Informática - Redes e Multimédia

Caixa Multibanco

Helder Correia 20102556 **Miguel Ferreira** 20102552 **Rodolfo Mendonça** 20102581

Introdução à Programação

Índice

1 Descrição geral do projecto

O intuito deste projecto é o desenvolvimento de um programa que simula uma caixa multibanco, com a possibilidade de efectuar levantamentos, consulta de saldo e movimentos, pagamento de serviços, e depósitos.

Apesar de ser um programa simples, este grupo sentiu-se motivado em usar este projecto para explorar áreas de interesse na linguagem Java, principalmente o uso de testes automatizados, implementação de uma interface gráfica em Swing e padrões de desenho populares como o MVC (*Model View Controller*).

As funcionalidades foram desenvolvidas primeiro através da criação de testes automatizados com separação de responsabilidades em mente, e só depois, as suas implementações. Como os testes estiveram disponíveis desde o início, houve a segurança de que bugs não eram introduzidos nas implementações ou refactorizações, e *feedback* rápido e automático de que essas implementações funcionavam correctamente.

Apenas no fim foram construídas duas interfaces de utilizador, uma em linha de comandos e outra gráfica, com um controlador que decide qual interface a executar.

Em termos de organização, foram criados três pacotes, para alojar o Main — que serve como um controlador simples (*Application Controller*) —, as classes do modelo — com lógica de negócio e persistência de dados —, e a apresentação.

2 Análise do problema

2.1 Caracterização das principais entidades do programa

Classe	Descrição
Atm	Representa uma caixa multibanco (do inglês, *Automatic Teller Machine*). Responsável por autenticar o utilizador com base no seu pin, e efectuar as operações de serviços que pertençam a um multibanco, tais como o levantamento, depósito e pagamento de serviços. Decidimos implementar o extra de inicializar o objecto com um valor de dinheiro em cofre, disponível para os levantamentos.
Account	Representa uma conta bancária. Tem como responsabilidade guardar o número de conta, dados do cliente (nome), saldo, movimentos de conta, e processar operações de débito e de crédito.
AccountMapper	Implementa o padrão *Data Mapper*, que serve como intermediário entre um objecto do tipo Account e a sua infraestrutura de armazenamento de dados.
Transaction	Representa um movimento de conta. Só podem existir dois tipos de movimentos: crédito ou débito.
Payment	Representa um pagamento de serviços. Tem a simples função de agregar os dados relacionados com um pagamento de serviços, e validar se os dados estão correctos.

2.2 Algoritmo

definir dinheiro em cofre a 500 euros
ler argumentos
se houver argumentos, então
    se último argumento for um numero, então
        dinheiro em cofre = número lido
    se primeiro argumento for “-gui”, então
        iniciar aplicação gráfica
        sair

iniciar aplicação em linha de comandos
    definir output para UTF-8
    inicializar multibanco com dinheiro em cofre definido
    fazer
        pedir pin
        obter conta para pin dado
    enquanto conta for null
    mostrar menu principal
    pedir opção
    …

3 Implementação na linguagem de programação

Desde o início, desenhámos a aplicação com a separação de responsabilidades em mente, para promover a extensibilidade, modularidade e manutenção.

No nível mais exterior temos a camada da **apresentação**, responsável pela interface do utilizador. A sua principal função é traduzir tarefas em algo que o utilizador consiga compreender, pedindo os dados necessários e mostrando os resultados no ecrã.

A seguir temos a **lógica de negócio**, que processa pedidos, efectua decisões e avaliações lógicas, e cálculos. Ela também move informação entre a apresentação e a persistência de dados, que é a camada mais interior.

Decidimos lidar com a persistência de **dados** separadamente da lógica. É aqui que a informação é guardada e recolhida do sistema de ficheiros, mas facilmente alterado para uma base de dados, chamadas remotas, ou outro sistema de armazenamento.

No entanto, enquanto existe uma classe para apenas este propósito em relação aos movimentos de conta — a classe *AccountMapper* implementa um padrão de desenho conhecido por *Data Mapper*, e é responsável por abstrair a classe *Account* da sua persistência de dados —, o acesso aos dados do cliente permaneceu em dois métodos na classe do multibanco (*atm.model.Atm*), para simplificar, uma vez que neste caso só precisamos efectuar uma operação simples de leitura.

Não foi por esquecimento que vários métodos estão definidos como *package private*. Esses métodos servem tipicamente para ajudar os testes, ou para ser usados noutras classes do modelo, e portanto não devem ser acessíveis a partir das interfaces. Estes intuitos geralmente vêm descritos nos *docblocks*.

Também há métodos criados com o intuito de ajudar a leitura do código (e.g. *printLineBreak()* em *atm.ui.Console*) e a manutenção ao permitir alterar algum aspecto do programa apenas num sítio (e.g. *printStatusMessage()* em *atm.ui.Console*).

No que toca à implementação do objecto de movimento de conta, houve a questão de como seria melhor guardar o tipo de movimento (crédito/débito). Onde usar uma String pode levar a uma fraca consistência de valores, e difícil manutenção, considerou-se mais elegante, prático e adequado criar um tipo enumerado (*enum*), para guardar este valor. Apesar de não ser uma estrutura aprendida nas aulas, faz parte das nossas explorações da linguagem neste projecto. Deste modo os textos “Crédito” e “Débito” apenas se encontram na conversão do tipo enumerado para String, usado apenas ao mostrar os movimentos de conta ao utilizador.

No pagamento de serviços, tinha-se uma série de dados relacionados entre si, e portanto, apesar de ser apenas necessário neste programa o valor e a descrição do tipo de pagamento, fez mais sentido criar um

objecto que une esses valores numa entidade que faz sentido, em vez de enviar os dados separados como argumentos para um método. Pensamos ter uma solução limpa e elegante para este caso, criando um objecto separado para este fim (*atm.model.Payment*). Apesar de só ser retirado dele o montante, posteriormente é fácil usá-lo para outras coisas ou simplesmente guardar em algum sítio.

Adicionalmente o grupo achou interessante implementar uma interface gráfica e um cofre de dinheiro ao multibanco, disponível para levantamentos. Esse valor pode ser passado pela linha de comandos como último argumento numérico passado ao programa (por defeito, 500 euros), e irá reflectir o dinheiro em cofre durante toda a execução do mesmo.

A interface gráfica, criada com Swing pode ser executada adicionando um primeiro argumento "-gui". Será disponibilizado um ficheiro *gatm.bat*, que inicializa a interface gráfica automaticamente, no Windows, e outro *atm.bat* para correr em linha de comandos.

Como nota final, foram consultadas as convenções de código para o Java, dadas pela Oracle, e tentou-se seguir sempre que possível essas orientações, excepto em um caso ou outro. Consideramos isso uma boa prática, que promove uma leitura mais fácil entre todos.

