{{lastupdated_at}} by {{lastupdated_by}}

Unit testing

Introduction

Unit testing is a method by which individual units of source code, sets of one or more computer program modules together with associated control data, usage procedures, and operating procedures, are tested to determine if they are fit for use.

Actual implementation

In GammaLib, unit tests verify all classes that are provided by the library. Unit testing is initiated using the

```
$ make check
```

command. So far, the unit tests are implemented using C++ executable that test the various classes. The C++ code of the test executables can be found in the test directory. Unit tests either compare the results of an operation against the expected value, or our embedded in try - catch blocks to catch unexpected features. In the JUnit parlance (see below), the first types of tests will produce failures, while the second types to tests will produce errors. So far, no distinction is made between failures and errors.

Implementing a JUnit compliant test suite

Many frameworks (such as Jenkins or Sonar) support unit test reports in the JUnit format. Also GammaLib should comply to the [[JUnit XML format specification]], so that GammaLib test results can be analyzed and tracked using standard tools. This means, GammaLib unit tests should output the test results in form of XML files that inform about the numbers of failures and errors that occur for a given series of tests. Such series will be combined in test suites, and test suites will be combined in testsuite containers.

We should thus implement a GTestSuites container that contains GTestSuite class elements. Each C++ executable should be mapped into a GTestSuites container. The tests that are done generally within a given function of the test executable (these are the functions that start with a test name, followed by a number of dots, and end by an ok) should be mapped into the GTestSuite.

The GTestSuites container class will run all tests, gather the information about failures and errors, and produce a test report in XML format (see [[JUnit XML format specification]] or [[Jenkins specification]] for the format definition). The test suite container will dump the container name to the screen in the actual format, e.g.

```
*************
* CTA tests *
*************
```

Screen dumping should be optionally disabled. Maybe the GLog class can be used for this. If one of the tests failed or had en error, the class destructor should exit with an error so that the error will be reported by make check (TBC).

The GTestSuite class will start a series of tests and collects the test information for these series. It provides this information to the container class upon request. For each successful test it will print a . on the screen, failures will results in a F, errors in a E. If all tests are successful, ok will be added at the end of the test to the dump, in case of a failure, NOK will be added. Those are followed by a linefeed. The output will be as follows:

```
My test: ....F...FEF....F NOK
Another test: ..... ok
```

Screen dumping should be optionally disabled.

Each test function will now be a test class that derives from GTestSuite, and each test will be a method of this class. The class can have data members that hold information that is passed from one test to the other. It still has to be figured out how the different test methods will register to the class, so that the class can run them one after the other. One possibility is that the class holds a list of

pointers. Here a possible implementation (not tested):

```
class GTestSuite;

typedef bool (GTestSuite::*testpointer)(void);

class GTestSuite {
public:
    GTestSuite() {}
    ~GTestSuite() {}
    void run(void) {
        std::vector<testpointer>::iterator iter;

        for (iter = m_tests.begin(); iter != m_tests.end(); ++iter) {
            try {
                if ((*this.*(*iter))()) {
                    m_success++;
                }
                else {
                    m_failure++;
                }
            }
            catch(std:exception &e) {
                m_error++;
            }
        }
    }
    std::vector<testpointer> m_tests;
};

class MyTestSuite : GTestSuite {
public:
    MyTestSuite();
    ~MyTestSuite() { }
    bool f1(void) {  std::cout << "Unit test 1" << std::endl; return true; }
    bool f2(void) {  std::cout << "Unit test 2" << std::endl; return false; }
private:
    int                 m_a;
};

void MyTestSuite::MyTestSuite(void)
{
    m_tests.push_back(static_cast<testpointer>(&MyTestSuite::f1));
    m_tests.push_back(static_cast<testpointer>(&MyTestSuite::f2));
}

int main() {
    MyTestSuite tests;
    tests.run();
}
```

That this should work is illustrated by the code below that compiles on kepler (CentOS 5) and that demonstrates how a base class can call derived class methods.

```
#include<vector>
#include<iostream>

class base;

typedef bool (base::*pointer)(void);

class base {
public:
  base() {}
```

```
  ~base() {}
  std::vector<pointer> m_fct;
  void run(void) {
     bool r = (this->*(m_fct[0]))();
  }
};

class derived : public base {
public:
  derived() {}
  ~derived() {}
  bool f1(void) { std::cout << "f1" << std::endl; return true; }
  void set(void) {
     m_fct.push_back(static_cast<pointer>(&derived::f1));
  }
};

int main()
{
  derived d;
  d.set();
  d.run();
  return 0;
}
```

The set() method can be implemented in the constructor of the derived class, so that calling the run() method will execute all tests. The run() method is implemented in the base class and takes provision for bookkeeping and error catching.

Writing unit tests

To write a test suite it is necessary to create a class that derives from GTestSuite. This class is abstract so the method void set(void) should be implemented in the derivated class.
The set method contains initialisation of the members used by the test suite and the configuration of the test functions. You can also set the test suite name in this method.
To add a test function (which must be a method of the class), use add_test(static_cast<pfunction>(&myTestSuite::my_test), " Test name");

Each TestSuite should be appended in a TestSuites container and tests are launched by the run() method.

Here an example :

```
class TestGMyClass : public TestSuite
{
public:

 testGMyClass() : Testsuite() { return; }
 // A test function
 void my_test(void){
  test_assert(1==0,"Test if 1 ==0", "Message if failure");

  // ... some code

  test_try("A specific test")
  try{
     // Some code to test

     test_try_success();
   }
  catch(exception& e)
  {
     test_try_failure(e);
  }
  return;
```

```
  }

  void set(void){

   // Set test suite name
   name("Test suite name");

   // Initialisation members
   m_var=0;

   // Add tests
   add_test(static_cast<pfunction>(&myTestSuite::my_test), " Test name");

private:
  int m_var;

};

int main(){
  // Create a conainer of TestSuite
  TestSuites testsuites("GMyClass"");

  // Create our TestSuite
  TestGMyClass testsuite;

  // Append testsuite to the container
  testsuites.append(testsuite);

  // Run all the test suite
  bool was_successful = testsuites.run();

  //save a report
  testsuites.save("reports/GMyClass.xml");

  // Return 0 or 1 depending the tests result.
  return was_successful ? 0 : 1;
}
```

Reports are saved in the test/reports repertory.
To be compatible with the make check command, the main should return 0 or 1 depending if tests passed or not.

As the example shows, it is possibile to do test in a test function.

- test_asset : To test an assertion, if it is true, test pass.
- test_try : To test some code, if an exception is throw in a test_try, it would be notice in the report and the other tests of the function are exectuted.

We can also nest test_try. In the test report the name will appear like this :
TestFunctionName:TestTryName1:TestTryName2:TestAssertName