Techniques in Program Development for Statistical and Quantum Physics

統計物理・量子物理のためのプログラム開発技術

Synge Todo / 藤堂眞治

Department of Phys., UTokyo / 東京大学大学院理学系研究科

Agenda

- Part I: Test-Driven Development / テスト駆動開発
  - What's TDD？ / テスト駆動開発とは？
  - TDD Demonstration / テスト駆動開発の実演

- Part II: Version Control / バージョン管理
  - What's VCS？ / バージョン管理システムとは？
  - Branch & Merge / ブランチとマージ
  - Using GitHub / GitHubの利用
  - Continuous Integration (CI) / 継続的インテグレーション

- Sample Code Repository
  - https://github.com/wistaria/tdd-tutorial

Prerequisites

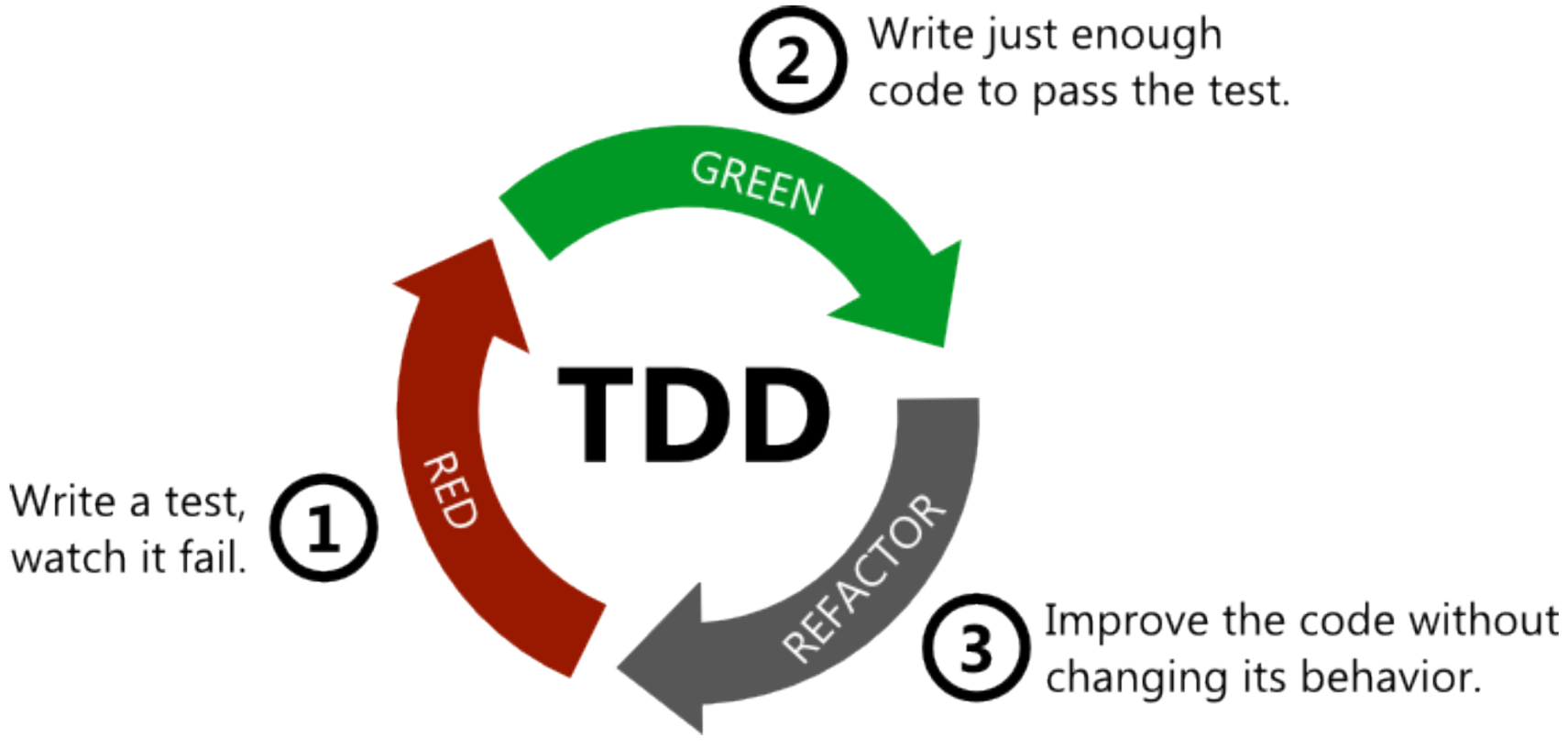
- Prerequisites / 前提とする基礎知識
  - Basic UNIX shell operations / 基本的なUNIXシェル操作
    - UNIX入門 https://github.com/cmsi/malive-tutorial/blob/master/introduction/shell.md
  - Basic Programming in C++ and/or Python / C++やPythonによるプログラミングの基礎

Part I: Test-Driven Development / テスト駆動開発

What's TDD？ / テスト駆動開発とは？

What’s Test-Driven Development?

- Test-driven development（TDD）/ テスト駆動開発
  - Red: Write a failing automated test before you write any code / コードを書く前に失敗する自動テストコードを必ず書く
  - Green: Write just enough code to pass the test / テストをパスする最低限のコードを書く
  - Refactor: Remove duplication / 重複を除去する



https://hashstrix.com/post/what-is-test-driven-development-tdd/

Traditional Program Development

- Traditional Program Development / 従来型のプログラム開発
  - Write specifications / 仕様書を書く
  - Write code / コードを書く
  - Perform tests / テストを行う
  - Debugging / デバッグする
  - Write documentation / マニュアルを書く
  - Release / リリース

- Problems / 問題点
  - Specifications change during the course of development / 開発の途中で仕様が変わる
  - Can not know if all of the specifications have been tested / 仕様の全てがテストできているか分からない
  - Can not know if bugs still exist / バグが残っているかどうか分からない
  - Can not know if it’s completed / 完成したかどうかが分からない

Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.

デバッグは、最初の段階でコードを書くより2倍は難しい。ゆえに、可能な限り賢いコードを書いたとしたら、そのコードにバグがあったとき、あなたがそれをデバッグできるほどの賢くないのは明白である

— Brian W. Kernighan

Test-Driven Development

- TDD =
  - Technology to develop ”clean code that works” effortlessly / 「動作するきれいなコード」をストレスなしに開発する技術

- Advantages of TDD / TDDの利点
  - Small time/cost to fix bugs / バグを修正するまでの時間・コストが小さい
  - The tests themselves are the specification / テストそのものが仕様書
  - No need to worry about if the bug is still there / バグが残っているか心配する必要がない
  - Can know if it is completed or not / 完成したかどうかがわかる
  - It feels good to be ”Green” so often. Less stress / 頻繁にGreenになるので気持ちがいい。ストレスが少ない
  - Can refactor with confidence / 自信をもってリファクタリングできる
  - Easier to maintain as there are no duplicates / 重複がないので、メンテナンスが楽
  - Clean code is actually faster / きれいなコードは実際速い

Writing Comments

- Comments in Source Code / ソースコード中のコメントについて
  - A comment is of zero (or negative) value if it is wrong / たとえコメントを入れても、それが不適切なものであれば価値はゼロ(あるいはマイナス)である (B. W. Kernighan and P. J. Plauger)
  - Comments are not running code / 実行されるのはコメントではない
  - Avoid duplication / 重複を避ける
  - Let the code speak for itself / コード自身に語らせる

- How to write comments / コメントの書き方
  - Write ”How” in code / コードには「How」
  - Write ”What” in tests / テストコードには「What」
  - Write ”Why” in commit logs / コミットログには「Why」
  - Write ”Why not” in code comments / コードコメントには「Why not」

Tools for TDD

- Tools for TDD / テスト駆動開発に必要なツール類
  - Text editor / テキストエディタ 
    - Visual Studio Code, Vim, Emacs, nano, Mousepad...
  - Programming language / プログラミング言語
    - C++ compiler, Python interpreter / C++コンパイラ, Pythonインタプリタ
  - Build tools / ビルドツール
    - Make , AutoTools, CMake 

  - Test framework / テストフレームワーク
    - C++: Google Test, Boost.Test, Catch2 

    - Python: DocTest, UnitTest, pytest 

  - Version control system (VCS) / バージョン管理システム
  - Continuous integration (CI) / 継続的インテグレーション
    - Subversion, Git, GitLab, GitHub, Jenkins, Travis CI 

Make & CMake

- Make (1976～！)
  - Automates a series of tasks based on dependencies / 依存関係に基づき、一連の作業を自動化
  - Describe dependencies and work procedures in the Makefile / 依存関係や作業手順はMakefileに記述する
  - UNIX/Linux standard tool / UNIX/Linux標準ツール

- CMake (1999～) https://cmake.org/
  - Utility that generates a Makefile / Makefileを生成してくれるユーティリティ
  - Settings are written in CMakeLists.txt / 設定はCMakeLists.txtに記述する
  - Automatic detection of file dependencies / ファイルの依存関係の自動検出
  - Automatic detection of various external libraries / さまざまな外部ライブラリの自動検出
  - test execution function / テスト実行機能 (ctest)

Tools Installation Example

- on macOS

```
$ xcode-select --install
$ /bin/bash -c ”$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/HEAD/install.sh)”
# for zsh on Apple silicon (M1/M2)
$ echo ’eval $(/opt/homebrew/bin/brew shellenv)’ >> $HOME/.zprofile
# for bash on Apple silicon (M1/M2)
$ echo ’eval $(/opt/homebrew/bin/brew shellenv)’ >> $HOME/.bash_profile
```

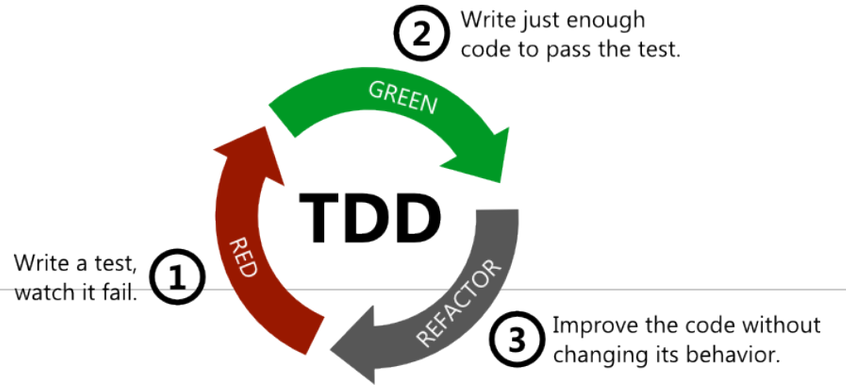
```
$ brew install cmake
```

- on Debian/Ubuntu Linux

```
$ sudo apt-get install -y build-essential python3-dev cmake git
```

- on Windows
  - use WSL2 ?

