

Automating Regression Testing for Real-time Software in a Distributed Environment

F. Zhu

S. Rayadurgam

W. T. Tsai

Department of Computer Science

University of Minnesota

Minneapolis, Minnesota

Abstract

Many real-time systems evolve over time due to new requirements and technology improvements. Each revision requires regression testing to ensure that existing functionality is not affected by such changes. Testing these systems often require specialized hardware and software, and both are expensive. While the overall regression testing process is similar across different organizations, the strategies and tools used by them vary according to their product needs. Hence a good framework for regression testing should provide the flexibility to configure it depending on the particular organization's needs while at the same time maximizing utilization. Manual processes are typically slow and error-prone and result in under-utilization of valuable test resources. This paper proposes an automated distributed regression testing framework that provides flexibility to the user to configure it to their needs while at the same time optimizing resource usage.

1 Introduction

When software is modified it must be tested to verify that the changes do not have undesirable effects on the functionality of the software. Test cases from test repositories are selected and run. Anomalies are detected by comparing the results of the execution with the expected results. For real-time systems, regression testing is even more significant. Unlike the run-of-the mill applications software for which *beta versions* of upgrades are *released* to end users for testing and evaluation, safety-critical embedded systems like pacemakers cannot be tested on patients. Statutory regulations are stringent and they must be met before the product can be released to the public. Hence upgrades to such systems take much more time due to testing. Further, testing of these systems sometimes requires specialized hardware and software, and both can be expensive. Due to increased competition, developers are looking for new techniques and tools that would cut down their testing cycle time and enhance productivity and utilization of test resources, while at the same time maintaining the quality of products.

In many real-time embedded systems, the control software is stored in non-volatile memory and is referred to as firmware. The testing environment for firmware of real-time embedded systems is usually different from its development environment. The reason is that testing of firmware requires special hardware. These are usually too expensive to be deployed at each developer's workstation. These specialized support tools are not required when the software is written. Further, the devices used to test a particular product may not be useful for a different product, even if the development and testing processes are the same. Hence the test environment is built and maintained separately. This results in the need for transporting test and data files from the developers' computers to the test environment and the results back to the developer. Doing this manually can be slow and error-prone.

Different companies produce different products using different technologies. Even companies producing similar products may use different tools to test their firmware. However, their goal of firmware testing is the same: to make sure that the design meets the requirement. Often their overall testing processes are similar.

This paper proposes a framework for the automation of regression testing for real-time software in a distributed environment. The authors have been involved in testing real-time safety-critical systems in an industrial environment [Tsai 97, Poonawala 97]. Cardiac Pacemakers Inc. (CPI) of Guidant Corporation develops safety-critical medical devices such as pacemakers and defibrillators. In the past, medical devices were mostly implemented by hardware, but modern devices are large and complex, and many of the features are implemented by software. Both the hardware (such as battery, capacitors and leads) and software must be reliable and safe because the consequences of failure can be serious.

A typical medical device (such as pacemakers and defibrillators) consists of two parts, a PG (Pulse Generator) which will be implanted, and a Programmer, which is a device used to set parameters in PGs. A typical PG contains capacitors, a battery, a CPU and computer memory. A Programmer consists of a user-friendly graphical user interface, various checking software to ensure that the parameters entered by physicians are safe and consistent, and devices for communicating with the PG. Both the PG and the Programmer are safety critical, and must meet various regulators' guidelines (such as FDA and ISO rules) for reliability, safety and documentation standards.

Due to the stringent requirements for reliability and safety, CPI follows the traditional waterfall-like development process. The process starts with capturing user requirements, followed by system analysis, system design and decomposition into mechanical, electrical and software sub-systems. After decomposition, each sub-system has its own requirement analysis, design, implementation and multiple

stages of testing. Finally, the sub-systems are integrated and it is followed by system and field-testing. The requirements are rigorously traced from the beginning of the system development process to the end to ensure that the stated requirements are fulfilled. Safety analysis is performed at each stage to ensure that safety requirements are met.

Safety is an important issue in critical systems like pacemakers. Any failure in hardware or software could cause considerable harm to patients, doctors and nurses. Hence products are subject to intensive testing. Timing is another important issue. In real-time systems tasks have deadlines. These could be hard real-time constraints (e.g. pacing pulse should be delivered within 10 milliseconds from the detection of an arrhythmia) or soft real-time constraints (e.g. no single task in the system can execute for more than 8 milliseconds). The testing process should handle such safety and timing issues. Hence CPI has adopted a multi-level checking approach. Testing is performed at unit, component, subsystem, and system levels. Further, acceptance testing is performed before the product is released. From the experience gained in developing safety-critical systems, CPI has found that testing safety and timing is quite tricky. Over the years CPI has built a huge database of test cases. When new products are developed, relevant test cases from the database are selected and run. These test cases were developed by analyzing usage patterns obtained from field studies.

The current testing process involves considerable manual effort. While parts of the process have been automated many activities are carried out manually. A major reason for this remaining manual is that the process is quite effective in addressing the prime concern, quality. Engineers are able to produce defect-free products, albeit taking a longer time. New techniques are typically looked at with some healthy skepticism. Our current effort is to propose a viable alternative that would take less time without compromising product quality. We are proposing a framework for regression testing that can be profitably employed in this environment. The objectives of this work are:

- To optimize utilization of test resources
- To speedup the testing process
- To automate the testing process and free engineers from tedious and clerical work
- To provide a uniform testing environment, even for testing different products or product families.
- To enable simultaneous testing of different products.

In this paper, a framework for regression testing in a distributed environment is explained. It provides a systematic approach for software and firmware testing in a distributed environment and has the potential of automating the whole process of regression testing. The framework has been designed using an object-oriented approach. We have carefully abstracted parts of the process that can be generalized and provided a common foundation over which testing systems can be built.

Before we discuss the framework itself we briefly review the regression testing process on which the framework is based. In section 3 the framework is presented in detail. Section 4 explores some possible applications of this framework and presents the results obtained by implementing parts of the framework in an industrial environment.

2 Process of Regression Testing

While different organizations follow different processes to develop and maintain software -- some use the waterfall model, some the spiral model, or variants of these models -- most of them follow similar processes for software testing. A necessary but expensive software testing activity is regression testing. This is performed on modified programs to ensure that changes are correct, and have not affected unmodified portions of the software. [Onoma 96] lists the following steps in a typical regression testing process:

- **Modification Request** – The software can be changed only if a defect has been found or there is a change in specification or design. Often requests for changes originate from field-support engineers.
- **Software artifact modification** – The software artifacts such as requirement documents, design documents and the code must be changed to meet the new requirements or to remove the known defect. Often, the source code is the central focus [Joiner 94, Onoma 95], and once it is changed every other document should be updated to reflect the change in the code. For regression testing, changes to the design and changes to the test cases are done by different teams independently. The changed test cases are then used to test the changed implementation.
- **Test case Selection** – Relevant test cases must be selected to run regression testing. Test cases are associated with software components. Changing a software component could potentially alter the results of its associated test cases. Hence, when a software component is changed all its associated test cases are candidates for re-testing. The goal of regression testing is to determine a minimal set of test cases that need to be rerun and select those to run against the new implementation. When changes involve addition of new software components, new test cases would have to be developed to test

them. In this case the goal is to obtain the right test cases instead of minimizing the number of test cases. Researchers are mostly concerned with reducing the number of test cases that have to be rerun after changes. This is regarded as the “test case selection” problem in [Rothermel 94a, 94b] and [White 93].

