ユニットテスト

2020年11月06日（金）
システム開発プロジェクト応用第一

東京工業大学
特任助教　内田公太

本講義の概要

- 実際のシステム開発プロジェクトの現場で使われている現代的な開発ツールや手法を学ぶ
  - 正しいツールや手法の選択はソフトウェア開発を効率的に，そして楽しいものにする

到達目標：

- 現代的な開発ツールの基本的な使い方と適する用途が分かる

講義で扱うトピック

- 情報収集
- GDB
- Git
- バグトラッキング
- GitHub & Pull Request
- ユニットテスト
- 継続的インテグレーション
- デプロイと冪等性
- コミュニケーション

自己紹介

- 内田公太
- Twitter @uchan_nos
- 週3日：サイボウズ・ラボ株式会社
  週2日：東工大の特任助教

- osdev-jpコアメンバー
- 『30日でできる! OS自作入門』の校正担当
- 『自作エミュレータで学ぶ
  x86アーキテクチャ』の著者 →

講義の進め方

スタイル：

- 少し講義して演習，の繰り返し

成績評価：

- 現代の開発技術・手法の理解度を評価する
- 各トピックを受講者自身のソフトウェア開発プロジェクトに適用し，レポートおよびリポジトリを提出する
- レポートおよびリポジトリの充実度で成績を決定する

レポート，リポジトリの充実度

- 色々な要素がある
- トピックに対する回答
- ドキュメント
- コミットメッセージ
- プルリクのやり取り
- Etc.

- 総合的に判断して評価します

課題提出の方法

- 課題を含めたリポジトリとレポートを作成し，提出
- 初回（10/2）説明したので詳しい話はしないつもり
  - 改めて聞きたい方がいたらお知らせください

ユニットテスト

テスト

中間・期末テスト

- 学生の理解が正しいことを確かめる
- 入力=問題文，出力=答案

関数のテスト

- 関数が正しく動くか確かめる
- 入力=引数，出力=戻り値

テストケース

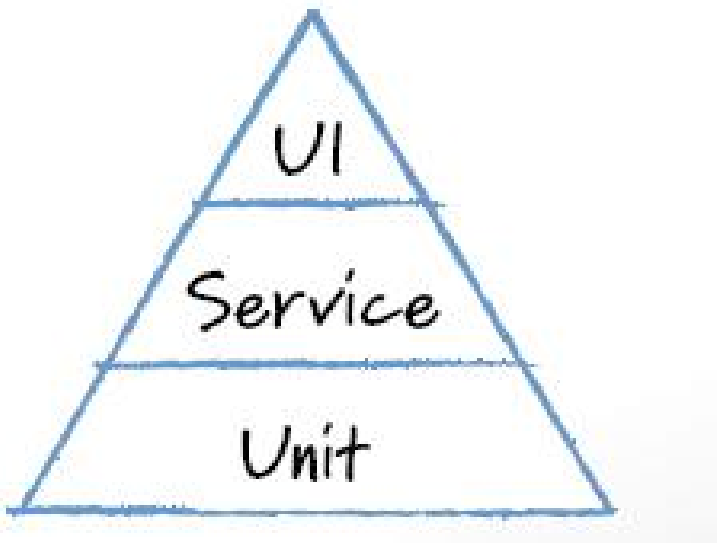
- テストの入力と出力の組
- テストの入力はテスト対象により異なる
- コマンドのテスト
  - コマンドライン引数，環境変数
- 関数のテスト
  - 関数の引数，グローバル変数
- その他，入力となり得るもの
  - ファイル，標準入力
  - パソコンの時刻設定
  - ネットワーク上のリソース

テストの出力

- テストの出力もテスト対象により異なる
- コマンドのテスト
  - Exit code
- 関数のテスト
  - 関数の戻り値，グローバル変数
- その他，出力となり得るもの
  - ファイル，標準出力，標準エラー出力
  - ネットワーク上の通信
  - 実行時間
  - 特定の関数が呼び出されたかどうか

テストピラミッド

- 実行コストが低いテストほど増やすことで，品質を維持したままテスト時間を減らす戦略
  - UI: E2Eテストとも。ユーザがシステムを使うシナリオに沿ったテスト
  - Service: 結合テストとも。機能（サービスやAPI）単位のテスト
  - Unit: 関数やクラス単位のテスト
- テスト種別に拘らず，高コストなテストを減らす意識が重要



https://www.mountaingoatsoftware.com/blog/the-forgotten-layer-of-the-test-automation-pyramid

UIテスト

- ユーザーが操作する画面でテストシナリオを流す
- ユーザー操作に起因する幅広いコンポーネントが動作
  - フロントエンド
  - Webサーバー
  - データベースサーバー
- 範囲は広いが，テストされる実行パスは少ない
- テスト環境の整備が大変
  - データベースにテスト用データを入れたり
- 1つのテストケースの実行時間が通常かなり長くなる

自作コンパイラのUIテスト

```
function test_exit() {
  want="$1"
  input="$2"

  echo "$input" | ./opelac > tmp.s
  nasm -f elf64 -o tmp.o tmp.s
  cc -no-pie -o tmp tmp.o cfunc.o
  ./tmp
  got=$?

  if [ "$want" = "$got" ]
  then
    echo "[  OK  ]: $input -> $got"
    (( ++passed ))
  else
    echo "[FAILED]: $input -> $got, want $want"
    (( ++failed ))
  fi
  rm tmp tmp.o tmp.s
}
```

テストの入力値

終了コードを取得

```
test_exit 42 'func main() { 42; }'
test_exit 11 'func main() { 1+23 - 13; }'
test_exit 2 ¥
  'func main() { 12/2 - 2 * (3 - 1); }'
test_exit 5  'func main() { -3 + (+8); }'
test_exit 0  'func main() { 3 < 1; }'
test_exit 1  'func main() { 2*3 >= 13/2; }'
test_exit 1  'func main() { 2>2 == 4<=3; }'
test_exit 0  'func main() { }'
test_exit 3  'func main() { 42; 3; }'
test_exit 15 ¥
  'func main() { foo:=5; bar:=3; foo*bar; }'
test_exit 3  'func main() { return 3; 42; }'
test_exit 2 ¥
  'func main() { 1; if 42 > 10 { 2; } }'
```

