Pràctica de Proves Unitàries (curs 2016/2017) v2.0

El domini de la pràctica serà el mateix que heu estat treballant a la part d'anàlisi, és a dir, el del *vot electrònic*.

Primerament introduiré algunes classe addicionals que considerarem bàsiques, ja que la seva única responsabilitat és la de guardar valors.

El paquet data

Aquest paquet contindrà vàries classes que només serveixen per emmagatzemar una dada (`String`s, `byte[]`, etc.). Per què ho fem? Per varis motius.

Un d'ells és que en cas de no fer-ho, gran part dels nostres mètodes rebrien i/o retornarien `String`s (o altres tipus bàsics) i, per tant, no quedaria gens clar què és el que estan processant.

A més, considereu un mètode que rep el vot que ha escollit un votant. Perfectament podríem definir-lo com:

```
public void registerVote(String vote) { ... }
```

Però, realment el vot és un `String`? Mètodes com `concat`, `startsWith`, `indexOf`, etc. tenen sentit pels vots? La resposta clarament és no.

Els `String`s poden servir per a representar un valor de tipus vot, però la gran majoria d'operacions de `String` no s'apliquen a Vot. És per això que hem d'acostumar-nos a definir-les com a *classes diferents*.

En llenguatges més complexos com Scala, podriem fer simplement:

```
case class Vote(getOption: String)
case class Party(getName: String)
case class MailAddress(getAddress: String)
case class Signature(getSignature: Array[Byte])
```

Però en Java, per aconseguir més o menys el mateix, hem de crear una classe com:

```
package data;

/**
 * Represents a vote.
 */
final public class Vote {

    private final String option;

    public Vote(String option) {
        this.option = option;
    }

    public String getOption() {
        return option;
    }

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

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

        Vote vote = (Vote) o;

        return option.equals(vote.option);

    }

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

Totes aquestes classes seran **inmutables** (d'ací els finals i la no existència de setters) i tenen definit un `equals` que fa que dues instàncies que tenen el mateix valor són iguals. Per això s'anomen *classes valor*, ja que de les seves instàncies ens interessa només el valor i no la seva identitat. A més, durant el disseny, aquestes classes, que inicialment no tenien responsabilitats complexes, les poden anar agafant.

De fet, tampoc és tanta la feina a fer ja que, una vegada creada la classe i el seu atribut, l'entorn de desenvolupament és capaç de generar el constructor, el getter, l'equals, el

hashCode i un toString que funciona amb la semàntica d'una classe valor.

Les classes `Party`, `MailAddress` i `Signature` són similars i tenen getters `getName`, `getAddress` i `getSignature` per accedir al valor del nom del partit, l'adreça de correu i la signatura (que, en aquest cas és un `byte[]`)

L'única cosa que podriem haver afegit és protegir-nos davant que en el constructor ens passin null, però he considerat que no cal ja que pràcticament segur que es generaria un `NullPointerException` en algun lloc del codi que usés la classe.

Aquestes classes normalment no cal provar-les, ja que no tenen responsabilitat complexes. En tot cas, la única cosa interessant de provar seria que `equals` i `hashCode` són compatibles (dos objectes que són iguals han de tenir el mateix codi de hash). Si voleu afegir tests per a provar-les no hi ha cap problema en que ho feu.

Apartat 1. La classe ActivationCard (2 punts)

La classe que representa un targeta d'activació és gairebé una classe immutable si no fos pel detall que, una vagada realitzat el vot, el codi d'activació queda esborrat (veure crida al mètode `borrarCodgActTarj()` del diagrama de seqüència d'emetre vot).

Per això us caldrà una classe què, a més del mètodes `equals`, del seu constructor (a partir d'un `String`) i del mètode `getCode` té dos métodes:

```
public boolean isActive() { ... }
public void erase() { ... }
```

Quan construïu una instància d'ActivationCard es tracta de un codi que està actiu, si l'esborreu, deixa d'estar actiu).

Respecte del mètode `equals` farem que **només** depengui del valor del codi.

Aquesta classe pertany al paquet **`kiosk`**.

NOTA: Com es una classe a la que podrem esborar el seu codi, no té sentit que la tractem com una classe valor i definim un mètode `hashCode` per ella.

Feu tests per a provar aquesta funcionalitat.

Apartat 2. El cas d'ús Emetre Vot (6 punts)

Anem a presentar, finalment, el cas d'ús que heu de provar i d'implementar: **Emetre vot** (corresponent al seu diagrama de seqüència del sistema que teniu publicat a sakai).

Com estem fent proves unitàries, no presentarem cap interfície d'usuari sinó que tindrem mètodes que seran els que usarà la interfície d'usuari per implementar la seva funcionalitat. Encara no ho hem vist, però tal i com es presenta al tema de *Patrons GRASP*, estarem implementant i provant la classe *controladora del cas d'ús*.

Aquest cas d'ús necessita de cinc serveis externs (que s'injectaran a la classe que

implementa el cas d´ús mitjançant **setters**):

- `ValidationService`: sistema encarregat de validar un usuari a partir de la seva ActivationCard i de desactivar aquesta una vegada s'ha votat.
- `VotePrinter`: sistema que imprimeix el vot realitzat.
- `VotesDB`: sistema que enregistra el vot realitzat.
- `SignatureService`: sistema que signa criptogràficament el vot per a després poder enviar el comprovant per correu.
- `MailerService`: sistema que envia la signatura del vot a l'adreça indicada per l'usuari.

`ValidationService` té la següent interfície:

```
package services;

/**
 * Local service for validating activation cards
 */
public interface ValidationService {

    boolean validate(ActivationCard card);
    void deactivate(ActivationCard card);
}
```

`VotePrinter`:

