TechBrew in 東京 ～技術的負債と共に歩むプロダクトの成長～

テストコードを負債化させない上手な付き合い方

Jan 25 2024 - Sora Ichigo

自己紹介

名前

市古 空 (Sora Ichigo)

所属

- ウォンテッドリー株式会社
- 新規プロダクト開発チーム
- **DevOps** 推進チームリード

SNS

- **X: @igsr5_**
- **GitHub: @igsr5**

自己紹介

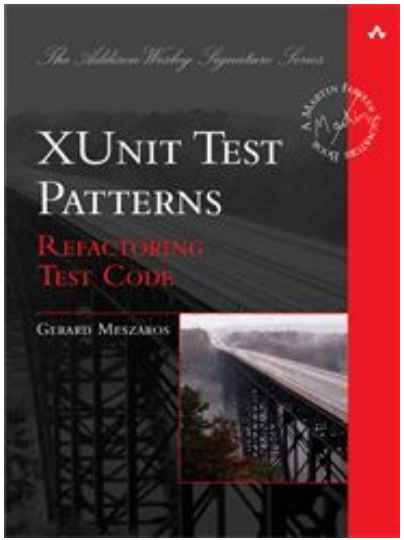
生産性とテストが好き

今日の話

「テストを書くだけで偉い」を卒業しよう👩‍🎓

「テストを書くだけで偉い」を卒業する

書籍 XUnit Test Patterns をベースに筆者の経験則を伝える



http://xunitpatterns.com/

本発表のターゲット

テストコードを

- なんとなくで書いている人
- 開発のボトルネックに感じている人

本発表のスコープ

テストコードを書きやすく保守しやすいものにする

スコープ内外　テスト自動化の目的 XUnit Test Patterns より筆者意訳

1. テストは品質向上に役立つべき
2. テストはテスト対象を理解するのに役立つべき
3. テストはリスクを減らすものであるべき
4. テストは簡単に実行できるべき
5. テストは書きやすく保守しやすくあるべき
6. テストはシステムの進化において最小限のメンテナンスであるべき

http://xunitpatterns.com/Goals%20of%20Test%20Automation.html

スコープ内

テスト自動化の目的 XUnit Test Patterns より筆者意訳

1. テストは品質向上に役立つべき
2. テストはテスト対象を理解するのに役立つべき
3. テストはリスクを減らすものであるべき
4. テストは簡単に実行できるべき
5. テストは書きやすく保守しやすくあるべき
6. テストはシステムの進化において最小限のメンテナンスであるべき

