ソフトウェアテスト
[11] テスト駆動開発演習

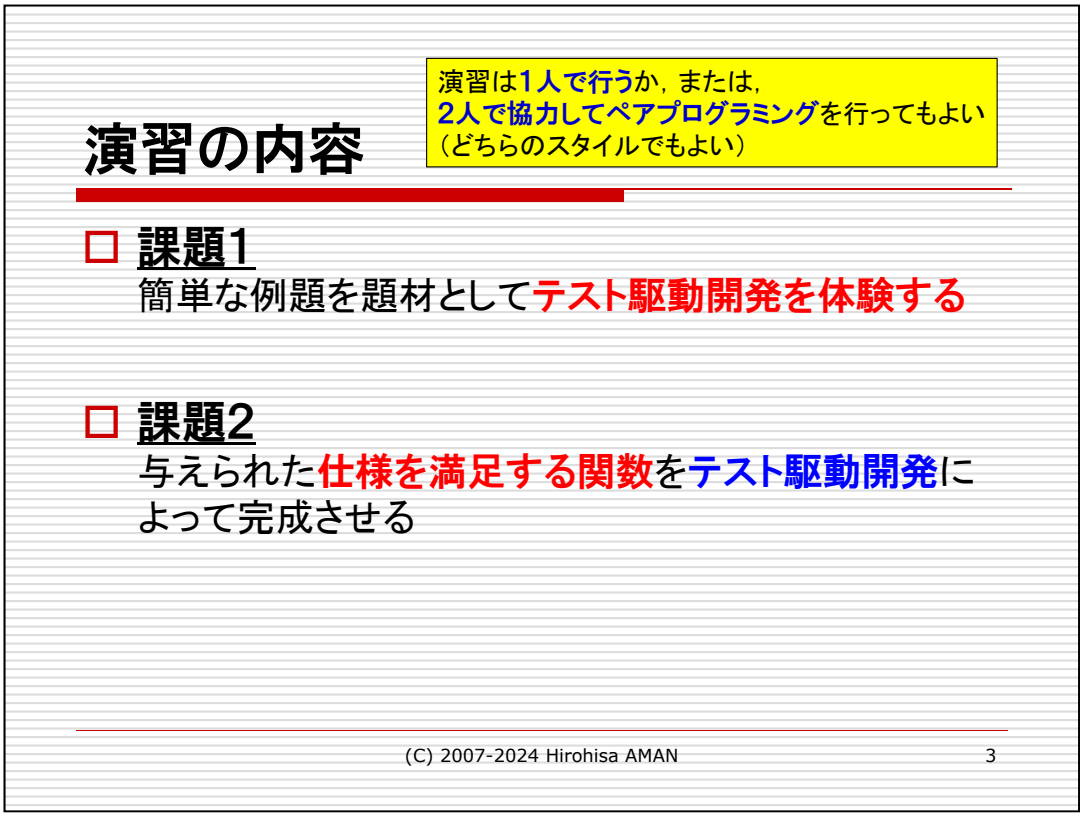
Software Testing
[11] Exercise: Test-Driven Development

あまん ひろひさ
阿萬 裕久(AMAN Hirohisa)
aman@ehime-u.ac.jp

演習の目的

- テストファーストに基づいたテスト駆動開発を体験・学習する
- 主な内容
  - 簡単な例題を使ったテスト駆動開発の練習
  - 仕様を満たす関数のテスト駆動開発

テスト駆動開発（テストくどうかいはつ）: test-driven development
体験（たいけん）: experience
仕様を満たす関数（しようをみたす関数）: a function meeting the requirements



Note: you can do these exercises in a group of two students (pair programming), or you can do them by yourself.

課題1（かだい1）: exercise-1

Experience a test driven development of a toy problem (a simple example).

簡単（かんたん）な例題（れいだい）: simple example, toy problem

題材（だいざい）: educational material

テスト駆動開発（テストくどうかいはつ）: test driven development

体験（たいけん）する: experience

課題2（かだい2）: exercise-2

Perform a test driven development of a function satisfying the given specification.

