По мере прохождения кода через процесс развертывания наборы тестов все более тщательно тестируют его в среде, приближенной к промышленной. Одновременно время выполнения каждого тестового набора обычно увеличивается. Основной смысл процедуры состоит в том, чтобы как можно скорее сообщить о непройденных тестах.

Процесс развертывания (см. рис. 9.9) состоит из следующих этапов.

- *Этап, предшествующий фиксации*, — выполняет модульные тесты. Запускается разработчиком перед фиксацией изменений.
- *Этап фиксации* — компилирует сервис, выполняет модульные тесты и производит статический анализ кода.
- *Интеграционный этап* — выполняет интеграционные тесты.
- *Компонентный этап* — выполняет компонентные тесты для сервиса.
- *Этап развертывания* — развертывает сервис в промышленной среде.

CI-сервер запускает этап фиксации, когда разработчик фиксирует изменение. Он выполняется чрезвычайно быстро, чтобы сразу предоставить сведения о фиксации. Дальнейшие этапы протекают дольше и предоставляют информацию не так быстро. Если все тесты пройдены, на заключительном этапе код развертывается в промышленную среду.

В этом примере автоматизирован весь процесс, от фиксации до развертывания. Однако в некоторых ситуациях требуется вмешательство человека. Например, вам может понадобиться этап ручного тестирования в предпромышленной среде. В этом сценарии код переходит на следующий этап, когда тестировщик отмечает успешное тестирование нажатием кнопки. Это может быть также выпуск новой версии сервиса в рамках процесса развертывания. Позже выпущенные сервисы будут упакованы и в качестве готового продукта отправлены заказчикам.

Теперь вы знаете, как организован процесс развертывания и на каких этапах выполняются разные типы тестов. Переместимся в самый низ пирамиды тестирования и посмотрим, как пишут модульные тесты для сервиса.

9.2. Написание модульных тестов для сервиса

Представьте, что вам нужно написать тест, который проверяет, вычисляет ли сервис `Order` приложения FTGO корректную промежуточную стоимость заказа. Код вашего теста может запустить сервис `Order`, обратиться к его REST API, чтобы создать заказ, и проверить, содержит ли HTTP-ответ ожидаемые значения. Однако при этом тест получится не только сложным, но и медленным. Если он выполняется на этапе компиляции класса `Order`, вы будете тратить много времени в ожидании его завершения. Написание модульных тестов для класса `Order` — куда более продуктивный подход.

Как видно на рис. 9.10, модульные тесты находятся на самом нижнем уровне пирамиды тестирования. Они ориентированы на технологии и могут использоваться в разработке. Модульный тест позволяет убедиться в корректной работе *модуля*, который представляет собой очень маленькую часть сервиса. Обычно в качестве модуля выступает класс, поэтому модульное тестирование проверяет, ведет ли он себя так, как от него ожидается.

