J. Software Engineering & Applications, 2009, 2: 209-220
doi:10.4236/jsea.2009.23029 Published Online October 2009 (http://www.SciRP.org/journal/jsea)

An Exploratory Case Study in Designing and Implementing Tight Versus Loose Frameworks

Manjari GUPTA[1], Ratneshwer GUPTA[2], A. K. TRIPATHI[3]

[1]Department of Computer Science, Faculty of Science, Banaras Hindu University, Varanasi, India; [2]Department of Computer Science, MMV, Banaras Hindu University, Varanasi, India;[3]Department of Computer Engineering, Institute of Technology, Banaras Hindu University, Varanasi, India.
Email: {manjari, ratnesh, anilkt}@bhu.ac.in

Received May 5th, 2009; revised June 20th, 2009; accepted June 24th, 2009.

ABSTRACT

Frameworks provide large scale reuse by providing skeleton structure of similar applications. But the generality, that a framework may have, makes it fairly complex, hard to understand and thus to reuse. Frameworks have been classified according to many criteria. This paper proposes two types of framework (based on the concept of 'generality') named as: tight framework and loose framework. A case study is done by developing loose and tight frameworks for the application sets of Environment for Unit testing (EUT) domain. Based on the experience that we got by during this case study, we tried to find out the benefits of one (tight or loose) framework over the other. This work attempts to provide an initial background for meaningful studies related to the concept of 'Design and Development of Framework'.

Keywords*: *Framework Reuse, Environment for Unit Testing, Condition Coverage Criteria

1. Introduction

"Frameworks are reusable designs of all or part of a software system described by a set of abstract classes and the way instances of those classes collaborate". It is always the result of domain analysis [1]. Frameworks may be classified according to many criteria such as manner of deployment in different applications, level of support that provides to applications, type of services they have (entity framework, control framework), level of abstraction they have (white box framework, black box frameworks) etc. Frameworks are normally developed by keeping in mind requirements of multiple similar applications. Frameworks should be developed and delivered in such a manner so that problems in instantiation (like determining the applicability of a framework, understanding and modifying if necessary, architectural mismatch etc. [2]) do not arise and overheads (requirements that are not required in a particular application), do not get transported with the framework. In order to develop a framework, its scope must not be an afterthought and it should be considered at the beginning i. e. at the time of designing frameworks. When one talks about frameworks, its scope and generality are necessary to consider.

Here, we classify frameworks by considering the generality they have. In software engineering literature, we could not find the formal categorization of frameworks based on 'generality' concept. We classify frameworks in two categories by considering their generality as follows.

A ***loose framework*** is a framework that does not fix the way of performing many activities (that may be performed differently in similar applications in a domain) in the framework itself. It only provides the control abstraction and thus may need to work together with other frameworks for some activities that will extensively interact with it. Such frameworks may be useful for those applications that have many possible variations in their requirements for example business application systems, E-governance systems etc.

A ***tight framework*** fixes the way of performing most of such activities in the framework itself. Thus, these frameworks are highly useful for (only) those similar applications that require performing those activities in a particular way as defined and implemented in the framework. Such frameworks may be useful for those applications that have very few variations in their properties like embedded systems, pervasive systems etc.

During design of a framework certain roles and responsibilities amongst the classes along with their collaboration are fixed. Variability among applications is represented as hot spots. Thus, by comparing the already fixed roles and responsibilities as well as the variability (hotspots) of different frameworks, for the same domain, one can say "what is the scope of a framework?" and

thus "whether a tight or a loose framework development would be beneficial for that domain?"

To the best of our knowledge, there is lack of such study/work in which tight and loose frameworks, for a same domain, are compared so that one can list the cases in which one (tight or loose framework) would be better than the other. In this paper, a tight and a loose framework for 'Environment of Unit Testing' have been developed and a comparative study has been made between the two types of frameworks. Based on the observations of this study, we tried to answer the following questions:

1) Which framework (tight or loose) is more reusable in terms of "ease of reuse"?

2) Which one is more reusable in terms of "number of reuses"?

3) Which one is easy to develop?

4) Which one is heavier in terms of size?

5) Which one is more complex?

Reminder of this paper is organized as follows. The section 2 attempts to present, in a concise manner, the research efforts related to the topic of discussion. In section 3, we briefly describe the domain 'EUT' for which frameworks have been developed. Designs of a loose and a tight framework for "EUT" have been described in section 4 and 5 respectively. A comparative study of loose and tight frameworks has been discussed in section 6. Finally, we conclude in section 7.

2. Related Work

Software engineering, over the last decades, has been promoting the development of software systems with software frameworks. Researchers and practitioners have been considering various aspects of framework development and related issues. Software frameworks are classified according to several criteria.

There are two styles of frameworks that are commonly used: called and calling frameworks. Sparks et al. [3] showed the reuse with Called and calling frameworks. *Called* frameworks are very much like traditional libraries in that the application code calls the framework when some framework service is needed. *Calling* frameworks on the other hand, reverse the role of the framework and the application, because the framework calls the application code, rather than the other way around. Some authors defined frameworks according to domain dependency: *vertical* and *horizontal* frameworks [4]. A framework dependent on specific domain is referred to as *vertical framework*. A framework independent on specific domain is referred to as *horizontal framework*. According to Taligent, Inc (now IBM) [5]) the problem domain that a framework addresses can encompass application functions, domain functions, or support functions. Application frameworks encapsulate expertise applicable to a wide variety of programs. These frameworks encompass a horizontal slice of functionality that can be applied across client domains. Current commercial graphical user interface (GUI) application framework, which supports the standard function required by all GUI applications, is one type of application frameworks. Domain frameworks encapsulate expertise in a particular problem domain. These frameworks encompass a vertical slice of functionality for a particular client domain. Examples of domain frameworks include: a control systems framework for developing control applications for manufacturing, securities trading framework, multimedia framework, or data access framework. Support frameworks provide system-level services, such as file access, distributed computing support, or device drivers. Several authors defined Frameworks based on the techniques used to extend them: White-box, Black-box and gray-box frameworks [4]. In a white box framework, the framework user is supposed to customize the framework behaviour through sub-classing of framework classes. On the other hand, a black box framework user does not have access to framework code. Gray-box frameworks lie between white and black box framework. Frameworks are also classified by their scopes: *System infrastructure frameworks*, *Middleware integration frameworks* and *Enterprise application frameworks*. *System infrastructure frameworks* simplify the development of portable and efficient system infrastructure. Communication frameworks, proposed by Schmidt [6], also belong to *System infrastructure frameworks. Middleware integration frameworks* are commonly used to integrate distributed applications and components. Common examples include ORB frameworks, message-oriented middleware, and transactional databases. *Enterprise application frameworks*, proposed by Fayad et al. [7], address broad application domains such as telecommunications, avionics, manufacturing, and financial engineering.

