The Pragmatic Bookshelf

PragPub

The First Iteration

IN THIS ISSUE

* Advanced Arduino Hacking
* Create your Own Arduino IDE
* Testing Arduino Code
* Test Abstraction
* When Did That Happen?

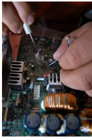
The Arduino Issue

In which we bring serious software development tools to the popular single-board platform.

Issue #22
April 2011

Contents

FEATURES



Advanced Arduino Hacking 4

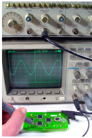
by Maik Schmidt

You want to get into this popular open-source electronics prototyping platform, but you don't want to have to work with development tools designed for artists and hobbyists. Maik shows you how to develop software for Arduino in a professional way.

Create your Own Arduino IDE 22

by Maik Schmidt

If you're going to do serious Arduino development, you may want to work in an environment more like your day job. Here Maik shows how to set up your own IDE.



Testing Arduino Code 24

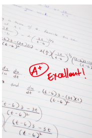
by Ian Dees

Ian brings the testing power of the Ruby-based Cucumber testing library to the Arduino.

Test Abstraction 33

by Jeff Langr, Tim Ottinger

Use the techniques in this article to sniff out problems and improve tests by increasing their level of abstraction.



When Did That Happen? 42

by Dan Wohlbruck

Claude Shannon was born in this month in 1916. Two decades later, he made history.

Up Front 1
by Michael Swaine

Just because it's a hobby that doesn't mean you don't need power tools.

Choice Bits 2
A few selected sips from the Twitter stream.

The Quiz 45
by Michael Swaine

Just your basic Arduino Sudoku.

Calendar 47
After a slow-ish winter, things are really heating up this spring.

Shady Illuminations 58
by John Shade

Some elements of Sun Microsystems were never going to survive the move to Oracle. John runs down the Doomed from the Get-Go list.

Except where otherwise indicated, entire contents copyright © 2011 The Pragmatic Programmers.

Feel free to distribute this magazine (in whole, and for free) to anyone you want. However, you may not sell this magazine or its content, nor extract and use more than a paragraph of content in some other publication without our permission.

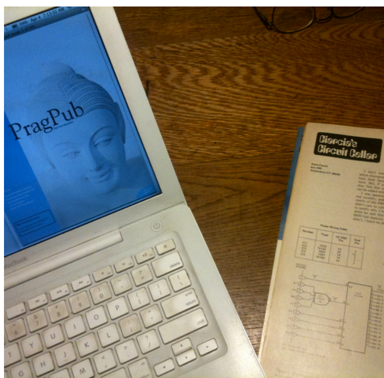
Published monthly in PDF, mobi, and epub formats by The Pragmatic Programmers, LLC, Dallas, TX, and Raleigh, NC. E-Mail support@pragprog.com, phone +1-800-699-7764. The editor is Michael Swaine (michael@pragprog.com). Visit us at http://pragprog.com for the lowdown on our books, screencasts, training, forums, and more.

ISSN: 1948-3562

Up Front

Bringing Serious Software Development to the Arduino

by Michael Swaine



Welcome to our first-ever Arduino issue!

Arduino, as the whole geek-speaking world knows by now, is an extremely accessible open-source single-board microcontroller designed for pursuing electronics projects. Also as the whole geek-speaking world knows by now, Arduino is very popular. Over 100,000 boards have been sold so far, and the fascination shows no signs of abating.

We're a hard-core software developers' magazine, so that's the approach we're taking to Arduino in this issue. We figure you want to get into this popular open-source electronics prototyping platform, but you don't want to have to work with development tools designed for artists and hobbyists. So Arduino master Maik Schmidt shows you how to develop software for Arduino in a professional way. Ian Dees gets into the act, too, showing how to bring serious software testing to the Arduino. Just because it's a hobby that doesn't mean you don't need power tools.

