

ソフトウェア工学

第6回

土田 隼之

週	授業内容・方法	週ごとの到達目標
1週	ソフトウェアの性質と開発	ソフトウェア開発の特徴および課題について少なくとも一つ上げられ、その理由を言える。
2週	ソフトウェア開発プロセス	複数の開発プロセスモデルを挙げ、それぞれの特徴を言える。
3週	要求分析	要求分析とプロトタイピングの関係性や有用性について言える。
4週	ソフトウェア設計	モジュールの結合度の低い場合と高い場合のモジュール間の依存関係について言える。
5週	プログラミングとテスト	誤り混入をさせないためのプログラミング手法およびテスト効率を向上させる方法について言える。
6週	テストと保守	保守容易性を確保するための方策について、考察し、述べることができる。
7週	グループワーク	前半6週に関する課題を、グループワークで取り組む。
8週	中間試験	前半に習得した項目について確認する。
9週	オブジェクト指向 1	身の回り
10週	オブジェクト指向 2	オブジェ
11週	ソフトウェア再利用	ソフトウ
12週	プロジェクト管理	プロジェ
13週	品質管理	品質管理
14週	ソフトウェア開発規模と見積もり	ソフトウェア開発規模の見積もり手法について言える。
15週	グループワーク	後半6週に関する課題を、グループワークで取り組む。
16週	期末試験	後半に習得した項目について確認する。

中間以降に、再度テストを扱う予定で
います。テストは重要で幅広いので

9週:中間試験を実施予定

8週:中間試験の模擬問題を予定

→傾向を見てもらう目的。ポートフォリオ点に入るので、試験点には影響なし。

#課題をきちんと出していれば、模擬試験0点でもポートフォリオ点満点

今日の内容

1) テスト例と有名な不具合

- インテルPentium
- 組み込みデータベースSQLite(OSS)
- JVMの商用実装

2) テストと保守

テストはめちゃめちゃ大切です

有名な不具合(インテルPentium)

Pentium FDIV バグ

出典: フリー百科事典『ウィキペディア (Wikipedia) 』

ある割り算すると、結果が間違う。。

1994年10月30日、リンチバーグ大学のThomas Nicely教授はPentiumプロセッサの浮動小数点演算ユニットにバグがあることを報告した。その内容は、とある割り算を行うと非常に小さな量だけ間違った値を返すというものだった。この結果はインターネットを通じて他の人々の手で素早く検証された。そして問題を起こすのがPentiumプロセッサのx87浮動小数点除算命令であることと、その二モニックFDIVからPentium FDIV バグとして知られるようになった。また、別の人々はPentiumが返す結果が引き起こす割り算問題は、100万回に高々61回までしか起こらないことを見つけた（これは、オペランドの色々な値に対して、という意味である。このバグは特定の値に対して必ず起きる）。

https://ja.wikipedia.org/wiki/Pentium_FDIV_%E3%83%90%E3%82%B0

12月19日にはNew York Timesがこれに関する記事を掲載。これによって当時は、「Pentium」という言葉が「バグを抱えているもの」の代名詞に使われるほどだった。結局インテルは12月20日に、公式に謝罪を表明。すでに出荷したPentiumにバグがあったこと。そのバグのあったPentiumはすべてバグを修正したものに無償交換すること。およびこの無償交換にともなう費用を4億2000万ドル(後に4億7500万ドルに上方修正)と見込んでいることを発表して、ようやく事態は収束に向かうことになった。

ちなみに、1994年度におけるインテルの売上は115億2000万ドル。純利益は22億9000万ドル(関連リンク3)と発表されていたから、純利益の17%ほどがこれで吹っ飛んだ計算になる。

利益2200億円で、交換費用500億円
#ざっくり1ドル100円とした

<https://ascii.jp/elem/000/000/757/757002/>

テストの例(OSSデータベース)

SQLiteのテストコードは4567万8000行！ 本体のコードは6万7000行

有名なOSSの組み込みデータベース

2010年4月22日

軽量なリレーショナルデータベースとして人気のSQLite。そのWebサイトに掲載されている「[How SQLite Is Tested](#)」の内容が、海外のプログラマなどのあいだで話題になっています。

3月に公開された最新バージョンの[SQLite 3.6.23](#)。本体のソースコードは約6万7200行（67.2KSLOC、Kilo Source Lines of Code：空行やコメントを除いた行数）なのに対し、テストコードはなんと4567万8300行（45678.3KSLOC）だと紹介されているのです！ これはテストコードが本体の約679倍もの大きさだということになります。

1)MySQL5.7で66万行くらい

<https://blog.s-style.co.jp/2018/04/1709/>

2)SQLite3.33(2020)で14万行くらい

<https://www.sqlite.org/testing.html?>

OSSにとっては、ミッションクリティカルなシステムで使われるのが重要となる。

→商用製品は誰かが会社員生命かけて実装している。一方、OSSは？今では、linuxが重要なシステムでも広く使われている。

https://www.publickey1.jp/blog/10/sqlite45678000_67000.html

テストの例(OSSデータベース)2

100%のブランチカバレッジ

SQLiteコアのライブラリをテストするテストコードとして、以下の3つが紹介されています。

TCL Tests

TCL Testsはもっとも古いテストコードで、[TCL scripting language](#)で記述されています。テストコードは1万4700行（14.7KSLOC）、536ファイルから構成されており、2万5400のテストケースが記述されています。それぞれのテストケースはパラメータ化されており複数回実行されるため、全体では220万回のテストを実行します。

TH3

TH3テストはCで書かれていて、SQLiteコアのライブラリに対して100%のブランチカバレッジ（すべての分岐に対してテストを行う）を実行します。TH3は公開されているインターフェイスのみを用いたテストです。テストコードは60万2900行（602.9KSLOC）で2万9644のテストケースが記述されています。それぞれのテストケースは徹底的にパラメータ化されており、全体で150万回のテストを実行します。

SQL Logic Test

SQL Logic Test（SLTテスト）は大量のSQL文のテストを、SQLiteとPostgreSQL、MySQL、SQL Server、Oracle 10gなどほかのデータベースに対して実行し、同じ結果が得られるかどうかをテストします。このテストは1.12GBのテストデータを用いて、720万回のクエリを実行します。

他のDBMSと結果が一致すればOKとのこと

https://www.publickey1.jp/blog/10/sqlite45678000_67000.html

JVMの商用実装

● 独自JVMの開発は、顧客の「困った」が起点に

――日立ではJVM (Java Virtual Machine) を独自に開発しているが、それは最初からか？

最初にリリースしたときは、JVMは提供していません。顧客がCosminexusを動かすには、サンもしくは各OSが提供するJVMを利用する必要がありました。しかし、Javaでのシステムが本格的に稼働し始めた2001年ごろから、原因究明が困難なトラブルも発生するようになりました。このようなトラブルは、JVMまで調査しないと分からない根が深いもので、各JVM開発元のバグ情報を調べ、該当する対策モジュールを顧客に適用してもらうといった対応を行っていました。このため、問題解決までに時間を要し、顧客に迷惑を掛けることがありました。この反省を踏まえ、顧客に安心してシステムを運用してもらうため、JVMを自製し2002年のバージョン5からCosminexusに搭載しました。

Java実行環境
(JREの一部)

● 開発者自らが保守・サポートして機能追加に生かしている

――国産である優位点とは？

例えば、「海外製アプリケーションサーバを使ったシステムでJVMがらみのトラブルがあった場合、パッチ対応完了まで40日ぐらいかかった」という話を聞いたことがありますが、CosminexusではJVMがらみのトラブルがあっても、長くても2、3日で解決しています。なぜなら、Cosminexusの保守／サポートはCosminexusの開発に携わる技術者自身が直接行っているからです。製品の中身が分かっている技術者が対応するので、「どう動かすべきか」「どうチューニングすべきか」という適切な内容を提案できます。さらに、この過程は顧客の要望を直に聞く密接なコミュニケーションの場にもなっています。この点は、国産ベンダだからこそ実現できる点ではないでしょうか。

