Práctica de diseño y pruebas unitarias (v3)

El objetivo de la práctica es diseñar e implementar un diseño sencillo usando pruebas unitarias para garantizar su funcionamiento. Además, como querremos controlar la precisión de los cálculos, usaremos profusamente la clase `BigDecimal`.
La funcionalidad objetivo consistirá en calcular el valor que tiene una determinada cartera de inversiones. En dicha cartera podremos encontrar diversos tipos de elementos:

- Dinero efectivo en una determinada divisa
- Conjunto de acciones de una compañía
- Contrato de futuro para comprar a un precio una cantidad de acciones de una compañía en una fecha determinada
- Contrato de futuro para vender a un precio una cantidad de acciones de una compañía en una fecha determinada

Para acceder tanto al valor bursátil actual como para saber el cambio entre dos divisas se disponen de dos clases para:

- Dadas dos divisas diferentes, obtener su ratio de cambio.
- Obtener el valor bursátil actual de una compañía

1. Clases valor testables independientemente

Las siguientes clases modelan valores **inmutables** que usaremos para representar elementos básicos como nombres de divisas, nombres de tickets, dinero (combinación de valor y divisa).
Sus operaciones serán los constructores, y quizás getters y operaciones de infraestructura como `equals`, `hashCode`, etc.
Para estas clases, solamente es estrictamente necesario realizar tests para las operaciones complejas, aunque podéis realizarlos para otros como `equals` y para comprobar si `equals` y `hashCode` cumplen su contrato[1].

Clase Currency

Esta clase representa el nombre de una divisa:

```
1 package cat.udl.eps.ep.data;
2
3 public class Currency
4   public Currency(String name) ...
5   public boolean equals(Object other) ...
6   public int hashCode() ...
7   public String toString() ...
```

Clase Ticket

Esta clase representa el nombre de un ticket (nombre de una empresa en un mercado). Por ejemplo “CBK” para Caixabank, etc.

[1] Dos objetos que son equals deben tener el mismo hashCode.

```
1 package cat.udl.eps.ep.data;
2
3 public class Ticket
4   public Ticket(String name) ...
5   public boolean equals(Object other) ...
6   public int hashCode() ...
7   public String toString() ...
```

Clase Money

Esta clase representa una cantidad monetaria en una determinada divisa. Las cantidades monetarias admiten dos operaciones:

- Suma/diferencia de dos operaciones monetarias (de la misma divisa)
- Producto de una cantidad monetaria por un entero
- Producto de una cantidad monetaria por un ratio de cambio, expresando el resultado en la divisa dada

Como ya se ha indicado, **la clase `Money` es inmutable**, por lo que las operaciones nunca modifican ni los argumentos ni los objetos receptores sino que revuelven objetos nuevos como resultados.
Los valores monetarios deberán tener **dos dígitos** de precisión y, en caso de necesitar redondeo, éste se realizará alejándose del cero (**round up**)[2].

```
1 package cat.udl.eps.ep.data;
2
3 public class Money
4   public Money(BigDecimal quantity, Currency currency) ...
5   public Currency getCurrency() ...
6   public Money add(Money other) ...
7   public Money subtract(Money other) ...
8   public Money multiply(int multiplier) ...
9   public Money change(BigDecimal ratio, Currency to) ...
10  public boolean equals(Object other) ...
11  public int hashCode() ...
12  public String toString() ...
```

- `add/subtract` lanzan `IllegalArgumentException` (excepción predefinida no comprobada) si las cantidades monetarias no se corresponden con la misma divisa.
- `change` lanza también `IllegalArgumentException` si se intenta cambiar a la misma divisa que tiene la cantidad monetaria
- las cantidades monetarias pueden ser negativas (para expresar pérdidas)

La clase `Money` es una clase que se puede comprobar de forma independiente de las demás (aunque usa la clase `Currency`, como ésta simplemente representa valores, podemos que su uso no constituye una dependencia de la que las pruebas de `Money` deban aislarse)[3].

[2] Considerad el método `setScale` de la clase `BigDecimal`.
[3] Desde el punto de vista de los tests, se comportaría igual que si usáramos la clase `String`, o del uso que hacemos de `BigDecimal`.

