

# Using a PDU and Scenario Based Methodology in Testing Object-Oriented Programs

**Pierre Innocent, Member, IEEE**

Tsert Inc.

C.P. André-Grasset,

Box#23801

Montréal, Québec,

Canada,

H2M2W6

contact@tsert.com

<http://www.tsert.com/>

**Abstract** - This paper presents our black-box approach (*Tsert Method* ©<sup>®</sup><sup>™</sup>) in testing object-oriented programs based on the use of protocol data units to communicate with a test-harness, which are built by processing the methods of a given class. Testing object-oriented programs has always been difficult, especially in handling inheritance and polymorphism. The approach to be presented, allows the tester, to test classes in a bottom-up manner, thereby handling inheritance and polymorphism, as the subclasses and classes are processed.

The use of Protocol Data Units (PDUs) eliminates the need to generate stubs for classes and constructors. Our black-box approach, by handling only publicly accessible constructs, retain the main

benefits of object-oriented programs, which are data hiding and abstraction.

**Index Terms** - PDU, Tsert, Black-Box, Stub, Scenario, Inheritance, Polymorphism.

## I. Introduction

**T**HE software application described in this document is used to test object-oriented software. It is used, by following the testing life-cycle methodology also described in this document called the *Tsert Method*.

This document applies to version 1.0 of *Tsert*. Earlier non-released versions did not have all of the features described in this document. The most recent version of the *Tsert* application can be obtained by accessing the website <http://www.tsert.com>.

It is assumed that the reader has a basic understanding of object-oriented languages such as *C++*, *Java*, or object-based languages such as *ADA*, as well as a basic knowledge of testing of object-oriented languages. Documentation for *Tsert* is accessible through the Online-Help module of *Tsert*.

#### A. Nomenclature

We use the word *method*<sup>1</sup> to mean the member functions of *C++* and *Java* programs and procedures and functions of *Ada* programs; the word *class* to refer to *C++* and *Java* classes and to packages in *Ada*; and the word *structure* to refer to *C++* structures and *Ada* records; the words *header file* to refer to any file containing class or package specifications.

## II. Problems

#### A. Class and Method Stubbing

Large object-oriented software projects usually have a large number of interacting components or classes. The dependencies [1] or linkage between these components include containment, inheritance, reference parameters, and return values. The use of isolation testing usually fails [1], because of the need to *create a large number of test stubs* to cover every dependency.

#### B. Inheritance and Polymorphism

Inheritance and polymorphism are mechanisms that allow re-use of components in most object-oriented

---

<sup>1</sup>One problem with dealing with object-oriented programming is the proliferation of terms. Each new programming language gives the same object-oriented idea a new name (Journal of Object Oriented Programming, June 1992).

languages. Testing problems may arise when a class is inherited. For example, when a class inherits a member function from its base class, it is re-used in a new context [1]. The behaviour of the inherited method can change when a virtual member function, overridden in the derived class, is called.

#### C. Test Adequacy

The main problem with test scenario, for object-oriented programs, is in generating adequate tests. Several criteria can be taken into account when generating test sequences [5] [6]. Test adequacy is especially important when dealing with multiple inheritance, virtual functions and encapsulation. For example, with two-way dependencies between classes and superclasses, different test sets may be needed at every point in the ancestor chain between the class defining the overriding method and its ancestor class defining the overridden method.

## III. Solution

Based on industry experience, unit testing has not been proven very cost-effective. Taking that fact into account, the philosophy encapsulated by the *Tsert Method*, in testing classes and methods, is to focus on what is usually referred to as *strategy testing* [3].

Test strategy is a process that is designed to search for errors. It is a sequence of calls to methods of the class under test, called a *test scenario*, that is intended to discover errors -- [4] *Steven P. Fielder, Hewlett Packard Journal, 1989*. Simple examples of test strategies are identity and set-examine tests.

The *Tsert* method is a black-box or specification-based approach [2] [5] to testing classes of object-oriented programs, may they be in *Ada*, *Java*, or *C++*. The method consists in pre-processing the header files, that is the files which contain the class specification and then generating test-harnesses and a pdu declaration files, based on the methods of the class or classes found in the header files.

The class implementation is of no interest, since the internals of the class itself are never examined. Data abstraction and hiding is maintained, since any private or protected structures are also never examined. The only constructs of interest are the public methods and variables which may be base types, structures, or classes. If the private or protected methods need to be

examined then the tester can simply make them temporarily public.

To test a class and its methods, the appropriate test-harness is executed. The test-harness waits for pdus to arrive from the test-manager of the Tsert. The pdu specifies which method is to be invoked and which parameters if any, to instantiate for that method call. A series of method calls can be made. This series of method calls is generally described as a *test scenario*.

