Universidad Católica de Córdoba
Universidad Jesuita

Ingeniería de software III

Automatización de Pruebas

Testing

✧Muchos proyectos solamente dependen del testing manual para verificar que el software cumpla con los requerimientos funcionales y no funcionales

✧A veces, existen tests automatizados pero están desactualizados y pobremente mantenidos y requieren que se los suplemente con test manual extensivo

Build quality in

✧*“No depender en inspecciones masivas para alcanzar la calidad, mejorar el proceso y contruir con la calidad incluida en el producto desde el principio”*

✧Testing es una actividad que involucra a todo el equipo y debería ser continua desde el principio del proyecto

✧“Building quality in” para testing significa
- ✧Escribir tests automatizados a diferentes niveles:
  - ✧Unidad
  - ✧Componente
  - ✧Aceptación
- ✧Ejecutarlos como parte del deployment pipeline el cual es iniciado cada vez que se cambia
  - ✧el código, la configuración, o el entorno dónde se ejecuta

Proyecto ideal

✧Colaboración entre testers, developers, usuarios para escribir tests automatizados desde el comienzo del proyecto

✧Los tests se escriben antes que los desarrolladores comiencen a trabajar en la feature que los tests testean

✧El conjunto de tests forman una especificación ejecutable del comportamiento del sistema

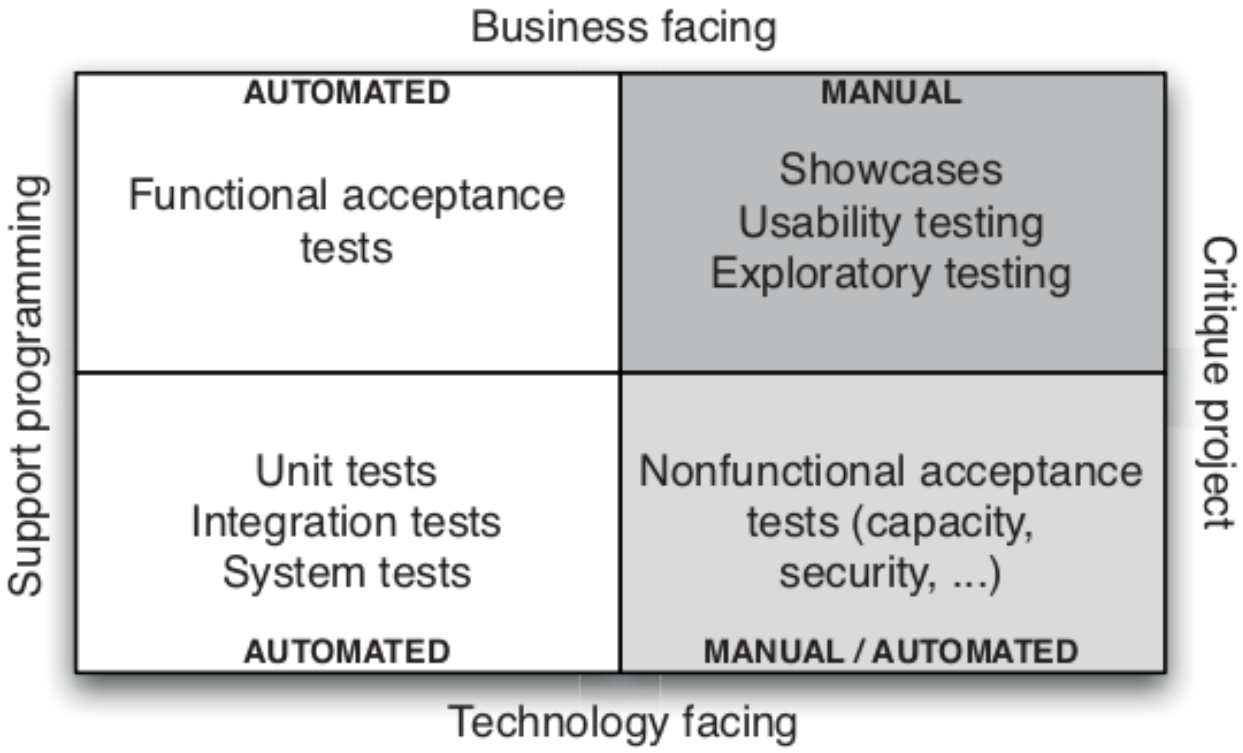
- ✧Cuando pasan demuestran que la funcionalidad requerida por el cliente ha sido implementada completamente y correctamente

✧El conjunto de tests es ejecutado por el sistema de CI

✧El conjunto de tests también sirve como tests de regresión

✧También se incluyen tests automatizados para los aspectos “no funcionales” del sistema

