INFORME DE PRÁCTICAS
TEST Y CALIDAD DE SOFTWARE

Verónica Lozada Pérez
Bianca Luna

Profesor encargado: Xavier Otazu

Verónica Lozada Pérez 1639552
Bianca Luna 1676394

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 sesion si está previamente registrado por una empresa.
  - No puede registrarse ni crear una cuenta por si mismo.
  - Una vez validado, tiene acceso unicamente a sus datos y calculos 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 qué pueden tomar estos están parametrizados o limitados a ciertas opciones, tales qué de 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 debera cumplir con la cantidad de digitos 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 numero entero mayor o igual a 0 (cero).
- El salario total anual de un empleado deberá ser mayor al salario minimo 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”, “Formacion en Alternancia”, “Formativo para la Obtencion de la Práctica Profesional”
- La categoría profesional de un empleado deberá ser un número entre 1 y 10.

- El correo electronico de una empresa deberá cumplir con el patron de un correo electronico como tal.
- El teléfono de una empresa deberá contar con 8 digitos, siendo cualquier otra opcion incorrecta.
- El CNAE de una empresa deberá formar parte del listado de códigos dispuestos por las autoridades economicas nacionales.

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

Verónica Lozada Pérez 1639552
Bianca Luna 1676394

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 ejemplo[1][2]:

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.

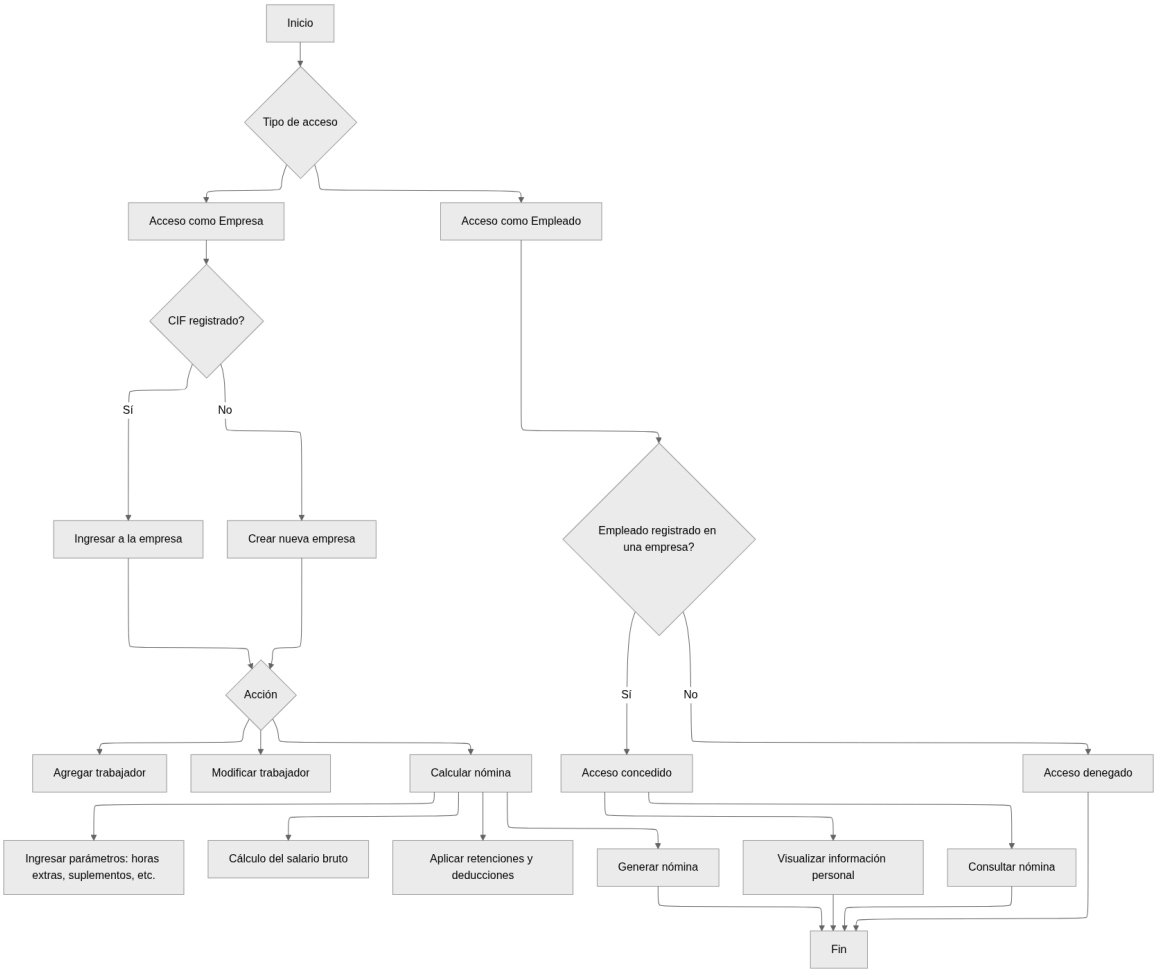
INGRESO ANUAL	RETENCIÓN
hasta 12.450	19%
12.450 a 20.199	24%
20.200 a 35.199	30%
35.200 a 59.999	37%
60.000 a 299.999	45%
más de 300.000	47%

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

Verónica Lozada Pérez 1639552
Bianca Luna 1676394

Diagrama de Flujo



Aplicación del contenido teórico

MVC

model	Worker.java, Payroll.java, Company.java
view	App
controller	CompanyRespository.java, WorkerRepositoty.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:

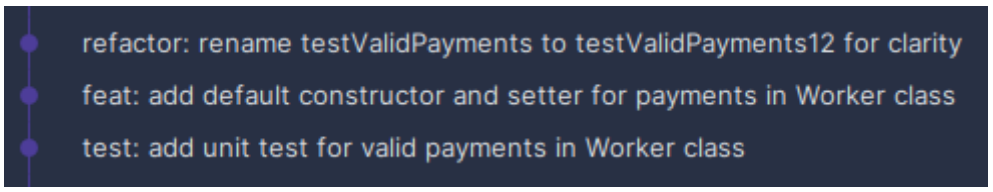
- `WorkerRepository.java`: Interfaz que define las operaciones **CRUD** (Create, Read, Update, Delete) para gestionar trabajadores: guardar, buscar por DNI, buscar por empresa y eliminar trabajadores.

- `CompanyRepository.java`: Interfaz que define las operaciones para gestionar empresas: guardar, actualizar, buscar por CIF y eliminar empresas.