http://xunitpatterns.com/Goals%20of%20Test%20Automation.html

本題

落ちて喜べるテストを書こう

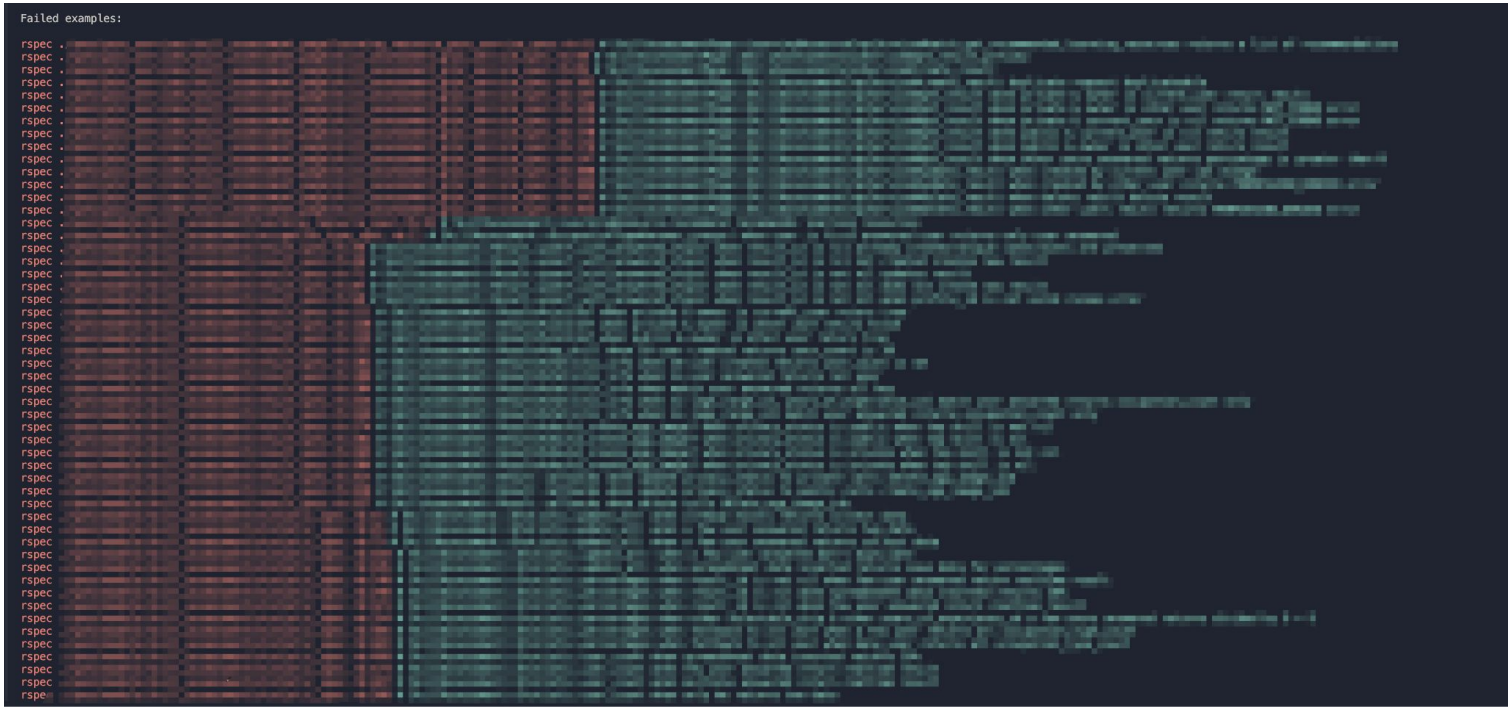
先に結論　テストが落ちて喜べる例

苗字・名前の処理が逆になってた！気付けてよかった！

```
  1) User#user_name when user is japanese returns only first_name and last_name
     Failure/Error:
       expect(user_name).to have_attributes(
         first_name: "太郎",
         last_name: "田中",
         middle_name: "",
       )

       expected #<UserName id: 16602, user_id: 8786, first_name: "田中", middle_name: "", last_name: "太
first_name => "太郎", :last_name => "田中", :middle_name => ""} but had attributes {:first_name => "田中
       Diff:
       @@ -1,4 +1,4 @@
       -:first_name => "太郎",
       -:last_name => "田中",
       +:first_name => "田中",
       +:last_name => "太郎",
        :middle_name => "",
```

先に結論　テストが落ちて喜べない例

小さい変更なのにテスト落ちすぎて途方に暮れる..