与（あた）えられた仕様（しよう）: given specification

満足（まんぞく）する: satisfy

関数（かんすう）: function

完成（かんせい）させる: complete

課題1

【対象プログラム】sample1101.c

【支援プログラム】run_test.h

- run_test.h では，テスト駆動開発の演習をサポートするための関数を提供しています
- この後の説明に従って sample1101.c を編集しながらプログラムを完成させなさい

run_test.h provides a function for supporting this exercise regarding the test driven development.

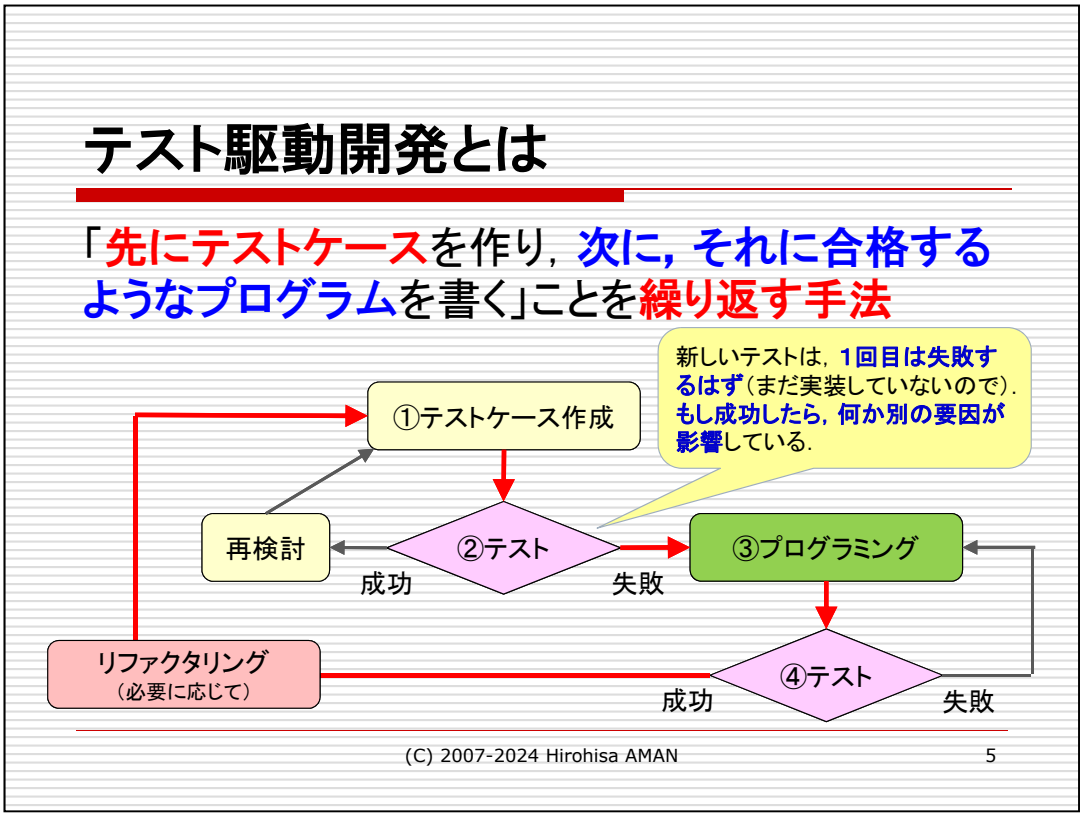
<exercise step>

Develop sample1101.c through editing and testing.

The procedure is explained in the following slides.

支援（しえん）: support

編集（へんしゅう）: edit



合格（ごうかく），成功（せいこう）: pass

失敗（しっぱい）: fail

何か別の要因が影響する（なにかべつのよういんがえいきょうする）: another factor influences

必要に応じて（ひつようにおうじて）: if necessary

<Test driven development>

1) Create a test case.
2) Test the program by the test case; it must fail because you have not write production code corresponding to the test.
   If your test passes, there may be another factor affecting the execution of the program, so you have to check it.
3) Do programming so that the production code would pass the test.
4) Test the code. If it fails, you have to fix the bug.
   After passing the test, you should do refactoring your code if necessary.
5) Go back to step 1 and create another test case.

