Refactoring is only refactoring when combined with rigorous unit testing. **Kevlin Henney** explains how to take the taskwork out of testing in C++

# Put to the test

PREVIOUS COLUMNS HAVE FREQUENTLY emphasised the benefits of travelling light and keeping your code fit. This advice applies both to the general guidelines to follow in structuring code[1, 2] and to the breadth of a type's interface[3]. Many problems can be tackled with simpler solutions than you might first think—solutions that are sufficient and surprisingly versatile. A good example is the lightweight regular expression matcher presented in the previous column[4].

In the case of that `regex` code, the algorithm was taken from some C code in *The Practice of Programming*[5]. I refactored the code into C++, and not just for cosmetic reasons—the code was adapted and generalised to use iterators. The result was a generic algorithm that worked on any kind of forward iterator range for any type of character; the original was constrained to work only with null-terminated arrays of `char`. Last time we looked at the code and this time I want to look at the process. If you want to travel light, you'd better have a process for doing it. Refactoring is part of such a process.

## Personal hygiene for code

Refactoring[6, 7] has grabbed a lot of programmer mind share. However, it is often still used as a front for rabid hacking. Refactoring is not just taking a piece of code that you don't like the look of and then playing around with it for a week until it satisfies your own preferred indentation, naming or design style, tossing in a raft of extra features while you're at it. Refactoring is actually bound by quite a precise definition that automatically disqualifies most of the programming practice that takes its name in vain[6]:

*Refactoring* (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior.

*Refactor* (verb): to restructure software by applying a series of refactorings without changing the observable behavior of the software.

Far from random code-cutting and splicing, you can see that refactoring is quite a precise and rigorous discipline. A number of projects use the term as a more fashionable and less management-upsetting alternative to the word 'rewrite'. There is certainly an element of rewriting in refactoring, but it is not the same as tearing down and reworking the whole architecture so that nothing builds successfully for a week, a month or longer.

**Many problems can be tackled with simpler solutions than you might first think—solutions that are sufficient and surprisingly versatile**

While refactoring, you should ensure that the code can be built at any time at the drop of a hat or function key. The code base should be as stable and buildable as possible, which means taking many small steps where you might otherwise be tempted to leap. And the code should never offer less functionality.

Refactoring clearly requires a more methodical

### FACTS AT A GLANCE

- Refactoring involves careful changes to code that modify its structure but do not affect its functionality.
- Successful refactoring is best carried out with one or any combination of uninterrupted work, pair programming and unit testing.
- Unit testing should be automated and frequent, not manual and rare.
- Designing for testability reduces interface bloat and coupling in a system.
- The minimum requirement for productive testing is the humble `assert`.

and double-checked approach than is often misguidedly suggested. So how do you refactor with balance, rather than with bugs? There are three ways that you can change code with confidence:

- Make changes with a clear state of mind. This means having continuous blocks of uninterrupted time when you are feeling alert. So, that excludes afternoons for most people—especially Friday—and suggests that Monday morning and any 'morning after the night before' is also probably inappropriate. It also means that unpartitioned open-plan offices are unsuitable environments. If you cannot reach the zone of total concentration, you will miss too many tricks to carry out refactoring effectively.
- Make changes with a colleague. Two pairs of eyes are better than one when it comes to spotting careless mistakes and suggesting alternative designs. This kind of pair programming is effective in many different office environments and most times of day. One person's attention low may coincide with the other's high, flattening out the peaks and troughs. Apparently some managers regard any form of pair programming as a waste of resources. Many such managers indulge in long and frequent meetings that are, however, apparently another matter and should not be judged by the same productivity criteria. Pots. Kettles. 'Nuff said.
- Make changes against a set of unit tests. You break it, you find out very quickly. Footfall is surer because the feedback loop between change and effect is tightened. Change, compile, test. And then, depending on the outcome, you either move on to the next step or fix the bug just introduced with the change.

Or you can use any workable combination of the three. Each approach also has benefits above and beyond refactoring. For instance, working without interruption is always more of a pleasure than working with distractions. You are likely to write code more clearly and cleanly, introducing fewer bugs in the process, tackling design problems more easily and so on.