The *test scenario* can be built in many ways; one way is to use Unified Modeling Language ( *UML*<sup>2</sup> ) state charts. A state chart shows the sequences of states that an object or an interaction goes through during its life in response to received stimuli, together with its responses and actions.

#### A. ProcessingClassesandMethods

Normally, an object-oriented programming project consists of many files, files containing class specifications and others containing class implementations. Each header file is scanned and the classes are processed to extract all public constructs, i.e. *type definitions, enums, methods, structures, and classes*. These public constructs are cross-referenced and stored into a code-database. The newly created code database is then used, whenever a header file is loaded by *Tsert*<sup>3</sup>.

#### B. MethodCallProtocolDataUnit(PDU)

The pdu declaration file is created, by examining the public methods of every class, and a protocol data unit or frame is built, based on the sequence of constructor invocations required.

Each pdu is built the following way:

1. Take a method,
2. Create a pdu for the method,
3. Assign a message identifier to the pdu,
4. Take first parameter,
5. Check if the parameter is a basetype, structure, or class,
6. If a basetype, create a field for it in the pdu,
7. If a structure, create a new field consisting of the members of that structure,

8. If a class, find the appropriate constructor method and repeat from step 2,
9. Take next parameter and repeat from step 5,
10. After all parameters are processed, repeat from step 5 with return value, if any.

An alternative way to creating PDUs is to build a hierarchy of classes mimicking a PDU, starting with the top class being the PDU class and all subclasses being fields of the pdu. The pdu class could then be serialized, sent, and received like a frame built the default way.

The *Tsert* method enables the bottom-up testing of *C++*, *Java*, *Ada* classes by generating execution wrappers around the classes and methods under test. The wrapper is seen as a stand-alone executable, i.e. the *Test-Harness*, which receives commands from *Tsert* to execute a given method. The test-harness executes the sequence of class constructor invocations, to create instances of objects to be passed as parameters. This is done, by retrieving user-defined parameter test-data from the received pdu.

By directly passing test-data, as part of the protocol data unit (pdu) information, to the class method, and using the user defined classes; the *Tsert* method avoids the *unnecessary use of class and method stubs*. The use of *diff files* is also minimised by using the data specified for the return value for the equality test and object state verification; see *Design for Testability*.

*Unit testing* of each method is simplified by allowing test data to be automatically generated based on the tester's input. Our application allows test data adequacy [5] to be improved by the specification of constraints [7] on the test data; for example, constraints on test data, for pairs of parameters, can be specified.

#### Example:

Assuming that test data was specified for every parameter, test data would be generated only for sets that fit the constraints.

<sup>2</sup>Unified Modelling Language (UML) from Rational Corporation.

<sup>3</sup>*Tsert*-ITE, Babelart Communications Inc., <http://www.tsert.com>

**Method:**

```
bool addJob(const QString& Id, const QString& Description float Salary);
```

**Constraints:**

```
$any Description $in { "SalesPerson", "Engineer" }
```

```
$and Salary = 28000.00
```

```
$and $uniq Id
```

**Example C++ Class**

```
class ValueList: public BPtrList<GenValue>{
public:
    ValueList(GenValue* val=0);
    ValueList(const ValueList& list);
    ~ValueList();

    bool useAny() const { return _use_any; }
    bool useUniq() const { return _use_unique; }
    void setUseAny(bool flag) { _use_any = flag; }
    void setUseUniq(bool flag) { _use_unique = flag; }

    ValueList* addList(ValueList* list);
    ValueList* copy() const;
    ValueList* exclJoin(ValueList*);
    GenValue* getAnyValue();
};
```

**Example Pdus**

```
pdu ValueListPdu_18{
    msg_typeint (18);
    val{
        val int;
    }
}
pdu ValueListPdu_2_19{
    msg_typeint (19);
    list{
        val{
            val int;
        }
    }
}
pdu ValueListPdu_3_20{
    msg_typeint (20);
}
pdu useAnyPdu_21{
    msg_typeint (21);
    _retval_bool;
}
pdu useUniqPdu_22{
    msg_typeint (22);
    _retval_bool;
}
pdu setUseAnyPdu_23{
    msg_typeint (23);
    flag bool;
}
pdu setUseUniqPdu_24{
    msg_typeint (24);
    flag bool;
}
```

```
pdu addListPdu_25{
    msg_typeint (25);
    _retval_{
        val{
            val int;
        }
    }
    list{
        val{
            val int;
        }
    }
}
pdu copyPdu_26{
    msg_typeint (26);
    _retval_{
        val{
            val int;
        }
    }
}
pdu exclJoinPdu_27{
    msg_typeint (27);
    _retval_{
        val{
            val int;
        }
    }
    P1{
        val{
            val int;
        }
    }
}
pdu getAnyValuePdu_28{
    msg_typeint (28);
    _retval_{
        val int;
    }
}
```

