



**DesignNews**

Test Automation Design for Embedded Systems

# DAY 3 : Unit-Testing Using Test-Driven Development (TDD) Part 1

Sponsored by

**DigiKey**



## Webinar Logistics

- Turn on your system sound to hear the streaming presentation.
- If you have technical problems, click “Help” or submit a question asking for assistance.
- Participate in ‘Group Chat’ by maximizing the chat widget in your dock.

## THE SPEAKER



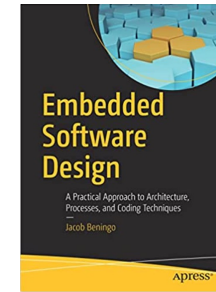
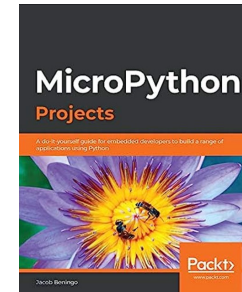
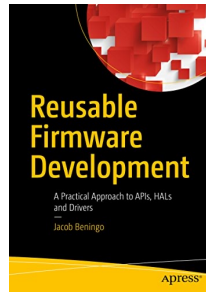
**Jacob Beningo**

Jacob@beningo.com

## Beningo Embedded Group – CEO / Founder

Focus: Embedded Software Consulting and Training

Help teams deliver higher-quality embedded software faster. We specialize in creating and promoting embedded software excellence in businesses around the world.



Blogs for:

- DesignNews.com
- Embedded.com
- EmbeddedRelated.com
- MLRelated.com

Visit [www.beningo.com](http://www.beningo.com) to learn more

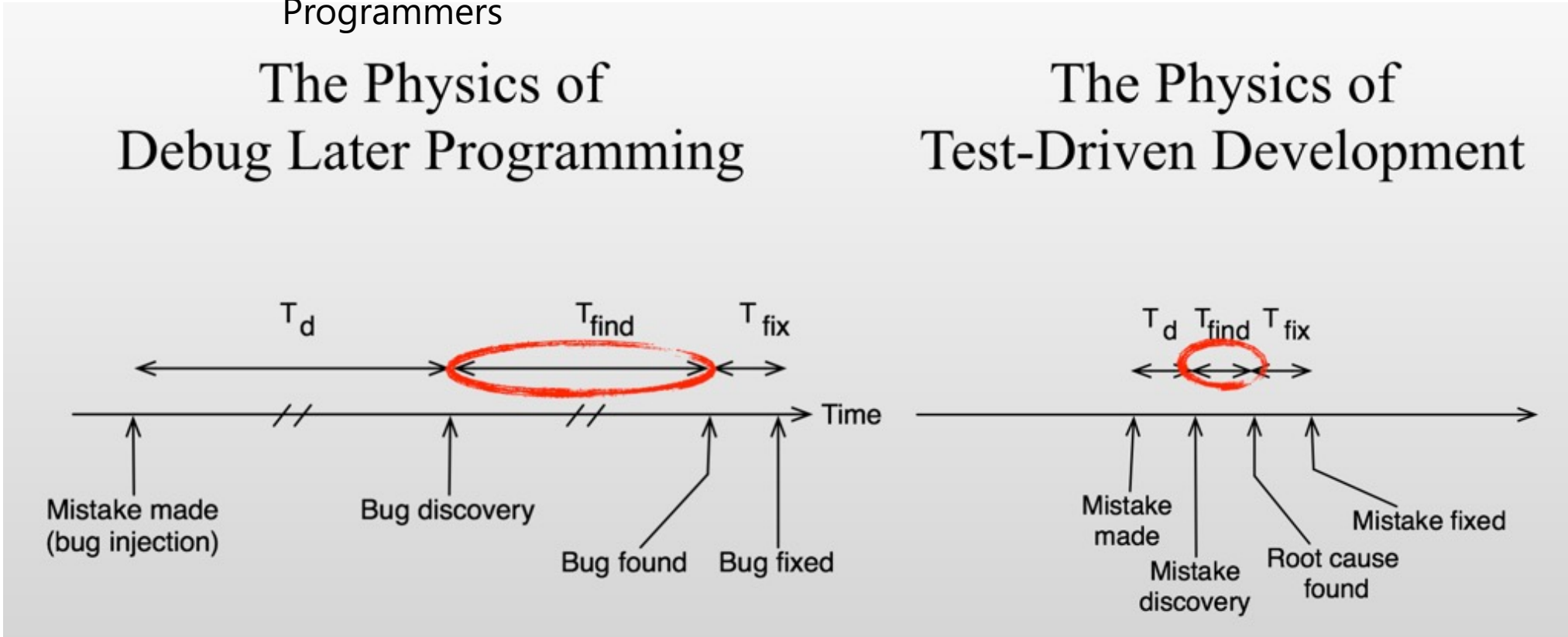


01

# Test-Driven Development TDD

# TDD Physics

Source: James Grenning; **TDD for Embedded C**; Pragmatic Programmers





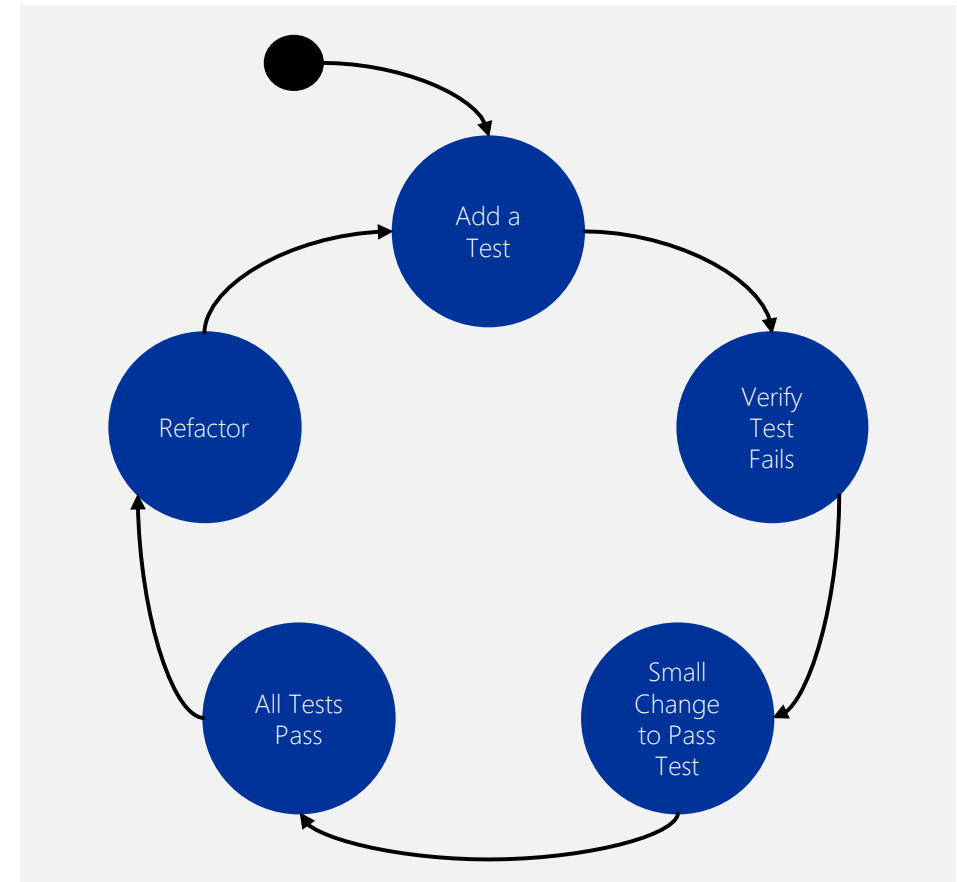
# Test-Driven Development (TDD)

**TDD** is a technique for building software incrementally that allows the test cases to drive the production code development.

