Simulacro segundo control de laboratorio

INSTRUCIONES

- Crea un nuevo workspace en el escritorio
- **Cambia el nombre del proyecto inicial a: apellido1_apellido2_nombre**

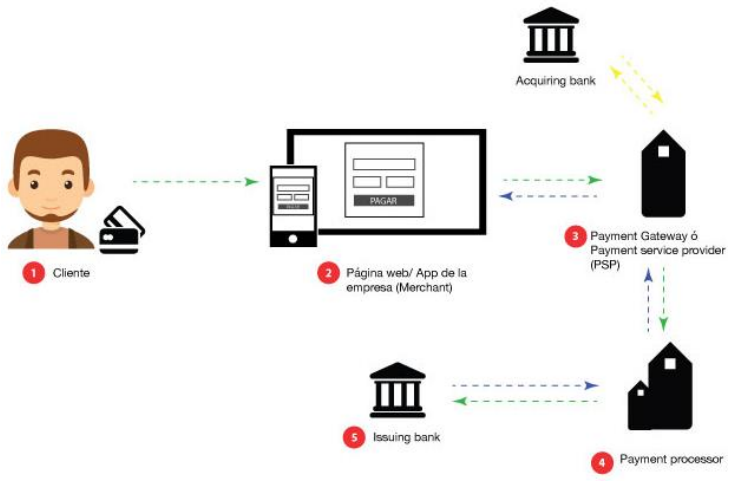
ESTÁ TOTALMENTE PROHIBIDO

- USAR CÓDIGO DE OTROS PROYECTOS (excepto `mp.2022.util`).
- Copiar/enviar información a través de la red

Observaciones importantes

1. **DEBES** dibujar el diagrama de clases UML **y debes hacerlo antes de la implementación**.
2. No solo se tendrá en cuenta la propia implementación, las pruebas unitarias y el manejo de excepciones se consideran partes esenciales de la nota.

1. Planteamiento del problema



PAYPAY, es un famoso proveedor de servicios de pago o PSP. Es decir, es un tercero que ayuda a las empresas a aceptar pagos de los clientes. En pocas palabras, los PSP permiten a las empresas aceptar pagos de crédito / débito, así como transferencias bancarias, cheques, pagos móviles,…. conectándolos con entidades financieras (bancos).

PAYPAY necesita una aplicación simple para validar las transacciones de pago realizadas por los clientes. Para el propósito de nuestra aplicación, consideramos una transacción como una operación comercial que se paga con un medio de pago.

En primer lugar, esta aplicación **cargará información sobre las transacciones de un fichero**. Luego, **la aplicación las clasificará en válidas / no válidas**. Finalmente, se deben **crear dos ficheros: uno con las transacciones válidas y otro con las inválidas**. Todas las *transacciones válidas* se guardan en un fichero comprimido y las *inválidas* se almacenan en un fichero de texto sin comprimir.

En aras de la simplicidad, solo consideraremos las tarjetas de crédito y las cuentas corrientes como medios de pago. En consecuencia, tenemos dos tipos de transacciones: *transacciones con tarjeta de crédito* y *transacciones con cuenta corriente.*

Cada **Transacción** tiene esta información:

- ***fecha*** de la operación (como un string, con el formato yyyy/mm/dd, ejemplo 2021/05/03).
- ***número*** del medio de pago (número de tarjeta de crédito o número de cuenta) (un string).
- **importe** de la operación (un double).
- **descripción** de la operación (un string).

Una *transacción con tarjeta de crédito* **(CreditCardTransaction)** tiene esta información extra:

- ***fecha de vencimiento*** de la tarjeta (no se puede usar para pagos una vez pasada esta fecha) (un string aaaa / mm / dd).

- **cantidad máxima** permitida por esta tarjeta para una operación (un valor double)

Una *transacción con cuenta corriente* (**CurrentAccountTransaction)** tiene, como información particular, el ***tipo*** **del cliente**: *Normal* (N) o *Premium* (P).