TDD Workflow



- **Red**
  - Write a failing automated test before you write any code / コードを書く前に失敗する自動テストコードを必ず書く
    - git fetch → git merge → git checkout
    - vim → cmake → make → ctest
- **Green**
  - Write just enough code to pass the test / テストをパスする最低限のコードを書く
    - [vim → cmake → make → ctest ...] → git add → git commit ...
    - git checkout → git merge → git push
- **Refactor**
  - Remove duplication / 重複を除去する
    - git checkout
    - [[vim → cmake → make → ctest ...] → git add → git commit ...]
    - git checkout → git merge → git push

Don’t Repeat Yourself（DRY）

- DRY principle / DRY原則
  - Fundamental principle for creating simpler, more maintainable, higher-quality programs / シンプルで、品質が高く、保守もしやすいプログラムを作るための最も基本的な指針

- Don’t Repeat Yourself / 繰り返しを避ける
  - Duplication is waste / 重複は無駄
  - Repetition in process calls for automation / 作業の重複は自動化で避ける
  - Repetition in logic calls for abstraction / ロジックの重複は抽象化で避ける

- DRY ⇔ WET
  - ”write everything twice”
  - ”write every time”
  - ”we enjoy typing”
  - ”waste everyone’s time”

Measuring programming progress by lines of code is like measuring aircraft building progress by weight.

プログラミングの進展をコードの行数で測るのは、飛行機建造の進展を重量で測るようなものだ

— Bill Gates

What Are Good Tests?

- F.I.R.S.T. principle
  - Fast / 速い
  - Independent / 互いに独立している
  - Repeatable / 再現可能性
  - Self-Validating / 自律的検証？ → 自動化されている？
  - Timely / 適切なタイミング

- A-TRIP principle
  - Automated / 自動化されている
  - Thorough / 徹底的
  - Repeatable / 再現可能性
  - Independent / 互いに独立している
  - Professional / プロの？

Part I: Test-Driven Development / テスト駆動開発

TDD Demonstration / テスト駆動開発の実演

TDD Demonstration

- TDD Demonstration / テスト駆動開発実演
  - Fibonacci sequence / Fibonacci数列
    - $f(0) = 0, f(1) = 1, f(n) = f(n-1) + f(n-2)$
    - 0, 1, 1, 2, 3, 5, 8, 13, ...

- TDD iterations
  - 1st iteration: cmake && make
  - 2nd iteration: cmake && make && ctest
  - 3rd iteration: fibonacci(0) == 0
  - 4th iteration: fibonacci(1) == 1
  - 5th iteration: fibonacci(2) == 1 && fibonacci(3) == 2
  - 6th iteration: fibonacci(10) == 55
  - 7th iteration: fibonacci(-1) == 1 && fibonacci(-2) == -1
  - 8th iteration: speed up
  - 9th iteration: fibonacci(100) == 354224848179261915075
  - 10th iteration: Lucas series

TDD Demonstration: 1st iteration (C++)

- Test to pass: cmake && make
- Create source & build directories / ソース・ビルドディレクトリの作成
  - fibonacci: source directory / ソースコードを作成するディレクトリ
  - fibonacci/build: build directory / ソースコードをコンパイルするディレクトリ
- Pass the location of source directory (directory with CMakeLists.txt) to cmake / cmakeの引数にソースディレクトリ(CMakeLists.txtのあるディレクトリ)を指定

```
$ mkdir fibonacci
$ mkdir fibonacci/build
$ cd fibonacci/build
$ cmake ..
CMake Error: The source directory ”.../fibonacci” does not appear to contain
CMakeLists.txt.
```

TDD Demonstration: 1st iteration (C++)

- Test to pass: cmake && make
- Create CMakeLists.txt

fibonacci/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.14)
project(fibonacci CXX)
set(CMAKE_CXX_STANDARD 14)
add_executable(fibonacci fibonacci.cpp)
target_compile_options(fibonacci PRIVATE -Wall -Wextra)
```

```
$ cd fibonacci/build
$ cmake ..
CMake Error at CMakeLists.txt:5 (add_executable):
  Cannot find source file: ...
```

TDD Demonstration: 1st iteration (C++)

- Test to pass: cmake && make
- Create fibonacci.cpp

fibonacci/fibonacci.cpp

```
#include <iostream>

int main() {
  std::cout << ”hello, world” << std::endl;
}
```

```
$ cd fibonacci/build
$ cmake ..
-- Build files have been written to: .../fibonacci/build
$ make
$ ./fibonacci
hello, world
```

TDD Demonstration: 1st iter. (Python)

- Test to pass: cmake && make
- Create CMakeLists.txt

fibonacci/CMakeLists.txt

```
cmake_minimum_required(VERSION 3.14)
project(fibonacci NONE)
find_package(Python)

configure_file(fibonacci fibonacci FILE_PERMISSIONS OWNER_READ
OWNER_WRITE OWNER_EXECUTE GROUP_READ GROUP_EXECUTE
WORLD_READ WORLD_EXECUTE)
```

```
$ cd fibonacci/build
$ cmake ..
CMake Error at CMakeLists.txt:5 (configure_file):
  configure_file Problem configuring file
```

TDD Demonstration: 1st iter. (Python)

- Test to pass: cmake && make
- Create fibonacci

fibonacci/fibonacci

```
#!@Python_EXECUTABLE@

print(”hello, world”)
```

```
$ cd fibonacci/build
$ cmake ..
-- Build files have been written to: .../fibonacci/build
$ make
$ ./fibonacci
hello, world
```

Some Good Practices for C++/Python

- C++
  - Specify options to make the compiler issue maximum warning messages / コンパイラが最大限の警告メッセージを出すようにオプションを指定する
    - for GCC: ”-Wall -Wextra”
  - Fix code until all warnings go away / 警告がなくなるまで修正する
- Python
  - Do not use version 2, use version 3 / version 2ではなく、version 3を使う
  - Install additional modules in the user area, not the system area / 追加モジュールはシステム領域ではなくユーザ領域にインストール
    - create dedicated environment / 専用の環境を作る: venv, conda
    - for venv
      - python3 -m venv venv
      - source venv/bin/activate
      - pip3 install pytest
- Common
  - Always follow a ”coding standard” / ”コーディング規格”に常に従う

TDD Demonstration: 2nd iter. (C++)

- Test to pass: cmake && make && ctest
- Modify CMakeLists.txt

fibonacci/CMakeLists.txt

```
...
include(FetchContent)
FetchContent_Declare(catch2
  GIT_REPOSITORY https://github.com/catchorg/Catch2.git
  GIT_TAG v3.1.0
)
FetchContent_MakeAvailable(Catch2)
enable_testing()

add_executable(fibonacci fibonacci.cpp)
target_compile_options(fibonacci PRIVATE -Wall -Wextra)

add_executable(test_fibonacci test_fibonacci.cpp)
target_compile_options(test_fibonacci PRIVATE -Wall -Wextra)
target_link_libraries(test_fibonacci Catch2::Catch2WithMain)
add_test(test_fibonacci test_fibonacci)
```

TDD Demonstration: 2nd iter. (C++)

- Test to pass: cmake && make && ctest
- Create test_fibonacci.cpp

fibonacci/test_fibonacci.cpp

```
#include <catch2/catch_test_macros.hpp>

TEST_CASE("fibonacci", "test") {
  REQUIRE(2 * 2 == 4);
}
```

```
$ cd fibonacci/build
$ cmake ..
-- Build files have been written to: .../fibonacci/build
$ make
...
$ ctest
Test project .../fibonacci/build
    Start 1: test_fibonacci
1/1 Test #1: test_fibonacci ...................   Passed    0.02 sec
100% tests passed, 0 tests failed out of 1
```

TDD Demonstration: 2nd iter. (Python)

- Test to pass: cmake && make && ctest
- Modify CMakeLists.txt

fibonacci/CMakeLists.txt

```
...
execute_process(COMMAND ${Python_EXECUTABLE} -m venv
  ${CMAKE_BINARY_DIR}/venv)
set(ENV{VIRTUAL_ENV} ${CMAKE_BINARY_DIR}/venv)
set(Python_FIND_VIRTUALENV FIRST)
unset(Python_EXECUTABLE)
find_package(Python)
execute_process(COMMAND ${Python_EXECUTABLE} -m pip install -r
  ${CMAKE_SOURCE_DIR}/requirements.txt)
enable_testing()

configure_file(fibonacci fibonacci FILE_PERMISSIONS OWNER_READ
OWNER_WRITE OWNER_EXECUTE GROUP_READ GROUP_EXECUTE
WORLD_READ WORLD_EXECUTE)

add_test(test_fibonacci ${Python_EXECUTABLE} -m pytest
  ${CMAKE_CURRENT_SOURCE_DIR}/test_fibonacci.py)
```

TDD Demonstration: 2nd iter. (Python)

- Test to pass: cmake && make && ctest
- Create requirements.txt & test_fibonacci.py

fibonacci/requirements.txt

```
pytest
```

fibonacci/test_fibonacci.py

```
def test_fibonacci():
    assert 2 * 2 == 4
```

```
$ cd fibonacci/build
$ cmake ..
-- Build files have been written to: .../fibonacci/build
$ make
...
$ ctest
Test project .../fibonacci/build
    Start 1: test_fibonacci
1/1 Test #1: test_fibonacci ..................   Passed    0.02 sec
100% tests passed, 0 tests failed out of 1
```

TDD Demonstration: 3rd iter. (C++)

- Test to pass: fibonacci(0) == 0
- Modify test_fibonacci.cpp and create fibonacci.hpp

fibonacci/test_fibonacci.cpp

```
#include <catch2/catch_test_macros.hpp>
#include ”fibonacci.hpp”

TEST_CASE(”fibonacci”, ”test”) {
  REQUIRE(fibonacci(0) == 0);
}
```

fibonacci/fibonacci.hpp

```
#pragma once

int fibonacci(int) {
  return 0;
}
```

TDD Demonstration: 3rd iter. (Python)

- Test to pass: fibonacci(0) == 0
- Modify test_fibonacci.py and create fibonacci.py

fibonacci/test_fibonacci.py

```
from fibonacci import fibonacci

def test_fibonacci():
    assert fibonacci(0) == 0
```

fibonacci/fibonacci.py

```
def fibonacci(n):
    return 0
```

TDD Demonstration: 4th iter. (C++)

- Test to pass: fibonacci(1) == 1
- Modify test_fibonacci.cpp and fibonacci.hpp

fibonacci/test_fibonacci.cpp