落ちて喜べるテストを書こう

1. 落ちて喜べるテストは良いテスト
2. 落ちて喜べないテストは負債になる
3. 意図の明確化・重複排除・実行結果の最適化がテスト実装のコツ

Agenda

1. 問題のあるテスト
2. 見分け方
3. 原因
4. 対処法

問題のあるテスト

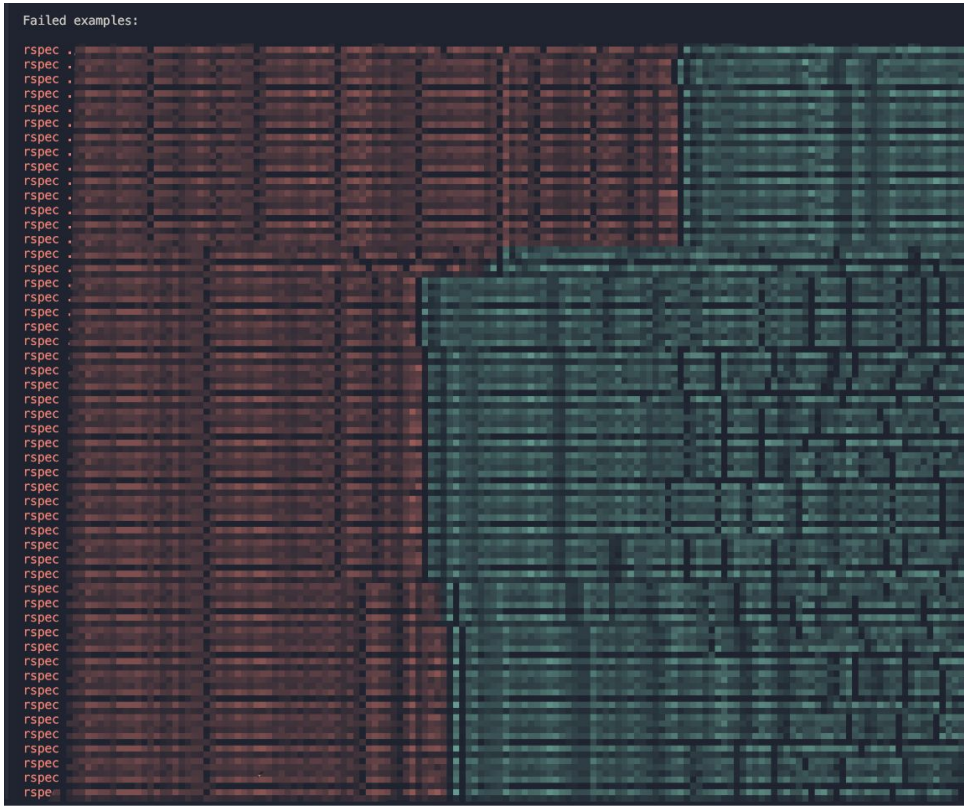
テスト = 開発プロセスの単一障害点

になっている状態

テストが負債化するとは

具体例

- 落ちても直し方が分からないテスト
- なぜ落ちたか分からないテスト
- 落ちた時の修正に時間がかかるテスト
- 偽陽性・偽陰性が高いテスト
- 頻繁に変更が発生するテスト
- モックが間違っているテスト
- テスト自体がバグっているテスト
- 実行が遅いテスト
- 確率的に落ちるテスト
- 何もしていないのに壊れるテスト
- **…etc**