今日の内容

1) テスト例と有名な不具合

- インテルPentium
- 組み込みデータベースSQLite(OSS)
- JVMの商用実装

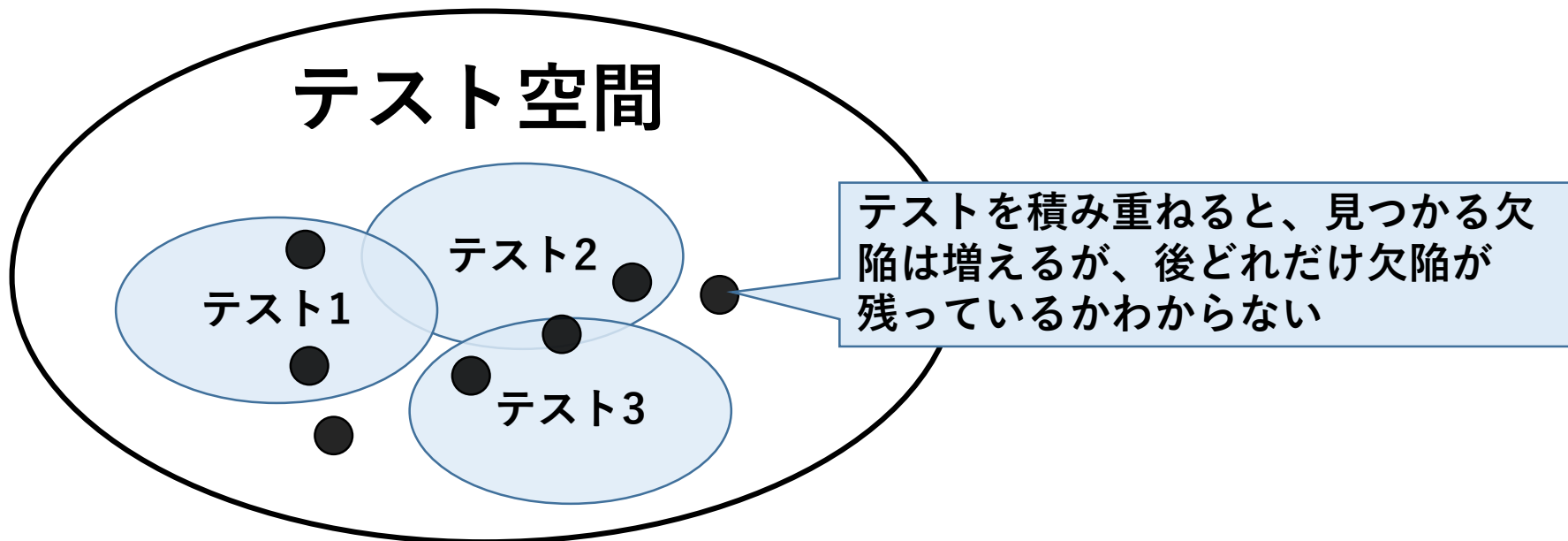
2) テストと保守

テスト工程とは

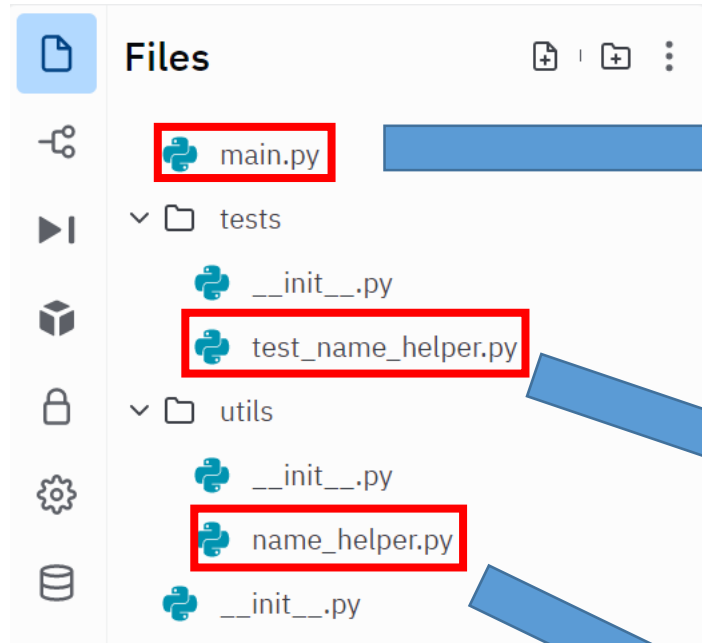
- テストとは

品質管理活動における検査手段であり、限られた時間内に、目標指標を達成しているかを判定できる必要がある。

テストでは、プログラム内に残存する欠陥を検知するためにプログラムを実行する。被テストプログラムに対して、欠陥がないことを保証できるテストケースの全空間をテスト空間と呼ぶこととする。



単体テストの例(前回内容)



```
main.py ×  
1 import pytest  
2 from utils import name_helper  
3  
4 string1= name_helper.split_name("John Smith")  
5 print(string1)  
6 pytest.main()
```

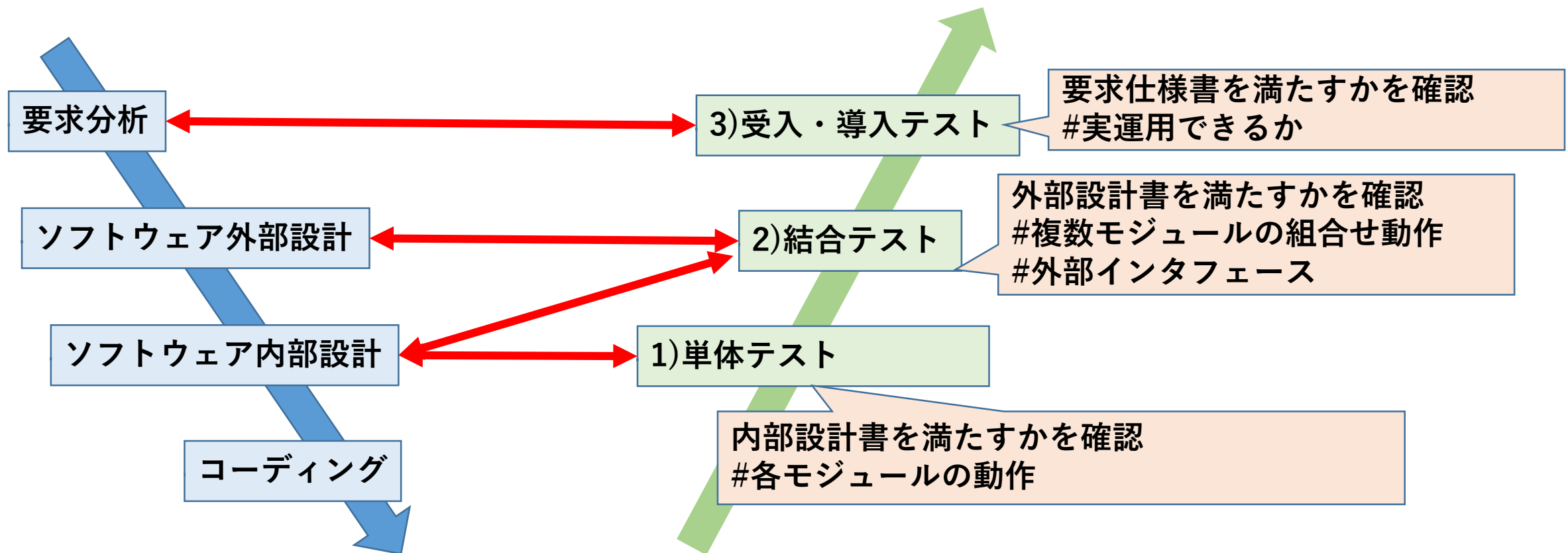
```
tests/test_name_helper.py ×  
1 from namesplit.utils import name_helper  
2  
3 def test_two_names():  
4     assert name_helper.split_name("John Smith")  
       == ["John", "Smith"]
```

```
utils/name_helper.py ×  
1 def split_name(name):  
2     first_name, last_name = name.split()  
3     return [first_name, last_name]
```

入力"John Smith"に対して
返り値が["John","Smith"]に
なることを確認

テスト工程とVモデル

要求分析からコーディングまでの全工程に対応したテストを実施する必要がある。各開発工程に対応させてテスト工程を分割したものをVモデルという。#他に、Wモデルなどがある。



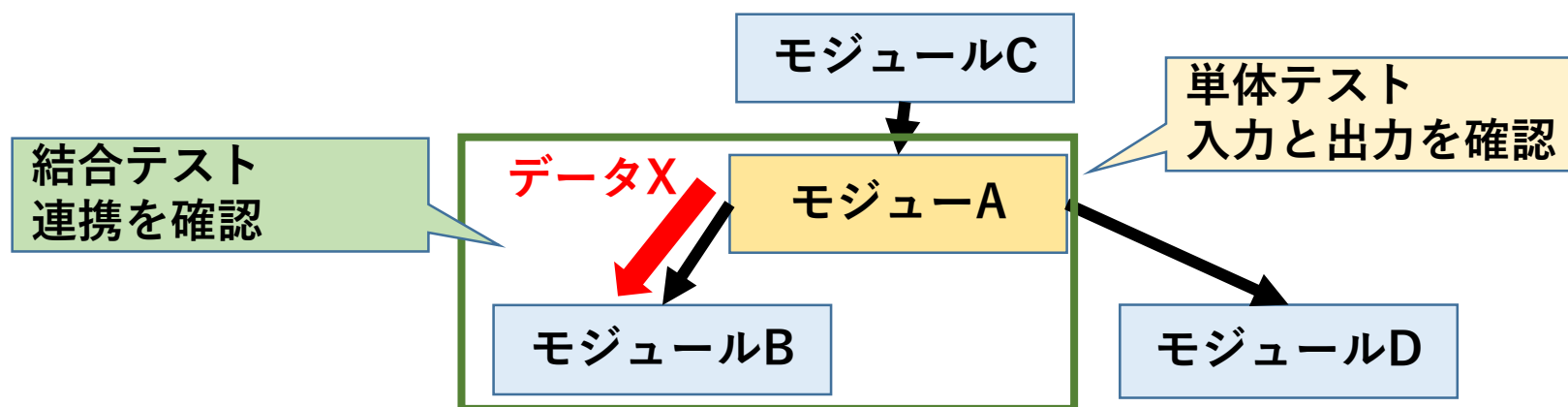
テスト工程の詳細

1)単体テスト:ソフトウェアの最小単位である「モジュール」ごとに行うテストである。内部設計書どおりにモジュールが動くかどうかをテストし、コーディングされたソフトウェアの論理構造が適切かどうかを確認する。

