## 【实践】培训上下文的领域实现建模

假定整个培训上下文通过领域设计建模获得了以聚合为自治单元的领域设计模型，识别了每一个领域场景，需求分析人员为每个领域场景都编写了具有验收标准（Acceptance Criteria）的用户故事，然后通过场景驱动设计给出了分解好的任务，分配了职责的时序图或时序图脚本，则开发人员在领域实现建模阶段要做的工作，就是针对每个领域场景的任务编写测试用例，然后进行测试驱动开发。

在编写领域实现代码时，不能死板地照搬领域设计建模的成果。为了快速地推进领域建模，在进行领域设计建模时，总有可能存在考虑不周之处，尤其是通过场景驱动设计获得的时序图脚本，属于伪代码的编码形式，并未完全真实地呈现最终的实现模型。此外，在领域实现建模阶段，需要以迭代的形式完成任务，加强开发人员与需求分析人员、测试人员的沟通，通过 Kick Off 与 Desk Check 等交流手段统一对需求的认识，就每个用户故事的验收标准达成一致，如此才能够保证编写的测试用例满足客户的需求。

### 测试驱动开发

#### **聚合的测试驱动开发**

从一个领域场景开始，选择一个表达领域概念和领域行为的原子任务，开始为其编写测试用例。以“提名候选人”领域场景为例，首先选择“提名”原子任务，它的测试用例包括：

* 验证票的状态必须为“Available”
* 提名给候选人后，票的状态更改为“WatiForConfirm”
* 为票生成提名历史记录

由于场景驱动设计已经识别出该任务由 Ticket 聚合履行其职责，故而创建 TicketTest 测试类。为第一个测试用例编写测试如下：

public class TicketTest {  
 private String trainingId;  
 private Candidate candidate;  
  
 @Before  
 public void setUp() {  
 trainingId = "111011111111";  
 candidate = new Candidate("200901010110", "Tom", "tom@eas.com", trainingId);  
 }  
  
 @Test  
 public void should\_throw\_TicketException\_given\_ticket\_is\_not\_AVAILABLE() {  
 Ticket ticket = new Ticket(TicketId.next(), trainingId, TicketStatus.WaitForConfirm);  
  
 assertThatThrownBy(() -> ticket.nominate(candidate))  
 .isInstanceOf(TicketException.class)  
 .hasMessageContaining("ticket is not available");  
 }  
}

遵循简单设计原则与测试驱动设计三大支柱，只需要编写让该测试通过的实现代码即可：

package xyz.zhangyi.ddd.eas.trainingcontext.domain.ticket;  
  
import xyz.zhangyi.ddd.eas.trainingcontext.domain.candidate.Candidate;  
import xyz.zhangyi.ddd.eas.trainingcontext.domain.exceptions.TicketException;  
import xyz.zhangyi.ddd.eas.trainingcontext.domain.tickethistory.TicketHistory;  
  
public class Ticket {  
 private TicketId ticketId;  
 private String trainingId;  
 private TicketStatus ticketStatus;  
  
 public Ticket(TicketId ticketId, String trainingId, TicketStatus ticketStatus) {  
 this.ticketId = ticketId;  
 this.trainingId = trainingId;  
 this.ticketStatus = ticketStatus;  
 }  
  
 public TicketHistory nominate(Candidate candidate) {  
 if (!ticketStatus.isAvailable()) {  
 throw new TicketException("ticket is not available, cannot be nominated.");  
 }  
 return null;  
 }  
}

由于当前测试并没有验证 nominate(candidate) 方法返回的结果，为了让测试快速通过，可以保留返回 null 的简单实现。

接下来为第二个测试用例编写测试方法：

public class TicketTest {  
 @Test  
 public void ticket\_status\_should\_be\_WAIT\_FOR\_CONFIRM\_after\_ticket\_was\_nominated() {  
 Ticket ticket = new Ticket(TicketId.next(), trainingId);  
  
 ticket.nominate(candidate);  
  
 assertThat(ticket.status()).isEqualTo(TicketStatus.WaitForConfirm);  
 assertThat(ticket.nomineeId()).isEqualTo(candidate.employeeId());  
 }  
}

