Software testing on RISC OS

Some methods and mechanisms

Gerph, May 2021

Software testing on RISC OS

Some methods and mechanisms

0. Introduction

0. Introduction

This is a pretty rushed talk, as I've only had a week to prepare so... if it's a jumbled mess, we'll just have to revel in the disaster it becomes!

Introduction

What we'll talk about

- Problem area
- Testing background
- Testing examples
- Conclusions

I'll take questions between the sections, but if you feel you want to ask something part way through, feel free to ask in the chat and I'll try to keep an eye on that.

Introduction

What we'll talk about

I'm going to talk about a number of things in software testing.

- I'm going to talk about some of the things that are hard to test on RISC OS.
- I'll talk about what testing is for, and some test theory.
- I'll give some examples of how you might create some tests.
- And there'll be some small demos of tests running and explanation of them.

I'll take questions between the sections, but if you feel you want to ask something part way through, feel free to ask in the chat and I'll try to keep an eye on that.

Introduction

Who am I and why do I care about testing?

- A RISC OS architect and engineer.
- Work in groups providing test and tooling to engineers.
- Lots of experience of code I thought was good, failing...
- ... and the deep mistrust that it produces.

Introduction

Who am I and why do I care about testing?

I'm a RISC OS architect, and in my day job, I'm a software engineer - I work in the engineering infrastructure group at a company which produces a time travelling debugger. Essentially I provide the test systems and support for a company that produces a tool for testing and debugging.
During my time doing work with RISC OS, I've always cared about testing, but it's been hard, and there's so many reasons for that. But over the years I've learnt one or two things which are useful, and I find that not having testing is... unsettling.
Finding bugs in something that you think is good makes you question what else you've given too much credit to. Whilst working on RISC OS I called this the 'shifting sand problem'. Once you start delving into parts of the system that you thought was ok (or maybe just ok-ish) you find more and more things wrong. And suddenly you feel that the whole OS is built on shifting sand - everything could give way underneath you, and you wouldn't be able to trust anything.
This was one of my development strategies at that time was 'month of development, month of testing and stablisation'. Anyone using TaskWindow or PipeFS will know that there be demons lying there, and poking them can lead to uncovering horrors.

Introduction

Why do this presentation? (1)

- I created a build service to make it possible for people to do testing...
- ... but nobody's using it. Why?
- Probably because testing just isn't important to many people.
- Also, it's kinda alien to people.
- In the past I've seen engineers frustrated at the lack of testing:

> *(~1997) This is stupid, but this is the second release where the non-function of a simple binary search has turned out to be a bug, and I am tired of it.*

Introduction

Why do this presentation? (1)

You may know from the earlier presentation that I've been quite passionate about things being testable well, and that I put together the RISC OS Build service a year ago. Nobody's used the service in anger as far as I can tell, and nobody's come to talk to me about it. So this annoys me. And it's good, 'cos it means I don't have to answer people's questions. And it's expected, 'cos it is a bit special.
There's a quote that I always come back to when it comes to testing. Back in about '97 I ported a load of Mozilla to RISC OS, and encountered the quote in the code that parses colour names - along side a test which will check every colour.

> *This is stupid, but this is the second release where the non-function of a simple binary search has turned out to be a bug, and I am tired of it.*

This quote is one of the earliest expressions I'd seen of why testing was important - and even on the things that 'surely you can't get wrong'. If Netscape, who were a hugely important company, couldn't get a simple colour selection right and had made multiple releases with it broken, then anyone could make that mistake. And all it takes to fix the problem is to have some a little bit of code to test that things work correctly.

Introduction

Why do this presentation? (2)

- Frustration at lack of testing isn't restricted to ancient things...

> *(2021) Replying to @nemo20000 and @oflaoflaofla*
> *Confirmed. When there's a Prefix set, the RO5 DDEUtils treats a filename starting with spaces as a null string, so prepends the Prefix (which is a directory) and suddenly your (font) file is a directory.*
> *That's the bug. Insufficient familiarity with API inputs. No testing.*

- Without a testing attitude, a build and test service isn't useful.
- So... this presentation is intended to show why and how I do testing...

Introduction

Why do this presentation? (2)

But the thing that kicked me was a recent comment from my friend Chris quoting Nemo on twitter...

> *Replying to @nemo20000 and @oflaoflaofla*
> *Confirmed. When there's a Prefix set, the RO5 DDEUtils treats a filename starting with spaces as a null string, so prepends the Prefix (which is a directory) and suddenly your (font) file is a directory.*
> *That's the bug. Insufficient familiarity with API inputs. No testing.*

My response? 'I'm sure I documented fixing that, and surely someone would have applied similar fixes.'

> *I'm sure I documented my changes to ddeutils' handling of calls somewhere.*

Actually the specific reply I gave was...

> *It REALLY annoys me that it's not even a matter of trying to reunify the branches. information about the changes in Select was made available at the API and changelog levels and nobody even tried.*

But hey-ho, that's enough about my problems.
I didn't have any tests for DDEUtils at the time either, so can't be too complacent. I had, however, fixed that bug in 2004, along with a raft of others. Yeah, it's hard to test some things some times, but it seems like testing is just bad on RISC OS. Not just that it's hard, but people seem to have the wrong attitude to doing it.
And so this is the thing that made me want to do a talk on testing. It was my accepting that the build service isn't something that RISC OS users can use. Testing just isn't something RISC OS users do, so expecting a web service which allows automated testing in the cloud to be taken up is not going to help.
It's like trying to introduce giving JCB to the kids building sandcastles, and expecting them to make better models. So if testing isn't something that RISC OS users do, let's explain what it is and how it can be done.

1. The Problem Area

1. Problem area

Problem area

Why is testing a problem on RISC OS?

- Testing is mostly done ad-hoc.
- Automated testing is non-existent.
- You can see this by the code that is available just not having any useful tests.
- You can see stupid problems that would have been caught by testing.
- A lack of testing encourages a continued lack of testing.

Problem area

Why is testing a problem on RISC OS?

It seems - to me - that testing on RISC OS is something that gets done ad-hoc by the developer, and maybe beta testers, and then pushed out to the world to test. Automated testing is not a thing. It's verifiably not a thing because nobody at all has taken up the build service, and there is (to the best of my knowledge) no equivalent. In any professional software company this would be a cause for concern. Even if we assume that developers are awesome, the chance that something stupid creeps into the code is still there.

There's a whole operating system with zero tests. Assuming that everyone that worked on it was awesome is completely unreasonable, and evidence shows that there are stupid bugs that would have been caught by testing. And of course, introducing a test where there already exist tests is a lot easier - so not having tests encourages not adding more.

Problem area

Why is testing hard on RISC OS?

- Tools and frameworks don't exist to allow testing.
- Lack of process model or system security is a deterrent to automated testing.
- System is large and complex.
- Desktop doesn't lend itself to testing.
- Modules can easily destroy the system.
- User mode programs can also easily destroy the system.

Problem area

Why is testing hard on RISC OS?

The thing is that testing on RISC OS is kinda hard. There aren't the tools that you expect on other systems, the process model isn't anything like how it works elsewhere, and the system can just blow up underneath you if you sneeze at the wrong time. Not true, of course - but it misses the truth by mere inches.

The system is large, it gets in the way, and isolating yourself so that you're only testing small things at a time is tricky.

The desktop doesn't lend itself to easy testing, and so people don't bother automating those tests. There's very little in the way of libraries and frameworks which can help you in this regard, and tools that you might use on other platforms are unlikely to be helpful.

Modules are considered hard to test because they execute in SVC mode and any mistakes there are generally fatal - largely because of the lack of any protections against problems.

And of course your regular command line program can also be fatal because of a lack of any protection.

Are there any questions up to this point, before I move on to some testing theory.

2. Testing Background

2. Testing Background

Testing background
My anecdote

- Manually testing my new Portable module had all gone well.
- So I started writing some automated tests for it...

Testing background
My anecdote

Let's start with a small anecdote from a couple of weeks ago. I was updating my implementation of the Portable module - which gives information about the battery and charging. I had changed it from being a single implementation with only a small amount of information to reading lots of information from my MacBook. I have a little BASIC program that I use to read information from it. It's pretty simple and it'll crash if something's wrong, or show invalid values. I look at it after making changes and see if things work. When they don't I fix things, and when they do I am happy.
So I was happy and it was reporting sensible values from my laptop. Then I started writing tests. The tests actually check that what they are getting back is actually what they expect, and they are they things that must work before I can commit the code to my branch.

Testing background

My anecdote

- Manually testing my new Portable module had all gone well.
- So I started writing some automated tests for it...
    - First test added... found a bug.
    - Second test added... ok
    - Third test added... found a bug.
    - Fourth test added... found a bug.

STEP

- First test added... found a bug.
- Second test added... ok
- Third test added... found a bug.
- Fourth test added... found a bug.

Testing background

My anecdote

- Manually testing my new Portable module had all gone well.
- So I started writing some automated tests for it...
    - First test added... found a bug.
    - Second test added... ok
    - Third test added... found a bug.
    - Fourth test added... found a bug.
- But surely I'd done some tests? How is it so broken?
- Lack of rigour and care.

STEP

But I'd tested it by running my example code - why did my manual tests not find the problem?

Because in the manual tests I'm not applying rigour. I'm human, so I try things out and they seem ok, and I feel good about it. Then I apply some rigour and am reminded that it's very easy to be complacent because a cursory test says that it works.

Testing background

Manual and automatic?

- Manual testing is any time you're just eyeballing the results.
  - Good to give you a feeling that things work.
- Automated testing is any time that the computer checks the results.
  - Good to give you confidence that it keeps working.

