第３９研究コース（アジャイルと品質）

付録

実験に至る経緯と各実験での実施状況を，録画した動画を元に要約した会話形式で記す．
(Dev：開発者，QA：QA 担当者，PO：プロダクトオーナー)

1. 実験に至るまで

Dev:酒井，QA:原田，大泉，PO: 金子

All：どうしたら PO/QA/Dev でのモブプロが実現できるか・・
Dev：同じプロダクツを見ながら会話が出来ると，認識の相違が減り効率的だが・・
PO/QA：ソースコードを一緒に見ても，なかなか理解できない・・
Dev：記法の工夫をすることで，PO/QA により寄り添うことができるのかもしれない．
　　テストシナリオを自然言語で記載する記法があるので，採用してみよう！
QA：振舞い駆動開発（BDD）の考え方ですね．
PO：ToDo リスト(アプリ)を作成するという前提で，
　　実際にテストを実施してみよう！モックはざっと
　　こんな感じかな(図 1)．
All：いいね．これで進めよう．

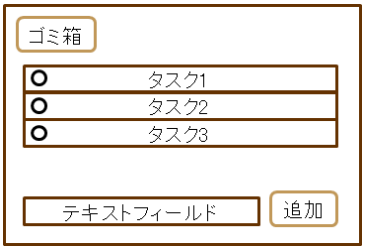

図 1 ToDo リストイメージ

2. 実験１　Gherkin 記法を使ってみた

Dev:酒井，QA: 原田，大泉，PO:金子

De：BDD でよく使われる記法に RSpec 記法と Gherkin 記法がある．
　　RSpec 記法では describe/it/expect の構文になっていて，it のあとにテスト内容を
　　日本語で記載できる．厳密性があまり高くなく，比較的柔軟な記述ができそうだ．
　　(図 2)
PO/QA：テストコード中のコメントとして記述するので，読みにくいかもしれない．

```
RSpec.describe '計算' do
  it '1 + 1 = 2 になること' do
    expect(1 + 1).to eq 2
  end
  it '2 - 1 = 1 になること' do
    expect(2 - 1).to eq 1
  end
end
```

図 2 RSpec 記述イメージ

```
Feature: 計算
  計算結果が正しいことを確認する

  Scenario: 1 + 1 の計算結果が2であること
    Given 1と1を加算する
    When 計算を実行する
    Then 結果は2である

  Scenario: 2 - 1 の計算結果が1であること
    Given 2から1を減算する
    When 計算を実行する
    Then 結果は1である
```

図 3 Gherkin 記法 記述イメージ

Dev：Gherkin 記法はシナリオには仕様に該当する要件，Given/When/Then 等には詳細条件
　　を記述していく．(図 3)
PO/QA：構文がとてもシンプルなので，何を実行しているか理解し易そう．また，シナリオ
　　は仕様書としても活用できそうですね．
All：では，Gherkin 記法でテストをモブプロしてみよう！

```
シナリオ: 新しいタスクを追加
  前提 タスク入力フィールドが表示されている
  もし "牛乳を買う" というテキストをタスク入力フィールドに入力する
  かつ "追加" ボタンをクリックする
  ならば タスク一覧に "牛乳を買う" が表示される

シナリオ: タスクを削除できる
  前提 タスク一覧に "牛乳を買う" が表示されている
  もし "牛乳を買う" というタスクを左のラジオボタンを選択し
  かつ "ゴミ箱" ボタンをクリックする
  ならば "牛乳を買う" タスクが削除される

####

# バックログ
# - タスクの内容を変更できる
# - 削除
# やってる最中に気づいた
# - 最大文字数の制限
# - ファイルの添付
# - 一定時間はアンドゥできる
```