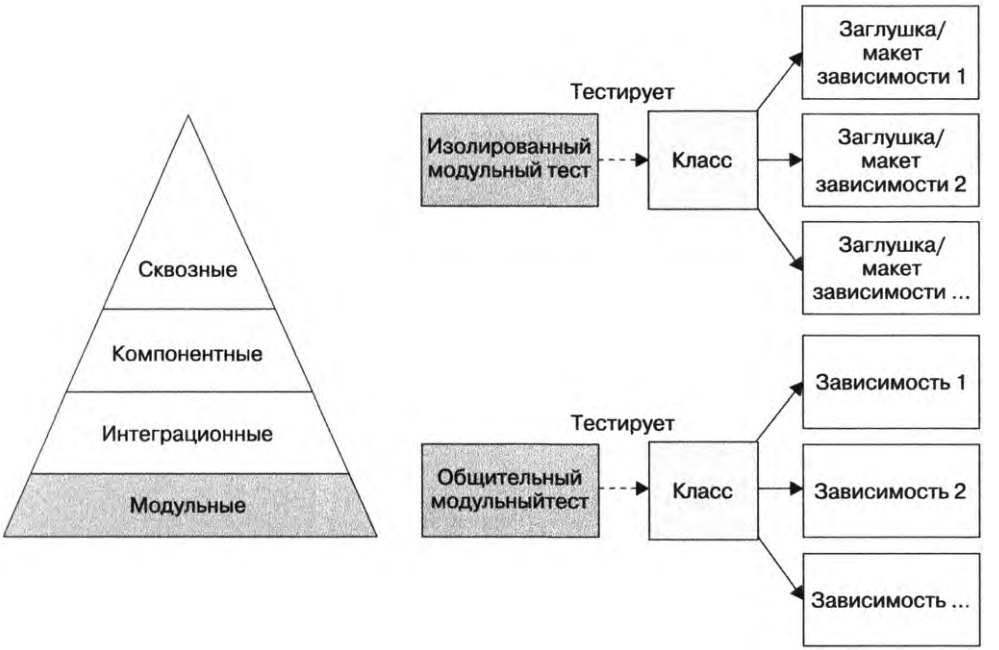

Рис. 9.10. Модульные тесты лежат в основании пирамиды. Они быстрые, простые в написании и надежные. Изолированный модульный тест тестирует отдельно взятый класс, задействуя макеты и заглушки вместо его зависимостей. Общительный модульный тест тестирует класс вместе с его зависимостями

Существует два типа модульных тестов (martinfowler.com/bliki/UnitTest.html):

- *изолированный* — тестирует отдельно взятый класс, заменяя его зависимости объектами-макетами;
- *общительный* — тестирует класс и его зависимости.

Тип теста, который следует использовать, зависит от назначения класса и его роли в архитектуре. Шестигранная архитектура типичного сервиса и типы модульных тестов, применяемые для определенного вида классов, показаны на рис. 9.11. Классы контроллеров и сервиса обычно тестируются изолированно. Доменные объекты, такие как сущности и объекты значений, чаще всего тестируются с помощью общительных модульных тестов.

