

Verónica Lozada Pérez
Bianca Luna

1639552
1676394

INFORME DE PRÁCTICAS

TEST Y CALIDAD DE SOFTWARE

Verónica Lozada Pérez
Bianca Luna

Profesor encargado: Xavier Otazu

Funcionamiento del sistema

• Introducción

El presente informe describe el diseño, funcionamiento y características del sistema de gestión de empresas y empleados desarrollado con el objetivo de administrar organizaciones, gestionar trabajadores y calcular nóminas de manera eficiente.

Dentro del sistema se han establecido dos tipos de acceso diferenciados, empresa y empleado, y define un conjunto de funciones, restricciones y mecanismos de control para garantizar la integridad, coherencia y seguridad de la información.

• Descripción general del sistema

- El sistema permite gestionar información administrativa sobre empresas y sus trabajadores, proporcionando herramientas para:
 - Registrar empresas y trabajadores.
 - Administrar información laboral.
 - Realizar cálculos de nómina considerando múltiples variables personalizables.
 - Establecer un control estricto sobre qué tipo de usuario puede acceder a modificar datos.

• Roles de acceso

3.1 Accesos como empresa

El acceso empresarial se realiza mediante el CIF (Código de Identificación Fiscal). El sistema ofrece dos procedimientos:

3.1.1 Acceso mediante CIF existente

- El sistema verifica el CIF en la base de datos.
- Si existe, la empresa accede a su panel de administración.
- Desde este entorno puede gestionar trabajadores, consultar información registrada y realizar cálculos de nómina.

3.1.2 Creación de una nueva empresa

- Si el CIF no está registrado, la empresa debe crear un nuevo perfil.
- Se almacenan datos básicos, los cuales se describen en los atributos más adelante.
- La empresa queda habilitada automáticamente para gestionar trabajadores.

3.2 Acceso como empleado

El acceso del empleado se realizara mediante su DNI, existiendo restricciones específicas:

- El empleado solo puede iniciar sesión si está previamente registrado por una empresa.
- No puede registrarse ni crear una cuenta por si mismo.
- Una vez validado, tiene acceso únicamente a sus datos y cálculos personales.

● Gestión de actores en el sistema

A continuación se detallan las especificaciones de qué datos contiene cada actor del sistema.

Worker (Trabajador)

```
private String name;  
private String dni;  
private String civilStatus;  
private int children;  
private float totalIncome;  
private int payments;  
private String contract;  
private int category;  
private String cifEmpresa;
```

Company (Empresa)

```
private String name;  
private String cif;  
private String email;  
private String phone;  
private String address;  
private String website;  
private String cnae;
```

Por otro lado, los valores que pueden tomar estos están parametrizados o limitados a ciertas opciones, tales que si no cumplir con los requisitos estos no serán aceptados, procedemos a describir los mismos:

- El nombre de un trabajador no debe tener números o caracteres especiales.
- El DNI ingresado deberá cumplir con la cantidad de dígitos y la letra deberá coincidir con la calculada.
- El estado civil de un empleado puede ser Soltero, Divorciado, Casado o Viudo.
- La cantidad de hijos de un empleado deberá ser un número entero mayor o igual a 0 (cero).
- El salario total anual de un empleado deberá ser mayor al salario mínimo anual dictaminado por las autoridades nacionales o entidad pertinente.
- El número de pagas a realizar al empleado deberá ser 12 o 14, sin admitir ninguna otra opción.
- El contrato por el cual el empleado se relaciona con la empresa deberá ser: “Indefinido”, “Temporal”, “Formación en Alternancia”, “Formativo para la Obtención de la Práctica Profesional”
- La categoría profesional de un empleado deberá ser un número entre 1 y 10.

- El correo electrónico de una empresa deberá cumplir con el patrón de un correo electrónico como tal.
- El teléfono de una empresa deberá contar con 8 dígitos, siendo cualquier otra opción incorrecta.
- El CNAE de una empresa deberá formar parte del listado de códigos dispuestos por las autoridades económicas nacionales.

Esta es una breve introducción al funcionamiento del sistema, a continuación detallaremos el funcionamiento lógico así como la implementación de las técnicas de test y calidad de software vistas a lo largo del periodo lectivo.

Cálculo de Nómina y como se realiza

Ante la confección de una nómina, deben tenerse en cuenta muchos aspectos, sobre todo las cargas y descargas del empleado y empleador. Esto está sujeto a muchas variables administrativas, por lo que las detallaremos un modelo de nómina.

La nómina se encuentra dividida en **4** partes para este ejemplo¹²:

- 1) **Encabezado** : se incluyen los datos básicos de la empresa, así como los datos del trabajador.
- 2) **Devengos**: son los ingresos (en bruto) que percibe el trabajador, es decir que a este valor no se le han aplicado ninguna de las deducciones. Aparecen divididos en:
 - a) Salariales: son aquellas se obtienen en base al trabajo realizado y están sujetas a cotización, conformadas por el salario base y los complementos salariales (horas extras, pagas extraordinarias, etc.).
 - b) No Salariales: no corresponden al periodo trabajado ni son computables, siendo percibidos como indemnización o contraprestación a un desembolso realizado previamente.
- 3) **Deducciones** : son las retenciones que se le realizan al trabajador y cuya aplicación da lugar al salario neto. Los tipos de deducciones son varios:
 - a) Seguridad Social: compuestas por las contingencias comunes (4,7%), desempleo (1,55% si contrato es indefinido y 1,60% si el contrato es de duración determinada), formación (0,10%), horas extraordinarias (4,70% si son normales y 2% si son de fuerza mayor) y MEI (0,12%).
 - b) IRPF: el porcentaje de esta deducción es variable y depende de los ingresos del trabajador.

