



JAVA 达摩班

测试与TDD

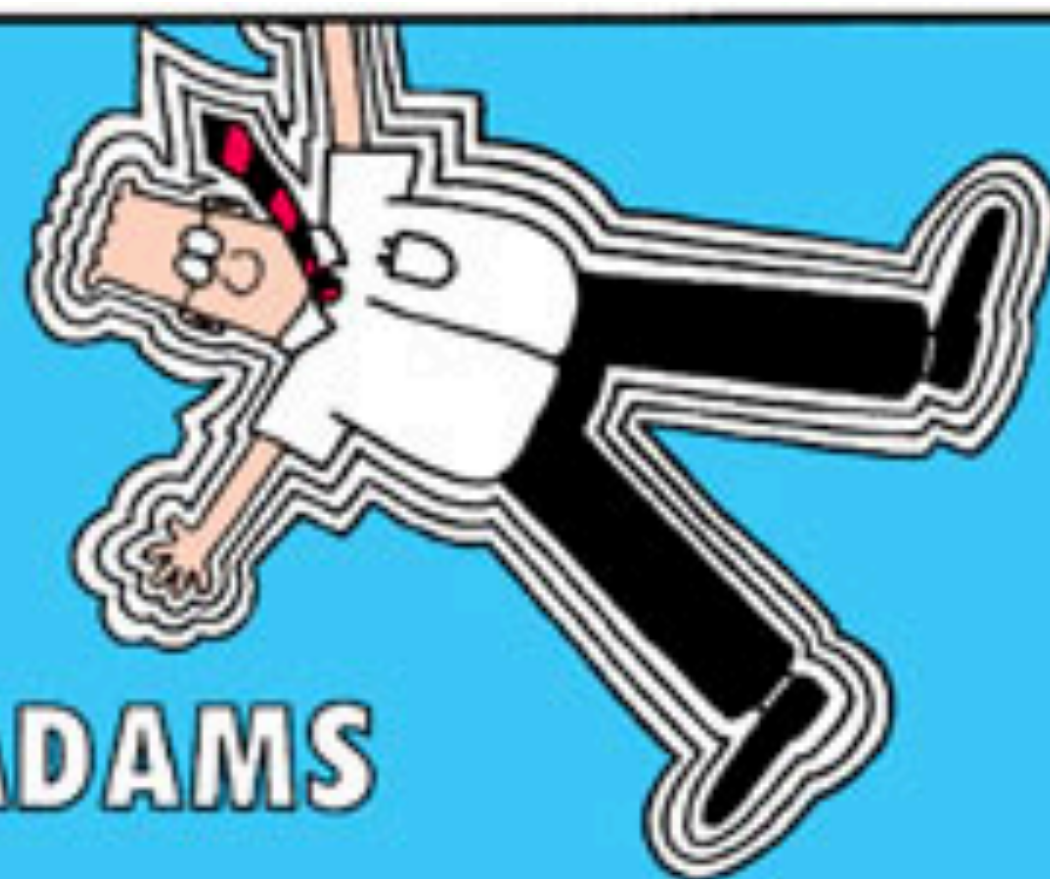


软件质量 与测试策略

1



DILBERT[®]