- **Test case building and execution** – After the test cases are selected, they will be built as executables for the testing platforms and will be scheduled to run. Since the number of test cases is often large, the building of test cases is usually automated and is carried out during off-peak hours such as evenings and weekends. Test cases are often scheduled to run in batches during off-peak time and their results are collected and recorded for later verification.
- **Failure identification** – Results from the execution stage must be compared with expected output to see if modifications to the software produced any undesired effects on the software. If the actual output is consistent with the expected output, it enhances our confidence in the belief that the existing functionality did not get affected by changes. If not, we need to examine whether the test case, the code, or both are faulty. If test cases have not been revalidated earlier, they will be revalidated at this time, especially the ones that failed.
- **Fault identification** – If the source code is suspected to be faulty, the programmer needs to examine the component of the software that caused the test case to fail. Fault identification for small programs is not an issue. However, it can be a difficult problem if we have many modules with multiple versions with many modifications submitted for regression testing. It is a non-trivial task to identify the right components, the right versions and the right modifications that caused the failure.

Fault mitigation – Once the components that caused the failure are identified, we may remove the fault by either removing the modifications, or proposing a new modification. Whatever we do at this stage will result in another round of regression testing.

The process is illustrated in Figure 1.

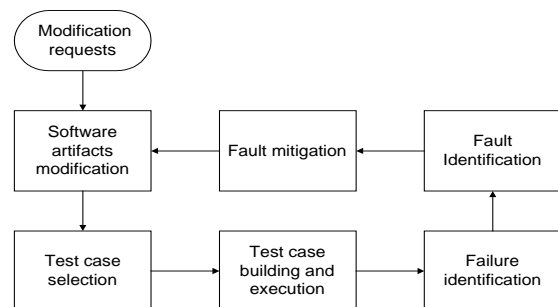


Figure 1 Process of regression testing

3 Distributed Regression Testing Framework

There are many important issues in performing regression testing in a distributed environment. A typical distributed testing system is shown in Figure 2. Test cases run on test stations. These may be general-purpose computers or some specially designed systems. A test station may be configured with special hardware devices and be capable of running certain type of test cases. For example, it may provide interfaces to the firmware against which the test cases are executed.

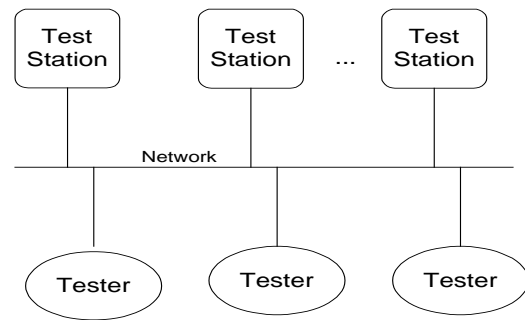


Figure 2 A distributed testing environment

Physically, test stations are wired to a local area network to which testers also have access. Such a set up allows testers to move test cases and test results between developers' computers and test stations. Usually the regression testing process is the responsibility of a group of test engineers. Programs and scripts are used to automate certain routine tasks in the process, such as test case building, test case execution and test results processing.

The following issues have to be addressed in doing regression testing in a distributed environment:

- **Test case allocation** – Test cases that are selected for regression testing need to be assigned to certain test stations to run. Each test station in the system is configured with a set of hardware devices that decide what test cases the station can run. Also, each test case requires certain hardware devices to run. A test case is said to be executable on a test station if the hardware configuration of the station satisfies the hardware requirement of the test case.

Sometimes, several test cases are executable on only one test station. In this situation, we need to have a test queue that serializes the execution of the test cases.

- **Test load balancing** – It is often the case that one test case is executable on multiple test stations. The question now is how to allocate test cases to test stations so that we can have the best utilization of the test resources. Several criteria exist regarding the optimizing test case execution and test resource usage, such as average test case waiting time, test case completion time, test station idle time, and test case allocation with priority.
- **Test interruption and recovery** – During a batch execution of test cases, it is possible that some test cases fail. Failed test cases may cause suspension of further execution of test cases or even affect the

correctness of test cases that follow them. This requires that test cases be executed in a controlled way so that recovery is possible in case of failures.

- **Composite test cases** – It is usually the case that a group of certain test cases are required to run in a specific order. For example, test case A may be run independently. But test case B may require the data values that test case A left in the raw registers. So the order of the execution in this case must be $A \rightarrow B$.
- **Dynamic test station configuration** -- During testing process, it often happens that the configurations of test stations have to be changed. For example, we may need to upgrade certain devices attached to a test station or we may need to switch some devices between test stations in order to make certain verifications, which is usually the case when we are doing firmware testing. This dynamic reconfiguration must be achieved with minimum user intervention, if at all possible.

The framework we are proposing here is a generic design of a distributed regression testing system that can be used virtually by any organization to perform regression testing in a distributed environment. It is based on the authors' experience of developing a test management system to automate regression testing process in a distributed environment for cardiac pacemaker firmware. Users can configure the framework by plugging in their own testing platform and define their own test cases to build up distributed testing environments that are specific to their application.

3.1 Components of the Framework

The framework we are proposing runs in a distributed environment as shown in Figure 3.

- **Test server** -- The center of the framework is the test server, which has access to the test database that stores the system configuration and test case source files.
- **Test stations** – Test stations are platforms that actually execute test cases.
- **Test clients** – Test clients provide interfaces to test engineers to define test cases, submit tests, and monitor and control test case execution. Test clients also have access to the test database.