However, no such study has been done till now to develop different types of frameworks for a domain and compare them to show in which situation which one is more applicable for reuse. We had earlier done risk analysis in framework development and its reuse [8]. This paper attempts to extend the above contributions further by considering the comparative differences of loose and tight frameworks.

3. Environment for Unit Testing

We briefly explain the domain "EUT" for which frameworks are developed in the next sections. To develop a framework, it is necessary to understand the domain for which it is to be developed. Thus, first we briefly describe the unit testing process.

Unit testing is a dynamic method for verification, where the smallest unit of software design- *the software component or module* (unit code) is actually compiled and executed. The unit testing focuses on the internal processing logic and data structures within the boundary

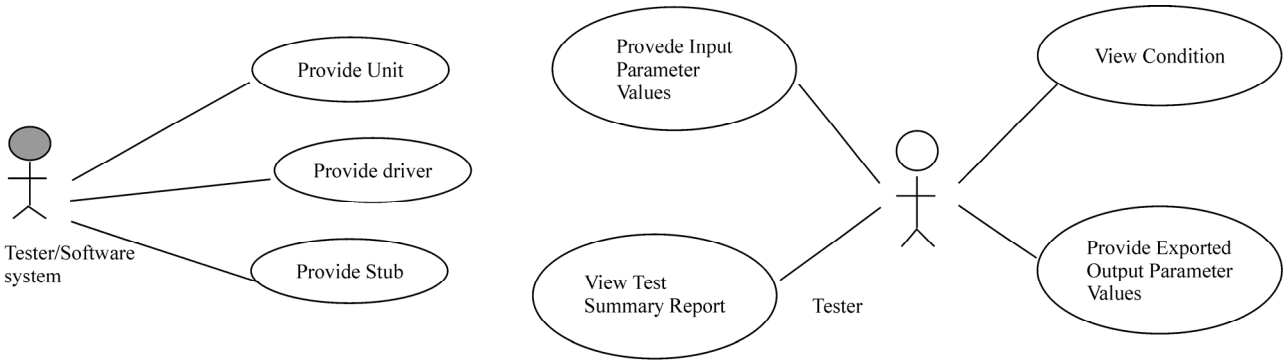

Figure 1. Use case diagram for tight framework

of a component [9]. As the focus of this testing level is on testing the code, *structure testing* is best suited for it. Unit testing commences with generating test drivers and stubs for the unit. Next, test cases are generated. A test case is a set of test inputs, on which the unit code, to be tested, is executed. The output of the program for each test case is evaluated to verify compliance with the corresponding requirement using test oracles. During unit testing several test deliverables (test case specification, error report and test log etc.) are generated. At last, test summary report is generated that specify the result of testing process.

To test the structure of a program, structure testing aims to achieve test cases that will force the desired coverage of different structures. Various criteria have been proposed for this. Most common *structure based criteria* are based on the *control flow* of the program for example statement coverage, branch coverage, decision/condition coverage and path coverage.

We developed both of these (tight and loose frameworks) for the domain of 'EUT'. Both of these frameworks test a unit written in C language. In the tight framework we fixed the test case generation activity (based on condition coverage criteria). Because of that, this framework can only be used if the unit testing criteria is 'condition coverage'. For rest of the unit testing criteria, this reusable framework is useless. However, the test case generation activity was not fixed in the loose one and thus any test case generator (developed by considering any unit testing criteria) that can generate test cases to test a C unit can be integrated with the loose framework. Both the frameworks (tight and loose) have been developed in C++ language.

4. The Proposed Tight Framework for 'EUT'

We first describe the design and implementation of tight framework for 'EUT'.

The tight framework is developed by considering the test cases generation based on the *condition coverage* criterion. Thus, this framework supports the development of a family of applications that would test a unit based on the condition coverage criteria but differ in the way of getting unit, driver, stubs. For example, a system developed by using this framework may accept these from a human being while others may get these from software systems (that will be generating these automatically).

As we know, any object oriented software and in particular any object oriented framework is a collaboration of domain, control logic, utility and interface classes. In this remaining section and in the next section, we identify these classes for both tight and loose framework for 'EUT' and show how they collaborate with each other.

4.1 Analyzing the Requirements of Tight Framework for 'EUT'

By *studying the above problem statement*, we specify how a user will interact with this tight framework in the *use case diagram* shown in Figure 1. Gray ovals show the functionalities that are needed to be customized and black ovals show functionalities that are prefixed in the framework. Way to provide unit to be tested, drivers and stubs (if any) to the framework (manually by the tester or automatically using a software system) may be different for different applications that will be developed by using this framework. Thus, in the use case shown on the left hand side of Figure 1, actor may either be a human being or a software system and that's why has shown as gray (as per the convention usually used to describe a framework). As shown in the use case on the right hand side of Figure 1, each condition, to be tested, is displayed to the tester and then a tester provides test cases corresponding to each condition. These test cases are run and test summary report is delivered to the tester after the completion of the testing process.

