Durham University

COMP51915 attendance

Durham University

Code Analysis & Continuous Integration

COMP51915 – *Collaborative Software Development*

Michaelmas Term 2024

Christopher Marcotte[1]

[1]For errata and questions please contact `christopher.marcotte@durham.ac.uk`

Outline

Outline

Learning Goals

- Knowledge of Unit Testing approaches,
- Engagement with standard code analysis practices,
- Understanding of continuous integration practices,
- Ability to employ continuous integration with GitHub Actions

Outline

Learning Goals

- Knowledge of Unit Testing approaches,
- Engagement with standard code analysis practices,
- Understanding of continuous integration practices,
- Ability to employ continuous integration with GitHub Actions

There will be a **task** given at the end of these slides.

Scenario

You have just joined a software development company and have been tasked with ensuring the product is reliable before it launches.

Scenario

You have just joined a software development company and have been tasked with ensuring the product is reliable before it launches.

The code base is 160,000 lines long, in a custom *C++* dialect.

Scenario

You have just joined a software development company and have been tasked with ensuring the product is reliable before it launches.

The code base is 160,000 lines long, in a custom *C++* dialect.

You need to ensure that a feature being added next week (feature B) won't break a feature from six weeks ago (feature A).

Scenario

You have just joined a software development company and have been tasked with ensuring the product is reliable before it launches.

The code base is 160,000 lines long, in a custom *C++* dialect.

You need to ensure that a feature being added next week (feature B) won't break a feature from six weeks ago (feature A).

How can you be sure that the new features will not break the existing codebase?

Scenario

You have just joined a software development company and have been tasked with ensuring the product is reliable before it launches.

The code base is 160,000 lines long, in a custom *C++* dialect.

You need to ensure that a feature being added next week (feature ***B***) won't break a feature from six weeks ago (feature ***A***).

How can you be sure that the new features will not break the existing codebase?

You write tests that pass if (old) feature *A* gives expected results.

Unit Testing

Unit Testing is a *dynamic code analysis* process which identifies issues by restricting the attention of a test to a small *unit* of code.

[2] In C++ you will typically write a `void` function with no arguments which calls a testing macro, while in Python you would write a function which uses `assert` without a return which is driven from `main()`.

Unit Testing

Unit Testing is a *dynamic code analysis* process which identifies issues by restricting the attention of a test to a small *unit* of code.

Unit test specification varies by language and testing library[3] but focuses on *small* portions with *succinct* test functions (a *unit*).

[3] In C++ you will typically write a `void` function with no arguments which calls a testing macro, while in Python you would write a function which uses `assert` without a return which is driven from `main()`.

Unit Testing

Unit Testing is a *dynamic code analysis* process which identifies issues by restricting the attention of a test to a small *unit* of code.

Unit test specification varies by language and testing library[4] but focuses on *small* portions with *succinct* test functions (a *unit*).

A unit test is just a *function* you write to verify an assumption.

```
TEST(FactorialTest, Zero) {
  EXPECT_EQ(1, Factorial(0));
}
```

```
assert 2 + 2 == 5 #fails
@test 2 + 2 == 4  #success
```

[4] In C++ you will typically write a `void` function with no arguments which calls a testing macro, while in Python you would write a function which uses `assert` without a return which is driven from `main()`.

Unit Test Example

We have a function which computes the sum of `double* x` up to element `int n`:

```
double sum(double* x, int n){
  double s = 0.0;
  for (int i=0; i<n; i++){
    s += x[i];
  }
}
```

Unit Test Example

We have a function which computes the sum of `double* x` up to element `int n`:

```
double sum(double* x, int n){
  double s = 0.0;
  for (int i=0; i<n; i++){
    s += x[i];
  }
}
```

We immediately see:

Unit Test Example

We have a function which computes the sum of `double* x` up to element `int n`:

```
double sum(double* x, int n){
  double s = 0.0;
  for (int i=0; i<n; i++){
    s += x[i];
  }
}
```

We immediately see:

- n is the length of array x
  - if $\mathbf{len}(x) < n$ ...error?
- if $\mathbf{type}(x) \neq \mathbf{double}$
  - if an `int`...error?
- This sums *any* `double*`
  - can the sum yield inaccurate results?

