P3

Francisco Javier Hernández Martín, Jose Luis Bueno Pachón, Carlos Marín Corbera, Carmen González Ortega, Altair Bueno Calvente

UNIVERSIDAD
DE MÁLAGA

1 jan 2022

Contents

Preámbulo

Para la realización de esta práctica, hemos optado por utilizar un estilo de programación para el lenguaje de programación Java estandarizado, conocido como **Google Java Style Guide**[1]. Además, para verificar el correcto funcionamiento de la implementación de los distintos ejercicios, hemos creado una serie de test unitarios usando el framework jUnit5[2].

Ejercicio 1: Expedientes médicos

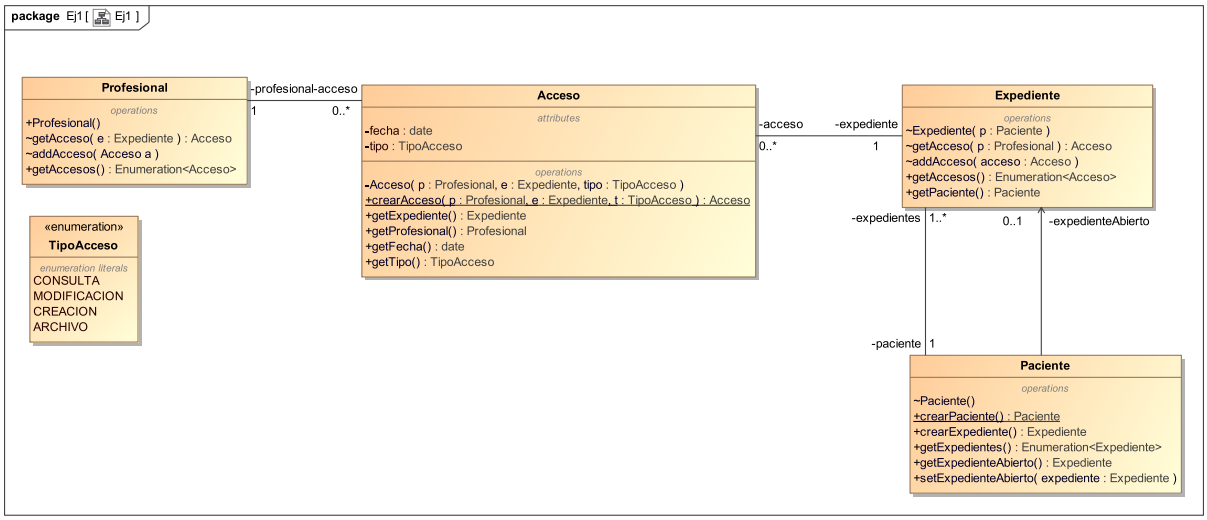

Figure 1: Modelo de diseño para Java en Magic Draw

Para la creación del modelo de diseño hemos usado la mayor parte del modelo de dominio, modificando aquellos elementos que no puedan ser directamente traducidos a Java o que pongan en peligro la consistencia del modelo.

La clase asociación Acceso ha sido reificada al ser la única forma de trabajar con clases asociación en Java. Para mantener el mismo comportamiento descrito en el modelo de dominio, hemos bloqueado el acceso al constructor y ofrecido en su lugar el método público estático `crearAcceso()`. Este método se encargará de actualizar los datos en un acceso ya existente o bien de crear una nueva instancia de la clase Acceso, emulando el comportamiento esperado de una clase asociación. El resto de la interfaz pública está compuesta unicamente por *getters* para la consulta de la fecha, tipo, profesional asociado y expediente asociado.

[1] https://google.github.io/styleguide/javaguide.html
[2] https://junit.org/junit5/

La clase Profesional no presenta grandes cambios, tan solo hemos limitado su interfaz pública a lo esencial: un constructor sin argumentos para poder instanciar la clase y un *getter* para consultar los todos los accesos.

La clase Expediente ha sido restringida a visibilidad de paquete para evitar ser instanciada fuera del paquete, siendo necesario el método `crearExpediente()` de Paciente para poder crear nuevos expedientes.Por último, su interfaz pública solo está compuesta de *getters* que permiten realizar consultas sobre los accesos existentes y el paciente al con los que dicho expediente está asociado.

La clase Paciente tampoco tiene su constructor visible, sino que utiliza el método estático `crearPaciente()` para generar nuevas instancias. Esto es así para garantizar la consistencia del modelo: no podemos tener pacientes sin ningún expediente, por lo que este método se encarga de generar un primer expediente inicial y asociarlo al paciente. También se ofrece el método `crearExpediente()`, encargado de crear un nuevo expediente para el paciente actual y asegurar la consistencia del modelo. El resto de la interfaz pública está compuesta por *getters* para consultar el expediente abierto y todos los expedientes actuales.