TDD

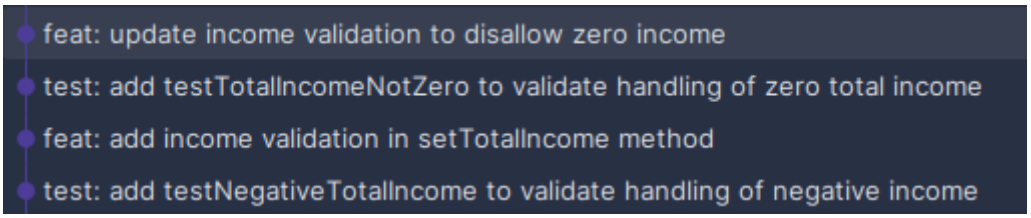
El *Test Driven Development* consiste en el desarrollo de código a partir de la generación de los casos de prueba primeros. Durante la realización de todo nuestro proyecto, tuvimos extremo cuidado en aplicar esta metodología para la confección de las clases principales del código, sobre todo aquellas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



TDD en clase Worker.java

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



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

Verónica Lozada Pérez 1639552
Bianca Luna 1676394

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 relevenate 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 (**CompanyRespositoryMock**) y la otra una base de datos con los empleados de todas las empresas (**WorkerRespositoryMock**).

CompanyRespositoryMock	
Mock que implementa la interfaz de CompanyRepository para que sea usado desde la Vista como manera de acceder y gestionar las empresas.	

```java
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
}
```

WorkerRespositoryMock	
Mock que implementa la interfaz de WorkerRespository para que sea usado desde la Vista como manera de acceder y gestionar los diferentes trabajadores de las múltiples empresas.	

```java
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
}
```

Coverage

Coverage | All in tqs_project

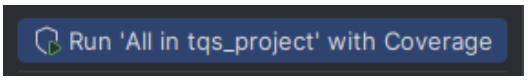
Element	Class, %	Method, %	Line, %	Branch, %
all	50% (3/6)	56% (50/88)	34% (144/412)	37% (67/179)
repository	0% (0/2)	0% (0/16)	0% (0/77)	0% (0/52)
WorkerRepositoryMock	0% (0/1)	0% (0/7)	0% (0/32)	0% (0/18)
CompanyRepositoryMock	0% (0/1)	0% (0/9)	0% (0/45)	0% (0/34)
App	0% (0/1)	0% (0/22)	0% (0/191)	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:



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 as 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

Verónica Lozada Pérez 1639552
Bianca Luna 1676394

Condición compuesta: `if (dni == null || dni.isEmpty())`
Ambas condiciones evaluadas como TRUE y FALSE

<table>
<tr><th>Test</th><th>dni == null</th><th>dni.isEmpty()</th><th>Resultado decisión</th></tr>
<tr><td rowspan="6"><code>testValidDni()</code></td><td>TRUE</td><td>-</td><td>TRUE</td></tr>
<tr><td colspan="3">

```
// 5. Asignar un DNI nulo
worker.setDni(null);
assertEquals(validDni, worker.getDni(),
    message: "El worker no debería aceptar DNIs nulos");
```

</td></tr>
<tr><td>FALSE</td><td>TRUE</td><td>TRUE</td></tr>
<tr><td colspan="3">

```
// 6. Asignar un DNI no nulo pero vacío
worker.setDni("");
assertEquals(validDni, worker.getDni(),
    message: "El worker no debería aceptar DNIs vacíos");
```

</td></tr>
<tr><td>FALSE</td><td>FALSE</td><td>FALSE</td></tr>
<tr><td colspan="3">

```
// 1. Asignar un DNI válido inicial
String validDni = "11768496V";
worker.setDni(validDni);
assertEquals(validDni, worker.getDni());
```

</td></tr>
<tr><td colspan="4">Líneas generadas por el “run with coverage”</td></tr>
</table>

```java
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: `calculateIrpf()` en Payroll
Condición compuesta: `if (contract != null && contract.equalsIgnoreCase("temporal"))`
Ambas condiciones evaluadas como TRUE y FALSE

Test	contract != null	equalsIgnoreCase ("temporal")	Resultado decisión
testIrpf Deduction()	TRUE	TRUE	TRUE

	`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.calculateIrpf(), delta: 0.01); // 22%`		

	TRUE	FALSE	FALSE

	`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.calculateIrpf(), delta: 0.01); // 40%`		

	FALSE	-	FALSE

```
// Equivalence Partition 2: grup 5-7
payroll.setWorker(cat5Worker);
assertEquals( expected: 6400.0, payroll.calculateSocialSecurity());
```

Líneas generadas por el “run with coverage”

```
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( anotherString: "temporal")) {
83        contractIncrease = 0.03;
84      }
85      else {
86        contractIncrease = 0.0;
87      }
```

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
testSocialSecurityDeduction()	0	FALSE	FALSE	FALSE
	2	TRUE	TRUE	TRUE
	5	TRUE	FALSE	FALSE

```
Exception exceptionLow = assertThrows(IllegalArgumentException.class, () -> {
  Worker invalidLow = new Worker( name: "Test", dni: "71239485K", civilStatus: "Soltero", children: 0, totalIncome:
      payments: 12, contract: "Indefinido", category: 0, cifEmpresa: "J12345678");
  payroll.setWorker(invalidLow);
  payroll.calculateSocialSecurity();
});
assertEquals( expected: "Categoria inválida. Debe estar entre 0 y 10.", exceptionLow.getMessage());
```

```
// Equivalence Partition 1: grup 1-4
payroll.setWorker(cat1Worker);
assertEquals( expected: 6350.0, payroll.calculateSocialSecurity
payroll.setWorker(cat2Worker);
assertEquals( expected: 6350.0, payroll.calculateSocialSecurity
```

