Software testing on RISC OS

Some methods and mechanisms

Gerph, May 2021

0. Introduction

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

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

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? (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...

1. The 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 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.

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

- 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.

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.

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

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? (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

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

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

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

Types of testing

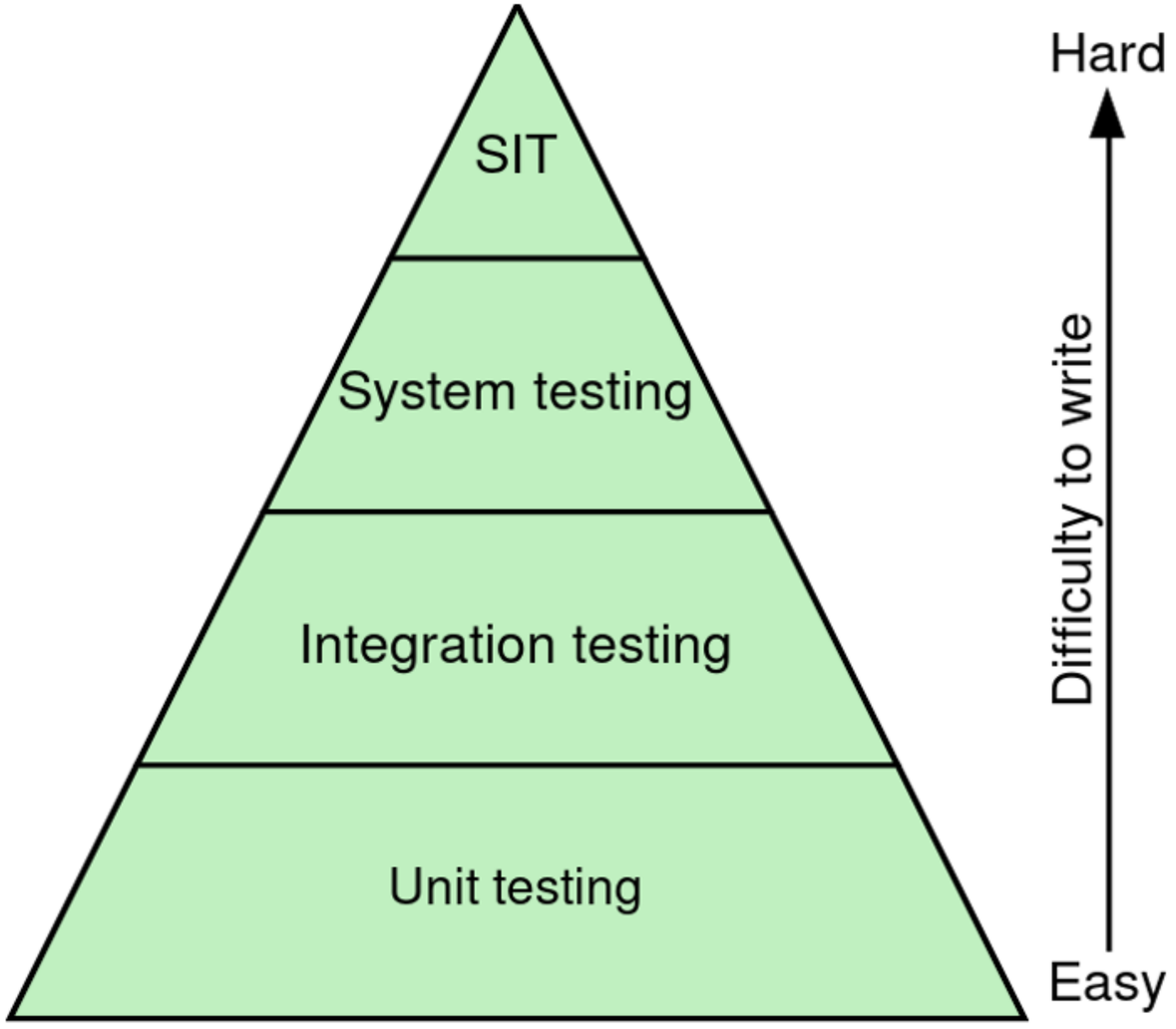
Commonly discussed testing scopes:

- *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



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

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

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

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.

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 (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 (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 (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

What if your code isn't that nice?

