

# CSD Retrospective

認定スクラムデベロッパー研修を  
受けてみてのふりかえり

2018.9

田地 将也  
(@otajisan)

# 概要

- ・この度機会をいただき、  
CSD (Certified Scrum Developer) の研修を  
受講しました
- ・研修で学んだことをふりかえりつつ、  
皆さんに共有しようと思います（私のYWTを元に説明します）
- ・なお、トレーナーや研修を開催する会社によって、  
研修のメニューは異なる可能性がありますので、  
研修の受講内容はご参考程度、と捉えてくださいmm
- ・今回私はOdd-eさんの研修を受講しました

# アジェンダ

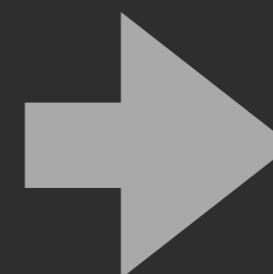
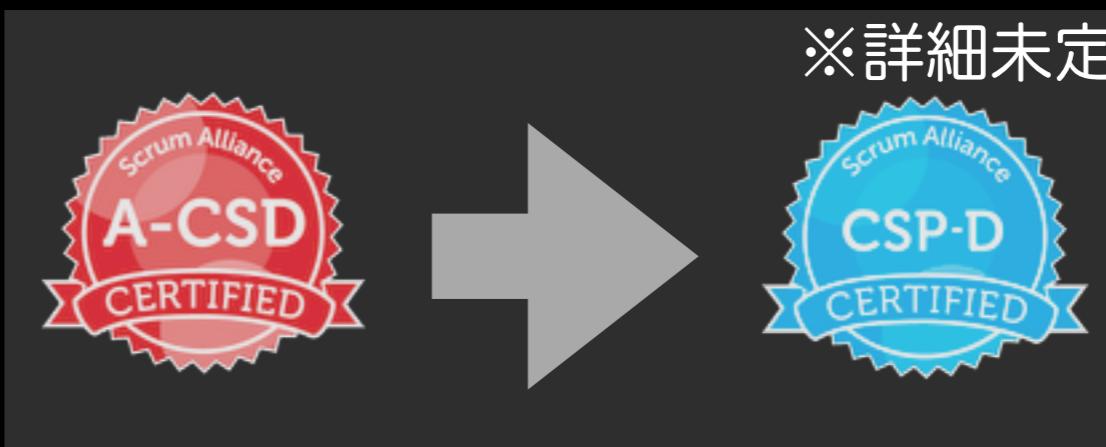
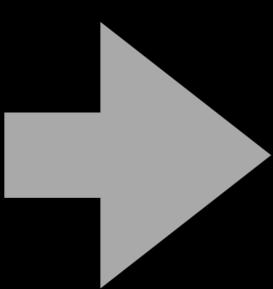
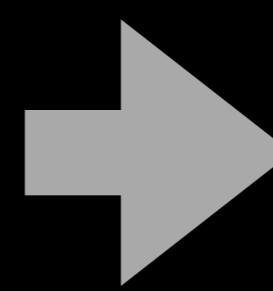
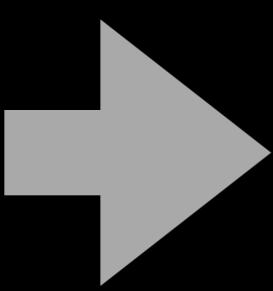
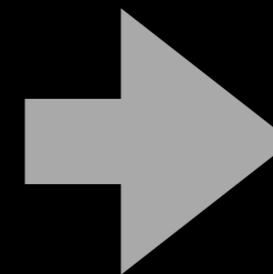
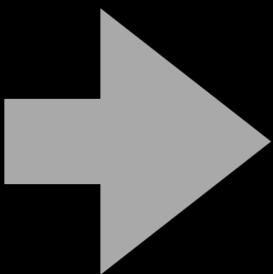
- これ、何？（CSDとは）
- 研修の大まかな流れ
- 研修で学んだこと
  - 1. TDD
  - 2. リファクタリング
  - 3. レガシーコードとの戦い方
  - 4. コンポーネントチーム vs フィーチャーチーム
- まとめ

# これ、何？（CSDとは）

- Scrum Allianceの認定資格の一つです
- 特に、スクラムにおける「開発チーム」の一員として、あるべき姿を学べます



# ちなみに、認定制度は 最近ややこしくなった



※詳細未定

※2018.9時点

# 研修の大まかな流れ

- 研修の期間は5日間
- この5日間を1スプリントと見立て、  
2チームに分かれて1つのプロダクトを開発しました
- 初日にお題が提示されて、  
5日の夕方（スプリントレビュー）までに  
フィーチャーを完成させ、PO（研修トレーナー）に  
お披露目する、という流れ

1日目

リファインメント  
スプリントプランニング

1～5日目

スプリント実行  
(開発)

5日目

スプリントレビュー

# 研修の大まかな流れ

- ・ 開発期間中、トレーナーが状況を見て、  
そのとき参加者に必要な講義を間に挟んでくれます
- ・ 私たちの受講時は、ペアプロは割とみんな  
うまくやっていたので講義を間引かれ、  
逆にTDDやリファクタリングの講義は  
厚めに実施していただいた印象です
- ・ 座学 -> 開発 -> 座学 -> 開発 ・ ・ ・ と繰り返す感じ

1日目

リファインメント  
スプリントプランニング

1～5日目

スプリント実行  
(開発)

5日目

スプリントレビュー

# 研修の大まかな流れ

- 初期のワーキングアグリーメントにて、  
幾つかのチームの決まりごとが定められている
  - 全てのコードをTDDで書くこと
  - 開発は全てペアプロで行うこと
  - 全てのコードに責任を持つこと
  - etc.
- 今回の研修での利用言語はJavaで、  
途中まで作られた既存システムにフィーチャーを追加する、  
という内容(研修によってはRubyなどもあるらしい)

# 研修で学んだこと

- 1. TDD
- 2. リファクタリング
- 3. レガシーコードとの戦い方
- 4. コンポーネントチーム vs フィーチャーチーム

研修で学んだこと

1. TDD

# TDD

- ・ テストファーストな開発手法
- ・ 「動作するキレイなコード」を書くことが目的
- ・ とにかく今回の研修では一番コレの重要性と、自分の慣れさに気付かされた・・・