2)結合テスト:複数のモジュールを組合わせて動作を確認するテストである。外部設計書どおりに、モジュール間の連携が実現されているかを確認する。

例:モジュールAで求めたデータX(計算結果)が、別のモジュールBへ入力データとして渡され、それをつかってモジュールBが適切に処理結果を出力しているかを確認する。

3)受入・導入テスト:個々のモジュールや機能を統合した状態で、要求定義の内容が実現されているかを確認するテストです。



1)単体テスト

1)単体テスト:ソフトウェアの最小単位である「モジュール」ごとに行うテストである。内部設計書どおりにモジュールが動くかどうかをテストし、コーディングされたソフトウェアの論理構造が適切かどうかを確認する。

→関数や手続きといった最小単位のプログラムを単体で動かしてみる。
テストを行う際には、テスト項目を作成し、テスト項目に対応するテストデータを定める。

例:映画館のチケット算出

大学生以上:1800, 高校生以下:1200, 会員:1500, 毎週水曜:1000

→年齢、曜日、会員かを入力し、チケット代金を出力

単体テストでは、プログラムの内部構造を考慮したテストである**ホワイトボックステスト**が用いられることが多い。

テストケース設計(例:ホワイトボックステスト)

限られた時間で完全なテストを行うことは不可能なので、できるだけ少ない数で多くの欠陥を発見できるテストケース集合を見つけるのが望ましい

→『入力A(0～999)と入力B(0～999)から、出力Cを得る関数Dのテスト』だと、 $1000 \times 1000 = 100$ 万の「入力と出力の組合せ」を全てテスト？関数Dの結果と関数Eの結果から出力Fを得る関数だと、 100 万 \times ?の組合せ？**全て確認するのは不可能→各種テスト設計を使う**

- テストケースは、テストを実行するための入力の定義だけではなく、テスト結果を判定するために、期待する出力または結果の定義を含む。
- テストケース設計には、1)ホワイトボックステスト、2)ブラックボックステスト、3)妥当性確認テストの3種類の方法がある。

ホワイトボックステスト

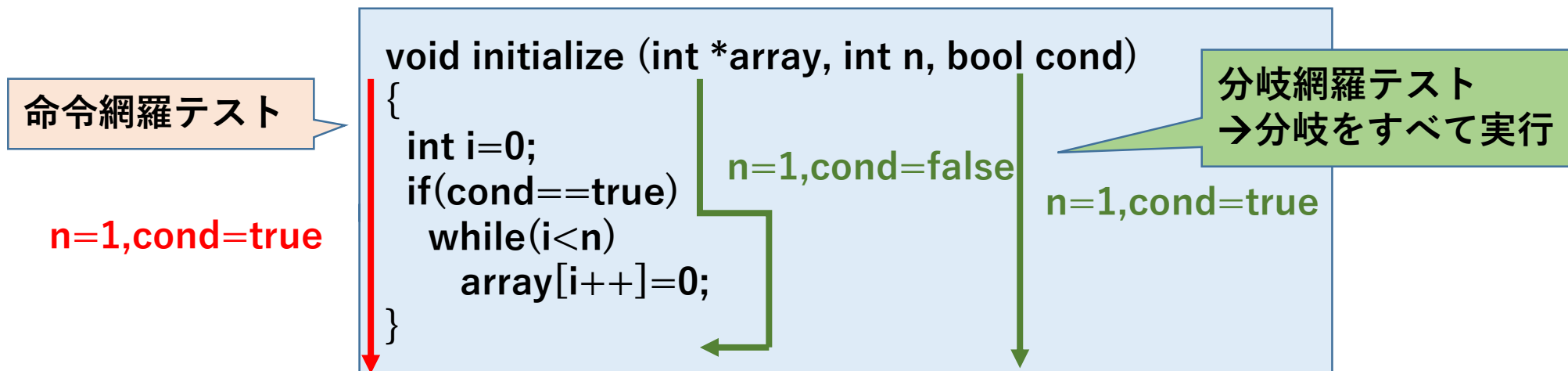
- プログラムの内部処理に注目し、プログラム構造を網羅するようにテストケースを見いだす方法である。

