



Scenario Testing with Executable Use Cases

James W Grenning

james@wingman-sw.com

wingman-sw.com

Originally Presented to Embedded Systems Conference

ESC-4020

Boston 2012

Introduction	2
Manual Testing is Unsustainable	2
Retest Time Must be Close to Zero	3
Use Cases	4
Use case example	5
Use Cases in a Typical Development Flow	6
Use Cases and User Stories	6
FitNesse Tests, a.k.a. Story Tests, or Executable Use Cases	7
Editing a FitNesse Test	9
A Passing FitNesse Test	10
A Failing FitNesse Test	11
What is being tested?	11
<i>Product with Testable Architecture</i>	12
FitNesse Test Architecture	13
Where do these tests run?	13
FitNesse/CSlim Fixture Examples	14
FitNesse Test Suites	18
Progress Tracking	19
When Something Bad Happens	20
When Fixtures are Broken or Not Ready	20
How Would You Like to Manually Test This?	21
Summary	21
Bibliography	23

Introduction

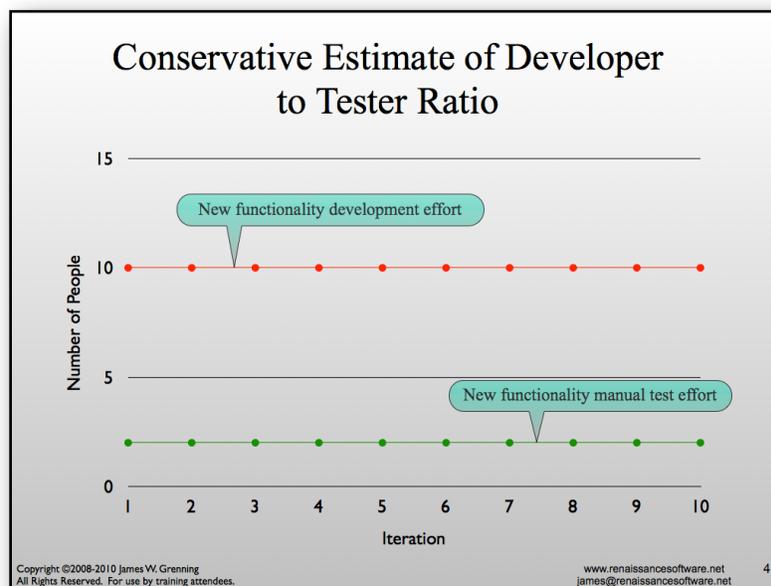
Manual testing of embedded software is unsustainable for all but the simplest products. Waiting until the end of the development cycle is a recipe for unpredictable cycles of test and fix. Testing must be done incrementally and the bulk of a product's tests must be automated. This paper presents a simple model of why manual test is unsustainable and look at an alternative, the executable use case. We compare use cases to the most commonly used technique to manage requirements in an Agile development effort, user stories. Then we explore using an open source story testing tool, FitNesse¹, to test specific product scenarios or use cases.

Manual Testing is Unsustainable

Software is fragile, as is natural for discrete systems. An apparently simple change can result in unintended consequences, affectionately known as bugs. A single wrong bit can cause disaster. Software is complex; people programming computers make mistakes. Mistakes can go unnoticed for long periods of time, but eventually show themselves as bugs.

Because of these realities of software development, we need to retest software with every change. To prevent defects, we have to test frequently to uncover mistakes before they become bugs.

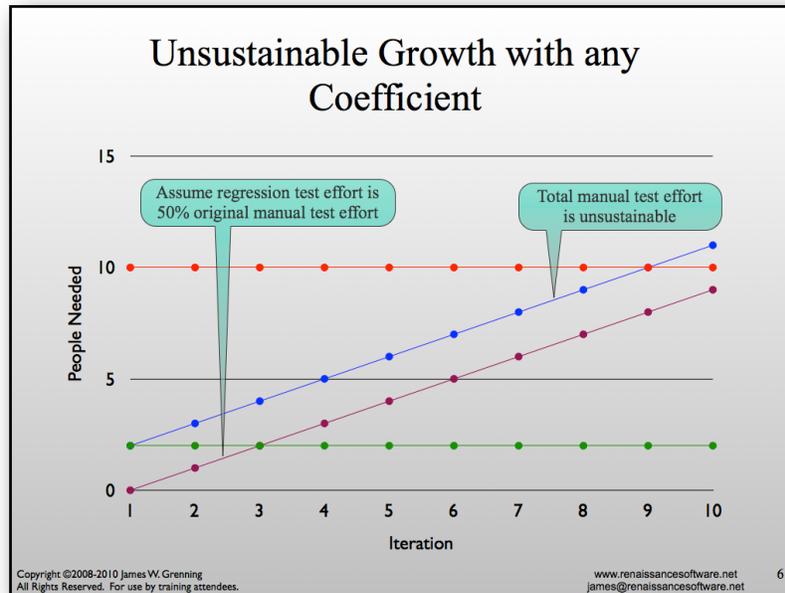
This simple model of test effort assumes that the test effort for a new feature is proportional to the development effort.



The model is wrong, but conservative. Some features have a higher test burden and some lower. Let's just agree that the model is weak and assume there is some average test load that is proportional to the new feature development effort.

¹ Pronounced like fitness.

