

Auditoría, calidad y gestión de sistemas de información - Notas

Francesco Lorenzoni

Febrero 2025

Contents

1 Calidad	5
1.1 ¿Qué entendemos por “sistema de información”	5
1.2 Calidad	5
1.2.1 Modelos de Calidad	5
1.2.2 Métricas	5
1.3 Requisitos	6
1.3.1 Matrices de trazabilidad	6
1.4 Gestión de la Calidad	6
1.4.1 ISO 9001	7
1.4.1.1 Quality control and assurance	8
1.4.1.2 SPI - Software Process Improvement	8
1.4.2 Coste de la calidad	9
2 Estrategias de Testing	11
2.1 Tipos de Testing	11
2.1.1 Testeo Unitario	11
2.1.1.1 JUnit	12
2.1.1.1.1 Test Suite	12
2.1.1.2 Testeo Parametrizado	13
2.1.2 Testeo de Integración	14
2.1.2.1 Top-Down	15
2.1.2.2 Mockito	16
2.2 Testeo de Sistema	16
2.3 Testeo de aceptación	17
2.3.1 Regresión	17
2.4 Gestión de Defectos, métricas y mas	17
2.4.1 Métricas	17
2.4.1.1 Cobertura	18
2.4.2 Mutación	18
2.4.3 Organización del testing	18
3 Diseño de casos de prueba	19
3.1 Clases de equivalencia	19
3.1.1 Coverage criterion	20
3.2 Design test cases	21
3.3 Tablas de decisión	21
3.3.1 Criterio de cobertura	22
3.4 Valores Límites	22
3.5 Matriz del testeo de dominio	23
3.5.1 Money class ejemplo	24
3.6 Máquinas de estado	24
4 Metodologías de Desarrollo	27
4.1 Definiciones	27
4.2 Metodologías	28
4.3 Metodologías modernas	29
4.3.1 Test Driven Development (TDD)	30

Chapter 1

Calidad

1.1 ¿Qué entendemos por “sistema de información”

Definition 1.1 (Sistema de información) *Conjunto único de hardware, software, bases de datos, telecomunicaciones, personas y procedimientos configurado para recolectar, manipular, almacenar y procesar datos para convertirlos en información*

Entonces un sistema de información necesita de una **entrada** de datos, un **procesamiento** de los mismos y una **salida** de información.

1.2 Calidad

Definition 1.2 (Calidad - 1) *La calidad para Pressman (1998) es el cumplimiento con:*

- ◊ *los requerimientos funcionales y de rendimiento explícitamente establecidos,*
- ◊ *los estándares de desarrollo explícitamente documentados*
- ◊ *con las características implícitas que se esperan de todo software desarrollado profesionalmente.*

Definition 1.3 (Calidad - 2) *Según las standards ISO e IEEE la calidad es El grado con el que un sistema, componente o proceso cumple con los requisitos especificados y las necesidades o expectativas del cliente o usuario.*

Definition 1.4 (Calidad - 3) *Según la ISO 91260, la calidad es el conjunto de propiedades La totalidad de características de un producto de software que tienen como habilidad, satisfacer necesidades explícitas o implícitas.*

1.2.1 Modelos de Calidad

Los modelos de calidad son herramientas que permiten evaluar la calidad de un producto o servicio. Ellos apuntar a identificar características estándar relacionadas con la calidad del software a través de atributos de calidad. Atributos de calidad incluyen:

- ◊ Adeguación Funcional
- ◊ Seguridad
- ◊ Fiabilidad (reliability)
- ◊ Usabilidad
- ◊ Eficiencia
- ◊ Mantenibilidad
 - Reparabilidad
 - Adaptabilidad
 - Portabilidad

En relación con Mantenibilidad, el OPEN/CLOSED principle dice que un software debe estar abierto para extensión pero cerrado para modificación.

- ◊ Compatibilidad

Estos atributos pueden cambiar según los modelos

En general los atributos pueden ser esternos o internos. Los primeros derivados de la relación entre el entorno y el sistema (para ello, el proceso o el sistema debe ejecutarse), e.g. reliability, robustness, usability. Los segundos derivados directamente de la descripción del producto o del proceso.

1.2.2 Métricas

Es necesario desarollar métricas de calidad, que deben ser:

- ◊ Simples y fáciles de usar
- ◊ Empírica e intuitivas
- ◊ Consistente y objetivas

Por ejemplo, centrémonos en la mantenibilidad. Podemos medirla con las siguientes métricas:

- ◊ **Aclopamiento** - Degree of interdependence between modules. High coupling is bad, it means that a change in one module will affect many other modules.
- ◊ **Cohesión** - Degree to which the elements of a module belong/work together to fulfill a single well-defined purpose. High cohesion is good, it means that a module has a single responsibility and is easier to maintain. Low cohesion means that elements are loosely related and serve multiple purposes
- ◊ Complejidad Ciclomática de McCabe
- ◊ Código Chum
- ◊ Code Coverage
- ◊ Código Muerto
- ◊ Duplicación de Código

1.3 Requisitos

Los requisitos son fundamentales en el software, y pueden ser utilizados para medir la calidad de este. Pero es importante notar que es necesario poder verificar si los requisitos están satisfechos con la implementación. Además, necesitamos también algunos controles sobre los requisitos, como complejidad, consistencia, completitud, corrección, claridad, verificabilidad, rastreabilidad, prioridad, viabilidad, flexibilidad, no ambigüedad, no redundancia, no contradicción, no vaguedad, no sobre-especificación, no sub-especificación.

Requisitos no funcionales son más fáciles de verificar que los funcionales, porque son más objetivos.

Definition 1.5 (Requisito no funcional verificable) *Una frase que incluye alguna medida que puede ser objetivamente probada.*

1.3.1 Matrices de trazabilidad

	Test 1	Test 2	Test 3	Test 4
Req 1		x		
Req 2			x	
Req 3	x		x	
Req 4				x

Figure 1.1: Matriz de trazabilidad

Matrices de trazabilidad son herramientas que permiten verificar la trazabilidad de los requisitos. Resulta fundamental que la trazabilidad siempre esté actualizada y refleje la realidad del proyecto en tiempo real

- ◊ Controlar cambios en requisitos
- ◊ Ayuda a encontrar inconsistencias
- ◊ Verificación de requisitos
- ◊ Mejor gestión de requisitos
- ◊ Asegurar eficiencia y calidad para el sistema

Hay muchas matrices en las slides, algunas relacionan requisitos con casos de uso, otros con pruebas, otros con componentes del sistema.

Ejercicio 1 / Análisis Requisitos

Javier dice cosas correctas. Aparte de las cosas que mencionó, se puede ver cómo no hay muchos números en los requisitos; el documento dices que es necesario hacer backups, pero, ¿cuantos backups? ¿con qué frecuencia? ¿con qué software?

1.4 Gestión de la Calidad

La calidad del proceso contribuye a la calidad del producto, y la calidad del producto contribuye a la calidad en uso.

- ◊ **Producto:** entregado al cliente

- ◊ **Proceso:** conjunto de actividades que se realizan para producir un producto

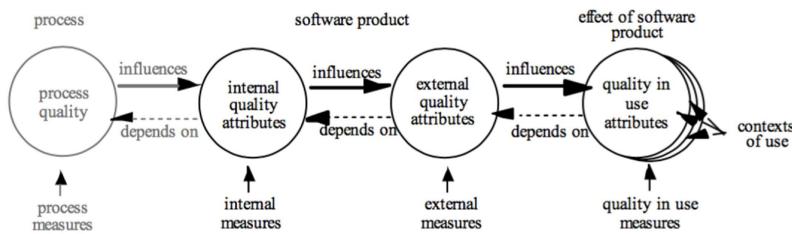


Figure 1.2: Calidad de proceso y producto

El producto como el proceso requieren recursos y la calidad también depende de la calidad de los recursos utilizados

Definition 1.6 (SGC) *Un Sistema de gestión de calidad es un conjunto de elementos relacionados entre sí para la planificación, la coordinación y la ejecución de acciones que fomenten la mejora continua en una organización, orientados a alcanzar la calidad de producto o servicio.*

Un SGC está formado por un conjunto de políticas, procesos y procedimientos documentados y está diseñado para asegurar que los productos o servicios desarrollados por la organización donde se implementa cumplen con los requisitos de calidad y con los exigentes estándares de un mercado específico.

Hay algunos pasos importantes en la gestión de la calidad:

- ◊ Preparación y utilización de **recursos**
 - Humanos
 - Económicos
 - Infraestructura
 - Conocimientos y experiencia
- ◊ Documentación y evaluación de **Procesos**
 - Estratégicos
 - Operativos
 - Soporte
- ◊ Establecer **Políticas de trabajo** y calidad

Una política de trabajo de calidad es una manifestación que realiza una empresa acerca de cómo actúa y cuáles son las pautas o reglas que establecen en el día a día para trabajar que le lleven a mejorar continuamente y hacer las cosas bien a la primera.

