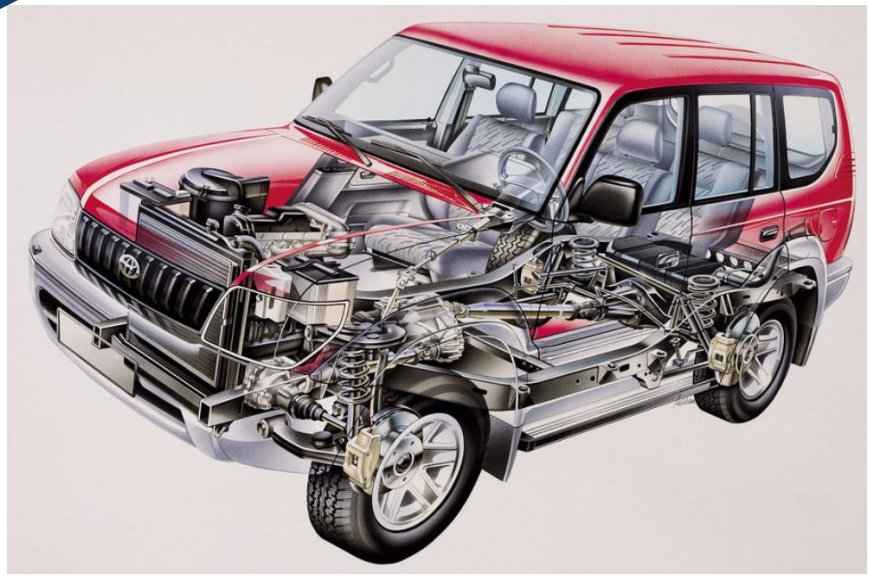
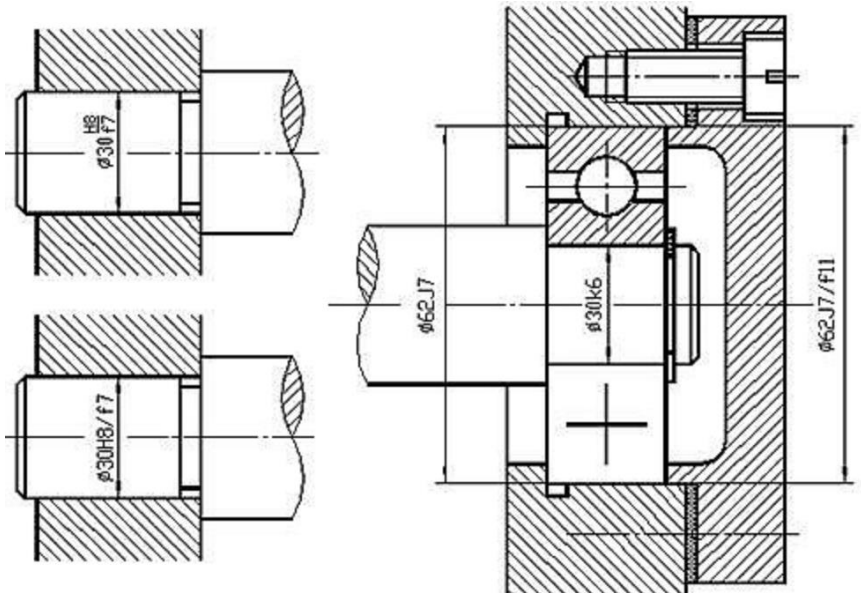
CS2001
软件工程