課題1での開発目標

- 関数 **get_digit_number** を作る
  - この関数は引数で与えられた整数 x について，その**桁数を戻り値**としてもつ
  - ただし，0 ＜ x ＜ 1000000（=10^6）とする
  - 上の範囲外の整数が与えられた時は**エラーとして -1 を戻り値**とする

（例1） `get_digit_number(256)` … この値は 3 となる
（例2） `get_digit_number(0)` … この値は -1 となる（エラー）

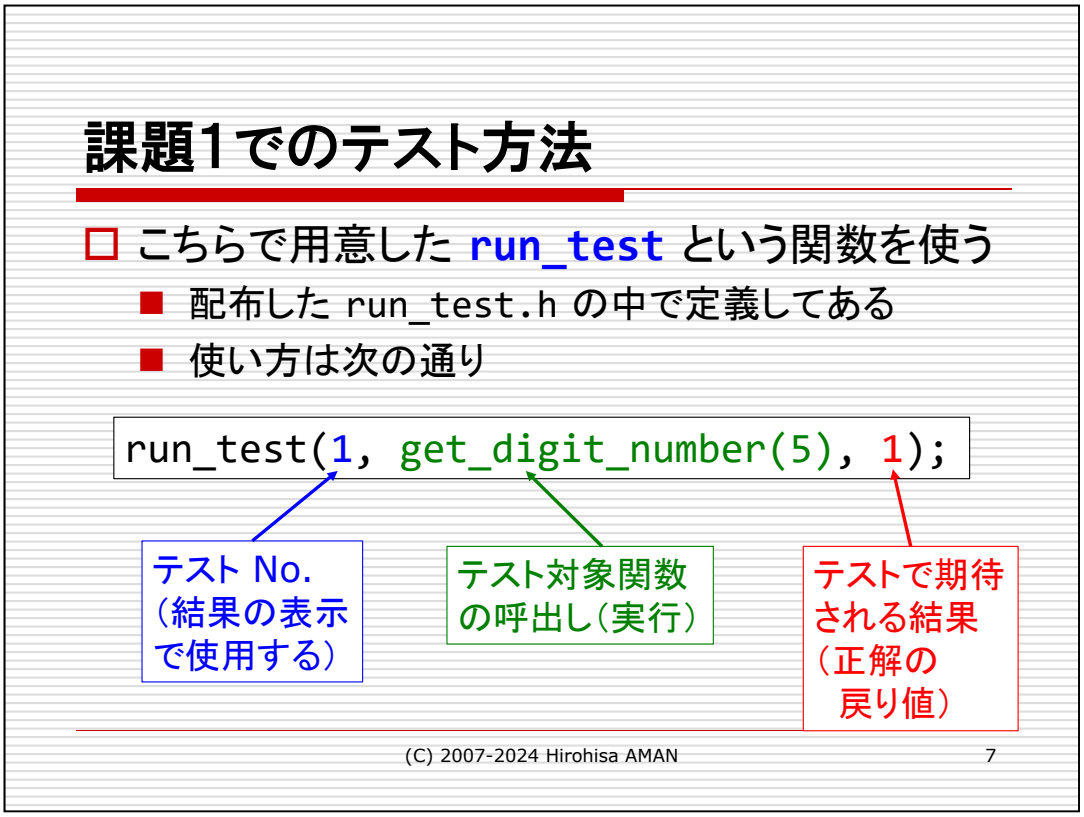
 6

開発目標（かいはつもくひょう）: the goal of development

Your goal is to develop the function “get_digit_number.”
It gets an integer x as the argument, and returns the number of digits of x.
The integer x must be 0 < x < 10^6.

When x is out of the above range, the function must return -1 representing an error.

引数（ひきすう）:argument
桁数（けたすう）: number of digits
戻り値（もどりち）: return value
範囲外（はんいがい）: out of range



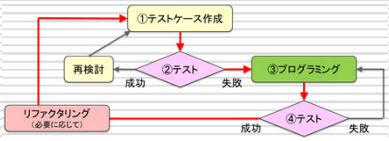
How to run your tests in the exercise-1: use “run_test” as written in sample1101.c.