4 Utilização do programa

```
PIN: 1234

1. Levantamentos
2. Consulta de saldo de conta
3. Consulta de movimentos de conta
4. Pagamento de serviços
5. Depósitos

> 5


|  Depósito  |

Montante: 1000
Obrigado pelo seu depósito.


PIN: 1234

1. Levantamentos
2. Consulta de saldo de conta
3. Consulta de movimentos de conta
4. Pagamento de serviços
5. Depósitos

> 1

|  Levantamento  |
1. 20       2. 50
3. 100      4. 150
5. 200      6. Outros Valores

> 2


PIN: 1234

1. Levantamentos
2. Consulta de saldo de conta
3. Consulta de movimentos de conta
4. Pagamento de serviços
5. Depósitos

> 1

|  Levantamento  |
1. 20       2. 50
3. 100      4. 150
5. 200      6. Outros Valores
> 6
```

```
Montante: 30


PIN: 1234

1. Levantamentos
2. Consulta de saldo de conta
3. Consulta de movimentos de conta
4. Pagamento de serviços
5. Depósitos

> 2

|  Saldo de conta  |

Conta número: 0010029289641272009
Saldo Actual: 920,00 euros


PIN: 1234

1. Levantamentos
2. Consulta de saldo de conta
3. Consulta de movimentos de conta
4. Pagamento de serviços
5. Depósitos

> 4

| Pagamento de Serviços |

1. Conta de Electricidade
2. Conta da Água
3. Carregamento Telemóvel

> 2

Entidade: 12345
Referência: 123456789
Montante: 15

Pagamento efectuado com sucesso

PIN: 1234

1. Levantamentos
2. Consulta de saldo de conta
3. Consulta de movimentos de conta
4. Pagamento de serviços
5. Depósitos

> 4

| Pagamento de Serviços |

1. Conta de Electricidade
2. Conta da Água
3. Carregamento Telemóvel

> 3

Telemóvel: 916489523
```

```
| Montante |

1. 5 euros
2. 10 euros
3. 20 euros

> 1
Pagamento efectuado com sucesso


PIN: 1234

1. Levantamentos
2. Consulta de saldo de conta
3. Consulta de movimentos de conta
4. Pagamento de serviços
5. Depósitos

> 3

| Movimentos de conta |

Saldo Actual: 900,00 euros

07/01/2011 00:46:55   Pagamento de serviços Telemóvel     Débito       5,00
07/01/2011 00:43:49   Pagamento de serviços Água          Débito      15,00
07/01/2011 00:41:08   Levantamento MB                     Débito      30,00
07/01/2011 00:40:59   Levantamento MB                     Débito      50,00
07/01/2011 00:37:50   Depósito MB                         Crédito   1000,00

PIN:_
```

5 Conclusões e recomendações

JUnit

Os testes foram fundamentais para o desenho e desenvolvimento incremental da aplicação. Por várias vezes verificamos que uma implementação era difícil testar, o que nos indicou que seria uma implementação mais difícil de usar também. Isto levou-nos a várias refactorizações, o que inevitavelmente levou a uma implementação mais extensível, e mais adaptável a mudanças.

Encoraja-se fortemente a aprendizagem e uso do JUnit.

Programar para o computador vs humano

Durante o projecto, foi preferencial favorecer uma melhor leitura do código, ao invés de micro-optimizar milisegundos de ciclos de relógio no processador. Na maior parte dos casos essas micro-optimizações não valem a pena e sai bem mais caro a manutenção posterior, caso tenha sido desfavorecida a leitura do código. No nosso caso foi mesmo preferível uma boa leitura por parte dos avaliadores.

No entanto, é claro que optimizações significativas devem ser consideradas com importância e ambos os factores foram ponderados em equilíbrio.

Como exemplo, no início, o ficheiro de movimentos de conta estava a ser guardado em texto, com os valores separados por uma vírgula. Posteriormente decidimos passar o ficheiro para binário, onde apenas se guarda o valor do saldo e o objecto de colecção de movimentos (*ArrayList<Transaction>*). Esta mudança simplificou em grande parte o código de leitura e escrita do ficheiro.

Uma optimização que foi discutida, mas não houve tempo de ser testada foi, não guardar o saldo nem o objecto de colecção, mas sim cada movimento (*atm.model.Transaction*) em modo *append*, uma vez que toda a informação de recuperação do saldo está nos valores dos movimentos e assim não seria necessário guardar toda a informação, em cada operação. Apenas adicionar o último movimento.

Só não houve mais investimento da parte do grupo para experimentar esta solução, talvez porque o código de recuperação do saldo (envolvendo descobrir se se trata de uma operação de crédito ou débito para somar ou subtrair) traria mais complexidade à implementação actual, e preferiu-se simplificar.

nextInt

Um problema comum dá-se quando se lê um inteiro ou decimal com o *Scanner* e não se tem o cuidado de capturar o <*Enter*> deixado no buffer. Isso foi resolvido de forma centralizada com os métodos *askXXX*, onde se resolve também o problema de o utilizador introduzir caracteres onde deveria ser um número, por exemplo.

Foi também preferível mover todas as chamadas ao *System.out* para métodos dedicados, para evitar dependência a este objecto e manter a interface consistente e um código mais fácil de ler.

Encoding

Um ponto interessante foi de que não foi pensado inicialmente que para este caso seria necessário ter que alterar o *System.out*, mas de facto, ao compilar e executar na linha de comandos do sistema operativo, verificou-se a má apresentação de caracteres acentuados. A solução passou por alterar então todas as chamadas a *System.out* para fazer o output em UTF-8, numa tentativa de resolver o problema, embora o Windows não ajudar, uma vez que a sua linha de comandos não vem com UTF-8 por defeito. Mesmo assim, e mesmo tendo a linha de comandos no Windows em UTF-8 ("`chcp 65001`"), sempre que é encontrado um caractere acentuado, é repetido os últimos *x* caracteres da linha em output, sendo *x* o número desses mesmos caracteres.

Este problema foi considerado mínimo e não foi a tempo de ser corrigido.

Longas cadeias de excepções

O Java ajuda-nos com as *checked exceptions*, mas por vezes são indesejadas. Como as operações de leitura e escrita de ficheiros lançam excepções do tipo *IOException*, e uma vez que os erros mostrados ao utilizador devem ser feitos da interface de utilizador e não no modelo, tinha-se no início toda uma cadeia de métodos com um *throws IOException*.

Isso traz dois problemas. Em primeiro lugar, "IO" sugere leitura/escrita de ficheiros, e é algo que a interface não precisa estar consciente. O que nos leva ao segundo problema. O que acontece quando queremos mudar a persistência de dados para uma base de dados? Teríamos que mudar todos os *throws*, o que é uma tarefa árdua e prova a manutenção difícil.

A solução encontrada foi re-lançar excepções desse tipo para umas do tipo *RuntimeException*, com o intuito de não serem verificadas pelo Java e não ser-mos obrigados a declarar os métodos com *throws*.

Possíveis Melhorias

Inicialmente a idea da interface gráfica era incluir imagens e ter uma série de botões representativas das teclas usadas nos multibancos, mas devido à limitação de tempo, a preocupação principal foi manter o *atm.ui.Swing* funcional e o mais simples possível, enquanto houvesse possibilidade de demonstrar o poder da separação de responsabilidades, através de duas interfaces de utilizador diferentes, que acedem à mesma lógica de negócio.

Outra melhoria seria adicionar testes automatizados às interfaces de utilizador, o que seria muito desejado, mas mais complexo de se criar.

Finalmente, o código poderia estar em inglês na sua totalidade, internacionalizado e localizado na formatação das datas, fusos horários, diferentes moedas, formatos de números, mas essencialmente permitindo a tradução de todo o texto para qualquer língua (I18n e L10n).