図 4 テストシナリオ記述例

```
@when('"追加" ボタンをクリックする')
def step_when_click_add(context):
    context.viewmodel.add_task()

@then('タスク一覧に "{task_text}" が表示される')
def step_then_task_added(context, task_text):
    tasks = context.viewmodel.get_tasks()
    assert task_text in tasks

@given('タスク一覧に "{task_text}" が表示されている')
def step_given_task_displayed(context, task_text):
    context.viewmodel.model.add_task(task_text)
    assert task_text in context.viewmodel.get_tasks()

@when('"{task_text}" というタスクを左のラジオボタンを選択し')
def step_when_select_task(context, task_text):
    context.viewmodel.select_task(task_text)

@when('"ゴミ箱" ボタンをクリックする')
def step_when_click_delete(context):
    context.viewmodel.delete_task()

@then('"{task_text}" タスクが削除される')
def step_then_task_deleted(context, task_text):
    tasks = context.viewmodel.get_tasks()
    assert task_text not in tasks
```

図 5 テストコード記述例

All：まずはプロダクトバックログアイテム(PBI)に相当するシナリオを洗い出してみよう．
All：次はシナリオ毎に詳細条件を追記していこう．思ったより細かく設定が必要で，出し切るのは大変だ・・．時間がかかる．

（PO/QA が条件を洗い出している間，Dev はテストコードを平行して作成.）（図 4）

Dev：テストコードにして実行すると・・(図 5)

Dev：エラーが出た・・・インスタンスがないといっている．「フィールドが表示されている」という以前に「アプリが起動している」という大前提の条件が足りないようだ．
Dev：前提条件が適切でないため書き換えたほうがよいかもしれない・・シナリオの前提条件の定義が細かく必要となるため，事前準備をしっかりしないと厳しい．
Dev：厳密性が高いため，慣れていないメンバーの場合，記述負荷が高いかもしれない．
Dev：気付いたら，テストをパスさせることばかり気にしてしまい，テストをパスさせるための記述を書かないと，という意識でいっぱいになってしまう！

（テストがなかなか通らず焦る Dev.）

PO：テスト仕様を記述しながらシナリオの条件も変更されると，少し混乱する．PBI 通りに入力して欲しいが，合っているのか段々わからなくなってきた・・．
QA：仕様に沿ったテストコードの記述に時間がかかりすぎる．仕様の確認をしたいのだけど，そこまで辿り着かない・・

（ここで実験 1 終了）

実験 1 の振り返りは以下の通り．

- ビューの表示等はテストに含めず，今回は View と Model の IF テストと限定したい．
- オーバーオールでテストできると良いが，どのテストで何をテストするか，という点を見定めてテストを実施することは QA のチャレンジでもある．
- 今回は単純すぎるため単体テストのようになって見える．もっとスムーズにテストコードが進み，あるべきストーリーベースのテストに辿りついて，やっと本来の QA の出番（仕

様を満たしているか，異常系も問題ないかを確認する）になる，という感触であった．
- テスト結果を受けて，シナリオ以下の条件文が追記・変更されていった．シナリオに関する条件が変化していくことに違和感を感じた．この流れが PO に一般的に受け入れてもらえるかが疑問である
- PBI を定めてからテストするが，PBI の内容はふわっとしていた．PBI 間の齟齬を見つけることはテストを進めると可能となるはず．
- ビジネスロジックの妥当性テストは PO が入るとより効果的になるため，フロント/バックエンドではなく要求に近いテストを実施したかった．
- ユースケースは PO，ToDo は QA，テストコードは Dev が記載するのがよい流れである．

3. 実験２　実例マッピングを参考にしてみた

Dev：酒井，QA：大泉，金子，PO：芳沢

Dev:BDD のプロセスの中で紹介されている実例マッピングを参考にして，バックログを掘り下げていくと，PO/QA/開発者間の認識の齟齬を減らせそうだ．（図 6）