4. 个人开发技能
—测试驱动的开发



按规格生产并进行质量检验，合格后才能集成装配！

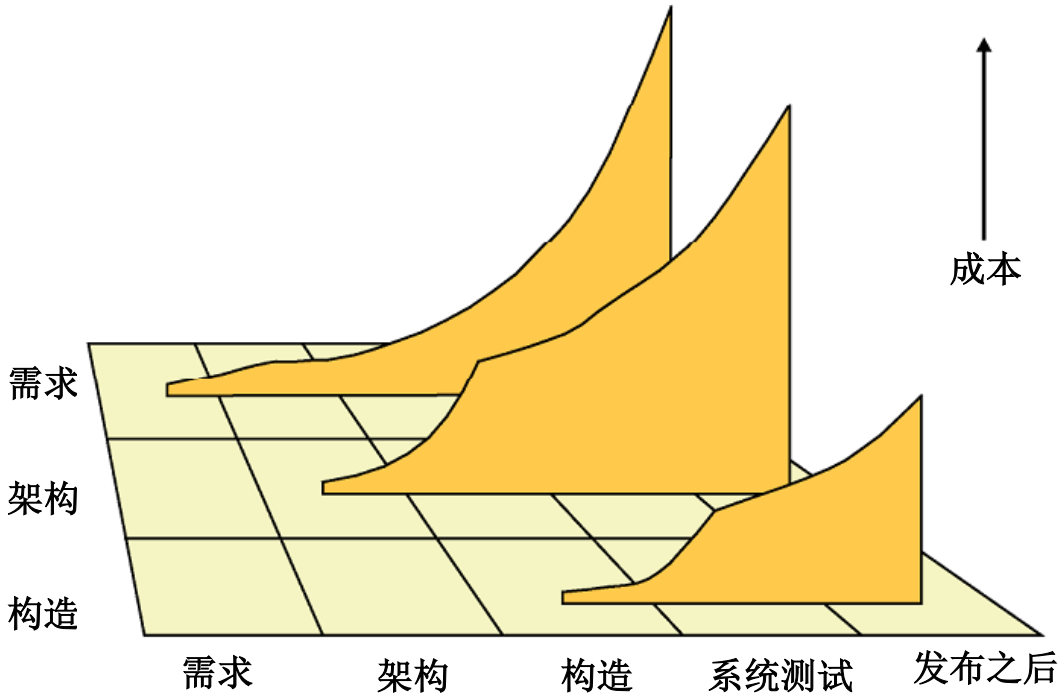


规格 SPECIFICATION / 机型 MODEL	SRL-4S (7B)		SRL-6S (10B)		SRL-8S (13B)		SRL-13S (19B)		SRL-16S (24B)		SRL-20S (30B)		SRL-25S (36B)		SRL-30S (45B)	
模数 FORGING STATION	5	6	5	6	5	6	5	6	5	6	5	6	5	6	5	6
最大切断直径 CUT OFF DIA.MAX (mm)	7		12		16		22		26		30		34		45	
最大切断长度 CUT OFF LENGTH MAX (mm)	40		45		60		70		80		100		140		180	
后托顶出长度 KICK OUT LENGTH MAX (mm)	40		45		70		100		110		125		150		180	
前托顶出长度 PKO LENGTH MAX (mm)	10		10		20		20		30		35		40		55	
夹仔翻转长度 MAX CHUCK TURN LENGTH (mm)	18		20		40		50		60		80		90		100	
切模 CUT OFF DIA.ILL(DXL) (mm)	24×30		40×60		50×80		70×75		85×90		95×100		110×155		138×165	
母模 MAIN DIE HOLE(DXL) (mm)	38×80		50×85		75×110		90×140		100×160		125×200		155×220		195×250	
母模中心距 DIES PITCH (mm)	48		69		87		114		126		138		174		210	
公模束 PUNCH HOLE(DXL) (mm)	32×70		40×80		50×90		65×120		75×155		90×200		110×220		135×280	
主滑板行程 RAM STROKE (mm)	100		100		140		175		200		250		280		320	
压造力 FORGING POWER (ton)	25		55	65	110	135	185	220	220	260	300	400	460	550	800	900
每分钟最大产能 MAX OUTPUT (pcs/minute)	180		160	150	140	130	120	110	90		65		60		50	
主马达 MAIN MOTOR (kw)	10		15	20	30	40	50	75	100		125		150		300	
润滑油用量 LUBRICANT OIL (L)	60		200		200		400		400		500		600		600	
冷却油用量 COOLANT OIL (L)	60		300		300		600		600		700		800		800	
长×宽 LENGTH×WIDT (M)	25×1.3	25×1.4	35×2.3	35×2.5	35×2.5	45×3.5	6×4	6×4.5	7×5	7×5.5	9.5×6	9.5×6	11×7	11×7.5	12.5×7	13×7.5
机械重量 APPROX.WT (ton)	6	7	11	13	16	18	26	30	45	54	74	80	98	108	140	155

软件的“零部件”问题

- 软件的基本单元：模块、类、方法、函数等
- 软件单元的问题
  - ✓ 软件单元的开发要求不清楚
  - ✓ 开发人员对软件单元开发要求的误解或疏漏
  - ✓ 软件单元内部实现变化导致外部特性改变
  - ✓ 软件单元性能等非功能性质量不符合要求

缺陷成本递增规律



	检测到缺陷的时间				
引入缺陷的时间	需求	架构	构造	系统测试	发布之后
需求	1	3	5-10	10	10-100
架构	-	1	10	15	25-100
构造	-	-	1	10	10-25

尽早测试，尽早发现缺陷！

单元测试

- 针对程序基本组成单元的一种测试
  - ✓ 方法、函数
  - ✓ 面向对象程序中的类
- 一种基本的开发者测试
- 单元测试的价值
  - ✓ 比系统测试更靠近开发阶段，能够更快的发现软件中存在的错误，从而避免在后续阶段的浪费
  - ✓ 隔离了代码单元和其它部分的依赖
  - ✓ 在测试执行上所用的时间和所需的信息更少，对错误定位的速度也更快