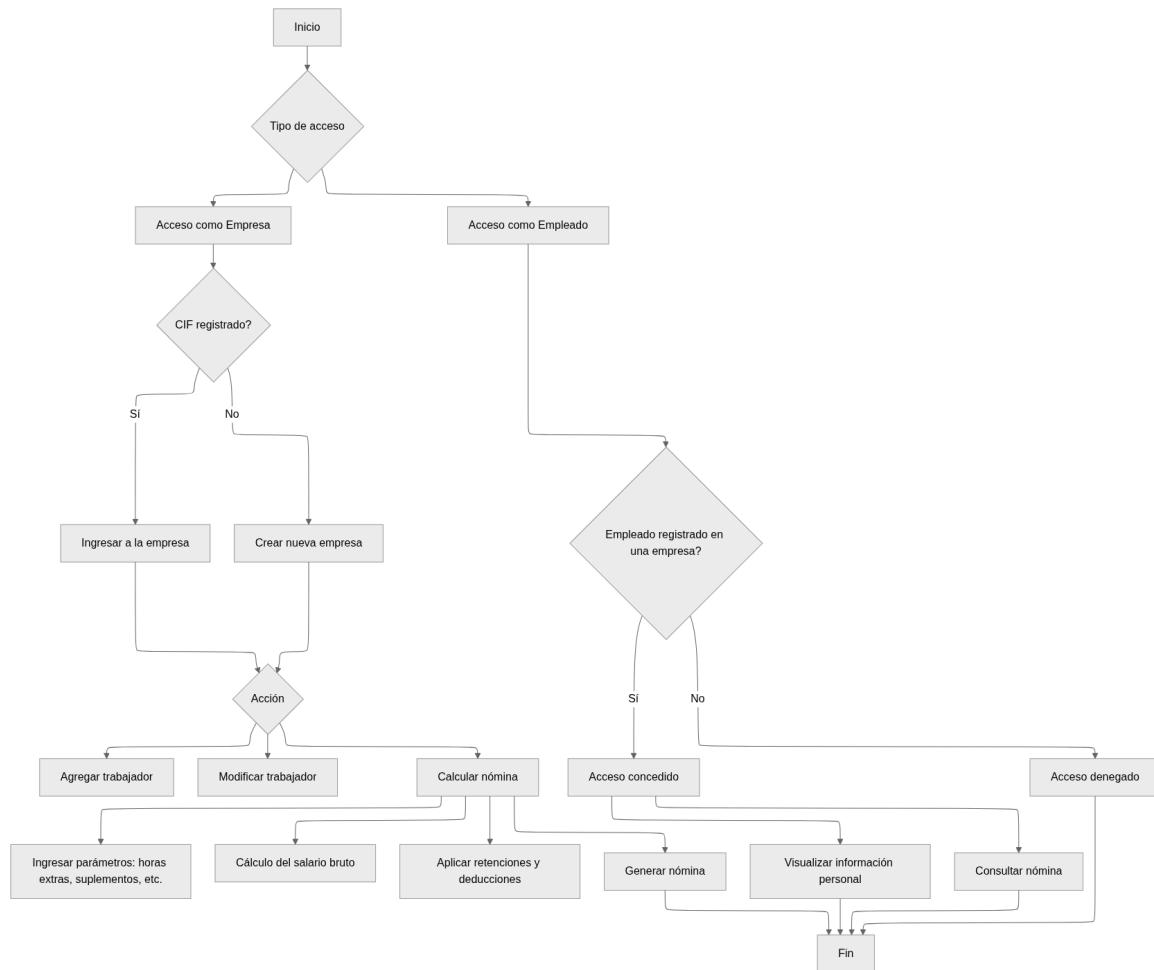
INGRESO ANUAL	RETENCIÓN
hasta 12.450	19%
12.450 a 20.199	24%
20.200 a 35.199	30%
35.200 a 59.999	37%
60.000 a 299.999	45%
más de 300.000	47%

- 4) **Liquidez a percibir**: corresponde al sueldo neto que se ingresará en la cuenta bancaria, para calcularlo solo se debe restar las deducciones a los devengos calculados anteriormente.

¹ Información obtenida del portal de BBVA en el siguiente enlace
<https://www.bbva.es/finanzas-vistazo/ef/banca-digital/este-es-un-ejemplo-de-nomina.html>

² Seguimiento de modelo del Boletín Oficial del Estado (BOE)
<https://www.boe.es/boe/dias/2014/11/11/pdfs/BOE-A-2014-11637.pdf>

Diagrama de Flujo



Aplicación del contenido teórico

MVC

model	Worker.java, Payroll.java, Company.java
view	App
controller	CompanyRespository.java, WorkerRepository.java

El proyecto implementa el Model-View-Controller (MVC) para separar las responsabilidades y facilitar el testing del código.

Model (Modelo)

El modelo contiene la lógica de negocio y la representación de los datos del sistema. En nuestro proyecto incluye:

- **Worker.java**: Representa a un trabajador con sus atributos (nombre, DNI, estado civil, hijos, salario, etc.) e incluye la lógica de validación de datos como la verificación del formato del DNI, validación del estado civil, categoría profesional y tipo de contrato.
- **Company.java**: Representa a una empresa con sus datos identificativos (CIF, nombre, email, teléfono, dirección, web, CNAE). Contiene validaciones básicas como la verificación de que el CIF no sea vacío.
- **Payroll.java**: Gestiona la propia nómina en sí a través de múltiples cálculos, incluyendo deducciones de IRPF y Seguridad Social, así como el cálculo del salario neto. Implementa la lógica compleja de los tramos de IRPF y los porcentajes de cotización según la categoría profesional del trabajador.

