

プログラミング応用
<http://bit.ly/kosen02>

Week3@後期
2016/10/13

本日の内容

- 講義
 1. ソフトウェア開発の現状と問題
 2. ソフトウェア開発工程・方法論
 3. ソフトウェア開発の効率化
 - コーディングの効率化
 - 共同開発の効率化
- 演習
 - ペアプログラミング
 - Gitを用いた共同開発

本日の内容

- 講義

1. ソフトウェア開発の現状と問題
2. ソフトウェア開発工程・方法論
3. ソフトウェア開発の効率化
 - コーディングの効率化
 - 共同開発の効率化

- 演習

- ペアプログラミング
- Gitを用いた共同開発

例) Windowsのソースコード

- SLOC(Source lines of code): 行数

年	製品名	100万SLOC
1993	Windows NT 3.1	4-5
1994	Windows NT 3.5	7-8
1996	Windows NT 4.0	11-12
2000	Windows 2000	29以上
2001	Windows XP	40
2003	Windows Server 2003	50

例) レジシステムの複雑化

- 2000年以前
 - レジシステムは店舗内での会計処理に使われていた



例) レジシステムの複雑化

- 2000年代
 - レジシステムに公共料金の支払、チケットの発券、通販で購入した商品の受け取りなどを管理する機能が組み込まれた
 - 顧客の購買データは中央のサーバーに自動的に転送されデータ分析に利用されている

チケット販売

公共料金支払い

通販サービス



コンビニチェーン
本部
(のサーバ)

例) レジシステムの複雑化

- 2010年代
 - これまで大規模チェーンしか導入できなかった多機能レジが小規模店舗にも普及



タブレット端末で会計と
顧客情報を入力



世界中から収集した
データを解析し経営
改善に活かす

ソフトウェア開発の現状

- 大規模化
 - 必要なソースコードの行数が増大
 - 多くの開発者が共同で開発する必要あり
- 複雑化
 - 社会が複雑化するのに合わせ、ソフトウェアへの要求も複雑化

ソフトウェア開発の問題

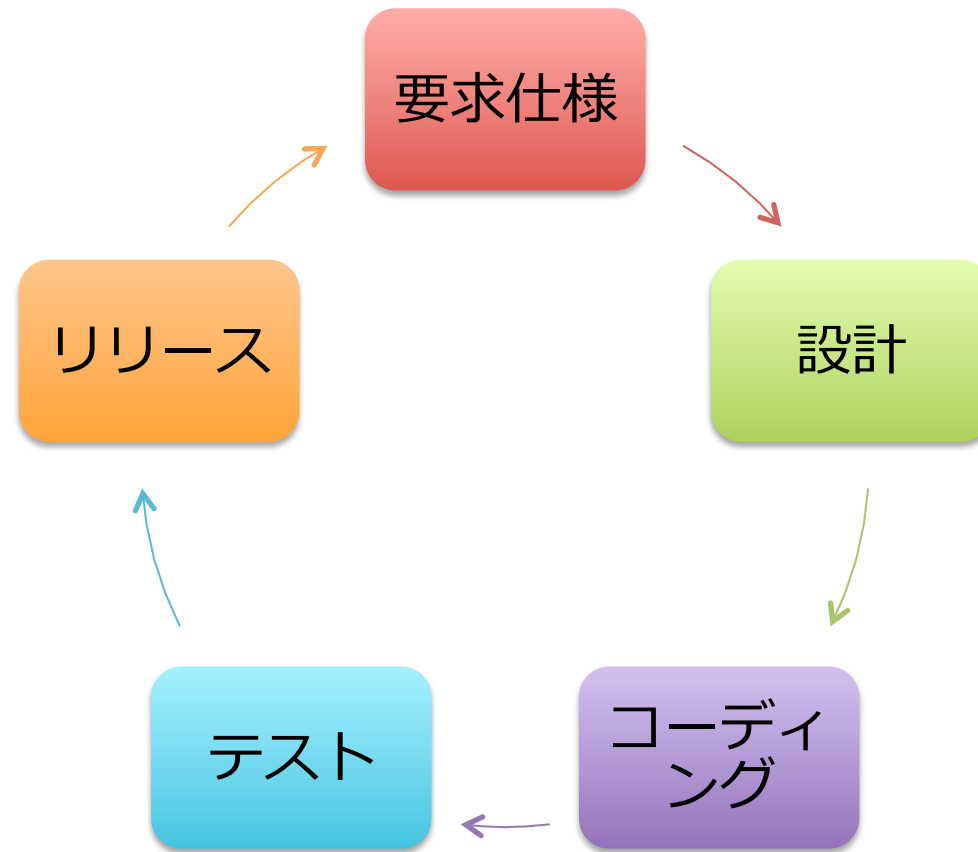
- 大規模化
 - 必要なソースコードの行数が増大
 - 多くの開発者が共同で開発する必要あり
 - 共同開発は難しい
 - 共同開発によるバグの増大
- 複雑化
 - 社会が複雑化するのに合わせ、ソフトウェアへの要求も複雑化
 - 複雑化により設計、実装がより難化

大規模化/複雑化したソフトウェアを
完成させる方法論が必要

本日の内容

- 講義
 1. ソフトウェア開発の現状と問題
 2. ソフトウェア開発工程・方法論
 3. ソフトウェア開発の効率化
 - コーディングの効率化
 - 共同開発の効率化
- 演習
 - ペアプログラミング
 - Gitを用いた共同開発