Tipos de tests



Soporte del proceso de desarrollo, de cara al negocio

- Tests funcionales de aceptación
- Aseguran que el criterio de aceptación de una story se cumple
- Deberían ser escritos y automatizados antes que el desarrollo de la story comience
- Son críticos en un entorno ágil responden a preguntas
  - ¿Cómo sé cuándo terminé la implementación? -> desarrollador
  - ¿Conseguí lo que quería? -> usuario
- En un proyecto ideal los usuarios deberían escribir el criterio de éxito de una story que debería traducirse a un test de aceptación
  - Cucumber es un ejemplo de tool que le permite al usuario escribir el test script (en su lenguaje)
  - Y los testers y developers trabajan en su implementación
  - La tool provee el mecanismo para mantenerlos sincronizados

Soporte del proceso de desarrollo, de cara al negocio

- Modelo given-when-then para tests
- ¿Cuáles testear?
- Happy path
  - Given [estado inicial],
  - when [acciones],
  - then [nuevo estado] will result
- Alaternative path
  - Variaciones del estado inicial, acciones y o estado final
  - Particiones equivalentes y boundary values para determinar cual tests implementar
- Sad paths
  - Causan errores

Soporte del proceso de desarrollo, de cara al negocio

- Automatizar los tests de aceptación
  - Costo
  - Retroalimetación más rápida
  - Reducen la carga de los testers
  - Los testers se pueden centrar en el test exploratorio
  - El conjunto de tests de aceptación también representan un conjunto de test de regresión poderoso
  - Usando nombres legibles por los humanos en los tests, se puede generar la documentación del sistema a partir de los tests
  - Usados para regresión

Soporte del proceso de desarrollo, de cara a la tecnología

- Escritos y mantenidos exclusivamente por desarrolladores
  - Unit tests
  - Componente tests
  - Deployment tests

Soporte del proceso de desarrollo, de cara a la tecnología

- Pruebas de unidad
  - Prueban una parte en particular del software aislada
  - Necesitan simular otras partes del sistema (test doubles)
  - No deberían llamar, bases de datos, otros sistemas, etc
    - Deberían ser muy rápidos
  - Deberían por lo menos cubrir el 80%, idealmente el 100%
  - Pierden la posibilidad de detectar bugs que provienen de la interacción entre diferentes partes del software

Soporte del proceso de desarrollo, de cara a la tecnología

- Pruebas de componente
  - Prueban un cluster de funcionalidad mayor, para poder encontrar problemas que surgen de la interacción
  - Son más lentos
  - A veces se los conoce como tests de integración

Soporte del proceso de desarrollo, de cara a la tecnología

- Pruebas de Despliegue
  - Se ejecutan cada vez que se instala la aplicación
  - Verifican que el deployment funcionó
  - Que la aplicación fue
    - correctamente instalada
    - correctamente configurada
    - capaz de contactar cualquier servicio que necesite y que este responda

Critican el proyecto de cara al negocio

✧Manuales

✧Verifican que la aplicación le entrega al usuario el valor que este está esperando

✧No es solo validar que la aplicación cumpla con las especificaciones, es también validar que las especificaciones están correctas

- ✧Cuando los usuarios utilizan la aplicación en vida real siempre descubren que se puede mejorar
- ✧Tal vez se descubren nuevas features inspiradas en la entrega actual
- ✧El desarrollo de software es iterativo en su esencia

Critican el proyecto de cara al negocio

- Demos o reviews
  - Los teams ágiles muestran al final de cada iteración el resultado de lo que se desarrollo
  - Una demo siempre tendrá comentarios, son el usuario y el team los que deciden cuanto quieren cambiar el plan para incorporar el resultado de la review
  - Cuando estos tests pasan es cuando uno realmente puede decir que ha cumplido la labor -> cliente contento :-)

- Canary testing

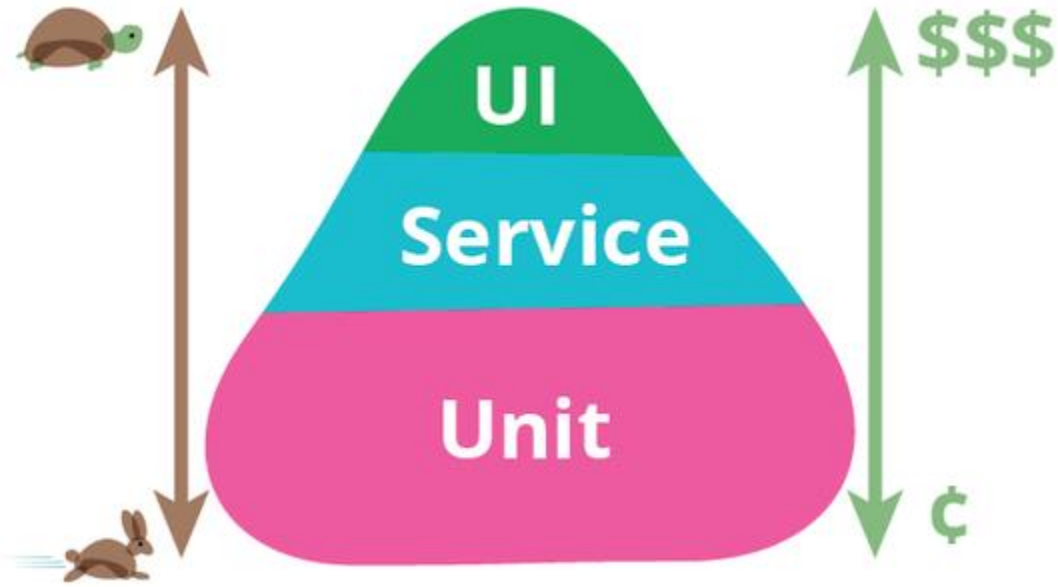
Critican el proyecto de cara a la tecnología

