Test-Driven Development (TDD)

Victor Eijkhout, Susan Lindsey

Fall 2023

last formatted: October 24, 2023

Intro to testing

1. Dijkstra quote

> *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. Unit testing

- Every part of a program should be testable
- $\Rightarrow$ good idea to have a function for each bit of functionality
- Positive tests: show that code works when it should
- Negative tests: show that the code fails when it should

4. Unit testing

- Every part of a program should be testable
- Do not write the tests after the program:
  write tests while you develop the program.
- Test-driven development:
  1. design functionality
  2. write test
  3. write code that makes the test work

5. Principles of TDD

Develop code and tests hand-in-hand:

- Both the whole code and its parts should always be testable.
- When extending the code, make only the smallest change that allows for testing.
- With every change, test before and after.
- Assure correctness before adding new features.

6. Unit testing frameworks

Testing is important, so there is much software to assist you.

Popular choice with C++ programmers: Catch2
`https://github.com/catchorg`

Intro to Catch2

7. Toy example

Function and tester:

```
// catch/require.cpp
#define CATCH_CONFIG_MAIN
#include "catch2/catch_all.hpp"

int five() { return 5; }

TEST_CASE( "needs to be 5" ) {
    REQUIRE( five()==5 );
}
```

The define line supplies a main:
you don't have to write one.

8. Tests that fail

```
// catch/require.cpp
float fiveish() { return 5.00001; }
TEST_CASE( "not six" ) {
  // this will fail
  REQUIRE( fivish()==5 );
  // this will succeed
  REQUIRE( fivish()==Catch::Approx(5) );
}
```

9. Compiling toy example

```
icpc -o tdd tdd.cxx \
    -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB} \
    -lCatch2Main -lCatch2
```

- Files:

  ```
  icpc -o tdd tdd.cxx
  ```

- Path to include and library files:

  ```
  -I${TACC_CATCH2_INC} -L${TACC_CATCH2_LIB}
  ```

- Libraries:

  ```
  -lCatch2Main -lCatch2
  ```

Make a script file!

10. CMake setup for Catch2

```
find_package( PkgConfig REQUIRED )
pkg_check_modules( CATCH2 REQUIRED catch2-with-main )
target_include_directories(
    ${PROGRAM_NAME} PUBLIC ${CATCH2_INCLUDE_DIRS} )
target_link_directories(
    ${PROGRAM_NAME} PUBLIC ${CATCH2_LIBRARY_DIRS} )
target_link_libraries(
    ${PROGRAM_NAME} PUBLIC ${CATCH2_LIBRARIES} )
```

Exercise 1: Simple test

1. Write a function

   ```
   double f(int n) { /* .... */ }
   ```

   that has only positive values as output.
2. Write a unit test that tests the function for a number of values.

You can base this off the file `tdd.cpp` in the repository

11. Slightly realistic example

We want a function that

- computes a square root for $x \geq 0$
- throws an exception for $x < 0$;

```
// catch/sqrt.cpp
double root(double x) {
  if (x<0) throw(1);
  return std::sqrt(x);
};

TEST_CASE( "test sqrt function" ) {
  double x=3.1415, y;
  REQUIRE_NOTHROW( y=root(x) );
  REQUIRE( y*y==Catch::Approx(x) );
  REQUIRE_THROWS( y=root( -3.14 ) );
}
```

What happens if you require:

```
  REQUIRE( y*y==x );
```

12. Correctness through 'require' clause

Simple test case:

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

- *TEST_CASE* acts like independent main program.
  can have multiple cases in a tester file
- *REQUIRE* is like `assert` but more sophisticated

13. Tests

Boolean:

```
REQUIRE( some_test(some_input) );
REQUIRE( not some_test(other_input) );
```

Integer:

```
REQUIRE( integer_function(1)==3 );
REQUIRE( integer_function(1)!=0 );
```

Boolean expressions need to be parenthesized:

```
REQUIRE( ( x>0 and x<1 ) );
```

14. Output for failing tests

Run the tester:

```
-------------------------------
test the increment function
-------------------------------
test.cpp:25
...............................

test.cpp:29: FAILED:
  REQUIRE( increment_positive_only(i)==i+1 )
with expansion:
  1 == 2

===============================
test cases: 1 | 1 failed
assertions: 1 | 1 failed
```