```
// 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.calculateIrpf(), delta: 0.01); // 37%
```

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

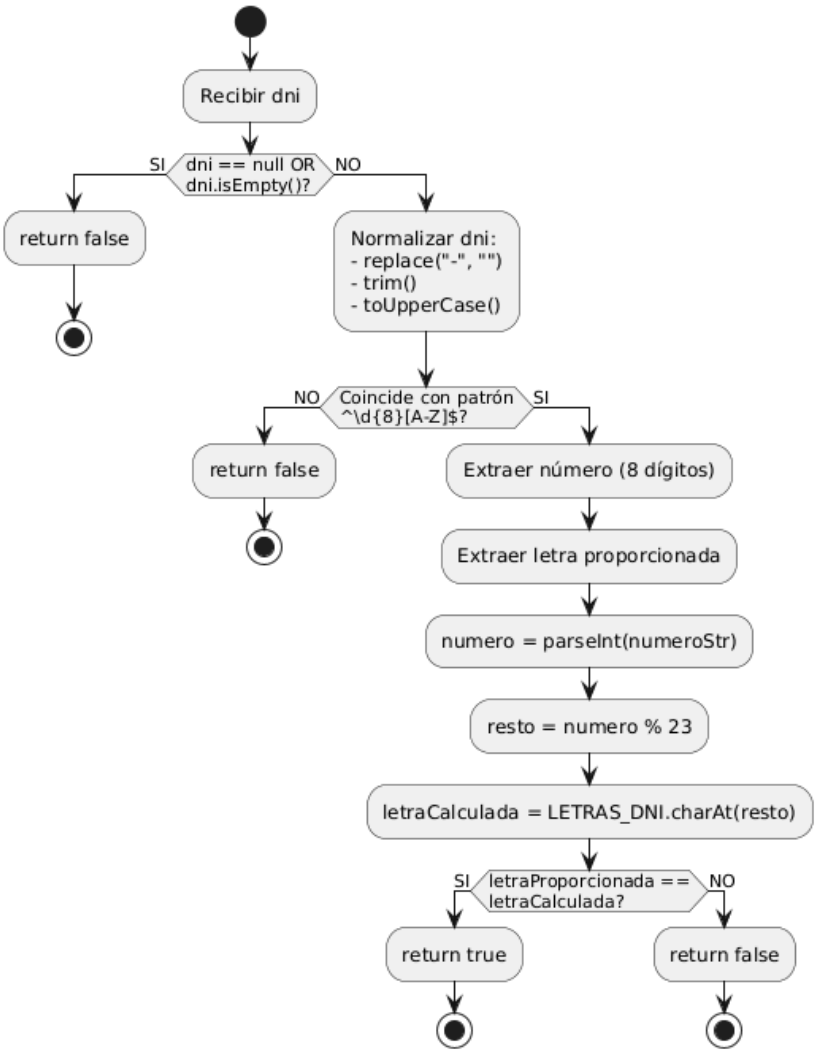
La cobertura de caminos es el criterio más exhaustivo de pruebas de caja blanca. Requiere que se ejecuten todos los caminos posibles a través del código, desde el punto de entrada hasta el punto de salida. Esto incluye todas las combinaciones posibles de decisiones.

Método 1: `isValidDni(String dni)`

- Complejidad ciclomática: 4

```
144     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 }
```

Verónica Lozada Pérez 1639552
Bianca Luna 1676394

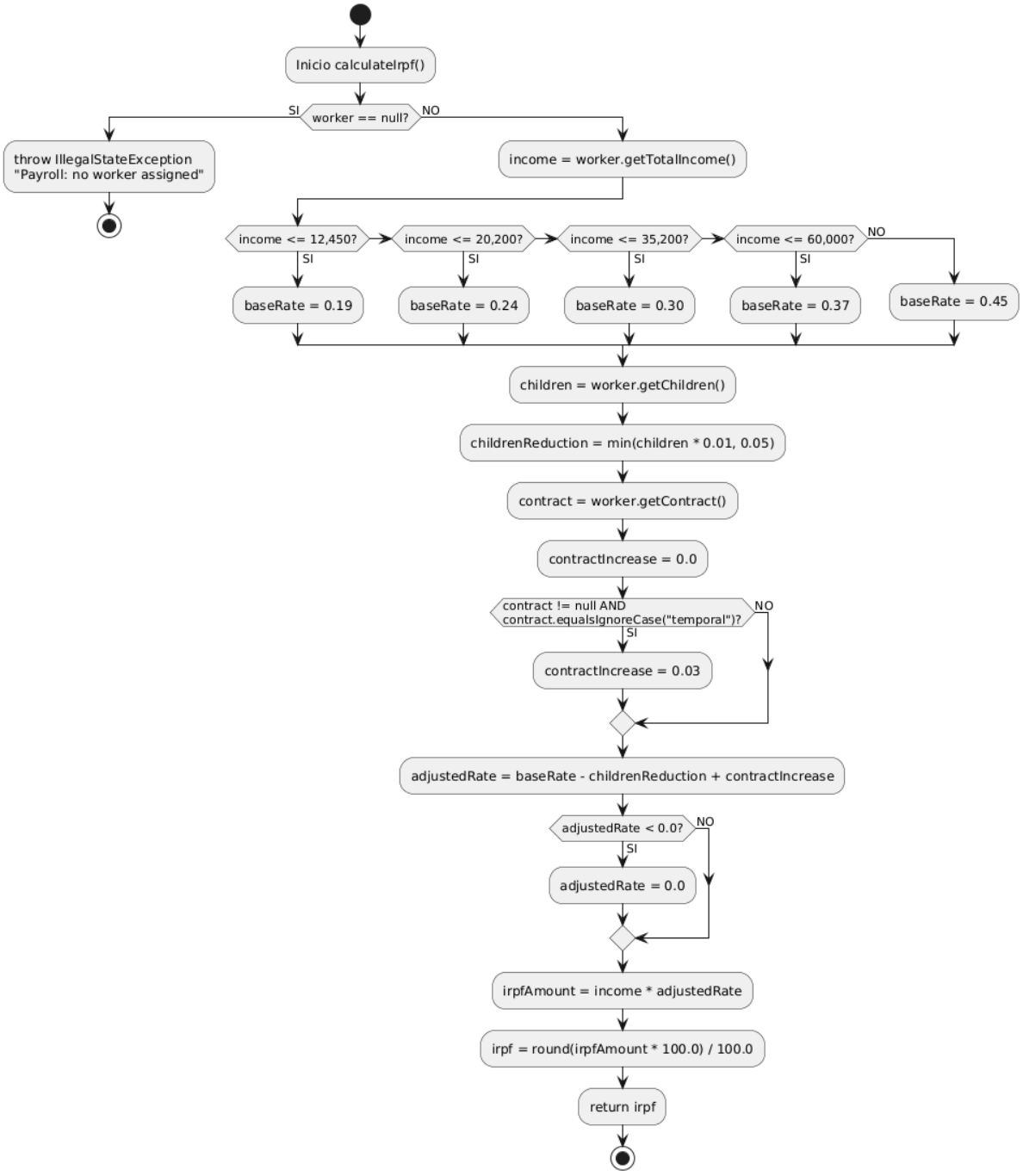


Verónica Lozada Pérez 1639552
Bianca Luna 1676394

Método 1: `calculateIrpf()`

- Complejidad ciclomática: 9

```java
public double calculateIrpf() {  24 usages  veronloz
    double income = worker.getTotalIncome();