- Los tests de aceptación vienen en dos categorías
  - funcionales
  - no funcionales
- No funcionales
  - Todas las características de un sistema diferentes a la funcional: por ejemplo: capacity, availability, security, etc
- El usuario muchas veces no especifica los requerimientos no funcionales desde un principio pero los sobreentiende muchas veces
- Requieren una cantidad considerable de recursos
- Requieren tools especificas
- Son más lentos
- Tienden a correrse más hacia el final del deployment pipeline

Test en el Commit Stage

Test en el Commit Stage

- La mayoría deberían ser test de unidad
  - Deben ser rápidos
  - Deben tener buena cobertura (~80% o más)
- Los unit tests por sí solos no son suficiente para asegurar que la aplicación funciona, para eso está el resto del deployment pipeline



Test en el Commit Stage

- ¿Cómo lograr que los test de unidad sean rápidos?
  - Evitar la interfaz de usuario
    - Tiempos de los humanos son extremadamente lentos para las máquinas
    - Gran cantidad de componentes requiere mucho esfuerzo
  - Injección de dependencias o inversión de control
    - Dependencias provistas desde fuera
    - Buen diseño, permiten introducción de test doubles
  - Evitar la base de datos
    - O cualquier otro subsistema que no sea el testeado
    - Debería ser sencillo de evitar
  - Evitar el asincronismo
    - En el commit stage debe ser evitada
    - Dividir los tests

Test en el Commit Stage

- ¿Cómo lograr que los test de unidad sean rápidos?
  - Usar test-doubles
  - Minimizar el estado en los tests
    - Testear comportamiento y minimizar el estado
    - Evitar tests complejos que lleven efuerzo considerable para preparar los datos de entrada
  - Simular el tiempo
    - Abstraer la información del tiempo en una clase separada
    - Injectar una implementación que simula el tiempo
  - Fuerza bruta
    - Paralelizar la ejecución
    - Separar los tests suites para poder paralelizar

Test doubles

- Una parte clave del test automatizado involucra reemplazar parte del sistema con una version simulada
- Esto se hace para lograr tener más control sobre el comportamiento
- Estas partes simuladas se denominan “test doubles”
- Tipos
  - Dummy objects, se pasan pero nunca se usan
  - Fake objects, tienen implementaciones funcionales, pero tomando atajos que los hacen no usables en producción - in-memory db
  - Stubs, proveen respuestas predeterminadas a las llamadas que reciben durante el tests
  - Spies, registran información en base a como son llamados durante el tests
  - Mocks, son objetos pre-programados con “expectations” que forman una especificación de las llamadas que esperan recibir

Test doubles

✧Mocks

```
class Car {
  private Engine engine;

  public Car(Engine engine) {
    this.Engine = engine;
  }

  public void drive() {
    engine.start();
    engine.accelerate();
  }
}

Interface Engine {
  public start();
  public accelerate();
}
```

```
class TestEngine implements Engine {
  boolean startWasCalled = false;
  boolean accelerateWasCalled = false;
  boolean sequenceWasCorrect = false;

  public start() {
    startWasCalled = true;
  }
  public accelerate() {
    accelerateWasCalled = true;
    if (startWasCalled == true) {
      sequenceWasCorrect = true;
    }
  }
  public boolean wasDriven() {
    return startWasCalled && accelerateWasCalled && sequenceWasCorrect;
  }
}
```

```
class CarTestCase extends TestCase {
  public void testShouldStartThenAccelerate() {
    TestEngine engine = new TestEngine();
    Car car = new Car(engine);

    car.drive();

    assertTrue(engine.wasDriven());
  }
}
```

Test doubles

- Mocks

```
class Car {
  private Engine engine;

  public Car(Engine engine) {
    this.Engine = engine;
  }

  public void drive() {
    engine.start();
    engine.accelerate();
  }
}

Interface Engine {
  public start();
  public accelerate();
}
```

