Enunciado de Práctica
Diseño y Pruebas Unitarias

Temática

El dominio de esta práctica es el mismo que habéis estado trabajando en la parte de análisis, y que conocéis en detalle: la *Plataforma accesible de Gestión Ciudadana con reconocimiento biométrico*.

Se pide implementar el código y desarrollar test unitarios de una versión simplificada del caso de uso *Ejercer derecho al voto*. La primera parte de la práctica se centra en el escenario más simple: la *identificación manual del votante*. La segunda parte se centra en el escenario de la *identificación mediante verificación biométrica*.

Se recomienda escribir el código en orden creciente de complejidad, tal y como se propone en este documento, e ir probándolo progresivamente a lo largo del desarrollo.

Comenzaremos formalizando algunas clases consideradas básicas (igual que lo son String, BigDecimal, etc.), dado que su única responsabilidad es la de guardar ciertos valores. Todas ellas se definirán en un paquete denominado data.

El paquete data

El paquete data contendrá algunas clases, la única responsabilidad de las cuales es la de guardar un valor de tipo primitivo o clase de java (concretamente String). Pensad los diversos motivos que hacen que sea conveniente hacerlo así.

Se trata de las clases Nif (el nif del votante), VotingOption (opción de voto para unos determinados comicios) y Password (el password del personal de soporte para la identificación manual).

A continuación se presenta la clase VotingOption.

La clase VotingOption

Representa una opción de voto válida para unos determinados comicios.

Esta es su implementación:

```
package data;
```

```java
/**
* Essential data classes
*/

final public class VotingOption {

// The tax identification number in the Spanish state.

    private final String party;

    public VotingOption (String option) { this. party = option; }

    public String getParty () { return party; }

    @Override
    public boolean equals (Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        VotingOption vO = (VotingOption) o;
        return party.equals(vO.party);
    }

    @Override
    public int hashCode () { return party.hashCode(); }

    @Override
    public String toString () {
        return "Vote option {" + "party='" + party + '\'' + '}';
    }
}
```

Cabe decir que el conjunto de partidos políticos válidos para unos determinados comicios será establecido de antemano, al iniciar la jornada electoral (método específico interface Scrutiny).

La decisión acerca de los valores para los votos **nulo** y **en blanco** se deja para vosotros.

Estas clases serán **inmutables** (por eso el **final** y la no existencia de setters), y tienen definido un **equals**, que comprueba si dos instancias con el mismo valor son iguales. Estas clases se denominan también **clases valor**, ya que de sus instancias nos interesa tan sólo el valor.

Definid vosotros las otras clases Nif y Password.

Es conveniente añadir las excepciones que consideréis oportunas. Por ejemplo, para el caso de la clase Nif, podemos definir las dos situaciones siguientes: que al constructor le llegue null (objeto sin instanciar), y también un nif mal formado. ¿Qué excepciones convendrá contemplar para Password?

Implementar y realizar test para estas clases (es suficiente con comprobar las excepciones consideradas).

Parte I

El caso de uso *Ejercer derecho al voto (7 puntos)*

En lo que queda de documento se presenta el caso de uso a desarrollar. Se trata de **Ejercer derecho al voto**[1].

Este caso de uso resuelve el sistema de voto electrónico, que habilita el ejercicio del derecho al voto a la ciudadanía a través de medios y tecnologías digitales en todas las fases que lo componen. Por supuesto, la realización de este trámite requiere la identificación del ciudadano con el fin de garantizar los principios electorales. En esta primera parte se pide resolver la identificación manual.
Todo ello se desarrollará en concordancia con los diagramas utilizados como referencia durante el análisis y, en particular, con los DSS que se incluyen como anexo al final de este documento.
Los casos de uso implicados en esta primera parte de la práctica son los siguientes: **Ejercer derecho al voto**, **Emitir voto-e** y **Autenticarse**.

Definiremos la clase VotingKiosk[2] para implementar el caso de uso. Será la responsable de manejar los eventos de entrada (*controlador de fachada*).