結合テスト

- サービスやAPIが仕様通りに動くかテストする
- 基本的にそのサービスしか動作しないため，テスト実行はそこそこ短時間
  - そのため，UIテストよりは数をこなせる
- サービスの組み合わせで発生する不具合は見つからない

ユニット（単体）テスト

- 関数やクラス（ユニット）の単位で動作を確かめる
- 1つの関数やクラス，および依存する部品しか動作しないため超高速にテスト可能
  - 多くのテストケースを準備しても大丈夫
- ユニットテストはプログラミング行為と直接結びつく
  - 関数を実装する≒関数の入出力の仕様を決める≒テストケースを書く

ユニットテストのテストケース

```
uchan@workstation:~/workspace/cpptest$ cat a.c
#include <stdio.h>
#include <string.h>

char* concat_str(char* s, const char* addend) {
  while (*addend) {
    *s++ = *addend++;
  }
  *s = '\0';
  return s;
}

int main() {
  char s[128] = "foo";
  concat_str(s, " bar");
  if (strcmp(s, "foo bar") != 0) {
    printf("want 'foo bar', got '%s'\n", s);
    return 1;
  }
}
uchan@workstation:~/workspace/cpptest$ ./a.out
want 'foo bar', got ' bar'
uchan@workstation:~/workspace/cpptest$
```

- concat_strは文字列を結合する関数
  - sの末尾にaddendを結合する
- テスト対象：concat_str()
- テストケース：main()の中

演習1

自分のOSSにテストを1つ追加せよ

- テストの種類は何でも良い
  - UI，サービス，ユニット
- テストフレームワークを使っても良いが必須ではない

- 制限時間15分

ユニットテストが持つべきでない性質

- 『レガシーコード改善ガイド』より
- 実行に0.1秒もかかる単体テストは，遅い単体テストである
- 次に当てはまるものは単体テストではない
  1. データベースとやり取りする
  2. ネットワークを介した通信をする
  3. ファイルシステムにアクセスする
  4. 実行するために特別な環境設定を必要とする（環境設定ファイルの編集など）

回帰テスト（regression test）

- プログラムの変更による劣化を検出する目的
- プログラム変更による劣化とは
  - とある関数を「改良」したら，その関数を使う機能がバグる
  - 変数名を変更したら思わぬところに影響が出る
- プログラムを変更する度にテストしたい
- 手動でやる
  - 実装を固定化し，テスト期間を設けて回帰テストを実施
- 自動でやる
  - コードがPushされたらテストを走らせる

ユニットテストフレームワーク

- ユニットテストの記述，実行，集計をサポート
- 記述
  - 出力値の比較機能（配列等の構造体の比較）
- 実行
  - 通常のプログラムとテストを分離してテストだけ実行
  - 通常実行時はテストが実行されないようにもする
- 集計
  - テスト成功数，失敗数を計算
  - テストに失敗した箇所を表示
  - テストカバレッジの表示

演習2

テストフレームワークを使って
自分のOSSにテストを追加せよ

- ユニットテストをいくつか追加する
- 制限時間15分

ユニットテストが書きにくい関数

- 外側から制御できないパスにあるファイルを開く
  - テストのために本番と同じ場所にファイルを置く必要がある
- ネットワーク送受信をする
  - 接続先のダミーを用意しなければならない
  - 接続先が固定だと超絶面倒
- 標準入出力を使う
  - 標準入出力の置き換えが必須
- 現在時刻に依存した処理をする
  - 前回の処理からN秒以上経っている，みたいなロジック
  - 現在時刻を使ってしまうとテストケースに時刻を埋められない

具体例：外側から制御できないパス

```
int get_value(const char* key) {
  FILE* conf = fopen("/etc/myapp.conf");
  // conf から key に対応する値を探す
  // int value = ...;
  return value;
}
```

- /etc/myapp.confは関数外から変更不能
- テスト時はそこにファイルを置く必要がある
- /etcはrootでしか書けないのでテストに不適

具体例：現在時刻に依存した処理

```
void push_item(ITEM* item) {
  item->next = NULL;
  item->time = current_time();

  item_list_last->next = item;
  item_list_last = item;
}
```

```
ITEM* pop_item() {
  ITEM* item = item_list;
  if (add_time(item->time, TIME_1HOUR)
        < current_time()) {
    item_list = item->next;
    return item;
  }
  return NULL;
}
```

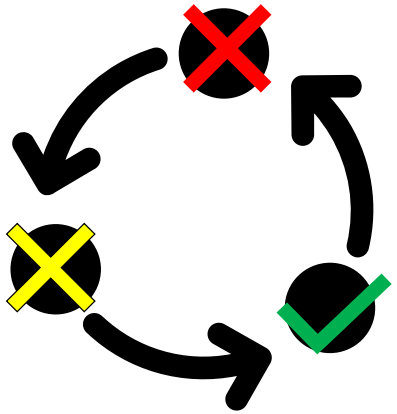
- pop_item():
  現在時刻+1時間経たないとテスト不能
- 無理やりテストするにはPCの時計を変更するしかない

テスト駆動開発（Test Driven Development）

- テストを充実させる戦略
- テストを書く＝関数の仕様を決める
  - 仕様を先に決めると実装方針が立ちやすい
- テストを先に書く→必然的にテストを書きやすい実装になる

TDDのやり方

1. 作りたい関数の仕様を決める
2. テストを書く
3. テストを動かす→コンパイルが失敗する
4. 実装が空の関数を書く
5. テストを動かす→テストが失敗する
6. 関数を実装する
7. テストを動かす→テストが成功する