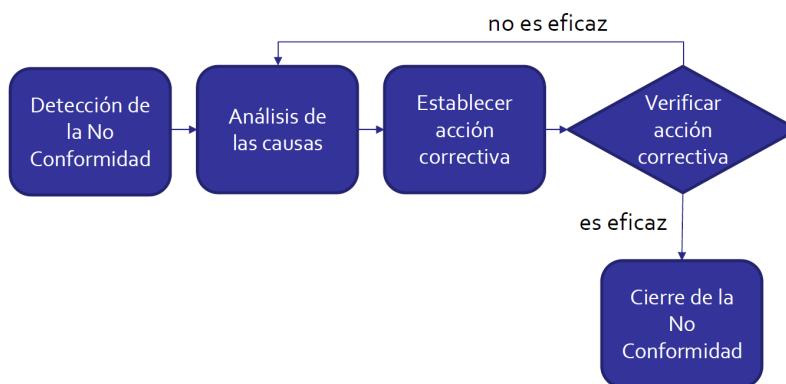


Figure 1.3: Política de calidad

Una parte crítica de la política de calidad es detectar y analizar los problemas en una empresa y los resultados no alcanzados

◊ **Objetivos**

“Conseguir una productividad del 100% de los desarrolladores” o “Aumentar la satisfacción de los cliente” son objetivos mal planteados.

1.4.1 ISO 9001

La ISO 9001 es una norma internacional que especifica los requisitos para un sistema de gestión de la calidad (SGC). Las organizaciones utilizan la norma para demostrar la capacidad para proporcionar productos y servicios que cumplen

con los requisitos del cliente y los reglamentarios aplicables. El principal objetivo de la ISO 9001:2015 es lograr que una compañía consiga la satisfacción del cliente mediante el establecimiento de procesos de mejora continuada dentro de la misma.

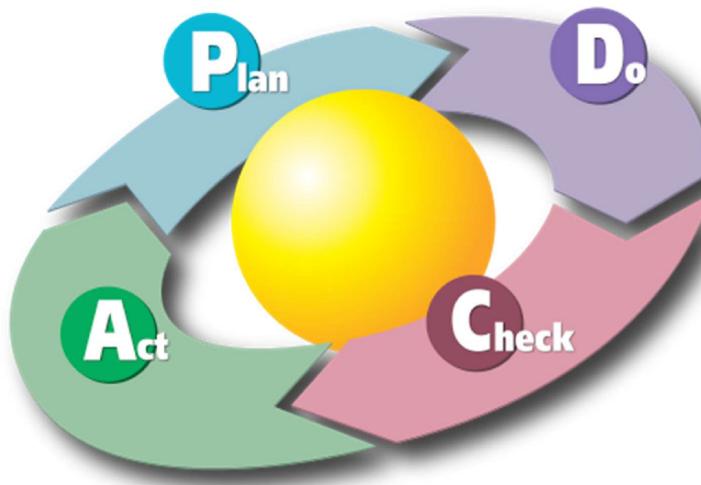


Figure 1.4: PDCA plan

Propone también un método de **mejora continua**, el PDCA (Plan, Do, Check, Act).

1.4.1.1 Quality control and assurance

- ◊ **Quality Control:** “A part of quality management focused on fulfilling quality requirements”
Parte de la gestión de calidad de software que comprueba que el proyecto sigue sus estándares, procesos y procedimientos, y que el proyecto produce los productos (entregables) internos y externos requeridos.
El principal propósito del control de calidad es asegurar que el producto satisface los requisitos mediante pruebas y revisiones de los requisitos funcionales y no funcionales. El control de calidad aprovecha las pruebas e inspecciones integrales para monitorear la calidad de la producción de un fabricante.
- ◊ **Quality Assurance:** “A part of quality management focused on providing confidence that quality requirements will be fulfilled”
Parte de la gestión de calidad de software que asegura que los estándares, procesos y procedimientos son apropiados para el proyecto y se han implementado correctamente.

El aseguramiento de la calidad incluye aquellas actividades planificadas y sistemáticas necesarias para aportar la confianza de que el software satisfará los requisitos dados de calidad. Esas actividades aseguran que el proceso del software y productos cumplen los requerimientos, estándares, y procedimientos:

- ◊ los *procesos* incluyen todas las actividades involucradas en el diseño, codificación, pruebas y mantenimiento;
- ◊ los *productos* incluyen software, datos asociados, documentación, y toda la documentación para soporte y reportes.

Los ingenieros del software realizan el trabajo técnico. En cambio, un grupo de **SQA** (*Software Quality Assurance*) se responsabiliza en la planificación de aseguramiento de la calidad, supervisión, mantenimiento de registros, análisis e informes.

El principal propósito del control de calidad es asegurar que el producto satisface los requisitos mediante **pruebas y revisiones** de los requisitos funcionales y no funcionales.

1.4.1.2 SPI - Software Process Improvement

Los **Estándares de Calidad** son normas establecidas que definen las mejores prácticas, guías o requisitos que un proceso, producto o sistema debe cumplir para garantizar la Calidad.

Los **Modelos de Madurez** establecen marcos de referencia que evalúan la capacidad de una organización para desarrollar productos o servicios de calidad de manera consistente y predecible; van a evaluar la madurez de los procesos de desarrollo de software o sistemas.

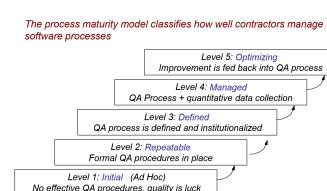


Figure 1.6: Capability Maturity CMMI

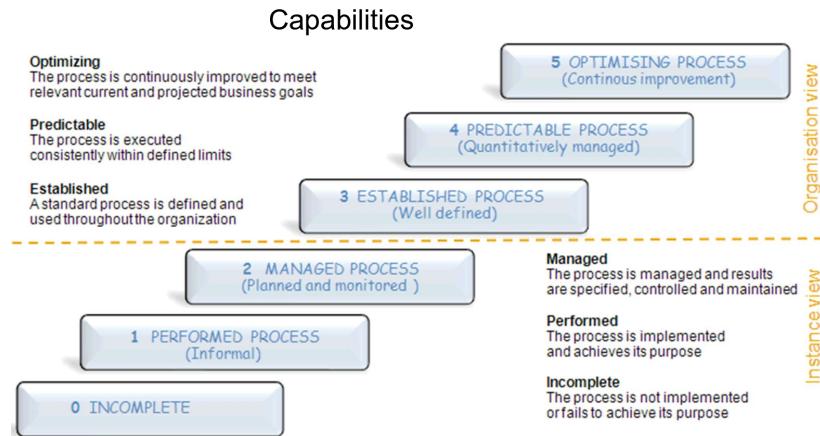


Figure 1.5: Spice Model for Process Improvement

Complaints about Models

- ◊ No siempre se ven como tan relevantes y actuales por los ingenieros del software
- ◊ Pueden conllevar mucha burocracia
- ◊ Pueden requerir trabajo manual tedioso si no tienen herramientas software de soporte

Entonces, *to effectively apply standards, limit overhead!*

1.4.2 Coste de la calidad

$$\text{Coste de la calidad} = \text{Coste de conformidad} + \text{Coste de no conformidad} \quad (1.1)$$

Definition 1.7 (Coste de conformidad) *costes de las actividades para:*

- ◊ *Valoración/Estimación: detección de defectos (testeo)*
- ◊ *Prevención: de errores (aseguramiento de calidad, etc)*

Definition 1.8 (Coste de no conformidad) *costes de las actividades para:*

- ◊ *Corrección: de defectos (reparación de errores)*
- ◊ *Fallos: en el producto (retrabajo, etc)*
- ◊ *Pérdida de clientes*
- ◊ *Costes de juicios*
- ◊ *etc.*

Boehm's law

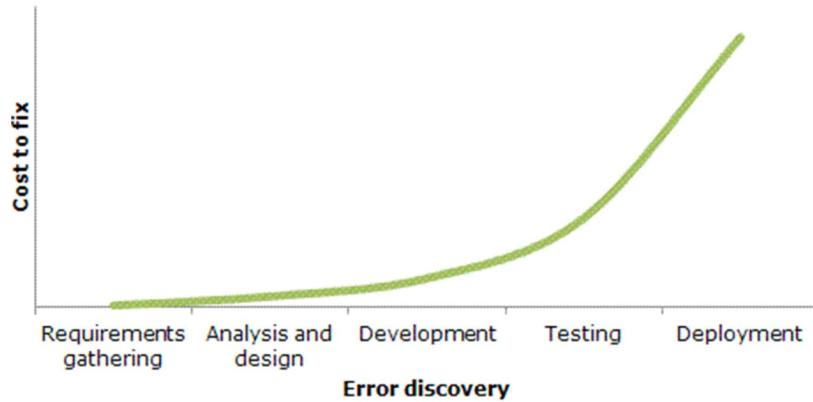


Figure 1.6: Segundo Boehm, el coste de corregir un error aumenta exponencialmente con el tiempo. Entonces, el **testing** es un momento crítico en el desarrollo de software, porque permite de encontrar errores antes de que el coste de corregirlos sea más alto.

Chapter 2

Estrategias de Testing

“ Probar programas sirve para demostrar la presencia de errores, pero nunca para demostrar su ausencia ” —
Edsger W. Dijkstra

El proceso de testing dentro de un proyecto de software es una actividad que tiene como objetivo verificar y validar que el software cumple con los requisitos especificados.

El proceso de testing conlleva:

- ◊ planificación,
- ◊ diseño,
- ◊ ejecución de pruebas
- ◊ evaluación

La gestión de calidad debe garantizar que el proceso de testing se lleve a cabo de manera eficiente y eficaz.

