Test-first编程的思想

让程序尽早失败，从而尽早地发现bug，以节省调试时间

Test-first编程的过程（以单元测试为例）：

1. 先写spec
2. 再写符合spec的测试用例
3. 写代码、执行测试、有问题再改、再执行测试用例，直到通过它

Test-first 编程的好处：

1. 可以让你更早的发现bug
2. 可以帮助你更加深入的理解spec
3. 让你写代码更加自信

Test Case

> 测试用例 = 输入 + 执行条件 + 期望结果

设计Test Case的一般方法：将输入按等价类划分，每个等价类代表着对输入约束加以满足/违反的有效 /无效数据的集合，在使用一种策略来根据等价类来设计测试用例

策略：

1. 笛卡尔积，多个划分维度上的多个取值，要组合起来，每个组合都要有一个用例
2. 覆盖每个取值，每个维度的每个取值至少被1个测试用例覆 盖一次即可

例如：

输入的学号no需满足的条件：
- 长度为10位：10、>10、<10
- 以119开头：以此开头、以其他开头
- 之后两位数应为03/36/37：03、36、37、其他

覆盖每个取值：最少1次即可

等价类	测试用例
长度为10	1190311111
长度>10	11111111111
长度<10	119361111
以119开头	1190311111
不以119开头	11111111111
之后两位是03	1190311111
之后两位是36	119361111
之后两位是37	1193711111
之后两位是其他	11111111111

笛卡尔积：全覆盖

长度=10	以119开头	之后两位是03	1190300000
		之后两位是36	1193600000
		之后两位是37	1193700000
		之后两位是其他	1194000000
	不以119开头	之后两位是03	1200300000
		之后两位是36	1203600000
		之后两位是37	1203700000
		之后两位是其他	1204000000
长度>10	以119开头	之后两位是03	11903000001
		之后两位是36	11936000001
		之后两位是37	11937000001
		之后两位是其他	11940000001
	不以119开头	之后两位是03	12003000001
		之后两位是36	12036000001
		之后两位是37	12037000001
		之后两位是其他	12040000001
长度<10	以119开头	之后两位是03	119030000
		之后两位是36	119360000
		之后两位是37	119370000
		之后两位是其他	119400000
	不以119开头	之后两位是03	120030000
		之后两位是36	120360000
		之后两位是37	120370000
		之后两位是其他	120400000

还有一点值得注意，一般情况下我们都会将程序的边界情况加入等价类中。因为很多程序就是在边界情况最容易出错。

使用JUnit来编写单元测试

教程：https://www.vogella.com/tutorials/JUnit/article.html#junit_testorganization

比较常用的：

```
assertArrayEquals("failure - byte arrays not same", expected, actual);
assertEquals("failure - strings are not equal", "text", "text");
assertFalse("failure - should be false", false);
assertNotNull("should not be null", new Object());
assertNotSame("should not be same Object", new Object(), new Object());
assertNull("should be null", null);
assertSame("should be same", aNumber, aNumber);
assertTrue("failure - should be true", true);

assertThat("albumen", both(containsString("a")).and(containsString("b")));
assertThat(Arrays.asList("one", "two", "three"), hasItems("one", "three"));
assertThat("good", allOf(equalTo("good"), startsWith("good")));
```

黑盒测试和白盒测试

黑盒测试：用于检查代码的功能，不关心内部实现细节，主要用于检查程序是否符合spec。测试用例完全由spec导出，上面讲的都是黑盒测试

白盒测试：要更据程序·具体实现细节来写测试用例，例如一个程序可能跟据输入规模选择了不同的算法来实现，这是就要更具不同的规模来设计测试用例

回归测试

1. 一旦程序被修改，重新执行之前的所有测试
2. 一旦发现bug，要马上写一个可重现该bug的测试用例，并将其加入测试库

Test Strategy

相当于测试的说明书，用于记录你是如何测试的。写给未来的你和你的同事。

例如：

An example

```
/**
 * Reverses the end of a string.
 *
 * For example:
 *   reverseEnd("Hello, world", 5)
 *   returns "Hellodlrow ,"
 *
 * With start == 0, reverses the entire text.
 * With start == text.length(), reverses nothing.
 *
 * @param text    non-null String that will have
 *                its end reversed
 * @param start   the index at which the
 *                remainder of the input is
 *                reversed, requires 0 <=
 *                start <= text.length()
 * @return input text with the substring from
 *               start to the end of the string
 *               reversed
 */
static String reverseEnd(String text, int start)
```

Document the strategy at the top of the test class:

```
/*
 * Testing strategy
 *
 * Partition the inputs as follows:
 * text.length(): 0, 1, > 1
 * start:         0, 1, 1 < start < text.length(),
 *                text.length() - 1, text.length()
 * text.length()-start: 0, 1, even > 1, odd > 1
 *
 * Include even- and odd-length reversals because
 * only odd has a middle element that doesn't move.
 *
 * Exhaustive Cartesian coverage of partitions.
 */
```

Each test method should have a comment above it saying how its test case was chosen, i.e. which parts of the partitions it covers:

```
// covers test.length() = 0,
//        start = 0 = text.length(),
//        text.length()-start = 0
@Test public void testEmpty() {
    assertEquals("", reverseEnd("", 0));
}
```