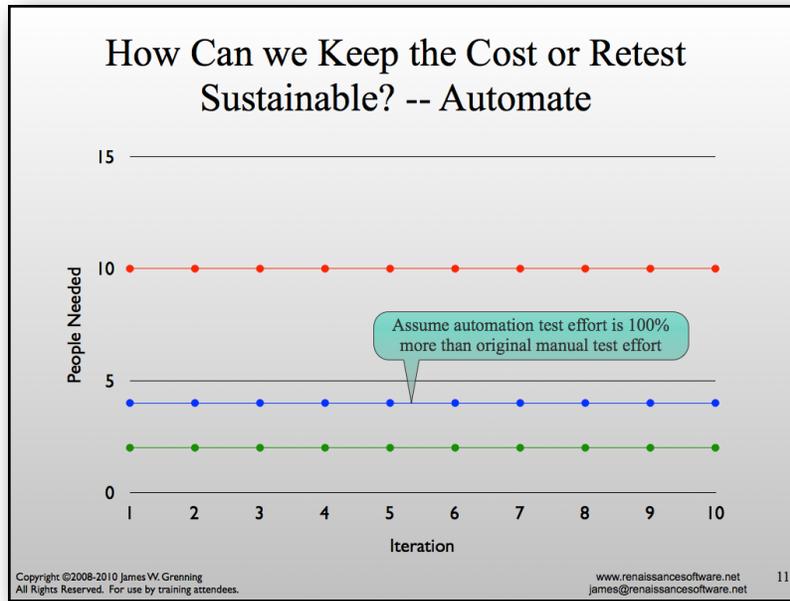
The first model does not take into account the retest time for features implemented in previous development iterations. So, let's adjust the model. Assume that the retest time is 50% of the original test time. It is less than 100% because new test procedures are not needed.



Whether these relationships are precisely correct or not, the two people that are doing test for the ten developers quickly get swamped and have to make tradeoffs. They test the new features and make educated guesses on what to test from previous iterations. Defects start to build up because a complete test is not affordable, and mistakes go undetected. A test strategy based on manual regression tests is unsustainable.

Retest Time Must be Close to Zero

A sustainable process needs to invest in automation. Retest time needs to be close to zero. What if an automated test procedure can be created and run in only twice the effort needed to create and run the manual test procedure? We get a sustainable test effort, with very low retest time.



Again, the coefficients in this model are probably wrong, but with an almost zero retest cost retest can be thorough and inexpensive at the same time.

Use Cases

A popular technique for specifying system behavior is the use case. Alistair Cockburn's book, *Writing Effective Use Cases*, is the definitive work on use cases. [COCKBURN1]

A use case specifies a usage or behavior scenario. They don't always refer to just human users, but could be used by other systems or other components in the same system. Alistair suggests providing this (and other not shown) information when documenting use cases: [COCKBURN2]

Use Case: <number> <the name should be the goal as a short active verb phrase>

CHARACTERISTIC INFORMATION

- Goal in Context: <a longer statement of the goal, if needed>
- Scope: <what system is being considered black-box under design>
- Level: <one of: Summary, Primary task, Subfunction>
- Preconditions: <what we expect is already the state of the world>
- Success End Condition: <the state of the world upon successful completion>
- Failed End Condition: <the state of the world if goal abandoned>
- Primary Actor: <a role name for the primary actor, or description>
- Trigger: <the action upon the system that starts the use case, may be time event>

MAIN SUCCESS SCENARIO

- <put here the steps of the scenario from trigger to goal delivery, and any cleanup after>
- <step #> <action description>

EXTENSIONS

- <put here there extensions, one at a time, each referring to the step of the main scenario>
- <step altered> <condition> : <action or sub.use case>
- <step altered> <condition> : <action or sub.use case>

SUB-VARIATIONS

- <put here the sub-variations that will cause eventual bifurcation in the scenario>
- <step or variation # > <list of sub-variations>
- <step or variation # > <list of sub-variations>

Use case example

In this paper I'll use a home automation system as an example. Here is an example use case for scheduling lighting controls.

Information	Description
Name	Schedule light control
Goal	Allow system users to schedule lights to turn on, off, or dim
Preconditions	System has controllable lights attached
Success End Condition	The scheduled light has been controlled
Failed End Condition	The scheduled light has not been controlled
Primary Actor	Home owner
Trigger	Scheduled time is reached
Main Success Scenario	<ol style="list-style-type: none"> 1.The home owner schedules a light to turn on at a specific time on a specific day 2.The scheduler wakes up at the right time of the right day 3.The light scheduled for this minute is turned on
Extensions/Variations	<ol style="list-style-type: none"> 1a. Homeowner can schedule the light to turn on 1b. Homeowner can schedule the light to turn off 1c. Homeowner can schedule the light to set to a dim level 1d. Homeowner can specify weekend schedule 1e. Homeowner can specify weekday schedule 2a - Scheduler does nothing when it wakes up when there are no scheduled controls. 3a - Light is turned on when on is scheduled 3b - Light is turned off when off is scheduled 3c - Light is set to a specified level when dim is scheduled

You can see that this use case starts out rather general, then gets more specific in the extensions and variations.

Use Cases in a Typical Development Flow

Use cases are usually written by technical marketing, systems engineering, or some other group that is responsible for specifying the product in the organization. The use case is handed off to development and to test, both doing their best to meet the intended need. Test uses the use case to develop test cases. They often contain much of the same information. My “ah ha!” moment... Maybe the detailed use case should never be developed. Stories name each part of the feature, while story tests can provide example data and detail to drive development. The test cases are executable, so unlike a prose description, they can play a part in keeping the code working and meeting its requirements. This approach also eliminates duplication and error prone translation of use cases to test cases.

Use Cases and User Stories

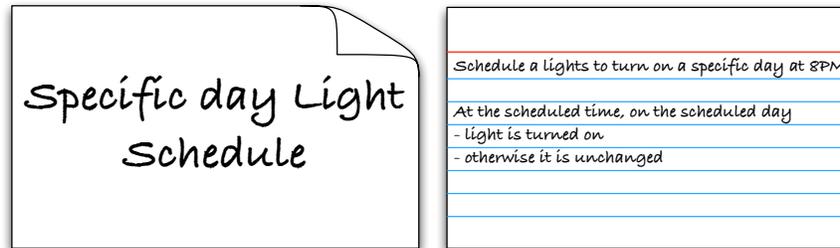
In agile development, we use user stories as the schedule-able unit of work. A user story is the name of some feature or a part of a feature. Like a use case, it is something the system should do.