図 6 実例マッピングイメージ

PO：やってみよう．まずは PBI に相当するストーリーを数件挙げていきます．「タスクを追加する」，「タスクを削除する」，・・・

QA：「連打しても変な挙動にならない」・・

Dev：えっ！　なんていうか，やさしくテストして貰えると嬉しいな・・

PO：ストーリーに合わせて実際の動きをルールに落とし込むと・・（図 7）

Dev：あ，連打しても大丈夫にするには，ボタンの活性化/非活性化も追記しないと．

QA：ルール１の下に実例として記載しましょう．（図 8）

QA：PBI 間の整合がとれ，見落としも防げていいですね．

Dev：PO/QA/Dev で共通理解が深められて嬉しいです．

All：この調子でストーリー/ルール/実例を考えていこう！！

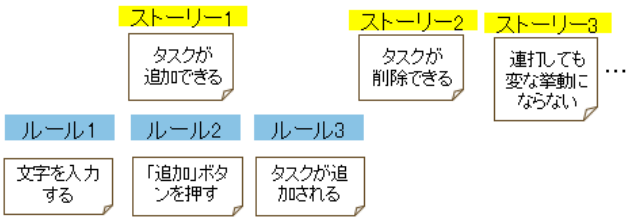

図 7 ストーリー/ルール

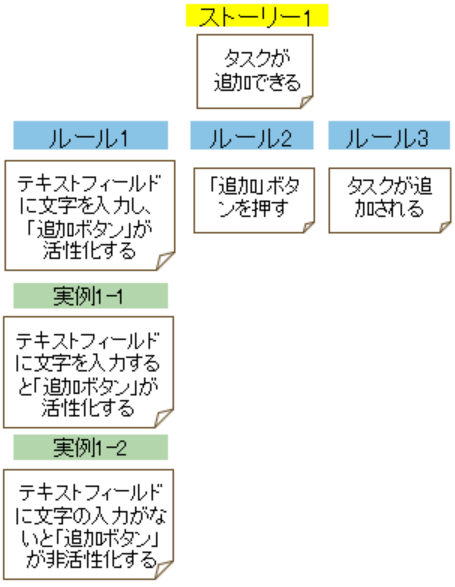

図 8 ストーリー/ルール/実例

Dev：そして，実例マッピングの「ストーリー」，「ルール」，「実例」に合わせたテストコードが Python で書けることが分かったので，試してみます．言語やフレームワークに依存しないテスト方法です．（図 9）

```
43    # Story: TODOリストにタスクを追加する   ストーリー1
44
45    class Rule_追加ボタンはテキストフィールドにテキストが入力されている場合に活性化する(unittest.TestCase):   ルール1
46        def test_テキストフィールドに1文字入力された場合_追加ボタンが活性化する(self):   実例1-1
47            model = TodoListModel()
48            model.enter_text('A')
49            self.assertTrue(model.add_button_enabled)
```

図 9 テストコード例

QA：RSpec に似ている書き方で，柔軟な記載ができますね．
PO：PBI との対比もとれるので分かりやすい．
All：このやり方で試してみよう！！

Dev：テストモデルを準備しました．（図 10）

```
3  + # Model for a simple TODO list application
4  + class TodoListModel:
5  +     def __init__(self):
6  +         self.items = []
7  +         self.add_button_enabled = False
8  +
9  +     def enter_text(self, text):
10 +         if text:
11 +             self.add_button_enabled = True
12 +         else:
13 +             self.add_button_enabled = False
14 +
15 +     def add_item(self, item_name):
16 +         if self.add_button_enabled:
17 +             self.items.append(item_name)
18 +             self.add_button_enabled = False  # Reset the button state
```

図 10 テストモデル（実験開始時）

Dev：では，早速「タスクを 1 つ追加する」をテストします．
PO/QA：はい．（ドキドキ）
Dev：テストコードを記述していきます．
PO：Class でルールを定めて，def に実例を記載していくんだったな．
QA：なんとなく，どんな処理が書かれているか想像できそうだぞ．

```
21 + # Story: TODOリストにタスクを追加する
22 +
23 + class Rule_追加ボタンはテキストフィールドにテキストが入力されている場合に活性化する
   (unittest.TestCase):
24 +     def test_テキストフィールドに1文字入力された場合_追加ボタンが活性化する(self):
25 +         model = TodoListModel()
26 +         model.enter_text('A')
27 +         self.assertTrue(model.add_button_enabled)
28 +
29 +     def test_テキストフィールドが空の場合_追加ボタンが非活性になる(self):
30 +         model = TodoListModel()
31 +         model.enter_text('')
32 +         self.assertFalse(model.add_button_enabled)
```