单元测试测什么

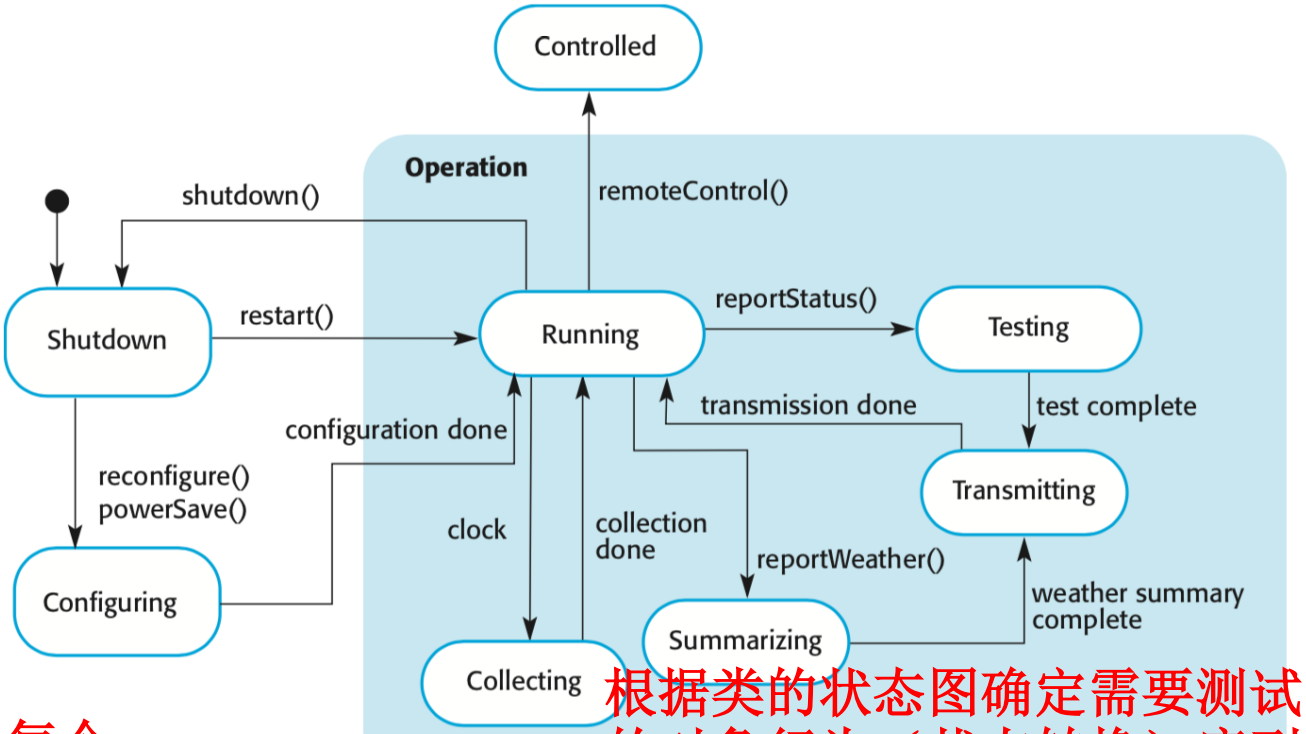
- 单个方法/函数
  - ✓ 测试方法/函数针对不同输入组合的行为
  - ✓ 对于有状态的对象可能需要一些初始化准备
  - ✓ 预期结果判断除了输出结果外还需考虑对象状态变化、异常抛出、外部数据（如文件）变化等
- 对象行为（有状态对象）
  - ✓ 对象从初始化到执行一系列操作的过程中的状态变化是否符合预期
  - ✓ 不仅取决于单个方法/函数正确性，还取决于对象的整体状态和行为控制
- 可以以类和方法契约为依据设计测试用例集合

WeatherStation类测试示例

WeatherStation

identifier

reportWeather ( )
reportStatus ( )
powerSave (instruments)
remoteControl (commands)
reconfigure (commands)
restart (instruments)
shutdown (instruments)



针对各个方法进行测试（每个方法都需要考虑多种输入组合）

根据类的状态图确定需要测试的对象行为（状态转换）序列

需要测试的对象行为序列（部分）

Shutdown → Running → Shutdown
Configuring → Running → Testing → Transmitting → Running
Running → Collecting → Running → Summarizing → Transmitting → Running

野外气象站系统案例介绍详见《软件工程》教材

单元测试的一些要点

- 单元测试是开发者测试
  - ✓ 开发人员能够更容易地理解要测试的目标
  - ✓ 发现问题时能够更快的修复
- 单元测试应尽可能自动化
  - ✓ 自动化的方式有助于降低单元测试的总体运行成本，提高开发人员运行单元测试的频率
- 单元测试应该隔离依赖
  - ✓ 如果被依赖的部分发生变化，很可能会导致需要测试的代码单元出现错误的行为

好的单元测试

- 基本单元：应该在最基本的功能/参数上验证程序的正确性
- 状态无损：测试过后机器状态保持不变
- 可重复：应该产生可重复、一致的结果
- 覆盖性：应该尽可能覆盖所有代码路径
- 独立性：测试结果不依赖于其他单元或外部因素，可用各种“替身”隔离外部影响
- 自动化：单元测试需要频繁重复执行，应该自动化且执行速度很快

自动化单元测试框架xUnit

```
Public class TestString {

        @Test
        Public void test_compare_to_same_string_should_be_zero() {
                String s_1 = "abc";
                String s_2 = "abc";
                assertEquals(0, s_1.compareTo(s_2));
        }

        @Test
        Public void test_compare_to_smaller_string_should_be_positive() {
                String s_1 = "abc";
                String s_2 = "ab0";
                assertTrue(s_1.compareTo(s_2) > 0);
        }
}
```

