Conduite de Projet
Cours 8 — Test-Driven Development

Stefano Zacchiroli
zack@pps.univ-paris-diderot.fr

Laboratoire IRIF, Université Paris Diderot

2015-2016

URL http://upsilon.cc/zack/teaching/1516/cproj/
Copyright © 2013-2016 Stefano Zacchiroli
License Creative Commons Attribution-ShareAlike 4.0 International License
http://creativecommons.org/licenses/by-sa/4.0/deed.en_US

Outline

1. Development processes and testing
2. xUnit & Check
3. Test-Driven Development
4. Bootstrapping TDD

Test-Driven Development (TDD)

Test-Driven Development, or TDD, is an iterative software development process which uses very short development cycles and leverages tests to provide constant feedback to software developers.

Goal: “clean code that works”, i.e., develop better software, less stressfully.

Test-Driven Development (TDD)

Test-Driven Development, or TDD, is an iterative software development process which uses very short development cycles and leverages tests to provide constant feedback to software developers.

Goal: "clean code that works", i.e., develop better software, less stressfully.

The "Test-Driven Development" expression is often (ab)used to talk about 2 distinct things:

- the TDD development process
- the xUnit family of testing frameworks
  - e.g., JUnit, [Python] unittest, cppUnit, OUnit, NUnit, PHPUnit, ...

  which have been designed to support the TDD development process, but can also be used when adopting different development process

Test-Driven Development (TDD)

Test-Driven Development, or TDD, is an iterative software development process which uses very short development cycles and leverages tests to provide constant feedback to software developers.

Goal: “clean code that works”, i.e., develop better software, less stressfully.

The “Test-Driven Development” expression is often (ab)used to talk about 2 distinct things:

- the TDD development process
- the xUnit family of testing frameworks
  - e.g., JUnit, [Python] unittest, cppUnit, OUnit, NUnit, PHPUnit, ...

  which have been designed to support the TDD development process, but can also be used when adopting different development process

We’ll cover both in this lecture.

Outline

1. Development processes and testing
2. xUnit & Check
3. Test-Driven Development
4. Bootstrapping TDD

Reminder — Development process

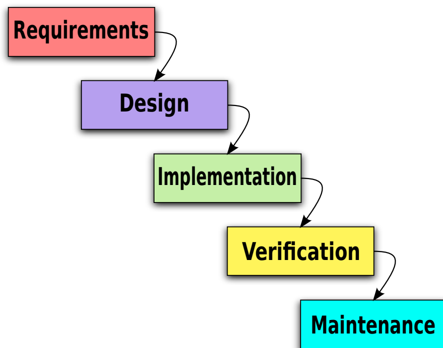
Definition (Software development process)

A *software development process* is a structured set of activities which lead to the production of some software.

Some software development activities:

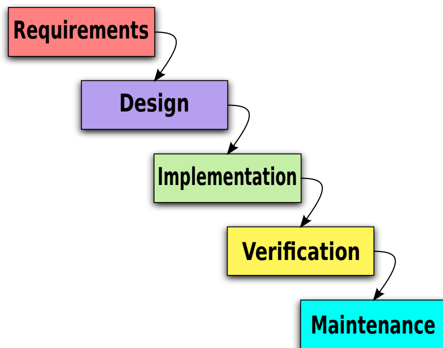
1. requirement specification
2. design
3. implementation
4. verification
5. maintenance

Reminder — Waterfall model



https://en.wikipedia.org/wiki/File:Waterfall_model_(1).svg

Reminder — Waterfall model

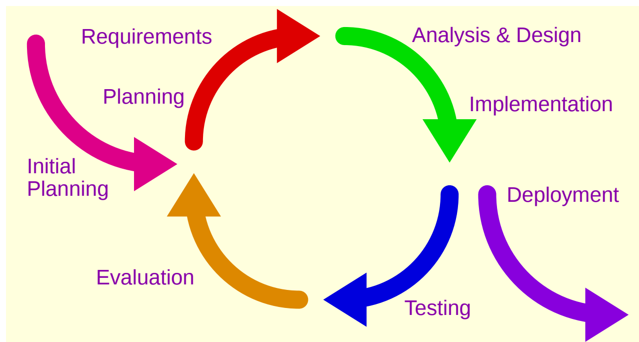


https://en.wikipedia.org/wiki/File:Waterfall_model_(1).svg

- that's the theory
- in practice:
  - feedback loops
  - increasingly more costly
  - that need to be prevented

Reminder — Iterative development

A family of models where development happens incrementally, through repeated iterations of development activities.

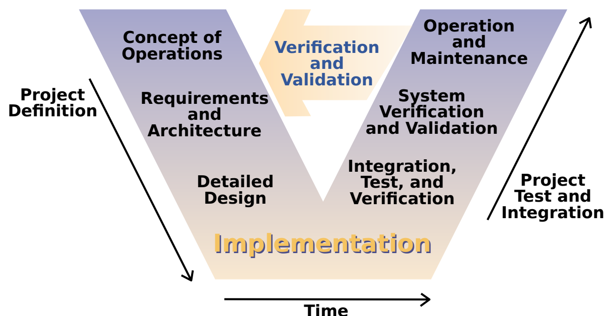


https://commons.wikimedia.org/wiki/File:Iterative_development_model.svg

Key benefit: feedback loop.

V-Model

Not really a software development model.



https://en.wikipedia.org/wiki/File:Systems_Engineering_Process_II.svg

Rather a (simplistic) view on the waterfall model that correlates the initial “definition” phases with the final “delivery” ones.

The V-model helps to think about test purposes.

A hierarchy of tests

Disclaimers:

- there are other hierarchies/taxonomies, on different angles
- terminology is not clear cut (as it often happens in SWE)
- the granularity trend—from small to big—however matters and is agreed upon

Test hierarchy

acceptance	*Does the whole system work?*
integration	*Does our code work against (other) code (we can't change)?*
unit	*Do our code units[a] do the right thing and are convenient to work with?*

a. in a broad sense: might be classes, objects, modules, etc. depending on the available abstraction mechanisms

Acceptance test

Does the whole system work?

Acceptance tests represent features that the system should have. Both their lack and their misbehaviour imply that the system is not working as it should. Intuition:

- 1 feature → 1+ acceptance test(s)
- 1 user story → 1+ acceptance test(s) (when using user stories)

Exercise (name 2+ acceptance tests for this "user login" story)

After creating a user, the system will know that you are that user when you login with that user's id and password; if you are not authenticated, or if you supply a bad id/password pair, or other error cases, the login page is displayed. If a CMS folder is marked as requiring authentication, access to any page under that folder will result in an authentication check. http://c2.com/cgi/wiki?AcceptanceTestExamples

Preview: we will use acceptance tests to guide feature development

Integration test

Does our code work against (other) code (we can't change)?