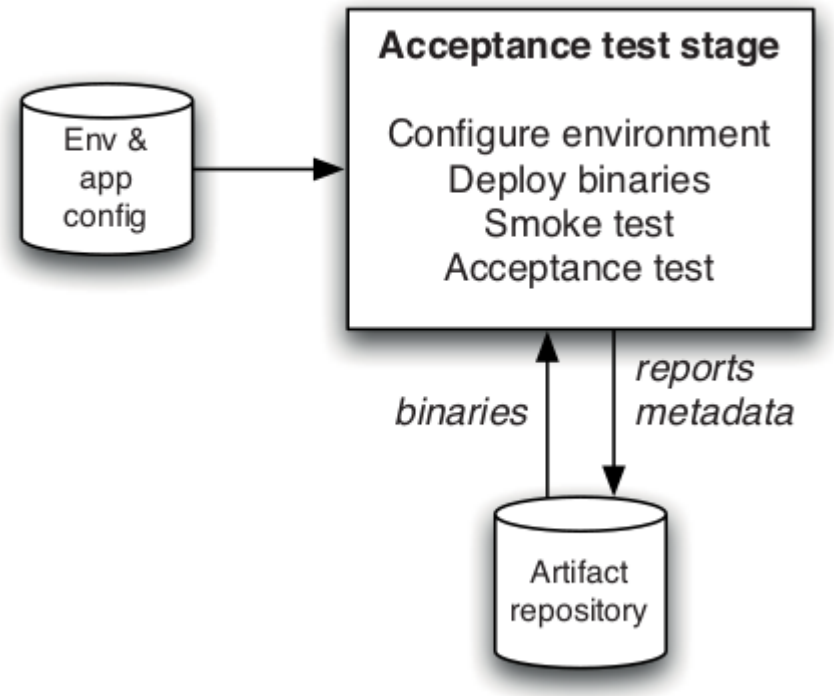
```
class CarTestCase extends TestCase {
  public void testShouldStartThenAccelerate() {
    TestEngine engine = new TestEngine();
    Car car = new Car(engine);

    car.drive();

    assertTrue(engine.wasDriven());
  }
}
```

```
class CarTestCase extends MockObjectTestCase {
  public void testShouldStartThenAccelerate() {
    Mock mockEngine = mock(Engine);
    Car car = new Car((Engine)mockEngine.proxy());

    mockEngine.expects(once()).method("start");
    mockEngine.expects(once()).method("accelerate");

    car.drive();
  }
}
```

Tests en el Acceptance Test Stage

Test de Aceptación Automatizado

- ✧Son una parte crucial en el deployment pipeline
- ✧Lleva a los equipos más allá del simple CI
- ✧Testean el criterio de aceptación del negocio
- ✧Se corren contra toda versión de la aplicación que pase el commit stage



Test de Aceptación Automatizado

- ¿Por qué son esenciales?
  - El costo de testear manualmente es alto, dado que tiene que repetirse con cada nueva version para comprobar si puede ser releseada o no
  - Testing manual suele hacerse al último
    - Los defectos se capturan tarde
    - Suele planificarse incorrectamente
  - Los tests de unidad y de componente no se basan en testear escenarios por lo que no son suficiente
  - Los tests de aceptación proveen una protección contra cambios grandes en el diseño o la arquitectura
    - Los tests de unidad y de componente tienen que rehacerse pero no los tests de aceptación
  - Proveen una presión saludable para tener requerimientos de calidad

Test de Aceptación Automatizados Mantenibles

✧Están basados en los criterios de aceptación por lo que estos deben escribirse con la automatización en mente
✧Una vez que se tiene el COS se debe automatizar
✧Los tests de aceptación deben siempre ser implementados en capas

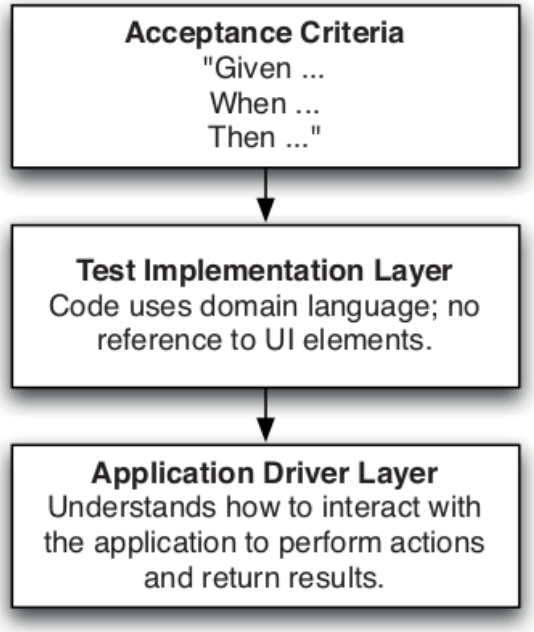


Figure 8.2 *Layers in acceptance tests*

Test de Aceptación Automatizados Mantenibles

- Primera capa
  - Criterio de aceptación
  - Herramientas como cucumber permiten poner el COS directamente en el test y linkearlo automaticamente con la implementación
  - También puede utilizarse los nombres de los métodos para lograr esto
  - Es fundamental que se haga en lenguage del dominio y no contenga detalles de la interacción con la aplicación porque se vuelven frágiles
  - Los tests de aceptación no deberían ir contra la UI
    - Si bien la UI es la forma de interactuar del usuario, no debería contener lógica de negocios
    - Tests contra la misma API que utiliza la UI es preferible