From the problem specification, described above, we can identify the following classes:

Unit – is an important class that would be tested using this framework,

Driver – would invoke and provide environment for the execution of the unit (if required),

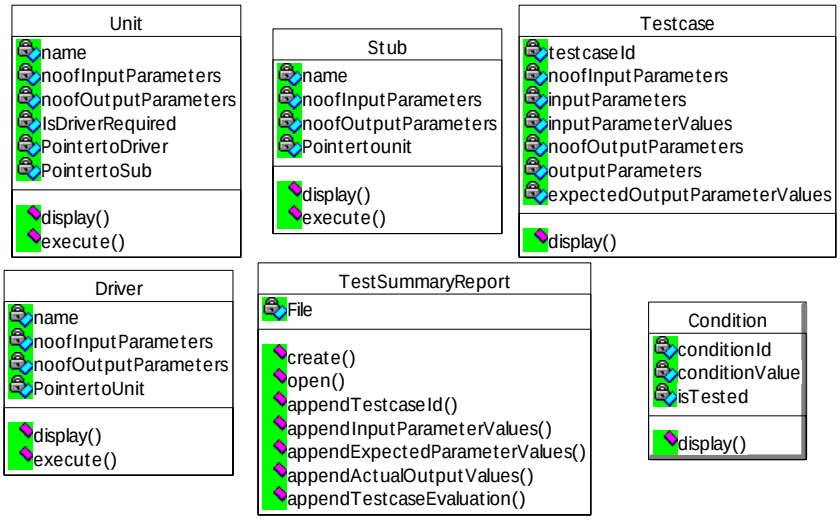

Figure 2. Domain classes for tight framework

Stub – would be called by the unit during its execution (if required),

Test case – is a class that would represent a test case,

Condition – is a class that represents a condition to be tested and

Test summary report – represents the test summary report delivered to the tester after testing. These domain classes along with their attributes and methods are shown in Figure 2.

4.2 Designing the Tight Framework for 'EUT'

To represent the abstract dynamic behavior of the system developed as a tight framework, we first describe scenarios.

The success scenario is as follows:

1) Unit is provided (way would be specific to the application) to the system.

2) If driver is needed, it is provided (way would be specific to the application) to the system.

3) If stubs are needed, these are provided (way would be specific to the application) to system.

4) Until all the conditions, in the unit, are tested

a) Next condition is identified.

b) System displays this condition to the tester and asks the number of test cases (N) that need to be generated to test this condition.

c) For this number (N) of times

i) System asks the next test case.

ii) Tester inputs the test cases.

iii) System executes this test case and redirects the output value of the execution to a file.

iv) The actual output value is compared with the expected output value and is shown to the tester.

v) System appends the testing result of this test case (condition, corresponding test case, actual output value, execution status of the test case (successful/unsuccessful, executed or not) etc.) in the test summary report.

vi) System asks whether to proceed further or quit.

vi) If tester wants to quit (in case of getting wrong result to correct the unit), the path and name of the Test Summary Report is displayed to the tester and system stops.

d) System asks whether to proceed further or quit.

e) If tester wants to quit, the path and name of the Test Summary Report is displayed to the tester and system stops.

5) A message "all the conditions have been tested" and the path and name of the Test summary report is displayed to the tester.

An *exception* scenario could be: if the format of any of the information related to the program (unit, driver or stub) is unacceptable by the system, for example if the values of the input parameters, needed as a string containing all these values separated by a space, is not provided in the required format. In all these types of situations, system would display different error messages according to the situation.

Another abnormal scenario could be: if a test case cannot be run successfully and thus the execution fails. System will display an error message to show this situation.

The activity diagrams (Figure 3 (a) and Figure 3 (b)) show the sequence of activities performed during *test case generation* and *execution and comparison of actual and expected output* activities in the framework. All the activities in this tight framework for 'EUT' are shown in Figure 3 (c).