"Code we can't change" =

- 3rd party libraries/framework
  - be them proprietary or Free/Open Source Software
- code developed by other teams that we don't "own"
  - (strict code ownership is bad, though)
- code that we do not want/cannot modify in the current phase of development, for whatever reason

Example

- our `BankClient` should not call the `getBalance` method on `BankingService` before calling `login` and having verified that it didn't throw an exception
- `xmlInitParser` should be called before any other parsing function of `libxml2`
- the DocBook markup returned by `CMSEditor.save` should be parsable by `PDFPublisher`'s constructor

Unit test

Do our code units do the right thing and are convenient to work with?

Before implementing any unit of our software, we have (to have) an idea of what the code should do. Unit tests show convincing evidence that—in a limited number of cases—it is actually the case. [1]

Example (some unit tests for a `List` module)

1. remember: tests reveal bugs, but don't prove their absence!

Unit test

Do our code units do the right thing and are convenient to work with?

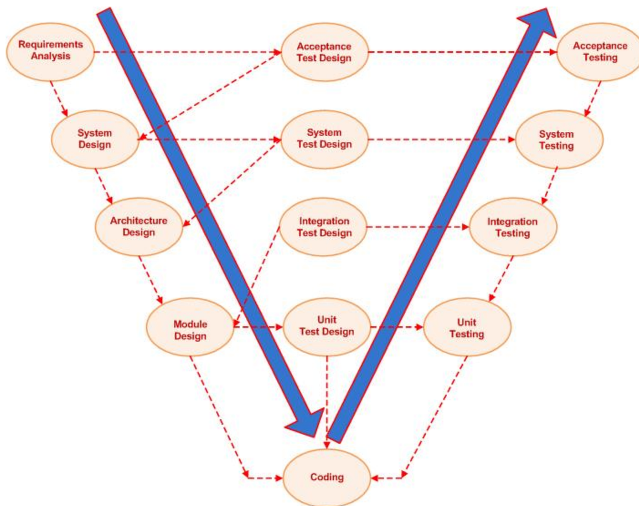
Before implementing any unit of our software, we have (to have) an idea of what the code should do. Unit tests show convincing evidence that—in a limited number of cases—it is actually the case. [1]

Example (some unit tests for a `List` module)

- calling `List.length` on an empty list returns 0
- calling `List.length` on a singleton list returns 1
- calling `List.last` after `List.append` returns the added element
- calling `List.head` on an empty list throws an exception
- calling `List.length` on the concatenation of two lists returns the sum of the respective `List.length`s
- ...

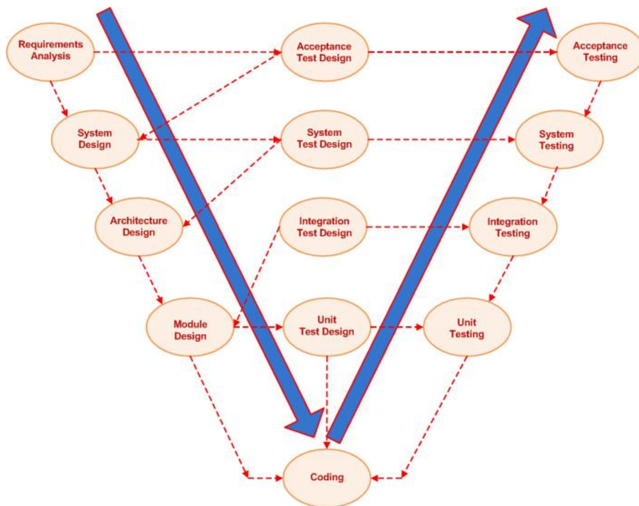
1. remember: tests reveal bugs, but don't prove their absence!

Tests in the V-Model



https://en.wikipedia.org/wiki/File:V-model.JPG

Tests in the V-Model



https://en.wikipedia.org/wiki/File:V-model.JPG

For TDD we will “hack” unit, integration, acceptance tests, and use them in an arguably more clever way. . .

Outline

1. Development processes and testing
2. xUnit & Check
3. Test-Driven Development
4. Bootstrapping TDD

xUnit

xUnit collectively refers to a set of frameworks for automated unit testing which share a common test coding style.

Each xUnit framework includes:

- **test case** abstraction used to define tests
- **test suite** abstraction used to organize test in test suites
- **assertion API** to implement test case oracles to verify outcomes
- **test fixture** mechanisms to factorize test initialization and clean up code
- **test runner** end-user program to discover and (selectively) run test suites
- **test result formatters** summarize test results and present them for human consumption

xUnit implementations

Teting frameworks that adhere to the xUnit paradigm exist for most languages and platforms, e.g.:[2]

- SUnit (Smalltalk)
- JUnit (Java)
- CppUnit (C++)
- OUnit (OCaml)
- Test::Unit (Ruby)
- HUnit (Haskell)
- NUnit (.NET)
- unittest (Python)
- Check (C)

2. https://en.wikipedia.org/wiki/List_of_unit_testing_frameworks

Check

Check[3] is one of the most well-known unit testing framework for the C programming language.

Features overview

- C library
- xUnit style
- fixtures
- address space isolation
- autotools integration
- ~~mocking~~

References

- API reference: https://libcheck.github.io/check/doc/doxygen/html/check_8h.html
- source code (LGPL): https://github.com/libcheck/check/

Check examples in the following slides have been adapted from the Check manual. Copyright © 2001–2014 Arien Malec, Branden Archer, Chris Pickett, Fredrik Hugosson, and Robert Lemmen. License: GNU GFDL, version 1.2 or any later version.

3. https://libcheck.github.io/check/

Code organization for testing

Logical organization

- structure the code to be tested as a library...
- ...with a well-defined API
- corollary: almost empty `main()`, that just calls the main API entry point

> Unit testing encourages to think at your API early in the project life-cycle, as your unit tests become your first client code.

