Research and Practice of Distributed Test Platform

FENG QINQUN, YUN WENFANG, PENG SHEQIANG
Computer Application Department
Command Communication Academy
Wuhan, Hubei Province, 430010
CHINA

Abstract: - After concise introduction of Software test and its significance, an integration environment of Distributed Test Platform for software testing is proposed and its two departments, including Test Server and Test Driver, are described subsequently. The composition of Test Server based on Metadata Service is presented. The Metadata Service's element and its runtime principle are described in details. Finally summarization is listed.

Key-Words: - Software Test, Distributed Test Platform, Test Driver , Test Server, Metadata Service

1 Introduction

Software testing is the important approach to assure the quality of software and it's accepted widely as a "best practice" for software development. There are already many systematic, integrated and mature test theory, method and tools, but it either has limited test scope or is very expensive. It's difficult for software developers, project managers and advanced managers to control and monitor the quality of software with projects' complex improvement, and testing consumes an ever-increasing amount of time and resources. Although the investment of software test can be cut down to the lowest level and it can be delayed, the total capital invested will increase in the following procedure. Traditional testing meet the big challenge, so it's necessary to build automatic software test platform to maintain software approach, reduce the cost, and use these tools to promote software development. In fact, test platform can find the bug and hidden problem shortly, and avoid many invalidate or less-effective test repetition, what's more, it could support different operate system.

2 Distributed Test Platform

Distributed Test Platform is different from other test tools, such as local test, which local testing involves running test cases on a local computer system. Because a distributed test case consists of two or more parts that interact with each other. Each part of constructed a test packages is processed on a different system and there are interaction between the different test case components that sets distributed testing apart. Typically it is this interaction between different computer systems that is under test. All of the test cases processed on all of the different processors contribute towards a single, common, result. This is not the same as simultaneous testing. Because even though simultaneous testing involves different test case components being carried out on different processors, and contributing towards a single result, there is no interaction between the test cases or the processors. As noted above it is this interaction that sets distributed testing apart.

A further challenge that we have to face with distributed testing is that of platform. For example tester could run a test case using a Windows client to access one or more UNIX servers. So our environment has to be written at a level capable of working across all of these platforms.

2.1 Architectures

Over the past few years, tools with graphical user interfaces (GUI) that help programmers quickly create applications have dramatically improved programmer productivity. This has increased the pressure on testers, who are often perceived as bottlenecks to the delivery of software products. Testers are being asked to test more and more code in less and less time. Test automation is one way to do this, as manual testing is time consuming. As different versions of software are released, the new features will have to be tested manually time and again. But, now there are tools available that help the testers in the automation of the GUI which reduce the test time as well as the cost; other test automation tools support execution of performance tests.

Many test automation tools provide record and playback features that allow users to record interactively user actions and replay it back any number of times, comparing actual results to those expected. However, reliance on these features poses major reliability and maintainability problems. Most successful automatons use a software engineering approach, and as such most serious test automation is undertaken by people with development experience.

A growing trend in software development is to use testing frameworks such as the xUnit frameworks which allow the code to conduct unit tests to determine whether various sections of the code are acting as expected under various circumstances. Test cases describe tests that need to be run on the program to verify that the program runs as expected. All three aspects of testing can be automated.

Another important aspect of test automation is the idea of partial test automation, or automating parts but not all of the software testing process. If, for example, an oracle cannot reasonably be created, or if fully automated tests would be too difficult to maintain, then a software tools engineer can instead create testing tools to help human testers perform their jobs more efficiently. Testing tools can help automate tasks such as product installation, test data creation, GUI interaction, problem detection (consider parsing or polling agents equipped with oracles), defect logging, etc., without necessarily automating tests in an end-to-end fashion.

Test automation is expensive and it is an addition, not a replacement, to manual testing. It can be made cost-effective in the longer term though, especially in regression testing. One way to generate test cases automatically is model-based testing where a model of the system is used for test case generation, but research continues into a variety of methodologies for doing so.

Distributed Test Platform is designed to “black-box test” with two departments: Test Server and Test Driver. And implementation under test is a series of applications on different operate system. Test Server can run test suites on local or remote targets and schedule test cases (or its package test suite) automatically and repeatedly, which organized in xml language and generated automatically by the Test Scripter. Test reporter collects the information of test cases, analyze its results and create a reporter and log, which could be distributed automatically on the network. The distributed test platform architecture is displayed in Fig.1.