- TDD improves code quality through cleaner, less buggy code
- Improves design due to developers thinking more carefully about what they are doing
- Code is debugged more efficiently due to failing tests that pinpoint exactly what the problem is
- Reduced development time and cost by catching issues earlier in the development cycle
- Developers can refactor with confidence due to existing tests

## The TDD Microcycle

1. Add a small test
2. Run all the tests and see the new one fail. (Maybe not even compile!)
3. Make the small change(s) needed to pass the test
4. Run all the tests and see the new one pass
5. Refactor to remove duplication and improve the expressiveness of the tests



## Audience POLL Question

How do you feel about TDD?

- For TDD
- Cautiously optimistic
- Skeptical, but see the value in it
- Rubbish! I can't support this concept





02

# What makes a great test?

# What makes a great test?

Edsger W. Dijkstra stated that:

**“Testing shows the presence, not the absence of bugs”**

## What makes a great test?

Great tests:

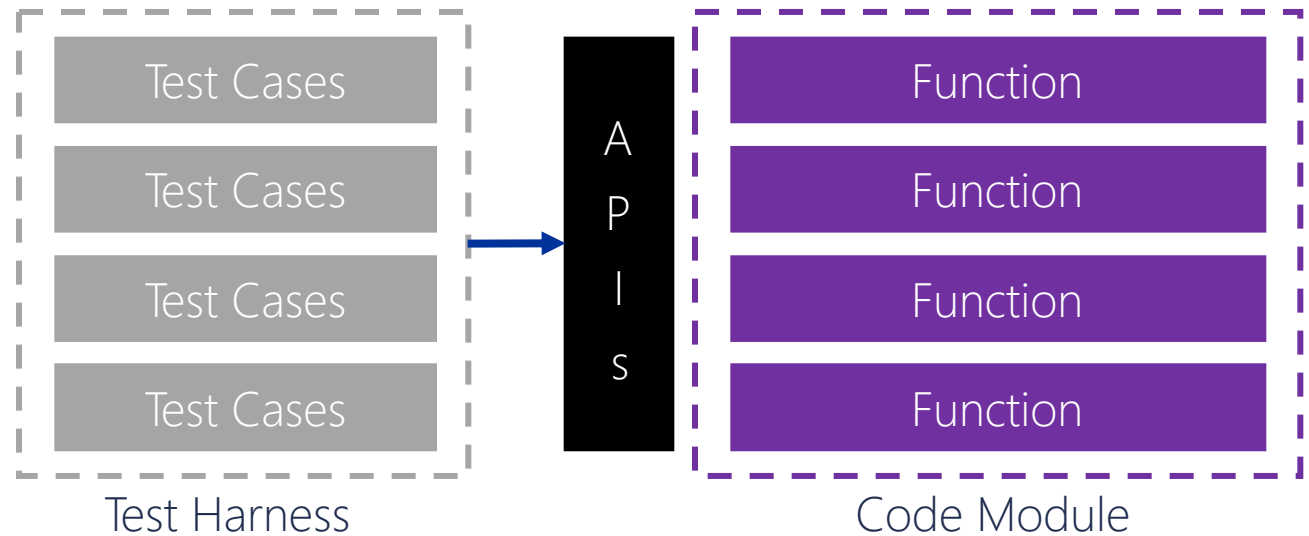
- Are limited in scope and test only one thing at a time
- Communicate their intent clearly
- Be self-documenting
- Automated
- Easily change with time (maintainable)

```
TEST(PacketDecoding, crc_correct)
```

# What makes a great test?

## Questions to ask yourself:

- What are the inputs to the interface?
- What are the outputs from the interface?
- Are there errors that are reported or can be tracked?
- What ranges can inputs and outputs be within?