Physical organization

Up to you, but typically:

- `src/` (top-level dir): library code + `main()`
- `tests/` (top-level dir): Check tests
  - **#include** "../src/mylib.h" or equivalent

SUT — money.h

```
#ifndef MONEY_H
#define MONEY_H

typedef struct Money Money;

Money *money_create(int amount, char *currency);
void money_free(Money * m);

int money_amount(Money * m);
char *money_currency(Money * m);

#endif /* MONEY_H */
```

SUT — money.c I

```c
#include <stdlib.h>
#include "money.h"


struct Money {
        int amount;
        char *currency;
};

void money_free(Money *m) {
        free(m);
        return;
}
```

SUT — money.c II

```
Money *money_create(int amount, char *currency) {
        Money *m;

        if (amount < 0)
                return NULL;

        m = malloc(sizeof(Money));
        if (m == NULL)
                return NULL;

        m->amount = amount;
        m->currency = currency;

        return m;
}
```

SUT — money.c III

```c
int money_amount(Money *m) {
        return m->amount;
}

char *money_currency(Money *m) {
        return m->currency;
}
```

Unit test skeleton

- the smallest units of executable tests in Check are unit tests
- unit tests are defined in regular .c files, using suitable preprocessor macros

```
#include <check.h>

START_TEST (test_name)
{
    /* unit test code */
}
END_TEST
```

Exercise

find the macro definitions of START_TEST and END_TEST and describe what they do

Hello, world

```
#include <check.h>  // testing framework
#include "../src/money.h"  // SUT

START_TEST(test_money_create) {
    Money *m;  // setup

    m = money_create(5, "USD");  // exercise SUT

    // test oracle
    ck_assert_int_eq(money_amount(m), 6);
    ck_assert_str_eq(money_currency(m), "USD");

    money_free(m);  // clean up
}
END_TEST
```

Assertion API

For basic data types, pre-defined “typed” assertions are available and can be used as simple and readable test oracles:

ck_assert_int_eq
: asserts that two signed integers values are equal; display a suitable error message upon failure

ck_assert_int_{ne,lt,le,gt,ge}
: like ck_assert_int_eq, but using different comparison operators

ck_assert_uint_*
: like ck_assert_int_*, but for unsigned integers

Assertion API

For basic data types, pre-defined "typed" assertions are available and can be used as simple and readable test oracles:

ck_assert_int_eq
: asserts that two signed integers values are equal; display a suitable error message upon failure

ck_assert_int_{ne,lt,le,gt,ge}
: like ck_assert_int_eq, but using different comparison operators

ck_assert_uint_*
: like ck_assert_int_*, but for unsigned integers

ck_assert_str_*
: like ck_assert_int_*, but for `char *` string values, using `strcmp()` for comparisons

Assertion API

For basic data types, pre-defined "typed" assertions are available and can be used as simple and readable test oracles:

ck_assert_int_eq
: asserts that two signed integers values are equal; display a suitable error message upon failure

ck_assert_int_{ne,lt,le,gt,ge}
: like ck_assert_int_eq, but using different comparison operators

ck_assert_uint_*
: like ck_assert_int_*, but for unsigned integers

ck_assert_str_*
: like ck_assert_int_*, but for `char *` string values, using `strcmp()` for comparisons

ck_assert_ptr_{eq,ne}
: like ck_assert_int_*, but for `void *` pointers

Assertion API (cont.)

For other data types, you can cook your own test oracles on top of more basic assertion primitives:

ck_assert fails test if supplied condition evaluates to false

ck_assert_msg `ck_assert` + displays user provided message

Assertion API (cont.)

For other data types, you can cook your own test oracles on top of more basic assertion primitives:

ck_assert fails test if supplied condition evaluates to false

ck_assert_msg `ck_assert` + displays user provided message

ck_abort unconditionally fails test with predefined message

ck_abort_msg unconditionally fails with user supplied message

Assertion API — examples

```
ck_assert_str_eq(money_currency(m), "USD");
```

is equivalent to the following alternative formulations:

```
ck_assert(strcmp(money_currency(m), "USD") == 0);

ck_assert_msg(strcmp(money_currency(m), "USD") == 0,
    "Was expecting a currency of USD, but found %s",
    money_currency(m));

if (strcmp(money_currency(m), "USD") != 0)
  ck_abort_msg("Currency not set correctly on creation");
```

Assertion API — examples (cont.)

```
ck_assert(money_amount(m) == 5);
```

is equivalent to:

```
ck_assert_msg(money_amount(m) == 5, NULL);
```

and, if money_amount(m) != 5, will automatically synthesize the message:

```
"Assertion 'money_amount(m) == 5' failed"
```

Test suites

In Check terminology:

- unit tests are grouped into test cases
- test cases are grouped into test suites

Test suites are what you can ask a test runner to run for you, recursively down to individual unit tests.

Example:

```
START_TEST(test_money_create) {
    // as before
}
END_TEST

// other unit tests here
```

Test suites (cont.)

```c
Suite *money_suite(void) {
    TCase *tc_core;
    Suite *s;

    tc_core = tcase_create("Core");
    tcase_add_test(tc_core, test_money_create);
    // tcase_add_test(tc_core, test_foo);
    // ...

    s = suite_create("Money");
    suite_add_tcase(s, tc_core);
    // suite_add_tcase(s, tc_bar);
    // ...

    return s;
}
```

Suite runner

In Check terminology, a suite runner is responsible for recursively running all unit tests reachable from a (set of) suite(s).

```
// create a suite runner, including a single suite
SRunner *srunner_create(Suite *);

// add a suite to a suite runner
void srunner_add_suite (SRunner *sr, Suite *s)

// destroy a suite runner
void srunner_free(SRunner *);
```

Suite runner (cont.)

```
// run all unit tests reachable from all suites
void srunner_run_all(SRunner *sr,
                     enum print_output print_mode);

// run the unit tests corresponding to suite sname and
// test case tcname. Either can be NULL to mean "all"
void srunner_run(SRunner *sr,
                 const char *sname,
                 const char *tcname,
                 enum print_output print_mode);
```

`print_output` controls the on-screen output of the test runner:

- **CK_SILENT** no output
- **CK_MINIMAL** summary output
- **CK_NORMAL** summary + list of failed tests
- **CK_VERBOSE** summary + list of all tests
- **CK_ENV** deduct from env CK_VERBOSITY (default: CK_NORMAL)

Suite runner (cont.)

After tests have been run, test result information can be extracted from the suite runner.

srunner_ntests_failed number of failed tests

srunner_ntests_run number of tests ran

srunner_failures

srunner_results

access to detailed, per test result information (see API reference)

Suite runner (cont.)

```
int main(void) {
    int failures;
    Suite *s;
    SRunner *sr;

    s = money_suite();
    sr = srunner_create(s);

    srunner_run_all(sr, CK_VERBOSE);
    failures = srunner_ntests_failed(sr);
    srunner_free(sr);

    return (failures == 0) ? EXIT_SUCCESS : EXIT_FAILURE;
}
```

Suite runner (cont.)

```
./check_money
Running suite(s): Money
100%: Checks: 1, Failures: 0, Errors: 0
check_money.c:13:P:Core:test_money_create:0: Passed
```

Demo

More tests

```
START_TEST(test_money_create_neg) {
    Money *m = money_create(-1, "USD");

    ck_assert_msg(m == NULL,
                  "NULL should be returned on attempt "
                  "to create with a negative amount");
}
END_TEST

START_TEST(test_money_create_zero) {
    Money *m = money_create(0, "USD");

    if (money_amount(m) != 0)
        ck_abort_msg("0 is a valid amount of money");
}
END_TEST
```