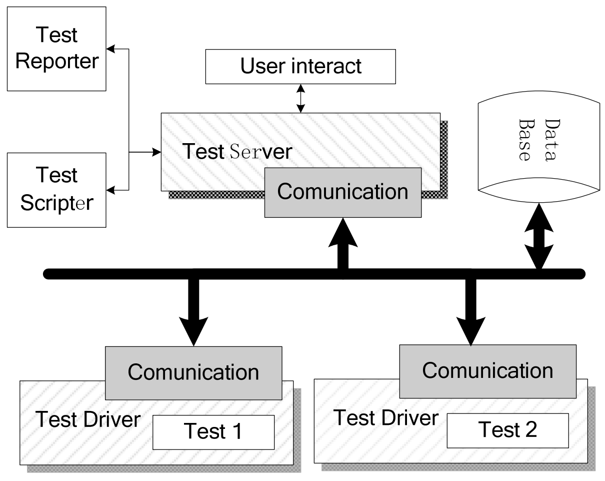

Fig.1 Architecture of Distributed Test Platform

Modules of Test Server are made up by Result Comparator, Test Scripter, Data Generator, Test Result Reporter, Communication service, Log service, Display service and Metadata service. Main modules list in Fig.2

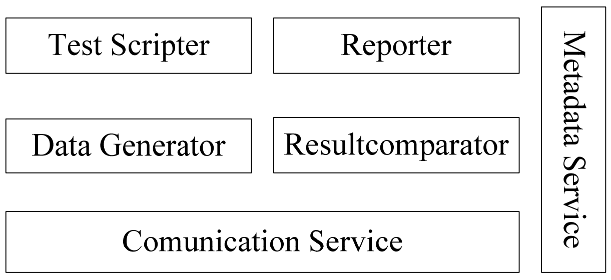

Fig.2 Main Modules of Test Server

Test Server exchanges data between these tested objects and itself and drives these tested objects through the network based communication service and metadata service. Test Driver transport the test data accepted from the server to the tested modules after building the link between Test Server and Test Driver. One of the powerful features of Test Driver is its ability of isolate, reusing. This feature allows user saving considerable time in generating the Test Driver program and stub code, which often need for the test.

After a test process is active, Test Server generates a test case using the Test Scripter, which can be created by any other text editor in the appointed format. The test case drives the test data generator to bring out test data, including input data and the expected value, which will be transferred to the Test Driver by communication service through network. Test Driver located in different operation system and different location load the tested modules with the accepted input data and executes the test procedure. Output data is collected and sent back to the comparator in the Test Server to finish the comparison between the output data and the expected value. The result and vital event will be written into the database, which will be prepared for the test record and test reporter.

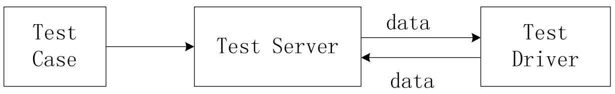

Fig.3 Test Procedure

Ensuring that all of the tests happen on all of the systems in the correct order is the greatest challenge in distributed testing. To do this we synchronize the test cases either automatically, at system determined points, or user determined points. Synchronization is the key to distributed testing and is important enough to merit separate consideration. The challenge of distributed testing lies both in synchronization and the administration of the test process: Configuring the remote systems; generating the scenario files; and processing the results to produce meaningful reports. Being able to repeat the tests consistently, and to select tests for repetition by result i.e. regression testing. And doing this repeatedly and across many different platforms, UNIX, Windows and Linux and so on.

2.2 Test Server and Test Driver

Software test automation involves automating a manual testing process that uses a formalized testing procedure. Test automation involves the use of software in order to control the execution of the tests, setting up the pre-conditions for the test, comparisons of the predicted outcomes to the actual outcomes and many more test reporting functions.

Test Server and Test Driver are designed based on Black box testing, which is a type of testing in which the system is considered as a 'black-box' and the testers don't use their knowledge of internal structure or code to validate the application against the specifications. Black box testing is also termed as 'behavioral testing' as it mainly focuses on the functionality of the system as a whole. However, the tests can be non-functional as well. The black box testing method is applicable to all the levels of the software testing i.e. unit, integration, system, functional and acceptance testing.

