INSTITUTO FEDERAL
Paraíba

Campus
Cajazeiras

PROGRAMAÇÃO P/ WEB 2

12. TESTANDO MICROSSERVIÇOS

PROF. DIEGO PESSOA

✉ DIEGO.PESSOA@IFPB.EDU.BR

@DIEGOEP

CST em Análise e Desenvolvimento de Sistemas

ROTEIRO

- Estratégias de testes para microsserviços
- Usando mocks e stubs para testar um componente isolado
- Usando o conceito de pirâmide de testes para determinar onde focar os esforços de teste
- Testes unitários para classes dentro de um serviço

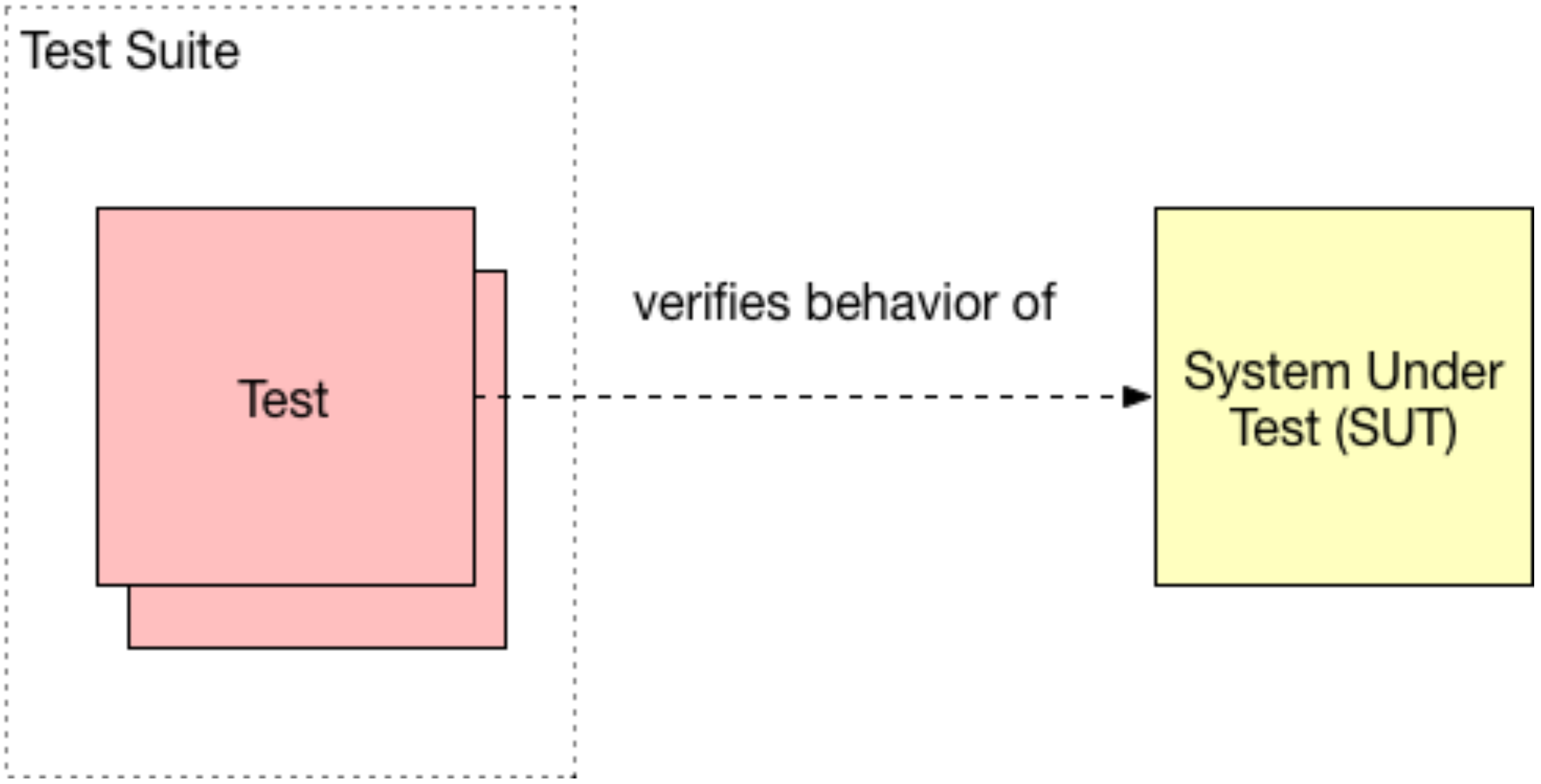
MOTIVAÇÕES

- "You should never ask a human to do what a machine can do better."
- "The second problem with the traditional approach to testing is that testing is done far too late in the delivery process."
- "A much better approach is for developers to write automated tests as part of development. It improves their productivity since, for example, they'll have tests that provide immediate feedback while editing code. "

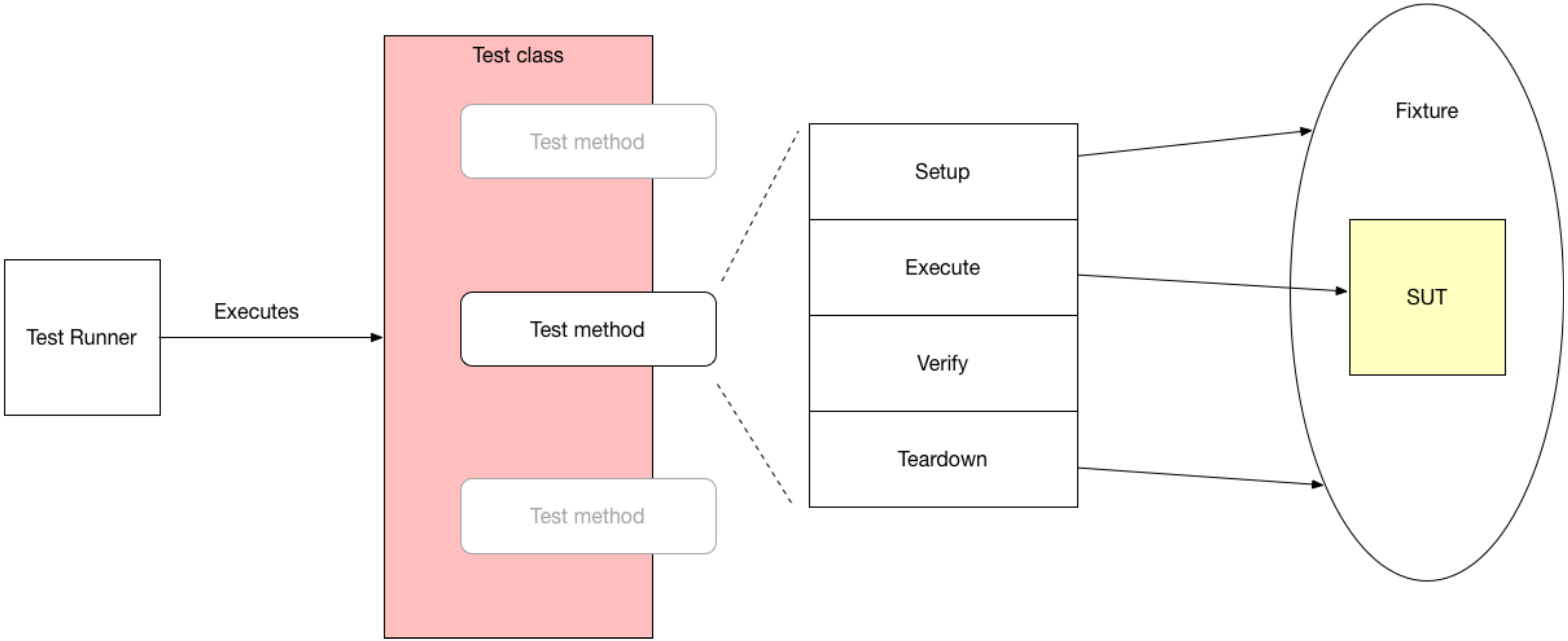
RELEMBRANDO… VISÃO GERAL DE TESTES

- A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.
- https://en.wikipedia.org/wiki/Test_case