15. Diagnostic information for failing tests

INFO: print out information at a failing test

```
TEST_CASE( "test that f always returns positive" ) {
  for (int n=0; n<1000; n++)
    INFO( "iteration: " << n );
    REQUIRE( f(n)>0 );
}
```

16. Exceptions

Exceptions are a mechanism for reporting an error:

```
double SquareRoot( double x ) {
  if (x<0) throw(1);
  return std::sqrt(x);
};
```

More about exceptions later;
for now: Catch2 can deal with them

17. Test for exceptions

Suppose function `g(n)`

- succeeds for input $n > 0$
- fails for input $n \leq 0$:
  throws exception

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

18. Tests with code in common

Use *SECTION* if tests have intro/outtro in common:

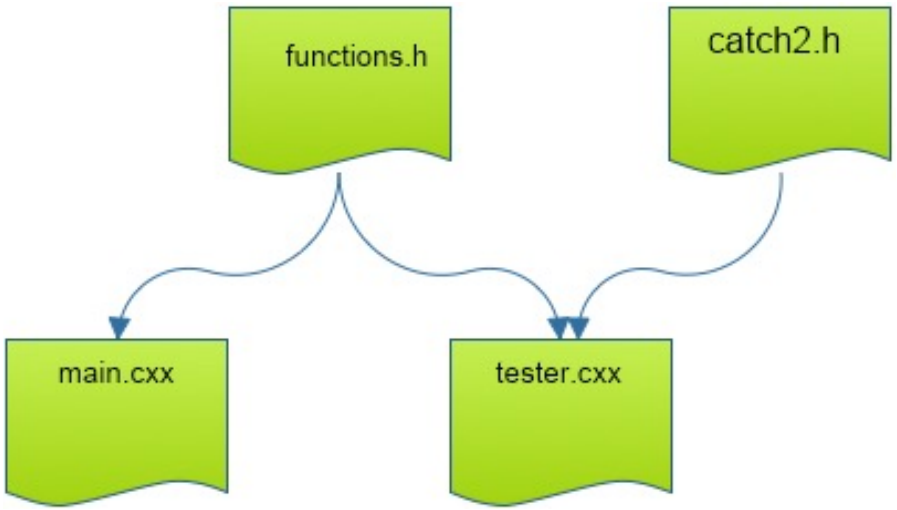
```
TEST_CASE( "commonalities" ) {
  // common setup:
  double x,y,z;
  REQUIRE_NOTHROW( y = f(x) );
  // two independent tests:
  SECTION( "g function" ) {
    REQUIRE_NOTHROW( z = g(y) );
  }
  SECTION( "h function" ) {
    REQUIRE_NOTHROW( z = h(y) );
  }
  // common followup
  REQUIRE( z>x );
}
```

(sometimes called setup/teardown)

Catch2 file structure

19. Realistic setup

- All program functionality in a 'library' file
- Main program really short
- Tester file with only tests.
- (Tester also needs the `catch2` stuff included)



Exercise 2: File structure

Make three files:

1. Include file with the functions.
2. Main program that uses the functions.
3. Tester main file, contents to be determined.

20. Function to be developed

File `functions.h` contains the function.

We know the structure:

```
// susan/functions.hpp
int increment_positive_only( int i ) {
  // this function returns one more than the inpuut
  // input has to be positive, error otherwise
    /* ... */
}
```

function body to be developed,

for now: `return 0;`

21. Functionality testing

File `tester.cxx`:

Same include file for the functionality;
the testing framework creates its own main.