如Junit、CPPUnit

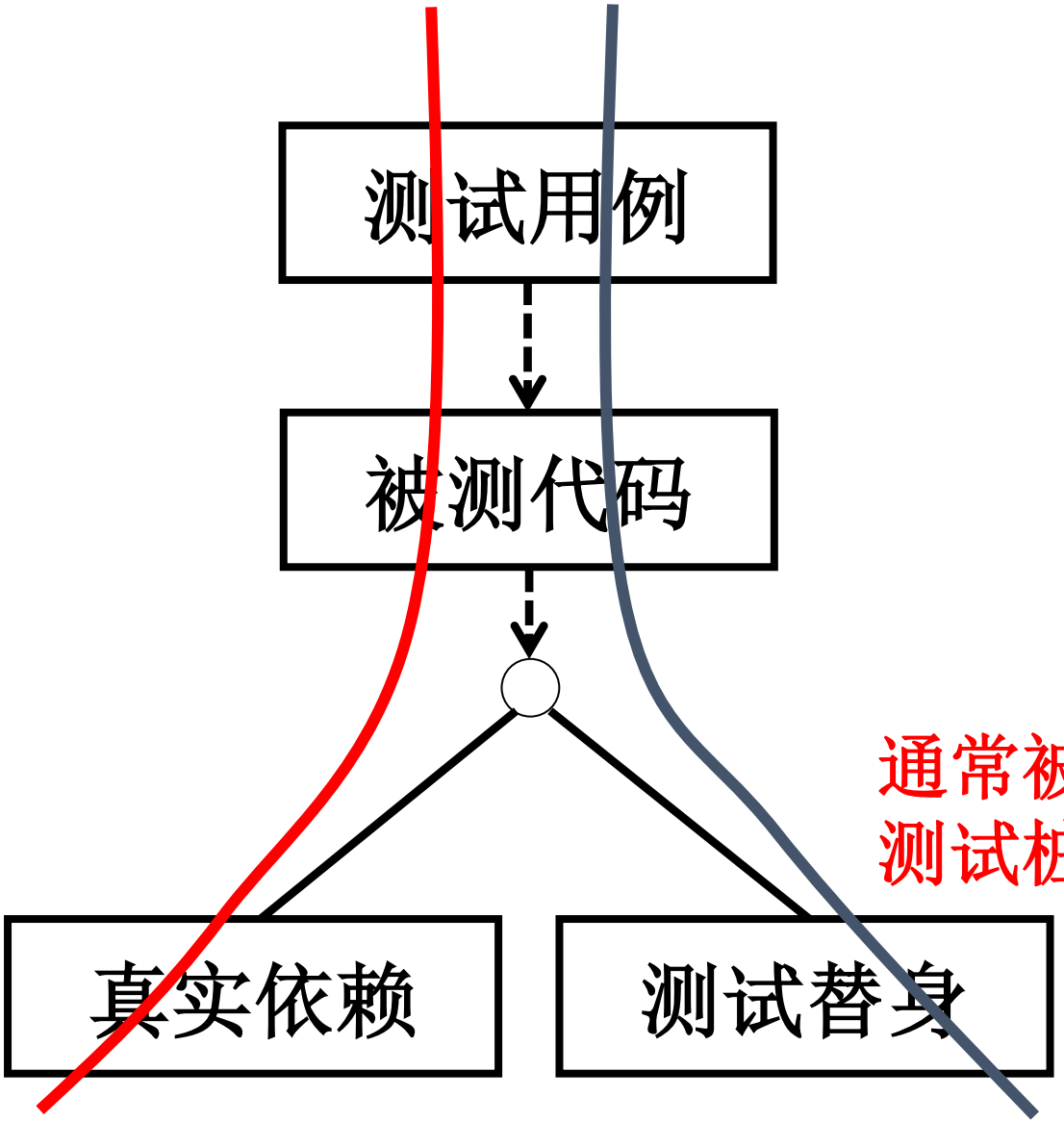
自动化单元测试

- 测试用例是可执行的测试类和测试方法
- 测试用例的编写以规格说明为依据，无需了解被测类或方法的内部实现
- 哪怕单个方法/函数也需要一组测试用例（考虑各种不同情况的组合）
- 测试用例可以多次重复运行，并随着被测代码一起进行维护和更新

单元测试的四阶段模式

- 建立：建立被测代码的前置条件，从而能够开始进行测试
- 执行：调用被测单元的接口
- 验证：通过断言确定是否获得了预期的结果，从而判断被测代码的正确性
- 拆卸：拆卸测试夹具，从而将被测目标及环境恢复到测试前的初始状态，避免影响后续的测试

依赖和测试替身



通常被称为
测试桩（**Stub**）

测试替身的类型

- 从来不会被真正调用，仅仅是为了使得链接器可以进行链接
- 能够按照要求在特定时刻返回特定的值，从而使得被测代码能够跳转到相应的路径或者执行特定的行为
- 除了返回特定值外，还能够捕获由被测代码传出的参数以便确定被测代码所调用的服务或者传出的参数是否正确