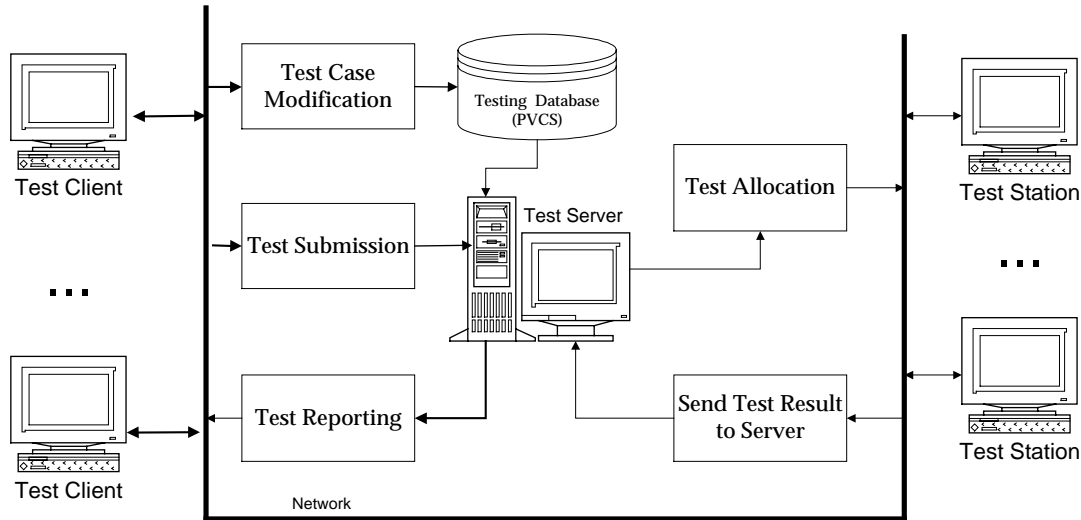


Figure 3: Regression testing in distributed environment

All the test clients and test stations are connected to the network which can be a LAN or the Internet. Whenever a test engineer makes some changes to the test source files, the changes will be saved into the test database. This change will take effect in the next round of test case building by default. When specified, we can also do test building using previous versions of test case source files. A regression test is initiated when a test engineer submits a test case using test client program to the test server. Upon receiving the submission, the test server first retrieves the test source files from the test database and then builds the test case executables according to the building method and target platform information encapsulated in the test case object. The test executables are then allocated to some test station to run. If the test station is busy at the time, the test executables are spooled at the test server and wait for the test station to finish its current job. When the test station is available, the test executables will be transferred to the test station and test station will execute the test case. On completion of test execution, the test results will be sent back to the test server by the test station. The test server will process the results and generate reports. Finally, the test server will send these reports to the test engineers responsible for the test case.

3.2 Software Structure of the Framework

3.2.1 The layered structure

The software for the framework is structured in such a way that it can be easily adapted to different platforms. All the three components of the framework, namely the test client, the test server and the test station, should support TCP/IP.

- **Network layer** – Network layer is provided by the operating systems that run on test client, test server and test station computers. The framework software is built on top of the Internet/intranet communication mechanism that is widely available.

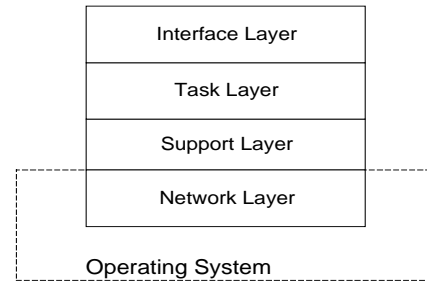


Figure 4 Software layer of the framework

- **Support layer** – This layer provides support for the following services:
 - Database connection to test database and version control system
 - The transportation of test files including test case source files, executables and test results between the server, clients and test stations.
 - Remote method invocation between framework classes such as test clients to test server or test server to test stations.
- **Task layer** – Programs that accomplish various testing related tasks such as test submission, test case selection, test case execution and monitoring, test result processing and test reporting, belong to this layer.
- **Interface Layer** – This layer provides visual interfaces to the human users.

Each layer calls the services provided by the layer beneath it and at the same time provides services to the layer above it. For example, the Support Layer calls the networking services provided by the operating systems in order to implement the services it provides to the task layer such as database connection, file transportation and remote method invocation. At a higher level, the Task Layer implement testing tasks by using the services provided by the Support Layer. Meanwhile, the Task Layer provides service to the Interface Layer, which manages the interaction between human and the system and let engineers to access the testing tasks.

3.2.2 An object-oriented design

The main classes and their responsibilities in the framework are discussed below.

- Abstract test case class** – This class encapsulates the attributes and methods that are required for a test case to be manipulated within the framework. It also provides an objected-oriented wrapper for the test cases that run within a conventional environment. For example, a test case could have been designed to run in the DOS environment and the source code for the test case could have been written in C or assemble language. However, when we create an object for this test case in the framework, we create a test case object by specifying its source files, building method and target platform. Then the test case can be built within the framework and test source files, test executables and test results can be transported between the test server and test stations. A sample definition of the TestCase class in C++ is shown at the right.

```

class TestCase {
Public:
    TestCase();
    virtual int Build();
    virtual int Execute();
    virtual char *GetHardwareReq();
    . . .

private:
    TestCase *parent;
    char *caseName;
    char *creator;
    char *sourceFiles;
    char *version;
    char *buildingMethod;
    char *hardwareReq;
    . . .
}

```

- Composite test case class** – Composite test case is a group of test cases that are required to be executed in a specified order. By defining the composite test case class we can form up complex test cases. There are two reasons for creating complex test cases: (1) In a test case group, some test cases need other test cases’ output as their input and (2) Execution of some test cases may not be meaningful unless the test cases before them have been successfully executed. These require that we be able to specify a group of test cases that run in certain order. The composite test case class allows us to view such a group of test cases as a whole and treat them just like a single test case within the framework.

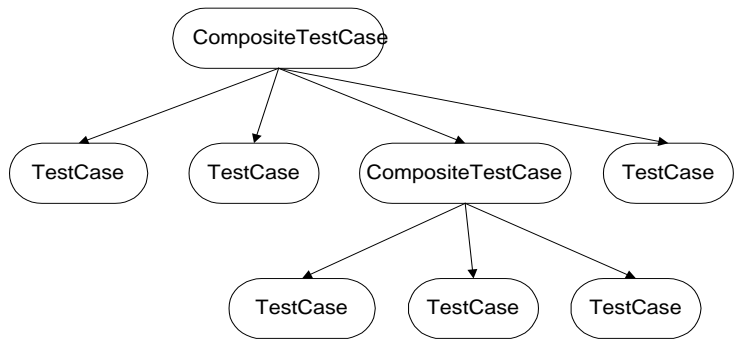


Figure 5 Composing complex test cases

If the execution of one test case B is dependent on the results from another test case A, then we denote this as $A \rightarrow B$. For example, in pacemaker firmware regression testing, one requirement is stated as: *whenever any changes in hardware are made to a test station, trimming programs must be successfully executed before any new testing execution on the test station.* Here, trimming programs can typically be denoted as trim_01, trim_02 and trim_03 and they must be executed in this

order successfully. By composing, the trimming programs can be treated as a single composite test case `Trimming` which contains three primitive test cases `trim_01`, `trim_02` and `trim_03`:

```
Trimming ::= { trim_01 -> trim_02 -> and trim_03 }
```

The above representation gives the information about both the group of test cases and the order in which they are required to run. Later, we will discuss techniques of generating composite test cases.

- **Abstract test case generation class** – This class lets us define how composite test cases can be generated. Basically, user can manually specify a group of test cases and the order in which they are executed. But with test case generation class, strategies for composite case generation can be encapsulated in test case generation objects that implement the abstract class. The design pattern *Strategy* [Gamma 94] is used in designing this class. As examples, we will discuss two test case generation methods: scenario based test case generation method [Poonawala 97] and MtSS [Kirani 94] based test case composition method. We will also show how these methods can be incorporated into the framework to create composite test cases and free test engineers from doing the job manually.
- **Abstract test case selection class** – Based on the same rationale, the abstract test case selection class allows us to incorporate different test case selection strategies into the framework, although manual selection is still available to test engineers as an option. As an example, we will discuss a test selection strategy that is based on a test case classification technique.
- **Abstract test station class** – The test station class is used to encapsulate configuration and execution status details of a test station. It is defined as an abstract class. We can derive different test station subclasses from this abstract class corresponding to different type of test stations. Test stations can be very different from each other. They may run different operating systems, have different hardware setup and test different product families or different products in the same product family. With the abstract test station class, we can have all the physically different test stations in the same testing framework and treat them uniformly.
- **Test server class** – Test server is actually a program that runs on the test server computer. It does the following:
 - Accepting test submissions from test clients
 - Building test case executables

- Allocating test cases to test stations
- Receiving status report from test stations
- Receiving test results from test stations
- Processing test results
- Sending reports to appropriate engineers who are responsible for the test cases
- Managing test stations

The test server class provides interface for test clients to submit tests. It also provides interface for test stations to report their execution status, send back test results. The framework implements the mechanism described in the design pattern *remote proxy* [Gamma 94] so that the interface of the server class is transparent to test clients and test stations.