More tests (cont.)

```
Suite *money_suite(void) {
    TCase *tc_core, *tc_limits;
    Suite *s;

    tc_core = tcase_create("Core");
    tcase_add_test(tc_core, test_money_create);

    tc_limits = tcase_create("Limits");
    tcase_add_test(tc_limits, test_money_create_neg);
    tcase_add_test(tc_limits, test_money_create_zero);

    s = suite_create("Money");
    suite_add_tcase(s, tc_core);
    suite_add_tcase(s, tc_limits);

    return s;
}
```

More tests (cont.)

```
./check_money
Running suite(s): Money
100%: Checks: 1, Failures: 0, Errors: 0
check_money.c:13:P:Core:test_money_create:0: Passed
check_money.c:24:P:Limits:test_money_create_neg:0: Passed
check_money.c:28:P:Limits:test_money_create_zero:0: Passed
```

Compiling with Check

src/Makefile

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = $(CFLAGS)

all: main
main: main.o money.o money.h
money.o: money.h

clean:
	rm -f main *.o
```

Exercise

Where are defined the actual compilation commands?

Compiling with Check (cont.)

tests/Makefile

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = $(CFLAGS)
LDLIBS = -Wall $(shell pkg-config --libs check)

all: check_money
check_money: check_money.o ../src/money.o

clean:
	rm -f check_money *.o
```

Exercise

Where are defined the actual compilation commands?

Compiling with Check (cont.)

```
$ cd src/
src$ make
gcc -Wall -c -o main.o main.c
gcc -Wall -c -o money.o money.c
gcc -Wall main.o money.o money.h -o main

src$ cd ../tests
tests$ make
gcc -Wall-c -o check_money.o check_money.c
gcc -Wall check_money.o ../src/money.o -lcheck_pic -pthread \
    -lrt -lm -lsubunit -o check_money

tests$ ./check_money
Running suite(s): Money
100%: Checks: 1, Failures: 0, Errors: 0
check_money.c:13:P:Core:test_money_create:0: Passed

tests$
```

Unit testing C: memory safety

```c
#include <check.h>

START_TEST(null_deref) {
    int *p = NULL;
    ck_assert_int_eq(p[1], 42);
}
END_TEST

int main(void) {
        TCase *tc;  Suite *s;  SRunner *sr;

        tc = tcase_create("segfault");
        tcase_add_test(tc, null_deref);
        s = suite_create("memsafety");
        suite_add_tcase(s, tc);
        sr = srunner_create(s);
        srunner_run_all(sr, CK_VERBOSE);
        return (srunner_ntests_failed(sr) == 0 ? 0 : 1);
}       // what will this program do?
```

Unit testing C: memory safety (cont.)

```
$ ./test-segfault
Running suite(s): memsafety
0%: Checks: 1, Failures: 0, Errors: 1
test-segfault.c:3:E:segfault:null_deref:0:
  (after this point) Received signal 11 (Segmentation fault)

$ echo $?
$ 1
```

- the program did not crash
- Check reported a test failure and “detected” the segfault
- how come?

Address space separation

- unit testing C might be difficult in general because all tests are run in the same address space
- if a test induces memory corruption, *other* tests will suffer the consequences too (including crashes)

Address space separation

- unit testing C might be difficult in general because all tests are run in the same address space
- if a test induces memory corruption, *other* tests will suffer the consequences too (including crashes)
- as a way around, several C test frameworks run tests in separate processes and address spaces, `fork()`-ing before each test

Address space separation

- unit testing C might be difficult in general because all tests are run in the same address space
- if a test induces memory corruption, *other* tests will suffer the consequences too (including crashes)
- as a way around, several C test frameworks run tests in separate processes and address spaces, `fork()`-ing before each test
- by default Check runs each unit test in a separate process ("fork mode")
- "no fork mode" can be requested explicitly
  - define the environment variable CK_FORK=no
  - ```
    void srunner_set_fork_status(SRunner *,
                                 enum fork_status);
    ```

    with `fork_status` = CK_FORK / CK_NOFORK
    ```

Check (no) fork mode — example

```
$ ./test-segfault
Running suite(s): memsafety
0%: Checks: 1, Failures: 0, Errors: 1
test-segfault.c:3:E:segfault:null_deref:0:
  (after this point) Received signal 11 (Segmentation fault)

$ echo $?
$ 1

$ CK_FORK=no ./test-segfault
Running suite(s): memsafety
[1]    5750 segmentation fault  CK_FORK=no ./test-segfault

$ echo $?
$ 139
```

- after disabling fork mode the program did crash :-(
- the suite runner would have been unable to run further tests in the suite

Test suite best practices

- often a group of tests should be run on the same initial state…
- …but tests execution might alter that state

1. we want test isolation: each test should behave the same no matter the test execution order *(dynamic requirement)*
   - each test should initialize all of its required state (setup)
   - each test should clean up after itself (tear down)
2. we also wish to not duplicate test initialization across tests, as it violates the DRY principle *(static requirement)*

Note: fork mode helps with (1), but not with (2).

We want a mechanism to factor out setup and tear down code across multiple tests.

Test fixtures

A test fixture, or test context, is a pair $\langle setup, teardown \rangle$ of functions to be executed before and after the test body.

- setup should create all the state needed to evaluate the test (i.e., exercising SUT + test oracle)
- teardown should clean up all the state affected by the test execution (i.e., setup + exercising SUT)

The code that implements text fixtures is independent from the actual test code.
Therefore, it can be shared across multiple tests.

Test fixtures in Check

In Check, test fixtures are associated with test cases.
They are hence shared among all unit tests of the same test case.

In terms of isolation, Check distinguishes two kinds of fixtures:

checked fixtures
: are run within the address space of unit tests (if fork mode is on), once for each unit test

unchecked fixtures
: are run in the address space of the test program, once for each test case

Warning: memory corruption in unchecked fixtures might crash the whole test suites.

Test fixtures in Check (cont.)

For a Check test case with 2 unit tests—`unit_test_1` and `*_2`—the execution order of test and fixture functions will be:

1. unchecked_setup();
2. fork();
3. checked_setup();
4. unit_test_1();
5. checked_teardown();
6. wait();
7. fork();
8. checked_setup();
9. unit_test_2();
10. checked_teardown();
11. wait();
12. unchecked_teardown();

Test fixtures in Check — example

```
Money *five_dollars;

void setup(void) {
    five_dollars = money_create(5, "USD");
}

void teardown(void) {
    money_free(five_dollars);
}

START_TEST(test_money_create_amount) {
    ck_assert_int_eq(money_amount(five_dollars), 5);
}
END_TEST

START_TEST(test_money_create_currency) {
    ck_assert_str_eq(money_currency(five_dollars), "USD");
}
END_TEST
```

Test fixtures in Check — example (cont.)