Unit Test Example

We have a function which computes the sum of `double* x` up to element `int n`:

```
double sum(double* x, int n){
  double s = 0.0;
  for (int i=0; i<n; i++){
    s += x[i];
  }
}
```

We immediately see:

- n is the length of array x
  - if $\mathbf{len}(x) < n$ ...error?
- if $\mathbf{type}(x) \neq \mathbf{double}$
  - if an `int`...error?
- This sums *any* `double*`
  - can the sum yield inaccurate results?

There are issues with `sum( ... )` that we might hope to catch through unit tests.

Unit Test Example

We should first test for what is *expected* and verify code *correctness*.

[5]Unexpected input testing is sometimes called *fuzzing* – or testing against unexpected inputs to prevent security vulnerabilities, typically.

[6]Using `googletest`, the Google Test framework.

Unit Test Example

We should first test for what is *expected* and verify code *correctness*.

We might write a test[8] like:

```
TEST(SumFunction, ValidInput) {
  double x[] = {1.0, 2.0, 3.0};
  int n = 3;
  double result = sum(x, n);
  EXPECT_DOUBLE_EQ(result, 6.0);
}
```

[7]Unexpected input testing is sometimes called *fuzzing* – or testing against unexpected inputs to prevent security vulnerabilities, typically.

[8]Using `googletest`, the Google Test framework.

Unit Test Example

We should first test for what is *expected* and verify code *correctness*.

This example may also test for unexpected inputs, eventually.[9]

We might write a test[10] like:

```
TEST(SumFunction, ValidInput) {
  double x[] = {1.0, 2.0, 3.0};
  int n = 3;
  double result = sum(x, n);
  EXPECT_DOUBLE_EQ(result, 6.0);
}
```

Notice that this test will not catch all the issues we mentioned – types, accuracy.

[9]Unexpected input testing is sometimes called *fuzzing* – or testing against unexpected inputs to prevent security vulnerabilities, typically.

[10]Using `googletest`, the Google Test framework.

How do we start testing?

You should begin writing unit tests early in development – tests can serve as a demonstration of code usage, i.e. they're self-documenting.

Your tests should begin by codifying expectations of the *correct* path through the codebase – once we've exhausted our expectations, we should consider how to handle unexpected situations.

How do we start and when do we stop testing?

You should begin writing unit tests early in development – tests can serve as a demonstration of code usage, i.e. they're self-documenting.

Your tests should begin by codifying expectations of the *correct* path through the codebase – once we've exhausted our expectations, we should consider how to handle unexpected situations.

Unit tests are *sufficient* when there's *no more capacity to surprise*.

How do we start and when do we stop testing?

You should begin writing unit tests early in development – tests can serve as a demonstration of code usage, i.e. they're self-documenting.

Your tests should begin by codifying expectations of the *correct* path through the codebase – once we've exhausted our expectations, we should consider how to handle unexpected situations.

Unit tests are *sufficient* when there's *no more capacity to surprise*.

You are *not* testing the language constructs, just your *productive usage*!

Structuring Testing in a Program

Normally, we write an program to solve a problem, and testing is thought about during development but rarely considered once the program is ‘done’.

Structuring Testing in a Program

Normally, we write an program to solve a problem, and testing is thought about during development but rarely considered once the program is 'done'.

You use your build system to: include test execution in the *debug* or *test* build, and exclude tests from the *release* build.

Structuring Testing in a Program

Normally, we write an program to solve a problem, and testing is thought about during development but rarely considered once the program is ‘done’.

You use your build system to: include test execution in the *debug* or *test* build, and exclude tests from the *release* build.

Your build system then swaps your entrypoint to the program e.g.,

Test:

```
int main(int argc, char **argv) {
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Release:

```
int main(int argc, char **argv) {
  our_real_main(&argc, argv);
  return 0;
}
```

Unit Testing v Testing Frameworks

There is a difference between:

- knowing how to write *unit tests* and
- knowing how to use a particular *testing framework*.

Unit Testing v Testing Frameworks

There is a difference between:

- knowing how to write *unit tests* and
- knowing how to use a particular *testing framework*.

The former is a *skill*, the latter is a matter of *habit*.

Unit Testing v Testing Frameworks

There is a difference between:

- knowing how to write *unit tests* and
- knowing how to use a particular *testing framework*.

The former is a *skill*, the latter is a matter of *habit*.

We have time for you to *learn the skill*…
…but the habit is a matter of practice and time.

Code Linting

Linting is a *static code analysis* process which identifies issues in the *source code* defining a program, like bugs and stylistic issues.[11]

[11] [1] S. C. Johnson, *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977. [Online]. Available: http://squoze.net/UNIX/v7/files/doc/15_lint.pdf

[12] That you, no doubt, ignore usually...

Code Linting

Linting is a *static code analysis* process which identifies issues in the *source code* defining a program, like bugs and stylistic issues.[13]

Linting tools exist for nearly all languages and can give you valuable feedback on your code.

These range from poor variable name choices to inefficient patterns – think of linting as dealing with all the *warnings* from the compiler[14] rather than errors.

[13][1] S. C. Johnson, *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977. [Online]. Available: http://squoze.net/UNIX/v7/files/doc/15_lint.pdf

[14]That you, no doubt, ignore usually...

Linting, by example

We have a python program

`factorial.py`:

```python
def calculate_factorial(n):
  if n < 0:
    raise ValueError("n < 0!")
  elif n == 0: #note: == != =
    return 1
  else:
    result = 1
    for i in range(1, n + 1):
      result *= i
    return result

if __name__ == "__main__":
  n = calculate_factorial(10)
print(n)
```

Linting, by example

We have a python program `factorial.py`:

If we run

```
pytest factorial.py
```

It will print a number of outputs to the terminal enumerating lexical and logical issues with the code.

```python
def calculate_factorial(n):
  if n < 0:
    raise ValueError("n < 0!")
  elif n == 0: #note: == != =
    return 1
  else:
    result = 1
    for i in range(1, n + 1):
      result *= i
    return result

if __name__ == "__main__":
  n = calculate_factorial(10)
print(n)
```

Linter Output

`pytest factorial.py`:

```
************* Module factorial
factorial.py:1:0: C0114: Missing module docstring (missing-module-
docstring)
factorial.py:1:0: C0116: Missing function or method docstring (missing-
function-docstring)
factorial.py:1:24: W0621: Redefining name 'n' from outer scope (line 13)
(redefined-outer-name)
factorial.py:2:4: R1720: Unnecessary "elif" after "raise", remove the
leading "el" from "elif" (no-else-raise)

------------------------------------------------------------------
Your code has been rated at 6.67/10 (previous run: 7.50/10, -0.83)
```

Linting Practice & Limits

Software development is “the process of putting bugs in code”[15] – linting is a guide for identifying code issues *before they become bugs* (during execution).

[15] “If debugging is the process of removing software bugs, then programming must be the process of putting them in.” — Dijkstra, likely apocryphal.

[16] Compared to *compiling* or *running* the code.

Linting Practice & Limits

Software development is “the process of putting bugs in code”[17] – linting is a guide for identifying code issues *before they become bugs* (during execution).

Using a Linter can be a great way to improve the quality of your code, especially when you don’t have ready access to an expert reviewer.

[17] “If debugging is the process of removing software bugs, then programming must be the process of putting them in.” — Dijkstra, likely apocryphal.

[18] Compared to *compiling* or *running* the code.

Linting Practice & Limits

Software development is “the process of putting bugs in code”[19] – linting is a guide for identifying code issues *before they become bugs* (during execution).

Using a Linter can be a great way to improve the quality of your code, especially when you don’t have ready access to an expert reviewer.

However, because Linters are a distinct, additional, way to interpret your code[20] they can generate false positives – identifying *useful* feedback is difficult.

[19] “If debugging is the process of removing software bugs, then programming must be the process of putting them in.” — Dijkstra, likely apocryphal.

[20] Compared to *compiling* or *running* the code.

Linting with C

Configuring a linter with *C/C++* is <u>less trivial</u> than the Python example – but often worthwhile and very good practice.

```c
#include <stdio.h>
int r_factorial(int n){
  return n * r_factorial(n-1);
}
int main(){
  int n = 10;
  printf("%d! = %d\n",
    n, n = r_factorial(n));
}
```

Using `clang-tidy bad_factorial.c`:

Linting with C

Configuring a linter with *C/C++* is less trivial than the Python example – but often worthwhile and very good practice.

