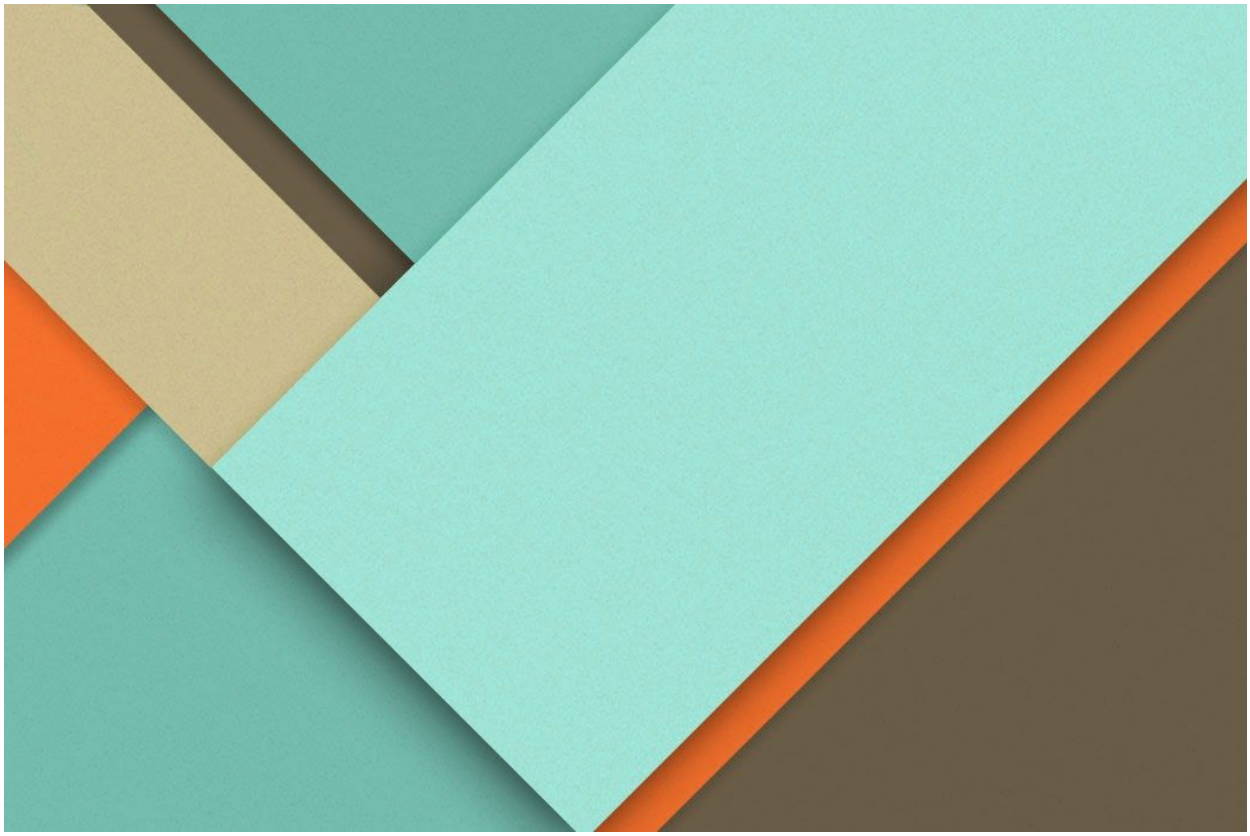


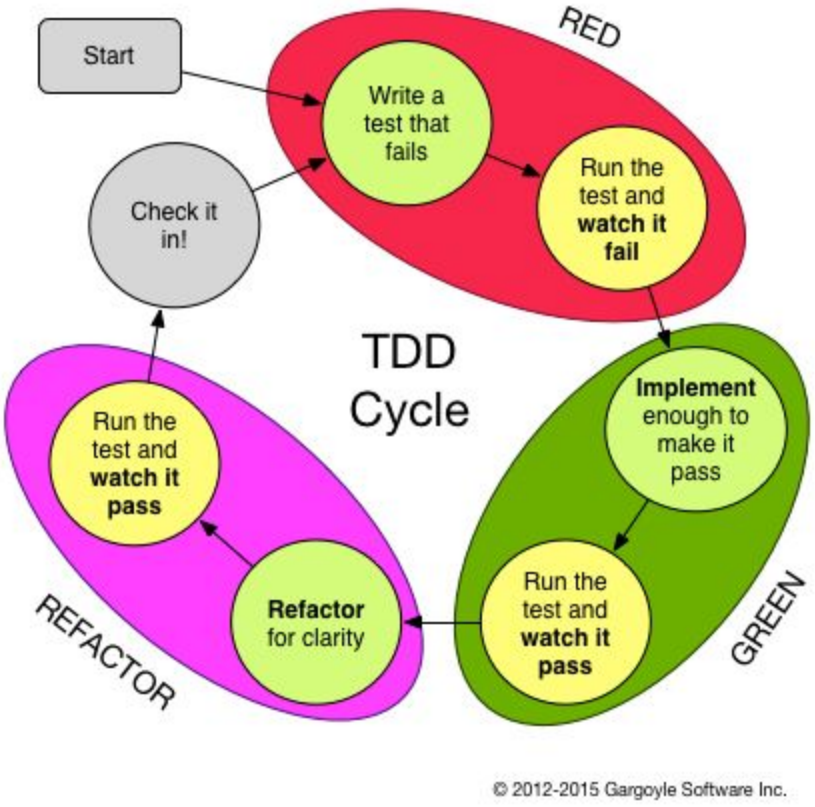
Software Engineering Lab #1

TDD + SOLID principles

محمدرضا غمخوار ۹۵۱۰۶۴۹۴
محمدرضا احمدخانی‌ها ۹۵۱۰۵۳۱۳

مقدمه

در این آزمایش خواسته‌های مسئله را با استفاده از شیوه test driven development و با رعایت اصول SOLID انجام می‌دهیم.



چرخه red-green-refactor

برای اجرای این مراحل از زبان برنامه‌نوسی python و ابزار pytest استفاده می‌کنیم.

روند کلی این است که برای پیاده سازی هر feature جدید در ابتدا تست‌هایی برای آن می‌نویسیم؛ این تست‌ها باید همه شرایط مختلف و مرزی را پوشش دهند و همچنین از طرز کار درست کد با توجه به نیازمندی پروژه مطمئن شوند.

سپس تست‌ها را اجرا می‌کنیم و fail شدنشان را به نظاره می‌نشینیم. سپس با ایجاد interface ها و signature کد خطاهای سینتکسی را از بین می‌بریم.

در گام بعدی حداقل کدی که برای اجرای موفق تست‌ها نیاز است را پیاده سازی می‌کنیم تا پس از اجرای تست‌ها همه آن‌ها pass شوند.

در گام آخر نیز کد را refactor می‌کنیم تا درنهایت کدی خوانا و تمیز در اختیار داشته باشیم. همچنین برای آخرین بار تست‌ها را اجرا می‌کنیم تا مطمئن شویم همه چی طبق برنامه پیش رفته است.

انجام این مراحل به ما اطمینان بیشتری در طی توسعه کد می‌دهد زیرا می‌دانیم که در هر مرحله کد تست دارد و از درستی برنامه مطمئنیم؛ از طرفی با نوشتن تست‌ها تا حد خوبی از پوشش دادن اکثر حالات مطمئنیم در نتیجه احتمال برخورد به bug در ادامه مسیر حداقل می‌شود.