Bibliografia

Livros

- S. Freeman e N. Pryce, *Growing Object-Oriented Software, Guided By Tests*, Addison Wesley, 2009
- M. Fowler, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002
- F. M. Martins, *Java 6 e Programação Orientada Pelos Objectos*, FCA Editores, 2009

Material da Internet

- Oracle, *The Java Tutorials*. Disponível em: <http://download.oracle.com/javase/tutorial/>
- Oracle, *Code Conventions for the Java Programming Language*. Disponível em: <http://www.oracle.com/technetwork/java/codeconv-138413.html>

Apêndice I — atm.model

Atm.java

```
package atm.model;

import java.io.File;
import java.io.FileNotFoundException;
import java.util.Scanner;

/**
 * Objecto que representa uma caixa de multibanco. Responsável por
 * autenticar o utilizador com base no seu pin, e efectuar operações de
 * serviços que pertençam a um multibanco.
 */
public class Atm {

    /** Ficheiro de dados com as contas de clientes */
    private final String DATA_SOURCE = "clientes.txt";

    /** Quantia mínima que se pode levantar */
    private final int MINIMAL_WITHDRAWAL = 10;

    /** Quantia máxima que se pode levantar */
    private final int MAXIMAL_WITHDRAWAL = 200;

    /** Dinheiro disponível no cofre */
    private double funds;

    /**
     * Permite abstracção na localização dos ficheiros de dados
     *
     * @param filename  nome do ficheiro de dados
     * @return          objecto File
     */
    public static File getFile(String filename) {
        return new File("data" + File.separator + filename);
```

```
}

/**
 * Construtor
 *
 * @param startingFunds  quantidade disponível no cofre
 */
public Atm(double startingFunds) {
    funds = Math.abs(startingFunds);
}

/** Retorna a quantidade disponível no cofre */
double getFunds() {
    return funds;
}

/**
 * Verifica se tem dinheiro suficiente no cofre
 * para efectuar um levantamento
 */
public boolean hasEnoughWithdrawalFunds() {
    return funds >= MINIMAL_WITHDRAWAL;
}

/**
 * Deposita dinheiro numa conta de cliente
 *
 * @param d        quantidade a depositar
 * @param account  conta para onde depositar o dinheiro
 */
public void deposit(double d, Account account) {
    d = Math.abs(d);
    account.processTransaction(
        Transaction.newCredit("Depósito MB", d)
    );
}

/**
 * Faz um levantamento, desde que haja dinheiro no cofre e:
 *
 * - tem que ser no mínimo 10 e no máximo 200
 * - tem que ser uma quantia possível de se juntar com notas
 *   (múltiplo de 5)
 *
```

```
     * @param d        quantia a levantar
     * @param account  conta de onde levantar o dinheiro
     */
    public void withdraw(double d, Account account) {
        d = Math.abs(d);
        if (d > funds) {
            throw new IllegalArgumentException(
                "Montante não disponível. Levante até "+funds+
                " ou dirija-se à caixa de multibanco mais próxima."
            );
        }
        if (d < MINIMAL_WITHDRAWAL || d > MAXIMAL_WITHDRAWAL) {
            throw new IllegalArgumentException(
                "Mínimo "+MINIMAL_WITHDRAWAL+", máximo "+MAXIMAL_WITHDRAWAL+"."
            );
        }
        if (d % 5 != 0) {
            throw new IllegalArgumentException(
                "Não existem notas de "+(d%5)+" euros."
            );
        }
        account.processTransaction(
            Transaction.newDebit("Levantamento MB", d)
        );
        funds -= d;
    }

    /**
     * Pagamento de serviços Água
     *
     * @param payment  objecto de um pagamento de serviços
     * @param account  objecto de conta
     */
    public void payWaterBill(Payment payment, Account account) {
        payBill("Pagamento de serviços Água", payment, account);

    }

    /**
     * Pagamento de serviços Electricidade
     *
     * @param payment  objecto de um pagamento de serviços
     * @param account  objecto de conta
     */
```

```
public void payElectricityBill(Payment payment, Account account) {
    payBill("Pagamento de serviços Electricidade", payment, account);

}

/**
 * Pagamento de serviços Telemóvel
 *
 * @param payment  objecto de um pagamento de serviços
 * @param account  objecto de conta
 */
public void payPhoneBill(Payment payment, Account account) {
   payBill("Pagamento de serviços Telemóvel", payment, account);

}

/**
 * Pagamento de um serviço.
 *
 * @param description  descrição a registar nos movimentos de conta
 * @param payment      objecto do pagamento de serviços
 * @param account      objecto de conta
 */
private void payBill(String description, Payment payment, Account account) {
    // É possível, posteriormente, fazer algo com o objecto do pagamento
    account.processTransaction(
        Transaction.newDebit(description, payment.getAmount())
    );
}

/**
 * Retorna uma entidade para um número de telemóvel em operadora conhecida.
 *
 * @param phone  número de telemóvel
 * @return       entidade para ser usada num pagamento de serviços
 */
public String getPhoneEntity(String phone) {
    if (Payment.isValidReference(phone)) {
        switch (phone.charAt(1)){
            case '1': return "10158";  // vodafone
            case '3': return "20638";  // optimus
            case '6': return "10559";  // tmn
        }
    }
```

```
        throw new IllegalArgumentException(
            "Número inválido ou operadora desconhecida."
        );
    }

    /**
     * Efectua o "login", procurando uma conta com base no seu pin,
     * e partindo do princípio que o ficheiro de clientes está bem
     * formatado de acordo com as especificações.
     *
     * @see parseValidAccount(String, String)
     *
     * @param pin  o pin da conta, para autenticar
     * @return     objecto de conta se encontrada, null caso contrário
     */
    public Account getAccountWithPin(String pin) {
        Account account = null;
        File dataFile = getFile(DATA_SOURCE);
        try {
            Scanner fileScanner = new Scanner(dataFile, "UTF8");
            while (fileScanner.hasNextLine() && account == null) {
                String client = fileScanner.nextLine();
                account = parseValidAccount(client, pin);
            }
        } catch (FileNotFoundException e) {
            throw new RuntimeException("Ficheiro de clientes não encontrado!");
        }
        return account;
    }

    /**
     * Interpreta uma linha do ficheiro de clientes para um
     * objecto do tipo Account, partindo do princípio que
     * o ficheiro está bem formatado.
     *
     * Formato de cada linha:
     * [pin],[número da conta],[nome do cliente],[ficheiro de dados da conta]
     *
     * @param client  linha formatada do ficheiro de clientes
     * @param pin     o pin da conta, para autenticar
     * @return        objecto de conta
     */
    private Account parseValidAccount(String client, String pin) {
        Account account = null;
```

```
        String[] tokens = client.split(",");
        if (tokens.length != 4) {
            throw new RuntimeException (
                "Ficheiro de clientes formatado incorrectamente."
            );
        }
        if (tokens[0].equals(pin)) {
            String nmbr = tokens[1];
            String name = tokens[2];
            String data = tokens[3];
            AccountMapper datamapper = new AccountMapper(getFile(data));
            account = new Account(nmbr, name, datamapper);
        }
        return account;
    }
}
```