# 従来の開発手法とTDD

従来



キレイに設計 -> 動かす

# 従来の開発手法とTDD

TDD



動かす -> キレイに書く

# では、なぜTDDなのか？

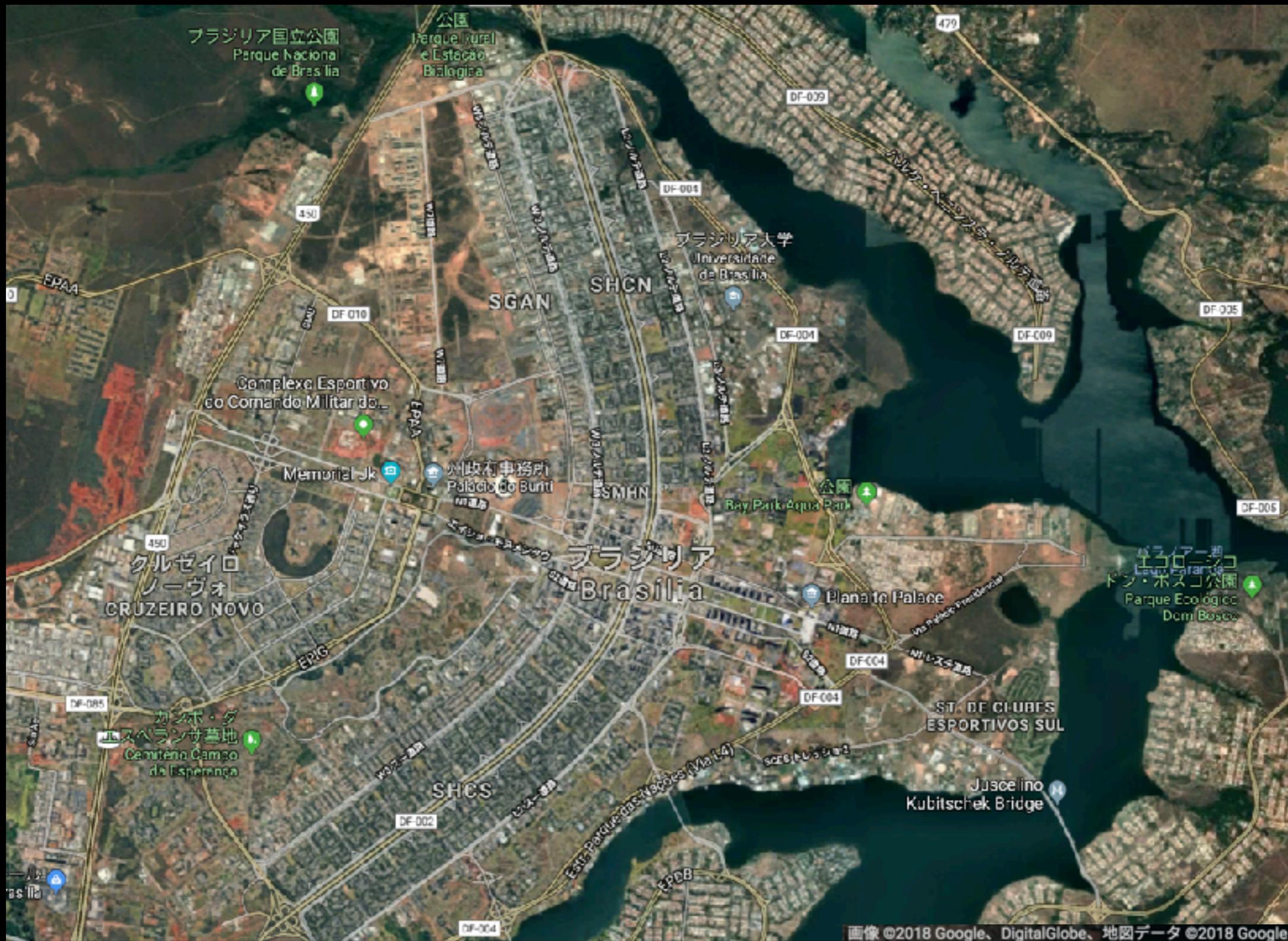
従来



TDD



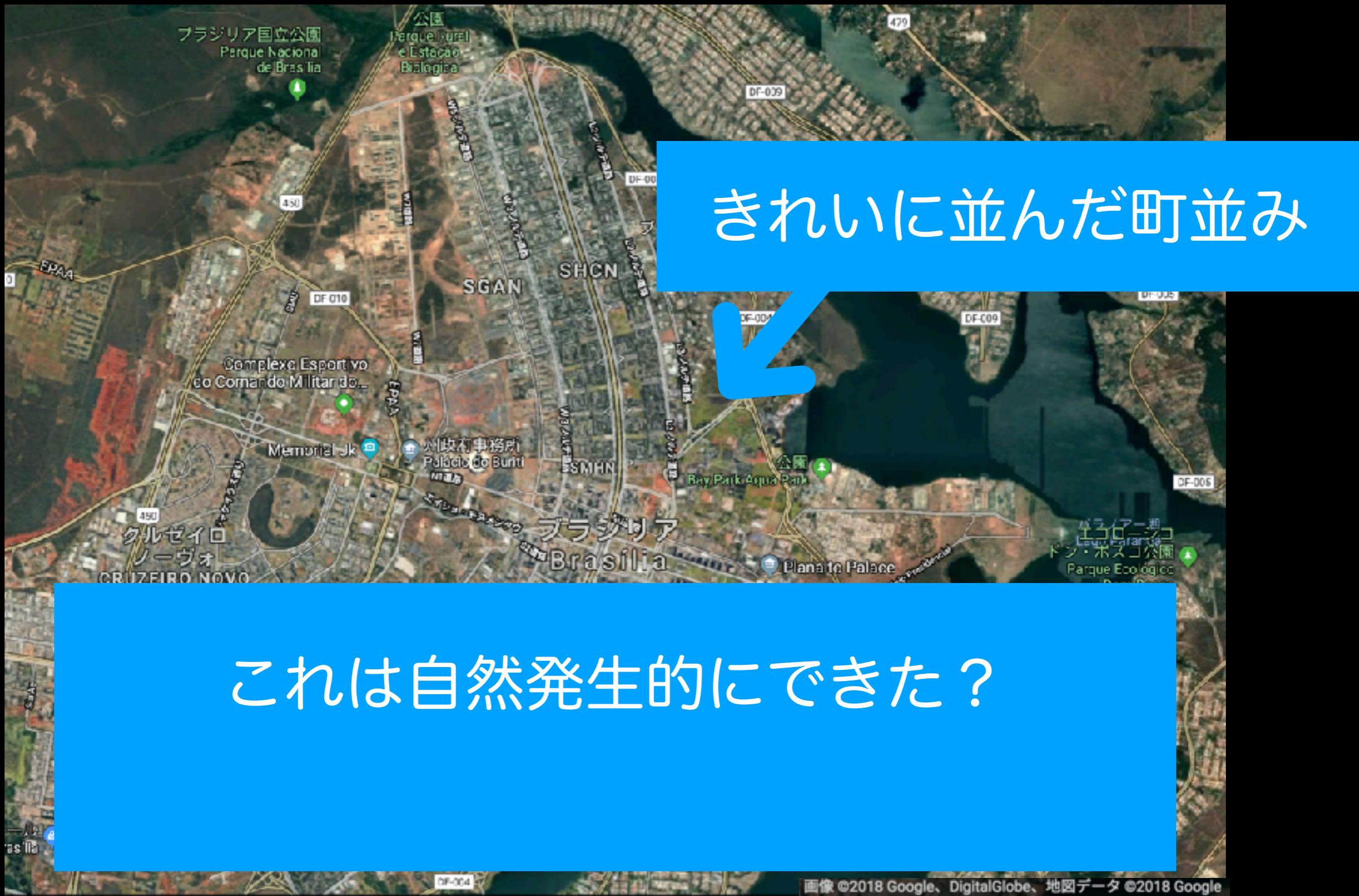
# ブラジリアの話



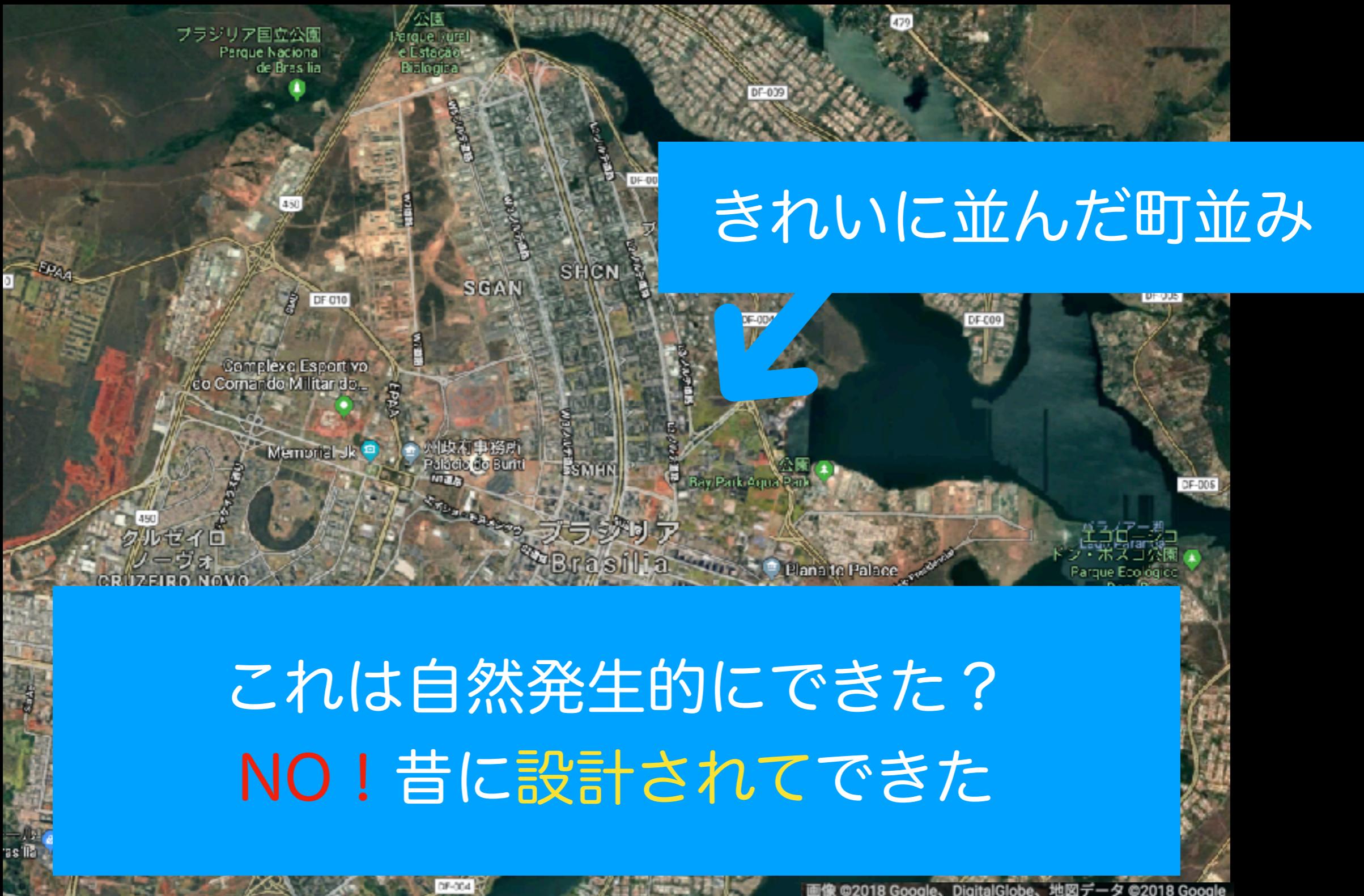
# ブラジリアの話



# ブラジリアの話



# ブラジリアの話



# ブラジリアの話

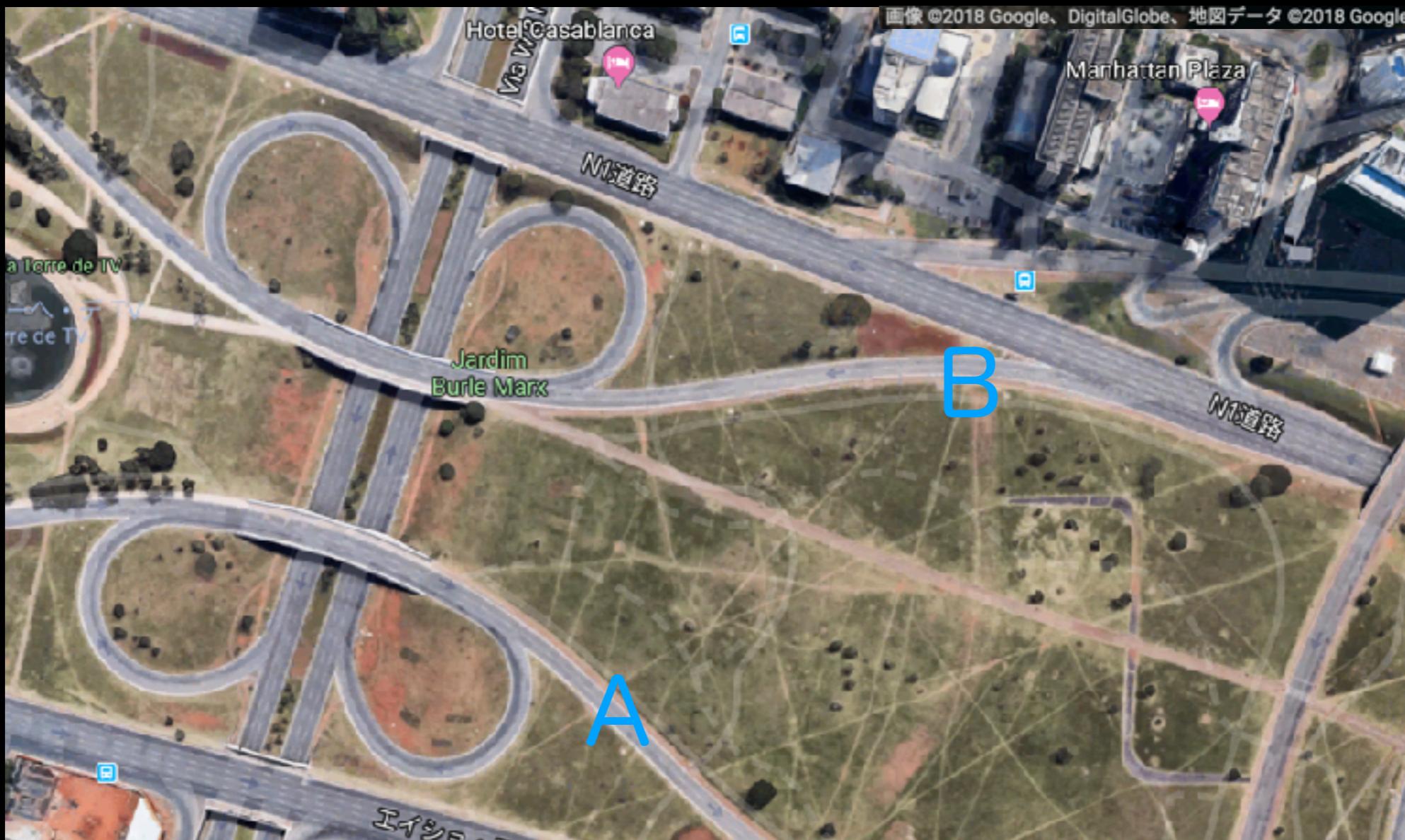


実際の人々の動線は？

# 実際の人々の動線は？



# 実際の人々の動線は？



地点A -> Bに移動する、  
というケースを考える

# 実際の人々の動線は？



# 実際の人々の動線は？



# 実際の人々の動線は？



足跡を見ると  
「こう行きたい！」感満載

つまり、当初の意図通りに  
全てうまくいくわけではない

それって「設計力」の  
問題なのでは？

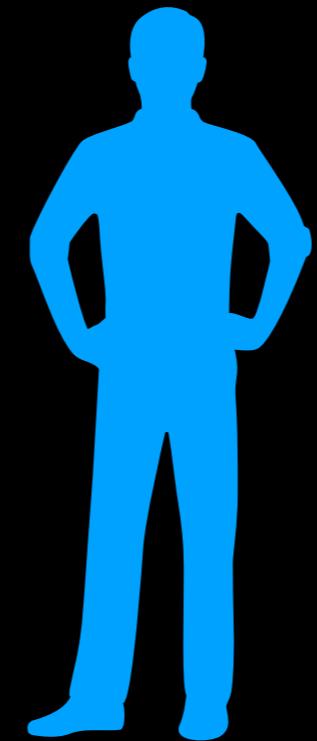
「設計力」って  
何ですか？

「経験」のことですか？

# 「経験」に見る「設計力」

俺はアーキテクトだ！

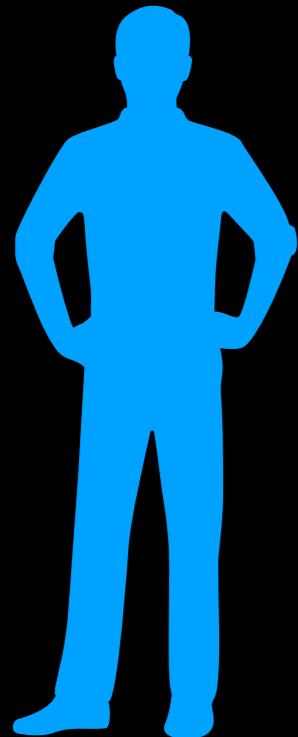
開発経験豊富で、  
なんでも設計できるぜ！



Architect

# 「経験」に見る「設計力」

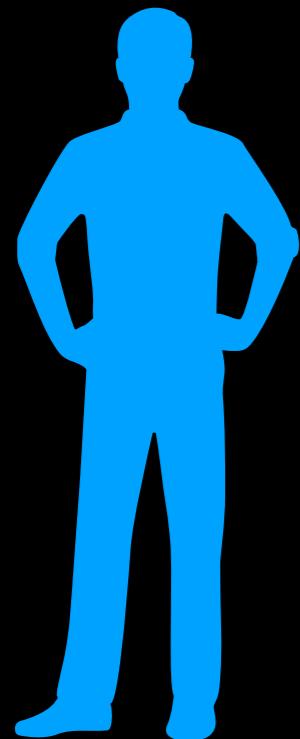
これから「新規システム」を作るとして、  
アーキテクトがそのシステムを設計するケースを考える



System Design

# 「経験」に見る「設計力」

これから「新規システム」を作るとして、  
アーキテクトがそのシステムを設計するケースを考える

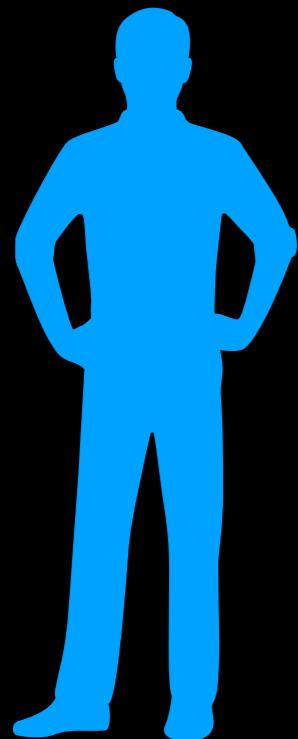


System Design

「経験」というのは  
「以前作ったことがある」ということである

# 「経験」に見る「設計力」

これから「新規システム」を作るとして、アーキテクトがそのシステムを設計するケースを考える

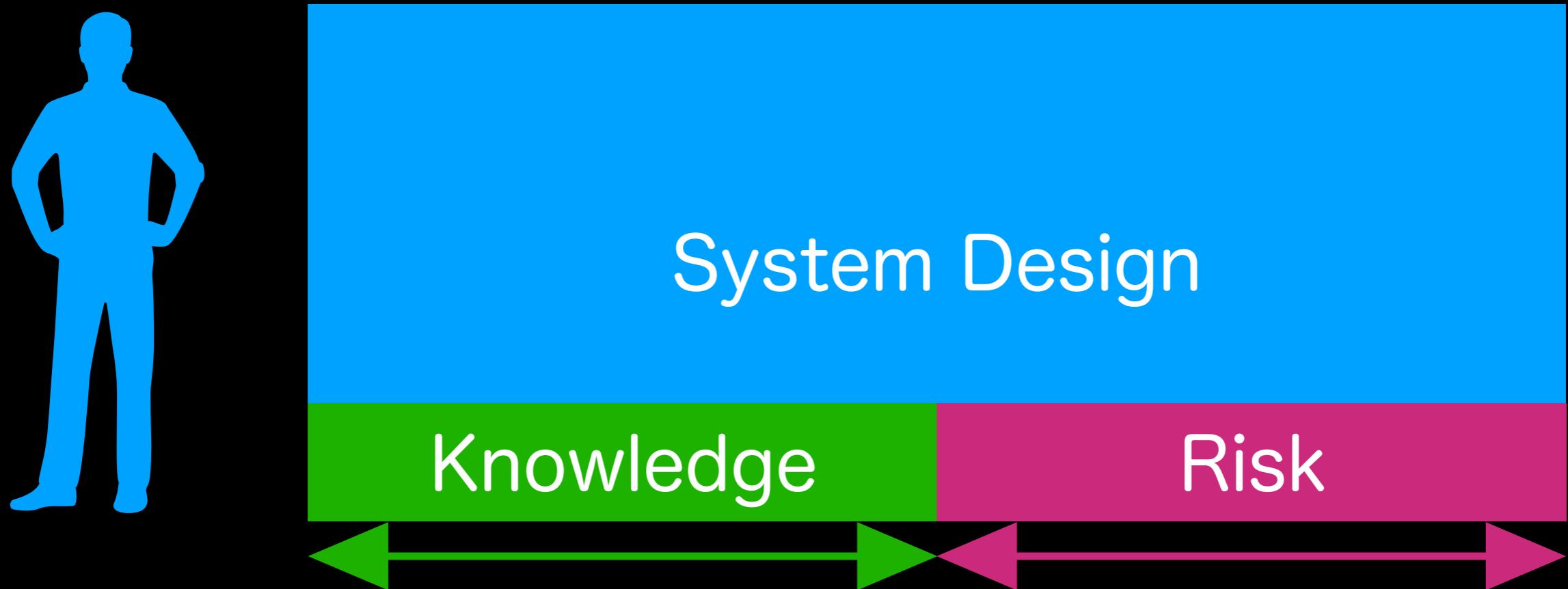


System Design

しかし、「これから作るもの」は「以前の失敗を踏襲して」検討するケースが多い  
(つまり、未知のものを含んでいる)

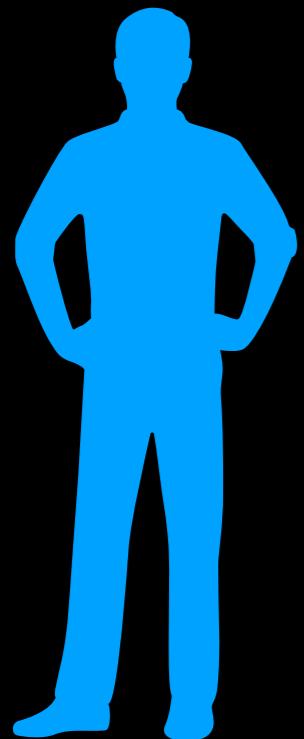
# 「経験」に見る「設計力」

これから「新規システム」を作るとして、アーキテクトがそのシステムを設計するケースを考える



なので、結局「経験していないモノ」を考えることになり、それはリスクを含む

# 「経験」に見る「設計力」



アーキテクチャは最初に検討しない。  
実装が進んだら都度設計と合わせて検討する  
(先に決めると制約となり、変更しづらい)

なので、地に足の着いたところから、  
ということでTDDが注目されている  
従来~~X~~



TDD ✓



# TDDの進め方(テスト)

- ・リファインメント時に、  
ユーザーストーリーからテストシナリオを作成し、  
POと合意する
- ・ツールを使ってテストコードを生成する
- ・今回の研修では受け入れテストのツールとして  
cucumberを利用しました

# TDDの進め方(テスト)

プロダクトバックログを分解し、  
以下のシナリオを明確化します

Scenario:

Given

When

Then

# TDDの進め方(テスト)

例えばこんな感じ  
(Web上で回答可能な試験問題を考える)

**Scenario:** 問題に正答すると次の問題が出題される

**Given** ユーザtajiとしてログインしている

**When** 正解となる問題を選択する

**And** “回答”ボタンをクリックする

**Then** 次の問題が表示される

# TDDの進め方(テスト)

例えばこんな感じ  
(Web上で回答可能な試験問題を考える)

**Scenario:** 問題に正答すると次の問題が出題される

**Given** ユーザtajiとしてログインしている

**When** 正解となる問題を選択する

**And** “回答”ボタンをクリックする

**Then** 次の問題が表示される

これをPOと合意しておく、というのが重要

# TDDの進め方(テスト)

また、このシナリオの中には  
幾つかのドメインが含まれている

Scenario: **問題**に正答すると次の問題が出題される

Given ユーザtajiとして**ログイン**している

When 正解となる問題を選択する

And “回答”ボタンをタップする

Then 次の問題が表示される

例えば、このあたりの単語が**ドメイン**

# TDDの進め方(テスト)

例えば“回答”ボタンを例に取ると、  
このシステムでは「次へ」ボタンでなく、  
「回答」ボタンを利用してしていることが分かる

Scenario: **問題**に正答すると次の問題が出題される

Given ユーザtajiとして**ログイン**している

When 正解となる問題を選択する

And “回答”ボタンをタリックする

Then 次の問題が表示される

例えば、このあたりの単語がドメイン

# TDDの進め方(テスト)

これによって、PO、ステークホルダーと  
共通の言葉を使ってシステムを開発できる  
(※詳しくはドメイン駆動設計を(ry))

Scenario: **問題**に正答すると次の問題が出題される

Given ユーザtajiとして**ログイン**している

When 正解となる問題を選択する

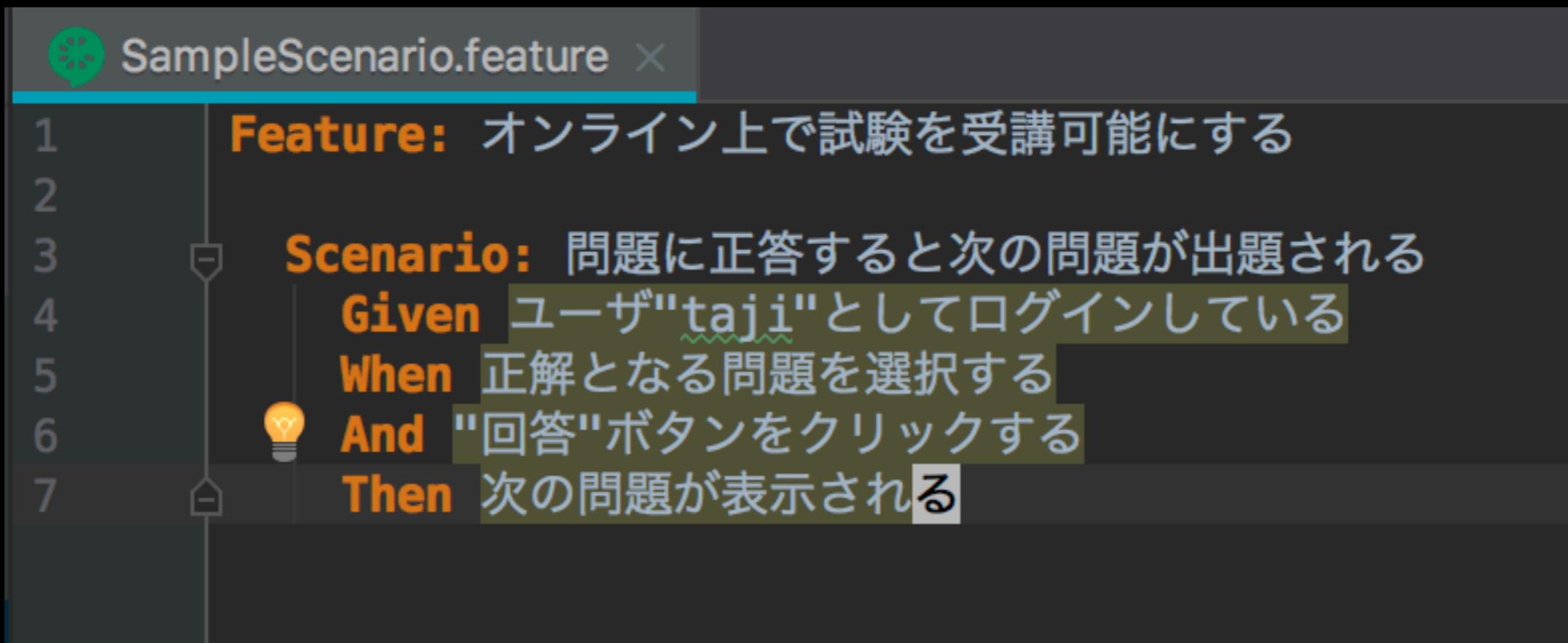
And “回答”ボタンをタリックする

Then 次の問題が表示される

例えば、このあたりの単語が**ドメイン**

# TDDの進め方(テスト)

そして、ここで決めたシナリオを  
cucumberの.featureファイルに書く



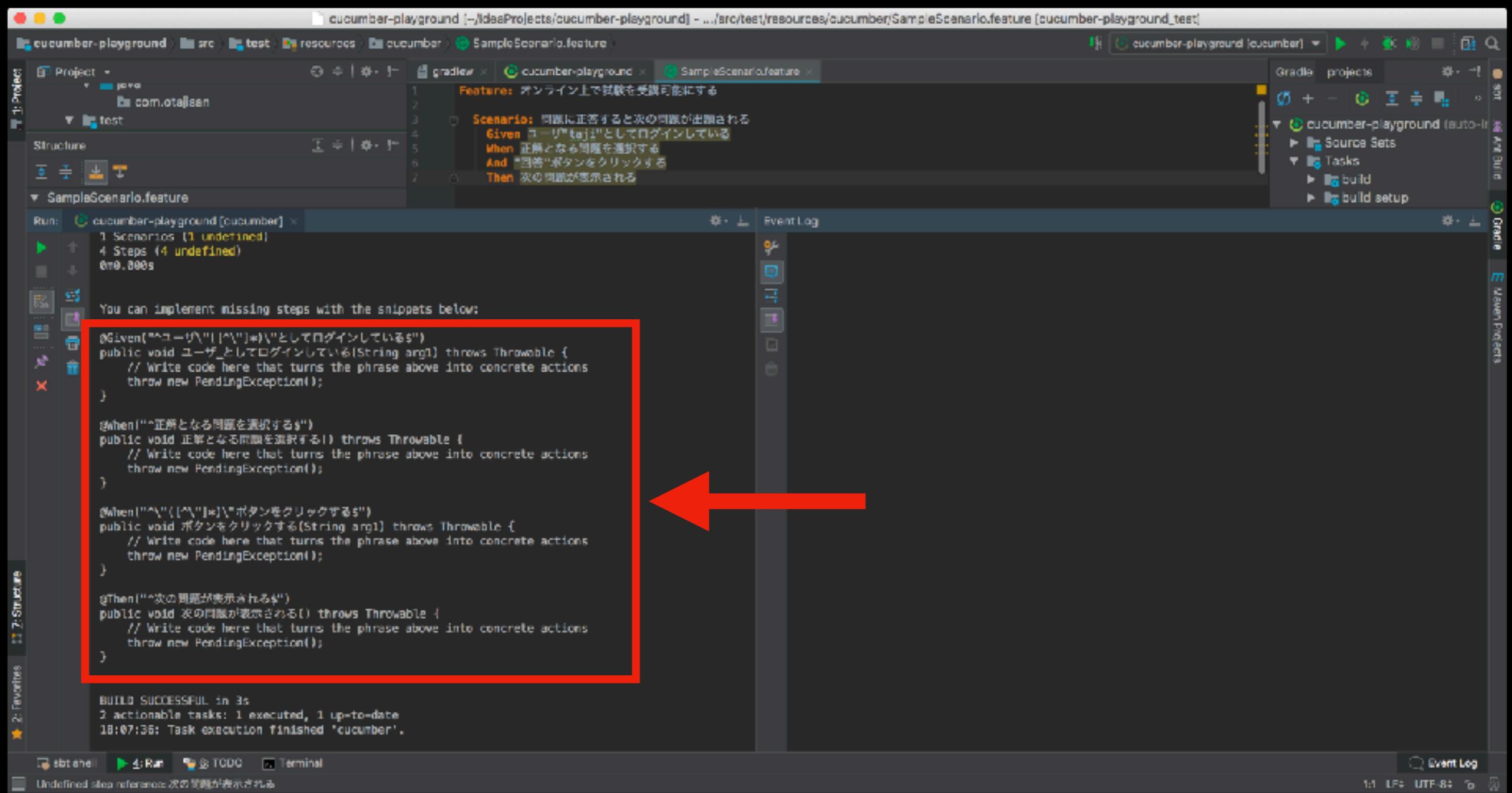
The screenshot shows a code editor window with a dark theme. The title bar says "SampleScenario.feature". The content of the file is as follows:

```
1 Feature: オンライン上で試験を受講可能にする
2
3   Scenario: 問題に正答すると次の問題が出題される
4     Given ユーザ"user"としてログインしている
5     When 正解となる問題を選択する
6     And "回答"ボタンをクリックする
7     Then 次の問題が表示される
```

The "Feature:" line is bolded. The "Scenario:" line has a small orange icon to its left. The "Given", "When", "And", and "Then" steps each have their own icons: a user icon for "Given", a question mark icon for "When", a lightbulb icon for "And", and a checkmark icon for "Then".