- **Test client class** – Test client is also a program. But it runs from test engineer’s desktop. The test client class provides interfaces for human engineers to interact with the testing system to accomplish testing tasks such as submitting test, defining test cases and controlling test execution. It also provides interfaces to the test server so that testing tasks are transferred to the test server and accomplished by the server and test stations.

3.2.3 The object model

The object model of the distributed testing environment is shown in the following diagram.

In the diagram, the three elements of the framework are shown: TestClient, TestServer and TestStation. When any of them sends a message to another, it sends the message to its remote proxy. The proxy then sends the message to its real object. In this way, classes can send messages to other classes that are on remote machine just like they are on the local machine. There is no direct communication between test clients and test stations and the test server is responsible for the information exchange between them. By subclassing TestStation class to create different real test station subclasses, we are able to incorporate different test station hardware into the framework and treat them uniformly. In order to simplify the view, we didn’t include other classes in the diagram. We will discuss them later in this paper.

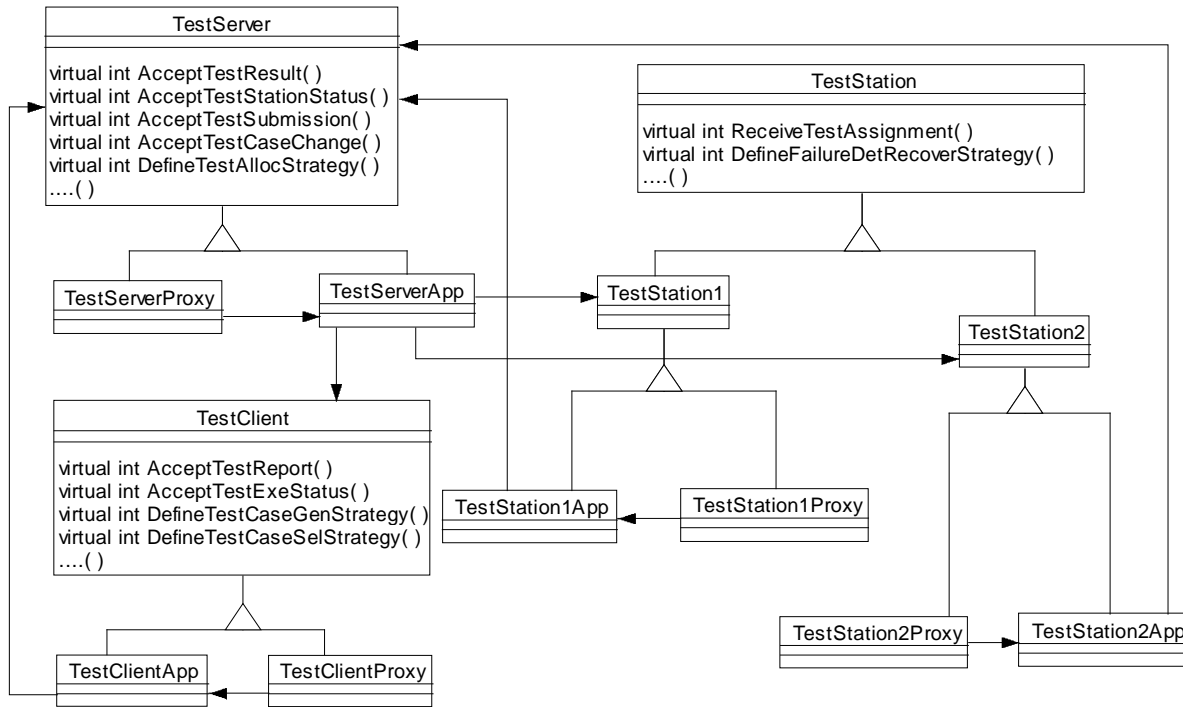


Figure 6 Distributed view of the framework

3.2.4 Distributed running mechanism

The running mechanism of the framework is show in Figure 7.

- Creating test cases** – Test cases are created by test clients. While defining a simple test case, an engineer can specify its properties such as test case ID, version, creator, hardware requirement, source file set, building method, etc. If a composite test case is being created, different test generation strategies can be incorporated to automate test case generation process. The information about the newly created test cases are then sent to the test server and stored into the test database.
- Doing Testing** – Whenever any changes occur in software artifacts and modifications have been made, it is necessary to perform regression testing. In the framework, testing requests are submitted from the test clients. While submitting a testing job, different test case selection strategies can be incorporated to make the testing efficient or to automate the selection procedure. As an option, test engineers can choose to select a test case manually. The test case selection information is then sent to the test server. Test server is responsible for test case building so that the test cases can be executed on the test stations. Information encapsulated in the test case objects such as building method, source file set etc. are then used while test case building. The test server will retrieve relevant information

