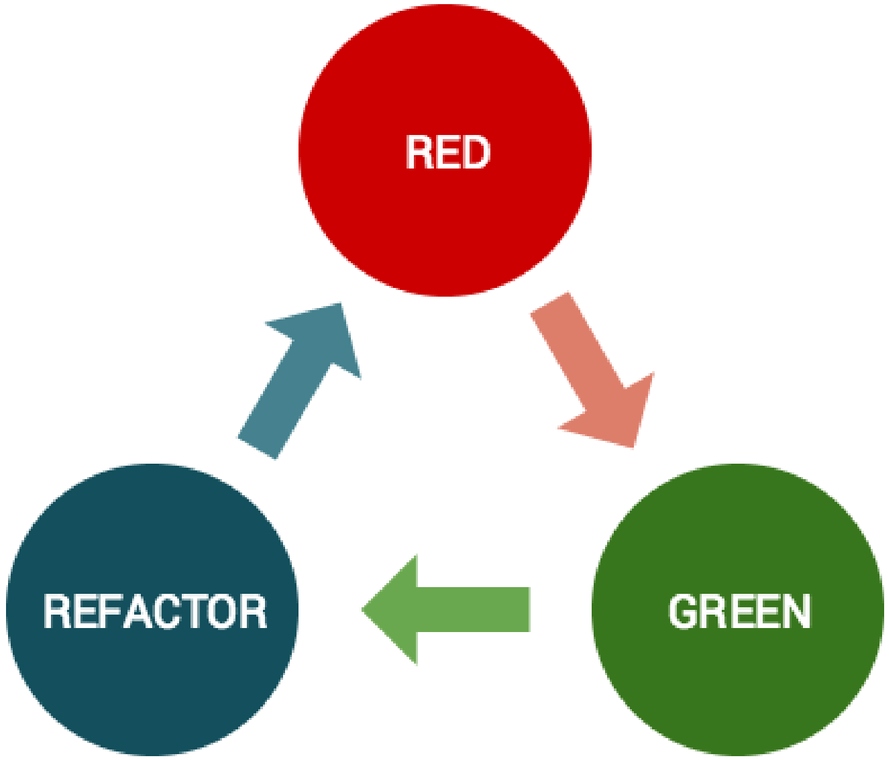
COMP 3550

3.4 — TEST-DRIVEN DEVELOPMENT (TDD) CYCLE

Week 3: Version Control & Testing Foundations

WHAT IS TDD?

- Red – Write a failing test (define the desired behavior)
- Green – Write the simplest code to make the test pass
- Refactor – Clean up the code while keeping the test green



Medium Post by Merce Bauza

WHY USE TDD?

- TDD encourages designing for testability
- Forces you to think about how your code will be used before you write it
- Leads to cleaner, modular, and more maintainable code
- Keeps you in line with YAGNI (ya ain’t gonna need it. More acronyms to come!)

EXAMPLE WALKTHROUGH: PASSWORDVALIDATOR

- Validate that passwords meet basic rules using TDD
- Test 1: Password Too Short

```
def test_password_too_short():
    assert not is_valid("abc")
```

EXAMPLE WALKTHROUGH: PASSWORDVALIDATOR

- Validate that passwords meet basic rules using TDD
- Test 1: Password Too Short

```
def test_password_too_short():
    assert not is_valid("abc")
```

```
def is_valid(pw):
    return len(pw) >= 8
```

EXAMPLE WALKTHROUGH: PASSWORDVALIDATOR

- Validate that passwords meet basic rules using TDD
- Test 1: Password Too Short

```
def test_password_too_short():
    assert not is_valid("abc")
```

- Missing Uppercase Letter

```
def test_password_needs_uppercase():
    assert not is_valid("lowercase8")
```

```
def is_valid(pw):
    return len(pw) >= 8
```

EXAMPLE WALKTHROUGH: PASSWORDVALIDATOR

- Validate that passwords meet basic rules using TDD
- Test 1: Password Too Short

```
def test_password_too_short():
    assert not is_valid("abc")
```

- Missing Uppercase Letter

```
def test_password_needs_uppercase():
    assert not is_valid("lowercase8")
```

```
def is_valid(pw):
    return len(pw) >= 8
```

```
def is_valid(pw):
    return len(pw) >= 8 and any(c.isupper() for c in pw)
```

TDD IN ACTION

- Each test exposes a missing requirement
- Code evolves gradually — with confidence
- Tests always stay **green** before you refactor

WHEN TDD WORKS BEST**

- Clear, Testable Logic
  - Business rules, validation logic, or data transformations
  - e.g., Password checkers, calculators, parsers
- Small, Modular Functions
  - TDD thrives when code is broken into reusable units
  - Encourages cleaner design from the start
- Team Collaboration
  - Writing tests first clarifies requirements and expectations
  - Helps avoid misunderstandings between devs, testers, and stakeholders

WHEN TDD WORKS BEST**

🚫 **When TDD Doesn't Shine:**

- UI-heavy code
- Spikes or prototypes
- Poorly defined requirements

COMMON PITFALLS

- Writing Tests That Are Too Broad
  - Tests should focus on one behavior, avoid trying to cover too much in a single test
- Skipping the “Red” Step
  - Don’t write code before the test fails
  - You lose the feedback loop TDD is built on
- Over-Refactoring Too Soon
  - Make it pass first, clean it up second
  - Premature refactoring often leads to confusion or broken tests
- Ignoring Test Readability
  - Tests are documentation — they should be easy to read
  - Use clear names and Arrange–Act–Assert structure
- Only Testing Happy Paths (for all types of testing really)
  - Don’t forget edge cases, invalid input, and error handling
  - Good tests cover both success and failure scenarios

If you’re not confident breaking things without fear — you’re not doing TDD yet.

TDD VS. JUST TESTING LATER

TDD (Test-Driven Development)	Testing After the Code
Write the test *before* the code	Write code first, then test
Guides design from the start	Tests adapt to finished code
Immediate feedback on design flaws	Bugs may go unnoticed longer
Encourages small, testable units	Risk of tightly coupled, hard-to-test code

- TDD gives you constant checkpoints while coding
- The process shapes better code, not just checks it afterward

PAUSE & REFLECT

Using TDD, write the tests and implementation for one of the following functions:

convert_celsius_to_fahrenheit(double temp)

get_top_scores(double[] scores, int n) // get top n scores from []scores

Once completed, what changed in your approach?