View (Vista)

La vista gestiona la interfaz con el usuario. En nuestro caso se hace a través del siguiente fichero:

- **App.java**: Actúa como capa de presentación del sistema. Mediante la terminal, gestiona toda la interacción con el usuario, mostrando menús diferenciados según el tipo de usuario (empresa o trabajador), capturando datos de entrada y presentando información en formato de tablas y mensajes con colores para mejorar la experiencia visual.

Todo se hace siguiente el flujo explicado anteriormente en el apartado de funcionamiento del sistema.

Controller (Controlador)

El controlador actúa como intermediario entre el modelo y la vista, definiendo las operaciones que se pueden realizar sobre los datos:

- **WorkerRepository.java**: Interfaz que define las operaciones **CRUD** (Create, Read, Update, Delete) para gestionar trabajadores: guardar, buscar por DNI, buscar por empresa y eliminar trabajadores.
- **CompanyRepository.java**: Interfaz que define las operaciones para gestionar empresas: guardar, actualizar, buscar por CIF y eliminar empresas.

TDD

El *Test Driven Development* consiste en el desarrollo de código a partir de la generación de los casos de prueba primeros. Durante la realización de todo nuestro proyecto, tuvimos extremo cuidado en aplicar esta metodología para la confección de las clases principales del código, sobre todo aquellas que tuvieran lógica.

La imagen que adjuntamos a continuación, hace referencia al desarrollo de la función `testValidPayment()`. Como podemos observar que ha seguido el flujo de:

TEST → FEATURE → REFACTOR

- refactor: rename testValidPayments to testValidPayments12 for clarity
- feat: add default constructor and setter for payments in Worker class
- test: add unit test for valid payments in Worker class

TDD en clase Worker.java

testValidIncome() – otro ejemplo donde hay 2 versiones del test desarrollado y del test de prueba.

- feat: update income validation to disallow zero income
- test: add testTotalIncomeNotZero to validate handling of zero total income
- feat: add income validation in setTotalIncome method
- test: add testNegativeTotalIncome to validate handling of negative income

En el *log* de todos nuestros *commits*, siempre se verá el mismo ciclo a la hora de enfocar el desarrollo de las clases que contienen la lógica, es decir, todas aquellas que conformen el apartado de *model*.

Particiones Equivalentes, Valores límites y Valores frontera

La técnica de particiones equivalentes consiste en dividir el dominio de entrada de una función en clases o grupos donde todos los valores de una misma partición deberían comportarse de manera similar. Esto permite reducir el número de casos de prueba necesarios mientras se mantiene una cobertura efectiva.

En los métodos test de la clase Worker.java

testValidPayments()

```
/*
 * Test cases for validating number of payments within a year, which can only be 12 or 14.
 *
 * Equivalence Partitions:
 * 1. Valid payments: 12, 14
 * 2. Invalid payments: any other number (-1, 0, 15, 60) (numbers outside the valid range)
 * 3. Boundary cases: 11, 12, 13, 14, 15
 */
```

fichero: WorkerTest.Java

testTotalIncome()

```
/*
 * Test cases for validating total income, which must be a positive value and with
 * minimum value of 16000 euros, which is the
 * interprofessional minimum salary within Spain.
 *
 * Equivalence Partitions:
 * 1. Valid total income: positive values greater than or equal to 16000 euros (e.g., 16000, 25000.60, 800000)
 * 2. Invalid total income: negative values and zero (e.g., -5000, -0.01, 0, 15999.99)
 * 3. Boundary case: exactly 16000.00 euros, -0.01, 0, 1, 15 999.99, 16 000.01
 */
```

fichero: WorkerTest.Java

testChildren()

```
/*
 * Test cases for validating number of children, which must be a non-negative integer.
 *
 * Equivalence Partitions:
 * 1. Valid number of children: non-negative integers (e.g., 0, 1, 2, 3, ...)
 * 2. Invalid number of children: negative integers (e.g., -1, -2, -3, ...)
 * 3. Boundary case: -1, 0, 1
 */
```

fichero: WorkerTest.Java

testChildren()

```
/*
 * Test cases for validating number of children, which must be a non-negative integer.
 *
 * Equivalence Partitions:
 * 1. Valid number of children: non-negative integers (e.g., 0, 1, 2, 3, ...)
 * 2. Invalid number of children: negative integers (e.g., -1, -2, -3, ...)
 * 3. Boundary case: -1, 0, 1
 */
```

fichero: WorkerTest.Java

testValidCategory()

```
/*
 * Test cases for validating worker category, which must be an integer between 1 and 10.
 * Equivalence Partitions:
 * 1. Valid category: integers between 1 and 10 (e.g., 1, 5, 11)
 * 2. Invalid category: integers less than 1 or greater than 10 (e.g., -1, 0, 12, 20)
 * 3. Boundary cases: -1, 0, 1, 10, 11
 */
```

fichero: WorkerTest.java

testContract()

```
/*
 * Test cases for validating contract type, which must be one of the following strings:
 * "Indefinido", "Temporal", "Formacion en Alternancia", "Formativo para la Obtencion de la Práctica Profesional"
 * Equivalence Partitions:
 * 1. Valid contract types: "Indefinido", "Temporal", "Formacion en Alternancia",
 * "Formativo para la Obtencion de la Práctica Profesional"
 * 2. Invalid contract types: any other string (e.g., "indefinido-invalid", "permanent", "temporary")
 */
```

