wellbehaved

Выпуск 0.1.2.0

19 January 2017

Оглавление

Содержание:

Глава 1

BDD и мы

1.1 Введение

На протяжении всего времени, что я работаю в этой компании в воздухе витает дух тестирования. Стремление сдвинуть стрелку code coverage с 0% и с тем постепенно повышать надежность ПО знакомо каждому разработчику, и многие из них с пеной у рта готовы доказывать своим менеджерам о необходимости подобного подхода.

Обычно пропагандируется использование подхода TDD (test-driven development): сначала мы пишем тесты, затем уже пишем код приложения и путем прогона тестов выясняем, какие части приложения работают не так, как мы хотим. По сути, мы делаем ставку на равенство суммы корретно работающих компонент корретно работающему приложению. На практике же про этот подход вспоминают слишком поздно и применение его сводится к покрытию тестами уже существующего кода. Но даже перед этим, им необходимо "продать" менеджеру продукта идею выделения части оплачиваемого разработчикам времени на написание тестов.

Вот тут и начинаются до боли знакомые вопросы...

- "Как мы это сможем продать?"
- "Какая нам от этого польза?"
- "Зачем тратить время на старый код, когда мы можем запилить много нового?!"

И они по-своему верны.

Для примера, рассмотрим классический случай теста на основе unittest:

```
def test_group_counter(self):
    ...
    self.failUnlessEqual(self.grup1.pupil_set.count(), self.pupil_in_gr )
    self.failUnlessEqual(Group.objects.count(), self.group_count)
    self.failUnlessEqual(Pupil.objects.count(), self.pupil_in_gr * self.group_count)
    ...
```

Если вы покажете такое менеджеру, то он вряд ли сможет разобрать хоть что-то. С другой стороны, разработчик системы без всяких проблем сможет увидеть в этих строчках проверку вычисления определенных внутренних аттрибутов объекта и их соответствие с ситуацией в БД.

И вот здесь заключается одна из основных проблем работы по TDD: навыками написания и чтения тестов владеет только разработчик, менеджер и аналитик выключены из процесса. Соответственно, им трудно оценить приносимую тестами пользу.

Это приводит к целому ряду негативных последствий:

- Проблемы в общении между членами команды
- Разные цели в тестировании продукта: разработчик думает про корректность кода, менеджер душой болеет за корректность продукта
- Покрытие тестами представляется менеджеру как бесполезная трата времени

1.2 BDD и с чем его едят

Описанные выше проблемы привели к разработке нового подхода, во многом базирующегося на идеях TDD - BDD (behaviour driven development). Он объединяет основные принципы и техники BDD с идеями из доменно-ориентированного проектирования для предоставления аналитикам и разработчикам единого инструмента для взаимодействия в процессе разработки продукта.

Ниже приведен один из примеров теста, написанного с использованием BDD:

```
Функция: вычисление очереди
       Предыстория: имеется база с учреждением и заявлениями в разных статусах
               Дано имеется учреждение "ДОУ №1" (вид: "ДОУ")
               И имеются заявления
                       | Желаемый ДОУ | Статус заявления     | Дата и время подачи | Дата выбора ДОУ     |
                       | ДОУ №1           | Зарегистрировано     | 01.01.2011 01:02:03 | 04.05.2011 01:00:00 |
                       | ДОУ №1           | Направлен в ДОУ      | 02.01.2011 01:02:03 | 05.06.2012 02:00:00 |
                       | ДОУ №1           | Подтверждение льгот  | 02.01.2011 01:02:03 | 06.07.2013 03:00:00 |
               И диапазон возрастов от 3 до 7 лет
               И установлены стандартные параметры расчета очереди
               И очередь считается по учреждению "ДОУ №1"
       Сценарий: фильтрация заявлений
               Когда вычисляется очередь
               То в очередь попадают только заявления в статусах
                       | Статус заявления    |
                       | Зарегистрировано    |
                       | Подтверждение льгот |
                       | Выбор желаемого ДОУ |
                       | Желает изменить ДОУ |
               И у заявлений в очереди нет направлений в статусах
                       | Статус направления  |
                       | Предложено системой |
                       | Зачислен                        |
                       | Не явился               |
                       | Желает изменить ДОУ |
       Сценарий: сортировка очереди по тульской модели комплектования
               Когда включена тульская модель комплектования
               И вычисляется очередь
               То заявления отсортированы по возрастанию даты выбора ДОУ
```