    // Determinar tasa base por tramos
    double baseRate;
    if (income <= 12_450.0) {
        baseRate = 0.19;
    } else if (income <= 20_200.0) {
        baseRate = 0.24;
    } else if (income <= 35_200.0) {
        baseRate = 0.30;
    } else if (income <= 60_000.0) {
        baseRate = 0.37;
    } else {
        baseRate = 0.45;
    }

    // Modificador por hijos: -1% por hijo, hasta -5%
    int children = worker.getChildren();
    double childrenReduction = Math.min(children * 0.01, 0.05);

    // 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;
    }

    // Tasa ajustada
    double adjustedRate = baseRate - childrenReduction + contractIncrease;

    // Importe IRPF anual
    double irpfAmount = income * adjustedRate;

    // Redondear a 2 decimales (céntimos)
    irpf = Math.round(irpfAmount * 100.0) / 100.0;
    return irpf;
}
```



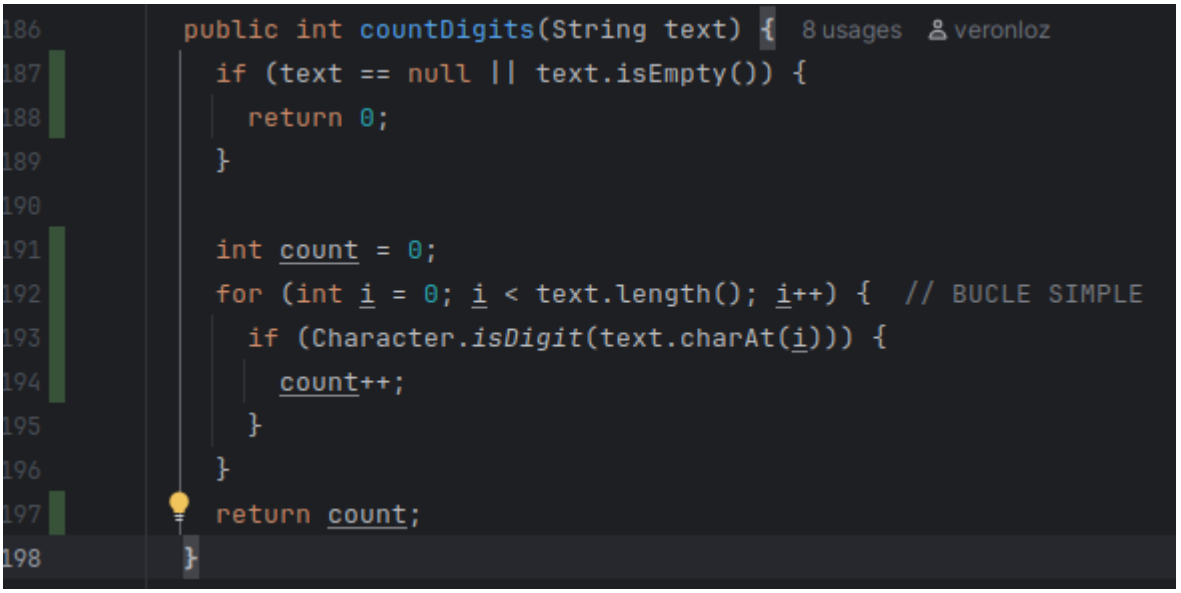
Loop Testing

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.

Verónica Lozada Pérez 1639552
Bianca Luna 1676394

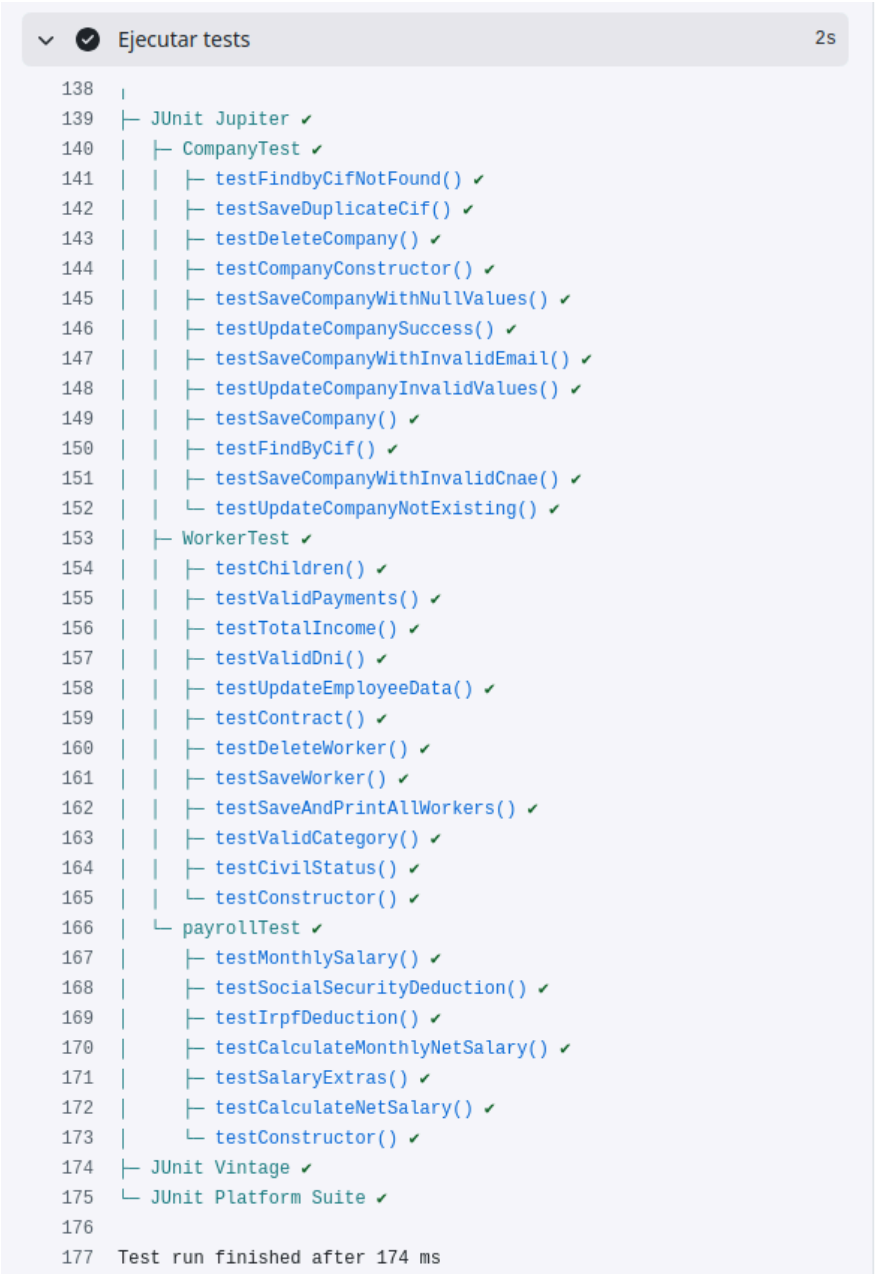


