Software Best Practices for Physicists

Lessons learned in a year as a professional software engineer

Laura Kogler

Pivotal Labs

7 March 2014

Outline

Outline

Physics work

- Undergrad: University of Washington
  - Atomic physics in the Fortson lab
  - Learned C for DAQ software
- Grad school: UC Berkeley / LBL
  - CUORE / Cuoricino
  - C++ analysis framework
- Postdoc: Sandia
  - Radiation detection for nonproliferation group
  - Worked on C++ data acquisition framework
  - C++ analysis software for neutrino experiments

Pivotal Labs

- Software consulting company
- Mostly web and mobile (Ruby, javascript, objective C)
- Projects I've worked on include:
  - tool for data scientists
  - in-browser CAD software
  - iPhone app for stadiums
- Practice and teach agile development methods

Outline

Embrace Change

The core principle of agile development is that requirements for software inevitably change, and instead of trying to predict and plan for every possible requirement up front, we should create systems that are resilient to change.

Change is a fact of life. In my experience this is just as true in physics as it is in industry.

Example reasons software requirements may change

- New discovery changes experiment focus or needs
- Budget considerations reduce scope
- Colleague asks to repurpose code for their application

Extreme Programming

Extreme Programming (XP) is a style of Agile Development invented by Kent Beck in the late 90s. The name comes from the idea that we should take commonly accepted "good" software practices and apply them to the extreme.

- Code reviews $\Longrightarrow$ continuous code review (pair programming)
- Testing $\Longrightarrow$ test-driven development
- Simplicity $\Longrightarrow$ simplest thing that could possibly work
- Good design and architecture $\Longrightarrow$ continuous refactoring
- Integration testing $\Longrightarrow$ continuous integration

Extreme Programming Explained: Embrace Change, 2nd ed, Kent Beck (2004)

Feedback loops

One of the key concepts of Agile Development is **feedback**, applied at every step of the development process.

Shortening feedback loops is a way to increase agility and respond more quickly to change.

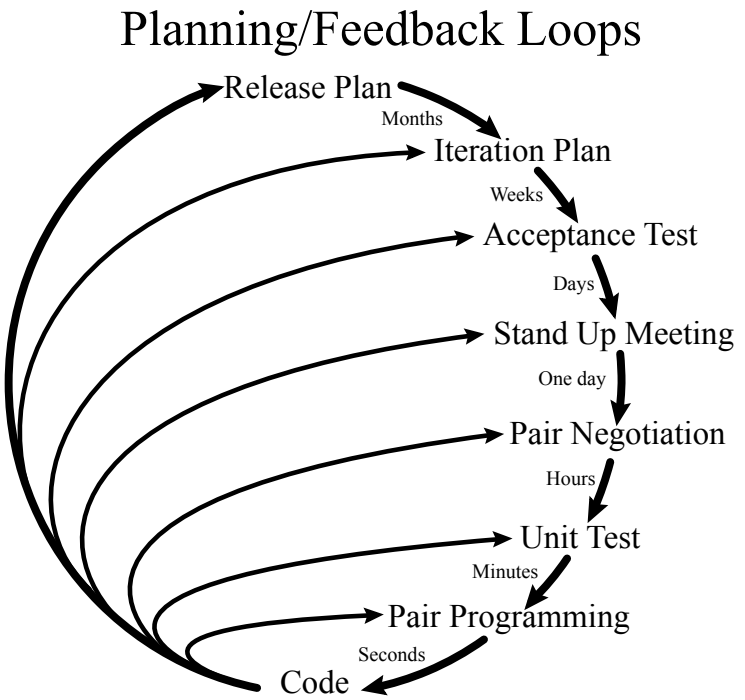

Figure by Don Wells

Outline

Why pair program?

- Continuous code review – improve code quality, reduce bugs
- Improve focus, use time more effectively
- Spread knowledge among team members – eliminate knowledge silos
- Ramp up new team members easily
- Maintain team alignment on goals, priorities, and practices
- It's fun!

But doesn't pair programming slow you down?

A common misconception is that pair programming is half as productive as individuals programming alone. In my experience this is far from true. Here's why:

- Save time on debugging
- Get stuck less often
- Get distracted less often
- Spend less time trying to decipher other people's cryptic code

Combined with the other benefits of pair programming listed above, pairing turns out to be a great value overall.

How to pair program effectively

- One computer, two keyboards (and monitors and mice)
- Check out `http://screenhero.com/` for easy to set up collaborative screen sharing (sadly no linux support yet)
- Work on listening, as well as talking. Take turns!
- Pairing works best with co-located teams. If that's not possible, consider remote pairing or modularizing code base and splitting work by location.

Outline

Daily and weekly cycles

The process we teach at Pivotal includes three types of meetings:

Daily standup	A brief checkin to discuss previous day's events and determine pairing for current day
Iteration planning meeting	Discuss the team's upcoming work for the week
Team retrospective	Review the events of the past week. Identify action items to make improvements in coming week.