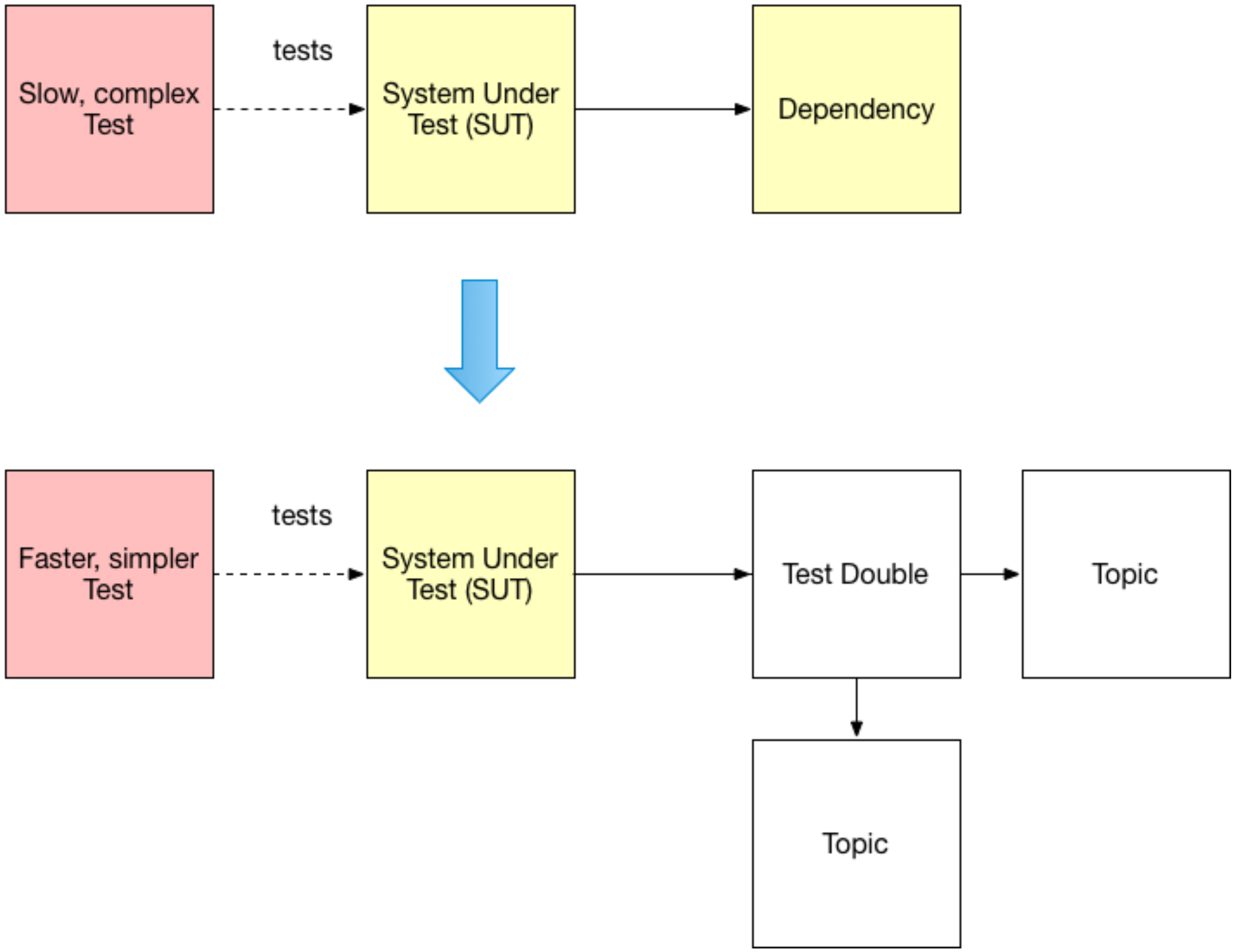
RELEMBRANDO… VISÃO GERAL DE TESTES



ESCREVENDO TESTES AUTOMÁTICOS



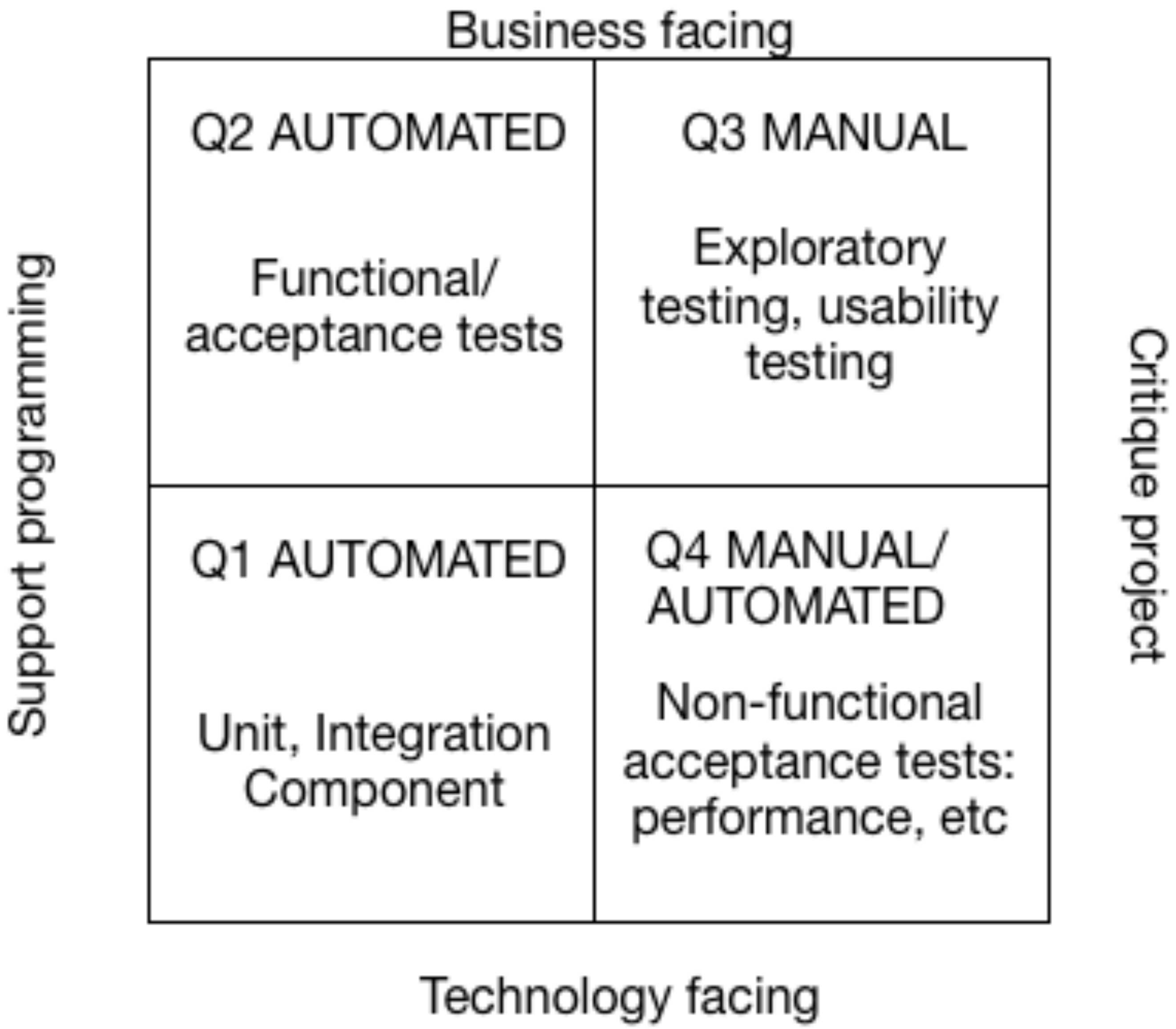
TESTANDO USANDO MOCKS E STUBS



DIFERENTES TIPOS DE TESTE

- Unit tests - tests a small part of an service, such as a class
- Integration tests - verifies that a service can interact with infrastructure services, such as databases, and other application services
- Component tests - acceptance tests for an individual service
- end-to-end tests - acceptance tests for the entire application

USANDO O “TEST QUADRANT” PARA CATEGORIZAR TESTES



Business facing

Q2 AUTOMATED Functional/ acceptance tests	Q3 MANUAL Exploratory testing, usability testing
Q1 AUTOMATED Unit, Integration Component	Q4 MANUAL/ AUTOMATED Non-functional acceptance tests: performance, etc