Valoramos la posibilidad de utilizar un patrón **factoría abstracta** con una clase Hospital, encargada de los métodos `crearPaciente()` y `crearExpediente()`. Terminamos descartando esta idea en favor del patrón **método fábrica**, ya que los expedientes y los pacientes tienen alto acoplamiento y no consideramos necesario introducir otra clase.

Clase Profesional

Interfaz pública

- `Profesional()`/*{Constructor}*.
- `getAccesos()`/*{Getter}*: Devuelve una enumeración[3] que itera sobre todos los accesos actuales que tiene el profesional.

Interfaz visible desde paquete

- `getAcceso()`/*{Getter}*: Dado un expediente no nulo, busca cuales de todos sus accesos le relaciona con dicho expediente y lo devuelve. En caso de no existir dicho acceso devolverá `null`.
- `addAcceso()`: Registra el acceso no nulo recibido como parámetro en su colección de accesos.

[3] `java.util.Enumeration<T>` es una colección inmutable de Java

Interfaz privada

- `acceso`/*{Attributes}*: Conjunto de accesos a los que el profesional tiene acceso.

Clase Acceso

Interfaz pública

- `getExpediente()`/*{Getter}*: Devuelve el expediente asociado.
- `getProfesional()`/*{Getter}*: Devuelve el profesional asociado.
- `getFecha()`/*{Getter}*: Devuelve la fecha de registro.
- `getTipo()`/*{Getter}*: Devuelve el tipo de acceso.
- `crearAcceso()`/*{Static}*: Permite crear o actualizar los datos de acceso entre un profesional y un expediente. En el caso de que ya existiera un expediente entre ambas instancias, se actualiza la fecha y el tipo de acceso y se devuelve. En caso de no existir acceso previo, se crea una instancia de Acceso y se asocia al profesional y al expediente.

Interfaz privada

- `Acceso()`/*{Constructor}*.
- `fecha`/*{Attributes}*: Fecha de registro.
- `tipo`/*{Attributes}*: Tipo de acceso.
- `expediente`/*{Attributes}*: Expediente asociado.
- `profesional`/*{Attributes}*: Profesional asociado.

Clase Expediente

Interfaz pública

- `getAccesos()`/*{Getter}*: Devuelve una enumeración que itera sobre todos los accesos existentes sobre el expediente.
- `getPaciente()`/*{Getter}*: Devuelve el paciente al que pertenece el expediente.

Interfaz visible desde paquete

- `Expediente()`/*{Constructor}*.

- `getAcceso()`/*{Getter}*: Dado un profesional no nulo, busca cuales de todos sus accesos le relaciona con dicho profesional y lo devuelve. En caso de no existir dicho acceso devolverá `null`.
- `addAcceso()`: Registra el acceso no nulo recibido como parámetro en su colección de accesos.

Clase Paciente

Interfaz pública

- `crearExpediente()`: Crea un nuevo expediente y lo asocia al paciente actual. Establece dicho expediente como el expediente abierto. Devuelve la instancia de expediente creada.
- `getExpedientes()`/*{Getter}*: Devuelve una enumeración que itera sobre todos los expedientes asociados al paciente.
- `getExpedienteAbierto()`/*{Getter}*: Devuelve el expediente abierto.
- `setExpedienteAbierto()`/*{Setter}*: Sustituye el expediente abierto por el recibido como parámetro. El expediente debe o bien ser `null` (sin expediente abierto) o un expediente del mismo paciente.
- `crearPaciente()`/*{Static}*: para crear nuevos pacientes. Se encarga de instanciar un nuevo paciente, instanciar un nuevo expediente para el paciente y asignarlo como expediente abierto del paciente. Devuelve la instancia de paciente creada.

Interfaz visible desde paquete

- `Paciente()`/*{Constructor}*.

Interfaz privada

- `expedienteAbierto`/*{Attributes}*: Último expediente asociado al paciente.
- `expedientes`/*{Attributes}*: Conjunto de expedientes del paciente.

Enumeración TipoAcceso

Enumeración compuesta de las siguientes variantes: `Consulta`, `Modificacion`, `Creacion` y `Archivo`.