a) 命令網羅テスト:プログラムの全ステートメントが実行されているかどうかを網羅度の基準とする。この基準を命令網羅と呼ぶ。

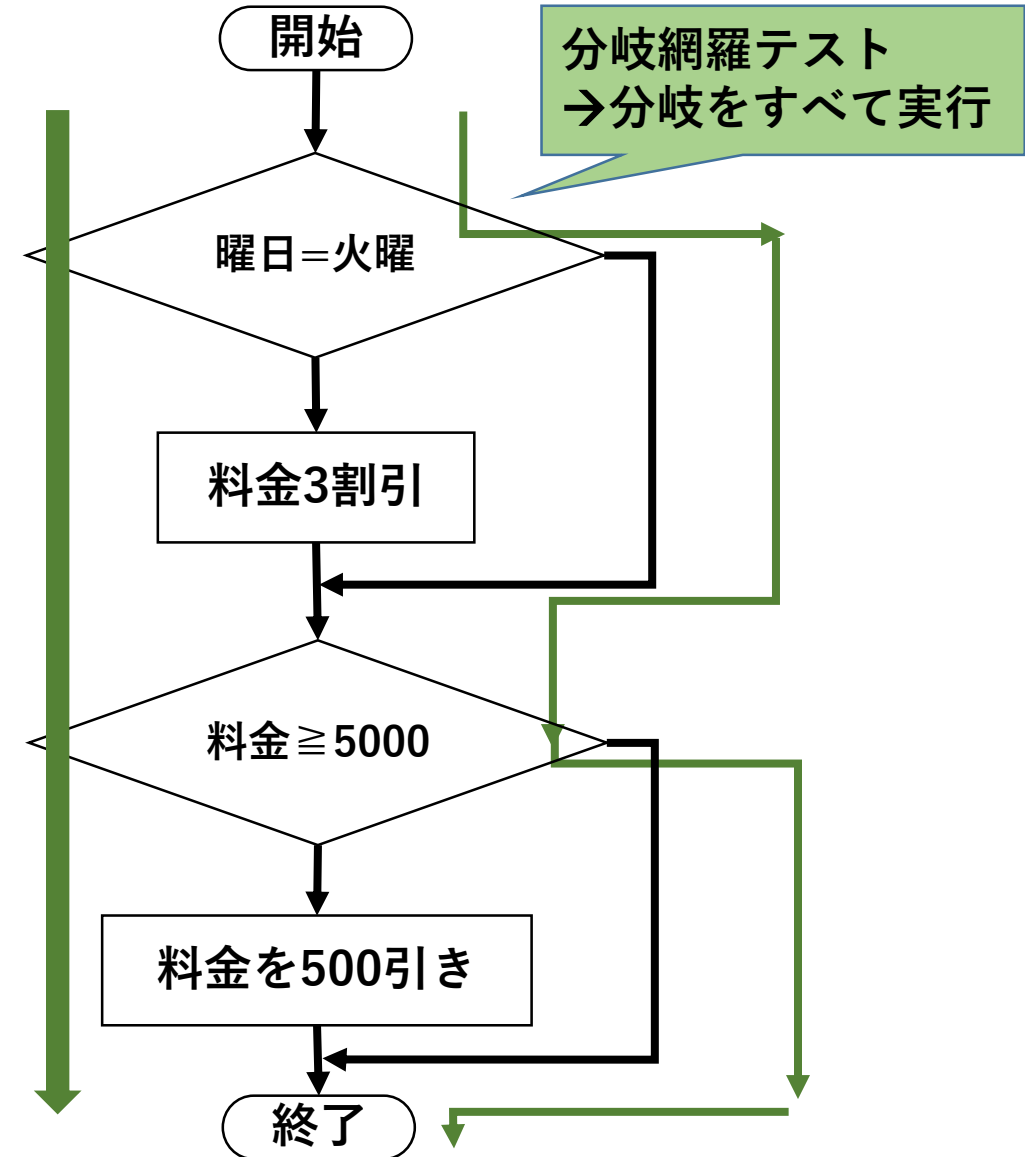
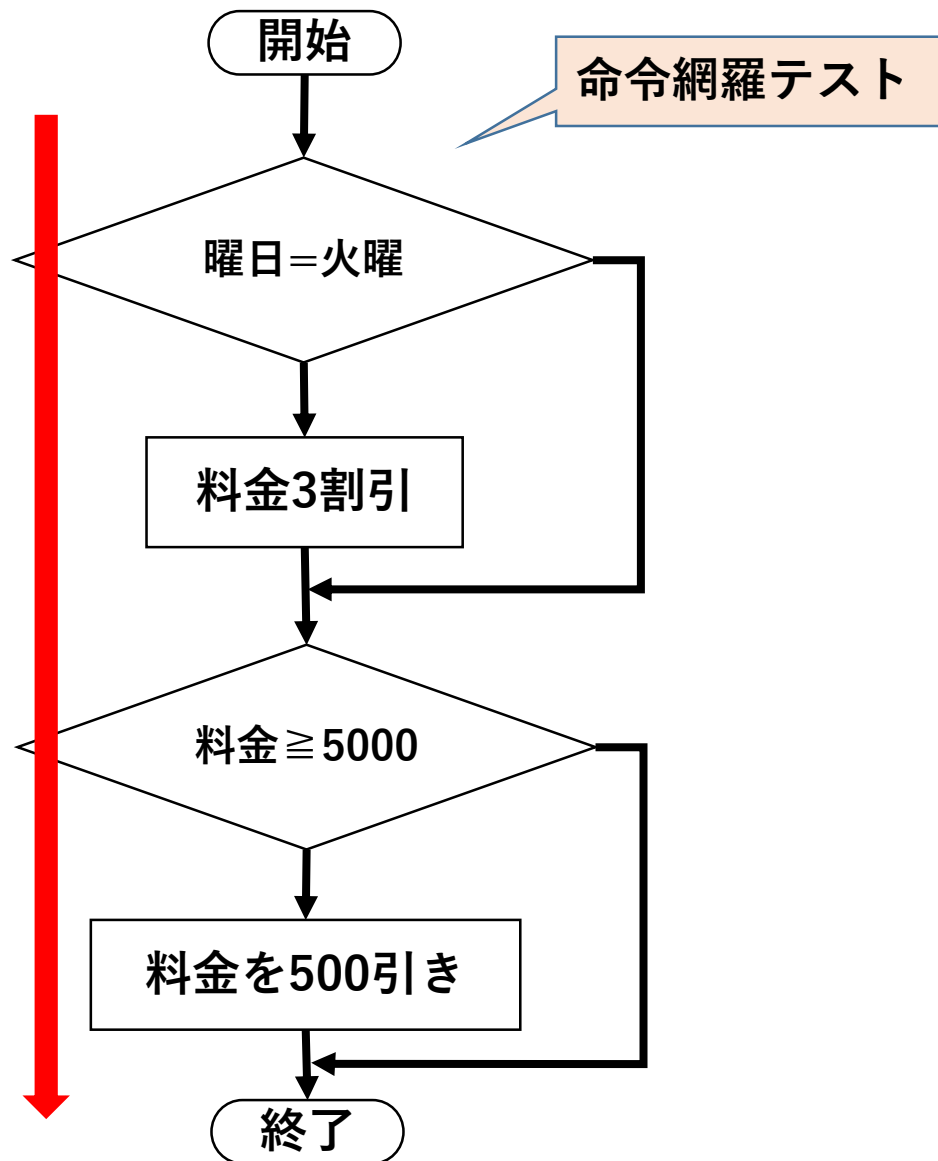
例:下記では「n=1,cond=true」で網羅度を達成できる。

b) 分岐網羅テスト:プログラムの条件分岐がすべて実行されているかどうかを網羅度の基準とする。この基準を分岐網羅と呼ぶ。

例:下記では「n=1,cond=true」「n=1,cond=false」で網羅度達成できる。



ホワイトボックステスト(命令網羅・分岐網羅)



課題6-1(締切:6/9)

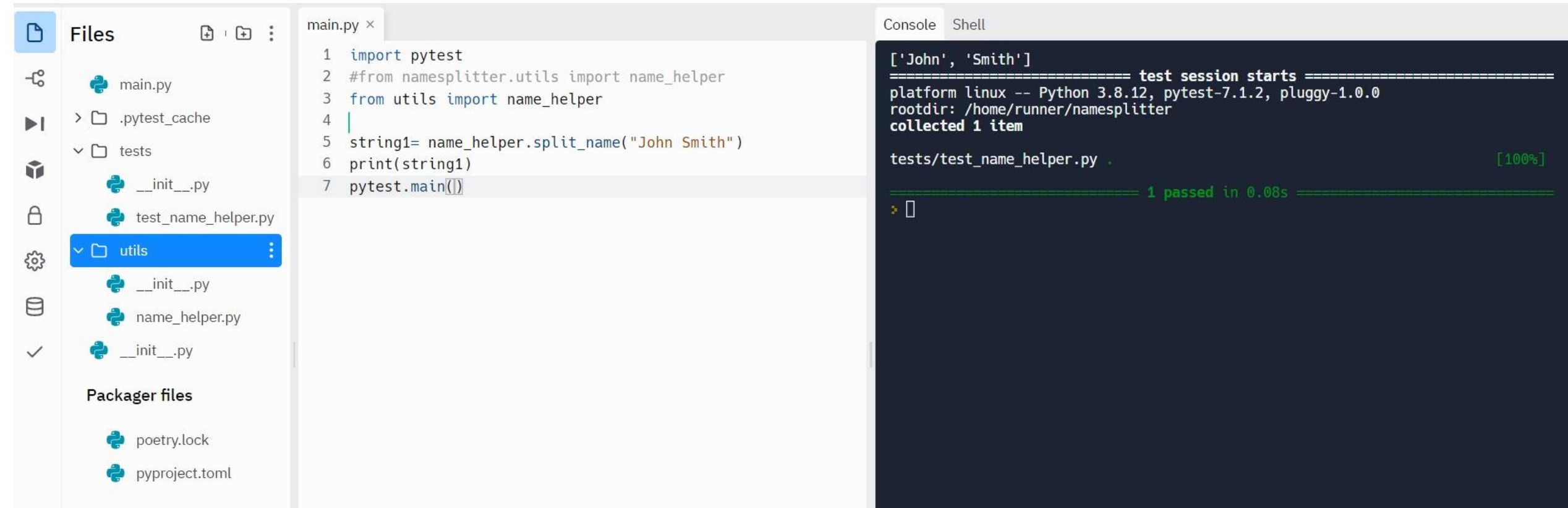
- 前スライドの処理について、命令網羅と分岐網羅で必要なテスト条件を記載してください。なお、テスト条件は(入力:料金,曜日 出力:割引後料金)としてください。

まずは、1)エクセルなどの表で作成し、その後に2)python実装して「ソースコードのテキスト」と「テスト結果のスクリーンショット」を提出ください。

提出:1)2)の両方を提出ください

Replit環境について(前回資料)

replit環境で、単体テストを行い、実行結果(テスト成功,テスト失敗)のスクリーンショットを提出ください。



The screenshot displays the Replit web interface. On the left, the 'Files' sidebar shows a project structure with a 'utils' folder containing 'name_helper.py'. The main editor shows 'main.py' with the following code:

```
1 import pytest
2 #from namesplitter.utils import name_helper
3 from utils import name_helper
4
5 string1= name_helper.split_name("John Smith")
6 print(string1)
7 pytest.main()
```

On the right, the 'Console' tab shows the output of the test session:

```
['John', 'Smith']
===== test session starts =====
platform linux -- Python 3.8.12, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/runner/namesplitter
collected 1 item

tests/test_name_helper.py . [100%]

===== 1 passed in 0.08s =====
```

Replit環境について(前回資料)

Google

replit

すべて ニュース 画像 動画 ショッピング

約 1,240,000 件 (0.78 秒)


https://replit.com ▼

Replit: The collaborative browser based IDE

Replit is a simple yet powerful online IDE, Editor, Compiler, Interpreter, compile, run, and host in 50+ programming languages.
このページに 4 回アクセスしています。前回のアクセス: 22/04/1

[Log In](#)

Replit is a simple yet powerful online IDE, Editor, Compiler ...



replit Features ▾ Blog Pricing Jam Teams Pro Teams for Education Careers [Log in](#) [Sign up](#)

Code, create, and learn together

Use our free, collaborative, in-browser IDE to code in 50+ languages — without spending a second on setup.