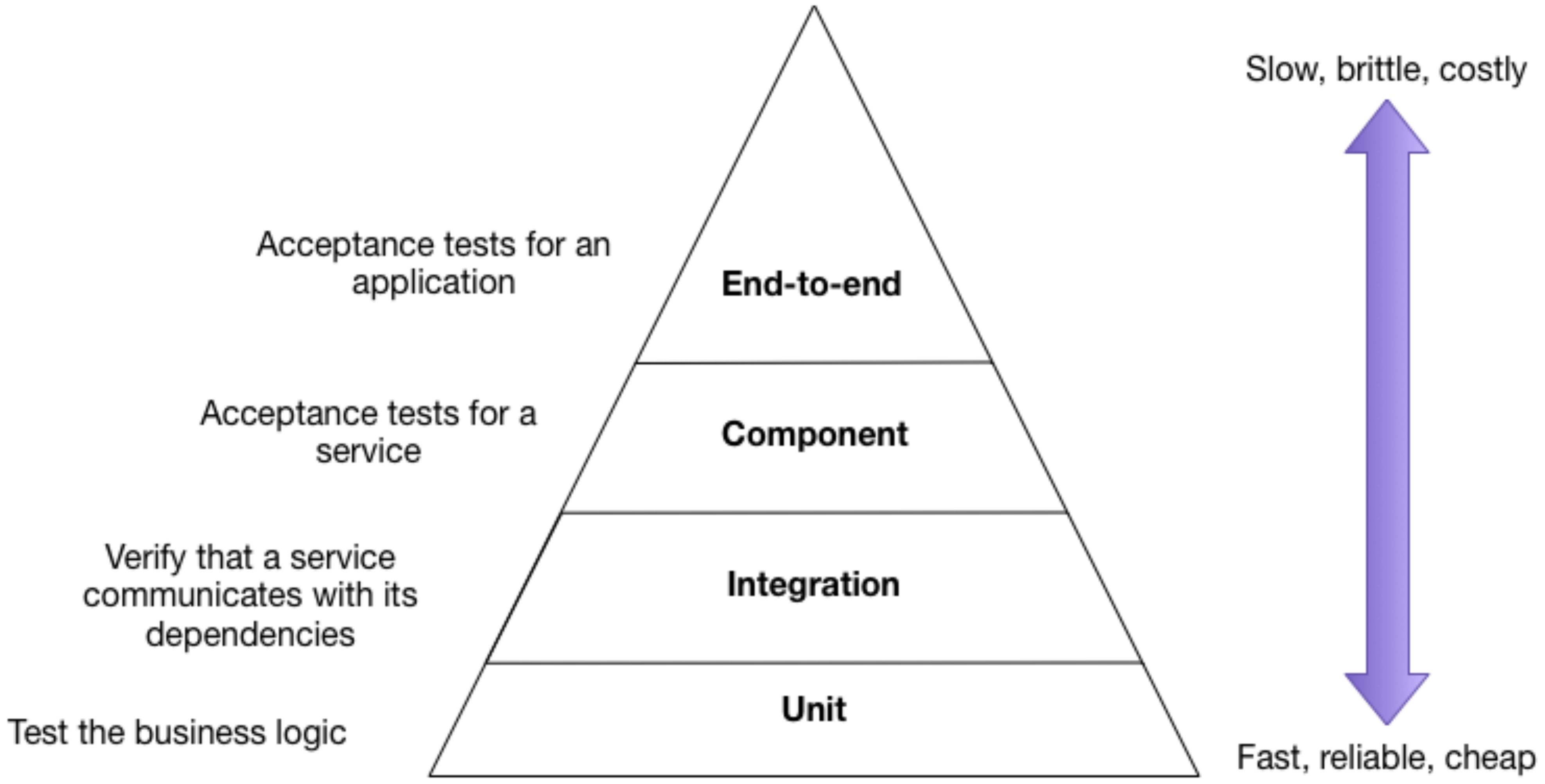
Technology facing

Support programming

Critique project

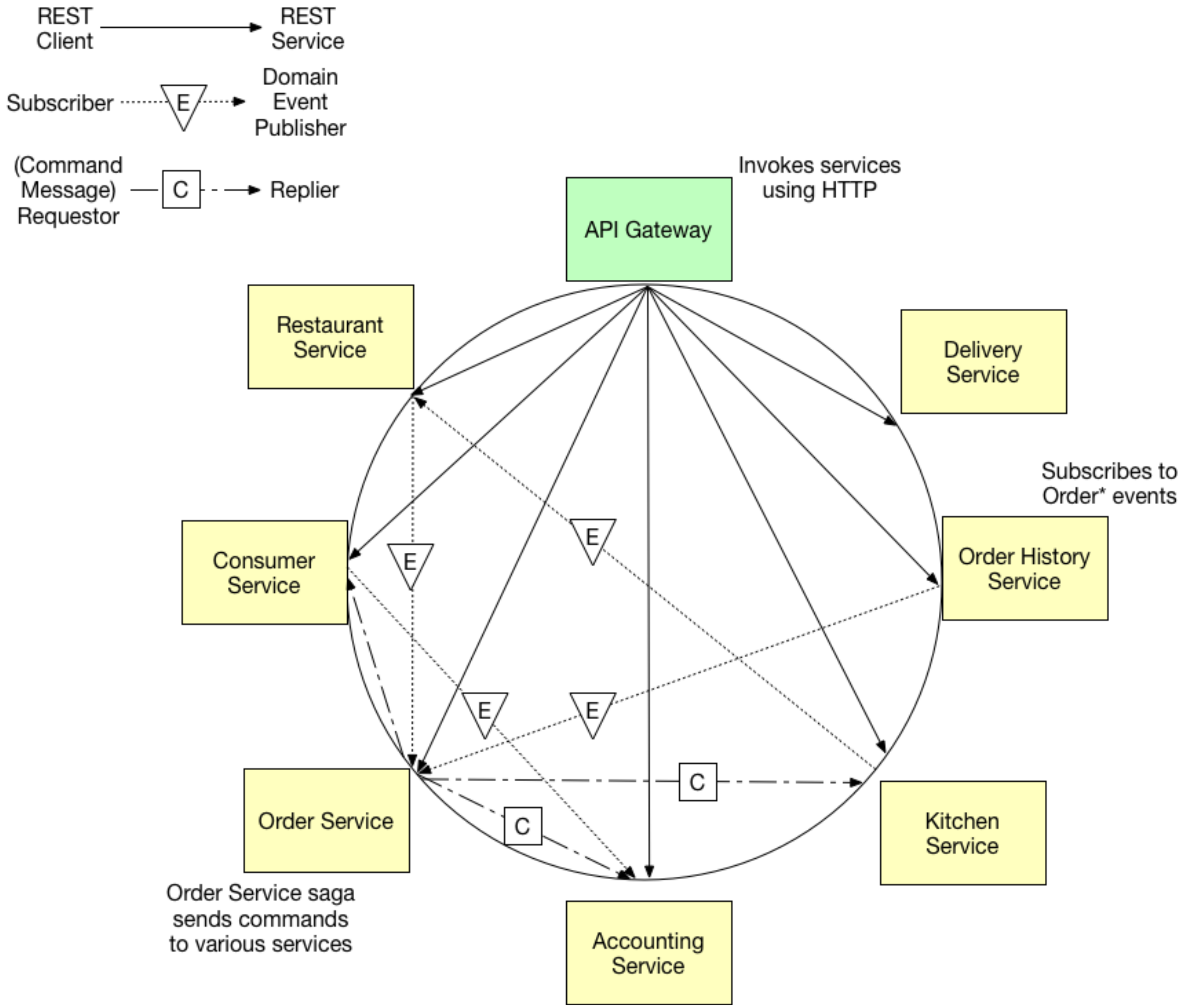
- Q1 - support programming/technology facing - unit and integration tests
- Q2 - Support programming/business facing - component and end-to-end test
- Q3 - Critique application/business facing - usability and exploratory testing
- Q4 - Critique application/technology facing - non-functional acceptance tests, such as performance tests

USANDO A “TEST PYRAMID” COMO GUIA PARA SABER ONDE FOCAR ESFORÇOS

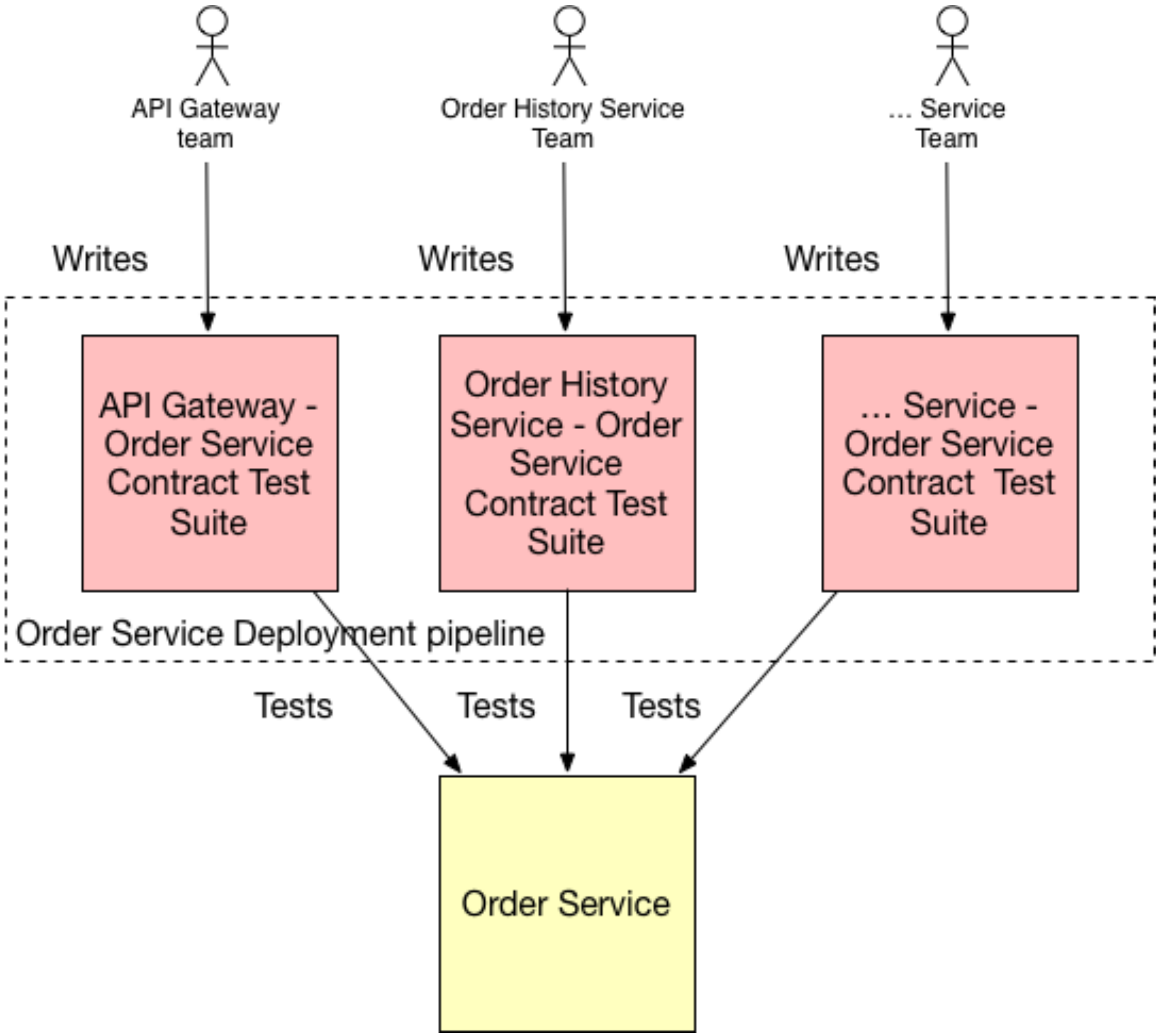

Slow, brittle, costly
Acceptance tests for an application
End-to-end
Acceptance tests for a service
Component
Verify that a service communicates with its dependencies
Integration
Test the business logic
Unit
Fast, reliable, cheap