# TDDの進め方(テスト)

.featureからテストコードのスケルトンを吐き出す



The screenshot shows an IDE interface with a Cucumber project named "cucumber-playground". The "SampleScenario.feature" file is open, displaying a single scenario:

```
Feature: オンライン上で試験を受講可能にする
  Scenario: 問題に正答すると次の問題が表示される
    Given ユーザー"taji"としてログインしている
    When 正解となる問題を選択する
    And "回答"ボタンをクリックする
    Then 次の問題が表示される
```

Below the feature file, the "Run" tab shows:

- 1 Scenarios (1 undefined)
- 4 Steps (4 undefined)
- 0m0.000s

A message indicates missing steps can be implemented with snippets:

```
You can implement missing steps with the snippets below:
```

```
@Given("^ユーザー\"([^\"]*)\"としてログインしている$")
public void ユーザーとしてログインしている(String arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^正解となる問題を選択する$")
public void 正解となる問題を選択する() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@When("^\"([^\"]*)\"ボタンをクリックする$")
public void ボタンをクリックする(String arg1) throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}

@Then("^次の問題が表示される$")
public void 次の問題が表示される() throws Throwable {
    // Write code here that turns the phrase above into concrete actions
    throw new PendingException();
}
```

A red box highlights the generated Java code, and a large red arrow points to it from the bottom left.

At the bottom of the screen, the terminal output shows:

```
BUILD SUCCESSFUL in 3s
2 actionable tasks: 1 executed, 1 up-to-date
18:07:36: Task execution finished 'cucumber'.
```

# TDDの進め方(テスト)

これを、Step(受け入れテストのコード)にコピペ

```
SampleScenarioStep.java ×
1 package com.otajisan.steps;
2
3 import cucumber.api.PendingException;
4 import cucumber.api.java.en.*;
5
6 import static org.junit.Assert.*;
7
8 public class SampleScenarioStep {
9     @Given("^ユーザ\"([^\"]*)\"としてログインしている$")
10    public void ユーザ_としてログインしている(String arg1) throws Throwable {
11        // Write code here that turns the phrase above into concrete actions
12        throw new PendingException();
13    }
14
15    @When("^正解となる問題を選択する$")
16    public void 正解となる問題を選択する() throws Throwable {
17        // Write code here that turns the phrase above into concrete actions
18        throw new PendingException();
19    }
20
21    @When("^\"([^\"]*)\"ボタンをクリックする$")
22    public void ボタンをクリックする(String arg1) throws Throwable {
23        // Write code here that turns the phrase above into concrete actions
24        throw new PendingException();
25    }
26
27    @Then("^次の問題が表示される$")
28    public void 次の問題が表示される() throws Throwable {
29        // Write code here that turns the phrase above into concrete actions
30        throw new PendingException();
31    }
32}
```

# TDDの進め方(テスト)

\$ gradle cucumber

してテストがFailすることを確認する

```
Scenario: 問題に正答すると次の問題が出題される # cucumber/SampleScenario.feature:3
  Given ユーザ"taji"としてログインしている # SampleScenarioStep.ユーザ_としてログインしている(String)
    cucumber.api.PendingException: TODO: implement me
      at com.otajisan.steps.SampleScenarioStep.ユーザ_としてログインしている(SampleScenarioStep.java:12)
      at *.Given ユーザ"taji"としてログインしている(cucumber/SampleScenario.feature:4)

  When 正解となる問題を選択する          # SampleScenarioStep.正解となる問題を選択する()
  And "回答"ボタンをクリックする          # SampleScenarioStep.ボタンをクリックする(String)
  Then 次の問題が表示される             # SampleScenarioStep.次の問題が表示される()

1 Scenarios (1 pending)
4 Steps (3 skipped, 1 pending)
0m0.129s

cucumber.api.PendingException: TODO: implement me
  at com.otajisan.steps.SampleScenarioStep.ユーザ_としてログインしている(SampleScenarioStep.java:12)
  at *.Given ユーザ"taji"としてログインしている(cucumber/SampleScenario.feature:4)
```

ここから開発を開始する！！

(正確に言うと、更にユニットテストを書いてから)

ここで質問です

あなたはここまで  
コーディングを  
我慢できますか？

ぶっちゃけ  
先にコード

書きたくないですか？

この「我慢」が開発手法を  
変えることにつながります

この「我慢」が開発手法を  
変えることにつながります

と、いうことを  
体に叩き込まれる研修です♪

繰り返しになりますが、  
これを叩き込むのが難しい  
従来~~X~~



TDD



繰り返しになりますが、  
これを叩き込むのが難しい  
従来~~X~~



TDD



訓練するしかない

訓練を苦行としないために  
成功体験を味わうのが良い

# TDDの進め方(テスト)

ここで養いたい感覚は、

「デバッグは不要で、不具合は全てテストが報告するものだ」

ということ

```
Scenario: 問題に正答すると次の問題が出題される # cucumber/SampleScenario.feature:3
  Given ユーザ"taji"としてログインしている # SampleScenarioStep.ユーザ_としてログインしている(String)
    cucumber.api.PendingException: TODO: implement me
      at com.otajisan.steps.SampleScenarioStep.ユーザ_としてログインしている(SampleScenarioStep.java:12)
      at *.Given ユーザ"taji"としてログインしている(cucumber/SampleScenario.feature:4)

  When 正解となる問題を選択する
  And "回答"ボタンをクリックする
  Then 次の問題が表示される
    # SampleScenarioStep.正解となる問題を選択する()
    # SampleScenarioStep.ボタンをクリックする(String)
    # SampleScenarioStep.次の問題が表示される()

1 Scenarios (1 pending)
4 Steps (3 skipped, 1 pending)
0m0.129s

cucumber.api.PendingException: TODO: implement me
  at com.otajisan.steps.SampleScenarioStep.ユーザ_としてログインしている(SampleScenarioStep.java:12)
  at *.Given ユーザ"taji"としてログインしている(cucumber/SampleScenario.feature:4)
```

これが結果的に良いコードにつながり、  
機能追加を容易化させることができる

Try TDD!

# 研修で学んだこと

## 2. リファクタリング

# リファクタリング

- なぜやるのか？

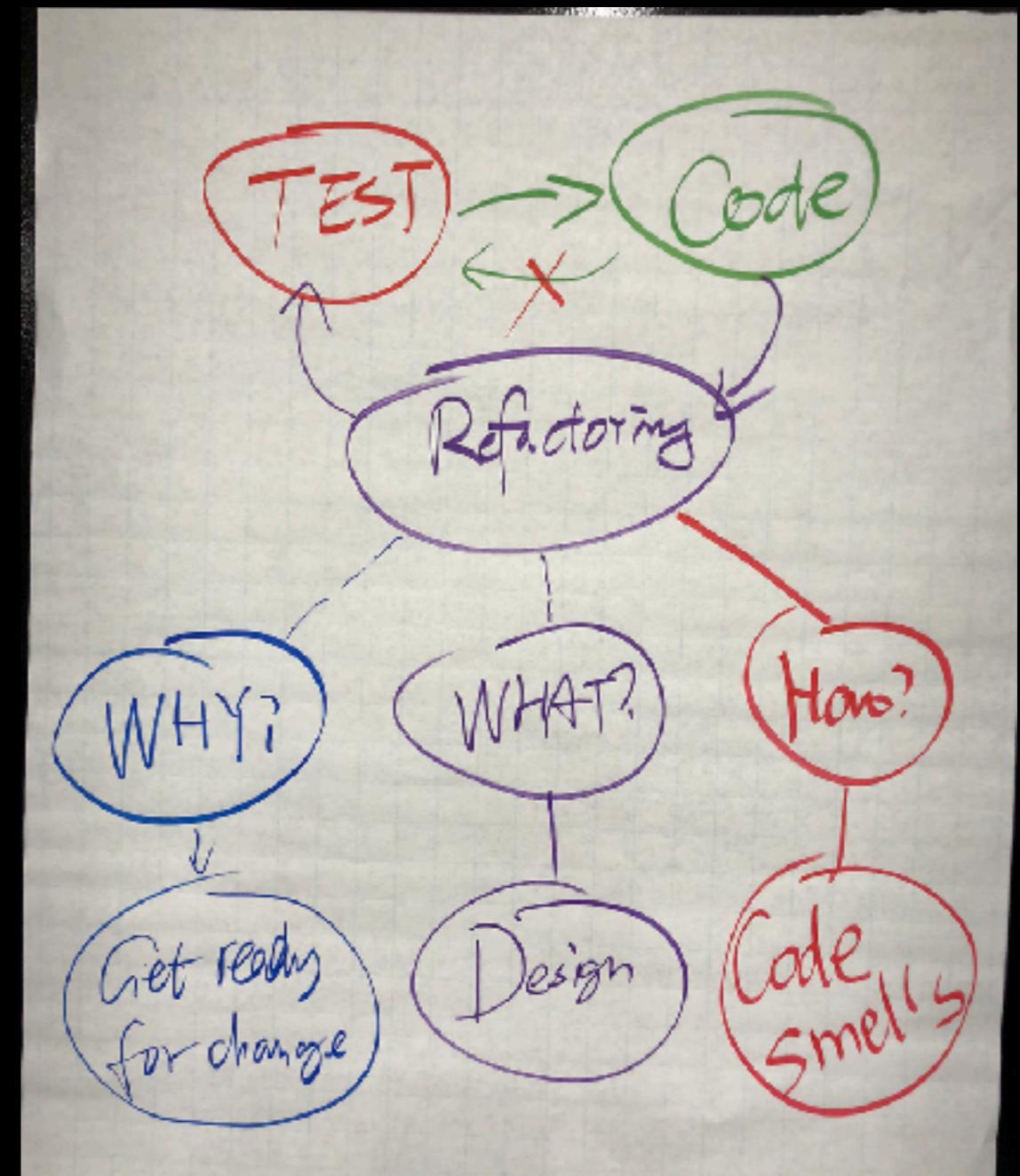
- 変更に備えるため

- 何をやるのか？

- 設計をする

- どう(いつ)やるのか？

- コードが”臭い”を発したら



# リファクタリング

- なぜやるのか？

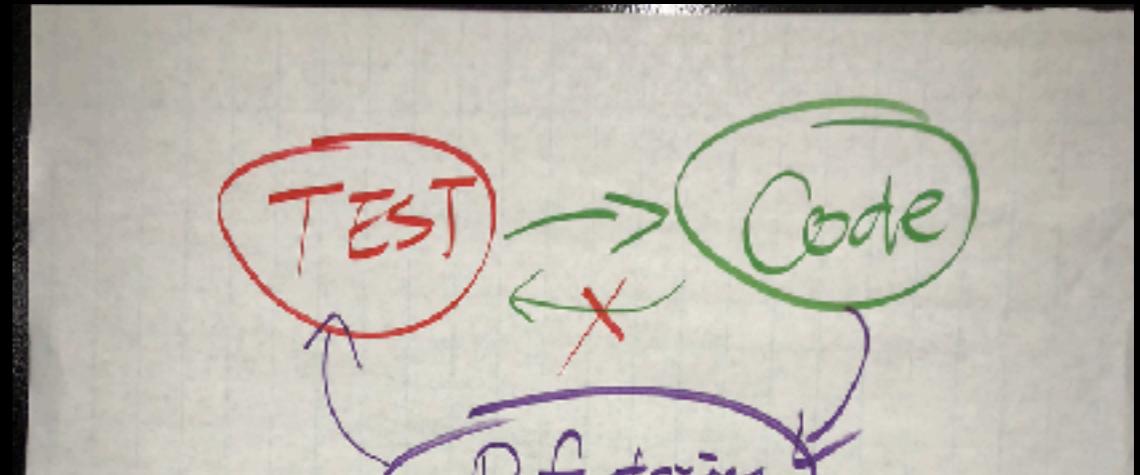
- 変更に備えるため

- 何をやるのか？

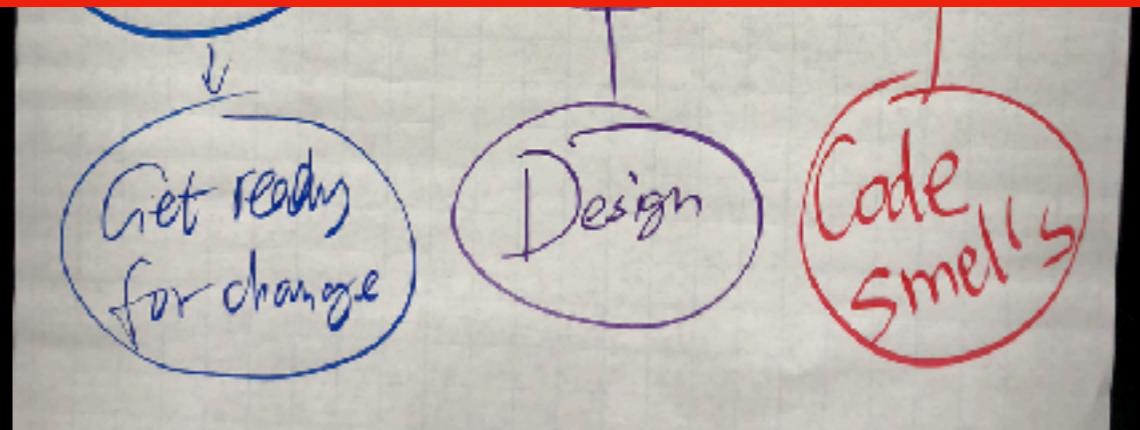
- 設計をする

- どう(いつ)やるのか？

- コードが”臭い”を発したら



自分が一番響いたのはココ



設計は

リファクタリング時にやる

設計は

リファクタリング時にやる

マジか

# リファクタリング

- 結局のところ、「先に設計をしない」のは、オーバースペックな設計配慮や、そもそも動作保証がなく、これらをリカバリする「ムダな手戻りを避けたい」、というモチベーションを強く感じる
- テストをパスし、動作することが保証された上で、Code Smellを感じたら設計(リファクタリング)する、というのがTDDスタイル、ということらしい

# リファクタリング(ポイント)

- 機能させる(Make It Work)
- キレイにする(Make It Right)
- 高速化する(Make It Fast)

# リファクタリング(ポイント)

- 高凝集・疎結合なコードを目指す
- 高凝集
  - 役割(責任範囲)が明確に表現されている
  - 理解しやすさ、再利用性の高さ、などのメリットがある
- 疎結合
  - クラス間の依存関係が弱い
  - 高凝集を促進させることで、結果的に疎結合になることが多い

# リファクタリング(ポイント)

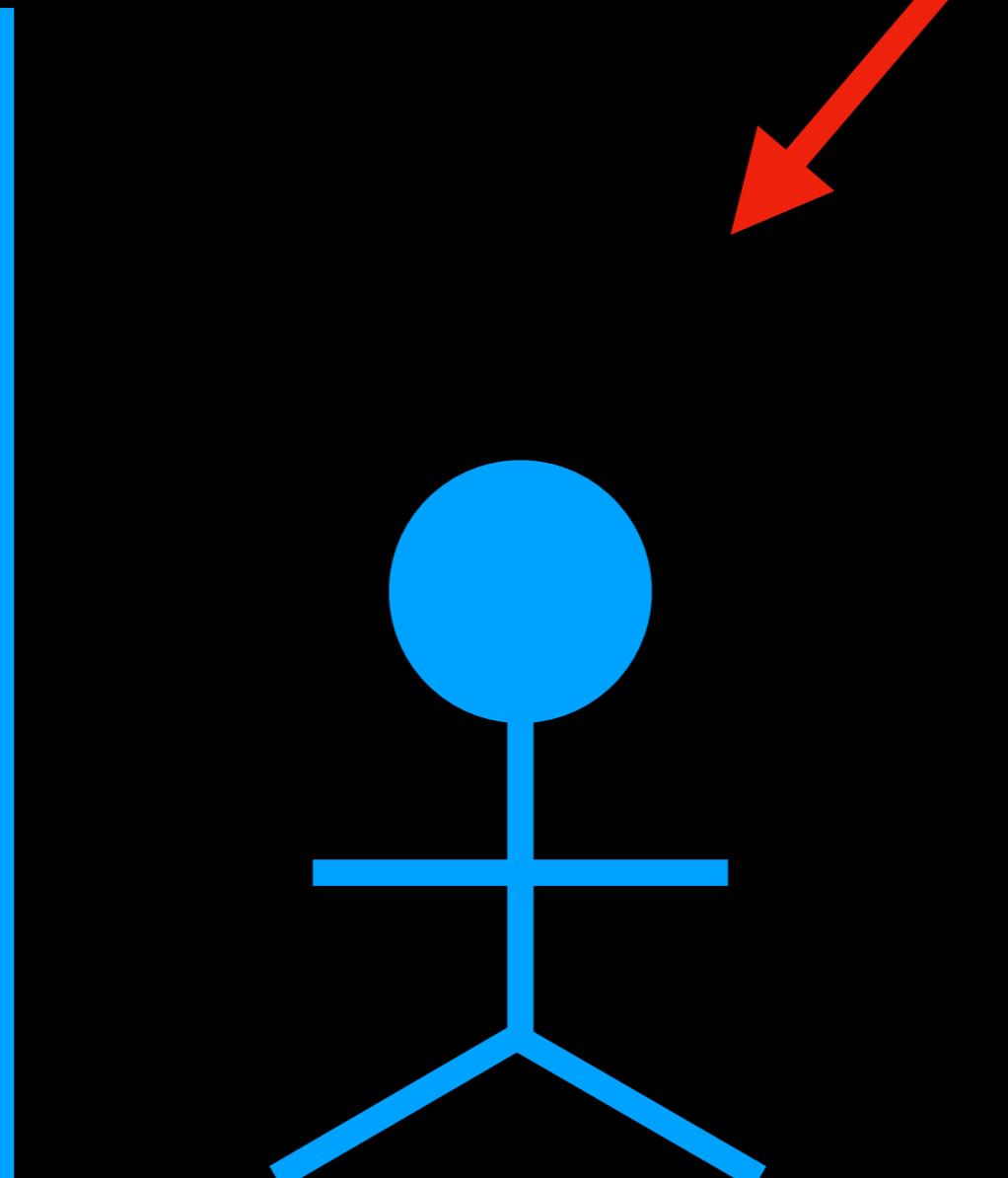
- 高凝集・疎結合なコードを目指す
- 高凝集 特に、高凝集のほうが重要、とされている
  - 役割(責任範囲)が明確に表現されている
  - 理解しやすさ、再利用性の高さ、などのメリットがある
- 疎結合
  - クラス間の依存関係が弱い
  - 高凝集を促進させることで、結果的に疎結合になることが多い

