Test First
Test Better

@phil_nash

CPP-Summit

TDD

CPP-Summit

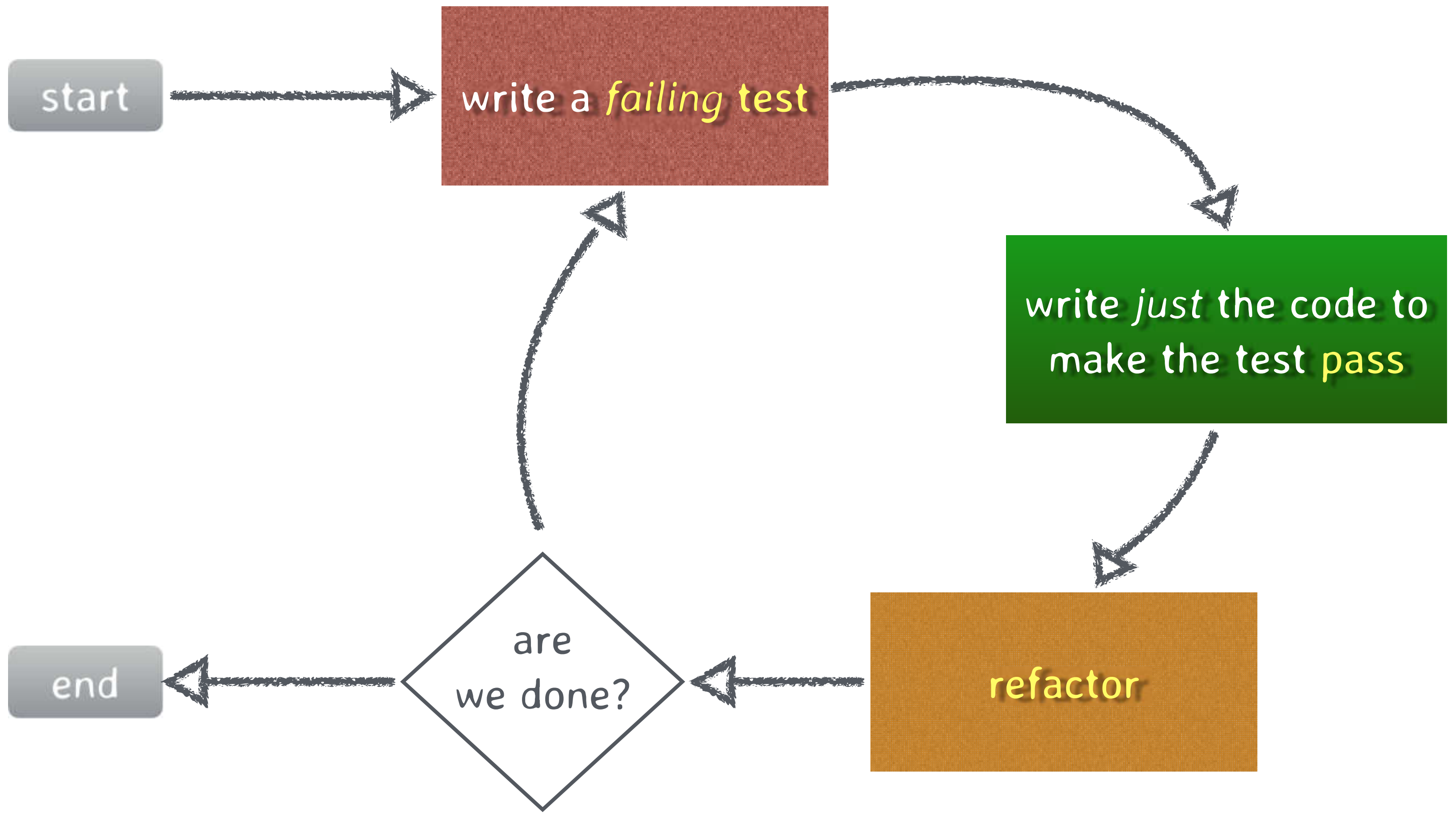
What is TDD?