In embedded development I find it useful to refer to user stories as Product Stories, as the user is not always evident for some of the stories that drive embedded development (See my other paper for ESC-404 <http://renaissancesoftware.net/papers.html>)

Think of a product story as a really light use case, so light that all is left is the name of the use case or the extension. Stories are vague. Stories are often written on notecards, at least initially. Here are some of the stories that could replace the “schedule light control” use case.



Story details come from conversations with subject matter experts, specs if you have them and from test cases. Sometimes story details are written on the back of the cards, and describe how to test the story. Tests define what it means for a story to be done.



Think of a story as a token for the feature. Stories are useful for planning an iteratively developed product. It is a reminder that we have to dig deeper. In a use case centered approach the activity of use case identification attempts to get a broad view of the development effort by naming all the use cases. Story writing is a similar activity. On a priority basis, use case identification is followed by incrementally elaborating use cases. This activity is where the details are brought in. I am proposing doing the details in an executable form, story tests in FitNesse.

FitNesse Tests, a.k.a. Story Tests, or Executable Use Cases

The story details are worked out just-in-time, keeping an iterative development effort moving towards its release goals. The just-in-time approach allows elaborating requirements concurrently with the development effort.

FitNesse is a tool for writing story tests. There are two main aspects to FitNesse. [FITNESSE]

- It is a wiki² that supports team collaboration and composition of test cases and other shared content.
- It is a test execution engine that takes specially formatted wiki pages (test pages), interprets them and interacts with the system under test (SUT)
 - Initializing the SUT
 - Feeding in event scenarios to the SUT
 - Checking responses from the SUT

Lets look at a FitNesse test case. This is the test case scenario where a light is scheduled to turn on every Monday at 7:30.

² A wiki is an easily edited website used for collaboration.

[HomeAutomationTests.LightScheduler.TestSuite.](#)

LightShouldComeOnAtTheRightTime [add child]

▼ *Set Up:* [.HomeAutomationTests.LightScheduler.SetUp \(edit\)](#)

Home Automation Start up

A light scheduled for a specific day of the week, should turn on at the scheduled time and no sooner or later.

script	Light Schedule Script							
schedule	turn on	for light	13	where day is	Monday	and time is	7:30	
check	transition to	Monday	at	7:29	then light	13	should be	unchanged
check	transition to	Monday	at	7:30	then light	13	should be	on
check	transition to	Monday	at	7:31	then light	13	should be	unchanged

▼ *Tear Down:* [.HomeAutomationTests.LightScheduler.TearDown \(edit\)](#)

Home Automation Shut down

This is a screen shot from a browser page in the FitNesse wiki. The text in the tables are test instructions and data. Free form text, the text not in a table, can add context and explanation. The executable use case goes a step further than the use case text by providing specific test data. Some think of story tests as specification by example.

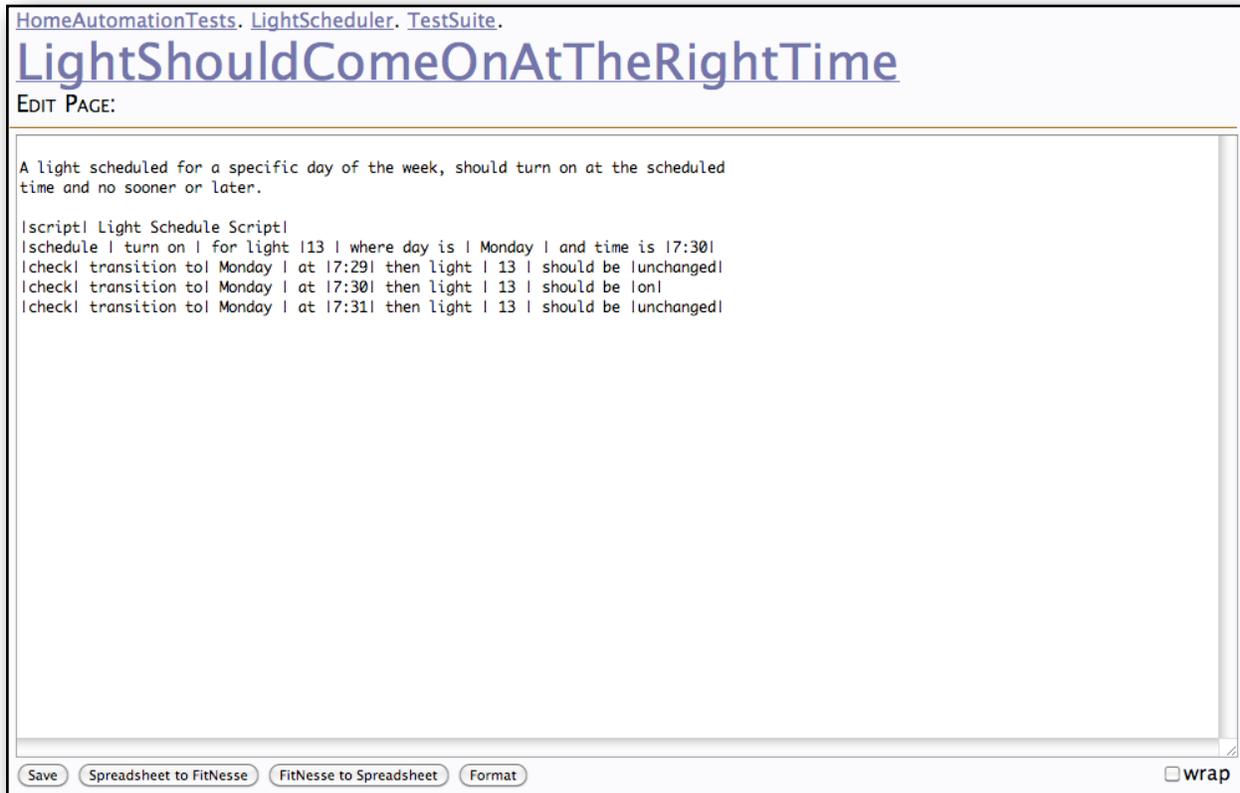
FitNesse pages are written in a very regular language, a language you customize to your application needs. This is called a Domain Specific Language. All the data pertinent to scheduling a light to be turned is specified in the test case. The test case assures the light does not turn on at 7:29, or 7:31, but does turn on at 7:30 on Monday.