2.1 Tipos de Testing

- ◊ **Testeo unitario**
Testeo de unidades de software individuales
- ◊ **Testeo de integración**
Testeo de los interfaces y de la interacción entre las unidades previamente testeadas
- ◊ **Testeo de sistema**
Testeo del sistema entero
- ◊ **Testeo de aceptación**
Testeo del sistema y los criterios de aceptación previamente establecidos con el cliente

2.1.1 Testeo Unitario

Las **unidades** son Clases/Métodos (en OO), Procedimientos, Módulos o Componentes. Como se definen las unidades depende del diseño, de la criticidad del software (más crítico implica unidades más pequeñas), de la empresa (estrategia), o del tiempo disponible.

Una **prueba unitaria (unit test)** es una pieza de código escrita por un desarrollador que pone a prueba un pequeño y específico trozo de código. Es una manera relativamente barata para mejorar la calidad del código producido, pero *NO* es para usuarios, gerentes, jefes de proyectos: está hecha para los programadores. El programador examina o ejecuta una pequeña parte de su código, para testear la funcionalidad que él o ella piensa que tiene que tener

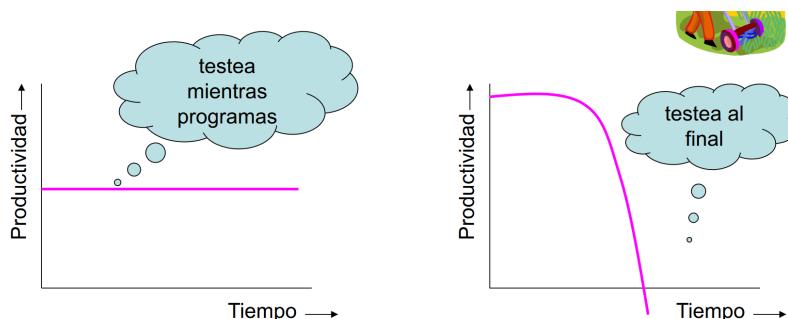


Figure 2.1: Posponer el testeo unitario cuesta mucho tiempo

“No es mi trabajo hacer testeo, tenemos un departamento de calidad” — Bad programmer

Entonces ¿cuál es el trabajo de un programador? El trabajo de cada programador es producir código que funciona y que no contenga “muchos” errores

2.1.1.1 JUnit

JUNIT es un framework de testing para Java. JUNIT es una herramienta que ayuda a los programadores a escribir pruebas unitarias en Java. JUNIT ha sido importante en el desarrollo de la metodología de programación extrema (XP). @Test son los métodos de prueba.

- ◊ @BeforeEach se invoca antes de la ejecución de cada test
- ◊ @AfterEach se invoca después de la ejecución de cada test
- ◊ @BeforeAll se invoca antes de la ejecución de todos los tests
- ◊ @AfterAll se invoca después de la ejecución de todos los tests

Estas anotaciones son de JUNIT 5, en JUNIT 4 se usaba @Before y @After, además de otras diferencias.

El código de testo para una clase `MyClass` se pone en una clase `TestMyClass`, y el método para testear el método `myMethod` se llama `testMyMethod`, o algo similar, como `testMyMethodSimple`, `testMyMethodAll`, etc.

En el ejemplo anterior, se muestra un test de una clase `Account` que tiene una conexión a una base de datos. Se puede ver que se crea una conexión a la base de datos en el método `setUpBeforeAll` y se cierra en el método `tearDownAfterAll`. Además, se crea una instancia de la clase `Account` en el método `setUp` y se destruye en el método `tearDown`.

```
public class TestDB {
    static private Connection dbConn;
    static private Account acc;
    @BeforeAll
    public static void setUpBeforeAll(){
        dbConn = new Connection("Oracle",15,fred,
                               "f");
        dbConn.connect();
    }
    @AfterAll
    public static void tearDownAfterAll(){
        dbConn.disconnect();
        dbConn = null;
    }
    @BeforeEach
    protected void setUp(){
        acc = new Account();
    }
    @AfterEach
    protected void tearDown(){
        acc = null;
    }

    public void testAccountAccess(){
        .../Uses dbConn and acc
    }
    public void testEmployeeAccess(){
        .../Uses dbConn and acc
    }
}
```

Los métodos son ejecutados en el siguiente orden:

- ◊ `setUpBeforeAll`
 - `setUp`
 - `testAccountAccess`
 - `tearDown`
 - `setUp`
 - `testEmployeeAccess`
 - `tearDown`
- ◊ `tearDownAfterAll`

2.1.1.1.1 Test Suite Un test suite es una combinación de clases de prueba que permite ejecutar todas las pruebas contenidas en dichas clases a la vez en un orden establecido. Para crear una Test suit se debe:

- ◊ Crear una clase Java
- ◊ Anotar la clase con la etiqueta `@RunWith(JUnitPlatform.class)`
- ◊ Indicar las clases que forman parte de la suite con la etiqueta `@SelectClasses`, o `@SelectPackages`

A continuación, se muestra un ejemplo de código de la test suite `AllTests` que agrupa las pruebas de las clases `LargestTest` y `MiClaseDeTest`.

```
import org.junit.platform.runner.JUnitPlatform;
import org.junit.platform.suite.api.SelectClasses;
import org.junit.runner.RunWith;

@RunWith(JUnitPlatform.class)
@SelectClasses({ LargestTest.class, MiClaseDeTest.class })
public class AllTests { }
```

Cuando lancemos a ejecución el test suite `AllTests` se ejecutarán primero las pruebas de la clase `LargestTest`, y a continuación las pruebas de la clase `MiClaseDeTest`.

2.1.1.2 Testeo Parametrizado Los test parametrizados posibilitan ejecutar un test con diferentes argumentos. Estos tests son declarados como los tests básicos (`@Test` methods) pero usan la etiqueta `@ParameterizedTest`.

Es necesario declarar al menos una fuente que proporcionará los argumentos para cada invocación del método. Esto se puede hacer con la anotación `@ValueSource` y un array de valores literales, pero existen también otras formas de hacerlo, por ejemplo, con valores de un fichero csv utilizando la anotación `@CsvFileSource`.

```
@ParameterizedTest
@ValueSource (strings = { "racecar", "radar", "able was I ere I saw elba" } )
void palindromesTest (String candidate) {
    assertTrue (StringUtils.isPalindrome (candidate) );
}
```

Listing 2.1: Ejemplo de clase de test de Factorial

Listing 2.1: Ejemplo de clase Factorial

```
public class Factorial {
    public static int factorial(int n) {
        if (n < 0) {
            throw new IllegalArgumentException("n
                                         must be nonnegative");
        }
        int result = 1;
        for (int i = 1; i <= n; i++) {
            result *= i;
        }
        return result;
    }
}
```

```
public class FactorialTest {
    @Test
    public void testFactorialOfZero() {
        assertEquals(1, Factorial.factorial(0));
    }

    @Test
    public void testFactorialOfOne() {
        assertEquals(1, Factorial.factorial(1));
    }

    @Test
    public void testFactorialOfPositiveNumber() {
        assertEquals(24, Factorial.factorial(4));
    }

    @Test(expected = IllegalArgumentException.
          class)
    public void testFactorialOfNegativeNumber() {
        Factorial.factorial(-1);
    }
}
```

```
@RunWith(Parameterized.class)
public class FactorialParametrizedTest {
    private int input;
    private int expected;

    public FactorialParametrizedTest(int input, int expected) {
        this.input = input;
        this.expected = expected;
    }

    @Parameterized.Parameters
    public static Collection<Object[]> data() {
        return Arrays.asList(new Object[][] {
            { 0, 1 }, { 1, 1 }, { 2, 2 }, { 3, 6 }, { 5, 120 }, { 9, 362880 },
            { 12, 479001600 }, { -1, 1 }
        });
    }

    @ParameterizedTest
    @MethodSource("data")
    public void testFactorial() {
        assertEquals(expected, Factorial.factorial(input));
    }
}
```

}

2.1.2 Testeo de Integración

Definition 2.1 (Testeo de Integración) *El testing de las interfaces y la interacción entre las unidades previamente testeadas mientras que se ensambla el sistema entero*

- ◊ ¿Qué componentes son el foco del testeo de integración?
- ◊ ¿En qué orden vamos a testear las interfaces?
- ◊ ¿Qué técnicas utilizamos para testear las interfaces?