ソフトウェア開発工程

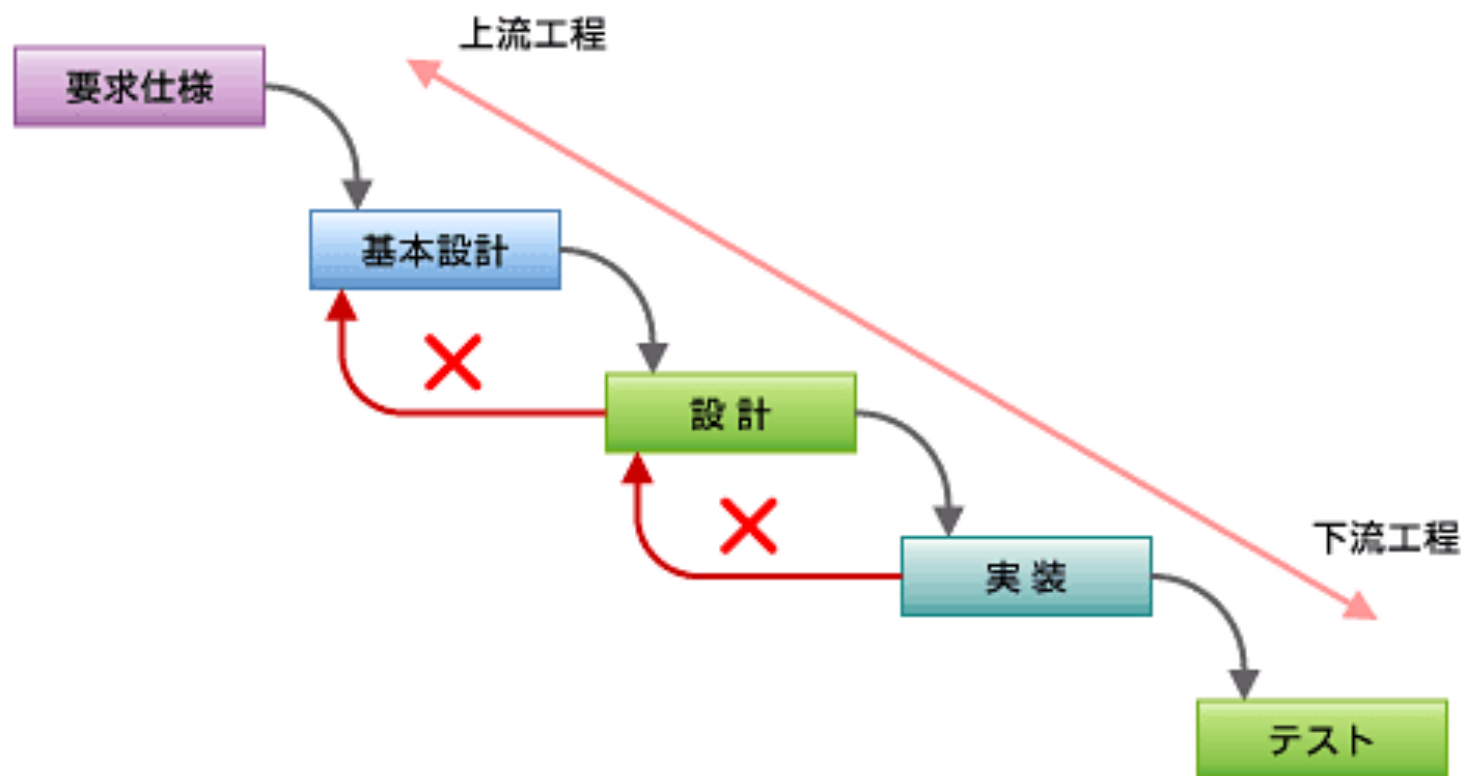


代表的な開発手法

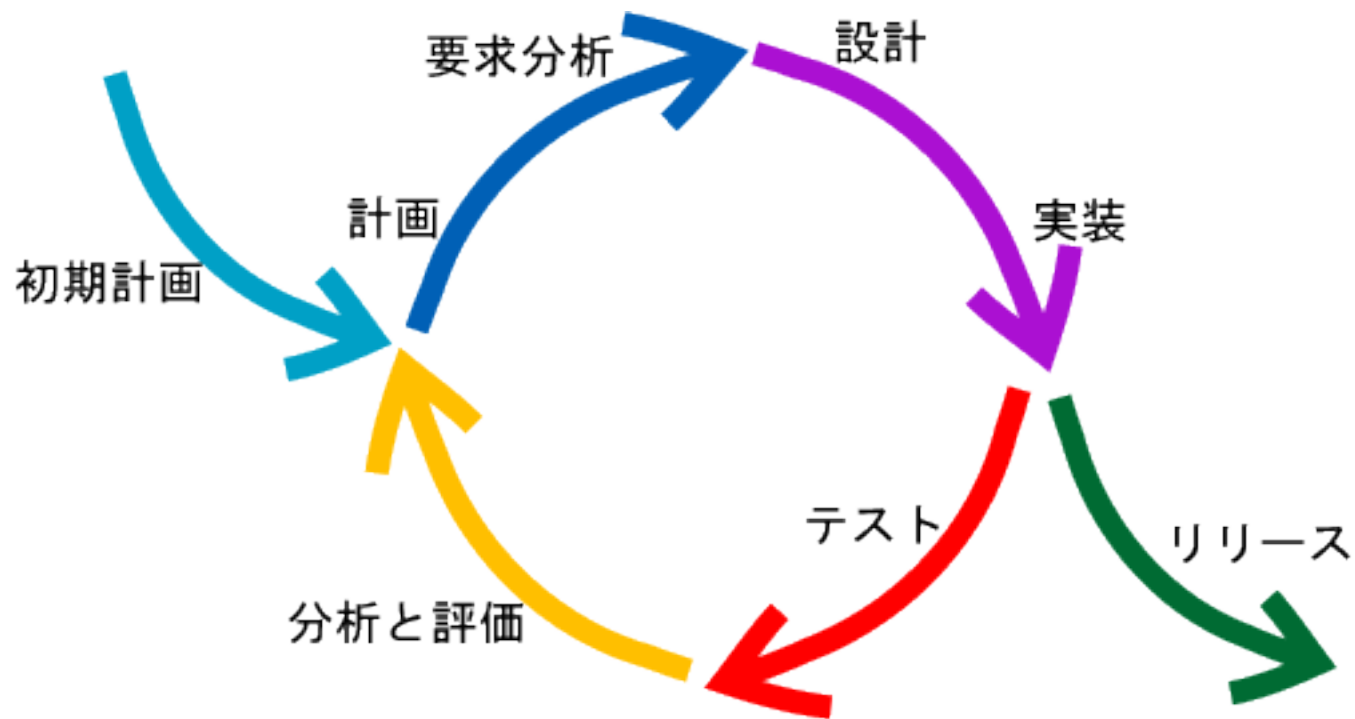
1. ウォーターフォールモデル
2. 反復モデル
3. アジャイル開発

ウォーターフローモデル

- メリット： 作業の見積もりが容易
- デメリット： 上流工程に問題があると破綻

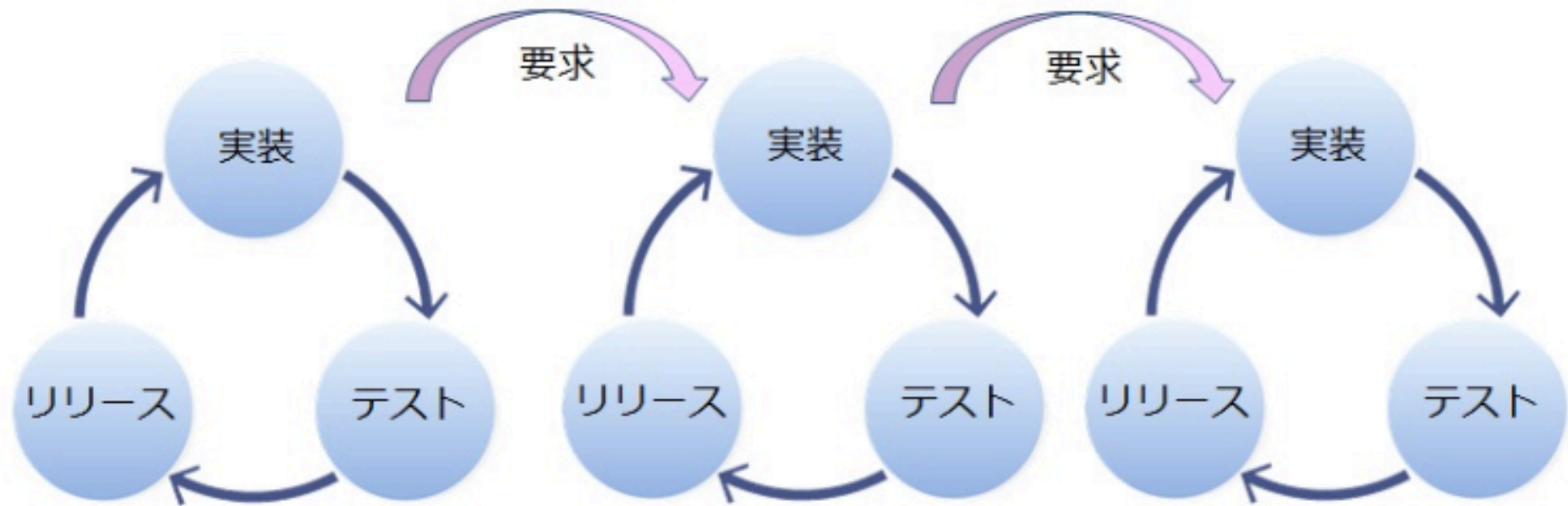


反復モデル



- 初期計画・計画 : 納期やおおまかな方針を決める
- 要求分析 : どのようなソフトウェアが必要か、顧客と話し合う
- 設計 : 顧客の要求に合うソフトウェアを仕様書に落とし込む
- 実装 : コーディングしソフトウェアにする
- テスト : 顧客の要求を満たすかテスト
- リリース : ユーザが使える状態にして公開

アジャイル開発



アジャイル開発では開発を短い期間（イテレーション）に区切り、イテレーションごとに必要最低限の機能を持った製品(MVP; Minimum Viable Product)を完成させる

完成後すぐに顧客と話し合い次に作るMVPを決める。

どの開発手法が最適か

- どの開発手法も一長一短
 - ウォーターフロー
 - 計画を重視
 - 仕様変更が少ない大規模金融システムや要求がすでに具体的なシステムに有用
 - アジャイル開発
 - 仕様変更に対応することを重視
 - 仕様変更が頻繁に起こる可能性がある場合に有用
 - 小規模なシステムでは近年スタンダードになりつつある