Как можно заметить, тест стал гораздо проще для чтения и понимания всеми участниками проекта. Теперь уже не нужно знать тонкости технической реализации внутренних механизмов системы для написания тестов и этим может заниматься кто-то кроме разработчика. Во многом это достигается использованием “единого языка”, полуформального набора терминов для описания бизнес-логики системы. Этот набор разрабатывается совместно разработчиками и аналитиками, устраняя двойственные трактовки одних и тех же аспектов бизнес-логики.

Формат описания теста напоминает описание User Story, знакомую использующим Agile командам. Действительно, они выполняют схожую функцию - описание поведения, которое представляет ценность для конечного клиента. Также как и в User Story здесь присутствуют Feature (“Функция” и

“Предыстория”) и Acceptance Criteria (“Сценарий”, “Дано-Когда-То”), помогающие описать требуемое поведение и условия корретности его реализации.

Но как этот человекочитаемый текст транслируется в проверяющий условия код? За это отвечает файл с “шагами”:

```
@given(u'диапазон возрастов от {start_age} до {end_age} лет')
def get_age_range(context, start_age=0, end_age=7):
    context.ages = (float(start_age), float(end_age))
...
@when(u'вычисляется очередь {queue_type}')
def calculate_specific_queue(context, queue_type):
    types = {
        u'сводная': -1,
        u'общая': 1,
        u'льготная': 2,
        u'переводников': 3
    }
    context.type = types[queue_type]
    context.queue = DeclarationQueue(context.queue_ctx)
    context.queue_decls = context.queue.get_list()[0]
...
@when(u'вычисляется очередь')
def calculate_default_queue(context):
    context.queue = DeclarationQueue(context.queue_ctx)
    context.queue_decls = context.queue.get_list()[0]
...
@then(u'в очередь попадают только заявления в статусах')
def check_declaration_selection(context):
    allowed_statuses = [row[u'Статус заявления'] for row in context.table]
    for decl in context.queue_decls:
        assert decl['status__name'] in allowed_statuses, \
              u'Status "%s" not allowed in queue!' % decl['status__name']
```

Каждый шаг в написанном тесте должен соответствовать одному обработчику. Разделение кода выполнения теста облегчает дальнейшую компоновку шагов в сценарии, позволяя также использовать их многократно в качестве библиотеки стандартных действий.

Глава 2

Gherkin

(здесь и далее - перевод и легкая адаптация статьи [”Writing Features - Gherkin Language”][Writing Features])

Gherkin - человеко-читаемый язык для описания поведения системы, который использует отступы для задания структуры документа, (пробелы или символы табуляции). Каждая строчка начинается с одного из ключевых слов и описывает один из шагов.

Пример:

```
Функция: Короткое, но исчерпывающее описание требуемого функционала
    Для того, чтобы достичь определенных целей
    В качестве определенного участника взаимодействия с системой
    Я хочу получить определенную пользу

    Сценарий: Какая-то определенная бизнес-ситуация
        Дано какое-то условие
        И ещё одно условие
        Когда предпринимается какое-то действие участником
        И им делается ещё что-то
        И вдобавок он совершил что-то ещё
        То получается какой-то проверяемый результат
        И что-то ещё случается, что мы можем проверить
```

Обработчик разбивает файл с тестами на функции, сценарии и входящие в них шаги. Давайте разберем этот пример:

1. Строка **Функция: Короткое, но исчерпывающее описание требуемого функционала** начинает собой описание функционала и дает ему название.
2. Следующие три строчки не обрабатываются и не несут никакой смысловой нагрузке для обработчика тестов, но они задают контекст тестирования и одновременно описывают, какую пользу мы получим от этого функционала.
3. Строка **Сценарий: Какая-то определенная бизнес-ситуация** начинает сценарий и содержит его описание.
4. Следующие 7 строчек описывают шаги теста, каждому из которых впоследствии будет сопоставлен определенный программный код, выполняющий описанное действие. Сопоставлению подлежат части строк лежащие после ключевых слов “Дано”, “И”, “Когда” и т.д.

