (Lower 48 and Offshore Waters)
Hemisphere:	Northwest

- Click the Next button. The Map Coordinate System page appears.
- On the Map Coordinate System page, adjust these settings as follows:

Region:	United States
System:	Universal Transverse Mercator (US Survey Feet)
Zone:	13, 108W-102W
Datum:	NAD 1927 - North American Datum of 1927 (Mean)
Area:	Mainland United States (Lower 48 and Offshore Waters)

- Click the Next button. The New Project Wizard page appears.
- Accept the default, click the Next button, GES32 will set extents to project data.

- Click the Finish button. The new project appears in the Project Explorer window. The new created project New Sample is active, and the New Sample icon appears as an octagon.

Import LandNet data.

- In the Project Explorer, choose Tools>>GES32 Applications>>LandNet menu, the LandNet launches.
- Minimize Project Explorer and continue with the testing.
- In LandNet window, in the title bar, verify the Active Project is New Sample. Choose File>>Import>>GES CDF Cartigraphic Data Format. The Import dialog box opens.
- To select a file for import, click the Browse button. The Open dialog box appears.
- If you installed GES32 using the default settings, the sample files are in the C:\Program Files\GeoGraphix\SampleFiles\Sample. For the New Sample project, import the SAMPLEGRID.CDF file by selecting SAMPLEGRID.CDF and clicking Open.
- In the Import Data File box, the name and path of the selected file appears.
- In the Import As box, select the Display Layer and LandGrid Database options.
- For the display layer name, type New Sample Grid.

- After inputting all information into the Import dialog box, click Next. The ASCII Data Coordinate System dialog box appears, displaying the project coordinate system information.
- Click Finish. A Meridian Selection dialog box appears.
- In the Meridian Code column, click the 6 to select the 6th Principal Meridian and click OK. LandNet imports the SAMPLEGRID.CDF file.
- To import cultural data, choose File>>Import>>GES CDF Cartigraphic Data Format. The Import dialog box appears.
- To select a file for import, click the Browse button. The Open dialog box appears.
- If you installed GES32 using the default settings, the sample culture file is in the C:\Program Files\GeoGraphix\SampleFiles\Sample. For the New Sample project, import the SAMPLECULTURE.CDF file by selecting SAMPLECULTURE.CDF and clicking Open.
- In the Import Data File box, the name and path of the selected file appear.
- In the Import As box, the Display Layer option is selected and the LandGrid Database option is blank.
- For Display Layer Name, type New Sample Culture.
- After inputting all information into the Import dialog box, click Next. The ASCII Data Coordinate System dialog box appears, displaying the project

coordinate system information. To import the SAMPLECULTURE.CDF file and create the Culture display layer, click Finish. While LandNet imports the data, a progress box appears.

- Close LandNet by choosing File>>Exit and continue with remainder of the testing.

Import lease data

- The procedure for importing lease data, adding a record using the landgrid database, filtering lease data, and creating a filtered lease layer follows. Choose File>>Import>>LDF Import.
- The LDF Import dialog box appears.
- To select a file for import, click the Browse button. The Open dialog box appears.
- Set Look In to Sample. If you installed GES32 using the default settings, the path to this folder is C:\Program Files\GeoGraphix\SampleFiles\Sample. For the Beta Sample project, import the SAMPLE.LDF file by selecting the SAMPLE.LDF file and clicking Open.
- The name and path of the selected .ldf file appears in the Import Data File text box.
- Click Next. The ASCII Data Coordinate System dialog box appears. To import the SAMPLE.LDF file, click FINISH.

- When the import is done, choose File>>Tract Navigator.
- The Tract Navigator opens to Note 10. Maximize window if necessary.
- To add a new parcel to this tract, in the Legal Description Information grid, select the first blank row.
- In the Legal Description Information box, click the Landgrid button. The Survey Polygons dialog box appears.
- Select the row that contains Section 7 and click OK.
- The Tract Navigator displays the new parcel polygon and its corresponding legal description.
- On the Child Description column title, right-click and choose Quarter. The Quarters palette appears.
- Click the E2 button twice to quarter the new parcel to the E2E2.
- The polygon created from the landgrid database appears in the Legal Descriptions view box.
- Close the Quarters palette.
- To see both polygons in the Legal Description Information Viewer, click the Select All Polygons box. The Tract Navigator appears.

Create LeaseMap Layers

- Choose Layer>> Create. The Choose LeaseMap Template dialog box appears.

