Correction de l'examen JEE 2014

JEE-1 : Premier test

Ce premier test est le test nominal : on prend une carte de crédit valide, et on vérifie qu'un validateur nous renvoie bien true. Cela nous fait créer deux classes : CreditCardImpl et CCValidatorImpl.

```
@Test
public void itCanValidateAVisaCard() throws Exception {

    CreditCard card = new CreditCardImpl("1111-2222-3333-
4444",10,2014,CCType.VISA);
    CCValidator validator = new CCValidatorImpl();
    assertTrue(validator.isValid(card, 370));

}
```

Une première implémentation de isValid() peut juste retourner true : on a un premier test qui passe

JEE-2 : test d'invalidation d'une carte

Un deuxième test va nous obliger à écrire une implémentation moins naïve

```
@Test
public void itCanInvalidateAVisaCard() throws Exception {
    CreditCard card = new CreditCardImpl("1111-2222-3333-
0000",10,2014,CCType.VISA);
    CCValidator validator = new CCValidatorImpl();
    assertFalse(validator.isValid(card, 370));
}
```

En se basant sur l'exemple StringCalculator du cours, on va utiliser une expression régulière pour matcher le format de la carte et en extraire les valeurs.

Une fois que le test passe, on va refactorer la méthode de test pour avoir moins de redondance au niveau du code.

Une solutions à base de String#split() et de vérification du format était bonne aussi. Cependant la solution à base d'expression régulière est la plus efficace.

JEE-3 : Vérification de formats

On veut vérifier que certain format de nombre ne passent pas. On va donc ajouter un test

avec différents nombres de carte malformés. Pour renforcer le test (et suite à une bonne idée de Rémi Roussel), on peut tester qu'aucun CCV ne marche pour une carte malformée. Cela se fait par l'intermédiaire de la méthode suivante :

```
private void assertNoCCVCanValidateCard(CreditCard card) {
    for(int ccv = 0; ccv < card.getType().getMaxCCVNumber(); ccv++) {
        assertFalse("Card " + card.getNumber() + " was validated by " +
ccv,validator.isValid(card, ccv));
    }
}
```

JEE-4 : Vérification de la date

On va vérifier maintenant qu'une date incorrecte ne permet pas de valider une date.

Cela se fait en ajoutant une clause isValidDate à l'implémentation du validateur. Cela nous permet aussi d'extraire la validation par CCV dans une méthode distincte.

Il faut aussi mettre à jour les tests précédents pour qu'il ne plantent pas une fois que la date prise au hasard soit dépassée.

JEE-5 : Vérification d'une AMEX

```
On ajoute un test pour vérifier le cas de l'AMEX

@Test
public void ItCanValidateAnAmexCard() throws Exception {
    CreditCard card = new CreditCardImpl("1111-2222-3333-0000", 10, 5000,
            CCType.AMEX);
    assertTrue(validator.isValid(card, 2180));
    assertFalse(validator.isValid(card, 3456));
}
```

JEE-6 : Servlet

On crée une classe CreditCardServlet qui étends HttpServlet. On l'annote avec @WebServlet("/cc") et on décommente la ligne de JettyHarness : on a une servlet qui démarre et qui peut commencer à servir des requêtes.

En implémentant la méthode doGet comme ceci :

```
@Override
protected void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws ServletException, IOException {
    PrintWriter out = resp.getWriter();
    out.print("ok");
}
```

Les test passent. Il faut effectivement ajoute une clause fail au troisième test que est censé vérifier qu'il y a une erreur.

On peut maintenant récupérer les paramètres de la requête pour construire une carte et appeler le validateur. Cela se fait par l'intermédiaire de req.getParameter(). Cette méthode peut renvoyer null et c'est un cas qu'il faut traiter. On ajoute d'ailleurs un test sur la vérification de la carte (cas d'une carte nulle).

JEE-7 JPA : création d'un Account

En exécutant AccountTest, on a un première erreur :

```
    javax.naming.NameNotFoundException: Name "global/2-jpa/AccountDAOImpl"
not found.
```

Il s'agit de notre EJB qui n'est pas trouvé par le conteneur. Pour qu'il soit trouvé, il faut d'ajouter l'annotation @Singleton sur la classe. Une fois cette annotation ajoutée, il nous faut implémenter la méthode createAccount.

```
    @PersistenceContext(name="account")
    EntityManager em;

    @Override
    public Account createAccount(String ownerName, int initialBalance,
            String ccNumber) {
        Account account  = new AccountImpl(ownerName, initialBalance,
ccNumber);
        em.persist(account);
        return account;

    }
```

Il faut aussi demander au contenur d'injecter un EntityManager lié au contexte de persistence `account`. Cela se fait par le biai de l'annotation @PersistenceContext. Le nom de l'unité de persistence est celui renseigné dans le fichier `META-INF/persistence.xml` qui était déjà fourni.

Maintenant, pour que la classe AccountImpl soit persistable, il faut l'annoter avec @Entity.

Ensuite il faut définir un propriété id annoté avec @GeneratedValue et @Id qui servira de clé primaire et remplir le corp des différents getter et setter.

Enfin, il faut dans le DAO coder la méthode findAccount.

```
    @Override
    public Account findAccount(Long id) {
        return em.find(AccountImpl.class, id);
    }
```

JEE-8 : suppression d'un account

Il faut implémenter la méthode deleteAccount :

```
@Override
public void deleteAccount(Long id) {
    Account account = findAccount(id);
    if(account != null) {
        em.remove(account);
    }

}
```

JEE-9 Lister les account

il faut implémenter la méthode list() :

```
@Override
public List<Account> list() {
    return em.createQuery("SELECT a FROM AccountImpl a").getResultList();
}
```

La syntaxe JQPL d'OpenJPA est un peu plus restrictive que celle d'Hibernate. Il faut donc bien spécifier le SELECT. On pouvait aussi passer par des requêtes déclarées au niveau de l'entité via les @NamedQuery.

JEE-10 Trouver un compte en fonction du numéro de CB

C'est encore la même API que l'on utilise en passant en plus un paramètre.

```
    @Override
    public Account findAccountByCC(String number) {
        return em
                .createQuery(
                        "SELECT a FROM AccountImpl a WHERE a.ccNumber =
:number",
                        Account.class).setParameter("number", number)
                .getResultList().get(0);
    }
```

JEE-11 Ajout d'une transaction

L'ajout d'une transaction nous demande de modifer l'entité pour gérer une relation @OneToMany :

```
    @OneToMany(mappedBy="account", fetch=FetchType.EAGER)
    private List<TransactionImpl> transactions = new ArrayList<>();
```

au niveau du DAO, l'implémentation de addTransaction devra créer une transaction et mettre à jour la balance du compte correspondant.

```
    @Override
    public Transaction addTransaction(Long id, TxType type, DateTime date,
            String description, int amount) {
        Account  acc = findAccount(id);
        Transaction tx = new TransactionImpl((AccountImpl) acc, type, date,
description, amount);
        em.persist(tx );
        acc.setBalance(acc.getBalance() + (TxType.CREDIT.equals(type) ?
amount : -amount));
        em.merge(acc);
        return tx ;
    }
```

JEE-12 JAX-RS

Comme j'avais oublié d'enlever l'implémentation de l'objet AccountService, il suffisait de donner un moyen de sérialiser un Account en utilisant JAXB et ajouter l'annoation @XmlRootElement à la classe AccountImpl.