```
package services;

/**
 * This service prints physically the vote to allow checking.
 */
public interface VotePrinter {
    void print(Vote vote);
}
```

`VotesDB`:

```
package services;

/**
 * Local service that registers votes.
 */
public interface VotesDB {
    void registerVote(Vote vote);
    List<Vote> getVotes();
}
```

`SignatureService`:

```
package services;

/**
 * External service for signing votes
 */
public interface SignatureService {
    Signature sign(Vote vote);
}
```

`MailerService`:

```
package services;

/**
 * External service for sending mails
 */
public interface MailerService {
    void send(MailAddress address, Signature signature);
}
```

Ara ja podem presentar l'estructura de la classe `VotingMachine`:

```
package kiosk;

/**
 * Implements a simplification of Use Case: Emit Vote
 */
public class VotingMachine {

    ???

    public VotingMachine() { ??? }

    public void setValidationService(ValidationService validationService) {
??? }

    ... // pels altres serveis

    public void activateEmission(ActivationCard card) throws
IllegalStateException { ??? }

    public boolean canVote() { ??? }

    public void vote(Vote vote) throws IllegalStateException { ??? }

    public void sendReceipt(MailAddress address) throwa
IllegalStateException { ??? }
}
```

Les operacions tenen les següents especificacions:

- `activateEmission(ActivationCard card)`: Usa el `ValidationService` per a veure si el votant té un codi vàlid. Si la màquina ja estava activada llença l'exepció (predefinida) `IllegalStateExeception`. Només si el codi és vàlid, ja es podrà votar a la màquina.
- `canVote()`: Indica si el votant pot votar. No es pot votar si prèviament no s'ha activat la màquina amb un codi vàlid. Després de votar no es podrà votar de nou fins a activar de nou la màquina amb un codi vàlid.
- `vote(Vote vote)`: Aquesta acció indica el vot del votant al sistema. Si s'està en un estat en el que no es pot votar, llença l'exepció `IllegalStateException`. Si es pot votar, el vot s'enregistra i s'imprimeix i es desactiva el codi d'activació. Després ja no es pot votar a la màquina fins que el proper votant l'activi.
- `sendReceipt(MailAddress address)`: Aquesta acció indica l'adreça de correu a la que enviar el codi de comprovació. Si s'està en un estat en el que no es pot votar o bé encara no s'ha fet l'acció `vote`, llença l'exepció `IllegalStateException`. Si tot es correcte, es demana al sistema de validació la signatura del vot i s'envia, usant el `MailService` el correu a l'adreça que es àssa com paràmetre. Per simplificar, **el**

sistema no comprova que el correu s'ha enviat sense problemes.

Aquest apartat consisteix en implementar i fer tests per aquesta versió del cas d'ús, utiltzant dobles per a tots els sistemes col·laboradors

Apartat 3 El control biomètric (Opcional, 2 punts)

Aquest darrer apartat opcional inclou la verificació de les dades biomètriques (escaneig de l'iris) en el moment de fer la votació.

Per tal de fer-ho:

- Haureu de crear una nova classe valor `IrisScan` per a representar l'escaneig de l'iris (que guardarà com un `byte[]`)
- Tindreu una interfície `IrisScanner` que tindrà un mètode `scan()` que retornarà un `IrisScan`
- Les `ActivationCard`s podran tenir emmagatzemat un `IrisScan`. Afegiu els mètodes adequats a `ActivationCard`
- En `activateEmission` es procedirà a comprovar l'escanig i no es podrà votar si l'escaneig de l'ull no coincidei amb l'enregistrar a la targeta.

Obviament, si la targeta no disposa de cap escaneig, la votació ha de procedir com a l'apartat 2.

Implementeu tot el que calgui per a que aquesta nova versió del cas d'ús, utilitzant dobles per a tots els sistemes col·laboradors

Consideracions generals

- La pràctica s'ha de realitzar **individualment** i si es detecten pràctiques copiades tots les còpies (incloent l'original) tindran com a nota un **0**.
- Aquesta pràctica té un valor d'un **20%** sobre la nota final i **no és recuperable**.
- Feu servir **control de versions** (git):
    - Cada vegada que feu un test i el sistema que esteu provant el passa, feu un commit.
    - Si decidiu refactoritzar el codi, feu un commit.
    - No cal que desenvolupeu coses a branques diferents, però no està prohibit.

  Com entregareu un **ZIP** amb el directori del projecte, entregareu també el repositori git (que estarà al sotsdirectori `.git`), així que podré comprovar els commits que heu fet.
- No cal que en els tests feu servir @Before. M'estimo més que dediqueu el temps en

pensar en quines coses provar i com fer-ho que en distribuïr-los en diferents classes. Ara, no està prohibit fer-ho i ho podeu fer com a refactorització, una vegada tot us funciona.

- Recordeu de posar la classe de test al mateix paquet que la classe que està provant.
- No feu dobles complicats per a poder reaprofitar-los en més d'un lloc. Feu el doble més simple que us permet passar la prova.
- Els dobles dels serveis els podeu col·locar a les classes de prova o bé en un paquet separat en la carpeta del codi de proves. Jo per exemple, he creat un paquet mocks per guardar-los (que està al directori tests/mocks).

Què heu d'entregar?

Un *ZIP* (no un rar, tgz, tar.bz, etc, etc) que contingui:

- Projecte de codi amb el projecte desenvolupat (podeu usar Netbeans o Intellij)
- Un petit informe en el que expliqueu **en llenguatge natural** la situació que voleu provar en cadascun dels tests que feu i qualsevol altre cose que m'ajudi a valorar millor el vostre treball.