Además, se implementarán los servicios involucrados. Estos se presentan a continuación.

Servicios involucrados

Agruparemos los servicios involucrados en este caso de uso en un paquete denominado services.

Nos referimos al servicio externo ElectoralOrganism (Organismo Electoral). Adicionalmente, serán considerados como servicios del sistema Scrutiny, que será quien gestione el escrutinio a nivel del colegio electoral y el LocalService, para verificar las cuentas de acceso al sistema.

A continuación, se presentan con más detalle, así como su interacción con la plataforma de gestión ciudadana.

ElectoralOrganism. Este servicio interviene en dos ocasiones a lo largo del proceso de ejercer el voto, y se vale de los siguientes métodos:

---

[1] Como estamos realizando pruebas unitarias, no se presenta ninguna interfaz de usuario, sino que tendremos métodos que son los que usará la interfaz de usuario (los eventos de entrada). Tal y como se presenta en el tema de *Patrones GRASP*, implementaremos y probaremos un *controlador de fachada* para el caso de uso (es decir, un objeto del dominio escogido específicamente como controlador).

[2] Corresponde a la clase del dominio VotingKiosk.

- canVote(Nif nif): a partir del nif verifica si el votante cumple todas las condiciones para votar, antes de proceder a emitir el voto.
  Excepciones: Además de ConnectException[3], NotEnabledException, para indicar que el votante ya ha votado o no está en un colegio electoral que le corresponde.
- disableVoter(Nif nif): a partir del nif registra en el censo electoral que dicho votante ya ha ejercido el derecho al voto (el votante queda inhabilitado, evitando así la posibilidad de duplicar votos).

La definición del servicio ElectoralOrganism queda tal y como se muestra a continuación:

```
package services;

/**
 * External services involved in managing the electoral roll
*/

public interface ElectoralOrganism { // External service for the Electoral Organism

    void canVote(Nif nif) throws NotEnabledException, ConnectException;
    void disableVoter(Nif nif) throws ConnectException;
}
```

LocalService. Se encarga de verificar las cuentas de acceso al sistema por parte del personal de soporte. Para ello se utiliza el siguiente método:

- verifyAccount(String login, Password pssw): verifica la cuenta de acceso (login, password) del personal de soporte.

Por cuestiones de simplicidad, se le asigna esta única responsabilidad a LocalService. En este sentido, las conexiones con el exterior serán gestionadas directamente por la clase VotingKiosk. Y, como ya se ha mencionado, la gestión del escrutinio se delega en Scrutiny.

La definición de LocalService queda tal y como se muestra a continuación:

```
public interface LocalService {  // Solves the login step for the support staff
    void verifyAccount (String login, Password pssw)
                throws InvalidAccountException;
}
```

Como situación excepcional a tratar tenemos la siguiente:

- InvalidAccountException: indica que la cuenta de acceso proporcionada por el personal de soporte es inválida.

[3] Excepción proporcionada por la API. Señala un error producido al intentar conectar un socket a una dirección y puerto remotos. Por lo general, es debido a que la conexión fue rechazada remotamente (e.g. ningún proceso fue escuchado en la dirección/puerto remoto).

Scrutiny. Es una de les componentes más destacadas, puesto que su responsabilidad es registrar y escrutar los votos a medida que éstos son emitidos. Recordad que estamos implementando un sistema de e-voting de *registro electrónico directo*, es decir, que los votos son escrutados conforme son emitidos. Por simplicidad, consideraremos que Scrutiny es colaboradora directa de VotingKiosk.

- initVoteCount (List<VotingOption> validParties): recibe la lista de partidos políticos válidos para estos comicios e inicializa el recuento de votos.
- scrutinize(VotingOption vopt): incorpora la opción de voto (vopt) en el recuento de votos emitidos.

Para acceder a los resultados tenemos los métodos siguientes:

- getVotesFor(VotingOption party): retorna el número de votos que ha obtenido el partido party.
- getNulls(): retorna el número de votos nulos.
- getBlanks(): retorna el número de votos en blanco.
- getTotal(): retorna el número total de votos contabilizados.
- getScrutinyResults(): vuelca por pantalla (dispositivo de salida) los resultados del escrutinio, partido a partido, así como también el número de votos en blanco y votos nulos.