# 研修で学んだこと

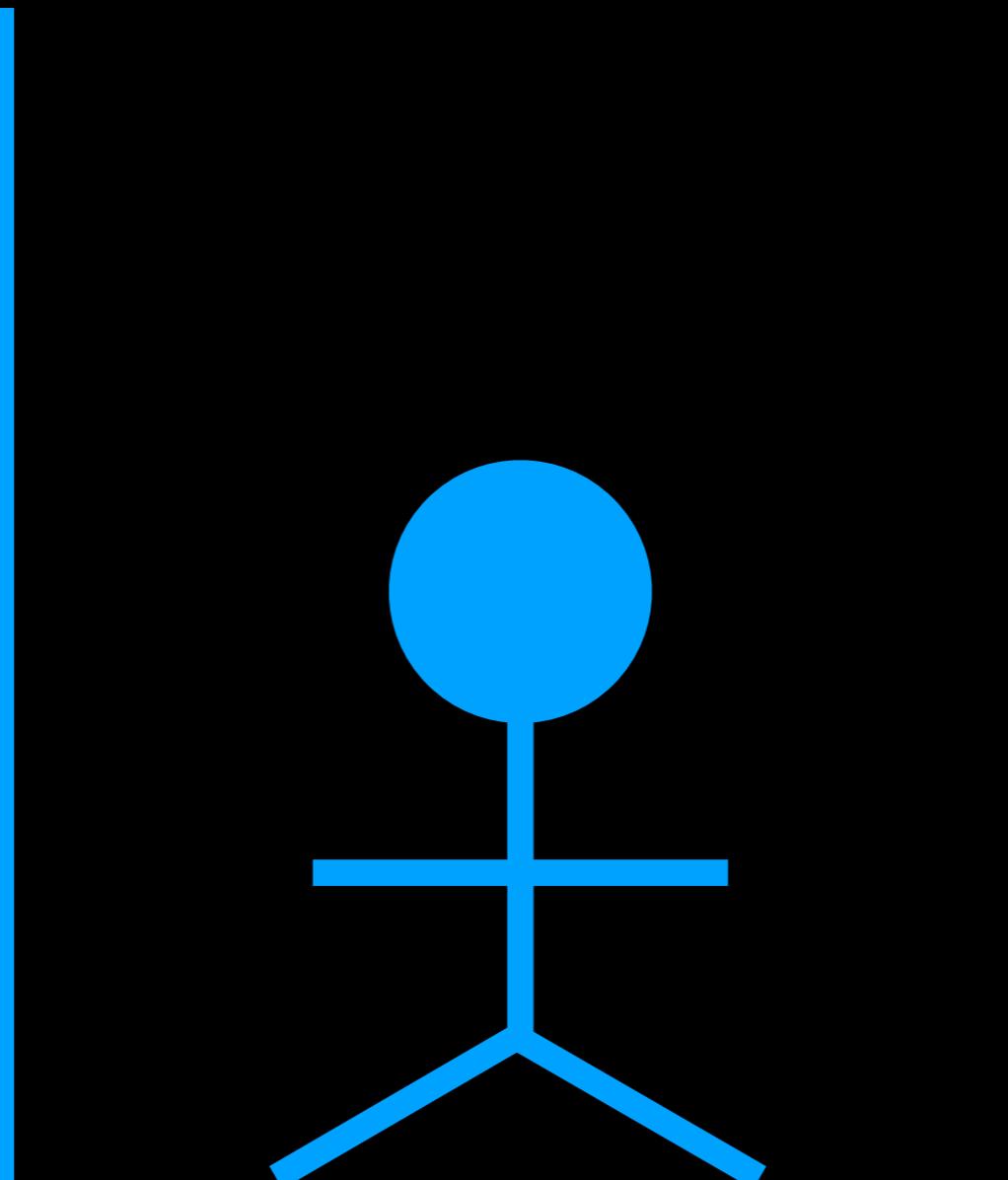
## 3. レガシーコードとの戦い方

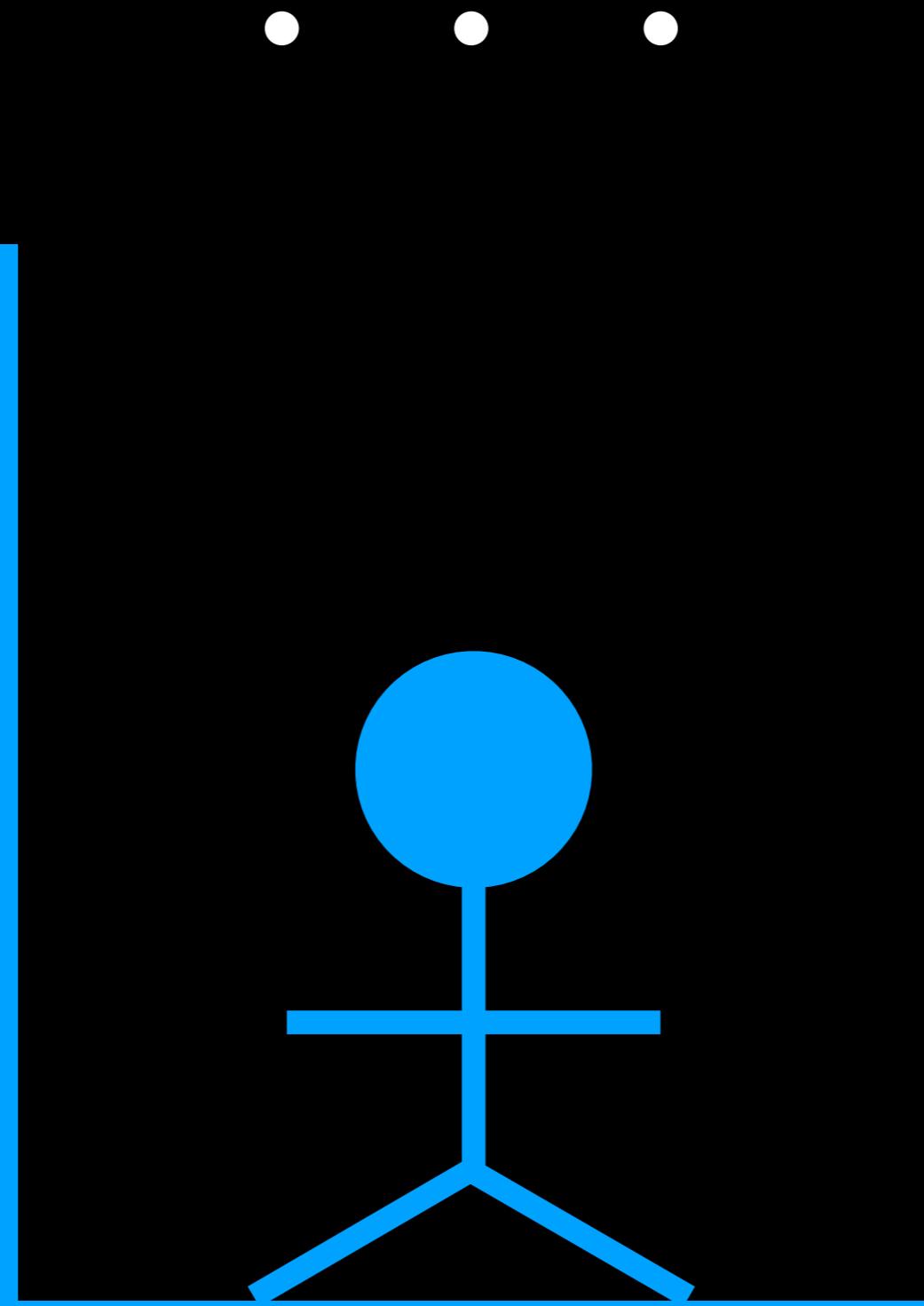
あなたは穴に  
ハマっています

穴

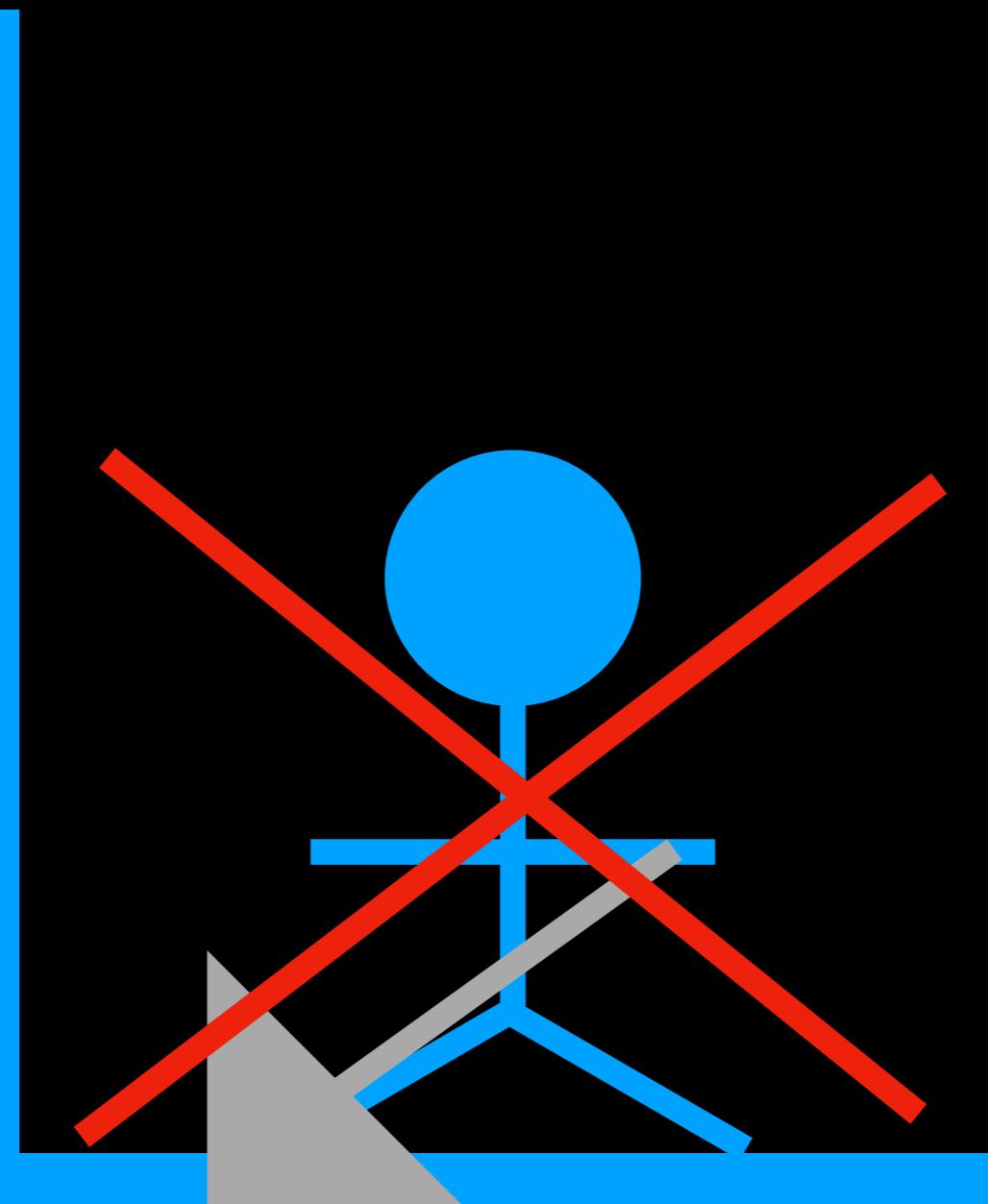


# 最初に何をすべきか？





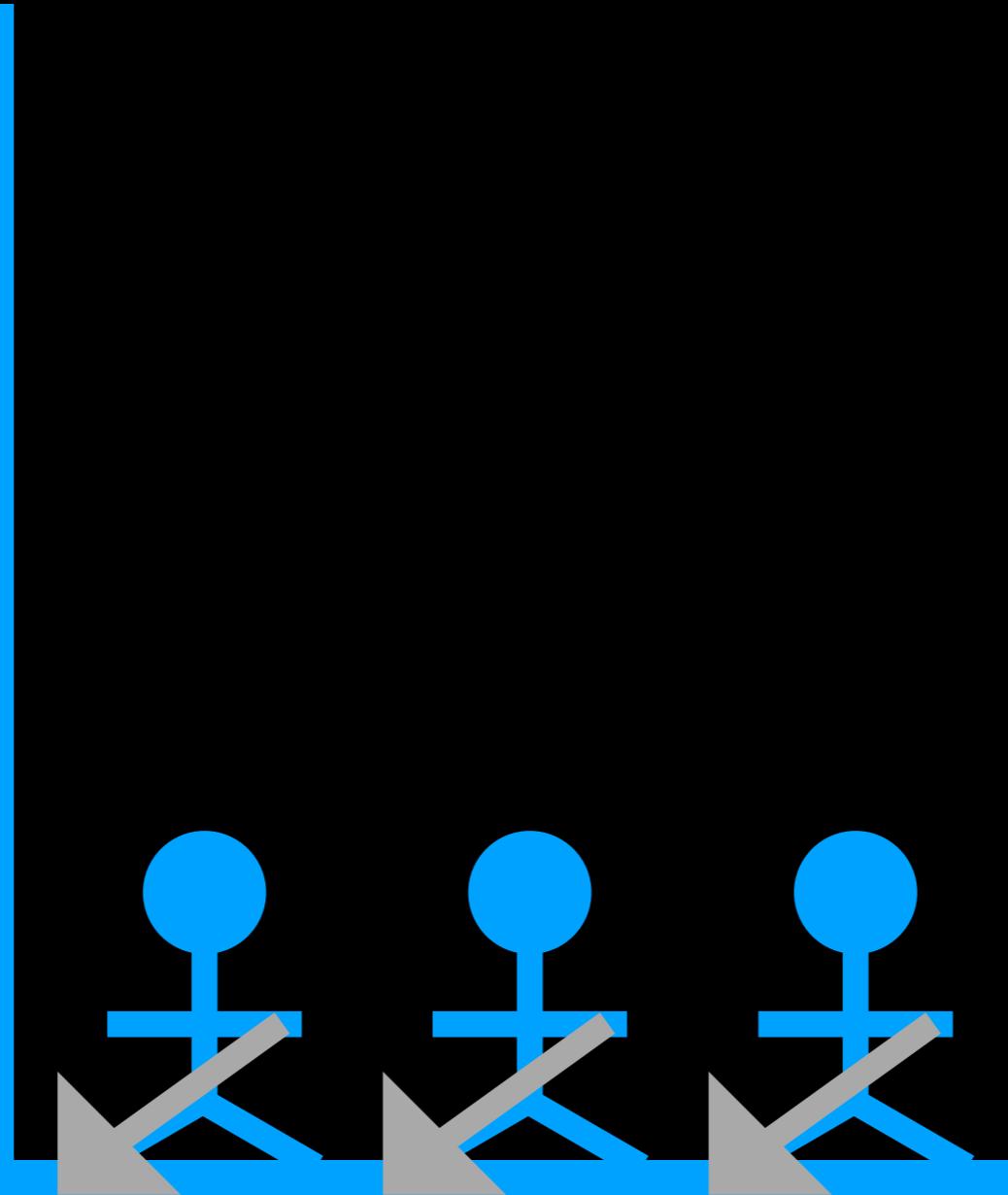
これ以上穴を掘ることを  
止めましょう



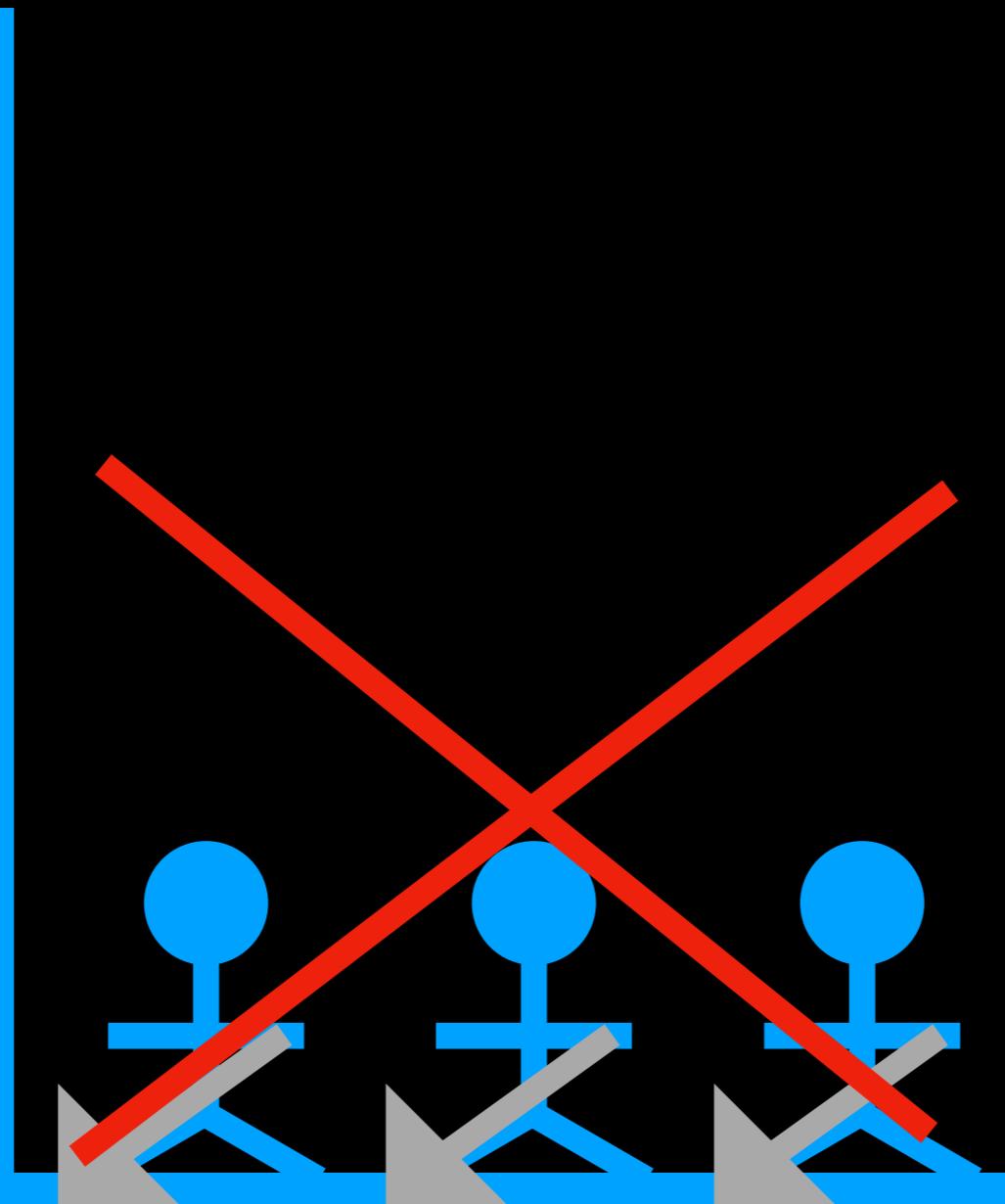
# レガシーコードとの戦い方

- ・ 数1000ステップのコードを含むメソッドがあったとする
- ・ そこをいじらなければ機能追加ができない、とする
- ・ まず、「何で数1000ステップのメソッド」ができたのかを考える

