# Design Manual

**for**

# Automatic Random Regression Testing with Human Oracle

*Prepared by: Ivan Maguidhir, C00002614, February 2011*

# Table of Contents

# 1. Introduction

## *1.1 Purpose*

This software design document describes a model of the architecture and system design of the AutoUnit product (an implementation of the Automatic Random Regression Testing with Human Oracle project). Its purpose is to describe the different parts of the system and to demonstrate how they cooperate in order to satisfy the requirements outlined in the Software Requirements Specification. The document should serve as a reference to developers working on the implementation of the system as code.

## *1.2 Scope*

AutoUnit is a software tool for analysing ANSI C source code, provided to it by a user, generating tests for each function in the source code using the CppUnit implementation of the xUnit testing framework and running these tests periodically. The objectives of the software are:

- To reduce the effort required by the developer for testing by suggesting tests and running them automatically going forward.
- To reduce errors in the developers code and therefore save time which would normally be spent debugging.
- To continuously attempt to uncover errors in previously error-free code

From a management and business point of view the objective of the software is to reduce the time and cost associated with development and the provision of technical support.

## *1.3 Goals and Requirements*

The model presented in the document addresses the following goals and requirements:

- Functional Requirements:
  - Create tests for specified source-code
  - Configure application settings
  - List tests currently managed by the application
  - Remove tests which are no longer needed or invalid due to merged code etc.
- Design: The core functionality must be contained within a library which can be used by the command-line application. This allows the product features to be reused in other applications in the future.
- Localization: The application must use Unicode throughout.
- Reporting: The application must log test failures.
- Auditing: The application must log its start time, end-time and any errors.
- System Management: The application uses two configuration files: one for application settings and one containing the list of current tests (test database). These are stored as XML which can be read, edited and copied between installations of the application.
- Workflow: Commands should be individually accessible from the command-line where

possible (i.e. with the exception of the create command which requires interaction always)

- Usability: Command-line interface should provide a usage screen, short and long versions of commands and detailed error messages
- Reliability: Application should be able to cope with missing configuration file / test database
- Localization: Application text strings with the exception of terminology should be stored externally (e.g. in XML with a locale identifier) where it can be translated easily.
- Compatiblity: The application needs to be compatible with all major operating systems Linux, Windows and Mac.

# 2. System Overview

AutoUnit is an application which generates unit tests with random values for source-code written in the C programming language. The application consists of a library which contains the core functionality of the software and a command-line user interface which accesses the library functions making them available to the user. In order to generate tests the application parses C source-code and identifies function definitions. It then observes the data-type of each function's parameters to determine what values need to be generated. The command-line application consults with the user following initial test generation and execution to validate the result. Once validated, the tests are added to a test database. The command-line application is designed to be run periodically using a task scheduler in which case it calls on the library to run a set of tests from the test database and save the test results for later retrieval by the user.

# 3. System Architecture

## *3.1 Architectural Strategies*

### 3.1.1 Third-party libraries

#### *CppUnit*

The CppUnit library implements Kent Beck's unit testing patterns for the C++ language. Unit tests written with CppUnit are written in C++ as well as testing C++ source-code. This product will process C source-code only however the test cases which it generates will consist of C++ source-code using the CppUnit library. Each generated test case will call a C function. We will use this xUnit framework instead of one specifically written for the C programming language as most of our research was carried out on it. Additionally, it will be easier to add support for the C++ language later if we use CppUnit from the outset.

#### *Xerces-C*

The Xerces-C library provides advanced XML parsing and generation functionalities. XML has been decided upon to represent the application configuration and the test database on disk. Using XML for these files allows an administrator to visually inspect both, backup, restore, deploy them across many computers running the product. The XML format used must be human-readable and only contain elements (attributes will not be used).

### *Cross-platform Library*

A decision has been made not to use a cross-platform library such as Qt or wxWidgets for the following reasons:

- The application does not contain a large amount of operating system specific code.
- It can be difficult to build libraries within an environment such as the Minimal GNU on Windows.
- Our developers are familiar with the Standard Template Library which is universally available. We will use it where we would have used classes provided by the cross-platform library (e.g. string and array classes).

Instead, a set of files containing operating system specific code will be developed. The files will contain conditional definitions of the required functions for each OS.