بخش اول - مستطیل

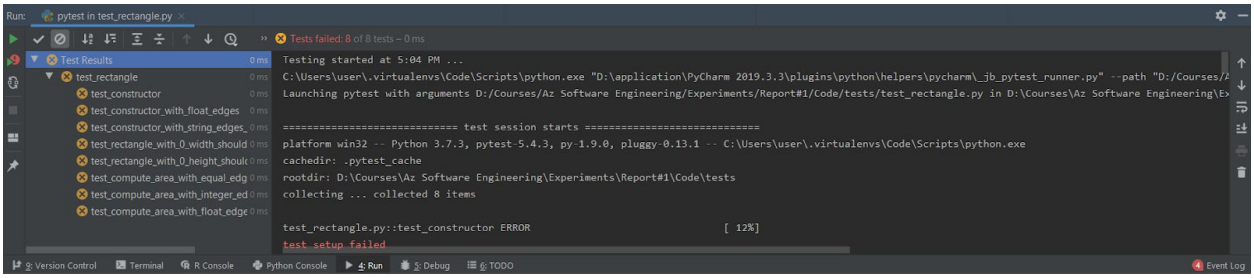
مرحله اول - Red

در بخش اول تست‌های مورد نیاز را می‌نویسیم :

```
def test_constructor(rectangle):
    assert isinstance(rectangle, Rectangle)


def test_constructor_with_float_edges():
    try:
        rectangle = Rectangle(2.5, 4.5)
    except Exception as e:
        pytest.fail("should not throw exception")
    else:
        assert True


def test_constructor_with_string_edges_should_throw_exception():
    try:
        rectangle = Rectangle(12, "Hello")
    except InvalidEdgeType as e:
        assert True
    else:
        pytest.fail("expected error but got none")


def test_rectangle_with_0_width_should_not_be_possible():
    try:
        rectangle = Rectangle(height=4, width=0)
    except ZeroDivisionError as e:
        assert isinstance(e, Exception)
    except ZeroEdgeError as e:
        assert isinstance(e, Exception)
    else:
        pytest.fail("Expected error but found none")


def test_rectangle_with_0_height_should_not_be_possible():
    try:
```

```
        rectangle = Rectangle(height=0, width=4)
    except ZeroDivisionError as e:
        assert isinstance(e, Exception)
    except ZeroEdgeError as e:
        assert isinstance(e, Exception)
    else:
        pytest.fail("Expected error but found none")


def test_compute_area_with_equal_edges():
    rectangle = Rectangle(height=2, width=2)
    assert rectangle.compute_area() == 4


def test_compute_area_with_integer_edges(rectangle):
    assert rectangle.compute_area() == 15


def test_compute_area_with_float_edges():
    rectangle = Rectangle(2.5, 4.5)
    assert rectangle.compute_area() == 2.5*4.5
```

حال آن‌ها را اجرا کرده و شاهد اشتباهات (به واسطه‌ی پیاده سازی نشدن کدها نه اشتباه بودن رفتار کد) آن هستیم:



fail شدن همه تست‌ها بواسطه پیاده سازی نشدن کد

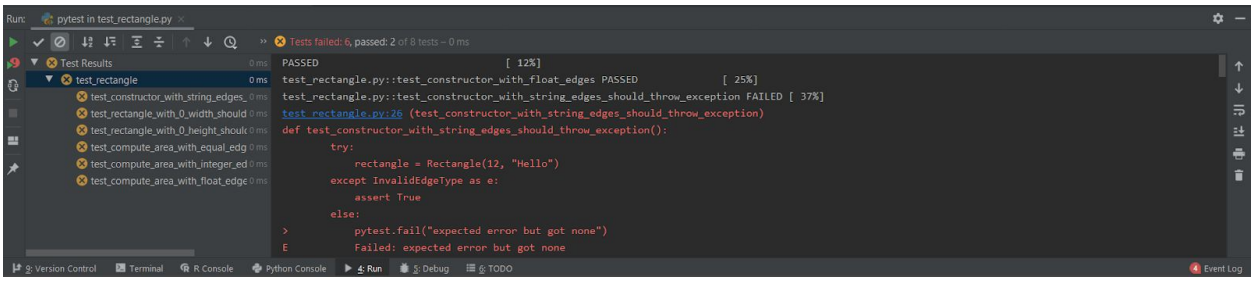
حال به سراغ پیاده سازی رابطهای کد می‌رویم تا تست‌ها صرفا به واسطه اشتباه بود fail شوند و نه پیاده سازی نشدن.

پیاده سازی:

```
class Rectangle:
    def __init__(self, height, width):
        self.height = height
        self.width = width

    def compute_area(self):
        return -1


class InvalidEdgeType(Exception):
```

```
    pass


class ZeroEdgeError(Exception):
    pass
```

حال دوباره تست‌ها را اجرا می کنیم تا شاهد fail شدنشان باشیم:



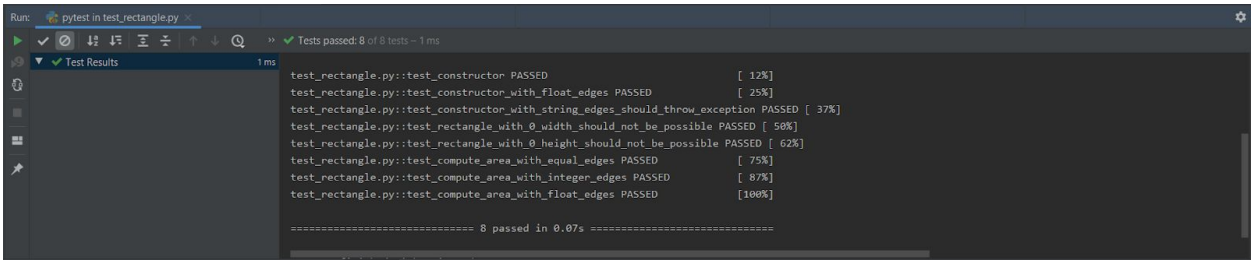
مشاهده شد که تست‌ها fail شدند بدون هرگونه اشتباه syntaxای.