2. Formato de ficheros

Los datos de entrada vienen en un fichero de texto plano con extensión ".trx". Tiene dos tipos de líneas.

- Para transacciones con tarjeta de crédito:

  cc;*<date>;<card number>;<exp date>;<max amount per trx>;<amount>;<description>*

- Para transacciones de cuenta corriente:

  acc;*<date>;<account number>;<client type>;<amount>;<description>*

  El separador entre campos es el carácter de punto y coma (";"). Lo siguiente es una muestra de este tipo de fichero.

```
cc;2021/05/03;5552468190471987;2021/12/03;500;400;Carrefour Travel
cc;2021/04/05;6011898652586869;2021/12/05;1000;12;Valentin's cafeteria
acc;2021/04/08;ES0310250124910021214445;N;150;Car repairing at Caris
acc;2021/04/08;ES0712452342121241551584;N;1000;Shopping at CFRD
```

  Una vez que el proceso de validación ha finalizado, se deben crear dos ficheros nuevos

  - **El *fichero de transacciones válido*** es un fichero comprimido con la extensión **.trx.gz** y el mismo formato que el fichero de entrada.
  - **El *fichero de transacciones no válidas*** es un fichero de texto plano, con el mismo formato que el de entrada, pero con la extensión **.invalid.trx**

  Se debe tener en cuenta que las transacciones en el fichero de entrada pueden no estar ordenadas, pero la aplicación tiene que producir ficheros con las transacciones ordenadas por *fecha y número* de forma ascendente.

3. Procesamiento de transacciones

El propósito del proceso de validación es separar las transacciones válidas de las inválidas. Para eso, la clase ***TransactionValidator*** mantiene dos listas diferentes: las válidas (irán al fichero "valid.trx.gz") y las inválidas (irán al fichero "file.invalid.trx").

Una transacción se considera inválida si no pasa **todas las reglas de validación** de la aplicación para el tipo de transacción. Ten en cuenta que una transacción puede tener más de un error y el programa debe indicar todos los errores de cada transacción

Para cada fallo detectado, se debe generar un mensaje explicativo y adjuntarlo a la transacción. Por ejemplo:

"(cc) La cantidad 850.0 es mayor que el máximo: 500.0 para la tarjeta de crédito", o
"(acc) El número IBAN[1] es inválido: ES6524155245212844135465".

3.1 Operaciones de validación

Para las ***transacciones de cuenta corriente***, se aplican estas reglas de validación:

- Si el tipo de cliente es *Normal*, la cantidad debe ser <= 1.000 €. Si es *Premium*, no hay límite para la cantidad cargada contra la cuenta.

[1] IBAN: International Bank Account Number

- El número de cuenta debe ser un IBAN válido. Eso evita errores tipográficos en caso de introducir de forma manual el número.

Para las ***transacciones con tarjeta de crédito***, se deben cumplir estas otras reglas de validación:

- La fecha de la operación no puede ser posterior a la fecha de vencimiento de la tarjeta.
- El importe cobrado no puede ser mayor que el máximo por transacción para la tarjeta.
- El número de tarjeta debe ser válido para Lhun[2]

Notas:

- Para verificar si un número de cuenta es un IBAN válido, usa la clase IBAN provista con el esqueleto inicial (package *uo.mp.transaction.model.util*) de esta manera:
  ```
  if ( IBAN.isValid( number ) ) …
  ```
- La fecha de la operación no puede ser posterior a la fecha de vencimiento de la tarjeta.
- Del mismo modo, para verificar si un número de tarjeta de crédito es válido, usa la clase Lhun proporcionada en (package *uo.mp.transaction.model.util*):
  ```
  if ( Lhun.isValid( number ) ) …
  ```
- Para comparar dos fechas usa String.compareTo (...). Esto funciona por la forma en que se representan las fechas (aaaa / mm / dd).
  ```
  2019/11/04 > 2019/11/03
  ```

Para cada una de las validaciones anteriores que no se cumplen, se debe adjuntar un mensaje de error a la transacción.