fichero: WorkerTest.java

testCivilStatus()

```
/*
 * Test cases for validating civil status, which must be one of the following strings:
 * "Soltero", "Casado", "Viudo", "Divorciado"
 * Equivalence Partitions:
 * 1. Valid civil status: "Soltero", "Casado", "Viudo", "Divorciado"
 * 2. Invalid civil status: any other string (e.g., "Separado", "Solo", "Matrimonio")
 */
```

fichero: WorkerTest.java

En los métodos test de la clase Payroll.java

testSocialSecurityDeduction()

```
/*
 * Test cases for calculating social security deduction based on contribution group.
 *
 * Equivalence Partitions:
 * 1. Group 1-4 → 6.35%
 * 2. Group 5-7 → 6.40%
 * 3. Group 8-10 → 6.45%
 * 4. Invalid group (<1 or >10) → IllegalArgumentException
 * 5. Boundary cases: 1, 0, 2, 3, 6, 9, 10, 11
 */
```

fichero: PayrollTest.Java

testIrpfdeduction()

```
/* Equivalence Partitions:
 * 1. Base cases (no children, indefinite contract):
 *     - Income ≤ 12,450 → 19%
 *     - 12,451 – 20,200 → 24%
 *     - 20,201 – 35,200 → 30%
 *     - 35,201 – 60,000 → 37%
 *     - > 60,000 → 45%
 *
 * 2. Children cases (1 to 5 children):
 *     - Each child reduces IRPF by 1% up to a maximum of 5%
 *
 * 3. Temporary contract cases:
 *     - Add 3% to the base IRPF rate
 *
 * 4. Boundary cases:
 *     - income: negative income, 0 income, 12449, 12451, 20199, 20201, 35199, 35201, 59999, 60001
 *     - children: 0, 4, 6
 *     - contract type: "Indefinido", "Temporal", invalid types
 * */
```

fichero: PayrollTest.Java

Pairwise Testing

Al momento de realizar diferentes casos de prueba encontramos que esta técnica sería la qué mejor cobertura de casos nos haría tener, ya qué para tener en cuenta diferentes situaciones particulares de los empleados, hay qué también tener en cuenta la amplia cantidad de variables que se deben tener en cuenta así como las distintas variaciones de la

misma. A continuación detallamos en formato de tabla el uso de esta técnica en nuestro desarrollo:

caso	civilStatus	children	totalIncome	payments	contract	category
1	Soltero	0	1200	12	Indefinido	1
2	Soltero	1	1800	14	Temporal	3
3	Soltero	3	2500	12	Formacion en Alternancia	5
4	Casado	0	1800	12	Temporal	1
5	Casado	1	2500	14	Indefinido	3
6	Casado	3	1200	14	Formacion en Alternancia	1
7	Soltero	0	2500	14	Temporal	5
8	Casado	0	1200	12	Formacion en Alternancia	3
9	Soltero	1	1200	14	Indefinido	5
10	Casado	1	1800	12	Formacion en Alternancia	1
11	Soltero	3	1800	14	Indefinido	3
12	Casado	3	2500	12	Temporal	5

Todos estos casos pueden ser consultados en el apartado de `// ==> PAIRWISE TESTING` en el código de prueba de deducciones del IRPF. Sitio que consideramos más relevante para comprobar tantos casos y tantas entradas

Mock Object

Hemos utilizado la técnica de “Mock Object” en la simulación de dos bases de datos qué consideramos esenciales para el funcionamiento de nuestro sistema, siendo estas una donde se guardan las empresas con sus datos (**CompanyRepositoryMock**) y la otra una base de datos con los empleados de todas las empresas (**WorkerRepositoryMock**).

CompanyRepositoryMock	<pre>public interface CompanyRepository { 2 usages 1 implementation & Bianca /** * Guarda una compañía en el repositorio. * @param company La compañía a guardar * @return true si se guardó correctamente, false en caso contrario */ boolean save(Company company); no usages 1 implementation & Bianca boolean update(Company company); no usages 1 implementation & Bianca Company findByCif(String cif); 2 usages 1 implementation & Bianca boolean delete(String cif); no usages 1 implementation & Bianca }</pre>
WorkerRepositoryMock	<pre>import model.Worker; import java.util.List; public interface WorkerRepository { 1 usage 1 implementation & Bianca boolean save(Worker worker); 4 usages 1 implementation & Bianca Worker findByDni(String dni); 5 usages 1 implementation & Bianca List<Worker> findByCompany(String companyCif); no usages 1 implementation & Bianca boolean delete(String dni); 1 usage 1 implementation & Bianca }</pre>

Coverage

Coverage All in tqs_project ×				
Element	Class, %	Method, %	Line, %	Branch, %
all	50% (3/6)	56% (50/88)	34% (144/412)	37% (67/179)
repository	0% (0/2)	0% (0/16)	0% (0/77)	0% (0/52)
WorkerRepositoryMock	0% (0/1)	0% (0/7)	0% (0/32)	0% (0/18)
CompanyRepositoryMock	0% (0/1)	0% (0/9)	0% (0/45)	0% (0/34)
App	0% (0/1)	0% (0/22)	0% (0/191)	0% (0/53)
controller	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
WorkerRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
CompanyRepository	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
model	100% (3/3)	100% (50/50)	100% (144/144)	90% (67/74)
Payroll	100% (1/1)	100% (11/11)	100% (54/54)	80% (21/26)
Company	100% (1/1)	100% (15/15)	100% (24/24)	100% (4/4)
Worker	100% (1/1)	100% (24/24)	100% (66/66)	95% (42/44)

Resultado de ejecutar el coverage sobre todos los tests del proyecto.