```
...
  REQUIRE(fibonacci(0) == 0);
  REQUIRE(fibonacci(1) == 1);
...
```

fibonacci/fibonacci.hpp

```
...
int fibonacci(int) {
  if (n == 0)
    return 0;
  return 1;
}
```

TDD Demonstration: 4th iter. (Python)

- Test to pass: fibonacci(1) == 1
- Modify test_fibonacci.py and fibonacci.py

fibonacci/test_fibonacci.py

```
...
def test_fibonacci():
    assert fibonacci(0) == 0
    assert fibonacci(1) == 1
```

fibonacci/fibonacci.py

```
def fibonacci(n):
    if n == 0:
        return 0
    return 1
```

TDD Demonstration: 5th iter. (C++)

- Test to pass: fibonacci(2) == 1 && fibonacci(3) == 2
- Modify test_fibonacci.cpp and fibonacci.hpp

fibonacci/test_fibonacci.cpp

```
...
TEST_CASE("fibonacci", "test") {
  std::vector<std::tuple<int, int>> cases {{0, 0}, {1, 1}, {2, 1}, {3, 2}};
  for (auto c : cases) {
    REQUIRE(fibonacci(std::get<0>(c)) == std::get<1>(c));
  }
}
```

fibonacci/fibonacci.hpp

```
...
int fibonacci(int) {
  if (n == 0)
    return 0;
  if (n <= 2)
    return 1;
  return 2;
}
```

TDD Demonstration: 5th iter. (Python)

- Test to pass: fibonacci(2) == 1 && fibonacci(3) == 2
- Modify test_fibonacci.py and fibonacci.py

fibonacci/test_fibonacci.py

```
...
def test_fibonacci():
    cases = [[0, 0], [1, 1], [2, 1], [3, 2]]
    for n, v in cases:
        assert fibonacci(n) == v
```

fibonacci/fibonacci.py

```
def fibonacci(n):
    if n == 0:
        return 0
    if n <= 2:
        return 1
    return 2
```

TDD Demonstration: 6th iter. (C++)

- Test to pass: fibonacci(10) == 55
- Modify test_fibonacci.cpp and fibonacci.hpp

fibonacci/test_fibonacci.cpp

```
...
  std::vector<std::tuple<int, int>> cases {{0, 0}, {1, 1}, {2, 1}, {3, 2},
     {10, 55}};
...
```

fibonacci/fibonacci.hpp

```
...
int fibonacci(int) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

TDD Demonstration: 6th iter. (Python)

- Test to pass: fibonacci(10) == 55
- Modify test_fibonacci.py and fibonacci.py

fibonacci/test_fibonacci.py

```
...
def test_fibonacci():
    cases = [[0, 0], [1, 1], [2, 1], [3, 2], [10, 55]]
    for n, v in cases:
        assert fibonacci(n) == v
```

fibonacci/fibonacci.hpp

```
def fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

TDD Demonstration: 7th iter. (C++)

- Test to pass: fibonacci(-1) == 1 && fibonacci(-2) == -1
- Modify test_fibonacci.cpp and fibonacci.hpp

fibonacci/test_fibonacci.cpp

```
...
  std::vector<std::tuple<int, int>> cases {{-2, -1}, {-1, 1}, {0, 0}, {1, 1}, {2, 1},
     {3, 2}, {10, 55}};
...
```

fibonacci/fibonacci.hpp

```
...
int fibonacci(int) {
  if (n < 0)
    return fibonacci(n+2) - fibonacci(n+1);
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  return fibonacci(n-1) + fibonacci(n-2);
}
```

TDD Demonstration: 7th iter. (Python)

- Test to pass: fibonacci(-1) == 1 && fibonacci(-2) == -1
- Modify test_fibonacci.py and fibonacci.py

fibonacci/test_fibonacci.py

```
...
    cases = [[-2, -1], [-1, 1], [0, 0], [1, 1], [2, 1], [3, 2], [10, 55]]
...
```

fibonacci/fibonacci.py

```
def fibonacci(n):
    if n < 0:
        return fibonacci(n+2) - fibonacci(n+1)
    if n == 0:
        return 0
    if n == 1:
        return 1
    return fibonacci(n-1) + fibonacci(n-2)
```

TDD Demonstration: 8th iter. (C++)

- Refactoring for speed up for large N (~45)
- Modify fibonacci.hpp

fibonacci/fibonacci.hpp

```
#include <tuple>
int fibonacci(int) {
  int v0 = 0;
  int v1 = 1;
  if (n < 0) {
    for (int i = -1; i >= n; --i)
      std::tie(v0, v1) = std::make_tuple(v1 - v0, v0);
    return v0;
  }
  if (n == 0)
    return v0;
  if (n == 1)
    return v1;
  for (int i = 2; i <= n; ++i)
    std::tie(v1, v0) = std::make_tuple(v0 + v1, v1);
  return v1;
}
```

TDD Demonstration: 8th iter. (Python)

- Refactoring for speed up for large N (~45)
- Modify fibonacci.py

fibonacci/fibonacci.py

```
def fibonacci(n):
    v0 = 0
    v1 = 1
    if n < 0:
        for i in range(-1, n-1, -1):
            v0, v1 = (v1 - v0, v0)
        return v0
    if n == 0:
        return v0
    if n == 1:
        return v1
    for i in range(2, n+1):
        v1, v0 = (v0 + v1, v1)
    return v1
```

TDD Demonstration: 9th iter. (C++)

- Test to pass: fibonacci(100) == 354224848179261915075
- Modify CMakeLists.txt to use Boost.multiprecision library

fibonacci/CMakeLists.txt

```
...
find_package(Boost REQUIRED)

include(FetchContent)
...
enable_testing()

add_executable(fibonacci fibonacci.cpp)
target_compile_options(fibonacci PRIVATE -Wall -Wextra)
target_link_libraries(fibonacci PRIVATE Boost::boost)

add_executable(test_fibonacci test_fibonacci.cpp)
target_compile_options(test_fibonacci PRIVATE -Wall -Wextra)
target_link_libraries(test_fibonacci PRIVATE Boost::boost
  Catch2::Catch2WithMain)
add_test(test_fibonacci test_fibonacci)
```

TDD Demonstration: 9th iter. (C++)

- Test to pass: fibonacci(100) == 354224848179261915075
- Modify test_fibonacci.cpp

fibonacci/test_fibonacci.cpp

```
...
  std::vector<std::tuple<int, mp::cpp_int>> cases {{-2, -1}, {-1, 1}, {0, 0},
    {1, 1}, {2, 1}, {3, 2}, {10, 55}, {40, 102334155},
    {100, mp::cpp_int("354224848179261915075")}};
  for (auto c : cases) {
    REQUIRE(fibonacci(std::get<0>(c)) == std::get<1>(c));
    if (std::get<0>(c) > 0)
      REQUIRE(fibonacci(std::get<0>(c)) > 0);
  }
...
```

TDD Demonstration: 9th iter. (C++)

- Test to pass: fibonacci(100) == 354224848179261915075
- Modify fibonacci.hpp

fibonacci/fibonacci.hpp

```
#pragma once

#include <tuple>
#include <boost/multiprecision/cpp_int.hpp>
namespace mp = boost::multiprecision;

mp::cpp_int fibonacci(int n) {
  mp::cpp_int v0 = 0;
  mp::cpp_int v1 = 1;
  if (n < 0) {
...
```

TDD Demonstration: 9th iter. (Python)

- Test to pass: fibonacci(100) == 354224848179261915075
- Modify test_fibonacci.py and it just works as a charm

fibonacci/test_fibonacci.py

```
...
    cases = [[-2, -1], [-1, 1], [0, 0], [1, 1], [2, 1], [3, 2], [10, 55], [40,
102334155], [100, 354224848179261915075]]
...
```

TDD Demonstration: 10th iter. (C++)

- Test to pass: lucas(0) == 2 && lucas(1) == 1 && ...
- Create test_lucas.cpp and lucas.hpp, and modify CMakeLists.txt

fibonacci/CMakeLists.txt

```
...
foreach(name fibonacci lucas)
  add_executable(${name} ${name}.cpp)
  target_compile_options(${name} PRIVATE -Wall -Wextra)
  target_link_libraries(${name} PRIVATE Boost::boost)

  add_executable(test_${name} test_${name}.cpp)
  target_compile_options(test_${name} PRIVATE -Wall -Wextra)
  target_link_libraries(test_${name} PRIVATE Boost::boost
Catch2::Catch2WithMain)
  add_test(test_${name} test_${name})
endforeach(name)
```

TDD Demonstration: 10th iter. (C++)

- Test to pass: lucas(0) == 2 && lucas(1) == 1 && ...
- Refactor fibonacci.hpp and lucas.hpp by creating recursion.hpp

fibonacci/fibonacci.hpp

```
...
#include ”recursion.hpp”

mp::cpp_int fibonacci(int n) {
  return recursion(0, 1, n);
}
```

fibonacci/recursion.hpp

```
...
mp::cpp_int recursion(mp::cpp_int v0, mp::cpp_int v1, int n) {
...
```

TDD Demonstration: 10th iter. (Python)

- Test to pass: lucas(0) == 2 && lucas(1) == 1 && ...
- Create test_lucas.py and lucas.py, and modify CMakeLists.txt

fibonacci/CMakeLists.txt

```
...
foreach(name fibonacci lucas)
  configure_file(${name} ${name} FILE_PERMISSIONS OWNER_READ
    OWNER_WRITE OWNER_EXECUTE GROUP_READ GROUP_EXECUTE
    WORLD_READ WORLD_EXECUTE)

  add_test(test_${name} ${Python_EXECUTABLE} -m pytest
    ${CMAKE_CURRENT_SOURCE_DIR}/test_${name}.py)
endforeach(name)
```

TDD Demonstration: 10th iter. (Python)

- Test to pass: lucas(0) == 2 && lucas(1) == 1 && ...
- Refactor fibonacci.py and lucas.py by creating recursion.py

fibonacci/fibonacci.py