## *3.2 Architectural Design*

The software consists of two components:

- a library which contains functions fulfilling most of the requirements of the software
- a command-line application which acts as a user interface and fulfils requirements that involve user interaction (i.e. the “Human Oracle” part of the project proposal description)

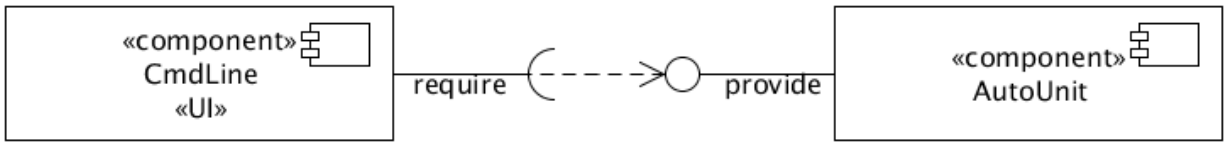



*Illustration 1: Component diagram*

## *3.3 Decomposition Description*

### 3.3.1 CmdLine component

This component consists of a single class which processes command-line parameters, calls the appropriate library functions and displays output on the console.

### 3.3.2 AutoUnit library component

This component consists mainly of the following classes:

- AutoUnit – a class which provides instances the library
- Parser – a class containing code mostly generated by Lex and Yacc for parsing C source-code
- Log – a class which records messages with a time-stamp
- Configuration – a class which stores the location of the compiler and linker and any flags

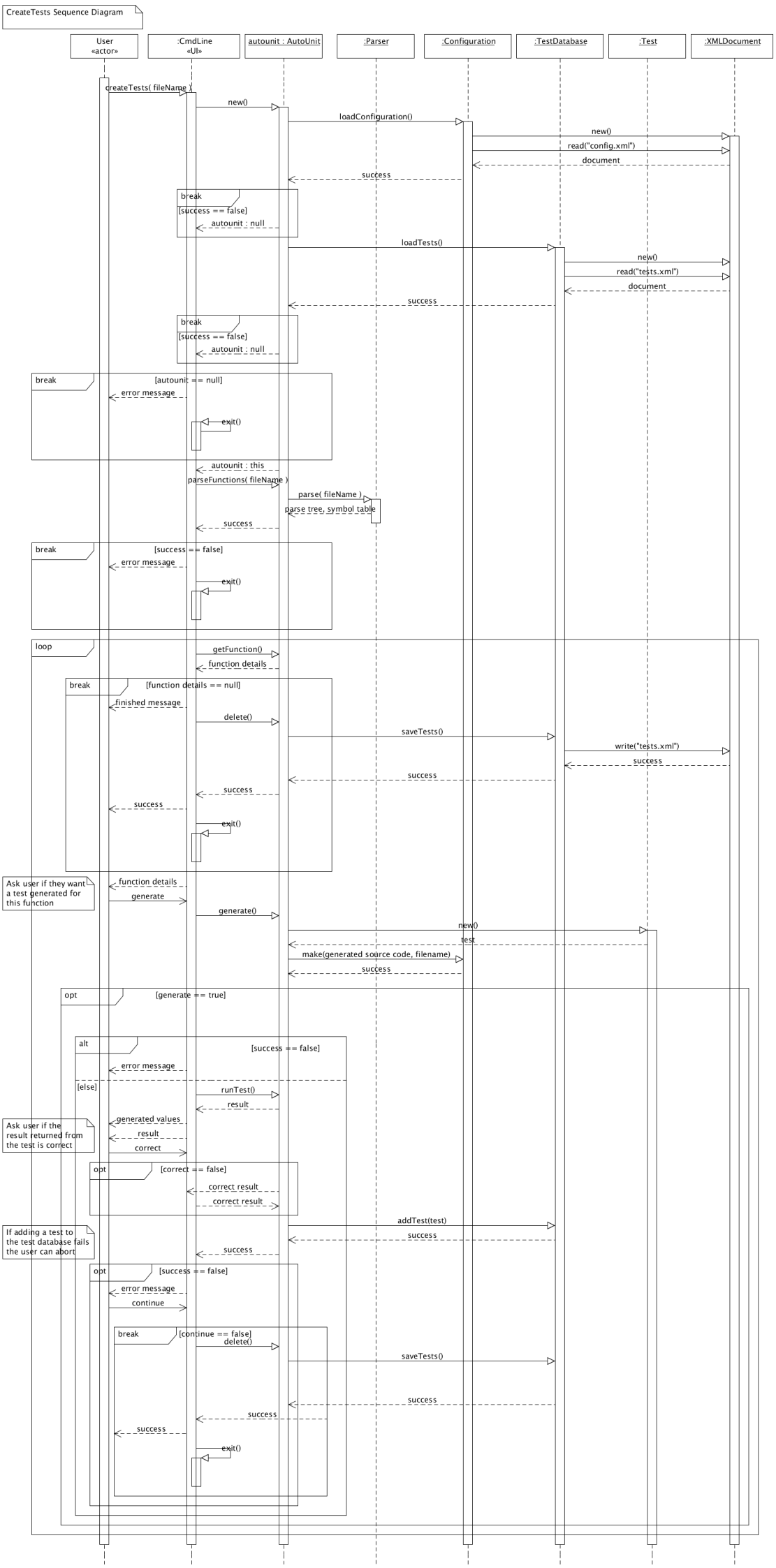
that they require. It is also responsible for building source-code.

