## 【案例】薪资管理系统的测试驱动开发（下）

### 测试驱动开发的过程

#### 满足简单设计并编写新的测试

当代码满足重用性和可读性之后，就应遵循简单设计的第四条原则“若无必要，勿增实体”，不要盲目地考虑为其增加新的软件元素。这时，需要暂时停止重构，编写新的测试。

现在，要测试加班的用例，需提供超过 8 小时的工作时间卡。测试代码已经定义了创建工作时间卡的方法，新测试的需求差异仅在于工作时长，为了测试代码的重用，可以提取 createTimeCards() 方法的参数，允许传入不同的工作时长。因此，新编写的测试如下所示：

@Test  
 public void should\_calculate\_payroll\_by\_work\_hours\_with\_overtime\_in\_a\_week() {  
 //given  
 List<TimeCard> timeCards = createTimeCards(9, 7, 10, 10, 8);  
 Money salaryOfHour = Money.of(10000, Currency.RMB);  
 HourlyEmployee hourlyEmployee = new HourlyEmployee(timeCards, salaryOfHour);  
  
 //when  
 Payroll payroll = hourlyEmployee.payroll();  
  
 //then  
 assertThat(payroll).isNotNull();  
 assertThat(payroll.beginDate()).isEqualTo(LocalDate.of(2019, 9, 2));  
 assertThat(payroll.endDate()).isEqualTo(LocalDate.of(2019, 9, 6));  
 assertThat(payroll.amount()).isEqualTo(Money.of(465000, Currency.RMB));  
 }

提供的工作时间卡包含了加班、正常工作时间、低于正常工作时间三种情况，综合计算钟点工的薪资。运行测试，测试失败。

#### 让失败的测试刚好通过

为该测试编写实现。

按照业务规则，加班时间的报酬会按照正常报酬的 1.5 倍进行支付，这就需要 Money 支持对 1.5 的乘法。在最初定义的 Money 类中，使用了 long 类型来代表面值，并以分作为货币单位。原本的 multiply() 方法支持的因数为 int 类型，现在需要改为支持 double 类型。为保证薪资的精确计算，可考虑使用 BigDecimal 来实现，这就需要先修改 Money 类的定义。

这相当于新的测试对原有产品代码提出了新的要求。需要暂时搁置对新测试的实现，而是对已有产品代码按照新的需求进行调整，修改 Money 类的定义，并在修改后运行已有的所有测试，确保这一修改并未破坏原有测试。接下来，实现刚才编写的新测试：

public Payroll payroll() {  
 int regularHours = timeCards.stream()  
 .map(tc -> tc.workHours() > 8 ? 8 : tc.workHours())  
 .reduce(0, (hours, total) -> hours + total);  
  
 int overtimeHours = timeCards.stream()  
 .filter(tc -> tc.workHours() > 8)  
 .map(tc -> tc.workHours() - 8)  
 .reduce(0, (hours, total) -> hours + total);  
  
 Money regularSalary = salaryOfHour.multiply(regularHours);  
 // 修改了 multiply() 方法的定义，支持 double 类型  
 Money overtimeSalary = salaryOfHour.multiply(1.5).multiply(overtimeHours);  
 Money totalSalary = regularSalary.add(overtimeSalary);  
  
 return new Payroll(  
 settlementPeriod().beginDate,  
 settlementPeriod().endDate,  
 totalSalary);  
 }