Ejercicio 2: Trabajadores

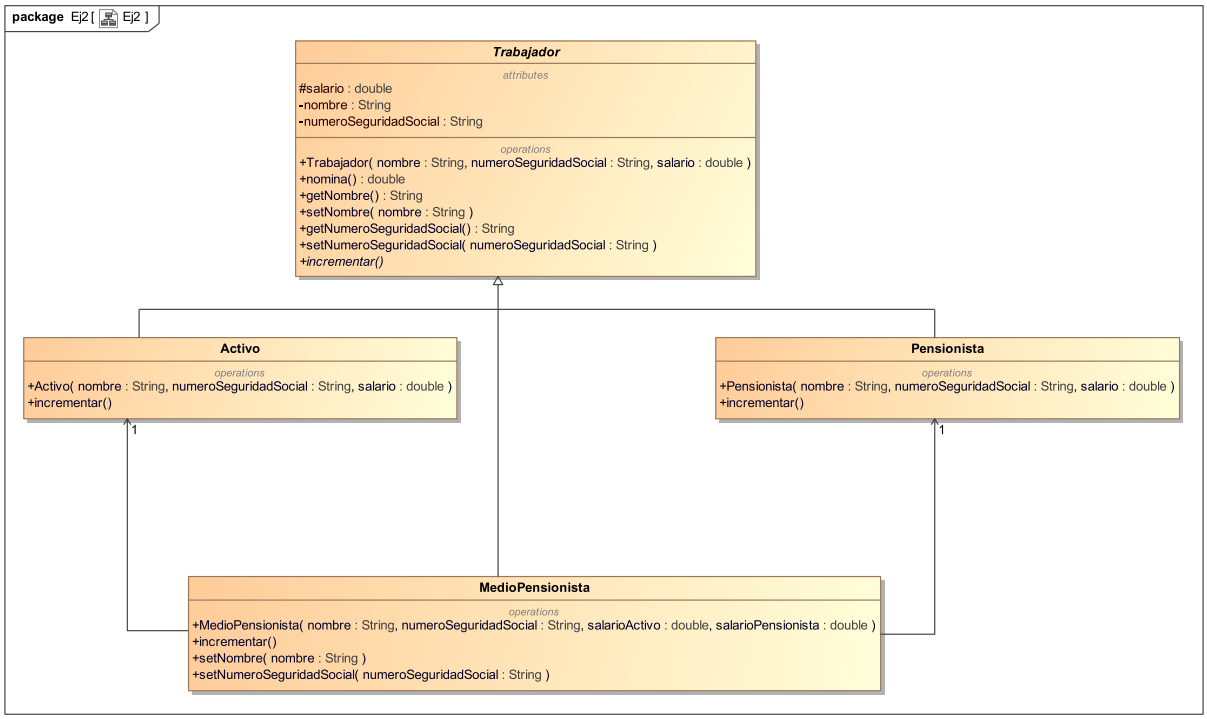

Figure 2: Modelo de diseño para Java en Magic Draw

El modelo estructural no podía ser implementado en Java debido a que la clase MedioPensionista heredaba tanto de la clase Activo como de Pensionista. En respuesta al primer apartado del ejercicio, en Java no se permite herencia múltiple, es decir, una clase no puede heredar simultáneamente de dos clases distintas, por lo que el modelo suponía un problema para la implementación.

La solución que proponemos consiste en hacer que MedioPensionista herede directamente de la clase abstracta Trabajador. Una de las restricciones del ejercicio establece que deben reutilizarse los métodos de la clase Trabajador, Activo y Pensionista, por lo que hemos relacionado la clase MedioPensionista con Activo y Pensionista. La idea es mantener como atributos privados una instancia de cada una de estas clases por cada instancia de MedioPensionista, y de esta manera utilizar sus métodos para calcular el salario de la misma. Se detallará el funcionamiento más adelante, en el apartado correspondiente.

La clase Trabajador ha sufrido alguno cambios. En primer lugar, hemos cambiado la visibilidad de los atributos `nombre` y `numeroSeguridadSocial` de pública a privada y, en su defecto, hemos añadido *getters* y *setters*, ya que cambiar el nombre de una instancia de la clase

MedioPensionista implica cambiar el nombre de las instancias de Activo y Pensionista que la componen, para así mantener la consistencia de dichos atributos. Como estos métodos pertenecen a la clase Trabajador, las clases que hereden de ella los mantienen, pudiendo sobreescribirla, como ocurre con la clase MedioPensionista. Por otro lado, se ha añadido un constructor con los tres parámetros necesarios para inicializar correctamente los valores de `nombre`, `numeroSeguridadSocial` y `salario` de un trabajador.

Las clases Activo y Pensionista no han sufrido apenas cambios, se ha añadido el constructor correspondiente para inicializar los valores de sus atributos, al igual que en la clase de la que heredan.

La clase MedioPensionista sí ha sufrido un mayor número de cambios. En primer lugar, el constructor recibe, no tres, sino cuatro argumentos, habiéndose dividido el parámetro `salario` en `salarioActivo` y `salarioPensionista`. Esto se debe a que las instancias de esta clase reciben un salario como trabajador activo y otro como trabajador pensionista, por lo que a cada uno de ellos se le debe aplicar un porcentaje de incremento de distinto. Además, se deben crear y almacenar en atributos de manera interna una instancia de la clase Activo y otra de la clase Pensionista, que se inicializarán con los mismos valores comunes y el tipo de salario correspondiente. Los *setters*, como ya se ha explicado antes, deben mantener la consistencia de los atributos, por lo que además de modificar su valor en la instancia de MedioPensionista, deben modificarlo en las instancias privadas de Activo y Pensionista. Por último, el método abstracto incrementar realiza a su vez llamadas al método incrementar se las instancias de Activo y Pensionista de los atributos de la clase, para posteriormente modificar el salario de MedioPensionista como la suma de ambos salarios.