テスト対象関数の呼出し（テストたいしょうかんすうのよびだし）: calling the function under test

期待される結果（きたいされるけっか）: expected result

正解の戻り値（せいかいのもどりち）: correct return value

課題1の進め方(1／7)



- まずは main 関数内に1個目のテストケースを用意する(手順①)

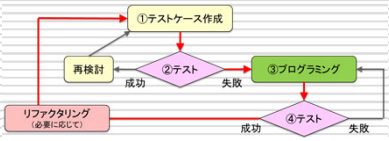
```c
#include <stdio.h>
#include "run_test.h"

int main(void){

  run_test(1, get_digit_number(5), 1);

  return 0;
}
```

この時点ではまだ,
関数 get_digit_number
は存在していない

Step 1/7 of Exercise-1:
Write the first test case in the main function.
Here, get_digit_number is not existed yet.

まだ存在していない(まだそんざいしていない): it does not exist

課題1の進め方(2／7)



□ いったんコンパイルして**エラーになるのを確認**する(手順②)

```
C:¥Users¥aman¥Desktop¥cprog>gcc sample1001.c
sample1001.c: In function 'main':
sample1001.c:6:15: warning: implicit declaration of function 'get_digit_number' [-Wimplicit-function-d
eclaration]
    6 |     run_test(1, get_digit_number(5), 1);
      |               ^~~~~~~~~~~~~~~~

c:/mingw/bin/../lib/gcc/mingw32/9.2.0/../../../../mingw32/bin/ld.exe: C:¥Users¥aman¥AppData¥Local¥Temp
¥ccdi785j.o:sample1001.c:(.text+0x6d): undefined reference to `get_digit_number'
collect2.exe: error: ld returned 1 exit status

C:¥Users¥aman¥Desktop¥cprog>
```

この場合，get_digit_number という関数が**存在しないという**
エラーになるのが正解である
(もしもここでエラーが出なかったら，既にその名前の関数が
存在していることになるため，この後のテストで混乱してしまう)

Step 2/7 of Exercise-1:

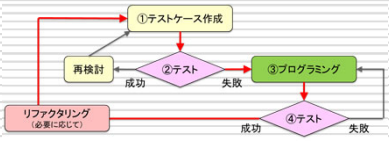
Compile sample1101.c and check it causes an error because there is not “get_digit_number” in the program.

If the compile succeeds, it means a function with the same name already exists. Such a function would cause a trouble in our test.

既にその名前の関数が存在している(すでにそのなまえのかんすうがそんざいしている): a function with that name exists already

混乱する(こんらんする): get confused

課題1の進め方(3／7)



- 次に関数 get_digit_number を用意する:
  とりあえず戻り値を 1 にしておけばテストには成功するはず(手順③)

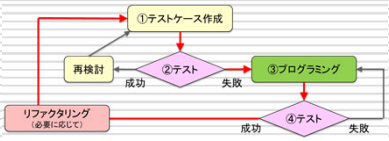
```
int get_digit_number(int x){
  return 1;
}

int main(void){

  run_test(1, get_digit_number(5), 1);

  return 0;
}
```

とりあえず「その場しのぎ」だが動くようにしてみる

Step 3/7 of Exercise-1:
Implement get_digit_number quickly.
It is OK that the code is tentative (ad hoc) to pass only the test case No.1.
(for example, “return 1;” works for the test case)

とりあえず: tentative
その場しのぎ(そのばしのぎ): ad hoc

課題1の進め方(4／7)



- 今度はコンパイルでエラーにはならず，実行すると，No.1 のテストに成功したと表示される（手順④）

```
C:¥Users¥aman¥Desktop¥cprog>gcc sample1001.c

C:¥Users¥aman¥Desktop¥cprog>a
No. 1: PASS
```

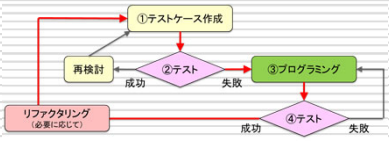
No. 1: PASS という表示は，
関数 run_test がやってくれている

Step 4/7 of Exercise-1:

Compile sample1101.c again, and run it.

Its test would pass.