Testing background

Manual and automatic?

I've said that I ran my example code and then I wrote some tests. The example code is, essentially, a manual test. It isn't able, itself, to tell me that things are working. It needs a human to interpret it. Pretty much anything that exercises your code could be called a manual test, so long as you have a criteria for what's right and what's wrong (even if it's only in your head).
Automated tests are similar, but they actually tell you whether their criteria have been met or not. Automated tests are something that you can run and at the end they'll tell you whether things are good or not. So they are great for running after you've changed some of your code. In most software development you'll find that automated tests are vital to ensuring that things keep working - they'll be run after anyone makes any change, automatically, and will shout at you by email, instant message or sometimes big red lights shining at a disco ball. Seriously, you don't want to be sitting under a red light in the office. It's creepy. Oh, and it's bad that the tests failed too.

Testing background

How many bugs does your code have? (1)

So how many bugs does your code have?

- Industry standards say 10-50 defects per 1000 lines.
- You're going to be worse than that.

How buggy is Pyromaniac?

- Around 60,0000 lines.
- So 600 bugs at 10 per 1000 lines? - Nah, much, much more than that
- 1250 tests is way too few tests!

Testing background

How many bugs does your code have? (1)

So how many bugs does your code have?
Different sources give some statistics, which you can look up, but... let's say that in general you expect 10 defects per 1000 lines. That's really optimistic by industry figures, but it makes the maths easier. Code that's had good review and has seen widespread use over years will have much lower numbers. Code that's written by a single developer without much oversight or rigour will have a much higher figure. Code that's written without a care for testing will be even higher.
So, let's give a specific example from my code...
Pyromaniac is ~60,000 lines. So we might expect 600 bugs in there. That's way off, in my opinion - I'm a single developer with no peer review, and it doesn't have widespread testing. So that's got to push the numbers higher. Plus, I've intentionally made the choice to not make it bullet proof in places where the implementation doesn't matter that much. It's arguable if that's a bug if you've made a note and a conscious decision to implement it that way, but what it comes down to is 'does the system work correctly'.
I've got around 1250 tests in Pyromaniac. Does that mean that I've caught most of the bugs - it's bigger than 600? No, and you knew that 'cos that's not the way that these numbers work!
And that's a pitifully small number of tests for an operating system. Some of those tests are targetted at specific interfaces and exercise them well. Others only hit an interface once - enough to give me confidence that in the most basic of cases it doesn't fail. And lots of interfaces aren't used. Never mind the fact that many of the interfaces can be used in combinations which aren't exercised.

Testing background

How many bugs does your code have? (2)

Does manual testing not count then?

- Only in that it gives you confidence.
- Doesn't really check that your code is working in more than the cursory cases.

Testing background

How many bugs does your code have? (2)

But I tested this stuff manually - how much does that count? You'll remember from my example of the Portable module, that I'd tested it manually, but then when I came to write automated tests I immediately found problems. Does that mean that my manual test was bad? No. Just that it increased my confidence in the code much more than was warranted. Basically manual tests are worthless except to increase your confidence in the code, and they do that disproportionately to the actual effect that the testing has. Confidence is important, but also understanding what it means.

Testing background

Do the bugs matter?

- Not all bugs matter.
- It always depends on context and your use case.
- Being able to say that there is a bug, is as useful as being able to fix it.

Testing background

Do the bugs matter?

So we think that there are bugs. Ok. That doesn't mean that all bugs 'matter' to users. Some might be corner cases that only happen if specific circumstances crop up. For a little toy application, that's probably fine. But for a commercial application, the user base is larger so the risk is greater. And for an operating system, the risk is even higher. So every bug needs to be treated with the respect its risk deserves. If it's a fault that can only happen if the user is using their computer under 20 metres of water, you might not care. On the other hand if it's part of your oxygen supply system, you may want to pay a little more attention to it.
This is the reason that you'll see a list of known faults with some software - those are things that the developers know about but they're ok with being there. Understanding what bugs exist and what are acceptable is actually as important sometimes as fixing them. At least when you know about them you can make an informed decision.

Testing background

How much testing should you do?

- "More!"
- Enough to make you feel confident that your product works.
- That means being realistic about how much and what you have tested.
- Think about the many combinations that you might test...
  - Different file system types, different access permissions, or strange environments
- How many do you actually exercise?
- How many combinations of those with other factors do you exercise?

Testing background

How much testing should you do?

This is an impossible question to answer, but it's one that is useful to explore. The trite answer is 'more', but that's not very helpful. What matters is whether the product is working for your users. If you see lots of problems coming from your users then you probably need more testing. Or possibly you need better documentation or user interface or design - but those aren't what I'm going to talk about here. Just remember that not everything comes down to changing code.

So that's one source of information about how much testing you need. Of course, before you release, you don't have that so the question becomes 'How confident are you that users won't have problems?'. This is a terrible question because human arrogance plays such a big role here, but if you can be honest with yourself then it makes it easier to say 'yes, I'm good with that'.

What you can do is destroy your ego by looking at the things that you have tested and things that haven't been tested. This works if you are doing adhoc testing or automated testing. If you've got an application that can save documents, you can look at the different ways that they can be saved. You can save to a local disc, or a remote disc, or a RAM disc, or a disc that's read only, or over a file that is locked, or into a directory that doesn't exist, or a filesystem that doesn't exist, or on to a disc that doesn't have enough space, or a disc that's been removed, or to PipeFS, or to null.

Those are just some of the cases that I can immediately think up for a single type of user's operation. How many of those have you hit in your testing? Probably 3 or 4. And they'll be the 3 or 4 that are common and important. Which is cool, but they probably won't be the ones that will give you errors. That's 'negative' testing - checking that the program keeps working when something goes wrong, rather than checking that the program does what it's meant to when things go right. It's a commonly missed thing, and probably you've missed it in the manual tests. There's so many ways that things can go wrong, that even if you have hit some failure cases, there will still be more behind the scenes.

Still feeling confident about your code? Great. Now let's apply that same strategy to all the other operations in your application that you can perform. Loading, processing, dealing with each type of content, each type of user input, each interaction from outside the application...

There are just too many combinations to consider. And your testing will hit a tiny tiny fraction of them. You should be humbled. Because you suck. Just like every other software developer in the world. Some people can hold a design in their head and can write things well so that they'll never fail. Some people can intuitively spot holes that others will miss. They're just that good. But that's not you. And if you accept that you're not that good, your confidence will be lower - or as I like to think of it, you'll be more realistic about your abilities and your code.

Testing background

What is testing?

- Testing means many things to different people.
- And it doesn't even have to involve running the product.
- Not going to cover all the dimensions and types of tests.

Testing background

What is testing?

Having said that it's hard to do testing and that we're all bad at it, let's step back and look at what testing is and why we do it. Fortunately this is pretty easy. Testing is anything that looks at your product and tells you that it's doing what it should, or that it's not. It's important that this definition doesn't require running the product, or for that matter even looking at the code. Peer review (having someone look over your code) is a form of testing, which doesn't involve running the code. And a tangential form of testing is seeing the effects that your product has on others - if you're breaking a remote server because your millions of users (or maybe 3 users in the case of some products) are all hitting it at once.
But whilst there's a lot of things that can be involved in testing, I'm only going to talk about a very fraction of the ways that you can test things.

Testing background

Types of testing

Commonly discussed testing scopes:

- *Unit test* - Tests individual parts of the code.
- *Integration test* - Tests a module works internally, without reference to other parts of the product; sometimes split into 'module' and 'integration' tests to distinguish between tests of a module, and tests of interactions between modules.
- *System test* - Tests that the product works in the form that it will be delivered.
- *System integration test* - Tests that when used within an environment that the product is expected to work.
- *Customer test* - Tests that when used by the customer in their environment, the product works.

Each involves more and more of the product and environment.

Testing background

Types of testing

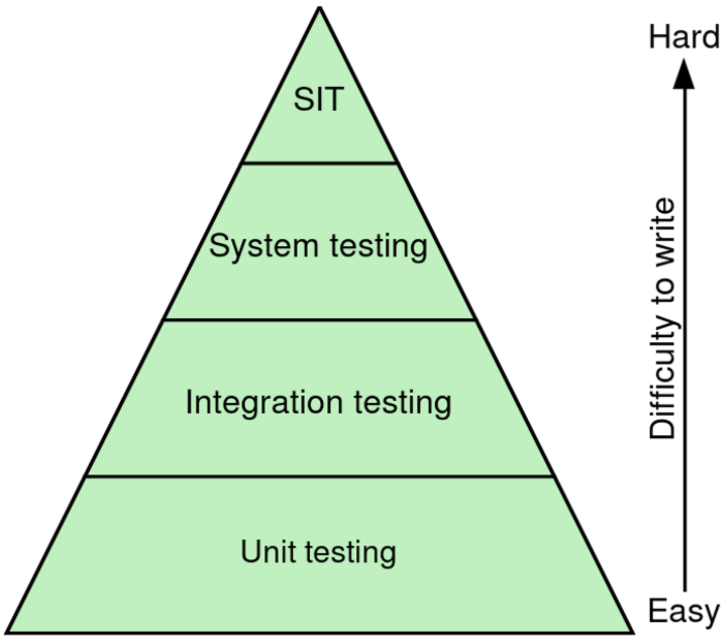
Let's have a look at some of the types of testing that we can do... There's a lot of different types, and specifically I'm going to look at what are traditionally focused on as the levels of testing in a project. These describe the scope of the tests - how much of the protect and its environment the tests will cover.
Commonly these are:

- Unit test - Tests individual parts of the tests
- Integration test - Tests a module works internally, without reference to other parts of the product; sometimes split into 'module' and 'integration' tests to distinguish between tests of a module, and tests of interactions between modules
- System test - Tests that the product works in the form that it will be delivered
- System integration test - Tests that when used within a product environment the product works
- Customer test - Tests that when used by the customer in their environment, the product works

As you progress from the lowest level (unit test) through to the highest level (customer test) you introduce more and more parts of the code and the environment into the system. More things can interact - and so more things can go wrong.