```
Suite * money_suite(void) {
    Suite *s;
    TCase *tc_core;
    TCase *tc_limits;

    s = suite_create("Money");

    tc_core = tcase_create("Core");
    tcase_add_checked_fixture(tc_core, setup, teardown);
    tcase_add_test(tc_core, test_money_create_amount);
    tcase_add_test(tc_core, test_money_create_currency);
    suite_add_tcase(s, tc_core);

    tc_limits = tcase_create("Limits");
    tcase_add_test(tc_limits, test_money_create_neg);
    tcase_add_test(tc_limits, test_money_create_zero);
    suite_add_tcase(s, tc_limits);

    return s;
}
```

Test fixtures in Check — example (cont.)

```
$ ./check_money
Running suite(s): Money
100%: Checks: 4, Failures: 0, Errors: 0
check_money.c:17:P:Core:test_money_create_amount:0: Passed
check_money.c:22:P:Core:test_money_create_currency:0: Passed
check_money.c:31:P:Limits:test_money_create_neg:0: Passed
check_money.c:35:P:Limits:test_money_create_zero:0: Passed

$
```

Selectively running tests

It might be important to run only a few tests

- e.g., when debugging a specific test failure
- e.g., to run fast vs slow tests in different phases of your development process

```
$ ./check_money
Running suite(s): Money
100%: Checks: 4, Failures: 0, Errors: 0
check_money.c:17:P:Core:test_money_create_amount:0: Passed
check_money.c:22:P:Core:test_money_create_currency:0: Passed
check_money.c:31:P:Limits:test_money_create_neg:0: Passed
check_money.c:35:P:Limits:test_money_create_zero:0: Passed

$ CK_RUN_CASE=Limits ./check_money
Running suite(s): Money
100%: Checks: 2, Failures: 0, Errors: 0
check_money.c:31:P:Limits:test_money_create_neg:0: Passed
check_money.c:35:P:Limits:test_money_create_zero:0: Passed
```

A similar CK_RUN_SUITE environment variable also exists.

Check boilerplate

```c
#include <stdlib.h>
#include <stdint.h>
#include <check.h>  // testing framework
#include "../src/money.h"  // SUT

START_TEST(test_money_create) {
    Money *m;  // setup

    m = money_create(5, "USD");  // exercise SUT

    // test oracle
    ck_assert_int_eq(money_amount(m), 6);
    ck_assert_str_eq(money_currency(m), "USD");

    money_free(m);  // clean up
}
END_TEST

Suite *money_suite(void) {
    TCase *tc_core;
    Suite *s;

    tc_core = tcase_create("Core");
    tcase_add_test(tc_core, test_money_create);
    // tcase_add_test(tc_core, test_foo);
    // ...

    s = suite_create("Money");
    suite_add_tcase(s, tc_core);
    // suite_add_tcase(s, tc_bar);
    // ...

    return s;
}
```

How do you like it?

Check boilerplate

```
#include <stdlib.h>
#include <stdint.h>
#include <check.h>  // testing framework
#include "../src/money.h"  // SUT

START_TEST(test_money_create) {
    Money *m;  // setup

    m = money_create(5, "USD");  // exercise SUT

    // test oracle
    ck_assert_int_eq(money_amount(m), 6);
    ck_assert_str_eq(money_currency(m), "USD");

    money_free(m);  // clean up
}
END_TEST

Suite *money_suite(void) {
    TCase *tc_core;
    Suite *s;

    tc_core = tcase_create("Core");
    tcase_add_test(tc_core, test_money_create);
    // tcase_add_test(tc_core, test_foo);
    // ...

    s = suite_create("Money");
    suite_add_tcase(s, tc_core);
    // suite_add_tcase(s, tc_bar);
    // ...

    return s;
}
```

How do you like it?

- quite a bit of boilerplate
- for relatively few lines of actual test code

checkmk

checkmk[4] can be used to reduce the amount of Check boilerplate to write and focus on the actual test code.

checkmk is used as a custom C preprocessor that expand specific #-directives to suitable calls to the Check API.

Some checkmk directives:

- **#suite** define a suite
- **#tcase** define a test case
- **#test** define a unit test
- **#main-pre** `main` preamble (e.g., to declare fixtures)

See the `checkmk(1)` manpage for a full list.

4. `http://micah.cowan.name/projects/checkmk/`

checkmk – example

```
#include <stdlib.h>
#include "../src/money.h"

#suite Money

#tcase Core

#test test_money_create_amount
    ck_assert_int_eq(money_amount(five_dollars), 5);

#test test_money_create_currency
    ck_assert_str_eq(money_currency(five_dollars), "USD")
```

checkmk – example (cont.)

```
#tcase Limits

#test test_money_create_neg
    Money *m = money_create(-1, "USD");
    ck_assert_msg(m == NULL,
                  "NULL should be returned on attempt "
                  "to create with a negative amount");

#test test_money_create_zero
    Money *m = money_create(0, "USD");
    if (money_amount(m) != 0)
        ck_abort_msg("0 is a valid amount of money");
```

checkmk – example (cont.)

```
Money *five_dollars;

void setup(void) {
    five_dollars = money_create(5, "USD");
}

void teardown(void) {
    money_free(five_dollars);
}

#main-pre
    tcase_add_checked_fixture(tc1_1, setup, teardown);
```

checkmk – example (cont.)

```
$ checkmk check_money.check > check_money.c

$ head check_money.c
/*
 * DO NOT EDIT THIS FILE. Generated by checkmk.
 * Edit the original source file "check_money.check" instead.
 */

#include <check.h>

#line 1 "check_money.check"
#include <stdlib.h>
#include "../src/money.h"
```

checkmk – example (cont.)

```
CC = gcc
CFLAGS = -Wall
LDFLAGS = $(CFLAGS)
LDLIBS = -Wall $(shell pkg-config --libs check)

all: check_money
check_money: check_money.o ../src/money.o

check_money.c: check_money.check
	checkmk $< > $@

clean:
	rm -f check_money *.o
```

checkmk – example (cont.)