低品質なテストコードは価値を生まず
費用のみ増やすため負債化しやすい

見分け方

テストが落ちて喜べるかどうか

喜べる

- これは見逃してた！
- 早くに気づけてよかった！
- やっぱり落ちるよね！

喜べない

- どう直せばいいんだ…
- なぜか大量に落ちた…
- どこでエラーになってるんだ…

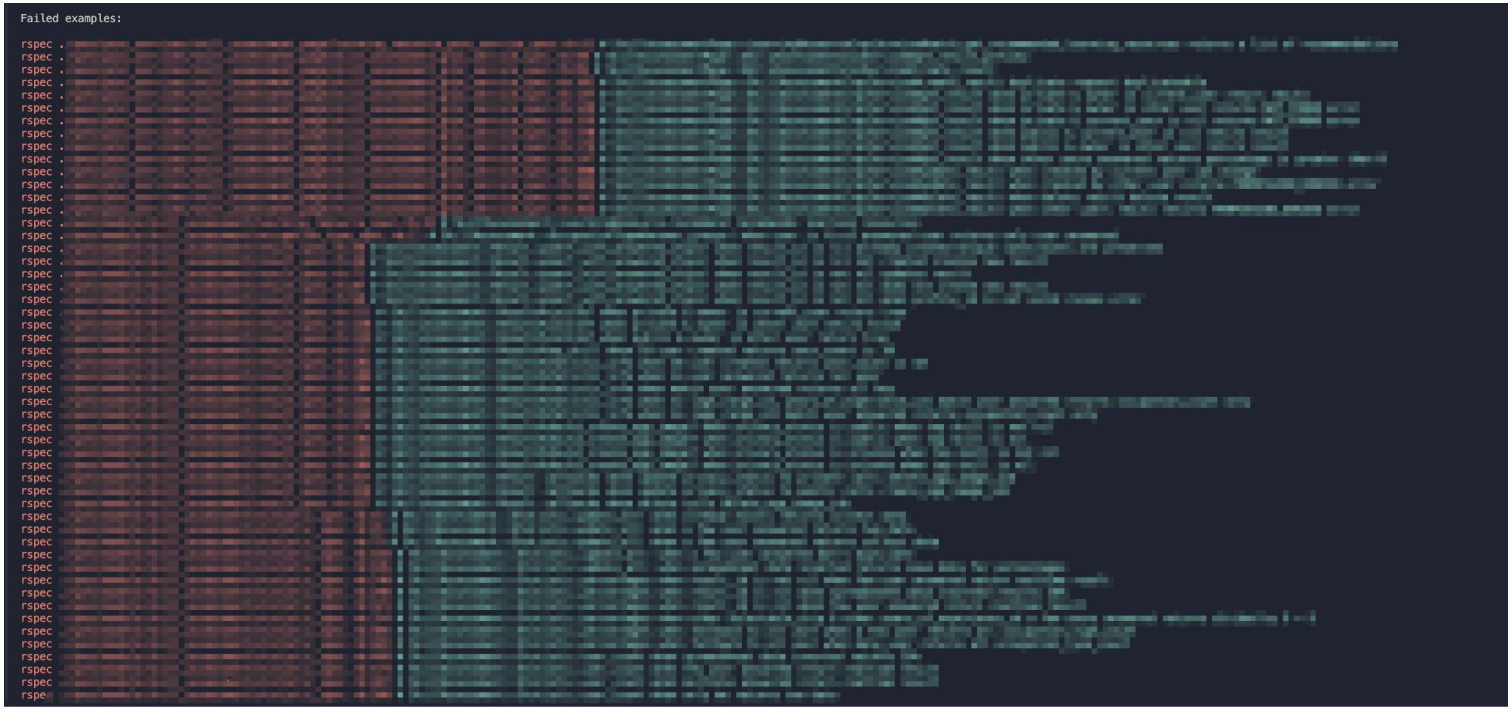
再掲　テストが落ちて喜べる例

苗字名前の処理が逆になってた..！ 気付けてよかった！

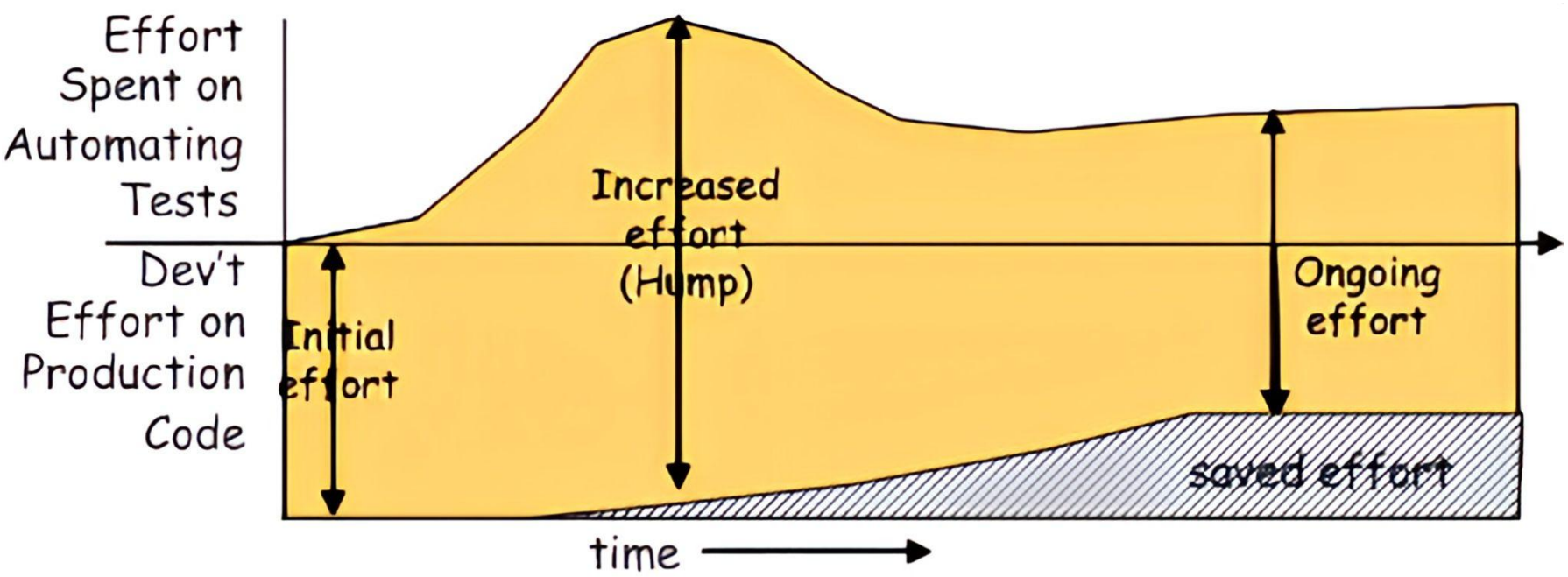
```
  1) User#user_name when user is japanese returns only first_name and last_name
     Failure/Error:
       expect(user_name).to have_attributes(
         first_name: "太郎",
         last_name: "田中",
         middle_name: "",
       )

       expected #<UserName id: 16602, user_id: 8786, first_name: "田中", middle_name: "", last_name: "太
first_name => "太郎", :last_name => "田中", :middle_name => ""} but had attributes {:first_name => "田中
       Diff:
       @@ -1,4 +1,4 @@
       -:first_name => "太郎",
       -:last_name => "田中",
       +:first_name => "田中",
       +:last_name => "太郎",
        :middle_name => "",
```

再掲　テストが落ちて喜べない例

小さい変更なのにテスト落ちすぎて途方に暮れる..