BY

SCOTT ADAMS



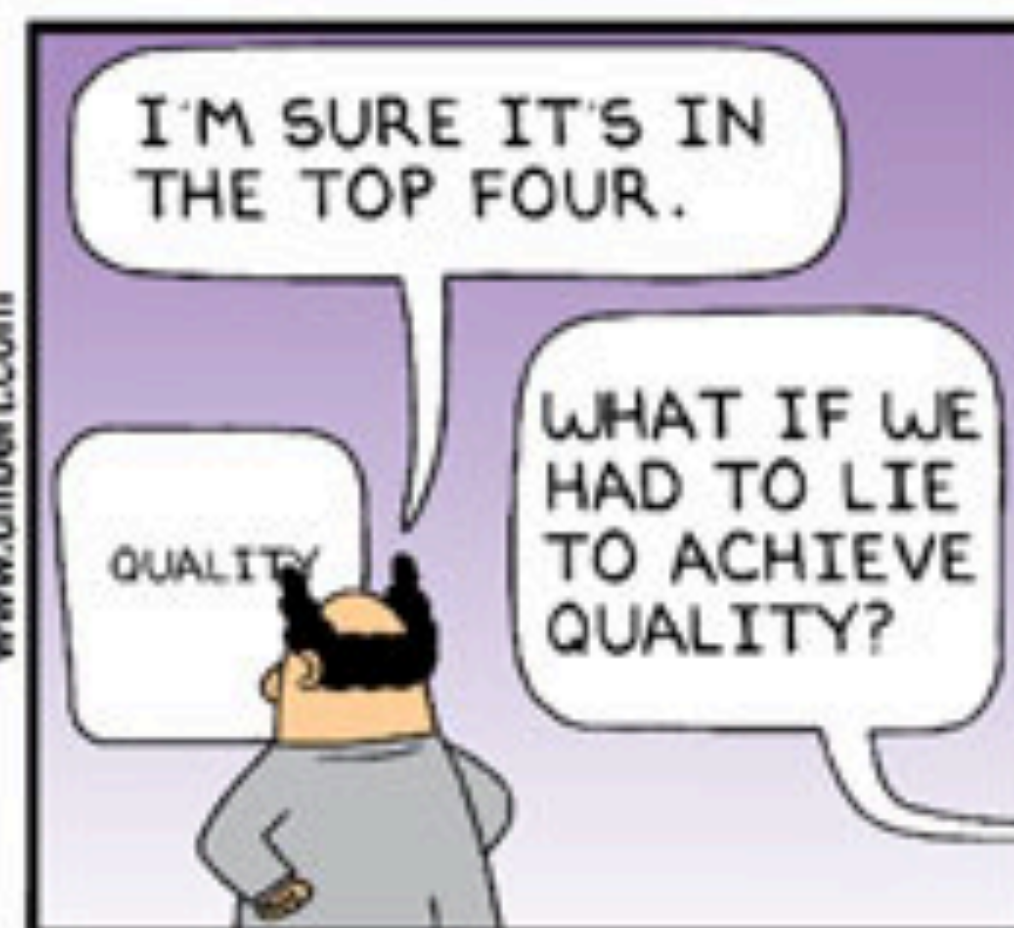
E-mail: SCOTTADAMS@AOL.COM



© 2004 Scott Adams, Inc. / Dist. by UFS, Inc.