演習2

テスト駆動開発により
自分のOSSに機能を追加せよ

- テストの種類はユニットテスト
- テストフレームワークを使っても良いが必須ではない
- 必ず「コンパイル失敗→テスト失敗→テスト成功」の順で

- 制限時間35分

レポート提出のトピック名

- 11/6の前半はTOPIC=github
- 11/6の後半はTOPIC=ut

ユニットテスト
後編

11/13（金） 14:20-16:00

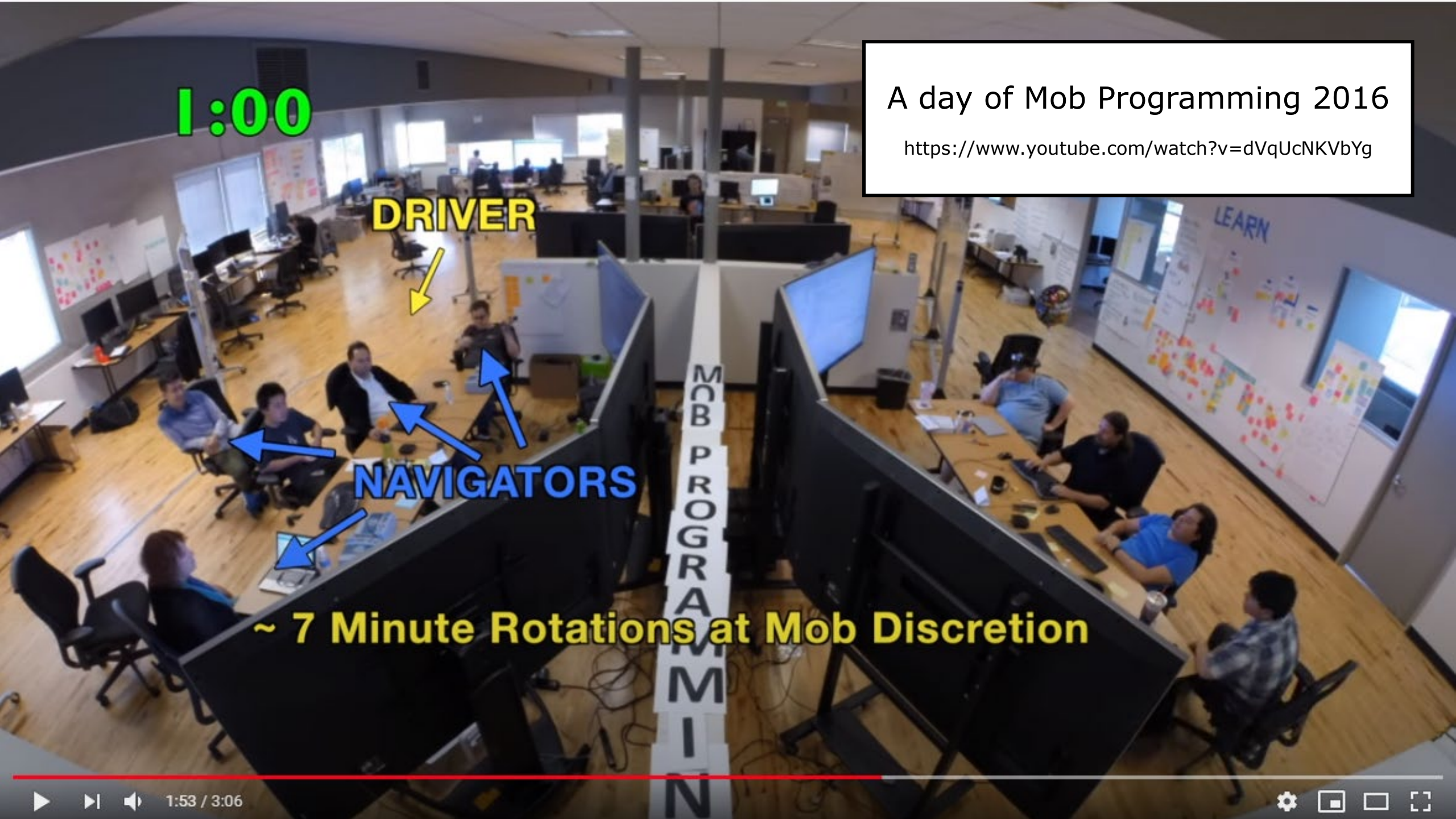
TDDワイワイ会

- TDD+モブプログラミングでワイワイする会
- https://tddyyx.github.io

- モブプログラミング
- 複数人が同時に同じ課題に取り組むプログラミング手法
- ドライバー（1人）：コードを書く
- ナビゲーター（その他）：ドライバーに指示する