Of course we also have lots of other good stuff in this packed issue. Jeff Langr and Tim Ottinger offer up a thoughtful article on Test Abstraction, Dan Wohlbruck offers another in his series on computer history, John Shade has some thoughts on Sun, Oracle, and the Doomed List, and there is a quiz (Arduino-themed, of course).

Arduino Resources

We recommend that you include in your Arduino toolkit Maik's book, Arduino: A Quick-Start Guide [U1]. But there are other books to check out, including these two new ones from O'Reilly: Arduino Cookbook [U2] by Michael Margolis and Make: Arduino Bots and Gadgets [U3] by Tero Karvinen and Kimmo Karvinen.

Also, here are three sites Maik tells us are must-visits for the Arduino hacker: makershed.com [U4], adafruit.com [U5], and hackaday.com [U6].

External resources referenced in this article:

[U1] http://pragprog.com/refer/pragpub22/titles/msard/arduino

[U2] http://oreilly.com/catalog/9780596802486/

[U3] http://oreilly.com/catalog/0636920010371/

[U4] http://makershed.com

[U5] http://adafruit.com

[U6] http://hackaday.com/

Test Abstraction

Eight Techniques to Improve Your Tests

by Jeff Langr, Tim Ottinger

Use the techniques outlined in this article to sniff out problems and improve tests by increasing their level of abstraction.

In this article we illustrate several useful techniques you can use to improve your tests by increasing their level of abstraction.

Expressive Tests

When doing TDD, the tests you create are the entry point into your system. Tests are where the coding starts. They are also how you drive changes into the system, whether those changes represent new features or defects that you must fix. When you've taken the care to craft your tests to act as "specifications by example," you can learn about system behaviors and class capabilities from reading the tests.

For those who haven't ingrained the discipline of TDD, the natural inclination is to find reasons not to write tests—to those developers, TDD is just more work. We hear many excuses:

- "This code was in a larger method that was already tested. All I did was move it to a new class, and there's no way I could have broken it."
- "The whole feature is covered by an acceptance test."
- "The method is so simple I can look at it and know it's not broken."
- "I don't have time."
- "Our coverage is good enough."

"... So why do I need to write additional tests?"

Why? First and foremost, because other developers will need to understand the code at some point down the road. They may need to change the behavior of the class, or they may want to reap the benefits of reuse and consume the class from other client code. Without tests, they'll have to spend extra time digging through the code to fully understand its behavior.

Describing your code with readable tests is an act of courtesy. It is also the mark of a professional who has moved out of the novice TDD phase—someone who is striving to craft *better* tests instead of seeking to avoid writing them.

Test Abstraction

In our *PragPub* article on abstraction [U1], we lightly touched on the topic of test abstraction—the idea that a test needs to clearly specify its intent. You may recall that we used Uncle Bob's definition of abstraction: "Amplification of the essential, elimination of the irrelevant." Let's talk about some concepts and techniques for making your tests suitably abstract.

Bill Wake's Arrange-Act-Assert (AAA) pattern [U2] for test organization tells you to visually organize your tests—by grouping lines of code—around their core three steps: setup ("arrange"), execution ("act"), and verification ("assert"). For a test reader to easily understand the intent of a test, these three test steps must be distinct from each other.

Kent Beck's rules for simple design (we have an Agile in a Flash card [U3] for that too) are the next things to consider. The second and third steps of the simple design rules are:

1. …
2. No code is duplicated
3. Code clearly expresses intent

The rules are in priority order. Getting rid of duplication trumps code expressiveness. However, this ordering has been debated often, with some developers suggesting a reversal, and other developers suggesting that these two rules are roughly on equal footing.

With respect to tests, if you strive to eliminate duplication without considering expressiveness, you're going to bury important concepts in setup methods. Sure, it's usually easy to navigate to a setup method and then return to the test method, but that navigation represents extra "travel" on the part of the test reader. Eventually, unnecessary travel adds up to significant wasted time and wears a test reader down. It is especially unwelcome if the reader is only visiting the test because his or her most recent code change caused it to fail.