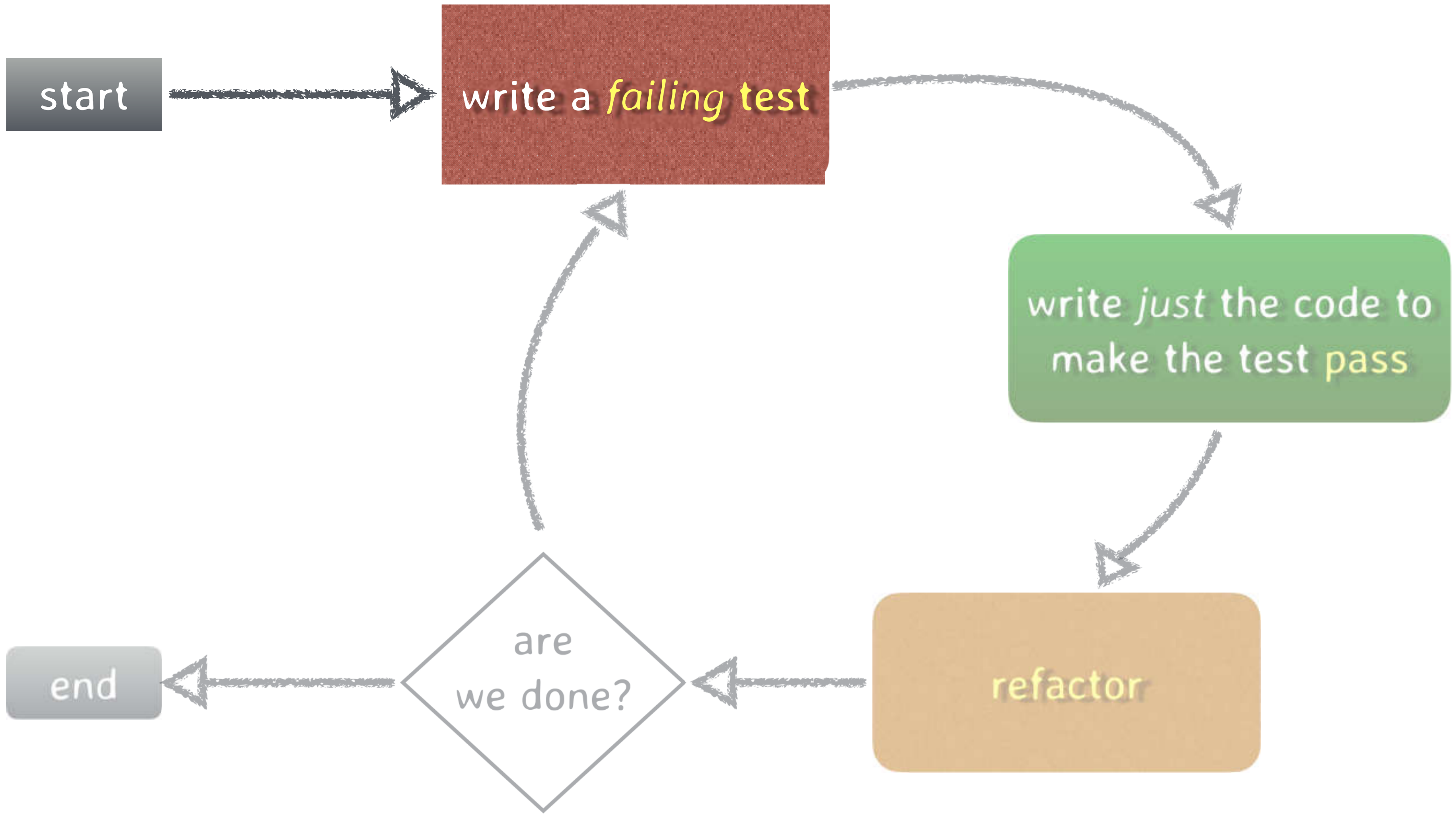
Test

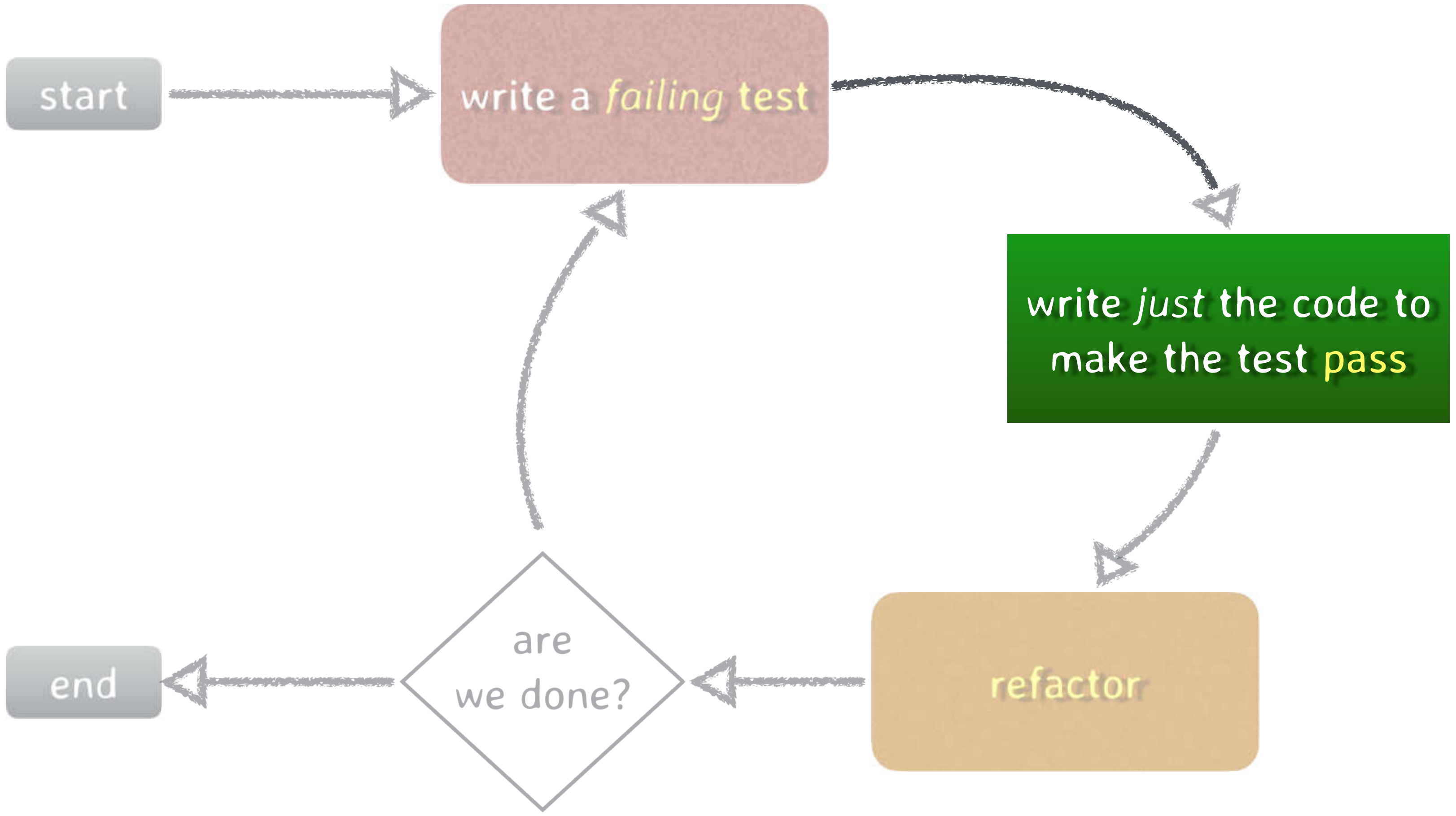
Driven

Development

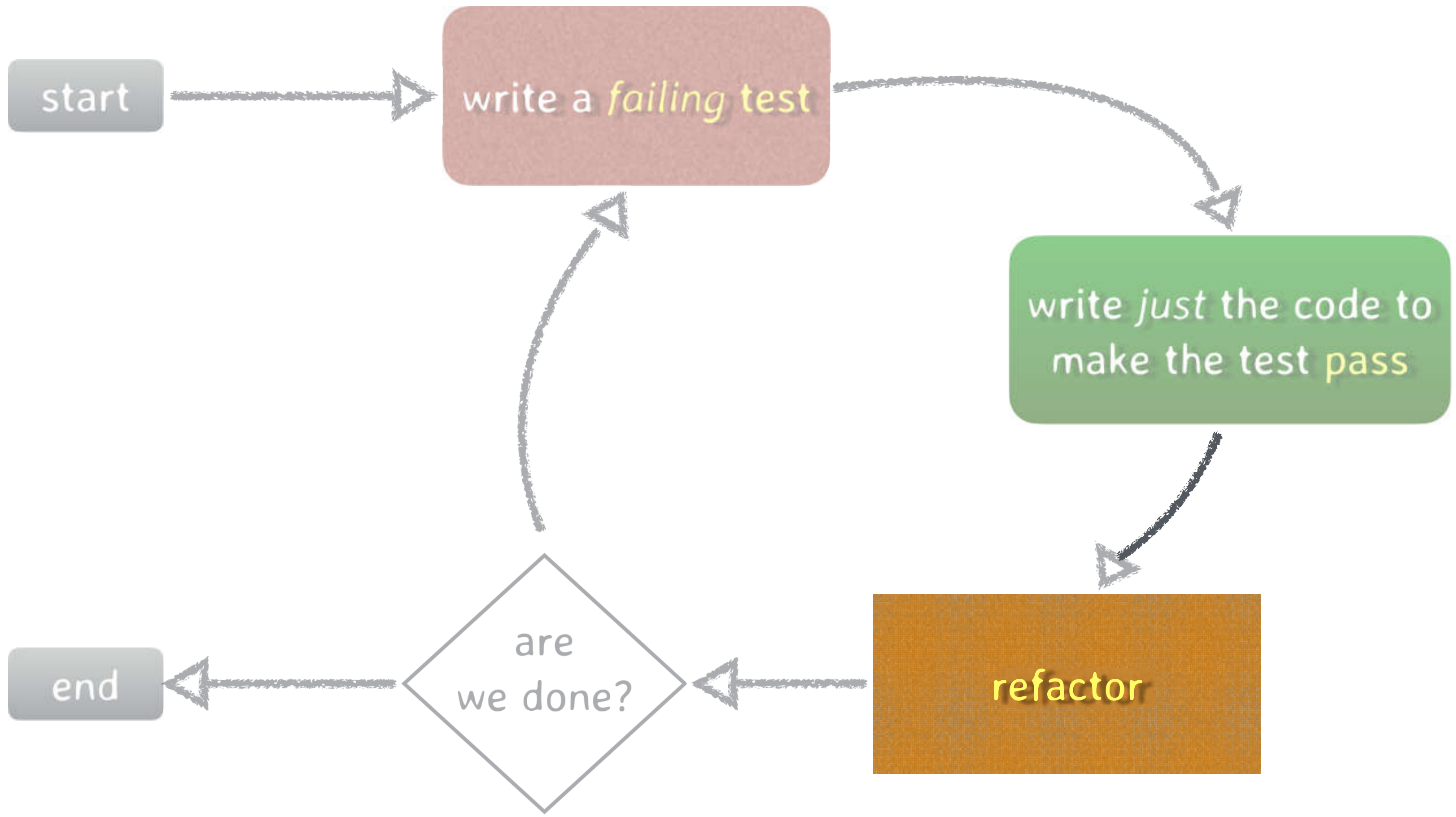
Test Driven Design


start
write a *failing* test