```
from recursive import recursive

def fibonacci(n):
    return recursive(0, 1, n)
```

fibonacci/recursion.py

```
...
def recursion(v0, v1, n):
...
```

Good Practices in TDD

- Keep iterations as short as possible / 各イテレーションをできるだけ小さくする
  - 10 min to several hours / 10分～数時間
- Always be **Green** at the end of the day / 1日の終わりには必ず**Green**に
- Review the test if the iteration seems to take too long / 時間がかかりすぎるようであればテストを見直す
  - Move the current test to the To-Do list and consider a smaller test / 今のテストはTo-Doリストに移し、もう少し小さなテストを考える
- Focus on making the **Red** in front of you **Green**, even if you want to fix something else while working / 作業中に別のところを修正したくなっても、目の前の**Red**を**Green**にすることに集中する
  - Write down the potential improvement on your To-Do list and start writing a test for it later / 修正候補はTo-Doリストに書いておいて、あとでそのテストを書くところから始める

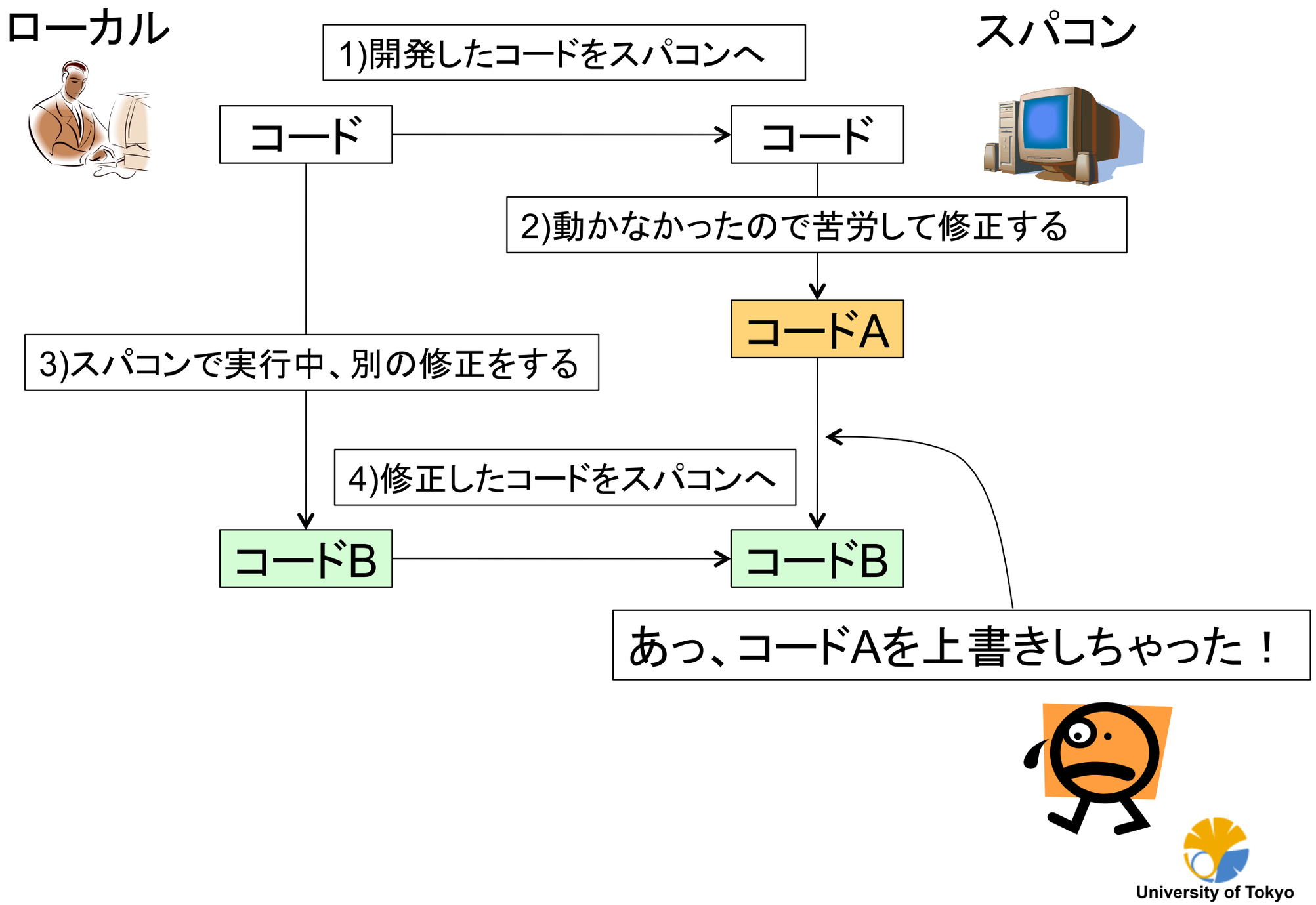
Part II: Version Control / バージョン管理

What's VCS？ / バージョン管理システムとは？

Traditional ”File Management”

- Traditional ”File Management” / 従来の「ファイル管理」
  - Based on file/directory name / ファイル名・ディレクトリ名による管理
    - date, person name, version number, etc. / 日付、人名、バージョン番号など
    - e.g.) main_v1.0.cpp, main_v1.0_todo-3.cpp, main_v1.0.2_synge.cpp
  - Recorded by handwritten log file / 手書きのログファイルによる記録
- Problems / 問題点
  - Forget to keep records, record mistakes, incomplete records / 記録を付け忘れる、記録を間違う、不完全な記録
- Different people have different naming conventions / 人により命名規則がばらばら
- Copying between computers makes it difficult to know which one was modified and which one is new / コンピュータ間でコピーを繰り返すと、どれを修正したか、どれが新しいか分からなくなる
- Another person would make independent modifications based on the same version / 同じバージョンを元に、別の人が独立に修正を行ってしまう
  - version branching / バージョンの分岐

A Common Example

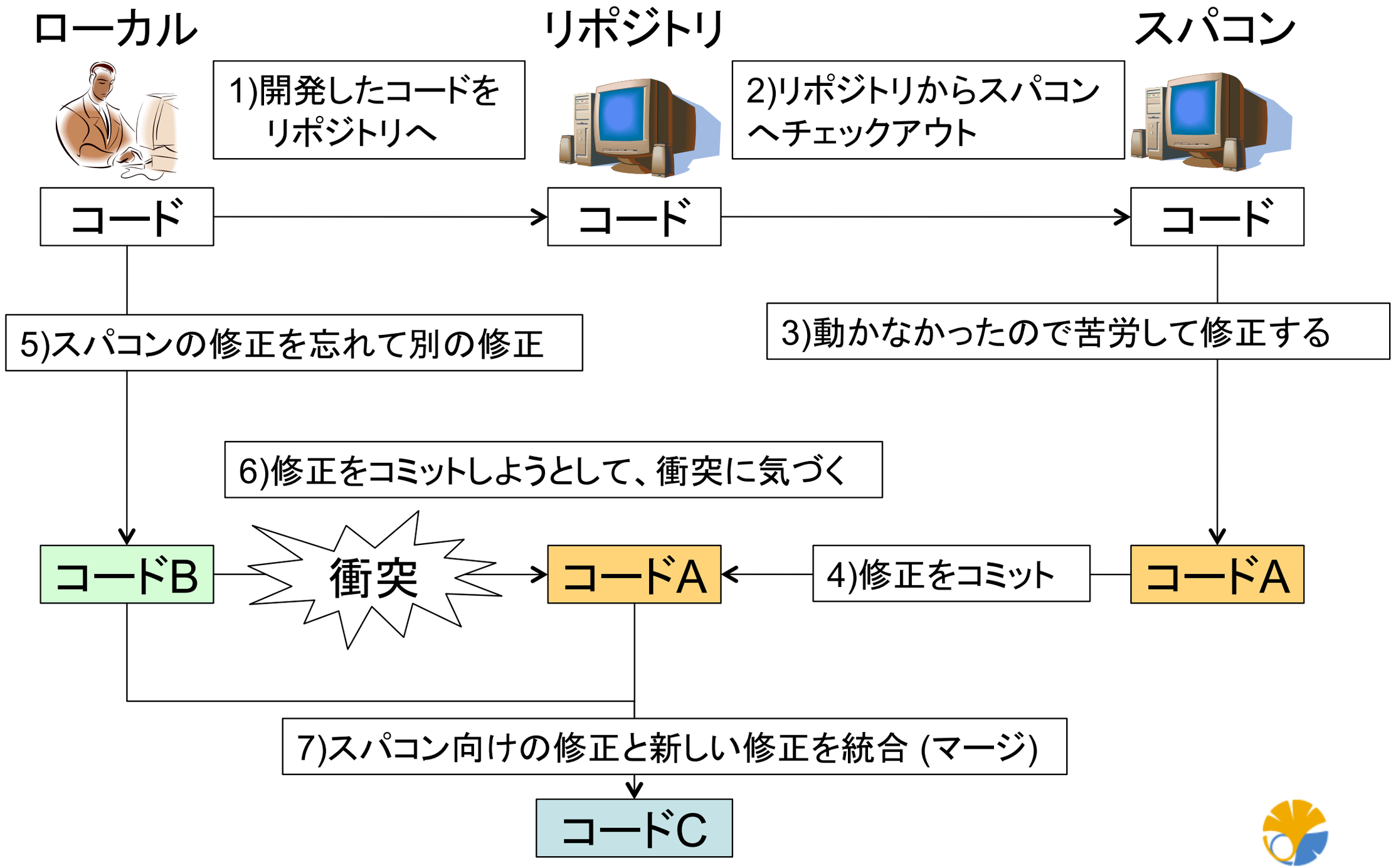


University of Tokyo

What’s Version Control System（VCS）?

- System for managing all file histories in a database（repository）/ ファイルの履歴をデータベース(リポジトリ)で一括管理するシステム
  - Save all revision history（diffs）/ 全ての修正履歴(差分)を保存
  - Unique version number（commit ID）for each update / 更新毎に一意なバージョン番号(コミットID)を付与
  - Can make comparison between any versions / 任意のバージョン間の比較が可能
- Support for working in team and distributed environments / チーム・分散環境での作業をサポート
  - Share repository over Internet / インターネット経由でリポジトリを共有
  - Detect and resolve collisions when updated by multiple people/ multiple locations simultaneously / 複数人で/複数場所から同時に更新した場合に衝突を検知・解決
  - Branch, merge, and tag management / ブランチ・マージ・タグの管理

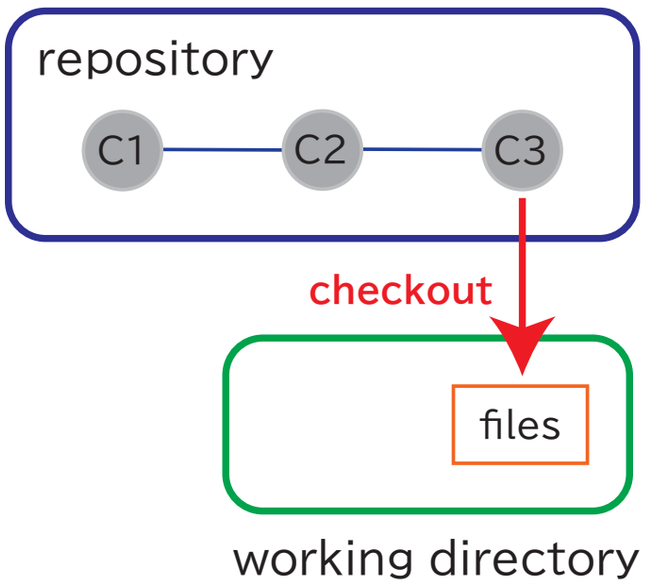
With Version Control System

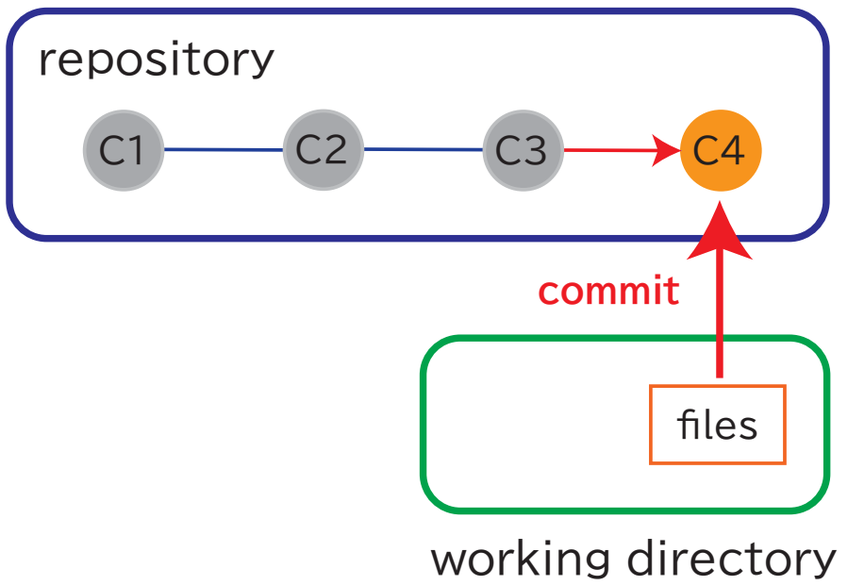


University of Tokyo

What's Repository

- Repository / リポジトリ
  - Database that manages all history (diffs) of files / ファイルの全ての履歴(差分)を管理するデータベース
- Interaction with repository / リポジトリとのやりとり
  - checkout / チェックアウト
    - Retrieve the latest (or a specified) version of files / 最新版(あるいは指定したバージョン)のファイルを取り出す
  - commit / コミット
    - Register file changes / ファイルの変更をリポジトリに登録する
    - Go one version further / バージョンが一つ進む





Disadvantages of VCS

- Disadvantages and troublesome points of VCS / バージョン管理システムの欠点・面倒な点
  - Must be updated to the latest state before modification / 修正前に最新の状態にアップデートしなければならない
    ⇒ Becomes a habit / 習慣になります
  - All changes must be ”committed” / 全ての修正点を「コミット」しなければならない
    ⇒ Becomes a habit / 習慣になります
  - Conflicts must be dealt with when they occur / コンフリクトへの対応が面倒
    ⇒ Horrible if fixed without realizing it / 気づかずに修正してしまうほうが怖い
  - Server setup is cumbersome / サーバのセットアップが面倒
    ⇒ GitHub, GitLab, etc are available / GitHub, GitLabなどが使えます
-

Advantages of VCS

- Advantages of Using VCS / バージョン管理システムを使う利点
  - All records are kept automatically / 全ての記録が自動的に残る
  - No need to be afraid of bold changes / 大胆な変更を恐れる必要がなくなる
  - No need to comment out and save old code / 古いコードをコメントアウトしてとっておく必要なし
  - No more stress of maintaining file history / ファイルの履歴管理というストレスから開放
- Using VCS more than doubles your work efficiency / バージョン管理システムを使うと作業効率が倍以上になる
  ⇒ If you don’t, you lose half your life / 使わないと人生を半分損する

diff & patch

- diff command
  - Output difference between two text files / 2つのテキストファイルの差分を出力
  - More compact than storing entire file / ファイル全体を保存するよりコンパクト
  - Easy to see the changes / 変更点を確認しやすい