 - 反復モデル
 - ウォーターフローを改善したモデル

本日の内容

- 講義
 1. ソフトウェア開発の現状と問題
 2. ソフトウェア開発工程・方法論
 3. ソフトウェア開発の効率化
 - コーディングの効率化
 - 共同開発の効率化
- 演習
 - ペアプログラミング
 - Gitを用いた共同開発

コーディングの効率化

- 信頼性のある(=バグ)プログラムをより早く作るには？
→ ソフトウェア工学の分野でいくつか効率化手法が提案
- 代表的な効率化手法
 - コードレビュー(Code review)
 - ペアプログラミング(Paired programming)
 - テスト駆動開発(Test-driven development)

コードレビュー

- 2人以上の開発者にコードをより良いものに出来ないか話し合う
 - 直感的に分かりやすい記述に出来ないか？
 - より短く簡潔にできないか？
 - バグを引き起こしそうな箇所はないか？
- コードレビューの利点
 - 他人の記述したコードを見ることで自分の担当部分の挙動も理解できる
 - 他人の良いところは学び、自分の悪いところは直すことができる

ペアプログラミング

- 2人で1つのマシンを使ってコーディングする手法
 - 1人はコードを書く(ドライバ)
 - 1人は助言を行う(ナビゲータ)
- 初心者同士 or 初心者と上級者のペアが望ましい
- ペアプログラミングの利点
 - プログラムの正確性が48%向上(=バグ取りにかかる時間が短縮できるので、2人で別々に作業するよりも実は早く完成する)