図 11 タスクを追加するテスト（その 1）

Dev：あ，どのメソッドを使ったらよいかすぐに思いつかない・・
ちょっと分からないところを調べてみます．（検索中もずっと見られていて緊張するな・・）

（時間が少し経過）

Dev：テストコード作成完了しました．テストします．

```
+ class Rule_追加ボタンを押下すると項目が追加される(unittest.TestCase):
+     def test_追加ボタンを押下するとTODOリストにタスクが追加される(self):
+         model = TodoListModel()
+         model.enter_text('New Task')
+         model.add_item('New Task')
+         self.assertIn('New Task', model.items)
+
+
+ class Rule_テキストフィールドに入力されたテキストがタスク名となる(unittest.TestCase):
+     def test_テキストフィールドにAAAを入力し追加ボタンを押下すると追加されたタスク名はAAAとなる
  (self):
+         model = TodoListModel()
+         model.enter_text('AAA')
+         model.add_item('AAA')
+         self.assertIn('AAA', model.items)
```

図 12 タスクを追加するテスト(その 2)

PO/QA：はい．（緊張が走る）
Dev：テスト通りました．（ふぅと一息）
PO/QA：よかった！！（待ちが少しあったけど，まぁこんなものか・・・）

Dev：では次は，「タスクを 1 つ削除する」をテストします．テストコードを記述します．

```
+ # Story: TODOリストのタスクを削除できる
+
+ # - ゴミ箱ボタンを押下すると選択されたタスクが削除される
+ class Rule_ゴミ箱ボタンを押下すると選択されたタスクが削除される(unittest.TestCase):
+     def test_タスク2を選択しゴミ箱ボタンを押下_タスク2が削除される(self):
+         model = TodoListModel()
+         model.items = ['Task1', 'Task2', 'Task3']
+         model.select('Task2')
+         model.push_trash_button()
+         self.assertNotIn('Task2', model.items)
```

図 13 タスクを削除するテスト

Dev：テストコード作成完了しました．テストします．
PO/QA：はい．
Dev： テスト通りません・・調査します．

Dev： モデルの変更も必要そうだ．修正範囲が広くなると時間がかかるな・・(焦る・・・)

（時間が大分経過）
（Dev が何回かコード修正してテストするも中々通らず）

QA：(申し訳ないが，テストコード作成時の待ちを長く感じてしまう・・)
QA：(協力したくても，分からないから口出しできないし・・・)
PO：(最初にストーリー出してから，ほとんど出番がないような・・・帰りたい.)

Dev： すみません，後でゆっくり時間をかけて変更したいので，モブは一旦終了で良いでしょうか・・・

PO/QA：わ，わかりました．大丈夫ですか？？
Dev：終始見られていることによる緊張と，待たせないよう急がないと，という気持ちでストレスが凄まじいです．
PO/QA：本当にお疲れ様です．休んでください．

（ここで実験 2 終了）

QA：アプローチ自体の流れは良かったので，あとはコーディングの負荷を如何に下げられるか，なんですが何かいい方法がないか・・・・・
QA：では生成 AI をテストコーディングのサポートに使ってみてはいかがでしょうか？
All：いいですね，やってみましょう！！

Dev：ではモデルの変更案を生成 AI(ChatGPT4)に聞いてみます．
しっかり返答内容を確認して，微修正を加えて・・・

```
5    # Model for a simple TODO list application
6  +
7    class TodoListModel:
8        def __init__(self):
9            self.items = []
10 +          self.selectedItemIndex = None
11           self.add_button_enabled = False
12
13       def enter_text(self, text):
14 +          self.add_button_enabled = bool(text)
15
16       def add_item(self, item_name):
17           if self.add_button_enabled:
18               self.items.append(item_name)
19               self.add_button_enabled = False  # Reset the button state
20
21 +     def select(self, selected_item_name):
22 +         try:
23 +             self.selectedItemIndex = self.items.index(selected_item_name)
24 +         except ValueError:
25 +             self.selectedItemIndex = None
26 +
27 +     def push_trash_button(self):
28 +         if self.selectedItemIndex is not None and self.selectedItemIndex < len(self.items):
29 +             self.items.pop(self.selectedItemIndex)
30 +             self.selectedItemIndex = None
```

図 14 テストモデル(実験 2 終了時)