测试替身示例

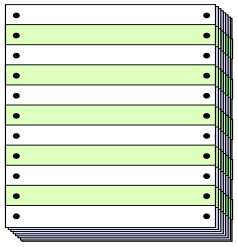
```
public interface ExchangeRate {
  double getRate(String inputCurrency, String outputCurrency);
}
public class TestCurrency {
 @Test
public void testToEuros() {
        Currency one_hundred_yuan = new Currency(100, "CNY");
        Currency ten_euro = new Currency(10, "EUR");
        ExchangeRate mock = EasyMock.createMock(ExchangeRate. class );
        EasyMock.expect(mock.getRate("CNY", "EUR")).andReturn(0.1);
        EasyMock.replay(mock);
        Currency exchange_result = one_hundred_yuan.toEuros(mock);
        assertEquals(ten_euro, exchange_result);
  }
}
```

使用EasyMock框架创建一个按照0.1返回人民币/欧元汇率的测试替身

测试用例的选择

测试耗时且昂贵，因此选择有效的单元测试用例是很重要的！

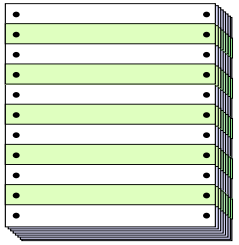
当按照期望的方式使用时，所测试的代码做了期望它做的事情



反映程序可以工作并正常运行的测试用例

例如，针对一个创建并初始化一条新的病人记录的程序，那么测试用例应当显示出所添加的记录在数据库中存在并且各字段取值按照所指定的值进行了设置

如果所测代码中存在缺陷，那么测试用例可以揭示这些缺陷



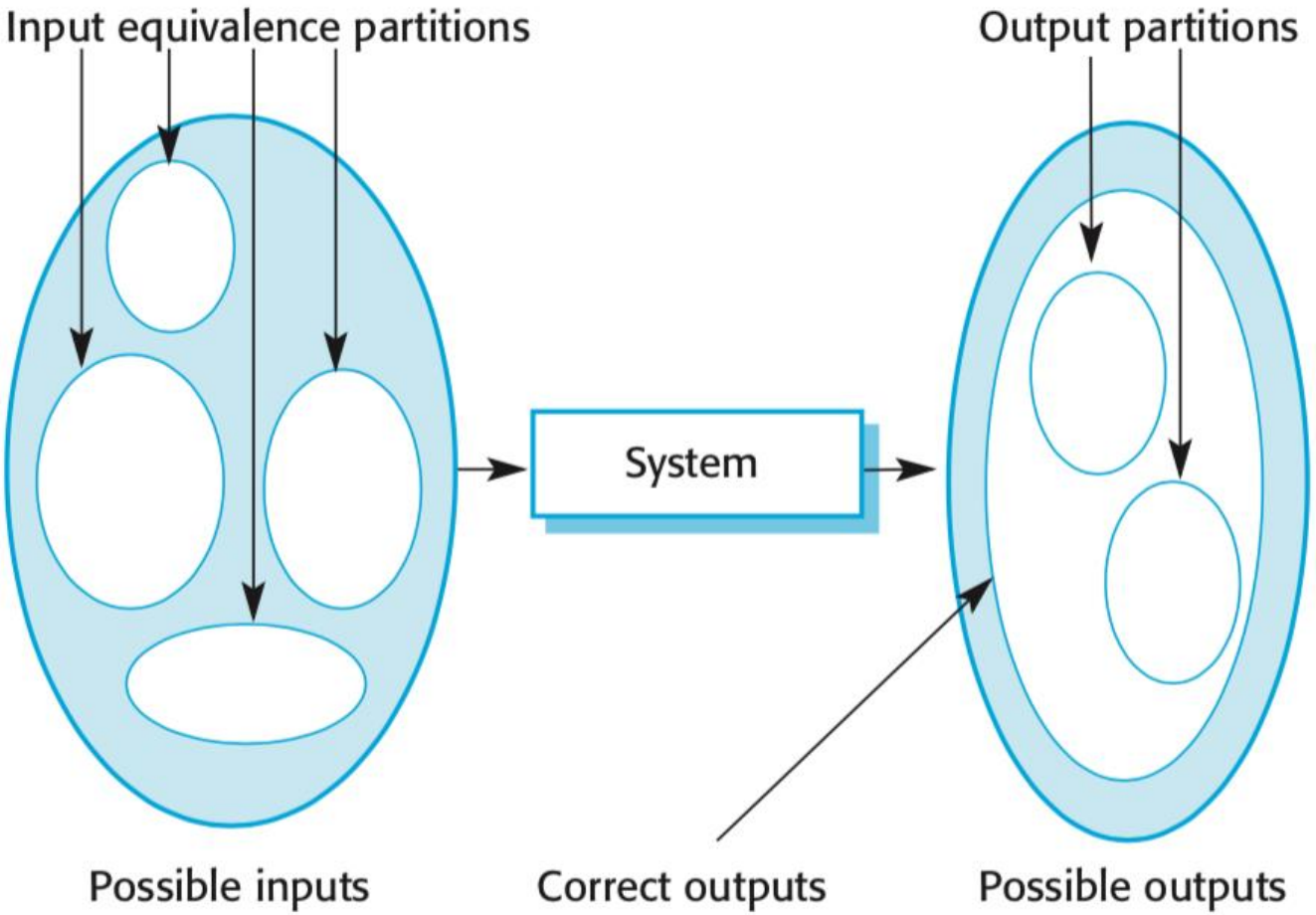
可以将常见的问题暴露出来的测试用例

例如，针对一个创建并初始化一条新的病人记录的程序，那么测试用例应当体现病人信息填写不全、身份ID格式不正确、姓名字段超长等容易导致程序出错的问题

常用的测试用例选取策略

- 划分测试
  - ✓ 识别出具有共同特性、处理方式相似的输入分组，然后从每个分组中选取测试用例
  - ✓ 例如，对于字符串比较可以划分如下分组：相等的非空字符串；不相等的非空字符串；两个空字符串；一个空另一个非空的字符串…
- 基于指南的测试
  - ✓ 根据关于程序员常犯的错误的经验指南来选择测试用例
  - ✓ 例如，输入值取值范围的上下界；超长或包含特殊字符的字符串输入；高频次服务请求中的资源泄漏…