مرحله دو - Green

پیاده سازی را به صورت حداقلی کامل می‌کنیم تا تست‌ها pass شوند:

```
class Rectangle:
    def __init__(self, height, width):
        if not (type(height) == int or type(height) == float) or not (type(width) ==
int or type(width) == float):
            raise InvalidEdgeType
        if height <= 0 or width <= 0:
            raise ZeroEdgeError

        self.height = height
        self.width = width

    def compute_area(self):
        return self.height * self.width


class InvalidEdgeType(Exception):
    pass


class ZeroEdgeError(Exception):
    pass
```

حال تست‌ها را اجرا می‌کنیم:



همانطور که مشاهده می‌شود همه تست‌ها با موفقیت pass شدند.

مرحله سوم - Refactor

اکنون کدهای توسعه داده شده را refactor می‌کنیم تا کدی زیباتر و خواناتر داشته باشیم.

بخش‌های تغییر کرده در تست‌ها:

```
@pytest.fixture
def rectangle():
   return Rectangle(height=5, width=3)
```

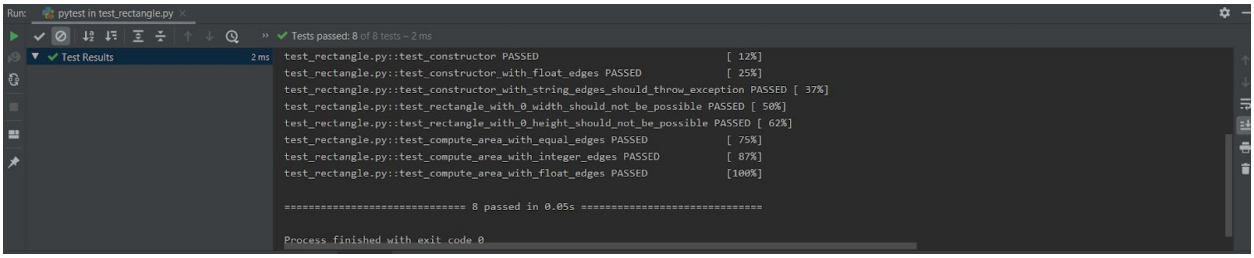
از آنجایی که در بخش زیادی از تست‌ها در ابتدا یک شی از کلاس مستطیل می‌ساختیم از این طریق این شی به صورت خودکار ساخته می‌شود.

بخش‌های تغییر کرده در کد اصلی:

```
class Rectangle:
   def __init__(self, height, width):
       self.check_edge_value_validity(height, width)
       self.height = height
       self.width = width

   def check_edge_value_validity(self, height, width):
       if not self.is_integer_or_float(height) or not
self.is_integer_or_float(width):
           raise InvalidEdgeType
       if height <= 0 or width <= 0:
           raise ZeroEdgeError

   def is_integer_or_float(self, number):
       return type(number) == int or type(number) == float
```

با منتقل کردن شروط اولیه ایجاد مستطیل به توابعی جدا کد بسیار سادمتر و خواناتر شده است.

مشاهده می‌شود که همچنان تست‌ها کاملا pass می‌شوند:



بخش دوم - امکان تغییر طول و عرض مستطیل

مرحله اول - Red

در این بخش تست‌های زیر توسعه داده شدند:

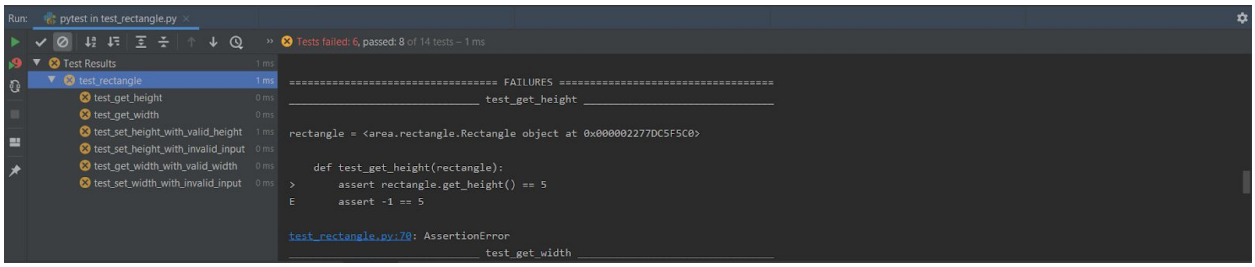
```
def test_get_height(rectangle):
    assert rectangle.get_height() == 5


def test_get_width(rectangle):
    assert rectangle.get_width() == 3


def test_set_height_with_valid_height(rectangle):
    assert rectangle.get_height() == 5
    rectangle.set_height(12)
    assert rectangle.get_height() == 12
    rectangle.set_height(1.1)
    assert rectangle.get_height() == 1.1


def test_set_height_with_invalid_input(rectangle):
    assert rectangle.get_height() == 5
    invalid_heights = ["Moo!", 0, -2, "cool"]
    for invalid_height in invalid_heights:
        try:
            rectangle.set_height(invalid_height)
        except (InvalidEdgeType, ZeroEdgeError) as e:
            assert isinstance(e, Exception)
        else:
            pytest.fail("Error expected but none found")


def test_get_width_with_valid_width(rectangle):
```

```
    assert rectangle.get_width() == 3
    rectangle.set_width(10)
    assert rectangle.get_width() == 10
    rectangle.set_width(2.2)
    assert rectangle.get_width() == 2.2


def test_set_width_with_invalid_input(rectangle):
    assert rectangle.get_width() == 3
    invalid_widths = ["Moo!", 0, -2, "cool"]
    for invalid_width in invalid_widths:
        try:
            rectangle.set_width(invalid_width)
        except (InvalidEdgeType, ZeroEdgeError) as e:
            assert isinstance(e, Exception)
        else:
            pytest.fail("Error expected but none found")
```

با ایجاد تغییرات زیر در کد این تست‌ها به مرحله قابل اجرا شدن رسیدند:

```
def get_height(self):
    return -1

def set_height(self, height):
    pass

def get_width(self):
    return -1

def set_width(self, width):
    pass
```

حال مشاهده می‌شود که تست‌ها fail می‌شوند:

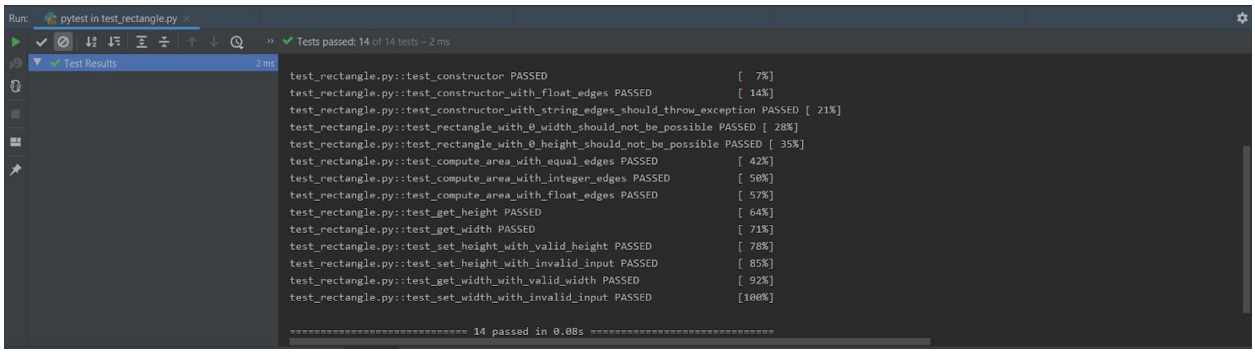


مرحله دو - Green

حال پیاده سازی حداقلی را انجام می‌دهیم تا تست‌ها pass شوند:

```
def get_height(self):
    return self.height

def set_height(self, height):
    if not self.is_integer_or_float(height):
        raise InvalidEdgeType
    if height <= 0:
        raise ZeroEdgeError

    self.height = height

def get_width(self):
    return self.width

def set_width(self, width):
    if not self.is_integer_or_float(width):
        raise InvalidEdgeType
    if width <= 0:
        raise ZeroEdgeError

    self.width = width
```

نتیجه مطابق انتظار همان pass شدن همه تست‌هاست:

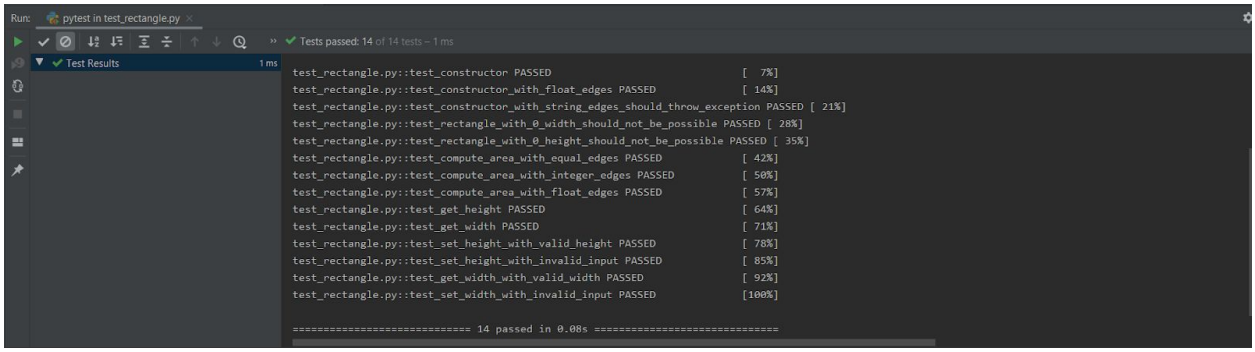


مرحله سوم - Refactor

با کمی دقت متوجه می‌شویم با اندک تغییری در تابع check_edge_value_validity می‌توانیم از آن در توابع set طول و عرض استفاده کنیم:

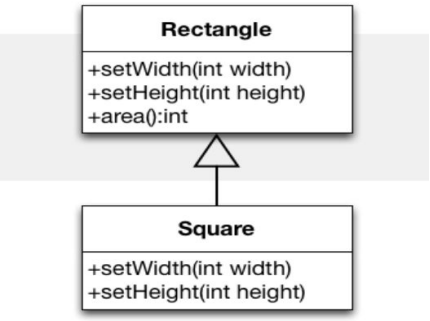
```
def check_edge_value_validity(self, edge):
    if not self.is_integer_or_float(edge):
        raise InvalidEdgeType
    if edge <= 0:
        raise ZeroEdgeError

def set_height(self, height):
    self.check_edge_value_validity(height)
    self.height = height

def set_width(self, width):
    self.check_edge_value_validity(width)
    self.width = width
```

همچنان مطمئن می‌شویم که تست‌ها قابل اجرا باشند:



بخش سوم - گسترش برنامه برای در نظر گرفتن مربع

در نگاه نخست می‌دانیم که از نظر ریاضیاتی هر مربع یک نوع مستطیل نیز به شمار می‌رود، اما آیا در نظر گرفتن مربع به عنوان زیرکلاسی از مستطیل در کد، کار درستی است؟ (مانند این شکل)



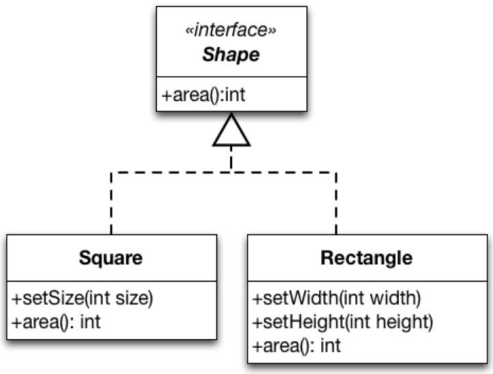
جواب **خیر** است.

این عمل Liskov substitution Principle را نقض می‌کند زیرا مربع از نظر رفتاری، با مستطیل تفاوت دارد.

دلیل این اتفاق استفاده از متدهایی است که مستطیل را قابل تغییر میکنند(mutable) زیرا اگر قابل تغییر نباشد، در صورتی که بخواهیم مستطیلی با طول و یا عرض جدید داشته باشیم، شی جدید ایجاد می‌کنیم و همچنان مربع بدون مشکل می‌تواند زیر کلاس مستطیل باقی بماند.

اما مشکل جایی پدید می‌آید که امکان تغییر طول و یا عرض مستطیل را اضافه می‌کنیم. در این حالت به عنوان مثال با تغییر طول، مربع دیگر با متد ما قابل انطباق نیست و این مسئله باعث پدید آمدن باگ های زیادی ممکن است بشود. در حقیقت شرایط اولیه مربع (هر چهار ضلع با هم برابرند) از شرایط اولیه مستطیل (اضلاع روبه‌رو با هم برابرند) سخت تر است.

برای حل این موضوع می‌توانیم از کلاس دیگری مانند shape استفاده کنیم. در این حالت با در نظر گرفتن مستطیل و مربع به عنوان sibling از مشکلات ذکر شده در حالت قبل جلوگیری می‌کنیم.