www.dilbert.com



什么是QA?

测试是一组活动，它的目的是保证产品满足用户对系统性和可靠性需求。

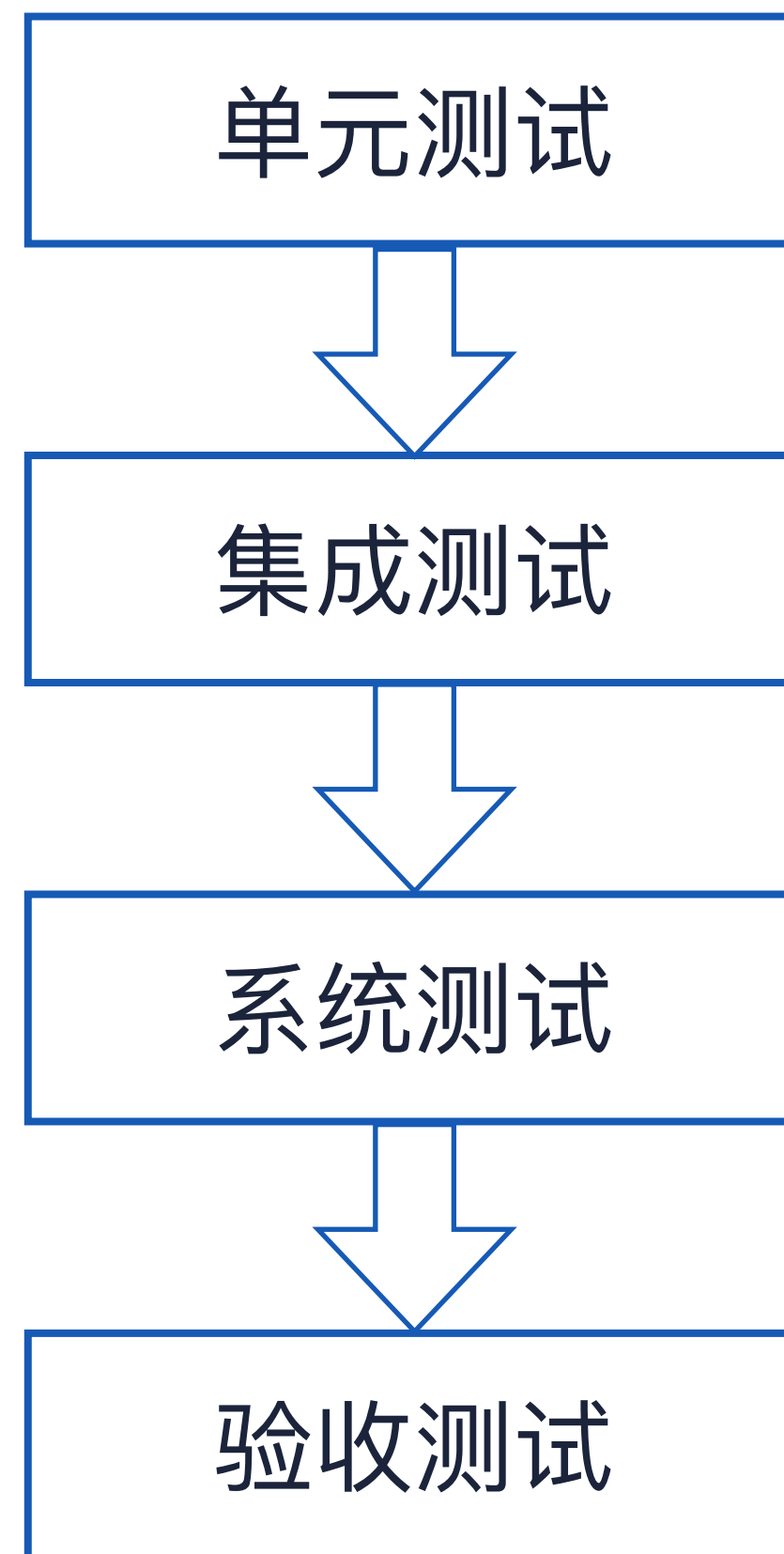
在敏捷文化中，Quality Assurance(QA)不仅仅是测试团队的职责，也是所有研发人员的职责。