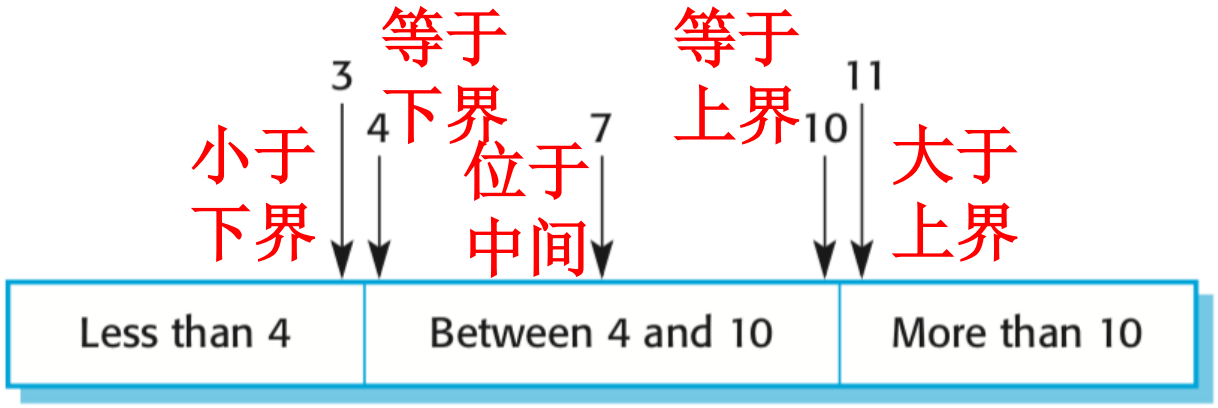
等价划分



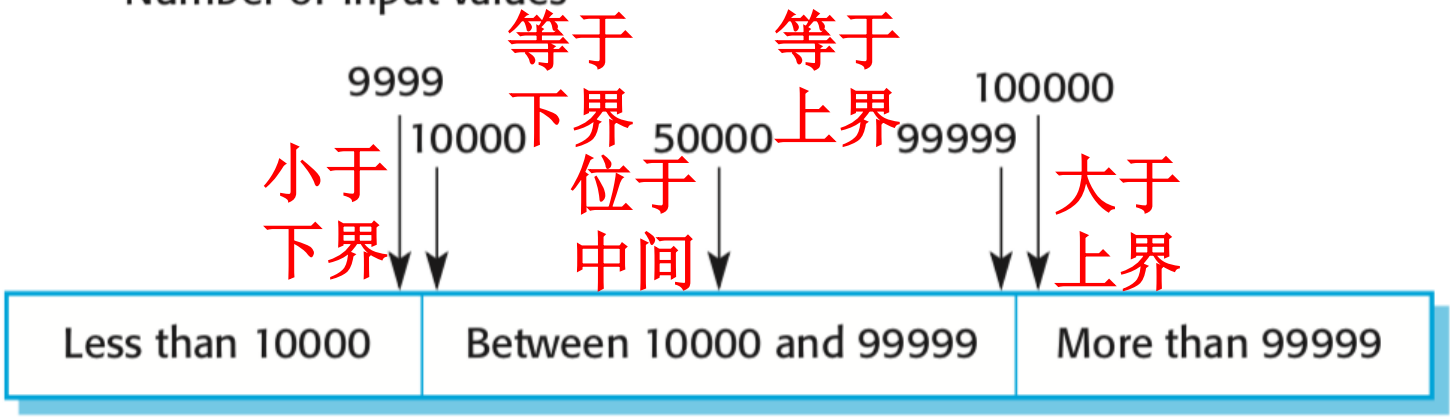
基于程序规格说明，针对输入和输出分别划分等价类，然后基于这些等价类选取测试用例

等价划分示例

程序规格说明：该程序接受4到10个大于10,000的五位整数作为输入



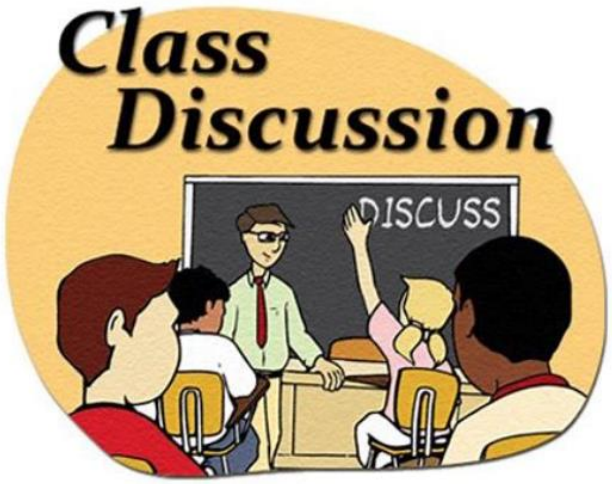
Number of input values



Input values

根据输入取值范围的上下界进行等价类划分，
然后根据多个输入的等价类组合选取测试用例

课堂讨论：单元测试用例设计



课堂讨论：根据DateUtil类的规格说明定义相应的单元测试用例

DateUtil：工具类，提供日期相关的一些计算和判断方法

DateUtil类（isValid方法）

isValid(CalendarDate)参数为日期对象，返回值是布尔值，true 表示日期对象为一个有效的日期，false 表示日 期对象为一个非法的日期。输入为 null 时返 回 false。日期有效是指现实中确实存在这一天。例如，2016-2-29是一个合法的日期，2017-2-29 是一个非法的日期。

测试用例1：传入null
测试用例2：合法日期（例如2018-04-02）
测试用例3：日超范围的非法日期（例如2018-04-32）
测试用例4：月超范围的非法日期（例如2018-13-01）
测试用例5：针对2月29日的非法日期（例如2018-02-29）
测试用例6：针对2月29日的合法日期（例如2016-02-29）
测试用例7：千年整年的特殊年份（1000-02-29）

isValid方法测试用例（部分）

传入null

```
@Test
public void testIsValidNull() {
    assertFalse(DateUtil.isValid(null));
}
```

传入非法日期

```
@Test
public void testIsValidFalse(){
    CalendarDate date = new CalendarDate(1900, 2, 29);
    assertFalse(DateUtil.isValid(date));
}
```

传入正常日期

```
@Test
public void testIsValidTrue(){
    CalendarDate date = new CalendarDate(2018, 4, 1);
    assertTrue(DateUtil.isValid(date));
}
```

DateUtil类（getDaysInMonth方法）

getDaysInMonth(CalendarDate)参数为日期对象，返回值为List<CalendarDate>，要求能够根据参数返回该日期所属的年份和月份的完整日期列表。List 大小应根据实际日期情况保证在 [28,31] 之间，并且已经按照日期由小到大顺 序排列。对于不合法的日期输入（包括空值和非法输入），返回 null 值

测试用例1：传入null
测试用例2：传入非法日期（存在多种情况）
测试用例3：传入合法日期（存在多种情况）