2.1 Функции (Feature)

Каждая функция описывается в отдельном файле с расширением *.feature*. Первая строчка должна начинаться с ключевого слова "Функция:", за которой могут идти три строчки с описанием, размеченные отступами. Каждая функция обычно состоит из списка сценариев.

Каждый сценарий состоит из списка *шагов*, каждый из которых должен начинаться с одного из ключевых слов:

- Дано
- Когда
- То
- Но
- И

Шаги *"Но"* и *"И"* существуют исключительно для удобства чтения и по своим функциям повторяют ключевое слово, с которого начиналась предыдущая строчка.

Вдобавок к сценариям, описание функционала может также содержать *структуры сценариев* и *предыстории*.

2.2 Сценарий (Scenario)

Сценарий представляет собой одну из ключевых структур в языке *Gherkin*. Каждый сценарий начинается с ключевого слова **"Сценарий:"**, и может содержать в себе название сценария. Описание функционала может содержать в себе один или больше сценариев, и каждый сценарий состоит из одного или более шага.

Каждый из следующих сценариев содержит три шага:

```
Сценарий: Вася создает новую запись
    Дано я вошел в систему как Вася
    Когда я пытаюсь добавить запись в справочник "Лекарства"
    То мне должен быть ответ "Ваша запись успешно добавлена."

Сценарий: Вася не может добавлять запись в справочник лечений
    Дано я вошел в систему как Вася
    Когда я пытаюсь добавить запись в справочник "Виды лечений"
    То мне должен быть ответ "У вас нет прав доступа!"
```

2.3 Структура сценария (Scenario Outline)

Достаточно часто приходится писать множество мелких сценариев, которые различаются буквально парой переменных. Эти повторения могут быстро надоесть:

```
Сценарий: удалить 5 записей из 12
    Дано есть 12 записей
    Когда я удаляю 5 записей
    То у меня должно остаться 7 записей

Сценарий: удалить 5 записей из 20
    Дано есть 20 записей
```

```
    Когда я удаляю 5 записей
    То у меня должно остаться 15 записей
```

Структуры сценариев позволяют нам более кратко описывать подобные наборы сценариев с помощью шаблонов:

```
Структура сценария: удаление записей
    Дано есть <было> записей
    Когда я удаляю <удалено> записей
    То у меня должно остаться <остаток> записей

    Примеры:
        | было  | удалено | остаток |
        | 12    |    5    |   7     |
        | 20    |    5    |   15    |
```

Шаги указанные в структуре сценария не выполняются напрямую, но используются для подстановки в них значений из таблицы примеров. Каждая строчка таблицы будет обрабатываться как отдельный сценарий с указанными значениями вместо заглушек “было”, “удалено” и “стало”.

2.4 Предыстории (Background)

Предыстории позволяют вам добавить определенный контекст ко всем сценариям в пределах функции. По сути, предыстория - сценарий без имени, состоящий из шагов. Основное отличие в запуске: предыстория запускается перед каждым сценарием:

```
Функция: поддержка многих справочников

    Предыстория:
        Дано есть пользователь с именем "Вася"
        И есть справочник "Лекарства"
        И у пользователя "Вася" есть право на запись в  "Лекарство"
        И есть справочник "Виды лечений"

    Сценарий: Вася создает новую запись
        Дано я вошел в систему как Вася
        Когда я пытаюсь добавить запись в справочник "Лекарства"
        То мне должен быть ответ "Ваша запись успешно добавлена."

    Сценарий: Вася не может добавлять запись в справочник лечений
        Дано я вошел в систему как Вася
        Когда я пытаюсь добавить запись в справочник "Виды лечений"
        То мне должен быть ответ "У вас нет прав доступа!"
```

2.5 Шаги

Функции состоят из шагов, также известных как *Данные*, *Действия* и *Результаты*.