In a scientific environment, where team members are not spending 100% of their time coding, this set of meetings may not be appropriate. Tailor to suit your needs. Retrospectives are helpful in guiding process changes and finding the right balance for your team.

Identifying features

One key way in which scientific programming is different from industry is there is typically no separation between the roles of requesting software features, implementing them, and using them. The product manager, developer, and end user are often the same person!

Nevertheless, it can be helpful to separate these roles conceptually, even if they are performed by the same person. Consider formalizing your feature request process. Use an issue tracker to record and prioritize features and bugs, and track your progress toward concrete milestones.

Popular issue tracking software

Pivotal Tracker	`http://www.pivotaltracker.com/why-tracker`
Jira	`https://www.atlassian.com/software/jira`
Bugzilla	`http://www.bugzilla.org/`
Trac	`http://trac.edgewall.org/`

Writing good "user stories"

In agile development, features are generally described in terms of **user stories**: incremental units of functionality that provide value to an end user.
A good user story should meet the following criteria:

- has a clear persona (who is the feature for?)
- unambiguously describes the desired behavior and has clear acceptance criteria
- represents the smallest unit of independently deliverable work that provides value

Example user stories

- As a shifter, I want to see the current temperature of the cryostat on the experiment control panel.
- As a shifter, I want to see the current temperature of the cryostat in red when it deviates by more than 3 sigma from the one week running average temperature.
- As an analyst, I want to see a plot of the detector resolution vs the temperature of the cryostat.

Why bother writing user stories?

It's true that one of the primary benefits of writing good user stories is to improve **communciation** within your team about software requirements. But even if you're the only person defining, implementing, and using features, it can still be beneficial to formally write and track user stories.

- Forces you to think through the desired behavior of the system **before** you start coding
- Gives you data-driven tools for velocity estimation
- Allows intentional and informed prioritization of tasks
- Provides an additional source of documentation of intended code behavior (*tip:* make a habit of referencing your issue tracker in your source control commit messages for cross-referencing later. Some issue trackers even provide source control integration features.)

Outline

Automated software testing

Testing is great! Here are some reasons why you should write automated tests for your code:

- Confidence that your code does what you think it does
- Prevent regresssions
- Encourage good design – classes that can be easily tested are by nature modular and reusable.
- Enable refactoring
- Living documentation that tells you when it goes out of date

Types of tests

Unit tests

- Test functions and objects in isolation
- Run fast
- Should cover all code paths
- Generally should not touch the filesystem, database, or network
- Especially useful for encouraging good design

Integration tests

- Test integration between different components of the system
- May be slow
- Usually only cover a few representative code paths
- Especially useful for catching regressions

Ideally, you will have both types of tests in your code base, and you will have 100% test coverage...

Legacy code

...But unfortunately, we don't live in an ideal world.

What if you have lots of existing code that doesn't have any tests yet?

Check out *Working Effectively with Legacy Code*, by Michael Feathers, which contains tons of valuable techniques for introducing tests into a code base that doesn't already have them.

"Legacy code is simply code without tests"

Test driven development

The idea of test driven development is to write your tests *before* you write your implementation. It's important to see the test fail first, so that you know your changes made a difference. You can think of it like an experimental control.

An algorithm for test driven development

1. Write a failing test case
2. Make it compile
3. Make it pass
4. Remove duplication (refactor)
5. Repeat

From *Working Effectively with Legacy Code*, Michael Feathers

I often find it useful to start with a high level integration test for the feature I am implementing and then work down into unit tests.

But won't writing tests slow me down?

This is a common objection to writing automated tests for code. It's true that writing test requires a substantial investment of up front effort. In a fully tested code base, there may be 2–3 times as much test code as production code.

However, once you get over the initial hurdle, the investment of time and effort pays itself back many times over. You'll be surprised how fast you start to see improvements in reliability and maintainability, and how much time you save debugging and tracking down problems.

A real live example

Imagine we want to make a class called `MeasuredQuantity` that can hold a value and an error, for use in building up more complicated calculations. Let's start by writing some tests, using a lightweight test framework called "Catch":

MeasuredQuantityTest.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "MeasuredQuantity.h"

TEST_CASE( "Fractional Error", "fractional error" ) {
    MeasuredQuantity a = MeasuredQuantity(4, .1);
    REQUIRE( a.FractionalError() == Approx(0.025) );
}
```

Make it pass

MeasuredQuantity.cpp

```
MeasuredQuantity::MeasuredQuantity(double value, double error)
    : fValue(value)
    , fError(error)
{}

double MeasuredQuantity::FractionalError() const
{
    double result = fError/fValue;
    return result;
}
```

Add more tests

MeasuredQuantityTest.cpp

```
TEST_CASE( "If the value is zero", "[MeasuredQuantity]" ) {
    MeasuredQuantity a = MeasuredQuantity(0, .1);
    REQUIRE_THROWS_AS( a.FractionalError(), DivideByZero );
}
```

MeasuredQuantity.cpp