Account.java

```
package atm.model;

import java.util.ArrayList;
import java.util.List;

/**
 * Representa uma conta bancária.
 */
public class Account {

    /** O número de conta */
    private String number;

    /** O nome do cliente */
    private String client;

    /** Saldo da conta */
    private double balance = 0.0;

    /** Colecção com os movimentos */
    private ArrayList<Transaction> transactions = new ArrayList<Transaction>();

    /** Objecto de persistência, com a fonte de dados */
```

```
    private AccountMapper mapper;

    /**
     * Construtor por defeito.
     *
     * Objectos deste tipo apenas devem ser instanciados
     * pelo Atm, ou pelos testes.
     *
     * @see Atm.getAccountWithPin(String)
     *
     * @param number  número de conta
     * @param client  nome do cliente
     * @param mapper  objecto de persistência, com a fonte dos dados
     */
    Account(String number, String client, AccountMapper mapper) {
        this.number = number;
        this.client = client;
        this.mapper = mapper;
        load();
    }

    /** Carrega os dados da persistência */
    private void load() {
        mapper.load(this);
    }

    /** Retorna o número de conta passado para o construtor */
    public String getNumber() {
        return number;
    }

    /** Retorna o nome do cliente passado para o construtor */
    public String getClient() {
        return client;
    }

    /** Retorna o saldo */
    public double getBalance() {
        return balance;
    }

    /**
     * Define o saldo. Usado pelos testes e pela persistência de dados
     *
```

```
     * @param balance  o saldo a definir
     */
    void setBalance(double balance) {
        this.balance = balance;
    }

    /** Retorna os movimentos de conta */
    ArrayList<Transaction> getTransactions() {
        return (ArrayList<Transaction>) transactions.clone();
    }

    /**
     * Regista um movimento de conta
     *
     * @param transaction  objecto de movimento de conta Transaction
     */
    void addTransaction(Transaction transaction) {
        if (transaction != null) {
            transactions.add(transaction);
        }
    }

    /** Retorna o último movimento adicionado */
    Transaction getLastTransaction() {
        return !transactions.isEmpty()
                    ? transactions.get(transactions.size()-1)
                    : null;
    }

    /**
     * Retorna os últimos max movimentos,
     * por ordem do mais recente ao mais antigo
     */
    public List<Transaction> getLatestTransactions(int max) {
        if (transactions.size() < max) {
            max = transactions.size();
        }

        List<Transaction> latest = new ArrayList<Transaction>(max);
        for (int i = transactions.size()-1; max > 0; i--, max--) {
            latest.add(transactions.get(i));
        }

        return latest;
```

```
}

/**
 * Processa um movimento, debitando ou creditando da conta o valor
 * movimentado, assim como registando o movimento. Por último,
 * se tudo tiver corrido bem, os dados são gravados.
 *
 * @param transaction  objecto de movimento de conta
 */
public void processTransaction(Transaction transaction) {
    double amount = transaction.getAmount();
    if (amount > 0) {
        switch (transaction.getType()) {
            case CREDIT:
                credit(amount);
                break;
            case DEBIT:
                debit(amount);
                break;
        }
        addTransaction(transaction);
        mapper.save(this);
    }
}

/**
 * Coloca dinheiro na conta.
 *
 * @param d  valor a creditar
 */
public void credit(double d) {
    balance += Math.abs(d);
}

/**
 * Retira dinheiro da conta, tipicamente para um levantamento ou
 * pagamento de serviços.
 *
 * @param d  valor a debitar
 */
public void debit(double d) {
    d = Math.abs(d);
    if (d > balance) {
        throw new IllegalArgumentException("Não tem saldo suficiente");
```

```
        }
        balance -= d;
    }
}
```

AccountMapper.java

```
package atm.model;

import java.io.EOFException;
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.util.List;

/**
 * Persistência de dados das contas.
 *
 * Esta classe não deve ser usada fora do modelo (package private),
 * e é usada para separação de responsabilidades,
 * o que facilita uma futura manutenção.
 *
 * Em vez de armazenar os dados num ficheiro, pode ser desejado usar
 * uma base de dados ou chamadas pela rede (RPC), como faz um
 * multibanco verdadeiro. As alterações são só precisas aqui.
 */
class AccountMapper {

    /** Ficheiro onde estão armazenados os dados */
    private File data;

    /**
     * Constructor.
     * Não deve ser conhecido fora da camada do modelo.
     *
     * @see Atm.parseValidAccount(String, String)
     *
     * @param data  ficheiro onde estão armazenados os dados
     */
```

```
AccountMapper(File data) {
    this.data = data;
    if (!data.exists()) {
        try {
            data.createNewFile();
        } catch (IOException e) {
            throw new RuntimeException(
                "Não foi possível criar novo ficheiro de dados."
            );
        }
    }
}

/**
 * Guarda o saldo e movimentos de conta no ficheiro
 *
 * @param account  objecto do tipo Account para guardar
 */
public void save(Account account) {
    ObjectOutputStream out = null;
    try {
        out = new ObjectOutputStream(new FileOutputStream(data));
        out.writeDouble(account.getBalance());
        out.writeObject(account.getTransactions());
        out.close();
    } catch (IOException e) {
        throw new RuntimeException("Problema ao guardar dados.");
    }
}

/**
 * Carrega o saldo e movimentos de conta do ficheiro
 *
 * @param account  objecto do tipo Account para onde carregar os dados
 */
public void load(Account account) {
    ObjectInputStream in = null;
    try {
        in = new ObjectInputStream(new FileInputStream(data));
        account.setBalance(in.readDouble());
        for (Transaction trans : (List<Transaction>) in.readObject()) {
            account.addTransaction(trans);
        }
        in.close();
```

```
        } catch (ClassNotFoundException e) {
            throw new RuntimeException(
                "Objecto desconhecido ao carregar dados."
            );
        } catch (EOFException e) {
        } catch (IOException e) {
            throw new RuntimeException("Problema ao recuperar dados.");
        }
    }
}
```

Transaction.java

```
package atm.model;

import java.text.DateFormat;
import java.text.SimpleDateFormat;
import java.util.Date;

/**
 * Representa um movimento de conta.
 */
public class Transaction implements java.io.Serializable {

    /**
     * Versão de serialização
     *
     * @see http://www.mkyong.com/java-best-practices/understand-the-serialversionuid/
     */
    private static final long serialVersionUID = 1L;

    /** Tipo enumerado com os tipos de movimentos */
    enum Type {
        DEBIT  { @Override public String toString() { return "Débito"; } },
        CREDIT { @Override public String toString() { return "Crédito"; } };
    }

    /** Data (e hora) do movimento */
    private Date date;

    /** Descrição do movimento */
    private String description;
```

```
    /** Tipo de movimento */
    private Type type;

    /** Valor do movimento */
    private Double amount;

    /**
     * Construtor.
     *
     * Apenas deve ser usado pelos testes, métodos fábrica abaixo,
     * e AccountMapper, para poder definir uma data específica.
     *
     * @param date          data do movimento
     * @param description   descrição do movimento
     * @param type          tipo do movimento
     * @param amount        valor movimentado
     */
    Transaction(Date date, String description, Type type, double amount) {
        this.date        = (Date) date.clone();
        this.description = description;
        this.type        = type;
        this.amount      = new Double(amount);
    }

    /**
     * Cria um movimento de crédito
     *
     * @param description   descrição do movimento
     * @param amount        valor movimentado
     * @return              novo objecto de movimento Transaction
     */
    public static Transaction newCredit(String description, double amount) {
        return new Transaction(new Date(), description, Type.CREDIT, amount);
    }

    /**
     * Cria um movimento de débito
     *
     * @param description   descrição do movimento
     * @param value         valor movimentado
     * @return              novo objecto de movimento Transaction
     */
    public static Transaction newDebit(String description, double value) {
```

```
        return new Transaction(new Date(), description, Type.DEBIT, value);
    }

    /** Retorna uma string representativa da data do movimento */
    public String getDateString() {
        DateFormat df = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        return df.format(date);
    }

    /** Retorna a descrição do movimento */
    public String getDescription() {
        return description;
    }

    /** Retorna o valor do movimento */
    public double getAmount() {
        return amount.doubleValue();
    }

    /** Retorna o tipo do movimento */
    Type getType() {
        return type;
    }

    /** Retorna uma string representativa do tipo de movimento */
    public String getTypeString() {
        return type.toString();
    }

    /**
     * String formatada que representa este objecto.
     * Usado para mostrar os movimentos no ecrã.
     */
    @Override
    public String toString() {
        return String.format("%s   %-35s   %-7s   %6.2f",
              getDateString(), description, type, amount.doubleValue()
        );
    }

    /**
     * Verifica se um objecto é igual a este.
     * Utilizado nos testes, com assertEquals().
     *
```

```
     * @param obj  a referência do objecto para comparar
     * @return     true se este objecto é igual ao argumento obj;
     *             false caso contrário
     */
    @Override
    public boolean equals(Object obj) {
        if (obj == null) {
            return false;
        }
        if (getClass() != obj.getClass()) {
            return false;
        }
        final Transaction other = (Transaction) obj;
        if (this.date != other.date &&
                (this.date == null || !this.date.equals(other.date))) {
            return false;
        }
        if ((this.description == null)
                ? (other.description != null)
                : !this.description.equals(other.description)) {
            return false;
        }
        if (this.type != other.type) {
            return false;
        }
        if (this.amount != other.amount &&
                (this.amount == null || !this.amount.equals(other.amount))) {
            return false;
        }
        return true;
    }

    /**
     * Retorna um código único para este objecto.
     * Deve ser implementado sempre que se faz o override do equals(Object).
     *
     * @see equals(Object)
     *
     * @return  um código único para este objecto
     */
    @Override
    public int hashCode() {
        int hash = 3;
        hash = 59 * hash + (date != null ? date.hashCode() : 0);
```

```
        hash = 59 * hash + (description != null ? description.hashCode() : 0);
        hash = 59 * hash + (type != null ? type.hashCode() : 0);
        hash = 59 * hash + (amount != null ? amount.hashCode() : 0);
        return hash;
    }
}
```