[<> Start coding](#)

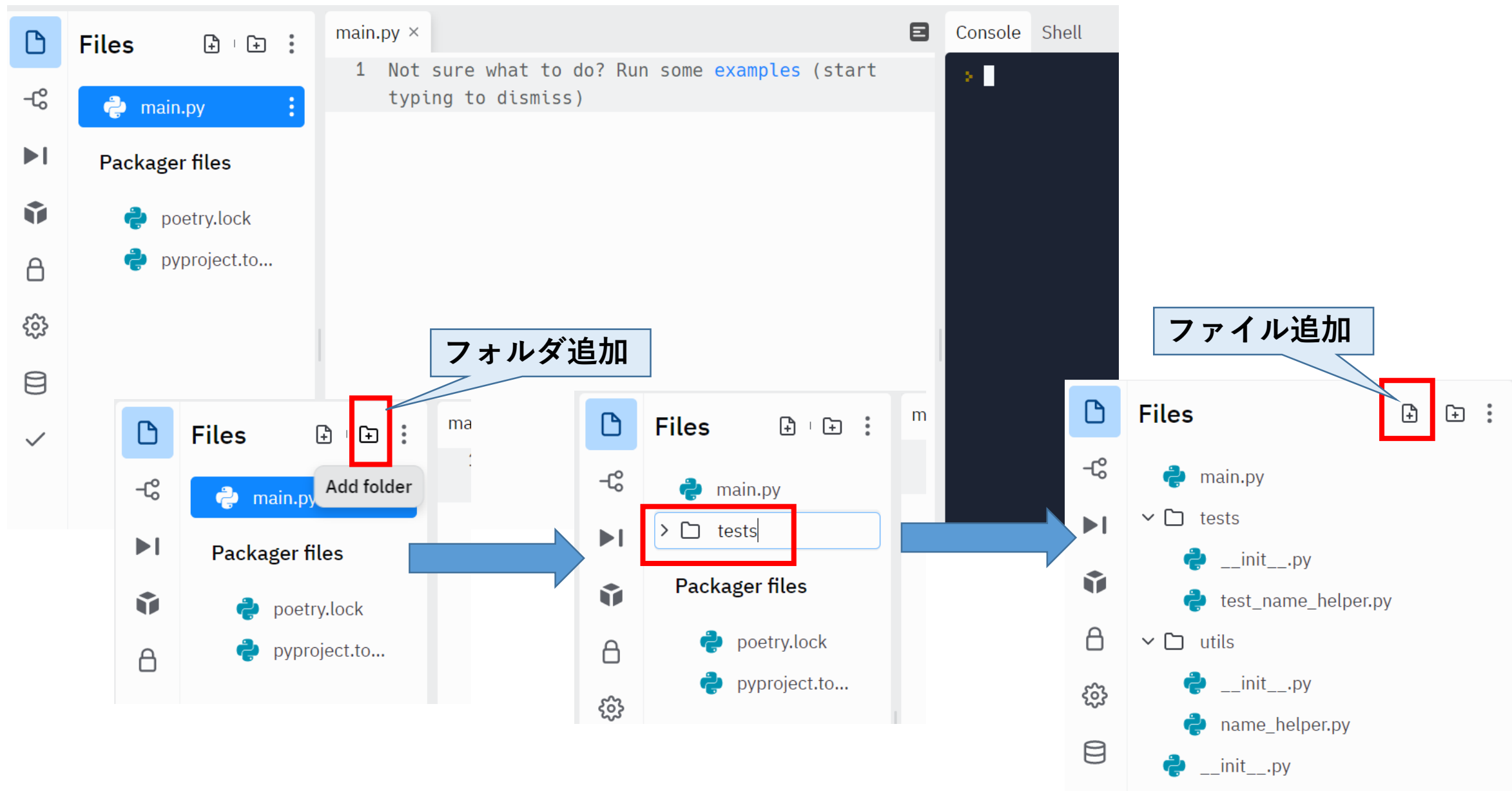
Replit環境について(前回資料)

The image illustrates the process of creating a new Repl environment on the Replit website. It consists of three sequential screenshots showing the user interface and the steps taken:

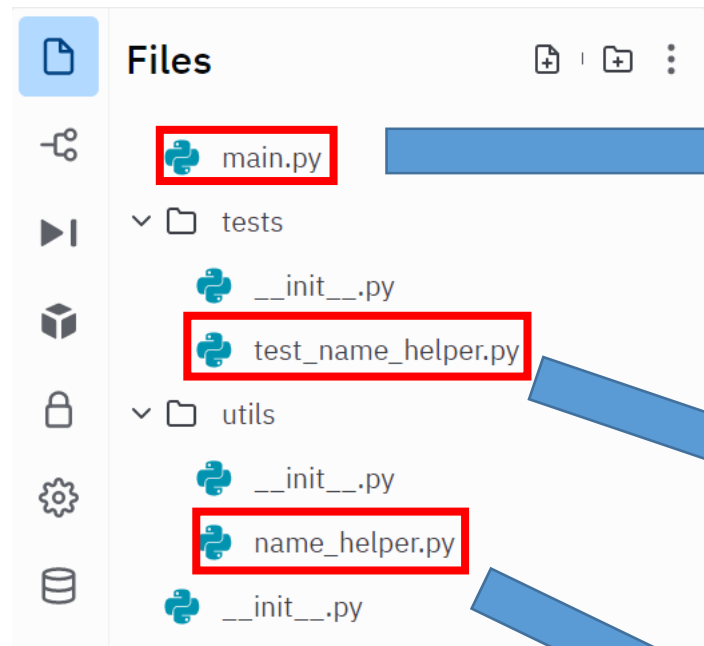
- Top Screenshot:** Shows the Replit homepage with the user profile `@tsuchidat`. The `+ Create` button in the top-left navigation bar is highlighted with a red box.
- Middle Screenshot:** Shows the "Create a repl" modal. The `Python` template is selected from the "Template" dropdown, and the `Python` language is chosen from the "Languages" section. The `+ Create Repl` button at the bottom is highlighted with a red box.
- Bottom Screenshot:** Shows the "Create a repl" modal with the title field filled with `namesplit` (highlighted with a red box) and the `+ Create Repl` button at the bottom highlighted with a red box.

Blue arrows indicate the flow of the process: from the `+ Create` button to the `Python` option, and from the `namesplit` title to the `+ Create Repl` button.

Replit環境について(前回資料)



Replit環境について(前回資料)






```
main.py ×  
1 import pytest  
2 from utils import name_helper  
3  
4 string1= name_helper.split_name("John Smith")  
5 print(string1)  
6 pytest.main()
```



```
tests/test_name_helper.py ×  
1 from namesplit.utils import name_helper  
2  
3 def test_two_names():  
4     assert name_helper.split_name("John Smith")  
       == ["John", "Smith"]
```

引数と返り値が想定通りになることを確認

```
utils/name_helper.py ×  
1 def split_name(name):  
2     first_name, last_name = name.split()  
3     return [first_name, last_name]
```


it  

 Run

 Invite 

main.py ×

Console

Shell




```
1 import pytest
2 from utils import name_helper
3
4 string1= name_helper.split_name("John Smith")
5 print(string1)
6 pytest.main()
```


```
['John', 'Smith']
===== test session starts =====
platform linux -- Python 3.8.12, pytest-7.1.2, pluggy-1.0.0
rootdir: /home/runner/namespace
collected 1 item

tests/test_name_helper.py .
[100%]

===== 1 passed in 0.09s =====
>
```

テスト成功

plit   ▶ Run 👤 Invite 

utils/name_helper.py ×  Console Shell

```
1 ▼ def split_name(name):
2     #     first_name, last_name = name.split()
3     #     return [first_name, last_name]
4     ||| return name
```

tests/test_name_helper.py ×

collected 1 item

tests/test_name_helper.py F
[100%]

===== FAILURES =====

test_two_names

```
def test_two_names():
>     assert name_helper.split_name("John Smith") == ["John", "Smith"]
E       AssertionError: assert 'John Smith' == ['John', 'Smith']
E         + where 'John Smith' = <function split_name at 0x7fa8196ebdc0>('John Smith')
E         + where <function split_name at 0x7fa8196ebdc0> = name_helper.split_name

tests/test_name_helper.py:4: AssertionError
===== short test summary info =====
FAILED tests/test_name_helper.py::test_two_names - AssertionError: assert 'Jo...
```