Some of the common black box test design techniques are: Equivalence Partitioning. In this, the input domain of a program is divided into classes of data which can then be used to derive the test cases. In boundary value analysis technique, the extreme boundary values are chosen as the systems have a tendency to fail on the boundary. The extreme boundary values include minimum, maximum, typical, just inside/outside and error values.

Some of the advantages associated with black box testing are: Once the functional specifications are complete, it is fairly easy to design the test cases.

It is simple to check the contradictions present between the specifications and the actual system.
Even a non-technical person can perform black box testing as internal structure knowledge is not required to carry out the black box testing.
Some of the disadvantages associated with black box testing are: Writing test cases is a slow process as it is difficult to identify all the possible inputs in a limited time.

Black box testing requires the test inputs to be from large sample space. Since the internal structure knowledge is not required to carry out the black box testing, there's every chance of having some un-identified paths during the testing which can lead to degradation in the performance. To conclude, black-box testing is recommended to test the functionality of the system as a whole and it comes with its own set of advantages and disadvantages.

3 Test Case (Test Scripter)

The test script is the combination of a test case, test procedure, and test data. Initially the term was derived from the product of work created by automated regression test tools.

A test case in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system meets specifications. It may take many test cases to determine that a software program or system is functioning correctly. Test cases are often referred to as test scripts. Written test cases are usually collected into test packages. In order to fully test that all the requirements of an application are met, there must be at least one test case for each requirement unless a requirement has sub-requirements. In that situation, each sub-requirement must have at least one test case.

3.1 Test Case and It's Automatic Running

Test automation is not a panacea for all software testing challenges; however, it is an important consideration in every test strategy, along with organizational readiness, test process maturity, and expertise in testing tools. We find that many organizations acquire a testing tool without first establishing the objectives, how test automation fits in with manual testing, and how to optimize the tool's use given the organization's test maturity level. When designing a test automation suite, we consider its applicability across the IT portfolio, the need for ongoing maintenance, scalability and effectiveness of test execution

The major advantage of Automated testing is that tests may be executed continuously without the need for a human intervention. Another advantage over manual testing in that it is easily repeatable, and thus is favored when doing regression testing. It is worth considering automating tests if they are to be executed several times, for example as part of regression testing.

Disadvantages of automated testing are that automated tests may be poorly written and can break during playback. Since most systems are designed with human interaction in mind, it is good practice that a human tests the system at some point. Automated tests can only examine what they have been programmed to examine. A trained manual tester can notice that the system under test is misbehaving without being prompted or directed. Therefore, when used in regression testing, manual testers can find new bugs while ensuring that old bugs do not reappear while an automated test can only ensure the latter.

One shouldn't fall into the trap of spending more time automating a test than it would take to simply execute it manually, unless it is planned to be executed several times.

The method to generate test cases of software testing must be operational and high efficient. During the process of software testing, the generation of test case is critical and difficult. Presently, generation of test case is primarily done manually, relying heavily on testers' experience and capability. Techniques of generating test cases automatically is very significant as it can reduce the time and cost of testing. Test case's systematic generation using Test Scripter could be automatic. The Test Scripter is an application embedded in the Test Driver. The Scripter is an independent application, and it's embedded in the Test Server as listed in Fig.2. Any text editor can create a script, the Scripter can compile and check it. Test cases are serialized by the test script, and these serialized test cases make up of a test package. The test package includes the input data, output data and expected data. What characterizes a formal, written test case is that there is a known input and an expected output, which is worked out before the test is executed. The known input should test a precondition and the expected output should test a postcondition.

3.2 Scripting Languages

The test case is the description, recorded a test case or its procedure (or package). It's a text coded by XML language and is generated by the Scripter. such as Perl, Python, Tcl, and Java are very popular in the programming industry as a whole, as well as within test organizations, since they facilitate a rapid development cycle.1 As programming languages, scripting languages are not intended to directly solve either reuse or automation. Additionally, they are not directly targeted at the test environment, although their generality does not preclude their use in a test environment. Despite these limitations, we felt that given the wide popularity of scripting languages and the almost fanatical devotion of their proponents, we should examine their potential for solving our problems.

Although scripting languages are not a direct solution to reuse or automation, scripting languages do have some general applicability to the problem of reuse. To begin with, they are available on a wide variety of operating systems. They also have large well-established sets of extensions. Although not complete from a test perspective, these extensions would provide a solid base from which to build. Additionally, some languages (notably Tcl and Java) provide support for dealing with multiple codepages.