Following is a test that exhibits some test smells, culled from a well-known open source effort. Let's see what we can do to improve it.

The test `testMessageKey` isn't terribly long but it's not quickly digestible either. If it failed, how long would it take for you to understand why?

```
public void testMessageKey() {
  HashMap<String, Object> params = new HashMap<String, Object>();
  params.put("foo", "200");
                                                                       //
  HashMap<String, Object> extraContext = new HashMap<String, Object>();
  extraContext.put(ActionContext.PARAMETERS, params);
                                                                       //
  try {
     ActionProxy proxy = actionProxyFactory.createActionProxy("",
        MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
     ValueStack stack = ActionContext.getContext().getValueStack();
     ActionContext.setContext(new ActionContext(stack.getContext()));
     ActionContext.getContext().setLocale(Locale.US);
     proxy.execute();
     assertTrue(((ValidationAware) proxy.getAction()).hasFieldErrors());
                                                                       //
     Map<String, List<String>> errors =
        ((ValidationAware) proxy.getAction()).getFieldErrors();
     List<String> fooErrors = errors.get("foo");
     assertEquals(1, fooErrors.size());
                                                                       //
     String errorMessage = fooErrors.get(0);
     assertNotNull(errorMessage);
     assertEquals("Foo Range Message", errorMessage);
  } catch (Exception e) {
     e.printStackTrace();
     fail();
  }
}
```

So, what can we do to improve this?

Unnecessary Test Code

If you're running your test suite in an environment that suppresses stack traces on test failure, it may seem advantageous to have included the printStackTrace statement. But from the standpoint of cohesion, it's not necessary and represents irrelevant clutter in your tests.

A single JUnit test method should capture a single case, start to finish. Design the bulk of your tests as "happy path" cases that demonstrate useful functionality, blissfully ignoring any possible errors. When the system under test (SUT) throws an exception, JUnit catches it and reports a test failure. So you can remove extracurricular constructs, such as the try/catch block in testMessageKey.

Don't waste too much time designing tests that provide volumes of information when they fail. Once you've gotten them to pass initially, they will almost always pass. If tests fail in a non-obvious way, it's a simple matter to bolster them with helpful failure messages and then re-run them. Instead, make the message as brief and clear as possible.

Another piece of clutter is the assertNotNull statement near the end of the test. Such assertions are absolutely unnecessary when followed by a subsequent test that uses the same reference. In testMessageKey, the assertNotNull statement is immediately followed by assertEquals("Foo Range Message", errorMessage). If the reference is null, the subsequent assertion will make it perfectly clear. Run the test while it is failing and check the error message for yourself.

Here's the new code, with the try/catch block and unnecessary assertNotNull statement removed:

```
public void testMessageKey() {
  HashMap<String, Object> params = new HashMap<String, Object>();
  params.put("foo", "200");
                                                                     //
  HashMap<String, Object> extraContext = new HashMap<String, Object>();
  extraContext.put(ActionContext.PARAMETERS, params);
                                                                     //
  ActionProxy proxy = actionProxyFactory.createActionProxy("",
     MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
  ValueStack stack = ActionContext.getContext().getValueStack();
  ActionContext.setContext(new ActionContext(stack.getContext()));
  ActionContext.getContext().setLocale(Locale.US);
  proxy.execute();
  assertTrue(((ValidationAware) proxy.getAction()).hasFieldErrors());
                                                                     //
  Map<String, List<String>> errors =
     ((ValidationAware) proxy.getAction()).getFieldErrors();
  List<String> fooErrors = errors.get("foo");
  assertEquals(1, fooErrors.size());
                                                                     //
  String errorMessage = fooErrors.get(0);
  assertEquals("Foo Range Message", errorMessage);
}
```

Missing Abstractions

We want to quickly understand the core concepts of each test we read. Anywhere we find two or more lines used to express a single concept, there might be bloat that we can eliminate.