write *just* the code to make the test pass
refactor
are we done?
end

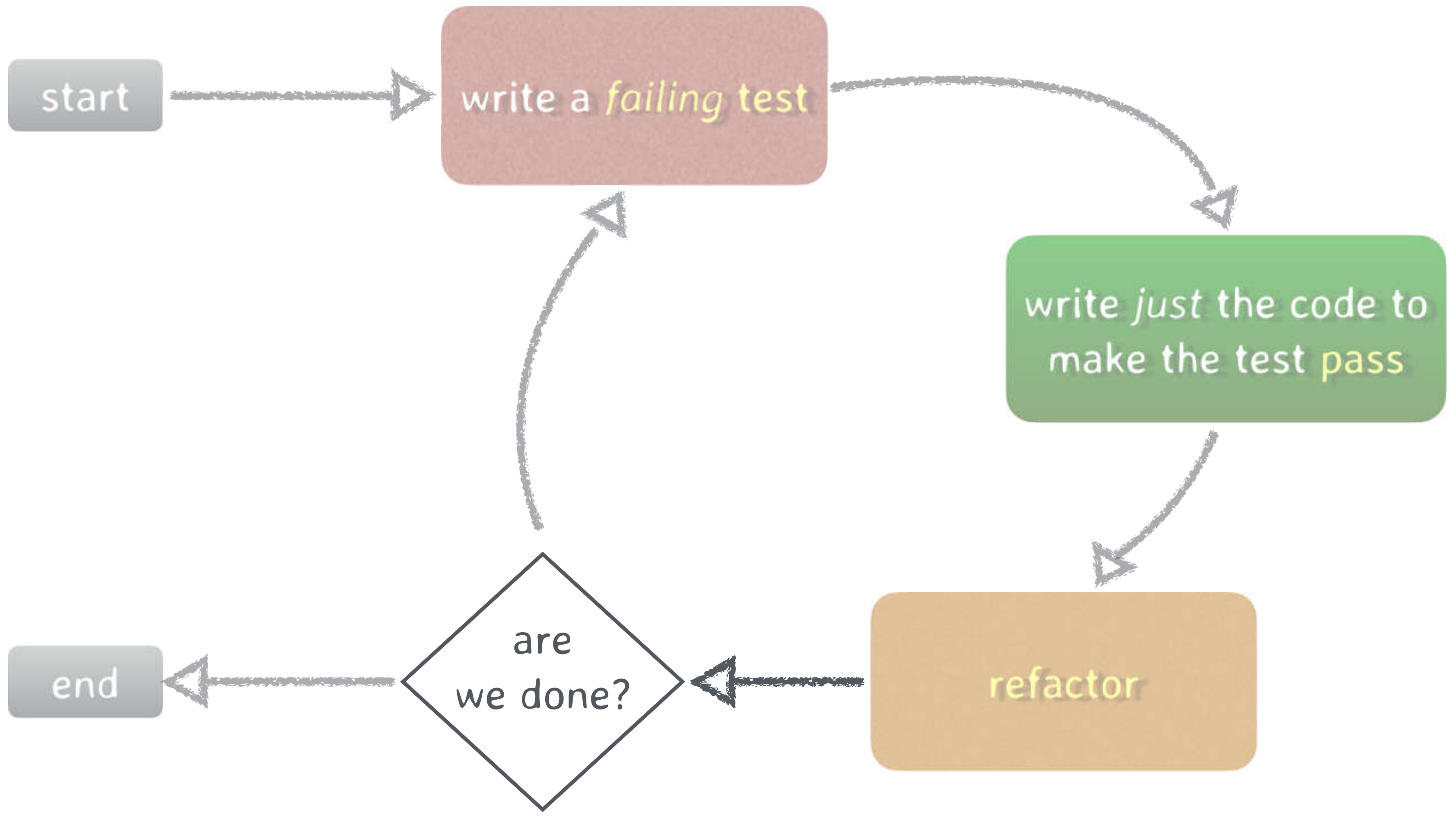

start
write a *failing* test
write *just* the code to make the test pass
refactor
are we done?
end