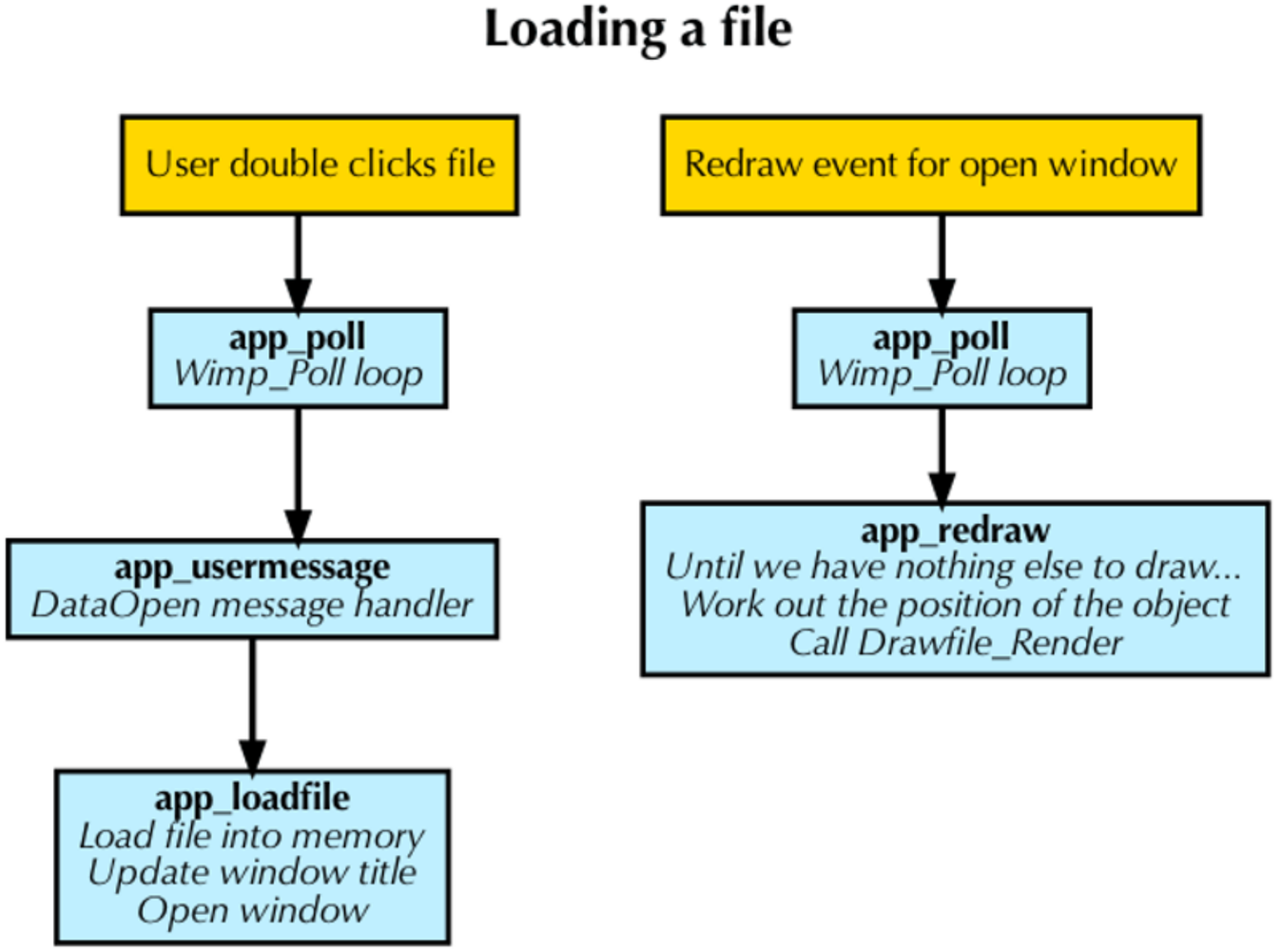
What if...

- 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.