The benefits of scripting languages would clearly place them in category 3 of our preferences. Unfortunately, these benefits are only available if one is willing to standardize on one language exclusively. As was mentioned earlier, our testers create tests in many different programming languages, and it would have been tremendously difficult to force them to switch to one common programming language. Even if we could have convinced all of the testers on our team, we could never have convinced all the testers in our entire organization (much less those in other divisions, or at other sites), with whom we hoped to share our solution. Therefore, we were unable to rely on scripting languages for our solution

The following is a part of a test case organized by xml. It declares an int8 object, generates a group of normal test data subsequently, which will be sent to the tested module.

```
<datatype>
  <classname>Int8</classname>
  <vary> inData  </vary>
<method>
  <name> GeneratNormaData() </name >
  <parameters>
<paratype></paratype>
<paraname></paraname>
</ parameters >
```

</method>
</datatype>

Of course, under special circumstances, there could be a need to run the test, produce results, and then a team of experts would evaluate if the results can be considered as a pass. This happens often on new products' performance number determination.

4 Test Data Generator

Test data used by test cases is generated by an isolated application named Test Data Generator. In order to easily extend the data generate methods, a data generating container is designed in Test Server, which contains some data generating strategies.

4.1 Data Generating Strategy

Performance tests require very large data set. Preparing proper test data is a core part of "project test environment setup". Tester cannot pass the bug responsibility saying that complete data was not available for testing. Tester should create test data additional to the existing standard production data. Test data set should be ideal in terms of cost and time. Data Generator is a automatic designing test data toolbox. Data Generator is designed references below: 1) No input data: test cases is scheduled on blank or default data to see whether the proper error messages are generated or not. 2) Valid data set: Create it to check if application is functioning as per requirements and valid input data is properly saved in database or files. 3) Invalid data set: Prepare invalid data set to check application behavior for negative values, alphanumeric string inputs. 4) Illegal data format: Make one data set of illegal data format. System should not accept data in invalid or illegal format. Also check proper error messages are generated. 5) Boundary Condition data set: Data set containing out of range data. Identify application boundary cases and prepare data set that will cover lower as well as upper boundary conditions. 6) Data set for performance, load and stress testing: This data set should be large in volume. This way creating separate data sets for each test condition will ensure complete test coverage.

4.2 Test Data Generator

The container is maintainable, reusable and extendable. Making a good choice of test data is the key factor of effective test case, the scope of input parameters is determined after analyzes their values and types. The input of these tested units is classified according to the principle of equivalent class. Every test data represents a set. The normal value, exception value and boundary value construct the set of input value. The input data enumerate all above types of data. Invalidate input data is eliminated and the clearing of redundant data makes the test be more efficient.

One good way to design valuable test data is use the existing sample test data and append new test case data each time to get same module for testing. This is the way to build a comprehensive data set. Test Data Generator can remember all of the tested data and could repeat it all.

5 Data Comparator

After a test is finished the test result must be identify its way that the outputs data. This data doesn't have to be text displayed on the screen, it could be binary data in the form of a file, network stream, or other. Test result comparator's main function is getting the difference between output and the expected value, it decides whether the test case is successful or not. The specific comparator method is important to the test system except for these tools operation system provided to check the result. The principle of comparing is clear when the test result is of simple data type, while the comparing principle is complicated as the output data is of complicated style. The comparing method container equips the existing comparator, which overcast static compare and dynamic compare of the simple data type, users can custom or reinforce their own comparators.

5.1 Strategy Pattern

In order to aim the dynamic selecting compare method from the comparing method container, a strategy pattern is used in the Data Comparator designed. In computer programming, the strategy pattern (also known as the policy pattern) is a particular software design pattern, whereby algorithms can be selected at runtime. The strategy pattern is useful for situations where it is necessary to dynamically swap the algorithms used in an application. The strategy pattern is intended to provide a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. The strategy pattern lets the algorithms vary independently from clients that use them.

For the comparing principle is complicated as the output data is of complicated style, the comparing method container must be opened and dynamic to allows tester to switch the algorithm in using at any time.