Test de Aceptación Automatizados Mantenibles

- Business Analyst / Product Owner / Representante del cliente y los usuarios
  - Trabajan con
    - los usuarios
      - para identificar y priorizar los requerimientos
    - los developers
      - para asegurarse que los developers entienden bien la aplicación y la perspectiva del usuario
      - para asegurarse que las stories entreguen el valor de negocio esperado
    - los testers
      - para asegurar que el CoS está especificado apropiadamente
      - para asegurar que la funcionalidad desarrollada cumple con el COS y entrega el valor esperado
- Este es un rol, no una persona

CoS como especificación ejecutable

- En un entorno ágil, los tests automatizados no solo sirven a su propósito de testear el software
- También son una especificación ejecutable del comportamiento del software que se está desarrollando
- BDD ->
  - CoS debe ser escrita en la forma de expectativas del usuario respecto del comportamiento del sistema
  - Entonces, es posible ejecutarla directamente contra la aplicación para verificar que cumple con la especificación
- Ventajas
  - No quedan out-of-date porque fallan en su ejecución apenas lo hacen
    - El deployment pipeline las verifica para cada cambio
    - Con los enfoques tradicionales para especificar un sistema las especificacionesse desfasan rápidamente y quedan atrasadas

CoS como especificación ejecutable

✧Los test de aceptación son business-facing
✧CoS captura el valor de negocio y debe usarse para validar
✧Cucumber permite escribir el CoS de una manera que puede automatizarse su validación

Given some initial context,
When an event occurs,
Then there are some outcomes.

✧Given
- ✧Estado inicial necesario para iniciar el test

✧When
- ✧Describe la interacción entre el usuario y la aplicación

✧Then
- ✧Describe el estado de la aplicación después que la interacción se ha completado

CoS como especificación ejecutable

✧Ejemplo

```
Feature: Placing an order
    Scenario: User order should debit account correctly
    Given there is an instrument called bond
    And there is a user called Dave with 50 dollars in his account
    When I log in as Dave
    And I select the instrument bond
    And I place an order to buy 4 at 10 dollars each
    And the order is successful
    Then I have 10 dollars left in my account
```

✧Archivo -> features/placing_an_order.feature

CoS como especificación ejecutable

```
Before do
   @admin_api = AdminApi.new
   @trading_ui = TradingUi.new
end

Given /^there is an instrument called (\w+)$/ do |instrument|
   @admin_api.create_instrument(instrument)
end

Given /^there is a user called (\w+) with (\w+) dollars in his account$/ do |user, amount|
   @admin_api.create_user(user, amount)
end

When /^I log in as (\w+)$/ do |user|
   @trading_ui.login(user)
end

When /^I select the instrument (\w+)$/ do |instrument|
   @trading_ui.select_instrument(instrument)
end

When /^I place an order to buy (\d+) at (\d+) dollars each$/ do |quantity, amount|
   @trading_ui.place_order(quantity, amount)
end

When /^the order for (\d+) of (\w+) at (\d+) dollars each is successful$/ do |quantity, instrument, amount|
   @trading_ui.confirm_order_success(instrument, quantity, amount)
end

Then /^I have (\d+) dollars left in my account$/ do |balance|
   @trading_ui.confirm_account_balance(balance)
end
```

✧features/step_definitions/placing_an_order_steps.rb

CoS como especificación ejecutable

✧Hay que desarrollar AdminAPI y TradingAPI que contienen la lógica de interacción con la aplicación, por ejemplo con Selenium o directamente con HTTP

```
Scenario: User order debits account correctly
        # features/placing_an_order.feature:3
Given there is an instrument called bond
        # features/step_definitions/placing_an_order_steps.rb:9
And there is a user called Dave with 50 dollars in his account
        # features/step_definitions/placing_an_order_steps.rb:13
When I log in as Dave
        # features/step_definitions/placing_an_order_steps.rb:17
And I select the instrument bond
        # features/step_definitions/placing_an_order_steps.rb:21
And I place an order to buy 4 at 10 dollars each
        # features/step_definitions/placing_an_order_steps.rb:25
And the order for 4 of bond at 10 dollars each is successful
        # features/step_definitions/placing_an_order_steps.rb:29
Then I have 10 dollars left in my account
        # features/step_definitions/placing_an_order_steps.rb:33

1 scenario (1 passed)
7 steps (7 passed)
0m0.016s
```

CoS como especificación ejecutable

- En resumen
  - Discutir el criterio de aceptación de la story con el cliente
  - Escribirlo en un formato ejecutable
  - Escribir la implementación del tests que usa solo lenguage del dominio y accede a la capa de aplicación “driver”
  - Crear la capa de aplicación “driver” que encapsula la forma de hablar con el sistema que está siendo testeado

- Esto representa un avance muy grande respecto de mantener criterios de aceptación en documentos word y tener que trackear que tests los valida