A day of Mob Programming 2016

https://www.youtube.com/watch?v=dVqUcNKVbYg



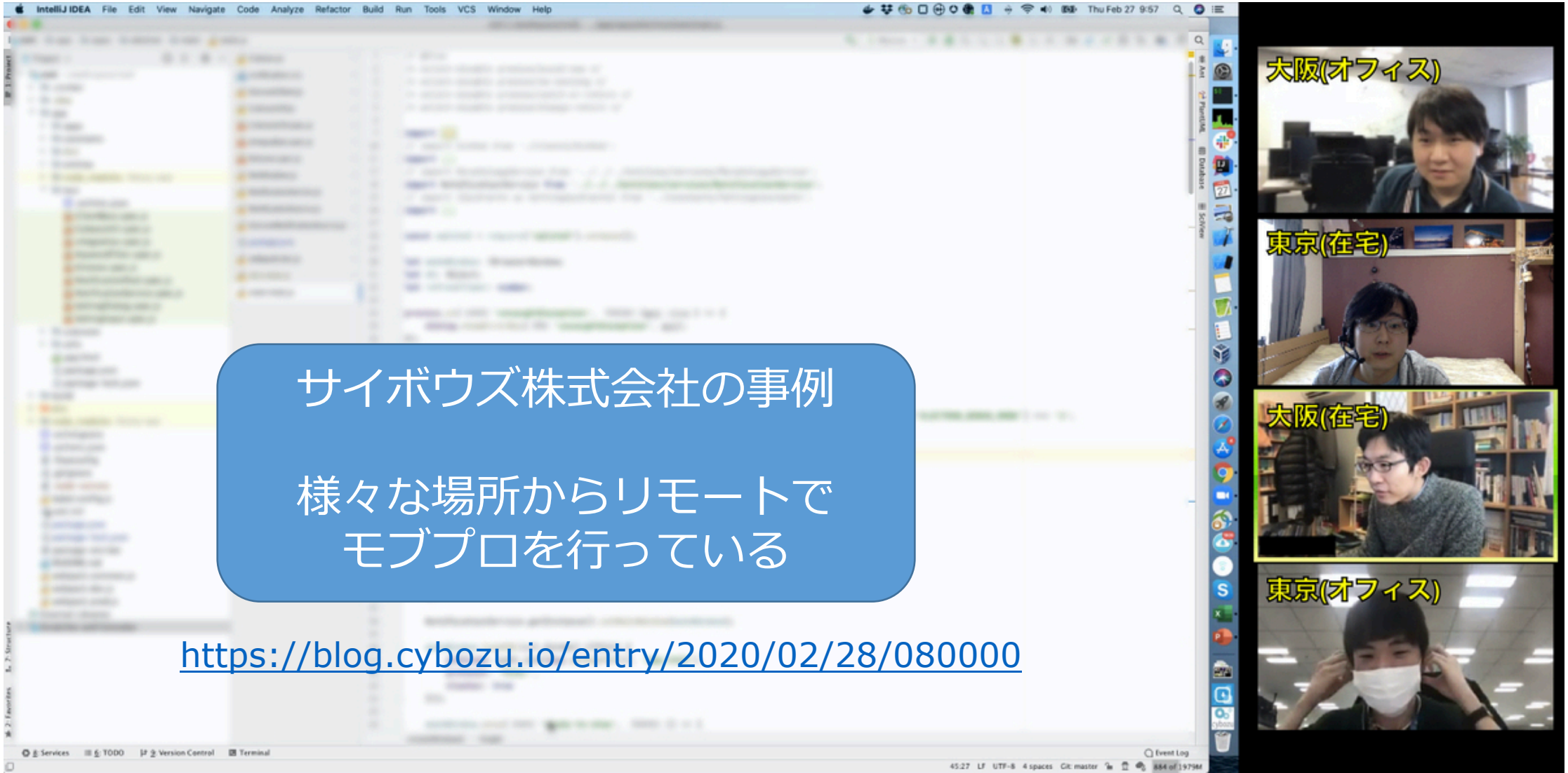
ドライバーとナビゲーター

- ドライバー
  - ナビゲーターの議論の結果をコードで表現する
  - 自分の考えだけでコーディングを進めない
- ナビゲーター
  - 何を実装するかを議論する
  - ドライバーが理解できる最も高い抽象度の表現で伝える
- チームの文化によって役割の差はある

モブプロの良さ

- 互いに分からないことを補完しあえる
  - 調べる時間が短縮される
  - 不正確な知識を得てしまう可能性が下がる
- チームメンバー皆が内部構造等に詳しくなる
  - ドキュメント量を削減できる
  - チームメンバーの知識レベルを揃えられる
  - 属人性をある程度防げる
- チームでわいわい楽しい！

リモート・モブプログラミング



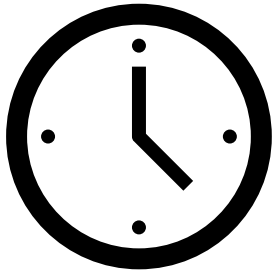
サイボウズ株式会社の事例

様々な場所からリモートで
モブプロを行っている

https://blog.cybozu.io/entry/2020/02/28/080000

リモート・モブプロのやり方

- ドライバーの順序を決める
- ドライバーの交代方法を決める
  - 時間で区切る？機能単位？他の何か？
  - リモートチーム用タイマー https://cuckoo.team/titech-sysdev



- ナビゲーターが議論し，ドライバーがコードを書く
- ドライバーの交代時間になったらコードを共有
  - https://docs.google.com/document/d/17EfZQZ-788giRuAWurezloz-1Va6P7c92ZKqzb9d0AY/edit?usp=sharing
- 適度に休憩時間を設ける

ドライバーの技術

- 自分の考えだけでコーディングを進めない
  - ナビゲーターが静かに見守るだけ，になりがち
  - ナビゲーターからの案を素直に実装するのが基本
    - もちろん，よりよいアイデアがあれば議論に参加しよう

- 自分の思考を垂れ流す（しゃべる）
  - どんなコードを書こうと思っているかを垂れ流す
  - 分からないことがあれば，分からないことを伝える
  - 難易度が高い技術なので練習しよう

演習4

TDD+モブプログラミング
で100 doorsを解け

- 適当に人数を分けます
- https://cyber-dojo.org/ Exercise ID = AXzDFQ

- 制限時間90分

課題提出

- 課題を含めたご自身のリポジトリとレポートを提出
- 提出先は内田のGitHubリポジトリ
  - https://github.com/uchan-nos/titech-sysdev-2020
  - プライベートリポジトリのためアカウント登録必須
    皆さんのGitHubアカウントを教えてください
- このリポジトリに対し，レポートを送る
  - レポートには，トピックに対する回答を含める
- 提出期限は講義の1週間後の10:00（JST）

