



Parallel Fortran Unit Testing Framework

Installation, Usage, and API

Tom Clune

Advanced Software Technology Group
Computational and Information Sciences and Technology Office
NASA Goddard Space Flight Center

April 10, 2014



1 Introduction

- Overview
- Quick review of testing

2 Introduction to pFUnit

3 Advanced pFUnit

4 Test-driven development



- 1 Introduction
 - Overview
 - Quick review of testing
- 2 Introduction to pFUnit
- 3 Advanced pFUnit
- 4 Test-driven development



Primary Goals

- Learn how to use pFUnit 3.0 to create and run unit-tests
- Learn how to apply test-driven development
- Gain greater appreciation for software testing

Prerequisites

- Familiarity with F95 syntax
- Familiarity with MPI¹
- Access to Fortran compiler supported by pFUnit 3.0

Possibly beneficial skills

- Various F2003 syntax
- Object-oriented programming
- CMake

¹Only for MPI-specific sections.



- **Thursday PM - Introduction to pFUnit**
 - ▶ Overview of pFUnit and unit testing
 - ▶ Build and install pFUnit
 - ▶ API for pFUnit
 - ▶ The pFUnit preprocessor
- **Friday AM - Advanced topics (including TDD)**
 - ▶ User-defined test subclasses
 - ▶ Parameterized tests
 - ▶ Introduction to TDD
 - ▶ Examples and exercises using TDD and pFUnit
- **Friday PM - Bring-your-own-code**
 - ▶ Introduce pFUnit into the build process for your own code
 - ▶ Apply TDD within your own code
 - ▶ Supplementray exercises will be available



- These slides can be downloaded at ...
- The exercises can be downloaded at ...



1 Introduction

- Overview
- Quick review of testing

2 Introduction to pFUnit

3 Advanced pFUnit

4 Test-driven development

Quick review of testing



- What is a (software) test?
- What are the different types of software tests?
- What are desirable properties for unit tests?
- What is the anatomy of a unit test?
- What is a test “fixture”

A test by any other name ...



A test is *any* mechanism that can be used to verify a software implementation. Examples include:

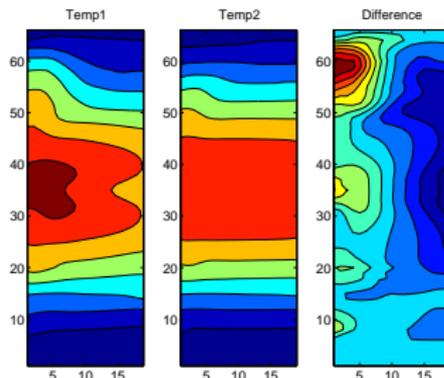
- Conditional termination during execution:

```
IF (PA(I,J)+PTOP.GT.1200.) &  
    call stop_model('ADVECM: Pressure diagnostic
```

- Diagnostic print statement

```
print*, 'loss of mass = ', deltaMass
```

- Inspection of rendered output:



Taxonomy of Testing



- Scope
 - ▶ Unit
 - ▶ Integration
 - ▶ System
- Acceptance
- Stress
- Performance
- Regression
- Access
 - ▶ Black box
 - ▶ White box

Desirable attributes for tests:



Desirable attributes for tests:



- Narrow/specific
 - ▶ Failure of a test localizes defect to small section of code.

Desirable attributes for tests:



- Narrow/specific
 - ▶ Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
 - ▶ Each defect causes failure in one or only a few tests.

Desirable attributes for tests:



- Narrow/specific
 - ▶ Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
 - ▶ Each defect causes failure in one or only a few tests.
- Complete
 - ▶ All functionality is covered by at least one test.
 - ▶ *Any defect is detectable.*

Desirable attributes for tests:



- Narrow/specific
 - ▶ Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
 - ▶ Each defect causes failure in one or only a few tests.
- Complete
 - ▶ All functionality is covered by at least one test.
 - ▶ *Any defect is detectable.*
- Independent - *No side effects*
 - ▶ No STDOUT; temp files deleted; ...
 - ▶ Order of tests has no consequence.
 - ▶ **Failing test does *not* terminate execution.**



Desirable attributes for tests:



- Narrow/specific
 - ▶ Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
 - ▶ Each defect causes failure in one or only a few tests.
- Complete
 - ▶ All functionality is covered by at least one test.
 - ▶ *Any defect is detectable.*
- Independent - *No side effects*
 - ▶ No STDOUT; temp files deleted; ...
 - ▶ Order of tests has no consequence.
 - ▶ Failing test does *not* terminate execution.
- Frugal
 - ▶ Execute quickly (think 1 millisecond)
 - ▶ Small memory, etc.

Desirable attributes for tests:



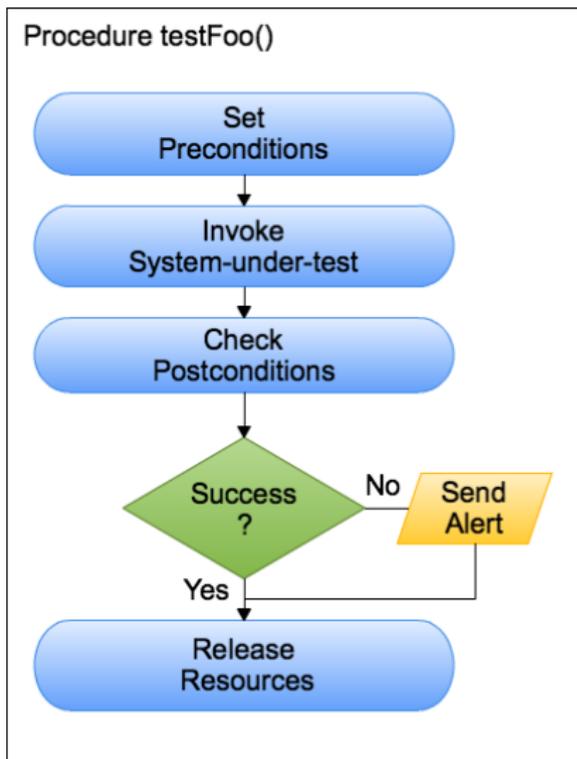
- Narrow/specific
 - ▶ Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
 - ▶ Each defect causes failure in one or only a few tests.
- Complete
 - ▶ All functionality is covered by at least one test.
 - ▶ *Any defect is detectable.*
- Independent - *No side effects*
 - ▶ No STDOUT; temp files deleted; ...
 - ▶ Order of tests has no consequence.
 - ▶ Failing test does *not* terminate execution.
- Frugal
 - ▶ Execute quickly (think 1 millisecond)
 - ▶ Small memory, etc.
- Automated and repeatable

Desirable attributes for tests:



- Narrow/specific
 - ▶ Failure of a test localizes defect to small section of code.
- Orthogonal to other tests
 - ▶ Each defect causes failure in one or only a few tests.
- Complete
 - ▶ All functionality is covered by at least one test.
 - ▶ *Any defect is detectable.*
- Independent - *No side effects*
 - ▶ No STDOUT; temp files deleted; ...
 - ▶ Order of tests has no consequence.
 - ▶ Failing test does *not* terminate execution.
- Frugal
 - ▶ Execute quickly (think 1 millisecond)
 - ▶ Small memory, etc.
- Automated and repeatable
- Clear intent

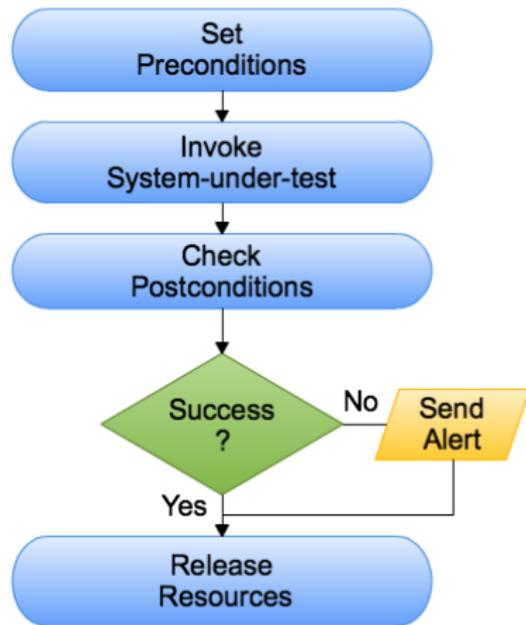
Anatomy of a Software Test Procedure



Anatomy of a Software Test Procedure



Procedure testFoo()

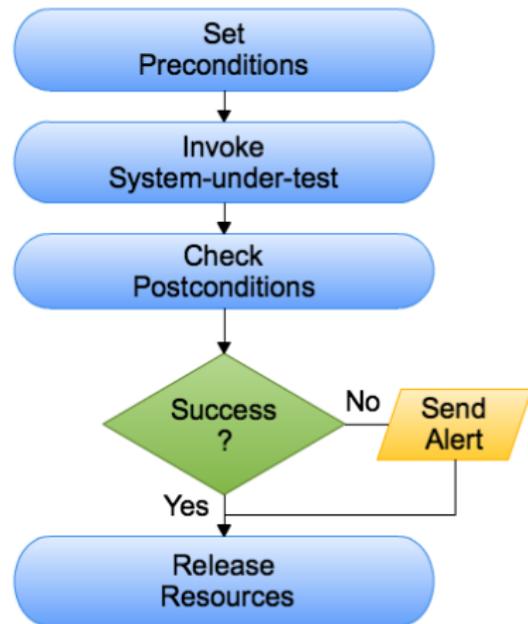


testTrajectory() ! $s = \frac{1}{2}at^2$

Anatomy of a Software Test Procedure



Procedure testFoo()



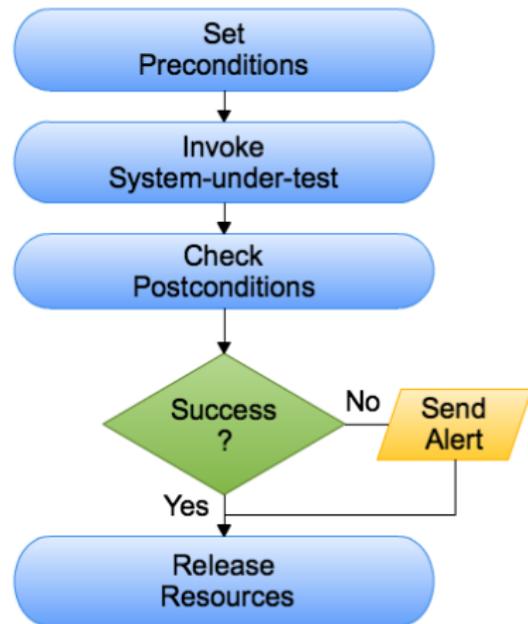
testTrajectory() ! $s = \frac{1}{2}at^2$