测试通过。

#### 重构以改进代码的质量

现在仍然按照简单设计原则消除重复，提高代码可读性。首先，可以提取 8 和 1.5 这样的常量，对代码作微量调整。仔细阅读实现代码对 filter 与 map 函数的调用，发现这些函数接收的lambda表达式，操作的数据皆为 TimeCard 类所拥有。遵循“信息专家模式”，且遵循对象之间采用行为协作，避免协作对象成为数据提供者，需要将这些表达式提取为方法，然后将提取出来的方法转移到 TimeCard 类：

public class TimeCard implements Comparable<TimeCard> {  
 private static final int MAXIMUM\_REGULAR\_HOURS = 8;  
 private LocalDate workDay;  
 private int workHours;  
  
 public TimeCard(LocalDate workDay, int workHours) {  
 this.workDay = workDay;  
 this.workHours = workHours;  
 }  
  
 public int workHours() {  
 return this.workHours;  
 }  
  
 public LocalDate workDay() {  
 return this.workDay;  
 }  
  
 public boolean isOvertime() {  
 return workHours() > MAXIMUM\_REGULAR\_HOURS;  
 }  
  
 public int getOvertimeWorkHours() {  
 return workHours() - MAXIMUM\_REGULAR\_HOURS;  
 }  
  
 public int getRegularWorkHours() {  
 return isOvertime() ? MAXIMUM\_REGULAR\_HOURS : workHours();  
 }  
}

这一重构说明，只要时刻注意对象之间正确的协作模式，就能避免出现贫血模型。领域对象持有的行为并非刻意追求，而是通过识别代码坏味道，遵循面向对象设计原则逐步形成的。经过如此重构后，payroll() 方法的实现为：

public Payroll payroll() {  
 int regularHours = timeCards.stream()  
 .map(TimeCard::getRegularWorkHours)  
 .reduce(0, (hours, total) -> hours + total);  
  
 int overtimeHours = timeCards.stream()  
 .filter(TimeCard::isOvertime)  
 .map(TimeCard::getOvertimeWorkHours)  
 .reduce(0, (hours, total) -> hours + total);  
  
 Money regularSalary = salaryOfHour.multiply(regularHours);  
 Money overtimeSalary = salaryOfHour.multiply(OVERTIME\_FACTOR).multiply(overtimeHours);  
 Money totalSalary = regularSalary.add(overtimeSalary);  
  
 return new Payroll(  
 settlementPeriod().beginDate,  
 settlementPeriod().endDate,  
 totalSalary);  
 }

显然，方法暴露了太多细节，缺乏足够的层次，因而无法清晰表达方法的执行步骤：先计算正常工作小时数的薪资，然后再计算加班小时数的薪资，即可得到该钟点工最终要发放的薪资。通过提取方法就能隐藏细节，使得主方法清晰地体现业务步骤：

public Payroll payroll() {  
 Money regularSalary = calculateRegularSalary();  
 Money overtimeSalary = calculateOvertimeSalary();  
 Money totalSalary = regularSalary.add(overtimeSalary);  
  
 return new Payroll(  
 settlementPeriod().beginDate,  
 settlementPeriod().endDate,  
 totalSalary);  
 }

这种将一个相对较长的主方法通过提取方法清晰表达其执行过程的模式，被 Kent Beck 称之为“组合方法”模式。他在著作 *Smalltalk Best Practice Patterns* 中阐释了这一模式：

* 把程序划分为方法，每个方法执行一个可识别的任务。
* 让一个方法中的所有操作处于相同的抽象层。
* 这会自然地产生包含许多小方法的程序，每个方法只包含少量代码。

提取方法是一种非常有效的重构手段。通过确定一个方法的高层目标，就可以识别和提取出无关的子问题域，让方法的职责变得更加单一，代码的层次更加清晰。方法在代码层次是一种非常有效的封装机制，它可以让细节不再直接暴露，只要提取出来的方法拥有一个“不言自明”的好名称，代码就能变得更加可读。