Statement Coverage

La cobertura de sentencias mide el porcentaje de líneas de código que han sido ejecutadas al menos una vez durante la ejecución de los tests. El objetivo es asegurar que cada instrucción del programa haya sido probada.

Según el análisis de cobertura realizado con la función de *coverage* del propio IDE de IntelliJ (visible en la captura de pantalla adjunta), nuestro proyecto alcanza los siguientes niveles de cobertura:

 Run 'All in tqs_project' with Coverage

Cobertura por paquetes:

Paquete	Class Coverage	Method Coverage	Line Coverage	Branch Coverage
all	50% (3/6)	56% (50/88)	34% (144/412)	37% (67/179)
repository	0% (0/2)	0% (0/16)	0% (0/77)	0% (0/52)
controller	100% (0/0)	100% (0/0)	100% (0/0)	100% (0/0)
model	100% (3/3)	100% (50/50)	100% (144/144)	90% (67/74)

Cobertura de las clases del modelo:

Clase	Class Coverage	Method Coverage	Line Coverage	Branch Coverage
Payroll	100% (1/1)	100% (11/11)	100% (54/54)	80% (21/26)
Company	100% (1/1)	100% (15/15)	100% (24/24)	100% (4/4)
Worker	100% (1/1)	100% (24/24)	100% (66/66)	95% (42/44)

Estos resultados son coherentes con el tipo de cobertura que intentamos conseguir. En el modelo, que es donde reside la lógica principal, hay una cobertura del 100% en las líneas de código, lo que indica que cada sentencia ha sido ejecutada al menos una vez. Vemos como las tres clases principales: Worker, Company y Payroll están completamente cubiertas en cuanto a métodos y líneas.

Por el resto de resultados, vemos también las interfaces al 100%, pero porque solo contienen definiciones. En cuanto al paquete de repository, como no contiene los test directamente, es coherente que tenga una cobertura del 0%, y lo mismo con App que no necesita de ningún tipo de test.

Decision Coverage

La cobertura de decisiones verifica que cada punto de decisión (como if, while, for) en el código se haya evaluado tanto como verdadero como falso al menos una vez.

En nuestro proyecto, aplicamos Decision Coverage a los siguientes métodos:

Condition Coverage

La cobertura de condiciones asegura que cada condición booleana individual dentro de una decisión se haya evaluado tanto como verdadera como falsa. Es más estricta que Decision Coverage porque considera cada parte de una condición compuesta.

En nuestro proyecto, aplicamos Condition Coverage a los siguientes métodos:

Método 1: `isValidDni(String dni)` en Worker

Condición compuesta: `if (dni == null || dni.isEmpty())`

Ambas condiciones evaluadas como TRUE y FALSE

Test	<code>dni == null</code>	<code>dni.isEmpty()</code>	Resultado decisión
<code>testValidDni()</code>	TRUE	-	TRUE
	<pre>// 5. Asignar un DNI nulo worker.setDni(null); assertEquals(validDni, worker.getDni(), message: "El worker no debería aceptar DNIs nulos");</pre>		
	FALSE	TRUE	TRUE
	<pre>// 6. Asignar un DNI no nulo pero vacío worker.setDni(""); assertEquals(validDni, worker.getDni(), message: "El worker no debería aceptar DNIs vacíos");</pre>		
	FALSE	FALSE	FALSE
	<pre>// 1. Asignar un DNI válido inicial String validDni = "11768496V"; worker.setDni(validDni); assertEquals(validDni, worker.getDni());</pre>		
Líneas generadas por el “run with coverage”			

```
144     private boolean isValidDni(String dni) { 1 usage  & Bianca
145         if (dni == null || dni.isEmpty()) {
146             return false;
147         }
148
149         dni = dni.replace( target: "-", replacement: "").trim().toUpperCase();
150
151         if (!dni.matches( regex: "^\d{8}[A-Z]$")) {
152             return false;
153         }
154
155         String numeroStr = dni.substring(0, 8);
156         char letraProporcionada = dni.charAt(8);
157
158         int numero = Integer.parseInt(numeroStr);
159         int resto = numero % 23;
160         char letraCalculada = LETRAS_DNI.charAt(resto);
161
162         return letraProporcionada == letraCalculada;
163     }
164 }
```

Método 2: `calculateIrpfp()` en Payroll

Condición compuesta: `if (contract != null`

`&& contract.equalsIgnoreCase("temporal"))`

Ambas condiciones evaluadas como TRUE y FALSE

Test	<code>contract != null</code>	<code>equalsIgnoreCase("temporal")</code>	Resultado decisión
<code>testIrpfpDeduction()</code>	TRUE	TRUE	TRUE
	<pre>Worker temp1 = new Worker(name: "T1", dni: "71239485K", civilStatus: "Soltero", children: 0, totalIncome: 12000, payments: 12, contract: "temporal", category: 3, cifEmpresa: "J12345678"); payroll.setWorker(temp1); assertEquals(expected: 2640.0, payroll.calculateIrpfp(), delta: 0.01); // 22%</pre>		
	TRUE	FALSE	FALSE
<pre>Worker child5 = new Worker(name: "A5", dni: "71239485K", civilStatus: "Soltero", children: 5, totalIncome: 80000, payments: 12, contract: "Indefinido", category: 3, cifEmpresa: "J12345678"); payroll.setWorker(child5); assertEquals(expected: 32000.0, payroll.calculateIrpfp(), delta: 0.01); // 40%</pre>			
FALSE		-	FALSE