みんなが掘っちゃったから



# 止めなければならない



# レガシーコードとの戦い方

- レガシーコード = テストが無いコード
- なので、穴を掘るのを止めるために、  
テストを書かなければならぬ
- テストが無ければリファクタリングもできない

# レガシーコードとの戦い方

- ・ レガシーコードと戦う場合に関しては、必ずしもテストから書く必要はない
- ・ 機能追加となるコードを記述後、局所的にテストを追加する
- ・ そうすることで、多少は安全地帯を増やすことができる  
(次回、その箇所の変更の際に、リスクが減る)
- ・ レガシーコードをテストファーストに作ろうとすると、テストが極端に複雑になる可能性が高い(ので、コードから)

# レガシーコードとの戦い方

- ・ 詳しくは「レガシーコード改善ガイド」を(ry

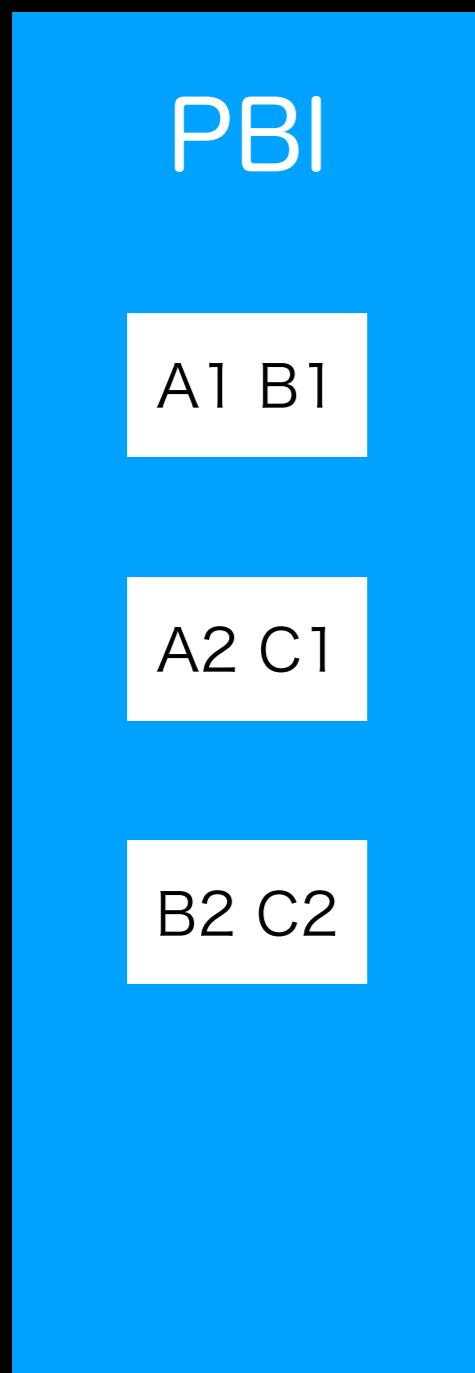
研修で学んだこと

4. コンポーネントチーム

vs

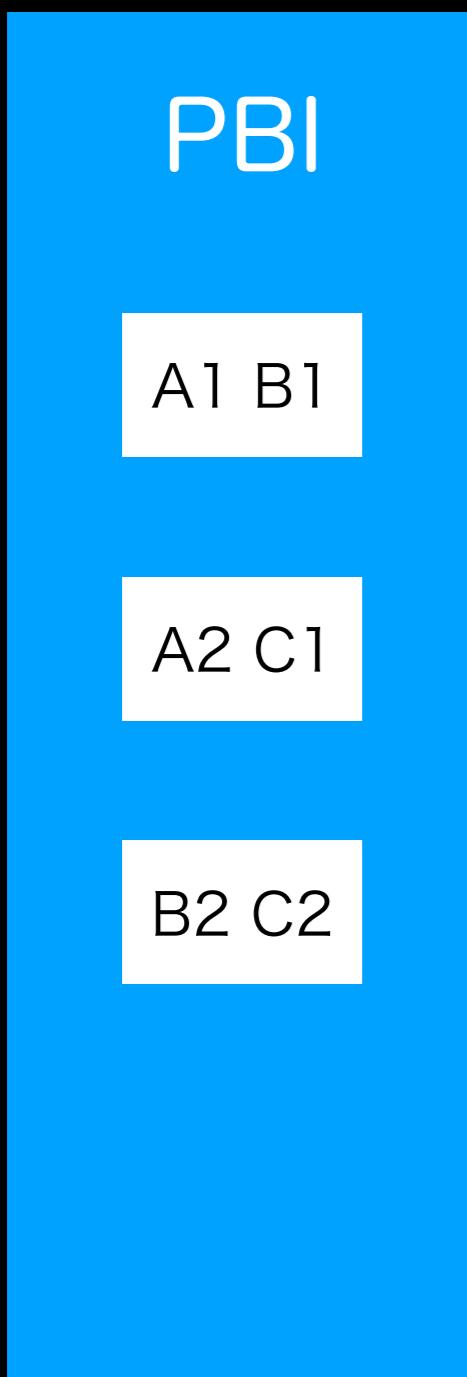
フィーチャーチーム

# コンポーネントチーム vs フィーチャーチーム



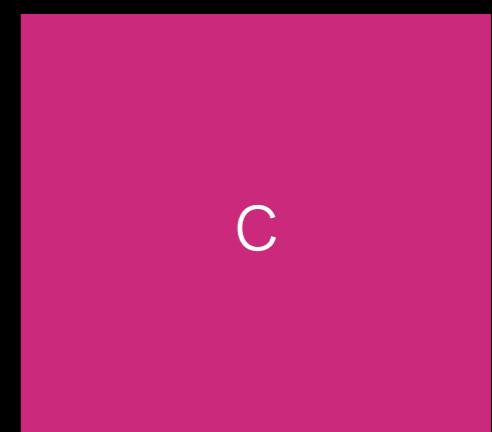
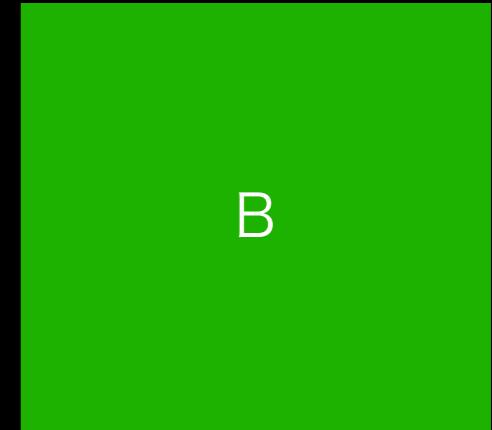
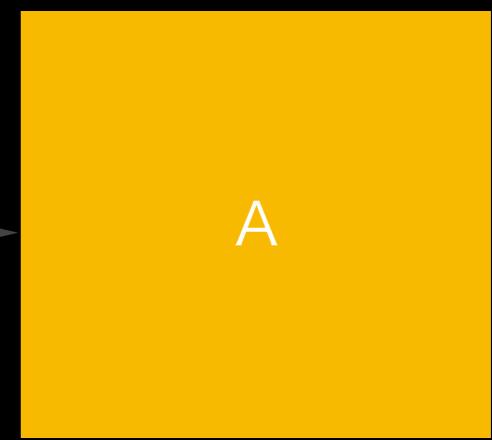
このようなプロダクトバックログを  
チームで開発することを考える