该测试仅验证了 ticket 的状态和提名人 ID，为了保证测试通过，只需做如下实现：

public class Ticket {  
 public TicketHistory nominate(Candidate candidate) {  
 if (!ticketStatus.isAvailable()) {  
 throw new TicketException("ticket is not available, cannot be nominated.");  
 }  
  
 this.ticketStatus = TicketStatus.WaitForConfirm;  
 this.nomineeId = candidate.employeeId();  
  
 return null;  
 }  
}

在针对第三个测试用例编写测试时，就需要结合业务需求通过验证来驱动出 TicketHistory 类。首先，nominate(candidate) 方法返回了 TicketHistory 对象，为了确保返回的结果是正确的，需要验证它的属性值。究竟要验证哪些属性呢？我们可以从测试出发，确定培训票需要保存的历史记录包括：

* 票的 ID
* 票的操作类型
* 状态迁移的状况
* 执行该操作类型后的票的拥有者
* 谁执行了本次操作
* 何时执行了本次操作

体现为测试方法，即对 ticketHistory 的验证：

@Test  
 public void should\_generate\_ticket\_history\_after\_ticket\_was\_nominated() {  
 Ticket ticket = new Ticket(TicketId.next(), trainingId);  
  
 TicketHistory ticketHistory = ticket.nominate(candidate, nominator);  
  
 assertThat(ticketHistory.ticketId()).isEqualTo(ticket.id());  
 assertThat(ticketHistory.operationType()).isEqualTo(OperationType.Nomination);  
 assertThat(ticketHistory.owner()).isEqualTo(new TicketOwner(candidate.employeeId(), TicketOwnerType.Nominee));  
 assertThat(ticketHistory.stateTransit()).isEqualTo(StateTransit.from(TicketStatus.Available).to(TicketStatus.WaitForConfirm));  
 assertThat(ticketHistory.operatedBy()).isEqualTo(new Operator(nominator.employeeId(), nominator.name()));  
 assertThat(ticketHistory.operatedAt()).isEqualToIgnoringSeconds(LocalDateTime.now());  
 }

在当前领域场景中，票的操作者 operator 就是作为协调者或培训主管的提名人（Nominator）。由于之前定义的 nominate(candidate) 方法并无提名人的信息，故而需要引入 Nominator 类，修改方法接口为 nominate(candidate, nominator)。

验证 TicketHistory 的属性值，也驱动出 TicketOwner、StateTransit、OperationType 与 Operator 类，这些类皆作为 TicketHistory 聚合内的值对象，它们在领域设计建模时并没有被识别出。相反，领域设计模型为 TicketHistory 聚合定义了 CancellingReason 与 DeclineReason 类，在当前的 TicketHistory 定义中并没有给出，这是因为当前的领域场景还未牵涉到这些领域概念。TicketHistory 类的定义为：

public class TicketHistory {  
 private TicketId ticketId;  
 private TicketOwner owner;  
 private StateTransit stateTransit;  
 private OperationType operationType;  
 private Operator operatedBy;  
 private LocalDateTime operatedAt;  
  
 public TicketHistory(TicketId ticketId,  
 TicketOwner owner,  
 StateTransit stateTransit,  
 OperationType operationType,  
 Operator operatedBy,  
 LocalDateTime operatedAt) {  
 this.ticketId = ticketId;  
 this.owner = owner;  
 this.stateTransit = stateTransit;  
 this.operationType = operationType;  
 this.operatedBy = operatedBy;  
 this.operatedAt = operatedAt;  
 }  
  