$a = 2.;$ $t = 3.$

Anatomy of a Software Test Procedure



Procedure testFoo()



testTrajectory() ! $s = \frac{1}{2}at^2$

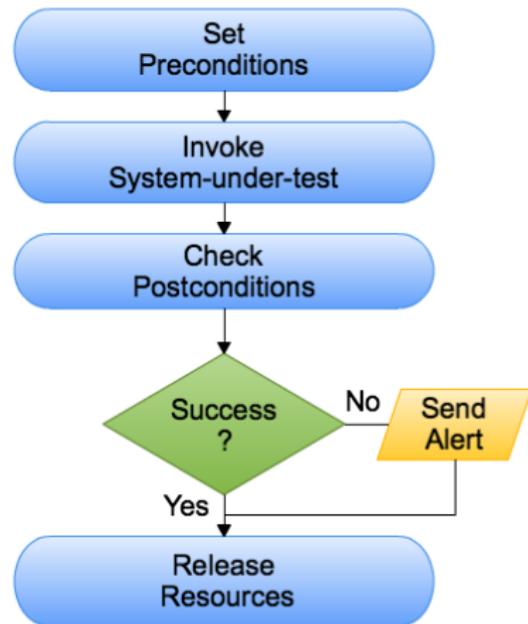
$a = 2.;$ $t = 3.$

$s = \text{trajectory}(a, t)$

Anatomy of a Software Test Procedure



Procedure testFoo()



testTrajectory() ! $s = \frac{1}{2}at^2$

$a = 2.$; $t = 3.$

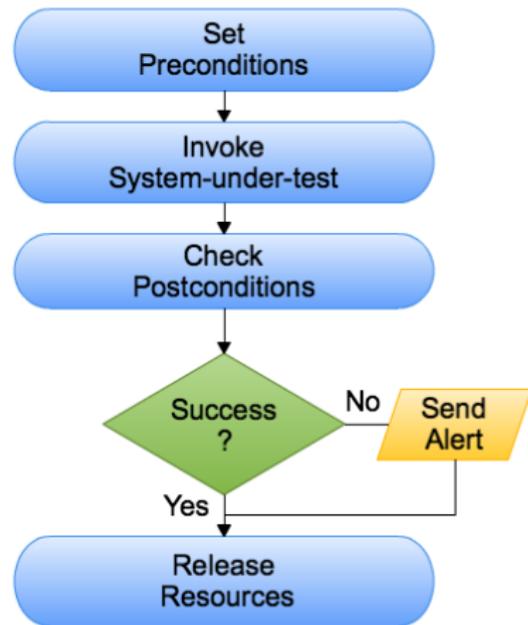
$s = \text{trajectory}(a, t)$

call `assertEqual(9., s)`

Anatomy of a Software Test Procedure



Procedure testFoo()



testTrajectory() ! $s = \frac{1}{2}at^2$

$a = 2.;$ $t = 3.$

$s = \text{trajectory}(a, t)$

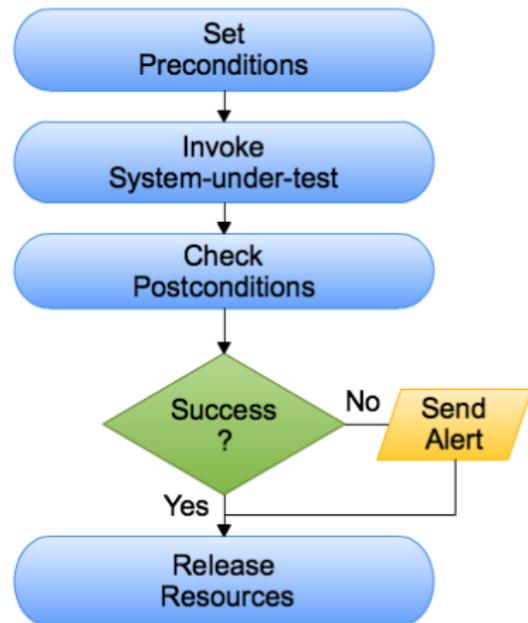
call `assertEqual(9., s)`

! no op

Anatomy of a Software Test Procedure



Procedure testFoo()



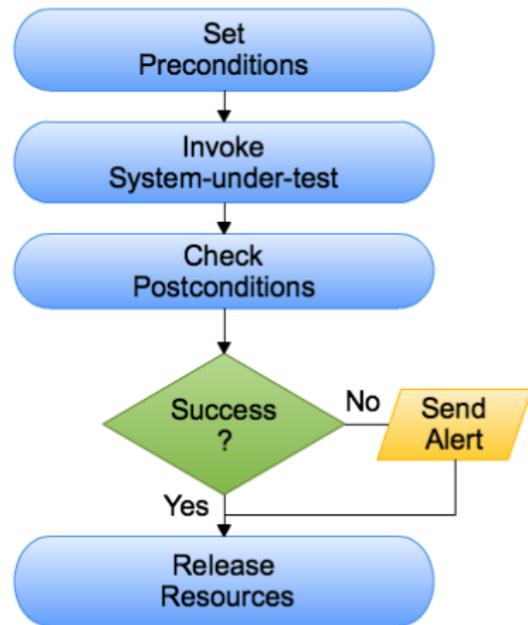
testTrajectory() ! $s = \frac{1}{2}at^2$

call `assertEqual(9., trajectory(2.,3.))`

Anatomy of a Software Test Procedure



Procedure testFoo()



testTrajectory() ! $s = \frac{1}{2}at^2$

```
@assertEqual(9., trajectory(2.,3.))  
(automatically includes  
file name and line number)
```