Hay tres estrategias:

1. **Big Bang** (todos los componentes a la vez)
2. **Bottom-up** (de abajo hacia arriba)
3. **Top-down** (de arriba hacia abajo)

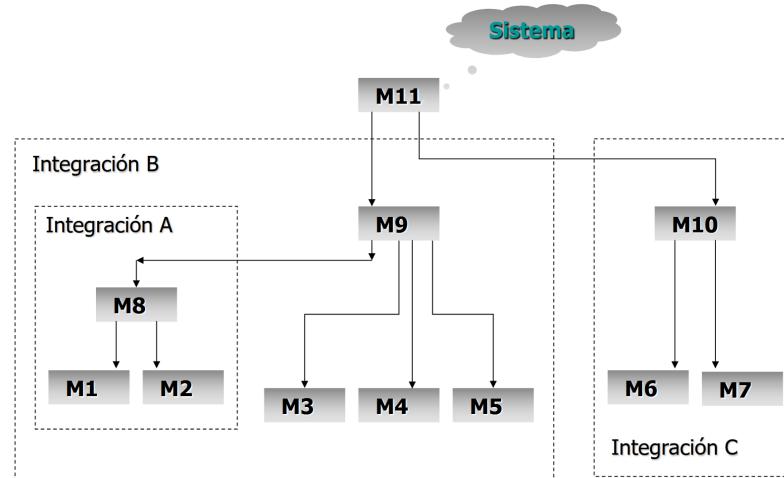


Figure 2.2: Árbol de dependencia y Bottom-up testing

Para hacer el testing *bottom-up* necesitamos **drivers**. Un driver es un programa que invoca un componente bajo testeo, por ejemplo, para simular un componente de un nivel superior cuyo código todavía no está disponible (está todavía en desarrollo).

Por ejemplo un driver puede simular el componente M9 que invoca M8, si estamos testeando M8.

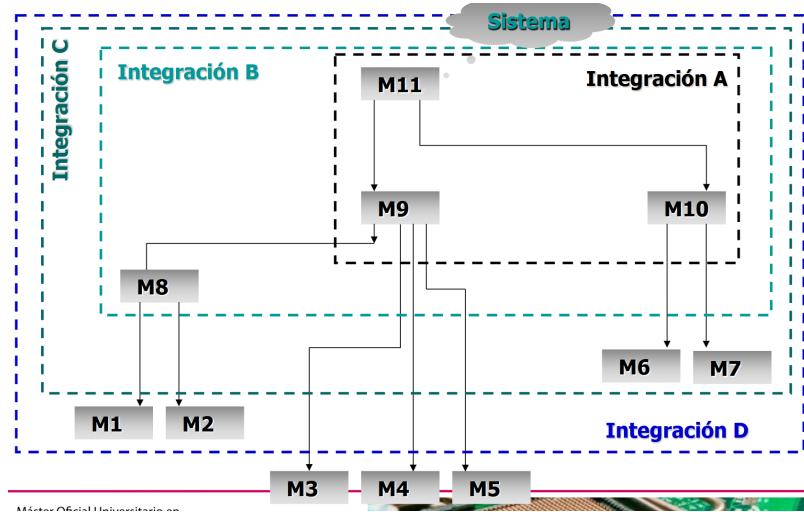


Figure 2.3: Top-down testing

Para hacer el testeo de integración de manera *top-down*, se necesita dobles, que simulan componentes de un nivel inferior ("Stub", "dummy", "fake", "mock").

Si estamos testeando M8, podemos sustituir M1 y M2 con mocks

El **Big Bang** testing se utiliza solo para sistemas pequeños, ya que es muy difícil de manejar en sistemas grandes.

2.1.2.1 Top-Down

```

public float calculaSalarioNeto(float salarioBruto) throws BREException {
    float retencion = 0.0f;
    if (salarioBruto < 0) {
        throw new BREException("El salario bruto debe ser positivo");
    }
    // PROXY HERE!
    ProxyAeat proxy = getProxyAeat();
    List<TramoRetencion> tramos;
    try {
        // Invoke the emulated function!
        tramos = proxy.getTramosRetencion();
    } catch (IOException e) {
        throw new BREException("Error al conectar al servidor de la AEAT", e);
    }
    for (TramoRetencion tr : tramos) {
        if (salarioBruto < tr.getLimiteSalario()) {
            retencion = tr.getRetencion();
            break;
        }
    }
    return salarioBruto * (1 - retencion);
}

public class MockProxyAeat extends ProxyAeat {
    boolean lanzarExcepcion;

    public MockProxyAeat(boolean lanzarExcepcion) {
        this.lanzarExcepcion = lanzarExcepcion;
    }

    @Override
    public List<TramoRetencion> getTramosRetencion()
        throws IOException {
        if (lanzarExcepcion) {
            throw new IOException("Error al conectar al servidor");
        }
        List<TramoRetencion> tramos = new ArrayList<TramoRetencion>();
        tramos.add(new TramoRetencion(1000.0f, 0.0f));
        tramos.add(new TramoRetencion(1500.0f, 0.16f));
        tramos.add(new TramoRetencion(Float.POSITIVE_INFINITY, 0.18f));
        return tramos;
    }
}

```

Listing 2.2: Utilizar en el testing

```

class TestableEmpleado extends Empleado {
    ProxyAeat proxy;

    public void setProxyAeat(ProxyAeat proxy) {
        this.proxy = proxy;
    }

    @Override
    ProxyAeat getProxyAeat() {
        return proxy;
    }
}

TestableEmpleado empleadoTest;

@BeforeEach
public void setUpClass() {
    empleadoTest = new TestableEmpleado();
    empleadoTest.setProxyAeat(new MockProxyAeat(false));
    empleadoTestFail = new TestableEmpleado();
    empleadoTestFail.setProxyAeat(new MockProxyAeat(true));
}

```

```

}

@Test
public void testCalculaSalarioNeto1() throws BRException {
    float resultadoReal = empleadoTest.calculaSalarioNeto(2000.0f);
    float resultadoEsperado = 1640.0f;
    assertEquals(resultadoEsperado, resultadoReal, 0.01);
}

```

2.1.2.2 Mockito

Mockito es un framework de testing para Java que permite crear objetos simulados (mocks) de clases y interfaces. Mockito se utiliza para simular objetos que son necesarios para realizar pruebas unitarias. Se puede integrar con JUnit para realizar pruebas unitarias en Java. En las slides se muestran ejemplos de cómo utilizar Mockito.

Mockito es muy comodo para implementar el testeо de integraciоn de manera *top-down*.

```

public class CalculatorServiceImpl implements CalculatorService {
    private DataService dataService;

    public double calculateAverage() {
        int[] numbers = dataService.getListOfNumbers();
        double avg = 0;
        for (int i : numbers) {
            avg += i;
        }
        return (numbers.length > 0) ? avg / numbers.length : 0;
    }

    public void setDataService(DataService dataService) {
        this.dataService = dataService;
    }
}

@RunWith(MockitoJUnitRunner.class)
public class CalculatorServiceTest {
    // Automatically set data service to point to mock
    @InjectMocks
    private CalculatorServiceImpl calculatorService;
    @Mock
    private DataService dataService;

    @Test
    public void testCalculateAvg_simpleInput() {
        // define what to yield when the mock methods are invoked
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1, 2, 3, 4, 5 });
        assertEquals(3.0, calculatorService.calculateAverage(), .01);
    }

    @Test
    public void testCalculateAvg_emptyInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] {});
        assertEquals(0.0, calculatorService.calculateAverage(), .01);
    }

    @Test
    public void testCalculateAvg_singleInput() {
        when(dataService.getListOfNumbers()).thenReturn(new int[] { 1 });
        assertEquals(1.0, calculatorService.calculateAverage(), .01);
    }
}

```

2.2 Testeo de Sistema

Se verifica que se cumple los requisitos especificados, es decir, que el sistema realiza correctamente todas las funciones que se han detallado en las especificaciones dadas por el usuario del sistema.

Es necesario automatizar las pruebas, y por esto se utilizan herramientas como *Selenium*, que permite automatizar pruebas en navegadores web: en este caso, hablamos de pruebas de **funcionalidad**.

Para testar el **rendimiento** (tiempo de respuesta, carga, memoria.) se utilizan herramientas como *JMeter* o *NeoLoader*.

Otros aspectos importantes que el testeo debe cubrir son **Seguridad** y **Usabilidad**.

- ◊ Evaluar la capacidad del sistema de software de prevenir acceso no autorizado.
- ◊ Casos de test Npicos examinan funciones como:
 - logon, logoff
 - cambio de permisos
 - chequeo de contraseñas
 - caducidad de contraseñas
- ◊ Usabilidad es el conjunto de características que influyen en el esfuerzo requerido para el aprendizaje, el uso, la preparación de entradas y la interpretación de salidas del programa por parte de un conjunto de usuarios dados
 - Memorabilidad (el esfuerzo del usuario para comprender el software una vez ya había aprendido a usarlo)
 - Facilidad de aprendizaje (el esfuerzo del usuario para aprender el software)
 - Operabilidad/eficiencia (el grado en que el usuario puede ejecutar las tareas que son objeto del software)
 - Eficacia (el grado en que el usuario comete errores)
 - Atractivo (ergonomía, colores, formas,)
- ◊ La mejor manera da evaluar si un producto es usable es involucrandolo a los usuarios, que son los que deciden si el producto es usable o no.

2.3 Testeo de aceptación

Esto testeo es dirigido a los criterios de aceptación previamente establecidos con el cliente. Se puede hacer de manera manual o automatizada.

Es importante recordar dos axiomas:

- ◊ Axioma de **decomposición**: Aunque se ha testeado un sistema adecuadamente, eso no significa que se ha testeado adecuadamente sus componentes.
- ◊ Axioma de **anticomposición**: Testear cada uno de los componentes de un sistema por separado no necesariamente significa que el sistema entero está testeado adecuadamente.

2.3.1 Regresión

Testeo que se necesita hacer después de cambios en el software para asegurar que no se ha introducido defectos.
El testeo de regresión se tiene que automatizar por el simple hecho de que el testeo de regresión manual *NO SE HACE*.