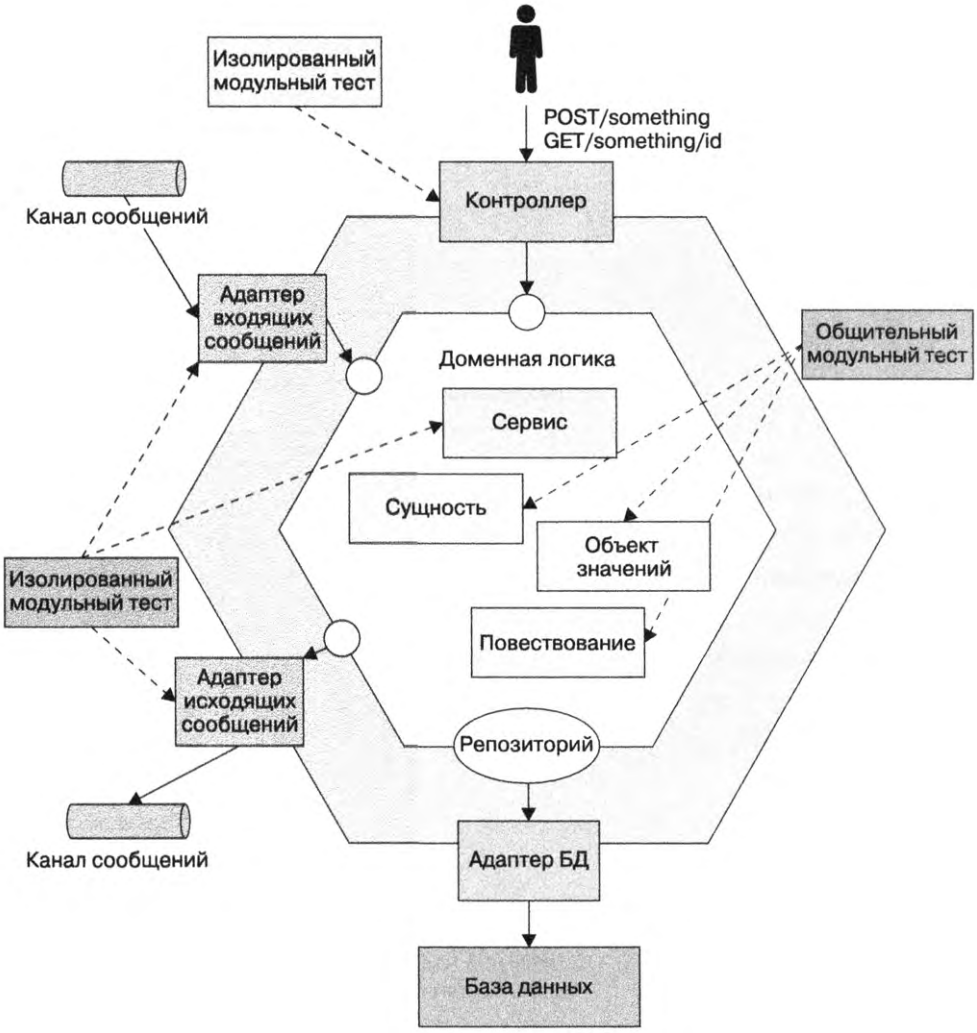

Рис. 9.11. Назначение класса определяет, какие модульные тесты нужно использовать — изолированные или общительные

Типичная стратегия тестирования класса выглядит так.

- Такие сущности, как `Order`, которые обладают постоянными идентификаторами (см. главу 5), тестируются с помощью общительных модульных тестов.
- Такие объекты, как `Money`, представляющие собой набор значений (см главу 5), тестируются с применением общительных модульных тестов.
- Повествования наподобие `CreateOrderSaga`, которые обеспечивают согласованность данных между сервисами (см. главу 4), тестируются общительными модульными тестами.

- Доменные сервисы, такие как `OrderService`, реализующие бизнес-логику, которая не подходит для сущностей или объектов значений (см. главу 5), тестируются с помощью изолированных модульных тестов.
- Контроллеры, обрабатывающие HTTP-запросы, такие как `OrderController`, тестируются с использованием изолированных модульных тестов.
- Шлюзы для входящих и исходящих сообщений тестируются с помощью изолированных модульных тестов.

Для начала посмотрим, как тестируются доменные сущности.

9.2.1. Разработка модульных тестов для доменных сущностей

В листинге 9.2 показан фрагмент класса `OrderTest`, реализующий модульные тесты для сущности `Order`. Этот класс содержит метод `@Before setUp()`, который создает объект `Order` перед запуском каждого теста. Методы, помеченные как `@Test`, могут выполнить инициализацию объекта `Order`, вызвать один из его методов и затем сделать утверждение о его возвращаемом значении и состоянии.

Листинг 9.2. Простой и быстрый модульный тест для сущности Order

```
public class OrderTest {

  private ResultWithEvents<Order> createResult;
  private Order order;

  @Before
  public void setUp() throws Exception {
    createResult = Order.createOrder(CONSUMER_ID, AJANTA_ID, CHICKEN_VINDALOO
     _LINE_ITEMS);
    order = createResult.result;
  }

  @Test
  public void shouldCalculateTotal() {
    assertEquals(CHICKEN_VINDALOO_PRICE.multiply(CHICKEN_VINDALOO_QUANTITY),
     order.getOrderTotal());
  }

  ...

}
```

Метод `@Test shouldCalculateTotal()` проверяет, возвращает ли `Order.getOrderTotal()` ожидаемое значение. Модульные тесты тщательно тестируют бизнес-логику. Они являются общительными и распространяются не только на класс `Order`, но и на его зависимости. Вы можете использовать их на этапе компиляции, так как выполняются они чрезвычайно быстро. Также важно протестировать объект значений `Money`, от которого зависит класс `Order`. Посмотрим, как это делается.