TESTES E MICROSSERVIÇOS

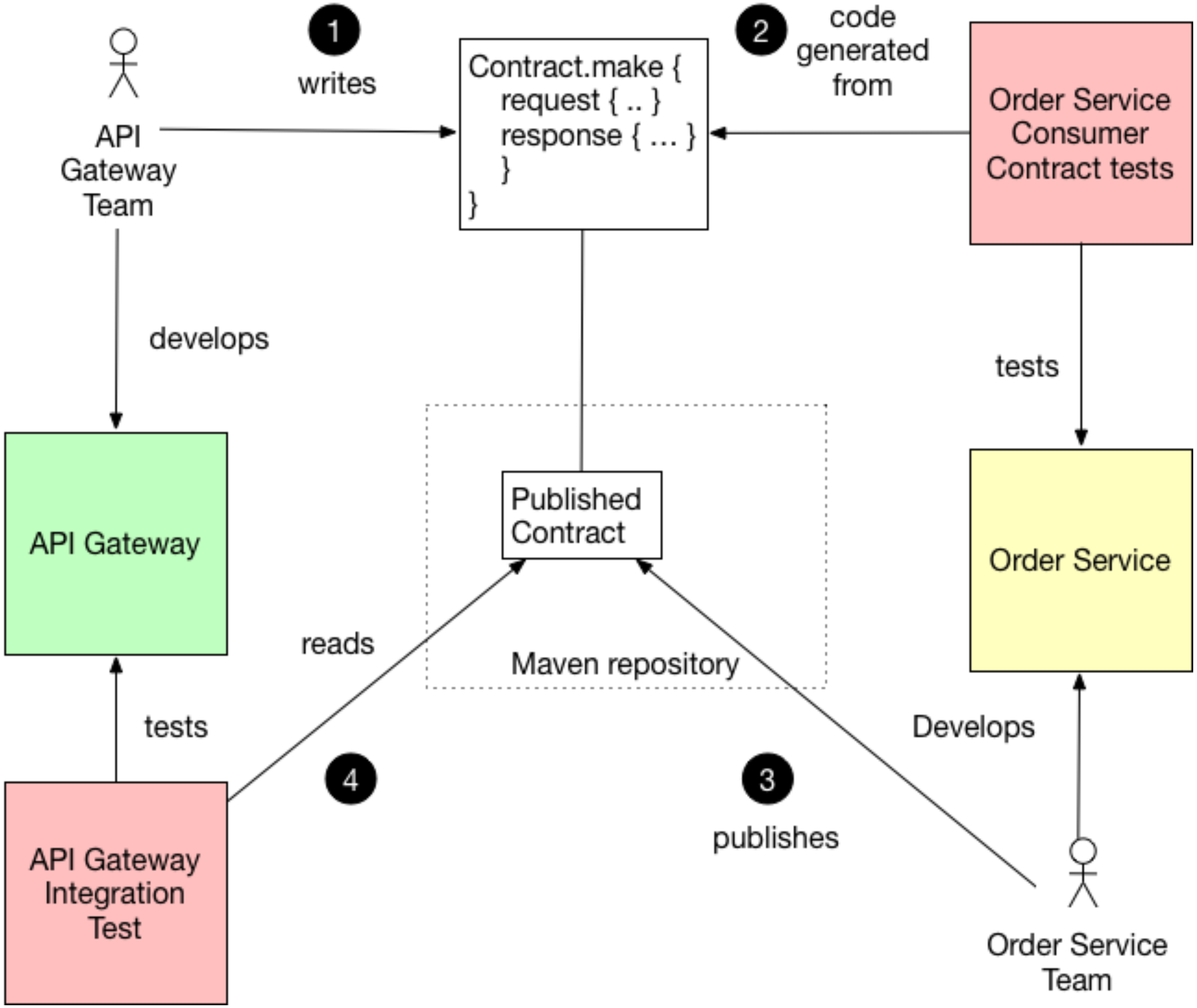
DESAFIOS PARA TESTAR MICROSSERVIÇOS



CONSUMER-DRIVEN CONTRACT TESTING

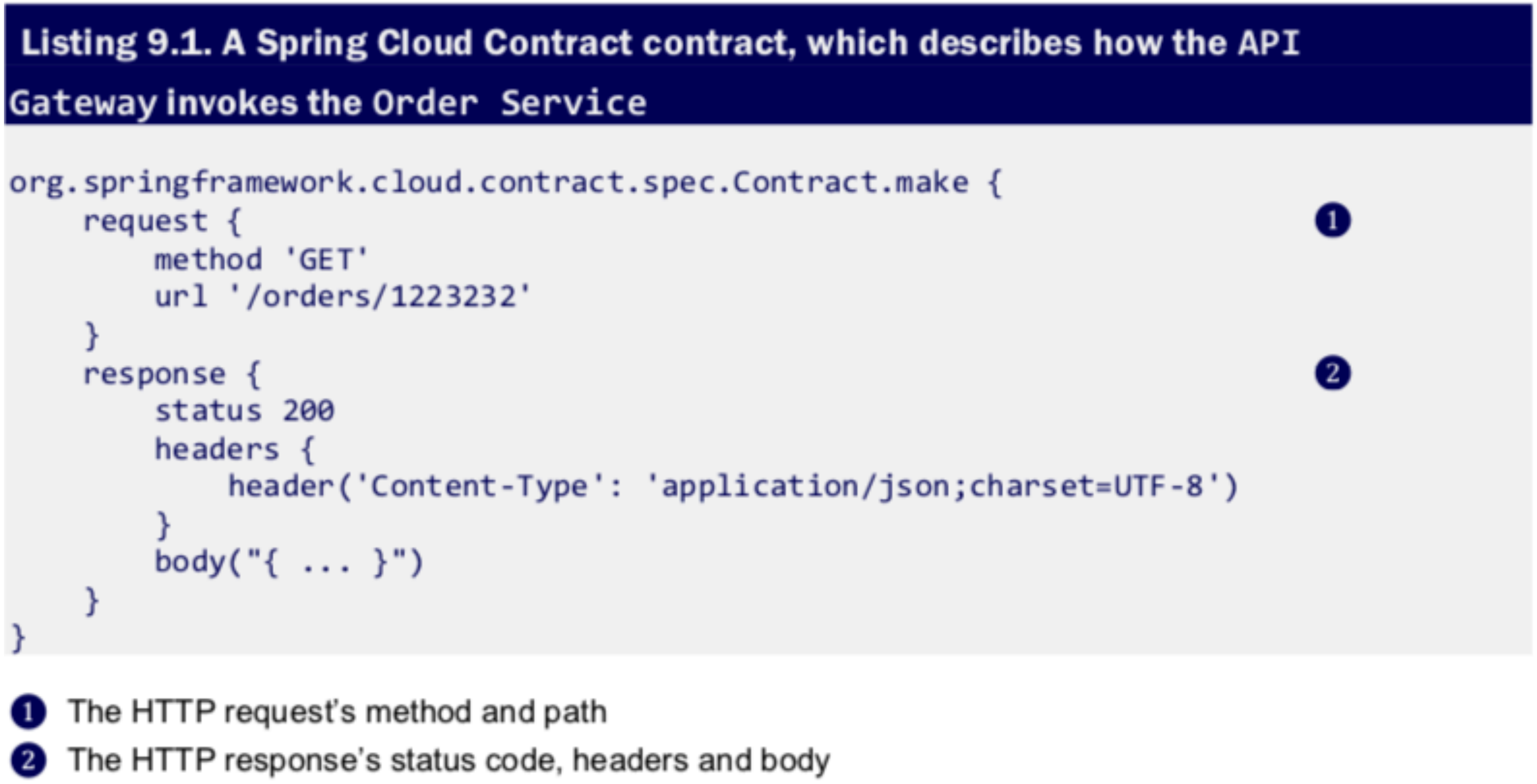

API Gateway team
Order History Service Team
... Service Team
Writes
Writes
Writes
API Gateway - Order Service Contract Test Suite
Order History Service - Order Service Contract Test Suite
... Service - Order Service Contract Test Suite
Order Service Deployment pipeline
Tests
Tests
Tests
Order Service

TESTANDO SERVIÇOS USANDO SPRING CLOUD CONTRACT



```
Contract.make {
   request { .. }
   response { … }
   }
}
```

TESTANDO SERVIÇOS USANDO SPRING CLOUD CONTRACT



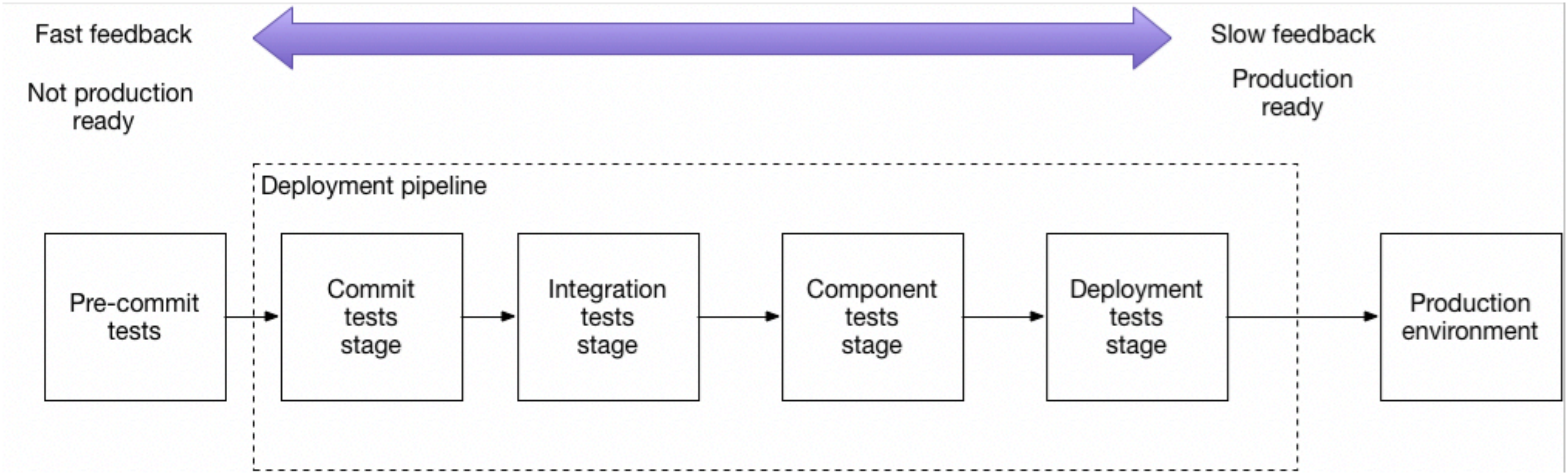
Listing 9.1. A Spring Cloud Contract contract, which describes how the API Gateway invokes the Order Service

```
org.springframework.cloud.contract.spec.Contract.make {
    request {                                                         ❶
        method 'GET'
        url '/orders/1223232'
    }
    response {                                                        ❷
        status 200
        headers {
            header('Content-Type': 'application/json;charset=UTF-8')
        }
        body("{ ... }")
    }
}
```

