

Original Behavior Driven Development Framework

自作BDDフレームワーク

Introduction of PySpec and its implementation

# PySpecの紹介と実装

Shibukawa Yoshiki

渋川 よしき



実は…

- 日本で最初のScrumの翻訳本に関わりました



## At First... 最初に...

- <http://pyunitx.shibu.jp>

Please tell me how to remove spam ticket on Trac

- だれかTracのスパムチケットの削除の仕方を教えて . . .



## History of PySpec(1)

# PySpecの歴史(1)

I envied NUnit2, JUnit4 and RSpec (Behavior Driven Development)

- NUnit2とJUnit4やRSpec(BDD)がうらやましかった

Deriving TestCase is boring

- TestCaseを継承するのはダサイ

Definitive Attribute is cool!

- Attributeで宣言って今時っぽくてイイ

```
[TestFixture]
public class SetupTest001 {
    [Test]
    public void FileCheckSample() {
        int a = 100;
        Assertion.AssertEquals("変数チェック", 100, a);
    }
}
```

## History of PySpec(2)

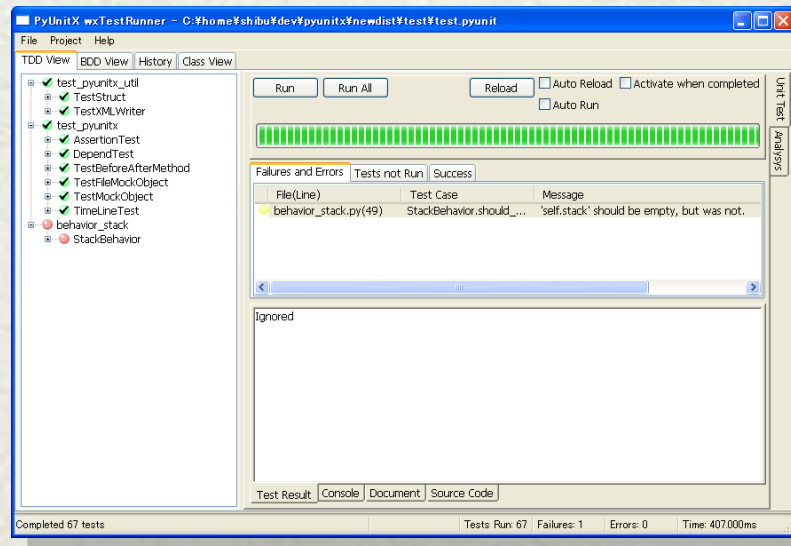
# PySpecの歴史(2)

I announced it as PyUnitX in LT at ObjectClub Summer Event

- 2006/6/30、PyUnitXとして、オブラブ夏イベントのライトニングトークスで発表

I got second prize!!

– あまのりょーさんに次いで二位獲得



## History of PySpec(3) PySpecの歴史(3)

I forgot it about 9 months

- その後9ヶ月ぐらい放置

Now I am restructuring and refactoring after I reminded

- また最近いろいろ手を加え始めた

## PySpec Sample Code

# PySpecのサンプルコード

```
class Stack Behavior(object):
    """スタックの動作仕様."""
    @context
    def New_stack(self):
        """新しいスタック."""
        self.stack = Stack()

    @spec
    def should_empty(self):
        """空でなければならない."""
        About(self.stack).should_be_empty()
```

## PySpec Sample Code

# PySpecのサンプルコード

```
class Stack Behavior(object):  
    """スタックの動作仕様."""
```

```
    @context
```

```
    def New_stack(self):  
        """新しいスタック"""  
        self.stack = Stack()
```

```
    @spec
```

```
    def should_empty(self):  
        """空でなければならない."""
```

```
        About(self.stack).should_be_empty()
```