 public TicketId ticketId() {  
 return this.ticketId;  
 }  
  
 public TicketOwner owner() {  
 return this.owner;  
 }  
  
 public StateTransit stateTransit() {  
 return this.stateTransit;  
 }  
  
  
 public OperationType operationType() {  
 return this.operationType;  
 }  
  
 public Operator operatedBy() {  
 return this.operatedBy;  
 }  
  
 public LocalDateTime operatedAt() {  
 return this.operatedAt;  
 }  
}

为了让当前测试快速通过，Ticket 的 nominate(candidate, nominator) 方法实现为：

public TicketHistory nominate(Candidate candidate, Nominator nominator) {  
 if (!ticketStatus.isAvailable()) {  
 throw new TicketException("ticket is not available, cannot be nominated.");  
 }  
  
 this.ticketStatus = TicketStatus.WaitForConfirm;  
 this.nomineeId = candidate.employeeId();  
  
 return new TicketHistory(ticketId,  
 new TicketOwner(candidate.employeeId(), TicketOwnerType.Nominee),  
 StateTransit.from(TicketStatus.Available).to(this.ticketStatus),  
 OperationType.Nomination,  
 new Operator(nominator.employeeId(), nominator.name()),  
 LocalDateTime.now());  
 }

考虑到 TicketOwner 的属性值来自 Candidate，Operator 的属性值来自 Nominator，可以将 Candidate 与 Nominator 分别视为它们的工厂。因而可以重构代码：

public TicketHistory nominate(Candidate candidate, Nominator nominator) {  
 if (!ticketStatus.isAvailable()) {  
 throw new TicketException("ticket is not available, cannot be nominated.");  
 }  
  
 this.ticketStatus = TicketStatus.WaitForConfirm;  
 this.nomineeId = candidate.employeeId();  
  
 return new TicketHistory(ticketId,  
 candidate.toOwner(),  
 transitState(),  
 OperationType.Nomination,  
 nominator.toOperator(),  
 LocalDateTime.now());  
 }

通过提取方法，该方法还可以进一步精简为：

public TicketHistory nominate(Candidate candidate, Nominator nominator) {  
 validateTicketStatus();  
 doNomination(candidate);  
 return generateHistory(candidate, nominator);  
 }

对比测试用例，你会发现重构后的方法包含的三行代码恰好对应这三个测试用例，清晰地展现了“提名候选人”的执行步骤。

当然，测试代码也可以进一步重构：

@Test  
 public void should\_generate\_ticket\_history\_after\_ticket\_was\_nominated() {  
 Ticket ticket = new Ticket(TicketId.next(), trainingId);  
 TicketHistory ticketHistory = ticket.nominate(candidate, nominator);  
 assertTicketHistory(ticket, ticketHistory);  
 }

#### **领域服务的测试驱动开发**

在为原子任务编写了产品代码和测试代码之后，即可在此基础上开始领域服务的测试驱动开发。领域服务对应一个组合任务，除了访问外部资源的原子任务之外，若其余原子任务都已完成编码实现，就能降低为领域服务编写单元测试的成本。与 TicketService 领域服务对应的组合任务为“提名候选人”。需要考虑的测试用例为：

* 没有符合条件的 Ticket，抛出 TicketException
* 培训票被成功提名给候选人

在考虑候选人被提名后的验收标准时，通过开发人员、需求分析人员与测试人员对需求的沟通，发现之前编写的用户故事中，忽略了两个功能：

* 添加票的历史记录
* 候选人被提名之后的处理，需要将被提名者从该培训的候选人名单中移除

故而需要调整该领域服务对应的时序图脚本：

TicketService.nominate(ticketId, candidate, nominator) {  
 TicketRepository.ticketOf(ticketId);  
 Ticket.nominate(candidate, nominator);  
 TicketRepository.update(ticket);  
 TicketHistoryRepository.add  
 CandidateRepository.remove(candidate);  
}