9.2.2. Написание модульных тестов для объектов значений

Объекты значений не изменяются, поэтому их обычно легко тестировать. Вам не нужно беспокоиться о побочных эффектах. Как правило, тест должен создать объект значений в определенном состоянии, вызвать один из его методов и сделать вывод о возвращаемом значении. В листинге 9.3 приводятся тесты для объекта `Money` — простого класса, который представляет денежное значение. Эти тесты проверяют поведение методов класса `Money`, включая `add()`, который складывает два экземпляра `Money`, и `multiply()`, который умножает объект `Money` на целое число. Эти тесты изолированы, поскольку класс `Money` не зависит ни от каких других классов приложения.

Листинг 9.3. Простой и быстрый тест для объекта значений Money

```
public class MoneyTest {

  private final int M1_AMOUNT = 10;
  private final int M2_AMOUNT = 15;

  private Money m1 = new Money(M1_AMOUNT);
  private Money m2 = new Money(M2_AMOUNT);

  @Test
  public void shouldAdd() {
     assertEquals(new Money(M1_AMOUNT + M2_AMOUNT), m1.add(m2));
  }

  @Test
  public void shouldMultiply() {
    int multiplier = 12;
    assertEquals(new Money(M2_AMOUNT * multiplier), m2.multiply(multiplier));
  }

  ...
}
```

Проверяем, можно ли сложить вместе два объекта Money

Проверяем, можно ли умножить объект Money на целое число

Доменные сущности и объекты значений — это кирпичики, из которых состоит бизнес-логика сервиса. Но иногда она может находиться также в повествованиях и других сервисах приложения. Посмотрим, как их тестировать.

9.2.3. Разработка модульных тестов для повествований

Повествования, такие как класс `CreateOrderSaga`, реализуют важную бизнес-логику, поэтому их тоже нужно тестировать. В данном случае мы имеем дело с сохраняемым объектом, которые рассылает командные сообщения участникам повествования и обрабатывает их ответы. Как говорилось в главе 4, класс `CreateOrderSaga` обменивается командными/ответными сообщениями с несколькими сервисами, включая `Consumer`

и Kitchen. Тест для этого класса создает повествование и проверяет, шлет ли тот ожидаемую последовательность сообщений своим участникам. Один из тестов должен быть написан для оптимистичного сценария. Но вы должны предусмотреть и тесты для различных случаев, когда повествование откатывается, получив сообщение об отказе от одного из своих участников.

Вы можете написать тесты, которые используют настоящие базу данных и брокер сообщений, но заменяют участников повествования заглушками. Например, заглушка для сервиса Consumer будет подписываться на командный канал consumerService и отправлять обратно сообщение с нужным ответом. Однако тесты, написанные таким образом, будут довольно медленными. Куда более эффективный подход заключается в применении макетов место брокера сообщений и классов для взаимодействия с базой данных. Это позволит вам сосредоточиться на тестировании основных функций повествования.

В листинге 9.4 показан тест для CreateOrderSaga. Это общительный модульный тест, который проверяет класс повествования и его зависимости. Он написан с использованием фреймворка тестирования Eventuate Tram Saga (github.com/eventuate-tram/eventuate-tram-sagas), который предоставляет простой в применении язык DSL, абстрагирующий подробности взаимодействия с повествованиями. С помощью этого языка вы можете создать повествование и убедиться в том, что оно отправляет корректные командные сообщения. При этом фреймворк тестирования подготавливает макеты для базы данных и инфраструктуры обмена сообщениями.

Листинг 9.4. Простой и быстрый модульный тест для CreateOrderSaga

```
public class CreateOrderSagaTest {

  @Test
  public void shouldCreateOrder() {
    given()
        .saga(new CreateOrderSaga(kitchenServiceProxy),
                new CreateOrderSagaState(ORDER_ID,
                     CHICKEN_VINDALOO_ORDER_DETAILS)).
    expect().
         command(new ValidateOrderByConsumer(CONSUMER_ID, ORDER_ID,
                CHICKEN_VINDALOO_ORDER_TOTAL)).
        to(ConsumerServiceChannels.consumerServiceChannel).
    andGiven().
        successReply().
     expect().
          command(new CreateTicket(AJANTA_ID, ORDER_ID, null)).
           to(KitchenServiceChannels.kitchenServiceChannel);
  }

  @Test
  public void shouldRejectOrderDueToConsumerVerificationFailed() {
    given()
        .saga(new CreateOrderSaga(kitchenServiceProxy),
                new CreateOrderSagaState(ORDER_ID,
                     CHICKEN_VINDALOO_ORDER_DETAILS)).
```