Plain Old Python Object

普通のPython  
オブジェクト

Decorators to define test

デコレータで  
テストを宣言

## PySpec Sample Code

# PySpecのサンプルコード

```
class Stack Behavior(object):  
    """スタックの動作仕様."""
```

```
@context
```

```
def New_stack(self):  
    """新しいスタック.  
    self.stack = Stack()
```

Method name and docstring  
are used for spec name

メソッド名とdocstringが  
仕様名になる

```
@spec
```

```
def should_empty(self):  
    """空でなければならない."  
    About(self.stack).should_be_empty()
```

## PySpec Sample Code

# PySpecのサンプルコード

Method name and docstring become specification name

- メソッド名とdocstringが仕様名になる

Test code becomes a specification document

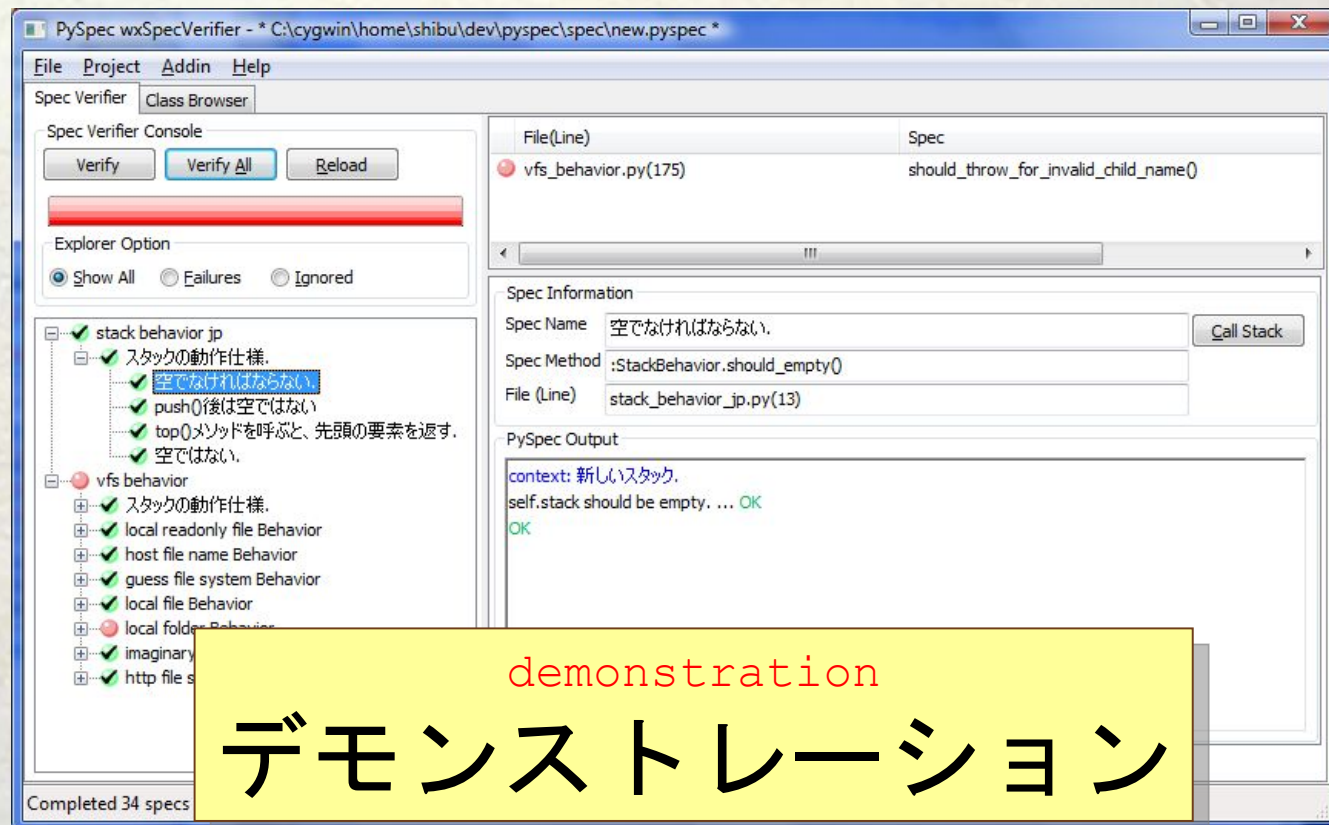
- テストコードが仕様書になる

	<b>Spec Name</b> 仕様名
<b>English Name</b> 英語名	New Stack: should be empty.
<b>Japanese Name</b> 日本語名	新しいスタック: 空でなければならない。