1 Introduction

2 Introduction to pFUnit

- pFUnit overview
- Build and Install
- Basic pFUnit Example
- Simple MPI
- Simple Fixtures (unencapsulated)
- Testing for error handling
- API
- Parser and Driver

3 Advanced pFUnit

4 Test-driven development



- 1 Introduction
- 2 Introduction to pFUnit
 - pFUnit overview
 - Build and Install
 - Basic pFUnit Example
 - Simple MPI
 - Simple Fixtures (unencapsulated)
 - Testing for error handling
 - API
 - Parser and Driver
- 3 Advanced pFUnit
- 4 Test-driven development

Noteworthy features of pFUnit 3.0



2005 Prototype²

- read a book and re-implemented with TDD

2006 Version 1.0 released as open source

2012 Began serious attempt at F2003 implementation

2013 Version 2.0 released - heavy reliance of OO

- new and improved preprocessor (test “annotations”)
- additional assertions

2013 Version 2.1 released - listened to user feedback

2014 Version 3.0 released³

- introduced cmake
- test extensions finally “easy”

²Proof to colleague that Fortran (F90) - I cheated

³Would have been 2.1, but bug in gfortran broke backwards compatibility



- Standard Fortran⁴
- Strong support for multidimensional arrays
- Testing of parallel procedures - MPI & OpenMP⁵
- Test fixtures
- Parameterized tests
- User-defined extensions (OO)
- Test annotations (via preprocessor) for greatly improved ease of use
- Improved (and maintained) examples
- Extensive regression testing with each push

⁴F2003 with a dash of F2008

⁵Threadsafe



- Website/documentation <http://pfunit.sourceforge.net>
- Mailing list: pfunit-support@lists.sourceforge.net
- This tutorial



1 Introduction

2 Introduction to pFUnit

- pFUnit overview
- **Build and Install**
- Basic pFUnit Example
- Simple MPI
- Simple Fixtures (unencapsulated)
- Testing for error handling
- API
- Parser and Driver

3 Advanced pFUnit

4 Test-driven development

Supported Configurations^{1,2}



OS	Vendor		Version
Linux	Intel	ifort	14.0.2
Linux	Intel	ifort	13.1.192
Linux	GNU	gfortran	4.8.3 ³
Linux	GNU	gfortran	4.9.0 ³
Linux	NAG	nagfor	5.3.2(981)
OS X	Intel	ifort	14.0.2
OS X	GNU	gfortran	4.8.3 ³
OS X	GNU	gfortran	4.9.0 ³
OS X	NAG	nagfor	5.3.2(979)
Externally contributed			
AIX	IBM	xlf	???
Windows	Intel	ifort	???

¹In many cases closely related compiler versions will also work.

²We are cautiously optimistic that PGI will soon be supported.

³Not yet released. 4.9.0 experimental currently works. Older versions contained an insurmountable bug.



For this discussion we will refer to 3 distinct directories:

- root - top directory of downloaded code
- build - directory in which build instructions are issued
- install - directory where various framework elements will be installed for later use.

There are 2 ways to obtain the source code for pFUnit:

- Via git (read-only):
`% git clone git://git.code.sf.net/p/pfunit/code pFUnit`
- Via tar:
`http://sourceforge.net/projects/pfunit/files/latest/download
% tar xzf ./pFUnit.tar.gz`

What's inside?



- ./source
- ./tests
- ./Examples
- ./documentation
- ./include
- ./bin
- README-INSTALL
- CMakeLists.txt
- GNUmakefile
- COPYRIGHT
- LICENSE



- 1 Set PFUNIT environment variable to desired installation path

```
bash: % export PFUNIT=<path>
```

```
csh: % setenv PFUNIT <path>
```

You may want to edit your various login scripts to automatically set this variable.

- 2 Decide whether to use CMake (recommended) or just GNU make
- 3 Choose configuration options
 - ▶ Build with MPI support
 - ▶ Build with OpenMP support



① % cd <builddir>

② % cmake <rootdir> <options>

Options:

- ▶ -DMPI=YES (include support for MPI)
- ▶ -DOPENMP=YES (include support for OpenMP)

③ % make tests

④ % make install



- ① % cd <rootdir>
- ② % make tests <options>
 - ▶ MPI=YES (include support for MPI)
 - ▶ OPENMP=YES (include support for OpenMP)
 - ▶ F90_VENDOR=<vendor> (override default vendor: Intel, GNU, NAG)
- ③ % make install INSTALL_DIR=\$PFUNIT



- 1 Download pFUnit 3.0
- 2 Build serial configuration
 - 1 Set \$PFUNIT
 - 2 Build with cmake/gmake
 - 3 Install
- 3 Build MPI configuration (unless skipping MPI exercises)
 - 1 Set \$PFUNIT (use different path than for serial)
 - 2 Build with cmake/gmake - use MPI option
 - 3 Install



1 Introduction

2 Introduction to pFUnit

- pFUnit overview
- Build and Install
- **Basic pFUnit Example**
- Simple MPI
- Simple Fixtures (unencapsulated)
- Testing for error handling
- API
- Parser and Driver

3 Advanced pFUnit

4 Test-driven development

Simple Example: testing factorial function



file: Exercises/SimpleTest/testFactorial.pf

```
@test
subroutine testFactorialA()
  use pFUnit_mod
  use Factorial_mod
  @assertEqual(6, factorial(3))
end subroutine testFactorialA
```

