

# Unit testing and test-driven development (TDD)

Victor Eijkhout, Susan Lindsey

COE 322 Fall 2021

# 1 Dijkstra quote, part 1

*Today a usual technique is to make a program and then to test it. But: program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence. (cue laughter)*

Still ...

## 2 Types of testing

- *Unit tests* that test a small part of a program by itself;
- *System tests* test the correct behavior of the whole software system; and
- *Regression tests* establish that the behavior of a program has not changed by adding or changing aspects of it.

### 3 Unittesting frameworks

Testing is important, so there is much software that assists you.

Popular choice with C++ programmers: Catch2

<https://github.com/catchorg>

## 4 Compile and link options

```
INCLUDES = -I${TACC_CATCH2_INC}
```

```
EXTRALIBS = -L${TACC_CATCH2_LIB} -lCatch2 -lCatch2main
```

## 5 Unittest driver file

```
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

#include "yourprogram.h"
/*
    here follow the unit tests
*/
```

## 6 Require correctness

- *TEST\_CASE* acts like independent program.
- Can contain (multiple) tests for correctness.

```
TEST_CASE( "test that f always returns positive", "[key1][key2]" )
{
    for (int n=0; n<1000; n++)
        REQUIRE( f(n)>0 );
}
```

## 7 Test for exceptions

```
TEST_CASE( "test that g only works for positive", "[key1][key2]" )
{
    for (int n=-100; n<+100; n++)
        if (n<=0)
            REQUIRE_THROWS( g(n) );
        else
            REQUIRE_NOTHROW( g(n) );
}
```



## 8 Setup and teardown

```
TEST_CASE( "commonalities","[key1][key2]" ) {  
    // common setup:  
    double x,y,z;  
    REQUIRE_NOTHROW( y = f(x) );  
    // two independent tests:  
    SECTION( "g function" ) {  
        REQUIRE_NOTHROW( z = g(y) );  
    }  
    SECTION( "h function" ) {  
        REQUIRE_NOTHROW( z = h(y) );  
    }  
    // common followup  
    REQUIRE( z>x );  
}
```

## Eight queens problem

## 9 Sort of test-driven development

You will solve the ‘eight queens’ problem by

- designing tests for the functionality
- then implementing it

# 10 Basic object design

Object constructor of an empty board:

```
placement(int n);
```

Test how far we are:

```
int next_row_to_be_filled() const;
```

First test:

```
TEST_CASE( "empty placement","[constructor]" ) {  
    constexpr int n=10;  
    placement empty(n);  
    REQUIRE( empty.next_row_to_be_filled()==0 );  
}
```

# Exercise 1: Placement object

Start writing the *placement* class, and make it pass the above test.

## Exercise 2: Placement method

Write a method for placing a queen on the next row, missing snippet placementdonext and make it pass this test:

```
auto one(empty);  
REQUIRE_THROWS( one.place_next_queen_at_column(-1) );  
REQUIRE_THROWS( one.place_next_queen_at_column(n) );  
REQUIRE_NOTHROW( one.place_next_queen_at_column(0) );  
REQUIRE( one.next_row_to_be_filled()==1 );
```

## Exercise 3: Test for collisions

Write a method that tests if a placement is collision-free:

```
bool feasible() const;
```

This test has to work for simple cases to begin with. You can add these lines to the above tests:

```
REQUIRE( empty.feasible() );
```

```
REQUIRE( one.feasible() );
```

```
auto collide(one);  
collide.place_next_queen_at_column(0);  
REQUIRE( not collide.feasible() );
```

## Exercise 4: Test full solutions

Make a second constructor to 'create' solutions:

```
placement( vector<int> cols );
```

Now we test small solutions:

```
placement five( {0,3,1,4,2} );  
REQUIRE( five.feasible() );
```



## Exercise 5: No more delay: the hard stuff!

Write a function that takes a partial board, and places the next queen:

```
optional<placement> place_queen(const placement& current);
```

Test that the last step works:

```
placement almost( {1,3,0,place_empty} );  
auto solution = place_queen(almost);  
REQUIRE( solution.has_value() );  
REQUIRE( solution->filled() );
```

Alternative to using *optional*:

```
bool place_queen( const placement& current, placement &next );  
// true if possible, false is not
```

## Exercise 6: Test that you can find solutions

Test that there are no  $3 \times 3$  solutions:

```
TEST_CASE( "no 3x3 solutions", "" ) {  
    placement three(3);  
    auto solution = place_queen(three);  
    REQUIRE( not solution.has_value() );  
}
```

but  $4 \times 4$  solutions do exist:

```
TEST_CASE( "there are 4x4 solutions", "" ) {  
    placement four(4);  
    auto solution = place_queen(four);  
    REQUIRE( solution.has_value() );  
}
```