2.5.1 Данные (Givens)

Назначение шагов *Дано* состоит в **приведение системы в известное состояние** перед тем как пользователь (или внешняя система) начнет взаимодействие с системой (в шагах *Когда*). Также можно рассматривать их как предусловия.

Пример: создавать объекты сущностей или настраивать БД

```
Дано нет пользователей в базе
Дано база данных пустая
```

Пример: вход пользователя в систему (исключение к правилу “никаких взаимодействий в шаге Дано”)

```
Дано я вошел в систему как "Вася"
```

2.5.2 Действия (Whens)

Назначение шагов *Когда* состоит в **описании ключевого действия, совершаемого пользователем**.

Пример: взаимодействие со страницей

```
Когда я открыл форму добавления учреждения
Когда я ввел "Институт радости" в поле "Наименование"
Когда я выбрал в поле "Тип" значение "Институт"
Когда я нажал на кнопку "Сохранить"
```

2.5.3 Результаты (Thens)

Назначение шагов *То* состоит в **наблюдении результатов выполнения действий**. Наблюдения должны быть связаны с явной пользой, которая указаны в описании функции. Также необходимо помнить, что должен проверяться *вывод системы* (отчеты, интерфейс, сообщения), а не что-то глубоко закопанное в систему.

2.5.4 Предлоги (And, But)

Если у вас есть несколько шагов *Дано*, *Когда*, или *То* то вы можете писать так:

```
Сценарий: множественные данные
    Дано что-то первое
    Дано что-то второе
    Дано и что-то ещё
    Когда я открою свои глаза
    То я увижу что-то
    То чего-то я не увижу
```

...или можете использовать шаги *И* и *Но*, превращая свой сценарий в нечто более читаемое:

```
Сценарий: множественные данные
    Дано что-то первое
    И что-то второе
    И и что-то ещё
    Когда я открою свои глаза
    То я увижу что-то
    Но чего-то я не увижу
```

2.5.5 Таблицы

Регулярные выражения, с помощью которых программисты получают данные из текстового описания шагов позволяют получать небольшие куски данных из самой строчки. Но бывает и такое, что

необходимо передать больший объем данных в один шаг. И здесь нам на помощь придут таблицы:

```
Сценарий:
    Дано существуют следующие пользователи:
        | Логин | E-mail        | Пароль |
        | user1 | user1@mail.ru | pass1  |
        | joe   | joe@gmail.com | hey    |
        | heyho | hey@hoe.com   | joe    |
```

Глава 3

Wellbehaved

Обертка вокруг питоновского проекта behave (который, в свою очередь, является портом Cucumber из Ruby), добавляющая возможность написания плагинов и шаблонизацию с использованием Jinja2.

3.1 Установка

Установите пакет `wellbehaved` со стандартного PyPi-сервера "БАРС Груп":

```
pip install wellbehaved -i https://<PyPi_сервер_БАРС_Груп>/simple
```

3.2 Запуск

Запустить тестирование можно с помощью обычной команды:

```
./wellbehaved --cfg-file cfg.yaml --var-file vars.py -- --lang ru
```

3.3 Настройки

Для удобства передачи опций `behave`, используется следующая схема:

```
wellbehaved [--cfg-file <файл конфигурации>] [--var-file <файл с переменными>] -- [опции behave]
```

Опции командной строки:

–cfg-file <имя файла>

Подключение YAML-файла с настройками плагинов.

–var-file <имя файла>

Python-файл с переменными для шаблонизатора. Весь словарь переменных и функций, полученный в ходе исполнения этого файла будет передан в качестве контекста Jinja2, после чего будет использоваться для подстановки переменных в к2аждом feature-файле.