	<pre>// Equivalence Partition 2: grup 5-7 payroll.setWorker(cat5Worker); assertEquals(expected: 6400.0, payroll.calculateSocialSecurity());</pre>
Líneas generadas por el “run with coverage”	
79 80 81 82 83 84 85 86 87	<pre>// Modificador por contrato temporal: +3% si es "Temporal" double contractIncrease = 0.0; String contract = worker.getContract(); if (contract != null && contract.equalsIgnoreCase(anotherString: "temporal")) { contractIncrease = 0.03; } else { contractIncrease = 0.0; }</pre>

Método 3: `calculateSocialSecurity()` en Payroll

Condición compuesta: `if (worker.getCategory() >= 1
&& worker.getCategory() <= 4)`

Ambas condiciones evaluadas como TRUE y FALSE

Test	category	>= 1	<= 4	Resultado decisión
	0	FALSE	FALSE	FALSE
testSocial Security Deduction()		<pre>Exception exceptionLow = assertThrows(IllegalArgumentException.class, () -> { Worker invalidLow = new Worker(name: "Test", dni: "71239485K", civilStatus: "Soltero", children: 0, totalIncome: 10000, payments: 12, contract: "Indefinido", category: 0, cifEmpresa: "J12345678"); payroll.setWorker(invalidLow); payroll.calculateSocialSecurity(); }); assertEquals(expected: "Categoría inválida. Debe estar entre 0 y 10.", exceptionLow.getMessage());</pre>		
	2	TRUE	TRUE	TRUE
		<pre>// Equivalence Partition 1: grup 1-4 payroll.setWorker(cat1Worker); assertEquals(expected: 6350.0, payroll.calculateSocialSecurity()); payroll.setWorker(cat2Worker); assertEquals(expected: 6350.0, payroll.calculateSocialSecurity());</pre>		
	5	TRUE	FALSE	FALSE
		<pre>// Null contract type (should be treated as indefinite, no +3%) Worker nullContract = new Worker(name: "NC", dni: "71239485K", civilStatus: "Soltero", children: 0, totalIncome: 50000, payments: 12, contract: null, category: 3, cifEmpresa: "J12345678"); payroll.setWorker(nullContract); assertEquals(expected: 18500.0, payroll.calculateIrp(), delta: 0.01); // 37%</pre>		

Líneas generadas por el “run with coverage”

```
public double calculateSocialSecurity() { 12 usages  ↗ veronloz
    float percentage = 0;

    if (worker.getCategory() >= 1 && worker.getCategory() <= 4) percentage = 6.35f;
    else if (worker.getCategory() >= 5 && worker.getCategory() <= 7) percentage = 6.40f;
    else if (worker.getCategory() >= 8 && worker.getCategory() <= 11) percentage = 6.45f;
    sgs = Math.round(worker.getTotalIncome() * (percentage / 100));

    return sgs;
}
```

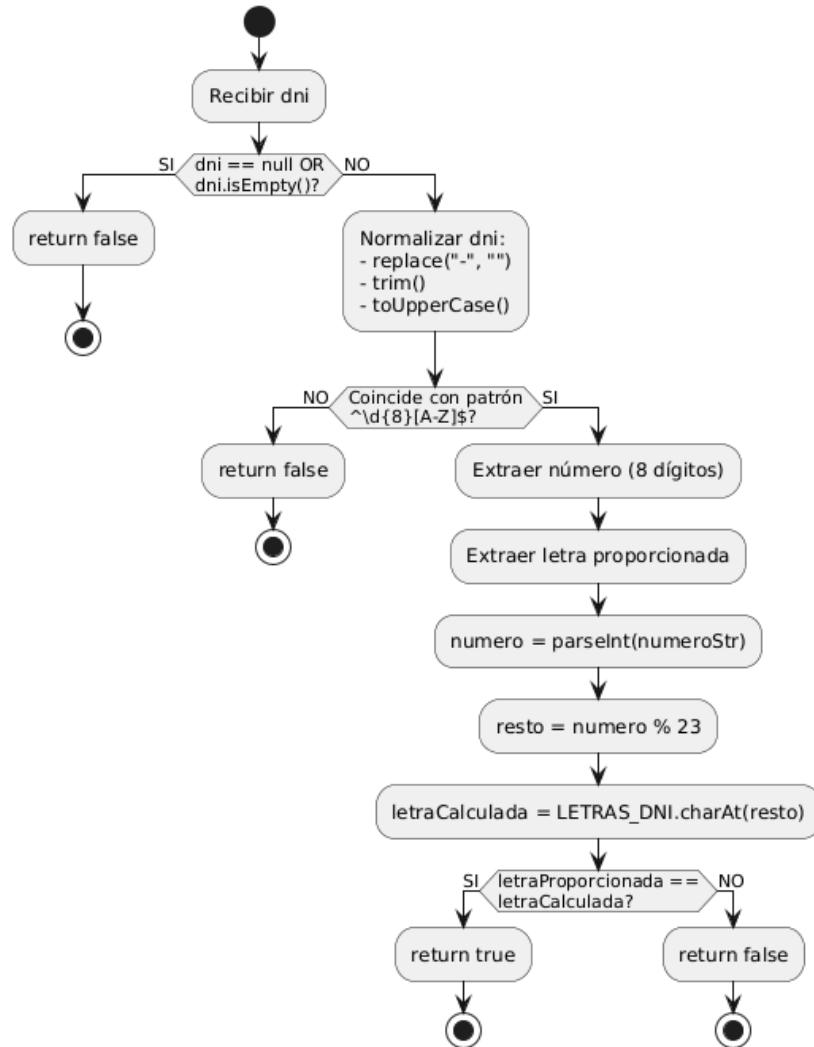
Path Coverage

La cobertura de caminos es el criterio más exhaustivo de pruebas de caja blanca. Requiere que se ejecuten todos los caminos posibles a través del código, desde el punto de entrada hasta el punto de salida. Esto incluye todas las combinaciones posibles de decisiones.