## Audience POLL Question

Testing can prove that there are no bugs in the system:

- True
- False



03

# Writing your first unit test



# Writing Your First Unit Test

A few simple steps to get started:

- 1) Start Docker Desktop
- 2) Open Visual Studio Code
- 3) Open the controller project
- 4) Click Terminal->New Terminal
- 5) In the terminal, type `make docker_image`
- 6) In the terminal, type `make docker_run`
- 7) Write your first test

# Writing Your First Unit Test

The screenshot displays the Visual Studio Code interface for a project named 'controller'. The Explorer view on the left shows a directory structure with folders for 'tests', 'tools', and 'docs', and files like 'LICENSE.md', 'Makefile', and 'README.md'. A red box labeled '1' highlights the 'tests' folder, and a red box labeled '2' highlights the 'tools' folder. The main editor shows the 'README.md' file with a red box labeled '3' around the terminal output and a red box labeled '4' around the 'Overview' section. The terminal shows the command 'make docker\_run' and its output. The Preview window on the right shows the rendered README content, including the title 'Baseline Embedded Build and Test System by Beningo', author information, and an overview section.

```
## Baseline Embedded Build and Test System by Beningo
**Author:** Jacob W. Beningo
**Contact:** jacob@beningo.com
**Company:** Beningo Embedded Group, LLC
**Website:** www.beningo.com
**Origin Date:** 05/01/2024
**Version:** 1.6.0

See LICENSE.md for copyright and licensing information.

## Overview

The Baseline Embedded Build and Test System is an example repo that demonstrates how you might set up an embedded software project using modern tools and techniques.

It includes the following key pieces:
- Makefile to manage builds and simplify invoking tools like unit-tests, software analysis, etc.
- project.sh, a script for managing builds using cmake and ninja
- Dockerfile, for creating a containerized build and test system

The example is configured with the assumption that you would be using Visual Studio Code.

You'll find that through these build and tests scripts, you can build a target for:
- Debug on target
- Release on target
- Simulation
- Testing
- Code Analysis

## Getting started
```

```
(base) beningo@Jacobs-MacBook-Pro controller % make docker_run
docker run --rm -it --privileged -v "/Users/beningo/Projects/02-Experiments/2024-05-BuildSystem/controller:/home/app" name/embedded-dev:latest bash
root@e25010e3006:/home/app#
```

There was an error connecting to beningo. Please log in again.  
Source: Jira and Bitbucket (Atlassian Labs) View Atlassian settings

# Writing Your First Unit Test

Start with a design:

- Add 2 integers of type `uint8_t`

The interface might look like:

```
uint8_t add(uint8_t a, uint8_t b);
```

Make a list of tests:

- Zero values:  $a = 0, b = 0$
- Zero and non-zero:  $a = 0, b = 10$
- Zero and non-zero:  $a = 10, b = 0$
- Positive Numbers:  $a = 50, a = 70$
- Max without overflow:  $a = 125, b = 130$
- Overflow:  $a = 200, b = 100$
- "Typical" Values:  $a = 23, b = 77$
- Symmetrical values:  $a = 123, b = 123$
- Boundary values:  $a = 254, b = 1$
- Overflow by 1:  $a = 255 + 1$

## Writing Your First Unit Test

- 1) Open adder\_tests.cpp
- 2) Add the test:

```
TEST(adder, onePlusOneEqualsTwo)
{
    CHECK_EQUAL(2, add(1, 1));
}
```