Aunque algunos de estos métodos no formen parte del caso de uso, no obstante, **resultarán útiles para llevar a cabo las pruebas de test**. Estos métodos deberán ser invocados por la clase principal, VotingKiosk, que es la que colabora con Scrutiny.

La definición de la interface Scrutiny queda de este modo:

```
public interface Scrutiny {  // Centralizes the vote counting

    void initVoteCount (List<VotingOption> validParties);
    void scrutinize (VotingOption vopt);
   int getVotesFor (VotingOption vopt);
   int getTotal ();
   int getNulls ();
   int getBlanks ();
   void getScrutinyResults ();
}
```

Pensad cuál es la estructura (atributos de la clase) más adecuada para la clase doble que implemente esta interface. Del mismo modo, procurad que el código para sus métodos (públicos y también privados) sea lo más entendedor posible, evitando la aparición de *code smells*. De todas las clases dobles necesarias, ésta será la más compleja.

Estos servicios se inyectarán a la clase pertinente, por ejemplo, mediante un *setter*.

El paquete evoting

Este paquete contiene las clases directamente involucradas con el ejercicio del voto, en colaboración directa con los servicios. Trabajaremos con una única clase, la clase VotingKiosk.

La clase VotingKiosk

A continuación, se presenta la clase VotingKiosk. Esta clase define, entre otros, los eventos de entrada para resolver el caso de uso que nos ocupa.

La estructura, aunque incompleta, de la clase votingKiosk es la siguiente:

```
package evoting;
/**
* Internal classes involved in in the exercise of the vote
*/
public class votingKiosk {

    ???   // The class members
    ???   // The constructor/s
    // Input events
  public void initVoting () { . . . }
  public void setDocument (char opt) { . . . }
  public void enterAccount (String login, Password pssw)
                throws InvalidAccountException { . . . }
  public void confirmIdentif (char conf) throws InvalidDNIDocumException
                { . . . }
  public void enterNif (Nif nif) throws NotEnabledException, ConnectException
                { . . . }
  public void initOptionsNavigation () { . . . }
  public void consultVotingOption (VotingOption vopt) { . . . }
  public void vote () { . . . }
  public void confirmVotingOption (char conf) throws ConnectException { . . .}
    // Internal operation, not required
  private void finalizeSession () { . . . }

    (. . .) // Setter methods for injecting dependences and additional methods
}
```

Las clases de test no deberán acceder directamente a los métodos de Scrutiny para obtener el resultado del escrutinio. Es por ello que VotingKiosk debe ofrecer métodos adicionales a los presentados aquí para facilitar la consulta de dichos resultados.

A continuación, se presentan a grandes rasgos dichos métodos. Los contratos de referencia de algunos de los eventos de entrada se pueden consultar en el documento *ModelCasosUs-PartContractes.pdf*[4].

Eventos de entrada:

- initVoting (): evento que emula la acción de seleccionar la funcionalidad de e-voting dentro de la plataforma.
- setDocument (char opt): evento que representa la acción de seleccionar o bien el DNI o bien el pasaporte, como documento identificativo. Se presentarán ambas opciones, de tal forma que la opción escogida por el votante se guarda como carácter.
- enterAccount (String login, Password pssw): evento que emula el paso de autenticación del personal de soporte, y que proporciona ambos valores (nombre de usuario y password), correspondientes a la cuenta asociada a esa persona.
  Excepciones: InvalidAccountException (la cuenta de acceso proporcionada es inválida).
- confirmIdentif (char conf): evento que representa la respuesta proporcionada por el personal de soporte tras comprobar si el DNI es válido y corresponde al votante. Mediante el carácter conf se le indica al sistema una de ambas opciones.
  Excepciones: InvalidDNIDocumException (el documento está caducado, no es válido o no corresponde a la persona).
- enterNif (Nif nif): evento que representa la acción de introducir manualmente el NIF del votante por parte del personal de soporte.
- initOptionsNavigation (): evento que representa la acción de indicarle al sistema que despliegue los menús y opciones de voto correspondientes a los comicios en curso.
- consultVotingOption (VotingOption vopt): evento que representa la acción de acceder a la información relacionada con la opción de voto correspondiente al argumento vopt (para votar por este partido es necesario acceder a esta información).
- vote (): evento que emula el paso de escoger la opción de voto que se presenta en estos momentos en pantalla (previamente accedida), para la emisión del voto.
- confirmVotingOption (char conf): evento que representa la acción de confirmar o no la opción de voto escogida en el paso anterior. Mediante el carácter conf se le indica al sistema una de ambas opciones.