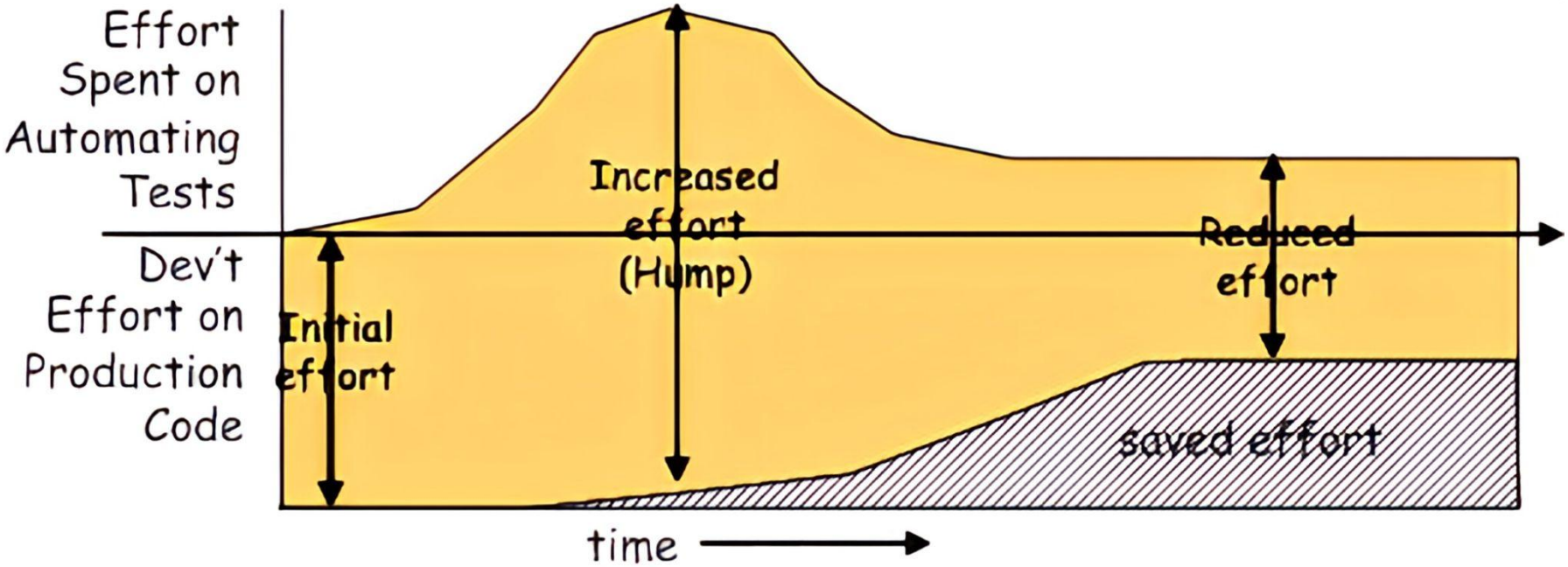


落ちて喜べないテストは対処に時間がかかる



http://xunitpatterns.com/Goals%20of%20Test%20Automation.html

理想はこちら



http://xunitpatterns.com/Goals%20of%20Test%20Automation.html

原因

テスト失敗を喜べない原因は3つ

1. テストの意図が不透明
2. テストが重複している
3. テストの失敗原因が結果に現れない

Obscure Test

1. テストの意図が不透明
    - 「どう直せばいいんだ...」のケース
    - 書籍 **XUnit Test Patterns** では **Obscure Test** と呼ばれる
2. テストが重複している
3. テストの失敗原因が結果に現れない

Test Code Duplication

1. テストの意図が不透明
2. テストが重複している
    - 「なぜか大量に落ちた...」のケース
    - 書籍 **XUnit Test Patterns** では **Test Code Duplication** と呼ばれる
3. テストの失敗原因が結果に現れない

Assertion Roulette

1. テストの意図が不透明
2. テストが重複している
3. テストの失敗原因が結果に現れない
    - 「どこでエラーになってるんだ...」のケース
    - 書籍 **XUnit Test Patterns** では **Assertion Roulette** と呼ばれる
    - ※ ただし正確には他にも原因あり(補足で後述)

対処法

工夫次第で問題は事前回避できる

1. 意図の明確化
2. 重複の排除
3. 実行結果の情報を増やす

① 何を確かめたいのか？を明確に示そう

Bad コンテキスト不足・複雑・冗長なテストが問題になりやすい

```ruby
# Bad コンテキスト不足かつ複雑
it 'successes' do # successes とは
 user = User.create(name: 'Alice', email: 'alice@example.com', password: 'secure123')
 User.confirm_email # 複雑
 User.update_last_login_time
 expect(user).to be_valid # 本当に必要？
 expect(user.email_confirmed?).to be true
 expect(user.last_login_time).to be_present
end
```

① 何を確かめたいのか？を明確に示そう

Good コンテキストが明確・シンプル・無駄のないテストを書くべき

```
let(:verified_email) { 'alice@example.com' }
let(:user) { User.create(email: verified_email) }
# Good 最低限の情報から意図が伝わる
it 'confirms user email' do
 user.confirm_email
 expect(user.email_confirmed?).to be true
end
# Good 意味のある単位で分割する
it 'updates last_login_time' do
 # ...
end
```

② 同じ確認を2回以上繰り返しても意味がない

Bad 重複したテストに対する変更は手間が大きい

```ruby
# Bad モデルスペックで確認したことをコントローラースペックでも確かめる（バリデーション）
describe 'POST #create' do
 it 'creates a new user' do
   post :create, params: { user: { name: 'Alice' } }
   expect(assigns(:user)).to be_valid # これはモデルのテスト
 end
 it 'does not create a user' do
   post :create, params: { user: { name: nil } }
   expect(assigns(:user)).not_to be_valid
 end
end
```