```
$ diff -u file1.txt file2.txt > file.diff
```

- patch command
  - Apply the diff generated by the diff command / diff コマンドが生成した差分をファイルに適用
  - Generate modified file from original file and diff / もとのファイルと差分から変更後のファイルを生成

```
$ patch < file.diff
```

diff & patch: Single File Example

```
$ cp /somewhere/shakespeare/prologue.txt prologue.txt
$ cp prologue.txt prologue-orig.txt

# edit prologue.txt
$ vim prologue.txt

# take a diff and look at it contents
$ diff -u prologue-orig.txt prologue.txt > prologue.diff
$ less prologue.diff

# apply patch
$ cp /somewhere/shakespeare/prologue.txt prologue.txt
$ patch < prologue.diff
$ less prologue.txt
```

diff & patch: Directory Example

```
$ cp -r /somewhere/shakespeare shakespeare
$ cp -r shakespeare shakespeare.orig

# edit contents of directory
$ (vim, cp, mv, etc)

# take a diff and look at its contents
$ diff -urN shakespeare.orig shakespeare > shakespeare.diff
$ less shakespeare.diff

# apply patch
$ rm -rf shakespeare
$ cp -r /somewhere/shakespeare shakespeare
$ patch -p0 < shakespeare.diff
```

Git Demonstration: initialize repository

- First setup / 初回の設定 (Just do it once for the first time on each PC/WS)
  - Setup of username etc (used for logs, etc / ログなどに使用される)

```
$ git config --global user.name ’Synge Todo’
$ git config --global user.email wistaria@phys.s.u-tokyo.ac.jp
$ git config --global init.defaultBranch main
```

- Create working directory and git repository / 作業用ディレクトリと git リポジトリの作成

```
$ mkdir fibonacci
$ cd fibonacci
$ git init
Initialized empty Git repository in .../fibonacci/.git
$ ls -a
. .. .git
```

- All history information is stored in .git. Never delete .git! / 全ての履歴情報は .git に保存される。けっして .git を消さないように!

Git Demonstration: first commit

・Create files and perform test / ファイルの作成とテスト

```
$ vim CMakeLists.txt
$ vim fibonacci.cpp
# cmake & make
```

・Add files as targets to be committed / ファイルをcommitの対象として追加

```
$ git add CMakeLists.txt fibonacci.cpp
$ git status
...
```

・Commit to repository / リポジトリにコミット

  ・Use ”-m” option for commit log / コミットログは ”-m” で指定 (Write ”Why”)

```
$ git commit -m ’cmake and make to work’
```

・Check log / ログの確認

```
$ git log
```

Git Demonstration: 2nd iteration

・Edit and add files / ファイルの編集・追加

```
$ vim CMakeLists.txt
$ vim test_fibonacc.cpp
# cmake & make & ctest
```

・Check status and diff / 状態と差分を見てみる

```
$ git status
...
$ git diff
...
```

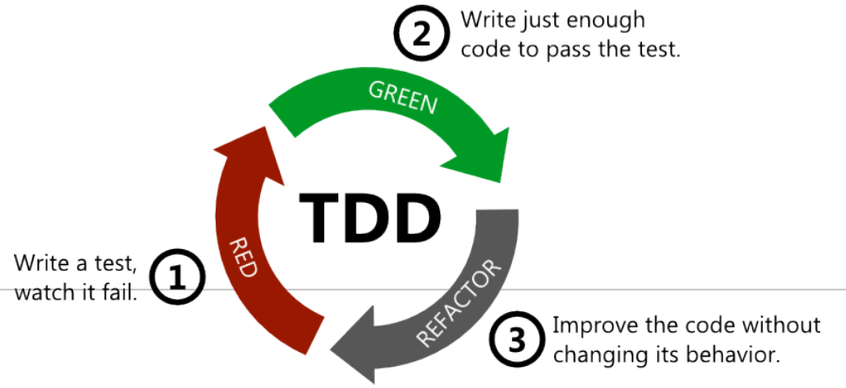
・Commit to repository / リポジトリにコミット

```
$ git add CMakeLists.txt test_fibonacci.cpp
$ git commit -m ’first test to work’
```

・Check log / ログの確認

```
$ git log
```

TDD Workflow



- Red
  - Write a failing automated test before you write any code / コードを書く前に失敗する自動テストコードを必ず書く
    - git fetch → git merge → git checkout
    - vim → cmake → make → ctest
- Green
  - Write just enough code to pass the test / テストをパスする最低限のコードを書く
    - [vim → cmake → make → ctest ...] → git add → git commit ...
    - git checkout → git merge → git push
- Refactor
  - Remove duplication / 重複を除去する
    - git checkout
    - [[vim → cmake → make → ctest ...] → git add → git commit ...]
    - git checkout → git merge → git push

”.gitignore”

- Only source code and CMakeLists.txt etc should be managed under Git / GitではソースコードやCMakeLists.txtのみを管理
  - Files generated by cmake and make under build should not be managed / build以下のcmakeやmakeで生成されたファイルは管理しない
- List unmanaged files in .gitignore / .gitignoreに管理しないファイルのリスト