**Example Method Calls in the Test-Harness**

```
case setUseAnyPdu_23{
    bool flag_5803 = iter.getBool((PduFields){ "flag", 0 }, idx);
    ValueList_Obj > setUseAny(flag_5803);
    channel->send(success_pdu);
}
break;
case useAnyPdu_21{
    bool result = ValueList_Obj > useAny();
    bool _retval = iter.getBool((PduFields){ "_retval", 0 }, idx);
    Value value_retval = _retval;
    Value value = result
    iter.setValue(success_pdu, (PduFields){ "bool_retval", 0 }, 0, value);
    iter.setValue(error_pdu, (PduFields){ "bool_retval", 0 }, 0, value);
    if (value_retval == value) {
        channel->send(success_pdu);
    } else {
        channel->send(error_pdu);
    }
}
```

```
break;
```

### C. Handling Inheritance and Polymorphism

We must take into account polymorphism and inheritance when testing *C++*, *Java*, and *Ada* classes. The bottom-up approach adopted by the *Tsert* method covers issues such as inheritance, by allowing the tester to incrementally load and test each class with their inherited subclass, if any.

The testing is done the following way:

1. Load a class,
2. Create a test-harness for the class,
3. Create test-scenarios for the class,
4. Test the class,
5. Load a super class of the class, if any,
6. Repeat step 1 to 5, until top class is found, or
7. Merge methods of the class, *if an abstract subclass*, with those of the class under test,
8. Repeat step 1 to 5, until top class is found.

Polymorphism is represented by the use of virtual functions; functions that are superseded by other declarations in super-classes. Polymorphism is partly handled by allowing the tester to test each class separately or in conjunction with their inherited subclasses, if any.

Dependency between classes is handled by allowing the tester to merge dependent classes into a single test-harness; and to create test scenarios based on these dependencies. Therefore up-down and side-ways dependencies, between classes, are covered with the use of a state chart which includes all the specified classes. This overall state chart is comparable to a protocol state machine with its Sub-FSMs (Sub finite State Machines).

### D. Design for Testability

In order to design software that can be tested easily with our *Tsert*<sup>4</sup> application, the developer needs to do the following:

1. Provide an equality method for most if not all non-base or non-abstract classes, which is usually represented by the equal operator =.
2. Specify a return value for most if not all non-constructor methods, e.g. a *boolean* return value specifying the result of the method invocation.
3. For applications with a Graphical User Interface (GUI), an equality method for images is required.

4. Write the class documentation in the header-file as if writing the requirements. Each method can be associated with an *XML/HTML* tag representing a keyword (*Tsert Process* ©<sup>5</sup>) linking the method requirement to a test sequence or scenario. That keyword is used to track the requirement with results obtained when executing the test scenario.
5. Specify in the requirements, which class state of set of class states may lead to an error.
6. Develop *UML* state charts for all if not most classes, which can be used to generate test scenarios. These scenarios are stored as separate test cases.
7. Develop parameter test data for the method invocations, with or without constraints, which can be easily inserted into the test scenarios.

## IV. Tsert Integrated Testing Environment

The *Tsert ITE* is a black box oriented testing environment. It helps to administer the creation and execution of testsuites, mainly for protocol entities such as modems, and other communication hardware. *Tsert* loosely follows *ISO* methodology. The user interface was developed with the Qt GUI toolkit on Linux. All file formats are stored and retrieved in *XML*.

The immediate users of the *Tsert ITE* are developers wanting to do unit-testing and testers doing integration testing. Developers may use the tool after completing the implementation of a class and wanting to do unit-test. *The Tsert ITE* helps the unit-tester by easing the generation of unit-test data with a test-data generation module called the *Tsert-Generator* and allows re-testing of previously developed classes from saved test scenarios with *the Tsert-Recorder* module.

*Tsert* provides the following features and modules:

- Graphical User Interface.
- QA project creation module.
- Test plan creation module.
- Automated test data generation module, with **C++**, **Java**, and **Ada** class/package testing support.

<sup>4</sup>*Tsert-ITE*, Babelart Communications Inc., <http://www.tsert.com>

<sup>5</sup>*Tsert Process*, testing lifecycle used by the *Tsert-ITE* application.