现在，针对测试用例编写测试方法：

public class TicketServiceTest {  
 @Test  
 public void should\_throw\_TicketException\_if\_available\_ticket\_not\_found() {  
 TicketId ticketId = TicketId.next();  
 TicketRepository mockTickRepo = mock(TicketRepository.class);  
 when(mockTickRepo.ticketOf(ticketId, Available)).thenReturn(Optional.empty());  
  
 TicketService ticketService = new TicketService();  
 ticketService.setTicketRepository(mockTickRepo);  
  
 String trainingId = "111011111111";  
 Candidate candidate = new Candidate("200901010110", "Tom", "tom@eas.com", trainingId);  
 Nominator nominator = new Nominator("200901010007", "admin", "admin@eas.com", TrainingRole.Coordinator);  
  
 assertThatThrownBy(() -> ticketService.nominate(ticketId, candidate, nominator))  
 .isInstanceOf(TicketException.class)  
 .hasMessageContaining(String.format("available ticket by id {%s} is not found", ticketId.id()));  
 verify(mockTickRepo).ticketOf(ticketId, Available);  
 }  
}

通过 Mockito的mock() 方法模拟 TicketRepository 获取 Ticket 的行为，并假定返回 Optional.empty()，以模拟未能找到培训票的场景。注意，在验证该方法时，除了要验证指定异常的抛出之外，还需要通过 Mockito 的 verify() 方法验证领域服务与资源库的协作。实现代码为：

public class TicketService {  
 private TicketRepository tickRepo;  
  
 public void setTicketRepository(TicketRepository tickRepo) {  
 this.tickRepo = tickRepo;  
 }  
  
 public void nominate(TicketId ticketId, Candidate candidate, Nominator nominator) {  
 Optional<Ticket> optionalTicket = tickRepo.ticketOf(ticketId, TicketStatus.Available);  
 if (!optionalTicket.isPresent()) {  
 throw new TicketException(String.format("available ticket by id {%s} is not found.", ticketId));  
 }  
 }  
}

驱动出来的 TicketRepository 定义为：

public interface TicketRepository {  
 Optional<Ticket> ticketOf(TicketId ticketId, TicketStatus ticketStatus);  
}

为 TicketService 编写的第二个测试需要验证提名候选人的结果。由于原子任务“提名”已经被 Ticket 的测试完全覆盖，故而在领域服务的测试中，只需要验证聚合与资源库之间的协作逻辑即可。如此既能保证代码质量和测试覆盖率，又可减少编写和维护测试的成本：

@Test  
 public void should\_nominate\_candidate\_for\_specific\_ticket() {  
 // given  
 String trainingId = "111011111111";  
 TicketId ticketId = TicketId.next();  
 Ticket ticket = new Ticket(TicketId.next(), trainingId, Available);  
  
 TicketRepository mockTickRepo = mock(TicketRepository.class);  
 when(mockTickRepo.ticketOf(ticketId, Available)).thenReturn(Optional.of(ticket));  
  
 TicketHistoryRepository mockTicketHistoryRepo = mock(TicketHistoryRepository.class);  
 CandidateRepository mockCandidateRepo = mock(CandidateRepository.class);  
  
 TicketService ticketService = new TicketService();  
 ticketService.setTicketRepository(mockTickRepo);  
 ticketService.setTicketHistoryRepository(mockTicketHistoryRepo);  
 ticketService.setCandidateRepository(mockCandidateRepo);  
  
 Candidate candidate = new Candidate("200901010110", "Tom", "tom@eas.com", trainingId);  
 Nominator nominator = new Nominator("200901010007", "admin", "admin@eas.com", TrainingRole.Coordinator);  
  
 // when  
 ticketService.nominate(ticketId, candidate, nominator);  
  
 // then  
 verify(mockTickRepo).ticketOf(ticketId, Available);  
 verify(mockTickRepo).update(ticket);  
 verify(mockTicketHistoryRepo).add(isA(TicketHistory.class));  
 verify(mockCandidateRepo).remove(candidate);  
 }