Operaciones internas:

- finalizeSession ():finaliza la sesión de voto para volver a la pantalla inicial.
  No se pide su implementación.

<u>Consideraciones</u>:

- **Deben tratarse también las situaciones descritas en las precondiciones** (documento *ModelCasosUs-PartContractes.pdf*), para detectar si se han completado con éxito los pasos que preceden a cada evento del caso de uso. Manejaréis una única excepción: ProceduralException, para representar todas las situaciones relacionadas con este aspecto.
  A incluir en las cabeceras de los métodos.

[4] Este documento tan sólo incluye los contratos de los eventos de entrada que tienen efecto en los objetos del dominio. El resto de eventos o bien únicamente emulan la acción de un click, introducen información para continuar con el proceso, o bien se limitan a generar datos temporales.

- Por lo que respecta a las excepciones correspondientes a las clases del paquete data **quedan para vosotros** (deberán ser incorporadas también en las cabeceras de los métodos).
- Definiréis las excepciones como clases propias (subclases de Exception –excepciones *checables*) y, adicionalmente, la excepción proporcionada por la API: ConnectException, tal y como aparece en las cabeceras de los métodos.

Implementar y realizar test para el caso de uso descrito aquí, utilizando dobles para los servicios colaboradores.

Parte II

El Caso de Uso Verificar identidad de manera automática (*3 puntos*)

Se trata de implementar las clases inmutables, excepciones, operaciones y eventos de entrada específicos para el caso de uso **Verificar identidad de manera automática**, a fin de resolver la verificación biométrica mediante el uso del pasaporte.

El escaneo facial consiste en captar imágenes de una serie de puntos clave de la cara (los puntos biométricos faciales; e. g. lóbulo del oído, esquinas de ambos ojos, etc.). Tras procesar cada uno de esos puntos y tomar medidas entre ellos, finalmente se obtiene como resultado una puntuación numérica. Es la **clave biométrica facial**, la cual es única para cada individuo.

Para la huella dactilar el proceso es el mismo, ahora aplicado sobre el dedo, y obteniendo en este caso la **clave biométrica de la huella dactilar**.

Al tratarse de valores numéricos bastante grandes, representaremos cada una de las claves biométricas con un byte[]. Para ello definiréis dos nuevas clases valor, a incluir también en el paquete data:

- SingleBiometricData para representar una clave biométrica.
- BiometricData para agrupar las dos claves biométricas (facial y huella dactilar) correspondientes a un individuo o pasaporte.

Definiremos dos interfaces que representan los periféricos de lectura y escaneo de los datos biométricos (HumanBiometricScanner y PassportBiometricReader), a incorporar en un subpaquete de evoting llamado biometricdataperipheral. Son las siguientes:

HumanBiometricScanner. Representa el periférico para el escaneo facial y de la huella dactilar del individuo. Consta de dos métodos:

- scanFaceBiometrics(): retorna los datos biométricos de la cara.
- scanFingerprintBiometrics(): retorna los datos biométricos de la huella dactilar.

Excepciones: Unificaremos en una única excepción cualquier situación de error en el proceso de escaneo del individuo. La excepción: HumanBiometricScanningException.

