CubeSat Flight Software Workshop

The F Prime Unit Test Framework

Rob Bocchino

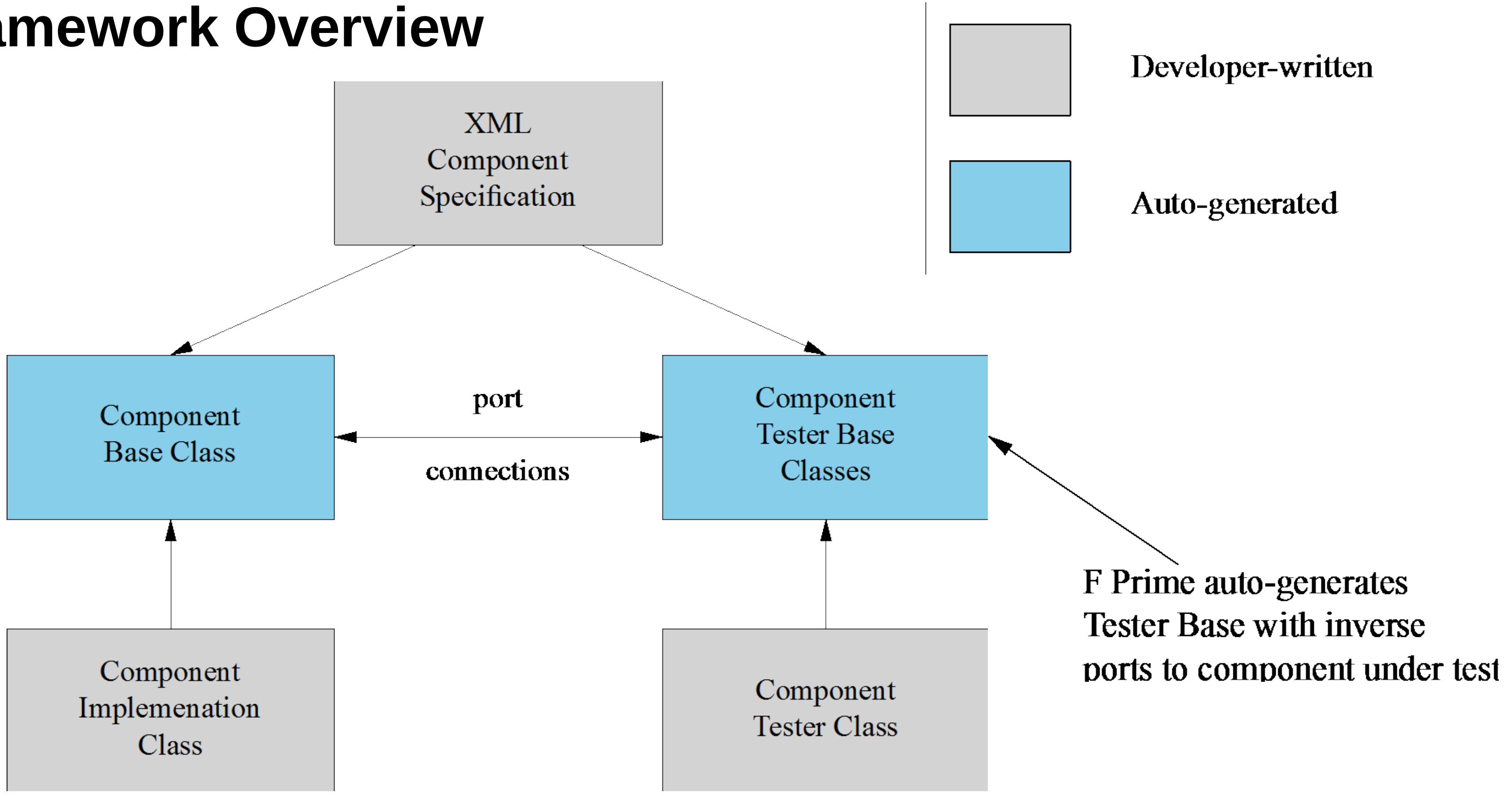
June 5, 2019

Jet Propulsion Laboratory
California Institute of Technology

Introduction

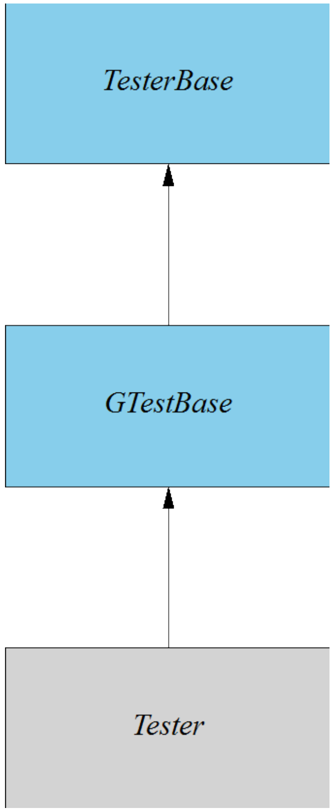
- Unit testing is important
- F Prime provides support for unit testing at the component level
- This section will explain how to
  - Auto-generate base classes for testing
  - Write unit tests
  - Send commands
  - Check the values emitted on output ports
  - Provide test values for time and parameters
  - Run unit tests
  - Check code coverage