Simple Example: testing factorial function



file: Exercises/SimpleTest/testFactorial.pf

```
@test
subroutine testFactorialA()
  use pFUnit_mod
  use Factorial_mod
  @assertEqual(6, factorial(3))
end subroutine testFactorialA
```

Things to notice:

- Test is preceded by `@test`
- Expected results are indicated with `@assertEqual`
- One must “use” the `pFUnit_mod` module
- The “.pf” suffix is an arbitrary convention

Simple Example: testing factorial function (contd)



file: Exercises/SimpleTest/testSuites.inc

```
! Register your test suites here  
ADD_TEST_SUITE(testFactorial_suite)
```



file: Exercises/SimpleTest/testSuites.inc

```
! Register your test suites here  
ADD_TEST_SUITE(testFactorial_suite)
```

- This file is included by the driver
- Used to register all your tests
- One entry per suite; one suite per file
- Default test suite name is derived from the test file name
- “testSuites.inc” is mandatory – must be in the include path



- Assert that two items are equal:

```
@assertEqual (a, b)
```

- Assert that two items are equal to some tolerance:

```
@assertEqual (a, b, tolerance)
```

- Register a test procedure

```
@test  
subroutine testProc()
```



Exercise 1: Simple unit tests



- 1 Build the “SimpleTest” example in the distribution
 - 1 Change directory to `./Exercises/SimpleTest`
 - 2 `% make tests`
 - 3 Verify that 1 test ran successfully.
- 2 Add a new test in the file `./Exercises/SimpleTest/testFactorial.pf`
 - 1 Create a new test procedure that verifies $5! = 120$
 - 2 `% make tests`
 - 3 Create a new test procedure that verifies $0! = 1$
 - 4 `% make tests` (uh oh!)
 - 5 Fix the implementation
 - 6 `% make tests` (whew!)
- 3 Demonstrate tests as a harness
 - 1 Edit `factorial.F90`
 - 2 Insert a bug (e.g., change `'*` to `'+`)
 - 3 `% make tests`



1 Introduction

2 Introduction to pFUnit

- pFUnit overview
- Build and Install
- Basic pFUnit Example
- **Simple MPI**
- Simple Fixtures (unencapsulated)
- Testing for error handling
- API
- Parser and Driver

3 Advanced pFUnit

4 Test-driven development

Simple MPI Example: matrix transpose



file: Exercises/SimpleMpiTest/TestTranspose.pf

```
module TestTranspose_mod
  use pFUnit_mod
  use Transpose_mod
  implicit none

contains

  @test(npes=[1])
  subroutine testTranspose_1by1(this)
    class (MpiTestMethod), intent(inout) :: this

    real :: a(1,1)
    real :: at(1,1)
    integer :: comm

    ! preconditions
    a = 1
    at = 0

    comm = this%getMpiCommunicator()
```



- Declare with `@test(npes=[<integer_list>])`
- Framework runs test once for each value of `npes`
- Framework creates a *new* subcommunicator each time test is run
- Framework passes communicator through a mandatory argument
- declare as `class(MpiTestMethod), intent(inout) :: <arg>`⁶
- Passed object has several useful methods:

```
comm = this%getMpiCommunicator()  
npes = this%getNumProcesses()  
rank = this%getProcessRank()
```

- Any assertions that fail will attach information about
 - ▶ `npes` for failing case
 - ▶ `rank` for failing case

⁶The intent *must* be *inout*.

Exercise 2: Extend transpose tests



- 1 Set \$PFUNIT to the MPI build
- 2 Build the “SimpleMpiTest” example in the distribution
 - 1 Change directory to ./Exercises/SimpleMpiTest
 - 2 % make tests
 - 3 Verify that 2 tests ran successfully.
- 3 Create a new test that works for multiple values of npes.
 - 1 Create a helper function that specifies

$$a_{ij} = a_{ji}^T = f(i, j) = n_p * i + j$$

- 2 Make arrays a and a^T (expected and found) allocatable
- 3 Fill a and expected a^T using helper function
- 4 Call transpose
- 5 Assert that the found value is the same as the expected value.



1 Introduction

2 Introduction to pFUnit

- pFUnit overview
- Build and Install
- Basic pFUnit Example
- Simple MPI
- **Simple Fixtures (unencapsulated)**
- Testing for error handling
- API
- Parser and Driver

3 Advanced pFUnit

4 Test-driven development

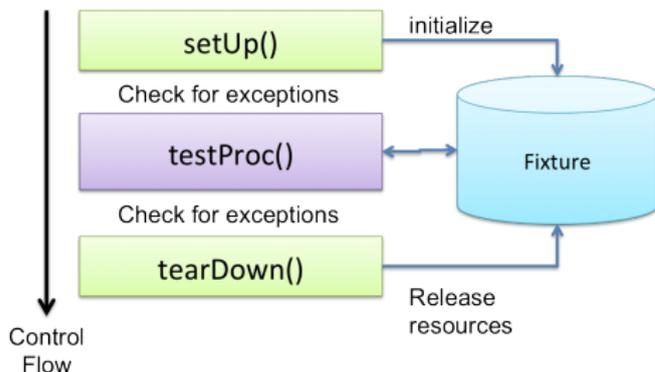
Test Fixtures



A test *fixture* is any mechanism that allows a consistent initialization for test preconditions.

- Group of tests have same initial conditions
- Complex sequence of steps to create preconditions
- Ensures release of system resources (memory, files, ...)

Testing frameworks generally provide mechanisms to encapsulate the logic for test preconditions and cleanup. Usually this is in the form of procedures named `setUp()` and `tearDown()`.