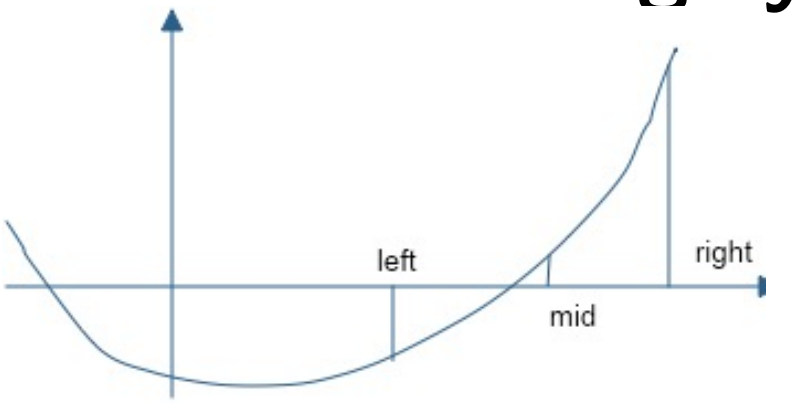
```
// susan/test.cpp
#include "functions.hpp"

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

TEST_CASE( "test the increment function" ) {
    /* ... */
}
```

TDD example: Bisection

22. Root finding by bisection



- Start with bounds where the function has opposite signs.

$$x_- < x_+, \qquad f(x_-) \cdot f(x_+) < 0,$$

- Find the mid point;
- Adjust either left or right bound.

23. Coefficient handling

$$f(x) = c_0 x^d + c_1 x^{d-1} \cdots + c_{d-1} x^1 + c_d$$

We implement this by storing the coefficients in a `vector<double>`.
Proper:

```
// root/testzeroarray.cpp
TEST_CASE( "coefficients represent polynomial" "[1]") {
  vector<double> coefficients = { 1.5, 0., -3 };
  REQUIRE( coefficients.size()>0 );
  REQUIRE( coefficients.front()!=0. );
}
```

Exercise 3: One test for properness

Write a function *is_proper_polynomial* as described, and write unit tests for it, both passing and failing:

```
vector<double> good = /* proper coefficients */ ;
REQUIRE( is_proper_polynomial(good) );
vector<double> notso = /* improper coefficients */ ;
REQUIRE( not is_proper_polynomial(notso) );
```

24. Handy shortcut

Are you getting tired of typing `vector<double>`?
put

```
// root/findzerolib.hpp
using polynomial = vector<double>;
```

somewhere high in your file.

25. Test on polynomials evaluation

Next we need to evaluate polynomials.

Equality testing on floating point is dangerous:
use *Catch::Approx(sb)*

```
// root/zeroclasstest.cpp
  polynomial second( {2,0,1} );
  // correct interpretation: 2x^2 + 1
  REQUIRE( second.evaluate_at(2) == Catch::Approx(9) );
  REQUIRE( second(2) == Catch::Approx(9) );
  // wrong interpretation: 1x^2 + 2
  REQUIRE( second.evaluate_at(2) != Catch::Approx(6) );
```

Exercise 4: Implementation

Write a function *evaluate_at* which computes

$$y \leftarrow f(x).$$

and confirm that it passes the above tests.

```
double evaluate_at( polynomial coefficients,double x);
```

For bonus points, look up Horner's rule and implement it.

26. Odd degree polynomials only

With odd degree you can always find bounds x_-, x_+.
For this exercise we reject even degree polynomials:

```
// root/zeroclassmain.cpp
  if ( not third_degree.is_odd() ) {
    cout << "This program only works for odd-degree
    polynomials\n";
    exit(1);
  }
```

This test will be used later;
first we need to implement it.

Exercise 5: Odd degree testing

Implement the *is_odd* test.

Gain confidence by unit testing:

```
// root/testzeroarray.cpp
  polynomial second{2,0,1}; // 2x^2 + 1
  REQUIRE( not is_odd(second) );
  polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
  REQUIRE( is_odd(third) );
```

27. Finding initial bounds

We need a function *find_initial_bounds* which computes x_-, x_+ such that

$$f(x_-) < 0 < f(x_+) \quad \text{or} \quad f(x_+) < 0 < f(x_-)$$

(can you write that more compactly?)

```
void find_initial_bounds
  ( polynomial coefficients,double &left,double &right);
```

Since we reject even degree polynomials,
throw an exception for those.

Exercise 6: Test for initial bounds

Unit test:

```
// root/testzeroarray.cpp
  right = left+1;
  polynomial second{2,0,1}; // 2x^2 + 1
  REQUIRE_THROWS( find_initial_bounds(second,left,right) );
  polynomial third{3,2,0,1}; // 3x^3 + 2x^2 + 1
  REQUIRE_NOTHROW( find_initial_bounds(third,left,right) );
  REQUIRE( left<right );
```

Can you add a unit test on the left/right values?