- Select New LeaseMap Template and click OK. The LeaseMap Layer Create dialog box appears.
- On the Data Posting page, in the Tables box, open the Tract Header table.
- Select the TractID field and drag it to the Data Posts area.
- Click the Fill/Line tab. The Fill/Line page appears.
- Select Foreground for Pattern, and select light gray for Foreground color.
- At the bottom of the LeaseMap Layer Create dialog box, in the Name box, type New Sample All Tracts.
- Select the Template option to apply these layer attributes to another layer at another time.
- Click OK. The LeaseMap Layer Create message box appears. Click OK.

- Choose Tools>>GES Applications>>QueryBuilder. QueryBuilder launches.
- Select View >> LeaseMap and adjust the Tables area to display a full view of the LeaseMap tables. Open the Working Interest Partnership table.
- In the Working Interest Partnership table, select the WI Partner field and drag it into the first (leftmost) column of the QueryBuilder grid.
- In the Working Interest Partnership table, select the %Interest field and drag it into the next column of the QueryBuilder grid.
- In the QueryBuilder grid Spy cell for the WI Partner field, select "?". "=?" appears in the Criteria cell.

- In the QueryBuilder grid Criteria cell for the %Interest field, select >#. The > # appears in the Criteria cell.
- Replace the # with 75.
- In the QueryBuilder dialog box, for WI Partner ='?'. enter GeoGraphix Inc. and click OK. The filter results appear in the Query Results area.
- To save the filter for use in LeaseMap, choose File>>Save Filter As.
- When the LeaseMap Filters dialog box appears, name the filter "GeoGraphix".
- Close QueryBuilder and return to LeaseMap.
- In LeaseMap, choose Filter>>Activate. The Open Filter dialog box appears.
- Select the GEOGRAPHIX.LMF file and click Open.
- Only the tracts and leases included in the filter are in the information managers.
- To create a LeaseMap layer from this filtered dataset, choose Layer>>Create. The Choose Leasemap Template dialog box appears.
- Select the New Leasemap Template and click OK. The LeaseMap Layer Create dialog box opens to the Data Posting page.
- Along the lower edge of the page, in the Name text box, type New Sample GeoGraphix Leases.
- Open the Mineral Interest Ownership table and select the Lease ID field.
- Drag the Lease ID field to the Data Posts box two or three lines from the top.

- With the Lease ID data post selected, right-click and choose List>>All.
- Open the Working Interest Partnership Table and select the WI Partner field.
- Drag the WI Partner field to the Data Posts box just below the LeaseID.
- Select the %Interest field and then drag it on top of the WI Partner.
- On the WI PARTNER + INTEREST data post, right click and choose Properties. The Data Post Properties page appears.
- Select the INTEREST field and in the Suffix Text box, type a % sign. Click OK.
- In the LeaseMap Layer Create dialog box, click the Filters tab. The Filters page appears.
- Select the Apply filter to layer option. In the Available Filters list, select the GeoGraphix.lmf.Click the Fill/Line tab. The Fill/Line page appears.
- For Pattern, select Left Hatch. For Foreground color, select red.
- To create the GeoGraphix Leases layer, click OK.

Importing well data

- Choose Tools>>GES Applications>>WellBase. WellBase launches.
- In the WellBase title bar, verify that the active project is New Sample.
- Choose File>>Import>>ASCII2 Import. The ASCII2 Import dialog box appears.

- To select a file for import, click the Browse button. The Open dialog box appears.
- If you installed GES32 using the default settings, the sample files is in the C:\Program Files\GeoGraphix\SampleFiles\Sample. For the New Sample project, import the SAMPLE.WLS file by selecting SAMPLE.WLS and clicking Open.
- The name and path of the selected .wls file appear in the Import File box.
- Click Next. The ASCII Data Coordinate System dialog box appears. To import the SAMPLE.WLS file, click FINISH.
- After WellBase finishes importing the data, the Information Manager displays the imported data.

Create WellBase Layer.

- Choose Layer>>Create. The Choose WellBase Template dialog box appears.
- Select New WellBase Template and click OK. The WellBase Layer Create dialog box appears.
- On the Data Posting tab, open the Well Header table.
- Select the Operator field and drag it to the Posting Area.
- Open the Formations table.
- Select the Top Subsea field and drag it to the Posting Area. The Additional Information Needed dialog box appears.

- From the Select Formation list, select MDDY (Muddy formation).
- In the Select Source list, select default as the source, click the greater than button. Default appears on the right side.
- Click OK. In the Posting Area, drag a box around the two posted fields.
- Right-click the selected area and choose Font. The Font Attributes dialog box appears.
- Adjust the Font to Times New Roman, the Font Style to Bold, and the Height to 160 Feet and click OK.
- Right-click on the location symbol in the posting area and select Bottom Hole Properties. The Bottom Hole Properties dialog box appears.
- Confirm/select the following settings in the Bottom Hole Properties dialog box:

Symbol:	Library Symbols
Size:	490 Feet
Library:	Default
Attribute key:	Well Status

- Click OK.
- At the bottom of the WellBase Layer Create dialog box, in the Name box, type New Sample Wells Muddy and click OK to build the layer.