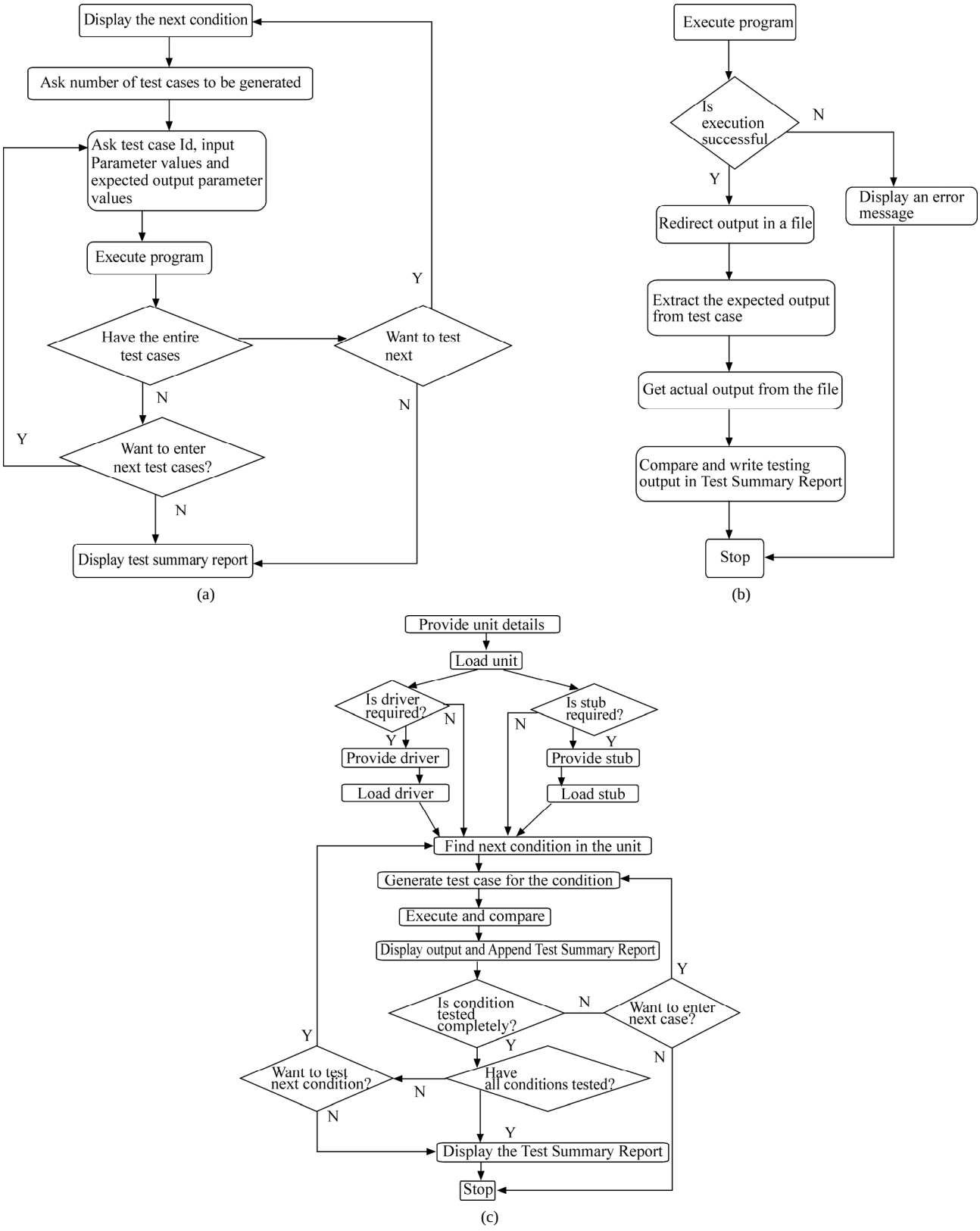

Figure 3. (a): Activity diagram for generating test case for tight framework, (b): Activity diagram for executing and comparing output for tight framework, (c): Activity diagram for tight framework

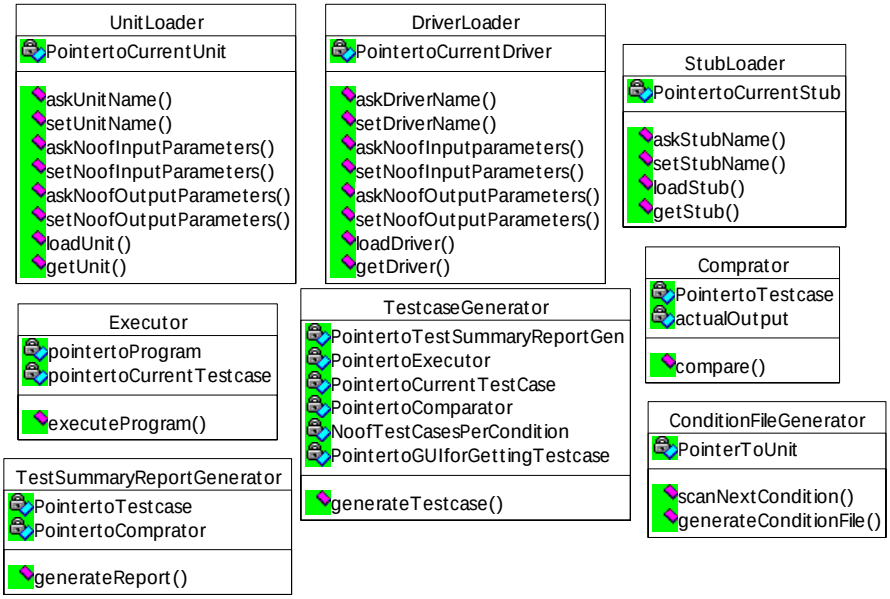

Figure 4. Control logic classes for tight framework

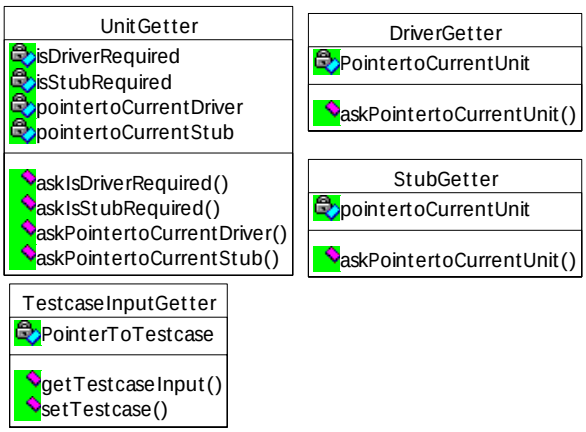

Figure 5. Graphical user interface class for tight framework

Scenarios and activity diagram, described above, hint for the following control *logic classes*:

UnitLoader – responsible for loading the unit to be tested,

DriverLoader – responsible for loading a driver (if any),

StubLoader – responsible for loading a stub (if any),

TestcaseGenerator – responsible for generating test cases with the help of tester,

ConditionFileGenerator – responsible for identifying all the conditions, in the unit, to be tested,

Executor – responsible for executing the unit (if there is no driver) or driver,

Comparator – responsible for comparing the actual output from the expected output parameter values for a given test case and

TestSummaryReportGenerator – responsible for generating test summary report.

These control logic classes along with their attributes and methods are shown in the Figure 4.

Following are the interface classes that are designed for this tight framework. These classes with their attributes and methods are shown in Figure 5.

UnitGetter – responsible for providing unit, to be tested, to the rest of the system.

DriverGetter – responsible for providing driver, if any, to the rest of the system.