运行所有测试，全部通过。

#### 全面考虑任务的测试用例

现在还要考虑一种特殊的测试用例，即在结算薪资时，钟点工没有提交任何工作时间卡。在考虑这一测试方法的编写时， **发现一个问题** ：如何获得薪资的结算周期？之前的实现是通过提交的工作时间卡来获得结算周期的，如果钟点工根本没有提交工作时间卡，意味着该钟点工的薪资为 0，但并不等于说没有薪资结算周期。事实上，如果提交的工作时间卡存在缺失，也会导致获取薪资结算周期出错。以此而论，即可发现确定薪资结算周期的职责本身就不应该由 HourlyEmployee 来承担，该聚合也不具备该“知识”；然而，payroll() 方法返回的 Payroll 对象又需要有结算周期，故而结算周期的信息事实上应该由外部传入，这就 **保证了聚合的自给自足** ，无需访问任何外部资源。因此，在编写新测试之前，还需要先修改已有代码：

public Payroll payroll(Period period) {  
 Money regularSalary = calculateRegularSalary();  
 Money overtimeSalary = calculateOvertimeSalary();  
 Money totalSalary = regularSalary.add(overtimeSalary);  
  
 return new Payroll(  
 period.beginDate(),  
 period.endDate(),  
 totalSalary);  
 }

新增测试覆盖了没有工作时间卡的情况。注意以下两个测试应分别实现、分别验证，严格遵守测试驱动开发的红绿黄节奏：

@Test  
public void should\_be\_0\_given\_no\_any\_timecard() { }  
  
@Test  
public void should\_be\_0\_given\_null\_timecard() { }

在编写第二个测试时，发现之前的测试并没有考虑 List 为 null 的情况，导致测试未能通过。修改实现，对 null 与空列表二者都做了判断：

public Payroll payroll(Period period) {  
 if (Objects.isNull(timeCards) || timeCards.isEmpty()) {  
 return new Payroll(period.beginDate(), period.endDate(), Money.zero());  
 }  
  
 Money regularSalary = calculateRegularSalary();  
 Money overtimeSalary = calculateOvertimeSalary();  
 Money totalSalary = regularSalary.add(overtimeSalary);  
  
 return new Payroll(period.beginDate(), period.endDate(), totalSalary);  
 }

这充分说明，倘若编写的测试用例不完整，就会影响到产品代码的外部质量，甚至引入未知的缺陷。测试驱动开发确乎可以推动开发人员编写大量的单元测试或集成测试，并利用这些测试形成重构的保护网，但测试驱动开发自身并不能决定测试的质量与覆盖率。故而从分解的子任务到测试用例的编写是从设计迈向实现的关键一步，若能结合用户故事的验收标准来确定测试用例，又或者由测试人员参与，结对编写测试方法，会在一定程度保证测试用例的完整性。

#### 领域驱动设计对测试的影响

仔细检查测试的断言，发现对 Payroll 的验证并不完全，因为工资条必须要与员工对应，故而需要验证 Payroll 与员工相关的信息，其中一个关键属性是员工的 ID。因为 HourlyEmployee 是一个聚合根，按照领域驱动设计的要求，需要为实体定义一个唯一的身份标识。倘若 Payroll 也具有员工的ID，就能与 HourlyEmployee 关联。因此，为已有测试增加新的验证，由此驱动我们修改 HourlyEmployee 与 Payroll 模型的定义：

assertThat(payroll.employeId()).isEqualTo(employeeId);

现在我们已经完成了一个原子任务，可以删去该任务。这一做法一方面真实体现了项目开发的进展情况，让开发人员及时收获任务完成的成就感，另一方面也能指导我们的开发工作小步前进，有条不紊：

* 计算钟点工薪资
  + 获取钟点工雇员与工作时间卡
  + ~~根据雇员日薪计算薪资~~

#### 新任务的测试驱动开发

选择下一个任务。通常应选择与前一任务相关的任务，保证每个组合任务的原子任务能够有序地完成。由于原子任务“获取钟点工雇员与工作时间卡”会访问数据库，从单元测试的角度来讲，应以 Mock 形式实现，因此可以直接为组合任务“计算钟点工薪资”编写测试，由此驱动出 Repository 的接口。

在进行测试驱动开发时， **需要考虑测试用例的正交性** ：无需为相同的业务场景和相同的测试数据重复编写测试，尤其要考虑组合任务与原子任务之间的关系。由于测试驱动开发的方向是由内至外，即先为内部的原子任务编写测试，再为外部的组合任务编写测试。倘若原子任务的测试用例足够完整，在为组合任务编写测试时，就只需要为组合点的集成逻辑编写测试方法即可。