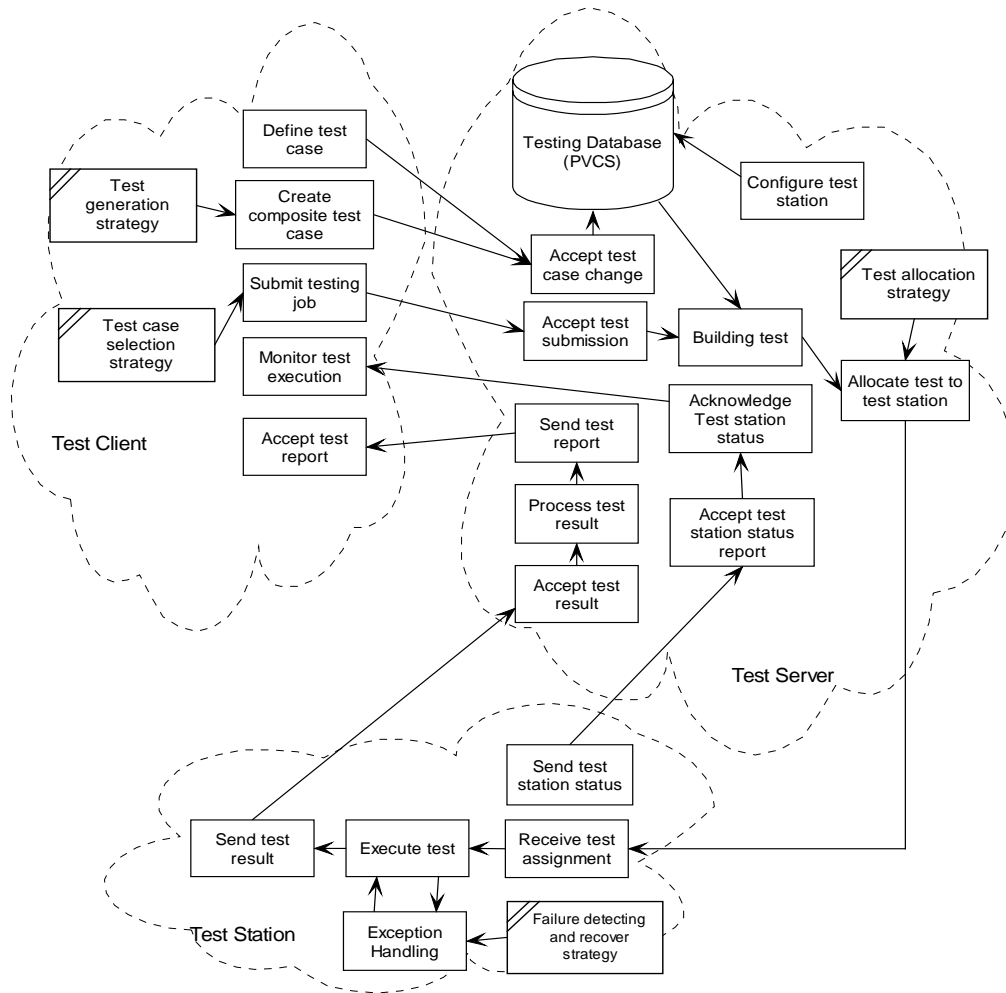


Figure 7 Running mechanism of the framework

from the test database and build the test executable. The server then allocates the test case to a test station that is suitable for its execution. If the allocated test station is busy, the test case will be put into a waiting queue for the test station. The test station, while receiving test assignment, will also acquire all the data and executables from the server related to the test case. Then it will execute the test case. After it finishes the test case execution, it will send the test results back to the test server. The server will then process the results and send the test reports to the appropriate test clients. This completes a round of regression testing. During test case execution, exceptions, interruptions and failures are handled by test stations. Suitable measures can be incorporated into the test station program to enable failure detection and recovery.

- **Test station status reporting** – Whenever the status of a test station changes, the station sends the status to the test server. And the server will send the status to all the test clients who are interested in the test station. During this process, the status of the test stations can be logged at the test server.

- **Test station configuration** – Configuration change occurs whenever any of the following happens:
 - Addition of a new test station
 - Removal of an existing test station
 - Changes to the hardware/software associated with a test station

The configuration information is stored in the test database.

By configuring test stations individually, the framework provides a uniform way of treating test stations. For testers, test stations are just test stations. They are the same. The differences only have effect when the test server is allocating test cases to test stations. And they only affect those test cases that are actually executed on the test stations. With this approach, we can place physically different test stations into a single framework and consequently we can do regression testing for different products simultaneously with the framework.

3.3 Composing complex test cases

In practice, individual test cases are designed to test specific functions or features of a system. However, it is usually required that certain group of test cases execute in a specified order. If it is required that B executes after A finishes, then we say that B is dependent on A. We denote this as $A \rightarrow B$. So a group of test cases that are executed in certain order can be expressed as $\{A_1 \rightarrow A_2 \rightarrow \dots A_n\}$. If we rewrite this as $\{\dots\{A_1 \rightarrow A_2\} \rightarrow A_3\}\dots A_{n-1}\} \rightarrow A_n\}$, then we find that we can build complex test cases by recursive

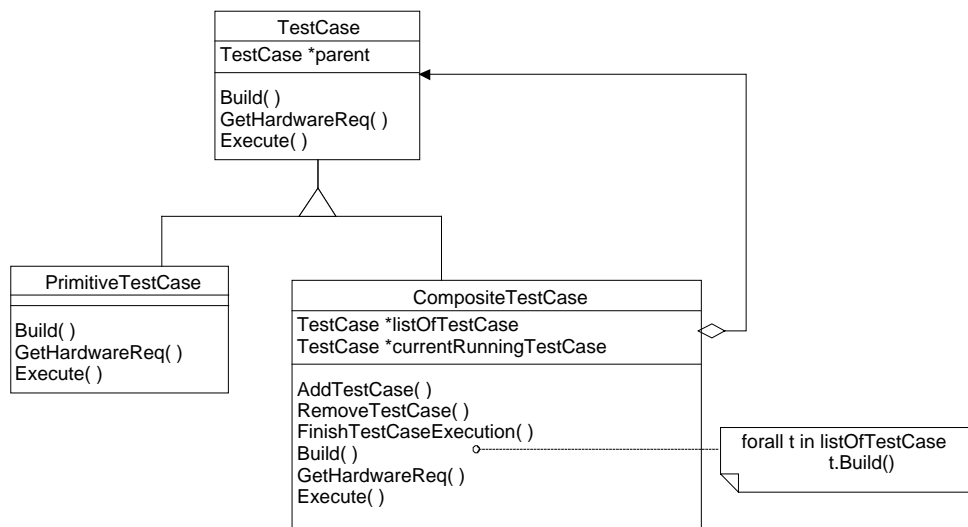


Figure 8 Object model for composite test case class

composition. Actually, we build complex test cases using the *Composite* design pattern [Gamma 94]. The class hierarchy is shown in Figure 8.

The composite test case class allows us to view a group of test cases as a whole and treat them just like a single test case within the framework.

For the example of composite test case given in Section 3.2.2, *Trimming*, its creation and usage are shown in the sample code on the right.

Now we describe a method that can be used with the framework to compose composite test cases.

```

TestCase *Trimming, *Trim_01, *Trim_02, Trim_03;

Trim_01 = new PrimitiveTestCase("trim_01");
Trim_02 = new PrimitiveTestCase("trim_02");
Trim_03 = new PrimitiveTestCase("trim_03");

/* Set up attributes for the three objects here
*/
...

Trimming = new CompositeTestCase("trimming");

Trimming->AddTestCase(Trim_01);
Trimming->AddTestCase(Trim_02);
Trimming->AddTestCase(Trim_03);

/* other operations */
...

Trimming->Build();

/* other operations */
...

Trimming->Execute();

```

First we create a new class `TPrimitiveTestCase` by subclassing from `PrimitiveTestCase` and add two new attributes to it: `mustFollow` specifies the test cases that must be executed before this test case. `followedBy` specifies the test cases that must be executed after this test case (Figure 9). The following table shows several test cases that have such information.

	mustfollow	followedBy
TA01		TA02, TA04
TA02	TA01	TA03, TA04
TA03		
TA04		TA05
TA05	TA04	

Some possible composite test case derived from the above table can be:

```

{TA01 -> TA02 -> TA03}, {TA01 -> TA02 -> TA04 -> TA05},
{TA01 -> TA04 -> TA05}, {TA04 -> TA05}

```

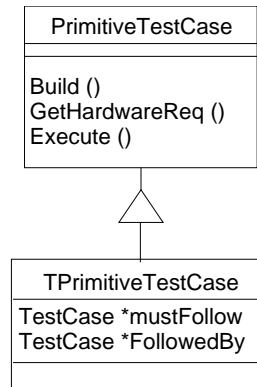


Figure 9 New primitive test case class

It is not difficult to devise an algorithm that can generate all the possible composite test cases based on the information specified by `TPrimitiveTestCase`. The algorithm can be used together with test case selection strategy since `mustfollow` and `followedBy` actually specify the dependency of test case execution. For example, when test case `TA02` is selected, we know that a composite test case must be selected from those composite test cases that contain `TA02`. Thus,

given a group of individual test cases, instead executing them directly, we execute a group of composite test cases so that the dependency is satisfied.