1 failed in 0.21s

テスト失敗

提出:テスト成功とテスト失敗のスクリーンショットを提出ください

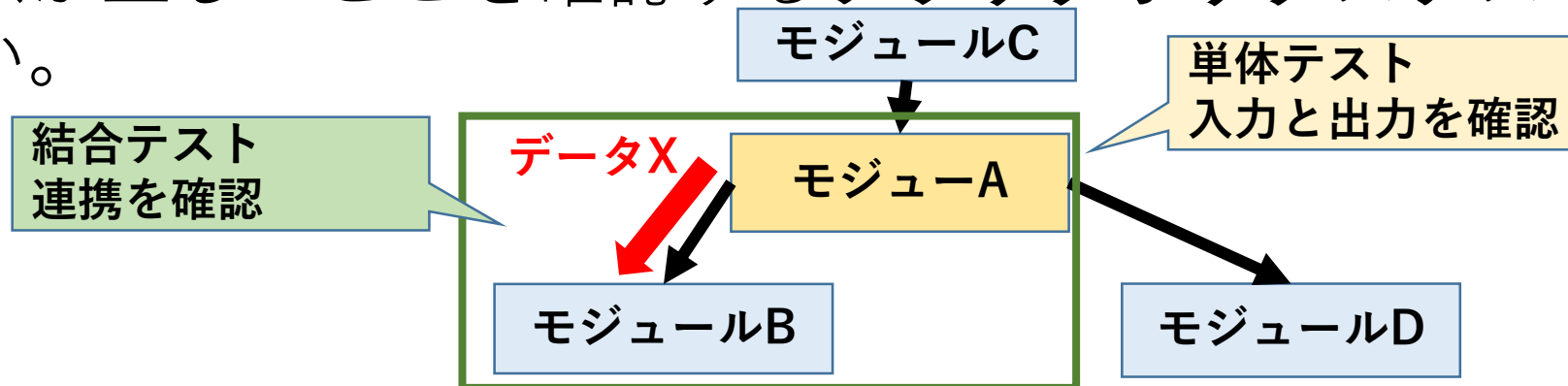
2)結合テスト

2)結合テスト:複数のモジュールを組合わせて動作を確認するテストである。外部設計書どおりに、モジュール間の連携が実現されているかを確認する。

例:モジュールAで求めたデータX(計算結果)が、別のモジュールBへ入力データとして渡され、それをつかってモジュールBが適切に処理結果を出力しているかを確認する。

結合テストでは、単体テストが終了したプログラムを組合わせて動作を確認する。プログラム間のインタフェースに問題がないか、必要な機能が実現されているかどうかの確認を行う。

結合テストでは、プログラムの内部構造には関知せず、入力と出力の対応関係が正しいことを確認する**ブラックボックステスト**が用いられることが多い。



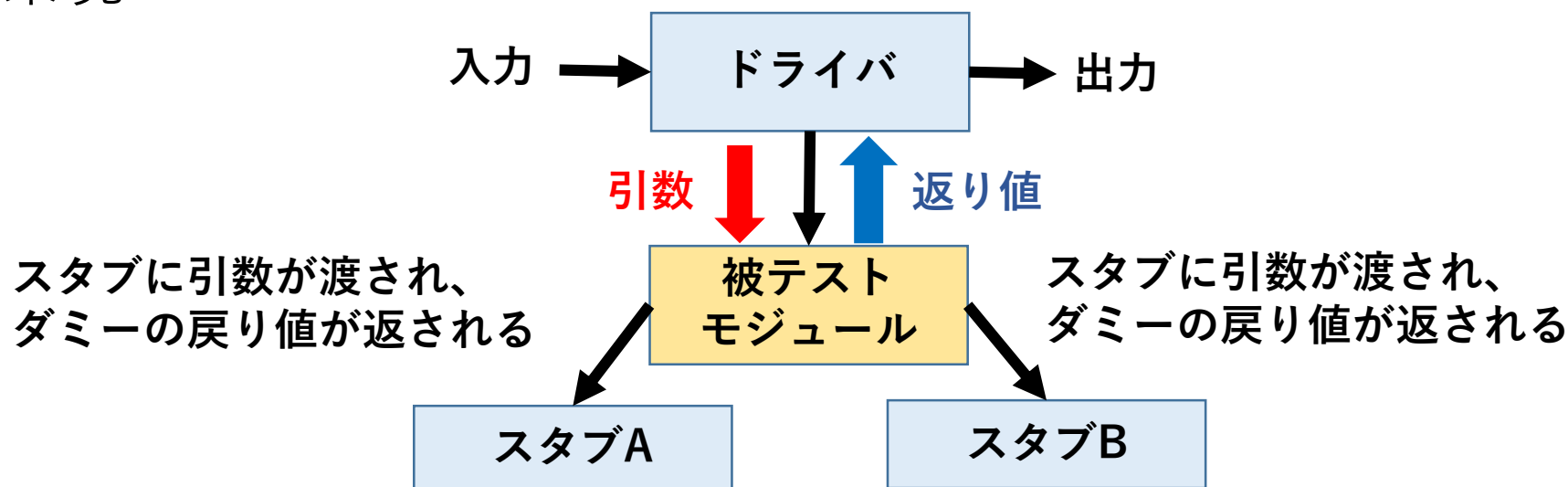
テスト環境の準備

単体・結合テストを行う際、被テストモジュールだけでは、テストを実行できない。被テストモジュールに渡す引数が必要。

ドライバ:実行を制御し、テスト入力と結果出力を行うプログラム

スタブ:被テストモジュールから呼び出すモジュールがない場合、呼び出すダミーのモジュール

テスト環境:ドライバとスタブなど、被テストモジュールのテスト実行に必要な環境



ブラックボックステスト(同値分割)

- プログラムの機能記述(仕様)に基づき、可能な入力の組合せとその入力に対する出力を選択する方法である。

a)同値分割:プログラムの入力条件を使って、テスト入力空間を分割する方法である。プログラムにとって同じ扱いを受けるはずの値の範囲である同値クラスという概念を考慮する。プログラムにとって有効な入力を同値有効クラス、無効なものを無効同値クラスと呼ぶ。

例えば、以下の要求仕様について考える。

入力A:1～99まで入力可能、 入力B:1～99まで入力可能、 出力C:A×B

```
if(a>0&&a<=99)
    //正しい値が入力されたときの処理
else
    //間違った値が入力されたときの処理
```

境界値(例:0,1,99,100)と代表値(例:範囲の中央値 49)

無効同値(~0)

有効同値(1～99)

無効同値(100～)

それぞれの代表値だけをテストすることで、テスト数増加を防ぐ

ブラックボックステスト(同値分割)

メリット:入力条件を網羅する最少のテストケースを作り出すことができ、系統的な手法なので個人差が現れにくい。応用範囲が広く、有効性が高い。

発見できる欠陥:プログラム処理での論理ミスによる欠陥が見つけられる。

例)入力値0～99で有効範囲1～64の場合に、`if(input<=64){/*有効処理*/}`としている欠陥

ブラックボックステスト2(限界値分析)

b)限界値分析:同値分割法では見つからないプログラミング上の実装ミスを見い出すための技法である。プログラム処理の変わり目と思われる値に焦点を絞り、テストケースを設計する。

例: `if(input>0&&input<64){/*処理*/}`で、正しくは`input<=64`

例:ユーザがエディタの印刷機能を使う場合を考える。1ページ未満の印刷をユーザが要求した場合はエラーとする。

```
if(a>=1)
    //印刷機能
else
    //エラー処理
```

```
if(a>1)
    //印刷機能
else
    //エラー処理
```

```
if(a>=2)
    //印刷機能
else
    //エラー処理
```

```
if(a>=1)
    //印刷機能
/*else
    //エラー処理
*/
```

```
if(a>=1&&a<10)
    //印刷機能
/*else
    //エラー処理
*/
```


ブラックボックステスト2(限界値分析)

例:ユーザがエディタの印刷機能を使う場合を考える。1ページ未満の印刷をユーザが要求した場合はエラーとする。

```
if(a>=1)
    //印刷機能
else
    //エラー処理
```

//タイプ1のバグ

```
if(a>1)
    //印刷機能
else
    //エラー処理
```

```
if(a>=2)
    //印刷機能
else
    //エラー処理
```

```
if(a>=1)
    //印刷機能
/*else
    //エラー処理
*/
```

```
if(a>=1&&a<10)
    //印刷機能
/*else
    //エラー処理
*/
```

正しいテストケース例:

テストケース1:1を入力

結果:正常な処理がなされること

テストケース2:0を入力

結果:エラー処理がなされること

適切でないテストケース例:

テストケース1:-1を入力

テストケース2:2を入力

とすると、タイプ1のバグが見つからない

異なる処理が行われる一付近の二地点をテストケースとする

ブラックボックステスト3(ランダムテスト)

- 被テストプログラムの入力空間から、入力データをランダムに抽出することでテストケースを作り出す方法である。
- 通常の運用環境での入力に関する確率モデルが存在するなら、その確率モデルの入力データを用いることで、実運用に近い状況のテストが行える。