2. Interfaces que representan servicios externos

Estos interfaces representan servicios externos y serán los que querremos implementar con dobles en los tests.

```
1 package cat.udl.eps.ep.services;
2
3 public interface MoneyExchange
4   BigDecimal exchangeRatio(Currency from, Currency to)
5     throws RatioDoesNotExistException;
```

No se debe llamar al `MoneyExchange` para hacer cambios de una divisa a sí misma, es decir, **`from` y `to` deben ser divisas diferentes**.

```
1 package cat.udl.eps.ep.services;
2
3 public interface StockExchange
4   Money value(Ticket ticket)
5     throws TicketDoesNotExistException;
```

`RatioDoesNotExistException` y `TicketDoesNotExistException` son clases que representan **excepciones comprobadas** y tienen un único constructor que recibe un `String` con un mensaje asociado. Indican que no hay cambio entre las divisas dadas y que no hay cotización del ticket dado.

Residen en el mismo paquete que las interfaces descritas.

3. Clases con dependencias a sustituir por dobles

Las siguientes clases representan el núcleo del diseño y serán la que deberán testearse aisladamente de los servicios que utilizan para su funcionamiento. Aunque estas clases también utilizan `Ticket`, `Currency` y `Money`, dado que éstas se consideran que representan valores básicos, no se han de aislar de ellas a la hora de hacer testing, por lo que dichas clases no han de sustituirse por dobles.

Interfaz Investment

Para tratar los diferentes tipos de inversiones disponemos de una interfaz a la que se deberán de adaptar todos ellos.

```
1 package cat.udl.eps.ep.portfolio;
2
3 public interface Investment
1   Money evaluate(Currency currencyTo,
2                  MoneyExchange moneyEx,
3                  StockExchange stockEx)
4     throws EvaluationException;
```

Vamos a ver, por fin, los diferentes productos de inversión que tenemos en la cartera[4]. En ellos ya no indicaremos el método `evaluate`, pues será el mismo para todos ellos.
`EvaluationException` es una excepción controlada, definida en el mismo paquete, y que tiene un único constructor que recibe como parámetro un `String` indicando la razón de su lanzamiento. Sirve para indicar que ha habido algún tipo de error al evaluar la inversión.

Clase Cash

Representa una inversión en una cantidad de una moneda concreta:

```
1 package cat.udl.eps.ep.portfolio;
2
3 public class Cash implements Investment
4   public Cash(Money money) ...
```

El valor de un producto en moneda es el que se obtiene actualmente en el mercado (es decir, según el cambio indicado por el `MoneyExchanger` en caso de ser necesario).

Clase Stock

Representa una inversión en un número de acciones de una determinada compañía:

```
1 package cat.udl.eps.ep.portfolio;
2
3 public class Stock implements Investment
4   public Stock(Ticket ticket, int numShares) ...
```

El valor de un Stock es el que actualmente tiene en el mercado (según el `StockExchange`) en la divisa que se pide (con el cambio de divisa del `MoneyExchange`).

Clase FutureBuy

Representa una compra que se realizará en una fecha futura de una determinada cantidad de acciones de una compañía a un precio pactado:

```
1 package cat.udl.eps.ep.portfolio;
2
3 public class FutureBuy implements Investment
4   public FutureBuy(Ticket ticket,
5                    int numShares,
6                    Money pricePerShare) ...
```

[4] El enunciado está vagamente inspirado en un ejemplo que sale en el libro "DSLs in Action". Mi conocimiento de cómo funcionan las carteras de valores y de cómo se evalúa su valor es nulo. Así que las reglas de evaluación que sigue el enunciado son las que me resultan cómodas para que probéis técnicas de testing.

El valor será la diferencia entre lo que costaría comprarlos a su precio actual en el mercado y lo que costaría adquirirlos al precio pactado. Es por ello que su valor actual puede ser negativo (si el precio de compra pactado es superior al actual).

Por ejemplo, un `FutureBuy` de 10 acciones a un precio de 12,45€ cada una de ellas, tendrá una valoración de -20,80€ si el valor actual de las mismas es 10,37€; en cambio la valoración sería de 34,40€ si el valor actual fuera 15,89€.