Another approach is random composing. In this approach, the first test case is randomly selected. Then during the propagation of the dependency specified in `mustfollow` and `followedBy`, dependent test cases are selected randomly to form a composite test case. For example, suppose TA02 is randomly chosen as the first test case, then TA01 must be added since it is required to execute before TA02. At this moment the composite test case is {TA01->TA02}. By random choosing from TA03 and TA04, the final composite test case will be one of {TA01 -> TA02 -> TA03} and {TA01 -> TA02 -> TA04 -> TA05}.

Scenario based test case generation technique ([Poonawala 97]) and Method Sequence Specification technique(MtSS) ([Kirani 94]) and its extension ([Vishnuvajjala 96]) can be used to generate test cases. The important thing here is that different test case generation techniques can be used in the same framework. In fact, we encapsulate test case generation techniques into test case generator subclasses (Figure 10). While test cases are being generated, an object of specific test case generator subclass is used to do the actual generation. Again we are using the *Strategy* design pattern to accomplish the design.

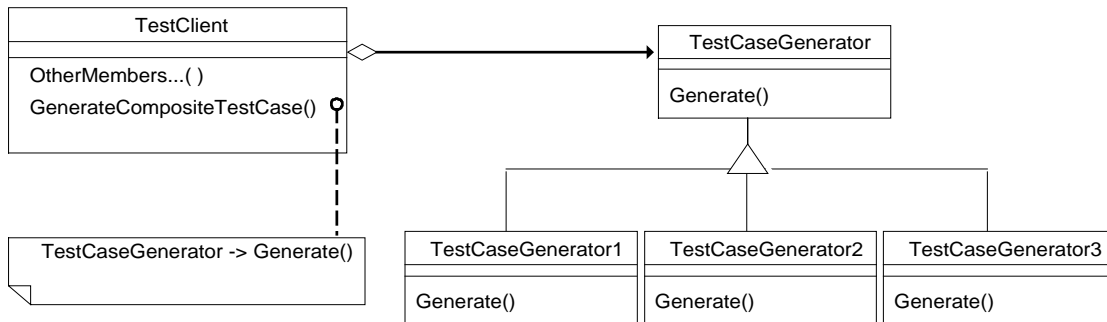


Figure 10 Using different test case generation strategies

Let us examine how a MtSS based test case generation technique would fit into our framework. MtSS specify the set of all possible sequences of messages that can be sent to an object. For example, the Programmer for pacemakers is a PC like device that provides physicians with a GUI to set various parameters. Due to interaction between these parameters not all values for a given parameter may be valid in a particular situation. The logic to verify the parameter values is in the Programmer software. For simplicity let us assume that there is a *Programmer* object in the software, that encapsulates a set of parameters. Let us further suppose that there are three methods *AcceptInput*, *CheckValidity*, and *SetParameters*, which respectively take user input, check the values and set the parameters in the device. A MtSS for such an object would be specified as:

$MtSS(Programmer) \Rightarrow (AcceptInput\ CheckValidity\ SetParameters)^*$

This specifies that the three methods should be called in sequence and the sequence itself could be repeated any number of times. A test case might then see if *SetParameters* fails when called before *CheckValidity*.

Test Case: Programmer.AcceptInput() followed by Programmer.SetParameters() Expected Result: SetParameters() should fail
--

The *Generate()* method of a *MtSSBasedGenerator* would take as input a MtSS and produce test cases like the one above.

Scenario based test case generation is a second approach. A scenario provides a high-level description of a possible sequence of actions that the system is expected to take. Each requirement can be associated with a scenario in which the functionality specified in the requirement is needed. For example, a requirement could be that *when a magnet is applied on the device for 30 seconds the device switches to therapy mode*. A scenario for this could be *switch to therapy mode by applying magnet for 30 seconds*. A test case based on this scenario would then be: *apply magnet for 30 seconds and check that the device is in therapy mode*. The description here is at an informal level. Formal notations are available and have been used to describe the scenarios and test cases could be generated automatically from those [Poonawala 97]. The *Generate()* method for a *ScenarioBasedGenerator* will take such a formal specification of the scenarios and automatically generate test cases based on those scenarios.

3.4 Test case selection

Similar design approach has also been adopted for incorporating test case selection strategies in the framework. The framework provides an interface to support virtually any test case selection strategy. The object model of the test case selection mechanism is shown in Figure 11.

It should be noted that all the different test case selection strategies may be used at the same time. For example, in preparing a set of test cases for the next round of regression testing, we may use test selection strategies based on requirements and put the selected test cases in the set. At the same time, we may also use a strategy based on the execution results of the previous regression testing, say choose those test cases that failed, to select test cases and put them into the same set. The resulting set contains the test cases that are to be executed during the next regression testing.

Let us consider how a requirements based test case selection strategy would fit into this framework. A requirements based selection strategy would select test cases from the database that are associated with a given set of requirements. Test cases are associated with requirements and this association information is maintained along with the test cases. For example, a test case could be: *Make the heart simulator fire a sequence of heart-beats that would require a brady pacing response from the pacemaker and measure the time taken before the response starts.* This test case may be associated with the requirement, *the pacemaker should start pacing therapy within 8msec of detecting an arrhythmia.* The *TestClient* has a list of *TestCaseSelector* objects, one of which would be a *RequirementsBasedSelector*. *SelectTestCase* method of *TestClient* would present the user with the list of selection strategies. When the user selects a particular strategy the corresponding *Select()* method would be invoked. The *Select()* method of a *RequirementsBasedSelector* would accept a list of target requirements that are to be tested, from the user, look at the requirements associated with each test case in the database and select the test case if any of its associated requirement is in the target set.

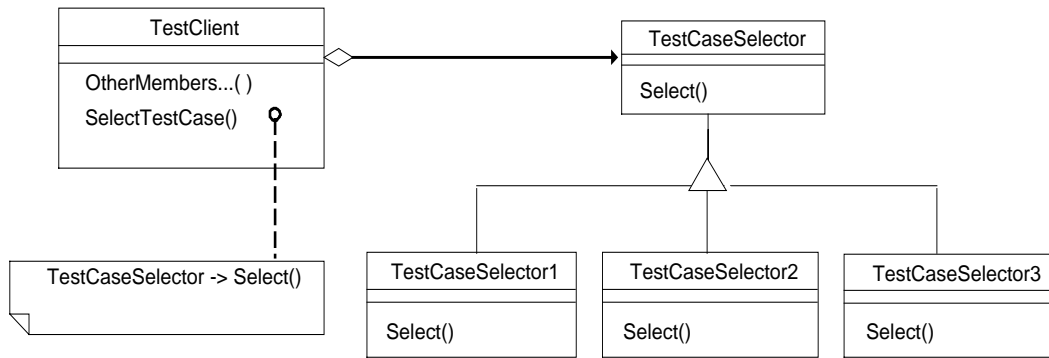


Figure 11 Using different test case selection strategies

Another selection strategy is test result based selection. Typically regression tests are run in cycles. In each cycle, the engineers select and execute a set of test cases, examine the results, make modifications to source code to fix problems detected by failed test cases and start another cycle. Typically they would want to run those test cases that failed in the previous cycle. A result based selection strategy could then be used to select those test cases that failed in the previous cycle. The *Select()* method for such a strategy would take the test cases and their results in the previous cycle and return a subset of test cases that have failed.