以组合任务“计算钟点工薪资”为例，由于原子任务“根据雇员日薪计算薪资”已经考虑了为单个钟点工计算薪资的各种情况，那么，组合任务“计算钟点工薪资”的测试用例就无需重复覆盖这些用例。这意味着，编写的测试只需考虑获取钟点工雇员数量的各种情形，但单个雇员与工作时间卡之间的各种情形就无需再考虑，否则就会形成测试用例的“组合爆炸”。因此，组合任务的测试用例就分为：

* 没有符合条件的钟点工雇员
* 只有一个钟点工雇员符合条件
* 多个钟点工雇员符合条件

根据场景驱动设计的要求，由领域服务 HourlyEmployeePayrollCalculator 履行组合任务“计算钟点工薪资”的职责。由于对 HourlyEmployeeRepository 资源库接口采用模拟的手段，由此可以站在调用者的角度驱动出它的接口。现在，按照测试驱动开发三定律的要求为新功能编写测试：

public class HourlyEmployeePayrollCalculatorTest {  
 @Test  
 public void should\_calculate\_payroll\_when\_no\_matched\_employee\_found() {  
 //given  
 HourlyEmployeeRepository mockRepo = mock(HourlyEmployeeRepository.class);  
 when(mockRepo.allEmployeesOf()).thenReturn(new ArrayList<>());  
  
 HourlyEmployeePayrollCalculator calculator = new HourlyEmployeePayrollCalculator();  
 calculator.setRepository(mockRepo);  
  
 Period settlementPeriod = new Period(LocalDate.of(2019, 9, 2), LocalDate.of(2019, 9, 6));  
  
 //when  
 List<Payroll> payrolls = calculator.execute(settlementPeriod);  
  
 //then  
 verify(mockRepo, times(1)).allEmployeesOf();  
 assertThat(payrolls).isNotNull().isEmpty();  
 }  
}

通过该测试可以驱动出领域服务、资源库和聚合之间的协作关系。测试的 Given 部分需要考虑如何定义 HourlyEmployeeRepository 接口及接口方法。When 部分驱动出领域服务的接口设计，包括方法名、输入参数和返回结果。场景驱动设计的时序图脚本可以作为参考，但不可拘泥于设计模型。由于在编写测试时能够更多地站在调用者角度去考虑，领域设计模型的结果就能在进一步细化的过程中向着正确的方向演化。Then 部分通过 Mockito 的 verify() 方法强调了领域服务与资源库的协作，并对领域服务方法返回结果的正确性进行了验证。

运行测试，失败。然后针对测试编写刚好令其通过的实现：

public class HourlyEmployeePayrollCalculator {  
 private HourlyEmployeeRepository employeeRepository;  
  
 public void setRepository(HourlyEmployeeRepository employeeRepository) {  
 this.employeeRepository = employeeRepository;  
 }  
  
 public List<Payroll> execute(Period settlementPeriod) {  
 List<HourlyEmployee> hourlyEmployees = employeeRepository.allEmployeesOf();  
 return hourlyEmployees.stream()  
 .map(e -> e.payroll(settlementPeriod))  
 .collect(Collectors.toList());  
 }  
}

组合任务除了引入了 Mockito 来模拟 HourlyEmployeeRepository 的行为之外，驱动开发的步骤与过程与之前实现的原子任务“根据雇员日薪计算薪资”并没有任何差别。为了节省文字，这里就不再继续赘述。唯一需要注意的是，随着测试的增加，必然需要为测试数据的准备提供工具方法。这在单元测试中，往往被称之为测试夹具（Fixture）。例如，为钟点工雇员提供创建方法：

public class EmployeeFixture {  
 public static HourlyEmployee hourlyEmployeeOf(String employeeId, List<TimeCard> timeCards) {  
 Money salaryOfHour = Money.of(100.00, Currency.RMB);  
 return new HourlyEmployee(employeeId, timeCards, salaryOfHour);  
 }  
  
 public static HourlyEmployee hourlyEmployeeOf(String employeeId, int workHours1, int workHours2, int workHours3, int workHours4, int workHours5) {  
 Money salaryOfHour = Money.of(100.00, Currency.RMB);  
 List<TimeCard> timeCards = createTimeCards(workHours1, workHours2, workHours3, workHours4, workHours5);  
 return new HourlyEmployee(employeeId, timeCards, salaryOfHour);  
 }  
}