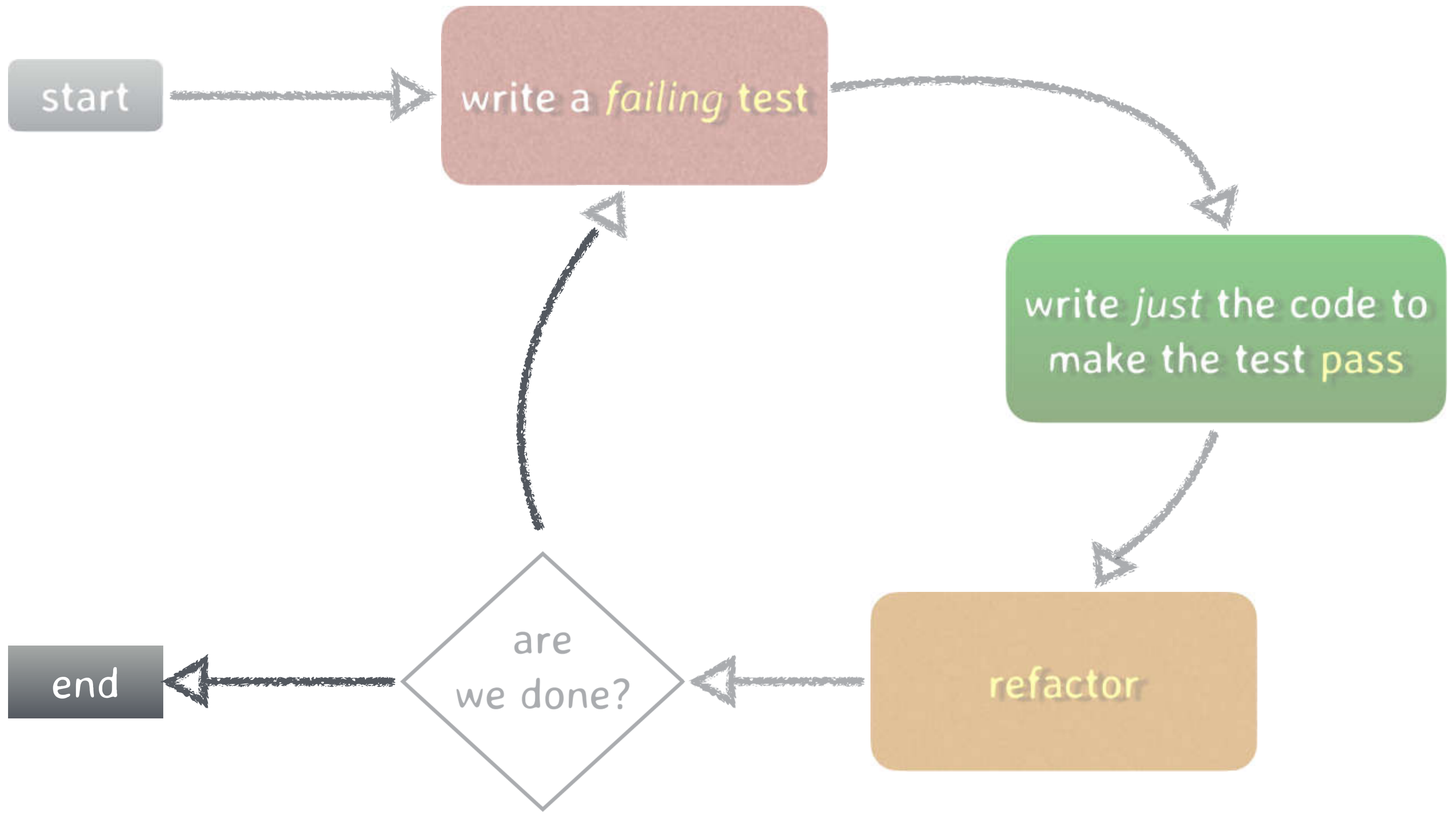

start
write a *failing* test
write *just* the code to make the test pass
refactor
are we done?
end

CPP-Summit

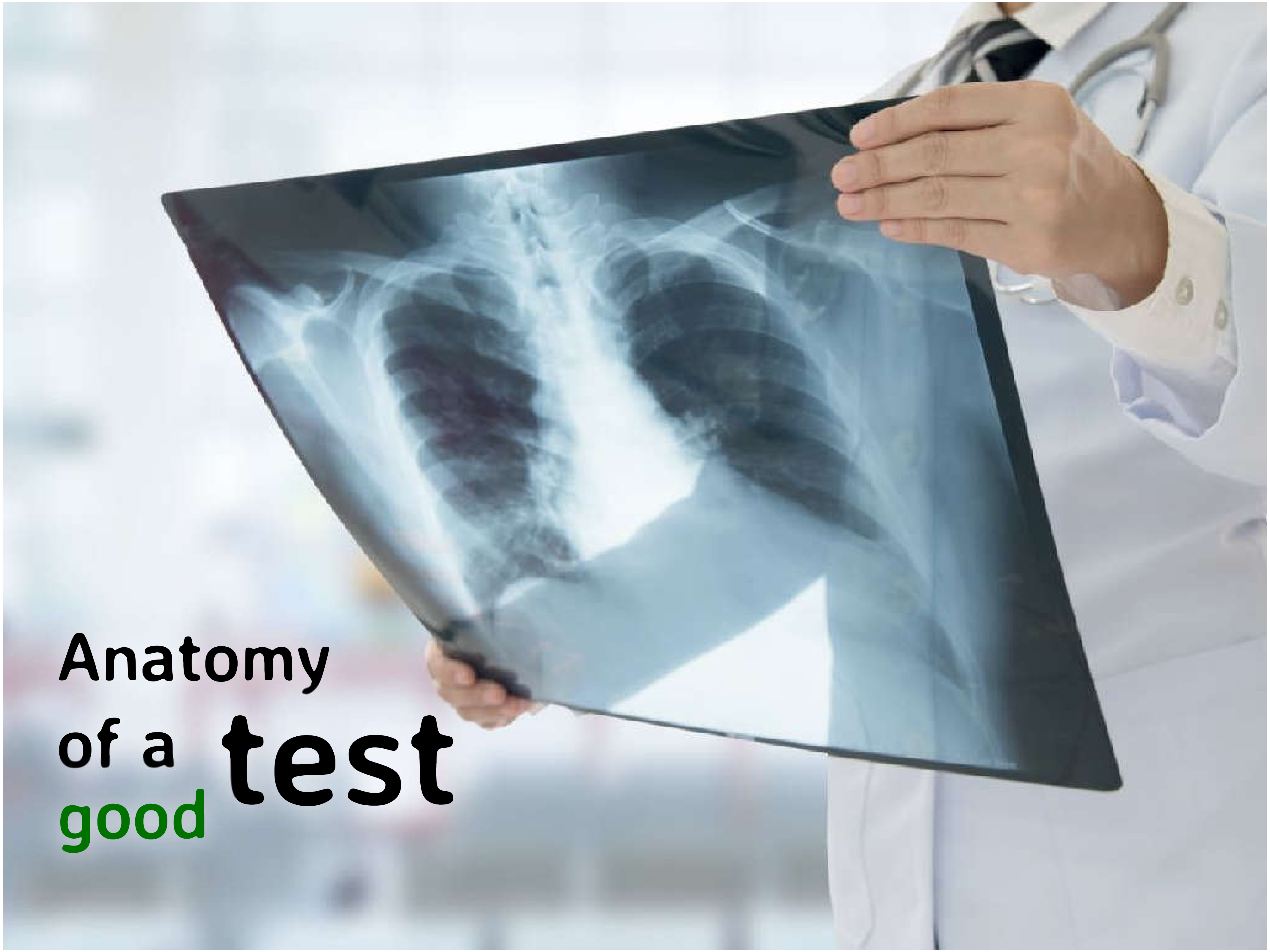

start
write a *failing* test
write *just* the code to make the test pass
refactor
are we done?
end

CPP-Summit


start
write a *failing* test
write *just* the code to make the test pass
refactor
are we done?
end


start
write a *failing* test
write *just* the code to make the test pass
refactor
are we done?
end

Questions?