Framework Overview


Developer-written
Auto-generated
XML Component Specification
Component Base Class
port
connections
Component Tester Base Classes
Component Implementation Class
Component Tester Class
F Prime auto-generates Tester Base with inverse ports to component under test

Test Framework Classes

- *TesterBase* (auto-generated)
  - The base class for testing a component
  - Provides a harness for unit tests
- *GTestBase* (auto-generated)
  - Derived from TesterBase
  - Includes
    - Headers for the Google Test framework
    - F Prime-specific macros
- *Tester* (developer-written from generated template)
  - Class that contains tests as members
  - Contains the component under test as a member



The TesterBase Class (Auto-Generated)

- Its interface is the “mirror image” of the component *C* under test
  - For each output port in *C*, an input port called a **from port**
  - For each input port in *C*, an output port called a **to port**
  - For each from port
    - A history *H* of data received
    - A virtual input handler that stores its arguments into *H*
- It provides utility methods for writing tests
  - Send commands to *C*
  - Send invocations to ports of *C*
  - Get and set parameters of *C*
  - Set the time

The GTestBase Class (Auto-Generated)

- Derived from *TesterBase*
- Includes headers for Google Test framework
  - *https://github.com/google/googletest*
  - Supports test assertions such as *ASSERT_EQ(3, x)*
- Adds F Prime-specific macros for checking
  - Telemetry received on telemetry from ports
  - Events received on event from ports
  - Data received on user-defined from ports
- Factored into a separate class so its use is optional

The Tester Class

- Autocoder provides a template
- You add tests as public methods
- You can also write tests in a derived class of *Tester*

Writing Unit Tests

- Generate the test classes
  - In the component directory, run *make testcomp*
  - Move the classes to the *test/ut* directory
- Add public test methods to *Tester*
- Write a *Main.cpp* file that calls the test methods:

```
#include "Tester.hpp"

TEST(TestCaseName, TestName) {
  Namespace::Tester tester;
  tester.testName();
}
...
int main(int argc, char **argv) {
  ::testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Sending Commands

```
// Send command
this->sendCOMMAND_NAME(
  cmdSeq, // Command sequence number
  arg1, // Argument 1
  arg2 // Argument 2
);
this->component.doDispatch();
// Assert command response
ASSERT_CMD_RESPONSE_SIZE(1);
ASSERT_CMD_RESPONSE(
  0, // Index in the history
  Component::OPCODE_COMMAND_NAME, // Expected command opcode
  cmdSeq, // Expected command sequence number
  Fw::COMMAND_OK // Expected command response
}
```

Checking Events

```
// Send command and check response
...
// Assert total number of events in history
ASSERT_EVENTS_SIZE(1);
// Assert number of a particular event
ASSERT_EVENTS_EventName_SIZE(1);
// Assert arguments for a particular event
ASSERT_EVENTS_EventName(
    0, // Index in history
    arg1, // Expected value of argument 1
    arg2 // Expected value of argument 2
);
```

Checking Telemetry

```
// Send command and check response
...
// Assert total number of telemetry entries in history
ASSERT_TLM_SIZE(1);
// Assert number of entries on a particular channel
ASSERT_TLM_ChannelName_SIZE(1);
// Assert value for a particular entry
ASSERT_TLM_ChannelName(
    0, // Index in history
    value // Expected value
);
```

Checking User-Defined Output Ports

```
// Send command and check response
...
// Assert total number of entries on from ports
ASSERT_FROM_PORT_HISTORY_SIZE(1);
// Assert number of entries on a particular from port
ASSERT_from_PortName_SIZE(1);
// Assert value for a particular entry
ASSERT_from_PortName(
    0, // Index in history
    arg1, // Expected value of argument 1
    arg2 // Expected value of argument 2
);
```

Setting Parameters

- In a test of component *C*, you can write

```
this->paramSet_ParamName(
    value, // Parameter value
    Fw::PARAM_VALID // Parameter status
)
```

- This call stores the arguments into member variables of *TesterBase*
- When *C* invokes its *ParamGet* role port, it will receive the arguments

Setting the Time

- In a test of component *C*, you can write

```
this->setTime(time)
```

- *time* is an *Fw::Time* object
- When *C* invokes its *TimeGet* role port, it will receive the value *time*

Building and Running Unit Tests

- To build unit tests
  - Go to the **component** directory (not the *test/ut* directory)
  - Run
    - *make ut* to build with code coverage enabled; or
    - *make ut_nocov* to build with code coverage disabled
- To run unit tests
  - Go to the **component** directory (not the *test/ut* directory)
  - Run
    - *make run_ut* to run with coverage enabled; or
    - *make run_ut_nocov* to run with coverage disabled

Analyzing Code Coverage

- Generate the analysis
  - Go to the **component** directory (not the *test/ut* directory)
  - Build the tests: *make ut*
  - Run the tests: *make run_ut*
  - Generate the analysis: *make cov*
- Review the analysis
  - Go to the ***test/ut*** directory (not the component directory)
  - Review summary output in files whose name ends in *_gcov.txt*
  - Go back to the **component** directory
    - Review coverage annotations for source files
    - File names end in *.hpp.gcov* and *.cpp.gcov*

Jet Propulsion Laboratory
California Institute of Technology

jpl.nasa.gov