若要了解薪资管理系统的整个驱动开发过程，可以在GitHub上获取我按照场景驱动设计与测试驱动开发结合的方式为该系统编写的代码，代码库为 <https://github.com/agiledon/payroll-ddd>。

### 场景驱动设计与测试驱动开发的相互作用

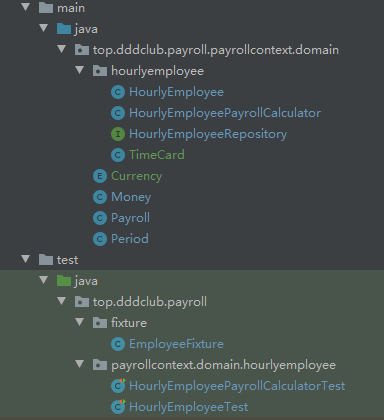
观察测试驱动开发的过程，若已有领域设计模型包括场景驱动设计作为指导，会让测试驱动开发的过程变得更加简单；反过来，测试驱动开发的过程又能通过编写测试代码和产品代码来验证领域模型的设计是否合理。例如组合任务“支付”的时序图脚本设计为：

PayingPayrollService.execute(employees) {  
 TransferClient.transfer(account);  
 PaymentRepository.add(payment);  
}

通过测试驱动开发完成了“计算雇员薪资”组合任务后，我们确定了领域服务方法返回的类型为 List。支付时，应对雇员的银行账户发起转账。在薪资管理系统中，账户 Account 属于另一个聚合，它与雇员之间存在关联关系。由于 Payroll 包含了 employeeId，故而领域服务 PayingPayrollService 还需要通过 AccountRepository 获得对应的账户信息，然后再对账户发起转账。

显然，在场景驱动设计的过程中，通过时序图的消息传递可以帮助我们思考接口设计，并编写出用例图脚本。然而，这个设计过程毕竟没有得到代码的验证，难免会出现设计误差或缺失。更何况，在没有编码实现之前，我们也不适宜在设计上花费太多的时间，以免导致过度设计。测试驱动开发能以快速反馈的实证主义，帮我们验证接口的合理性，然后在测试的保护下利用重构来改进代码质量。这正是我之所以建议将测试驱动开发与场景驱动设计相结合的主要原因。

随着测试驱动开发的逐步进行，驱动出越来越多的领域模型对象。我们需要按照领域驱动设计的要求与设计原则调整代码结构，例如按照限界上下文与聚合的粒度将不同的领域模型对象放到不同的模块与包中，对应的测试代码也应做响应的调整。如下图所示：



eaaf2f63-c29a-47c2-9609-2632be6adc1f.png

整个代码结构按照限界上下文进行划分，在限界上下文内部则按照分层架构划分。由于目前仅仅驱动出领域模型对象，上图的 payrollcontext 限界上下文中仅有 domain 层。在限界上下文中的领域层，则按照聚合的边界进行划分，跨聚合重用的领域模型对象直接放在限界上下文领域层的命名空间下。调整了代码结构后，一定要运行所有测试，避免在调整类的命名空间时，引入未知的编译错误。确定代码结构的工作越早进行越好，当要大范围调整的类数量越来越多时，调整的成本也会越来越高，影响也会越来越大。

下一章

还没有评论



评论

## 更多资源下载交流请加微信：Morstrong,加入永久会员,网盘更新更快捷！

# 本资源由微信公众号：光明顶一号，提供支持