- TestDatabase – a class which manages a set of test objects
- Test – a class whose objects contain every detail of a test (e.g. location in source-code, function name, parameters and expected result)
- XMLDocument – a class used by Configuration and TestDatabase to read and write XML

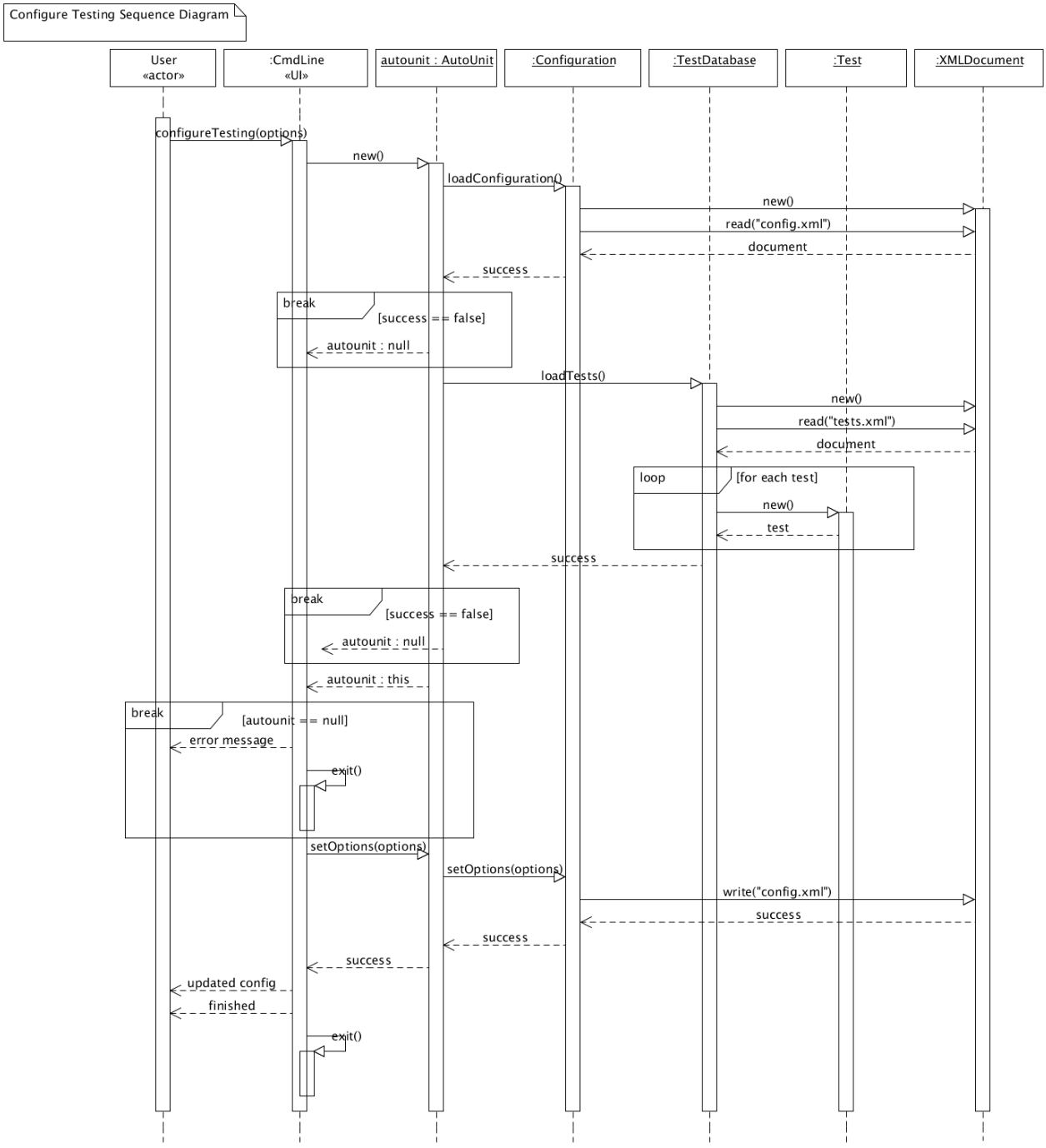
### 3.3.3 Sequence diagrams

The following sequence diagrams show the cooperation between the two components and their classes when each of the Use Cases defined in the Software Requirements Specification occurs.