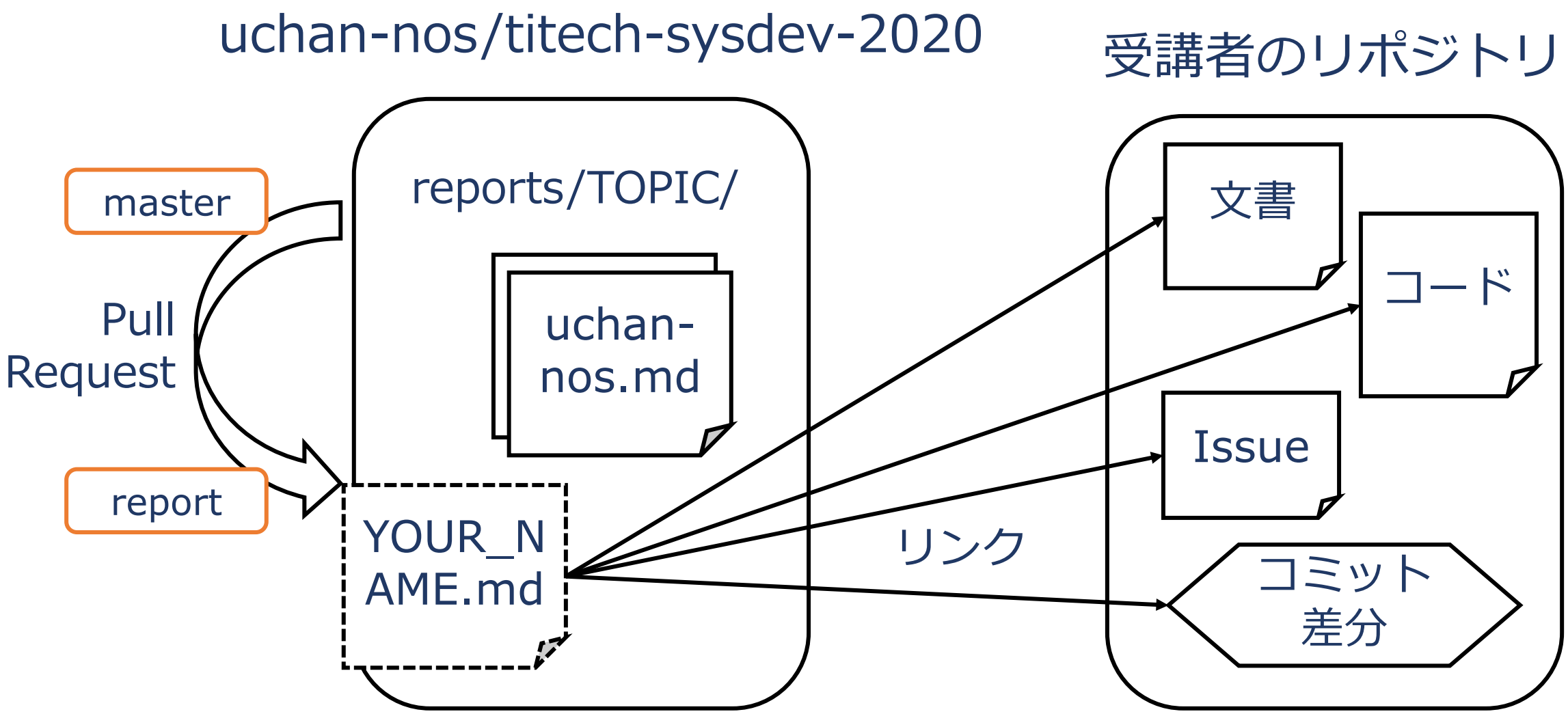
レポートの送り方

1. 独自のブランチを作る
   1. titech-sysdev-2020:master
      ↓branch
      titech-sysdev-2020:report-YOUR_NAME
2. 回答の概要をまとめたファイルを加える
   1. titech-sysdev-2020/reports/TOPIC/YOUR_NAME.md
   2. Commit & Push
3. プルリクを送る（リポジトリ内プルリク）
   1. titech-sysdev-2020:report-YOUR_NAME
      ↓pull request
      titech-sysdev-2020:master

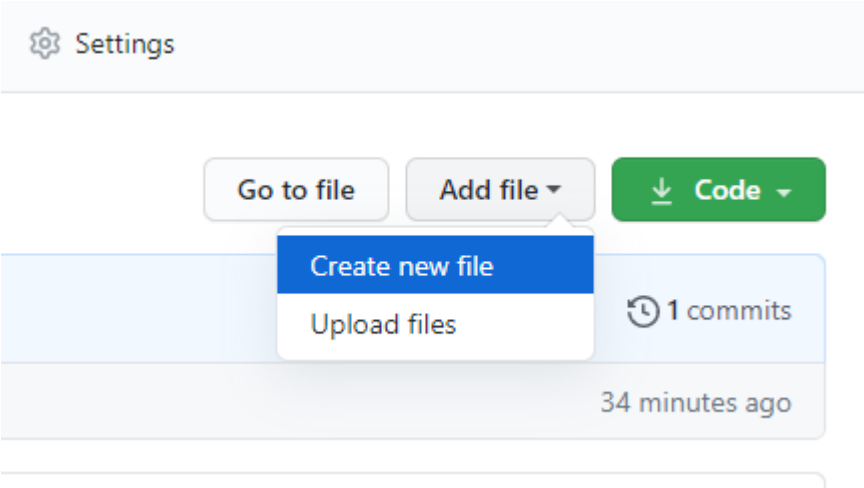
レポート

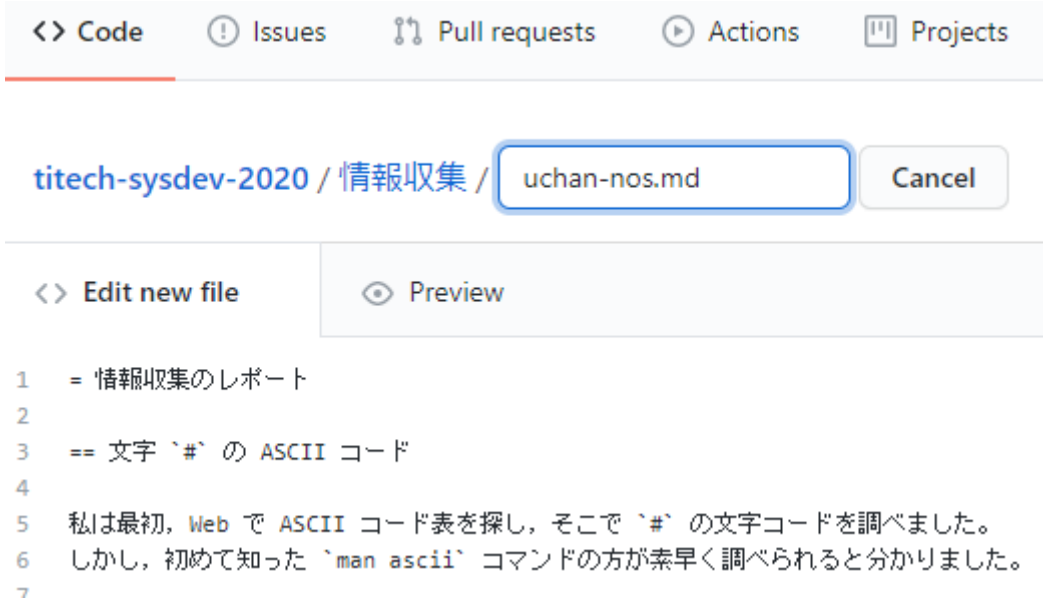
- reports/TOPIC/YOUR_NAME.md
- このファイルに課題への回答を記載する
- 必要なら以下のものを含める
  - Issueへのリンク
  - コミット差分へのリンク
    https://github.com/HOGE/REPO/compare/COMMIT1...COMMIT2
  - その他
- 要するに，成績評価に必要な情報をYOUR_NAME.md自体に記載するか，そこから辿れるようにする