```
$ git status
```

fibonacci/.gitignore

```
build
```

```
$ git status
```

- Manage .gitignore itself by Git / .gitignore自体はGitで管理する

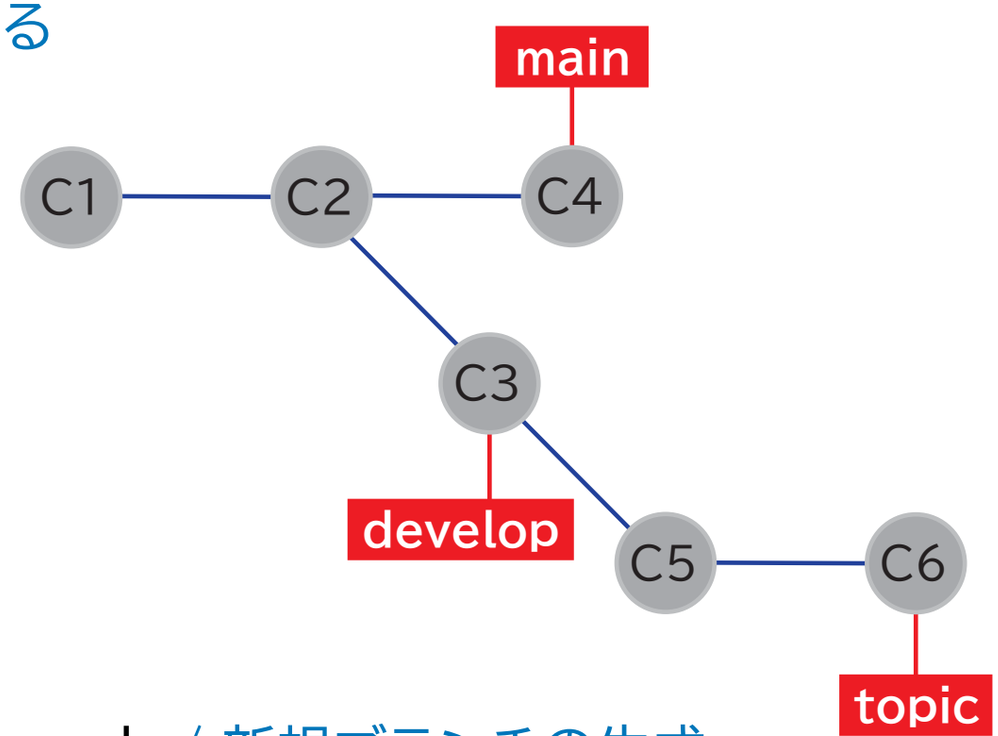
```
$ git add .gitignore
$ git commit -m ’ignore build’
$ git status
```

Part II: Version Control / バージョン管理

Branch & Merge / ブランチとマージ

What’s Branch?

- Branch / ブランチ
  - Working in a separate stream from the main / メインとは別の流れで作業を続ける機能
  - Can work without affecting other branches / 他のブランチには影響を与えずに作業ができる



- Creating new branch / 新規ブランチの生成
  - git branch ...
- Switching between branches / ブランチの切り替え
  - git checkout ...

Git Demonstration: branch

・Assume done up to 6th iteration / 6th iteration まで終わっていると仮定

・Create a topic branch (negative-n) for 7th iteration and work there / 7th iterationのためのトピックブランチ(negative-n)を作成してそこで作業

```
$ git status
$ git branch negative-n
$ git checkout negative-n
$ git status
# [vim → cmake → make → ctest]
$ git add fibonacci.hpp test_fibonacci.cpp
$ git commit -m ’for supporting negative n’
```

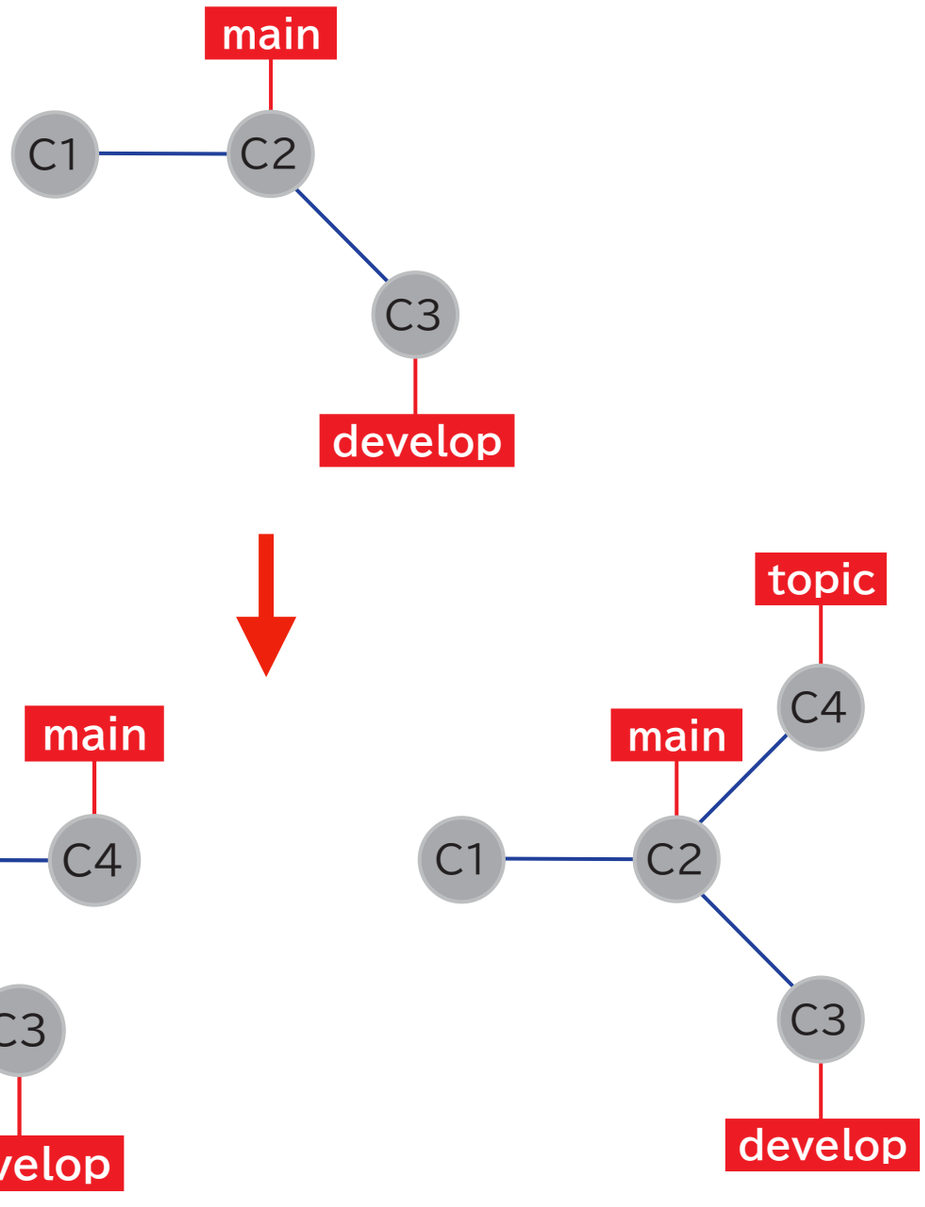
・Go back to the main branch / mainブランチに戻ってみる

```
$ git checkout main
$ less fibonacci.hpp
```

・Can go back and forth between main and negative-n with git checkout / git checkout で main と negative-n の間を行ったり来たりできる

Multiple Branches

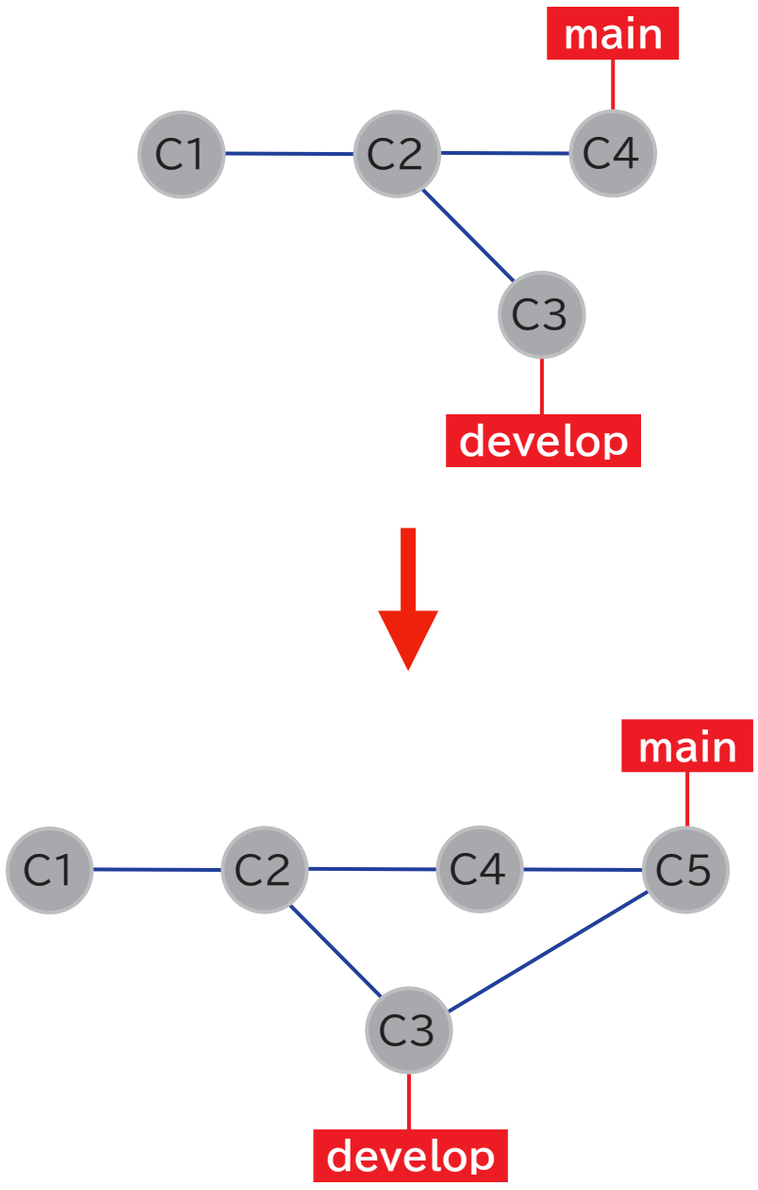
- Go back to main (or create a new branch) and commit there, then the world line branches off / mainに戻って(あるいは新しいbranchを作って)そこでcommitすると世界線が枝分かれする



What’s Merge

・Merging branch / ブランチのマージ

・Merge divergent world lines / 分岐した世界線を統合する



Git Demonstration: merge

・Once work on negative-n branch is completed, merge it into main branch / negative-nブランチの作業が完了したので、mainブランチにmergeする

```
$ git status
$ git checkout main
$ git status
$ git log --graph
$ git merge negative-n
$ git log --graph
```

・Remove negative-n branch as it is no longer needed / negative-nブランチはもう不要なので削除

```
$ git branch -D negative-n
$ git log --graph
```

・The divergent world lines have come back together / 分岐していた世界線が一つに戻った

Git Demonstration: branch & merge

- Try to do 8th iteration and 9th iteration in parallel / 8th iteration と 9th iterationを並行してやってみる
  - Create a branch (linear) for 8th iteration and work there / 8th iterationのためのブランチ(linear)を作成してそこで作業

```
$ git status
$ git branch linear
$ git checkout linear
# [vim → cmake → make → ctest]
$ git add fibonacci.hpp test_fibonacci.cpp
$ git commit -m 'for supporting negative n'
```

- Return back to main branch and create a branch (multiprec) for 9th iteration and work there / mainブランチに戻り、9th iterationのためのブランチ(multiprec)を作成して作業

```
$ git checkout main
$ git branch multiprec
$ git checkout multiprec
# [vim → cmake → make → ctest]
$ git add CMakeLists.txt fibonacci.hpp test_fibonacci.cpp
$ git commit -m 'for supporting arbitrary large n'
```

Git Demonstration: conflict

- Merge linear branch into main and then try merging multiprec branch / linearブランチをmainにマージしてから、multiprecブランチをマージしてみる