4. Manejo de errores

No se deben confundir los siguientes errores con el incumplimiento de las reglas de validación de las transacciones.

En el manejo de errores se deben tener en cuenta los siguientes.

4.1 Validación de parámetros

Es necesario validar los parámetros de los métodos que se implementen. Se considerará un error de programación que no cumple con las condiciones del contrato y se lanza `IllegalArgumentException`, con un mensaje apropiado, para indicar que a un método se le ha pasado un parámetro no válido.

- Todos los parámetros deben ser no nulos, incluidas las listas.
- Las cadenas tampoco deben estar en blanco, y los números deben ser mayores que cero.

[2] https://en.wikipedia.org/wiki/Luhn_algorithm

4.2 Errores del Parser

Durante el análisis del fichero de entrada, se debe tener en cuenta la posibilidad de errores en el fichero de entrada. Una línea no válida genera una excepción `InvalidLineException`. Estos errores hacen que se ignore la línea, pero continúe analizando las siguientes y registre este mensaje exacto:

`PARSING ERROR en línea` *`line-number`*`:` *`descripción-del-error`*

Las líneas no son válidas cuando:

- Está vacía. La descripción del error será LÍNEA VACÍA
- El primer campo contiene una cadena diferente de cc o acc. El error será: PALABRA CLAVE NO VÁLIDA
- Número incorrecto de campos. El error será: NÚMERO DE CAMPOS INCORRECTO
- El límite de operación y la cantidad no son números. El error será: FORMATO DE NÚMERO NO VÁLIDO
- Fechas incorrectas: no separadas por /, cualquiera de dd, mm o aaaa no son números, dd no está entre 1 y 31, mm no está entre 1 y 12 y aaaa es antes 2021. La descripción del error será: FORMATO DE FECHA NO VÁLIDO

4.3 Errores de Usuario/Aplicación

Hacen que la aplicación pare mostrando el mensaje:

APPLICATION ERROR revise este error: descripción-del-error

- El fichero de entrada no existe. La descripción del error será: `el fichero de entrada no existe`
- Si al añadir transacciones a **TransactionValidator**, sucede que se realizaron dos transacciones con el mismo sistema de pago en la misma fecha (mismo número y fecha de operación), el fichero de entrada se considera corrupto. La descripción del error será: `operación repetida`

4.4 Errores de Programación y Sistema

Hacen que el programa se detenga. El siguiente mensaje debe imprimirse en la salida estándar

```
PROGRAMMING ERROR la aplicación finaliza debido a un error interno
```

Además de mostrar el error por consola, se debe registrar en el fichero log la traza de la pila.

5. Desarrollo de la aplicación

- Dibuja un diagrama UML para mostrar el diseño de la solución. Esto debe entregarse al final del examen, pero la primera versión debe elaborarse al principio.
    - Te dan un esqueleto con algunas clases que dan algunas pistas sobre el diseño. Por favor, asegúrate de comprender el diseño previsto y verifica tu UML
- Aplica el polimorfismo
- Manejo de excepciones
    - Crea una clase de excepción específica para indicar errores de aplicación o lógica (descrito en 4.3).
    - Usa/maneja las excepciones correctamente para tratar posibles problemas con el manejo de ficheros y otros tipos de errores.

- Completa la clase Main con las cláusulas catch de captura adecuadas. La clase Main es la única que puede imprimir información en la salida estándar.

- Entrada/Salida
- Añade Junit tests para probar los siguientes métodos:
  - *TransationValidator::validate()*.
  - Transaction.*validate() en todas las clases de la jerarquía*
  - TransactionLoader::load (). Se proporciona un fichero con errores en el esqueleto.

Evaluación

Un programa que funcione NO significa un aprobado. Hay cinco aspectos principales que deben revisarse y **debe tener un mínimo** en cada uno:

- Diseño expresado como un diagrama de clase UML.
- Polimorfismo
- Manejo de excepciones
- Streams
- Calidad de las pruebas unitarias.

Si utilizas tus propias colecciones y clases de ordenación, tienes puntuación extra en la calificación final.