Создаем повествование

Проверяем, шлет ли оно сообщение ValidateOrderByConsumer сервису Consumer

Успешно отвечаем на это сообщение

Проверяем, шлет ли оно сообщение CreateTicket сервису Kitchen

```
    expect().
        command(new ValidateOrderByConsumer(CONSUMER_ID, ORDER_ID,
                CHICKEN_VINDALOO_ORDER_TOTAL)).
        to(ConsumerServiceChannels.consumerServiceChannel).
    andGiven().
       failureReply().                          <-- Возвращаем отрицательный ответ:
                                                    сервис Consumer отклонил заказ
    expect().
       command(new RejectOrderCommand(ORDER_ID)).
       to(OrderServiceChannels.orderServiceChannel);   <-- Проверяем, шлет ли повествование
  }                                                        сообщение RejectOrderCommand сервису Order
}
```

Метод `@Test shouldCreateOrder()` тестирует оптимистичный сценарий. Метод `@Test shouldRejectOrderDueToConsumerVerificationFailed()` тестирует случай, когда сервис `Consumer` отклоняет заказ. Он проверяет, шлет ли `CreateOrderSaga` команду `RejectOrderCommand`, чтобы компенсировать отклонение действий клиента. Класс `CreateOrderSagaTest` содержит методы для тестирования и других отрицательных сценариев.

Теперь посмотрим, как тестировать доменные сервисы.

9.2.4. Написание модульных тестов для доменных сервисов

Бóльшая часть бизнес-логики сервиса реализуется его доменными сущностями, объектами значений и повествованиями. Остальное содержится в таких классах, как `OrderService`. Это типичный класс доменного сервиса. Его методы вызывают сущности и репозитории и публикуют доменные события. Чтобы его эффективно протестировать, нужно использовать в основном изолированные модульные тесты, которые предоставляют макеты для таких зависимостей, как репозитории и классы обмена сообщениями.

В листинге 9.5 представлен класс `OrderServiceTest`, который тестирует `OrderService`. Он определяет изолированные модульные тесты, заменяющие зависимости сервиса макетами из состава Mockito. Каждый тест состоит из следующих этапов.

1. *Подготовка* — конфигурирует объекты-макеты для зависимостей сервиса.
2. *Выполнение* — вызывает метод сервиса.
3. *Проверка* — проверяет корректность значения, возвращенного методом сервиса, и убеждается в том, что зависимости были вызваны правильно.

Метод `setUp()` создает экземпляр `OrderService` с внедренными макетами зависимостей. Метод `@Test shouldCreateOrder()` проверяет, обратился ли вызов `OrderService.createOrder()` к `OrderRepository`, чтобы сохранить только что созданный заказ, опубликовал ли он событие `OrderCreatedEvent` и создал ли `CreateOrderSaga`.

