



Maintainable Code

Test-Driven Development

Traditional development workflow

1. Initial idea
2. Describe some behaviors / requirements
3. Design
4. Implement \leftarrow (write code)
 - Design problems may surface
5. Maybe write a few tests

■ Test-driven development workflow

- 💡 1. Initial idea
- ☰ 2. Describe some behaviors / requirements
- ⚙️ 3. Write 1 test
- 📄➡️ 4. Iterate on design
- 📁➡️ 4. Implement ← (write code for 1 test)
- 🔄 5. GOTO 3 until you're done
 - A good design emerges automatically
 - (You were using the code as you built it)

■ Why TDD?

- ◆ Maintenance
 - Most things are tested
 - Tests written first → easy to test
 - Fast feedback when something breaks
- ◆ Better APIs
 - API written before implementation
 - ▶ Converges towards easier-to-use code

■ Drawbacks

- ◆ Learning curve
 - Going from “code first” to “tests first” can be challenging
- ◆ TDD is a design & coding process, not a testing process
 - Not helpful when adding tests to an untested codebase
 - Not for full-application (e2e) testing
 - ▶ Use other automation tools

TDD Framework – AAA

```
#[test]
fn sample() {
    // Arrange (set up initial state)
    // Act (change the state)
    // Assert (check the changed state)
}
```

TDD Framework – GWT

```
#[test]
fn sample() {
    // Given an initial state
    // When the state changes
    // Then we check it
}
```

■ Properly written tests

- ◆ Test a single behavior
 - *Usually* 1 assertion
- ◆ Consistent
 - Test does not fail randomly between runs
- ◆ Everything tested locally
 - No communication with other machines
- ◆ Self-contained
 - Setup & teardown all happens within test
 - No reliance on test order or other tests

■ Recap

- ◆ TDD is a design & coding process
 - Not for testing untested code
 - Not for e2e testing
- ◆ TDD gives higher confidence in code reliability
 - Tests are part of the coding process
- ◆ TDD code is easier to change
- ◆ It will take practice to learn