La definición de la interface HumanBiometricScanner queda tal y como se muestra a continuación:

```
package evoting.biometricdataperipheral;

/**
 * Peripherals for reading and scanning biometric data
*/

public interface HumanBiometricScanner {// Peripheral for scanning human biometrics

    SingleBiometricData scanFaceBiometrics ()
                        throws HumanBiometricScanningException;
    SingleBiometricData scanFingerprintBiometrics ()
                        throws HumanBiometricScanningException;

}
```

PassportBiometricReader. Representa el periférico para la validación del pasaporte y lectura de los datos incrustados en él. Consta de los siguientes métodos:

- validatePassport(): valida que el documento identificativo esté en vigor, así como que los datos incrustados no estén corruptos.
  Excepciones: NotValidPassportException (se detecta algún problema al respecto).
- getNifWithOCR(): retorna un Nif que representa el nif del votante.
- getPassportBiometricData(): retorna una instancia de BiometricData con las dos claves biométricas (cara y huella dactilar) incrustadas en el pasaporte.
  Excepciones: La excepción PassportBiometricReadingException representa cualquier situación de error en el escaneo de datos biométricos del pasaporte.

Esta es su definición:

```
public interface PassportBiometricReader {// Perip. for reading passport biometrics

    void validatePassport () throws NotValidPassportException;
    Nif getNifWithOCR ();
    BiometricData getPassportBiometricData ()
                        throws PassportBiometricReadingException;

}
```

Finalmente, se presentan las incorporaciones a añadir a VotingKiosk para este escenario.

Clase VotingKiosk. Se procede a detallar aquí toda la funcionalidad asociada a la clase VotingKiosk para el escenario que nos ocupa (verificación de identidad automática).

Comenzamos con las operaciones internas y acciones relacionadas con la verificación biométrica, que también deben ser implementadas y testeadas.

- verifyBiometricData(BiometricData humanBD, BiometricData passpBD): verifica que los datos biométricos del individuo (humanBD) coinciden con los del pasaporte proporcionado (passpBD).
  Si la verificación biométrica falla lanza la excepción BiometricVerificationFailedException, invalidando por tanto el proceso de votación.
- removeBiometricData(): se encarga de eliminar cualquier rastro de los datos biométricos manejados, tanto del individuo como del pasaporte.
  Este método es invocado internamente, en el método verifyBiometricData() *(¡Al tanto, deberá invocarse antes de lanzar la excepción!*).

```
private void verifiyBiometricData
          (BiometricData humanBioD, BiometricData passpBioD)
          throws BiometricVerificationFailedException { . . . }
private void removeBiometricData () { . . . }
```

Por último, se detallan los eventos de entrada específicos para este escenario:

```
public void grantExplicitConsent (char cons) { . . . }
public void readPassport ()
          throws NotValidPassportException, PassportBiometricReadingException
          { . . . }
public void readFaceBiometrics () throws HumanBiometricScanningException
          { . . . }
public void readFingerPrintBiometrics ()
          throws NotEnabledException, HumanBiometricScanningException,
                 BiometricVerificationFailedException, ConnectException
          { . . . }
```

cuyos objetivos son los siguientes:

- grantExplicitConsent (char cons): evento que representa la acción de otorgar o denegar el consentimiento explícito para proceder a la lectura de los datos biométricos. La respuesta proporcionada por el votante se guarda como carácter.
- readPassport (): evento para indicar al sistema que puede proceder con la validación y lectura de los datos del pasaporte.
- readFaceBiometrics (): evento para indicar al sistema que puede proceder con el escaneo de la cara del votante.
- readFingerPrintBiometrics (): evento para indicar al sistema que puede proceder con el escaneo de la huella dactilar del votante.

Implementar, utilizar dobles para los sistemas colaboradores y añadir todo lo que sea necesario para realizar test sobre este escenario.