Листинг 9.5. Простой и быстрый модульный тест для класса OrderService

```
public class OrderServiceTest {

  private OrderService orderService;
  private OrderRepository orderRepository;
  private DomainEventPublisher eventPublisher;
  private RestaurantRepository restaurantRepository;
  private SagaManager<CreateOrderSagaState> createOrderSagaManager;
  private SagaManager<CancelOrderSagaData> cancelOrderSagaManager;
  private SagaManager<ReviseOrderSagaData> reviseOrderSagaManager;

  @Before
  public void setup() {
    orderRepository = mock(OrderRepository.class);
     eventPublisher = mock(DomainEventPublisher.class);
    restaurantRepository = mock(RestaurantRepository.class);
    createOrderSagaManager = mock(SagaManager.class);
    cancelOrderSagaManager = mock(SagaManager.class);
    reviseOrderSagaManager = mock(SagaManager.class);
    orderService = new OrderService(orderRepository, eventPublisher,
             restaurantRepository, createOrderSagaManager,
            cancelOrderSagaManager, reviseOrderSagaManager);
  }

  @Test
  public void shouldCreateOrder() {
    when(restaurantRepository
       .findById(AJANTA_ID)).thenReturn(Optional.of(AJANTA_RESTAURANT_);
    when(orderRepository.save(any(Order.class))).then(invocation -> {
       Order order = (Order) invocation.getArguments()[0];
      order.setId(ORDER_ID);
      return order;
    });

    Order order = orderService.createOrder(CONSUMER_ID,
                     AJANTA_ID, CHICKEN_VINDALOO_MENU_ITEMS_AND_QUANTITIES);

    verify(orderRepository).save(same(order));

    verify(eventPublisher).publish(Order.class, ORDER_ID,
           singletonList(
             new OrderCreatedEvent(CHICKEN_VINDALOO_ORDER_DETAILS)));

    verify(createOrderSagaManager)
           .create(new CreateOrderSagaState(ORDER_ID,
                      CHICKEN_VINDALOO_ORDER_DETAILS),
                 Order.class, ORDER_ID);
  }

}
```

Создаем макеты Mockito для зависимостей класса OrderService

Создаем экземпляр OrderService с внедренными макетами зависимостей

Делаем так, чтобы метод RestaurantRepository.findById() вернул ресторан Ajanta

Делаем так, чтобы метод OrderRepository.save() сохранил ID заказа

Вызываем OrderService.create()

Проверяем, сохранил ли класс OrderService только что созданный заказ в БД

Проверяем, опубликовал ли класс OrderService событие OrderCreatedEvent

Проверяем, создал ли класс OrderCreatedEvent повествование CreateOrderSaga

Итак, мы обсудили то, как выполнить модульное тестирование классов бизнес-логики. Теперь поговорим о том, как сделать то же самое с адаптерами, которые взаимодействуют с внешними системами.

9.2.5. Разработка модульных тестов для контроллеров

Сервисы, такие как `Order`, обычно содержат один или несколько контроллеров для обработки HTTP-запросов от других сервисов и API-шлюза. Класс контроллера состоит из набора методов, обрабатывающих запросы. Каждый такой метод реализует конечную точку REST API, а его параметры представляют значения HTTP-запроса, такие как переменные пути. Обычно он обращается к доменному сервису или репозиторию и возвращает объект с ответом. `OrderController`, к примеру, обращается к `OrderService` и `OrderRepository`. Эффективная стратегия тестирования контроллеров предполагает использование изолированных модульных тестов, которые заменяют макетами сервисы и репозитории.

Вы могли бы написать класс вроде `OrderServiceTest`, который создает экземпляр контроллера и вызывает его методы. Однако такой подход не позволяет проверить некоторые важные возможности, такие как маршрутизация запросов. Куда более эффективным будет применение фреймворков для тестирования в стиле MVC, например Spring Mock Mvc, который входит в состав Spring Framework, или Rest Assured Mock MVC, основанный на Spring Mock Mvc. Тесты, написанные с помощью одной из этих технологий, выполняют нечто похожее на HTTP-запрос и делают заключение об HTTP-ответах. Эти фреймворки позволяют тестировать маршрутизацию HTTP-запросов и преобразование объектов Java в формат JSON и обратно, избегая при этом реальных сетевых вызовов. Spring Mock Mvc автоматически создает экземпляры ровно того количества классов Spring MVC, которого должно хватить для тестирования.

Действительно ли это модульные тесты?

Эти тесты используют Spring Framework, поэтому можно предположить, что они не модульные. Действительно, они тяжеловесней тех, что я описывал ранее. В документации Spring Mock Mvc они называются внеконтейнерными интеграционными тестами (docs.spring.io/spring/docs/current/spring-framework-reference/testing.html#spring-mvc-test-vs-end-to-end-integration-tests). В то же время в Rest Assured Mock MVC они считаются модульными (github.com/rest-assured/rest-assured/wiki/Usage#spring-mock-mvc-module). Несмотря на такое различие в терминологии, написание этих тестов очень важно.