Testing background

Types of testing



STEP
At each stage the difficulty in setting up and writing the tests increases, usually not much, but enough to make each step harder to do. And because of the complexity you're introducing at each stage, you find it's harder to diagnose problems, and sometimes harder to reproduce because there are hidden variables you didn't know about. And because they're more complex, the time to find solutions to problems increases as you get more involved with the testing of the whole product. That means that the cost is higher, in your time and, if you're doing this as part of a job, the monetary cost too.
This sort of diagram turns up all over the place to show how much more benefit you get from lower level tests, which are easier to write. It also gets used to show the cost of bugs at the higher levels is greater. It's not the whole picture - because a good test at the highest level may actually have caught many of your problems, and testing against your business requirements is more effective in ensuring time isn't wasted.
But having many simple but effective low level and mid level tests gives you a lot of confidence in your product.

Testing background

Rationalising testing

- Testing is never an activity considered in isolation.
- Features, and visible changes, will always compete for your time.
- Time spent hunting for the cause of a bug might be better spent adding tests to prevent it.
- Adding tests proactively reduces bugs escaping to users.
- You can always make headway in automated testing, even if there are competing pressures.

Testing background

Rationalising testing

There's another cost is usually implicit in how individuals and companies decide what to focus on, and that's opportunity cost. That's the business name for it anyhow - it's the implicit cost that by spending your time looking for and fixing this bug, you're not doing something else.
As an individual, that might be the fact that looking for bugs is boring and you want to do something interesting - you don't want to lose the incentivising buzz of a new feature because you're looking at some silly little bug that came up once. You also know that that bug might be something important, and might bite you later.
As a company, you know that customers don't pay for bug fixes, so features are vital. You also know that customers leave if the product is bad.
So you rationalise what the best use of your time and resources is. If you found yourself being swamped by bug reports from users that were all impossible to fix, you certainly change your tune that features are important, and focus on testing.
But then you have to know where to spend your time doing testing. There's a strong appeal of doing system and system integration testing, because that's close to the environments and that the customer uses. What do I mean by that? I mean, you receive a bug report from a user, and you create a little test that does exactly what the user said to reproduce the problem - and that's your new test for the problem.

Testing background
Example bug report

> *User has reported that when loading a file, the application crashed, and they give you the file.*

- Create a program to load file into app, check it doesn't crash.
- You program fails - bug reproduced! Now we have a test.
- Can you see where it crashed?

This is a system integration test - running in the environment the user will use, using all the product.

- System tests take longer to run and to write - there is more in them.
- They are more involved to debug - there are more moving parts.
- They involve more of the environment - so you might have to run them multiple times in different environments to exercise the product fully.
- They require a clean(ish) environment to be reliable.

Testing background
Example bug report

Let's consider that more explicitly.

> *User has reported that when loading a file, the application crashed, and they give you the file.*

So you create a little program that runs your application, drags the file to it, and checks that it didn't crash. If it does crash then the test fails.
And your test reports that it fails - you've reproduced the bug and you've got a test that you can now run in the future to check that this hasn't happened again. But where did that problem occur? The test might have given you a backtrace, or maybe some other indication of what happened, but it probably won't be much. Maybe you can add more diagnostics to make sure that you get more information, but you're poking at a large machine with a small stick and trying to see where it's breaking - you generally cannot see inside the machine.
This is just for a small example of loading a file - if the sequence of operations that's required to trigger the problem is longer, the problem may be anywhere in the interactions of those operations. And maybe if you find the cause if the crash, you only fix the specific case that failed and your test passes, but some other very slightly different case would fail but you don't know about it. That depends on how dilligent you are about changing the code.
One other equally important part of running this type of tests is that because there's more of the system involved, they take longer to run. In general you'll want a clean system before each test (so that previous test runs haven't introduced problems themselves), you'll want the application to start, do its test, and then to collect results and clean up, or terminate.

Testing background

System tests are great

- They can test what your users see.
- They can test your feature requirements - the things you tell people they can do.
- They make changing your system safer - you can see the same effect that the user will see.
- They can test what your users report as bugs!

Testing background

System tests are great

Having said some of the problems of testing at this higher level, let's point out why it's good.
There's a big advantage in testing this way because you're testing what the customer sees. That's useful for regression tests, but that's not where it shines. The best thing about testing the things that the user is doing to the product is that you can integrate your feature requirements into your testing. Essentially that means that if you have decided that double clicking a file will load the file into the application, you can put a test together that checks this. This may catch bugs if it turns out that your application is broken, which is great, but also it checks that one of your key requirements for the system is still true.
Why's that useful? Well, those sorts of requirements are what you'll be explaining to your customer in documentation, or in examples. In a command line program, it might be an example command line that you will say "This is how you use our product". So having a test that that example command works is great. I expect there are very few people here who have not had the experience of following a manual and finding that what it says does not work - because the manual is outdated, or the program has become broken in a subtle way. That's a case where a test like this would have been great.
And, of course, like any form of automated testing, you can begin to parameterise the tests - instead of loading just one example file, you can make the test able to be run with any file, and run it multiple times. It'll obviously take longer to run it many times, but you can check many different variants of your files just be adding them to a list that the test will run.

Testing background

Unit and integration tests

Unit testing:

- Keeps the scope of the test to just a small unit - a single function or file.
- Doesn't need specialised environment.
- Limited to just testing the inputs, outputs and insides of the unit.

Integration testing:

- Multiple units tested together.
- May involve parts of the environment, like files and system components.

Testing background

Unit and integration tests

So I've said why system integration testing is less effective in general, and in the introduction I mentioned that interface testing on RISC OS is pretty hard. What can we do about that?

Unit and integration testing are below the level of system integration, and generally need fewer resources to be executed and are simpler to write because of that.

With unit testing, you usually focus on just one small area (the 'unit') and you check that it works. The idea here is that the small section of code has well defined inputs, outputs and conditions of use. Its error cases are easier to manage, or at least to identify. In unit testing, you have no other interactions other than inputs and outputs. In general you never read from disc or communicate with any external libraries. That's not always true in all definitions, but it's the general idea.

With integration testing, you're looking a step above that, where you have multiple units being used together. You might hear people talk about 'module' testing and 'integration' testing separately - where 'module' testing is applying the unit test philosophy

You might hear people say that unit tests must run fast. They're wrong. Unit test describes merely the scope of what you're testing. Generally companies and groups have policies that their unit tests should run fast, and there are good reasons for that, but if you feel you need to write an exhaustive test of a module which will take hours, or days, to run, do it. Maybe you'll run it only a few times, or only as part of your releases, but if that's what you need to make you feel confident in your tests then go for it. The reason that people want unit tests to run fast is that they're your lowest level test and you want them to run regularly and rapidly - each test level, because of its complexity, increases the time it takes to run and the complexity of finding the cause of failures. So having unit tests that run fast means that your confidence in things working is confirmed quickly. You could have integration tests that run faster than your unit tests. It depends on what you're making them do.

Similarly, you may hear that people want their unit tests to be deterministic - that they shouldn't change their behaviour between runs. That's great and it's sensible for automated use, but again the scope of the test doesn't define the method you use for testing, and if you want to use an exploratory test that gives random input to your module, go for it. The reason for avoiding tests that use randomness as part of automation is so that your confidence is high that your test has exercised what you have changed. With randomised testing, your run of the test might find a real problem that someone else introduced weeks ago but which isn't your fault. You may waste time hunting it down because the failure isn't reproducible. But if you're looking for possibilities you never even thought of, random tests are great.

At this point I'll take any questions, before we go on to work through some examples.

3. Test examples

3. Test examples

Test examples

Introducing tests to applications (1)

- Application crashed whilst loading a file - an invalid font was being used.
- Here's the function that gets a font handle...

```
/*************************************************** Gerph *********
 Function:     font_findfont
 Description:  Find a font
 Parameters:   fontname-> the name of the font to find
               xsize = the size (points * 16)
               ysize = the size (points * 16)
 Returns:      font handle, or NULL if could not claim
 ******************************************************************/
font_t font_findfont(const char *fontname, int xsize, int ysize);
```

Test examples

Introducing tests to applications (1)

Let's think back to that bug report example I gave. The application crashed when loading a file. Ok, so how might we have found this with a unit test. Let us say that their document contains a font name that doesn't exist. And that causes something to crash. We don't have any tests for this, so we look at the code and we've got a little library that handles the loading of fonts, and there's a routine that locates a RISC OS font, and returns you a handle for using it, or NULL if it couldn't claim it.

```
/*************************************************** Gerph *********
 Function:     font_findfont
 Description:  Find a font
 Parameters:   fontname-> the name of the font to find
               xsize = the size (points * 16)
               ysize = the size (points * 16)
 Returns:      font handle, or NULL if could not claim
 ******************************************************************/
font_t font_findfont(const char *fontname, int xsize, int ysize);
```

Test examples

Introducing tests to applications (2)

```
void test_find(void)
{
    font_t font;

    /* We should be able to find the font */
    font = font_findfont("Homerton.Medium", 16*16, 16*16);
    assert(font != NULL && "Should be able to find Homerton.Medium");
    font_losefont(font);

    /* We should be able to report a non-existent font */
    font = font_findfont("NonExistent.Font.Name", 16*16, 16*16);
    assert(font == NULL && "Should not be able to find non-existent font");
}


int main(int argc, char *argv[])
{
    test_find();

    return 0;
}
```

Test examples

Introducing tests to applications (2)

So, let's create a small program which checks that this works...

```
void test_find(void)
{
    font_t font;

    /* We should be able to find the font */
    font = font_findfont("Homerton.Medium", 16*16, 16*16);
    assert(font != NULL && "Should be able to find Homerton.Medium");
    font_losefont(font);

    /* We should be able to report a non-existant font */
    font = font_findfont("NonExistant.Font.Name", 16*16, 16*16);
    assert(font == NULL && "Should not be able to find non-existant font");
}


int main(int argc, char *argv[])
{
    test_find();

    return 0;
}
```

It's not a complicated program - it just needs to fail when something has gone wrong so that you know that there's a problem. You now have a test that you can use to prove that in those circumstances your font library is working. You can add it to a list of tests that you can run when you want to be sure that things are working, and if it breaks you'll know that something's wrong. Better still, you've now got somewhere you can put some more tests the next time you find that the font library isn't working right.