Simple fixture (unencapsulated)



Can be expedient to use a global variable or a persistent file as a quick-and-dirty fixture:

```
module SimpleFixture_mod
  use pFUnit
  use Reader_mod

contains

  @before
  subroutine init()
    open(10,file='tmp.dat',status='new')
    write(10) 1
    write(10) 2
    close(10)
  end subroutine init

  @after
  subroutine done()
    open(10,file='tmp.dat',status='unknown')
    close(10, status='delete')
  end subroutine done

  ...
```



- `@before` indicates procedure to run before each test in file
 - ▶ Convention is to call procedure `setUp()`
- `@after` indicates procedure to run after each test in file
 - ▶ Convention is to call procedure `tearDown()`
- Because no arguments are passed to procedures, fixture data must be in the form of global variables (module, common) or the file system (persistent file)
 - ▶ Dangerously close to violating rule: "No Side Effects!"



1 Introduction

2 Introduction to pFUnit

- pFUnit overview
- Build and Install
- Basic pFUnit Example
- Simple MPI
- Simple Fixtures (unencapsulated)
- **Testing for error handling**
- API
- Parser and Driver

3 Advanced pFUnit

4 Test-driven development



1 Introduction

2 Introduction to pFUnit

- pFUnit overview
- Build and Install
- Basic pFUnit Example
- Simple MPI
- Simple Fixtures (unencapsulated)
- Testing for error handling
- **API**
- Parser and Driver

3 Advanced pFUnit

4 Test-driven development



Role: Used to notify/detect “undesired” states during execution. *Limited* emulation of exceptions provided by other high-level languages (C++, Java, Python, etc).

Implementation:

- Manages a global, private stack of Exception objects.
- Each Exception object has a message, and a location (file+line).



```
subroutine throw(message[, location])  
  
subroutine catch(
```



Role: Used to express *intended/expected* relationships among variables.

Implementation:

- Heavily overloaded suite of procedures with consistent style for interface.
- When the intended relationship does not hold, the layer pushes a self-explanatory Exception onto the global exception stack.



```
call assertTrue(condition)
call assertFalse(condition)
call assertAny(conditions)
call assertAll(conditions)
call assertNone(conditions)
call assertNotAll(conditions)
```



```
call assertEquals(expected, found)
```



```
call assertEquals(expected, found)
```

- Overloaded for up to rank 2⁷
- Only supports default KIND

The following are only supported for scalars:

```
call assertLessThan(a, b) ! a < b
```

```
call assertLessThanOrEqualTo(a, b) ! a <= b
```

```
call assertGreaterThan(a, b) ! a > b
```

```
call assertGreaterThanOrEqualTo(a, b) ! a >= b
```

⁷Use a support request if you need more.



```
call assertEquals(expected, found)
```

API - AssertEqual (Real)



Compare two values and throw exception if different

$$|a - b| \leq \delta$$

```
call assertEqual(expected, found[, tolerance])
```

- Uses *absolute* error (as opposed to *relative* error)
- Overloaded for multiple KINDs (32 and 64 bit)
- Overloaded for multiple ranks (up through 5D)
- Optional tolerance – default is *exact* equality
- Uses L_∞ norm
- To reduce exponential number of overloads:
 - ▶ `KIND(expected) <= KIND(found)`
 - ▶ `KIND(tolerance) == KIND(found)`
 - ▶ `RANK(expected) == RANK(found)` or scalar

Example message:

```
expected: +1.000000 but found: +3.000000;  
difference: |+2.000000| > tolerance:+0.000000.
```



```
call assertLessThan(expected, found)
call assertGreaterThan(expected, found)
call assertLessThanOrEqualTo(expected, found)
call assertGreaterThanOrEqualTo(expected, found)
```

If relative tolerance is desired:

$$\frac{|a - b|}{|a|} \leq \delta$$

```
call assertRelativelyEqual(expected, found[, tolerance])
```



Compare two values and throw exception if different

$$|a - b| \leq \delta$$

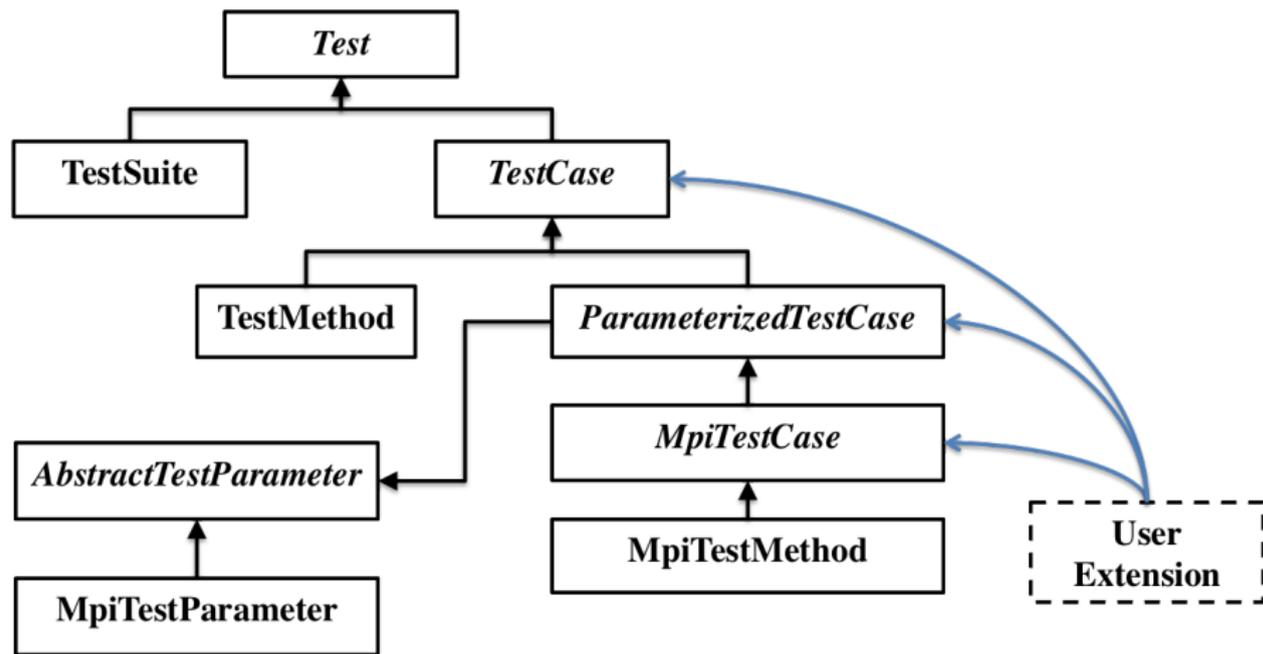
```
call assertEqual(expected, found[, tolerance])
```