Dev：テスト通りました！生成 AI のサポートがあれば，やれる気がしてきました！
All：良かった！！では，生成 AI を使って，再度実験を行ってみましょう．

4. 実験 3 生成 AI をコーディングのサポートに活用してみた

Dev：酒井，QA：大泉，金子，PO：芳沢

QA：今までは具体的に追加や削除のタスク数について明示していないので・・
「タスクが３つある場合，３つのタスクを全て選択し削除する」というテストを行いたいと思います．
QA：いいですね．表示タスクが 0 の挙動も併せて確認しましょう．

Dev：テストコードを記述します．えっと，全てのタスク削除後，リストが空であることを確認するメソッドが分からない・・
Dev:メソッドを ChatGPT に聞いてみます.(回答の中から適切な案を選択すればよいから，決めやすい)
Dev：テストコード完成したので，テストします．
PO/QA：はい．(実験 2 より速い!!)

```
75    # - タスクが3つ存在する場合タスクを3つ選んでゴミ箱ボタンを押下するとタスクが全て消える
76  + class Rule_複数のタスクを選択しゴミ箱を押下すると複数のタスクが削除される(unittest.TestCase):
77  +     def test_タスク1から3を選択しゴミ箱ボタンを押下_タスク1から3が削除される(self):
78  +         model = TodoListModel()
79  +         model.items = ['Task1', 'Task2', 'Task3']
80  +         model.select('Task1')
81  +         model.select('Task2')
82  +         model.select('Task3')
83  +         model.push_trash_button()
84  +         self.assertEqual(len(model.items), 0)
```

図 15 リスト上のタスク(3 つ)を全て削除するテスト

Dev：タスクの削除を実行しているのにタスクが全て消えないです・・・
あ，モデルが複数選択に対応していない！
だからタスクが 1 つしか消えず，2 つは消えずに残っていたのか．
QA：なるほど．モックの絵(図 1)のまま，ラジオボタンでモデルを作っていたのですね．
PO：これはチェックボックスで複数選択が正しいです．変更をお願いします．
QA：意思決定がタイムリーなので素早い対応が可能になりますね．
Dev：QA 視点のテストシナリオで仕様(モデル)変更に早く気付くことが出来きました！自分では思いつかないシナリオを，すらすら出してくれて助かります．
All：これがモブでやる効果なのか!!（あとはコーディングの速さ・・・）

(Dev，呼吸を整える)

Dev：メソッドを複数選択可能にして，と ChatGPT に聞いてみます．
修正コード案がすぐ出てきた．リスト構造にするのがやはりよいか・・
よし，この記述方法を採用しよう．(コピペして修正を加える)
複数選択したときのインデックスが持てるような構造に変えました．
PO/QA：修正が格段に早くなっている!!

```
7     class TodoListModel:
8         def __init__(self):
9             self.items = []
10    +       self.selectedItemIndices = []  # 複数選択用のリスト
11            self.add_button_enabled = False
12
13        def enter_text(self, text):
14
15
16        def add_item(self, item_name):
17            if self.add_button_enabled:
18                self.items.append(item_name)
19    +           self.add_button_enabled = False
20
21        def select(self, selected_item_name):
22            try:
23    +           index = self.items.index(selected_item_name)
24    +           if index in self.selectedItemIndices:
25    +               self.selectedItemIndices.remove(index)
26    +           else:
27    +               self.selectedItemIndices.append(index)
28            except ValueError:
29    +           pass
30
31    +   def push_trash_button(self):
32    +       for index in sorted(self.selectedItemIndices, reverse=True):
33    +           if index < len(self.items):
34    +               self.items.pop(index)
35    +       self.selectedItemIndices = []
```

図 16 テストモデル(実験 3 終了時)

Dev：では，テストします.
PO/QA：はい．(緊張が走る)
Dev：テスト通りました．(ガッツポーズ！)
All：いい流れですね！この調子でどんどんテストを回していこう！

QA：次は「タスクが選択されていない時はゴミ箱ボタンが非活性になる」のテストを！
QA：「削除したタスクを復元する」のテストもやりたいです！
Dev：じゅ，順番にやりますから落ち着いてください～
PO：仕様に関する疑問はその場で解決していこう！（あれ，タスク復元は可能だったかな・・？）

（テストを高速に繰り返す・・・）

Dev：そろそろリファクタしないと～！！　生成 AI，リファクタお願い～！！

完