In testMessageKey, we see a common construct: verifying that a collection has only a single element, and then verifying the value of that single element:

```
List<String> fooErrors = errors.get("foo");
assertEquals(1, fooErrors.size());
                                                                     //
String errorMessage = fooErrors.get(0);
assertEquals("Foo Range Message", errorMessage);
```

A bit of refactoring and name-brainstorming can lead to a single-line abstraction (which might read even better in Hamcrest form):

```
assertSoleElementEquals("Foo Range Message", errors.get("foo"));
```

Our test is starting to trim down. We're also starting to build a library of reusable functions which will help us reduce the amount of code application-wide.

```
public void testMessageKey() {
  HashMap<String, Object> params = new HashMap<String, Object>();
  params.put("foo", "200");
                                                                      //
  HashMap<String, Object> extraContext = new HashMap<String, Object>();
  extraContext.put(ActionContext.PARAMETERS, params);
                                                                      //
  ActionProxy proxy = actionProxyFactory.createActionProxy("",
     MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
  ValueStack stack = ActionContext.getContext().getValueStack();
  ActionContext.setContext(new ActionContext(stack.getContext()));
  ActionContext.getContext().setLocale(Locale.US);
  proxy.execute();
  assertTrue(((ValidationAware) proxy.getAction()).hasFieldErrors());
                                                                      //
  Map<String, List<String>> errors =
     ((ValidationAware) proxy.getAction()).getFieldErrors();
  assertSoleElementEquals("Foo Range Message", errors.get("foo"));
}
```

Irrelevant Information

In a well-abstracted test, every fact in the test is relevant to understanding the test. Take care not to include arbitrary values in a way that suggests relevance. In the statement:

```
params.put("foo", "200");
```

… the word "foo" is a standard *metasyntactic* variable indicating unimportance, but what's the meaning of the value 200? Is it an HTTP status code? You have to spend time reading the rest of the test to see if 200 correlates with anything else. If you suspect it doesn't, you can always change the value and see if the test still passes. You might be able to get away with passing the empty string or null:

```
params.put("foo", "");
```

If null or empty values don't work, you can use a meaningful constant name:

```
params.put("foo", ARBITRARY_NUMERIC_VALUE);
```

or even pass a value that imparts the arbitrary meaning:

```
params.put("foo", "bar");
```

The unfortunate intertwining of "essential" and "irrelevant" code in a test requires the reader to do even more work to figure out what's important.

In our case, it turns out that 200 is indeed relevant. More on that later—for now we choose to leave the magic number in the code and revisit it when the tests are easier to follow.

Bloated Construction

Our test dedicates its first four lines of code to a single relevant concept: constructing an "extra context" containing our single key/value pair of foo/200.

```
HashMap<String, Object> params = new HashMap<String, Object>();
params.put("foo", "200");
                                                                    //
HashMap<String, Object> extraContext = new HashMap<String, Object>();
extraContext.put(ActionContext.PARAMETERS, params);
```

Elsewhere in the open source application containing testMessageKey, we found a handful of four-line chunks that were exactly the same—duplication!

Understanding testMessageKey does not require one to know that extra context is created by stuffing one map into another. You can introduce a single-line call to a new factory method and hide this detail without losing any useful information:

```
HashMap<String,Object> extraContext =
   createExtraContextWithParameters("foo", "200");
```

Our test is now a few lines shorter still:

```
public void testMessageKey() {
  HashMap<String,Object> extraContext =
     createExtraContextWithParameters("foo", "200");
                                                                        //
  ActionProxy proxy = actionProxyFactory.createActionProxy("",
     MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
  ValueStack stack = ActionContext.getContext().getValueStack();
  ActionContext.setContext(new ActionContext(stack.getContext()));
  ActionContext.getContext().setLocale(Locale.US);
  proxy.execute();
  assertTrue(((ValidationAware) proxy.getAction()).hasFieldErrors());
                                                                        //
  Map<String, List<String>> errors =
     ((ValidationAware) proxy.getAction()).getFieldErrors();
  assertSoleElementEquals("Foo Range Message", errors.get("foo"));
}
```

Sometimes new test helper methods are useful not only for testing but for production code as well! The “move method” refactoring can be used to relocate methods from test to production code, and some search-and-replace can help find all the places where the production code is simplified by the new method.