利点:ランダムテストは、コンピュータによって容易に多数のテストケースを作り出せ、テストケース設計者が思いもよらぬテストケースを作ることができる。

欠点:テストケース数に対してプログラム構造に対する網羅度が低い

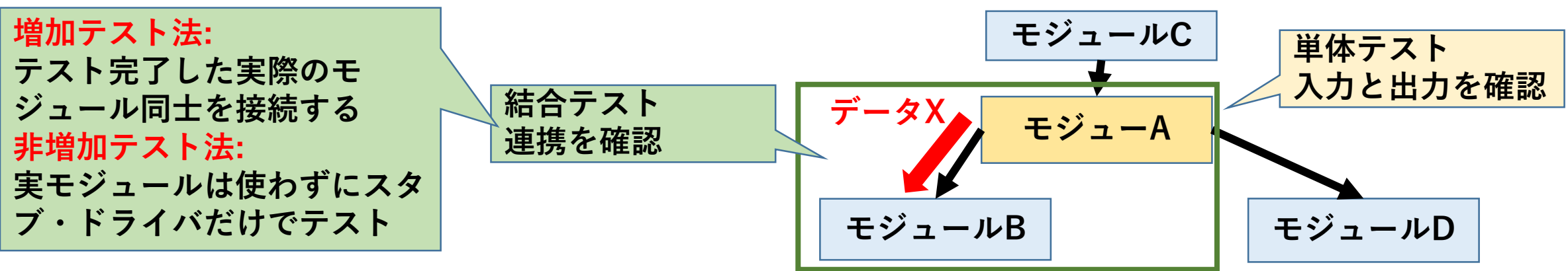
テスト戦略

- プログラムをどの部分からテストしていくかは、テストの効率に著しく影響を与える。

増加テスト法: モジュールを結合させながら単体・結合テストを順次すすめていくテスト戦略である。

① **トップダウンテスト:** 呼ぶ側のモジュール、つまり上位モジュールから先にテストを実施し、順次下位モジュールを加えながらテストを進める。

② **ボトムアップテスト:** 呼ばれる側のモジュール、つまり下位モジュールから先にテストを実施し、順次上位モジュールを加えながらテストを進める。



3)受入・導入テスト

3)受入・導入テスト:個々のモジュールや機能を統合した状態で、要求定義の内容が実現されているかを確認するテストです。

結合テストにより、外部設計書通りの機能が実現できていることを確認したプログラムと、そのプログラムが動作するハードウェア、ネットワーク、利用者端末などを組合わせ、本番と同じ環境、あるいは本番に近い環境で動作を確認する。

テスト妥当性確認テスト

- ユーザ要求の非機能的な側面を満足していることを確認するテストである。これまで述べてきたテスト設計技法とは異なる視点からテスト設計を行うことが必要である。妥当性確認のために汎用的に用いることができるテスト設計技法はないため、創造性や知識、経験が必要である。下記は主な例である。

	テスト種類	テスト対象
1	大容量テスト	想定データ量をはるかに超えた場合におけるシステムのふるまい
2	ストレステスト	過負荷状態におけるシステムのふるまい
3	セキュリティテスト	データ保全性
4	性能テスト	応答時間や処理速度
5	記憶域テスト	プログラムで使用する記憶域のサイズ
6	構成テスト	種々のシステム構成下での動作

妥当性確認テストの例(性能テスト)

超高速データベースエンジン Hitachi Advanced
Data Binder : 世界初、日立が「TPC-H」 100TB
クラスに登録

世界初、

**統計情報取得ユーティリティの実行時間の性能を確認
まずは、手元環境(小規模なブレードサーバとディスク
アレイ)で確認し、その後に本番環境**

「Hitachi Advanced Data Binder プラットフォーム」
が打ち立てた記録とは？

この記事に記載されている情報は2014年3月

2013年10月19日、「Hitachi Advanced Data Binder プラットフォーム」が、データ性能に関する業界標準の**ベンチマーク**である「TPC-H」の最大クラス(100TBクラス)で初めて登録されました。

機能要件:システムは～できる

例:統計情報取得ユーティリティは、データベースの適切な実行プランが生成できる統計情報を取得できる。

→各表のレコード数の有効数字はどれくらいまで取得するのか？→クエリ最適化アルゴリズムにより決まる

非機能要件:システムの持つべき性能、信頼性、セキュリティなど

例:統計情報取得ユーティリティの実行時間は(ベンチマークの主な評価項目である)クエリ実行時間ではなくデータロード時間に含まれる。そのため、ある程度の実行時間までは許容されるが、極端に長いのは望ましくない。

https://www.hitachi.co.jp/products/it/bigdata/platform/data-binder/activity/special_tpc/index.html

妥当性確認テストの例(性能テスト)

超高速データベースエンジン Hitachi Advanced

導入事例	金融：不正取引検知
課題	監督官庁からの要請により、取引履歴から不正取引を調査したいが、夜間バッチ処理の関係で前日分の取引履歴までしか調査できないという課題があった。
効果	◎ HADBのデータインポートは、オンライン処理中でもバックグラウンドで高速インポートできるため、当日の最新データまで調査対象に含めることができた。 ◎ HADBの高速検索により、データマートの開発は不要。大量の取引履歴データベースに対して直接検索できた。

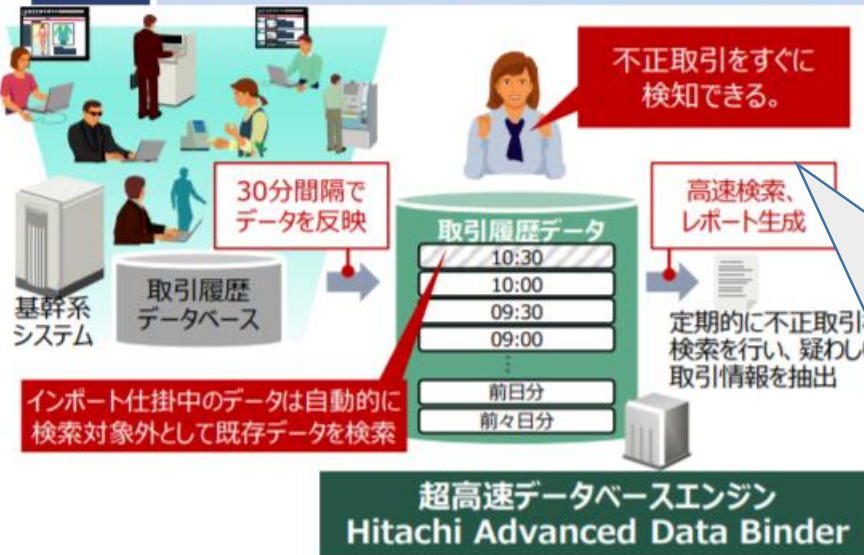
機能要件:システムは～できる

例:バックグラウンドインポートは、データ検索を中断することなく、データの一括インポートができる。

#普通のデータベースは、データ追加しながら検索は不可

仕様(マニュアルに書いてある使い方)の範囲で問題無く動くことを実測測定する。

→アプライアンス(ソフトとハードのセット販売)の場合は、検証環境が限定されてよい。



非機能要件:システムの持つべき性能、信頼性、セキュリティなど

例:バックグラウンドインポートは、仕様(例:何回までインポート可能)で定められている全ての組合せにおいて、問題無く動作する。

問題無くとは、例えば極端な性能低下やデータ容量の肥大化が起こらないことを指す。

→一回(どんなデータに対して)バックグラウンドインポートすると、どれだけの二次記憶やメモリ容量が必要かを記載し、その通りに動作する必要

テスト妥当性評価

- プログラムの品質を直接測ることはできない。テストを品質管理活動と捉えると、テストをどのように、どれだけ行えば、プログラムの品質をどれだけ保証できるのかが重要になる。これを測る基準がテスト妥当性である。
 - テスト妥当性は、あるテスト集合の実行結果と、被テストプログラムの信頼度を結びつけるための概念である。
- テストの品質を測る尺度で有り、テスト終了判定に用いる。
- 1)網羅性基準、2)欠陥除去基準、3)運用的基準がある。

網羅性基準

- テストに使用するテストケースが、テスト空間をどれだけ検査したかという基準である。テスト空間を絶対的な尺度で計測する手段は無いため、被テストプログラムの測定可能なある側面を捉えて網羅性を計算する。

a)仕様に基づく基準:被テストプログラムの仕様を基準としたもので、プログラムが仕様を満たしている尺度として利用される。

①**機能記述:**仕様書内の自然言語の機能記述に基づくもの
状態ベースの仕様記述に基づくもの

②**入力/出力条件:**入力条件に基づくもの
#2つの入力についての任意の組合せなどもある
入力条件/出力条件両方にに基づくもの

b)プログラムに基づく基準:

①**制御構造:**全ステートメントを網羅(命令網羅)、条件分岐の真偽両方を網羅(分岐網羅)