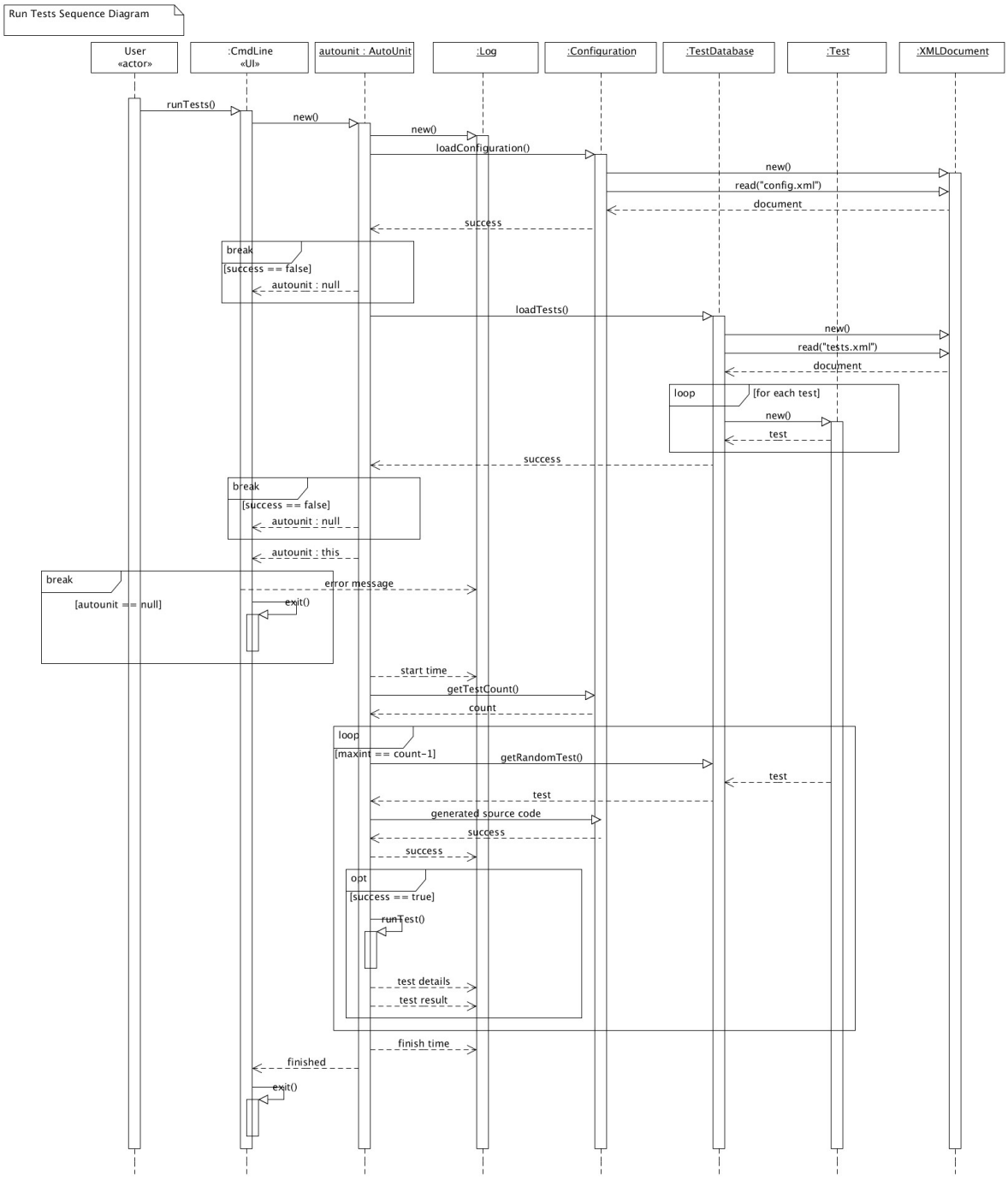
#### *Create Tests*


CreateTests Sequence Diagram
User
«actor»
:CmdLine
«UI»
autounit : AutoUnit
:Parser
:Configuration
:TestDatabase
:Test
:XMLDocument
createTests( fileName )
new()
loadConfiguration()
new()
read("config.xml")
document
success
break
[success == false]
autounit : null
loadTests()
new()
read("tests.xml")
document
success
break
[success == false]
autounit : null
break
[autounit == null]
error message
exit()
autounit : this
parseFunctions( fileName )
parse( fileName )
parse tree, symbol table
success
break
[success == false]
error message
exit()
loop
getFunction()
function details
break
[function details == null]
finished message
delete()
saveTests()
write("tests.xml")
success
success
success
success
exit()
Ask user if they want a test generated for this function
function details
generate
generate()
new()
test
make(generated source code, filename)
success
opt
[generate == true]
alt
[success == false]
error message
[else]
runTest()
result
generated values
result
Ask user if the result returned from the test is correct
correct
opt
[correct == false]
correct result
correct result
addTest(test)
success
If adding a test to the test database fails the user can abort
success
opt
[success == false]
error message
continue
break
[continue == false]
delete()
saveTests()
success
success
success
exit()