Systems are usually specified using generalizations, with occasional dives into the detail showing examples. Though systems are never used in a general way, they are always used in specific ways. Tests can't be executed in a general way. They must exercise specific scenarios.

FitNesse tests specify the details in the test tables, and allow the high-level, or more general ideas to be expressed as plain text.

Editing a FitNesse Test

A page can be edited by pressing the edit button on the wiki page. The edit window for the `LightShouldComeOnAtTheRightTime` test looks like this.



The vertical bar character (“|”) is wiki markup language for a table column separator. You can see the freeform text before the table in this example. There can be multiple tables in a test page and multiple freeform comments. There are other markups but they are not essential to understanding how to define a test case in FitNesse.

FitNesse pages are organized in a hierarchy of pages. This page hierarchy above `LightShouldComeOnAtTheRightTime` is

[HomeAutomationTests](#). [LightScheduler](#). [TestSuite](#)

In the example test case you might have noticed *Set Up* and *Tear Down*. These are special named pages that get automatically included in a group of related test cases. In the example, the set up page tells the *Home Automation* system to *Start up*, while the tear down page tells the *Home Automation* system to *Shut down*. Each test is an independent run, not leaving any history for other tests to depend upon.

A Passing FitNesse Test

Fitnesse tests can do a number of pass/fail checks. The test is run by pressing the test button on the wiki page (not shown). The passing test looks like this:

[HomeAutomationTests](#). [LightScheduler](#). [TestSuite](#).

LightShouldComeOnAtTheRightTime

TEST RESULTS

Assertions: 7 right, 0 wrong, 0 ignored, 0 exceptions

▼ *Set Up:* [.HomeAutomationTests.LightScheduler.SetUp \(edit\)](#)

Home Automation
Start up

A light scheduled for a specific day of the week, should turn on at the scheduled time and no sooner or later.

script	Light Schedule Script							
schedule	turn on	for light	13	where day is	Monday	and time is	7:30	
check	transition to	Monday	at	7:29	then light	13	should be	unchanged
check	transition to	Monday	at	7:30	then light	13	should be	on
check	transition to	Monday	at	7:31	then light	13	should be	unchanged

▼ *Tear Down:* [.HomeAutomationTests.LightScheduler.TearDown \(edit\)](#)

Home Automation
Shut down

The green highlighted fields indicate where there are successful return values. The essence of this test is in the three green areas on the right. The light did not change a minute early or late, but turned on at exactly the right minute.

A Failing FitNesse Test

FitNesse tests graphically show when a test is failing.

HomeAutomationTests. LightScheduler. TestSuite.

LightShouldComeOnAtTheRightTime

TEST RESULTS Tests Executed OK

Assertions: 6 right, 1 wrong, 0 ignored, 0 exceptions

▼ *Set Up:* [.HomeAutomationTests.LightScheduler.SetUp \(edit\)](#) [Expand All](#) / [Collapse All](#)

Home Automation Start up

A light scheduled for a specific day of the week, should turn on at the scheduled time and no sooner or later.

script	Light Schedule Script							
schedule	turn on	for light	13	where day is	Monday	and time is	7:30	
check	transition to	Monday	at	7:29	then light	13	should be	unchanged
check	transition to	Monday	at	7:30	then light	13	should be	[unchanged] expected [on]
check	transition to	Monday	at	7:31	then light	13	should be	unchanged

▼ *Tear Down:* [.HomeAutomationTests.LightScheduler.TearDown \(edit\)](#) [Expand All](#) / [Collapse All](#)

Home Automation Shut down

The red highlight shows that the light was expected to be *on*, but was *unchanged*. If a development group is provided a test like this before the work to support the feature is done, it is normal to see FitNesse tests fail. When this test passes, and the code is well structured, the story is done.

You are probably wondering what is really being tested. No lights went on or off, and it's not Monday around 7:30.

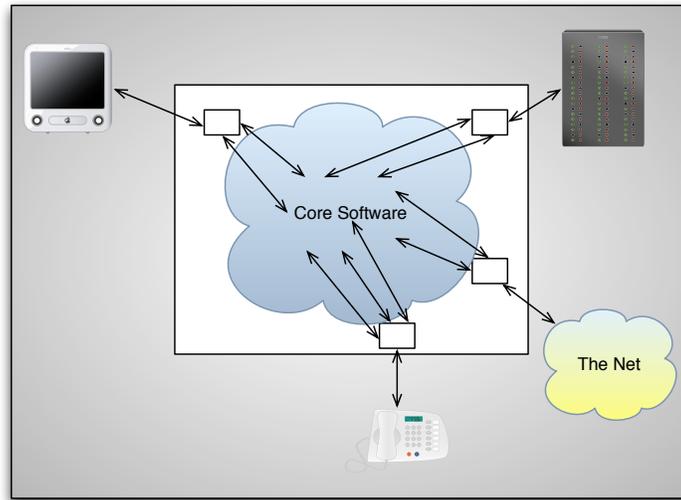
What is being tested?

In this example the core application logic (part of the production code) is being exercised. The design supports an API for scheduling light operations. The design also has separated clock and hardware dependencies so that the core application can be tested independently of the hardware and operating system environment. A system must be designed to support tests like these.