getDaysInMonth方法测试用例（部分）

传入null

```
@Test
public void testGetDaysInMonthNull() {
    assertNull(DateUtil.getDaysInMonth(null));
}
```

传入非法日期

```
@Test
public void testGetDaysInMonthIllegal(){
    CalendarDate date1 = new CalendarDate(2018, 2, 29);
    List<CalendarDate> actualList1 = DateUtil.getDaysInMonth(date1);
    assertNull(actualList1);

    CalendarDate date2 = new CalendarDate(2018, 22, 3);
    List<CalendarDate> actualList2 = DateUtil.getDaysInMonth(date2);
    assertNull(actualList2);

    CalendarDate date3 = new CalendarDate(2018, 3, 100);
    List<CalendarDate> actualList3 = DateUtil.getDaysInMonth(date3);
    assertNull(actualList3);

    CalendarDate date4 = new CalendarDate(2019, 33, 42);
    List<CalendarDate> actualList4 = DateUtil.getDaysInMonth(date4);
    assertNull(actualList4);
}
```

getDaysInMonth方法测试用例（部分）

传入合法日期

```
@Test
public void testGetDaysInMonthNotNull(){
    CalendarDate date = new CalendarDate(2018, 4, 2);
    List<CalendarDate> actualList = DateUtil.getDaysInMonth(date);

    List<CalendarDate> expectedList = new ArrayList<>();
    CalendarDate temp = null;
    for(int i = 0; i < 30; i++){
        temp = new CalendarDate(2018, 4, i+1);
        expectedList.add(temp);
    }

    assertEquals(expectedList.size(), actualList.size());

    for(int i = 0; i < expectedList.size(); i++){
        assertEquals(expectedList.get(i), actualList.get(i));
    }
}
```