You can argue that this isn't a unit test because it's using an external component - the FontManager, and that's fair. In that sense, it's actually a System test, because it's working with things outside your code. Whatever you call it, it's still a useful test to have, and it works quickly, and it should be reliable to confirm that you've got a working interface. At least, for the two cases that we've defined here.

Test examples

Introducing tests to applications (3)

```
/**************************************************** Gerph *********
 Function:         fontfamily_create
 Description:      Create a selection of fonts for a family
 Parameters:       name-> the font name to use
 Returns:          fontfamily pointer, or NULL if cannot allocate.
 ******************************************************************/
fontfamily_t fontfamily_create(const char *name, int xsize, int ysize);
```

Test examples

Introducing tests to applications (3)

So we start to look at the next module up. Let's say that you've got a library `fontfamily` which handles getting the bold and italic variants of a font, and it uses `font_findfont` to do so.

```
/**************************************************** Gerph *********
 Function:         fontfamily_create
 Description:      Create a selection of fonts for a family
 Parameters:       name-> the font name to use
 Returns:          fontfamily pointer, or NULL if cannot allocate.
 ******************************************************************/
fontfamily_t fontfamily_create(const char *name, int xsize, int ysize);
```

Test examples

Introducing tests to applications (4)

```
void test_create(void)
{
    fontfamily_t family;

    /* We should be able to find the font */
    family = fontfamily_create("Homerton.Medium", 16*16, 16*16);
    assert(family != NULL && "Should be able to create Homerton.Medium family");
    fontfamily_destroy(family);

    /* We should be able to report a non-existent font */
    family = fontfamily_create("NonExistent.Font.Name", 16*16, 16*16);
    assert(family == NULL && "Should not be able to find non-existent font family");
}
```

Test examples

Introducing tests to applications (4)

And again write some test code to check that it works...

```
void test_create(void)
{
    fontfamily_t family;

    /* We should be able to find the font */
    family = fontfamily_create("Homerton.Medium", 16*16, 16*16);
    assert(family != NULL && "Should be able to create Homerton.Medium family");
    fontfamily_destroy(family);

    /* We should be able to report a non-existant font */
    family = fontfamily_create("NonExistant.Font.Name", 16*16, 16*16);
    assert(family == NULL && "Should not be able to find non-existant font family");
}
```

If at this point we see that there is a crash during the `fontfamily_create` calls for the non-existant font family, we now have our bug. And we have a test that can reproduce it without any special use of the application. Hopefully from here it's easier to see how to fix it - you know that it's between the boundaries of the `font` and `fontfamily` libraries, and you know the conditions that trigger it.

And of course, if it's not in this library, and that test passes, we move further up the application. At each stage we add more and more functionality and integrations of libraries until we find the issue.

Test examples

What if your code isn't that nice?

What if...

- Your code isn't in isolated chunks?
- Your code isn't able to be split up to do this sort of test?
- Your code mixes interface and functional calls?

Note: There's a great book 'Working with Effectively with Legacy Code' which talks in more detail about some of these things.

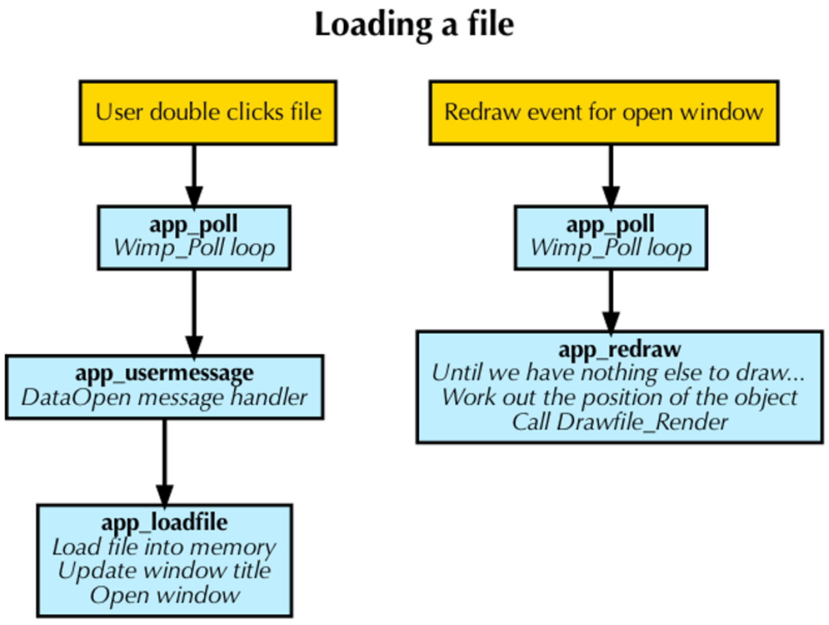
Test examples

What if your code isn't that nice?

In the example application I've discussed, I've made the assumption that parts of the code are broken into small sections in libraries which deal with only their own concerns. That's how I write my own code, usually, and that's because I know I'll want to test it in the future. Or I might want to reuse it in another project. But not everyone works that way, and that makes it hard to do this sort of testing.
The idea of having code split into chunks is really easy to see - you reuse sections where you need them, and you just do the minimum you need to in each part of your code. But it's not always that easy to do in code that hasn't been designed that way.
There's a lot of different ways to attack this problem, and I'm only going to show a couple of small examples because it will be really specific to the type of application you have.

Test examples

Refactoring an application (1)



Test examples

Refactoring an application (1)

Let's say we have an application that loads a Drawfile and can draw it in a window. It really doesn't matter, but it gives us something concrete to think about which is hopefully familiar and self-contained.

DIAGRAM loadcode1

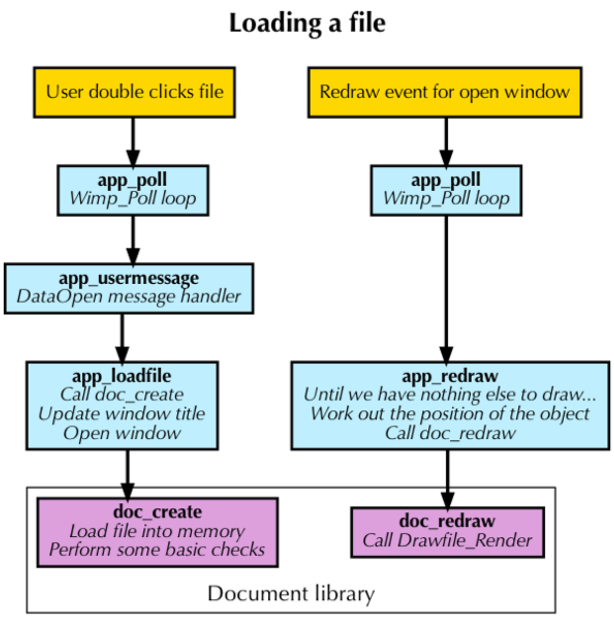
Here's what what we have happening when events occur in the desktop. You can see that the user can double click the application and that passes to the user message handler and then the application loads the file and opens the window. Then, as we know, RISC OS will trigger a redraw event for that window, so the application will catch that and then call the application redraw handler which calls `DrawFile_Render`.

You cannot test this code as it stands without having a desktop - at least not unless you esssentially replace the parts that do wimp operations. Which, actually, is a reasonable way that you could attack it. I'm not sure that I'd recommend it because it's a lot of work but it's got its own benefits and could be a really neat way to do things. However, it would be largely specific to whatever desktop library and interface you were using, so what worked for (say) RISC_OSLib would not be useful for a Toolbox application, or for anyone writing code in BASIC.

Anyhow, how do you attack the application to make it testable. The first thing to do is to identify the parts that are interface-facing, and the parts that are interally facing. Ideally the things that are internally facing have no dependency on the interface, so they can be calved off into their own library, with their own interface. You might find that that's not the case - for example, you might have button which processes the file and needs to update a progress bar in the interface. If that's been implemented with the progress bar updates in the middle of the processing code, it's not going to be a simple refactor to calve that processing function out of your application code. However, there are always ways around that - you could make the processing code operate in steps, and return its progress counter on each call, or you could give the processing code a function to call to update the progress bar.