- When the progress bar shows complete and disappears, layer create is complete.
- Choose Filter>>New. QueryBuilder launches.
- In QueryBuilder, open the Formations table.
- Select Formation and then drag it to the left column of the grid area.
- Click the Spy cell and select MDDY.
- Choose Query>>Run. A list of all the wells that penetrate the muddy formation appear in the Results Area.
- Choose File>>Save Filter As. In the WellBase Filters dialog box, save this filter as New Sample Muddy Wells.

Display Layers.

- Choose Tools>>GES Applications>>GeoAtlas. GeoAtlas launches.
- To open a new map, choose File>>New>>Map.
- Choose Layers>>Select. The Layer Selection dialog box appears.
- Select the New Sample Grid.
- Select the New Sample Culture.
- Select the New Sample All Lease.
- Select the New Sample GeoGraphix Lease.
- Select the New Sample Wells Muddy layer.
- The Map appears.

Create IsoMap Layer

- Maximize GeoAtlas and choose Layers>>Select
- The Layer Selection dialog box appears.
- Select the New Sample Wells Muddy layer.
- The Wells Muddy layer appears in the right pane. Click OK.
- Use Zoom In to magnify a concentration of wells.
- In the Active Layer List, select the New Sample Wells Muddy layer.
- Using the Query Selection Tool, click any Top Subsea value.
- The selected text is highlighted with a green box.
- Right click the map and choose Create IsoMap Layer.
- The IsoMap LT dialog box appears.
- In the Layer Name box, type Top Subsea MDDY and click OK. The map appears (Figure ????).

Appendix C

SAMPLE AUTO TEST CODES

C.1 Main.inc

```
' File name: main.inc
' Project:  GES32
Const FINDWINDOWFLAGS_IF =  FW_FOCUS or FW_PART or FW_ERROR

const PROJECT_NAME$ = "Build578b"
'const PROJECT_NAME$ = "Auto Test"
const PROGRAM_PATH$ = "D:\Program Files\GeoGraphix"

'const INSTALL_PATH$ = "\\Geoden03\Development\Ges32 Builds\"
'const INTEGRATED_INSTALL_PATH$ =
"\\Geoden01\groups\Teams\Products\GeoGraphixProducts\R99.1\Integrated Install"
const INTEGRATED_INSTALL_PATH$ =
"\\Geoden03\Development\GGXBuild\IntegratedCD\Setup.exe "
```

C.2 Util.inc

```
' File name: util.inc
' Project:  GES32 Auto testing
'=============================================================================
' Purpose:    Declare Currency application user-interface related
'             constants, for use in individual test scripts.
'
'-----------------------------------------------------------------------
' Dependencies: None
'
' Notes:      Test cases should not contain hard-coded information, unless
'             absolutely necessary.  Containing and "abstracting" this
'             information into an include file allows all test cases to be
             updated instantly by simply modifying this file.
```

```
'-------------------------------------------------------------------

'-------------------------------------------------------------------
'                    Subroutines
'-------------------------------------------------------------------

'Subroutine:  TestBegin

'This subroutine opens Project Explorer or returns an error message
'if PE can't be opened.

dim CurrentWindow as long
dim hwndApp as long


SUB TestBegin( Title as string, ProgramName as string)

        'viewport clear
        Sleep(10.0)
        SetDefaultWaitTimeout(10)

        'finds the PE window and returns the handle for the PE window
        dim hwndApp as long

        hwndApp = wFndWnd( Title, FW_ALL OR FW_EXIST OR FW_PART _
                                OR FW_FOCUS OR FW_RESTORE)

        if (hwndApp = 0) then
                'runs PE.exe or prints an error message if it can't be run (run<>0)
                if (run(ProgramName)) then
                        print ("Unable to run " + ProgramName )
                        stop
                endif

                sleep (2.0)
                hwndApp = WFndWnd(Title, FW_ALL OR FW_EXIST OR FW_PART _
                                OR FW_FOCUS OR FW_RESTORE)
        endif

end SUB
'end FUNCTION  'TestBegin()


SUB ChangeDIR(Dir as string)
        WButtonClick("B&rowse...",5.0)
                Sleep(2.0)
                if (GetText(NULL) = "Choose Folder") then
                        WEditSetText( "@1", Dir, 5.0)
```

```
                    Sleep(1.0)
                    WButtonClick("OK", 5.0)
                    if (GetText(NULL) = "Setup") then
                            Sleep(1.0)
                            WButtonClick("&Yes", 5.0)
                    endif
            else
                    print "Error! Expect Choose Folder"
            endif
end SUB
```