Anatomy of a good test

```
TEST_CASE("add() returns the sum of its arguments") {
   REQUIRE( add( 1, 2 ) == 3 );
}
```

```
TEST_CASE("add() returns the sum of its arguments") {
   REQUIRE( add( 1, 2 ) == 3 );
}
```

Tests should have a good name

```
TEST_CASE("add() returns the sum of its arguments") {
    REQUIRE( add( 1, 2 ) == 3 );
}
```

```cpp
TEST_CASE( "Most recently used list" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

```cpp
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

CPP-Summit

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

STATE EXPECTATIONS

Unit Tests should have a regular structure

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

CPP-Summit

Unit Tests should have a regular structure

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

"Arrange"

CPP-Summit

Unit Tests should have a regular structure

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

"Arrange"

"Act"

CPP-Summit

Unit Tests should have a regular structure

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

"Arrange"

"Act"

"Assert"

Tests should:

Have a good name

State expectations

Unit Tests should have a regular structure

"Arrange" "Act" "Assert"

Tests should:

Have a good name

State expectations

Unit Tests should have a regular structure

"Arrange" "Act" "Assert"

Have a single "Logical" Assert

CPP-Summit

Tests should: Have a single "Logical" Assert

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "An empty list has no elements" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );
    }
    SECTION( "Adding to an empty list increases the size to 1" ) {
        list.add("item1");
        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

Single assert

TESTS SHOULD: HAVE A SINGLE "LOGICAL" ASSERT

```
TEST_CASE( "An MRU list acts like a stack, "
           "but duplicate entries replace existing ones" ) {

    MRUList<std::string> list;

    SECTION( "…?" ) {
        REQUIRE( list.empty() );
        REQUIRE( list.size() == 0 );

        list.add("item1");

        REQUIRE( list.empty() == false );
        REQUIRE( list.size() == 1 );
    }
}
```

DEPENDENCY

MULTIPLE ASSERTS

Tests should:

Have a good name

State expectations

Unit Tests should have a regular structure

“Arrange” “Act” “Assert”

Have a single “Logical” Assert

CPP-Summit

Tests should:

Have a good name

State expectations

Use Public Interface

Unit Tests should have a regular structure

"Arrange" → "Act" → "Assert"

Have a single "Logical" Assert

CPP-Summit

Tests should: Use Public Interface

Q. How do you test private methods?

TESTS SHOULD: USE PUBLIC INTERFACE

Q. How do you test private methods?

A. You don't

Tests should: Use Public Interface

Q. How do you test private methods?

A. You don’t

Q. But what if you really need to?

CPP-Summit

Tests should: Use Public Interface

```
friend class TestAccess;
```

Tests should: Use Public Interface

```
friend class TestAccess;
```

To test “non-functional” requirements

(ref counts, caches etc)

Tests should: Use Public Interface

```
friend class TestAccess;

extern name_not_in_header;
```

Tests should: Use Public Interface

```
friend class TestAccess;

extern name_not_in_header;

#define private public;
```

Tests should:

Have a good name

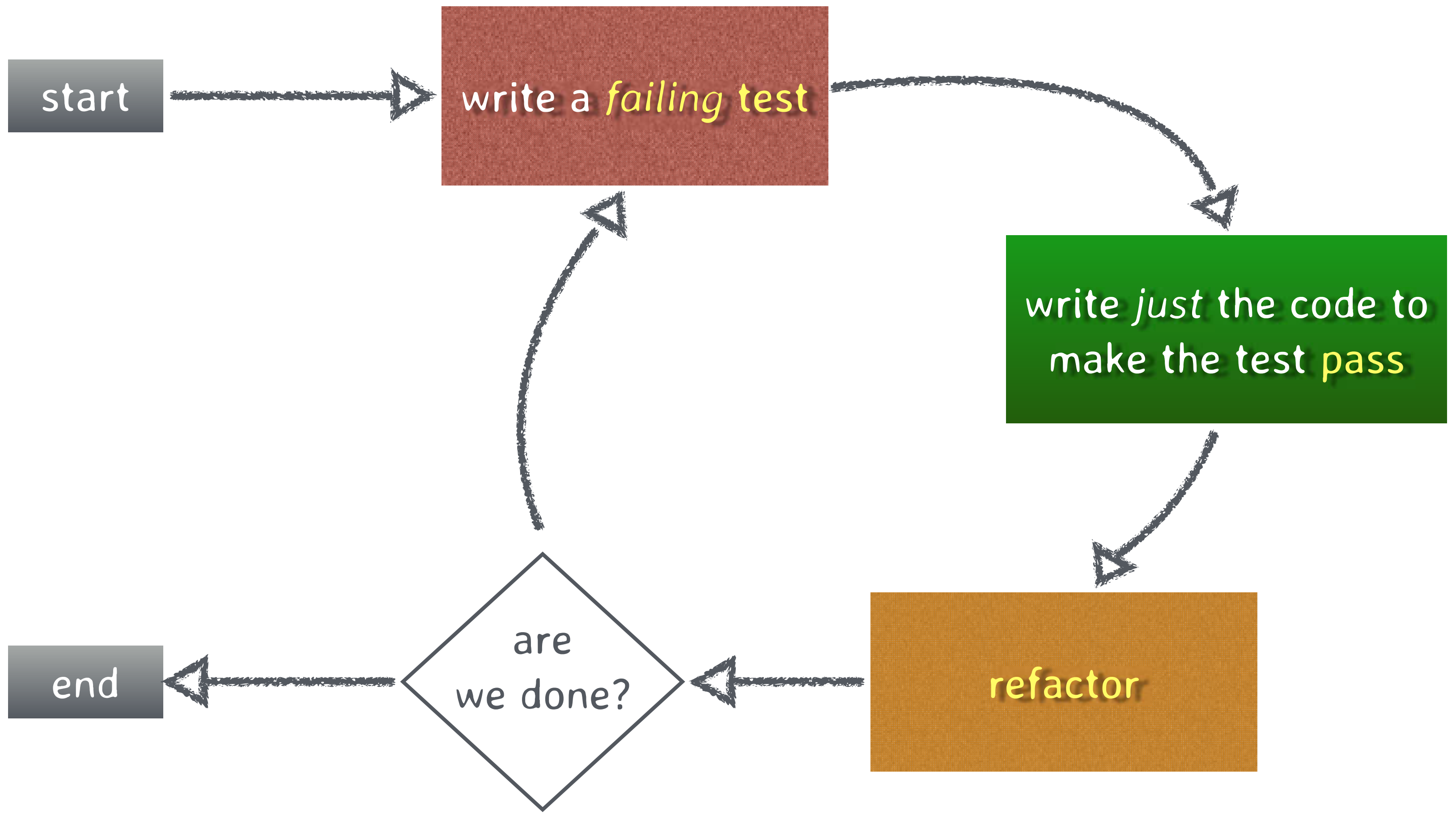
State expectations

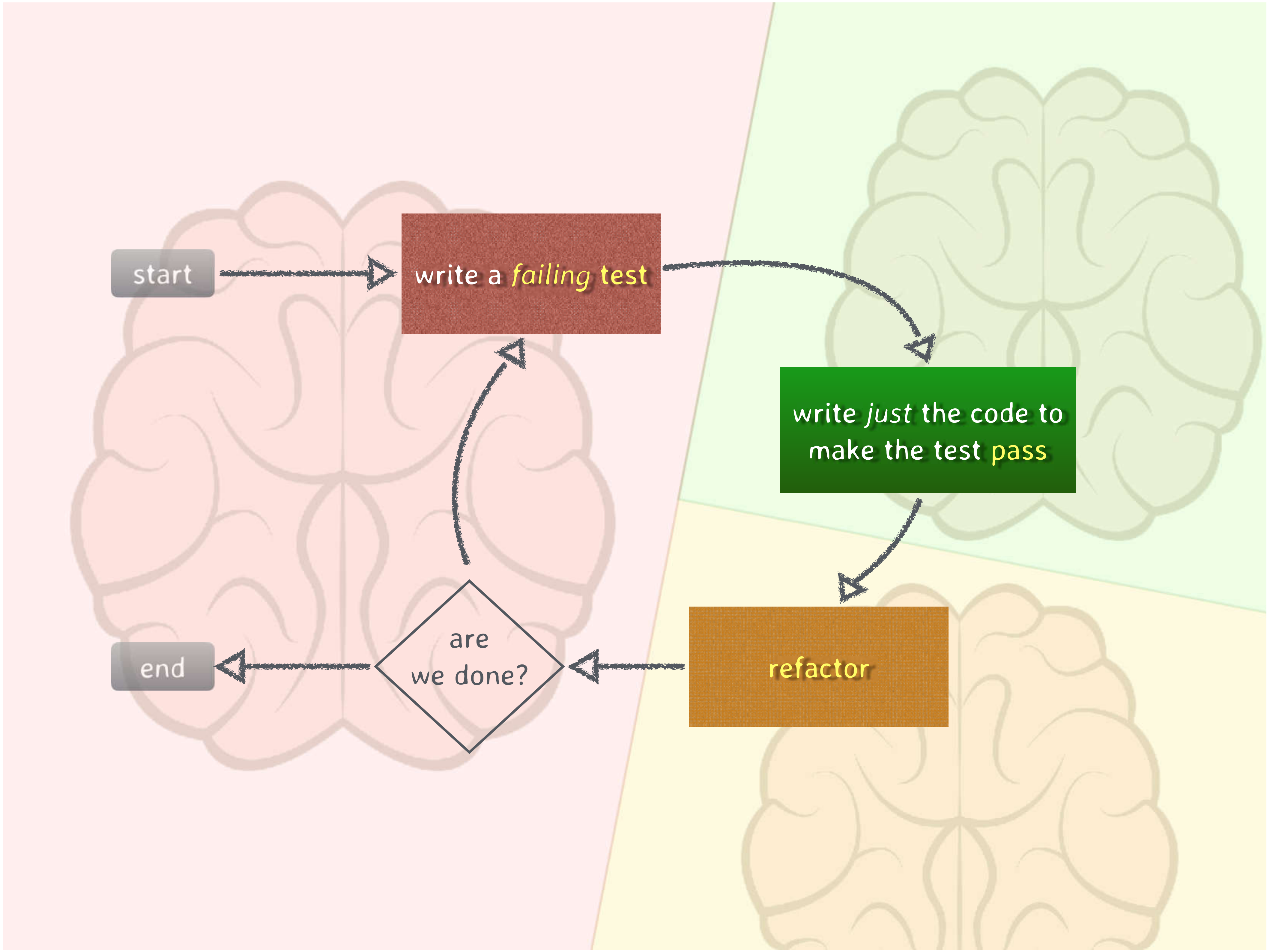
Use Public Interface

Unit Tests should have a regular structure

“Arrange” “Act” “Assert”