3.5 Allocating test cases

The motivation for allocating test cases to a group of test station is the need to maximize resource utilization. Usually we have a group of test stations that have different hardware configurations. So different test stations will be capable of running different test cases. On the other hand, we may have a group of test cases, each of which may require a different hardware configuration to run. The basic rule is to schedule execution of a test case on a test station that is capable of executing it. If we use $\text{HardwareReq}(T)$ to represent the hardware requirement of the test case T and use $\text{Capability}(S)$ to represent the capability of the test station S , then the rule can be expressed as:

$$\text{HardwareReq}(T) \subseteq \text{Capability}(S)$$

If this relationship holds, we say that the test case T is *executable* on station S . Generally, both $\text{HardwareReq}()$ and $\text{Capability}()$ represent a list of hardware devices from the same set of possible hardware used in the testing framework. For example, in pacemaker firmware testing, each test station is configured with some or all of the following devices: $DEV01$, $DEV02$, $DEV03$, $DEV04$ and $DEV05$. The following table gives the hardware configuration of three test stations:

	DEV01	DEV02	DEV03	DEV04	DEV05
STATION_01		x		x	x
STATION_02	x	x	x	x	x
STATION_05		x	x		x

The capability for the three test stations are:

$$\text{Capability}(\text{STATION_01}) = \{\text{DEV02}, \text{DEV04}, \text{DEV05}\}$$

$$\text{Capability}(\text{STATION_02}) = \{\text{DEV01}, \text{DEV02}, \text{DEV03}, \text{DEV04}, \text{DEV05}\}$$

$$\text{Capability}(\text{STATION_05}) = \{\text{DEV02}, \text{DEV03}, \text{DEV05}\}$$

The following table gives the hardware requirement of some test cases:

	DEV01	DEV02	DEV03	DEV04	DEV05
TB01_03					
TB07_05			x		
TB08_26	x	x	x	x	x
TB18_13		x			
TB19_01			x	x	
TB22_01		x	x		x
TC01_24	x	x			x
TC01_26	x	x		x	x

And we have:

$$\text{HardwareReq}(\text{TB01_03}) = \{\},$$

$$\text{HardwareReq}(\text{TB07_05}) = \{\text{DEV03}\},$$

HardwareReq(TB08_26) = {DEV01, DEV02, DEV03, DEV04, DEV05},

HardwareReq(TB18_13) = {DEV02}, HardwareReq(TB19_01) = {DEV03, DEV04},

HardwareReq(TB22_01) = {DEV02, DEV03, DEV05},

HardwareReq(TC01_24) = {DEV01, DEV02, DEV05},

HardwareReq(TC01_26) = {DEV01, DEV02, DEV04, DEV05}

The executabilities of the test cases versus the test stations are shown in following table.

	TB01_03	TB07_05	TB08_26	TB18_13	TB19_01	TB22_01	TC01_24	TC01_26
STATION_01	x			x				
STATION_02	x	x	x	x	x	x	x	x
STATION_05	x	x		x		x		

If we consider HardwareReq() and Capability() as binary vectors of hardware devices, then the executability of test case T on test station S can be calculated by the following formula:

$$\text{Executability}(T, S) = \text{HardwareReq}(T)^T \bullet \text{Capability}(S)$$

where Executability(T, S) = 1 means T is executable on S and Executability(T, S) = 0 means T is not executable on S. In the framework, we implement these two functions as the public member methods of test case class and test station class respectively. The server program calls these two methods while allocating test cases to stations. For composite test case T which contains test cases T₁, T₂, ... T_k, it is easy to see that:

$$\text{HardwareReq}(T) = \bigcup_{i=1}^k \text{HardwareReq}(T_i)$$

It is often the case that when a test case is submitted, there are more than one available test stations on which the test case can be executed. Hence, another factor that needs to be taken into account in test case allocation is load balancing. If we have many test stations in the testing system and many test cases to execute, load balancing becomes very important. Usually, different test stations have different capabilities due to differences in hardware configurations. Even if all the test stations have the same configuration and the same capability, we still want to balance test case execution in order to minimize the overall waiting time for test cases that are ready to run.

Balancing strategies can be based on various criteria:

- Minimize average waiting time for test cases

- Minimize overall execution time for test cases
- Minimize average finishing time for test cases
- Maximize average usage of all test stations. This is meaningful because the same test case may have different execution time on different test stations. This will let the slow test stations to have chances to run some test cases.

In our practice, the running time of the test cases in the previous execution was recorded as a base for load calculation in the next round of regression testing. Balancing is achieved by incorporating test case allocation strategies into the framework (See Figure 12). When the test server is trying to allocate test cases to test stations, it uses test case allocation strategies encapsulated in one of the subclasses that are derived from `TestCaseAllocator` base class. Again, we are using the *Strategy* design pattern. This design makes it possible to choose different allocation strategies dynamically. The framework thus also allows its users to incorporate their own strategies for test case allocation.

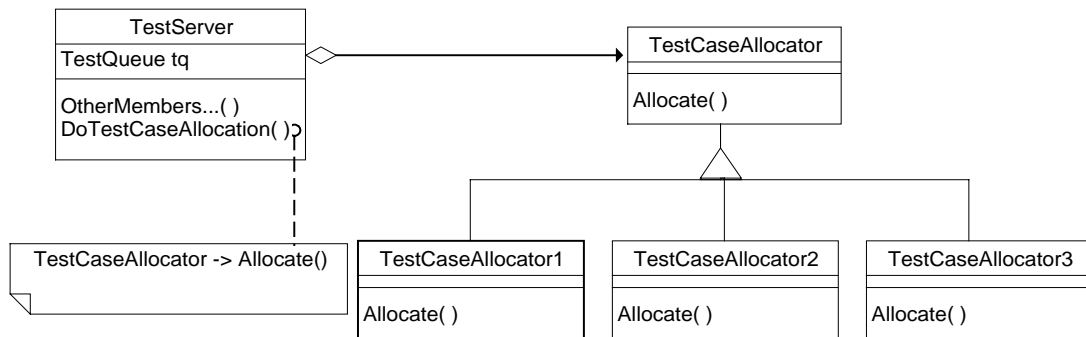


Figure 12 Using different test case allocators

For the test cases discussed above, the expected running time (ExpRunTime) are given below (assume the time is same on different test stations, for simplicity):

	TB01_03	TB07_05	TB08_26	TB18_13	TB19_01	TB22_01	TC01_24	TC01_26
ExpRunTime(min)	120	30	60	40	40	40	10	20

If the allocation strategy we encapsulate in `TestCaseAllocator1` is to minimize the overall finishing time for the test cases and at the time of allocating there are no test case running on any of the test stations, then the allocation given by `Allocate()` method of `TestCaseAllocator1` is as below:

	STATION_01	STATION_02	STATION_05
	TB01_03 (120)	TB08_26 (60)	TB07_05 (30)
		TB19_01 (40)	TB18_13 (40)
		TC01_24 (10)	TB22_01 (40)
		TC01_26 (20)	
Total	120	130	110

If the strategy is to minimize the average waiting time for test cases, then the allocation is similar, but the order in which the test cases are executed on STATION_02 will be different. The test cases require less time will be executed first.