Pairing is a practice that can address many different development activities. Two pairs of eyes are better than one for writing tests, debugging code, deciphering existing code and training or familiarising someone else with the existing system. Pairing also serves as a form of active code review that is superior to the slow cycle of organising review meetings, printing out the code, involving many people who make vague suggestions or get caught up in the wrong level of detail—and then finally making the changes. Depending on the social skills of the code reviewing peers, a meeting can also be hard on the code author's self esteem. Code reviews are a good idea, but they are better when they are more interactive and intimate.

**Perhaps the best advice is also the most succinct: Test early. Test often. Test automatically**

Unit tests make you think about the design of the code you are testing and, in particular, the interface to classes and functions. Unit tests can be seen as part of a design-driven approach[8] that reduces the coupling of your system and sharpens the interfaces. Highly coupled code is hard to test because the unit of test must be that much larger: loosely coupled code has more easily separable and testable chunks. You also find out what you can and can't test easily and what it's like to use your classes and functions. This experience feeds back positively. You write fewer of those lame 'setter' and 'getter' functions that plague so much code that masquerades as object-oriented and you cut out functions that aren't used. You also instantiate all of the template code that you've written, which can potentially hide basic compilation errors for a long time after it was first produced.

## The joy of testing

So how do you go about testing? Perhaps the best advice is also the most succinct[9]: *Test early. Test often. Test automatically.*

This advice goes against the test approach adopted by many organisations (even the thorough and methodical ones). Let's get one thing straight: writing code is a lot more fun than most of the other activities in software development, and testing has to be one of the activities traditionally associated with the least amount of fun. Many projects that explicitly include a testing activity often include it as a test phase, which is a sure way of guaranteeing the absence of fun. You've done all the interesting stuff and now you've got to test it all. For a very small project, this may not be a major obstacle. But for anything larger, it is quite a serious demotivator. And you may find out late in the day that you have written something that is quite hard to test. An interface may be broader than is strictly useful and more highly coupled than is strictly necessary. Additionally, if you leave testing until late and then discover that the project is late, guess what gets squeezed? Test early. Test often.

The significance of automated testing is often overlooked, but is perhaps one of the most important pieces of the puzzle. Many systems are tested by manual inspection. Someone sits staring at a screen, either pumping data in or pushing buttons, waiting for the right result. This is neither engaging nor effective. It is, however, time consuming, which means that you are never in a hurry to do it early and often. It is also not really unit testing, unless you consider the whole application to be the unit. This approach to testing makes expensive monkeys out of bright programmers.

If writing code is considered more fun than testing, then make testing a matter of writing code. If you write code to test code, you can rerun it automatically. Regression testing is no longer a chore and you can be more selective about what parts of your system you include in a test.

So what tools do you need for this? `#include <cassert>` (or `#include <assert.h>` if you want the original C header) is not a bad place to start. Yes, there are expensive tools on the market that will do all kinds of fabulous things for you, but the minimum toolset is `assert`—a macro in C and C++, a built-in facility in Python and Java (as of SDK 1.4) and an easily provided feature in most other languages. This is a very low entry level indeed, but one that gets you into testing straight away without any fuss or ceremony.

Once you have played around with assert-based testing for a while, you will have a better idea of what you want from your test harness and you can move on to something more powerful. Many people refactor such a harness out of their test code to keep total control over it. Others adopt one of the open source xUnit family of test harnesses (most famously, JUnit for Java[10]), which are simple but effective testing frameworks. If you want a level of static checking, automatic wrapping of library calls, memory

leak detection, coverage statistics and so on, a commercial tool is probably the right choice. However, none of these change the basic need for a testing mindset and knowing what you need for your own application. Go to product vendors with clear empirical requirements, rather than letting them suggest their requirements to you.

## Testing the ins and outs

The presentation layer—whether console, browser or GUI—is often held up as an obstacle to automated testing. In part, this is a valid objection, but only in part. A GUI test should test the appearance, rather than the logic. That means separating the model logic from all the window presentation logic. The problem is not so much that GUIs are hard to test, but that many programs are not designed to be tested. When you design for testability, you put into practice the common design recommendation to decouple the actual presentation details from all the event handling and core logic. The manual element of testing is then reduced to testing the appearance and usability rather than the underlying logic. The real challenge here is not the technology, but the quality of the design presented by so many books, courses and code samples for windowing APIs and frameworks. Developers model their code after these examples, but the aims of the two sorts of code are incompatible: the samples show sufficient code to demonstrate how the API is used, not how to build a large, testable commercial product.