Application Driver Layer

- ✧Es la capa que entiende como hablar con la aplicación
- ✧La interfaz de esta capa está expresada en lenguaje del dominio
- ✧Si está bien diseñada podría incluso obviarse la capa de CoS y expresar el COS en la capa de implementación
- ✧Ejemplo de capa de implementación usando una driver layer bien diseñada

```
public class PlacingAnOrderAcceptanceTest extends DSLTestCase {
    @Test
    public void userOrderShouldDebitAccountCorrectly() {
        adminAPI.createInstrument("name: bond");
        adminAPI.createUser("Dave", "balance: 50.00");
        tradingUI.login("Dave");
        tradingUI.selectInstrument("bond");
        tradingUI.placeOrder("price: 10.00", "quantity: 4");
        tradingUI.confirmOrderSuccess("instrument: bond", "price: 10.00", "quantity: 4");
        tradingUI.confirmBalance("balance: 10.00");
    }
}
```

- ✧Si está bien diseñada hace más fácil la lectura
- ✧DSL abstrae la complejidad
- ✧Suele utirlizarse aliases en la driver layer
  - ✧permite ejecución de los tests en paralelo

¿Qué es major?

- Acceptance criteria layer
  - ¿Si o no?

Windows Driver Pattern

- ✧La parte del aplication driver layer que interactúa con la GUI se conoce como windows driver
- ✧Permite desacoplar los tests de aceptación de la GUI del sistema
- ✧Hace los tests más robustos, aislando los tests de los detalles de la UI
- ✧Un driver para cada parte de la GUI
  - ✧Equivalente a los device drivers
- ✧El código de los tests solo hablan con la GUI a través del driver apropiado