# コンポーネントチームの場合

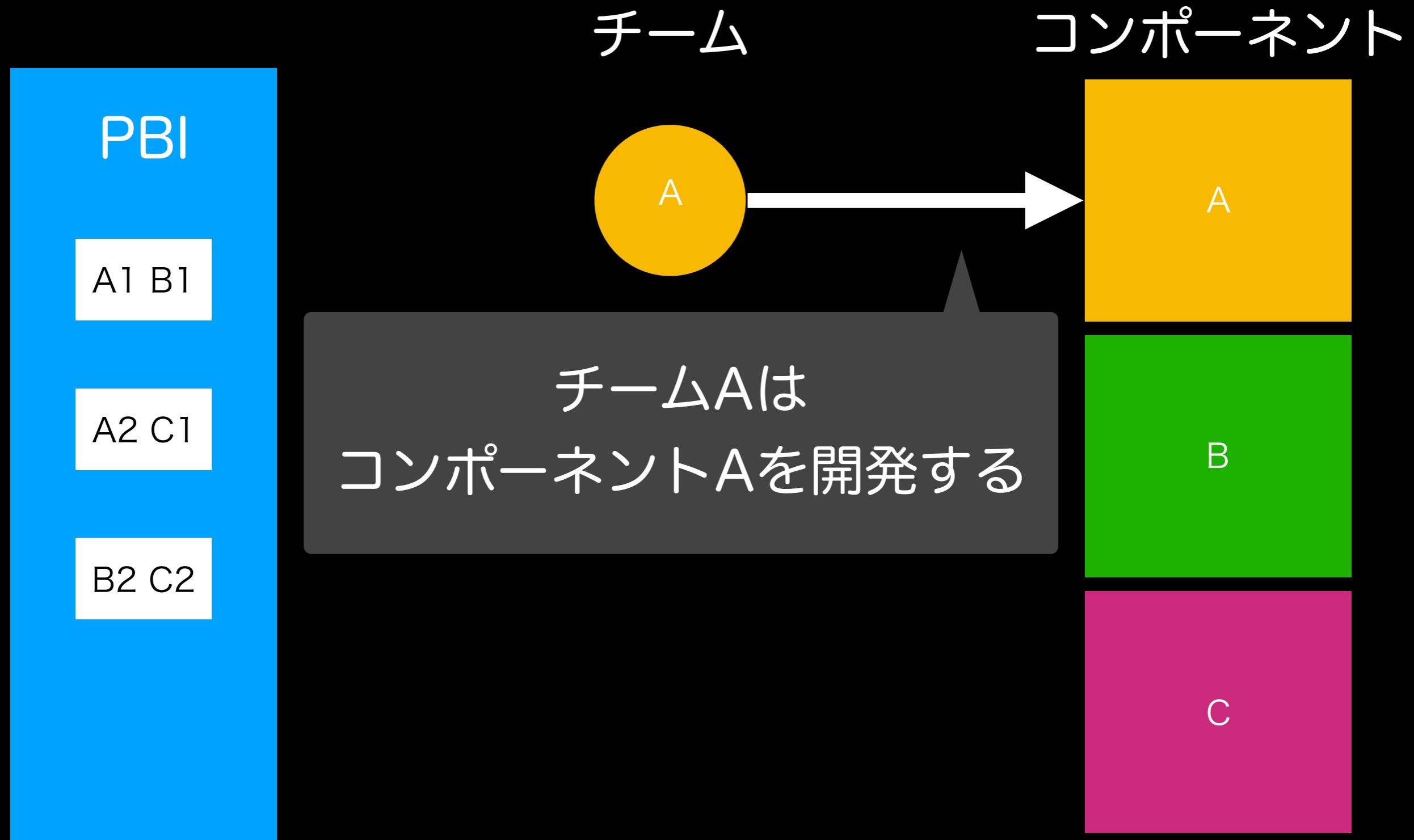


機能を満たすのに必要な  
コンポーネントがあって

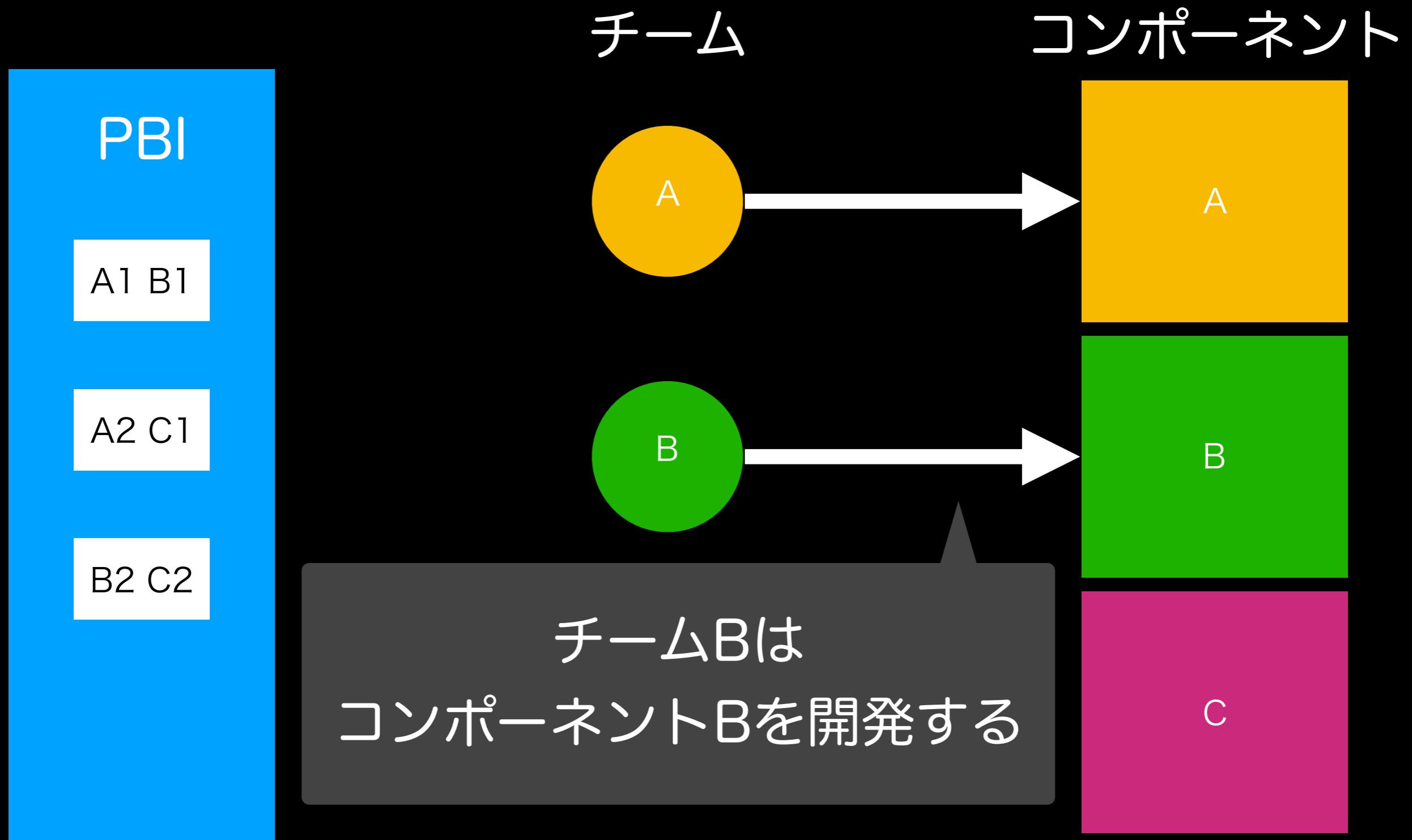
コンポーネント



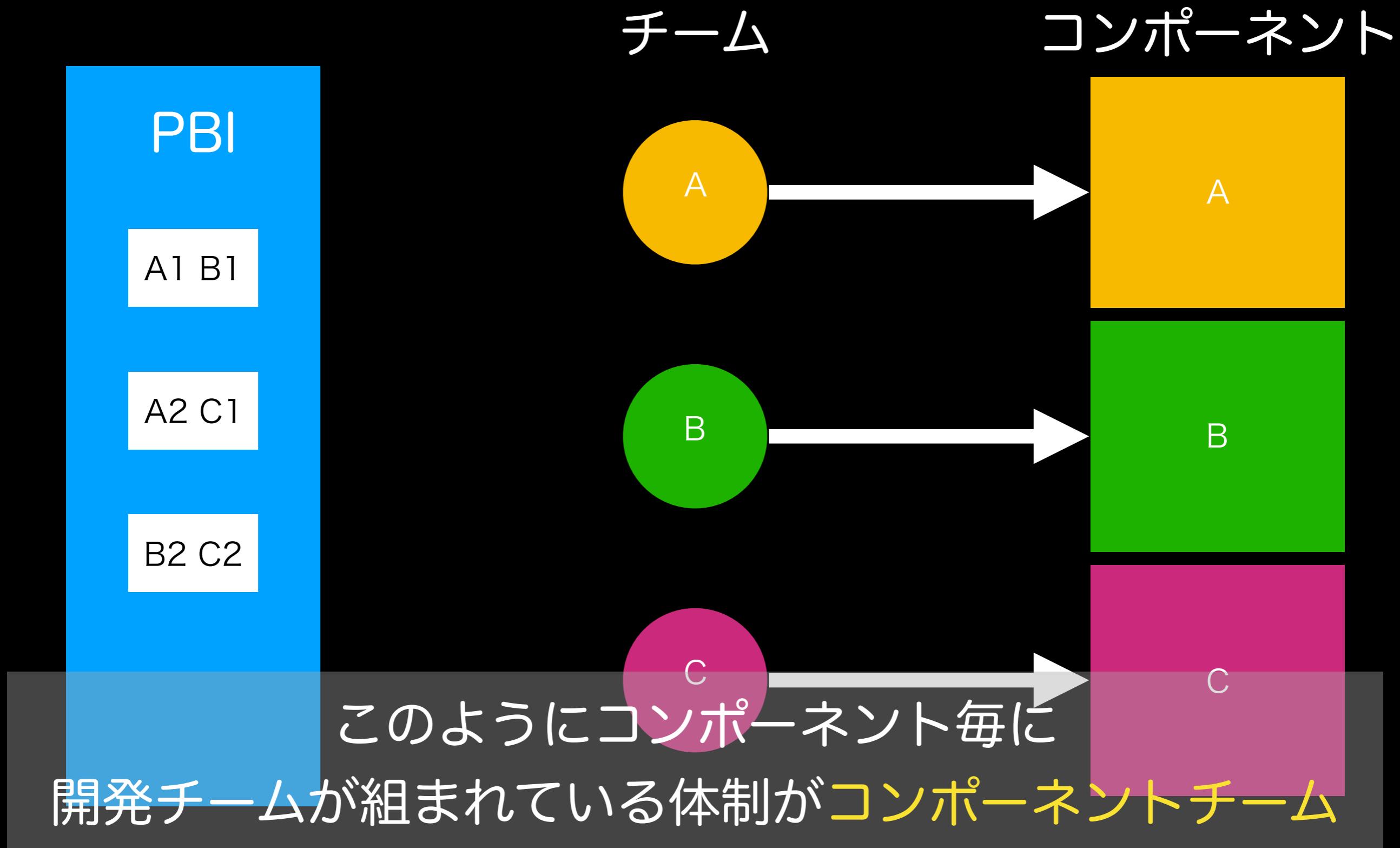
# コンポーネントチームの場合



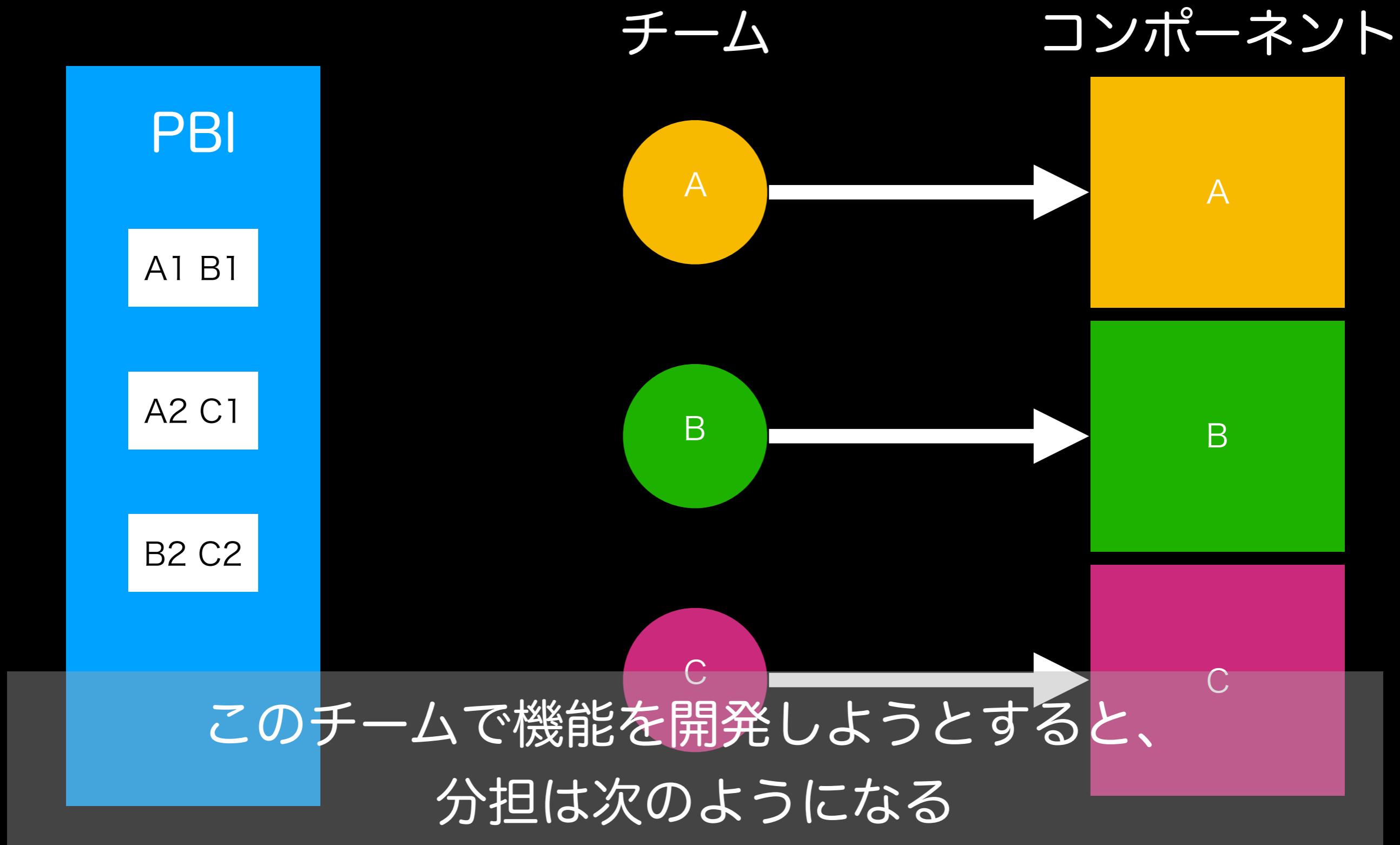
# コンポーネントチームの場合



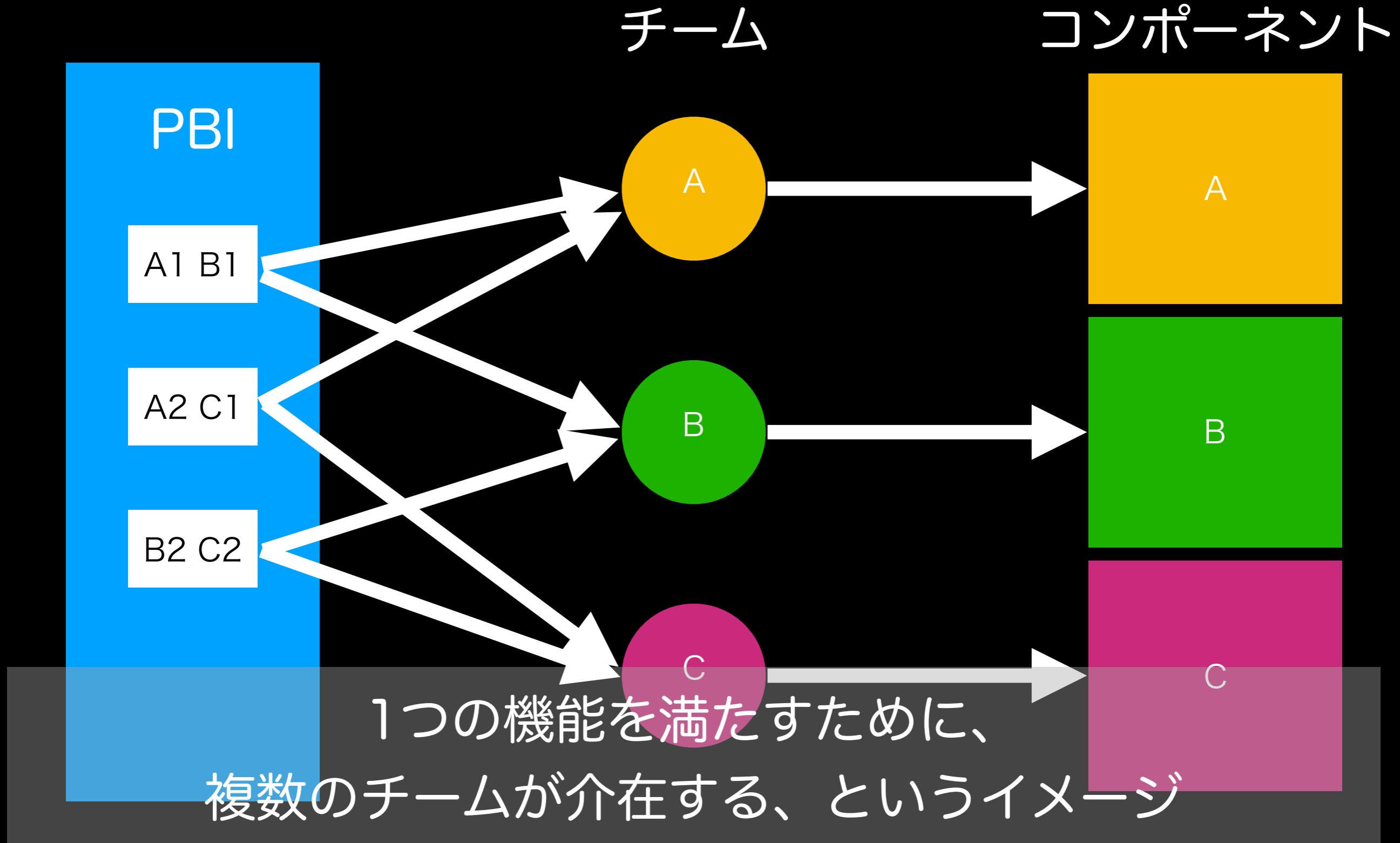
# コンポーネントチームの場合



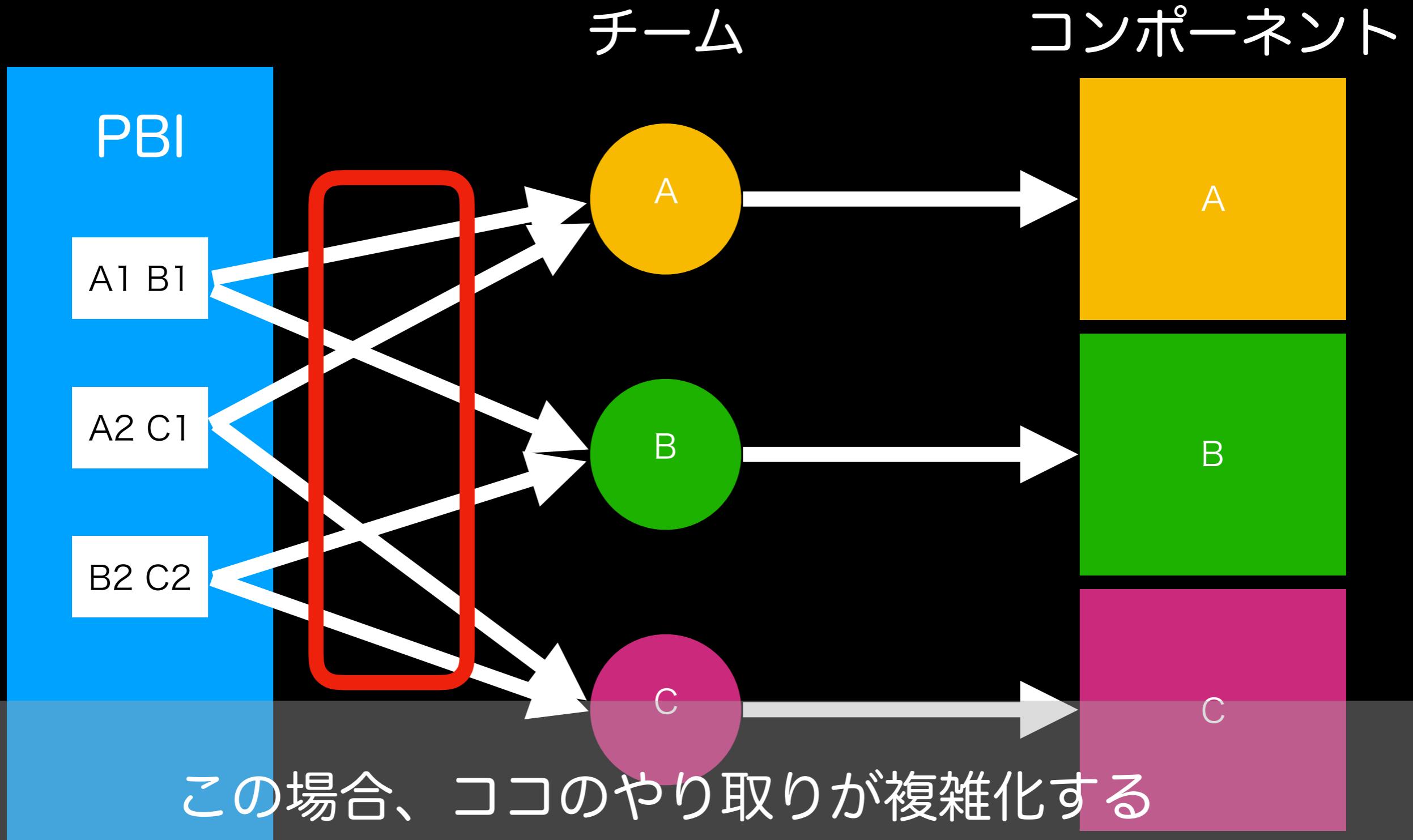
# コンポーネントチームの場合



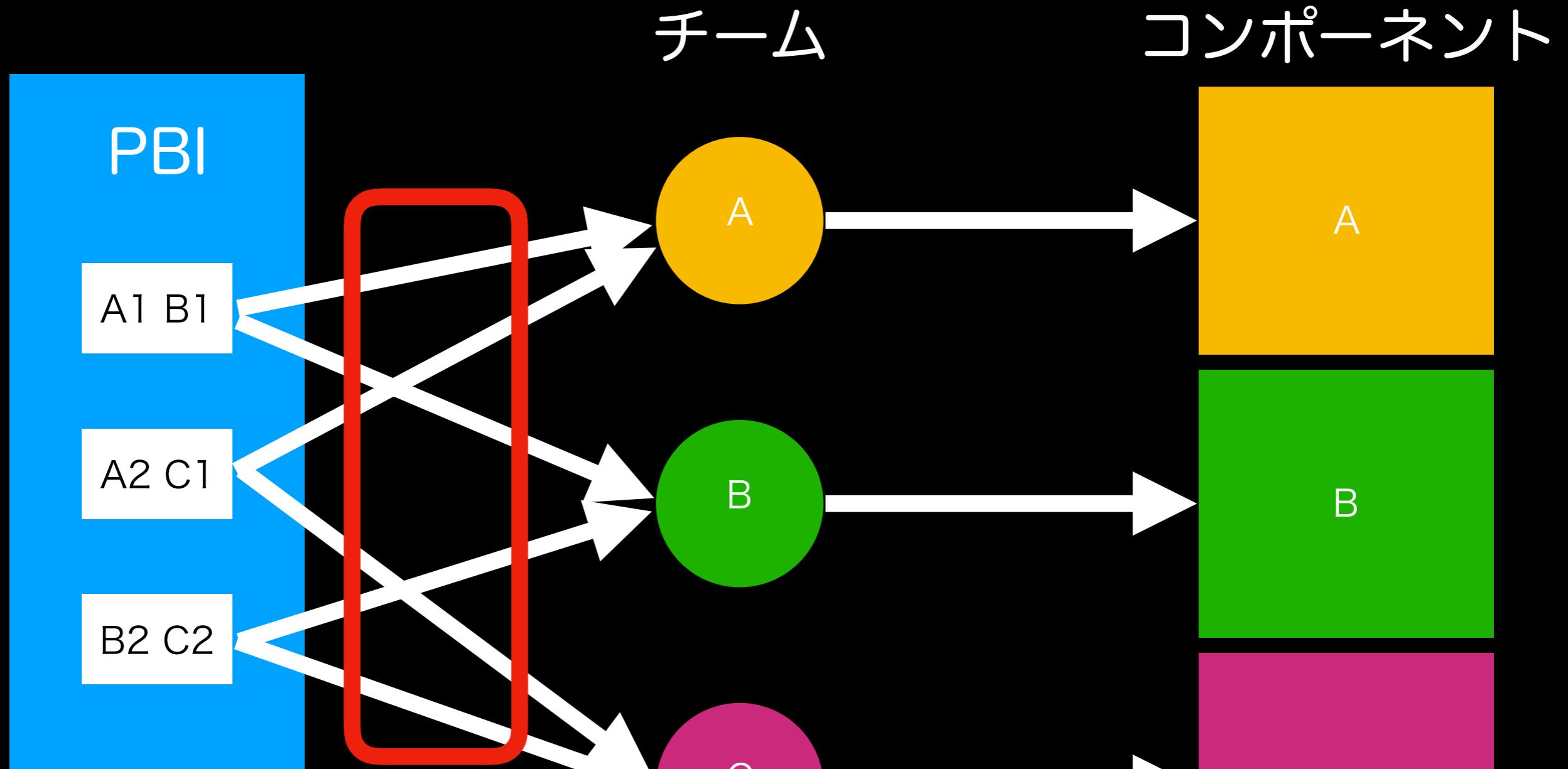
# コンポーネントチームの場合



# コンポーネントチームの場合



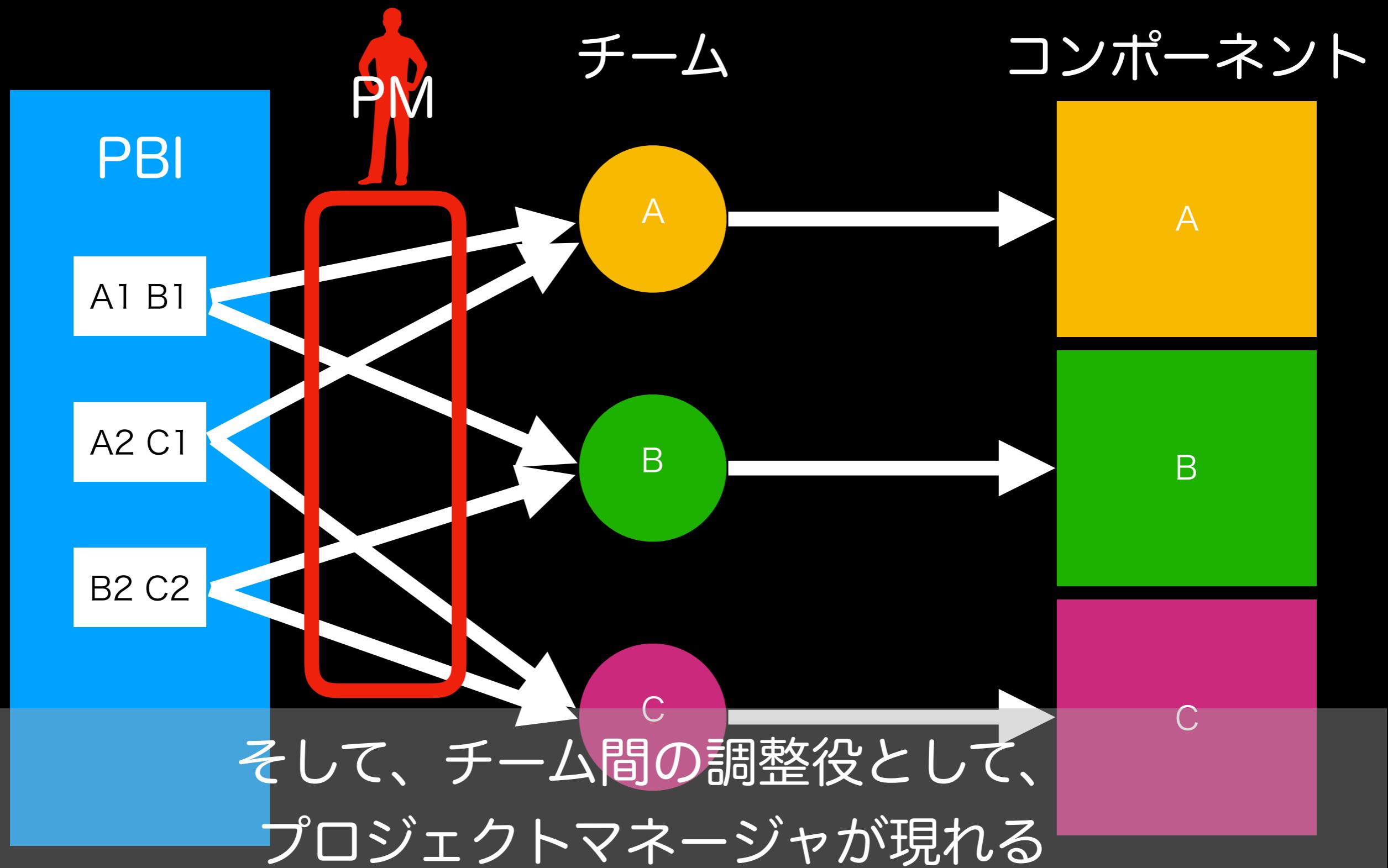
# コンポーネントチームの場合



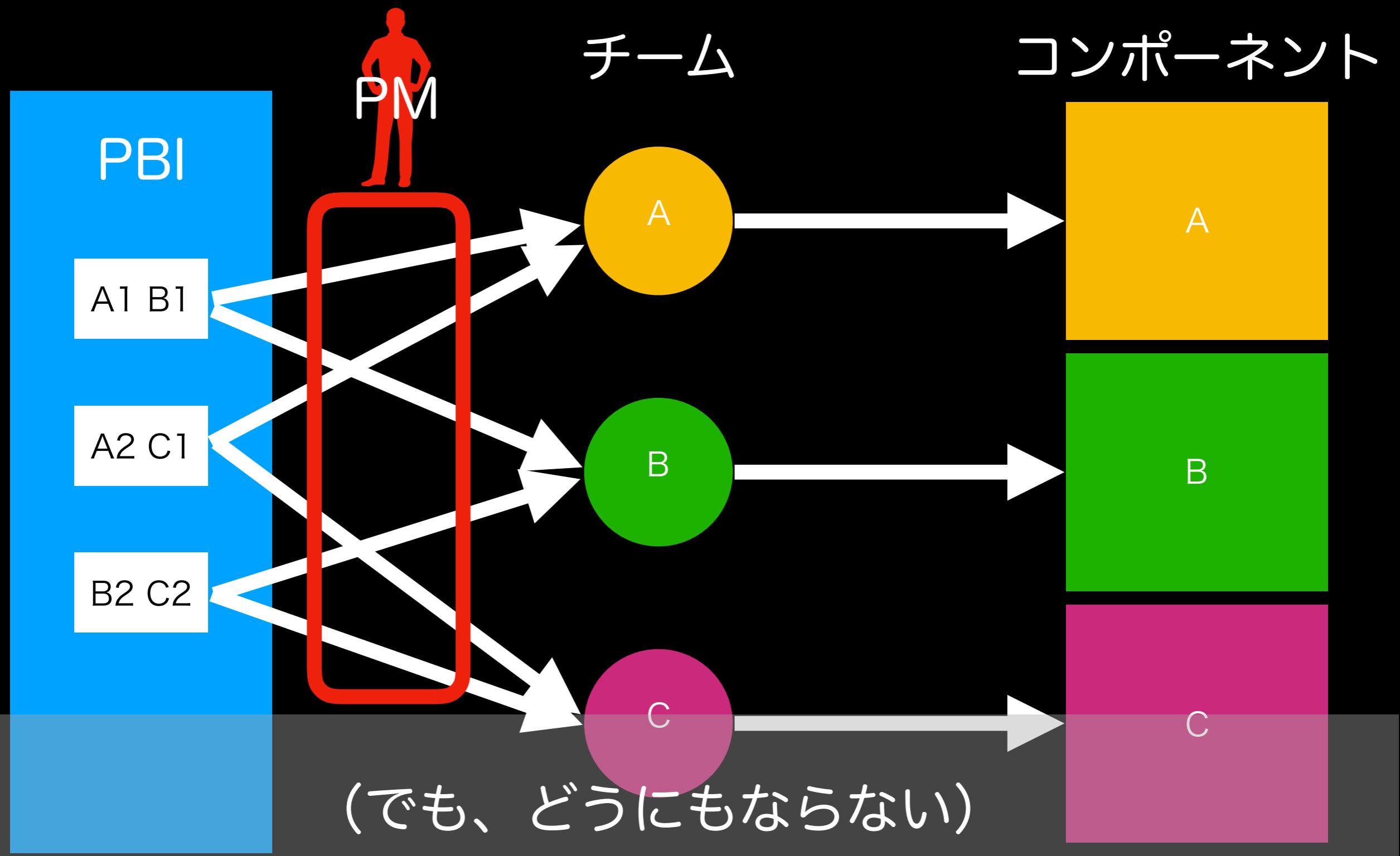
例えば、チケットベースで他のチームとやり取りをする

A、B両チームの開発が完了しなければリリースできないetc.