Método 1: `isValidDni(String dni)`

- Complejidad ciclomática: 4

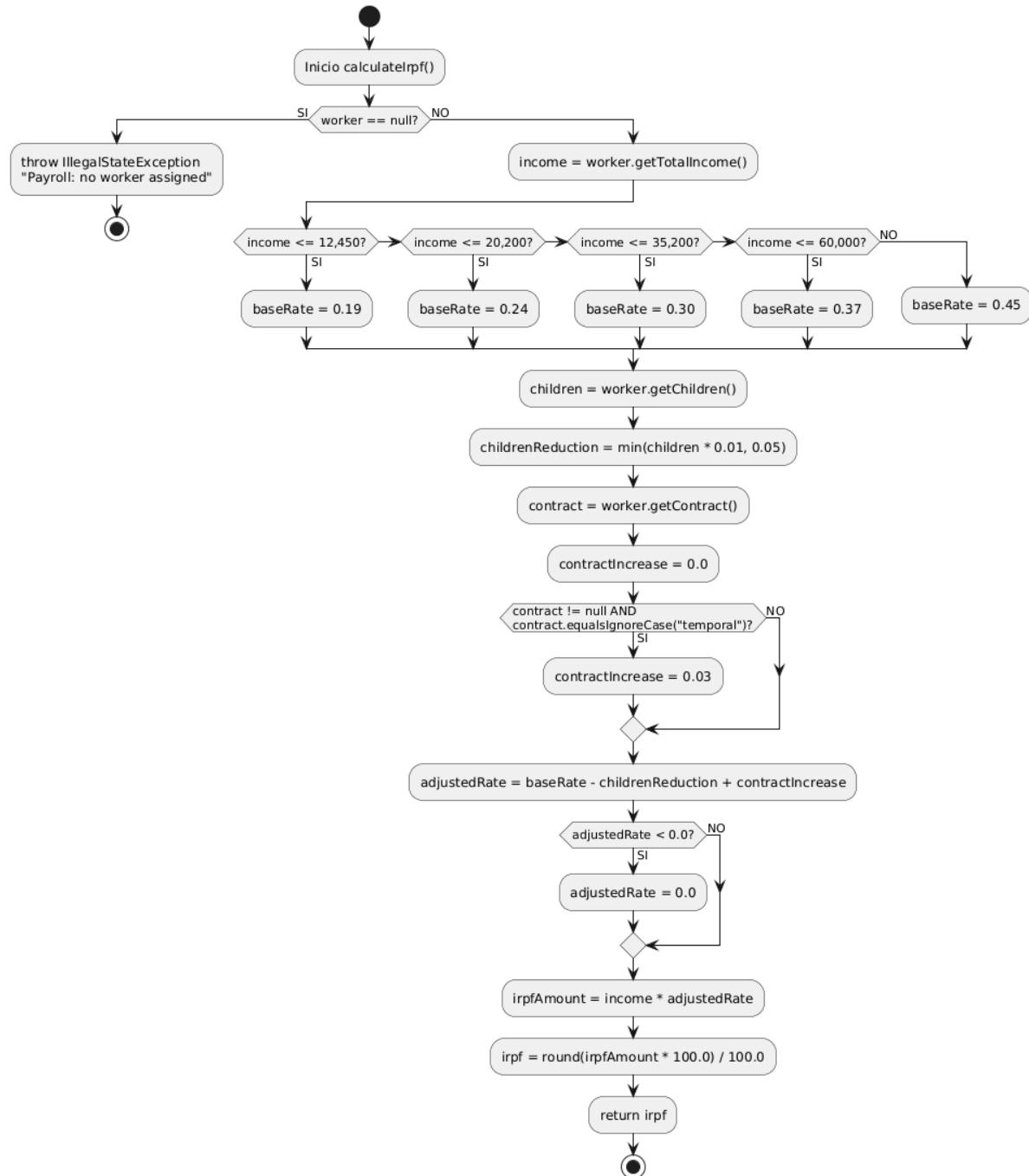
```
144
145     private boolean isValidDni(String dni) { 1 usage  ↗ Bianca
146         if (dni == null || dni.isEmpty()) {
147             return false;
148         }
149
150         dni = dni.replace( target: "-", replacement: "").trim().toUpperCase();
151
152         if (!dni.matches( regex: "^\d{8}[A-Z]$")) {
153             return false;
154         }
155
156         String numeroStr = dni.substring(0, 8);
157         char letraProporcionada = dni.charAt(8);
158
159         int numero = Integer.parseInt(numeroStr);
160         int resto = numero % 23;
161         char letraCalculada = LETRAS_DNI.charAt(resto);
162
163         return letraProporcionada == letraCalculada;
164     }
```



Método 1: calculateIrpf()