- 3) Run "make unit\_tests", watch it fail:

```
/usr/bin/ld: tests/obj/tests/adder_tests.o: in function `TEST_adder_onePlusOneEqualsTwo_Test::testBody()':
/home/app/tests/adder_tests.cpp:25: undefined reference to `add'
/usr/bin/ld: /home/app/tests/adder_tests.cpp:25: undefined reference to `add'
/usr/bin/ld: /home/app/tests/adder_tests.cpp:25: undefined reference to `add'
/usr/bin/ld: /home/app/tests/adder_tests.cpp:25: undefined reference to `add'
collect2: error: ld returned 1 exit status
make[1]: *** [/home/cpptest/build/MakefileWorker.mk:521: tests/main_tests] Error 1
make[1]: Leaving directory '/home/app'
make: *** [Makefile:106: unit_tests] Error 2
root@de25010e3006:/home/app#
```

- 4) Add the function, and compile:

```
uint8_t add(uint8_t a, uint8_t b) {
}
```

- 5) Compile, and watch the test fail

```
Thus the value in the error message is probably incorrect.
tests/adder_tests.cpp:25: error: Failure in TEST(adder, onePlusOneEqualsTwo)
  expected <2>
  but was <4>
  difference starts at position 0 at: <      4      >
                                   ^
Errors (1 failures, 2 tests, 2 ran, 2 checks, 0 ignored, 0 filtered out, 0 ms)
```

# Writing Your First Unit Test

6) Write some code to make the test pass:

```
uint8_t add(uint8_t a, uint8_t b) {  
    return 2;  
}
```

7) Verify the test passes:

```
Running tests/main_tests  
**  
OK (2 tests, 2 ran, 2 checks, 0 ignored, 0 filtered out, 0 ms)  
  
make[1]: Leaving directory '/home/app'  
mkdir -p tests/coverage  
(INFO) Reading coverage data...  
(INFO) Writing coverage report...  
-----  
GCC Code Coverage Report  
Directory: .  
-----  
File                               Lines   Exec  Cover  Missing  
-----  
firmware/app/adder.c                2       2  100%  
TOTAL                               2       2  100%  
-----
```

## Audience POLL Question

Can you trust that 100% test coverage means there are no bugs in the code?

- a) Yes
- b) No



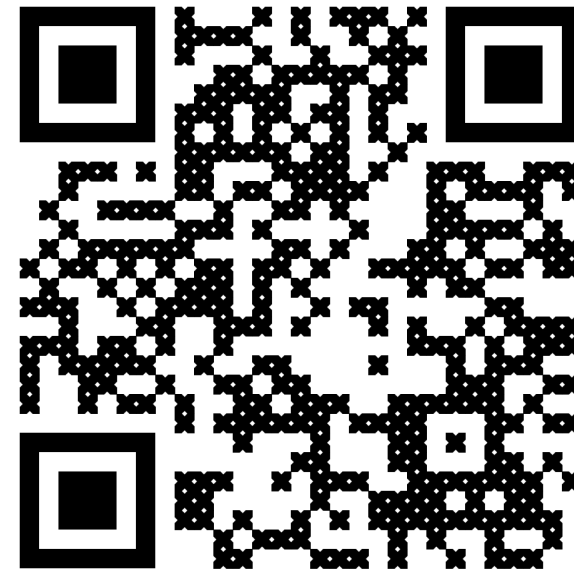
●● Next Steps

04

# Test Automation Build System

## Build System Example

- Docker container build system
- Makefile-based
- Cmake with Ninja Example
- Compilation scripts
- Integrated tools like cpputest



<https://mailchi.mp/beningo/beningo-devops>

## Additional Resources

Please consider the resources below:

- [Jacob's Blogs](#)
- [Jacob's CEC courses](#)
- [Embedded Software Academy](#)
- Embedded Bytes Newsletter
  - <http://bit.ly/1BAHYXm>

[www.beningo.com](http://www.beningo.com)



Consulting

Coaching

Training

## Next Steps

- ✓ Introduction to Test Automation Design
- ✓ Using Docker for a Test Automation Environment
- ✓ Unit-Testing Using Test-Driven Development Part 1
- Unit-Testing Using Test-Driven Development Part 2
- Automating System-Level Testing



**DesignNews**

Thank You

Sponsored by

**DigiKey**