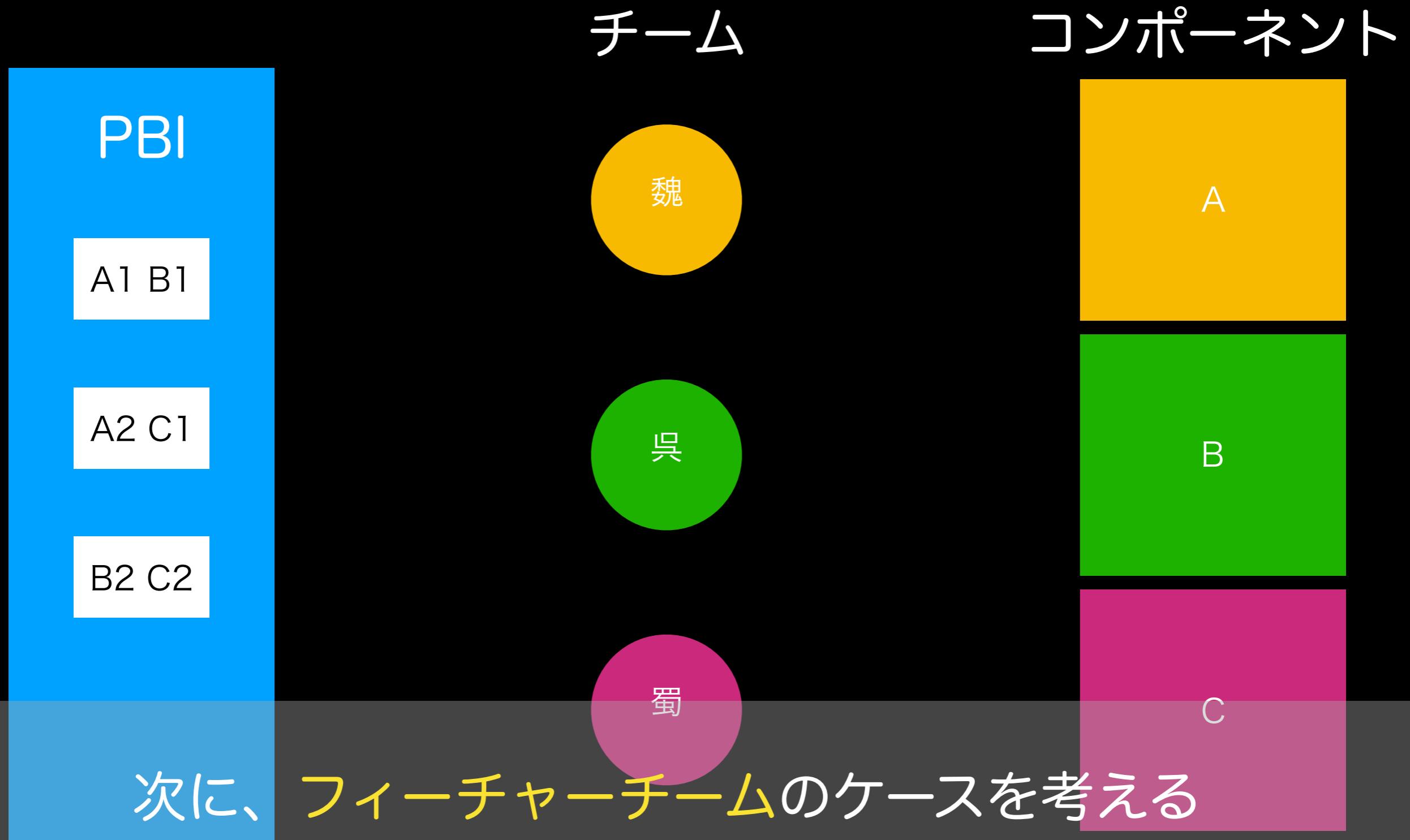
# コンポーネントチームの場合



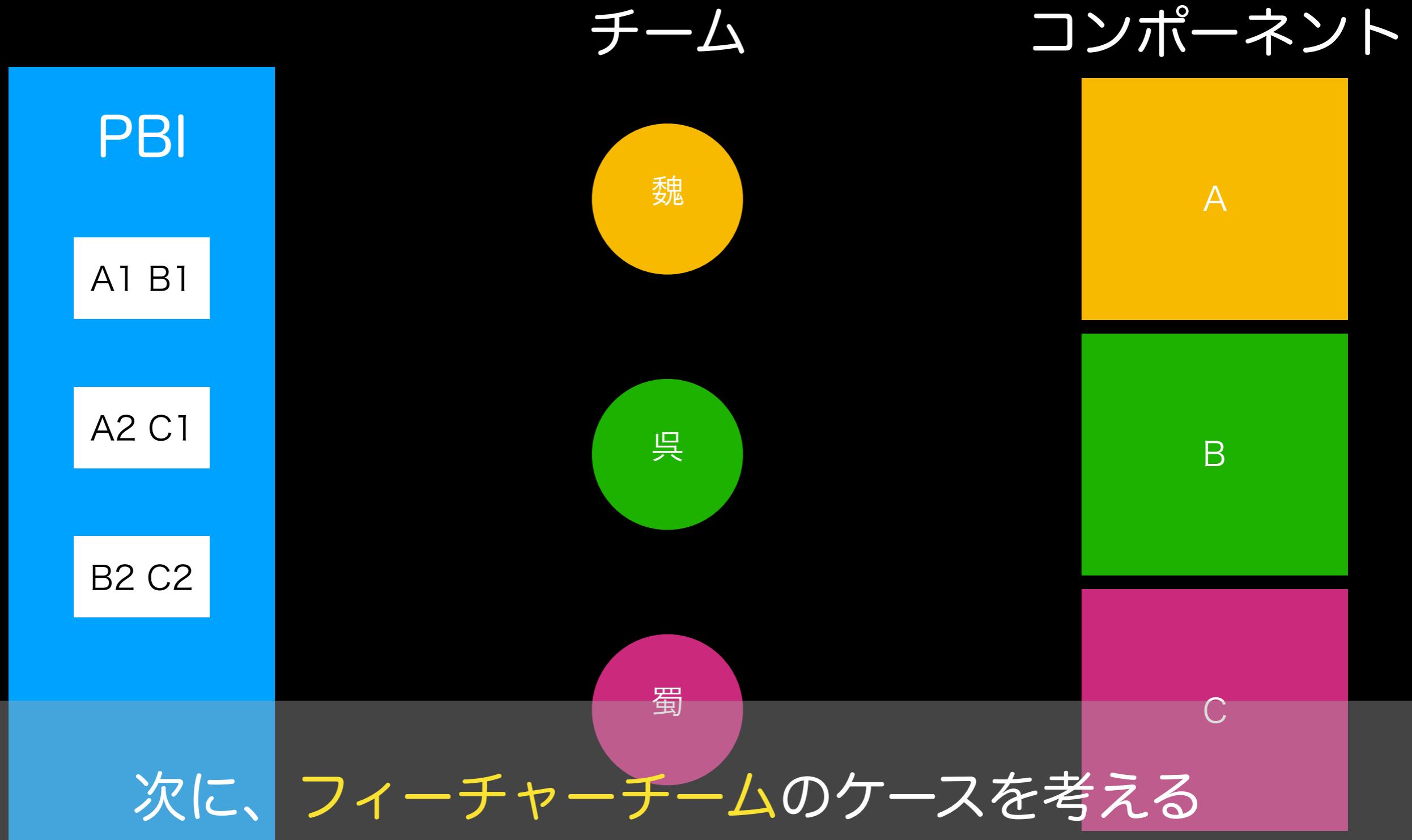
# コンポーネントチームの場合



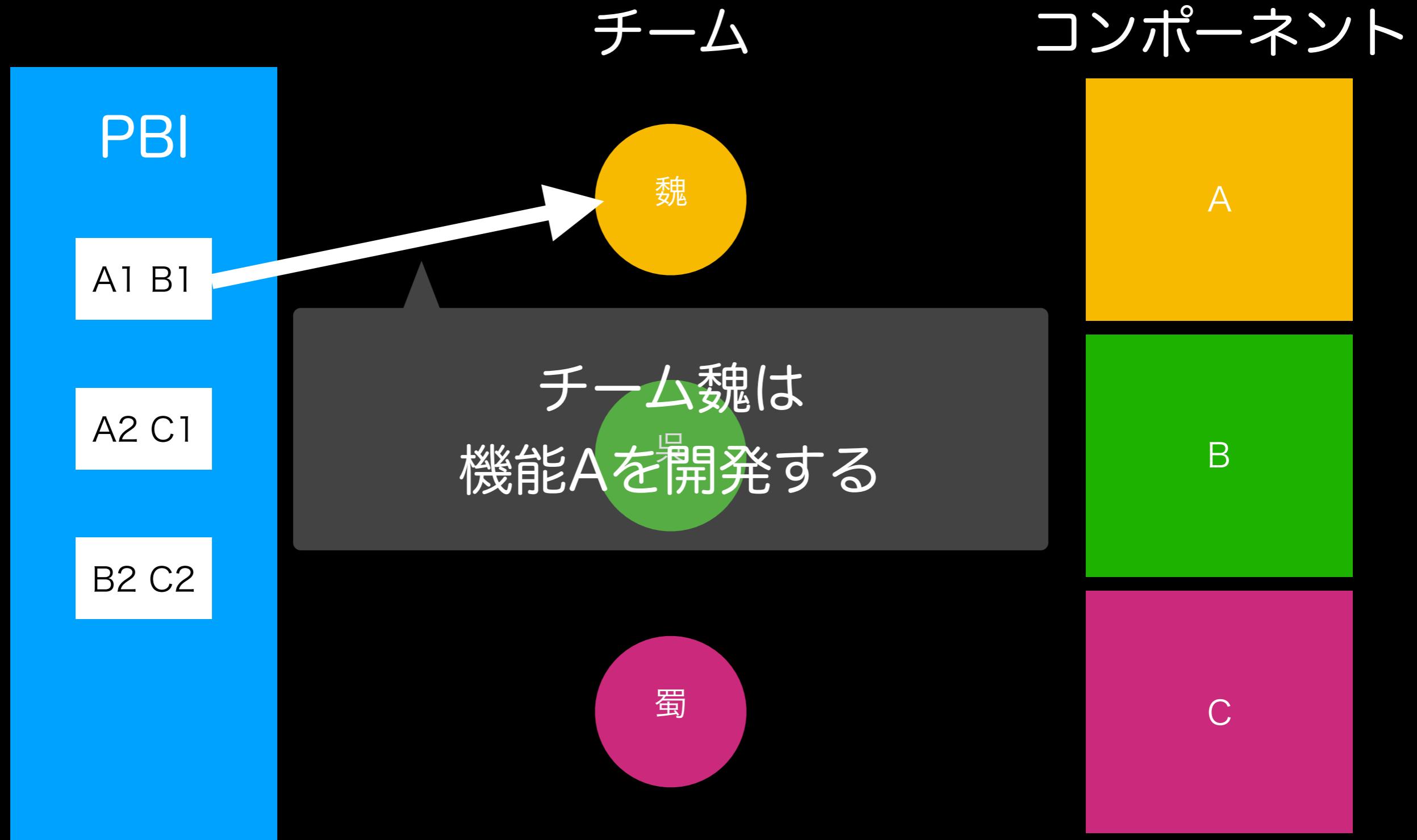
# フィーチャーチームの場合



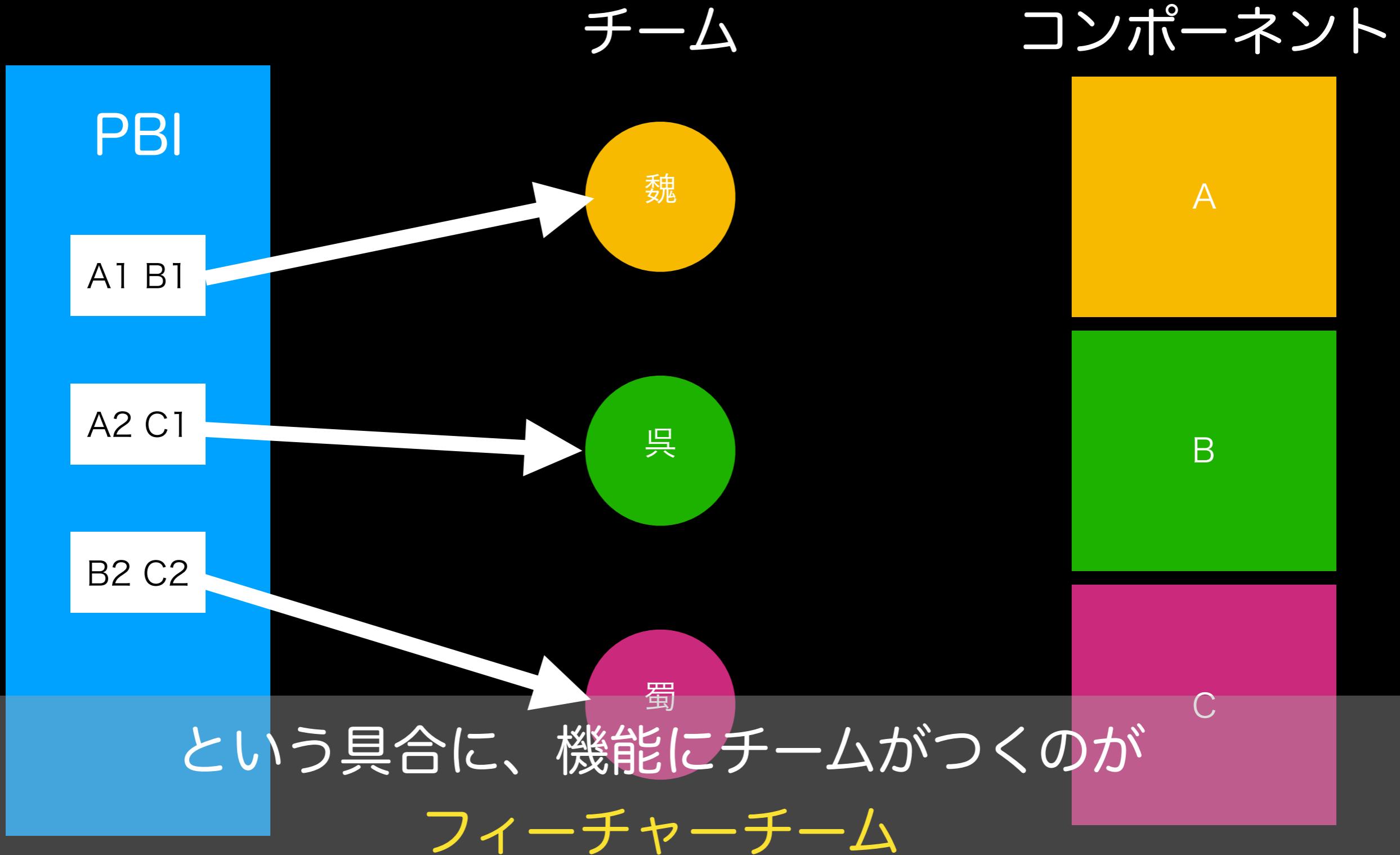
# フィーチャーチームの場合



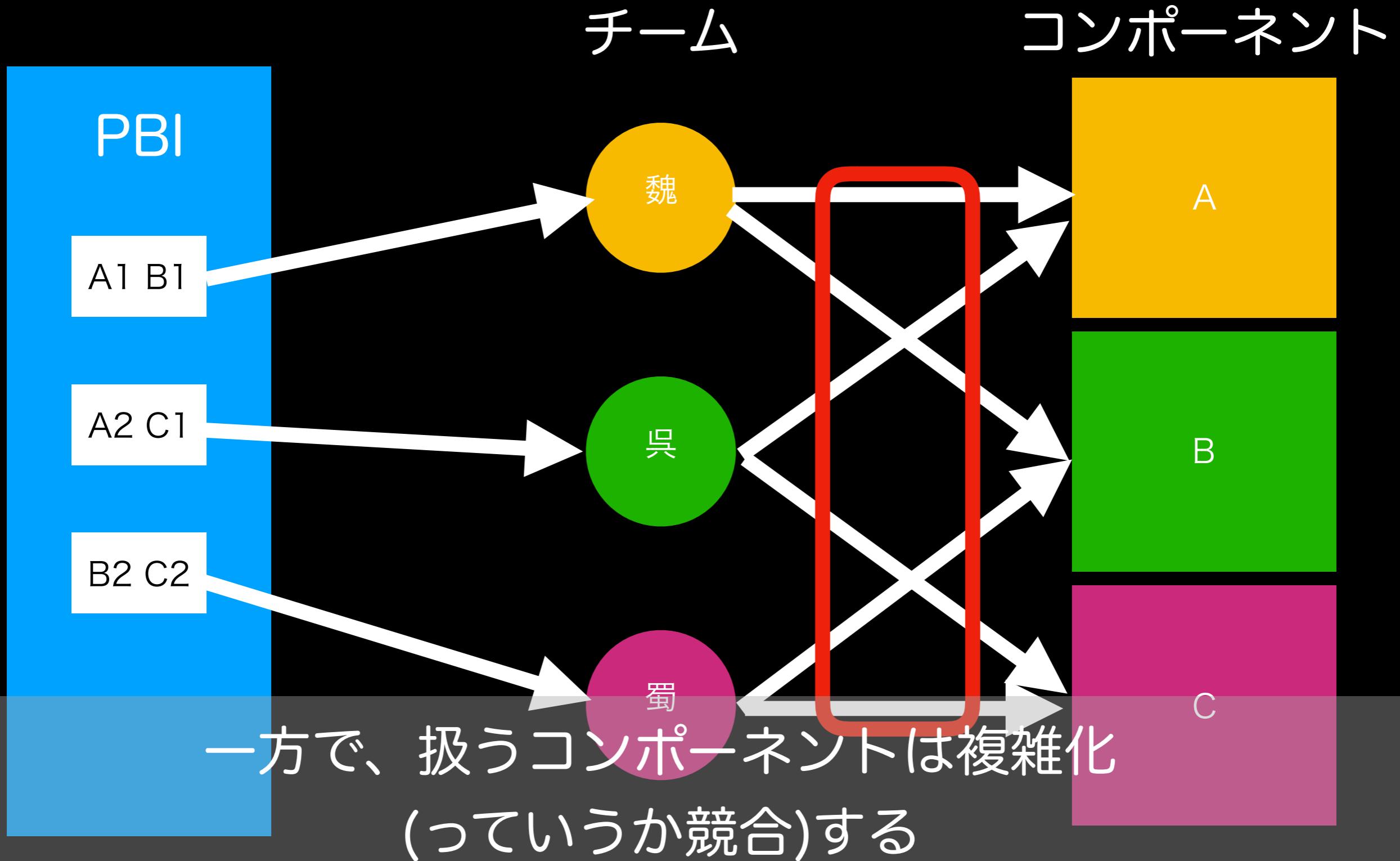
# フィーチャーチームの場合



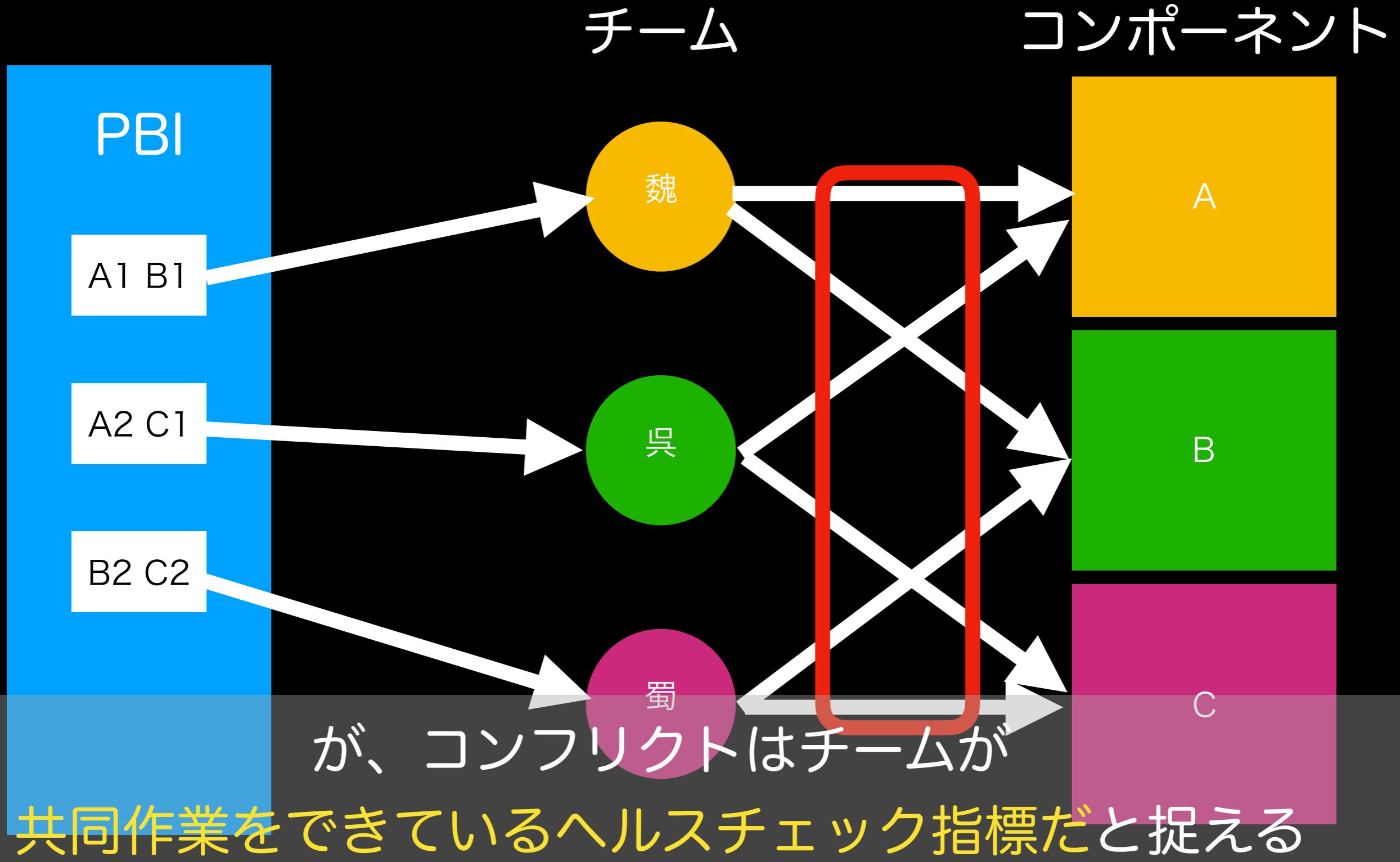
# フィーチャーチームの場合



# フィーチャーチームの場合



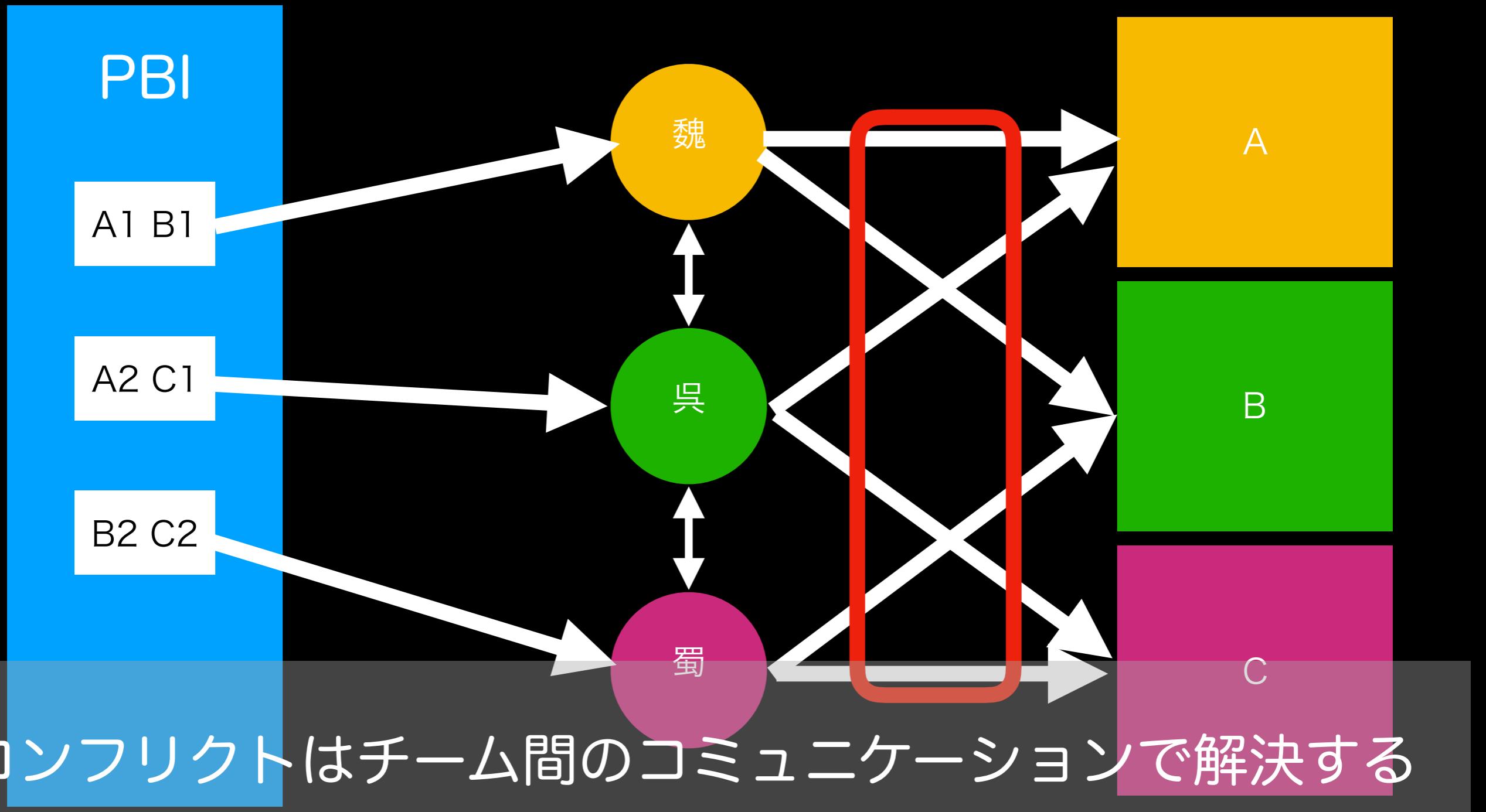
# フィーチャーチームの場合



# フィーチャーチームの場合

チーム

コンポーネント



# コンポーネントチーム vs フィーチャーチーム

- コンポーネントチーム

- コンポーネントごとの専門家  
(例えばDBチーム、APIチーム etc.)が集まっているので、対象コンポーネントの開発スキルには長けている
- 局所最適
- ウォーターフォール開発
- 一方で、機能開発をする際に、チーム間の調整コストが高くなり、融通がききづらかったりする

# コンポーネントチーム vs フィーチャーチーム

- フィーチャーチーム

- 組織の既存の枠やコンポーネントにとらわれず、顧客のフィーチャーを1つずつ完成させる長寿のチーム
- 全体最適
- 大規模アジャイル開発
- 一方で、チームには機能を実現するためにメンバーの幅広いスキルセットや高いコミュニケーション能力が要求される