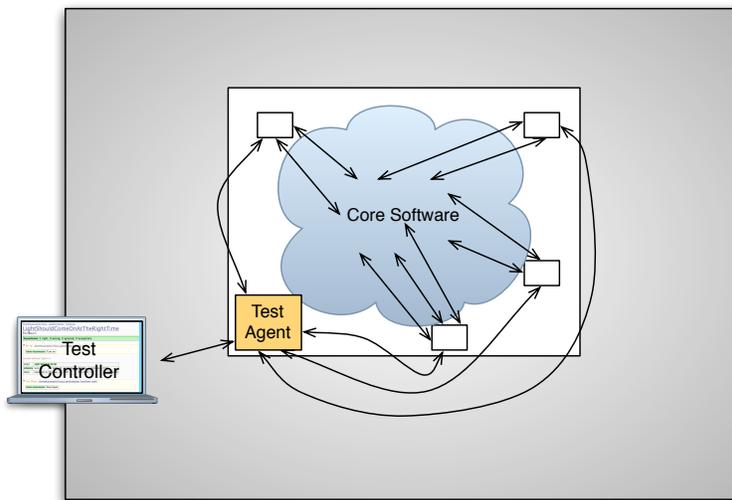
Product with Testable Architecture

When a product is not designed for test, core application logic often has direct dependencies on the hardware and operating system. For some tests that means spending your Friday night in the lab to see if the light comes on at 8PM, or continually messing with the clock. The dependencies can be hard to break.

If the system is architected for automated test, the core application communicates to its environment through interfaces as shown in the diagram below.



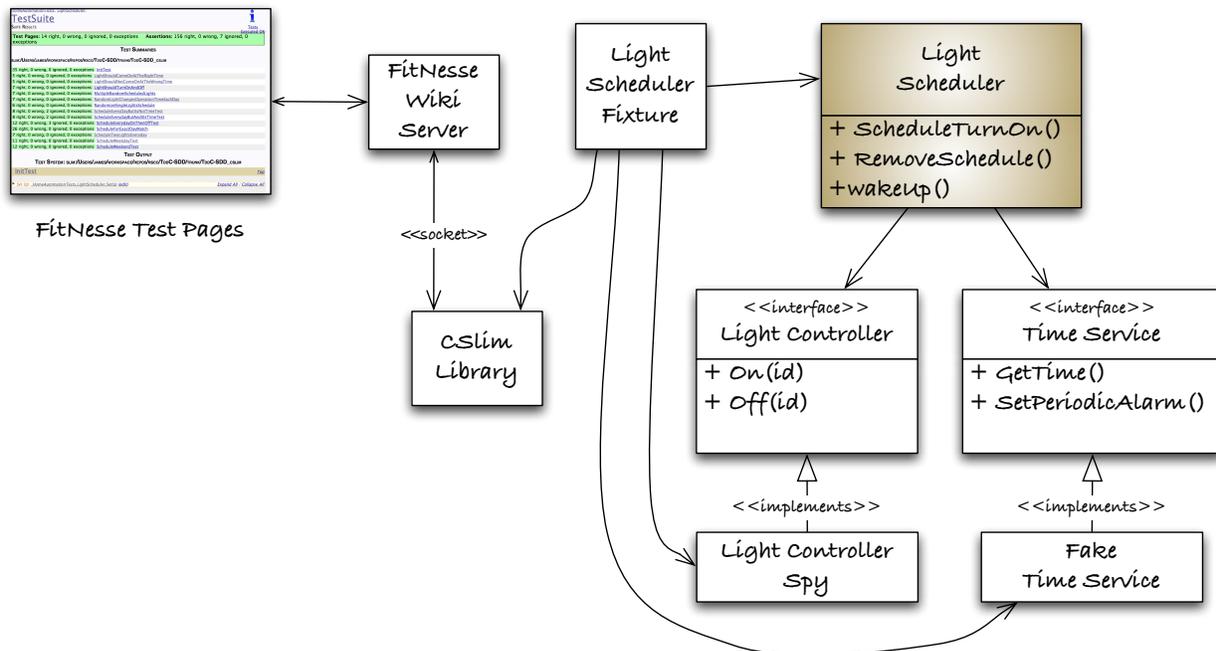
With the core application talking to the hardware and OS through interfaces, test implementations can be substituted for a test build. This diagram illustrates how a product is configured during test. Production drivers are substituted with drivers that can simulate input and capture outputs.



The test agent can generate events and capture responses at appropriate test points near the edges of the system and between subsystems. A test controller tells the test agent the events to simulate and the system states and responses to collect. Let's see how to use FitNesse to create a Test Controller and Test Agent.

FitNesse Test Architecture

Let's look at the LightScheduler component when put into a FitNesse based test harness.



The *Light Scheduler*, the application's bit of gold, is in a test fixture. A *Light Scheduler Fixture* interacts with API to program the scheduler. The fixture also simulates a one minute periodic wakeup call to the scheduler. When there is a wakeup call, the scheduler checks the time and when there is a light to be controlled it tells the *Light Controller* to turn on or off a light. Notice that the scheduler talks to the OS and hardware through interfaces, so that during tests the fixture can control the clock and capture system responses.

The Test Controller is made up of the wiki pages and FitNesse server, while the Test Agent is made up of the CSlim Library and the Fixtures. [CSLIM]

Briefly, it works like this: When someone presses the Test or Suite button on a FitNesse test page, FitNesse reads the test page and converts the test instructions into its SLIM protocol. The SLIM protocol drives remote function invocations through a socket connection, letting FitNesse talk to your application.

The CSlim library is a C implementation of the FitNesse/SLIM protocol. CSlim can be used with C or C++. There are SLIM implementations for other languages as well. Your application specific fixtures register with CSlim. CSlim waits dutifully for connections and SLIM messages, which call functions in the fixtures which stimulates the system under test and reports the SUTs reactions to the FitNesse wiki server.