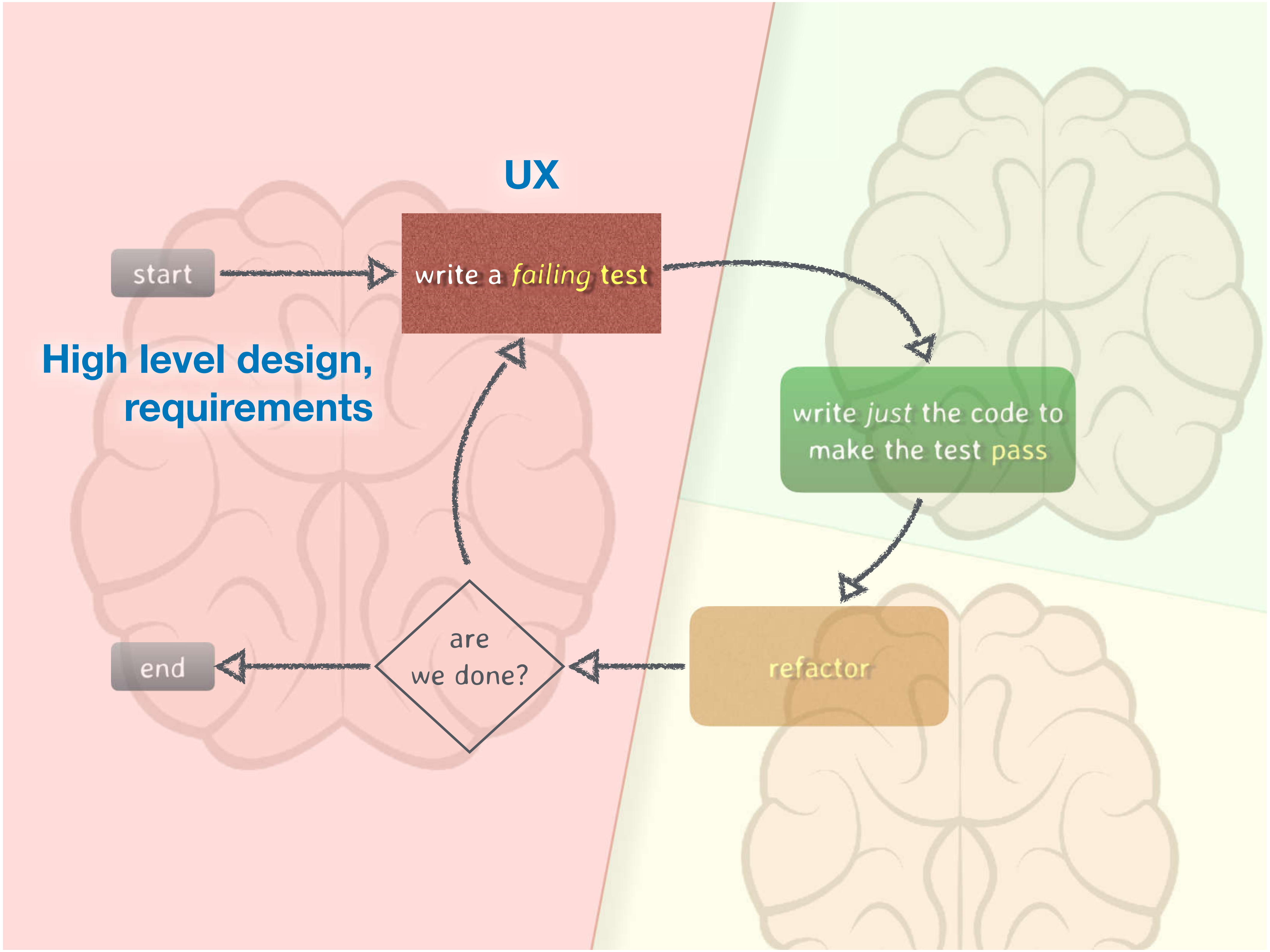
Have a single “Logical” Assert

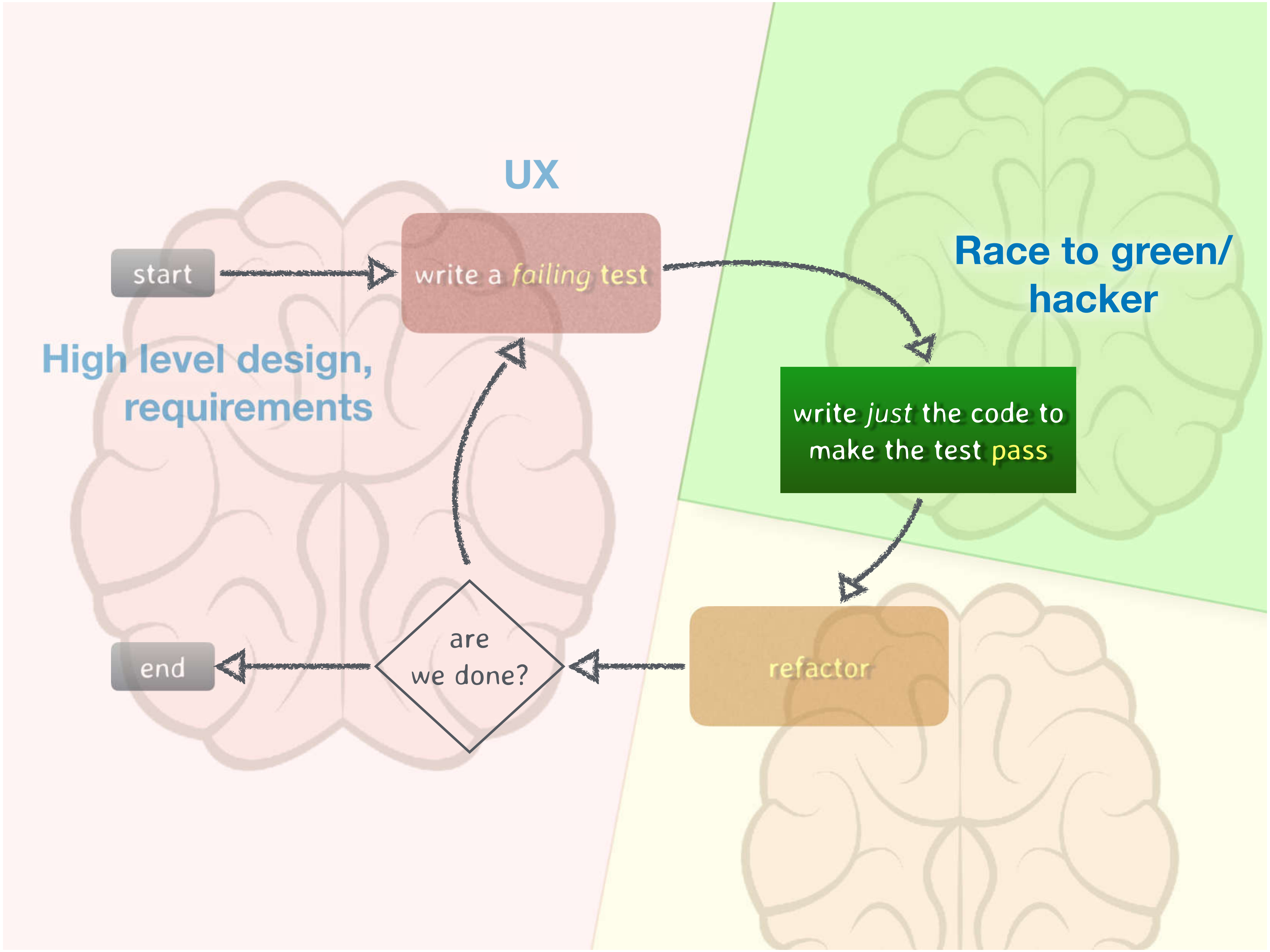

start
write a *failing* test
write *just* the code to make the test pass
refactor
are we done?
end