StubGetter – responsible for providing stubs, if any, to the rest of the system.

TestcaseInputGetter – responsible for displaying a condition to the tester and getting input parameter values to test that condition.

In this framework Unit, Driver and Stub classes share several attributes and methods thus we take a class **Program** from which all these three classes will inherit properties and operations. Similarly, UnitLoader, Driver Loader and StubLoader also share several properties and operations and we have taken a class **ProgramLoader** that have all the common properties and operations of these classes and these classes are taken as sub class of ProgramLoader class. Further, UnitLoader, DriverLoader, StubLoader are kept as abstract classes because we don't fix in this framework how the unit, drivers or stubs are generated. Thus, these classes are hot spots of the framework. At the time of instantiation of this framework one needs to refine these subclasses according to the application using the framework. The object diagram, that shows the relationships among all the types of classes identified during analysis and design, is shown in Figure 6.

4.3 Instantiation

We have instantiated this framework by adding three classes ***UnitLoader_Sub*** of *UnitLoader*, ***DriverLoader_Sub*** of *DriverLoader* and ***StubLoader_Sub*** of *StubLoader*. These sub classes allow us to provide unit, drivers and stubs (if required) manually using graphical user interface. Thus in this instantiation, we assume the application need is to provide these manually. In any other instantiation, some other method of providing a unit, drivers and stubs can be used for example as a result of some automation process that will generate them etc. Further one more class ***GUI*** is added that helps in implementation of these other classes added during instantiating. *GUI* class has a method that displays a string, passed it as a parameter, to the user. Thus, at the time of instantiation of this tight framework for 'EUT' only four additional classes were needed to be added.