```c
#include <stdio.h>
int r_factorial(int n){
  return n * r_factorial(n-1);
}
int main(){
  int n = 10;
  printf("%d! = %d\n",
    n, n = r_factorial(n));
}
```

Using `clang-tidy bad_factorial.c`:

```
  warning: Although the value stored to
'n' is used in the enclosing expression,
the value is never actually read from
'n' [clang-analyzer-deadcode.DeadStores]
9 |            n, n = r_factorial(n));
  |               ^   ~~~~~~~~~~~~~~
```

Linting with C

Configuring a linter with *C/C++* is <u>less trivial</u> than the Python example – but often worthwhile and very good practice.

```c
#include <stdio.h>
int r_factorial(int n){
  return n * r_factorial(n-1);
}
int main(){
  int n = 10;
  printf("%d! = %d\n",
    n, n = r_factorial(n));
}
```

Using `clang-tidy bad_factorial.c`:

```
  warning: unsequenced modification and
access to 'n' [clang-diagnostic-
unsequenced]
9 |              n, n = r_factorial(n));
  |              ~    ^
```

Linting with C

Configuring a linter with *C/C++* is <u>less trivial</u> than the Python example – but often worthwhile and very good practice.

```c
#include <stdio.h>
int r_factorial(int n){
  return n * r_factorial(n-1);
}
int main(){
  int n = 10;
  printf("%d! = %d\n",
    n, n = r_factorial(n));
}
```

Using `clang-tidy bad_factorial.c`:

Recursive factorial never exits – infinite loop!

```
r_factorial(n) = n * r_factorial(n-1) * ...
r_factorial(0) * ...
```

The Linter misses the obvious error here – discoverable (only) by running the code!

How do we ensure the code is still correct, when we make changes?

How do we ensure the code is still correct, when we make changes?

How do we automate the process of bug finding?

Continuous Integration (CI)

The idea behind Continuous Integration (CI) is exceptionally simple:

[21] From Freya Holmér.

Continuous Integration (CI)

The idea behind Continuous Integration (CI) is exceptionally simple:

Making large significant changes to a code base is difficult...

Continuous Integration (CI)

The idea behind Continuous Integration (CI) is exceptionally simple:

Making large significant changes to a code base is difficult...
...so instead you should make small, incremental, changes.

This is the *calculus-moment*[23] for software development.

[23] From Freya Holmér.

Continuous Integration (CI)

The idea behind Continuous Integration (CI) is exceptionally simple:

Making large significant changes to a code base is difficult...
...so instead you should make small, incremental, changes.

This is the *calculus-moment*[24] for software development.

CI is the *practice* of *checking* your program after changes – i.e. running tests.

[24] From Freya Holmér.

GitHub Actions

Github Actions[25] defines the Continuous Integration pipeline using a YAML file (`.yml`) containing instructions similar to those involved in the construction of a Docker image.

These include:

[25] There are alternative CI frameworks, e.g. TravisCI and those built into GitLab. We restrict ourselves to GitHub Actions because you should all have accounts with access to these facilities and they are industry standards.

GitHub Actions

Github Actions[26] defines the Continuous Integration pipeline using a YAML file (`.yml`) containing instructions similar to those involved in the construction of a Docker image.

These include:

- Triggers; for the execution of
- A build, specified with, e.g. OS and library installation; and finally
- the actual run job for the continuous integration, i.e. *testing*.

[26]There are alternative CI frameworks, e.g. TravisCI and those built into GitLab. We restrict ourselves to GitHub Actions because you should all have accounts with access to these facilities and they are industry standards.

Trigger Action: `on:`

```
name: Python package

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
# continued ...
```

The GitHub Action is specified by a `name` keyword[27], and triggered by instructions following `on`.

[27] The GitHub Action *Python package* example here will both lint your code as well as test and run it.

Trigger Action: `on:`

```yaml
name: Python package

on:
  push:
    branches: [ "main" ]
  pull_request:
    branches: [ "main" ]
# continued ...
```

The GitHub Action is specified by a `name` keyword[28], and triggered by instructions following `on`.

Here we run our job on the branch named `"main"` **every time** the repo is

- `push`ed to; or
- recieves a `pull-request`

ensuring it is running up-to-date.

[28]The GitHub Action *Python package* example here will both lint your code as well as test and run it.

Specifying `jobs:`