Clase Abstracta Trabajador

Interfaz pública

- `Trabajador()`/*{Constructor}*.
- `nomina()`: Devuelve el salario del trabajador.
- `getNombre()`/*{Getter}*: Devuelve el nombre del trabajador.
- `setNombre()`/*{Setter}*: Actualiza el nombre del trabajador por el que es pasado por parámetro.
- `getNumeroSeguridadSocial()`/*{Getter}*: Devuelve el número de la Seguridad Social del trabajador.
- `setNumeroSeguridadSocial()`/*{Setter}*: Actualiza el número de la Seguridad Social del trabajador por el que es pasado por parámetro.

- `incrementar()`/*{Abstract}*: Actualiza el salario del trabajador dependiendo del tipo que sea.

Interfaz protegida

- `salario`/*{Attributes}*: Salario del trabajador. No puede ser negativo ni cero.

Interfaz privada

- `nombre`/*{Attributes}*: Nombre del trabajador. No puede ser `null` ni la cadena vacía.
- `numeroSeguridadSocial`/*{Attributes}*: Número de la Seguridad Social del trabajador. No puede ser `null` ni la cadena vacía.

Clase Activo

Interfaz pública

- `Activo()`/*{Constructor}*.
- `incrementar()`: Actualiza el salario del trabajador activo con un incremento del 2%.

Clase Pensionista

Interfaz pública

- `Pensionista()`/*{Constructor}*.
- `incrementar()`: Actualiza el salario del pensionista con un incremento del 4%.

Clase MedioPensionista

Interfaz pública

- `MedioPensionista()`/*{Constructor}*.
- `incrementar()`: Actualiza el salario del medio pensionista con un incremento del 2% de su salario como activo más un incremento del 4% de su salario como pensionista.
- `setNombre()`/*{Setter}*: Actualiza el nombre del trabajador, así como del trabajador `activo` y `pensionista` asociados.

- `setNumeroSeguridadSocial()`/*{Setter}*: Actualiza el número de la Seguridad Social del trabajador, así como del trabajador `activo` y `pensionista` asociados.

Interfaz privada

- `activo`/*{Attributes}*: Utilizado para calcular el salario como activo del trabajador.
- `pensionista`/*{Attributes}*: Utilizado para calcular el salario como pensionista del trabajador.

Ejercicio 3: Cadena de montaje.

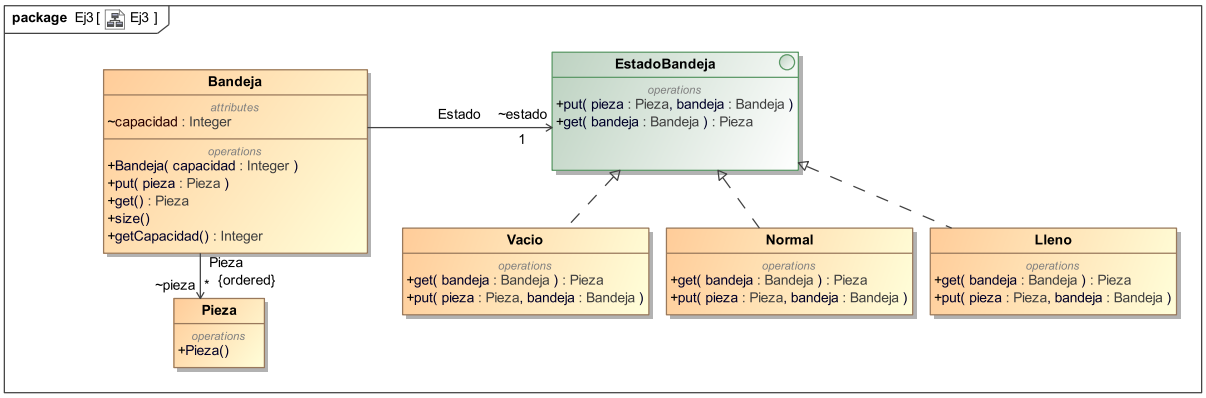

Figure 3: Modelo de diseño para Java en Magic Draw

Hemos creado el modelo de diseño basándonos en el modelo de dominio, aplicando el patrón de diseño Estado: para ello hemos creado una interfaz `EstadoBandeja`, la cual es implementada por tres clases (estados) distintas: `Vacio`, `Normal` y `Lleno`.

Se suelen comparar mucho los patrones Estrategia y Estado, y esto hizo que nos plantearamos usar el patrón Estrategia. Su comparación viene dada porque funcionan de manera muy similar: pretenden modificar el funcionamiento de un objeto en tiempo de ejecución de forma transparente a través de un proceso de composición. La diferencia que los separa es el objetivo del patrón Estrategia: proporcionar alternativas para realizar una misma tarea. Es decir, el patrón Estrategia es útil cuando queremos, por ejemplo, serializar un objeto. Por tanto, Estrategia encapsula el algoritmo, mientras que Estado encapsula un comportamiento que variará dependiendo del estado en el que se encuentre la ejecución, por estas razones fue que nos decantamos por el patrón Estado.