Duplication is a major drain on developer productivity for any application. Eliminating the bloat of duplication improves your future development speed in these ways:

- The improved abstraction level reduces comprehension time and thus also the time to maintain the tests and application.
- Future changes to the steps required to construct the “extra context” can be made in one place, not dozens or more.
- You can write new tests more rapidly, introducing a single line instead of having to find-copy-paste-and-alter another chunk of four lines.

Irrelevant Details in Test

Specifying the Locale for the ActionContext appears to have nothing to do with the rest of the test. We can bury it in a construction method (createActionProxy in the example here), killing another bird with this same stone—it also removes some of the clutter required for ActionProxy construction.

```
public void testMessageKey() {
  HashMap<String,Object> extraContext =
     createExtraContextWithParameters("foo", "200");
                                                                                    //
  ActionProxy proxy = createActionProxy(
     MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
  proxy.execute();
  assertTrue(((ValidationAware) proxy.getAction()).hasFieldErrors());
                                                                                    //
  Map<String, List<String>> errors =
     ((ValidationAware) proxy.getAction()).getFieldErrors();
  assertSoleElementEquals("Foo Range Message", errors.get("foo"));
}
```

As you can see, well-named extracted methods are a major key to cleaning up your tests. And once again, there's some chance that you can find and eliminate similar code chunks in other tests nearby, or even in the SUT itself.

Multiple Assertions

You can make the case for multiple assertions in one test method if a single behavior has multiple post-conditions that must hold true. However, additional assertions make it harder to comprehend the test at a glance, and decrease the amount of useful information that the test names can impart on their own. Multiple assertions can also be a code smell, suggesting violation of cohesion in the SUT.

You can consider separating each of these post-conditions into a separate test. There *is* a minor downside: readers must now poke around to find all conditions that hold true for a particular behavior, and hope that the developer has named and organized these consistently. But you get better fault isolation (single reason to fail), as well as an opportunity to improve test names, if you split the tests:

```
public void testActionValidationReportsFieldErrors() {
  HashMap<String,Object> extraContext =
     createExtraContextWithParameters("foo", "200");
                                                                                    //
  ActionProxy proxy = createActionProxy(
     MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
  proxy.execute();
  assertTrue(((ValidationAware) proxy.getAction()).hasFieldErrors());
}
                                                                                    //
public void testActionValidationAllowsMessageRetrievalByKey() {
  HashMap<String,Object> extraContext =
     createExtraContextWithParameters("foo", "200");
                                                                                    //
  ActionProxy proxy = createActionProxy(
     MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
  proxy.execute();
                                                                                    //
  // we could also eliminate this line of ugliness, replacing it
  // with a call to a local method named getFieldErrors. See the
  // next code section for the result.
  Map<String, List<String>> errors =
     ((ValidationAware) proxy.getAction()).getFieldErrors();
  assertSoleElementEquals("Foo Range Message", errors.get("foo"));
}
```

Test names are exceedingly important abstractions: they amplify the essential behaviors captured by a test case, and eliminate the contextual irrelevancy of how those behaviors are executed and verified.