مرحله اول - Red

برای این بخش فایل تستی مشابه با تست‌های مستطیل می‌سازیم ولی این بار به جای استفاده از rectangle به عنوان ورودی از square استفاده می‌کنیم. همچنین تست‌هایی اضافه می‌کنیم برای پوشش ویژگی‌های جدید مربع که از این قرارند:

```
def test_constructor_with_one_input():
    try:
        square = Square(4)
        assert square.height == 4
        assert square.width == 4
    except:
        pytest.fail("should not throw exception when there is 1 input")


def test_constructor_with_two_different_inputs():
    try:
        square = Square(4, 6)
    except UnequalEdgesError as e:
        assert isinstance(e, Exception)
    else:
        pytest.fail("Error expected but none found")
```

```
def test_set_height_with_valid_height(square):
    assert square.get_height() == 5
```

```
    square.set_height(12)
    assert square.get_height() == 12
    assert square.get_width() == 12
    square.set_height(1.1)
    assert square.get_height() == 1.1
    assert square.get_width() == 1.1


def test_set_width_with_valid_width(square):
    assert square.get_width() == 5
    square.set_width(10)
    assert square.get_width() == 10
    assert square.get_height() == 10
    square.set_width(2.2)
    assert square.get_width() == 2.2
    assert square.get_height() == 2.2
```

حال مراحل مربوط به پیاده سازی را طی می‌کنیم تا اشتباهات سینتکسی از میان بروند.

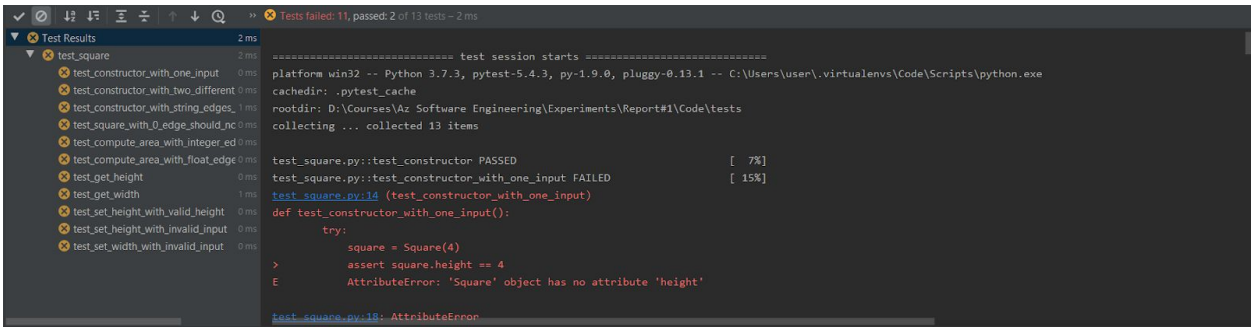
در قدم اول interface مربوط به shape را پیاده سازی می‌کنیم:

```
from abc import ABC, abstractmethod


class Shape(ABC):

    @abstractmethod
    def compute_area(self):
        pass

    @abstractmethod
    def get_height(self):
        pass

    @abstractmethod
    def set_height(self, height):
        pass

    @abstractmethod
    def get_width(self):
        pass

    @abstractmethod
    def set_width(self, width):
        pass
```

سپس کلاس square که فرزند shape هست را می‌سازیم:

```python
class Square(Shape):
    def __init__(self, height=None, width=None):
        pass

    def compute_area(self):
        return -1

    def get_height(self):
        return -1

    def set_height(self, height):
        pass

    def get_width(self):
        return -1

    def set_width(self, width):
        pass


class UnequalEdgesError(Exception):
    pass
```

حال fail شدن تست‌ها را مشاهده می‌کنیم:

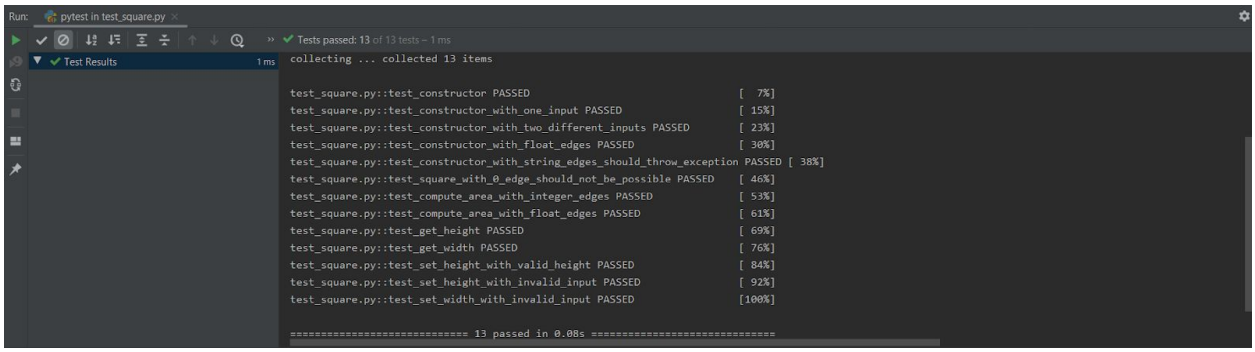


مرحله دو - Green

حال پیاده سازی حداقلی را انجام می‌دهیم تا تست‌ها pass شوند:

```
class Square(Shape):
   def __init__(self, height=None, width=None):
       if height is None:
           height = width
       elif width is None:
           width = height
       if (height is None and width is None) or height != width:
           raise UnequalEdgesError
       self.check_edge_value_validity(height)
       self.check_edge_value_validity(width)
       self.height = height
       self.width = width

   def compute_area(self):
       return self.height * self.width

   def get_height(self):
       return self.height

   def set_height(self, height):
       self.check_edge_value_validity(height)
       self.height = height
       self.width = height

   def get_width(self):
       return self.width

   def set_width(self, width):
       self.check_edge_value_validity(width)
       self.width = width
       self.height = self.height

   def check_edge_value_validity(self, edge):
       if not self.is_integer_or_float(edge):
           raise InvalidEdgeType
       if edge <= 0:
           raise ZeroEdgeError

   def is_integer_or_float(self, number):
       return type(number) == int or type(number) == float
```

مشاهده می‌کنیم همه تست‌ها pass می‌شوند:



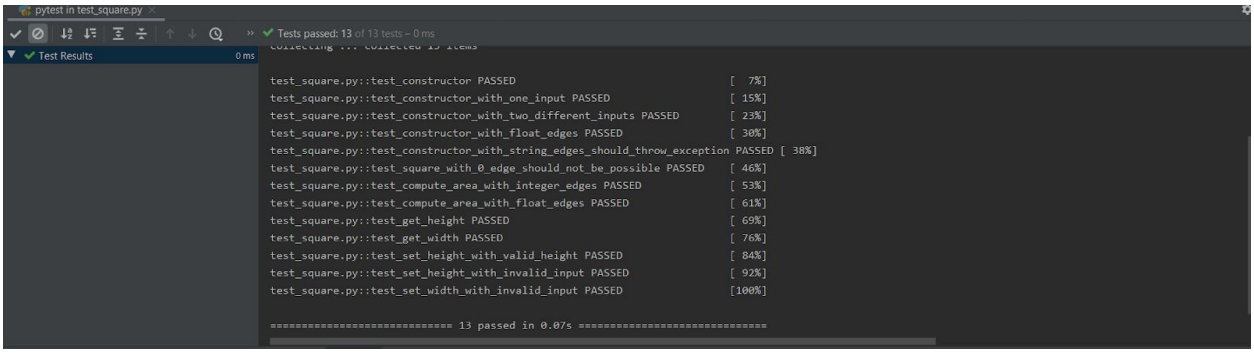
مرحله سوم - Refactor

اکنون کدهای توسعه داده شده را refactor می‌کنیم تا کدی زیباتر و خواناتر داشته باشیم.

بخش‌های تغییر کرده در کد:

```
def check_is_square(self, height, width):
    if (height is None and width is None) or height != width:
        raise UnequalEdgesError

def fill_width_and_height_if_not_given(self, height, width):
    if height is None:
        height = width
    elif width is None:
        width = height
    return height, width

def __init__(self, height=None, width=None):
    height, width = self.fill_width_and_height_if_not_given(height, width)
    self.check_is_square(height, width)
    ...
```

از pass شدن تست‌ها مطمئن می‌شویم:



نکاتی درباره‌ی تکنیک **Refactoring** :

تکنیکی است که کد را خواناتر و قابل فهم تر می‌کند و به صورت کلی پرفورمنس کد را بالا می‌برد. استفاده از این تکنیک می‌تواند در مواقعی که نیاز به تغییر در کد به وجود می آید بسیار کارآمد باشد. در حقیقت ساختار کد را بدون تغییر در فانکشنالیتی آن تغییر می‌دهد. این تکنیک همچنین "code smells" را هم از بین می‌برد و از به وجود آمدن تکرار در کد جلوگیری می‌کند.

از جمله کارهایی که در این تکنیک استفاده می‌شوند: Rename method, Encapsulate fields, Extract class,

(...,Pushdown method ,Introduce assertions