3.4 Плагины

3.4.1 redmine

Пропуск выполнения сценариев согласно информации из Redmine.

Этот плагин позволяет пропускать выполнения сценария (в результатах он будет отображаться как *skipped), если статус связанной со сценарием задачи соответствует одному из заданных.

Настройки:

- *host* - адрес сервера Redmine;
- *user* - имя пользователя;
- *pass* - пароль;
- *statuses* - разделенный запятыми список наименований статусов задач.

Сценарий связывается с задачей через тэг формы **@redmine{{taskID}}**:

```
@redmine12345
Scenario: hello, world!
```

3.4.2 coverage

Интеграция со стандартным coverage.

Этот плагин собирает и отображает информацию о покрытии кода шагов.

Настройки:

- *type* - формат сохранения собранной информации:
  - *report* - стандартный текстовый файл с отчетом;
  - *html* - папка со сгенерированным HTML-представлением отчета.
- *output* - имя файла/папки с результатами сбора информации.

3.5 Пример использованиия

Предположим, что пока задача *12345* находится в статусах *"Разработка"* и *"Анализ"* нам нет смысла прогонять сценарий тестирования личной страницы. Так как имя пользователя используется во многих тестовых файлах одновременно и со временем может изменится, было решено сделать его переменным.

Файл с переменными:

```
#coding: utf-8

username = u'Тестовый Пользователь'
```

Файл конфигурации:

```
plugins:
  redmine:
    host: http://task.bars-open.ru
    user: user1
    pass: pass1
```

```
    statuses: ['Разработка', 'Анализ']
  enabled_plugins: ['redmine']
```

Feature-файл:

```
Функция: работа с рабочим столом

  @redmine12345
  Сценарий: открытие личной страницы
    Дано пользователь "{{username}}"
    И пользователь вошел в систему
    Когда в меню выбрана "Личная страница"
    То открывается страница с иконками
```

Запуск:

```
./wellbehaved --cfg-file cfg.yaml --var-file vars.py -- --lang py
```

3.6 Контакты

С вопросами по доработкам, применению и с сообщениями об ошибках пишите на borisov@bars-open.ru

3.7 Благодарности

- Юле Касимовой (kasimova@bars-open.ru) - самоотверженное участие в тестировании продукта
- Сергею Чипиге (svchipiga@bars-open.ru) - нахождение багов
- Вадиму Малышеву (vvmalyshev@bars-open.ru) - продавливании идеи изучения концепций BDD

3.8 Ссылки

- Writing Features - Gherkin Language
- Behavior Driven Development (from behave documentation)
- List of behave formatters

Глава 4

Структура исходного кода

4.1 main - основной модуль

4.2 import_hooks - модификаторы поведения behave

class `wellbehaved.import_hooks.TemplateImportHooker`(*vars=None*)

Импорт-хук, который оборачивает стандартную функцию разбора фиич и трактует каждую из них как шаблон для Jinja2.

Выполнено это через подмену функции *parse_feature* модуля **behave.parser**.

`find_module`(*name*, *path=None*)

Фильтр модулей, которые обрабатываются этим хуком.

Параметры

- `name` – Имя импортируемого модуля.
- `path` – Путь к импортируемому модулю.

`load_module`(*name*)

Загрузчик модуля, который подменяет функцию разбора feature-файла нашей, которая предварительно преобразует её через шаблонизатор.

Параметры `name` – Имя модуля, во избежание повторной обработки.

4.3 utils - вспомогательные функции

Вспомогательный класс подсистемы плагинов.

class `wellbehaved.utils.StackedHookDictWrapper`(**args*, ***kwargs*)

Унаследованный от *dict* класс, "прозрачно" перехватываюиbщий установку обработчиков шагов тестирования в environment.py.

Каждый перехваченный обработчик добавляется в стэк, связанный с этим конкретным этапом тестирования и при каждом .

Глава 5

Индекс

- genindex
- modindex
- search

W

F

L

S

T

W