# GUI Spec Test Runner(1)

GUI Test Runner uses wxPython

- wxPythonで作成



# GUI Spec Test Runner(2)

- **Result Report**  
結果表示

Spec Information

Spec Name: 存在が確認できる. Call Stack

Spec Method: :http\_file\_system\_Behavior.should\_be\_able\_to\_check\_existance()

File (Line): vfs\_behavior.py(337)

PySpec Output

```
self.http_file.exists() should be True. ... OK
OK
```

Verify Result Console Out SourceCode

- **Console Output**  
コンソール出力

```
http://localhost:8000/sample.txt
MDX -- [29/Jun/2007 23:22:55] "GET /sample.txt HTTP/1.1" 200 -
self.http_file.exists() should be True.run remove_sample_file
run stop_http_server
MDX -- [29/Jun/2007 23:22:56] "QUIT / HTTP/1.1" 200 -
```

Verify Result Console Out SourceCode

- **Source Code Viewer**  
ソースコード表示

```
1 # -*- coding: utf_8 -*-
337 | @spec
338 | def should_be_able_to_check_existance(self):
339 |     """存在が確認できる."""
340 |     About(self.http_file.exists()).should_be_true()
341 |
```

Verify Result Console Out SourceCode

## PySpec Other feature

# PySpecのその他の機能

---

- Mock Objects
  - Object, MockFile, MockSocket
- Stack Trace Viewer
  - Call Graph Viewer (stable?)
- Class Browser (unstable)
- Module Dependency Map Viewer
- Auto Reloader (not implemented yet)

**PySpec Hacking Guide**

**PySpecの実装**



Most important fact to say...

もっとも重要なこと . . .

I love 'inspect' module

## Decorator デコレータ

```
@spec
def sample_spec():
    a = 1
    About(a).should_equal(1)
```

How is the test ran?

どのようにテストが実行されるのか？

## Decorator デコレータ

`pyspec.__init__.py`, `pyspec.framework.py` etc...

### (1) @spec

Add `'__pyspec_attribute'` attribute to the method

-> メソッドに `__pyspec_attribute` という属性を追加

### (2) TestRunner

Search methods that have the attribute from all  
Module name space, and run!

-> 全名前空間のメソッドを探索し、`__pyspec_attribute`属性  
のついたメソッドをリストアップして実行

Search the test methods from all name space

# 名前空間を探索して探しています

Why doesn't pyspec choose more elegant way?  
なぜこんな面倒なことしてるの？

---

Because I want to be able to write spec class that is not derived special class

- テストクラスをPOPOで実装できるようにするため

Then I can't get support from special parent class

- そうなると、親クラスによるサポートは受けられなくなるので

## Getting Variable Name 変数名の取得

```
@spec  
def fail_spec():  
    a = 1  
    About(a).should_equal(2)
```

⇒ **AssertionError:**  
    **a** should equal **2**, but was **1**.

Where the Variable Name 'a' comes from?