例) ペアプログラミング



テスト駆動開発

- TDD; Test-driven development
 - 実装を始める前にテストコード(ソフトが正しく動いているか確かめるコード)を書く
- 利点
 1. 先にテストを書くことで実装のゴールが明確になり、効率化に繋がる
 2. 将来実装を書き換えたときに、テストを動かせばバグがあるか分かるようになる

例) C言語によるテスト駆動開発

- 階乗を返す以下の関数が正しく動作するかテストする場合を考える
 - `int kaijou(int n);`
- 手順1：テストコードを書く(次項参照)

例) テストコード(kaijou_test.c)

```
1 #include <stdio.h>
2 #include "kaijou.h"
3
4 int main(void) {
5     // 1の階乗が正しく動くかテスト
6     if (kaijou(1) == 1) {
7         printf("1の階乗：テスト成功\n");
8     } else {
9         printf("1の階乗：テスト失敗\n");
10    }
11
12    // 3の階乗が正しく動くかテスト
13    if (kaijou(3) == 6) {
14        printf("3の階乗：テスト成功\n");
15    } else {
16        printf("3の階乗：テスト失敗\n");
17    }
18
19    return 0;
20 }
```


例) C言語によるテスト駆動開発

- 手順2：空の関数を作成(テストが失敗するようなコードをわざと作成)

kaijou.h

```
1 #ifndef _INCLUDE_KAIJOU_H_
2 #define _INCLUDE_KAIJOU_H_
3 // 関数 kaijou のプロトタイプ宣言
4 int kaijou(int);
5 #endif // _INCLUDE_KAIJOU_H_
-
```

kaijou.c

```
1 #include <stdio.h>
2 #include "kaijou.h"
3
4 int kaijou(n) {
5     return 0;
6 }
```

例) C言語によるテスト駆動開発

- 手順3：テストコードをコンパイル/実行しテストが失敗していることを確かめる

```
$ cc test_kaijou.c kaijou.c  
$ ./a.out
```

```
1の階乗：テスト失敗  
3の階乗：テスト失敗
```

例) C言語によるテスト駆動開発

- 手順4：関数を正しく実装する

kaijou.c

```
1 #include <stdio.h>
2 #include "kaijou.h"
3
4 int kaijou(n) {
5     int result = 1;
6     int i;
7     for(i=1; i <= n; i++) {
8         result *= i;
9     }
10    return result;
11 }
```

例) C言語によるテスト駆動開発

- 手順5：テストが正しく実行されることを確かめる

```
$ cc test_kaijou.c kaijou.c  
$ ./a.out
```

1の階乗：テスト成功

3の階乗：テスト成功

本日の内容

- 講義
 1. ソフトウェア開発の現状と問題
 2. ソフトウェア開発工程・方法論
 3. ソフトウェア開発の効率化
 - コーディングの効率化
 - 共同開発の効率化
- 演習
 - ペアプログラミング
 - Gitを用いた共同開発

共同開発の難しさ

- 共同開発：複数人による開発
 - 一緒にコーディングする(ペアプログラミング)
→ 3人以上のプロジェクトでは不可
 - 別々にコーディングしてしてあとで結合
→ しっかり議論して作業分担しても結合時に
バグが発生しやすい