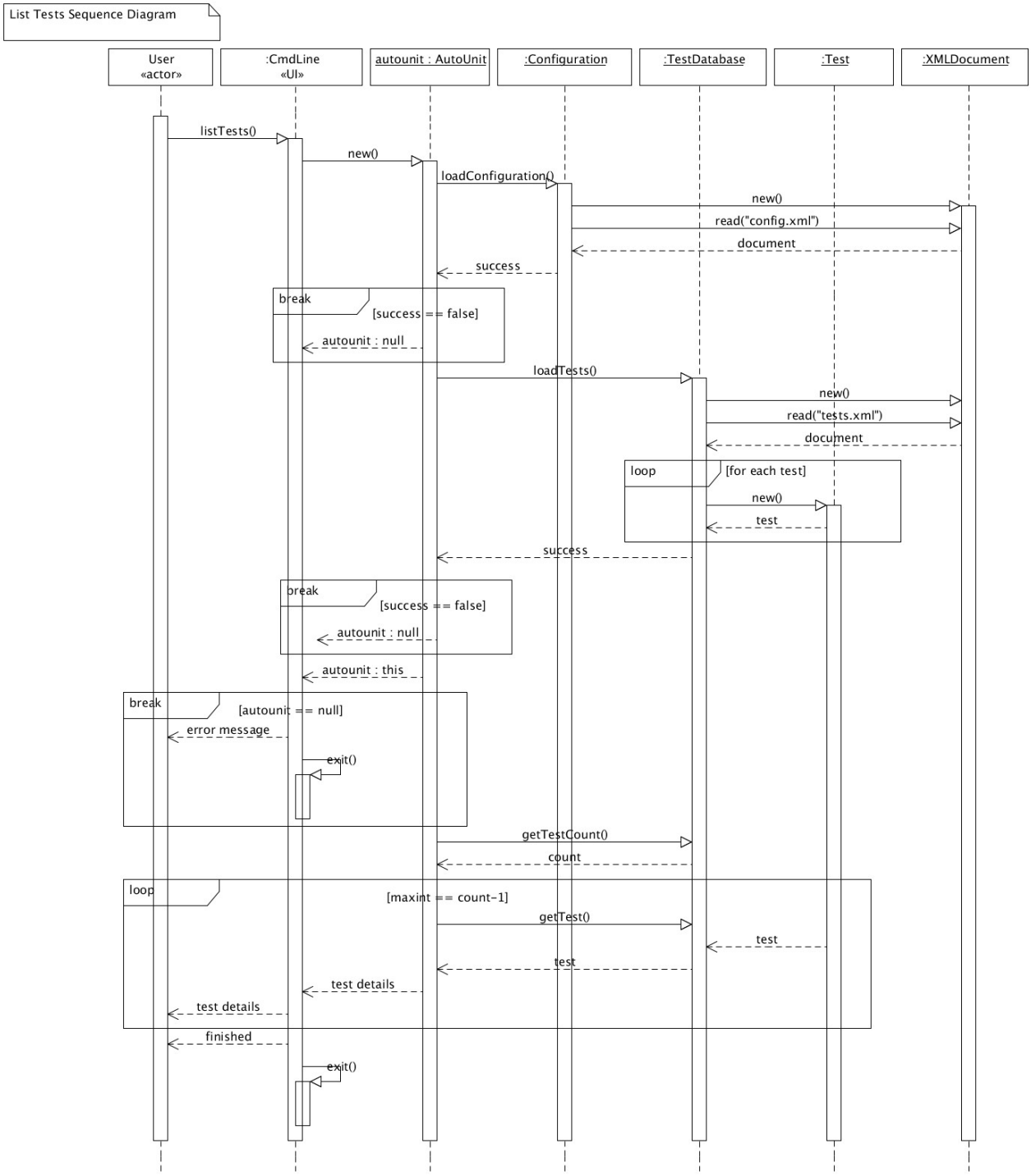
## *Configure Testing*



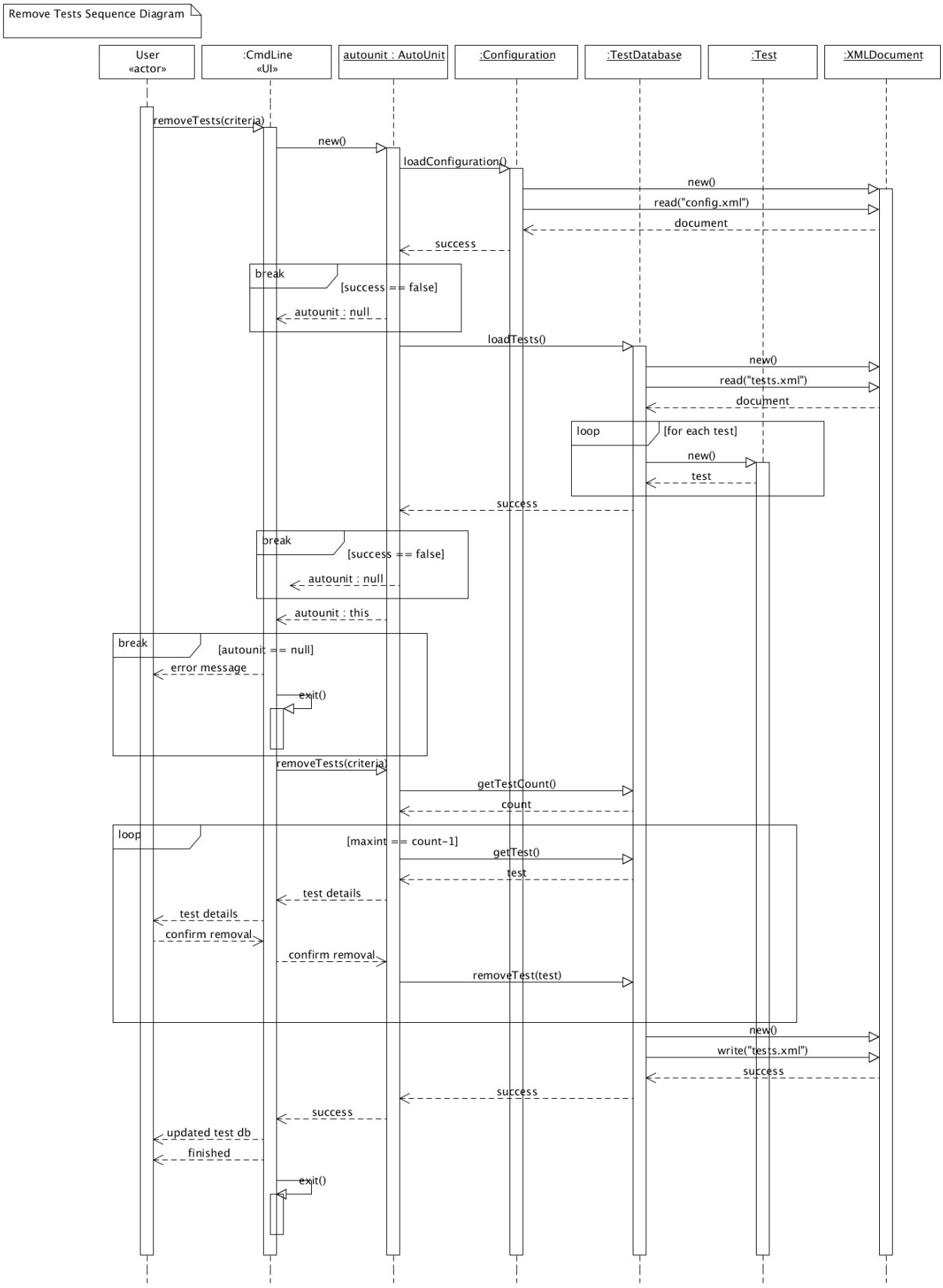
## *Run Tests*


Run Tests Sequence Diagram
User «actor»
:CmdLine «UI»
autounit : AutoUnit
:Log
:Configuration
:TestDatabase
:Test
:XMLDocument
runTests()
new()
new()
loadConfiguration()
new()
read("config.xml")
document
success
break
[success == false]
autounit : null
loadTests()
new()
read("tests.xml")
document
loop
[for each test]
new()
test
success
break
[success == false]
autounit : null
autounit : this
break
[autounit == null]
error message
exit()
start time
getTestCount()
count
loop
[maxint == count-1]
getRandomTest()
test
test
generated source code
success
success
opt
[success == true]
runTest()
test details
test result
finish time
finished
exit()

## *List Tests*


List Tests Sequence Diagram
User
«actor»
:CmdLine
«UI»
autounit : AutoUnit
:Configuration
:TestDatabase
:Test
:XMLDocument
listTests()
new()
loadConfiguration()
new()
read("config.xml")
document
success
break
[success == false]
autounit : null
loadTests()
new()
read("tests.xml")
document
loop
[for each test]
new()
test
success
break
[success == false]
autounit : null
autounit : this
break
[autounit == null]
error message
exit()
getTestCount()
count
loop
[maxint == count-1]
getTest()
test
test
test details
test details
finished
exit()

## *Remove Tests*



### *3.4 Design Rationale*

The command-line application and library together fulfil all of the requirements set out in the Software Requirements Specification. The rationale behind keeping them separate is that a different user interface (e.g. A graphical one) can be swapped into the project with ease because most of the work performed by the application is done in the library. We may also want to use the library to develop plug-ins etc., for different development IDEs in which case a UI might be generated by implementing some interface provided by the IDE.