Windows Driver Pattern

✧La como
winc
✧Pe
✧Ha
✧Ur
✧El ado

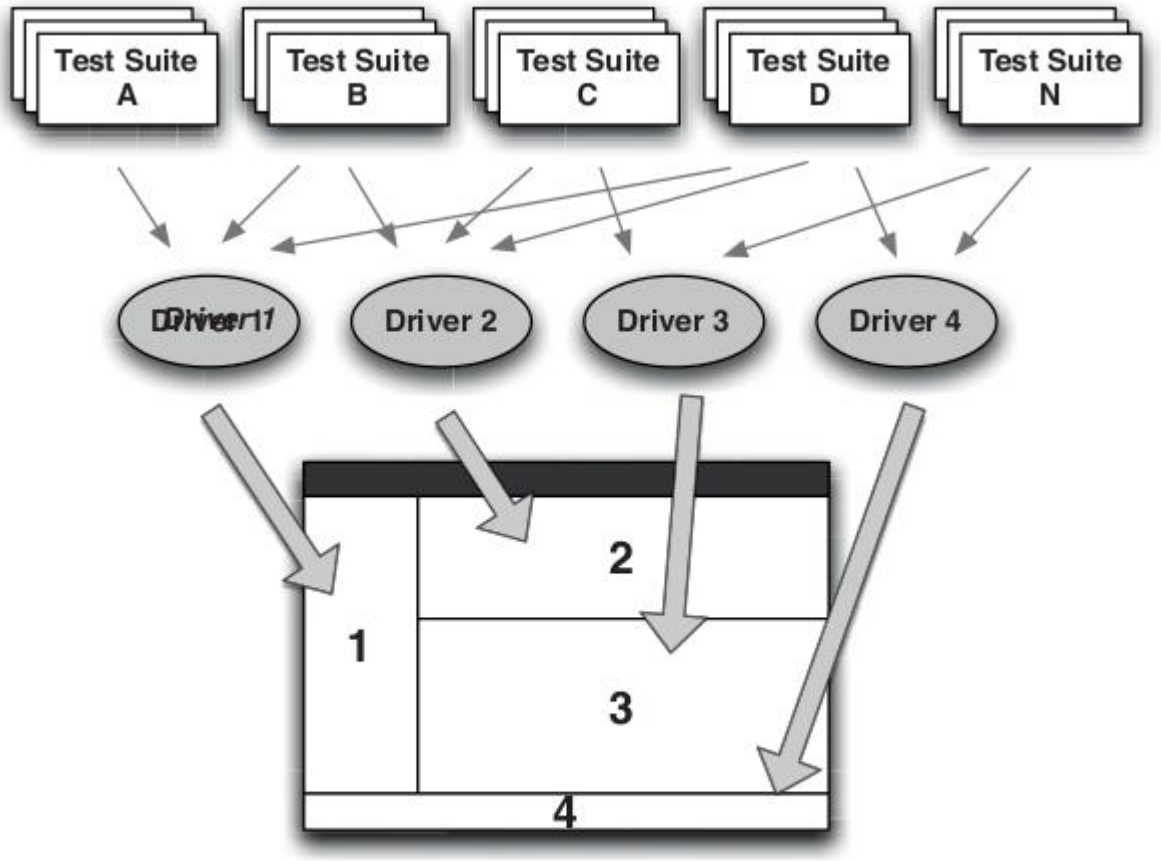


Figure 8.3 *The use of the window driver pattern in acceptance testing*

Windows Driver Pattern

✧Ejemplo de tests sin windows drivers

```
@Test
public void shouldDeductPaymentFromAccountBalance() {
  selectURL("http://my.test.bank.url");
  enterText("userNameFieldId", "testUserName");
  enterText("passwordFieldId", "testPassword");
  click("loginButtonId");
  waitForResponse("loginSuccessIndicator");

  String initialBalanceStr = readText("BalanceFieldId");

  enterText("PayeeNameFieldId", "testPayee");
  enterText("AmountFieldId", "10.05");
  click("payButtonId");

  BigDecimal initialBalance = new BigDecimal(initialBalanceStr);
  BigDecimal expectedBalance = initialBalance.subtract(new BigDecimal("10.05"));
  Assert.assertEquals(expectedBalance.toString(), readText("BalanceFieldId"));
}
```

Windows Driver Pattern

✧Ejemplo de tests con windows drivers

```
@Test
public void shouldDeductPaymentFromAccountBalance() {
  AccountPanelDriver accountPanel = new AccountPanelDriver(testContext);

  accountPanel.login("testUserName", "testPassword");
  accountPanel.assertLoginSucceeded();

  BigDecimal initialBalance = accountPanel.getBalance();
  accountPanel.specifyPayee("testPayee");
  accountPanel.specifyPaymentAmount("10.05");
  accountPanel.submitPayment();

  BigDecimal expectedBalance = initialBalance.subtract(new BigDecimal("10.05"));

  Assert.assertEquals(expectedBalance.toString(), accountPanel.getBalance());
}
```

Implementación de los tests de aceptación

- Manejando el estado
  - Los unit tests deberían ser stateless, pero no es posible lograrlo con los tests de aceptación
  - La intención de los tests de aceptación es
    - simular las interacciones del usuario con el sistema
    - ejercitar el sistema y
    - probar que este cumple con los requerimientos del negocio
  - Para poder probar el comportamiento de la aplicación el test depende de que la aplicación esté en un estado específico inicial
    - Esto puede ser difícil

Implementación de los tests de aceptación

- ✧Manejando el estado
  - ✧Recomendaciones
    - ✧evitar data dumps de producción para inicializar la base de datos de tests
    - ✧mantener un set mínimo, coherente y controlado de datos
      - ✧que permita explorar el comportamiento del sistema
    - ✧punto de comienzo conocido
    - ✧colección de scripts guardados en el VCS que se aplican al comienzo de la ejecución de los tests
    - ✧debería usarse la API pública de la aplicación para poner la aplicación en el estado correcto
      - ✧evitar correr los scripts directamente contra la base de datos

Implementación de los tests de aceptación

- Manejando el estado
  - Recomendaciones
    - el test ideal debería ser atómico
      - crear lo que necesita para su ejecución y luego “limpiar” y revertir la aplicación al estado anterior
    - tratar de usar características de la aplicación para aislar los tests, por elemplo
      - usuarios, areas de trabajo, particiones, etc
    - If you can’t avoid sharing state between test cases
      - Design them carefully
      - Make test very protective
      - Assert initial state

Implementación de los tests de aceptación

- Encapsulación y testing
  - los test de aceptación ponen un presión en hacer el diseño mejor
  - si hay que romper el encapsulamiento para poder acceder a información para el test, entonces se está perdiendo un diseño mejor
  - No debería tener que crearse código específico para que los tests puedan acceder a estado interno de la aplicación

  - Resistir la tentación de crear
    - Backdoors for testing
    - Code for testing mode
  - Se pueden usar magic values?
    - Solo para simular cosas que no se controlan

Implementación de los tests de aceptación

- Asincronía y timeouts
  - En los unit tests se debe evitar los tests asincronos, cada parte se debe testear por separado, esto no es posible en los tests de aceptación
  - Problema "¿ha fallado nuestro test o estámos aún esperando la respuesta?

```
[Test]
public void ShouldSendEmailOnFileReceipt() {
  ClearAllFilesFromInbox();
  DropFileToInbox();
  ConfirmEmailWasReceived();
}
```

```
// THIS VERSION WON'T WORK
private void ConfirmEmailWasReceived() {
  if (!EmailFound()) {
    Fail("No email was found");
  }
}
```

Implementación de los tests de aceptación

- Debemos “esperar”
  - ¿Cuánto tiempo?
  - El test se hace más lento

```
[Test]
public void ShouldSendEmailOnFileReceipt() {
  ClearAllFilesFromInbox();
  DropFileToInbox();
  ConfirmEmailWasReceived();
}
```

```
private void ConfirmEmailWasReceived() {
  Wait(DELAY_PERIOD);

  if (!EmailFound()) {
    Fail("No email was found in a sensible time");
  }
}
```

