Playbook Methods Repository

# **Test-Driven Development**

Use a technique of defining the target behaviour as a failing unit test, then creating the necessary logic to pass the test. This is achieved by means of a red/green/refactor feedback cycle which yields incremental changes, continuous refactoring, and simple design.

### Remote Agility: **•** High

### Linked Tactic(s): Agile Development

## Why we do it

Test-Driven Development (TDD) drives interface over implementation, a benefit complementary to design by contract leading to improved software design, reduction in bugs, and much faster test/development cycles leading to greater velocity. The test-first approach is central to extreme programming, a type of agile development, and facilitates other related tactics, enabling rapid pairing rotation through the red/green/refactor cycle, and enabling the necessary automation for successful CI/CD operation.

## 

## When to apply it

* During all phases of software development:

## Best Practices & Considerations

* Write tests first! That’s the “driven” in test-driven development
* Treat test code like production code
* KISS (Keep It Super Simple)
  + each test should be limited to a single requirement or specification
  + do not test implementation details
  + test code can be refactored too - as patterns emerge, e.g. setup and teardown tasks, take advantage of test harness facilities for those operations
* Tests should be independent of each other
* Bring a test from red to green with deliberate, incremental changes
  + Create the class, test
  + Create the method, test
  + Return a passing value, test
  + Refactor with implementation, test (repeat this step as needed)

## Responsible roles

* Product Manager:
* Software Engineers / Pair:

## Tools

### Software Testing Tools

* + Test harness or framework: software responsible for discovering and running tests, reporting failed and passed tests, expected and actual results, test coverage, and other test information
  + Assertion library: software that facilitates the writing of tests and the comparison between expected and actual results of a software process (function, method, etc)
  + Test interfaces: these interfaces facilitate the expression (in code) of test conditions, examples include should, expect, assert
    - TDD typically uses “assert”-style tests

### Examples

* + Javascript
    - Test harnesses: [mochajs.org](https://mochajs.org), [Jest](https://jestjs.io)
    - Assertion library: [Chai Assertion Library](https://www.chaijs.com) - Chai offers interfaces for should, expect, and assert style tests

## 

## Thoughtworks Examples - Linked

### Client working docs, airtable, miro/mural boards

* + xx

### Client polished presentations/deliverables

* + xx

### Internal assets - clinic materials / guild docs

* + [Microlearn - Pairing & TDD](https://drive.google.com/file/d/1yvmLJ9xqyf9Pz6arq_Ds2YuWcFEQ_5kN/view?usp=sharing)

## 

## Learn more: How we do this?

### Templates (docs, decks, sheets, miro, etc.)

* + xx

### How-To Resources (external or internal)

* + xx

### Outside References (articles, books, etc.)

* + xx

### Sub-set Activities

* + xx

## \*Practice Problem Sets

* [Katas in Ruby](https://github.com/connected-io/connected-learning-swe/tree/master/skills-based-technical-coaching-program/katas-ruby)