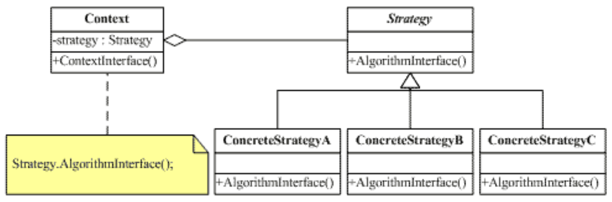

fig.4 – Strategy Pattern

The application must be aware of all the strategies to select the right one for the right situation. Strategy and Context classes may be tightly coupled. The Context must supply the relevant data to the Strategy for implementing the algorithm and sometimes, all the data passed by the Context may not be relevant to all the Concrete Strategies.

Context and the Strategy classes normally communicate through the interface specified by the abstract Strategy base class. Strategy base class must expose interface for all the required behaviors, which some concrete Strategy classes might not implement.

In most cases, the application configures the Context with the required Strategy object. Therefore, the application needs to create and maintain two objects in place of one. Since, the Strategy object is created by the application in most cases; the Context has no control on lifetime of the Strategy object. However, the Context can make a local copy of the Strategy object. But, this increases the memory requirement and has a sure performance impact.

Take into consideration that data outputted to a file may come from a location modifiable by the user and during the input phase is properly sanitized and appear harmless, however could cause damage to other portions of your application trying to read this file later on.

6 Test Reporter

The test report is the primary work deliverable from the testing phase. It disseminates the information from the test execution phase that is needed for project managers, as well as the stakeholders, to make further decisions. Anomalies and the final decomposition of the anomalies are recorded in this report to ensure the readers know the quality status of the product under test.

Test reporter can expediently generate test specification and test report after analyzes the collecting test logs and test input/output data. It provides some default test report document templates, and users can custom their own templates, or develop specific test record and report according to test data and these open interfaces.

Test Reporter is organized by xml file type, and can be published into two formats files: doc and html. And the Reporter contain the test summary identifier, objective, summary of testing activity, variances, testing activities and last but not least, the important piece of information, defects.

Test summary identifier need to be associated on each round of testing. In other words, each round of testing must have a unique identifier to ensure readability and traceability. Objective -- This is the objective of each round of testing. Does this round of testing cater for component testing, system testing, regression testing, integration testing or others? Test Summary includes the summary of testing activity in general. Information detailed here includes the number of test cases executed, the scope of testing, the number of defects found with severity classification, and test environments set up and used for the testing. Variances explain if there's a discrepancy between the complete product and the requirement. Variances can be on the plan, procedures and test items. Summarize all major testing milestones such as Test Plan, Test Case Development, Test Execution and Test Reporting in this section. Information on resource consumption, total staffing level and total lapsed time should be reported as well. Defects are the most essential section in the report. This is where you report defect information such as the number of defects found, defect severity classification, defect density, etc. Test metrics are important to complement this section. In general, the test report is important to make sure readers can make correct conclusions based on it.

6.1 Test Reporter Analysis

Software test results analysis plays a very important role in software development. The test process itself is crucial to the success of new software products. It is only through efficient system testing that the quality and safety of an application can be guaranteed. For software businesses, there is no excuse for a poor quality. This is the reason why companies use several testing methods - to perfect their product before introducing it to the market.

However, it must be noted that the process does not end with just system testing. A common mistake is forgetting the analysis of test results. While the tests find errors in the application, it is the analysis that interprets that same error. Without the interpretation, the testing conducted would be useless. To further explain the reason why failure to get the analysis is a mistake, let us take the software interface test.

Though there are many testing methods available, companies and developers have considered software interface testing as the most important system test to ensure the quality of a program. The interface is composed of sets of commands, images, messages, and other features that permit communication between the user and device. According to developers, the very advantage of interface testing is in the fact that it is anchored on the feedback of end users themselves. There are two characters involved in the process - the user and moderator. Each of them is assigned their respective tasks that are vital to the progress of the software application. First, the user is allowed to use the program. He or she is then tasked to note down comments about the program, its individual features and its general usage as well. These comments should be on how easy and efficient the program is when being used. Navigating can also be an area where the user can issue comments. The more detailed the comments of the user are, the better. Likewise, all aspects of the program must be looked into. Some of these aspects include functionality, being user-friendly, and the performance of the new application. The moderator, on the other hand, conducts the test. It is not necessary for him or her to communicate with the user. What he or she is assigned to do is document all the comments of the user regarding the program. At the end of the testing period, the moderator is expected to endorse the feedback made by the user to the developer. Here analysis starts to play a role. The comments must be interpreted in such a way that the error in the program should be given an appropriate solution.