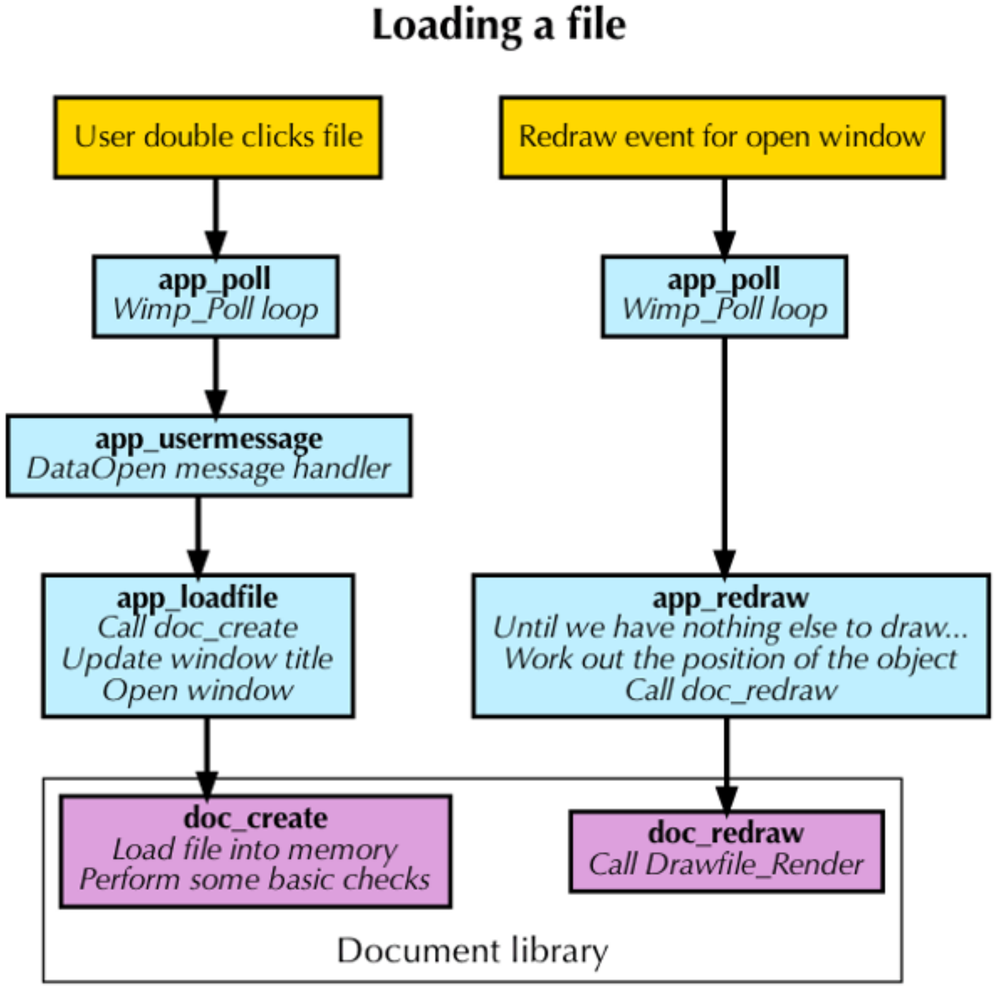
Test examples

Refactoring an application (1)