getDaysInMonth与isValid交叉检验

```
@Test
public void testIntegration(){
    List<List<CalendarDate>> generatedCalendar = new ArrayList<>();
    for (int i = 1; i <= 20; i++){
        List<CalendarDate> monthCalendar = new ArrayList<>();
        for (int j = 1; j <= 31; j++){
            CalendarDate temp = new CalendarDate(2018, i, j);
            monthCalendar.add(temp);
        }
        generatedCalendar.add(monthCalendar);
    }

    for (int i = 1; i <= 20; i++){
        List<CalendarDate> tempCalendar = generatedCalendar.get(i - 1);
        int length = tempCalendar.size();
        CalendarDate temp = tempCalendar.get(length - 1);
        if (DateUtil.isValid(temp)){
            for (int j = 0; j < length; j++){
                assertEquals(tempCalendar.get(j), DateUtil.getDaysInMonth(temp).get(j));
            }
        }else {
            assertNull(DateUtil.getDaysInMonth(temp));
        }
    }
}
```

某月31日是合法日期，那么该月从1日到31日的日期对象列表应该与getDaysInMonth方法返回的列表相同

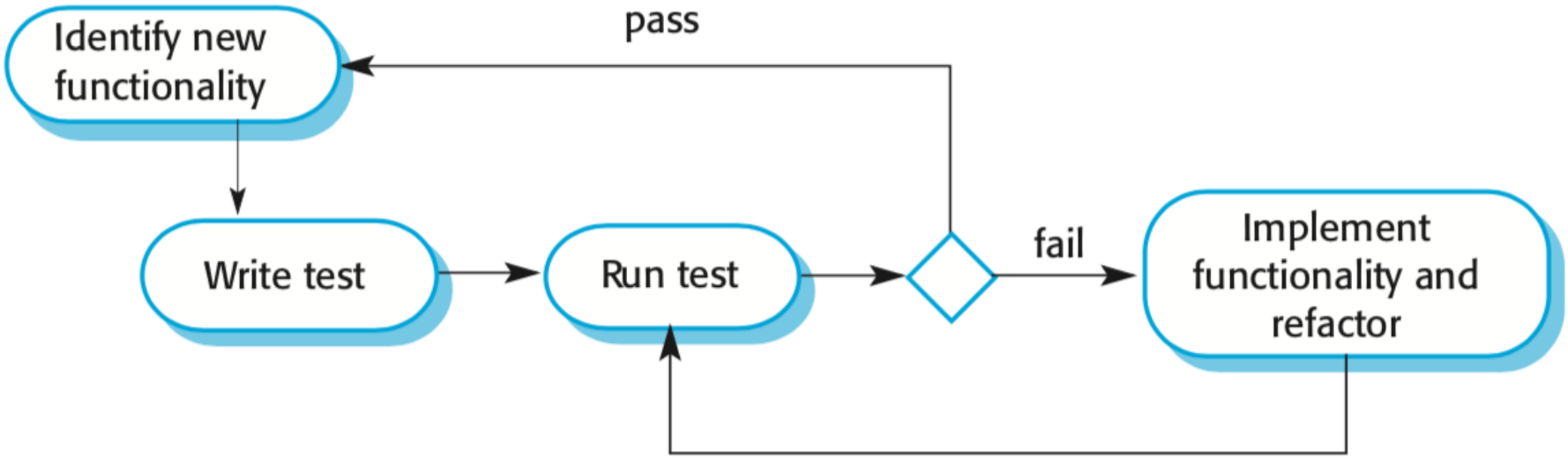
某月31日是非法日期，那么以该日期获得的getDaysInMonth方法返回结果是null

测试先行（Test First）

在设计和编码开始之前就编写测试

- 具备可行性：测试用例针对程序接口，不依赖于内部实现
- 精确的程序规格说明：明确定义程序接口及其规格说明，强化了需求理解
- 方便衡量开发进度：以通过的测试用例数量作为衡量开发进度的依据
- 提高测试效率：测试进度不会被落下，测试用例可在开发过程中反复使用

测试驱动的开发（Test Driven Development）



每次定义一个小的开发增量，为其定义好测试后然后进行代码编写和完善，同时不断运行测试直至测试用例都通过。

进行代码重构（内部优化）后也需要重新运行测试用例以确保程序的外部行为没有发生改变。

开发者测试的局限性

- 倾向于“干净”的测试
  - ✓ 主要关注于验证代码能否工作的测试（clean tests）而不是有可能让代码失效的测试（dirty tests）
  - ✓ 成熟的测试应当倾向于使二者的比例大约保持在1:5
- 对覆盖率的估计过于乐观
  - ✓ 程序员一般相信他们的测试覆盖度能达到95%，而事实上平均只有50-60%
- 往往忽略一些更复杂的覆盖度准则
  - ✓ 一般开发人员关注的是100%的语句覆盖度，这远远不够
  - ✓ 理想的覆盖度是100%的分支覆盖（可能代价很高）

从质量保障的角度看，开发者测试还远远不够，需要独立的软件测试作为补充

阅读建议

- 《软件工程》3.2.3、8.1.1、8.1.2、8.2
- 《构建之法》2.1
- 《代码大全》22.1、22.2

快速阅读后整理问题
在QQ群中提出并讨论

CS2001
软件工程

End

4. 个人开发技能
—测试驱动的开发