```java
public int countDigits(String text) {  8 usages  veronloz
    if (text == null || text.isEmpty()) {
        return 0;
    }

    int count = 0;
    for (int i = 0; i < text.length(); i++) {  // BUCLE SIMPLE
        if (Character.isDigit(text.charAt(i))) {
            count++;
        }
    }
    return count;
}
```

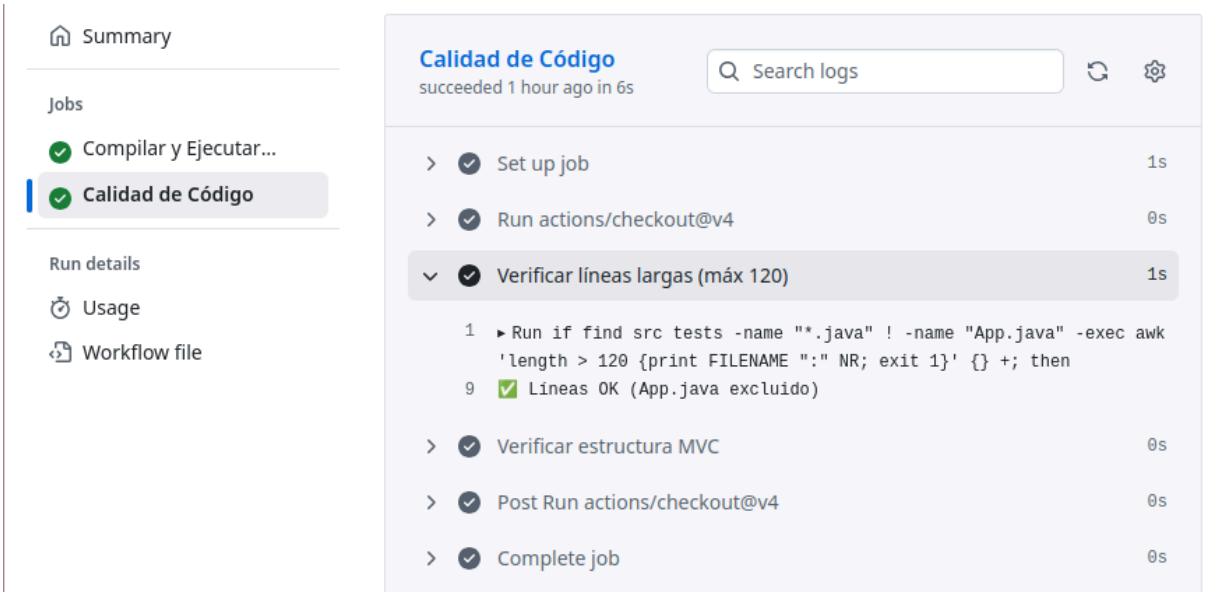
CI

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

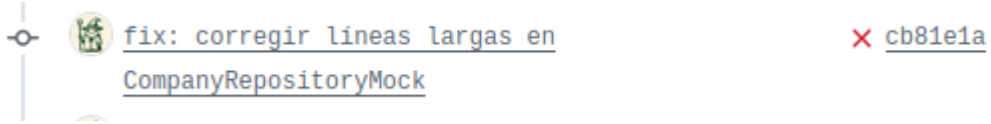
- Se ejecuten los tests inmediatamente al hacer un merge con main.



- Se compruebe la calidad del código



- Un error cualquiera impide un merge en main



Worker

```
public class Worker {

  private static final String LETRAS_DNI = "TRWAGMYFPDXBNJZSQVHLCKE";

  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;