# フィーチャーチームが やりづらいケース

- ・マイクロサービスの場合
- ・把握しなければならないコンポーネント数が多いので
- ・管理対象リポジトリが多岐に渡っている場合
- ・情報共有がしづらい

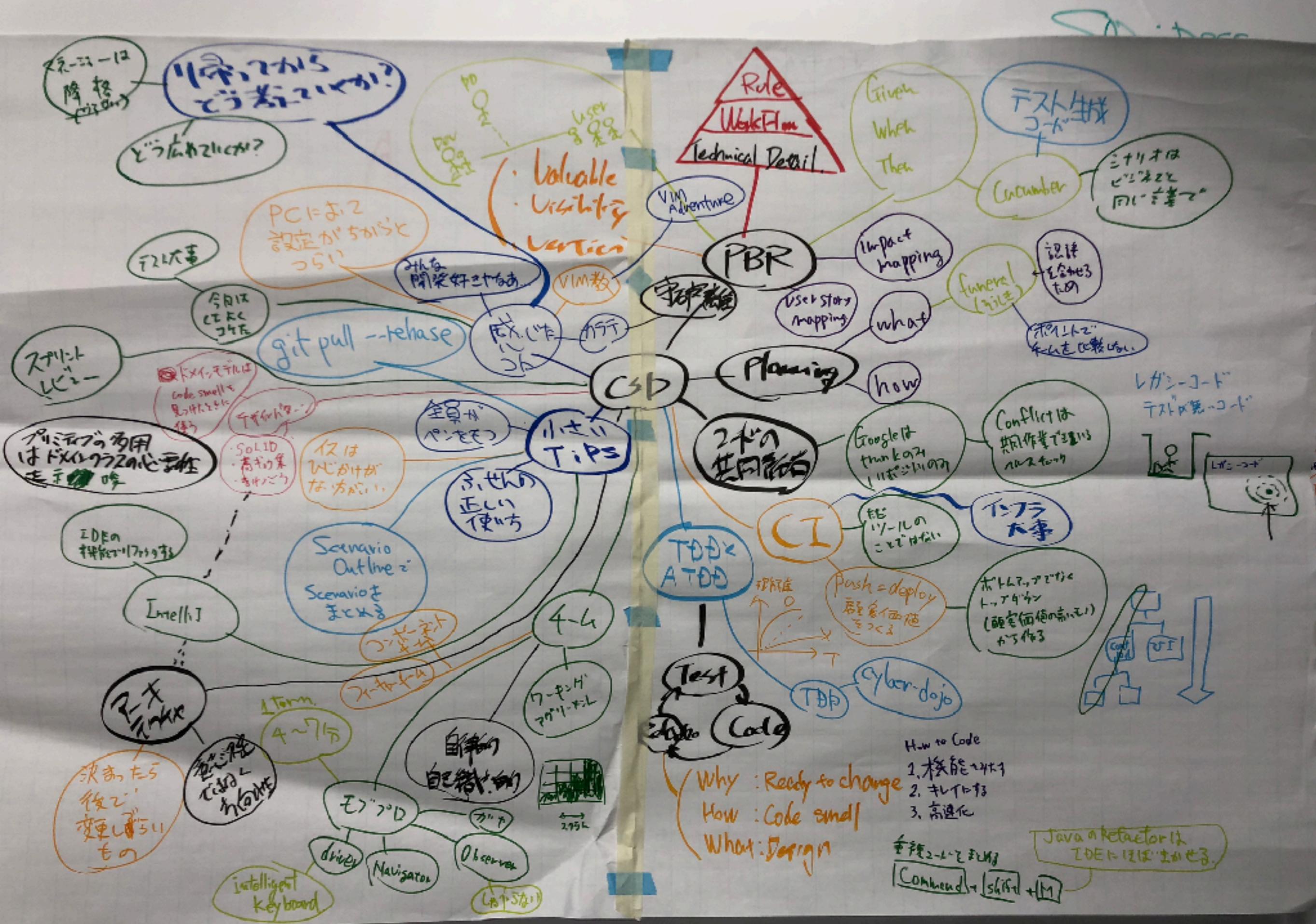
# フィーチャーチームが やりづらいケース

- ・例えば有名な話として、  
Googleは数十億行のソースコードを  
1リポジトリで管理している

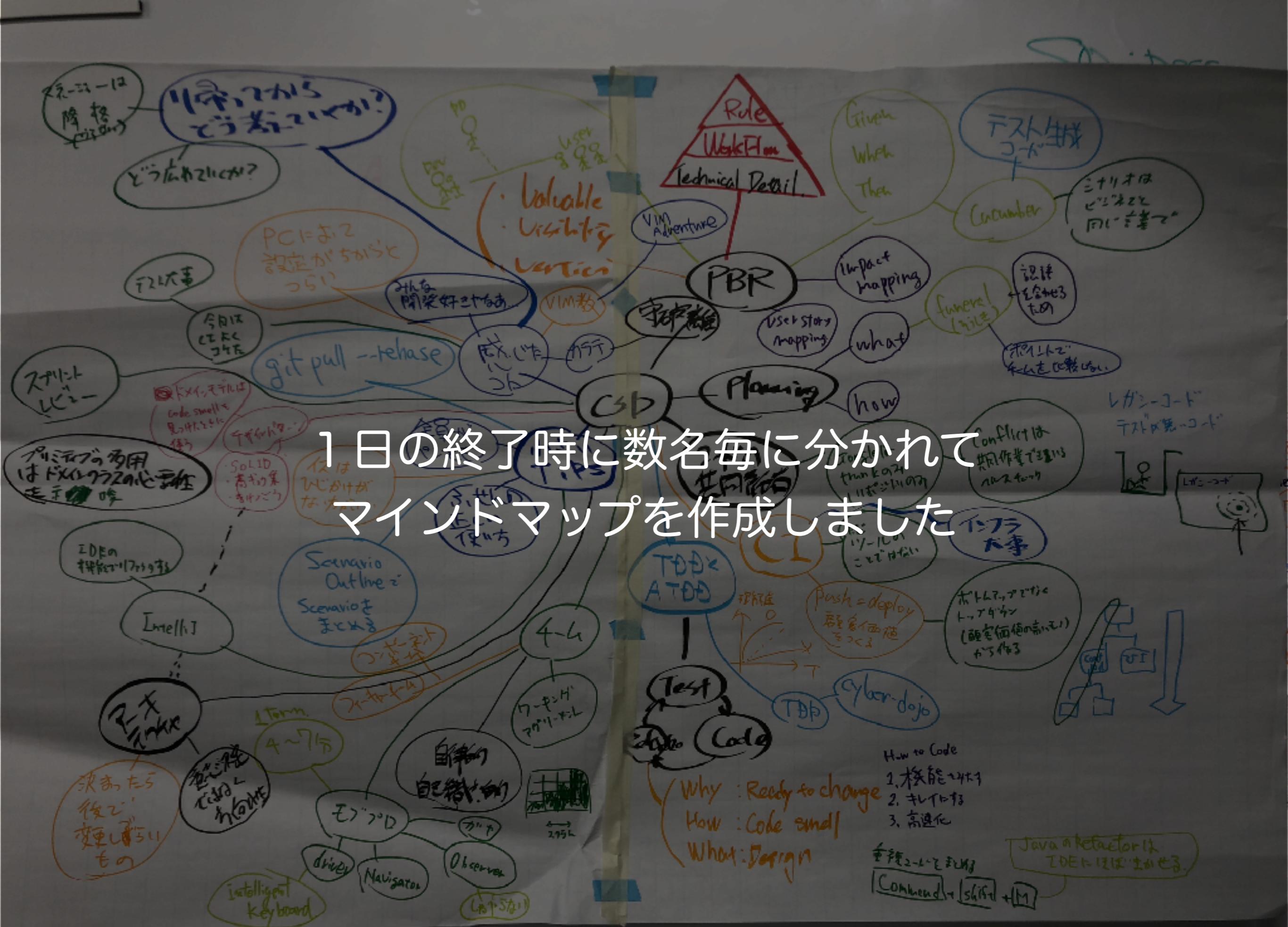
- ・参考

[https://www.publickey1.jp/blog/  
15/2045000google.html](https://www.publickey1.jp/blog/15/2045000google.html)

まとめ



1日の終了時に数名毎に分かれて  
マインドマップを作成しました



開発時、ウチのチーム(7名)は以下のように進めました

- 開発初期は不明瞭な点が多かったので、  
共通認識を持つために全員でモブプロ

- ある程度軌道に乗ったところで

3人×4人の2チーム

3人×2人×2人の3チーム

のように作業を並列化

- 次のフィーチャーに仕掛けたときは、  
また3人×4人の2チームで共通認識を持てるよう

# 学んだことまとめ

- ・ モブプロ/ペアプロは共通認識を持てるため作業効率が良い  
(ただしメッチャ疲れる)
- ・ TDDはそもそも働き方を根本的に変えなければできない  
(自分の理解が全然甘かった。逃げちゃダメだ)
- ・ TDDを身につける上で、以下のようなことが重要
  - ・ 機能を満たすシナリオを明確に書けるようになること
  - ・ リファクタリングしながらの設計スタイルに慣れること
  - ・ 継続的に価値を提供し続けるにはフィーチャーチームがオススメ