1. 作業分担した箇所の個々のバグ
2. 結合時のバグ

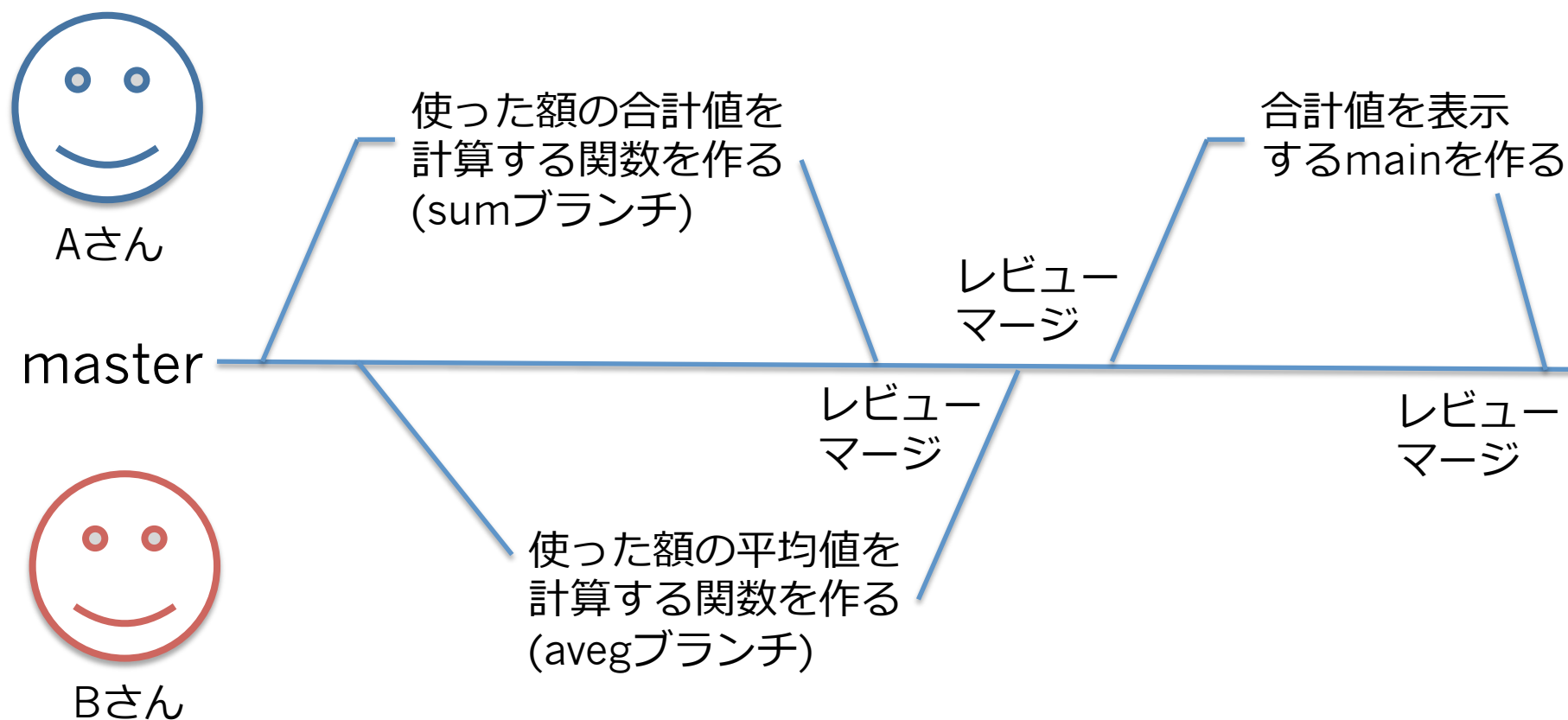
別々にコーディングしてもバグが発生しにくい
方法や工夫が必要

効率的な共同開発

- 工夫1：共同開発しやすい設計にする
 - (C言語の場合)関数をうまくわけると
 - 1つの関数を1人で担当し、複数の関数を同時開発する
 - オブジェクト指向に基づく設計にする(4/5回)
- 工夫2：共同開発の環境整備
 - Gitブランチを用いた共同開発
 - プルリクエストによるコードレビューとマージの効率化

ブランチを用いた共同開発

- 例) ブランチを用いた家計簿の共同開発



Gitを用いた共同開発(流れ)

1. 共同で使うリポジトリをGitHub上に作る
2. 2人がそれぞれのマシンにgit clone
3. 作業を始める前にmasterブランチをGitHubからpull
4. 新しいブランチを作り新しい機能を作る
5. 完成したらGitHubにブランチをpush
\$ git push origin ブランチ名
6. GitHub上でプルリクエストを作って共同開発者と議論
7. お互いが納得したらマージボタンを押す
8. 3に戻る

プルリクエスト

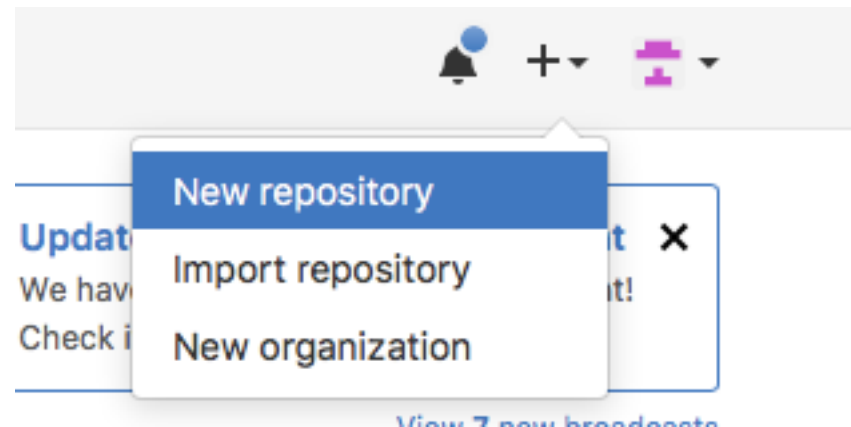
- プルリクエストとは？
 - コードレビューとマージを効率的に行う仕組み
 - GitHub上で議論し、お互いが納得したら
「マージ」ボタンを押す
→ 演習で詳しく確認

演習概要

- 代表的なソフトウェア開発技法を用いて簡単な家計簿を共同開発する
- 演習を通じてペアプログラミング、コードレビュー、Gitを用いた共同開発フローを修得する
- 2人組で作成：AとBに分かれること

演習0(A) : GitHubでのリポジトリ作成


- GitHubにログインし右上の+マークから「New repository」を選択する



Create a new repository

A repository contains all the files for your project, including the revision history.

Owner

 NIT-IBARAKI-Applied-Programming ▾

Repository name

/ kakeibo_ishigaki ✓

↑のように指定 kakeibo_stxxdxxと入力

Description (optional)

☐  **Public**
Anyone can see this repository. You choose who can commit.

☒  **Private** private
You choose who can commit to this repository.

☒ **Initialize this repository with a README** ここをチェック ✓
This will let you immediately clone the repository. ing an existing repository.