❶ The HTTP request's method and path

❷ The HTTP response's status code, headers and body

TESTES E O DEPLOYMENT PIPELINE

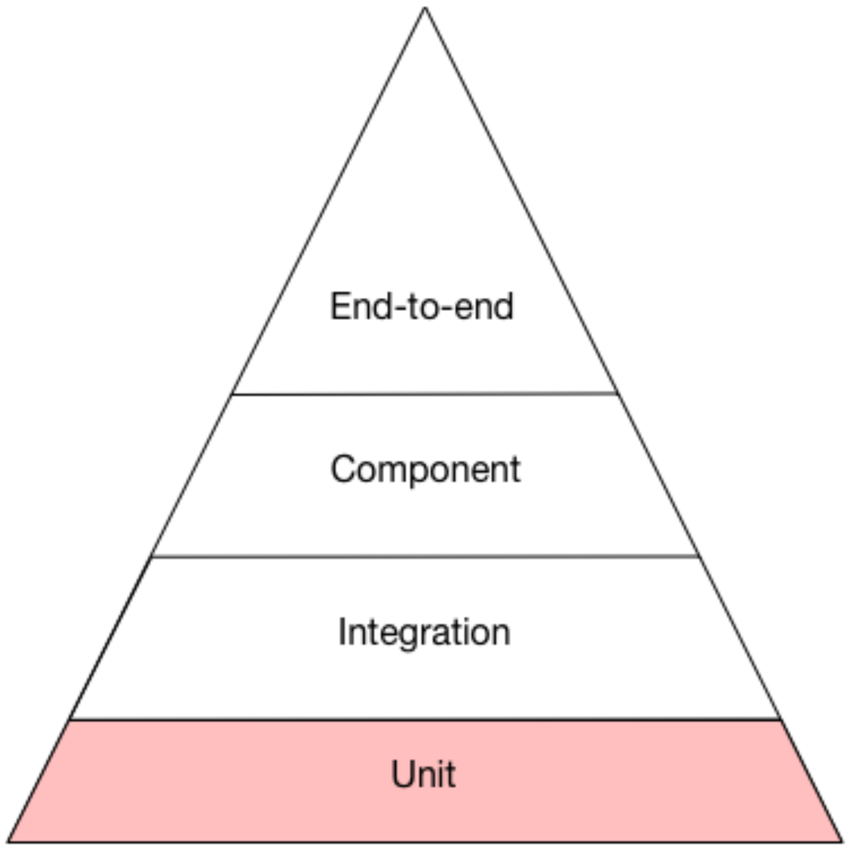


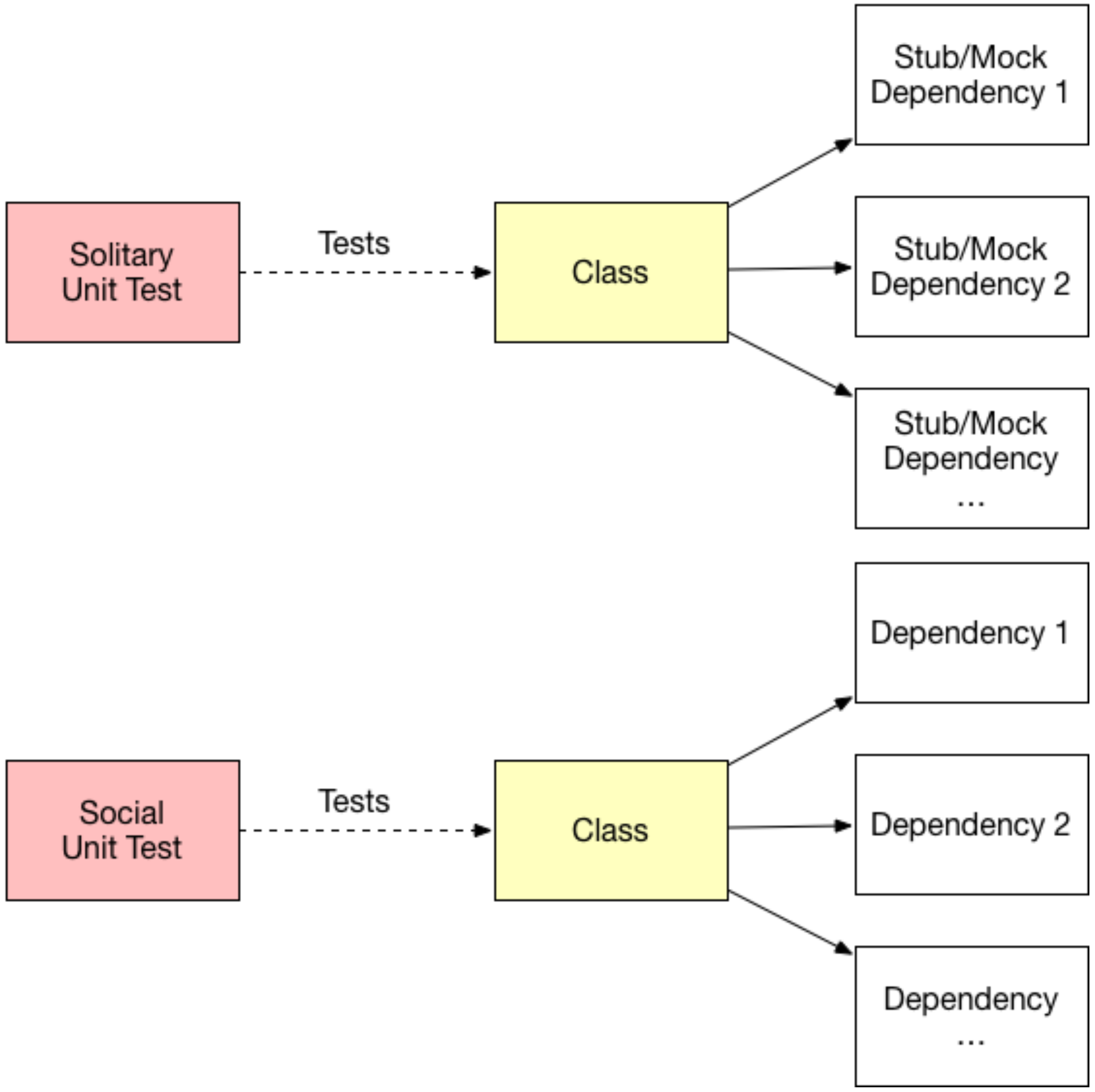
ALGUMAS REFERÊNCIAS

- https://cdn.agilitycms.com/sauce-labs/white-papers/sauce-labs-state-of-testing-2018.pdf
- http://xunitpatterns.com/Four%20Phase%20Test.html
- https://martinfowler.com/bliki/TestPyramid.html
- https://cloud.spring.io/spring-cloud-contract/
- https://github.com/pact-foundation

