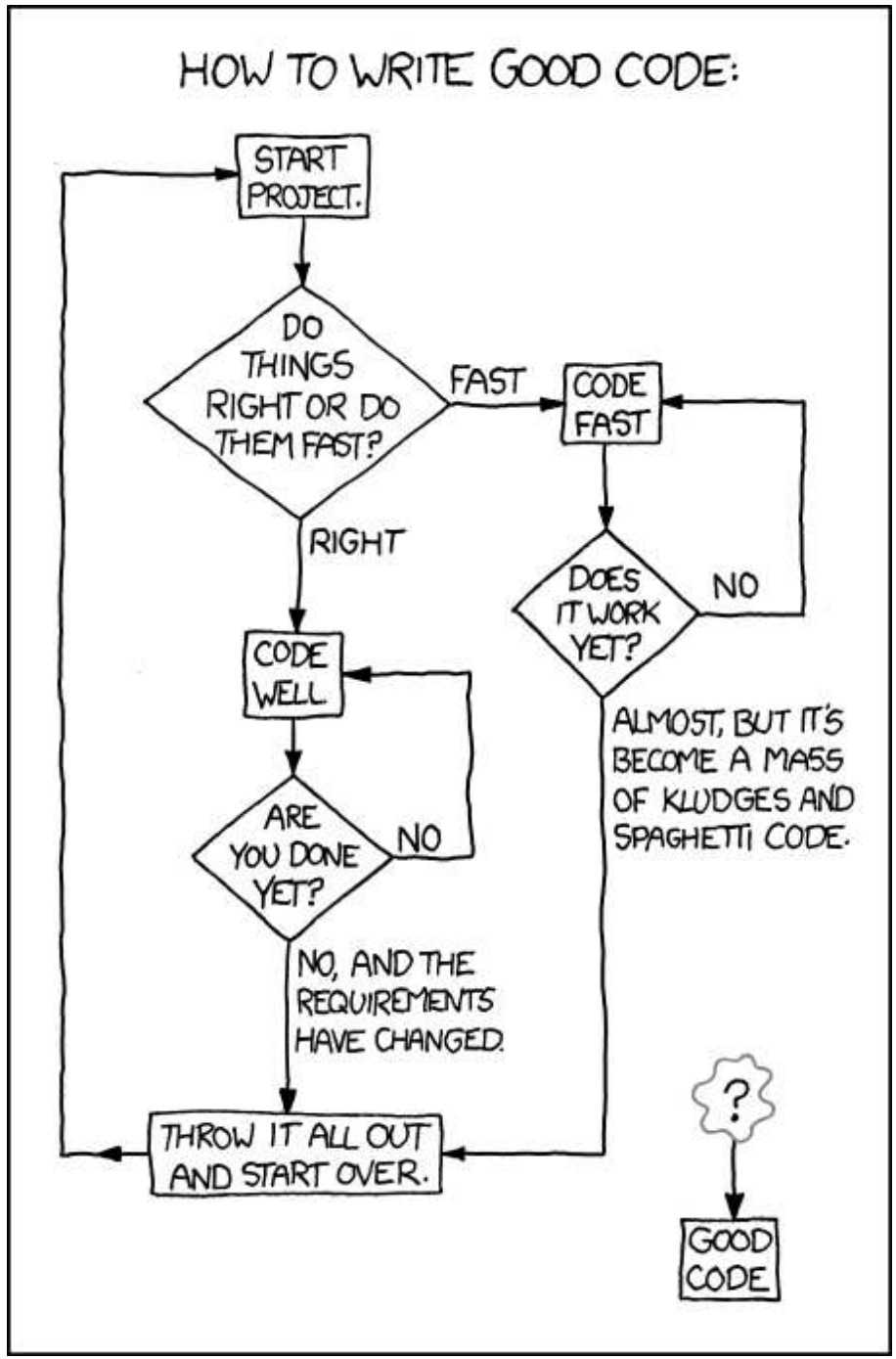
Automatic testing of client code

Luis Fernando Muñoz Mejías

Universiteit Gent

14^{th} Quattor Workshop


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
RIGHT
DOES IT WORK YET?
NO
CODE WELL
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
?
THROW IT ALL OUT AND START OVER.
GOOD CODE

Outline

A few common places

- Debugging sucks, testing rocks
- Tests must be repeatable
- I want to fix your bugs
- I want **you** you fix **my** bugs
- Design for testability
- Good code is easy to test
- Add test cases for your bugs

- Globals are evil
- Global state is just a global
- Bugfixes are likely to introduce new bugs
- Tests must be isolated
- High churn may imply high bug density

A few common places

- Debugging sucks, testing rocks
- Tests must be repeatable
- I want to fix your bugs
- I want **you** you fix **my** bugs
- Design for testability
- Good code is easy to test
- Add test cases for your bugs

- Globals are evil
- Global state is just a global
- Bugfixes are likely to introduce new bugs
- Tests must be isolated
- High churn may imply high bug density

Our current testing situation

1. We write some code
2. We push the code to some test node
   - If we can find one
3. We run the code with (maybe) a handful of profiles
4. Iterate

Our current testing situation

What it really means

Our desired testing situation

- Immediate feedback about a change
  - And I mean **immediate**
- On our own work stations
- Easy set up
- Repeat what others tested
  - In just one step!!