```
$ git checkout main
$ git merge linear
$ git merge multiprec
Auto-merging fibonacci.hpp
CONFLICT (content): Merge conflict in fibonacci.hpp
Auto-merging test_fibonacci.cpp
CONFLICT (content): Merge conflict in test_fibonacci.cpp
Automatic merge failed; fix conflicts and then commit the result.
```

- Git has decided it's not possible to integrate automatically / Gitは自動的に統合するのは無理と判断した

```
$ git status
On branch main
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)
...
```

Git Demonstration: resolving conflict

・Conflicts must be adjusted by hand / 衝突は手で調整しなければならない

fibonacci/fibonacci.hpp

```
#include <tuple>
<<<<<<< HEAD

int fibonacci(int n) {
  int v0 = 0;
  int v1 = 1;
=======
#include <boost/multiprecision/cpp_int.hpp>

namespace mp = boost::multiprecision;

mp::cpp_int fibonacci(int n) {
  mp::cpp_int v0 = 0;
  mp::cpp_int v1 = 1;
>>>>>>> multiprec
  if (n < 0) {
```

Git Demonstration: resolving conflict

- fibonacci.hpp and test_fibonacci.cpp have been modified / fibonacci.hpp と test_fibonacci.cppの修正が完了した

```
# [cmake → make → ctest]
$ git add fibonacci.hpp test_fibonacci.cpp
$ git commit -m 'merge branch multiprec'
$ git status
$ git log --graph
```

- The merge is successfully completed and the world lines are united / 無事mergeが完了して世界線が一つに

```
$ git branch -d linear multiprec
Deleted branch linear (was 38e6ae4).
Deleted branch multiprec (was dda8e49).
$ git status
$ git log --graph
```

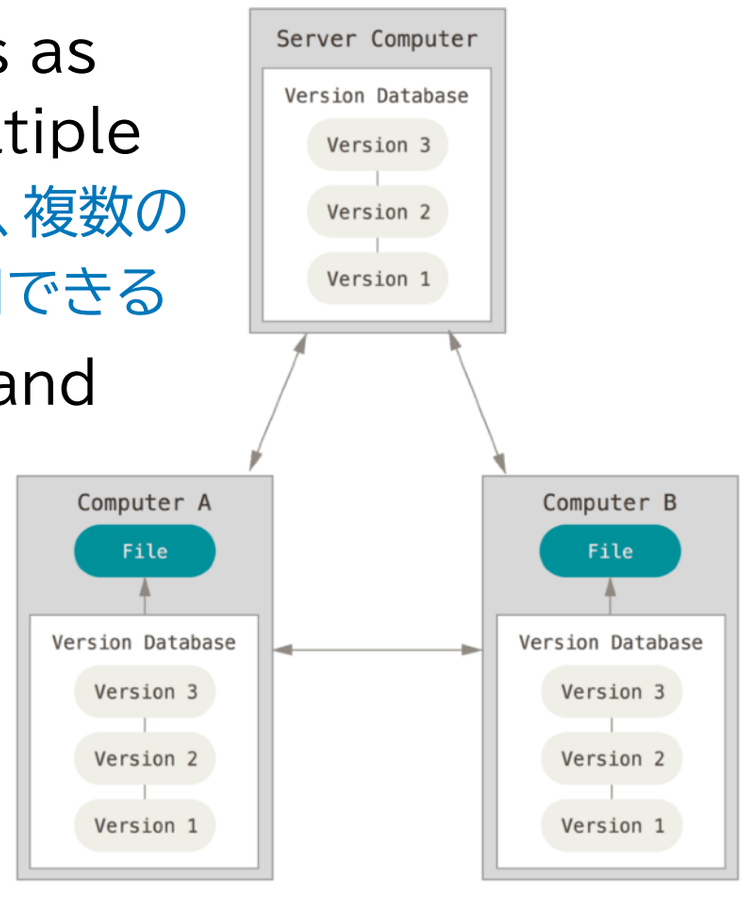
- Never merge into the main branch if there are still tests that do not pass / 通らないテストが残っている場合には絶対にmainブランチにマージしない

Part II: Version Control / バージョン管理

Using GitHub / GitHubの利用

Git & GitHub

- Git allows multiple repositories to be used in sync with each other / Gitでは複数のリポジトリを互いに同期しながら使うことができる
- GitHub can be used as one of remote repositories / GitHubはリモートリポジトリのひとつとして使える
  - Copy (backup) can be made by pushing from local repository / ローカルリポジトリからpushすることでコピー(バックアップ)を取ることができる
  - As we can access from anywhere, GitHub works as a hub for synchronizing repositories across multiple computers / どこからでもアクセスすることができるので、複数のコンピュータでリポジトリーを同期するためのハブとして利用できる
- Convenient for source code sharing by a team and source code publication / チームでのソースコード共有・ソースコード公開にも便利
- Use as a CI (Continuous Integration) tool / CI (継続的インテグレーション)ツールとしての利用



https://git-scm.com/book/

Using GitHub

- GitHub Account Registration / GitHubアカウント登録 (if you do not yet have registered / まだアカウントを持っていない場合)
  - https://github.com → Sign up for GitHub → Create an account

- Register SSH public key or obtain HTTPS access token / SSH公開鍵の登録あるいはHTTPSアクセストークンの取得
  - To write to GitHub repository with git command, either must be set up in advance / gitコマンドでGitHubリポジトリに書き込むためには、あらかじめどちらかの設定が必要
- SSH public key / SSH公開鍵の登録
  - Setting → SSH and GPG keys → New SSH Key
- HTTPS access token / HTTPSアクセストークンの取得
  - Setting → Developer settings → Personal access tokens → check ”repo” and ”workflow” → Generate new token

GitHub Demonstration: initialize repository

- Initialization of repository / リポジトリの初期化
  - Instead of doing a local git init, it is easier to create a repository on GitHub and copy it / localでgit initする代わりに、GitHubでリポジトリを作りそれをコピーするほうが楽
  - First create a repository on GitHub / 最初にGitHub上でリポジトリを作成
    - determine public or private / publicにするかprivateにするか決める
    - name of project / プロジェクトの名前 ← ultra-important / 超重要
- Clone the GitHub repository locally and start development / GitHubのリポジトリをローカルにcloneしてから開発開始
  - Copy repository URL / リポジトリのURLをコピー (https://... or git@github.com...)
  - git clone

```
$ git clone https://github.com/wistaria/fibonacci.git
```

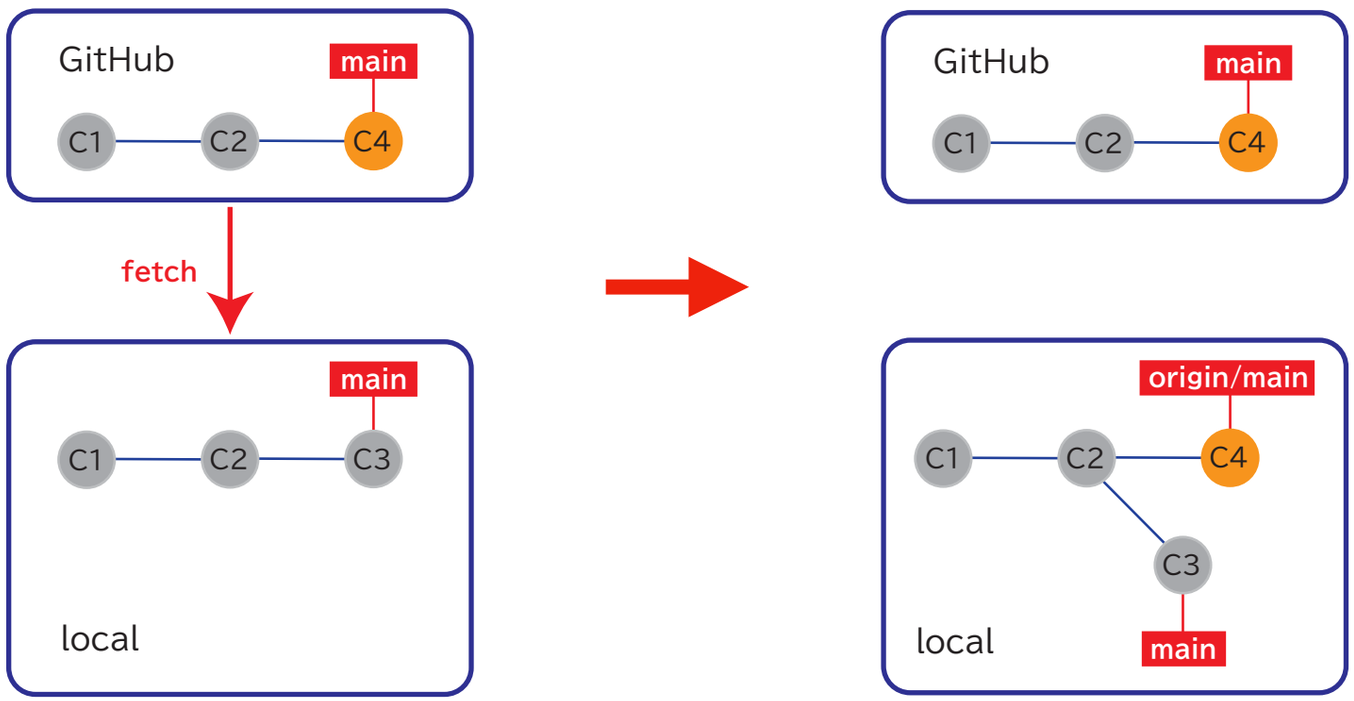
  - [vim → cmake → make → ctest ...] → git add → git commit ...

GitHub Demonstration: sync repositories

- GitHub → local repository / GitHub → ローカルのリポジトリ