Test examples

Refactoring an application (2)



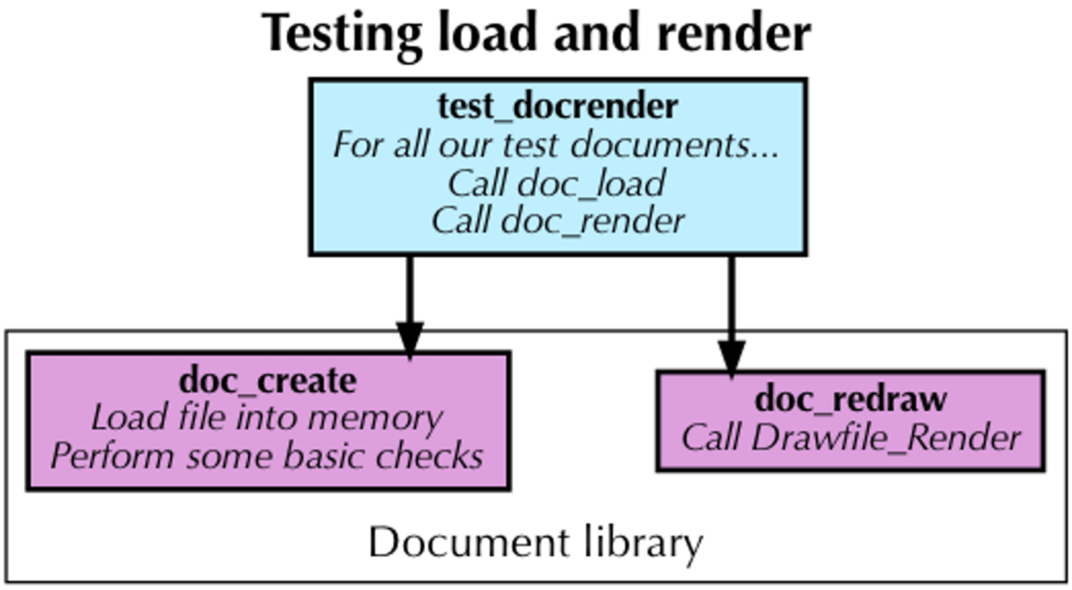
Test examples

Refactoring an application (2)

So your code will now look something like this:
DIAGRAM loadcode2
As you can see, the application code in blue is pretty much the same, but there's a new library for the document handling which you call down to when you need to do something with the document. In this case, the redraw operation is really quite simple, so maybe you think that it's not necessary to make it a separate function in the document library. That's a fair point, but from an efficiency perspective an extra function call won't make much difference, and we're trying to make it easier to test and to manage, which is sometimes in conflict with efficiency. Never forget that the efficiency of a broken program is Zero, it therefore follows that the efficiency of an untested program tends towards Zero.
Of course, having changed the code in this way, you'll have to do all the usual manual testing to be sure that the behaviour hasn't changed. As you're just moving things around, that should be relatively simple to do, and it should have been pretty safe.

Test examples

Refactoring an application (3)



Test examples

Refactoring an application (3)

DIAGRAM loadcode3
Finally you create a small test program that can call your document library to load and render the documents. This looks very simple - it's just running through a collection of test documents and loading and rendering them. The test code I've described here isn't even checking whether the documents are valid or that anything was rendered. Why? Because this is a valid test even without those checks - if anything crashes, you've failed your test. Obviously you'll want to add in more specific checks that things are right when you load the files, and maybe when you render, but knowing that the application doesn't crash when given garbage is really valuable. You could even have a special call to the test code that only supplies it garbage file - Drawfiles that have been truncated, that have objects that are the wrong size, an empty file, a sprite file. If it doesn't crash you're good.
The goal has been met that the code is now testable in a controlled and constrained manner, where it wasn't before. You can do other checks like saving screenshots to check that the documents being rendered are correct, if you want, or check that the bounds that it's rendered to haven't been exceeded, or all sorts of other things, but you have a starting point to work from that allows this.
Plus, you've also now got the possibility of a command line tool, and because your redraw code has been moved out of the redraw loop, that means that printing is a doddle (because printing is just the same as your redraw code, but inside a print loop).
I've over-simplified the problem case here, obviously.

Test examples

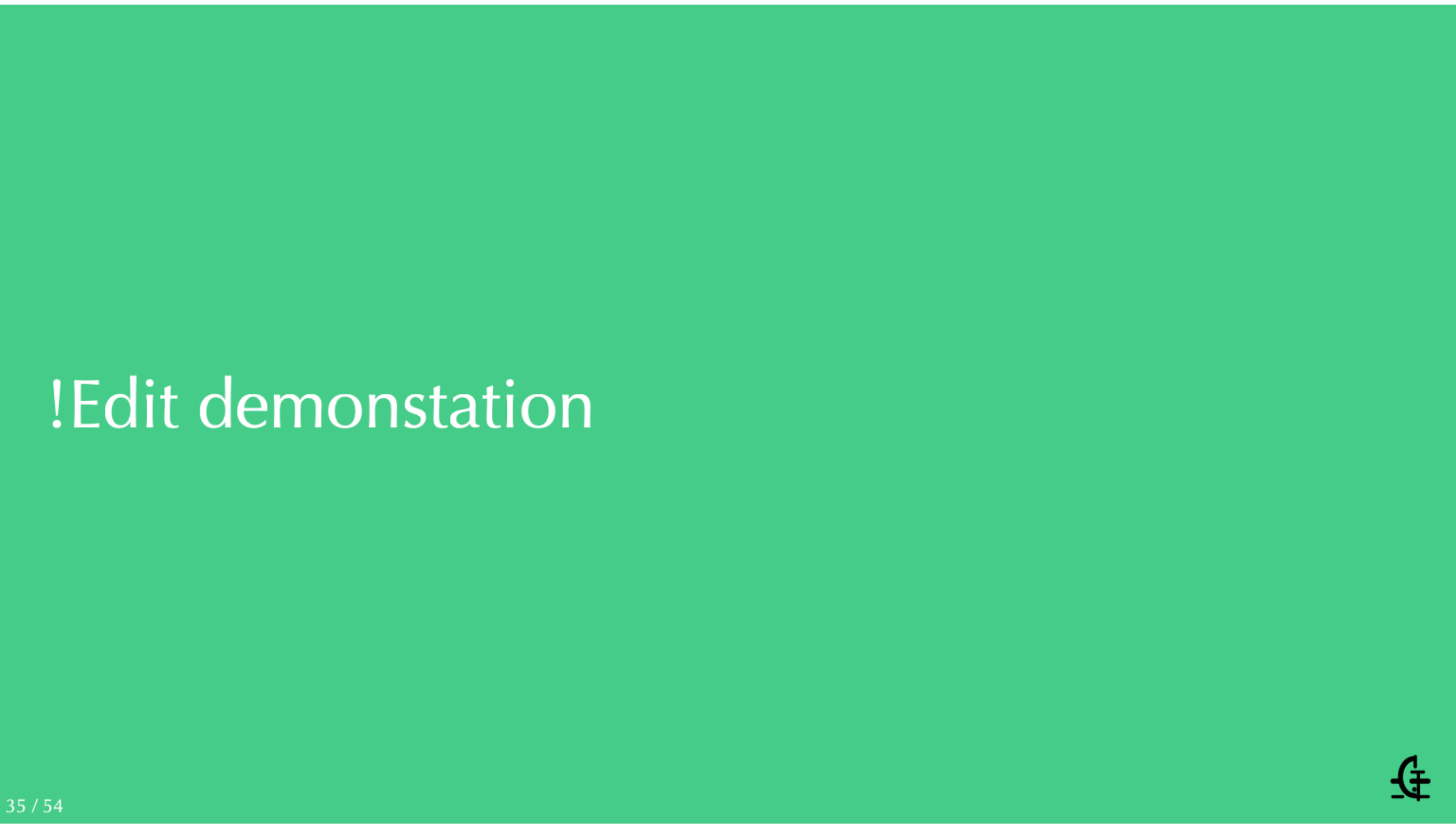
Concrete example

Is that a contrived example?
Let's have a look at some real code and I'll try to explain my thoughts...

Test examples

Concrete example

Is that an entirely contrived example? Well, it's possible that your code never looks like that. Let's have a look at an example of some real code that mixes the processing logic into the interface logic.



!Edit demonstation

!Edit demonstration

DEMO TIME

```
cd ~/projects/RO/riscos/Sources/Apps/Edit/
subl .
```

- Load up Edit, c/message, message_bkg_events around like 303 onwards.

This is the message handler for the Edit's TaskWindow operations. Let's have a look at what we might carve out to make the code testable.

So we have the `newtask` message, which is sent when a new taskwindow is being created. If we follow this to message_taskwindow, we see that this sets up some of Edit's `txt` handles (how it represents a window) and then starts a command in `message_starttask`. So `message_starttask` is probably able to be carved off, because it has no other dependencies on the interface. `message_taskwindow` is a bit more mixed up, with txt operations which are the window view, and the event library operations. It'd be nice if those could be split up more so that they don't mix up the different types of operations. The `txt` wouldn't be a simple library to split off, but it could at least be made easier if it didn't mix up the handling so much.

Anyhow, that was the `newtask` message; we move on to the `output` message, and this is doing a lot more work than we want in there. Firstly there's some processing of the message data. That's entirely specific to the taskwindow, so that should probably move into its own function. Then we read the caret... let's come back to that.

Next we insert the data into the taskwindow `txt` window and send a suspend message if we were out of memory. This could be better handled as a function call that asks the library code to add the data to the `txt`, and then returns a flag indicating whether it was successful or not. The message to suspend could then be sent based on that function's result.

There's a check whether the output is linked or not, which would probably go inside the function that handles the data insertion. Then finally there's some checks on whether the caret was in the window to move it with the output. So this caret check couple probably go around the taskwindow data processing, in the message handler, because it interacts with the Wimp. This message handler essentially reduces to the caret code, surrounding a call to the refactored code in a function.