- Complejidad ciclomática: 9

```
57     public double calculateIrpf() { 24 usages  ↗ veronloz
58
59         double income = worker.getTotalIncome();
60
61         // Determinar tasa base por tramos
62         double baseRate;
63         if (income <= 12_450.0) {
64             baseRate = 0.19;
65         } else if (income <= 20_200.0) {
66             baseRate = 0.24;
67         } else if (income <= 35_200.0) {
68             baseRate = 0.30;
69         } else if (income <= 60_000.0) {
70             baseRate = 0.37;
71         } else {
72             baseRate = 0.45;
73         }
74
75         // Modificador por hijos: -1% por hijo, hasta -5%
76         int children = worker.getChildren();
77         double childrenReduction = Math.min(children * 0.01, 0.05);
78
79         // Modificador por contrato temporal: +3% si es "Temporal"
80         double contractIncrease = 0.0;
81         String contract = worker.getContract();
82         if (contract != null && contract.equalsIgnoreCase("temporal")) {
83             contractIncrease = 0.03;
84         }
85         else {
86             contractIncrease = 0.0;
87         }
88
89         // Tasa ajustada
90         double adjustedRate = baseRate - childrenReduction + contractIncrease;
91
92         // Importe IRPF anual
93         double irpfAmount = income * adjustedRate;
94
95         // Redondear a 2 decimales (céntimos)
96         irpf = Math.round(irpfAmount * 100.0) / 100.0;
97
98         return irpf;
99     }
```



Loop Testing

Loop Testing es una técnica de caja blanca que se centra en verificar la validez de las construcciones de bucles. Se evalúan cuatro categorías principales de bucles: simples, anidados, concatenados y no estructurados.

Loop Testing 1: Bucle Simple - Validación de DNI con múltiples caracteres

Aunque el método isValidDni() no tiene un bucle explícito visible, internamente el método matches() itera sobre la cadena. Por lo que decidimos crear un método auxiliar que sí tenga un bucle explícito para demostrar Loop Testing.

```
186     public int countDigits(String text) { 8 usages & veronloz
187         if (text == null || text.isEmpty()) {
188             return 0;
189         }
190
191         int count = 0;
192         for (int i = 0; i < text.length(); i++) { // BUCLE SIMPLE
193             if (Character.isDigit(text.charAt(i))) {
194                 count++;
195             }
196         }
197         return count;
198     }
```

CI

A través del uso de GitHubActions, y la configuración en un fichero yaml en el directorio workflow, se consiguió que:

- Se ejecuten los tests inmediatamente al hacer un merge con main.



The screenshot shows a GitHub Actions test run interface. At the top, there's a header with a dropdown arrow, a checkmark icon, the text "Ejecutar tests", and a timer showing "2s". Below the header is a code listing of test cases, each followed by a green checkmark indicating success. The test cases are organized into several classes: JUnit Jupiter, CompanyTest, WorkerTest, payrollTest, JUnit Vintage, and JUnit Platform Suite. The code listing starts at line 138 and ends at line 177, with the final line being "Test run finished after 174 ms".

```
138 |
139 |   JUnit Jupiter ✓
140 |     CompanyTest ✓
141 |       |   testFindByCifNotFound() ✓
142 |       |   testSaveDuplicateCif() ✓
143 |       |   testDeleteCompany() ✓
144 |       |   testCompanyConstructor() ✓
145 |       |   testSaveCompanyWithNullValues() ✓
146 |       |   testUpdateCompanySuccess() ✓
147 |       |   testSaveCompanyWithInvalidEmail() ✓
148 |       |   testUpdateCompanyInvalidValues() ✓
149 |       |   testSaveCompany() ✓
150 |       |   testFindByCif() ✓
151 |       |   testSaveCompanyWithInvalidCnae() ✓
152 |       |   testUpdateCompanyNotExisting() ✓
153 |       |   WorkerTest ✓
154 |       |         testChildren() ✓
155 |       |         testValidPayments() ✓
156 |       |         testTotalIncome() ✓
157 |       |         testValidDni() ✓
158 |       |         testUpdateEmployeeData() ✓
159 |       |         testContract() ✓
160 |       |         testDeleteWorker() ✓
161 |       |         testSaveWorker() ✓
162 |       |         testSaveAndPrintAllWorkers() ✓
163 |       |         testValidCategory() ✓
164 |       |         testCivilStatus() ✓
165 |       |         testConstructor() ✓
166 |       |   payrollTest ✓
167 |         |   testMonthlySalary() ✓
168 |         |   testSocialSecurityDeduction() ✓
169 |         |   testIrpfdeduction() ✓
170 |         |   testCalculateMonthlyNetSalary() ✓
171 |         |   testSalaryExtras() ✓
172 |         |   testCalculateNetSalary() ✓
173 |         |   testConstructor() ✓
174 |   JUnit Vintage ✓
175 |   JUnit Platform Suite ✓
176
177 Test run finished after 174 ms
```

- Se compruebe la calidad del código

The screenshot shows a CI pipeline interface. On the left, there's a sidebar with links: 'Summary', 'Jobs' (which is expanded), 'Compilar y Ejecutar...', 'Calidad de Código' (which is selected and highlighted in blue), 'Run details', 'Usage', and 'Workflow file'. The main area is titled 'Calidad de Código' and shows a successful run from 1 hour ago. It lists several steps: 'Set up job' (1s), 'Run actions/checkout@v4' (0s), 'Verificar líneas largas (máx 120)' (1s, expanded). This step contains a command: 'Run if find src tests -name "*.java" ! -name "App.java" -exec awk 'length > 120 {print FILENAME ":" NR; exit 1}' {} +; then' and a checkmark indicating 'Lineas OK (App.java excluido)'. Other collapsed steps include 'Verificar estructura MVC' (0s), 'Post Run actions/checkout@v4' (0s), and 'Complete job' (0s).

- Un error cualquiera impide un merge en main