課題1の進め方(5／7)



□ 次のテストケースを追加する(手順①)

- No.2 として 2桁の整数(10)を引数として与え，答えが 2 となることを期待したテストケースにする

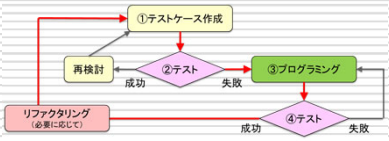
```
int main(void){

  run_test(1, get_digit_number(5), 1);
  run_test(2, get_digit_number(10), 2);

  return 0;
}
```

No.2 のテストケースを追加する

Step 5/7 of Exercise-1:

Add second test case (No.2), where the argument is 10 and the expected return value is 2.

課題1の進め方(6／7)



□ 再びコンパイルして実行し，今度は No.2 のテストで失敗することを確認する(手順②)

```
C:¥Users¥aman¥Desktop¥cprog>gcc sample1001.c

C:¥Users¥aman¥Desktop¥cprog>a
No. 1: PASS
No. 2: *** FAIL! ***: result = 1 (expected result = 2)
```

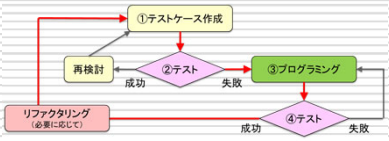
No.1 は PASS のままだが，
No.2 は失敗(FAIL)となっている
　　関数の実行結果(戻り値)は 1 だが （result = 1）
　　期待される結果は 2 であるという表示(expected result = 2)

Step 6/7 of Exercise-1:

Compile sample1101.c and run it.

Here, it must fail because get_digit_number is implemented only for test case No.1 and does not consider the test case No.2.

課題1の進め方(7／7)



- 次に No.1 と No.2 の両方のテストに合格するようにプログラムを修正していく(手順③)

- これを繰り返すことで徐々にプログラムの完成度を高めていくのがテスト駆動開発である
  - 途中でリファクタリングを行ってよい
  - テストケースが徐々に増えていくが，毎回すべてが再実行されるところがこの手法のポイント

Step 7/7 of Exercise-1:

So, you should fix get_digit_number so that it can pass both test cases No.1 and No.2.

By iterating those “adding new test case and fixing code” activities, you can get a perfect code.

That is the concept of “test driven development.”

Of course, you may do refactoring during the iteration.

The important point is that every iteration reruns all test cases, so you can easily find a bug when you made a mistake.

No1. と No.2 の両方のテストに合格(No.1 と No.2 のりょうほうのテストにごうかく): pass both No.1 and No.2 tests

徐々に(じょじょに): gradually

完成度を高める(かんせいどをたかめる): enhance its maturity (quality, perfection level)

毎回すべて再実行(まいかいすべてさいじっこう): re-run all tests every time

以上の作業を繰り返す

- 関数が仕様通り動作することを確認できるまで「テストケースを追加＆プログラムを修正」を繰り返す

- エラー処理も忘れずに！
  エラーの場合は -1 を戻り値とするが，直接 -1 と書くのは良くない（ハードコーディングになる）のでERROR というマクロとして定義するとよい

以上の作業を繰り返す（いじょうのさぎょうをくりかえす）: iterate the above steps

関数が仕様通り動作する（かんすうがしようどうりただしくどうさする）: the function behaves correctly; the behavior meets the specification

エラー処理（えらーしょり）: error handling

直接 -1 と書くのは良くない（ちょくせつ -1 とかくのはよくない）: it is not good practice to describe “-1” directly

ハードコーディングになる: it is a case of “hard cording”

ERROR というマクロとして定義するとよい: it is better to define it as macro “ERROR”