3.6 Interruption and recovery

Detecting interruption of test case execution is the responsibility of test stations. Sometimes, the execution of a composite test case can be suspended or aborted due to failure of one of its test cases.

Detecting failure of test case execution is important. For example, the execution of a test case B in a composite test case C may be dependent on the results from another test case A. Hence execution failure of a test case within a composite test case must be audited so that test cases that follow it do not start with improper data leftover by the failed test case. This also causes interruption in the execution of the composite test case C. To achieve an automated regression testing process, we also need to be able to recover from a test failure during the execution of a composite test case. One approach is to implement a method in the composite test case class called *Restore()*, which will restore the system to the state when the previous test case finished execution.

While the framework provides an interface for its users to plug-in their own techniques and strategies for failure detection and recovery, techniques for failure detection and recovery vary with respect to the variation of test stations. In a typical firmware testing environment, a test station is configured with a raw register programmer (RRP). In order to recover and resume execution of a composite test case, it is necessary to save the state of the test station before each individual test case is executed. If a test case failed to finish, a restoring utility is used to restore the initial scene of the test case, including the values of registers in RRP and the test case is executed again. Sometimes, failures may persist and there will be two options: continuing the execution of the test cases that follow or terminating the composite test case. The first option will require a finer granularity of failure classification while the second option is easier to implement.

3.7 Configuring test station

Configuring test stations is done by the test server program which provides an interface for test station management

A test station must be configured into the testing system before it can be used by test clients for testing. The framework provides a template for the definition of test stations. Different enterprises may need to define their own test station classes based on the generic design. Although a test station is a group of hardware devices attached together, it is wrapped by the abstract test station class and hence treated as a software object in the framework. Thus test stations can be changed or added to or removed from the framework dynamically without affecting stations that are running test cases, similar to printers in a shared network printing system. The information about a test station will be saved to the test database and used later when the server allocates test cases to the test stations.

4. Application

The following are some of the advantages that this framework provides:

- Different companies can use this framework to do regression testing
- Different test stations can be configured into the same testing framework
- Different products can be tested within the same testing framework
- Test engineers can access the framework from different platforms

Since the framework has a layered structure, it is easy to port it to different platforms. A possible future development of the framework is to implement the test client in Java applet, which can be run from a Web browser.

Several design patterns are extensively used in the framework including *Factory method*, *Proxy*, *Composite* and *Strategy* [Gamma 94]. By using these design patterns, the framework is ready to incorporate user preferences in test case generation strategy, test case selection strategy, test case allocation strategy and failure detecting and recovery strategy.

We have developed an automated regression testing system for the design verification testing of cardiac pacemaker firmware, based on the framework presented in this paper. The implementation is on top of MS-Windows. At the interface layer, we have implemented a Windows interface for test server, test client and test station program. A test station is typically a PC that is configured with a set of interface cards that

communicate with various hardware devices. Test cases are usually DOS applications that can be run from DOS-prompt of Windows systems. Regression testing using the framework is done in two ways: *scheduled and looped*. A full regression testing may last for up to 3 days. Scheduled regression testing, which will be activated on a specified time, is usually for the weekend full regression while looped testing, which usually runs again and again if not interrupted, is for smaller test case groups during weekdays. Test case selection strategy is based on coverage and previous testing results. We used PVCS (Program Version Control System) as the repository for test case source files. The test cases source files in the PVCS updated by test engineers will be used in the next test case building if the related test cases are selected for testing. The test server checks out test case source files and does the actual building. The test case allocation strategy is based on load balancing and earliest finishing time for test case. The test server uses the running time of the test cases in previous execution as a base for load calculation. With this system, human intervention was totally eliminated during the whole process of regression testing. We conclude from the successfulness of the system that (1) the design of the framework has achieved its goal in providing a usable and efficient guide of constructing distributed testing systems, and (2) the testing system has achieved its goal in automating regression testing process in a distributed environment.

References

- [Agrawal 93] H. Agrawal, J. R. Horgan, E. W. Krauser and S. A. London, "Incremental Regression Testing", Proceedings of IEEE Software Maintenance Conference, 1993, pp. 348-357.
- [Elliott 94] L. Elliott, R. Mojdehbakhsh and W.T. Tsai, "A Process for Developing Safe Software", Proceedings of the 7th Annual IEEE Symposium on Computer-Based Medical Systems, IEEE CS Press, Los Alamitos, California June 1994
- [Gamma 94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley 1994
- [Kirani 94] S. Kirani and W.T. Tsai, "Specification and Verification of Object-Oriented Programs", Technical Report, University of Minnesota, Minneapolis, MN 55455, 1994
- [Ness 97] B. Ness and V. Ngo, "Regression Containment through Source Change Isolation", Proceedings of COMPSAC 97, Washington D.C., August 1997, pp616-621
- [Onoma 95] A. K. Onoma, W. T. Tsai, F. Tsunoda, H. Sukanuma, and S. Subramanian, "Software Maintenance -- Industrial Experience", Technical Report, Dept. of Computer Science, University of Minnesota, 1994. *Journal of Software Maintenance*, Dec. 1995.
- [Onoma 96] A. K. Onoma, W.T. Tsai, M. Poonawala and H. Sukanuma, "Regression Testing in an Industrial Environment", *Communications of the ACM*, March 1996
- [Poonawala 97] M. Poonawala, S. Subramanian, W.T. Tsai, R. Vishnuvajjala, R. Mojdehbakhsh and L. Elliott, "Testing Safety-Critical Systems - A Reuse-Oriented Approach", *International*

Conference on Software Engineering and Knowledge Engineering, June 1997, pp. 271-278

- [Rumbaugh 91] J. Rumbaugh, et al., *OO Modeling and Design*, Prentice Hall, Englewood Cliffs, NJ, 1991
- [Rothermel 94a] G. Rothermel and M. J. Harrold, "Selecting Regression Tests for Object-Oriented Software", Proc. of IEEE Software Maintenance Conference, 1994, pp. 4-25.
- [Rothermel 94b] G. Rothermel and M. J. Harrold, "A Comparison of Regression Test Selection Techniques", technical report, Department of Computer Science, Clemson University, Clemson, SC 29634-1906, October 1994.
- [Tsai 97] W. T. Tsai, R. Mojdehbakhsh, S. Rayadurgam, "Experience in Capturing Requirements for Safety-Critical Medical Devices in an Industrial Environment", To appear in the Proceedings of IEEE High Assurance Systems Engineering, 1997.
- [Vishnuvajjala 96] R. Vishnuvajjala, W.T. Tsai, R. Mojdehbakhsh and L. Elliott, "Specifying Timing Constraints in Real-time Object-Oriented Systems", Proceedings of High Assurance Systems Engineering Workshop, Ontario, Canada, October 1996, pp. 32-39
- [White 93] L. J. White and V. Narayanswamy, "Test Manager: A Regression Testing Tool", Proceedings of IEEE Software Maintenance Conference, 1993, pp. 338-347.