QA是在新产品开发过程中的所有保证质量的行为。



概念

测试级别



测试方法

动态 vs 静态

开发 vs 独立

黑盒(行为) vs 白盒(结构) vs 灰盒

自动 vs 手动

测试类型

回归测试

性能测试

健全测试

探索性测试

冒烟测试

一致性/标准测试

可用性测试

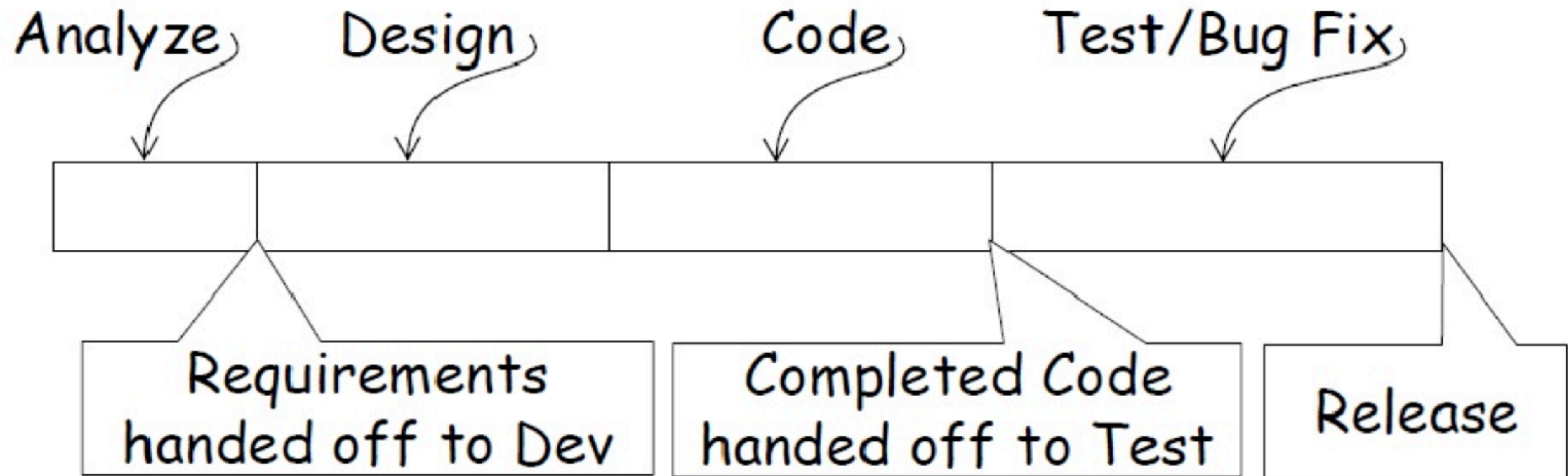
猴子/随机测试

功能性测试

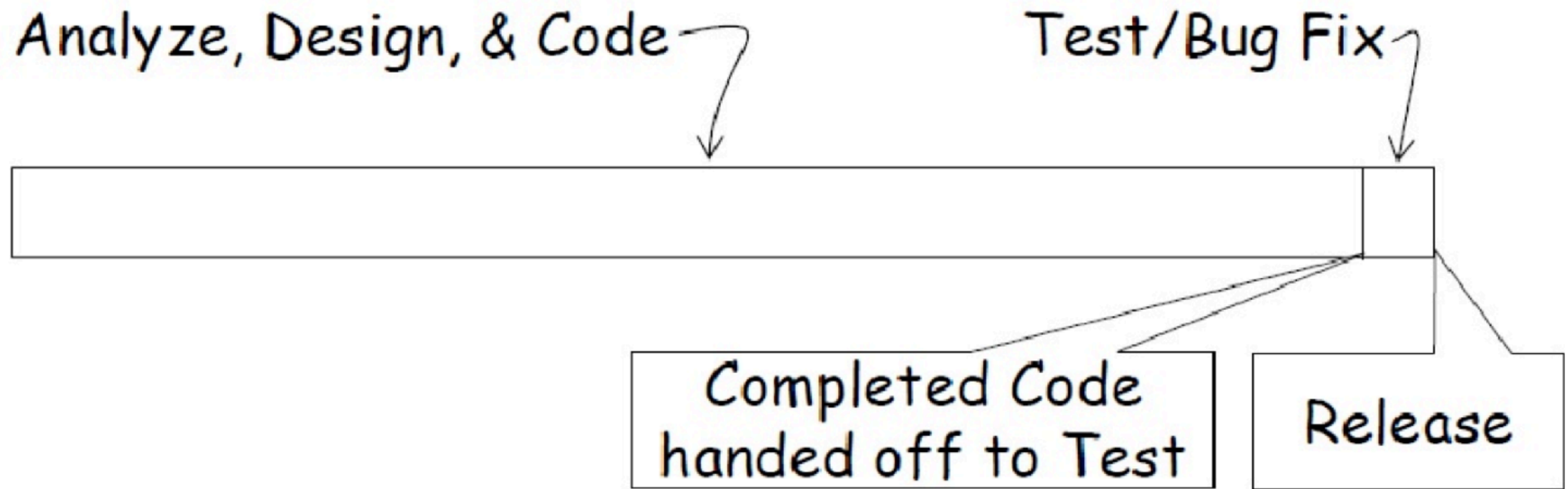
安全测试

.....