```
double MeasuredQuantity::FractionalError() const

    if(fValue==0) throw DivideByZero();
    double result = fError/fValue;
    return result;
```

...and more tests

MeasuredQuantityTest.cpp

```
SCENARIO( "Adding measured quantities", "[MeasuredQuantity]") {
    GIVEN( "Two measured quantities") {
        MeasuredQuantity a = MeasuredQuantity(5, .3);
        MeasuredQuantity b = MeasuredQuantity(2, .4);

        WHEN( "the numbers are added") {
            MeasuredQuantity c = a + b;

            THEN("the resulting value and error are correct") {
                REQUIRE( c.Value() == Approx(7) );
                REQUIRE( c.Error() == Approx(0.5) );
            }
        }
    }
}
```

Implementing addition

MeasuredQuantity.cpp

```
MeasuredQuantity MeasuredQuantity::operator+(
                          const MeasuredQuantity& right) const
{
    MeasuredQuantity result;
    result.SetValue(fValue + right.Value());
    result.SetError(sqrt(fError*fError
                         + right.Error()*right.Error()));
    return result;
}
```

Test for subtraction

MeasuredQuantityTest.cpp

```
WHEN("the numbers are added") {
    MeasuredQuantity c = a + b;
    THEN("the resulting value and error are correct") {
        REQUIRE( c.Value() == Approx(7) );
        REQUIRE( c.Error() == Approx(0.5) );
    }
}

WHEN("the numbers are subtracted") {
    MeasuredQuantity c = a - b;
    THEN("the resulting value and error are correct") {
        REQUIRE( c.Value() == Approx(3) );
        REQUIRE( c.Error() == Approx(0.5) );
    }
}
```

Implementing subtraction

MeasuredQuantity.cpp

```
MeasuredQuantity MeasuredQuantity::operator+(
                        const MeasuredQuantity& right) const
{
    MeasuredQuantity result;
    result.SetValue(fValue + right.Value());
    result.SetError(sqrt(fError*fError + right.Error()*right.Error()));
    return result;
}
MeasuredQuantity MeasuredQuantity::operator-(
                        const MeasuredQuantity& right) const
{
    MeasuredQuantity result;
    result.SetValue(fValue - right.Value());
    result.SetError(sqrt(fError*fError + right.Error()*right.Error()));
    return result;
}
```

Refactoring

Refactoring means changing the *structure* of the code without changing its *behavior*, in order to improve the underlying design and maintainability of the code.

Tests are extremely valuable for refactoring, because they provide a way to ensure that the behavior of the code is held constant when changes are made. In codebases without tests, refactoring is often avoided out of fear of breaking something, leading to increasingly unmaintainable code over time.

Refactor (remove duplication)

MeasuredQuantity.cpp

```
MeasuredQuantity MeasuredQuantity::operator+(
                        const MeasuredQuantity& right) const
{
    MeasuredQuantity result;
    result.SetValue(fValue + right.Value());
    result.SetError(additiveError(fError, right.Error()));
    return result;
}
MeasuredQuantity MeasuredQuantity::operator-(
                        const MeasuredQuantity& right) const
{
    MeasuredQuantity result;
    result.SetValue(fValue - right.Value());
    result.SetError(additiveError(fError, right.Error()));
    return result;
}
double MeasuredQuantity::additiveError(double e1, double e2) const {
    return sqrt(e1*e1 + e2*e2);
}
```

C++ testing frameworks

There are a lot of unit testing libraries for C++; here are a few that seem to be popular:

Popular C++ testing frameworks

Catch	`https://github.com/philsquared/Catch`
googletest	`https://code.google.com/p/googletest/`
Boost test	`http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html/index.html`
CppUnit	`http://sourceforge.net/projects/cppunit/`

For even more options, see: `http://en.wikipedia.org/wiki/List_of_unit_testing_frameworks#C.2B.2B`

Continuous Integration

Now that you have some great unit tests, you'll want a way to run them automatically! Continuous Integration (CI) is a system to automatically build and test your software whenever changes are submitted by developers.

Continuous integration resources

Jenkins `http://jenkins-ci.org/`

Travis `http://docs.travis-ci.com/`

Outline

Agile Physics?

I'm very interested in how the methods of Agile Development work and don't work in a scientific environment. Here are a few of the issues I've been thinking about:

- Pairing for general lab work, not just programming
- Exploratory programming
- Incentive structure and culture of academia regarding team vs. individual accomplishment

Take home message

1. Test your code
2. Collaborate! Try pair programming.
3. Use incremental, iterative planning and design
4. No really, test your code!

Recommended Reading

- "Best Practices for Scientific Computing", Wilson G, et al. (2014) PLoS Biol 12(1): e1001745. doi:10.1371/journal.pbio.1001745
- *Extreme Programming Explained: Embrace Change*, 2nd ed, Kent Beck (2004)
- *Working Effectively with Legacy Code*, Michael Feathers (2004)

Contact

Please send comments, questions, and complaints to:
lkogler@gmail.com

I'd love to hear your feedback and experiences!

Thank you!