Next message is `morio`, which just sets some state for the taskwindow - a no brainer, it goes into a new function.

The last one I'll look at is `ego`, which is essentially saying `I'm Mr Taskwindow, look at me!`. This is the point at which a taskwindow really is alive, so it would want to be refactored into a function as well.

It's certain that this isn't all you need to be able to test Edit's handling of TaskWindows - it's still highly dependant on the `txt` objects, which in turn relies on a lot of other things. But by beginning to split it off, it becomes easier to test some of the code. At the very least, that code which handles the trimming of control characters would be able to be isolated and could be tested almost immediately.

Not that I think that code is wrong, but ... as I look at it now, I see that it tries to step through the number of characters given in the first word of the data block, which

Test examples

System tests for a module

- System testing modules is quite possible - with care.
- The build service was already building ErrorCancel.
- Adding a simple test of it working is pretty easy...
- ... so let's see how it's done.

Test examples

System tests for a module

Following a little discussion on the forums, I returned to one of the examples that I created for the build service. I had wanted to create some examples that show how easy it was to feed source code to the build service and get it to build and return you a binary. One of those examples was Rick Murray's `ErrorCancel` module. It's a module that does a simple task - it spots that an error box has been shown and then after 5 seconds it presses Escape to cancel it.
As I was talking about testing, I said to myself "let's actually do something to show how easy this can be". If a picture is worth a thousand words, what's a demonstration worth?
Let's have a look at the test code I wrote, and oh my gosh is it simple...

ErrorCancel demonstration

ErrorCancel demonstration

DEMO

- In Terminalcd ~/projects/RO/ricks-errorcancel git checkout add-simple-test less TestError,fd1

```
REM >TestError

REM Rudimentary test that the error is cancelled
PRINT "Testing error box cancelling - if this hangs, we failed"
start = TIME
SYS "Wimp_ReportError", "0000My Error Message", %11, "Myapp" TO ,response
REM If we reached here this is a success, because it cancelled the error box without
REM any user interaction.
REM In the CI, this will hang.
elapsed = TIME-start

REM Report what we got
PRINT "Got response: ";response
PRINT "Took ";elapsed / 100; " seconds"

REM We can check these are what we expect.
IF response <> 2 THEN ERROR EXT 0, "Should have got response 2 (cancel)"
IF elapsed < 500 THEN ERROR EXT 0, "Terminated too quickly (should be about 5 seconds)"
IF elapsed > 550 THEN ERROR EXT 0, "Terminated too slowly (should be about 5 seconds)"
PRINT "Success"
```

What we do here is we just call `Wimp_ReportError` and try to get a response. We print the response and the time it took out, and then we check that it's about the right sort of duration. Remember I said about not liking tests that are non-deterministic... well that 5.5 seconds timeout is one place where things might go wrong. But that's ok, because you can increase it if you like, OR you can just accept that it might be a bit flakey, or you could remove that check entirely. The one that really matters is the one that says it mustn't be less than 5 seconds, 'cos then that'd mean it wasn't doing what it was meant to do.

Let's run that test code and see that it's working.

```
pyrodev --common --gos
Rmload rm.errorcancel
testerror
```

Wait for it... 5 seconds.

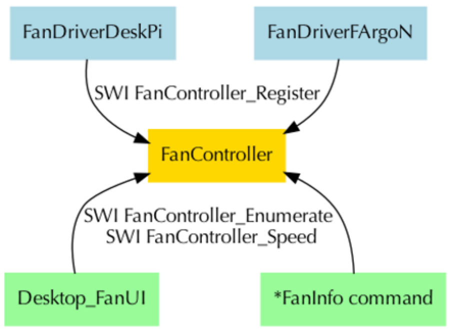
Test examples

Unit and integration tests for a module

- Unit and integration testing are better suited to modules.
  - It's safer.
  - They can be tested on small parts.

My FanController has some tests to verify it works.

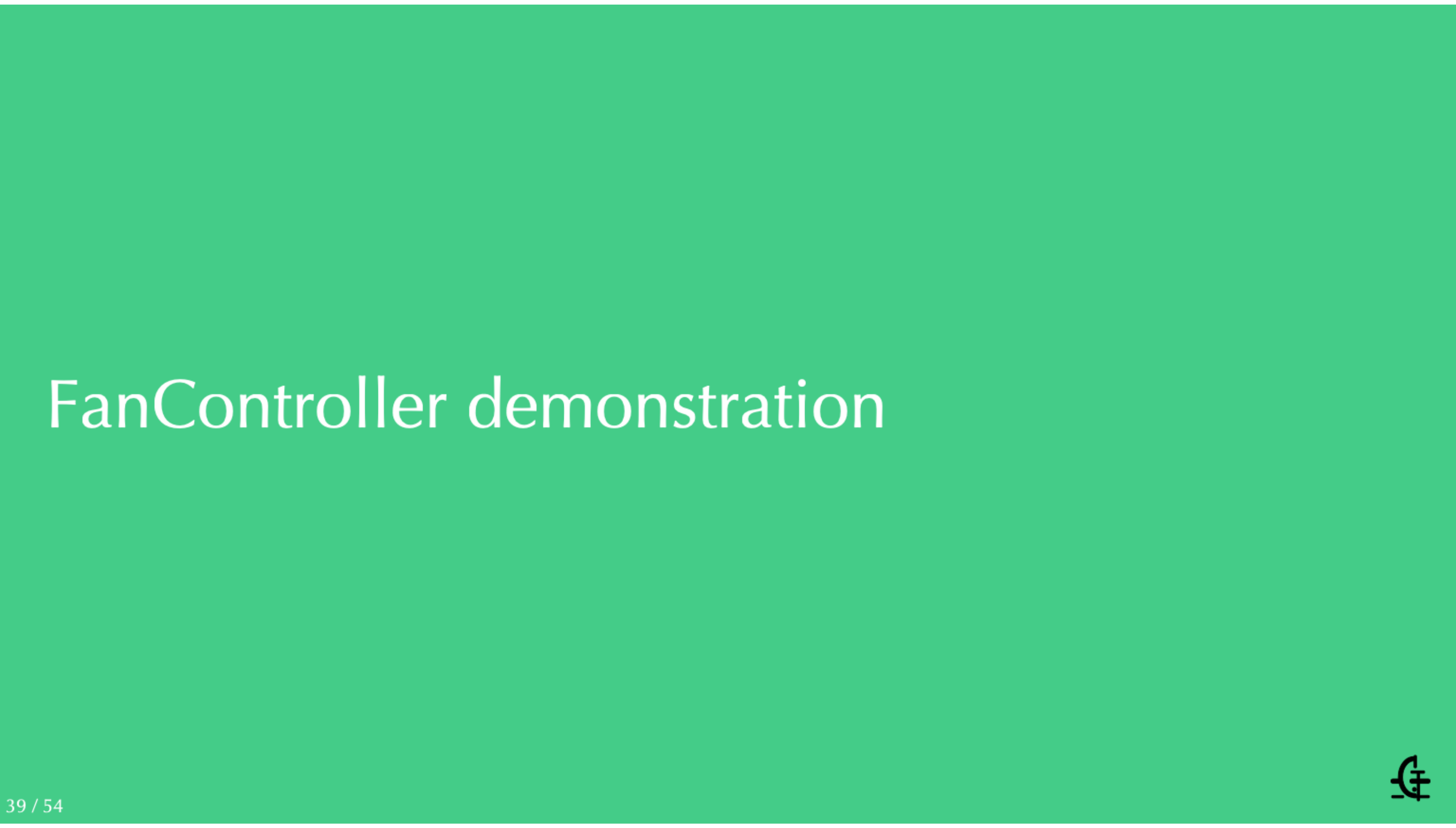
- It isn't a complex module, but even simple modules need testing.



Test examples

Unit and integration tests for a module

I've shown some testing of the effects and interface of the module with ErrorCancel - system testing. So let's move on to how you might do unit and integration testing of a module.
Modules are harder to test in general because they run in priviledged mode, and that means that you're always running the risk of blowing the system away. So don't test your code like that. That's the thing about unit and integration testing - you're testing things in isolation. And if you've written your code in a modular way, this process is pretty simple.
So I'm going to describe the testing that I've created for FanController. FanController is a dispatcher for registered FanDrivers. Drivers register with it, FanController announces to the world, the world talks to FanController, FanController talks to the drivers.
The module is pretty simple - and so it's actually pretty easy to test.


FanController demonstration

39 / 54

FanController demonstration

DEMO

- In a new terminal open the FanController source cd ~/projects/RO/Fancontroller subl .
- Sublime Text FanController window.
- Open the directories as you talk about them

This is the FanController repository. It's got the FanController module itself. A dummy driver module, which is intended for people to build upon to make their own fan drivers. There's some example code. And there's the python implementation from Pyromaniac. The pyromaniac version was written first, to provide a way to quickly iterate the design, but it's the native version that I'm interested in here.

- Open the MakefileTestStr,fe1

Let's start out with the string library. This is the makefile, which is defined as a LibraryCommand - these build AIF files from C and assembler. We have a dedicated test for the string code, and the string code itself. And we build the library with Fortify to check the memory allocations and frees are done properly.

- Open the h/str file.

So, this is the string utilities header - it doesn't do much. It's literally just a few functions that we'll use in the rest of the code. So they ought to work. If they're so simple, should you bother testing them? Depends really; if you assume that they're fine then maybe you'll fall over them later when you least expect it. On the other hand, it might just be wasting time you could be using on testing something more important.
In any case, it makes for an incredibly simple example of how you can write some tests, so it's a great plan.

- Open the c/str_test.