- Overloaded for multiple KINDs (32 and 64 bit)
- Overloaded for multiple ranks (up through 5D)
- Optional tolerance – default is *exact* equality
- To reduce exponential number of overloads:
 - ▶ `KIND(expected) <= KIND(found)`
 - ▶ `KIND(tolerance) == KIND(found)`
 - ▶ `RANK(expected) == RANK(found)` or scalar



```
call assertIsNaN(x)      ! single/double
call assertIsFinite(x)  ! single/double
call assertExceptionRaised()
call assertExceptionRaised(message)
call assertSameShape(expectedShape, foundShape)
```

Hierarchy of Test Classes





Role: Abstract base class for all test objects.

Implementation: Framework provides various subclasses for common/generic cases. Users can define custom subclasses for specific purposes. Provided subclasses include:

- *TestCase*
- *TestMethod*
- *MpiTestCase*
- *MpiTestMethod*
- *TestSuite*



Role: Aggregates collection of tests into single entity.

Implementation: TestSuite objects are simultaneously Test objects *and* collections of tests. Run() method applies run() to each contained test.



Role: *Abstract* Test subclass that provides some services that are common to most Test subclasses.

Implementation:



Role: Simple concrete Test subclass that supports the common case where test procedure receives no arguments.

Implementation: Constructor stores a procedure pointer to vanilla Fortran subroutine with no arguments. A restricted form of test fixture is permitted by specifying setUp() and tearDown() methods that also have no arguments. (I.e. fixture is not encapsulated.)



Constructor:

```
function TestMethod(name, method[, setUp, tearDown])  
  character(len=*), intent(in) :: name  
  procedure(empty) :: method  
  procedure(empty) :: setUp  
  procedure(empty) :: tearDown
```

Methods:



Role: Allows a single test procedure to be executed multiple times with different input values.

Implementation: *ParameterizedTestCase* objects contain an *AbstractTestParameter* object that encapsulates input. Subclasses of *ParameterizedTestCase* must generally also subclass *AbstractTestParameter*.



Role: (*Abstract*) Extends *ParameterizedTestCase* with support for MPI.

Implementation: *MpiTestCase* modifies the `runBare()` launch mechanism to create an appropriately sized MPI group and corresponding subcommunicator. Processes within that group then call the user's test procedure, while any remaining processes wait at a barrier.

MPI based tests *must not* use `MPI_COMM_WORLD`, and must instead obtain MPI context from the passed test object.

The following convenient type-bound procedures are provided:

```
getProcessRank() ! returns rank within group  
getNumProcesses() ! returns size of group  
getMpiCommunicator() ! returns the bare MPI com
```



Role: Simple concrete Test subclass that supports common MPI cases that just need basic MPI context.

Implementation: Analogous to the vanilla TestMethod, except that user test procedures are now passed an object which must be queried for any MPI context that the test needs.



Constructor:

```
function MpiTestMethod(name, method, numProcesses, [, setUp  
    character(len=*), intent(in) :: name  
    procedure(empty) :: method  
    integer :: numProcesses ! requested  
    procedure(empty) :: setUp  
    procedure(empty) :: tearDown
```



Role: “Scorecard” – accumulates information about tests as they run.

Implementation: Each `run()` method for Test objects has a mandatory TestResult argument. The *Visitor* pattern is used to allow the TestResult object to manage and monitor the test as it progresses.

Note: *Visitor* is a somewhat advanced pattern and uses OO capabilities in a nontrivial manner. Users should not need to be aware of this, but developers of framework extensions likely will.

Abstract BaseTestRunner class



Role: Runs a test (usually a TestSuite).

Implementation: Run() method constructs and configures a TestResult object, then runs the passed Test object.

TestRunner class



Role: Default Runner for pFUnit.



Role: Runner subclass that executes tests within a separate process.

Implementation: Collaborates with SubsetRunner. RobustRunner restarts SubsetRunner if it detects a hang or a crash. Currently a bit unreliable.
(Irony)



1 Introduction

2 Introduction to pFUnit

- pFUnit overview
- Build and Install
- Basic pFUnit Example
- Simple MPI
- Simple Fixtures (unencapsulated)
- Testing for error handling
- API
- **Parser and Driver**

3 Advanced pFUnit

4 Test-driven development



- `@suite`

Overrides default name for generated function which constructs test suite for the input file. Default is `<base>_suite` for file with external test procedures, and `<module_name>_suite` for files that contain a module.