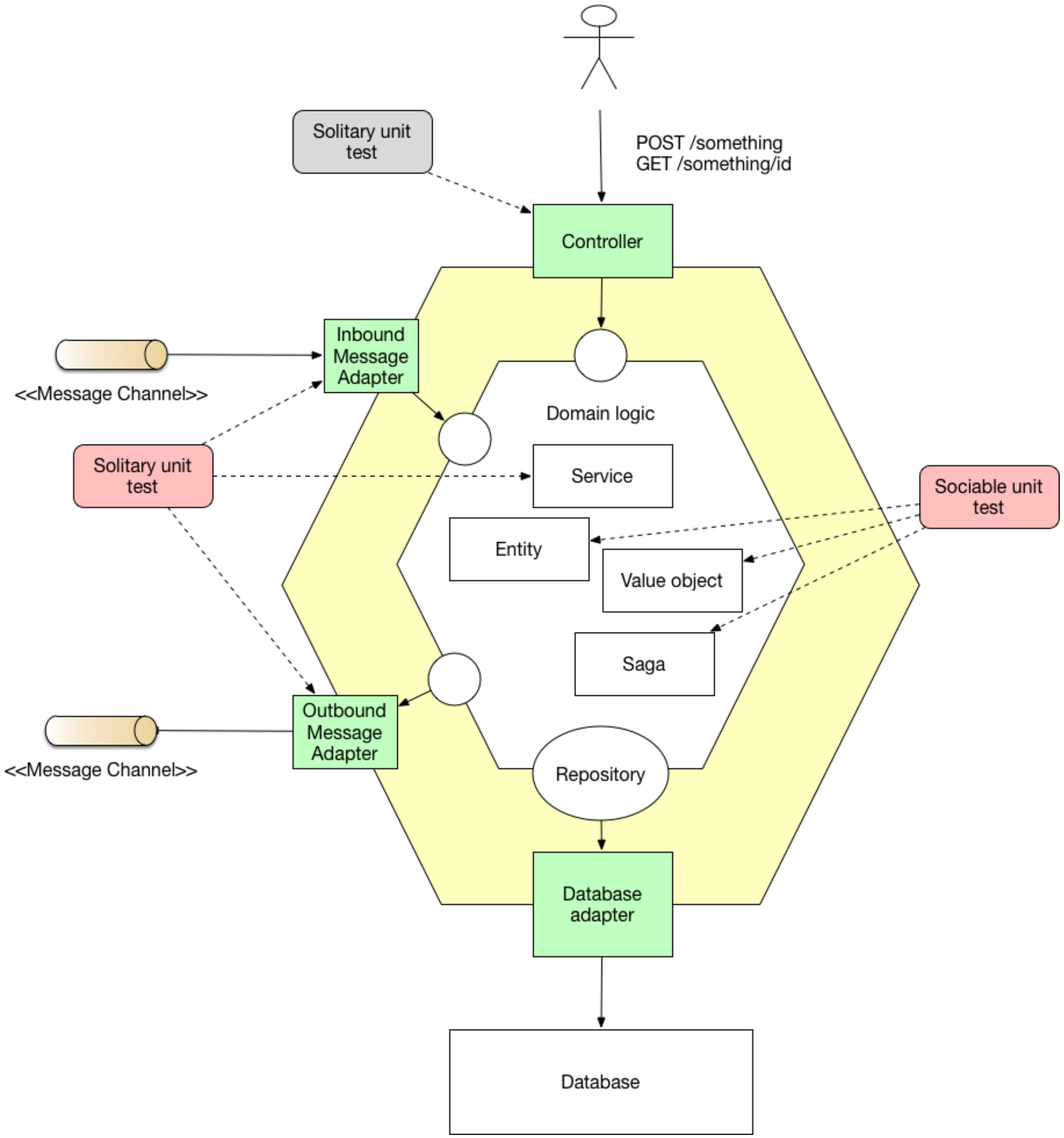
ESCREVENDO TESTES UNITÁRIOS PARA UM SERVIÇO

TESTES UNITÁRIOS

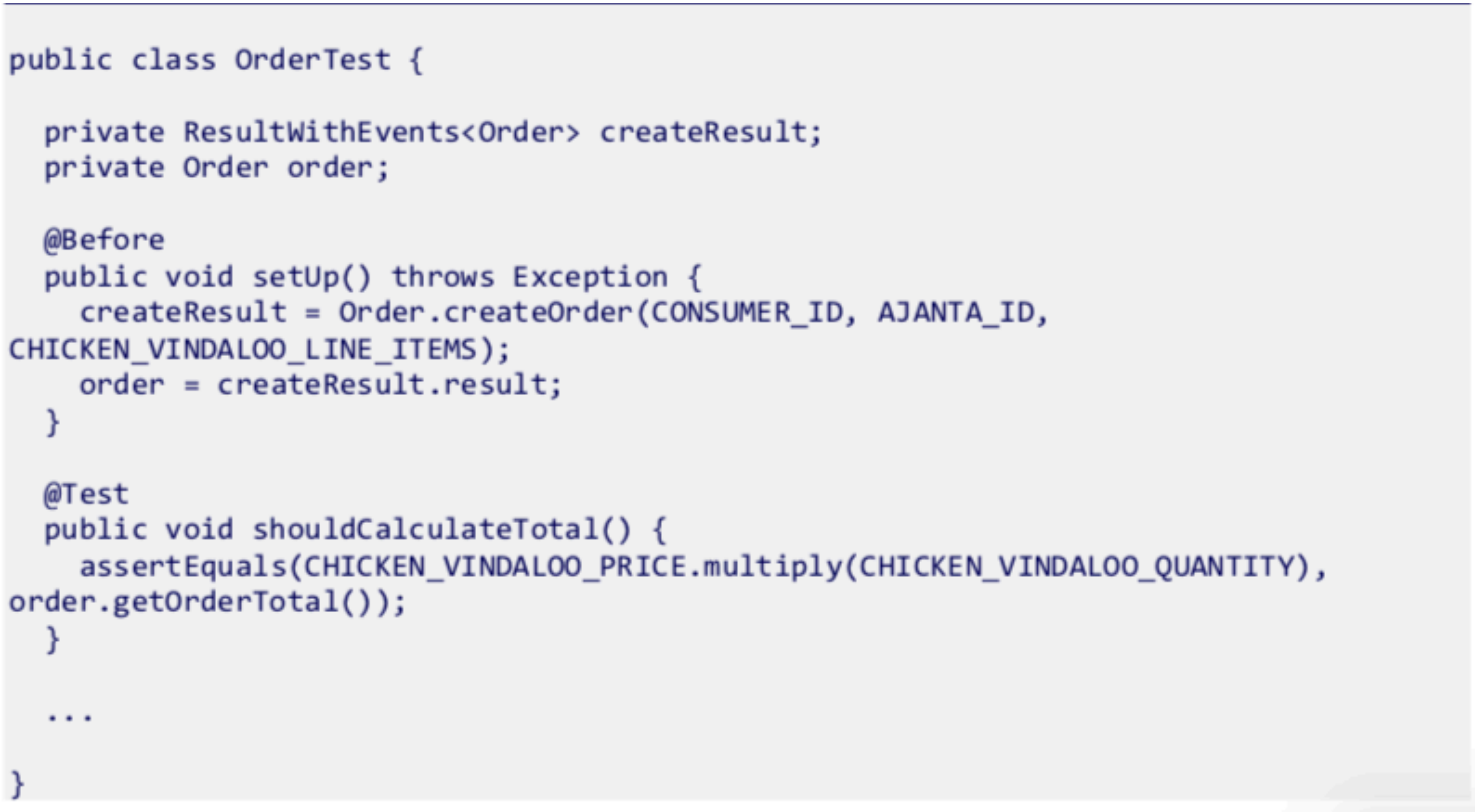




COMO DETERMINAR QUE TIPO DE TESTE UNITÁRIO UTILIZAR

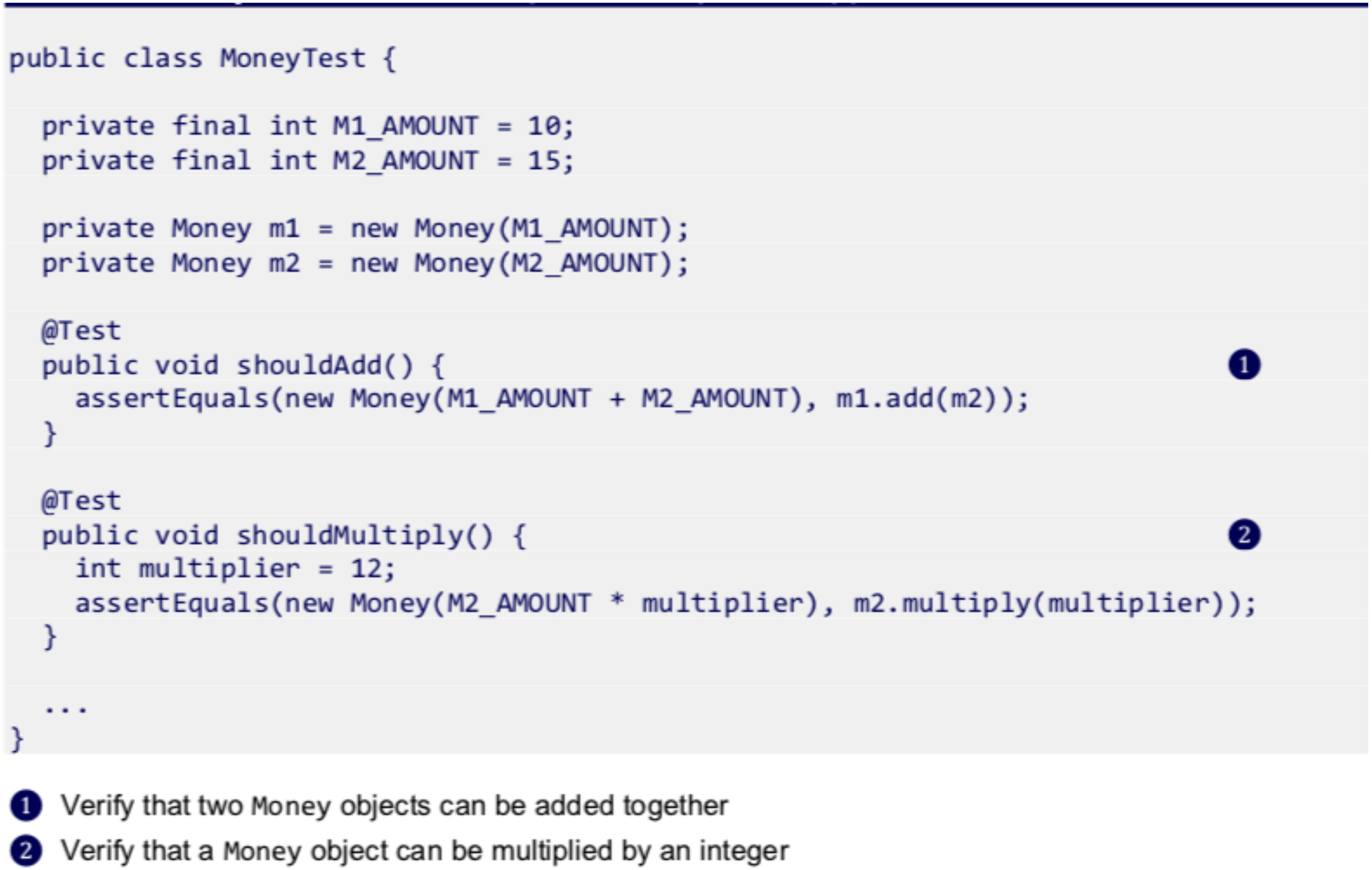


TESTES UNITÁRIOS PARA ENTIDADES



```
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());
  }

  ...

}
```

TESTES UNITÁRIOS PARA VALUE OBJECTS

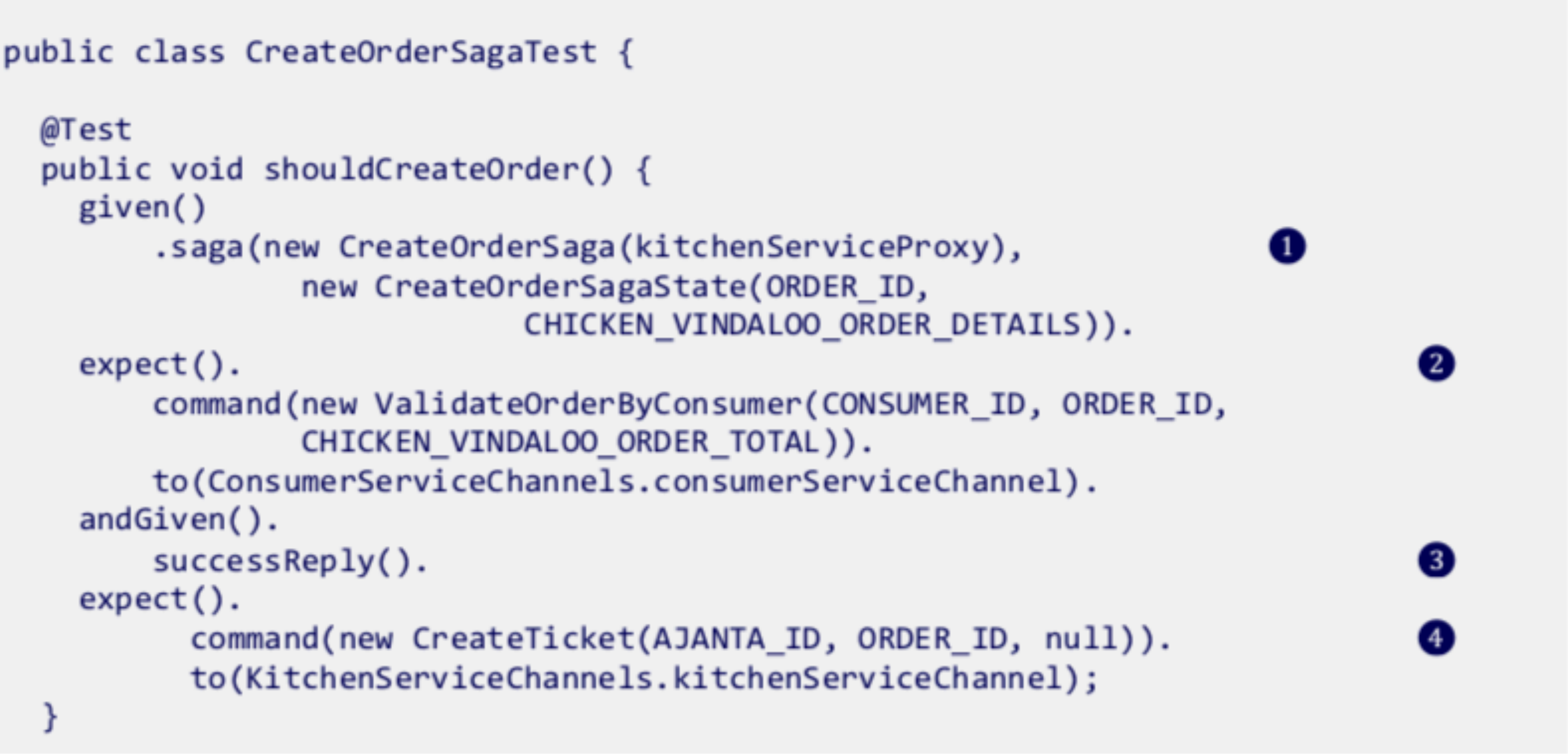


```
public class MoneyTest {

  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() {                                                ❶
    assertEquals(new Money(M1_AMOUNT + M2_AMOUNT), m1.add(m2));
  }

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

  ...
}
```