NOTA: Para simplificar, no consideraremos la fecha en la que el futuro se tendrá que ejecutar.

Clase FutureSell

Es la contrapartida del anterior, ya que representa a la otra parte de la transacción, es decir: la venta que se realizará en una fecha futura de una determinada cantidad de acciones de una compañía a un precio pactado.

```
1 package cat.udl.eps.ep.portfolio;
2
3 public class FutureSell implements Investment
4   public FutureSell(Ticket ticket,
5                     int numShares,
6                     Money pricePerShare) ...
```

Su valor será la diferencia entre lo que obtendríamos por la venta al precio pactado y lo que obtendríamos al venderlo a su precio actual en el mercado. Es por ello que, en este caso, su valor actual puede ser negativo (si el precio de venta pactado es inferior al actual).

Clase Portfolio

Esta clase representará una cartera de inversiones (el método `addInvestment` añadirá una inversión a la cartera).
La valoración de la cartera no es más que la suma de las valoraciones de las inversiones que contiene. En caso que la evaluación de algún activo de la cartera produzca un error, la valoración de la cartera, fallará también.

```
1 package cat.udl.eps.ep.portfolio;
2
3 public class Portfolio implements Investment
4   public Portfolio() ...
5   public addInvestment(Investment investment) ...
```

4. Orden de implementación y testing

Seguid el orden de implementación que se indica a continuación e id haciendo **commits** (con los mensajes adecuados) cada vez que completáis un "apartado". Podéis hacer commits más pequeños, por ejemplo en cada clase, cuando vais implementando y probando cada una de las operaciones que contiene. Un objetivo de la práctica es que probéis este ritmo de trabajo y, si queréis, podéis experimentar haciendo TDD, es decir, primero el test, luego el código que lo pasa, commit; y así hasta completar la funcionalidad.

1. Clases `Currency` y `Ticket` (pueden no tener tests ya que su código lo puede hasta generar la IDE) **[1 punto]**
2. Clase `Money` con sus tests. Debéis probar las operaciones, con sus casos de error, casos normales teniendo en cuenta la precisión de los cálculos. Pensad que el resto de clases usan exhaustivamente Money para realizar operaciones. **[3 puntos]**
3. Clase `Cash` y sus **tests**. Como mucho invoca a uno de los servicios externos (el MoneyExchange) **[1 punto]**
4. Clase `Stock` y sus **tests**. Aquí ya se invocarán, en algunos casos, ambos servicios externos. **[1 punto]**
5. Clase `FutureBuy` y sus **tests**. **[1,5 puntos]**
6. Clase `FutureSell` y sus **tests**. Factorizad el código común con la clase anterior para tener el mínimo de duplicidades. **[1,5 punto]**
7. Clase `Portfolio` y sus **tests**. **[1 punto]**

En cuanto a los tests, **no consideraré a la hora de evaluar si hay mucho código duplicado entre los tests de una clase**. Por tanto no es necesario que uséis `@Before`, etc., etc; prefiero en que os centréis en pensar y programar los tests que en organizarlos de la mejor manera posible.

5. ¿Qué se ha de entregar?

- La práctica se realizará de **forma individual** y, si se detectan copias, la nota de todas las copias será cero.
- Proyecto **Netbeans/Intellij** con las implementaciones de las clases necesarios y de los tests de prueba
    - todas las clases públicas, que no sean de prueba, han de ir acompañadas de su correspondiente javadoc en su parte pública.
    - usad nombres adecuados para las variables e indentad bien el código ("*readability counts*")
- **Usad control de versiones con git**
    - aseguraos de que el repositorio se incluye dentro del directorio que entregáis.
    - cuando hagáis el zip del proyecto el repositorio se incluirá en él.
    - no olvidéis de hacer un **clean** antes de comprimir el directorio del proyecto para no incluir las clases compiladas.
- Informe (1 ó 2 páginas sin contar portada, índice, contraportada y cualquier otro método que uséis para aumentar el peso del informe) que describa los puntos más importantes de vuestra implementación, las dificultades que habéis encontrado al plantear los tests, el ritmo de trabajo que habéis seguido, si habéis probado hacer TDD, dificulatdes encontradas, etc, etc.