2.4 Gestión de Defectos, métricas y mas

El proceso de clasificación de defectos incluye tres fases:

1. **Detección** de defectos
2. **Investigación** de defectos
3. **Resolución** de defectos

Para cada defecto tenemos que registrar algunas informaciones.

- ◊ **Actividad** (que estábamos haciendo cuando encontramos el fallo)
- ◊ **Fase del proyecto** (en que estábamos cuando encontramos el fallo)
- ◊ **Repetitividad** (¿se puede repetir el error?)
- ◊ **Síntoma** (fallos del sistema, mensaje de error, entrada no aceptada, resultado incorrecto, etc.)
- ◊ **Causa** (nuestro producto, componente externo, usuario, etc.)
- ◊ **Origen** (especificación de requisitos, diseño, programación, etc.)
- ◊ **Impacto** (misión: crítico, medio, ...; en la planificación; para el cliente, etc.)

2.4.1 Métricas

Las métricas son observaciones cuantitativas para:

- ◊ Informar sobre el **progreso del proyecto** de testeo
 - ¿Qué tareas han terminado en tiempo?
 - ¿Qué tareas han terminado antes?
 - ¿Qué tareas han tenido retraso?
 - ¿Cómo vamos siguiendo la planificación?
 - Si no vamos bien, ¿cuáles son las razones?
 - ¿Qué productividad ha tenido persona/equipo X?

- ◊ Informar sobre la **calidad del software**
 - ¿Podemos parar el testeo?
 - ¿Podemos entregar producto?
 - ¿Hemos resueltos todos los defectos?
 - ¿Cómo estamos gestionando los defectos?
 - ¿Cuántos defectos hemos encontrado por: subsistema, origen, causa, severidad, etc...?
- ◊ Informar sobre la **calidad del testeo**
 - ¿Estamos haciendo los tests necesarios?
 - ¿Están siendo efectivos los tests?
 - ¿Estamos utilizando los casos de prueba adecuados?
 - ¿Necesitamos tener en cuenta diferentes casos de prueba?

2.4.1.1 Cobertura

La cobertura del testeo es una métrica importante para evaluar la calidad del testeo. La cobertura del testeo es la medida de la cantidad de código que ha sido ejecutado por los tests, especialmente en lo que se refiere a **instrucciones, decisiones y condiciones** (múltiples).

Pero puede referirse también a la cobertura de los requisitos, de los casos de uso, de los casos de prueba, etc.

Listing 2.3: Con 1 test donde `x==y` se cubre todo el código, pero no todas las decisiones. En efecto, el defecto ocurre cuando `x!=y`

```
int foo_3 () {
    int* p = NULL;
    int x;
    if (x==y) {
        p = &x;
    }
    *p = 123;
}
```

- ◊ G = grafo de flujo de control
- ◊ A = cantidad de arcos en G
- ◊ N = cantidad de nodos en G
- ◊ $V(G) = A - N + 2$

Complejidad ciclomática $V(G)$ es una aproximación de la cantidad de los casos de test necesarios para cobertura de decisiones.

Está matemáticamente demostrado que 100% cobertura de decisiones se puede hacer con $\leq V(G)$ tests.

Sin embargo, se pueden elegir $V(G)$ caminos que NO consiguen 100% de cobertura de decisiones!

Se tiene que interpretar las métricas de cobertura con cuidado: cobertura alta no dice mucho, pero si es baja hay un problema.

$$PDD = \text{Porcentaje de Detección de Defectos} \quad (2.1)$$

$$PDD = \frac{\#\{\text{defectos encontrados durante el testeo}\}}{\#\{\text{defectos encontrados}\}} \times 100 \quad (2.2)$$

2.4.2 Mutación

La mutación es una técnica de testing que consiste en introducir errores en el código fuente para ver si los tests son capaces de detectarlos. Se puede utilizar para evaluar la calidad de los tests.

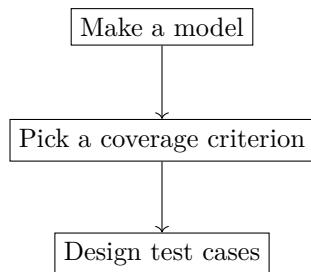
2.4.3 Organización del testing

Se puede dedicar un equipo de test integrado, con un jefe, o se puede haber que desarrolladores y testeadores son las mismas personas y no hay testeadores a tiempo completo. La ventaja de tener un equipo de testeo es que se puede tener una visión más objetiva del software, ya que los testeadores no han desarrollado el software; al contrario, si los desarrolladores hacen el testeo, no tienen que comunicar con los testeadores, y conocen el software.

Otra técnica de organización es *Outsourcing*, que consiste en contratar una empresa externa para hacer el testeo. La ventaja es que se puede tener una visión más objetiva del software, y se puede tener acceso a expertos en testeo. La desventaja es que se puede tener problemas de comunicación, y que se puede tener problemas de confidencialidad.

Chapter 3

Diseño de casos de prueba



Recuerda que el SUT (System under test) es el sistema bajo prueba.

Abajo vamos a ver algunos modelos.

3.1 Clases de equivalencia

Se modela el input domain, y para ello se utilizan clases de equivalencia. Es importante notar que las clases de equivalencia son una asunción, y no una verdad absoluta, pueden ser erróneas.

Sea R una relación de equivalencia sobre un conjunto A . Para cada $a \in A$, llamaremos clase de equivalencia de a , al conjunto formado por todos los elementos de A que estén relacionados con él a través de R .

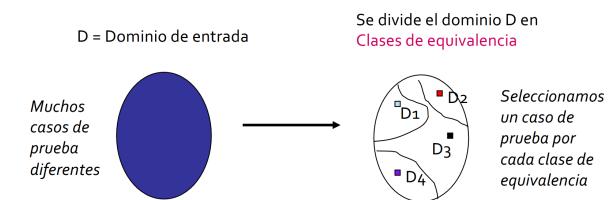


Figure 3.1: Dividir el dominio

Este modelo permite identificar conjuntos de pruebas de tamaño manejable seleccionando unos pocos casos de prueba para cada clase de equivalencia. Permite también medir la efectividad de la prueba en términos de cobertura relacionada con el modelo de partición creado. Además, el proceso de partición obliga al tester a pensar sistemáticamente sobre lo que importa.

Órdenes posibles: "cheque", "depósito", "pago factura", "retirada de fondos"		
Parámetro	Clase Válida	Clase Inválida
Código de área	1. $200 \leq \text{código} \leq 999$	2. $\text{código} < 200$ 3. $\text{código} > 999$ 4. no es un número
Clave identificativa	5. 6 caracteres exactos	6. menos de 6 caracteres 7. más de 6 caracteres
Ordenes posibles	8. cheque 9. depósito 10. pago factura 11. retirada de fondos	12. no es orden válida

Figure 3.1: Ejemplo de una aplicación bancaria

Los datos de entrada son:

- ◊ Código de area: número de tres dígitos que no empieza ni por 0 ni por 1
- ◊ Clave identificativa de la operación: 6 caracteres alfanuméricos
- ◊ Órdenes posibles: "cheque", "depósito", "pago factura" , "retirada de fondos"

Es importante enumerar las clases de equivalencia, y no mezclarlas sobre todo. Cuando se va a escribir las pruebas, cada prueba tiene que ser relacionada con una o más clases de equivalencia, y tenemos que indicar a cuales.

Parámetro	Clase Válida	Clase Inválida
Capital	$c > min$	$c \leq min$
Ingreso	$i \geq 0$	$i < 0$
Deuda	$d \geq 0$	$d < 0$
Impagados	$T \vee F$	
Impagados \wedge cond	1) $T \wedge (i - d) < c/3$ 2) $F \wedge (i - d) \geq c/3$ 3) $T \wedge (i - d) \geq c/3$ 4) $F \wedge (i - d) < c/3$	

Table 3.1: Ejemplo de clases de equivalencia para una aplicación bancaria

- ◊ si el cliente está en una lista de impagados y el sueldo bruto menos la deuda es inferior al capital solicitado dividido entre 3, la solicitud pasa a No Concedida.
- ◊ si el cliente no está en ninguna lista, y el sueldo bruto menos la deuda es mayor o igual al capital solicitado dividido entre 3, se clasifica como Pre-concedida.
- ◊ en cualquier otro caso pasa a En estudio

3.1.1 Coverage criterion

El criterio de cobertura es una forma de medir la efectividad de las pruebas. Puede ser:

- ◊ **ACoC** - All combinations coverage
para cada variable de entrada se consideran todas las clases válidas, y luego se combinan con todas las clases de las demás variables.
- ◊ **ECC** - Each choice coverage
solo se asegura que (un valor de) cada clase ocurra en al menos un caso de prueba

Reglas básicas:

1. Se pueden combinar valores de clases válidas en el mismo caso de prueba
2. No se pueden combinar valores de clases inválidas en el mismo caso de prueba. Se necesita un caso de prueba para cada clase inválida
3. Se debe intentar reducir el número de casos de prueba

All combination	Each choice	
1, 5, 8	1, 5, 8	Clases válidas
1, 5, 9	1, 5, 9	
1, 5, 10	1, 5, 10	
1, 5, 11	1, 5, 11	

All combination	Each choice	
2, 5, 8 4, 5, 8 1, 7, 8	2, 5, 10	Clases inválidas
2, 5, 9 4, 5, 9 1, 7, 9	3, 5, 9	
2, 5, 10 4, 5, 10 1, 7, 10	4, 5, 11	
2, 5, 11 4, 5, 11 1, 7, 11	1, 6, 9	
3, 5, 8 1, 6, 8 1, 5, 12	1, 7, 8	
3, 5, 9 1, 6, 9	1, 5, 12	
3, 5, 10 1, 6, 10		
3, 5, 11 1, 6, 11		

Figure 3.2: Aplicar los dos criterios de cobertura al ejemplo anterior en Fig. 3.1. Aquí se muestran las combinaciones de clases, son tripletas porque hay tres variables de entrada.