With the example on software interface testing, developers would definitely need analysis of the test result so he or she could make the corresponding revisions based on the feedback of the user. Remember that the aim of any developer is to perfect the new software application. This can only be done with comprehensive testing processes. Likewise, firms are very keen on improving the product quality. This is because their possible earnings from the software would rely so much on its potential. Thus, companies would require software test results analysis to be able to ensure quality and guarantee good returns.

6.2 Bug Trace

Finding the cause of a bug can be one of the most time-consuming activities in design verification. This is particularly true in the case of bugs discovered in the context of a random-simulation-based methodology, where bug traces, or counterexamples, may be several hundred thousand cycles long. So it’s important that the Test Reporter could record bugs details in it. Both the positive and negative results of testing can be tracked in order to provide a clear status to the project manager. As tests are run, the tester notes which tests passed and which failed; the failed tests result in documented defects. As defects are fixed and those fixes promoted into the test environment, tests are rerun. Test results tracking can let the tester see at a glance if previously failed tests now pass, or if previously passed tests now fail. In addition to tracing test cases back to requirements, a Baseline Traceability Matrix is a useful tool to track the status of tests. Testing can result in different kinds of issues, including defects, questions, actions and enhancements (feature requests). All these types of issues can be tracked in the same way, as long as there is some way to 'tag' the category.

Defects are logged and monitored from discovery to resolution (or stagnation). Once defects have been logged, and depending on the data captured, statistics can be gathered to track metrics such as increasing or decreasing defect counts, peak trouble times in a product's cycle, or volatile modules. For example, once you've captured details about a problem, if you see the same problem arise in production after code fixes, your regression tests need to be tweaked. Tracking defects arising out of tests is usually done with some kind of tool

7 Metadata Service

MS is the Metadata Service that that stores descriptive information about different data format and provides registration, retrieval, of metadata elements, metadata schemata, and metadata profiles of resources. It’s aim of MS is to support different operate system. MS is a standalone that stores information about data type. It also allows users to aggregate the data items into collections MS provides system-defined as well as user-defined attributes for logical items and collections. One distinguishing characteristic of MS is that users can dynamically define and add metadata attributes. MS can also provide the names of the user-defined attributes. As a result, different MS instances can be created with alternative contents. MS have been implemented to run on top of standard services. It provides secure access to the metadata.

MS may be used for storing and accessing metadata based data access and the particular storage system where the data resides.

7.1 Construction of MS

In order to adapt the test platform to different operate system, the data architecture, description and operations of metadata are defined in an abstract class, which exhibit a series of Methods, Events and Properties. The data description in binary module is independent of operation systems, which result in the test platform is independent of developing languages and operation systems. The metadata service is a part of test platform and they make test platform run smoothly. And all the other data types in different operations could be derived from the abstract class.

The *CTType* class is the main class for creating metadata object in runtime. The *CTType* class is an abstract class and that represents a type. By using this class, we could find the type name, the types used in a module (an assembly may contain one or more modules), and to see whether a given type is a value or a reference type, and so on. It also allows us to query the type's fields, methods, properties, and events by parsing the corresponding metadata tables. Serialization mechanism uses creating metadata object in runtime to determine what fields a type defines. The serialization formatter then can obtain the values of these fields and write them into the byte stream.

Late bindings can be achieved by using creating metadata object in runtime. For example, in some applications, we don't know which assembly to load during compile time, so we ask the user to enter the assembly name and type during run time and the application can load assembly. For this purpose, the CTType class offers three static methods that allow you to explicitly load an assembly: *Load*, *LoadFrom*, and *LoadWithPartialName*. These methods are something similar to the *LoadLibrary* Win32 API.

C++ object module is derived from simply object module. Every class will create virtual functions in a virtual table. The virtual function address is generated in the compile procedure and an index is assigned to every virtual function. In the runtime, the pointer to virtual table is found first and then the pointer to virtual function according to the virtual table, so the function is executed by the function index.

The data type objects are inherited from an abstract class: CTType, a child class of the CObject class. The Test Driver could explain all the data received from the other type operation system because the data organized in binary module. So, it's only need to expand the CTType's child class when need to meet the acquirement of adding new data type.