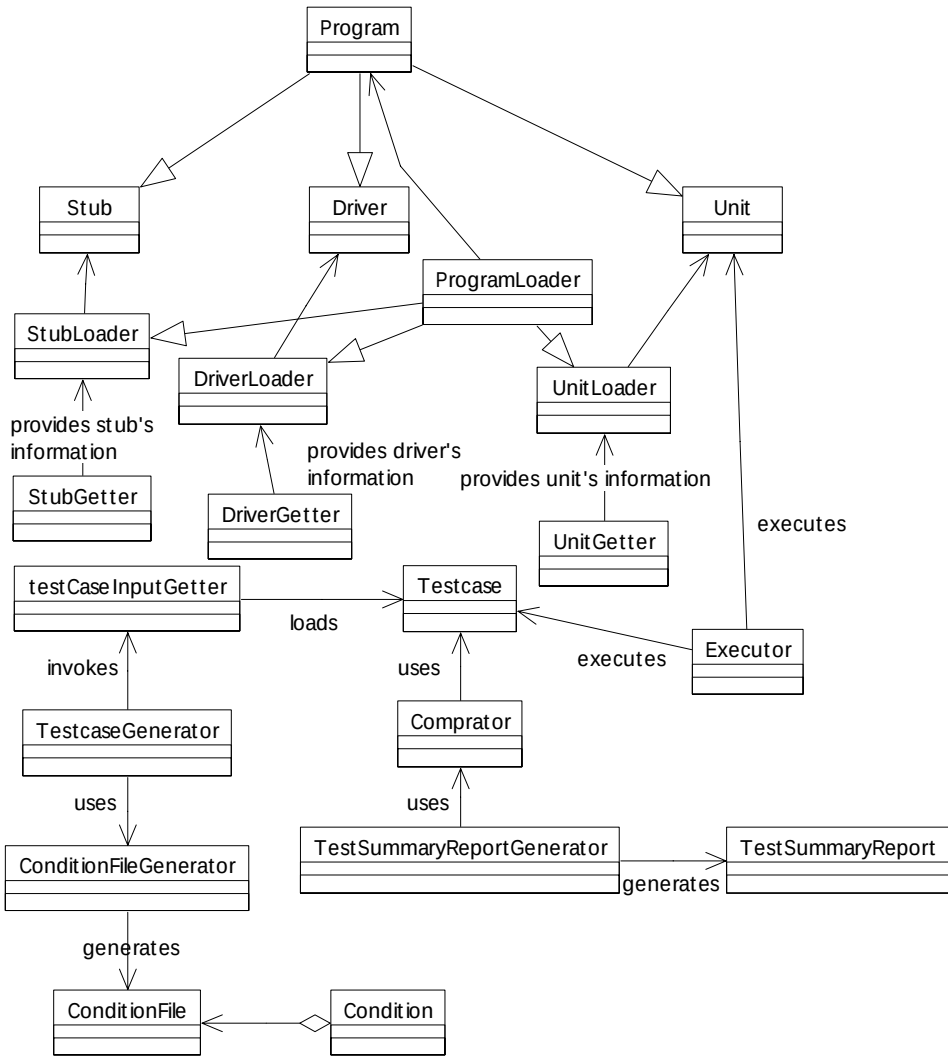

Figure 6. Object diagram of tight framework

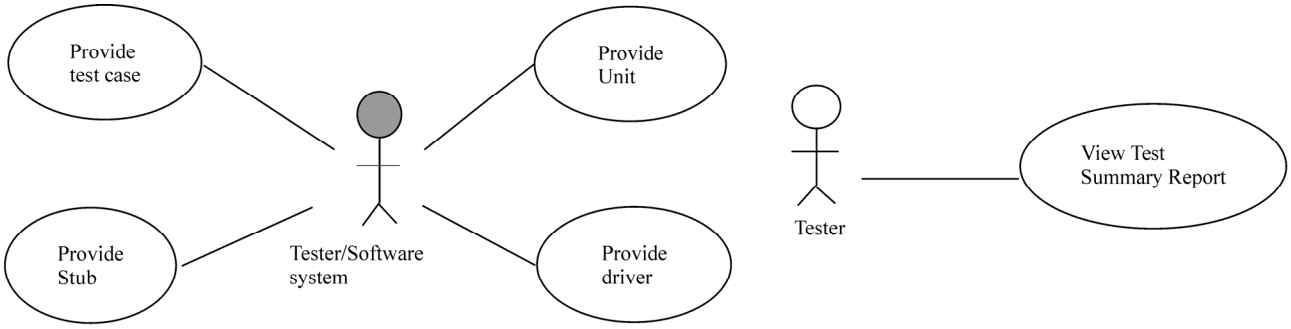

Figure 7. Use case diagram for loose framework

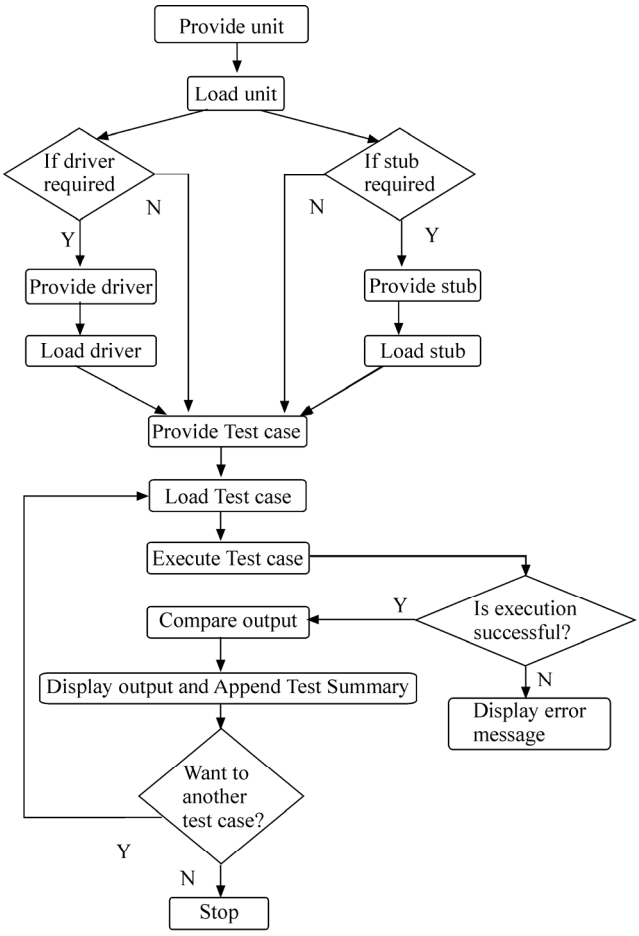

Figure 8. Activity diagram for loose framework

5. Loose Framework for 'EUT'

In the loose framework, for the same 'EUT' domain, we do not restrict the method of drivers, stubs or test case generation as we did in the above tight framework. However, in this loose framework the test oracle is also not fixed. That is, this framework would accept the drivers, stubs (if required for a unit) and test cases from somewhere else as the tight one did. These would be given to the framework either manually or generated automatically. This framework will accept and load these into the respective classes defined in the framework. This loose framework for 'EUT' does not fix the way of generating test cases, which is a big part of the framework. Thus, using this loose framework one can perform any type of structural testing. That is, the test case generation would also be application specific.

5.1 Analyzing the Loose Framework Requirements

By studying the problem statement, we specify the functionalities that the framework will support in the use case diagram in Figure 7. Gray ovals show functionalities that are needed to be customized and black ovals show functionalities that are prefixed in the framework. Unit, to be tested, drivers and stubs (if any) and test cases can be provided to the system either manually or can be loaded automatically using any software.

After analysis, here also, we get the same ***domain classes*** as we obtained in tight framework described above: ***Unit, Driver, Stub, Test case, and test summary report***. Only the domain class 'condition' that was identified during development of the tight framework, described above, is not a domain class for this framework. Rests of the classes are all same.

5.2 Designing the Loose Framework

To represent the dynamic behavior of a system that would be developed using this loose framework, we describe the following scenarios that explain how the system behaves when it performs some of its functions. The success scenario is as follows:

1) Unit is provided to the system.
2) If driver is needed, driver is provided to the system.
3) If stubs are needed, these are provided to the system.
4) Test data is given to the system.
5) Test oracle provides the correct output for a given test data.
6) System executes the test case and compares the actual output with the expected output parameter values.
7) System generates a test summary report.

Abnormal scenarios are as follows:

1) Unit is given to the system.
2) If driver is needed, driver is given to the system.
3) Test data is given to the system.
4) Test oracle provides the correct output for the test case.
5) System could not execute the test case and generates an error message "execution failed".

Similarly other abnormal scenarios would be if the unit and driver cannot be provided in the form used by the system. At that time system again generates error messages corresponding to the error occurred.

As we mentioned above, in this loose framework, we do not restrict the method of generating drivers, stubs or test case generation. Similarly the test oracle is also not fixed. That is, drivers and stubs (if required for a unit to be tested) and test cases can be provided manually or generated automatically and are supplied to this framework. This loose framework will accept and load these drivers, stubs and test cases into their respective classes defined in the framework. As the actual output would need to be compared with the expected output, these output variables would need to be stored in a file and then compared with the expected output. To redirect execution output of the unit it is required to add code in the unit for the purpose. Activity diagram (Figure 8) shows all the activities in this 'EUT' framework in brief.

Activity diagram hints some ***application logic classes***: UnitLoader, DriverLoader, StubLoader, TestcaseLoader, Executor, Comparator, TestSummaryReportGenerator. These classes are shown in Figure 9.