❶ Verify that two `Money` objects can be added together

❷ Verify that a `Money` object can be multiplied by an integer

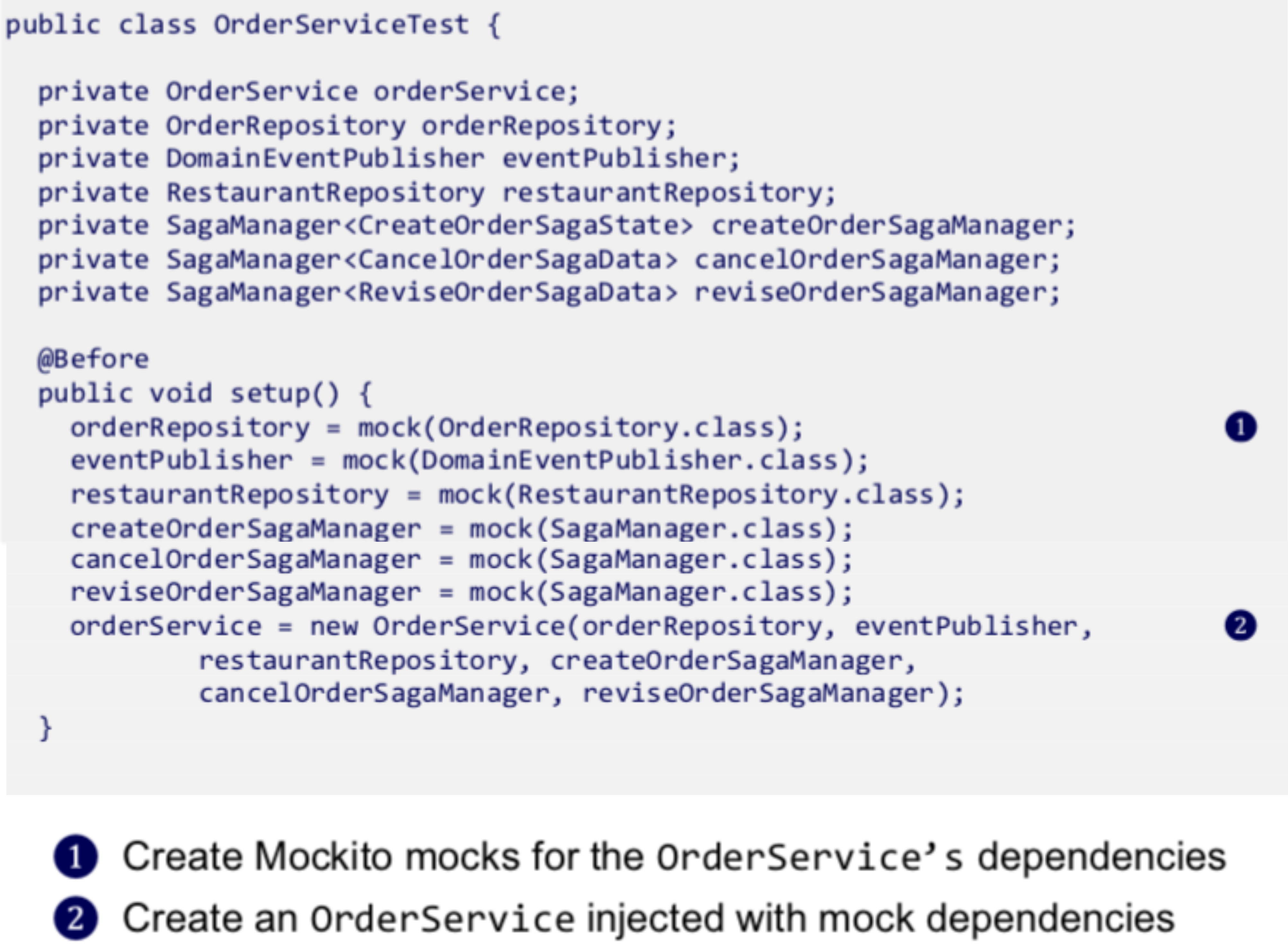
TESTES UNITÁRIOS PARA SAGAS



```
public class CreateOrderSagaTest {

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

❶ Create the saga
❷ Verify that it sends a ValidateOrderByConsumer message to the Consumer Service
❸ Send a Success reply to that message
❹ Verify that it sends a CreateTicket message to the Kitchen Service

TESTES UNITÁRIOS PARA SERVIÇOS DE DOMÍNIO

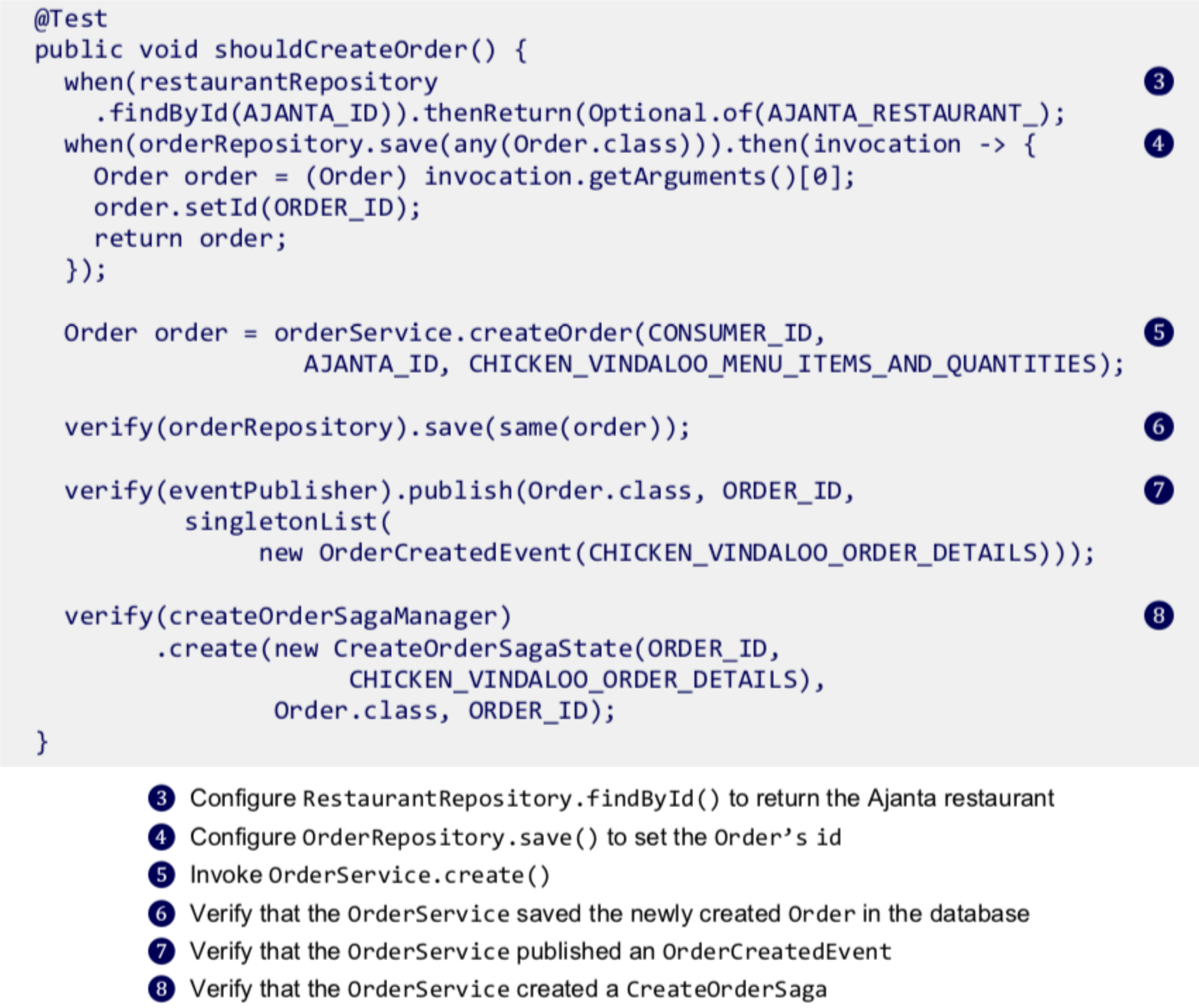