Add .gitignore: C ▾

Add a license: None ▾



Add .gitignore Cを選択

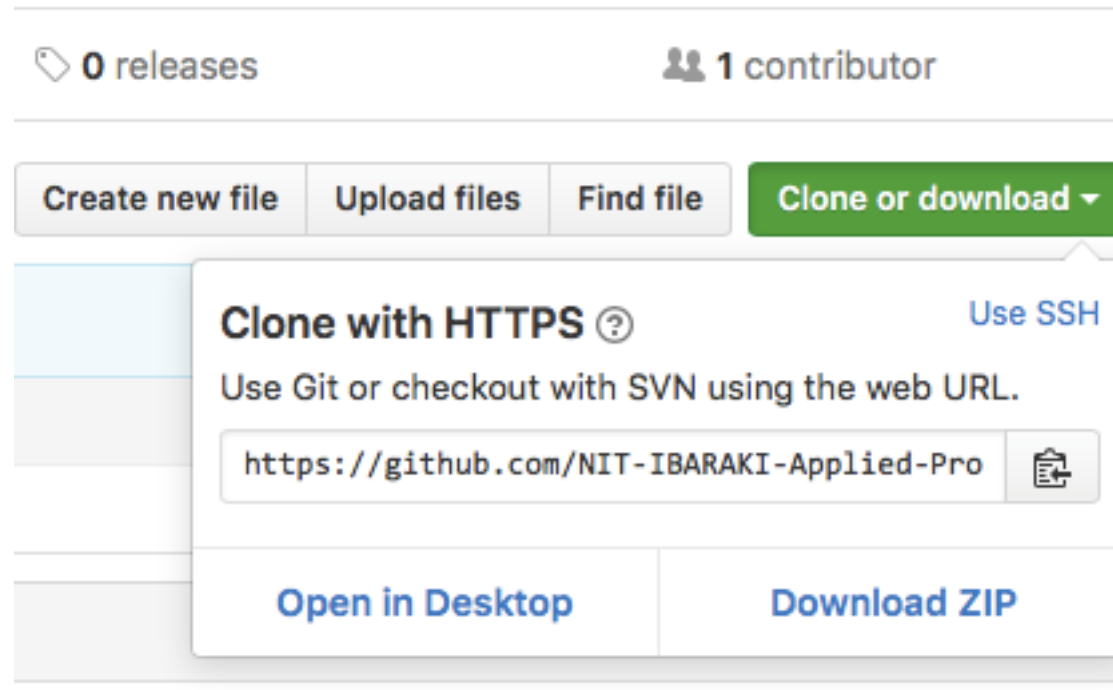
Create repository

最後にここをクリック

リポジトリのトップページが表示される。
次に共同開発者をリポジトリに追加する。
Setting→Collaboratorsと進み、共同開発者の
メールアドレス(GitHubに登録しているもの)を入力し
[Add collaborator]をクリック。

The screenshot shows the GitHub repository settings page. The top navigation bar includes links for Code, Issues (0), Pull requests (0), Projects (0), Wiki, Pulse, Graphs, and Settings. The left sidebar lists various settings: Options, Collaborators (selected), Branches, Webhooks, Integrations & services, and Deploy keys. The main content area is titled 'Collaborators' and includes a sub-header 'Push access to the repository'. Below this, a message states: 'This repository doesn't have any collaborators yet. Use the form below to add a collaborator.' A search bar is provided with the instruction: 'Search by username, full name or email address'. A note below the search bar says: 'You'll only be able to find a GitHub user by their email address if they've chosen to list it publicly. Otherwise, use their username instead.' The search bar is highlighted with an orange box containing the text 'ここに共同開発者のメールアドレス'. To the right of the search bar is an 'Add collaborator' button.

- 緑色のClone or downloadをクリックするとリポジトリURLが表示されるのでコピー



端末を開き、以下のコマンドでGitHubからリポジトリを各自(A&B)クローンしリポジトリに移動

```
$ git clone リポジトリURL  
$ cd kakeibo_stxxdyy
```

演習1(A, B)：ペアプログラミング

- Aのマシンを使い、2人でペアプログラミングしながら、以下の要件を満たすプログラムを作成せよ
 - ファイル名「main.c」からのCプログラムを用意
 - main()関数内に要素数5のint型配列 payments を定義
 - 配列の各要素にscanf()関数で整数を読み込む
 - ここまで2人で作ったら、以下のコマンドでGitHubにpushする
\$ git push origin master

演習1-1(A, B) : ペアプログラミング

- Aのマシンを使い、2人でペアプログラミングしながら、以下の要件を満たすプログラムを作成せよ
 - initialという名前のブランチを作る

```
$ git branch initial
```

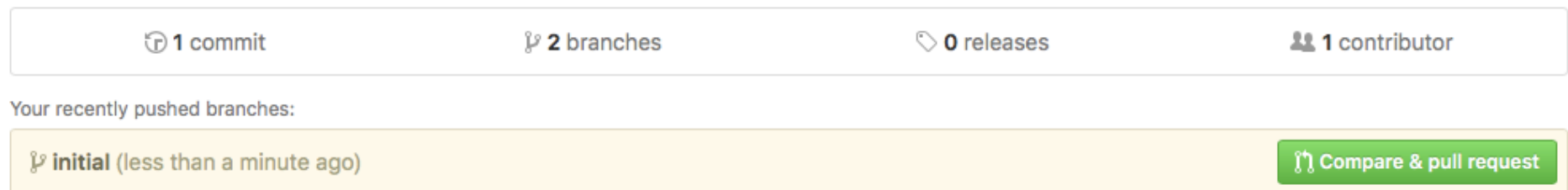
```
$ git checkout initial
```
 - ファイル名「main.c」のCプログラムを作成
 - main()関数内に要素数5のint型配列paymentsを定義
 - 配列の各要素にscanf()関数で整数を読み込む

演習1-2(A, B) : ペアプログラミング

- 続き
 - ここまで2人で作ったら、add/commitして以下のコマンドでGitHubにpushする
\$ git push origin initial

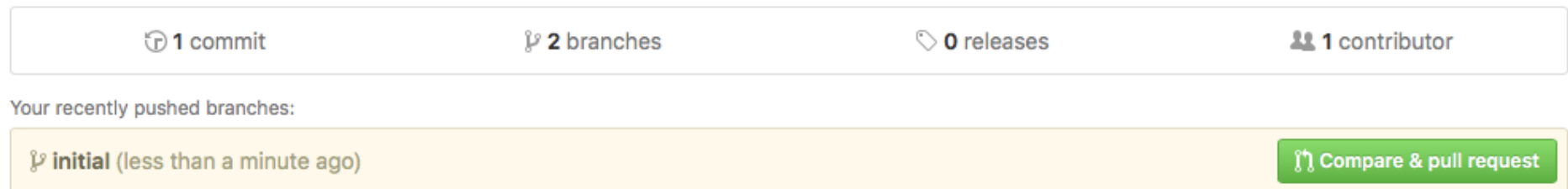
演習1-2(A, B)：プルリクの練習

- FirefoxでGitHubにアクセスし、リポジトリのトップページを開く
 - 以下のようにプルリクエストを作るボタンが
できているので、[Compare&pull request]を
クリック