La clase `Bandeja` no presenta grandes cambios con respecto al modelo de dominio, tan solo se ha limitado la visibilidad del atributo `capacidad` al paquete, manteniendo los métodos `put()` , `get()` y `size()` públicos. Para contener las piezas hemos optado por usar una cola: una colección que ordena los elementos según el algoritmo FIFO (First In, First Out). El estado en el que se encuentran las bandejas se almacena como un atributo dentro de la clase Bandeja

La clase Pieza no presenta ningún cambio, simplemente tiene un constructor que se encarga de instanciar la clase.

La clase Vacio implementa la interfaz EstadoBandeja. Esta clase representa el estado de la clase Bandeja cuando se encuentra vacía. Tiene dos métodos: `get()` el cual lanza una excepción del tipo `IllegalStateException`, ya que no se pueden obtener piezas de una bandeja vacía y el método `put()` el cual se encarga de añadir piezas a la bandeja y de transicionar del estado vacío al normal o al lleno, según la capacidad de la bandeja.

La clase Normal implementa la interfaz EstadoBandeja. Esta clase representa el estado de la clase Bandeja cuando no se encuentra ni vacía ni llena. Tiene dos métodos: `get()` que se encarga de devolver el primer elemento dentro de la lista de piezas y eliminarlo de la bandeja además de transicionar a estado vacío si se quedara vacía la bandeja. El método `put()`, que se encarga de añadir piezas a la bandeja y de transicionar al estado lleno si la bandeja llegara a su límite de capacidad.

La clase Lleno implementa la interfaz EstadoBandeja. Esta clase representa el estado de la clase Bandeja cuando se encuentra en su tope de capacidad. Tiene dos métodos: `get()` el cual se encarga de devolver el primer elemento dentro de la lista de piezas y eliminarlo de la bandeja además de transicionar a estado Normal. El método `put()` simplemente lanza una excepción del tipo `IllegalStateException`, ya que no se pueden añadir piezas a una bandeja que se encuentra llena.

Clase Bandeja

Interfaz pública

- `Bandeja(int capacidad)`/*{Constructor}*
- `put(Pieza pieza)`: Añade a la bandeja la pieza pasada por parámetro si es posible
- `get()`: Devuelve la primera pieza en la lista de piezas de la bandeja si es posible
- `size()`: Devuelve el número de elementos almacenados en la bandeja.

Interfaz visible desde el paquete

- `capacidad`/*{Attributes}*: Capacidad máxima de la bandeja
- `estado`/*{Attributes}*: Estado actual en el que la bandeja se encuentra
- `pieza`/*{Attributes}*: Cola FIFO que almacena las piezas de la bandeja

Clase Pieza

Interfaz pública

- `Pieza()`/*{Constructor}*

Interfaz EstadoBandeja

- `get(Bandeja bandeja)`
- `put(Pieza pieza, Bandeja bandeja)`

Clase Vacio

Interfaz EstadoBandeja

- `get(Bandeja bandeja)`/*{Getter}*: Lanza una excepción del tipo `IllegalStateException` al no existir más elementos que extraer
- `put(Pieza pieza, Bandeja bandeja)`/*{Setter}*: Añade a la bandeja pasada por parámetro la pieza recibida.

Clase Normal

Interfaz EstadoBandeja

- `get(Bandeja bandeja)`/*{Getter}*: Dada la bandeja por parámetro, devuelve la primera pieza dentro de la lista de piezas.
- `put(Pieza pieza, Bandeja bandeja)`/*{Setter}*: Añade a la bandeja pasada por parámetro la pieza recibida.

Clase Lleno (Implementa interfaz EstadoBandeja)

Interfaz EstadoBandeja

- `get(Bandeja bandeja)`/*{Getter}*: Dada la bandeja por parámetro, devuelve la primera pieza dentro de la lista de piezas.
- `put(Pieza pieza, Bandeja bandeja)`/*{Setter}*: Lanza una excepción del tipo `IllegalStateException` al estar la bandeja llena

Código Java

Ejercicio 1

`Acceso.java`

```
package ej1;

import java.util.Date;

public class Acceso {
  // Attributes
  private Expediente expediente;
  private Profesional profesional;

  private Date fecha;
  private TipoAcceso tipo;

  // Constructor
  private Acceso(Profesional profesional, Expediente expediente, TipoAcceso tipoAcceso) {
    this.fecha = new Date(System.currentTimeMillis());
    this.profesional = profesional;
    this.expediente = expediente;
    this.tipo = tipoAcceso;
  }
  // Methods
  public static Acceso crearAcceso(
      Profesional profesional, Expediente expediente, TipoAcceso tipoAcceso) {
```

```
    assert profesional != null;
    assert expediente != null;
    assert tipoAcceso != null;

    Acceso acceso;
    if (profesional.getAcceso(expediente) == null) {
      acceso = new Acceso(profesional, expediente, tipoAcceso);
      profesional.addAcceso(acceso);
      expediente.addAcceso(acceso);
    } else {
      acceso = profesional.getAcceso(expediente);
      acceso.fecha = new Date(System.currentTimeMillis());
      acceso.tipo = tipoAcceso;
    }

    assert profesional.getAcceso(expediente) != null;
    assert expediente.getAcceso(profesional) != null;
    return acceso;
  }

  public Expediente getExpediente() {
    return expediente;
  }

  public Profesional getProfesional() {
    return profesional;
  }

  public Date getFecha() {
    return fecha;
  }

  public TipoAcceso getTipo() {
    return tipo;
  }
}
```