28. Move the bounds closer

Root finding iteratively moves the initial bounds closer together:

```
move_bounds_closer(coefficients,left,right);
```

- on input, `left`$<$`right`, and
- on output the same must hold.

Design a test for this function;
implement this function.

29. Putting it all together

Ultimately we need a top level function

```
double find_zero( polynomial coefficients,double prec );
```

- reject even degree polynomials
- set initial bounds
- move bounds closer until close enough:
  $|f(y)| <$ `prec`.

Exercise 7: Put it all together

Make this call work:

```
// root/zeroclassmain.cpp
  auto zero = find_zero( coefficients, 1.e-8 );
  cout << "Found root " << zero
       << " with value " << evaluate_at(coefficients,zero) <<
    '\n';
```

Design unit tests, including on the precision attained, and make sure your code passes them.

Turn it in!

- If you think your functions pass all tests, subject them to the tester:

  `coe_bisection yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.
- The usual flags: `-s` for submit, `-d` for debug output, `-i` for submission as incomplete.
- The tester generates files `bisection_driver1.cxx` and such in your directory. You can use these to test partial solutions:

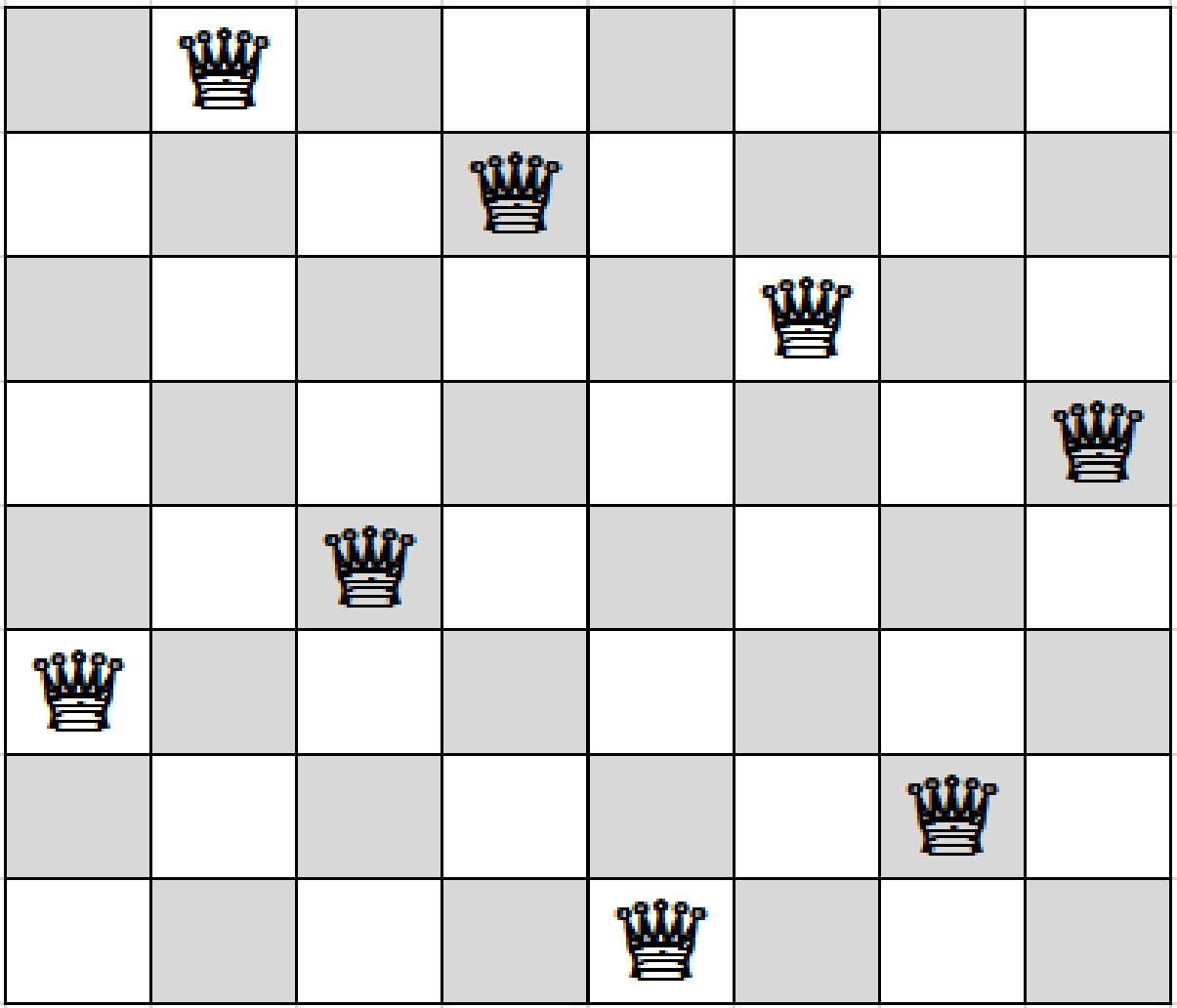
  `icpc -o bisection_driver bisection_driver5.cxx ### the`