Consideraciones generales

- Deberíais resolver esta práctica **en grupos de entre dos y tres personas**.
- Esta práctica tiene un valor de un **25%** sobre la nota final y **no es recuperable**.
- Utilizaréis el sistema de **control de versiones** git y un **repositorio remoto**, a fin de coordinaros con vuestros compañeros (e. g. GitHub o GitLab –podrán ser repositorios privados). Algunas recomendaciones:
  - Cada vez que hagáis un test y el sistema lo pase, haced un commit. Nunca incorporar a la rama remota código que no ha pasado un test.
  - Cada vez que apliquéis un paso de refactoring en el código, haced un commit indicando el motivo que os ha llevado a hacerlo (¿quizás algún *code smell* o principio de diseño?), así como del refactoring aplicado.
  - Con el fin de facilitar los test, podéis definiros otros constructores además de los sugeridos aquí, simplificando la inicialización de las clases.
  - Es recomendable ir trabajando cada miembro del grupo en ramas distintas para así lograr una mejor colaboración y sincronización de vuestro trabajo (e.g. desarrollo de distintos requisitos/funcionalidades/ramas por separado).
- Como entregaréis un ZIP con el directorio del proyecto, entregaréis también el repositorio git (subdirectorio .git), por lo que podré comprobar los commits (*no os lo dejéis para el último día, y colaborad todos los miembros del equipo !*).
- Por lo que respecta al SUT (System Under Test), los eventos de entrada **deben satisfacer los contratos** de referencia facilitados (documento *ModelCasosUs-PartContractes.pdf*), los cuales cumplen con el planteamiento expuesto aquí.
- **Nivel de exhaustividad en los test**:
  - Para las clases del paquete data es suficiente con probar las excepciones.
  - **Donde las pruebas deben ser exhaustivas** es en el paquete evoting. En ambas partes, **deberán tratarse todas las situaciones posibles** de cada escenario, así como todas sus situaciones excepcionales. Para ello **es imprescindible planear una cierta estructura para el código de test**, tanto para las clases de test, como para los test dobles (e.g., podrían definirse por separado los distintos casos de test: los de éxito y los de fracaso).
  - Además, podéis recurrir a la definición de **interfaces de test**, así como la definición de **métodos default**, tal y como se ha sugerido en clase para la resolución de los problemas de la colección de problemas de testing.
- Os indico, además, **cómo testear la clase controladora del caso de uso**:
  - Como ya se ha mencionado anteriormente, los eventos de entrada del caso de uso no se testean individualmente. Cada evento es testeado precediéndolo de los eventos que van por delante en el caso de uso. De esta forma se va probando el progreso del caso de uso. En el caso de no seguir el orden establecido, se lanza la excepción ProceduralException, asociada a las **precondiciones** de dichos eventos.
- No hagáis test dobles complicados para poder aprovecharlos más de una vez. Se trata de definir **test dobles lo más simples posible**, con objeto de probar los distintos escenarios. En general, un buen enfoque consiste en definir dobles por separado para diferenciar entre diferentes tipos de situaciones (e.g. de éxito y de fracaso).
  El test doble que requiere de más código es el de la interface Scrutiny.
- Los test dobles pueden definirse internamente en las clases de test, o bien como clases separadas en paquetes separados.

Entrega

¿Qué debéis entregar?