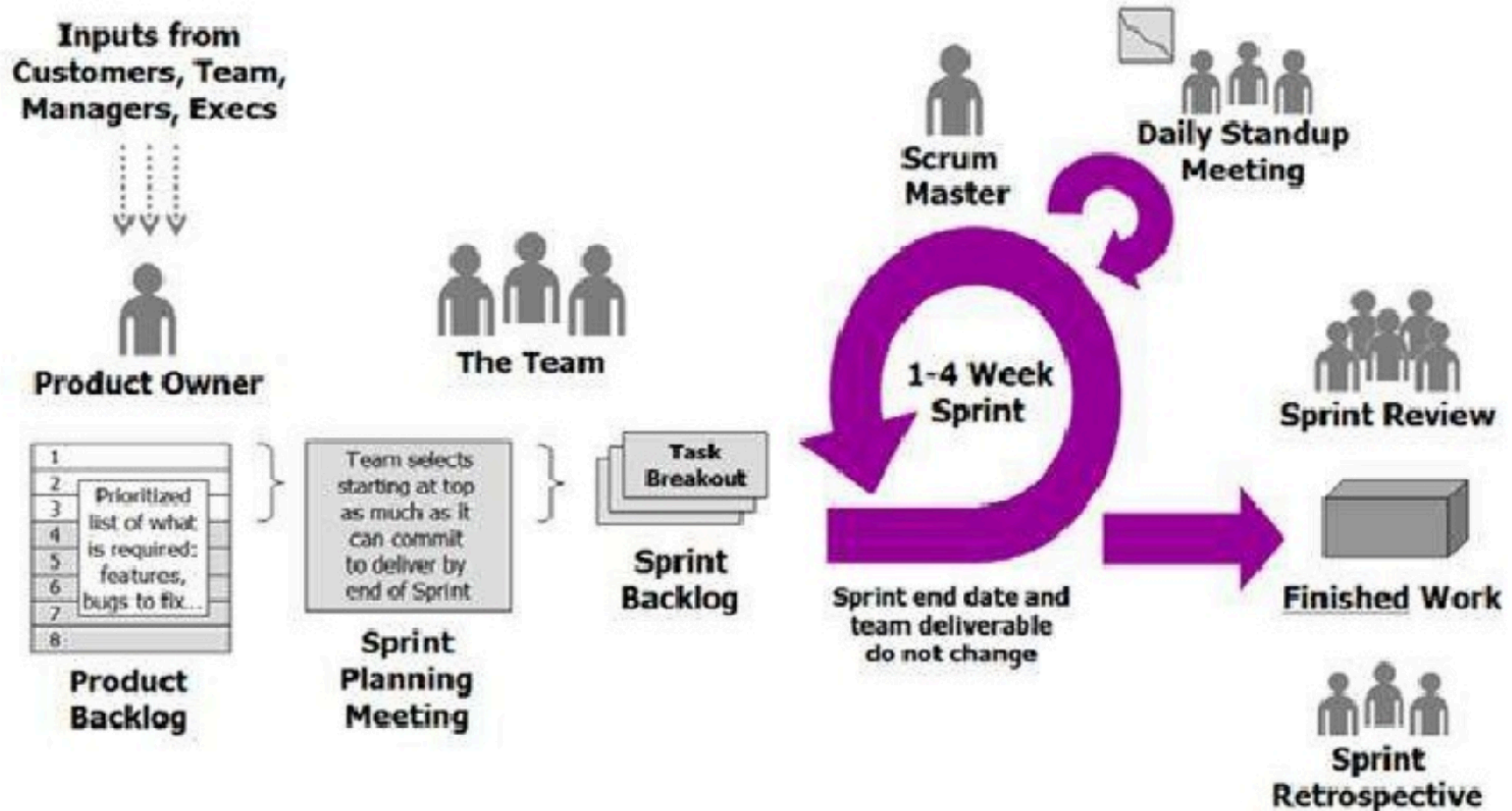
传统测试实践



传统测试实践

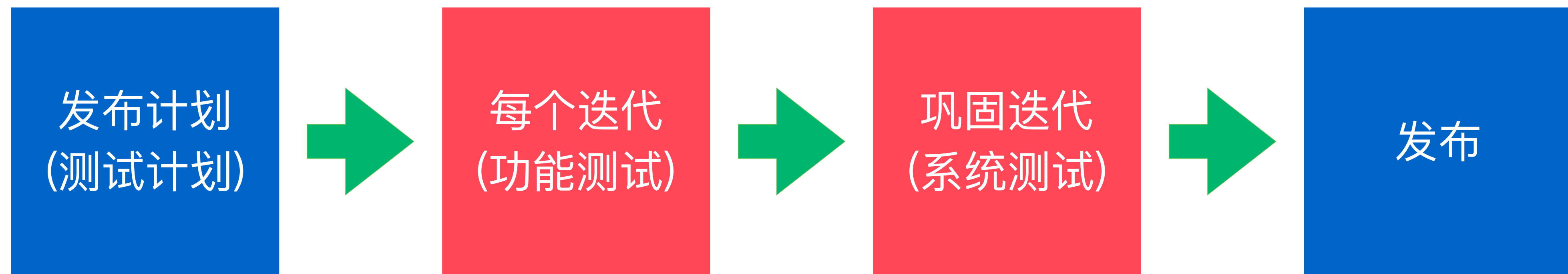


敏捷测试



敏捷策略

- 迭代测试，测试从早期就开始
- 面向团队的方法，测试人员参与感增强





单元测试 与Junit5

2

什么是单元测试？

单元测试是在应用**开发过程中**进行的质量保障活动

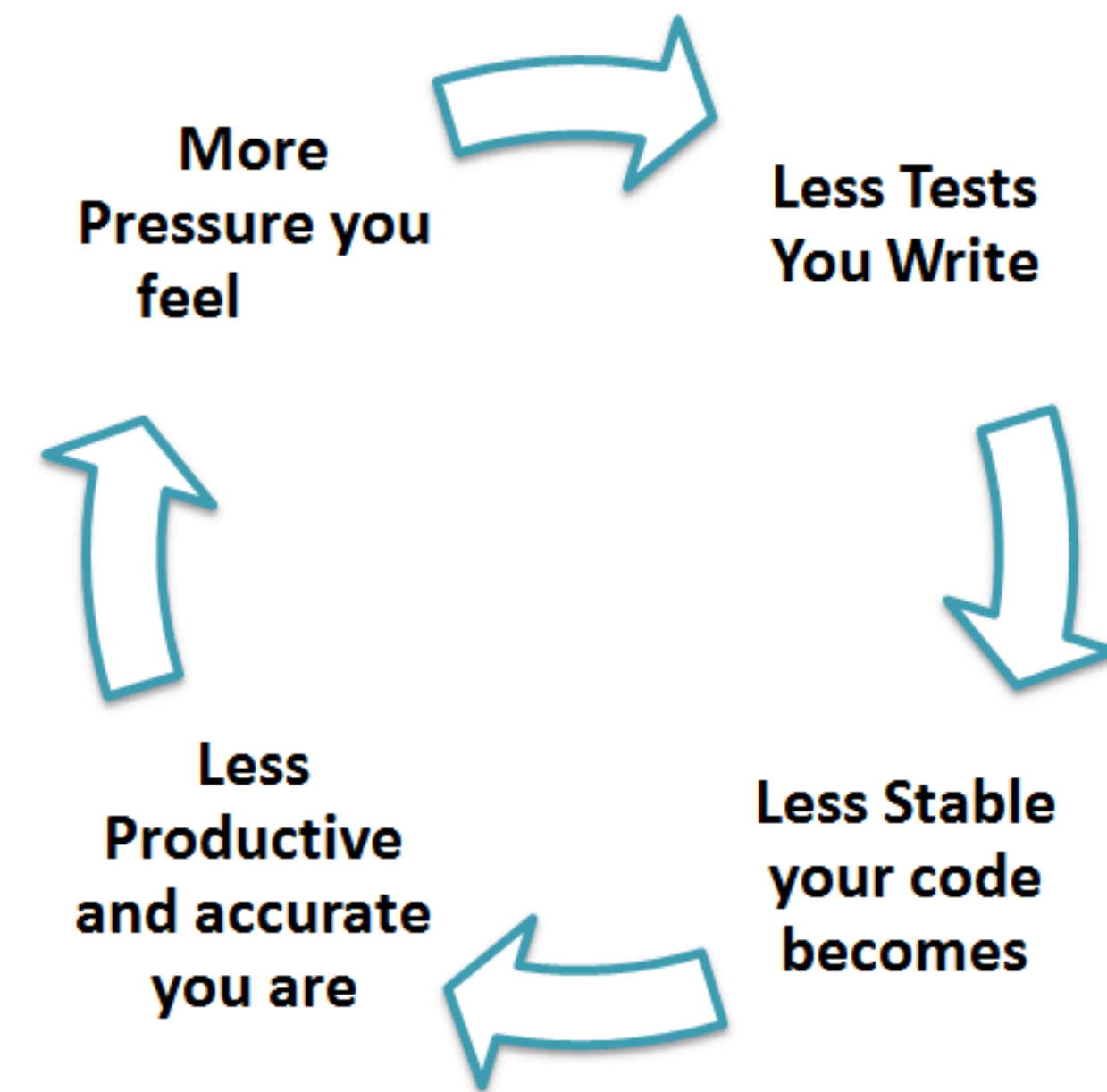
单元测试的目标隔离一块代码，并且验证它的正确性，它通常由**开发者完成**。

测试神话

“它需要时间，我没空”

“我的代码固若金汤，我不需要单元测试”

相反，单元测试会提高开发速度



什么是单元测试？

如何创建测试案例？

1. 自动化
 - 1.1 和业务代码在一起
 - 1.2 独立测试环境
2. 手动 - 步骤详细的文档

Mock对象

在单元测试中，为执行特定方法而创建的对象或者变量

单元测试工具

1. [Junit5](#)
2. [Mockito](#)

单元测试意义

优点：

- 通过单元测试了解功能
- 通过单元测试学习如何使用功能（调用）
- 对重构和bug修复友好
- 分模块测试（不需要等待其他模块完成）

局限性：

- 不要期望单元测试覆盖所有错误
- 单元测试仅限当前模块/功能，不考虑全局




单元测试方法

测试方法

- 结构化测试
- 功能性测试
- 基于错误的测试

最佳实践

- 测试案例是独立的
 - 每次仅覆盖一个特性
 - 清晰统一的命名规范
 - 功能修改之前，确保该功能的单元测试是正确的