Where do these tests run?

The tests can be run any number of places. An effective test strategy is to run the FitNesse based scenario tests on the host development system, rather than the target hardware. This helps to keep test fast and easy to run. It also results in a better design where hardware, OS, and component dependencies are well managed.

Because the FitNesse wiki server communicates with the CSLim library through a socket, the CSLim library and the application's fixtures could also run in the target hardware. You could also adapt the CSLim library to use some communications mechanism other than a socket if a socket is not available.

FitNesse/CSlim Fixture Examples

Let's look at what is on a FitNesse page that interacts with the Light Scheduler.

script	Light Schedule Script								
schedule	turn on	for light	13	where day is	Monday	and time is	7:30		
check	transition to	Monday	at	7:30	then light	13	should be	on	

The first row identifies that the table is a script named the Light Schedule Script. FitNesse transforms "Light Schedule Script" into the name `LightScheduleScript`. Here is a code snippet that defines the `LightScheduleScript` and two of its functions.

```
//The fixture function declarations precede this code snippet

SLIM_CREATE_FIXTURE(LightScheduleScript)
    SLIM_FUNCTION(schedule_ForLight_WhereDayIs_AndTimeIs)
    SLIM_FUNCTION(transitionTo_At_ThenLight_ShouldBe)
SLIM_END
```

The macro `SLIM_CREATE_FIXTURE` defines the fixture and the macro `SLIM_FUNCTION` defines each of its functions. The fixture and functions are wired into a lookup table that the CSLim library uses to interact with the fixture. The functions `schedule_ForLight_WhereDayIs_AndTimeIs` and `transitionTo_At_ThenLight_ShouldBe` are declared earlier in the same file.

Look at the second row of the FitNesse test table. The first, third, fifth, and seventh fields in the table are mangled into the function name `schedule_ForLight_WhereDayIs_AndTimeIs`. The even numbered fields are positional CSLim parameters passed to the function in CSLIM data lists.

`schedule_ForLight_WhereDayIs_AndTimeIs` is a function that return the string "true" if all goes to plan, or an error message when something is wrong.

The third row is much the same, except that the row starts with the keyword `check` which implies that the last field of the table is the expected result of that function invocation. The second, fourth, sixth, and eighth fields are mangled into `transitionTo_At_ThenLight_ShouldBe`.

To successfully invoke `SLIM_CREATE_FIXTURE(LightScheduleScript)`, these two functions are required.

```
typedef struct LightScheduleScript
{
    char result[80];
} LightScheduleScript;

//...

void* LightScheduleScript_Create(StatementExecutor* errorHandler, SlimList* args)
{
    LightScheduleScript* self =
        (LightScheduleScript*)calloc(1, sizeof(LightScheduleScript));
    return self;
}

void LightScheduleScript_Destroy(void* void_self)
{
    free(void_self);
}
```

The functions are responsible for initializing and cleaning up the `LightScheduleScript` data structure used by the fixture's functions. They must be named as `<FixtureName>_Create` and `<FixtureName>_Destroy`. In this case the create and destroy functions don't have much to do, but that is not always the case. Sometimes more data is needed by the fixture, and it must be initialized and cleaned up.

Let's look now at the code behind `schedule_ForLight_WhereDayIs_AndTimeIs`. This function follows a common fixture pattern: collect and validate parameters, operate on the production code, return a result string.

```
static char* schedule_ForLight_WhereDayIs_AndTimeIs(void* void_self, SlimList* args)
{
    LightScheduleScript* self = (LightScheduleScript*)void_self;
    int id, operation, day, minute;

    if (! checkArgCount(self, args, 4))
        return self->result;

    id = getId(self, args, 1);
    if (id < 0)
        return self->result;

    day = getDay(self, args, 2);
    if (day == NOT_A_DAY)
        return self->result;

    minute = getMinute(self, args, 3);
    if (minute < 0)
        return self->result;

    operation = getOperation(self, args, 0);
    if (operation == LIGHT_ON)
        LightScheduler_ScheduleTurnOn(id, day, minute);
    else if (operation == LIGHT_OFF)
        LightScheduler_ScheduleTurnOff(id, day, minute);
    else
        return self->result;

    return "true";
}
```

This fixture is simulating the user interacting with the system to schedule a light control. After getting checking the parameters the fixture calls the production code API (`LightScheduler_ScheduleTurnOn` or `LightScheduler_ScheduleTurnOff`) based on the specified operation. If control makes it to the bottom of this function it returns "true".

Here is `setResult` along with an example parameter getter/checker.

```
static void setResult(LightScheduleScript* self, const char * result)
{
    strncpy(self->result, result, sizeof(self->result));
}

static int getDay(LightScheduleScript* self, SlimList* args, int dayIndex)
{
    int day = convertDayStringToInt(SlimList_GetStringAt(args, dayIndex));
    if (day == NOT_A_DAY)
        setResult(self, SLIM_EXCEPTION("Having a bad day"));

    return day;
}
```

When there is a problem with the parameter, see how `getDay` forms an error message with `SLIM_EXCEPTION` and puts it into the result buffer.

The next function (`transitionTo_At_ThenLight_ShouldBe`) is responsible for advancing the clock, simulating the wakeup call to the Light Scheduler and collecting the system response. This fixture follows a similar pattern: collect and validate parameters, operate on the production code, get any system responses, return a result string.

```
static char* transitionTo_At_ThenLight_ShouldBe(void* void_self, SlimList* args)
{
    LightScheduleScript* self = (LightScheduleScript*)void_self;
    int id;
    int lightState;
    const char* result;

    if (!checkArgCount(self, args, 3))
        return self->result;

    id = getId(self, args, 2);
    if (id < 0)
        return self->result;

    if (setTimeResetLightsTransitionClock(self, args) == 0)
        return self->result;

    lightState = FakeLightController_getLightState(id);
    result = convertIntToOnOff(lightState);

    setResult(self, result);
    return self->result;
}
```

The checking of the returned result is done by FitNesse and not the fixture.