レポートの構成


uchan-nos/titech-sysdev-2020
受講者のリポジトリ
master
Pull Request
report
reports/TOPIC/
uchan-nos.md
YOUR_NAME.md
リンク
文書
コード
Issue
コミット差分

レポートの送り方 1/2

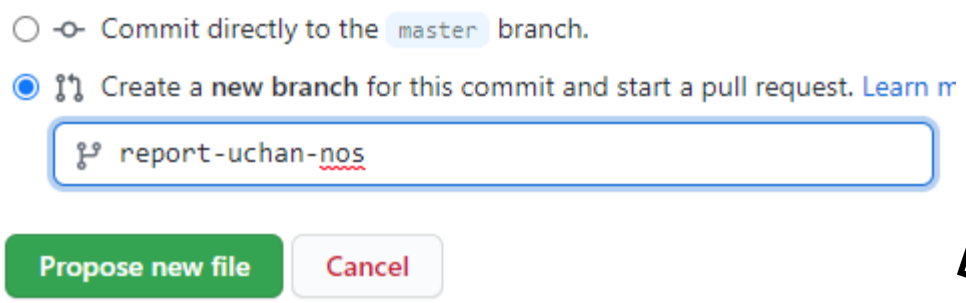




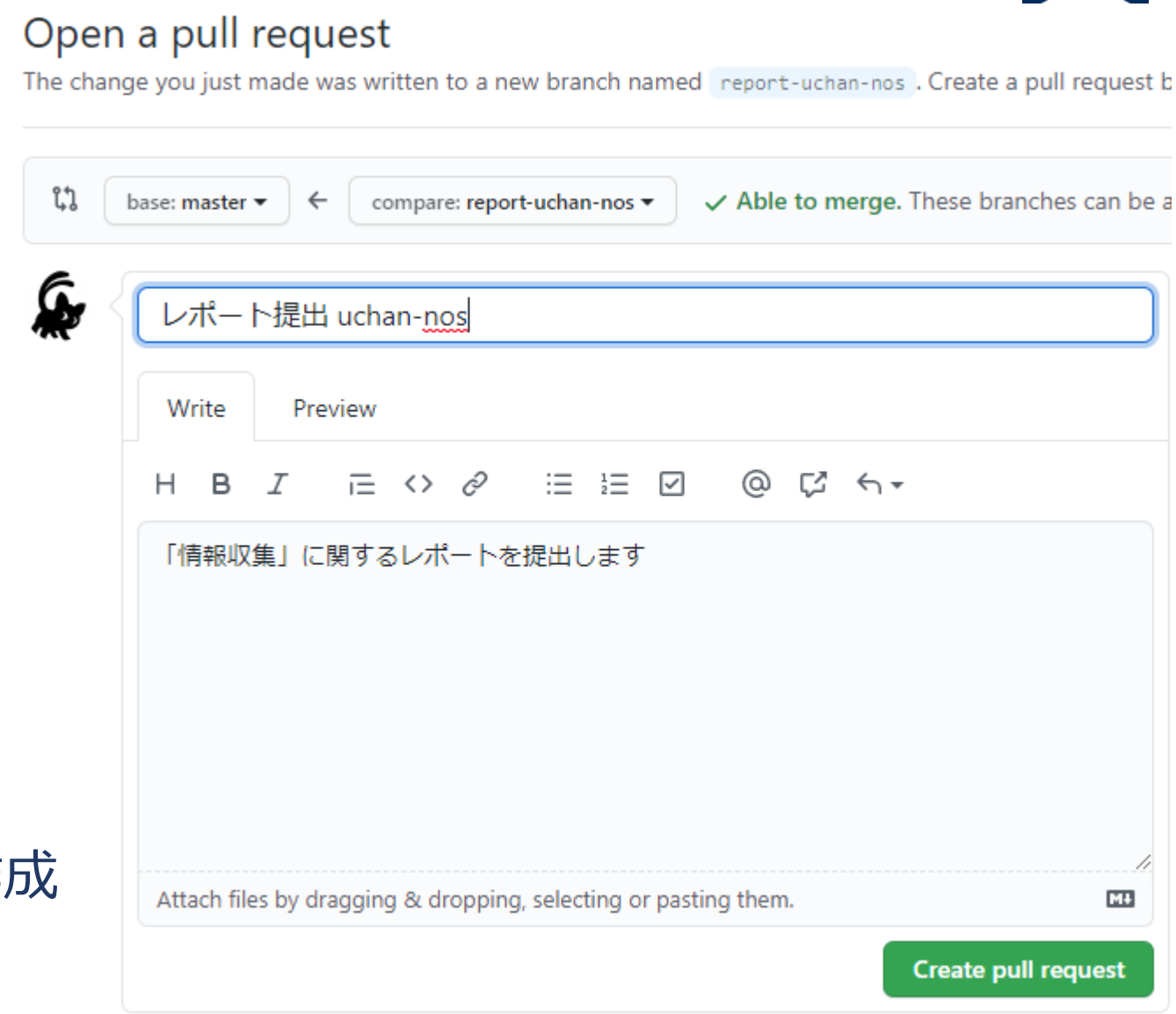
ファイルを新規作成

YOUR_NAME.mdの内容を記述

レポートの送り方 2/2

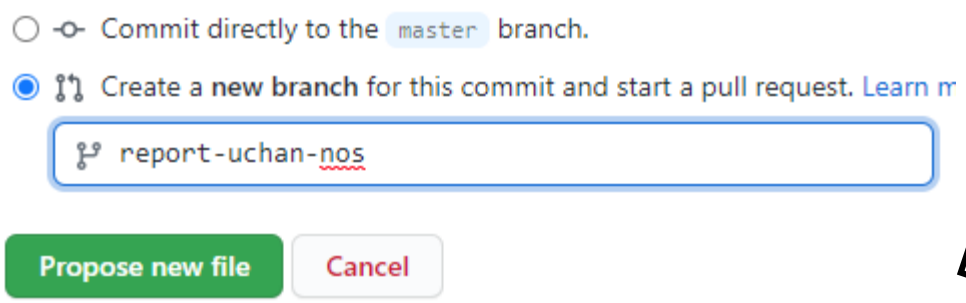