  These commandlines show up if you use the `-d` flag.

TDD example: Eight queens

30. Classic problem

Can you put 8 queens on a board so that they can't hit each other?



31. Statement

- Put eight pieces on an 8×8 board, no two pieces on the same square; so that
- no two pieces are on the same row,
- no two pieces are on the same column, and
- no two pieces are on the same diagonal.

32. Not good solution

A systematic solution would run:

1. put a piece anywhere in the first row;
2. for each choice in the first row, try all positions in the second row;
3. for all choices in the first two rows, try all positions in the third row;
4. when you have a piece in all eight rows, evaluate the board to see if it satisfies the condition.

Better: abort search early.

Exercise 8: Board class

Class *board*:

```
// queens/queens.hpp
  ChessBoard(int n);
```

Method to keep track how far we are:

```
// queens/queens.hpp
  int next_row_to_be_filled()
```

Test:

```
// queens/queentest.cpp
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

Exercise 9: Place one queen

Method to place the next queen,
without testing for feasibility:

```
// queens/queens.hpp
  void place_next_queen_at_column(int i);
```

This test should catch incorrect indexing:

```
// queens/queentest.cpp
  INFO( "Illegal placement throws" )
  REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
  REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
  INFO( "Correct placement succeeds" );
  REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
  REQUIRE( empty.next_row_to_be_filled()==1 );
```

Without this test, would you be able to cheat?

Exercise 10: Test if we're still good

Feasibility test:

```
// queens/queens.hpp
  bool feasible()
```

Some simple cases:
(add to previous test)

```
// queens/queentest.cpp
  ChessBoard empty(n);
  REQUIRE( empty.feasible() );

// queens/queentest.cpp
  ChessBoard one = empty;
  one.place_next_queen_at_column(0);
  REQUIRE( one.next_row_to_be_filled()==1 );
  REQUIRE( one.feasible() );
```

Exercise 11: Test collisions

```
// queens/queentest.cpp
  ChessBoard collide = one;
  // place a queen in a `colliding' location
  collide.place_next_queen_at_column(0);
  // and test that this is not feasible
  REQUIRE( not collide.feasible() );
```

Exercise 12: Test a full board

Construct full solution

```
// queens/queens.hpp
  ChessBoard( int n,vector<int> cols );
  ChessBoard( vector<int> cols );
```

Test:

```
// queens/queentest.cpp
  ChessBoard five( {0,3,1,4,2} );
  REQUIRE( five.feasible() );
```

Exercise 13: Exhaustive testing

This should now work:

```
// queens/queentest.cpp
  // loop over all possibilities first queen
  auto firstcol = GENERATE_COPY( range(1,n) );
  ChessBoard place_one = empty;
  REQUIRE_NOTHROW(
    place_one.place_next_queen_at_column(firstcol) );
  REQUIRE( place_one.feasible() );

  // loop over all possbilities second queen
  auto secondcol = GENERATE_COPY( range(1,n) );
  ChessBoard place_two = place_one;
  REQUIRE_NOTHROW(
    place_two.place_next_queen_at_column(secondcol) );
  if (secondcol<firstcol-1 or secondcol>firstcol+1) {
    REQUIRE( place_two.feasible() );
  } else {
    REQUIRE( not place_two.feasible() );
  }
```

Exercise 14: Place if possible

You need to write a recursive function:

```
// queens/queens.hpp
  optional<ChessBoard> place_queens()
```

- place the next queen.
- if stuck, return 'nope'.
- if feasible, recurse.