public Worker()
public Worker(String name, String dni,  String civilStatus, int
children, float totalIncome,
                int payments, String contract, int category, String
cifEmpresa)
```

Funcionalidad:	Constructores por defecto y paramétrico de la clase Worker con los atributos necesarios para hacer los cálculos pertinentes a la nómina del trabajador..
Localización:	src/model/Worker.java – Worker() / Worker(String name, String dni, String civilStatus, int children, float totalIncome, int payments, String contract, int category, String cifEmpresa)
Test:	test/WorkerTest.java – testConstructor()
Tipos de test:	Se han utilizado tests de caja negra para comprobar que las salidas correspondían con las esperadas al realizar las asignaciones.
Comentarios:	

public void setPayments(int payments)

Funcionalidad:	Establece el número de pagas anuales. Valida que el valor sea estrictamente 12 o 14. Lanza IllegalArgumentException si no se cumple.
Localización:	model.Worker.java, public void setPayments(int payments)

Test:	WorkerTest.java, testValidPayments
Tipos de test:	Tipo: Test Unitario, Test de Caja Blanca. Técnicas utilizadas: Particiones Equivalentes: Probando la partición válida (12, 14) y la partición inválida (e.g., 11, 13, 15). Análisis de Valores Límite: Se prueban los límites 11, 12, 13, 14, 15. Manejo de Excepciones: Se verifica que se lanza IllegalArgumentException con el mensaje correcto.
Comentarios:	Implementación directa de una regla de negocio binaria (12 o 14 pagas).

public boolean isValidIncome(float income)

Funcionalidad:	Función auxiliar que verifica si el salario introducido es un valor positivo.
Localización:	model.Worker.java, public boolean isValidIncome(float income)
Test:	WorkerTest.java, testTotalIncome()
Tipos de test:	Tipo: Test Unitario, Test de Caja Blanca. Técnicas utilizadas: Cobertura de Sentencias (verifica el retorno true o false según la condición income > 0).
Comentarios:	El comentario del código indica que la validación del mínimo legal de 16000 € se ha eliminado aquí para propósitos de prueba y se asume en una capa superior (App.java), aunque el test testTotalIncome sí comprueba el valor 15999.99f.

public void setTotalIncome(float totalIncome)

Funcionalidad:	Establece el ingreso total del trabajador. Valida que el ingreso sea positivo mediante isValidIncome(). Lanza IllegalArgumentException si no se cumple.
Localización:	model.Worker.java, public void setTotalIncome(float totalIncome)
Test:	WorkerTest.java, testTotalIncome
Tipos de test:	Tipo: Test Unitario, Test de Caja Blanca. Técnicas utilizadas: Particiones Equivalentes: Probando valores positivos (válido) vs. cero y negativos (inválido). Análisis de Valores Límite: Pruebas cerca de cero (e.g., -0.01, 0, 0.01).

	Manejo de Excepciones: Se verifica que se lanza IllegalArgumentException para los casos de salario no positivo.
Comentarios:	Asegura la coherencia de que el salario no puede ser negativo o cero.

public void setChildren(int children) (y su método isValidChildren)

Funcionalidad:	Establece el número de hijos a cargo. Valida que el número sea un entero no negativo (>= 0) mediante isValidChildren(). Lanza IllegalArgumentException si se intenta asignar un valor negativo.
Localización:	model.Worker.java, public void setChildren(int children) y public boolean isValidChildren(int children).
Test:	WorkerTest.java, testChildren
Tipos de test:	Tipo: Test Unitario, Test de Caja Blanca. Técnicas utilizadas: Particiones Equivalentes: Probando valores no negativos (válido) vs. negativos (inválido). Análisis de Valores Límite: Pruebas en -1, 0, y 1. Manejo de Excepciones: Se verifica el lanzamiento de IllegalArgumentException.
Comentarios:	Asegura la coherencia de que el salario noGarantiza que el número de hijos es un valor lógicamente posible.puede ser negativo o cero.

public void setCategory(int category) (y su método isValidCategory)

Funcionalidad:	Establece la categoría profesional del trabajador. Valida que el valor sea un entero en el rango $[0, 10]$ (ambos inclusive) mediante isValidCategory(). Lanza IllegalArgumentException si se intenta asignar un valor fuera de este rango.
Localización:	model.Worker.java, public void setCategory(int category) y public boolean isValidCategory(int category).
Test:	WorkerTest.java, testValidCategory
Tipos de test:	Tipo: Test Unitario, Test de Caja Blanca. Técnicas utilizadas:

	Particiones Equivalentes: Probando la partición válida (1 a 10) y las inválidas (menores a 0 o mayores a 10). Análisis de Valores Límite: Pruebas en los límites externos (-1, 11) e internos (0, 1, 10). Manejo de Excepciones: Se verifica el lanzamiento de IllegalArgumentException.
Comentarios:	Implementa la restricción del rango de categorías profesionales definido.

public void setContract(String contract) (y su método isValidContract)

Funcionalidad:	Establece el tipo de contrato. Valida que el contrato sea uno de los cuatro tipos predefinidos ("Indefinido", "Temporal", "Formacion en Alternancia", o "Formativo para la Obtencion de la Práctica Profesional") mediante isValidContract(). Lanza IllegalArgumentException si el valor es nulo o no coincide con los tipos permitidos.
Localización:	model.Worker.java, public void setContract(String contract) y public boolean isValidContract(String contract).
Test:	WorkerTest.java, testContract
Tipos de test:	Tipo: Test Unitario, Test de Caja Blanca. Técnicas utilizadas: Particiones Equivalentes: Probando todos los valores válidos de la lista cerrada y un valor inválido ("permanent"). Cobertura de Decisiones: Asegura que cada cláusula de la condición || en isValidContract se evalúa. Manejo de Excepciones: Se verifica el lanzamiento de IllegalArgumentException.
Comentarios:	Aplica una validación de lista cerrada (Enumeración de Contratos) sensible a mayúsculas/minúsculas.

public void setCivilStatus(String civilStatus) (y su método isCivilStatusValid)

Funcionalidad:	Establece el estado civil del trabajador. Valida que el estado civil sea uno de los cuatro tipos predefinidos ("Soltero", "Casado", "Divorciado", o "Viudo") mediante isCivilStatusValid(). Lanza IllegalArgumentException si el valor es nulo o no coincide con los tipos permitidos.

Localización:	model.Worker.java, public void setCivilStatus(String civilStatus) y public boolean isCivilStatusValid(String civilStatus)
Test:	WorkerTest.java, testCivilStatus
Tipos de test:	Tipo: Test Unitario, Test de Caja Blanca. Técnicas utilizadas: Particiones Equivalentes: Probando un valor válido y un valor inválido ("Hijo"). Cobertura de Decisiones: Asegura que cada cláusula de la condición || en isCivilStatusValid se evalúa. Manejo de Excepciones: Se verifica el lanzamiento de IllegalArgumentException.
Comentarios:	Aplica una validación de lista cerrada (Enumeración de Estados Civiles) sensible a mayúsculas/minúsculas.

public void setDni(String dni) (y su método isValidDni)

Funcionalidad:	Establece el DNI del trabajador. Valida la corrección del DNI mediante el método privado isValidDni. La validación comprueba: No nulo o vacío. Formato (8 dígitos seguidos de 1 letra: ^\\d{8}[A-Z]$). Cálculo de la letra (utiliza la tabla LETRAS_DNI con el módulo 23). Si el DNI es válido, se asigna. Si es inválido, imprime un error y no modifica el DNI actual.
Localización:	model.Worker.java, public void setDni(String dni) y private boolean isValidDni(String dni)
Test:	WorkerTest.java, testValidDni
Tipos de test:	Tipo: Test Unitario, Test de Caja Blanca (para isValidDni). Técnicas utilizadas: Análisis de Valores Límite: Pruebas con formato incorrecto, letra calculada incorrecta y valores nulos/vacíos. Particiones Equivalentes: DNI válido vs. DNI con letra incorrecta vs. DNI con formato incorrecto. Test de Mutación: Se prueba que un DNI inválido no sobreescribe el DNI válido preexistente.

Comentarios:	La lógica de validación del DNI es la más compleja, utilizando la tabla estándar de la policía española (TRWAGMYFPDXBNJZSQVHLCKE). El setter no lanza excepción, sino que no asigna el valor y notifica un error por consola (System.err).

public int countDigits(String text)

Funcionalidad:	Método auxiliar para contar el número de caracteres que son dígitos dentro de una cadena. Se utiliza principalmente para probar la cobertura del bucle for.
Localización:	model.Worker.java, public int countDigits(String text)
Test:	WorkerTest.java,testLoopCountDigitsZeroIterations, testLoopCountDigitsMaximum
Tipos de test:	Tipo: Test Unitario, Test de Cobertura de Bucles (Loop Testing). Técnicas utilizadas: Análisis de Bucle Simple: Cobertura de los siguientes casos del bucle for: Omitir el bucle (0 iteraciones: "" o null). Una iteración ("5"). Dos iteraciones ("5A"). Valor típico (8 iteraciones: "12345678"). Valor muy grande (100 iteraciones).
Comentarios:	Este método no forma parte de la lógica de negocio central, sino que ha sido añadido expresamente para demostrar la cobertura de las técnicas de Loop Testing sobre la estructura de control for.

Payroll