CTType is designed with series of virtual function which implement the polymorphs is defined like below:

```
class CTType : public CObject
{
	DECLARE_DYNCREATE(CTType)
public:
	static CRuntimeClass* PASCAL FromName(LPCSTR lpszClassName);
	static void GetValidedTypeList(CStringList &strTypeList);
	virtual DWORD GetTypeLength() const ;
	virtual BOOL PASCAL SetFixedValue(CTParameters &tParams);
	virtual BOOL PASCAL GenerateExpValue(CTParameters &tParams);
	virtual BOOL PASCAL ToString(CTParameters &tParams);
…
};
```

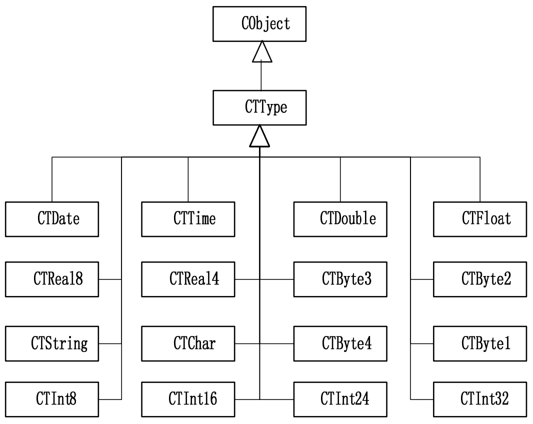

fig.5 – composite of metadata

7.2 C++Macro Implementation

When write a class in Visual C++, the Design Time Environment thinks of it as several things, including a class and a CodeElement. It has base classes, each of which is also a CodeElement. Your class, its bases, and even its functions can all be represented as objects with properties such as Name and Type-and your macros can manipulate those objects. For example, an object that represents a class has a method called AddFunction(), which actually adds functions to one of your classes, right from the macro. That's the heart of this macro: a loop that calls AddFunction() repeatedly to add each function that's in the interface your class implements. This generates the code inside your class, and it's quite fun to see. Some macros used in Distributed Test Platform are listed below:

DECLARE_CLASSSERVICES Enables objects of CObject-derived classes to be created dynamically at run time. The framework uses this ability to create new objects dynamically. For example, the new view created when you open a new document. Document, view, and frame classes should support dynamic creation because the framework needs to create them dynamically.

Add the DECLARE_DYNCREATE macro in the .h module for the class, then include that module in all .cpp modules that need access to objects of this class. If DECLARE_DYNCREATE is included in the class declaration, then IMPLEMENT_DYNCREATE must be included in the class implementation.

IMPLEMENT_DYNAMIC Generates the C++ code necessary for a dynamic CObject-derived class with run-time access to the class name and position within the hierarchy. Use the IMPLEMENT_DYNAMIC macro in a .cpp module, and then link the resulting object code only once.

RUNTIME_CLASS Gets the run-time class structure from the name of a C++ class.

7.3 Creating Metadata Object in Runtime

The dynamic run of the metadata object is simulated the c++ module by macro implement. A functions table including virtual function address and other information, such as counts of parameters and its type, is built in the runtime. When it's ready for run, it must search in the table from the head for the enter address by compare function name in the table, if the target is found and the function is actived by its index. The function virtual table is listed below in fig.6

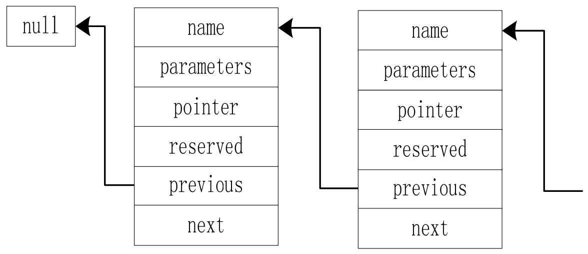

Fig.6 functions list

The saving function structure composed of function name, parameters counts, function pointer and two pointers to itself like below.

```
template<class T>
struct CClassService
{
LPCSTR m_lpszServiceName;
int m_nParameters;
unsigned int m_wSchema;
BOOL (PASCAL T::*m_pServicePointer)(void *p);
        static CClassService *m_pFirstService;
        CClassService *m_pNextService;
}
```

The dynamic steps of metadata object creating is listed below.