 - 单元测试发现的bug要立刻修复
 - 坚持“test as your code”原则，测试越少，错误越难定位
- 

JUnit

JUnit 是目前最流行的Java单元测试框架，当前稳定版为4.12，5.1.1代表的下一代JUnit 5已经逐渐普及，它需要java8及更高，同时支持更多测试类型。

JUnit Platform： 用于在JVM环境中运行测试框架。提供了构建工具和JUnit之间的接口。通过TestEngine API，可以实现自己的测试框架并集成到平台中。

JUnit Jupiter： JUnit 5的实现，包含很多新特性：@TestFactory，@DisplayName，@Nested，@Tag，@ExtendWith，@BeforeEach，@AfterEach，@BeforeAll，@AfterAll，@Disable

JUnit Vintage： 用于在JUnit 5 平台上运行JUnit 3和JUnit 4的测试

<https://junit.org/junit5/docs/current/user-guide/>

<https://github.com/junit-team/junit5/wiki/Core-Principles>



JUnit 5 注释

@RunWith和**@ExtendWith**

JUnit 5的ExtendWith已经替代了JUnit 4的RunWith，而为了保证向下兼容，RunWith仍然可以使用。

@RunWith(JUnitPlatform.class) - JUnitPlatform可以让JUnit 4运行在JUnit Platform上，是一个基于JUnit 4的运行环境