```
public class Payroll {
  public String payrollCode;
  Worker worker;
  public float annualGrossSalary; // worker.getTotalIncome()
  public double netSalary; // annualGrossSalary - deductions
  public double irpf;
  public double sgs;


public Payroll()
public Payroll(String payrollCode, Worker worker)
public void setWorker(Worker worker)
```

Funcionalidad:	Constructores por defecto y paramétrico, conjuntamente con el setter de un trabajador, de la clase Payroll para configurar los atributos internos auxiliares que se requieren para el cálculo de una nómina.
Localización:	src/model/Payroll.java – Payroll() / Payroll(String payrollCode, Worker worker) / setWorker(Worker worker)
Test:	test/PayrollTest.java – testConstructor() / testEmptyConstructor() / testSetWorker()
Tipos de test:	Se han utilizado tests de caja negra para comprobar que las salidas correspondían con las esperadas al realizar las asignaciones.
Comentarios:	

```
public double paymentsPerMonth()
```

Funcionalidad:	Cálculo de los pagos mensuales en bruto de un trabajador.
Localización:	src/model/Payroll.java – paymentsPerMonth()
Test:	test/PayrollTest.java – testMonthlySalary()

Tipos de test:	Se han utilizado tests de caja negra para comprobar que las salidas correspondían con las esperadas y que los números no se redondeaban incorrectamente.
Comentarios:	

```
public double calculateSocialSecurity()
```

Funcionalidad:	Cálculo de la deducción a realizar por parte de la Seguridad Social.
Localización:	src/model/Payroll.java – calculateSocialSecurity()
Test:	test/PayrollTest.java – testSocialSecurityDeduction()
Tipos de test:	Se han utilizado tests de caja negra para comprobar que las salidas correspondían con las esperadas. Se han implementado test de particiones equivalentes usando trabajadores de diferentes categorías, comprobando en cada caso el correcto funcionamiento de los valores límites y fronteras. Además de test de caja blanca como statement, path y decision coverage.
Comentarios:	Cabe destacar que hay comprobaciones que se realizan en la clase Worker que se propagan a Payroll y, por lo tanto, no es necesario volver a evaluar.

```
46      public double calculateSocialSecurity() {  11 usages  veronloz
47          float percentage = 0;
48
49          if (worker.getCategory() >= 1 && worker.getCategory() <= 4) percentage = 6.35f;
50          else if (worker.getCategory() >= 5 && worker.getCategory() <= 7) percentage = 6.40f;
51          else if (worker.getCategory() >= 8 && worker.getCategory() <= 11) percentage = 6.45f;
52          sgs = Math.round(worker.getTotalIncome() * (percentage / 100));
53
54          return sgs;
55      }
```

```
public double calculateIrpf()
```

Funcionalidad:	Cálculo de la deducción a realizar por parte del IRPF.
Localización:	src/model/Payroll.java – calculateIrpf()
Test:	test/PayrollTest.java – testIrpfDeduction()
Tipos de test:	Se han utilizado tests de caja negra para comprobar que las salidas correspondían con las esperadas. Se han implementado test de particiones

equivalentes usando el caso base, sin hijos y contrato indefinido. Comprobando, a su vez, que no hubiera fallos en los valores límites y frontera. Luego se hizo comprobaciones con los casos de 1,2, 3 y 5 hijos. Después comprobaciones del bonus por contrato temporal. Por último se realizó un pairwaise testing haciendo combinación de estos tres parámetros de entrada. También se validó tanto el statement, como el path como el decision coverage.

Comentarios:

```java
public double calculateIrpf() {  25 usages  veronloz *
    // Obtener salario bruto anual (Worker almacena totalIncome como float)
    double income = worker.getTotalIncome();
    // Determinar tasa base por tramos
    double baseRate;
    if (income <= 12_450.0) {
        baseRate = 0.19;
    } else if (income <= 20_200.0) {
        baseRate = 0.24;
    } else if (income <= 35_200.0) {
        baseRate = 0.30;
    } else if (income <= 60_000.0) {
        baseRate = 0.37;
    } else {
        baseRate = 0.45;
    }
    // Modificador por hijos: -1% por hijo, hasta -5%
    int children = worker.getChildren();
    double childrenReduction = Math.min(children * 0.01, 0.05);

    // 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;
    }

    // Tasa ajustada
    double adjustedRate = baseRate - childrenReduction + contractIncrease;

    // Importe IRPF anual
    double irpfAmount = income * adjustedRate;

    // Redondear a 2 decimales (céntimos)
    irpf = Math.round(irpfAmount * 100.0) / 100.0;
    return irpf;
}
```

`public double calculateNetSalary()`

Funcionalidad:	Cálculo del salario neto anual de un trabajador a partir de los atributos internos de la clase Payroll y del cálculo auxiliar de las deducciones.
Localización:	src/model/Payroll.java – calculateNetSalary()
Test:	test/PayrollTest.java – testCalculateNetSalary()
Tipos de test:	Se han utilizado tests de caja negra para comprobar que las salidas correspondían con las esperadas. Se han implementado test de particiones equivalentes usando el caso base, de un trabajador sin hijos, luego de uno con hijos y contrato temporal, y de un trabajador con categoría alta y muchos hijos. También se validó tanto el statement, path y decision coverage.
Comentarios:	

```
98   public double calculateNetSalary() {  5 usages  veronloz
99     irpf = calculateIrpf();
100    sgs = calculateSocialSecurity();
101
102    netSalary = annualGrossSalary - (irpf + sgs);
103    return Math.round(netSalary * 100.0) / 100.0;
104  }
```

`public double calculateMonthlyNetSalary()`

Funcionalidad:	Cálculo del salario neto mensual de un trabajador a partir de los atributos internos de la clase Payroll y del cálculo auxiliar de las deducciones.
Localización:	src/model/Payroll.java – calculateMonthlyNetSalary()
Test:	test/PayrollTest.java – testCalculateMonthlyNetSalary()
Tipos de test:	Se han utilizado tests de caja negra para comprobar que las salidas correspondían con las esperadas. Se han implementado test de particiones equivalentes usando casos de trabajadores con diferentes pagas. También se validó tanto el statement, path y decision coverage.
Comentarios:	

```
106     public double calculateMonthlyNetSalary() {  5 usages  veronloz
107         double netAnnual = calculateNetSalary();
108         int payments = worker.getPayments();
109
110         double monthly = netAnnual / payments;
111         return Math.round(monthly * 100.0) / 100.0;
112     }
```

Company

```
public Company (String name, String cif, String email, String phone,
String address, String website, String cnae) {}
```

Funcionalidad:	Inicialización de la entidad Company. Establece todos los atributos de la empresa (name, cif, email, phone, address, website, cnae) al crear una nueva instancia.
Localización:	src/model/Company.java – Company(String name, String cif, String email, String phone, String address, String website, String cnae).
Test:	test/CompanyTest.java – testConstructor_ValidCompany()
Tipos de test:	Test Unitario, Test de Caja Negra. Técnicas utilizadas: Verificación directa de la asignación de valores a los atributos (prueba de la postcondición de la construcción).
Comentarios:	Este constructor es la forma principal de crear objetos Company. Actualmente, no invoca a los setters, por lo que las validaciones que existan en los setters (como la del CIF) no se ejecutan durante la construcción.