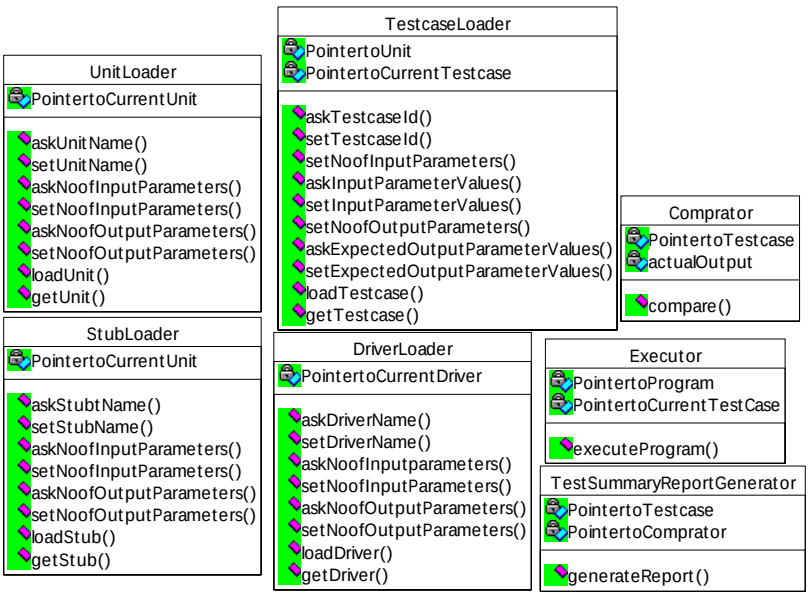

Figure 9. Control logic classes for loose framework

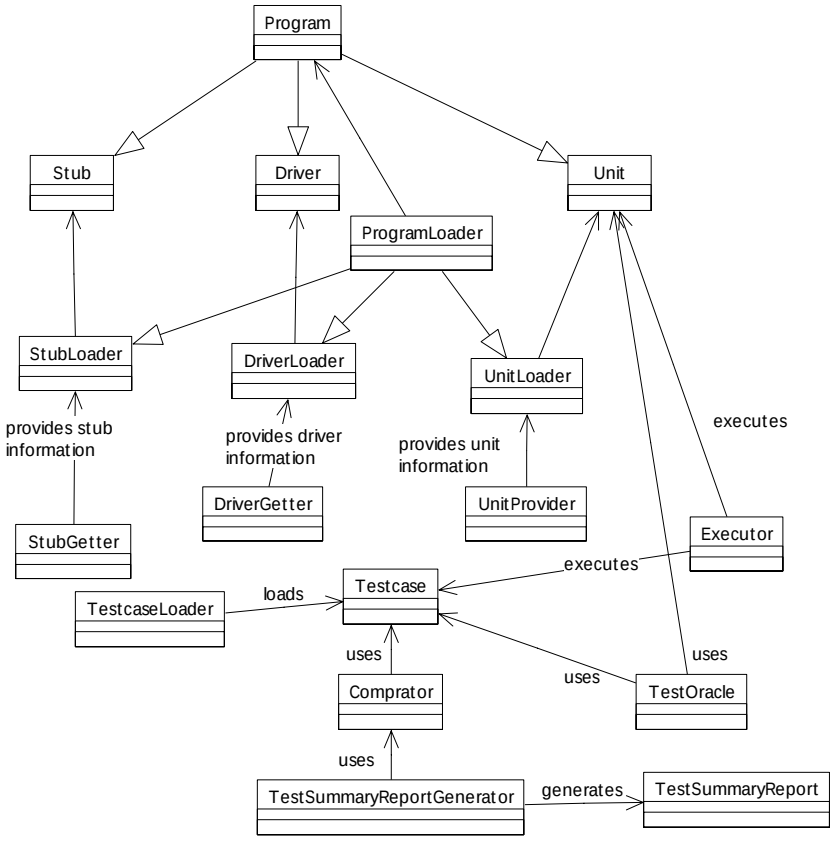

Figure 10. Object diagram of a loose framework

The user interface classes for this loose framework are same as they were in the above tight framework. Only the *TestCaseInputGetter* class is not included in this framework.

In this framework also, we take *Program* as an abstract class from which *Unit*, *Driver* and *Stub* classes would inherit properties and operations for the same reason as described in tight framework. Similarly, *ProgramLoader* is taken as an abstract class of which *UnitLoader*, *DriverLoader* and *StubLoader* are be implemented as sub classes for the same reason. Further, *UnitLoader*, *DriverLoader*, *StubLoader* and *TestcaseLoader* are also kept as abstract classes because we don't fix in this framework how the unit, drivers, stubs or test cases are generated. Thus, these classes are hot spots of the framework. At the time of instantiation of this framework one needs to refine these in subclass according to the application using the framework. After identifying the attributes and methods of these application-logic classes along with relationship among these and problem domain classes we get the object diagram (Figure 10).

5.3 Instantiation

We have instantiated this framework by adding five classes *UnitLoader_Sub* a sub class of *UnitLoader*, *DriverLoader_Sub* a sub class of *DriverLoader* and *StubLoader_Sub* a sub class of *StubLoader Test-CaseLoader_Sub* a sub class of the class *TestCaseLoader* and a class *GUI*. Except the TestCaseLoader_Sub rest of the classes have the same role and responsibilities that they have in the earlier described tight framework for 'EUT'. Since activities concerning with these classes were not fixed in the above tight framework also. Using this loose framework since the generation of test cases is not fixed in the framework there is a need to customize this activity also and thus the *TestCaseLoader_Sub* sub class is also added in the instantiation of this loose framework.

6. Observations

The objectives of the above case study are to obtain quantitative characteristics of frameworks for the purpose of comparing and understanding which framework (tight or loose) can better be reused in which scenario. The one main problem that we have encountered during this work is the lack of some good experimental data from real time environment that may help us to verify the proposed idea in an efficient manner.

Table 1. Comparison of tight and loose frameworks at code level

S.N.	Code Characteristics of frameworks	Tight framework	Loose framework
1.	Total number of methods	74	68
2.	Number of virtual functions	4	8
3.	Total Number of classes	18	13
4.	Number of abstract classes	4	5
5.	Size (Total number of non-commented lines of code)	624	485

A comparative table 1 shows different characteristics for loose and tight frameworks are drawn below.