There's only a couple of functions in the string test, and they're not exhaustive. There are loads of other things you could do. But here I'm mostly showing what you can do with very little code.
What do we have? We've got a main function that calls both our test functions. The string duplicator checks that we allocate memory, and that when given a `NULL` pointer we don't fail. Pretty simple stuff. At the point I wrote this code, I found that the `strdup(NULL)` actually crashed, despite it being expected that it would return NULL. Simple things can have overlooked bugs.
And the `strndup` code does the same sort of thing. I'm being lazy here in that I'm using `assert` checks to just fail where we see a problem. Some test systems will keep going, accumulating failures and showing reports like you want. It's quick, it's easy, and it gets you showing them as they're found. But I'm being lazy. The assertion will cause the program to fail, and will therefore terminate the testing.

Test examples

Operating system tests (1)

Ways of writing tests (there are many more, and you can mix them):

- *Manual tests*: Human checks the behaviour is what they expect.
- *Crash-based testing*: Fails only when the component crashes.
- *Assertion based testing*: Checks for specific features of the tests.
- *Expectation testing*: Lazy testing of the test's output.

Test examples

Operating system tests (1)

The testing I've discussed and shown so far has used a couple of styles of checking that the test has worked:

- I talked about manual testing, where a human checks the behaviour is what they expect.
- Crash-based testing, where you do no explicit checking of the behaviour other than that the product didn't crash.
- Assertion based testing, where you check for specific features of the tests.

There are a number of other ways of testing, but I'm going to discuss one of the ways that I chose to write most of the tests in Pyromaniac.
I do have some unit tests that use assertion testing, in the Python code, but a lot of the tests in Pyromaniac are system tests which use the whole product and check its outputs against a set of expectations. In this form of testing, you run your program with some inputs and you check the output - whether it be the text output stream, or files - against some master that you know to be correct. If it differs, your test fails.
This form of testing is relies heavily on the output you are checking being representative of what you are testing, and repeatable. Sometimes these tests are called 'gold master' tests, and they're an ideal way to write tests for legacy systems - that is anything that doesn't have any other form of tests. Because this form of test captures the behaviour of the system, they're likely to break if you change that behaviour, which is good if you want to know when you deviate from known good behaviour.

Test examples

Operating system tests (2)

RISC OS Pyromaniac's scripted tests look like this:

```
Group: OS_Write SWIs
Expect: expect/core/hello_world

Test: OS_WriteS
Command: $TOOL bin/hello_world_s

Test: OS_Write0
Command: $TOOL bin/hello_world
```

Test examples

Operating system tests (2)

In Pyromaniac there is a scripted test system which says how to run the tests, and what expectations should be met.

```
Group: OS_Write SWIs
Expect: expect/core/hello_world

Test: OS_WriteS
Command: $TOOL bin/hello_world_s

Test: OS_Write0
Command: $TOOL bin/hello_world
```

This is an example of a test specification, showing two tests that will be run. I created a test harness parser which takes these specifications and runs through them, reporting the results. This description says that you will run the tool, with the command `bin/hello_world_s` for the first test, and `bin/hello_world` for the second one. Both of them are expected to produce the same output, as the expectation is defined within the group.

Of course, the actual test specification scripts in Pyromaniac have much more in them.

Test examples

Operating system tests (3)

The `OS_WriteS` test looks like this:

```
        AREA    |Test$$Code|, CODE

        GET     hdr.swis

hello_world     ROUT
        SWI     OS_WriteS
        = "Hello world", 0
        ALIGN
        SWI     OS_NewLine
        MOV     pc, lr

        END
```

Test examples

Operating system tests (3)

What is the test itself? The `OS_WriteS` test looks like this:

```
        AREA    |Test$$Code|, CODE

        GET     hdr.swis

hello_world     ROUT
        SWI     OS_WriteS
        = "Hello world", 0
        ALIGN
        SWI     OS_NewLine
        MOV     pc, lr

        END
```

That's all the code does - it writes "Hello world" and then returns. The test harness will compare this to a file on disc and if it's different it'll fail. So it confirms that the `OS_WriteS` call is doing what it needs.

Test examples

Operating system tests (4)

The `OS_Write0` test looks like this:

```
        AREA    |Test$$Code|, CODE

        GET     hdr.swis

hello_world     ROUT
        ADR     r0, message
        SWI     OS_Write0
        ADR     r1, end
        CMP     r0, r1
        BNE     bad_return
        SWI     OS_NewLine
        MOV     pc, lr

bad_return
        ADR     r0, return_wrong
        SWI     OS_GenerateError

message = "Hello world", 0
end

return_wrong
        ALIGN
        DCD     1
        = "R0 on return from OS_Write0 was not correctly set to the terminator", 0

        END
```

Test examples

Operating system tests (4)

The `OS_Write0` test code is a little more involved:

```
        AREA    |Test$$Code|, CODE

        GET     hdr.swis

hello_world     ROUT
        ADR     r0, message
        SWI     OS_Write0
        ADR     r1, end
        CMP     r0, r1
        BNE     bad_return
        SWI     OS_NewLine
        MOV     pc, lr

bad_return
        ADR     r0, return_wrong
        SWI     OS_GenerateError

message = "Hello world", 0
end

return_wrong
        ALIGN
        DCD     1
        = "R0 on return from OS_Write0 was not correctly set to the terminator", 0

        END
```

Which you can see is pretty simple - this has the one assertion check that we need in this code, to test that the value returned in R0 is actually after the end of the string printed.
Shall we see one working?

RISC OS tests demonstration

RISC OS tests demonstration

DEMO

- In a terminal:cd ~/projects/RO/pyromaniac

I have a makefile which will run the unit tests and then can run an arbitrary test script, so let's see it run.

```
make tests TEST=core
```

As you can see we run all the tests in the script, grouped together, and then report the results. It's not all that exciting to watch, but when it completes you can see that we passed all the tests. Yay.

And at the end we even get an XML report of the test results:

```
less artifacts/results.xml
```

That's JUnit XML format, which is a semi-standardised way of reporting test results that systems like Jenkins, GitHub, GitLab and others all understand, and can give you nice tables of. Not too relevant to writing tests, but great when you have automated reporting on your test results.

Ok, so that's how I run tests in Pyromaniac. Does that help anyone else? Well... Last year I made some of the tests that I use with Pyromaniac available for others on GitHub - in the gerph/riscos-tests repository. The idea of publishing the tests repository was that it would show people that you can write tests, and that they don't have to be complex. And maybe someone would take them and use them to test RISC OS. Or tell me that they're broken.

Let's have a look at them.

```
cd ~/projects/RO/riscos-tests
ls
cat !RunTests,feb
```

There's a small test runner here so that we can run the tests:

```
| Run the tests
If "%*0" = "" Then Set Test$Args --show-command --show-output /bin.Capture testcode Else Set Test$Args %*0
If "%*1" = "" AND "%0" <> "" Then Set Test$Args --show-command --show-output --script "%0" /bin.Capture testcode
set Test$Dir <Obey$dir>

do perl testcode.test/pl <Test$Args>
Dir <Test$Dir>
```

It largely just sets up the command line so that I can run specific test suites easily. It's not polished, and I genuinely don't care that much about it - I have literally just set it

Test examples

Operating system tests (5)

- RISC OS still sucks for testing.
- Make the amount of things that you test small.
- Work your way up to system testing.
- Build service may help with that - when things die there, you don't lose all your work.

Test examples

Operating system tests (5)

Of course, there's a problem with this. RISC OS sucks for testing. It's too easy for RISC OS itself to be blown away by some rogue code. And when you're running tests, ALL code is rogue. Whilst putting together the tests above I managed to kill RISC OS 3.7 stone dead a few times. And then tried it with RISC OS 5 and Select, and it was pretty fatal in both of them as well. As in being forced to reboot immediately. Running it in Pyromaniac showed that I was getting zero page overwritten.
That wasn't whilst running that tests - that was the perl harness code killing things. That /is/ a reason I wrote Pyromaniac, after all.
Which is why it's vitally important to be able to test your code before you put it into RISC OS properly - which the RISC OS build service can do without you having to worry quite so much about killing the machine you're working on, or by leaving it up to the remote machine to do it whilst you get on with other things.

Test examples

Development practices to make it easier (1)

Make it easier to test:

- Design new features in a modular manner.
  - Write unit or integration tests for them.
- Separate out code sections that don't need to be integrated
  - Plug-ins, or external tools can isolate code and make it easier to develop and test.

The RISC OS Select Kernel has many modules which perform previously integrated functions:

- Smaller Kernel code, less complex.
- Easier to change extracted modules.
- Easier to test extracted modules.
- New features don't require a new Kernel.
- Reimplementation is easier.

Test examples

Development practices to make it easier (1)

There are things you can do to make things easier on you. I've already explained how you can carve up an application to make it possible to test the boundaries of the code. This technique applies to adding new code as well. When you create a piece of new functionality to go into your existing code, you can start out by making it a modular chunk of code, that just does what you need. And then you integrate that code into the rest of the program you're working on - after having implemented it and writing the tests for it. Doing the development this way means that you know that what you're adding should work. Obviously this only works for things that you've trying to add as distinct features.

But it does also mean that your development time is much shorter, because you're working on just a small part of the code and not the whole program.

Another thing you can do is to identify code that has no place in your program and rip it out. This isn't always possible, but if it's possible to call another program to do the complex work for you, then it simplifies each of them. That might not seem so common, but it's actually not that bad - extracting some chunk of code and making a second program that you communicate with, or a little tool that does the work isn't so hard. And then you can apply the testing just to that second program. And because it's smaller, it's easier to reason about and express what you're testing.