To complete the picture, here is the helper function that advances the clock and wakes the scheduler.

```
static int setTimeResetLightsTransitionClock(
    LightScheduleScript* self, SlimList* args) {
    if (FakeTimeService_SetTime(self, args, 0, 1) == 0)
        return 0;
    FakeLightController_resetAll();
    LightScheduler_WakeUp();
    return 1;
}
```

FitNesse Test Suites

Earlier I mentioned that FitNesse supports hierarchies of tests. Running the [HomeAutomationTests.LightScheduler.TestSuite](#) runs all the tests in the suite. There can be suites of tests and suites of suites.

[HomeAutomationTests.LightScheduler.](#)

TestSuite

SUITE RESULTS 
Tests Executed OK

Test Pages: 14 right, 0 wrong, 0 ignored, 0 exceptions **Assertions:** 156 right, 0 wrong, 7 ignored, 0 exceptions

TEST SUMMARIES

SLIM:/USERS/JAMES/WORKSPACE/REPOS/RSCC/TDDC-SDD/TRUNK/TDDC-SDD_CSLIM

35 right, 0 wrong, 0 ignored, 0 exceptions	InitTest
5 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldComeOnAtTheRightTime
5 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldNotComeOnAtTheWrongTime
7 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldTurnOnAndOff
7 right, 0 wrong, 0 ignored, 0 exceptions	MultipleRandomScheduledLights
7 right, 0 wrong, 0 ignored, 0 exceptions	RandomLightChangesOperationTimeEachDay
6 right, 0 wrong, 0 ignored, 0 exceptions	RandomizeSingleLightsSchedule
8 right, 0 wrong, 2 ignored, 0 exceptions	ScheduleEveryDayButItsNotTimeTest
8 right, 0 wrong, 2 ignored, 0 exceptions	ScheduleEverydayButAndItsTimeTest
12 right, 0 wrong, 3 ignored, 0 exceptions	ScheduleEverydayOnThenOffTest
26 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleForExactDayMatch
7 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleTwoLightsEveryday
11 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleWeekdayTest
12 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleWeekendTest

There is virtually no cost to re-running these tests. A button press and they show green in about a half a second.

Progress Tracking

Completed tests can provide a sense of progress. This shows there is one more test to go in this suite.

[HomeAutomationTests.LightScheduler.](#)

TestSuite

SUITE RESULTS  Tests Executed OK

Test Pages: 13 right, 1 wrong, 0 ignored, 0 exceptions **Assertions:** 154 right, 2 wrong, 7 ignored, 0 exceptions

TEST SUMMARIES

SLIM:/USERS/JAMES/WORKSPACE/REPOS/RSCC/TDDC-SDD/TRUNK/TDDC-SDD_CSLIM

35 right, 0 wrong, 0 ignored, 0 exceptions	InitTest
5 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldComeOnAtTheRightTime
5 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldNotComeOnAtTheWrongTime
7 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldTurnOnAndOff
7 right, 0 wrong, 0 ignored, 0 exceptions	MultipleRandomScheduledLights
7 right, 0 wrong, 0 ignored, 0 exceptions	RandomLightChangesOperationTimeEachDay
6 right, 0 wrong, 0 ignored, 0 exceptions	RandomizeSingleLightsSchedule
8 right, 0 wrong, 2 ignored, 0 exceptions	ScheduleEveryDayButItsNotTimeTest
8 right, 0 wrong, 2 ignored, 0 exceptions	ScheduleEverydayButAndItsTimeTest
12 right, 0 wrong, 3 ignored, 0 exceptions	ScheduleEverydayOnThenOffTest
26 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleForExactDayMatch
7 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleTwoLightsEveryday
11 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleWeekdayTest
10 right, 2 wrong, 0 ignored, 0 exceptions	ScheduleWeekendTest

When Something Bad Happens

With a good set of tests, a mistake with unintended bad side effects is evident. One changed line of code caused these failures. Timely mistake awareness leads to defect prevention.

HomeAutomationTests. LightScheduler.

TestSuite

SUITE RESULTS Tests Executed OK

Test Pages: 3 right, 11 wrong, 0 ignored, 0 exceptions **Assertions:** 129 right, 27 wrong, 7 ignored, 0 exceptions

TEST SUMMARIES

SLIM:/USERS/JAMES/WORKSPACE/REPOS/RSCC/TDDC-SDD/TRUNK/TDDC-SDD_CSLIM

35 right, 0 wrong, 0 ignored, 0 exceptions	InitTest
4 right, 1 wrong, 0 ignored, 0 exceptions	LightShouldComeOnAtTheRightTime
5 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldNotComeOnAtTheWrongTime
5 right, 2 wrong, 0 ignored, 0 exceptions	LightShouldTurnOnAndOff
5 right, 2 wrong, 0 ignored, 0 exceptions	MultipleRandomScheduledLights
5 right, 2 wrong, 0 ignored, 0 exceptions	RandomLightChangesOperationTimeEachDay
5 right, 1 wrong, 0 ignored, 0 exceptions	RandomizeSingleLightsSchedule
8 right, 0 wrong, 2 ignored, 0 exceptions	ScheduleEveryDayButItsNotTimeTest
7 right, 1 wrong, 2 ignored, 0 exceptions	ScheduleEverydayButAndItsTimeTest
10 right, 2 wrong, 3 ignored, 0 exceptions	ScheduleEverydayOnThenOffTest
19 right, 7 wrong, 0 ignored, 0 exceptions	ScheduleForExactDayMatch
5 right, 2 wrong, 0 ignored, 0 exceptions	ScheduleTwoLightsEveryday
6 right, 5 wrong, 0 ignored, 0 exceptions	ScheduleWeekdayTest
10 right, 2 wrong, 0 ignored, 0 exceptions	ScheduleWeekendTest

TEST OUTPUT

When Fixtures are Broken or Not Ready

Fixture errors show up in tables as yellow highlight.