Misleading Organization

The resulting two tests are clear candidates for the idiomatic organization of AAA:

```
public void testActionValidationReportsFieldErrors() {
  HashMap<String,Object> extraContext =
     createExtraContextWithParameters("foo", "200");
  ActionProxy proxy = createActionProxy(
     MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
                                                                        //
  proxy.execute();
                                                                        //
  assertTrue(((ValidationAware) proxy.getAction()).hasFieldErrors());
}
                                                                        //
public void testActionValidationAllowsMessageRetrievalByKey() {
  HashMap<String,Object> extraContext =
     createExtraContextWithParameters("foo", "200");
  ActionProxy proxy = createActionProxy(
     MockConfigurationProvider.VALIDATION_ACTION_NAME, extraContext);
                                                                        //
  proxy.execute();
                                                                        //
  // see previous code excerpt for how we got this single-line assertion
  assertSoleElementEquals("Foo Range Message", getFieldErrors().get("foo"));
}
```

Since both tests contain common setup, you can consider creating a separate fixture around action validation. An isolated fixture would allow you to move the common setup into a common initialization method (setUp or @Before).

And there's still room for a few additional tweaks. How do you know when you're done with a test? One great way is to call over a neutral party and ask them to paraphrase the test. If they're unable to clearly do so, you have work to do!

Implicit Meaning

Now that these tests are split apart and easy to read, it's pretty obvious that we're missing something: It's not at all clear why the key/value pair foo/200 generates a validation error! Unraveling this mystery requires you to read between the lines (aka blow time by digging around through other code or running a series of experiments).

The constant MockConfigurationProvider.VALIDATION_ACTION_NAME makes it obvious that a test double is in play somewhere, no doubt the arbitrator of whether or not validation passes. Unfortunately, the developer buried relevant test context in this test double. To fix the flaw, make the test double construction more explicit. Here's a stab at a clearer test:

```
static String INVALID_FOO_VALUE = "200";
                                                                         //
public void testActionValidationReportsFieldErrors() {
  ActionProxy proxy = createActionProxy(
     createExtraContextWithParameters("foo", "200"));
  proxy.injectValidator(
     createStubValidatorThatThrowsOnValue("200"));
                                                                         //
  proxy.execute();
                                                                         //
  assertTrue(((ValidationAware) proxy.getAction()).hasFieldErrors());
}
                                                                         //
private Validator createStubValidatorThatThrowsOnValue(String value) {
  Validator validator = mock(Validator.class);
  when(validator.validate(value).thenThrow(new ValidationException());
  return validator;
}
```

Your tests should be short and sweet, but it is more important to be clear than to be brief. Don't let your passion for shortening tests push you to bury relevant meaning!

Conclusion

Tests are documentation, or at least can be written as descriptive documents. Use the techniques outlined in this article to sniff out problems and improve tests by increasing their level of abstraction. Your coworkers will thank you, and the deciphering/debugging time you save may be your own.

About Jeff

Jeff Langr has been happily building software for three decades. In addition to co-authoring Agile in a Flash [U4] with Tim, he's written another couple books, *Agile Java* and *Essential Java Style*, contributed to Uncle Bob's *Clean Code*, and written over 90 articles on software development. Jeff runs the consulting and training company Langr Software Solutions from Colorado Springs.

About Tim

Tim Ottinger is the other author of Agile in a Flash [U5], another contributor to *Clean Code*, a 30-year (plus) software developer, agile coach, trainer, consultant, incessant blogger, and incorrigible punster. He writes code. He likes it.

Send the authors your feedback [U6] or discuss the article in the magazine forum [U7].

External resources referenced in this article:

- [U1] http://www.pragprog.com/magazines/2011-02/abstraction
- [U2] http://agileinaflash.blogspot.com/2009/03/arrange-act-assert.html
- [U3] http://agileinaflash.blogspot.com/2009/02/simple-design.html
- [U4] http://www.pragprog.com/refer/pragpub22/titles/olag/Agile-in-a-flash
- [U5] http://www.pragprog.com/refer/pragpub22/titles/olag/Agile-in-a-flash
- [U6] mailto:michael@pragprog.com?subject=Agile-cards
- [U7] http://forums.pragprog.com/forums/134