Expediente.java

```
package ej1;

import java.util.Collections;
import java.util.Enumeration;
import java.util.HashSet;
import java.util.Set;

public class Expediente {
  // Attributes
  private Paciente paciente;
  private Set<Acceso> acceso;

  // Constructor
  Expediente(Paciente paciente) {
    assert paciente != null;

    acceso = new HashSet<>();
    this.paciente = paciente;
  }

  // Methods
  Acceso getAcceso(Profesional profesional) {
    for (Acceso a : acceso) {
      if (a.getProfesional().equals(profesional)) {
        return a;
      }
    }
    return null;
  }

  void addAcceso(Acceso acceso) {
    this.acceso.add(acceso);
  }

  public Enumeration<Acceso> getAccesos() {
    return Collections.enumeration(acceso);
```

```
  }

  public Paciente getPaciente() {
    return paciente;
  }
}
```

Paciente.java

```
package ej1;

import java.util.Collections;
import java.util.Enumeration;
import java.util.HashSet;
import java.util.Set;

public class Paciente {
  // Attributes
  private Set<Expediente> expedientes;
  private Expediente expedienteAbierto;

  // Constructor
  Paciente() {
    expedientes = new HashSet<>();
  }

  // Methods
  public static Paciente crearPaciente() {
    Paciente paciente = new Paciente();
    Expediente expediente = paciente.crearExpediente();
    return paciente;
  }

  public Expediente crearExpediente() {
    Expediente expediente = new Expediente(this);
    expedientes.add(expediente);
    return expediente;
```

```
  }

  public Enumeration<Expediente> getExpedientes() {
    return Collections.enumeration(expedientes);
  }

  public Expediente getExpedienteAbierto() {
    return expedienteAbierto;
  }

  public void setExpedienteAbierto(Expediente expediente) {
    assert expediente == null || expediente.getPaciente().equals(this);
    this.expedienteAbierto = expediente;
  }
}
```

Profesional.java

```
package ej1;

import java.util.Collections;
import java.util.Enumeration;
import java.util.HashSet;
import java.util.Set;

public class Profesional {
  // Attributes
  private Set<Acceso> acceso;

  // Constructor
  public Profesional() {
    acceso = new HashSet<>();
  }

  // Methods
  Acceso getAcceso(Expediente expediente) {
    for (Acceso a : acceso) {
```

```
      if (a.getExpediente().equals(expediente)) {
        return a;
      }
    }
    return null;
  }

  void addAcceso(Acceso acceso) {
    assert acceso != null;
    assert !this.acceso.contains(acceso);

    int sizePre = this.acceso.size();

    this.acceso.add(acceso);

    assert this.acceso.size() == sizePre + 1;
    assert this.acceso.contains(acceso);
  }

  public Enumeration<Acceso> getAccesos() {
    return Collections.enumeration(acceso);
  }
}
```

TipoAcceso.java

```
package ej1;

public enum TipoAcceso {
  CONSULTA,
  MODIFICACION,
  CREACION,
  ARCHIVO
}
```

Ejercicio 2

Activo.java

```
package ej2;

public class Activo extends Trabajador {
  // Constructor
  public Activo(String nombre, String numeroSeguridadSocial, double salario) {
    super(nombre, numeroSeguridadSocial, salario);
  }

  // Methods
  @Override
  public void incrementar() {
    this.salario *= 1.02;
  }
}
```

MedioPensionista.java

```
package ej2;

public class MedioPensionista extends Trabajador {
  // Attributes
  private Activo activo;
  private Pensionista pensionista;

  // Constructor
  public MedioPensionista(
      String nombre,
      String numeroSeguridadSocial,
      double salarioActivo,
      double salarioPensionista) {
    super(nombre, numeroSeguridadSocial, salarioActivo + salarioPensionista);
    this.activo = new Activo(nombre, numeroSeguridadSocial, salarioActivo);
    this.pensionista = new Pensionista(nombre, numeroSeguridadSocial, salarioPensionista);
  }
```

```
  // Methods
  public void incrementar() {
    this.activo.incrementar();
    this.pensionista.incrementar();
    this.salario = this.activo.salario + this.pensionista.salario;
  }

  @Override
  public void setNombre(String nombre) {
    super.setNombre(nombre);
    activo.setNombre(nombre);
    pensionista.setNombre(nombre);
  }

  @Override
  public void setNumeroSeguridadSocial(String numeroSeguridadSocial) {
    super.setNumeroSeguridadSocial(numeroSeguridadSocial);
    activo.setNumeroSeguridadSocial(numeroSeguridadSocial);
    pensionista.setNumeroSeguridadSocial(numeroSeguridadSocial);
  }
}
```