# 4. Data Design

## *4.1 Data Description*

### Test data

The application creates an instance of the Lex & Yacc based Parser class when it needs to parse source-code (i.e. during a Create Tests operation). Once parsing has completed the Parser object provides the application with a set of parse trees, one for each function found in the source-code. Each of the nodes in these trees is actually an instance of a class defined by the application. A node is only included in a parse tree if the token it represents is one of the following:

- Function definition
- Parameter list
- Identifier
- Type specifier

The application creates a Test object for each parse-tree returned by the Parser object and populates this with information by traversing the parse-tree and interpreting the data which each node provides. These Test objects are temporary during the Create Tests use case until the user confirms the test and supplies the expected result (if the generated result was incorrect) at which time they are passed to the Test Database object for storage. The Test Database object, like the Configuration object, is instantiated when the application is initialized and is not destroyed until the application is exiting. The Test Database class contains code to allow persistence of Test objects by serializing them to XML using an XML Document object. The data component of each parameter is converted to Base64 encoding before being serialized to XML. Base64 is an encoding for binary data which allows it to be stored and transmitted as text. When the Test objects are being de-serialized from XML their data needs to be decoded from Base64 to binary again. Each Test object contains an array of Parameter objects which describe the individual parameters of the function the test is to be carried out on. The first Parameter in the array always represents the return type and value of the function. In order for the application to generate random values for a test, the Test object provides a method for generating random values for its array Parameter objects. This method, in turn, calls a method provided by each Parameter object which causes them to generate a random value for themselves.

### Configuration data

The configuration information for the application consists, simply, of a list of string and integer

values. The application instantiates a Configuration object when the application starts and this object remains alive for the period of time that the application is running. The Configuration object is responsible for storing the list of strings which represent the locations of the compiler and linker, flags and options (i.e. include paths, library names) which the user wants to be used when their source-code is being compiled to be tested. The responsibility of compiling and linking user source-code has been delegated to the Configuration object since it contains most of the relevant information. It provides this service to the rest of the application through its make() method.

## *4.2 Data Dictionary*

The following is an overview the objects which are required for handling the applications data:

**Test**

function : String

location : String

parameters : Parameter[]

...

generateParamValues()

**Parameter**

name : String

type : enum

data : byte[]

size : int

...

generate()

**Test Database**

tests : Test[]

addTest(Test)

removeTest(Test)

load(String)

save(String)

**Configuration**

compilerPath : String

linkerPath : String

…

load(String)

save(String)

make(String)

# 5. Component Design

## *5.1 CmdLine*

### 5.1.1 Classification

CmdLine is a class.