Payment.java

```
package atm.model;

/**
 * Representa um pagamento de serviços
 */
public class Payment {

    /** Entidade */
    private String entity;

    /** Referência */
    private String reference;

    /** Montante */
    private double amount;

    /** Valida se uma entidade tem 5 dígitos */
    public static boolean isValidEntity(String entity) {
        return validate(entity, 5);
    }

    /** Valida se uma referência tem 9 dígitos */
    public static boolean isValidReference(String reference) {
        return validate(reference, 9);
    }

    /**
     * Construtor
     *
     * @param entity     entidade
     * @param reference  referência
     * @param amount     montante
     */
```

```
    public Payment(String entity, String reference, double amount) {
        StringBuilder error = new StringBuilder();
        if (!isValidEntity(entity)) {
            error.append("Entidade inválida. ");
        }
        if (!isValidReference(reference)) {
            error.append("Referência inválida.");
        }
        if (error.length() != 0) {
            throw new IllegalArgumentException(error.toString());
        }
        this.entity    = entity;
        this.reference = reference;
        this.amount    = Math.abs(amount);
    }

    /** Retorna a entidade passada ao construtor */
    public String getEntity() {
        return entity;
    }

    /** Retorna a referência passada ao construtor */
    public String getReference() {
        return reference;
    }

    /** Retorna o montante passado ao construtor */
    public double getAmount() {
        return amount;
    }

    /**
     * Valida se uma string é um número com size dígitos
     *
     * @param numeric  número a validar
     * @param size     número de dígitos para verificar
     * @return         true se numeric tiver size dígitos;
     *                 false caso contrário
     */
    private static boolean validate(String numeric, int size) {
        java.util.Scanner sc = new java.util.Scanner(numeric);
        return sc.hasNextInt() && numeric.length() == size;
    }
}
```

Apêndice II — atm

Main.java

```
package atm;

public class Main {

    public static void main(String[] args) {
        double startingFunds = 500;

        if (args.length > 0) {
            try {
                startingFunds = Double.parseDouble(args[args.length-1]);
            } catch (NumberFormatException e) {}

            if (args[0].equals("-gui")) {
                atm.ui.Swing.run(startingFunds);
                return;
            }
        }

        atm.ui.Console.run(startingFunds);
    }
}
```

Apêndice III — atm.ui

Console.java

```
package atm.ui;

import atm.model.Atm;
import atm.model.Account;
import atm.model.Payment;
import atm.model.Transaction;

public class Console {

    private static java.util.Scanner input;
    private static java.io.PrintStream out;
    private static java.io.PrintStream err;

    private static Atm atm;
    private static Account account;

    public static void run(double startingFunds) {
        try { // Permitir acentuação na linha de comandos
            input = new java.util.Scanner(System.in);
            out = new java.io.PrintStream(System.out, true, "UTF-8");
            err = new java.io.PrintStream(System.err, true, "UTF-8");
        } catch (java.io.UnsupportedEncodingException e) {
            out = System.out;
            err = System.out;
        }

        try { // Iniciar a aplicação
            atm = new Atm(startingFunds);
            login();
        } catch (RuntimeException e) {
            printErrorMessage(
                "Erro do Sistema. Dirija-se ao multibanco mais próximo.\n"
              + "Diagnóstico: " + e.getMessage()
```

```
            );
            System.exit(1);
        }
    }

    /** Autentica o utilizador, fornecido um pin de acesso */
    private static void login() {
        String pin = askString("PIN: ");

        account = atm.getAccountWithPin(pin);
        printLineBreak();

        if (account == null) {
            printErrorMessage("Pin inválido!");
            login();
        }

        userMenu();
    }

    /** Menu de entrada ao utilizador */
    private static void userMenu() {

        printMenu(null,
            "1. Levantamentos",
            "2. Consulta de saldo de conta",
            "3. Consulta de movimentos de conta",
            "4. Pagamento de serviços",
            "5. Depósitos"
        );

        try {
            switch (getOption()) {
                case 1:
                    withdrawMenu();
                    break;

                case 2:
                    printHeader("Saldo de conta");
                    printStatusMessage(
                        "Conta número: " + account.getNumber() + "\n" +
                        "Saldo Actual: " + currency(account.getBalance())
                    );
                    break;
```

```
                case 3:
                    printHeader("Movimentos de conta");
                    printStatusMessage(
                        "Saldo Actual: " + currency(account.getBalance())
                    );
                    for (Transaction t : account.getLatestTransactions(10)) {
                        out.println(t);
                    }
                    break;

                case 4:
                    servicesPayment();
                    break;

                case 5:
                    printHeader("Depósito");
                    double dep = askDouble("Montante: ");
                    atm.deposit(dep, account);
                    printStatusMessage("Obrigado pelo seu depósito.");
                    break;

                default:
                    printErrorMessage("Opção inválida");
            }
        } catch (IllegalArgumentException e) {
            printErrorMessage(e.getMessage());
        }

        printLineBreak();
        login();
    }

    /** Menu dos levantamentos */
    public static void withdrawMenu() {

        printMenu("Levantamento",
            "1. 20       2. 50",
            "3. 100      4. 150",
            "5. 200      6. Outros valores"
        );

        try {
            switch (getOption()) {
```

```
                case 1: atm.withdraw(20, account); break;
                case 2: atm.withdraw(50, account); break;
                case 3: atm.withdraw(100, account); break;
                case 4: atm.withdraw(150, account); break;
                case 5: atm.withdraw(200, account); break;
                case 6: withdrawOther(); break;
                default:
                    printErrorMessage("Opção inválida");
            }
        } catch (IllegalArgumentException e) {
            printErrorMessage(e.getMessage());
        }

        printLineBreak();
    }

    /** Levantamento de outras importâncias */
    public static void withdrawOther() {
        try{
            int amount = askInt("Montante: ");
            atm.withdraw(amount, account);

        } catch (IllegalArgumentException e) {
            printErrorMessage(e.getMessage());
            withdrawOther();
        }
    }

    /** Menu do pagamento de serviços */
    public static void servicesPayment() {

        printMenu("Pagamento de Serviços",
            "1. Conta de Electricidade",
            "2. Conta da Água",
            "3. Carregamento Telemóvel"
        );

        switch (getOption()) {
            case 1: atm.payElectricityBill(getPayment(), account); break;
            case 2: atm.payWaterBill(getPayment(), account);       break;
            case 3: atm.payPhoneBill(getPhonePayment(), account);  break;
            default:
                printErrorMessage("Opção inválida");
                servicesPayment();
```

```
        }

        printStatusMessage("Pagamento efectuado com sucesso");
        printLineBreak();
    }

    /** Retorna um objecto de pagamento de serviço */
    public static Payment getPayment() {
        try {
            String entity    = askString("Entidade: ");
            String reference = askString("Referência: ");
            double amount    = askDouble("Montante: ");

            printLineBreak();
            return new Payment(entity, reference, amount);

        } catch (IllegalArgumentException e) {
            printErrorMessage(e.getMessage());
            return getPayment();
        }
    }

    /** Retorna um objecto de pagamento de serviço, para um telemóvel */
    public static Payment getPhonePayment() {
        try {
            String phone  = askString("Telemóvel: ");
            String entity = atm.getPhoneEntity(phone);
            double amount = getPhonePaymentAmount();

            printLineBreak();
            return new Payment(entity, phone, amount);

        } catch (IllegalArgumentException e) {
            printErrorMessage(e.getMessage());
            return getPhonePayment();
        }
    }

    /** Menu com quantias de carregamento do telemóvel */
    public static double getPhonePaymentAmount() {
        printMenu("Montante",
            "1. 5 euros",
            "2. 10 euros",
            "3. 20 euros"
```

```
        );
        switch (getOption()) {
            case 1: return 5;
            case 2: return 15;
            case 3: return 20;
            default:
                printErrorMessage("Opção inválida");
                return getPhonePaymentAmount();
        }
    }

    /* Métodos de ajuda */

    private static String currency(double amount) {
        return String.format("%.2f euros", amount);
    }

    private static int getOption() {
        int option = askInt("\n> ");
        printLineBreak();
        return option;
    }

    private static int askInt(String label) {
        out.print(label);
        try {
            int value = input.nextInt();
            clearInput();
            return value;

        } catch (java.util.InputMismatchException e) {
            printErrorMessage("Número inteiro inválido. Tente de novo.");
            clearInput();
            return askInt(label);
        }
    }

    private static double askDouble(String label) {
        out.print(label);
        try {
            double value = input.nextDouble();
            clearInput();
            return value;
```

```
        } catch (java.util.InputMismatchException e) {
            printErrorMessage("Número decimal inválido. Tente de novo.");
            clearInput();
            return askDouble(label);
        }
    }

    private static String askString(String label) {
        out.print(label);
        return input.nextLine();
    }

    private static void clearInput() {
        if (input.hasNextLine()) {
            input.nextLine();
        }
    }

    /*
     * Métodos de abstracção do output para tornar o código mais legível e
     * facilitar a manutenção ao remover uma dependência ao método de output,
     * assim como manter a interface consistente.
     */

    private static void printHeader(String header) {
        out.println("\n\n| "+header+" |\n");
    }

    private static void printMenu(String header, String ... entries) {
        if (header != null) {
            printHeader(header);
        }
        for (int i = 0; i < entries.length; i++) {
            out.println(entries[i]);
        }
    }

    private static void printStatusMessage(String msg) {
        out.println(msg);
        out.println();
    }

    private static void printErrorMessage(String msg) {
        err.println(msg);
```

```
        err.println();
    }

    private static void printLineBreak() {
        out.println();
    }
}
```