Pensionista.java

```
package ej2;

public class Pensionista extends Trabajador {

  // Constructor
  public Pensionista(String nombre, String numeroSeguridadSocial, double salario) {
    super(nombre, numeroSeguridadSocial, salario);
  }

  // Methods
  public void incrementar() {
    this.salario *= 1.04;
```

```
  }
}
```

Trabajador.java

```
package ej2;

public abstract class Trabajador {
  // Attributes
  protected double salario;
  private String nombre;
  private String numeroSeguridadSocial;

  // Constructor
  public Trabajador(String nombre, String numeroSeguridadSocial, double salario) {
    assert nombre != null;
    assert !nombre.equals("");
    assert numeroSeguridadSocial != null;
    assert !numeroSeguridadSocial.equals("");
    assert salario >= 0;

    this.nombre = nombre;
    this.numeroSeguridadSocial = numeroSeguridadSocial;
    this.salario = salario;
  }

  // Methods
  public double nomina() {
    return salario;
  }

  public String getNombre() {
    return nombre;
  }

  public void setNombre(String nombre) {
    assert nombre != null;
```

```
    assert !nombre.equals("");
    this.nombre = nombre;
  }

  public String getNumeroSeguridadSocial() {
    return numeroSeguridadSocial;
  }

  public void setNumeroSeguridadSocial(String numeroSeguridadSocial) {
    assert numeroSeguridadSocial != null;
    assert !numeroSeguridadSocial.equals("");
    this.numeroSeguridadSocial = numeroSeguridadSocial;
  }

  public abstract void incrementar();
}
```

Ejercicio 3

Bandeja.java

```
package ej3;

import java.util.LinkedList;
import java.util.Queue;

public class Bandeja {
  // Attributes
  Queue<Pieza> pieza;
  int capacidad;
  EstadoBandeja estado;

  // Constructor
  public Bandeja(int capacidad) {
    assert capacidad > 0;
    // Inicialmente la bandeja esta vacia.
    estado = new Vacio();
    pieza = new LinkedList<>();
```

```
    this.capacidad = capacidad;
  }

  // Methods
  public void put(Pieza pieza) {
    estado.put(pieza, this);
  }

  public Pieza get() {
    return estado.get(this);
  }

  public int size() {
    return pieza.size();
  }

  public int getCapacidad() {
    return capacidad;
  }
}
```

EstadoBandeja.java

```
package ej3;

interface EstadoBandeja {
  void put(Pieza pieza, Bandeja bandeja);

  Pieza get(Bandeja bandeja);
}
```

Lleno.java

```
package ej3;

class Lleno implements EstadoBandeja {
  @Override
```

```
  public void put(Pieza pieza, Bandeja bandeja) {
    throw new IllegalStateException("Bandeja llena");
  }

  @Override
  public Pieza get(Bandeja bandeja) {
    assert bandeja != null;
    assert bandeja.pieza.size() == bandeja.capacidad;

    Pieza pieza = bandeja.pieza.remove();
    bandeja.estado = bandeja.pieza.isEmpty() ? new Vacio() : new Normal();

    assert !bandeja.pieza.contains(pieza);
    assert bandeja.pieza.size() == bandeja.capacidad - 1;
    return pieza;
  }
}
```

Normal.java

```
package ej3;

class Normal implements EstadoBandeja {
  @Override
  public void put(Pieza pieza, Bandeja bandeja) {
    assert bandeja != null;
    assert pieza != null;
    assert bandeja.pieza.size() < bandeja.capacidad;
    assert bandeja.pieza.size() > 0;
    assert !bandeja.pieza.contains(pieza);

    var sizePre = bandeja.pieza.size();
    bandeja.pieza.add(pieza);
    if (bandeja.pieza.size() == bandeja.capacidad) bandeja.estado = new Lleno();

    assert bandeja.pieza.contains(pieza);
    assert sizePre == bandeja.pieza.size() - 1;
```

```
  }

  @Override
  public Pieza get(Bandeja bandeja) {
    assert bandeja != null;
    assert bandeja.pieza.size() < bandeja.capacidad;
    assert bandeja.pieza.size() > 0;

    var sizePre = bandeja.pieza.size();
    Pieza pieza = bandeja.pieza.remove();
    if (bandeja.pieza.isEmpty()) bandeja.estado = new Vacio();

    assert !bandeja.pieza.contains(pieza);
    assert sizePre == bandeja.pieza.size() + 1;

    return pieza;
  }
}
```

Pieza.java

```
package ej3;

public class Pieza {
  // Constructor
  public Pieza() {}
}
```