One presentation facility that can be tested automatically is I/O streams. Instead of hardwiring the input and output streams (e.g. to `std::cin` and `std::cout`), pass in `std::istream` and `std::ostream` parameters. In a test harness, these can be wired up to `std::stringstream` objects or `std::fstream` objects opened onto files prepared in advance with test data and correct responses.

If such substitution is not an option, you can get in under the wire and change the internal plumbing. Consider the toy function in listing 1. How would you test the decimal to hex conversion? You could have a `main` call it and script up a test externally in something like Perl or another scripting language. This would be fine, but given that this is a C++ column, let's look at a C++-only solution that doesn't involve running multiple processes.

Listing 2 shows a main function that runs the function through a single test case without changing any code in `dec2hex`. The technique is to substitute string buffers of type `std::stringbuf` in place of the normal console-connected `streambufs` used by `std::cin` and `std::cout`. You prime the input with a value and then collect and compare the output. Remember to restore the standard I/O streams to their former state. This is more than just politeness: if you don't, the streams will attempt to `delete` the local `stringbuf` variables on program exit, as well as leak their own original buffers. The catastrophic result of attempting to `delete` an object that was not created with `new` somewhat dwarfs the memory leak problem. The object ownership model in I/O streams is easy to trip over, in part because it fails to follow the recommendation to make acquisition and release symmetric[1]. If you wish, you can use objects to automate the setting and resetting of the buffers[2].

You can scale this approach up to many test cases by surrounding the whole lot with a loop and popping a predefined sequence of string values to test. You can use a little string manipulation to exclude the prompt text from the test condition. Although this is a very simple example, it demonstrates the anatomy of an automated I/O test.

## Refactoring match

To return to where we started out, how can testing work in support of refactoring? I refactored the `regex_match` functions in the last article[4] against a set of tests.

The original code was written in C[5], so that was where I started. The functions were copied verbatim. Compilation provided the basic reality check that the code was copied correctly and that the C was clean enough to be used as C++. I then wrote out 10 or so assertions that seemed briefly to exercise the range of its functionality. I felt that only a few tests were required because the code was published by authors of note. The whole chapter devoted to testing in the same book also did a lot to inspire confidence.

With the raw materials in place, I changed the names to be closer to the ones that I intended to use in the final code, which required some slight modifications. I then tightened up the types used, e.g. using `const` to qualify `char *` and `bool` instead of `int`, which did not require any test changes. Because I was aiming for an STL-styled algorithm, the next step was to switch the argument order around, which clearly necessitated changing test code. I then replaced some of the pointer-based expressions by expressions that would work for STL forward iterators, which did not require any test changes. Extra arguments were added so that iterator ranges were used instead of assuming null-terminated strings—the tests changed to accommodate the interface change, but no new tests were added.

I followed through with some more work ensuring that only forward iterator operations were used. I then introduced templating, which did not require any changes to the tests, and then new tests were added to check that pattern matching worked for embedded nulls and wide-character strings. In between each sentence of this paragraph, the tests were compiled and run.

The resulting interface for pattern matching is declared in listing 3 and the final tests are shown in listing 4. It is certainly possible to add more tests, and it is certainly possible to refactor the test code. However, for the immediate problem, the tests as structured were sufficient to give the right level of confidence. If I were to go back and extend the capabilities of the regular expression matching and therefore add to the number of tests, I would almost certainly refactor `main` before doing anything else. Don't refactor the test harness and the test subject in the coding spree between compilations. Driving the code from a table that held sample values and expected results would be the best way to capture the commonality in this case.