编写以上测试方法，不仅能验证 TicketService 的功能，同时还能驱动出各个资源库的接口。

与该测试对应的实现为：

public class TicketService {  
 private TicketRepository tickRepo;  
 private TicketHistoryRepository ticketHistoryRepo;  
 private CandidateRepository candidateRepo;  
  
 public void nominate(TicketId ticketId, Candidate candidate, Nominator nominator) {  
 Optional<Ticket> optionalTicket = tickRepo.ticketOf(ticketId, TicketStatus.Available);  
 Ticket ticket = optionalTicket.orElseThrow(() -> availableTicketNotFound(ticketId));  
  
 TicketHistory ticketHistory = ticket.nominate(candidate, nominator);  
  
 tickRepo.update(ticket);  
 ticketHistoryRepo.add(ticketHistory);  
 candidateRepo.remove(candidate);  
 }  
  
 private TicketException availableTicketNotFound(TicketId ticketId) {  
 return new TicketException(String.format("available ticket by id {%s} is not found.", ticketId));  
 }  
}

通过测试驱动开发进行领域实现建模是一个层层递进的过程。从领域场景分解的任务看，是从原子任务递进到组合任务；从领域模型对象的角色构造型来看，则是从聚合递进到领域服务。这样的实现既保证了各个类粒度的合理性，又能保证职责的合理分配，避免了所谓的“贫血模型”与“胀血模型”。测试驱动开发的单元测试又奠定了代码重构的基础。若在未来发生需求变更，需要改进现有设计或修改实现，就能保证开发人员进行安全的重构乃至于重写，确保了设计精进的可能性。

**领域驱动设计需要以迭代的方式进行增量开发** ，我不建议在未开始领域实现建模之前，花费大量的时间打磨领域设计模型。毕竟，一切未曾落地的设计，都可能是镜花水月。因此，我强调领域驱动设计结合敏捷迭代开发，并在敏捷管理流程的指导下合理安排项目开发。例如，在获得需求后，可以针对已有需求开展领域分析建模和领域设计建模，并在设计建模时只需要识别出领域场景即可。这时获得的领域设计模型包含了领域层最为关键的角色构造型：聚合与资源库。

一旦识别出领域场景，需求分析人员与测试人员就可以结对编写用户故事，并将这些用户故事放入到迭代计划中。开发团队在领取用户故事后，通过与需求分析人员、测试人员的 Kick Off，彻底了解其领域需求，包括用户故事的验收标准，并在确认统一语言之后，开始场景驱动设计，即分解任务，然后根据角色构造型编写时序图脚本。编写的时序图脚本以及对应的时序图可以作为领域设计模型的一部分，这个过程实际上是测试驱动开发的预研，相当于是在开发人员的心智模型中进行了业务流程与软件设计的演练。待最终确定了时序图脚本，完成了场景驱动设计，就可以开始编写测试用例，进行测试驱动开发了。

就一个用户故事而言，从场景驱动开发到测试驱动开发是一个连续的开发过程；就一个限界上下文而言，从领域分析建模到领域设计建模初期（到识别出领域场景为止），是整个特性团队参与建模的过程；识别出领域场景之后，需求分析与迭代增量开发就成了并行与串行交错的两条线，即需求分析人员在进行迭代 N+1 用户故事的分析与编写的同时，开发团队进行迭代N的场景驱动设计和测试驱动开发。

在完成领域实现建模的测试驱动开发之后，针对一个领域场景而言，只有完成了应用层和基础设施层的实现编码，才算真正完成整个用户故事。这就需要定义远程服务和应用服务，并完成基础设施层北向网关与南向网关的实现，即领域驱动设计魔方中，纳米层次技术维度要完成的框架应用开发与基础设施代码。它们的设计与开发并不属于领域实现建模的范畴，而应站在系统架构的角度，在分层架构、上下文映射以及前后端分离的背景之下，定义和实现系统的代码模型。

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