Vacio.java

```
package ej3;

class Vacio implements EstadoBandeja {
  @Override
  public void put(Pieza pieza, Bandeja bandeja) {
    assert pieza != null;
    assert bandeja != null;
```

```
    assert bandeja.pieza.size() == 0;

    bandeja.pieza.add(pieza);
    if (bandeja.pieza.size() == bandeja.capacidad) bandeja.estado = new Lleno();
    else bandeja.estado = new Normal();

    assert bandeja.pieza.contains(pieza);
    assert bandeja.pieza.size() == 1;
  }

  @Override
  public Pieza get(Bandeja bandeja) {
    throw new IllegalStateException("Bandeja vacia.");
  }
}
```

Tests jUnit

Test Ejercicio 1

```
import ej1.*;
import org.junit.Test;

public class Ej1 {

  @Test
  public void expedienteAbierto() {
    Paciente pac1 = Paciente.crearPaciente();
    Expediente x = pac1.getExpedienteAbierto();
    assert x == null;
  }

  @Test
  public void cerrarExpedienteAbierto() {
    var paciente = Paciente.crearPaciente();
    var expediente = paciente.getExpedientes().nextElement();
    paciente.setExpedienteAbierto(expediente);
```

```
    assert paciente.getExpedienteAbierto().equals(expediente);

    paciente.setExpedienteAbierto(null);

    assert paciente.getExpedienteAbierto() == null;
  }

  @Test(expected = AssertionError.class)
  public void expedienteAbiertoDistintoPaciente() {
    var paciente1 = Paciente.crearPaciente();
    var expediente = paciente1.getExpedientes().nextElement();
    var paciente2 = Paciente.crearPaciente();
    paciente2.setExpedienteAbierto(expediente);
  }

  @Test
  public void testEjecucion() {
    Profesional p1 = new Profesional();
    Profesional p2 = new Profesional();
    Paciente pac1 = Paciente.crearPaciente();
    var expediente = pac1.getExpedientes().nextElement();
    Acceso.crearAcceso(p1, expediente, TipoAcceso.CREACION);
    Acceso.crearAcceso(p2, expediente, TipoAcceso.MODIFICACION);
    Acceso.crearAcceso(p1, expediente, TipoAcceso.MODIFICACION);
  }
}
```

Test Ejercicio 2

```
import ej2.Activo;
import ej2.MedioPensionista;
import ej2.Pensionista;
import org.junit.Test;

public class Ej2 {
  @Test
  public void testActivo() {
```

```
    Activo a1 = new Activo("av", "N1", 1500.0);
    // El incremento del salario de un activo debe ser igual al salario  activo
    // sin incremento multiplicado por 1.02
    a1.incrementar();
    assert 1500.0 * 1.02 == a1.nomina();
  }

  @Test
  public void testPensionista() {
    Pensionista p1 = new Pensionista("dg", "N2", 800.0);
    // El incremento del salario de un pensionista debe ser igual al salario pensionista
    // sin incremento multiplicado por 1.04
    p1.incrementar();
    assert 800.0 * 1.04 == p1.nomina();
  }

  @Test
  public void testMedioPensionista() {
    MedioPensionista m1 = new MedioPensionista("ob", "N3", 1100.0, 450.0);
    // El incremento del salario de un pensionista debe ser igual al salario pensionista
    // sin incremento multiplicado por 1.04
    m1.incrementar();
    assert 1100.0 * 1.02 + 450.0 * 1.04 == m1.nomina();
  }
}
```

Test Ejercicio 3

```
import ej3.Bandeja;
import ej3.Pieza;

public class Ej3 {
  @org.junit.Test(expected = IllegalStateException.class)
  public void bandejaLlena() {
    var b1 = new Bandeja(2);
    b1.put(new Pieza());
    b1.put(new Pieza());
```

```
    b1.put(new Pieza());
  }

  @org.junit.Test(expected = AssertionError.class)
  public void bandejaInvalida() {
    var b1 = new Bandeja(-1);
  }

  @org.junit.Test(expected = AssertionError.class)
  public void piezaEnBandeja() {
    Bandeja b1 = new Bandeja(3);
    Bandeja b2 = new Bandeja(1);

    Pieza p1 = new Pieza();
    Pieza p2 = new Pieza();

    b1.put(p1);
    b1.put(p1);
    b1.put(p2);
    b1.get();
    b1.get();
    b1.get();

    b2.put(p1);
    b2.get();
  }

  @org.junit.Test(expected = IllegalStateException.class)
  public void bandejaVacia() {
    var b1 = new Bandeja(1);
    b1.get();
  }

  @org.junit.Test()
  public void normal() {
    Bandeja b1 = new Bandeja(2);
    Bandeja b2 = new Bandeja(2);
```

```
    Pieza p1 = new Pieza();
    Pieza p2 = new Pieza();

    b1.put(p1);
    b1.put(p2);
    b2.put(b1.get());
    b2.put(b1.get());
    assert b2.size() == 2;
    assert b1.size() == 0;
  }
}
```