```
$ git fetch
```

  - Entire repository on GitHub is copied / リポジトリ全体がコピーされる
  - Branch on GitHub (e.g. main) is locally named origin/main / GitHub上のブランチ(例: main)はローカルではorigin/mainという名前がつけられる
  - Branch diverges if repository is modified from elsewhere/by others / 他の場所から/他の人がリポジトリを変更している場合にはブランチが分岐する

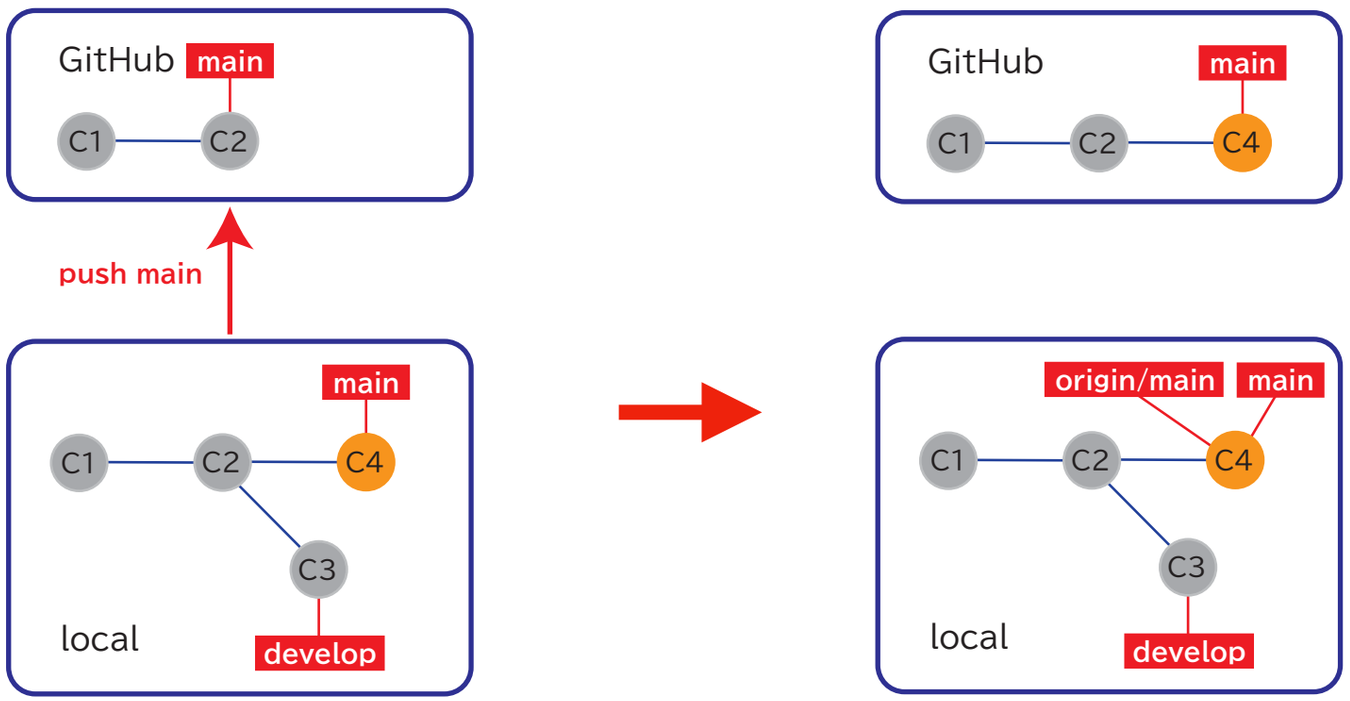


GitHub Demonstration: sync repositories

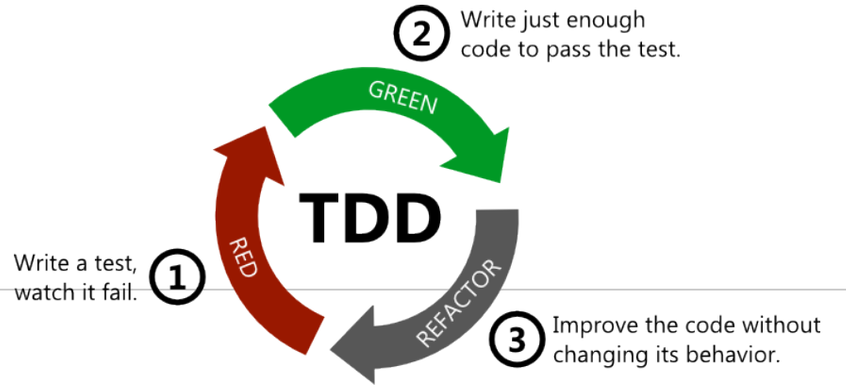
- local repository → GitHub / ローカルのリポジトリ → GitHub
  - Specify branch to be duplicated / 複製するブランチを指定する (e.g. main)

```
$ git push origin main
```

  - Fails if someone else has pushed to main branch on GitHub / 他の人がGitHub上のmainブランチにpushしている場合には失敗する
    → fetch & merge first, then push / fetch & merge してから push する



TDD Workflow



- **Red**
  - Write a failing automated test before you write any code / コードを書く前に失敗する自動テストコードを必ず書く
    - git fetch → git merge → git checkout
    - vim → cmake → make → ctest
- **Green**
  - Write just enough code to pass the test / テストをパスする最低限のコードを書く
    - [vim → cmake → make → ctest ...] → git add → git commit ...
    - git checkout → git merge → git push
- **Refactor**
  - Remove duplication / 重複を除去する
    - git checkout
    - [[vim → cmake → make → ctest ...] → git add → git commit ...]
    - git checkout → git merge → git push

Using Git & GitHub Successfully

- Tips for using Git & GitHub successfully / Git & GitHubをうまく使うコツ
  - Be well aware that there are 3 layers / 階層が3つあることを十分に認識する
    - local working space ⇔ local repository ⇔ remote repository (GitHub)
    - local working space ⇔ local repository
      - git checkout & git commit
    - local repository ⇔ remote repository
      - git fetch & git push
  - Always create a branch and work on it / 必ずブランチを作って作業する
  - Commit frequently / マメにコミットする
  - Think VCS as backup / バックアップだと考える
  - Test frequently / マメにテストする
  - Compile & make sure all tests pass before merging into main branch / コンパイル & 全てのテストが通ることを確認してから main ブランチにマージする

Part II: Version Control / バージョン管理

Continuous Integration（CI）/ 継続的インテグレーション

More Automated Testing

- Ideal / 理想
  - In TDD, main branch is always in Green state / テスト駆動開発においては、mainブランチは常にGreen状態
- Real / 現実
  - Forget to run tests / テストを忘れる
  - Tests may not pass in other environments (OS, compiler) / 他の環境(OS・コンパイラ)ではテストが通らないことも
  - Can’t know if others are running tests correctly / 他の人がテストを正しく実行しているか分からない

What’s Continuous Integration?

- Continuous Integration（CI）/ 継続的インテグレーション（CI）
  - Automatically run tests when code is commited to the repository / コードがリポジトリにcommitされたタイミングで自動的にテストを走らせる
  - Cumbersome to set up such a configuration on local PC / ローカルPCでそのような設定をするのは面倒

- GitHub Actions https://docs.github.com/ja/actions
  - Execute any job at the time of push to GitHub (or other) / GitHubにpushした(あるいは別の)タイミングで任意のジョブを実行
  - Use for CI / CIに利用する

- Setting up GitHub Actions / GitHub Actionsの設定
  - Create a YAML (.yml) file under .github/workflows / .github/workflows の下にYAML (.yml)ファイルを作成

GitHub Actions

- Create a YAML (.yml) file under .github/workflows / .github/workflows の下にYAML (.yml)ファイルを作成

.github/workflows/unittest.yml

```
name: unittest
on: [push]
jobs:
  build:
    runs-on: ubuntu-latest
    steps:
    - name: checkout
      uses: actions/checkout@v3
    - name: cmake
      run: mkdir build && cd build && cmake ..
    - name: make
      run: cd build && make
    - name: ctest
      run: cd build && ctest
```

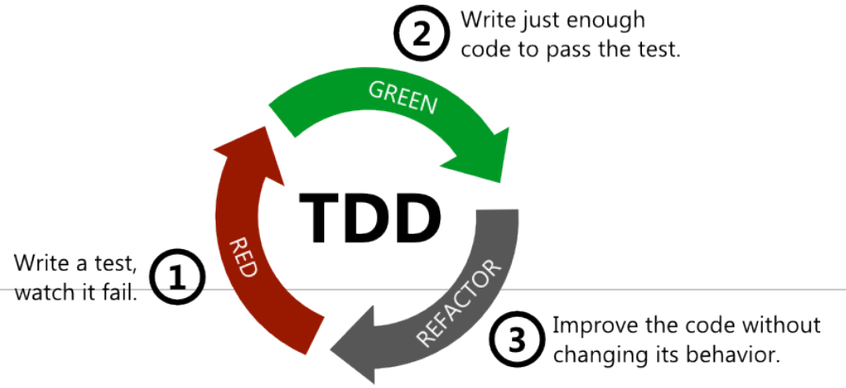
CI by GitHub Actions

- Confirmation of results / 結果の確認
  - GitHub Repository Web → 「Action」 tab
  - You will receive an email if the automated test fails / 自動テストがfailした場合にはメールが届く

- Using matrix
  - Matrix functionality allows to run tests on multiple versions of compilers, multiple operating systems, etc / Matrix機能を使うと、複数バージョンのコンパイラ、複数のオペレーティングシステムなどでテストを実行できる
  - https://docs.github.com/ja/actions/using-jobs/using-a-matrix-for-your-jobs

- Various other tasks can be automated / それ以外にも様々な作業を自動化できる

TDD Workflow



- **Red**
  - Write a failing automated test before you write any code / コードを書く前に失敗する自動テストコードを必ず書く
    - git fetch → git merge → git checkout
    - vim → cmake → make → ctest
- **Green**
  - Write just enough code to pass the test / テストをパスする最低限のコードを書く
    - [vim → cmake → make → ctest ...] → git add → git commit ...
    - git checkout → git merge → git push
- **Refactor**
  - Remove duplication / 重複を除去する
    - git checkout
    - [[vim → cmake → make → ctest ...] → git add → git commit ...]
    - git checkout → git merge → git push

Epilogue: Summary and Final Words

- By using TDD, VCS, etc
  - We can develop programs with fewer defects without being anxious / 不安をかかえこまずに、欠陥の少ないプログラムを開発できる
  - We can use TDD and VCS in various situations, from small tools to huge software, from individual to large-group development / 小さなツールから非常に大規模なソフトウェアまで、個人開発から大きなグループでの開発まで

- Don’t Repeat Yourself (DRY) principle / DRY原則

- Think of every line of code you write as a message for someone in the future / 自分の書くコードはすべて未来の誰かへのメッセージだと考える
  - including yourself :-) / 含む自分自身w
  - You yourself last month/last week/three days ago were a stranger / 先月の/先週の/3日前の自分は他人

References

- 「Test Driven Development: By Example」 by K. Beck
  - https://www.oreilly.com/library/view/test-driven-development/0321146530/
  - https://shop.ohmsha.co.jp/shopdetail/000000004967/

- 「Pro Git」 by S. Chacon and B. Straub
  - https://git-scm.com/book/ja/v2

- 「97 Things Every Programmer Should Know」 ed. K. Henney
  - https://github.com/97-things/97-things-every-programmer-should-know
  - https://ja.wikisource.org/wiki/カテゴリ:プログラマが知るべき97のこと

- Sample Code Repository
  - https://github.com/wistaria/tdd-tutorial