HomeAutomationTests. LightScheduler.

TestSuite

SUITE RESULTS Tests Executed OK

Test Pages: 11 right, 0 wrong, 0 ignored, 3 exceptions **Assertions:** 156 right, 0 wrong, 7 ignored, 6 exceptions

TEST SUMMARIES

SLIM:/USERS/JAMES/WORKSPACE/REPOS/RSCC/TDDC-SDD/TRUNK/TDDC-SDD_CSLIM

35 right, 0 wrong, 0 ignored, 0 exceptions	InitTest
5 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldComeOnAtTheRightTime
5 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldNotComeOnAtTheWrongTime
7 right, 0 wrong, 0 ignored, 0 exceptions	LightShouldTurnOnAndOff
7 right, 0 wrong, 0 ignored, 0 exceptions	MultipleRandomScheduledLights
7 right, 0 wrong, 0 ignored, 0 exceptions	RandomLightChangesOperationTimeEachDay
6 right, 0 wrong, 0 ignored, 0 exceptions	RandomizeSingleLightsSchedule
8 right, 0 wrong, 2 ignored, 2 exceptions	ScheduleEveryDayButItsNotTimeTest
8 right, 0 wrong, 2 ignored, 2 exceptions	ScheduleEverydayButAndItsTimeTest
12 right, 0 wrong, 3 ignored, 2 exceptions	ScheduleEverydayOnThenOffTest
26 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleForExactDayMatch
7 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleTwoLightsEveryday
11 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleWeekdayTest
12 right, 0 wrong, 0 ignored, 0 exceptions	ScheduleWeekendTest

How Would You Like to Manually Test This?

Maybe you think the only real test is one done in the target? Plan to work late. I can test this application logic in 0.049 seconds. A manual test, with clock tweaking will probably take 3 minutes per test.

[HomeAutomationTests.LightScheduler.TestSuite.](#)
[ScheduleWeekdayTest](#)

TEST RESULTS

Assertions: 11 right, 0 wrong, 0 ignored, 0 exceptions

▶ [Set Up: .HomeAutomationTests.LightScheduler.SetUp \(edit\)](#)

script	Light Schedule Script							
schedule	turn off	for light	1	where day is	Weekday	and time is	10:30	
check	transition to	Monday	at	10:30	then light	1	should be	off
check	transition to	Tuesday	at	10:30	then light	1	should be	off
check	transition to	Wednesday	at	10:30	then light	1	should be	off
check	transition to	Thursday	at	10:30	then light	1	should be	off
check	transition to	Friday	at	10:30	then light	1	should be	off
check	transition to	Saturday	at	10:30	then light	1	should be	unchanged
check	transition to	Sunday	at	10:30	then light	1	should be	unchanged

▶ [Tear Down: .HomeAutomationTests.LightScheduler.TearDown \(edit\)](#)

These tests can also be run on the target

Summary

Teams that use use cases may be able to shift toward executable use cases (a.k.a story tests) and not incur additional effort. The effort to create story tests could be paid for by not creating detailed use cases. If there is additional effort, keep in mind that the effort to automate has a positive return on investment. While the effort to test a release manually has a very short shelf-life, the executable use case serves as specification, provides example test data and can run to show conformance.

Story tests can help to get more value from an activity your organization is probably already doing, writing use cases, requirements, or manual test procedures. Some of that effort could be put into story tests. I encourage you to wean your organization off many or your manual tests and replace them with story tests as they provide a high return on investment. Manual test is unsustainable. It means your test effort, even though high, will have disappointing results. One of the powers of story tests is that once a few tests and fixtures are in place, programmers will not have to write all the tests. The testers, test engineers, or subject matter experts that write manual test procedures could write and maintain story tests. Story tests are often easier to add to a legacy code base, as it may be easier to break dependencies around the edges of your application.

In this paper, we looked at story tests running on the host development system. Because of the fitness architecture the tests can also be run on the target platform.

We've seen executable use cases implemented in FitNesse. These tests were run only on my host development system, but can also run in a target platform with some additional work. Keep in mind that these are not the only tests that a development team needs. There are lower level and high level tests. At the low level there are unit tests. A very effective way to develop on a minute to minute basis is to use Test Driven Development (See my book Test Driven Development for Embedded C). Unit testing might be the single most important tests to write, but a big challenge in a legacy code base.

These tests won't detect load and interaction problems in the system. Load tests are also needed. The act of adding unit and story tests provides the flexibility points in the production code that allow automated load tests to be written.

Both unit tests, story tests, and load tests are necessary, but not sufficient. Systems must also be tested as they will be used. So integrated systems must be tested in their target operating environment. Testing in the target is necessary, but also insufficient. When testing a fully integrated system we cannot get the code through all its branches to make sure those really odd error situations are handled properly. Though at the unit and the story level, we have a very good chance to exercise all the boundaries and edges of the system. An effective test strategy needs automated unit tests, story tests, and load tests. Some amount of manual and automated system tests will still be needed. The effort put in early and often into unit and story tests should result in many fewer problems derailing product delivery.

Bibliography

[COCKBURN1] Cockburn, Alistair, Writing Effective Use Cases, Addison Wesley, 2001

[COCKBURN2] <http://alistair.cockburn.us/Basic+use+case+template>

[FITNESSE] www.fitnessse.org

[CSLIM] <http://github.com/dougbradbury/cslim>

Contact me if you would like this example.

Find my other papers, articles and ESC submissions here:

<http://renaissancesoftware.net/papers.html>

<http://renaissancesoftware.net/blog>