What we'd gain

- Predictable behaviour
- Less bugs
- Ability to refactor code
  - We know what breaks with a change!!
- Faster development

Dependency Injection

Example (Bad)

```
package Home;
sub new
{
    my $class = shift;
    my $self = {kitchen => IKEA::Kitchen->new()};
}
```

If your city doesn't have an IKEA, bad things will happen

- The object is not responsible of constructing its members
- Quattor design follows the "bad" example too often

Dependency Injection

Example (Good)

```
package Home;
sub new
{
    my ($class, $kitchen) = @_;
    my $self = {kitchen => $kitchen};
}
```

I'm not locked into IKEA kitchens anymore

- The object is not responsible of constructing its members
- Quattor design follows the "bad" example too often

Mocking, faking and overriding

- So we have to make the code *believe* that it is really accessing an IKEA kitchen
- Requires a complicated layout
- But we have it at last!

Setting it up

- The build tools will run the tests if they find the PERL5LIB environment variable
- It must contain the paths to CCM, CAF, LC and NCM
- Then, run `mvn test`

Coverage reports

- Set the HARNESS_PERL_SWITCHES environment variable
- Mine is:

```
-MDevel::Cover=+ignore,/(test|LC|CCM|CAF|Test|
                         dependency)/
```

- Then, run `cover`

Disable all privileged operations

- We don't want to run commands that ruin our work station
- We don't want to modify files in our system!
- We want the component to believe it is actually modifying the system

```
use Test::Quattor;
```

Disable all privileged operations

- We don't want to run commands that ruin our work station
- We don't want to modify files in our system!
- We want the component to believe it is actually modifying the system

```
use Test::Quattor;
```

Testing mandates

- Tests must not access the network
  - Or you'll get spurious failures
- Tests must not share data or state among them
  - Or you won't be able to run them in parallel
- Tests must write their temporary results to `target/test/...`

Unit testing

- A single function is tested
  - Executed with different arguments or simulating different circumstances
- Any functions in your module called from the tested function must be mocked

```
use Test::MockModule;
```

Unit testing

- A single function is tested
  - Executed with different arguments or simulating different circumstances
- Any functions in your module called from the tested function must be mocked

```
use Test::MockModule;
```

MockModule example

Example

```
use Test::MockModule;

my $mock=Test::MockModule
    ->new('NCM::Component::spma');
$mock->mock("installed_pkgs", sub {
    my $self = shift;
    $self->{INSTALLED}->{args} = \@_;
    return $a_value;
    });
```

- We control the return value and side effects of `installed_pkgs`
- We can verify wether it was called

Validating the results

- We are interested usually in these three things
  - Return value
  - Contents and properties of any files opened
  - Commands executed
- And in how the code under test reacts to them
  - Use `get_file` and `get_command`

Injecting files and command statuses

- We have the ability to choose the status of a file or command before calling the function
  - `set_file_contents` `set_command_status` `set_desired_output` `set_desired_err`

Integration testing

- Test how a function behaves with its real callees
- Mock less
- You can afford less strict checks too

Time for an example

Test profiles

- Mocking CCM is difficult
- Better use real profiles
  - Place them under `src/test/resources`
- Pass arguments to `Test::Quattor`

Example

```
use Test::Quattor qw(profile);
my $cfg = get_config_for("profile");
```

- Use this for testing the `Configure` method

Globals are bad and subtle

- Avoid global variables
- Avoid singletons
- Watch out your constants!!

Constants and dependency injection

What's the difference?

Example

```
# Creates the repository dir if needed.
sub initialize_repos_dir
{
    my ($self) = @_;
    if (! -d REPO_DIR) {
        if (!eval{mkpath(REPO_DIR)} || $@) {
            $self->error("...: $@");
            return 0;
        }
    }
    return 1;
}
```

Constants and dependency injection

What's the difference?

Example

```
# Creates the repository dir if needed.
sub initialize_repos_dir
{
    my ($self, $repo_dir) = @_;
    if (! -d $repo_dir) {
        if (!eval{mkpath($repo_dir)} || $@) {
            $self->error("...: $@");
            return 0;
        }
    }
    return 1;
}
```

It's all about testability

- In the first example it's very difficult to control the behaviour of the component
  - It will fail unless executed as root
  - Or unless we pollute our code with mocks
  - The constant has became a global handle for a directory
- In the second one, we just need to call the function with different arguments
  - And see it succeed or fail

It's all about testability

- In the first example it's very difficult to control the behaviour of the component
  - It will fail unless executed as root
  - Or unless we pollute our code with mocks
  - The constant has became a global handle for a directory
- In the second one, we just need to call the function with different arguments
  - And see it succeed or fail

Example (Invocation in the component)

```
$self->initialize_repos_dir(REPOS_DIR);
```

It's all about testability

- In the first example it's very difficult to control the behaviour of the component
  - It will fail unless executed as root
  - Or unless we pollute our code with mocks
  - The constant has became a global handle for a directory
- In the second one, we just need to call the function with different arguments
  - And see it succeed or fail

Example (Invocations in tests)

```
$cmp->initialize_repos_dir("a");
$cmp->initialize_repos_dir("b");
```

Documenting your tests

- No test plans, please
  - Nobody reads them
- But use clear identifiers to each test called
- And use POD comments for each group of tests

Continuous integration server at UGent

- A Jenkins instance tests all components, CCM, CAF an the compiler
- Graphs, plots, reports
- Subscribe to the RSS of any modules you maintain!!
- Allows to test interactions of components, CCM, CAF...

Acceptance tests!

- This is when you really run the component in a machine with real profiles
- We have nothing automated yet
- May cloud (and Stratuslab) help here?
- May Jenkins coordinate this?

Conclusions

- We now can test automatically most of our client code
- A decent test suit will allow us to refactor some ugly code and fix bugs more easily
- We need to start writing those tests
- When you code, ask yourself how you will test that
- When you get a bug report, please **add a new test for it**

Questions?

Debugging Sucks!

Testing Rocks!

Credits

- Code diagram from XKCD
- Watch The clean code talks