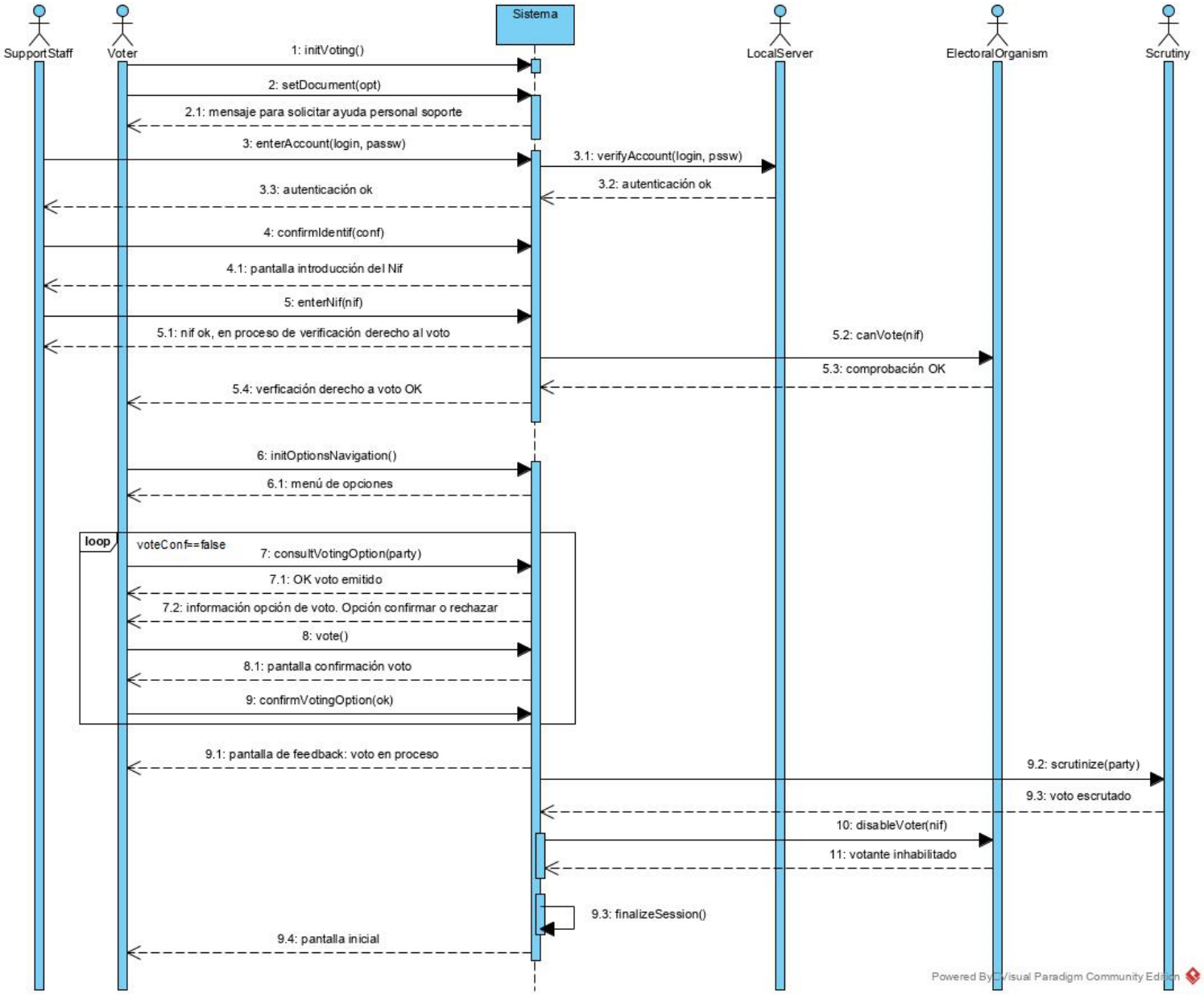
Un **ZIP** que contenga:

- El **proyecto desarrollado** (podéis utilizar IntelliJ IDEA o cualquier otro entorno).
- Un **informe** en el que expliquéis con vuestras palabras el/los criterios empleados para tomar vuestras decisiones de diseño (principios SOLID, patrones GRASP, etc.), y justificación de las mismas, si es el caso.
  Enumerad también los **métodos de refactoring** aplicados para resolver los posibles **code smell** detectados, ya sea en vuestro código o en el diseño que se propone en este enunciado.
  En cuanto a las situaciones que habéis probado en cada uno de los test realizados, no es necesario explicarlas. Tan sólo si hay algún aspecto o situación relevante, que valga la pena comentar, o bien cualquier otro detalle que pueda ayudar a valorar mejor vuestro trabajo.

Como siempre, haced la entrega a través del CV **tan sólo uno de los miembros del grupo**, indicando el nombre de vuestros compañeros.

Anexo. DSS adaptados a los dos escenarios

Sección A. DSS para la verificación manual (Parte I)



Sección B. DSS para la verificación biométrica (Parte II)