- `@before`

Indicates next line begins a `setUp()` procedure for subsequent test procedures.

- `@after`

Indicates next line begins a `tearDown()` procedure for subsequent test procedures.



```
@assert*(...)
```

- 1 Calls corresponding Fortran assert procedure
- 2 Inserts argument for file & line number
- 3 Inserts conditional return if exception is thrown

For example, if line 100 of file 'myTests.pf' is:

```
@assertEqual(x, y, tolerance)
```

Expands to

```
!@assertEqual(x, y, tolerance)  
call assertEqual(x, y, tolerance, &  
    & SourceLocation('myTests.pf', 100))  
if (anyExceptions()) return
```



```
@test
```

```
@test (<options >)
```

- Indicates that next line begins a new test procedure
- Appends test procedure in the file's TestSuite
- Accepts the following options:
 - ▶ `ifdef=<token>` Enables conditional compilation of test
 - ▶ `npes=[<list-of-integers>]` Specifies that test is to run in a parallel context on the given numbers of processes.
 - ▶ `esParameters={expr}` Run this test once for each value in `expr`. `Expr` can be an explicit array of `TestParameter`'s or a function that returns such an array.
 - ▶ `cases=[<list-of-integers>]` Alternative mechanism for specifying test parameters where a single integer is passed to the test constructor.



```
@testCase
```

```
@testCase (<options >)
```

- Indicates next line defines a new derived type which extends TestCase.
- All test procedures in file must accept a single argument of that extended type.
- Accepts the following options:
 - ▶ `constructor=<name>` Specifies the name of the function to construct corresponding test object. Default is a constructor with same name as derived type⁸
 - ▶ `npes=[<list-of-integers>]` Indicates that extension is a subclass of `MpiTestCase`, and provides a default set of values for NPES for all test procedures in the file. Individual tests can override.
 - ▶ `esParameters={expr}` Indicates that extension is a subclass of `ParameterizedTestCase`, and provides a default set of parameters for all tests in the file. Can be overridden by each test.
 - ▶ `cases=[<list-of-integers>]` Alternative mechanism for specifying default test parameters where a single integer is passed to the test constructor.

⁸This E2002 feature is somewhat unreliable, esp. prior to 14.0.2

Annotations: @testParameter







- 1 Introduction
- 2 Introduction to pFUnit
- 3 Advanced pFUnit**
 - OO Fortran - what you need to know
 - pFUnit test Hierarchy
 - Fixtures (encapsulated)
 - Parameterized Tests
 - Test Listeners and Runners
- 4 Test-driven development



- 1 Introduction
- 2 Introduction to pFUnit
- 3 Advanced pFUnit**
 - OO Fortran - what you need to know
 - pFUnit test Hierarchy
 - Fixtures (encapsulated)
 - Parameterized Tests
 - Test Listeners and Runners
- 4 Test-driven development



- 1 Introduction
- 2 Introduction to pFUnit
- 3 Advanced pFUnit**
 - OO Fortran - what you need to know
 - **pFUnit test Hierarchy**
 - Fixtures (encapsulated)
 - Parameterized Tests
 - Test Listeners and Runners
- 4 Test-driven development



- 1 Introduction
- 2 Introduction to pFUnit
- 3 Advanced pFUnit**
 - OO Fortran - what you need to know
 - pFUnit test Hierarchy
 - Fixtures (encapsulated)**
 - Parameterized Tests
 - Test Listeners and Runners
- 4 Test-driven development



- 1 Introduction
- 2 Introduction to pFUnit
- 3 Advanced pFUnit**
 - OO Fortran - what you need to know
 - pFUnit test Hierarchy
 - Fixtures (encapsulated)
 - Parameterized Tests**
 - Test Listeners and Runners
- 4 Test-driven development



- 1 Introduction
- 2 Introduction to pFUnit
- 3 Advanced pFUnit**
 - OO Fortran - what you need to know
 - pFUnit test Hierarchy
 - Fixtures (encapsulated)
 - Parameterized Tests
 - **Test Listeners and Runners**
- 4 Test-driven development



- 1 Introduction
- 2 Introduction to pFUnit
- 3 Advanced pFUnit
- 4 Test-driven development**



- pFUnit: <http://sourceforge.net/projects/pfunit/>
- Tutorial materials
 - ▶ <https://modelingguru.nasa.gov/docs/DOC-1982>
 - ▶ <https://modelingguru.nasa.gov/docs/DOC-1983>
 - ▶ <https://modelingguru.nasa.gov/docs/DOC-1984>
- TDD Blog
<https://modelingguru.nasa.gov/blogs/modelingwithtdd>
- *Test-Driven Development: By Example* - Kent Beck
- Müller and Padberg, "About the Return on Investment of Test-Driven Development," <http://www.ipd.uka.de/mitarbeiter/muellerm/publications/edser03.pdf>
- *Refactoring: Improving the Design of Existing Code* - Martin Fowler
- JUnit <http://junit.sourceforge.net/>



- This work has been supported by NASA's High End Computing (HEC) program and Modeling, Analysis, and Prediction Program.
- Many thanks to team members Carlos Cruz and Mike Rilee for helping with implementation, regression testing and documentation.
- Special thanks to members of the user community that have made contributions.