```
$ make
gcc -Wall   -c -o check_money.o check_money.c
gcc -Wall  check_money.o ../src/money.o  -Wall -lcheck_pic \
  -pthread -lrt -lm -lsubunit -o check_money

$ CK_VERBOSITY=verbose ./check_money
Running suite(s): Money
100%: Checks: 4, Failures: 0, Errors: 0
check_money.check:19:P:Core:test_money_create_amount:0: Passed
check_money.check:22:P:Core:test_money_create_currency:0: Passed
check_money.check:30:P:Limits:test_money_create_neg:0: Passed
check_money.check:35:P:Limits:test_money_create_zero:0: Passed
```

Outline

1. Development processes and testing
2. xUnit & Check
3. Test-Driven Development
4. Bootstrapping TDD

Development as learning

- every software development project tries *something* that has never been tried before
  - otherwise you wouldn't be doing it...
- due to constraints, developers often use technologies they don't completely master
  - new technologies, old technologies used in new contexts, etc
- all stakeholders (developers, managers, customers) learn as the project progresses

Problem: as we don't know everything at the beginning, there will be unexpected changes during the project.

How do we cope with them?

Nested feedback loops

Idea

1. we use empirical feedback to learn about the system
2. we store what we learn in the system itself, for future use

To do so, we organize development as nested feedback loops with increasing time periods and scopes in the organization (file, unit, product, team, etc.), e.g.:

- pair programming — period: seconds
- unit tests — seconds–1 minute
- acceptance tests — minutes
- daily meeting — 1 day
- iterations — 1 day–1 week
- releases — 1 week–months

We want feedback as quickly as possible. If something slips through an inner loop, it will (hopefully) be catched by an outer one.

Expecting the unexpected

Practices that (empirically) help coping with unexpected changes:

- constant testing
  - when we change something we might introduce regressions
  - to avoid that we need to constantly test our system
  - doing it manually doesn't scale ⇒ automated testing
- simple design

  keep the code as simple as possible
  optimize for simplicity
  - as we will have to change it, we want code that is easy to understand and modify
  - empirical studies show that developers spend more time reading code than writing it
  - clean design doesn't come for free, to achieve it we must constantly refactor
  - test suites give you *courage* to refactor, and apply other changes, thanks to their tight feedback loop

TDD principle

So we have test suites. Why do we need TDD?

TDD principle

So we have test suites. Why do we need TDD? Because:

- developers don't like writing tests
- testing is often seen as a 2nd class development activity

TDD principle

So we have test suites. Why do we need TDD? Because:

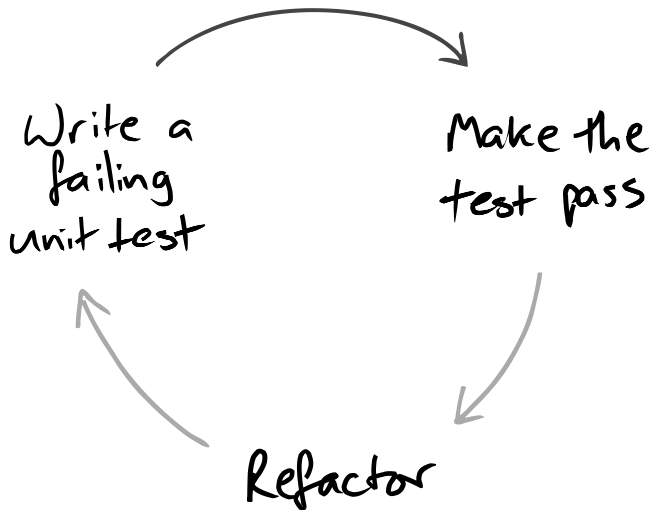
- developers don't like writing tests
- testing is often seen as a 2nd class development activity

TDD idea (i.e., a judo move on the above problem)

write tests *before* code

- don't write tests only to verify code *after* it's done
  leverage testing as a design activity
- write tests to clarify our ideas about what the code should do
- > *I was finally able to separate logical from physical design. I'd always been told to do that but no one ever explained how.* — *Kent Beck*
- write tests to get rapid feedback about design ideas
  - if *a test* is difficult to write, design is often wrong

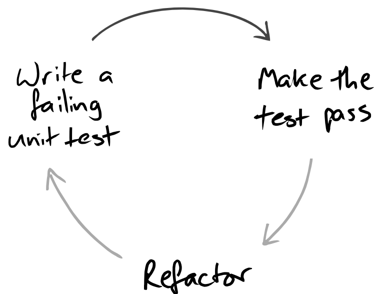
TDD in a nutshell



GOOS, Figure 1.1

The TDD development cycle

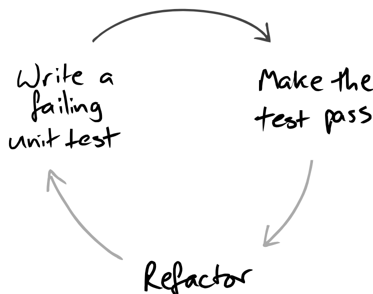
1. red: write a failing test
2. green: write code that makes it pass
3. refactor the code to be as simple as possible



GOOS, Figure 1.1

The TDD development cycle (cont.)

1. red: write a failing test
2. green: write code that makes it pass
3. refactor the code to be as simple as possible



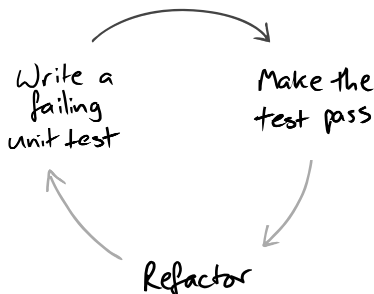
GOOS, Figure 1.1

By writing the test we:

- clarify acceptance criteria
- are pushed to design loosely coupled components
  - otherwise they are difficult to test
- document the code, via an executable description of it
- *incrementally* build a regression suite

The TDD development cycle (cont.)

1. red: write a failing test
2. green: write code that makes it pass
3. refactor the code to be as simple as possible



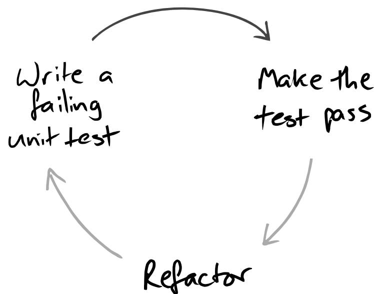
GOOS, Figure 1.1

By running the test we:

- detect errors when the context is fresh in our mind
- have a measure of progress, know when to stop (i.e., when we are "done enough")
  - avoid over-coding, "gold plating"

The TDD development cycle (cont.)

1. red: write a failing test
2. green: write code that makes it pass
3. refactor the code to be as simple as possible



GOOS, Figure 1.1

TDD golden rule

Never write a new functionality without a failing test.

Step 1 — write a failing unit test

This looks easy. But requires some care:

- write the test you'd want to read, ideally in 3 steps
  1. prepare test environment (input data and/or context)
  2. invoke the logic under testing
  3. verify that the results are correct

  If it cannot be done in a few lines (ideally: 3), write helper code.

  Remember: tests are documentation too.

Step 1 — write a failing unit test

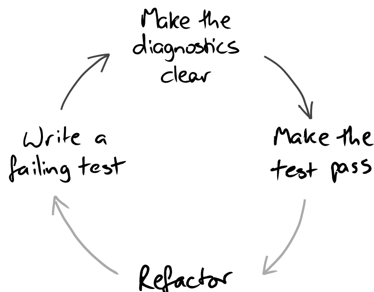
This looks easy. But requires some care:

- write the test you'd want to read, ideally in 3 steps