```
class board {
  /* stuff */
  optional<board> place_queens() const {
    /* stuff */
    board next(*this);
    /* stuff */
    return next;
  };
```

Exercise 15: Test last step

Test *place_queens* on a board that is almost complete:

```
// queens/queentest.cpp
  ChessBoard almost( 4, {1,3,0} );
  auto solution = almost.place_queens();
  REQUIRE( solution.has_value() );
  REQUIRE( solution->filled() );
```

Note the new constructor! (Can you write a unit test for it?)

Exercise 16: Sanity tests

```
// queens/queentest.cpp
TEST_CASE( "no 2x2 solutions","[8]" ) {
  ChessBoard two(2);
  auto solution = two.place_queens();
  REQUIRE( not solution.has_value() );
}

// queens/queentest.cpp
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

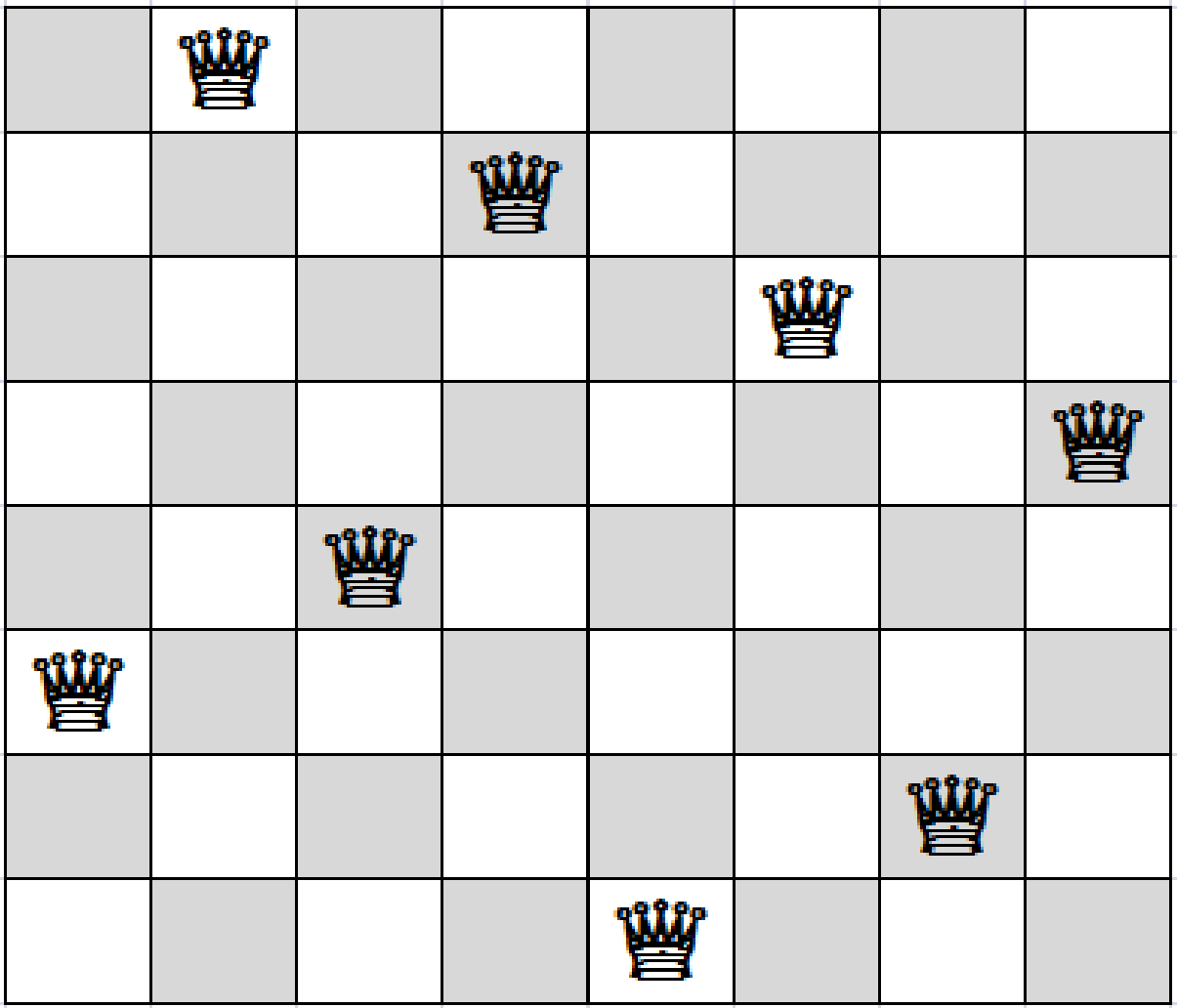
Exercise 17: O

ptional: can you do timing the solution time as function of the size of the board?

Eight queens problem by TDD (using objects)

33. Problem statement

Can you place eight queens on a chess board so that no pair threatens each other?

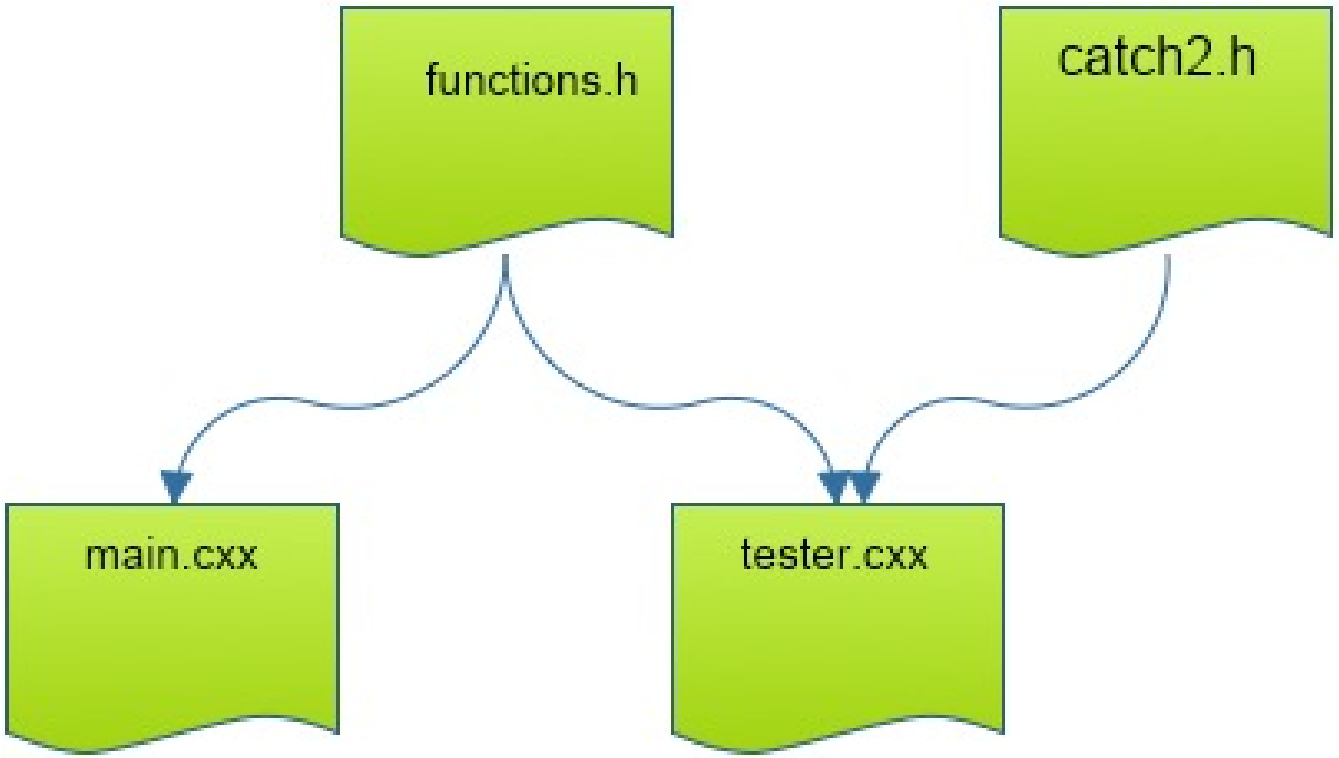


34. Sort of test-driven development

You will solve the 'eight queens' problem by

- designing tests for the functionality
- then implementing it

35. File structure



36. Basic object design

Object constructor of an empty board:

```
// queens/queens.hpp
  ChessBoard(int n);
```