### 5.1.2 Definition

CmdLine provides the software's command-line interface.

### 5.1.3 Responsibilities

- Parse command-line parameters
- Call the required AutoUnit library functions based on parameters
- Take input from the console
- Display messages (including errors) on the console

### 5.1.4 Constraints

None.

### 5.1.5 Composition

No subcomponents.

### 5.1.6 Uses/Interactions

CmdLine is statically linked to the AutoUnit library

### 5.1.7 Resources

None.

### 5.1.8 Processing

- Creates an instance of the AutoUnit library
- Checks each command-line parameter against a list of valid commands
- Checks that the required number of supplementary parameters have been provided
- Calls library functions corresponding to the particular command issued by the user

- Provides output on the console (to indicate success / failure, error messages etc.)
- Takes input from the user if required

### 5.1.9 Interface/Exports

None.

## *5.2 AutoUnit Library*

### 5.2.1 Classification

AutoUnit is a library.

### 5.2.2 Definition

AutoUnit provides the core functionality of the software through a Library class which it exposes using a software interface.

### 5.2.3 Responsibilities

- Allow creation and storage of test details
- Allow the application to be configured and store the configuration
- Compile user source-code using an external compiler and linker
- Dynamically link to compiled user source-code and call functions to test them
- Save test results
- Log application errors

### 5.2.4 Constraints

None.

### 5.2.5 Composition

Parser class – composed main of code auto-generated by Lex & YACC.

Config class – which manages settings for the application and serializes them to/from XML

Test class – which holds information describing a test

TestDb class – which manages a list of Test objects and serializes them to/from XML

XMLDocument class – which provides XML reading and writing capabilities to other classes

Log class – which appends time-stamped messages to a specified text document

### 5.2.6 Uses/Interactions

The AutoUnit library requires the Apache Xerces-C library for the XML Document class to compile

### 5.2.7 Resources

None.

### 5.2.8 Processing

Instantiates and returns a Library class object to a calling application

### 5.2.9 Interface/Exports

Library class containing methods for:

- Creating tests
- Running tests
- Listing tests
- Removing tests
- Configuration

# 6. Human Interface Design

## *6.1 Overview of User Interface*

The user interface provided with the software is command-line based.

## *6.2 Screen Images*

### Help Screen

```
user@computer:~$ autounit --help

Usage: autounit [OPTION...] [FILE...]

        Examples:
        autounit --cf compiler /usr/bin/gcc        # set the compiler path to /usr/bin/gcc
        autounit --create /home/user/functions.c   # interactively create tests for functions.c
        autounit --list                            # list all tests contained in the test database
        autounit --l --name calc                   # list all tests which test a function named calc
        autounit --r                               # run tests (generally called by a job/task scheduler)
        autounit --remove -p /home/user/console.c  # remove all tests associated with the /home/user/console.c file

        Main operation mode:
        -h, --help      display this help screen
        -c, --create    create tests for the specified source-code file
        -r, --run       run tests
        -l, --list      list tests
        -rm, --remove   remove tests associated with the specified file name or function name
        -cf, --config   set the specified configuration option
        -n, --name      specify a function name
        -p, --path      specify the path of a source-code file
```

### Example Create Tests interaction

```
user@computer:~$ autounit --create -p /home/user/foobar.c

Generate a test for int foo(int a, int b) y/n?y
a = -1852730991
b = 1263225675
result = 1263225675
Is this the expected result y/n?y
Test saved.

Generate a test for int bar(int a, int b) y/n?y
a = -1347440721
b = -303174163
result = -1347440721
Is this the expected result y/n?n
Enter the expected result: 54321

Test saved.

Finished.
```

# 7. Requirements Matrix

| **Requirement** | **Satisfied by** |
|---|---|
| Create tests | CmdLine, AutoUnit |
| Configure | CmdLine, AutoUnit |
| Run tests | CmdLine, AutoUnit |
| List tests | CmdLine, AutoUnit |
| Remove tests | CmdLine, AutoUnit |
| Design | CmdLine, AutoUnit |
| Localization | CmdLine, AutoUnit |
| Reporting | AutoUnit |
| Auditing | AutoUnit |
| System Management | AutoUnit |
| Workflow | CmdLine |
| Usability | CmdLine |
| Reliability | AutoUnit |
| Localization | CmdLine |
| Compatiblity | CmdLine, AutoUnit |

# 8. Appendices