```
public class OrderServiceTest {

  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);                   ❶
    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,  ❷
            restaurantRepository, createOrderSagaManager,
            cancelOrderSagaManager, reviseOrderSagaManager);
  }
```

❶ Create Mockito mocks for the `OrderService`’s dependencies

❷ Create an `OrderService` injected with mock dependencies

TESTES UNITÁRIOS PARA SERVIÇOS DE DOMÍNIO



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

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

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

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

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

❸ Configure `RestaurantRepository.findById()` to return the Ajanta restaurant

❹ Configure `OrderRepository.save()` to set the `Order`’s `id`

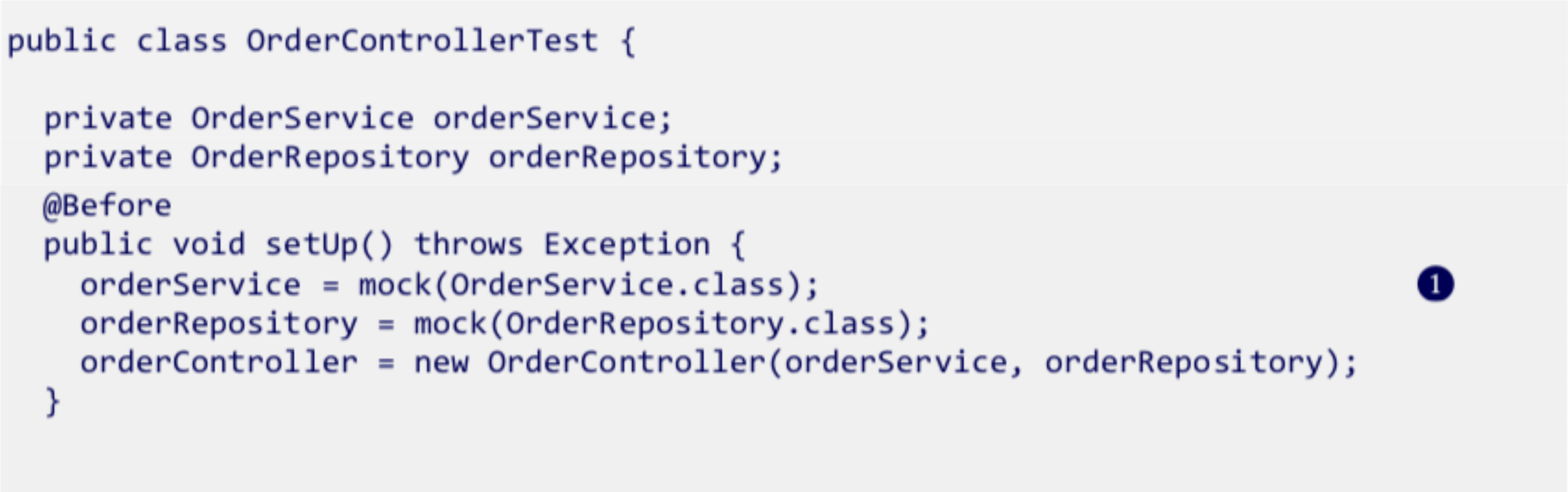
❺ Invoke `OrderService.create()`

❻ Verify that the `OrderService` saved the newly created `Order` in the database

❼ Verify that the `OrderService` published an `OrderCreatedEvent`

❽ Verify that the `OrderService` created a `CreateOrderSaga`

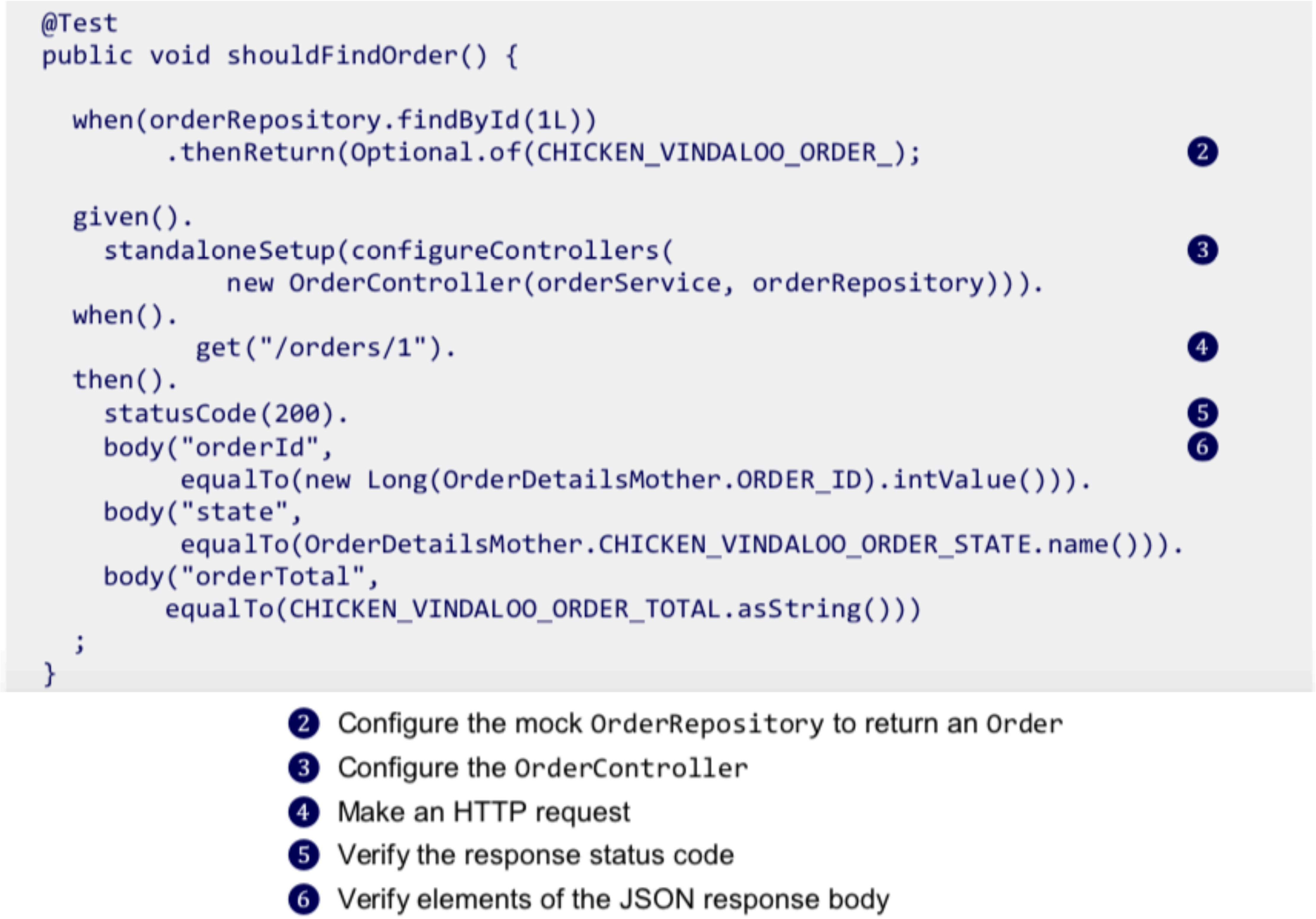
TESTES UNITÁRIOS PARA CONTROLADORES



```
public class OrderControllerTest {

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

❶ Create mocks for the OrderController’s dependencies

TESTES UNITÁRIOS PARA CONTROLADORES



```
@Test
public void shouldFindOrder() {

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

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

❷ Configure the mock `OrderRepository` to return an `Order`

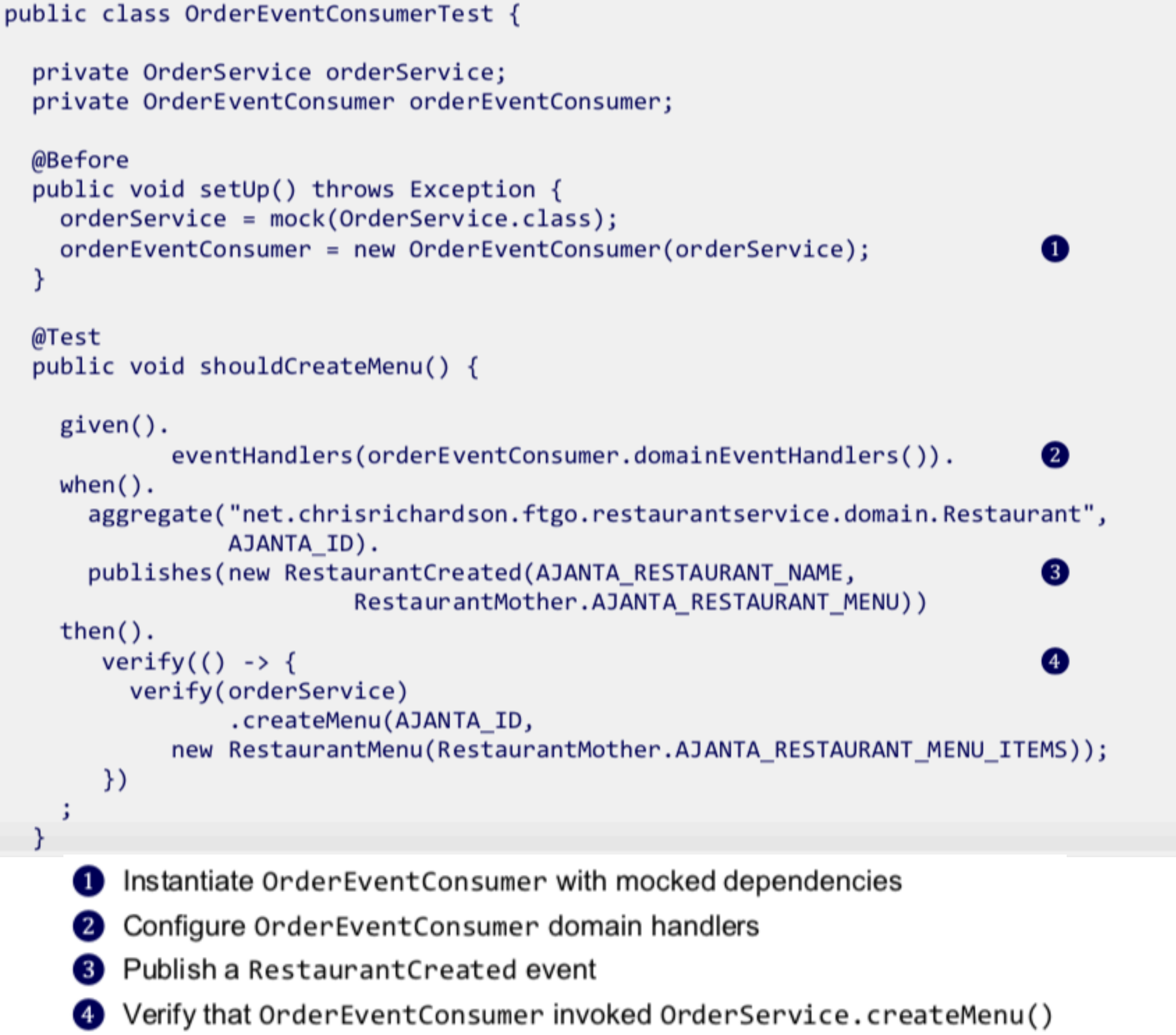
❸ Configure the `OrderController`

❹ Make an HTTP request

❺ Verify the response status code

❻ Verify elements of the JSON response body

TESTES UNITÁRIOS PARA EVENTOS E MESSAGE HANDLERS



```
public class OrderEventConsumerTest {

  private OrderService orderService;
  private OrderEventConsumer orderEventConsumer;

  @Before
  public void setUp() throws Exception {
    orderService = mock(OrderService.class);
    orderEventConsumer = new OrderEventConsumer(orderService);             ❶
  }

  @Test
  public void shouldCreateMenu() {

    given().
            eventHandlers(orderEventConsumer.domainEventHandlers()).       ❷
    when().
      aggregate("net.chrisrichardson.ftgo.restaurantservice.domain.Restaurant",
                AJANTA_ID).
      publishes(new RestaurantCreated(AJANTA_RESTAURANT_NAME,              ❸
                         RestaurantMother.AJANTA_RESTAURANT_MENU))
    then().
       verify(() -> {                                                      ❹
         verify(orderService)
                .createMenu(AJANTA_ID,
            new RestaurantMenu(RestaurantMother.AJANTA_RESTAURANT_MENU_ITEMS));
       })
    ;
  }
```

❶ Instantiate OrderEventConsumer with mocked dependencies

❷ Configure OrderEventConsumer domain handlers

❸ Publish a RestaurantCreated event

❹ Verify that OrderEventConsumer invoked OrderService.createMenu()

RESUMO

- Testes automáticos são a base da entrega rápida e segura de software. Por conta da complexidade da arquitetura de microsserviços, para se beneficiar completamente dela, devem ser escritos testes.
- O propósito de um teste é verificar o "behavior of the system under test (SUT). Nesta definição, sistema é um termo que significa o elemento que está sendo testado, que pode ser algo pequeno, como uma classe, como grande como a aplicação inteira ou algo no meio, como um conjunto de classes de um serviço individual.
- Uma boa forma de simplificar e agilizar os testes é usar os "dublês de teste", que são objetos que simulam o comportamento de uma dependência. Há dois tipos de "dublês": stubs e mocks. Um stub é um objeto que retorna valores para a SUT. Um mock é um objeto que o teste usa para verificar se o SUT invoca corretamente uma dependência.
- Usar a pirâmide de testes para determinar onde focar os esforços de testes para o seu serviço. A maior parte dos testes devem ser rápidos, confiáveis e fáceis de escrever. Deve-se procurar minimizar o número de testes fim-a-fim, visto que são lentos e dispendiosos.