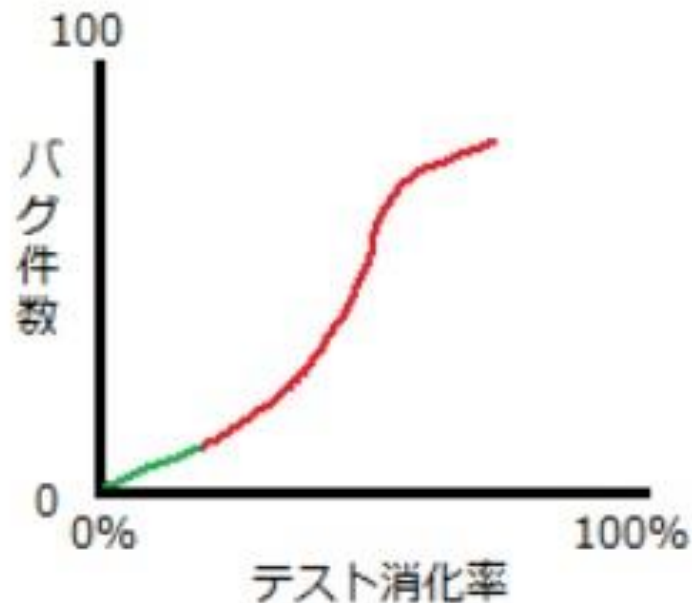
②**データフロー:**変数値のプログラム内での変更(変数への代入)パターンの組合せを追跡

欠陥除去基準

成長曲線モデル

あるプログラムを、異なるテストデータセットで順次テストすると、新たに検出される欠陥は徐々に減って限りなく0に近づくはずである。

→欠陥の累積数は、時間とともにプログラム内の欠陥数に収束する。この考えに基づき、欠陥の累積数を時間の関数 $f(t)$ で表現したものを欠陥検出累積数の成長曲線モデルと呼ぶ。



運用的基準

- 信頼性工学の分野では、被検査対象に欠陥が発生するまでの平均時間であるMTBF(Mean Time Between Failure)を基準に、信頼性を測定・評価する。この評価基準をソフトウェアに適用しようとしたものが運用基準である。
- 運用基準では、ソフトウェア製品での運用環境下において、欠陥が発生するまでの時間を測定する。

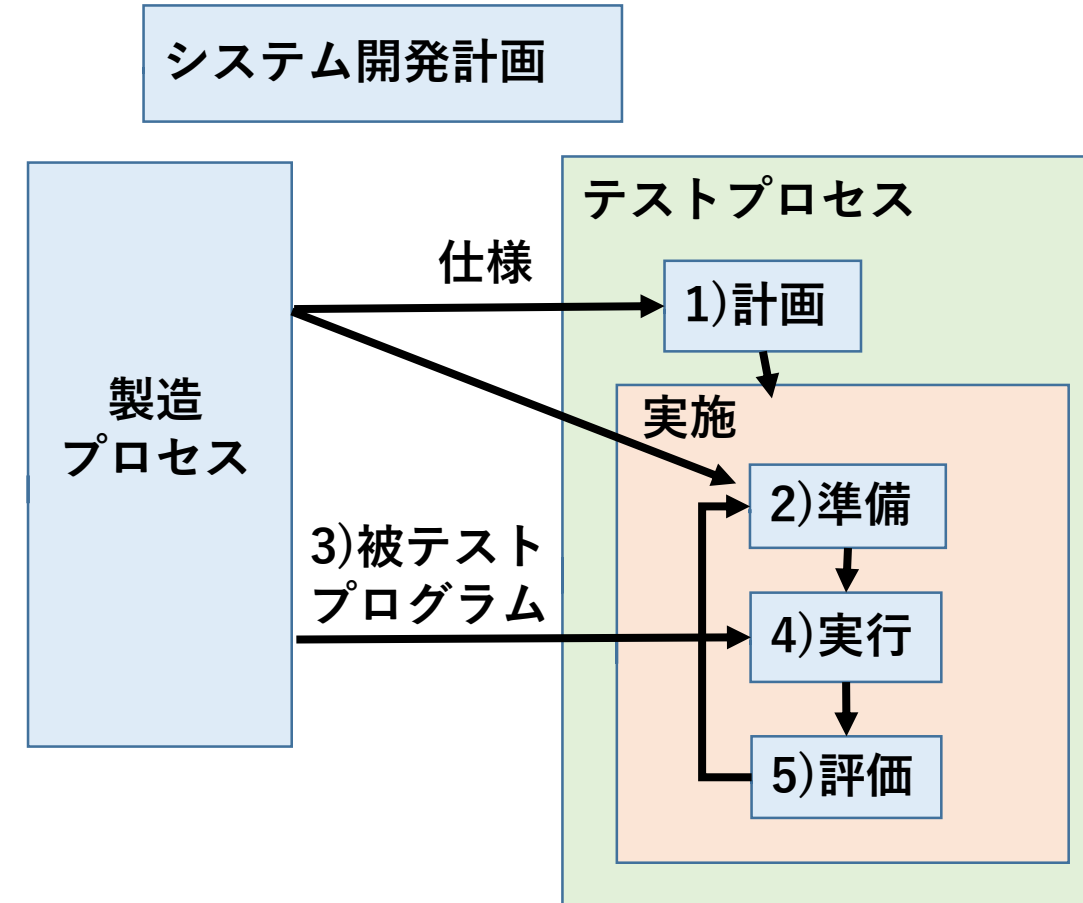
→ 何時間連続で使って大丈夫か？例:メモリリークなどあるかもしれないが、小規模なリークなら、定期的のリブートするなら問題無いかも？例:エクセル

MTBF=システムの稼働時間/故障回数

テストプロセスと技法

• テストプロセスと技法

- 1) 計画段階では、品質目標値の設定とテスト手順の計画を行う。
- 2) 準備段階では、テスト環境の準備とテストケース設計を行う。
- 3) 製造プロセスより被テストプログラムを受け取ると、テスト実行と評価を行えるようになる。
- 4) 実行段階では、テスト実行と結果確認・記録を行う。
- 5) 評価段階では、テスト実行結果が計画段階で設定した品質目標値を満たしているか判定し、満たせばテストプロセスを終了し、満たさなければプログラム修正後、さらにテスト実施を繰り返す。



保守

以前は「運用開始後の不具合を修正する活動」と定義されていたが、最近では「ソフトウェアをシステムの新たな目的と運用環境の変化に適合させていく継続的な活動」という認識に変化している。保守は、大きく3つの作業に分類できる。

a)不具合修正のための保守:運用開始後に生じたソフトウェアの運用上の問題や欠陥に対する修正

b)適合・改善・拡張のための保守:新たなユーザ要求と、ソフトウェア環境やハードウェア環境およびシステム環境の変化へ適合させるための修正

#ハードウェア変更、仕様OS変更、性能改善、追加機能要求

c)予防のための保守:設計時における保守容易性の確保の事を指す。つまり、ユーザの将来の要求と運用環境の変化に対応出来るように、プログラムの構造を柔軟にしておくことである。

課題6-2(締切:6/9)

ある動物飼育用の室温計

室温計は現在の室温表示欄と、ある動物にとって適温か表示するメッセージ欄がある。室温に応じて以下のメッセージが表示される。表示メッセージのpythonのテストケースを同地分割と限界値分析を用いて作成ください。なお、「表示メッセージは関数返り値」としてください。

前提:室温計の精度は0.1°C単位とする

室温計で計測できる下限および上限を考慮する必要はないとする。

提出:1)テストケースの一覧、2)実行結果のスクリーンショット、pythonソースコードのテキスト

No	室温	表示メッセージ
1	23.0°C未満	寒い
2	23.0°C以上25°C未満	快適
3	25°C以上	暑い

寒い、快適、熱いの
「境界値(下限)、代表値、境界値(上限)」
をテストケースとする。
#境界値の下限、上限は無い場合もある

(余裕があれば)課題6-3(締切:6/9)

料理の材料の重さを量るキッチンスケールのテストを考える

- ・電源を入れた時を0gとして、最大3000gまで、1g単位で重さを計測し、表示できる。3000gを超えた場合、表示欄にエラー「EEEE」が表示される。

#重さがマイナスの場合も、エラー「EEEE」が表示されるとする。

電源を入れる前に、器を載せて、電源入力後に外した場合はマイナス
pythonのテストケースを同地分割と限界値分析を用いて作成ください。なお、「表示(数値やEEEE)は関数返り値」としてください。

提出:1)テストケースの一覧、2)実行結果のスクリーンショット、pythonソースコードのテキスト

マイナスの無効クラス、有効クラス、プラスの無効クラス
「境界値(下限)、代表値、境界値(上限)」
をテストケースとする。
#境界値の下限、上限は無い場合もある