# 変数名はどこから取得しているのか？

## Getting Variable Name 変数名の取得

```
pyspec.__init__.py:47
```

```
(1) inspect.stack()
```

-> スタックフレームの探索 (search stack frame)

-> ソースコードの読込 (read source file)

-> 関数を呼んだ行のソースコード (get the line)

```
(2) re.search("About\\((.*?)\\).should", ...)
```

I get variable name through regular expression

-> 正規表現でソースコード解析

I get variable name from stack frame and source code  
スタックとソースコードから抜き出しています

Why doesn't pyspec choose more elegant way?  
なぜこんな面倒なことしてるの？

---

Because variable name is important to create spec from test

- 変数名を表示することは、テストを仕様書にするには必要不可欠

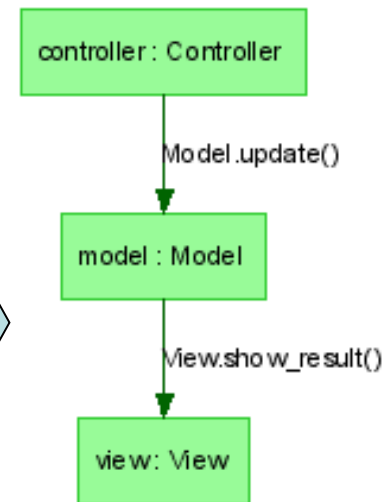
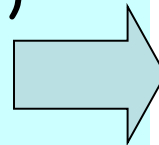
Read Refactoring[FOWLER 99]

- リファクタリング本を読むべし

## Getting instance name インスタンス名の取得

**@spec**

```
def MVC_test():  
  controller = Controller()  
  controller.update()
```



Where the instance name comes from?

インスタンス名は  
どこから取得しているのか？

Getting instance name

## インスタンス名の取得

```
pyspec.pyspec_util.py:259
```

```
(1) class IsolatedStackFrame
```

```
-> 呼び出し履歴取得 (use sys.settrace)
```

```
-> selfのidを取得 (get ID of self of function)
```

```
-> 関数を呼んだローカル名前空間からselfを探索  
(get 'self' from locals())
```

I uses API for debugger to get

デバッガ用のAPI (sys.settrace) 使ってます

Getting instance name

## インスタンス名の取得

I will enhance it more!

- まだまだ拡張していきます

Parameter of object (self )