Swing.java

```
package atm.ui;

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.border.*;
import atm.model.*;

public class Swing extends JFrame {

    // Este método existe para poder correr a aplicação gráfica directamente
    public static void main(String[] args) {
        run(500);
    }

    public static void run(final double funds) {
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                Thread.currentThread()
                      .setUncaughtExceptionHandler(new UncaughtException());
                new Swing(funds).setVisible(true);
            }
        });
    }

    private final String CONFIRM = "Confirmar";
    private final String OTHEROP = "Outras operações";
    private final String ABORT_T = "Abortar";

    private Atm atm;
    private Account account;
```

```
    private JButton confirm;
    private JButton abort;

    public Swing(double startingFunds) {
        atm = new Atm(startingFunds);
        Dimension screenSize = Toolkit.getDefaultToolkit().getScreenSize();

        setTitle("Multibanco");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setMinimumSize(new Dimension(400, 350));
        setLocation(screenSize.width/3, screenSize.height/5);

        reset();
    }

    private void reset() {
        JPanel root = new JPanel(new BorderLayout());
        root.setBorder(new EmptyBorder(10, 10, 10, 10));
        root.add(operations(), BorderLayout.PAGE_END);
        root.add(loginScreen(), BorderLayout.CENTER);
        root.revalidate();
        setContentPane(root);
        account = null;
        pack();
    }

    private void updateContent(JComponent component) {
        JPanel root = (JPanel) getContentPane();
        BorderLayout layout = (BorderLayout) root.getLayout();
        root.remove(layout.getLayoutComponent(root, BorderLayout.CENTER));
        root.add(component, BorderLayout.CENTER);
        root.revalidate();
    }

    private JComponent operations() {
        confirm = new JButton(CONFIRM);
        abort   = new JButton(ABORT_T);

        abort.addActionListener(new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                reset();
            }
        });
```

```
        Box buttonbox = Box.createHorizontalBox();
        buttonbox.setBorder(new EmptyBorder(10, 0, 0, 0));
        buttonbox.add(Box.createHorizontalGlue());
        buttonbox.add(confirm);
        buttonbox.add(Box.createHorizontalGlue());
        buttonbox.add(abort);
        buttonbox.add(Box.createHorizontalGlue());
        return buttonbox;
    }

    private JComponent loginScreen() {
        JLabel lblPin = new JLabel("Introduza o seu PIN:");

        final JPasswordField pwdPin = new JPasswordField(7);
        pwdPin.setMaximumSize(pwdPin.getPreferredSize());
        pwdPin.setHorizontalAlignment(JTextField.CENTER);

        ActionListener loginListener = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String pin = String.valueOf(pwdPin.getPassword());
                account = atm.getAccountWithPin(pin);
                if (account != null) {
                    showMainMenu();
                } else {
                    showError("Pin inválido. Tente de novo.", pwdPin);
                }
            }
        };

        pwdPin.addActionListener(loginListener);
        confirm.addActionListener(loginListener);

        Box login = Box.createVerticalBox();
        login.setPreferredSize(new Dimension(300, 220));

        login.add(Box.createVerticalGlue());
        login.add(centerComponent(lblPin));
        login.add(Box.createRigidArea(new Dimension(0, 5)));
        login.add(centerComponent(pwdPin));
        login.add(Box.createVerticalGlue());

        pwdPin.requestFocusInWindow();
        return login;
    }
```

```
private JComponent mainMenu() {
    JButton withdrawals  = new JButton("Levantamentos");
    JButton checkbalance = new JButton("Consulta de saldo da conta");
    JButton transactions = new JButton("Consulta de movimentos de conta");
    JButton payments     = new JButton("Pagamento de serviços");
    JButton deposits     = new JButton("Depósitos");

    withdrawals.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            updateContent(withdrawalScreen());
            confirm.setEnabled(true);
            confirm.addActionListener(new MenuListener());
        }
    });

    checkbalance.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            updateContent(balanceScreen());
            confirm.setEnabled(true);
            confirm.addActionListener(new MenuListener());
        }
    });

    transactions.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            updateContent(transactionsScreen());
            confirm.setEnabled(true);
            confirm.addActionListener(new MenuListener());
        }
    });

    payments.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            updateContent(paymentsScreen());
            confirm.setEnabled(true);
            confirm.addActionListener(new MenuListener());
        }
    });

    deposits.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            updateContent(depositScreen());
            confirm.setText(CONFIRM);
```

```
                confirm.setEnabled(true);
            }
        });

        confirm.setText(OTHEROP);

        JPanel grid = new JPanel(new GridLayout(0, 1, 0, 3));
        grid.add(withdrawals);
        grid.add(checkbalance);
        grid.add(transactions);
        grid.add(payments);
        grid.add(deposits);

        Box menu = Box.createVerticalBox();
        menu.add(screenTitle("Menu Principal"));
        menu.add(Box.createRigidArea(new Dimension(0, 15)));
        menu.add(grid);
        return menu;
    }

    private JComponent withdrawalScreen() {
        ActionListener withdrawalListener = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JButton source = (JButton) e.getSource();
                int amount = Integer.parseInt(source.getActionCommand());
                if (amount != 0) {
                    try {
                        atm.withdraw(amount, account);
                    } catch (IllegalArgumentException ex) {
                        showError(ex.getMessage());
                    }
                    showMainMenu();
                } else {
                    updateContent(otherWithdrawalScreen());
                }
            }
        };

        JButton btn;
        String[] order = {"20", "50", "100", "150", "200", "0"};

        JPanel grid = new JPanel(new GridLayout(0, 2));
        for (int i = 0; i < order.length; i++) {
            btn = new JButton(
```

```
                order[i].equals("0") ? "Outros valores" : order[i]
            );
            btn.setActionCommand(order[i]);
            btn.addActionListener(withdrawalListener);

            grid.add(btn);
        }

        Box screen = Box.createVerticalBox();
        screen.add(screenTitle("Levantamento"));
        screen.add(Box.createRigidArea(new Dimension(0, 15)));
        screen.add(grid);
        screen.add(Box.createVerticalGlue());
        return screen;
    }

    private JComponent otherWithdrawalScreen() {
        confirm.setText(CONFIRM);
        removeActionListeners(confirm);

        final JTextField withdrawal = new JTextField(6);
        withdrawal.setMaximumSize(withdrawal.getPreferredSize());
        withdrawal.setHorizontalAlignment(JTextField.CENTER);

        ActionListener otherWithdrawalListener = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    atm.withdraw(validateAmount(withdrawal.getText()), account);
                    showMainMenu();
                } catch (IllegalArgumentException ex) {
                    showError(ex.getMessage(), withdrawal);
                }
            }
        };

        withdrawal.addActionListener(otherWithdrawalListener);
        confirm.addActionListener(otherWithdrawalListener);

        Box screen = Box.createVerticalBox();
        screen.add(screenTitle("Levantamento"));
        screen.add(screenTitle("de outras importâncias"));
        screen.add(Box.createVerticalGlue());
        screen.add(centerComponent(withdrawal));
        screen.add(Box.createVerticalGlue());
```

```
        return screen;

    }

    private JComponent balanceScreen() {
        Box screen = Box.createVerticalBox();

        screen.add(screenTitle("Saldo de Conta"));
        screen.add(Box.createVerticalGlue());

        screen.add(centerComponent(boldLabel("Conta Número")));
        screen.add(Box.createRigidArea(new Dimension(0, 5)));
        screen.add(centerComponent(
            new JLabel(account.getNumber()))
        );
        screen.add(Box.createVerticalGlue());