C.3 Install Stand Alone.mst

```
'==========================================================================
' TEST CASE NAME:  Integrated Install
'
' GENERAL DESCRIPTION OF WHAT IS VERIFIED:
'       Make sure the GES32 is working after Install to a clean machine.
'
" NAME OF APPLICATION UNDER TEST:
'       GES32 Installation
'
'==========================================================================
'--------------------------------------------------------------------------
' OTHER FILES REQUIRED TO RUN TEST CASE:
'       GeoGraphix License file
'
'
'--------------------------------------------------------------------------
' REVISION HISTORY:
'
'       [ 0]    <DD-MM-YY>          <NAME>          : <CHANGES MADE>
'       [ 1]     <15-08 -99>          <Lixiu Huang>   : Created test.
'--------------------------------------------------------------------------


' '#DEFINES and '$INCLUDES for test case
'--------------------------------------------------------------------
'$INCLUDE 'main.inc'
'$INCLUDE 'util.inc'


' DIM statements, TYPE declarations, global variables,
' constants, and so on for test case
'--------------------------------------------------------------------
```

```
' ************************
' **** BEGIN TEST CASE ****
' ************************

'******* Test Case INITIALIZATION/SETUP *********
dim strText$
'dim CurrentWindow as long
'dim hwndApp as long

'make sure the view is cleared
'VIEWPORT CLEAR

if (run(INTEGRATED_INSTALL_PATH + "\Install.exe")) then
        print ("Unable to run setup.exe.")
        stop
endif
sleep(3.0)

'******* SCENARIO SECTION **********************

'*** Scenario 1:
Scenario ("Integrated Install")
        Sleep(40)
        Play "{Click 380, 300, Left}"
        Sleep(5)
        if (GetText(NULL) = "Welcome") then
                Play "{Click 380, 300, Left}"
                WButtonClick("&Next >", 30.0)
        else
                print "Error! Expect Welcome."
        endif
        Sleep(1.0)

        if (GetText(NULL) = "Software License Agreement") then
                WButtonClick("&Yes", 10.0)
        else
                print "Error! Expect Software License Agreement"
        endif
        Sleep(1.0)

        if (GetText(NULL) = "User Information") then
                WButtonClick("&Next >", 10.0)
        else
                print "Error! Expect User Information"
        endif
        Sleep(5.0)
```

```
if (GetText(NULL) = "Select Install Type") then
        WButtonClick("@@12001;1;255,0,255", 10.0)
else
        print "Error! Expect Select Install Type"
endif
Sleep(2.0)

if (GetText(NULL) = "Choose Destination Location") then
        ChangeDIR("D:\Program Files\GeoGraphix")
        Sleep(1.0)
        WButtonClick("&Next >", 10.0)
else
        print "Error! Expect Choose Destination Location"
endif
Sleep(3.0)

if (GetText(NULL) = "Choose Destination Location") then
        WButtonClick("&Next >", 10.0)
else
        print "Error! Expect Choose Destination Location"
endif
Sleep(5.0)

if (GetText(NULL) = "Select Components") then

'       hwndApp = WGetActWnd(0)
'       Sleep(1.0)
'       Play hwndApp, "{Click 157, 119, Left}"
'       Sleep(3.0)
'       Play hwndApp, "{Click 157, 136, Left}"
'       Sleep(3.0)
'       Play hwndApp, "{Click 157, 153, Left}"
'       Sleep(3.0)
'       Play hwndApp, "{Click 157, 102, Left}"
        Sleep(3.0)
        WButtonClick("&Next >", 10)
        Sleep(2.5)
else
        print "Error! Expect Select Components"
endif
Sleep(3.0)

if (GetText(NULL) = "Question") then
        WButtonClick("&Yes",10.0)
endif
Sleep(5.0)
```

```
if (GetText(NULL) = "Select Program Folder") then
        WButtonClick("&Next >",10.0)
else
        print "Error! Expect Select Program Folder"
endif
Sleep(5.0)

if (GetText(NULL) = "System Template Units") then
        WButtonClick("&Next >", 10.0)
else
        print "Error! Expect System Template Units"
endif
Sleep(5.0)

if (GetText(NULL) = "GeoGraphix License File") then
        WButtonClick("&Next >", 10.0)
else
        print "Error! Expect GeoGraphix License File"
endif
Sleep(5.0)

if (GetText(NULL) = "License File Source Location") then
        WButtonClick("OK", 10.0)          'copy license file
        Sleep(5.0)
else
        print "Error! Expect License File Source Location"
endif
Sleep(10.0)

While(GetText(NULL) = "GeoGraphix Products Setup")
sleep(25.0)                               'install Project
WEnd
Sleep(50)

if (GetText(NULL) = "Microsoft comctl32.dll Update") then
        WButtonClick("OK", 5)
endif

Sleep(150)
Play "{Click 380, 300, Left}"
'if (GetText(NULL) = "ReadMe File") then
        WCheckUncheck("Yes, I want to view the ReadMe file now.", 500)
        WButtonClick("&Next >", 10.0)

'else
'       print "Error! Expect ReadMe File"
'endif
Sleep(10.0)
```

```
        if (GetText(NULL) = "Setup Complete") then
                'select "No, I will restart my computer later"
                WOptionSelect("No, I will restart my computer later.")
                Sleep(2)
                WButtonClick("Finish", 10.0)        'shut down machine and restart it
        else
                print "Error! Expect Setup Complete"
        endif
        Sleep(5)
Scenario Cleanup 'and handling of scenario failure
End Scenario
'******* Test Case CLEANUP **********************
' ************************
' ***** END TEST CASE *****
' ************************
End
```