  1. prepare test environment (input data and/or context)
  2. invoke the logic under testing
  3. verify that the results are correct

  If it cannot be done in a few lines (ideally: 3), write helper code.

  Remember: tests are documentation too.

- watch the test fail *before* making it pass
  - otherwise you're not sure about your assumptions
  - if the test fails in unexpected ways, fix it (= the test)
  - if the diagnostic isn't clear, fix it



GOOS, Figure 5.2

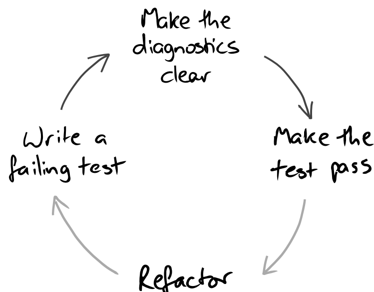
Step 1 — write a failing unit test

This looks easy. But requires some care:

- write the test you'd want to read, ideally in 3 steps
  1. prepare test environment (input data and/or context)
  2. invoke the logic under testing
  3. verify that the results are correct

  If it cannot be done in a few lines (ideally: 3), write helper code.
  Remember: tests are documentation too.

- watch the test fail *before* making it pass
  - otherwise you're not sure about your assumptions
  - if the test fails in unexpected ways, fix it (= the test)
  - if the diagnostic isn't clear, fix it
- unit test behavior, not functions
  - for TDD test coverage is less important than readable tests



GOOS, Figure 5.2

Step 2 — make the test pass (you won't like this)

To make the test pass we allow ourselves to take shortcuts.

Common strategies to make the test pass:

- fake it — all sorts of dirty tricks
  - e.g., return the constant value the test expects

Step 2 — make the test pass (you won't like this)

To make the test pass we allow ourselves to take shortcuts.

Common strategies to make the test pass:

- fake it — all sorts of dirty tricks
  - e.g., return the constant value the test expects
- obvious implementation — just type in the "obviously right" implementation
  - it takes experience to tune your confidence
  - too confident: you will have bad surprises
  - too prudent: you'll fake it too often
  - tip: use confidence increasingly, fall back when you get an unexpected "red bar" (i.e., test failure)

Step 2 — make the test pass (you won't like this)

To make the test pass we allow ourselves to take shortcuts.

Common strategies to make the test pass:

- fake it — all sorts of dirty tricks
  - e.g., return the constant value the test expects
- obvious implementation — just type in the "obviously right" implementation
  - it takes experience to tune your confidence
  - too confident: you will have bad surprises
  - too prudent: you'll fake it too often
  - tip: use confidence increasingly, fall back when you get an unexpected "red bar" (i.e., test failure)
- triangulation — when you have more than 2–3 tests that use the same implementation, factor out a common one
  - corollary: triangulation is commonly applied after several applications of the previous techniques

Step 3 — refactor

At this point: we have a test, some new code, and we are reasonably convinced that it is that code that makes the test pass.
We can now improve the code design, using tests as a safety net.

The goal of refactoring is to improve the design of existing code, without altering its external behavior (see Fowler 1999). We only give some of its intuitions here:

Code smells

- duplicate code
- long methods / large class
- too many parameters
- inappropriate intimacy
- Liskov principle violation
- complex conditionals
- ...

Techniques

- encapsulate field
- generalize type
- conditionals → polymorphism
- extract class / method
- rename method / field
- pull up / push down
- ...

TDD cycle — example

Goal: get a list of the elements contained in a binary tree

1. write a unit test that (pseudo-code)
   1. creates a binary tree containing the elements 2, 3, 1
   2. invokes a `toList` method on it
   3. asserts that `toList`'s return value = `[2; 3; 1]`

   Run all tests and ensures the new one fails *as we expect*

   - e.g., with a compilation error due to the lack of `toList`

TDD cycle — example

Goal: get a list of the elements contained in a binary tree

1. write a unit test that (pseudo-code)
    1. creates a binary tree containing the elements 2, 3, 1
    2. invokes a `toList` method on it
    3. asserts that `toList`'s return value = `[2; 3; 1]`

    Run all tests and ensures the new one fails *as we expect*

    - e.g., with a compilation error due to the lack of `toList`
2. implement `toList`
    - either by faking it: `return([2; 3; 1])`
    - or by writing the implementation you consider obvious

    Run all tests and ensures the new one succeeds

TDD cycle — example

Goal: get a list of the elements contained in a binary tree

1. write a unit test that (pseudo-code)
   1. creates a binary tree containing the elements 2, 3, 1
   2. invokes a `toList` method on it
   3. asserts that `toList`'s return value = `[2; 3; 1]`

   Run all tests and ensures the new one fails *as we expect*

   - e.g., with a compilation error due to the lack of `toList`

2. implement `toList`
   - either by faking it: `return([2; 3; 1])`
   - or by writing the implementation you consider obvious

   Run all tests and ensures the new one succeeds

3. refactor, e.g.:
   - write a proper implementation if you faked it
   - clean up, clean up, clean up

   Run all tests and ensures the new one still succeeds

TDD cycle — exercise

Exercise (bug fixing work-flow)

You have adopted TDD as the development process for your project. Describe the work-flow you would use to fix a bug.

?

TDD cycle — exercise