@Test

用于表示下面的方法是测试用例

测试用例既不能为私有，也不能返回值，否则会被忽略



JUnit 5 注释

@BeforeAll

```
static void setup() {  
    log.info("@BeforeAll - executes once before all test methods in this class");  
}
```

@BeforeEach

```
void init() {  
    log.info("@BeforeEach - executes before each test method in this class");  
}
```

@AfterEach

```
void tearDown() {  
    log.info("@AfterEach - executed after each test method.");  
}
```

@AfterAll

```
static void done() {  
    log.info("@AfterAll - executed after all test methods.");  
}
```



JUnit 5 注释

```
@DisplayName("Single test successful")
```

```
@Test
```

```
void testSingleSuccessTest() {  
    log.info("Success");  
}
```

```
@Disabled("Not implemented yet")
```

```
@Test
```

```
void testShowSomething() {  
}
```



JUnit 5 注释

断言 (Assertions) 定义在org.junit.jupiter.api.Assertions, 支持lambda表达式

```
@Test
void lambdaExpressions() {
    assertTrue(Stream.of(1, 2, 3)
        .stream()
        .mapToInt(i -> i)
        .sum() > 5, () -> "Sum should be greater than 5");
}

@Test
void groupAssertions() {
    int[] numbers = {0, 1, 2, 3, 4};
    assertAll("numbers",
        () -> assertEquals(numbers[0], 1),
        () -> assertEquals(numbers[3], 3),
        () -> assertEquals(numbers[4], 1)
    );
}
```


JUnit 5 注释

假设（Assumptions）类似if语句，是当某个外部条件满足时执行断言。

```
@Test
void trueAssumption() {
    assumeTrue(5 > 1);
    assertEquals(5 + 2, 7);
}
```

```
@Test
void assumptionThat() {
    String someString = "Just a string";
    assumingThat(
        someString.equals("Just a string"),
        () -> assertEquals(2 + 2, 4)
    );
}
```



JUnit 5 注释

异常 (Exception) 两种方式: 1.比较异常信息; 2.直接比较异常类型;

```
@Test
void assertThrowsException() {
    String str = null;
    assertThrows(IllegalArgumentException.class, () -> {
        Integer.valueOf(str);
    });
}
```

```
@Test
void shouldThrowException() {
    Throwable exception = assertThrows(UnsupportedOperationException.class, () -> {
        throw new UnsupportedOperationException("Not supported");
    });
    assertEquals(exception.getMessage(), "Not supported");
}
```

JUnit 5 注释

@Nested

嵌套测试是指在测试用例class中嵌套定义测试用例class

@Tag

Tag用来对测试用例分组，实现测试目录功能

```
@Tag("Test case")
public class TaggedTest {
    @Test
    @Tag("Method")
    void testMethod() {
        assertEquals(2+2, 4);
    }
}
```

JUnit 5 注释

测试套件（Test Suites）有两种方式：一种基于包，一种基于类，都是用于同时测试多个类

```
@RunWith(JUnitPlatform.class)
@SelectPackages("com.dharma")
public class AllTests {}
```

```
@RunWith(JUnitPlatform.class)
@SelectClasses({AssertionTest.class, AssumptionTest.class,
ExceptionTest.class})
public class AllTests {}
```



JUnit 5 注释

静态测试由 `@Test` 定义，它是在编译时期实现的测试；
动态测试由 `@TestFactory` 定义，它是在运行时动态生成的。

`@TestFactory` 方法必须返回 `Stream`, `Collection`, `Iterable`, or `Iterator` 实例，否则会报错 `JUnitException`。

`@TestFactory` 方法不能为 `static` 或 `private`。

动态测试不在标准测试生命周期内，即 `@BeforeEach` `@AfterEach` 方法不会执行。

动态测试的目的是为第三方框架或者插件提供扩展点。



JUnit 5 注释

@TestFactory

```
public Stream<DynamicTest> translateDynamicTestsFromStream() {  
    return in.stream()  
        .map(word ->  
            DynamicTest.dynamicTest("Test translate " + word, () -> {  
                int id = in.indexOf(word);  
                assertEquals(out.get(id), translate(word));  
            })  
        );  
}
```