- Testsuite editor/browser which manages the creation, modification, and version control of test scripts.
- Hierarchical Testsuite Structure with scoping rule for declarations.
- Default and Requirement Tracking with Mysql database engine.
- Execution of selected test scripts, in batch or interactive mode.
- Conditional execution of testcases, based on prior testcase verdict.
- Proprietary scripting language with threading capabilities.
- Protocol Data Unit (PDU) or Frame reception and transmission capability.
- Support for other scripts languages e.g. perl, expect, tcl, etc.

*Tsert* can act as a Test Management Tool, when used with scripts written with language other than the *Tsert* script language.

#### A. *Tsert-Project*

The *Tsert-Project* module allows the user to create or update a project. A *Tsert* project is a way for the user to specify, from the beginning the type of quality assurance tasks the user intends to perform. If the user intends to test *C++* classes; then the user must select the *C++* option. The user must provide the directories where the *C++* header files can be found. The user can optionally specify whether or not, requirement documents, test suite structure, and tracking database are supposed to be created automatically.

The *Tsert* application processes all *C++* header files, to create a text database of the *C++* constructs, found in the header files. It can therefore, automatically generate a test suite structure, and requirement documents based on the content of that database.

#### B. *Tsert-Planner*

The *Tsert-Planner* helps you write basic system test plans. A basic template is provided, and you may add or remove sections from that template, depending on your requirements. The template is structured as a tree or table of contents.

The *Tsert-Planner* also allows you to manage the access to a testsuite's groups and subgroups, within the system test plan. Each group of testcases can be assigned to a

different developer, by restricting access to them, according to the user's login and group id

The *Tsert-Planner* can also be used to write requirement documents. Each section in the requirements document is expected to have an associated keyword. It is these keywords that you are supposed to use when making a reference to a requirement in the test scripts that you will develop later. These same keywords are subsequently traced and logged to the default tracking database when the testsuite is run using the *Tsert-Engine*.

#### C. *Tsert-Browser*

The testsuite structure editor/browser *Tsert-Browser* allows the user to create, edit, and delete groups, subgroups, and testcases within a testsuite structure. The user can invoke version control functions on groups, subgroups, and testcases, or the whole testsuite. Syntax checks can be done on a single testcase, groups of testcases, or the whole suite. Execution is also invoked the same way, on parts of the testsuite or the whole testsuite.

#### D. *Tsert-Generator*

*Tsert-Generator* helps in the testing of *C++*, *Java*, *Ada* classes/packages, and in concert with the *Tsert-Recorder* enables the bottom-up testing of *C++*, *Java*, and *Ada* class hierarchies by generating execution wrappers around the classes/packages, and methods under test. The wrapper is a stand-alone executable, i.e. the Test-Harness, which receives commands from *Tsert* to execute a given method. *Tsert* passes the required method parameter data to the executable representing the class under test.

#### E. *Tsert-Recorder*

The *Tsert-Recorder* helps in the testing of *C++* and *Java* classes, and *Ada* packages. It executes method calls by sending a frame with the proper parameter values to the test-harness, and waits for a response. It helps the user generate parameter test data for method calls automatically. Sessions can be recorded and stored as straight session files or as test scripts; which can then be re-played by *Tsert-Recorder* itself or *Tsert-Browser* through a test script. The other principal feature of *Tsert-Recorder* is the automated generation of parameter test data for method calls, based on the user's input.

### F. *Tsert-Engine*

The *Tsert-Engine* is used to invoke a syntax check or execution on a set of testcase files. The declaration files are automatically selected for processing if they are found at or under the selected node. The *Tsert-Engine* dialog window shows you the following choices:

- Enter the testsuite and System Under Test names and versions.
- Check **Execute** to execute the selected testcases.
- If an official run is to be carried out; check **Official Run** and enter a run name.
- Select the execution time; if **Later**, enter a date.
- Select the repository from which the testcases are taken.
- If only failures are to be processed; check **Failure**.
- If testcases, where the given keyword matches one of the requirements, are to be processed; check the **Keyword** button and enter one or more keywords, separated by commas, in the text field. Otherwise, the testcases selected in the *Tsert-Browser* are processed.
- Specify the number of times a testcase is re-executed.
- Specify the time to wait after each testcase is executed.
- Initiate and Stop Syntax Checking or Execution.

### G. *Tsert-Tracker*

*Tsert-Tracker* is the module that allows the user to query the default tracking database. The default tracking database is the database that *Tsert* uses to track the results of testcase execution and the requirements that are associated to these testcases. The default tracking database schema is a simple one. Testcases can satisfy more than one requirement. Each testcase has a verdict, failure, reason, etc.