3.2 Design test cases

Test case	Código de area	Clave identificativa	Orden
1, 5, 8	300	nómina	cheque
1, 5, 9	400	viajes	depósito
1, 5, 10	500	coches	pago factura
1, 5, 11	600	comida	retirada de fondos

Table 3.2: Ejemplo: para cada tripleta que representa una combinación, elegimos un valor que pertenezca a la clase de equivalencia.

3.3 Tablas de decisión

Se puede utilizar para programas que toman decisiones basadas en condiciones lógicas sobre combinaciones de entradas, parámetros y/o variables, eligiendo diferentes acciones o respuestas.

- ◊ La acción o respuesta que se toma no depende del orden en que se evalúa los valores de los parámetros y variables.
- ◊ La acción o respuesta que se toma no depende de entradas o salidas anteriores
- ◊ Cada condición corresponde a una variable, relación
 - o predicado
- ◊ Valores posibles para las “Condition Entries”
 - Booleano (**True / False**) - Tabla de decisión de entrada limitada
 - Valores - Tabla de decisión de entrada extendida
 - Valor “No Importa” (Don’t care value)
- ◊ Cada acción es una operación que hay que ejecutar

	Rules			
	Rule 1	Rule 2	Rule 3	Rule 4
Conditions section {	Condition 1			
	Condition 2		Condition	
	Condition 3		entries	
	Condition 4			
Actions section {	Action 1		Action	
	Action 2		entries	

Figure 3.3: Tabla decision

The diagram illustrates two ways to represent the same set of 7 rules for insurance actions based on claim counts and age.

Variables de decisión:

- Conditions section:**

		1	2	3	4	5	6	7
Conditions section	num _claims	0	0	1	1	[2,4]	[2,4]	≥ 5
	age	≥ 16	> 25	≥ 16	> 25	≥ 16	> 25	≥ 16
- Actions section:**

Actions section	Incr	50	25	100	50	400	200	0
	Warning	no	no	yes	no	yes	yes	no
	Cancel	no	no	no	no	no	no	yes

7 rules (en forma columna): A bracket groups the Conditions section and Actions section tables, indicating they represent 7 separate rules.

Variables de decisión:

Conditions section:

	Conditions section		Actions section		
	num _claims	age	Increment premium	Send warning	Cancel insurance
1	0	[16, 25]	50	no	no
2	0	[25, 85]	25	no	no
3	1	[16, 25]	100	yes	no
4	1	[25, 85]	50	no	no
5	[2,4]	[16, 25]	400	yes	no
6	[2,4]	[25, 85]	200	yes	no
7	≥ 5	[16, 85]	0	no	yes

7 rules (en forma fila): A bracket groups the Conditions section and Actions section tables, indicating they represent 7 separate rules.

Figure 3.3: Forma columna y forma fila de tablas de decisión

3.3.1 Criterio de cobertura

- Un caso de test para cada variante explícita
- Un caso de test para seleccionar la acción por defecto (si hay)

3.4 Valores Límites

La técnica de valores límites es una técnica *complementaria* a las clases de equivalencia y tablas de decisión. Busca encontrar los errores que se pueden producir en los extremos de los datos de entrada, ya que el sentido común indica que los puntos cercanos a los límites pueden ser más propensos a errores.

- Intervalo:** un subconjunto del espacio de los valores de entrada de un programa.
- Punto límite** de un intervalo (*boundary*): aquel punto que según sea incrementado o decrementado un valor epsilon infinitesimal dará como resultado un valor perteneciente o no a dicho intervalo.
- Desigualdad del límite:** una expresión algebraica con $>$, $<$, \geq , \leq que define parte de los puntos que pertenecen a un intervalo o dominio.
- Condiciones de igualdad:** una expresión algebraica con $=$, \neq

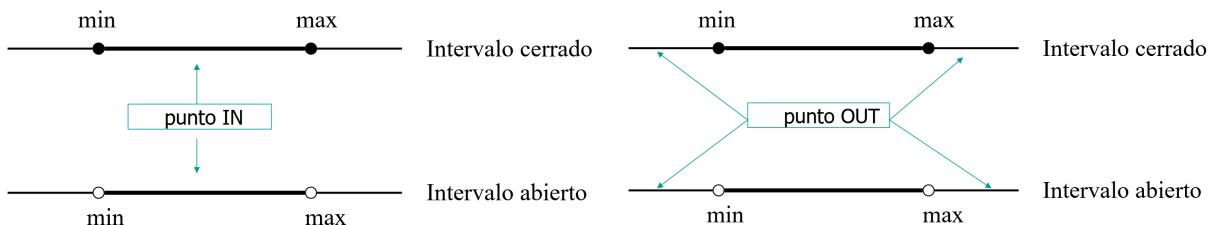


Figure 3.4: puntos IN y OUT

- PUNTO IN :** un punto que pertenece al intervalo
- PUNTO OUT :** un punto que no pertenece al intervalo

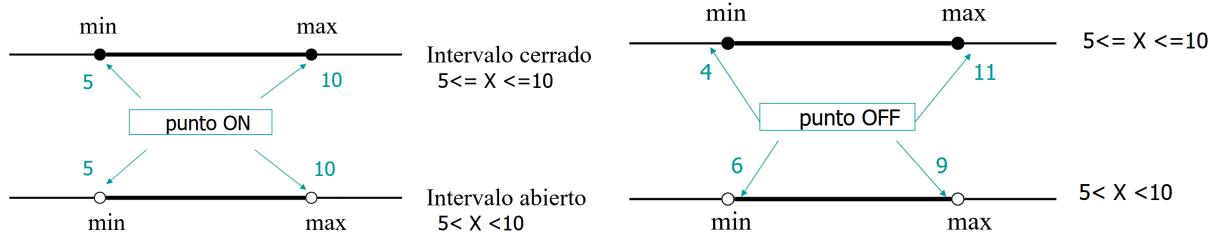


Figure 3.5: puntos ON y OFF

- ◊ PUNTO ON : un punto que pertenece al intervalo y es un límite
 - Intervalo cerrado - punto ON satisface la condición del intervalo
 - Intervalo abierto - punto ON *no* satisface la condición del intervalo
- ◊ PUNTO OFF : un valor cerca del punto límite de un intervalo
 - Intervalo cerrado - punto OFF *no* satisface la condición del intervalo
 - Intervalo abierto - punto OFF satisface la condición del intervalo

La estrategia para diseñar las pruebas de valores límites es 1 punto ON + 1 punto OFF para cada desigualdad de límite.

Ejemplos:

- ◊ $3 \leq \text{extras} < 10$
 - Puntos ON: $\text{extras} = 3$; $\text{extras} = 10$
 - Puntos OFF: $\text{extras} = 2$; $\text{extras} = 9$
- ◊ $2000 < \text{precio_base} \leq 5000$
 - Puntos ON: $p = 2000$; $p = 5000$
 - Puntos OFF: $p = 2001$; $p = 5001$

- ◊ $x > 0$ (x es número natural)
 - ON = 0
 - OFF = 1
- ◊ $x \leq 10$ (x es número natural)
 - ON = 10
 - OFF = 11
- ◊ $x > 0$ (x es número real)
 - ON = 0
 - OFF = 0,00001
- ◊ $x \leq 10$ (x es número real)
 - ON = 10
 - OFF = 10,0001

3.5 Matriz del testeo de dominio

Variable/ desigualdad/tipo		Casos de testeo															
		tipo	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
X1	C11	ON															
		OFF															
	C12	ON															
		OFF															
	ON															
		OFF															
	C1m	ON															
		OFF															
	Típico	IN															
	...																
Xn	Cn1	ON															
		OFF															
	ON															
		OFF															
	Cnm	ON															
		OFF															
	Típico	IN															
	Resultado esperado																

Una representación conveniente para representar el conjunto de casos de test desarrollados con el testeo de dominio.

Cada columna es un caso de test.