Implementación de los tests de aceptación

- Tratar de minimizar las esperas

```
[Test]
public void ShouldSendEmailOnFileReceipt() {
  ClearAllFilesFromInbox();
  DropFileToInbox();
  ConfirmEmailWasReceived();
}
```

```
private void ConfirmEmailWasReceived() {
  TimeStamp testStart = TimeStamp.NOW;
  do {
    if (EmailFound()) {
      return;
    }
    Wait(SMALL_PAUSE);
  } while (TimeStamp.NOW < testStart + DELAY_PERIOD);
  Fail("No email was found in a sensible time");
}
```

Implementación de los tests de aceptación

- Los test de aceptación se deben ejecutar en un entorno como el de producción
- No debería incluir integración con sistemas externos
- El entorno debería ser controlable
- Es debir poder controlar el estado de los tests
- Por otro lado queremos testear la integración con otros sistemas, hay una tensión en qué hacer
  - Crear tests doubles para los sistemas externos
  - Pero también tener tests suites pequeños y específicos para testear los puntos de integración, que deberán correrse en un entorno contra los sistemas reales
- Usar tests doubles permite
  - controlar el estado
  - controlar el comportamiento
  - simular fallas de comunicación
  - simular respuestas de error
  - etc

Implementación de los tests de aceptación

- Un buen diseño debería minimizar el acoplamiento entre la aplicación y el sistema externo
- Generalmente un solo componente de nuestra aplicación maneja la comunicación con un sistema externo
  - Un adapter o gateway por sistema externo
  - Permite implementar patrones como circuit breaker

- Desacoplar nuestro Sistema del Sistema externo agrega una capa donde Podemos agregar control
  - Retries
  - Circuit breaking
  - Timeouts
  - etc

Implementación de los tests de aceptación

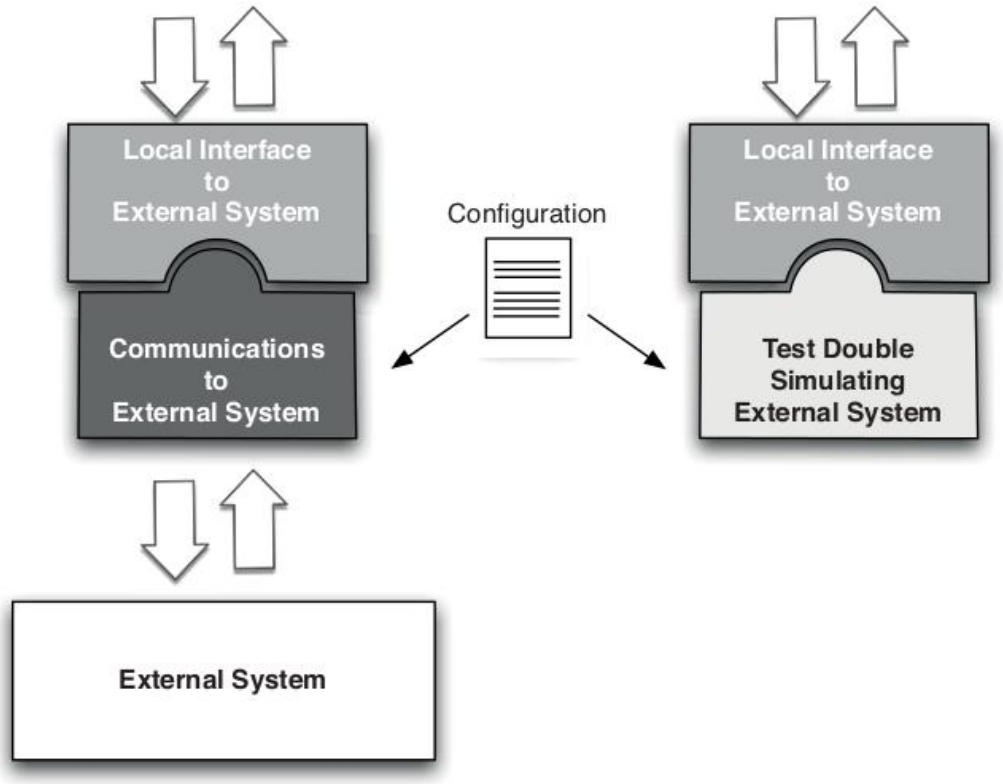


Figure 8.4 *Test doubles for external systems*

Implementación de los tests de aceptación

- Testear los puntos de integración externos
  - No testear el sistema externo
  - Testear la integración
- Testar puntos de interacción
  - Dependerán del estado del sistema externo
    - maduro en producción
    - o bajo desarrollo activo
  - Solo testear lo necesario para la interacción
    - no enfocarse en testear la interfaz del sistema externo completo
  - Adoptar un approach empírico, crear tests que cubran los problemas que se van descubriendo

Deployment tests

- El entorno para los aceptance tests debe ser lo más parecido a producción que se pueda
  - En lo posible idénticos
- Esto es una excelente oportunidad para testear los instaladores en un entorno como el de producción
- Además se debe testear que los componentes se pueden comunicar correctamente
- Deberían sumarse como otro tests suite al principio que hagan que el stage del pipeline falle si estos fallan de lo contrario puede que todos los tests esperen el máximo timeout antes de fallar

Preguntas