Test how far we are:

```
// queens/queens.hpp
  int next_row_to_be_filled()
```

First test:

```
// queens/queentest.cpp
TEST_CASE( "empty board","[1]" ) {
  constexpr int n=10;
  ChessBoard empty(n);
  REQUIRE( empty.next_row_to_be_filled()==0 );
}
```

Exercise 18: Board object

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

Exercise 19: Board method

Write a method for placing a queen on the next row,

```
// queens/queens.hpp
  void place_next_queen_at_column(int i);
```

and make it pass this test (put this in a *TEST_CASE*):

```
// queens/queentest.cpp
  INFO( "Illegal placement throws" )
  REQUIRE_THROWS( empty.place_next_queen_at_column(-1) );
  REQUIRE_THROWS( empty.place_next_queen_at_column(n) );
  INFO( "Correct placement succeeds" );
  REQUIRE_NOTHROW( empty.place_next_queen_at_column(0) );
  REQUIRE( empty.next_row_to_be_filled()==1 );
```

Exercise 20: Test for collisions

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

```
// queens/queens.hpp
  bool feasible()
```

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

```
// queens/queentest.cpp
  ChessBoard empty(n);
  REQUIRE( empty.feasible() );

// queens/queentest.cpp
  ChessBoard one = empty;
  one.place_next_queen_at_column(0);
  REQUIRE( one.next_row_to_be_filled()==1 );
  REQUIRE( one.feasible() );

// queens/queentest.cpp
  ChessBoard collide = one;
  // place a queen in a `colliding' location
  collide.place_next_queen_at_column(0);
  // and test that this is not feasible
  REQUIRE( not collide.feasible() );
```

Exercise 21: Test full solutions

Make a second constructor to 'create' solutions:

```
// queens/queens.hpp
  ChessBoard( int n,vector<int> cols );
  ChessBoard( vector<int> cols );
```

Now we test small solutions:

```
// queens/queentest.cpp
  ChessBoard five( {0,3,1,4,2} );
  REQUIRE( five.feasible() );
```

Exercise 22: No more delay: the hard stuff!

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

```
// queens/queens.hpp
  optional<ChessBoard> place_queens()
```

Test that the last step works:

```
// queens/queentest.cpp
  ChessBoard almost( 4, {1,3,0} );
  auto solution = almost.place_queens();
  REQUIRE( solution.has_value() );
  REQUIRE( solution->filled() );
```

Alternative to using `optional`:

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

Exercise 23: Test that you can find solutions

Test that there are no 3×3 solutions:

```
// queens/queentest.cpp
TEST_CASE( "no 3x3 solutions","[9]" ) {
  ChessBoard three(3);
  auto solution = three.place_queens();
  REQUIRE( not solution.has_value() );
}
```

but 4×4 solutions do exist:

```
// queens/queentest.cpp
TEST_CASE( "there are 4x4 solutions","[10]" ) {
  ChessBoard four(4);
  auto solution = four.place_queens();
  REQUIRE( solution.has_value() );
}
```

Turn it in!

- If you think your functions pass all tests, subject them to the tester:

  `coe_queens yourprogram.cc`

  where '`yourprogram.cc`' stands for the name of your source file.
- Is it reporting that your program is correct? If so, do:

  `coe_queens -s yourprogram.cc`

  where the `-s` flag stands for 'submit'.
- If you don't manage to get your code working correctly, you can submit as incomplete with

  `coe_queens -i yourprogram.cc`
- If you want feedback on what the tester thinks about your code do

  `coe_queens -d yourprogram.cc`

  with the `-d` flag for 'debug.