Cada caso de test:

- ◊ un ON o OFF para la variable en cuestión (diagonal)
- ◊ para las demás variables un IN (horizontales)

Tipos de datos “nonscalar” no tienen orden, por ejemplo strings, boolean o enumeraciones. Los límites del dominio son cerrados y binarios: la variable cumple la condición o no. Entonces,

- ◊ El Punto ON es valor que cumple la condición
- ◊ El Punto OFF es valor que no cumple la condición

Figure 3.6: Matriz de testeo de dominio

3.5.1 Money class ejemplo

```

class Money {
    private int fAmount;
    private String fCurrency;

    public Money(int amount, String currency) {
        fAmount = amount;
        fCurrency = currency;
    }

    public int amount() {
        return fAmount;
    }

    public String currency() {
        return fCurrency;
    }
}

```

Imagina que tenemos que testear un componente:

- ◊ entrada un objeto Money m
- ◊ el componente espera que
 - m.amount()>= 0
 - m.currency() in {EURO,USD, GBP, YEN}
- ◊ Si el cliente quiere pagar en USD,GBP o YEN, tiene que pagar 10% de costes de administración
- ◊ Si el cliente paga en EURO, no hay costes de administración

Variable/ desigualdad/tipo			Casos de testeo							
	condición	tipo	1	2	3	4	5	6	7	8
m.amount()	m.amount() ON	0			0					
	m.amount() OFF			-1		-1				
	Típico	IN					25	25	25	25
m.currency()	m.currency() ON = EURO						EURO			
	m.currency() OFF							USD		
	Típico	IN	EURO	EURO						
	m.currency() in {USD, GBP, YEN} ON								GBP	
	m.currency() OFF									EURO
	Típico	IN			YEN	YEN				
Resultado esperado			0%	Err	10%	Err	0%	10%	10%	0%

Figure 3.7: Matriz de testeo de dominio para la clase Money

Si tenemos una clase compleja que no se puede tratar como un escalar, es posible que podemos utilizar una “abstracción de estados” para definir los límites del dominio. Por ejemplo considera una clase Pila (*stack*), que se puede encontrar en tres estados:

1. vacío - stack.size()== 0
2. cargado - stack.size()> 0 && stack.size < MAXSTACK
3. lleno - stack.size()== MAXSTACK
 - ◊ Punto ON / vacío - stack.size()== 0
 - ◊ Punto IN / vacío - stack.size()== 0
 - ◊ Punto OFF / vacío - stack.size()== 1
 - ◊ Punto ON / cargado - stack.size()== MAXSTACK - 1
 - ◊ Punto IN / cargado - stack.size()== ceiling(MAXSTACK / 2)
 - ◊ Punto OFF / cargado - stack.size()== 0 || MAXSTACK
 - ◊ Punto ON / lleno - stack.size()== MAXSTACK
 - ◊ Punto IN / lleno - stack.size()== MAXSTACK
 - ◊ Punto OFF / lleno - stack.size()== MAXSTACK + 1

3.6 Máquinas de estado

La idea es modelar el comportamiento del sistema como una máquina de estados. Los casos de prueba diseñados con esta técnica tratarán de cubrir los diferentes estados y transiciones especificados en la máquina de estados que representa el sistema.

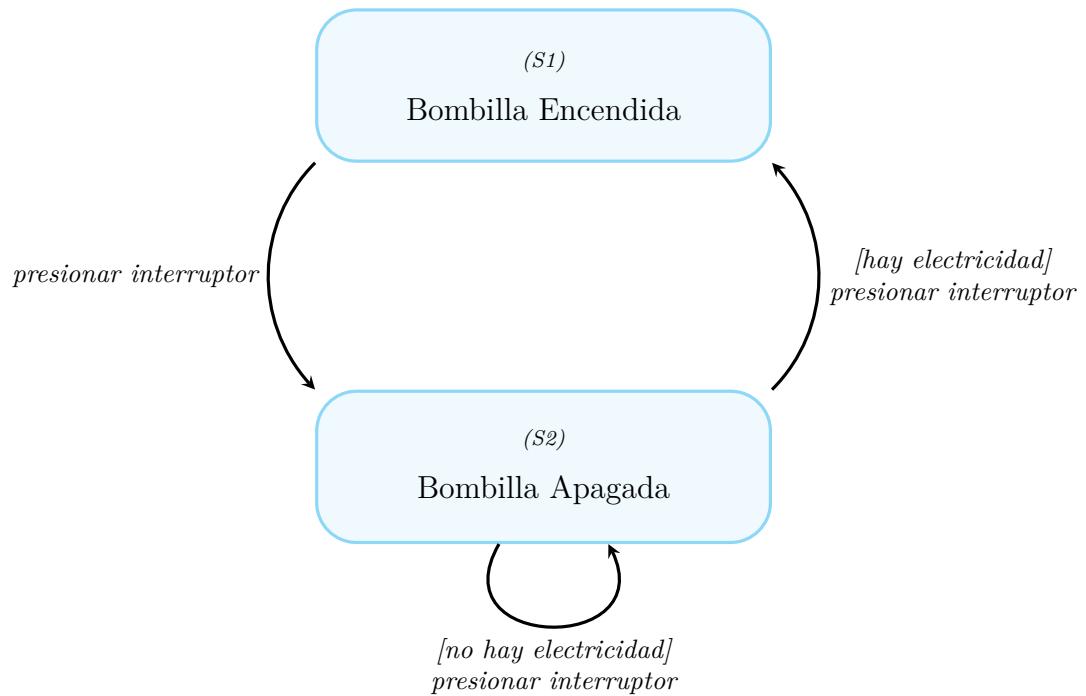


Figure 3.8: Ejemplo de máquina de estados

Test case	1	2	3
Estado inicial	S1 - Encendida	S2 - Apagada	S2 - Apagada
Entrada	Presionar interruptor	Presionar interruptor	Presionar interruptor
Condición	Hay electricidad	Hay electricidad	No hay electricidad
Estado final	S2 - Apagada	S1 - Encendida	S2 - Apagada

Table 3.3: Tabla de ejemplo para la máquina de estados de luz con interruptor

Cobertura

- ◊ Cobertura de **estados**: Este criterio se satisface cuando los casos de prueba recorren todos los estados.
Test case: 2
- ◊ Cobertura de **transiciones**: En este caso, se satisface el criterio cuando se recorren todas las transiciones.
Test case: 1,2,3
- ◊ Cobertura de **pares de transiciones**: Para cada estado se cubren las combinaciones de transiciones de entrada y salida.
Test case: 1,2,3

Cobertura de estados: 2 Cobertura de transiciones: 1, 2 y 3. Cobertura pares de transiciones: 1, 2 y 3

Chapter 4

Metodologías de Desarrollo

Una metodología de desarrollo es un conjunto de prácticas, técnicas y herramientas que se utilizan para llevar a cabo el proceso de desarrollo de software. Existen diferentes enfoques para el desarrollo de software, cada uno con sus propias características y ventajas.

4.1 Definiciones

Estos son enfoques, no son *metodologías* en sí: metodologías van a seguir uno de estos enfoques, pero son algo más:

- ◊ **Cascada** - Es muy sencillo y fácil de entender, pero no es flexible.
- ◊ **Prototipado** - Requiere diseño de prototipos muy rápido y consiguiente evaluación por el cliente.
- ◊ **Incremental** - Desarrollar el sistema en piezas, cada una de las cuales tiene un valor, y luego se integran.
- ◊ **Espiral** - Se va costruyendo el producto en partes y es muy flexible para incrementar (?), y es similar a incremental, pero hay también un análisis de riesgos en cada iteración.

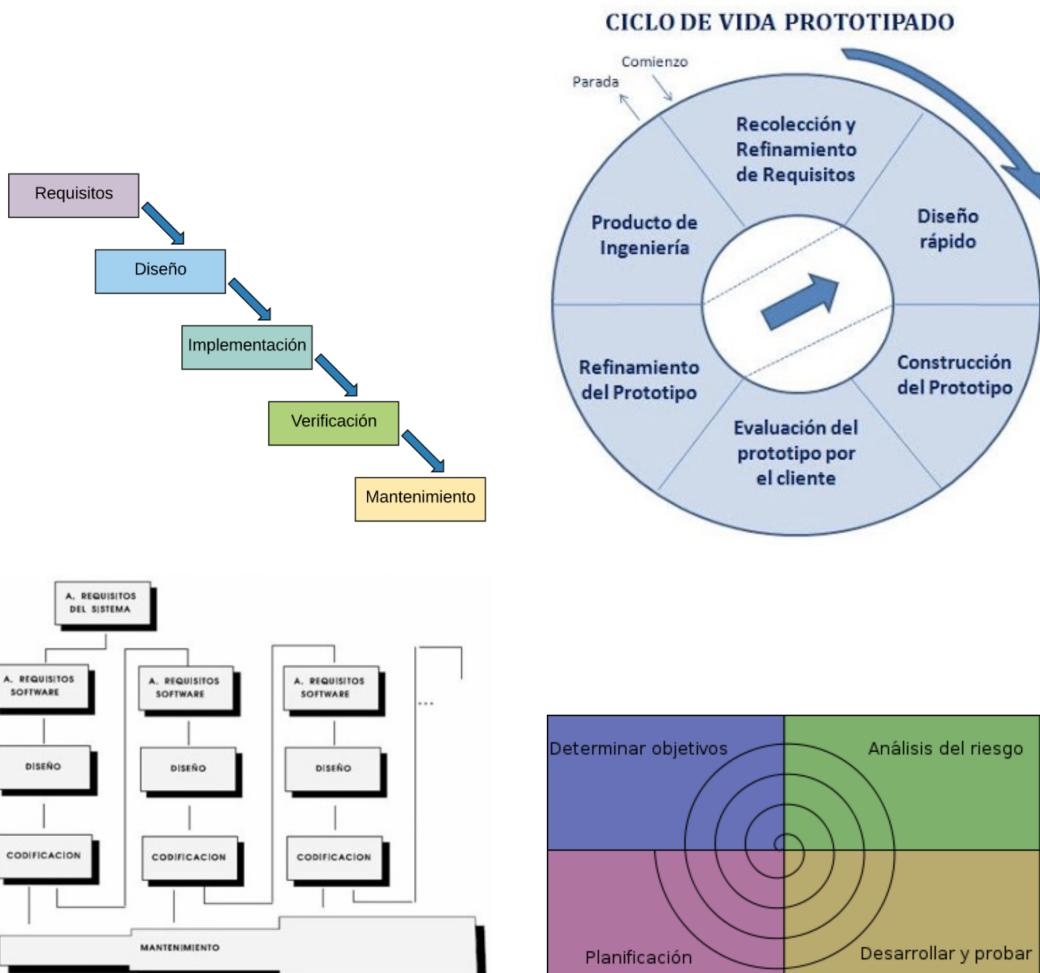


Figure 4.1: Enfoques generalistas: cascada, prototipado, incremental y espiral

Un desarrollo iterativo es aquel en el que, con cada entrega esperamos el feedback del usuario para decidir los siguientes pasos a seguir.