That might not be right for you, but it's a particular strategy I applied to the RISC OS Kernel, under RISC OS Select. ISTR that there were generally negative comments from people about the split of the Kernel, but it was all about confidence, speed of development and reliability. For anyone that hasn't seen that side of things, under Select many of the functions which had no need to be in the Kernel were ripped out and moved to their own modules. These separate modules could then be developed and tested entirely independant of the Kernel. It meant that...

- The Kernel got smaller and less complex.
- The difficulty changing the extracted modules became much lower.
- The ability to test those extracted modules was hugely improved.
- New features could be added to the extracted modules without influencing the Kernel.
- Those extracted modules could be replaced with other implementations when necessary - making transitions to C easier, when that came up.

Why does this matter for testing? Well, obviously it's easier to test a small, targetted component, than a monolithic one which is complex and liable to break at the slightest provocation. How did I achieve this split of the kernel functions? Well... exactly as I described for the applications. Identify the interface points, calve them out, move the code to another module and wire that module up to claim whatever OS SWIs it needed, or claim vectors, or whatever.

Test examples

Development practices to make it easier (2)

- Extracting code out of the Kernel isn't always the right choice.
- Writing assembler is rarely the right choice.
- Writing C is far nicer - more maintainable, more readable, and more testable.
- Integrating C code into assembler isn't actually that hard.

Test examples

Development practices to make it easier (2)

Sometimes, though, separating out the functions is not easy, but you still want to make your development easier and testable - and you've got a bunch of assembler you want to extend.
So write the code in C. You implement your new feature in C, and you test it and then you integrate that C code into the horrible assembler. Why do this? Because C is easier to manage, easier to reason about, and easier to test, and because dear god I don't want to write any more assembler ever. Plus, it means that once it's 'just' a matter of converting the rest of the assembler to C, and you're not having to re-do work.

Testable Kernel code demonstration

Testable Kernel code demonstration

DEMO

So let's have a look at an example of this. It won't be very exciting, but this talk is about showing how to do testing...

```
cd ~/home/RO/riscos/Sources/Kernel
subl .
```

So, this is the Kernel source. The assembler directory will look pretty familiar, but I'm going to show a little bit of the module header check code.

- Open c/checkmodhead

This is the code that was originally in assembler, and needed extending to be more resilient and to handle new flags. The original code was buried in the s.modhand file, and - as I've said - I don't really want to write assembler if I can help it. So this is a replacement.

We have some data structures declared, just as you would expect in C, and constants for some of the magic values that the modules use. There's a return enumeration that describes what the module errors might be. We can use conditional code to explain our checks using the preprocessor - this `MUCH_DEBUG` option would only be used outside of the Kernel, but it explains what values are being checked as we hit them. Nothing too special, but for visual inspection it's handy.

And this `check_module` function is the actual interface. It returns the enumeration that you saw above, and it's given the module's parameters - a pointer to it, its size and what bitness we're checking it for. Each of the header's offsets is checked and returns an error code if there's something wrong. And so it goes on. At the end of the file we have a section of code that's only used in the test, which is able to either check the entire system's modules or a specific module.

That test code can then be used to check whether things are being reported properly. In a modern environment I'd have a number of different correct or broken module files alongside the code which could be given to it to check that it was failing or working. But I don't have that here.

- Open s/modhand; search for CheckHeader as a symbol

And this is where it's called from. We marshal the registers into the form that the C code likes, and we use the CCall macro to do the call to the code. The result is then turned into an error block, and V set as per the expectation, or returned with V clear. Very little faffing in assembler.

- Back to the terminalcd TestModHeader less Makefile

This is the code that we have for building the test code... which we can do...

```
riscos-amu BUILD32=1
```

and then we can run it:

Test examples

Development practices to make it easier (3)

- Writing things in C only helps part way - you still need to integrate it.
- But the integration is usually trivial.
- And C is faster to develop, testable, and more maintainable.
- It lets you focus on algorithms instead of register assignments.
- RISC OS Select had a few assembler modules with C code in:
  - Filer
  - Wimp
  - CLIV
  - FileSwitch

Test examples

Development practices to make it easier (3)

So that's the way that you can write new code in an assembler module, but in C, and make it testable. You may still have problems with the integration, but those will only happen initially, and you can get back to making the feature work. Plus, of course, you're not sitting there waiting for the Kernel to build, patching the Kernel over a ROM image, firing it up, seeing whether it works and repeating.
Of course there are other examples where I'd done this - there's 9 implementations in C in the Kernel, at least 5 in the Wimp, and another 5 in the Filer. I think FileSwitch had some in one branch, too.
You remember those modules that I mentioned that extracted from the Kernel? Well one of them, CLIV, handles the CLI vector. In Ursula, module handling was updated to add hashed command lookups, which was quite entwined with the module code. Well. That sucks. Module code should handle modules, not be faffing around with command tables. So when the CLIV module was extracted, all that code that was heavily tied up with the module handling code was lost. In the process, the module code got simpler, which is cool. And for CLIV, I re-wrote the hashing algorithms from scratch in C. And there's test code that sits alongside it to check that the hashing works.
That's both the examples I've just discussed - a much easier to manage module, because it's outside the Kernel, and using C code to make the testing easier. Of course you could test against assembler, but - as if I haven't said it enough in this talk - life's too short to waste on writing assembler.
I made these changes to be easier on myself, because ...

- I don't like writing assembler
- I don't like waiting for ROM builds
- and when you're working on the entire operating system, the fewer magical interactions that are present in any part of the system, the better you can remember and reason about it.

Does this sort of stuff matter in an open source world? Yes, so much more. Because not everyone who might contribute will know what's safe. And not everyone that reviews things will know all the ins and outs of the system. And because the very few people who do know those magical interactions don't want to spent their time baby sitting each little change. I'd say that by separating components, and making it easier for people to make changes, without having that knowledge in their heads, that's got to be a bonus. And if you've got tests that prove that those changes work, then the amount of time needed reviewing every detail's theoretical effects is less.

Test examples
Collaboration

With open source work, collaboration on projects helps testing:

- Extra eyes improve code:
    - It encourages writing good code.
    - It encourages proving that your code works - through tests or otherwise.
    - Others can suggest ways of exercising the code that you won't have thought of.
- It allows for automated testing in the background.

Test examples
Collaboration

The intelligence community have a phrase - "Trust, but verify" - which is intended to avoid being accusatory about things. In those environments, you don't know who might be compromised - either with their knowledge or without - and if you were to treat some people above others, you might miss something.
Trust that other people put their code together well, but check it - with a review of the code, but better still, with automated tests that show that what they said is what it does. "It's not that I don't trust your code, but we test everything" should be the way that you think, and nobody gets a wave through without proving what they're doing.
Maybe that attitude's a little extreme, so maybe this is better... In more common use you may have heard people say "Pictures, or it didn't happen" - for engineers, this is very similar. If you didn't tested, your code isn't going in. You provide proof that you did it right, and made all efforts, and you may pass.
This sort of attitude towards collaborative working is intended to keep software in a good state, and ensure that the effects of changes are understood. And if, on every change, you provide some improvement to the manner in which things are tested, you make inroads into feeling confident about the product as a whole. If you feel confident about the product, and confident that its tests are sufficient to catch problems, then you'll feel more confident with Alice and Bob making changes.
How do you ensure that 3rd parties can show that their code changes work when your product doesn't have any tests? Well, there are ways around that, but it's tedious. Requiring that reviews of the code include the built product, with output from it changing - a command line output, a picture of it running, a video of the effects - that can help. But automated testing to show that things are working is so much better.

4. Conclusion

4. Conclusion

Conclusion
Summary

I have discussed...

- Theory and scopes of testing.
- Theoretical refactor of an application.
- System testing of a module
- Unit and integration testing of a module.
- Testing code that doesn't lend itself to testing.

Conclusion
Summary

Right, so I've talked about a number of things...

- I've explained some of the theory and scopes.
- I've given a worked theoretical example of refactoring part of a application.
- I've shown an example of adding some simple system testing to a mdoule, and using the build service to exercise it.
- I've shown an example of adding simple unit and integration tests to a module.
- I've shown an example of expectation testing used for system tests of Pyromaniac and RISC OS.
- I've shown some ways that code which doesn't lend itself to testing in place can be tested.

Conclusion

What now?

It's up to you but...

- Automated testing will honestly help you.
- Automated testing is what all the cool kids do.
- I've shown that you too can do it.

And ask yourself...

- How confident you feel using software that doesn't have any tests?
- What can you do about that?

Conclusion

What now?

What should you take away from this?
Well, that's up to you.
My view...
Avoiding doing testing is setting yourself up for a fall. Avoiding doing automated testing is making work for yourself. Not having sufficient testing is going to bite you or your customers in the ass. So do some.
My specific view is that UI testing is hard, and so energy is better spent on lower level unit, integration, and system testing which doesn't involve the UI. But there are always ways to do things, though, so don't be put off that it's hard, or that there aren't libraries available. If you need something, build it. Don't wait for someone else to do it, whilst moaning that it's hard. Actually that's good advice for non-testing things too.
Now, I get to do a little rant... why the heck is the RISC OS source still in the state that it's in, with no tests? As RISC OS is now over 30 years old, you might like to think that at least a few actual tests would exist? And that at least the core system calls would be exercised?
Finally, testing is all about giving you confidence. How confident do I feel about the testing I've got in RISC OS Pyromaniac? I'm pretty confident that there's not enough :-)
That's the end of the talk, thank you very much.

5. Questions

Resources: https://presentation.riscos.online/testing/

5. Questions

Resources: https://presentation.riscos.online/testing/