В листинге 9.6 показан класс `OrderControllerTest`, тестирующий `OrderController` сервиса `Order`. Он определяет изолированные модульные тесты, использующие

макеты вместо зависимостей `OrderController`. Этот класс написан с помощью фреймворка Rest Assured Mock MVC, который предоставляет простой язык DSL, абстрагирующий подробности взаимодействия с контроллерами. Rest Assured упрощает отправку контроллеру поддельных HTTP-запросов и проверку ответов. `OrderControllerTest` создает контроллер с внедренными макетами Mockito для `OrderService` и `OrderRepository`. Каждый тест конфигурирует макеты, выполняет HTTP-запрос, проверяет корректность ответа и, возможно, следит за тем, чтобы контроллер обратился к этим макетам.

Листинг 9.6. Простой и быстрый модульный тест для класса OrderController

```
public class OrderControllerTest {

  private OrderService orderService;
  private OrderRepository orderRepository;

  @Before
  public void setUp() throws Exception {
    orderService = mock(OrderService.class);
    orderRepository = mock(OrderRepository.class);
    orderController = new OrderController(orderService, orderRepository);
  }

  @Test
  public void shouldFindOrder() {

    when(orderRepository.findById(1L))
        .thenReturn(Optional.of(CHICKEN_VINDALOO_ORDER_));

    given().
      standaloneSetup(configureControllers(
          new OrderController(orderService, orderRepository))).
    when().
        get("/orders/1").
    then().
      statusCode(200).
      body("orderId",
          equalTo(new Long(OrderDetailsMother.ORDER_ID).intValue())).
      body("state",
          equalTo(OrderDetailsMother.CHICKEN_VINDALOO_ORDER_STATE.name())).
      body("orderTotal",
          equalTo(CHICKEN_VINDALOO_ORDER_TOTAL.asString()))
    ;
  }

  @Test
  public void shouldFindNotOrder() { ... }

  private StandaloneMockMvcBuilder controllers(Object... controllers) { ... }

}
```

Создаем макеты зависимостей класса OrderController

Делаем так, чтобы макет OrderRepository возвращал заказ

Конфигурируем OrderController

Выполняем HTTP-запрос

Проверяем код состояния ответа

Проверяем элементы тела ответа в формате JSON

Первым делом тестовый метод `shouldFindOrder()` конфигурирует макет `OrderRepository` так, чтобы тот возвращал `Order`. Затем он выполняет HTTP-запрос, чтобы извлечь заказ. В конце проверяет успешность запроса и то, содержит ли тело ответа ожидаемые данные.

Контроллеры — это не единственный вид адаптеров, обрабатывающих запросы из внешних систем. Есть также обработчики событий/сообщений. Посмотрим, как выполняется модульное тестирование для них.

9.2.6. Написание модульных тестов для обработчиков событий и сообщений

Сервисы часто обрабатывают сообщения, переданные внешними системами. К примеру, сервис `Order` содержит адаптер сообщений `OrderEventConsumer`, обрабатывающий доменные события, публикуемые другими сервисами. Как и контроллеры, адаптеры сообщений обычно представляют собой простые классы, которые обращаются к доменным сервисам. Каждый метод адаптера, как правило, вызывает метод сервиса, передавая ему данные из сообщения или события.

Для модульного тестирования адаптеров сообщений можно применить подход, аналогичный используемому для контроллеров. Каждый тест создает экземпляр адаптера, отправляет сообщение в канал и проверяет корректность обращения к макету сервиса. Одновременно автоматически создаются заглушки для инфраструктуры обмена сообщениями, поэтому брокер не участвует в этом процессе. Посмотрим, как протестировать класс `OrderEventConsumer`.

В листинге 9.7 показан фрагмент класса `OrderEventConsumerTest`, который тестирует адаптер `OrderEventConsumer`. Он проверяет, направляет ли тот каждое событие подходящему методу-обработчику, и следит за корректностью обращения к `OrderService`. Здесь применяется фреймворк Eventuate Tram Mock Messaging, который предоставляет простой в использовании язык DSL для создания макетов инфраструктуры обмена сообщениями. Он имеет тот же формат тестов «дано — когда — тогда», что и Rest Assured. Каждый тест создает экземпляр `OrderEventConsumer` с внедренным макетом `OrderService`, публикует доменное событие и проверяет, корректно ли `OrderEventConsumer` обращается к макету сервиса.