        screen.add(centerComponent(boldLabel("Saldo Actual")));
        screen.add(Box.createRigidArea(new Dimension(0, 5)));
        screen.add(centerComponent(
            new JLabel(formatCurrency(account.getBalance()))
        ));
        screen.add(Box.createVerticalGlue());

        return screen;
    }

    private JComponent transactionsScreen() {
        java.util.List transactions = account.getLatestTransactions(10);

        String[] columns = {"Data", "Descrição", "Tipo", "Valor"};
        Object[][] data  = new Object[transactions.size()][4];

        for (int i = 0; i < transactions.size(); i++) {
            Transaction trans = (Transaction) transactions.get(i);
            data[i] = new Object[] {
                trans.getDateString(),
                trans.getDescription(),
                trans.getTypeString(),
                trans.getAmount()
            };
        }

        JTable table = new JTable(data, columns);
```

```
        JScrollPane scrollPane = new JScrollPane(table);
        table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        table.setFillsViewportHeight(true);
        table.setEnabled(false);

        Box screen = Box.createVerticalBox();
        screen.add(screenTitle("Movimentos"));
        screen.add(Box.createRigidArea(new Dimension(0, 10)));
        screen.add(centerComponent(new JLabel(
            "<html><b>Saldo actual:</b> "
            +formatCurrency(account.getBalance())+"</html>"
        )));
        screen.add(Box.createRigidArea(new Dimension(0, 10)));
        screen.add(scrollPane);
        screen.add(Box.createVerticalGlue());

        return screen;
    }

    private JComponent paymentsScreen() {
        JButton elect = new JButton("Conta de Electricidade");
        JButton water = new JButton("Conta da Água");
        JButton phone = new JButton("Carregamento Telemóvel");

        elect.setActionCommand("e");
        water.setActionCommand("w");
        phone.setActionCommand("p");

        ActionListener choosePaymentListener = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                JButton source = (JButton) e.getSource();
                updateContent(
                    paymentScreen(source.getText(), source.getActionCommand())
                );
            }
        };

        elect.addActionListener(choosePaymentListener);
        water.addActionListener(choosePaymentListener);
        phone.addActionListener(choosePaymentListener);

        JPanel grid = new JPanel(new GridLayout(0, 1));
        grid.add(elect);
        grid.add(water);
```

```
        grid.add(phone);

        Box screen = Box.createVerticalBox();
        screen.add(screenTitle("Pagamento de Serviços"));
        screen.add(Box.createRigidArea(new Dimension(0, 15)));
        screen.add(grid);
        screen.add(Box.createVerticalGlue());
        return screen;
    }

    private JComponent paymentScreen(String title, final String command) {
        confirm.setText(CONFIRM);
        removeActionListeners(confirm);

        final JTextField fldEntity = new JTextField(6);
        final JTextField fldRefnce = new JTextField(10);
        final JTextField fldPhone  = new JTextField(10);
        final JTextField fldAmount = new JTextField(6);

        final String[] labels = command.equals("p")
            ? new String[] {"Telemóvel", "Montante"}
            : new String[] {"Entidade", "Referência", "Montante"};

        final JTextField[] fields = command.equals("p")
            ? new JTextField[] {fldPhone, fldAmount}
            : new JTextField[] {fldEntity, fldRefnce, fldAmount};

        ActionListener paymentListener = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                try {
                    String refnce = command.equals("p")
                        ? fldPhone.getText()
                        : fldRefnce.getText();

                    String entity = command.equals("p")
                        ? atm.getPhoneEntity(refnce)
                        : fldEntity.getText();

                    double amount = validateAmount(fldAmount.getText());

                    Payment payment = new Payment(entity, refnce, amount);

                    switch (command.charAt(0)) {
                        case 'p': atm.payPhoneBill(payment, account); break;
```

```
                    case 'w': atm.payWaterBill(payment, account); break;
                    case 'e': atm.payElectricityBill(payment, account);
                                                                    break;
                    default:
                        throw new RuntimeException(
                            "Pagamento de serviço desconhecido!"
                        );
                }
                showMainMenu();
            } catch (IllegalArgumentException ex) {
                showError(ex.getMessage(), fields);
            }
        }
    };

    confirm.addActionListener(paymentListener);

    Box screen = Box.createVerticalBox();
    screen.add(screenTitle(title));
    screen.add(Box.createVerticalGlue());

    for (int i = 0; i < labels.length; i++) {
        fields[i].addActionListener(paymentListener);
        fields[i].setHorizontalAlignment(JTextField.CENTER);
        fields[i].setMaximumSize(fields[i].getPreferredSize());

        screen.add(centerComponent(boldLabel(labels[i])));
        screen.add(centerComponent(fields[i]));
        screen.add(Box.createVerticalGlue());
    }

    return screen;
}

private JComponent depositScreen() {
    final JTextField deposit = new JTextField(6);
    deposit.setMaximumSize(deposit.getPreferredSize());
    deposit.setHorizontalAlignment(JTextField.CENTER);

    ActionListener depositListener = new ActionListener() {
        public void actionPerformed(ActionEvent e) {
            try {
                atm.deposit(validateAmount(deposit.getText()), account);
                showMainMenu();
```

```
                } catch (IllegalArgumentException ex) {
                    showError(ex.getMessage(), deposit);
                }
            }
        };

        deposit.addActionListener(depositListener);
        confirm.addActionListener(depositListener);

        Box screen = Box.createVerticalBox();
        screen.add(screenTitle("Depósito"));
        screen.add(Box.createVerticalGlue());
        screen.add(centerComponent(deposit));
        screen.add(Box.createVerticalGlue());
        return screen;
    }

    class MenuListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            showMainMenu();
        }
    }

    private void showMainMenu() {
        updateContent(mainMenu());
        removeActionListeners(confirm);
        confirm.setEnabled(false);
    }

    private void removeActionListeners(JButton button) {
        for (ActionListener al : button.getActionListeners()) {
            button.removeActionListener(al);
        }
    }

    private JComponent centerComponent(JComponent component) {
        component.setAlignmentX(Component.CENTER_ALIGNMENT);
        return component;
    }

    private JLabel screenTitle(String text) {
        JLabel title = new JLabel(text);
        title.setFont(new Font(Font.SERIF, Font.PLAIN, 20));
        title.setAlignmentX(Component.CENTER_ALIGNMENT);
```

```
        return title;
    }

    private JLabel boldLabel(String text) {
        JLabel newLabel = new JLabel(text);
        newLabel.setFont(newLabel.getFont().deriveFont(Font.BOLD));
        return newLabel;
    }

    private void showError(String error, JTextField reset) {
        showError(error, new JTextField[] {reset});
    }

    private void showError(String error, JTextField[] reset) {
        showError(error);
        for (int i = 0; i < reset.length; i++) {
            reset[i].setText("");
        }
        reset[0].grabFocus();
    }

    private void showError(String error) {
        JOptionPane.showMessageDialog(Swing.this, error, "Erro",
            JOptionPane.ERROR_MESSAGE
        );
    }

    private double validateAmount(String value) {
        try {
            return Double.parseDouble(value);
        } catch (NumberFormatException e) {
            throw new IllegalArgumentException("Montante inválido.");
        }
    }

    private String formatCurrency(double amount) {
        return String.format("%.2f euros", amount);
    }
}

class UncaughtException implements Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread th, Throwable ex) {
        System.err.println("Erro fatal: " + ex.getMessage());
        JOptionPane.showMessageDialog(null,
```

```
            "Erro do Sistema. Dirija-se ao multibanco mais próximo.",
            "Erro Fatal", JOptionPane.ERROR_MESSAGE
        );
        System.exit(1);
    }
}
```