– オブジェクトのパラメータ

```
self.viewer.show_result()
```

Return value of function

– 直前の関数の返り値

```
get_viewer().show_result()
```

Why doesn't pyspec choose more elegant way?  
なぜこんな面倒なことしてるの？

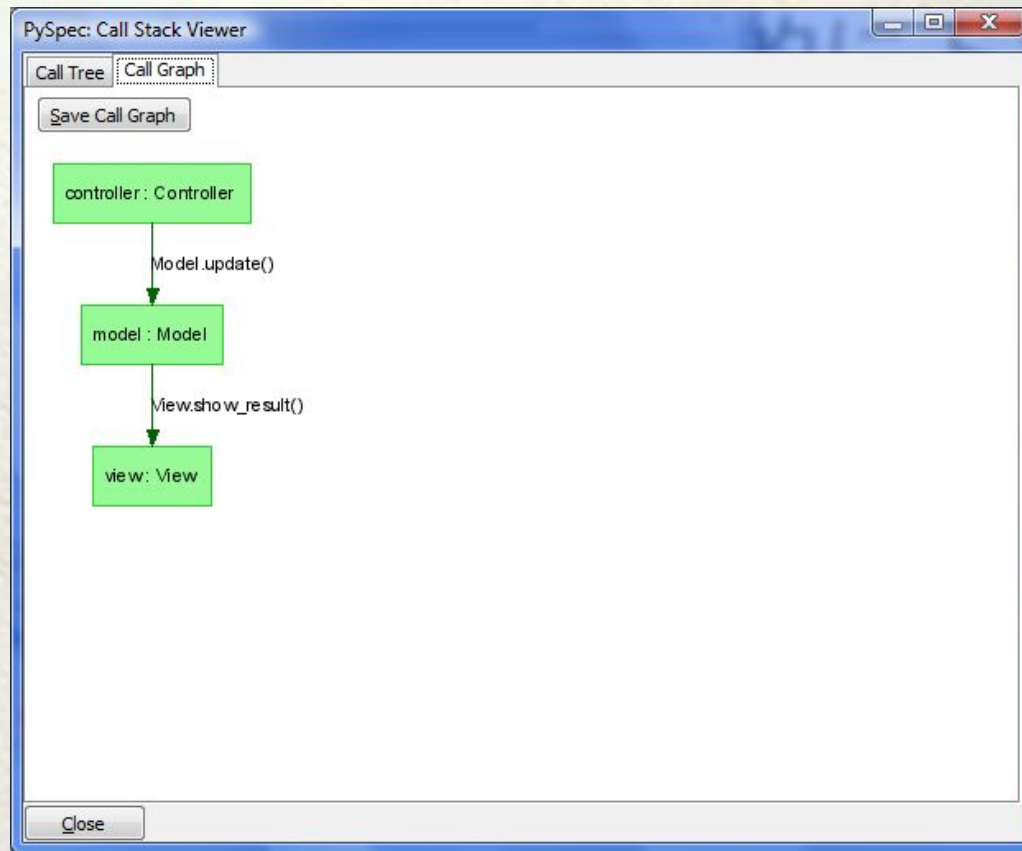
---

Automatic communication diagram generation

- コミュニケーション図の自動生成

Communication diagram is useful for analyzing code

- コミュニケーション図はコード解析に役立つ



# Problems for Porting PySpec to IronPython



---

- GUI?

- wx.NET?

- .NET framework?

- Compatibility

- `sys.settrace`, `sys.setprofile`

- inspect module

- Hooking `sys.stdout`, `sys.stderr`

**PySpec Reference Guid**

**PySpecリファレンスガイド**



## How to write test テスト書き方

Use decorator To define test

- デコレータでテストの指定をする

You can write test in following style: “the code should do something”

- テストは「～は、XXXであるべき」と言う書き方をする。

```
class Stack_Behavior:
    @spec
    def new_stack_should_be_empty(self):
        stack = Stack()
        About(stack.size()).should_equal(0)
```

## Decorators(1)

# デコレータの種類(1)

- @spec
- @spec(expected= ExceptionClass )

Define test method /function

– テストメソッド/関数の宣言

Add expected parameter, this method expects that exception will be raised

– expectedパラメータを与えると、「この例が投げられるべき」という意味になる

## Decorators(2)

# デコレータの種類(2)

---

- @context, @class\_context

Define setup method /function

- テスト用のオブジェクトなどの準備をする  
メソッドの宣言

- @spec\_finalize, @class\_finalize

Define tear down method /function

- 片づけをするメソッドの宣言

## Decorators(3)

# デコレータの種類(3)

- @ignored

If test method have this decorator, it will be skipped

– テストメソッドの実行をスキップすることができます

```
class Python3000_Behavior:
    @ignored
    @spec
    def not implemented function(self):
        """ まだ実装されていない関数 """
        About(format("{0}", "hello")).should_equal("hello")
```

## Verify Methods(1)

# 検証メソッド(1)

---

- `About(actual).should_equal(expected)`
- `About(actual).should_not_equal(not_expected)`
- `About(actual).should_equal_nearly(expected, delta)`
- `About(actual).should_not_equal_nearly(not_expected, delta)`
- `About(actual).should_be_true()`
- `About(actual).should_be_false()`
- `About(actual).should_be_none()`
- `About(actual).should_not_be_none()`

## Verify Methods(2)

# 検証メソッド(2)

---

- `About(actual).should_be_same(expected)`
- `About(actual).should_not_be_same(not_expected)`
- `About(sequence).should_include(expected)`
- `About(sequence).should_not_include(not_expected)`
- `About(sequence).should_be_empty()`
- `About(sequence).should_not_be_empty()`
- `Verify.fail(msg=None)`
- `Verify.ignore(msg=None)`