→ #define ERROR -1

課題1の提出

- Teams から
  sample1101.c
  を提出しなさい

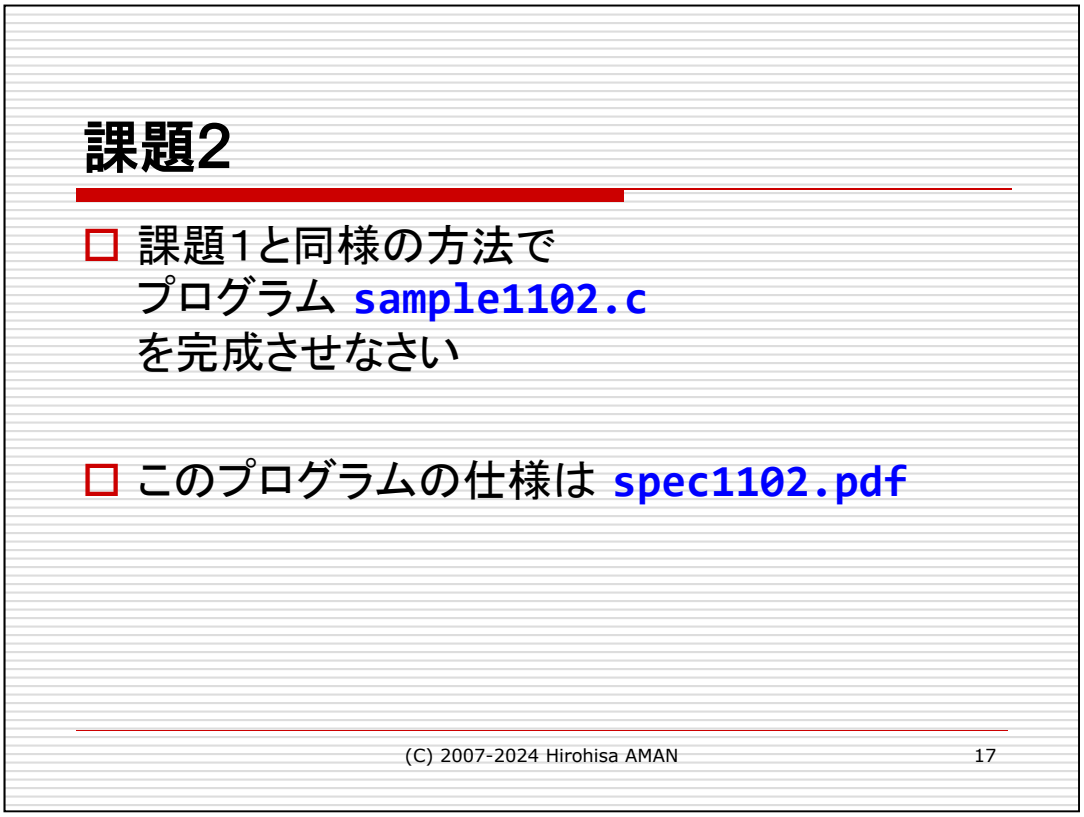
もしも2人組でペアプログラミングを行った場合は，2人とも同じものを提出しなさい

[11] Exercise-1 (TDD)

- 提出〆切：**23:59, May 24**

提出〆切（ていしゅつしめきり）：submission due (deadline)

If you did the exercise in a group of two students, each of the group members should submit the same file.



課題（かだい）2: exercise-2

<Exercise-2>

Develop sample1102.c by the test driven development method (similar to exercise-1).

Its specification is described in spec1102.pdf.

同様（どうよう）の方法（ほうほう）で: in a similar way

課題2の提出

- Teams から
  sample1102.c
  を提出しなさい

もしも2人組でペアプログラミングを行った場合は，2人とも同じものを提出しなさい

[11] Exercise-2 (TDD)

- 提出〆切：**23:59, May 24**

提出〆切（ていしゅつしめきり）：submission due (deadline)

If you did the exercise in a group of two students, each of the group members should submit the same file.

課題1，2で注意すること

- いずれの課題もテスト関数の呼出し

```
run_test(1, get_digit_number(5), 1);
run_test(2, get_digit_number(10), 2);
……
```

  をいろいろと考えて**追加していきなさい**

- 1 個の run_test が 1 個のテストケースになりますので**できるだけ多く作りなさい**

Notice: You should add “run_test”s as many as possible

Survey に答えなさい

- 今回の演習では1人でやってもよいし，2人組でペアプログラミングをやってもよい
- みなさんがどちらを選んだのかを知りたいので，最初に Survey に答えてください

[11] Survey (All students MUST answer first)

Answer the survey first.

You can do these exercise by yourself, or in a group of two students (pair programming).

You all must answer the survey to declare which style do you choose.