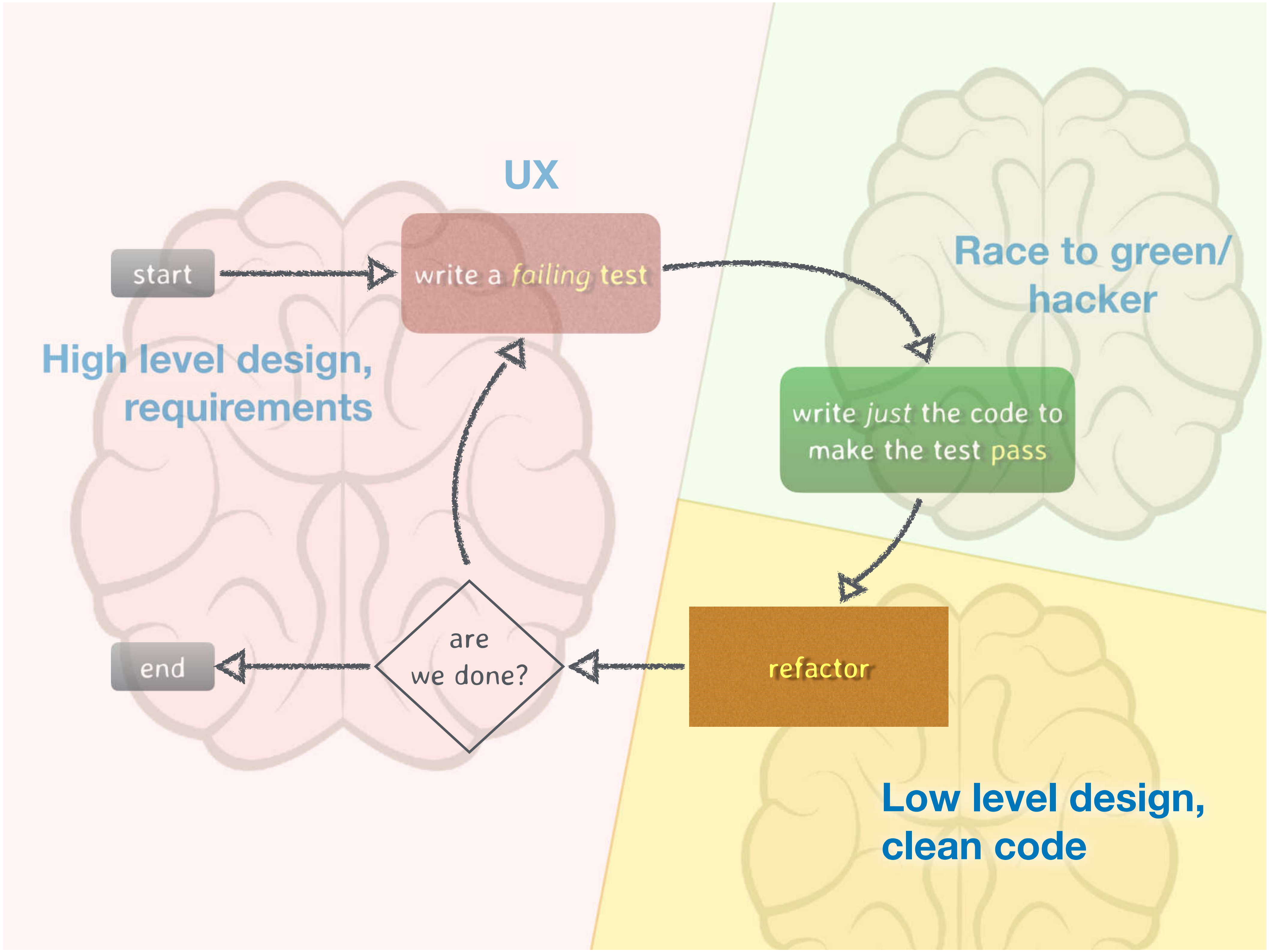

start
write a *failing* test
write *just* the code to make the test pass
refactor
are we done?
end

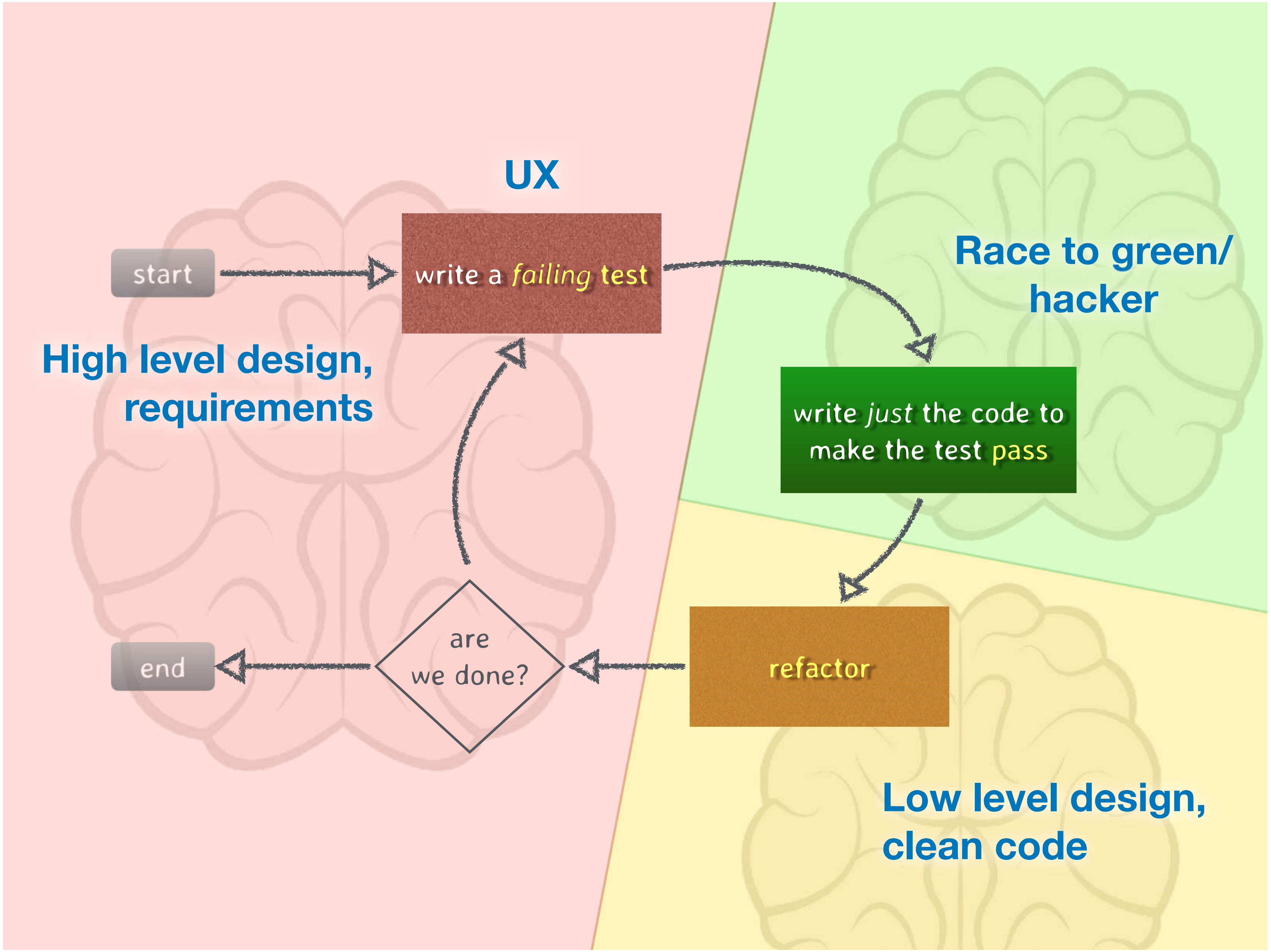
UX

High level design, requirements




UX
High level design, requirements
Race to green/ hacker
start
write a failing test
write just the code to make the test pass
refactor
are we done?
end


UX
start
write a failing test
High level design, requirements
Race to green/ hacker
write just the code to make the test pass
refactor
are we done?
end
Low level design, clean code


UX
start
write a *failing* test
Race to green/ hacker
High level design, requirements
write *just* the code to make the test pass
refactor
are we done?
end
Low level design, clean code

Property Based Testing

Mutation Testing

Legacy Code

Design Principles

Test First
Test Better

@phil_nash