新規ブランチにコミット

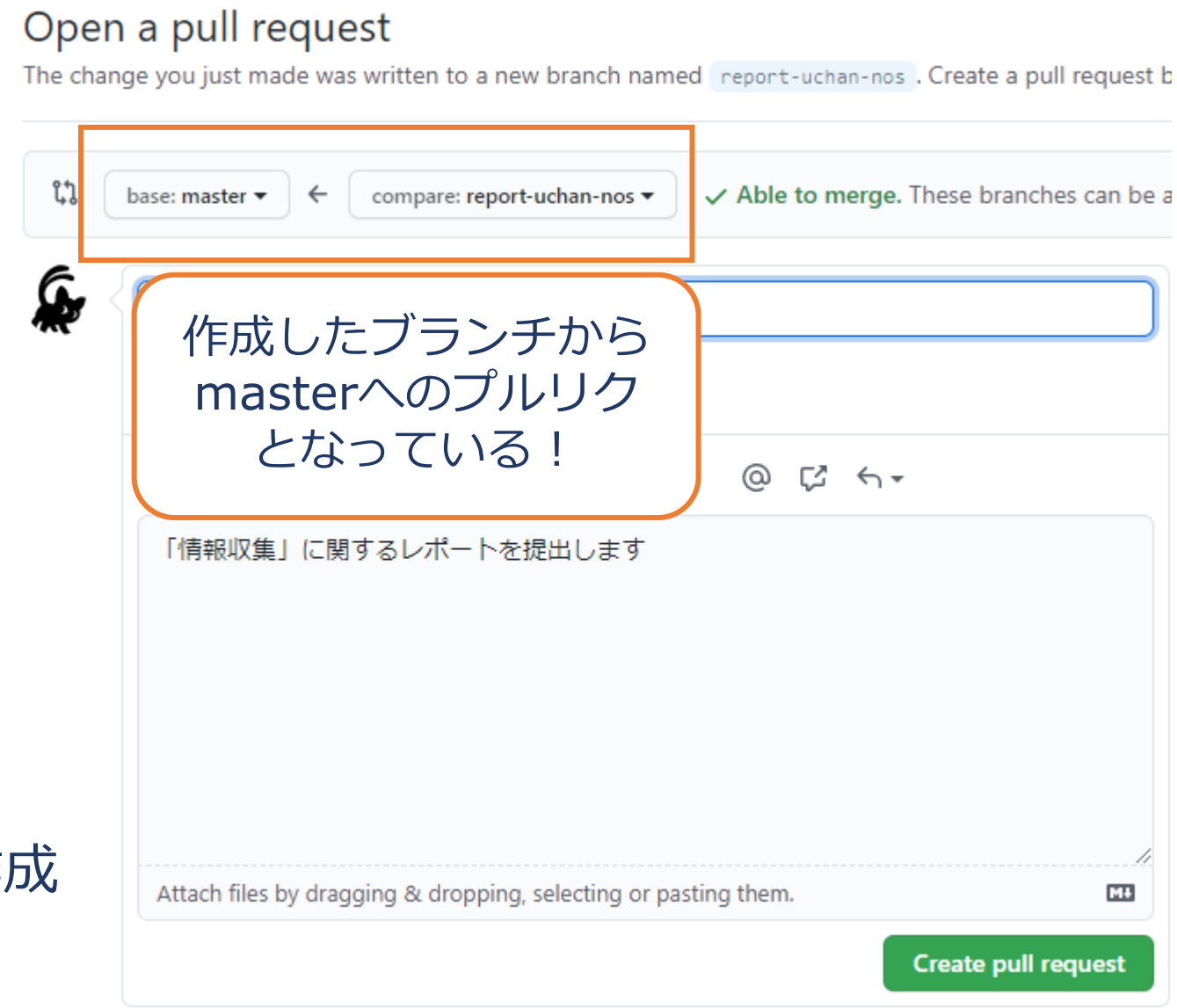


プルリクを作成

レポートの送り方 2/2



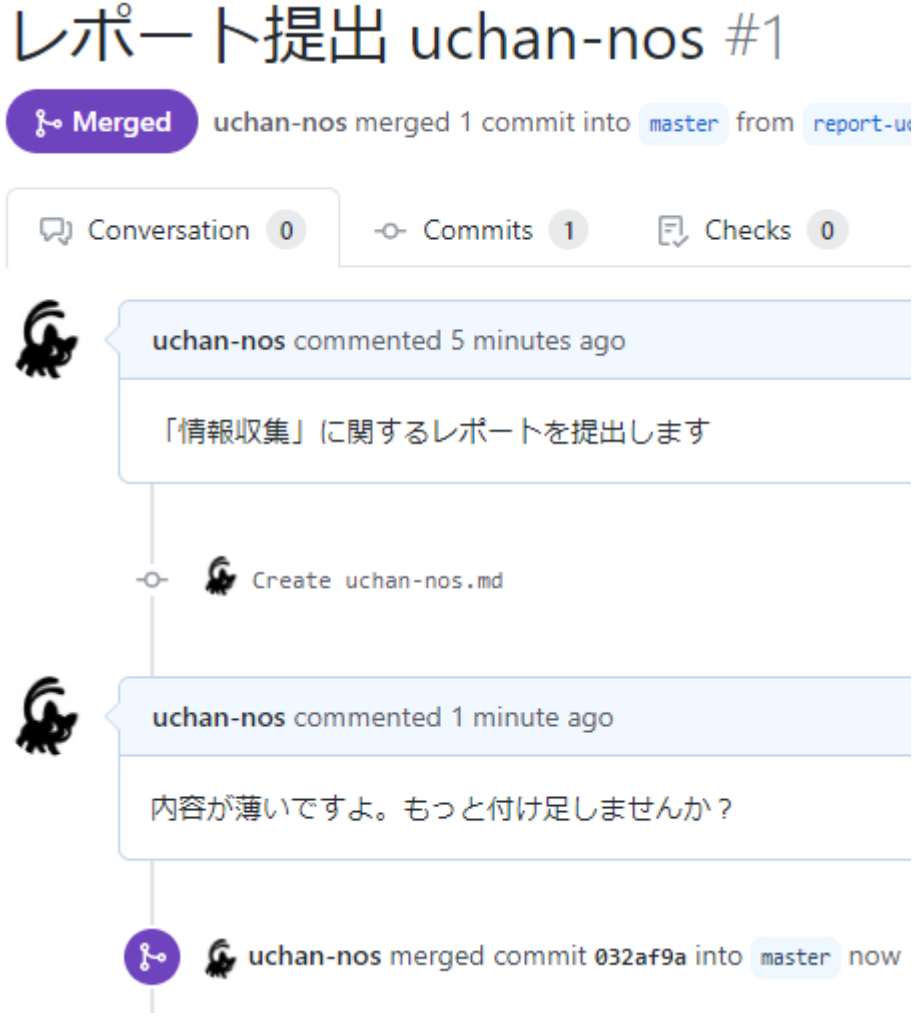
新規ブランチにコミット



作成したブランチから
masterへのプルリク
となっている！

プルリクを作成

レポートの受理



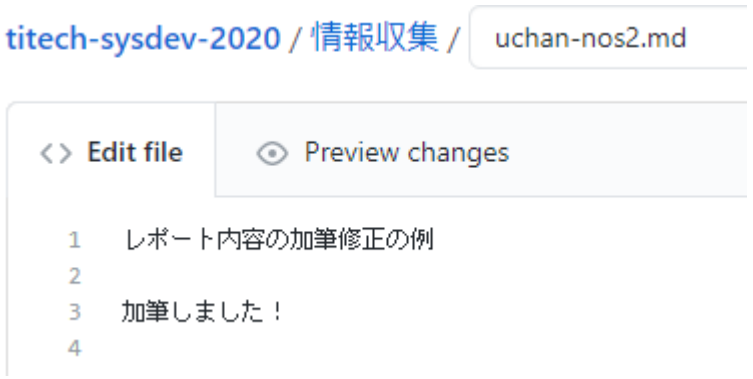
プルリクにコメントが付くことも

Merged：レポート受理済み

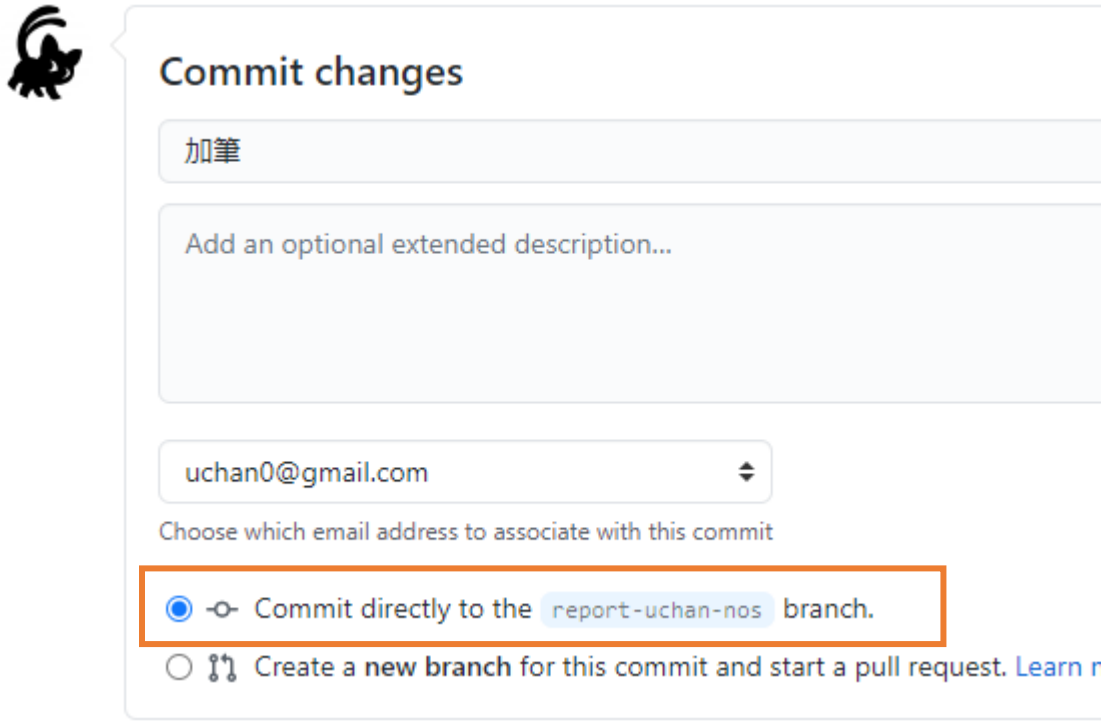
レポートの更新 1/2

レポートに不十分な個所があった！

まだマージされてないときの
更新方法を紹介



納得いくまで加筆修正



report-Xブランチにコミット

レポートの更新 2/2

一発目のコミット

Pull Requestに
コミットが追加されていく

レポートの送り方：その他の方法

- GitHubのWebインターフェースを使う必然性はない
- コマンドラインで作業してもよい
  - 具体的なコマンドラインは示しません
  - この講義は「情報収集」でしたね？
  - コマンドラインについて情報収集すれば，レポートをさらに充実させるネタになりますよ

次回予告

- 次回は11/13（金）14:20から
- 「ユニットテスト」の続きと
  「継続的インテグレーション」をやります