Exercise (bug fixing work-flow)

You have adopted TDD as the development process for your project. Describe the work-flow you would use to fix a bug.

Possible work-flow:

0. receive bug report
1. run all tests to ensure clean slate
2. create a new test that
   - recreates the context of the (alleged) bug
   - would succeed if the bug didn't exist
3. run all tests
   - new test fails → reproducible bug
   - new test passes → unreproducible bug → investigate with submitter

TDD cycle — exercise (cont.)

Exercise (bug fixing work-flow)

You have adopted TDD as the development process for your project. Describe the work-flow you would use to fix a bug.

Possible work-flow:

5. fix the bug
   - even with dirty workarounds, to ensure you've cornered it
6. run all tests
   - all test passes → bingo!
   - new test fails → try again (= go back to 5)
   - old tests fail → regression, try again
7. refactor as needed
   - from workaround, to proper fix
8. release fix (including the new test!)

Outline

1 Development processes and testing

2 xUnit & Check

3 Test-Driven Development

4 Bootstrapping TDD

TDD — the bigger picture

It is tempting to use only *unit* tests to implement TDD, but:

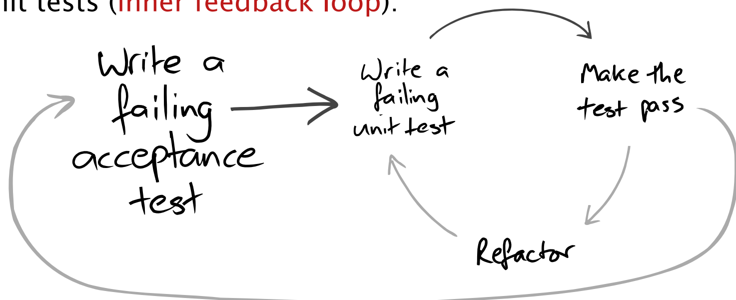
- you might end up having a lot of unused well-tested units
- you don't know where to start, nor when to stop

TDD — the bigger picture

It is tempting to use only *unit* tests to implement TDD, but:

- you might end up having a lot of unused well-tested units
- you don't know where to start, nor when to stop

That's why TDD leverages both acceptance (outer feedback loop) and unit tests (inner feedback loop):



GOOS, Figure 1.2

End-to-end testing

Acceptance tests should exercise the system end-to-end

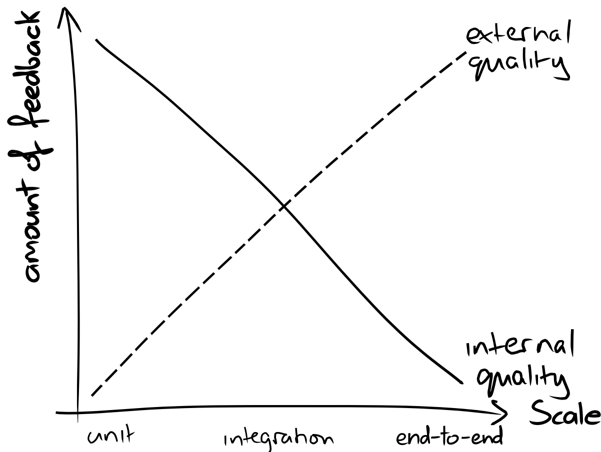
- black-box testing at system level
  - no instantiation/invocation of internal objects
  - use the system via its interfaces (user interface, external API, parsing its output and producing its inputs, etc.)
- test both the system and its processes
  - build
  - deployment in a realistic environment
    - don't trust the results of acceptance tests run in development environments
  - any other qualification mechanism
    - e.g. static analyses, stress testing, benchmark, etc.

Testing, quality, feedback

External quality: how well the system meets the needs of its users

Internal quality: how well the system meets the needs of its developers

- e.g. good design: low coupling & high cohesion
- it is often harder to push for internal than external quality, but we need to do so to cope with changes



GOOS, Figure 1.3

First feature paradox

Writing the first acceptance test at the beginning of a project is problematic:

- we want to test end-to-end the system and its processes
- but we don't have yet the tooling to make the test fail

First feature paradox

Writing the first acceptance test at the beginning of a project is problematic:

- we want to test end-to-end the system and its processes
- but we don't have yet the tooling to make the test fail

To get out of the paradox we compromise a bit, implementing a walking skeleton to kick start TDD.

Definition (walking skeleton)

An implementation of the smallest possible part of real functionality that we can automatically build, deploy, and test end-to-end.

To implement the walking skeleton we need to automate a lot of processes. That will force us to understand them better.

Example

The walking skeleton of a DBMS-backed web application will just show a static "Hello, World" web page.

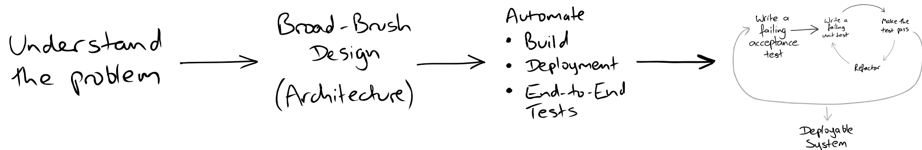
Walking skeleton

(Some of the) tasks to be completed as part of a walking skeleton:

- create a VCS repository, check in the code
  - requirements: choose Version Control System, choose hosting
- automate the build process
  - requirement: choose build tool (e.g. Ant, Maven)
  - note: “just click a button in Eclipse” $\neq$ automation
- automate deployment in a realistic environment
  - requirement: choose packaging/deployment mechanisms
- automate test execution
  - requirement: choose test framework
  - again: “just click a button in Eclipse” $\neq$ automation
- ...
- iteration 0: implement, deploy, test first feature

Yes, it’s a lot of work!

Kick-starting TDD



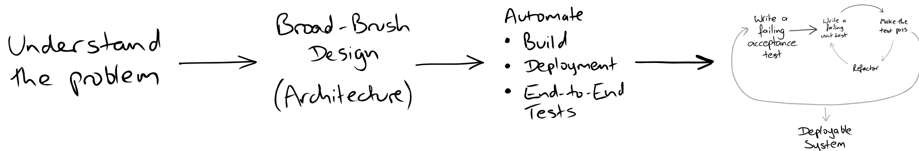
GOOS, Figure 4.2

Note: “Broad-Brush Design” ≠ “Big Design Up Front (BDUF)”

TDD as a whole

Periodically reassess both your understanding of the problem and the toolchain

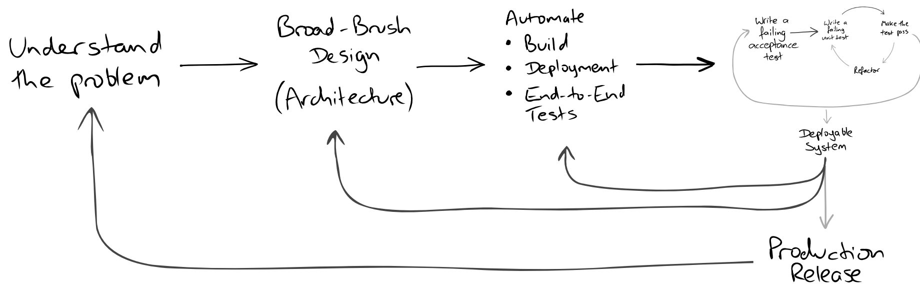
GOOS, Figure 4.2



TDD as a whole

Periodically reassess both your understanding of the problem and the toolchain

GOOS, Figure 4.3



Test suites organization

unit and integration test suites

- should always pass
- should run fast

acceptance test suite

- catch regressions
- should always pass
- might take longer to run

new acceptance test suite

- corresponds work in progress
- will keep on failing during inner loop iterations

Bibliography

Steve Freeman and Nat Pryce
Growing Object-Oriented Software, Guided by Tests[5]
Addison-Wesley, 2009.

Kent Beck
Test Driven Development: By Example
Addison-Wesley, 2002.

Martin Fowler
Refactoring: Improving the Design of Existing Code
Addison-Wesley Professional, 1999.

Kent Beck
Simple smalltalk testing: With patterns
The Smalltalk Report 4.2 (1994): 16-18.
available at `http://www.xprogramming.com/testfram.htm`

5. referred to as the "GOOS" book in this lecture slides