Метод `setUp()` создает экземпляр `OrderEventConsumer` с внедренным макетом `OrderService`. Метод `shouldCreateMenu()` публикует событие `RestaurantCreated` и проверяет, вызвал ли объект `OrderEventConsumer` метод `OrderService.createMenu()`. `OrderEventConsumerTest`, как и другие классы для модульного тестирования, выполняется очень быстро. Модульные тесты работают всего несколько секунд.

Однако эти тесты не проверяют, насколько корректно сервис наподобие `Order` взаимодействует с другими сервисами. Например, они не позволяют убедиться в том, что заказ можно сохранить в MySQL или что `CreateOrderSaga` шлет командные сообщения в правильном формате и в нужный канал. И с их помощью нельзя узнать, имеет ли событие `RestaurantCreated`, обработанное адаптером `OrderEventConsumer`, ту же структуру, что и события, публикуемые сервисом `Restaurant`. Для тестирования корректности взаимодействия между сервисами необходимо писать интеграци-

онные тесты. Также понадобятся компонентные тесты, которые тестируют отдельно взятый сервис целиком. Все это, а также выполнение сквозного тестирования, мы обсудим в следующей главе.

Листинг 9.7. Быстрый модульный тест для класса OrderEventConsumer

```
public class OrderEventConsumerTest {

  private OrderService orderService;
  private OrderEventConsumer orderEventConsumer;

  @Before                                                    Создаем экземпляр OrderEventConsumer
  public void setUp() throws Exception {                     с поддельными зависимостями
    orderService = mock(OrderService.class);
    orderEventConsumer = new OrderEventConsumer(orderService);
  }

  @Test
  public void shouldCreateMenu() {                           Конфигурируем доменные
                                                             обработчики OrderEventConsumer
    given().
            eventHandlers(orderEventConsumer.domainEventHandlers()).
    when().
      aggregate("net.chrisrichardson.ftgo.restaurantservice.domain.Restaurant",
                AJANTA_ID).
      publishes(new RestaurantCreated(AJANTA_RESTAURANT_NAME,
                          RestaurantMother.AJANTA_RESTAURANT_MENU))
    then().
       verify(() -> {
          verify(orderService)
                .createMenu(AJANTA_ID,
            new RestaurantMenu(RestaurantMother.AJANTA_RESTAURANT_MENU_ITEMS));
       })
    ;
  }

}
```

Публикуем событие RestaurantCreated

Проверяем, вызвал ли экземпляр OrderEventConsumer метод OrderService.createMenu()

Резюме

- Автоматическое тестирование лежит в основе быстрой и безопасной доставки программного обеспечения. К тому же микросервисная архитектура сложна по своей природе, поэтому, чтобы воспользоваться всеми ее преимуществами, вы *должны* автоматизировать свои тесты.
- Тест нужен для того, чтобы проверить поведение тестируемой системы. В данном случае под *системой* понимается тестируемый элемент программного обеспечения. Это может быть как отдельный класс, так и приложение целиком. Или же что-то среднее, например коллекция классов или отдельный сервис. Связанные между собой тесты объединяются в тестовый набор.

- Хороший способ упрощения и ускорения тестов — задействование дублеров. Дублер — это объект, который симулирует поведение зависимости тестируемой системы. Существует два типа дублеров: заглушки и макеты. Заглушка возвращает значение тестируемой системе. Макет используется тестом для проверки, корректно ли система вызывает свои зависимости.
- Применяйте пирамиду тестов, чтобы определить, где следует приложить усилия при тестировании сервисов. Большинство ваших тестов должны быть модульными, то есть быстрыми, надежными и простыми в написании. Вы должны минимизировать количество сквозных тестов, поскольку они медленные и нестабильные, а их написание занимает много времени.