```
public String getName()
public String getCif()
public String getEmail()
public String getPhone()
public String getAddress()
public String getWebsite()
public String getCnae()
```

Funcionalidad:	Los getters de la clase sirvem para obtener los atributos que otrosmetodos puedan necesitar de la clase.
Localización:	src/model/Company.java – getName() / getCif() / getEmail()/ getPhone()/ getAddress() / getWebsite()/ getCnae()
Test:	test/CompanyTest.java – testGetName() / testGetCif() / testGetEmail() / testGetPhone() / testGetAddress() / testGetWebsite / testGetCnae()
Tipos de test:	Test unitarios para la comprobación dela devolución correcta de los valores asignados.

Comentarios:

Class	Class, %	Method, %	Branch, %	Line, %
Company	100% (1/1)	100% (15/15)	100% (4/4)	100% (24/24)

```
public void setName(String name)
```

Funcionalidad:	Los setters de la clase sirven para asignar valores a los atributos que de la clase con los que más adelante se realizarán asociaciones y cálculos. El setName() ha de cumplir con ciertos requisitos, de ahí el testing realizado.
Localización:	src/model/Company.java – setName(String name)
Test:	test/CompanyTest.java – testSetName_ValidName() / testSetName_EmptyName() / testSetName_NullName()
Tipos de test:	Test unitarios para la comprobación de la asignación correcta de los múltiples valores. También la justificación de los múltiples tests viene por la búsqueda de tener el statement y el branch coverage de la función.

Comentarios:

```
public void setCif(String cif)
```

Funcionalidad:	Los setters de la clase sirven para asignar valores a los atributos que de la clase con los que más adelante se realizarán asociaciones y cálculos. El setCif(String cif) ha de cumplir con ciertos requisitos, de ahí el testing realizado.
Localización:	src/model/Company.java – setName(String name)
Test:	test/CompanyTest.java – testSetCif_ValidCif() / testSetCif_NullCif_ThrowsException() / testSetCif_EmptyCif_ThrowsException() / testSetCif_BlankCif_ThrowsException() /

	testSetCif_MultipleChanges / testSetCif_ConditionCoverage()
Tipos de test:	Test unitarios para la comprobación de la asignación correcta de los múltiples valores. También la justificación de los múltiples tests viene por la búsqueda de tener el branch coverage de la función.
Comentarios:	

```
public void setEmail(String email)
```

Funcionalidad:	Los setters de la clase sirven para asignar valores a los atributos que de la clase con los que más adelante se realizarán asociaciones y cálculos. El setEmail(String email) ha de cumplir con ciertos requisitos, de ahí el testing realizado.
Localización:	src/model/Company.java – setEmail(String email)
Test:	test/CompanyTest.java – testSetEmail_ValidEmail() / testSetEmail_EmailWithSubdomains() / testSetEmail_NullEmail() / testSetEmail_EmptyEmail()
Tipos de test:	Test unitarios para la comprobación de la asignación correcta de los múltiples valores. También la justificación de los múltiples tests viene por la búsqueda de tener el statement coverage de la función.
Comentarios:	

```
public void setPhone(String phone)
```

Funcionalidad:	Los setters de la clase sirven para asignar valores a los atributos que de la clase con los que más adelante se realizarán asociaciones y cálculos. El setPhone(String phone) ha de cumplir con ciertos requisitos, de ahí el testing realizado.

Localización:	src/model/Company.java – setEmail(String email)
Test:	test/CompanyTest.java – testSetPhone_ValidPhone() / testSetPhone_NullPhone()
Tipos de test:	Test unitarios para la comprobación de la asignación correcta de los múltiples valores. También la justificación de los múltiples tests viene por la búsqueda de tener el statement coverage de la función.
Comentarios:	

```
public void setAddress(String address)
```

Funcionalidad:	Los setters de la clase sirven para asignar valores a los atributos que de la clase con los que más adelante se realizarán asociaciones y cálculos. El setAddress(String address) ha de cumplir con ciertos requisitos, de ahí el testing realizado.
Localización:	src/model/Company.java – setAddress(String address)
Test:	test/CompanyTest.java – testSetAddress_ValidAddress() / testSetAddress_NullAddress()
Tipos de test:	Test unitarios para la comprobación de la asignación correcta de los múltiples valores. También la justificación de los múltiples tests viene por la búsqueda de tener el statement coverage de la función.
Comentarios:	

```
public void setWebsite(String website)
```

Funcionalidad:	Los setters de la clase sirven para asignar valores a los atributos que de la clase con los que más adelante se realizarán asociaciones y cálculos. El setWebsite(String website) ha de cumplir con ciertos requisitos, de ahí el testing realizado.
Localización:	src/model/Company.java – setWebsite(String website)

Test:	test/CompanyTest.java – testSetWebsite_ValidWebsite() / testSetWebsite_NullWebsite()
Tipos de test:	Test unitarios para la comprobación de la asignación correcta de los múltiples valores. También la justificación de los múltiples tests viene por la búsqueda de tener el statement coverage de la función.
Comentarios:	

```
public void setCnae(String cnae)
```

Funcionalidad:	Los setters de la clase sirven para asignar valores a los atributos que de la clase con los que más adelante se realizarán asociaciones y cálculos. El setCnae(String cnae) ha de cumplir con ciertos requisitos, de ahí el testing realizado.
Localización:	src/model/Company.java – setCnae(String cnae)
Test:	test/CompanyTest.java – testSetCnae_ValidCnae() / testSetCnae_NullCnae()
Tipos de test:	Test unitarios para la comprobación de la asignación correcta de los múltiples valores. También la justificación de los múltiples tests viene por la búsqueda de tener el statement coverage de la función.
Comentarios:	