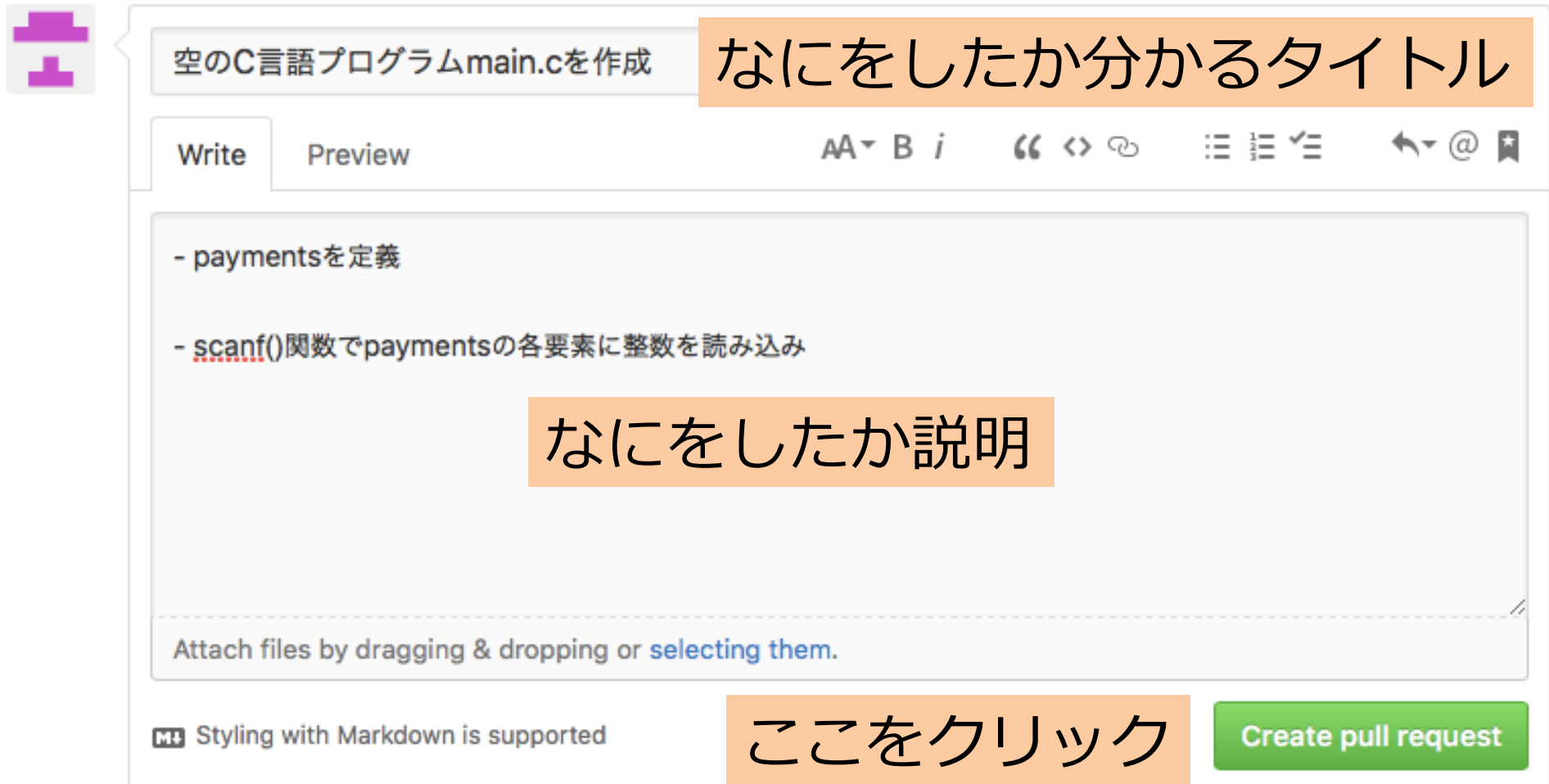
演習1-2(A, B)：プルリクの練習

- FirefoxでGitHubにアクセスし、リポジトリのトップページを開く
 - 以下のようにプルリクエストを作るボタンが
できているので、[Compare&pull request]をクリック



演習1-2(A, B)：プルリクの練習

- 以下のようにプルリクエストを作成



The screenshot shows the GitHub pull request creation interface. A purple icon is on the left. The title field contains the text "空のC言語プログラムmain.cを作成" (Create an empty C program main.c) and is annotated with "なにをしたか分かるタイトル" (Title that explains what was done). The description field contains two bullet points: "- paymentsを定義" and "- scanf()関数でpaymentsの各要素に整数を読み込み", with the second point annotated with "なにをしたか説明" (Explain what was done). At the bottom, there is a note "Attach files by dragging & dropping or selecting them.", a footer "Styling with Markdown is supported", and a green button labeled "Create pull request". An orange box with the text "ここをクリック" (Click here) points to the "Create pull request" button.

空のC言語プログラムmain.cを作成

なにをしたか分かるタイトル

Write Preview

AA B i “ <> 🔗 ☰ ☷ ✓ ☰ ↩ @ ★

- paymentsを定義
- `scanf()`関数でpaymentsの各要素に整数を読み込み

なにをしたか説明

Attach files by dragging & dropping or [selecting them](#).