Most developers do not work with such small, insular pieces of code. However, the principles and practices do scale. The trick is to make code sufficiently modular and independent that it becomes testable. Testing doesn't have to be tedious and it certainly isn't rocket science. It is a matter of documenting what you might otherwise have thought about testing by hand or would have looked for in a code inspection. However, instead of wasting time documenting the tests with a word processor, document these cases in code. That way you never actually have to do the tests yourself: you just compile and run the documentation. With a body of tests to lean on, you are much freer to modify the code to improve or extend it. ■

**References**

1. Kevlin Henney, "Six of the Best", *Application Development Advisor*, May 2002, www.appdevadvisor.co.uk.
2. Kevlin Henney, "The Rest of the Best", *Application Development Advisor*, June 2002, www.appdevadvisor.co.uk.
3. Kevlin Henney, "Stringing Things Along", *Application Development Advisor*, July-August 2002, www.appdevadvisor.co.uk.
4. Kevlin Henney, "The Next Best String", *Application Development Advisor*, October 2002, www.appdevadvisor.co.uk.
5. Brian W Kernighan and Rob Pike, *The Practice of Programming*, Addison-Wesley, 1999.
6. Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.
7. Refactoring home page, www.refactoring.com.
8. Kent Beck, "Aim, Fire", *IEEE Software*, 2001, http://computer.org/software/homepage/2001/05Design/index.htm.
9. Andrew Hunt and David Thomas, *The Pragmatic Programmer*, Addison-Wesley, 2000.
10. JUnit home page, www.junit.org.

*Kevlin Henney is an independent software development consultant and trainer. He can be reached at www.curbralan.com.*

## Listing 1: How would you test this function?

```
void dec2hex()
{
    std::cout << "decimal? ";
    int value = 0;
    std::cin >> value;
    std::cout << "hex: " << std::hex << value << std::endl;
}
```

## Listing 2: Running listing 1 through a single test case

```
int main()
{
    std::stringbuf out, in("42");
    std::streambuf *old_cout = std::cout.rdbuf(&out);
    std::streambuf *old_cin = std::cin.rdbuf(&in);

    dec2hex();

    std::cout.rdbuf(old_cout);
    std::cin.rdbuf(old_cin);

    assert(out.str() == "decimal? hex: 2a\n");

    std::cout << "OK" << std::endl;
    return 0;
}
```

## Listing 3: An interface for pattern matching in the refactored regex_match function

```
template<
    typename text_iterator,
    typename regex_iterator>
bool regex_match(
    text_iterator text_begin, text_iterator text_end,
    regex_iterator regex_begin, regex_iterator regex_end);

template<
    typename text_iterator,
```

```
    typename regex_char>
bool regex_match(
    text_iterator text_begin, text_iterator text_end,
    const regex_char *regex);
```

## Listing 4: The final tests

```
int main()
{
    // test three-agument regex_match
    const std::string text = "the cat sat on the mat";
    assert(regex_match(text.begin(), text.end(), "the"));
    assert(!regex_match(text.begin(), text.end(), "foo"));
    assert(regex_match(text.begin(), text.end(), "c.t"));
    assert(!regex_match(text.begin(), text.end(), "t.t"));
    assert(regex_match(text.begin(), text.end(), "^t"));
    assert(!regex_match(text.begin(), text.end(), "^c"));
    assert(regex_match(text.begin(), text.end(), "^t.*t"));
    assert(!regex_match(text.begin(), text.end(), "^t.*f"));
    assert(regex_match(text.begin(), text.end(), "t$"));
    assert(!regex_match(text.begin(), text.end(), "a$"));
    assert(regex_match(text.begin(), text.end(), "^t.*t$"));
    assert(!regex_match(text.begin(), text.end(), "^t.*f$"));
    assert(regex_match(
                text.begin(), text.end(), "t.*cat.*o"));
    assert(!regex_match(
                text.begin(), text.end(), "t.*rat.*o"));

    // test four-argument regex_match
    const std::string pattern = "s.*o";
    assert(regex_match(
                text.begin(), text.end(),
                pattern.begin(), pattern.end()));

    // test regex_match across embedded null
    std::string next = "abcd";
    next += '\0';
    next += "efgh";
    assert(regex_match(next.begin(), next.end(), "fg"));
    assert(regex_match(next.begin(), next.end(), "d.e"));

    // test regex_match with wide-character strings
    const std::wstring wtext = L"the cat sat on the mat";
    assert(regex_match(
                wtext.begin(), wtext.end(), L"^t.*t$"));
    assert(!regex_match(
                wtext.begin(), wtext.end(), L"^t.*f$"));
    assert(regex_match(
                wtext.begin(), wtext.end(), "t.*cat.*o"));
    assert(!regex_match(
                wtext.begin(), wtext.end(), "t.*rat.*o"));

    std::cout << "OK" << std::endl;
    return 0;
}
```