(1) Using macro DECLARE_CLASSSERVICES in the class define. The macro add a static vary to saving the information for the metadata dynamic create.

```
#define DECLARE_CLASSSERVICES(class_name, service_name) \
public:\
        static CClassService<class_name> service##service_name;
```

(2)Using macro IMPLEMENT_CLASSSERVICES in the class implement.

```
#define
IMPLEMENT_CLASSSERVICES(class_name,service_name, nNum,pfnNew)\
static char _lpsz##service_name[] = #service_name;\
CClassService<class_name>
class_name::service##service_name = \
{_lpsz##service_name,
nNum,0xFFFF,class_name::service_name,pfnNew};
\
static AFX_SERVICEINIT<class_name> \
        _init##service_name(&class_name::service
##service_name);\
```

For example, Test Server could create an "Int8" object in running a test case, and the "GeneratNormaData" functions define in the test case will be dynamic executed by searching for its address in the virtual table.

```
BOOL (PASCAL CTType::*pF)(void *p) ;
if( ((CTType *)pObj)->IsKindofServices(strNameOfService))
{
        ((CTType *)pObj)->GetFunctionPointer(strNameOfService,pF);
        (((CTType *)pObj)->*pF)(NULL);
}
```

The vary strNameOfService is read from a test case.

8 Key Features

Distributed Test Platform is composed with Test Server and Test Driver based on intranet/internet. It has some special features. The test platform includes extensive networking support. Test systems can network together to share test results and provide factory line monitoring.

The test platform has a geographically distributed test development against traditional setting of a standalone development. This provides improved collaboration and management of test assets. Web based development and user administrative options allows authorized team members at Site B to view, create or modify test scripts and data created by Site A team members. The test platform is designed from the ground up to manage parallel execution and test cases can run fully isolated and so they do not crash the executive.

Quality assurance team at each site can manage all aspects of testing, from initial test case test development, execution and analysis of test results from a central server. Distributed environment allows team members to share assets. The QA teams can perform functional, performance, web services, system and regression testing of the web applications/web sites. its architecture is cost effective to deploy and manage as it eliminates the installation and updates on each site. what's more, Both parametric and log data are automatically stored in the database on every test system. A variety of search and reporting functions work out of the box.

In short, Distributed Test Platform saves time, cuts costs, reduces risk and improves quality of test automation.

9 Conclusion

The software test has the important position in the whole software development procedure, but it's so expensive, labor-intensive, and times consuming that developer often leaves it out. It's valuable to develop an automated test tool which is effective, reusable and maneuverable. The Distributed Test Platform is the simplest to understand and to implement. It's also the easiest to incorporate with the other software framework. The Platform has the advantage of reuse and the reduced maintenance costs that come with reuse. Distributed Test Platform has been researched out with its merits. It generates test case scripted with xml and test data, it invokes test cases automatically and report the test result intelligently and automatically in xml or html format. Moreover, it's independent of developing languages and operation systems, and it can be easily integrated into other software platforms to promote the implementation of software procedure.

Distributed Test Platform is made up of Test Driver and Test Server which constructed of Test Scripter, Data Generator, Comparator and Reporter. And all its units are supported by Metadata Service, which includes a series of Methods, Events and Properties enabling Distributed Test Platform to accommodate different operation system. The server can run test suites on local or remote targets and log progress and results to HTML pages. The main purpose of Test Server is to act as engine inside customized test tools. A callback interface for such framework applications is provided.

Meanwhile, Test Data Generator can't meet users' requirements fully, and the tactics and methods of data generation need to emend and strengthen. It has been proved that the test platform can actually finish most of the Black Box test.

References:

[1] Rovert V. Binder, Testing Object-Oriented Systems:Models,Patterns, and Tools, 2001

[2] Stanley B.Lippman, C++ Primer, 2006

[3] David Chappell, Understanding .NET, Addison Wesley Longman, 2002

[4] Kit,Edward. Integrated Effective Test Design and Automation Software Development，1999

[5] Graham, Dorothy, MarkFewster. Software Test Automation: Effective Use of Test Execution Tools. Boston, Mass:Addison Wesley, 2000

[6] http://www.rational.com

[7] http://sourceforge.net/projects/httpunit/

[8] M.S.Lin, M.S.Chang, D.J.Chen, "Distruibuted-program Reliability Analysis: Complexity and Efficient Algorithms", IEEE Transaction on Reliability, 1999, 48(1):87-95

[9] http://msdn.microsoft.com

[10] Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

[11] Larman, Craig (2005). Applying UML and Patterns. Prentice Hall. ISBN 0-13-148906-2.