```yaml
#  ... continued
jobs:
  build:

    runs-on: ubuntu-latest
    strategy:
      fail-fast: false
      matrix:
        python-version: ["3.11"]
# continued ...
```

When specifying `jobs:`, your first will typically be a `build:` job which oversees the build and run.

You should specify the environment (OS, language versions, etc.) ahead of specifying the build `steps:`...

Note the `matrix:` this indicates a **set** of python versions to test.

Specifying a job `steps:`

```
# ... continued
    steps:
      - uses: actions/checkout@v4
      - name: Set up Python ${{ matrix.python-version }}
        uses: actions/setup-python@v3
        with:
          python-version: ${{ matrix.python-version }}
# continued ...
```

First we specify the shared properties of the remaining steps.[29]

[29] Note the `matrix.python-version` indicates that the python version will loop over those specified earlier.

Install Dependencies

```
# ... continued
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip
          python -m pip install flake8 pytest
# continued ...
```

This job installs the essential dependencies for the Python stack and ensures they're up-to-date with `pip`. E.g., `flake8` & `pytest`.[30]

[30] `flake8` is a well-respected python linter, though Ruff is a faster choice for large codebases, while `pytest` is a test framework.

Execution

Finally, we reach the invocation of the linter and testing:

```yaml
# continued...
      - name: Lint with flake8
        run: |
          # stop the build if there are Python syntax errors
          flake8 . --count --show-source --statistics
      - name: Test with pytest
        run: |
          pytest
```

...and the GitHub Actions file is complete.

GitHub Actions Description

Much like the Dockerfile description, *no one expects you to memorize the GitHub Actions description*.

Rather you should be able to parse, modify, and implement GitHub Actions, and critically assess its usefulness for your own code.[31]

There is a lot of subtle detail in the specification format that I encourage you to reference when writing your own.

[31] It is important to understand the use-cases of any technology and whether it is worthwhile for your project.

Session Task

Task: Quadrature Tests

For this session task, you will write a small set of unit tests for a Python program which implements numerical integration averaging:

$$I = \frac{1}{b-a}\int_a^b f(x)\,\mathrm{d}x,$$

for any user defined function $f(x) : \Omega \subseteq \mathbb{R} \to \mathbb{R}$, using `scipy.integrate.quad`, and including **tests** in a continuous integration environment.

Task Guidance

We encourage you to:

- work together,
- consult the web,
- find creative solutions,
- document your process,
- ask questions…

[32] `pytest` is a reasonable place to start!

Task Guidance

We encourage you to:

- work together,
- consult the web,
- find creative solutions,
- document your process,
- ask questions…

1. **Record** implementation progress in a `git` repository
2. Don't reinvent the wheel – use a **testing framework**[33]
3. You will need to **debug** the default GitHub Actions CI.

This task is meant to prepare you for the workshop coursework.

[33] `pytest` is a reasonable place to start!

Numerical Integration Test Suggestions

1. Test the simplest case, e.g. $f(x) = c$, first.
2. What about a general polynomial function, $f \in \mathbb{P} : \mathbb{C} \to \mathbb{C}$.
3. Test an integrable function, e.g. $f(x) = \exp\left(-x^2\right)$.
4. Test variations on the limits, e.g. $a \to -\infty$ or $b \to +\infty$.
5. Compare to the different methods of the `scipy` function.
6. Compare when using non-default tolerances.

Numerical Integration Test Suggestions

1. Test the simplest case, e.g. $f(x) = c$, first.
2. What about a general polynomial function, $f \in \mathbb{P} : \mathbb{C} \to \mathbb{C}$.
3. Test an integrable function, e.g. $f(x) = \exp\left(-x^2\right)$.
4. Test variations on the limits, e.g. $a \to -\infty$ or $b \to +\infty$.
5. Compare to the different methods of the `scipy` function.
6. Compare when using non-default tolerances.

How do you know when you are *done* testing?

Bibliography

[1] S. C. Johnson, *Lint, a C program checker*. Bell Telephone Laboratories Murray Hill, 1977. [Online]. Available: http://squoze.net/UNIX/v7/files/doc/15_lint.pdf