② 同じ確認を2回以上繰り返しても意味がない

Good テストの責務は漏れなくダブりなく

```ruby
# Good コントローラーの責務に焦点を当てたテスト
describe 'POST #create' do
 it 'redirects to the user page on successful creation' do
   allow_any_instance_of(User).to receive(:valid?).and_return(true)
   post :create, params: { user: { name: 'Alice' } }
   expect(response).to redirect_to(user_path(assigns(:user))) # コントローラーの責務
 end
 it 'renders new template on failure' do
   allow_any_instance_of(User).to receive(:valid?).and_return(false)
   post :create, params: { user: { name: 'Alice' } }
   expect(response).to render_template(:new)
 end
end
```

Bad テスト実行の学びが少ないと修正に時間がかかる

```
# Bad 全ての行が落ちる場合、1行毎にしか気づけない
it "creates a user with correct attributes" do
 user = create_user!(name: 'Alice', email: 'alice@example.com', age: 23)
 expect(user.name).to eq 'Alice' # ここで落ちると後続が実行されない!
 expect(user.email).to eq 'alice@example.com' # 全て誤りであれば3回テストを往復する必要がある
 expect(user.age).to eq 23
end
```

③ 1回のテスト実行から得られる情報を最大化しよう

Good Defect Localization (失敗原因が自ずと判明する状態)を目指す

```
# Good 一つのテストケースで確かめる事柄を1つにする
it "creates a user with correct name" do
 user = create_user!(name: 'Alice')
 expect(user.name).to eq 'Alice'
end
it "creates a user with correct email" do # 他のテストケース成否に関係なく実行される
 # ...
end
```

③ 1回のテスト実行から得られる情報を最大化しよう

Good Defect Localization (失敗原因が自ずと判明する状態)を目指す

```
# Good テストユーティリティを使う
it "creates a user with correct attributes" do
 user = create_user!(name: 'Alice', email: 'alice@example.com', age: 23)
   expect(user).to have_attributes( # attributesの差分を1度に確認できる
   name: 'Alice',
   email: 'alice@example.com',
   age: 23
 )
end
```

最後に

落ちて喜べるテストを書こう

1. 落ちて喜べるテストは良いテスト
2. 落ちて喜べないテストは負債になる
3. 意図の明確化・重複排除・実行結果の最適化がテスト実装のコツ

補足

- 時間の関係で対処法は全て紹介しきれていません
  - 残りは書籍 XUnit Test Patterns の Test Smell > Cause を見ると良いです
  - 例. Obscure Test > Cause: General Fixture
- p.25「どこでエラーになってるんだ...」については他にも原因がありえます
  - Erratic Test (確率的に落ちる)
  - Fragile Test (何もしてないのに壊れる)
  - これらのケースは対処が特殊で話すと長いのであえて割愛

宣伝

ウォンテッドリー、採用拡大中です

施策を自ら考えて改善し続けるグロースエンジニアWanted

Wantedly, Inc.

埋め込む

東京　中途　海外進出して...　知り合い +20

応募資格がありません

話を聞きに行くまでのステップ

Wantedly, Inc.のメンバー

杉本 貴昭
プロダクトマネージャー

原 剛士
Project Manager / Visit Growth Squad

川辺 慎太郎
Visit Growth Squad / Frontend Engineer

氏家 虎之介
Visit Tribe/ Visit Growth Squad

ゼロイチを繰り返して、気がつけば、売れないバンドマンからPdM / UIデザイナーになってました。
新規事業の立ち上げやリニューアル、チームの立ち上げなど、ゼロイチに関わることが多いです。
PdM、UIデザイン、Webマーケティング、SQLなど、必要なことは何でもやります。作る→届けるまでが仕事。
少人数でルールのない環境のほうが好き。

募集の特徴

オンライン面談OK

会社情報

Wantedly, Inc.

https://wantedlyinc.com
2010/09に設立
100人のメンバー
海外進出している /
社長がプログラミングできる /
1億円以上の資金を調達済み /
東京都港区白金台5-12-7　MG白金台ビル4階

宣伝

私の新卒入社エントリもあるのでよければ！(昨日公開)

新卒エンジニアが仕事に没頭したら DevOps チームが誕生しました【ウォンテッドリー 23卒入社エントリ】

市古 空
Visit Growth Squad

on 2024/01/24 589 views