For default tracking to work effectively, you must have, a priori, used the *Tsert-Planner* to document the requirements to be considered when a testsuite is developed. In that case, you are expected to specify a keyword for each section that corresponds to a requirement specified in a given testcase developed for a testsuite. The keyword is required because it is subsequently traced when the given testsuite is run. Each requirement specified in a testcase is entered into the default tracking database.

*Tsert-Tracker* connects to and queries the tracking database. You must first specify the appropriate settings for the database connection in the database preferences tab dialog, which is accessible, by clicking on the button with the **red check** mark in the **tools** toolbar. *Tsert-Tracker* can also be used as a general database viewer. When in test mode, *Tsert-Tracker* is used to test user-defined database schemas, allowing the user to automatically generate test data for SQL queries.

### H. *OnlineHelp*

You can access the *Online-Help* information by clicking on the button with the letter *i* displayed in blue in the *Tsert* tools toolbar. The On-line Help module is divided into two windows. One window displays the contents index that allows you to directly access the information that you desire. The keyword index contains all the links to the information found in the user manual information of each module of the *Tsert* application. Initially, the headers are loaded, as they are written in their respective documents, into the contents index window. You can sort these headers by clicking on the header bar at the top of the window.

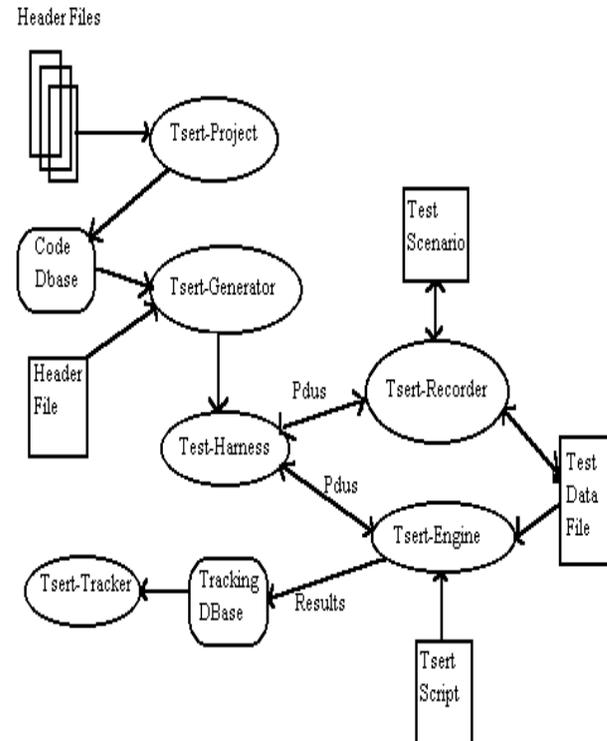


Fig. 1 - Architecture of Tsert ITE

## V. Conclusion

As we have seen, our method may not solve all object-oriented testing problems, but it does at least simplify some of the work, involved in properly generating adequate tests for these programs. The *Tsert* method negates the need for stubs. It eases test data generation for unit testing of class methods by allowing constraints, on the test data, to be specified. It automatically generates test-harnesses based on the classes found in the loaded header file(s). Inter-dependencies between classes can be covered, by generating a single test-harness using the *UML* state-chart for these classes. The testing can be eased further, if the programs are developed with the guidelines stated in the *Design for Testability* section.

## VI. References

- [1] M. Dorman, C++ “It’s Testing, Jim, But Not As We Know It”, *Proceedings of the Fifth European Conference in Software Testing, Analysis and Review*, November 1997.
- [2] B.D. Kurtz, D. Ho, T.A. Wall, An Objected-Oriented Methodology for Systems Analysis and Specification, *Hewlett-Packard Journal*, April 1989.
- [3] M.D. Smith, D.J. Robson, A Framework for Testing Objected-Oriented Programs, *Journal of Object-Oriented Programming*, June 1992.
- [4] S.P. Fieldler, Object- Oriented Unit Testing, *Hewlett-Packard Journal*, April 1989.
- [5] D.E. Perry, G.E. Kaiser, Adequate Testing and Objected-Oriented Programming, *Journal of Object-Oriented Programming*, Vol 2. No. 2, January/February 1990.
- [6] R.V. Binder, Modal Testing Strategies for OO Software, *IEEE Computer Vol. 29 No. 11*, November 1996.
- [7] D.M. Cohen, S. R. Dalal, J. Parelius, G.C. Patton, The Combinatorial Design Approach to Automatic Test Generation, *IEEE Software Vol. 13 No. 5*, September 1996.