Appendix D

SAMPLE AUTO TESTING LOG

[Start Scenario]

[Name] Integrated Install
[Test Location] C:\MSDEV\Projects\GES32\Install\IntegratedInstall.mst (75)
[Start Time] 10/30/1998 09:07:05
[Result] PASS
[Elapsed Time] 455.444
[End Scenario]

[Start Scenario]
[Name] Create New Project
[Test Location] C:\MSDEV\Projects\GES32\ProjectExplorer\File\New.mst (66)
[Start Time] 10/30/1998 09:50:59
[Result] PASS
[Elapsed Time] 33.609
[End Scenario]

[Start Scenario]
[Name] ImportLDF
[Test Location] C:\MSDEV\Projects\GES32\ProjectExplorer\Tools\ImportLDF.mst (65)
[Start Time] 10/30/1998 10:00:54
[Result] PASS
[Elapsed Time] 33.469
[End Scenario]

[Start Scenario]
[Name] ImportLDF

[Result] PASS
[Elapsed Time] 78.323
[End Scenario]

[Start Scenario]
[Name] ImportCDF
[Test Location] C:\MSDEV\Projects\GES32\ProjectExplorer\Tools\ImportCDF.mst (66)
[Start Time] 10/30/1998 10:08:34
[Result] PASS
[Elapsed Time] 180.320
[End Scenario]

[Start Scenario]
[Name] ImportWLS
[Test Location] C:\MSDEV\Projects\GES32\ProjectExplorer\Tools\ImportWLS.mst (66)
[Start Time] 10/30/1998 10:12:08
[Result] PASS
[Elapsed Time] 251.772
[End Scenario]

[Start Scenario]
[Name] GeoAtlas ShowLandNet - Build576b
[Test Location] C:\MSDEV\Projects\GES32\GeoAtlas\Show LandNet.mst (66)
[Start Time] 10/30/1998 10:21:46
[Result] PASS
[Elapsed Time] 68.669
[End Scenario]

[Start Scenario]
[Name] Lease Map
[Test Location] C:\MSDEV\Projects\GES32\LeaseMap\LeaseMap.mst (74)
[Start Time] 10/30/1998 10:26:04
[Result] PASS
[Elapsed Time] 475.473
[End Scenario]

[Start Scenario]
[Name] WellBase layer(DKOT)

[Test Location] C:\MSDEV\Projects\GES32\WellBase\WellBase layer.mst (64)
[Start Time] 10/30/1998 10:34:29
[Result] PASS
[Elapsed Time] 371.935
[End Scenario]

[Start Scenario]
[Name] GeoAtlas ShowLeaseMap-Build576b
[Test Location] C:\MSDEV\Projects\GES32\GeoAtlas\Show LeaseMap.mst (65)
[Start Time] 10/30/1998 10:42:28
[Result] PASS
[Elapsed Time] 122.376
[End Scenario]

[Start Scenario]
[Name] GeoAtlas Show WellBase
[Test Location] C:\MSDEV\Projects\GES32\GeoAtlas\Show WellBase.mst (66)
[Start Time] 10/30/1998 10:45:08
[Result] PASS
[Elapsed Time] 184.124
[End Scenario]