Un desarrollo incremental es aquel en el que con cada entrega, tenemos acabada una pieza más del sistema.

Lo mejor es un desarrollo iterativo e incremental, que es aquel en el que cada entrega es una pieza del sistema, y además esperamos el feedback del usuario para decidir los siguientes pasos a seguir.

4.2 Metodologías

- ◊ **Tradicionales** - Son orientadas al proyecto y presentan un enfoque lineal, con documentación formal.
 - **RUP (Rational Unified Process)** - Es un modelo incremental e iterativo que modela visualmente el software. Es un proceso dirigido por los casos de uso y centrado en la arquitectura

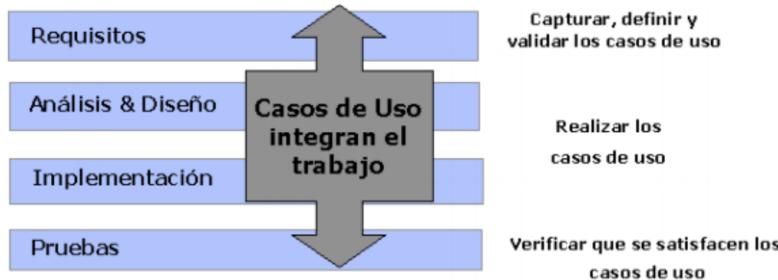


Figure 4.2: RUP schema

- **V-Model** - Es un modelo de desarrollo de software que se basa en la idea de que cada fase del desarrollo debe tener una fase de prueba correspondiente. Se representa como una V, donde la parte izquierda representa las fases de desarrollo y la parte derecha representa las fases de prueba.
No es muy flexible, requiere mucha documentación, es muy orientada al proyecto, y es bueno para sistemas que son muy críticos cuando son apropiadas muchas pruebas, para comprobar la calidad del producto cada vez que se entrega una pieza.

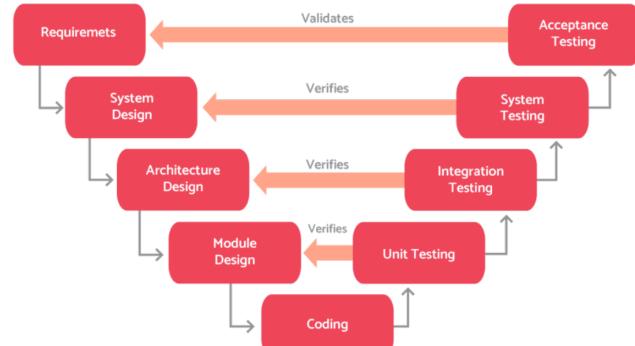


Figure 4.3: V-Model

- ◊ **Agile** - La palabra “agile” se relaciona con la respuesta al progreso que pueden ofrecer las metodologías agiles. Las iteraciones son de 2-4 semanas, y la planificación se hace al finalizar cada iteración. Los valores del manifesto agile son:

- **Individuos e interacciones** sobre procesos y herramientas
- **Software funcionando** sobre documentación extensiva
- **Colaboración con el cliente** sobre negociación contractual
- **Respuesta al cambio** sobre seguir un plan
- ...

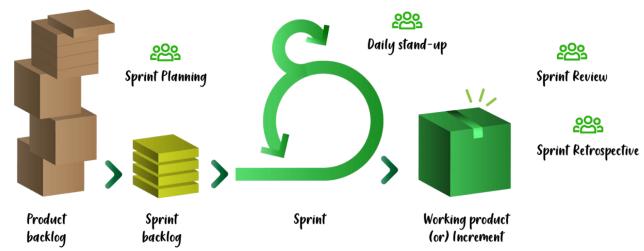


Figure 4.4: Agile workflow

- **Scrum** - Es un marco de trabajo ágil que se utiliza para gestionar proyectos de desarrollo de software. Se basa en la idea de que el desarrollo debe ser iterativo e incremental, y se centra en la colaboración entre los miembros del equipo.

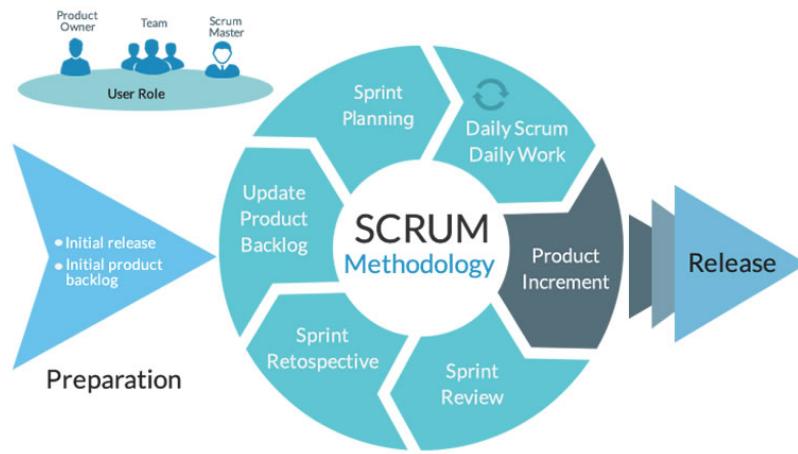


Figure 4.3: Scrum framework

- **Extreme Programming** - No tiene mucho diseño inicial, y entonces se va a hacer mucha refactorización.

4.3 Metodologías modernas

Características básicas de una estrategia de testing

- ◊ Proporciona feedback.
- ◊ Es rápida.
- ◊ Debe estar automatizada.

Ejemplos modernos de metodologías son CI/CD (Integración continua y entrega continua) y DevOps.

“El testing es una actividad integrada en el desarrollo, no una fase posterior al desarrollo” — Manuela

Las metodologías ágiles no ven al software testing como una fase separada, sino como parte integral del desarrollo de software al igual que la programación.

Los equipos ágiles utilizan un enfoque de “todo el equipo” al testing, con la finalidad de integrar la calidad al desarrollo del producto, al contrario de un enfoque de primero fabricar el producto y luego inspeccionar para determinar su nivel de calidad

4.3.1 Test Driven Development (TDD)

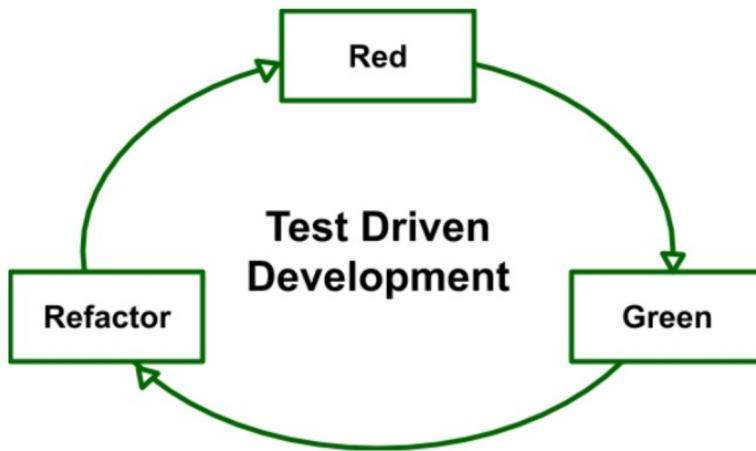


Figure 4.4: TDD schema

“*Red*” significa que tenemos que escribir una prueba que falle, “*Green*” significa que tenemos que escribir el código necesario para hacer que la prueba pase, y “*refactor*” significa que tenemos que refactorizar el código para mejorar su calidad, integrando el nuevo código con el antiguo.

El *Test Driven Development* (TDD)¹ es una metodología de desarrollo de software que se basa en la idea de que las pruebas deben ser escritas antes de escribir el código. El proceso de TDD se divide en tres pasos:

1. Escribir una prueba que falle
2. Escribir el código necesario para hacer que la prueba pase
3. Refactorizar el código para mejorar su calidad

Reglas para el TDD

1. No está permitido escribir código de producción sin una prueba que falle
2. No está permitido escribir más código de producción del necesario para hacer que la prueba pase
3. No está permitido escribir más pruebas de las necesarias para cubrir el código de producción

¹Desarrollo guiado por pruebas