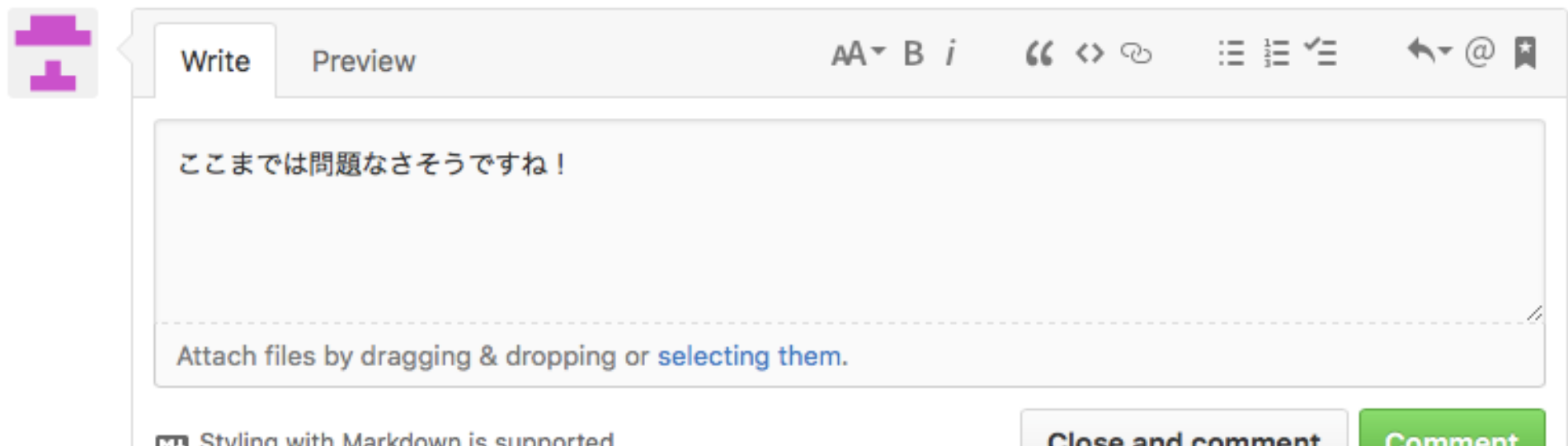
M Styling with Markdown is supported

ここをクリック

Create pull request

演習1-2(A, B)：プルリクの練習

- このページで共同開発者と議論することが出来る。試しになにかコメントを追加してみよう
- 上の方の[file changed]ではファイルの変更履歴を見ることが出来ます。



The screenshot shows a GitHub comment box interface. On the left is a purple icon of a person. The box has two tabs: 'Write' (active) and 'Preview'. The 'Write' tab contains a rich text editor with a toolbar at the top showing options for text formatting (bold, italic, underline, strikethrough), lists, links, and a mention button. The text area contains the Japanese text 'ここまでは問題なさそうですね！'. Below the text area is a dashed line and a prompt: 'Attach files by dragging & dropping or selecting them.' At the bottom of the box, there is a small text 'Styling with Markdown is supported' and two buttons: 'Close and comment' and a green 'Comment' button.

Write Preview

AA B i “ <> 🔗 ☰ ☷ ✓ ☷ ↩ @ 📌

ここまでは問題なさそうですね！

Attach files by dragging & dropping or selecting them.

Styling with Markdown is supported

Close and comment Comment

演習1-3(A, B) : マージ

- 問題なければ以下のボタン(Merge pull request)でmasterブランチにマージする



This branch has no conflicts with the base branch

Merging can be performed automatically.

Merge pull request



You can also [open this in GitHub Desktop](#) or view [command line instructions](#).

演習2(A)：テストコード

- Aは合計値を出す関数sumの実装を担当する。まずは、テストコードから作成する。
- 以下のコマンドでmasterブランチに移動し、GitHubから最新の内容をpull。その後、sumブランチを作り、テストコードsum_test.cを作成せよ。

```
$ git checkout master  
$ git pull  
$ git branch sum  
$ git checkout sum
```

合計値を出す関数sumは別ファイル(average.h/average.c)を用いて記述すること

関数の形式は以下とする

```
int sum(int payments[]);
```

※この段階ではsumの中身は作らず、テストが失敗することを確認せよ。

演習2(B)：テストコード

- Bは平均値を出す関数averageの実装を担当する。まずは、テストコード(average_test.c)から作成する。
- 以下のコマンドで以下のコマンドでmasterブランチに移動し、GitHubから最新の内容をpull。新たにaverageブランチを作る。
\$ git checkout master
\$ git pull
\$ git branch average
\$ git checkout average

平均値を出す関数averageは別ファイル(average.h/average.c)を用いて記述すること

関数の形式は以下とする
int average(int payments[]);

※この段階ではaverageの中身は空にしてテストが失敗することを確認めよ。

演習3(A)：平均値関数の実装

- `sum()`関数を実装しテストが通ることを確認せよ

演習3(B)：合計値関数の実装

- average()関数を実装しテストが通ることを確認せよ

演習4-1(A,B) : push

- GitHubにブランチをpushせよ
 - Aさん
\$ git push origin sum
 - Bさん
\$ git push origin average

演習4-2(A,B)：2回目のプルリク

- GitHubでプルリクエストを作成せよ
- プルリクを用いてお互いコードレビューし、改善点や良い点などをコメントとして投稿せよ
- 問題なければmerge、問題があれば改善した内容を再度pushしてもらいコードレビューを繰り返す

演習5(A, B)：2人の作業を統合

- 2人の作ったブランチ(sum, average)をmasterにマージしたら、以下のコマンドでマージした部分を自身のマシンの取り込む
\$ git checkout master
\$ git pull
- Aさん、Bさんどちらかのマシンでペアプログラミングし、sum()/average()関数をmain()関数から呼び出し、平均、合計をprintf()関数で表示しなさい。

演習6：追加機能の議論

- 家計簿に追加すべき機能について、2人でアイデアを出し合いなさい。
 - 例えば、「支出」を入力して、支出額の平均や合計を出せるようにするなど
- A,Bがどのように分担して関数を作るか決めなさい

演習7：追加機能の実装

- 以下のコマンドでmasterに戻り、GitHub上からmasterブランチの差分を入手しなさい

```
$ git checkout master  
$ git pull
```

- pull出来たら、以下のコマンドで新たなブランチを作り、演習2から演習6までの内容を繰り返し新機能を共同開発しなさい

```
$ git branch 新しいブランチ名  
$ git checkout 新しいブランチ名
```


次回

- 共同開発を進めるための
ソフトウェア設計技法を学習します
 - オブジェクト指向プログラミング
 - デザインパターン
- 演習室は放課後も使えるので活用してください！