@TestFactory

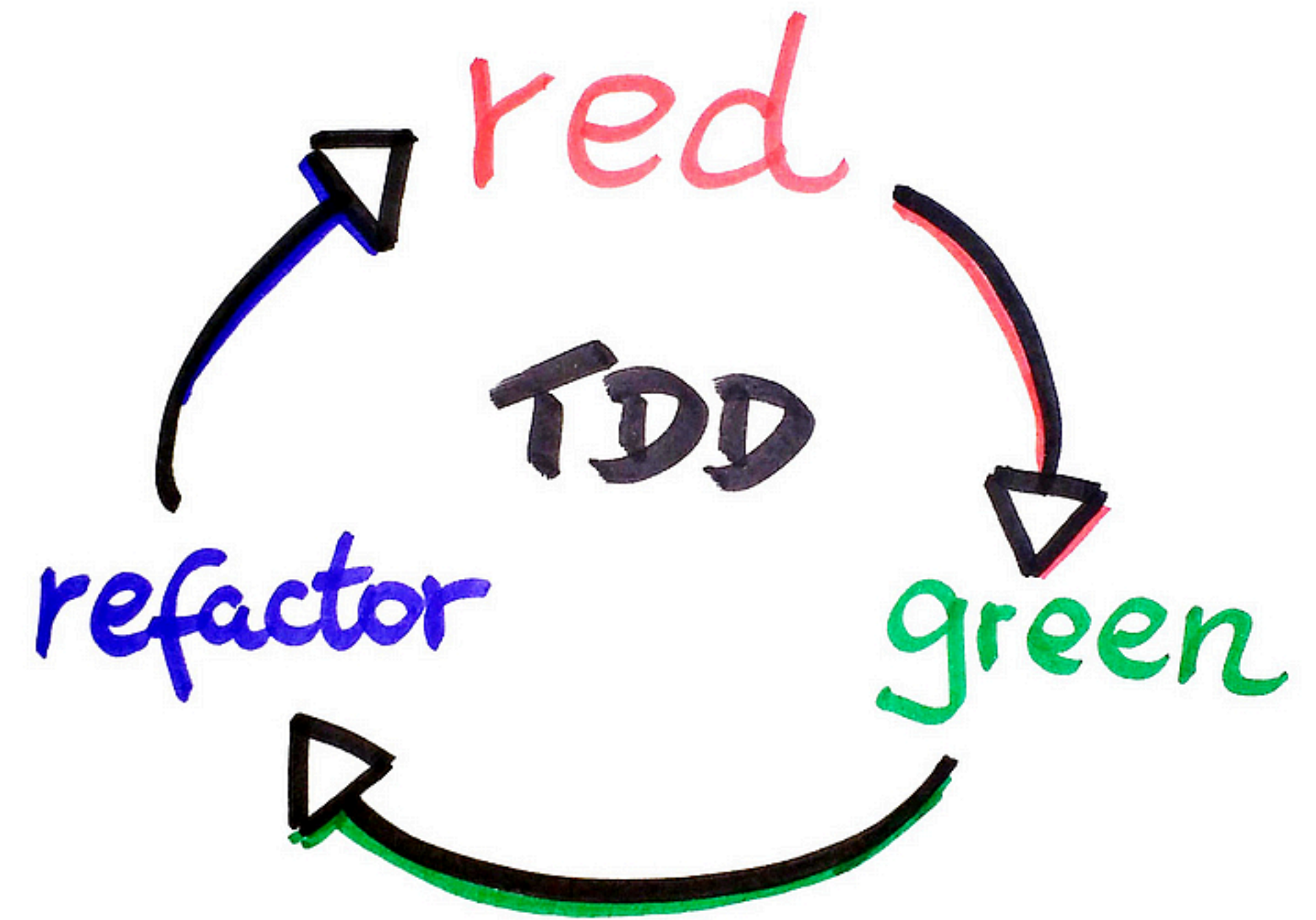
```
Collection<DynamicTest> dynamicTestsWithCollection() {  
    return Arrays.asList(  
        DynamicTest.dynamicTest("Add test", () -> assertEquals(2, Math.addExact(1, 1))),  
        DynamicTest.dynamicTest("Multiply Test", () -> assertEquals(4, Math.multiplyExact(2, 2)));  
}
```



测试驱动开发 3

TDD

1. 明确业务功能需求
2. 根据业务建立测试用例
 - 2.1 测试类
 - 2.2 测试用例
 - 2.3 测试函数体（满足业务）
3. 根据（失败）测试建立业务类
 - 3.1 业务类
 - 3.2 业务方法（空函数体）
4. 实现业务（red-green-refactor）
 - 4.1 运行测试（失败）
 - 4.2 实现业务
 - 4.3 运行测试（通过）
 - 4.4 重构代码
 - 4.5 运行测试（通过）



作业

1. TDD实现计算器程序（加减乘除）
2. 对比Sanity Test, Smoke Test和Regression Test





Thanks!

Any questions?