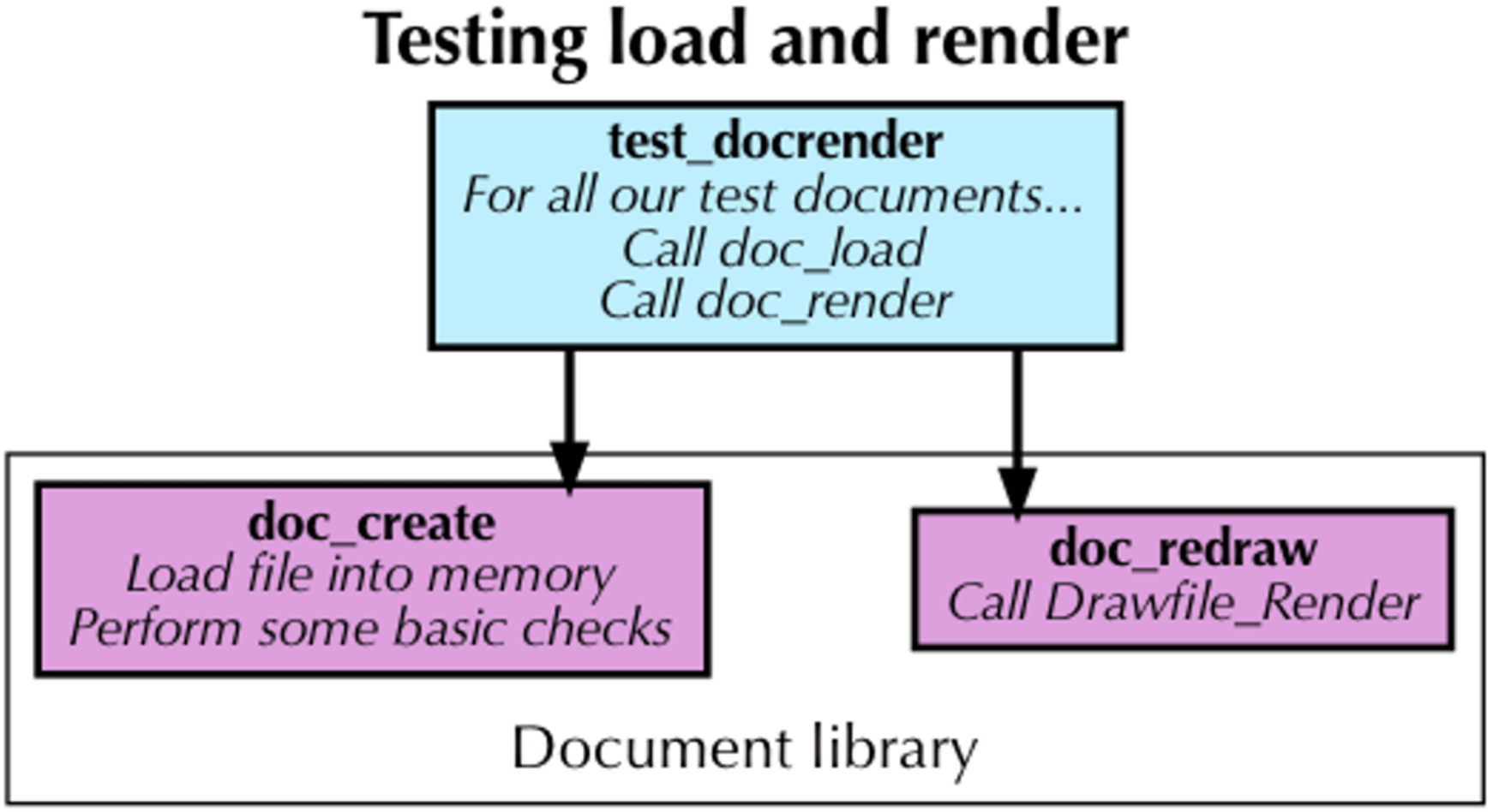
Test examples

Refactoring an application (2)



Test examples

Refactoring an application (3)



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...

!Edit demonstation

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.

ErrorCancel demonstration

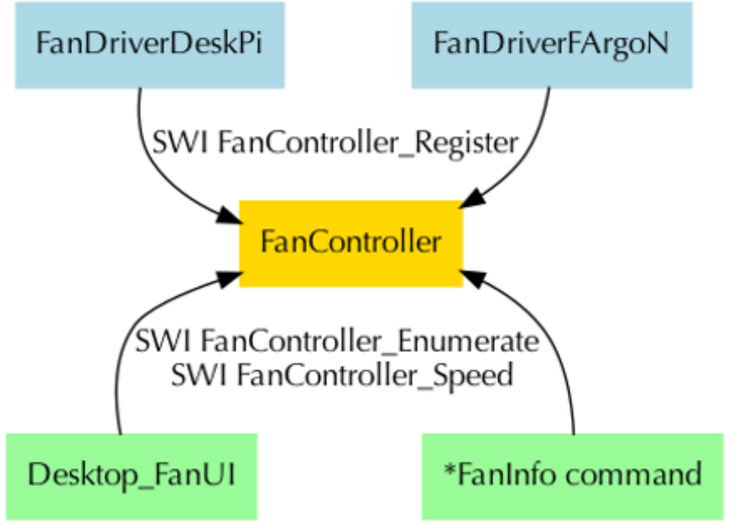
Test examples

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.



FanController demonstration

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 (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 (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 (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
```


RISC OS tests demonstration

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

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 (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.

Testable Kernel code demonstration

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

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.

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

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?

5. Questions

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