By comparing frameworks at code level, we can answer the questions discussed above at some extent.

1) If a software framework has more number of abstract classes and virtual functions, the possibility of its reuse will be higher. As we know, the abstract classes and virtual functions give freedom (flexibility) to designers to instantiate a framework according to the requirements of a specific application. So such a framework can be reused in more number of applications. It can be shown from the table that loose framework for 'EUT' have 8 virtual functions and 5 abstract classes while tight framework has 4 virtual functions and 4 abstract classes. It shows the loose framework would be more reusable in terms of "number of reuses" because it can be used in more number of application developments than the tight one.

In a software framework, the abstract classes and virtual functions have to be customized at the time of instantiation of that framework. A designer/implementer would have to extend abstract classes and virtual functions according to the requirements of any specific application. If a software framework has more number of abstract classes and virtual functions then more effort would be needed to customize them at the time of instantiation. However, as autonomous, in terms of its service providing responsibilities, a framework would be that better contribution it can make in functioning of the system. It is shown in the table that loose framework for 'EUT' have 8 virtual functions and 5 abstract classes while tight framework has 4 virtual functions and 4 abstract classes. As identified in the definition of tight framework that it fixes the way of performing most of such activities in the framework itself, it is our conjecture that tight framework would be more reusable in terms of "ease of reuse". As shown in Figure 9 'TestcaseLoader' is an abstract class in the loose framework that is to be implemented according to the application specific requirements and hence requires extra effort. Whereas the tight framework has a concrete 'TestcaseGenerator' class that has all the implementation and hence it can be directly used in the application.

2) As given in the definition of *loose framework* that it is a framework that does not fix the way of performing most of the activities in the framework itself. During design of loose framework one need not to write semi-code for the portion that is not concretely defined in the framework. As shown in section 4, we did not fix the unit testing criteria in loose framework so we need not to develop code for generating test cases (based on any criteria) in this framework; only the interface that would connect the test case generator is needed to be developed. However, in the case of tight framework, one has to collect all the requirements for a specific application. Since, in tight framework, we fix the way of performing most of the activities so we have to write semi-code for all the activities. As shown in section 5, for the tight framework, where we fixed the testing criteria as condition testing, we required to develop whole code for generating test cases, satisfying this condition coverage criterion, as part of the framework itself. Based on our design and development experience regarding both type of frameworks, it is our conjecture that a loose framework would always be easier to develop than a tight one.

3) Unlike a loose framework, in a tight framework we fix most of the activities, so we have to write semi-code for them. It can be shown from the table that size of tight framework (total number of non-commented line of code) is 624 while size of loose framework is 485. And thus, the tight framework will be heavier, in terms of size, as compare to loose framework.

In case of tight framework, the interdependence among the different component of the framework would be more because the way of performing most of the activities are fixed. In order to understand/modify one component, one has to understand all the related components that make the tight framework more complex. In case of loose framework, different components are loosely coupled to each other. It is easy to understand/ modify one component, without understanding/modifying other components, in a loose framework because for each activity there would be perhaps different loose frameworks that fulfill the application need by interact-

ing with each other. As in our case, loose framework for 'EUT' interact with any of the test case generator framework and perform the unit testing. Thus whenever we need to modify in test case generator framework there is no need to understand the 'EUT' framework and vice-versa. Thus we can say a loose framework would always be less complex than a tight framework for the same domain. In short, we can say that the complexity of the tight framework would be high because several activities considered in that framework may result in tightly coupled implementation of them. However, the loose framework that contains only the abstract code would be less complex since no detailed implementation is there in its code.

One can consider the basic guiding principles for designing a software framework based on the above observations.

7. Conclusions

For some situations a tight framework would be better than a loose framework for same domain if the ways of performing the activities, fixed in the tight framework, are exactly those that are required in the needed application. In this paper, we suggested some scenarios, which type of framework would be more appropriate as compared to other one. These observations will be useful at the time of selection of frameworks. We have focused on limited parameters. Results would be more visible for industrial applications. One can extend this study by considering other quality criteria.

REFERENCES

[1] D. Roberts and R. Johnson, "Evolving frameworks: A pattern language for developing object-oriented frameworks," in Pattern Languages of Program Design 3. Addison-Wesley, Illinois, USA, 1997.

[2] J. Bosch, P. Molin, M. Mattsson, and P. Bengtsson, Object-Oriented Frameworks – Problems & Experiences, 1997.

[3] S. Sparks, K. Benner, C. Faris, and S. Consulting, "Managing object-oriented framework reuse," IEEE 1996, pp. 52–61, 1996.

[4] Y. J. Yang, S. Y. Kim, G. J. Choi, E. S. Cho, C. J. Kim, and S. D. Kim, "A UML-based object-oriented framework development methodology," Software Engineering Conference, Proceedings. 1998 Asia Pacific, pp. 211–218, 1998.

[5] IBM, "Building Object-Oriented Frameworks", http: //www.ibm.com/java/education/oobuilding/index.html.

[6] D. C. Schmidt, "Applying design patterns and frameworks to develop object-oriented communication software," Handbook of Programming Languages, Volume I, edited by Peter Salus, MacMillan Computer Publishing, 1997.

[7] M. E. Fayad and D. S. Hamu "Object-oriented enterprise frameworks: Make vs. buy decisions and guidelines for selection," The Communications of ACM, 1997.

[8] A. K. Tripathi and M. Gupta, "Risk analysis in reuse-oriented software development," International Journal of Information Technology and Management, Vol. 5, No. 1, pp. 52–65, 2006.

[9] P. Jalote, "An integrated approach to software engineering," ISBN 81-7319-702-4, Narosa. 2005.