Apêndice IV — Testes

TestSuite.java

```
package atm;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import atm.model.*;

@RunWith(Suite.class)
@Suite.SuiteClasses({
    AcceptanceTests.class,
    AtmTest.class,
    TransactionTest.class,
    AccountMapperTest.class,
    AccountTest.class,
    PaymentTest.class
})
public class TestSuite {}
```

AcceptanceTests.java

```
package atm;

import atm.model.*;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class AcceptanceTests {

    private Atm atm;
    private Account account;

    @Before
```

```
    public void setUp() {
        atm = new Atm(1000);
    }

    @Test
    public void canGetAccount() {
        account = atm.getAccountWithPin("1234");
        assertEquals("0010029289641272009", account.getNumber());
        assertEquals("Rui Filipe Tavares Melo", account.getClient());
    }
}
```

AtmTest.java

```
package atm.model;

import java.io.IOException;
import org.junit.rules.TemporaryFolder;
import org.junit.Rule;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class AtmTest {

    @Rule public TemporaryFolder bucket = new TemporaryFolder();

    private final double PRECISION = 1e-6;

    private Atm atm;
    private AccountMapper mapper;
    private Account account;
    private Payment payment;

    @Before
    public void setUp() throws IOException {
        atm     = new Atm(300);
        mapper  = new AccountMapper(bucket.newFile("testFile.dat"));
        account = new Account("123456789", "Dummy client", mapper);
        payment = new Payment("12345", "123456789", 60.53);
        account.setBalance(600);
    }
```

```
@Test
public void testHasEnoughWithdrawalFunds() {
    atm = new Atm(5);
    assertFalse("can't allow when 5", atm.hasEnoughWithdrawalFunds());
    atm = new Atm(10);
    assertTrue("must allow when 10", atm.hasEnoughWithdrawalFunds());
}

@Test
public void depositDoesNotUpdateFunds() {
    atm.deposit(50, account);
    assertEquals(300, atm.getFunds(), PRECISION);
}

@Test
public void depositIsRegistered() {
    atm.deposit(300, account);
    Transaction expected = Transaction.newCredit("Depósito MB", 300);
    assertTrue(TransactionTest.equalTransactions(
            expected, account.getLastTransaction()
    ));
}

@Test
public void canMakeWithdrawal() {
    atm.withdraw(50, account);
    assertEquals(250, atm.getFunds(), PRECISION);
}

@Test
public void negativeValuesWithdrawAPositiveAmount() {
    atm.withdraw(-50, account);
    assertEquals(250, atm.getFunds(), PRECISION);
}

@Test(expected=IllegalArgumentException.class)
public void doesNotAllowWithdrawalsNotMultipleOfFive() {
    atm.withdraw(57, account);
}

@Test(expected=IllegalArgumentException.class)
public void doesNotAllowWithdrawalsBellowTen() {
    atm.withdraw(5, account);
```

```
    }

    @Test(expected=IllegalArgumentException.class)
    public void doesNotAllowWithdrawalsAboveTwoHundred() {
        atm.withdraw(210, account);
    }

    @Test(expected=IllegalArgumentException.class)
    public void doesNotAllowWithdrawalWhenNotEnoughFunds() {
        atm = new Atm(5);
        atm.withdraw(10, account);
    }

    @Test
    public void allowsWithrawingAllFunds() {
        atm = new Atm(100);
        atm.withdraw(100, account);
        assertEquals(0, atm.getFunds(), PRECISION);
    }

    @Test
    public void withdrawalIsRegistered() {
        atm.withdraw(200, account);
        Transaction expected = Transaction.newDebit("Levantamento MB", 200);
        assertTrue(TransactionTest.equalTransactions(
                expected, account.getLastTransaction()
        ));
    }

    @Test
    public void canPayWaterBill() {
        Transaction expected =
                Transaction.newDebit("Pagamento de serviços Água", 60.53);

        atm.payWaterBill(payment, account);

        assertEquals(expected, account.getLastTransaction());
        assertEquals(539.47, account.getBalance(), 1e-6);
    }

    @Test
    public void canPayElectricityBill() {
        Transaction expected =
            Transaction.newDebit("Pagamento de serviços Electricidade", 60.53);
```

```
        atm.payElectricityBill(payment, account);

        assertEquals(expected, account.getLastTransaction());
        assertEquals(539.47, account.getBalance(), 1e-6);
    }

    @Test
    public void canPayPhoneBill() {
        Transaction expected =
                Transaction.newDebit("Pagamento de serviços Telemóvel", 60.53);

        atm.payPhoneBill(payment, account);

        assertEquals(expected, account.getLastTransaction());
        assertEquals(539.47, account.getBalance(), 1e-6);
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantPayWaterBillForLackOfFunds() {
        account.setBalance(50);
        atm.payWaterBill(payment, account);
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantPayElectricityBillForLackOfFunds() {
        account.setBalance(50);
        atm.payElectricityBill(payment, account);
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantPayPhoneBillForLackOfFunds() {
        account.setBalance(50);
        atm.payPhoneBill(payment, account);
    }

    @Test
    public void canGetCorrectPhoneEntities() {
        assertEquals("10158", atm.getPhoneEntity("918135235"));
        assertEquals("20638", atm.getPhoneEntity("932352352"));
        assertEquals("10559", atm.getPhoneEntity("965234235"));
    }

    @Test(expected=IllegalArgumentException.class)
```

```
    public void cantGetEntityFromUnknownPhoneNetwork() {
        atm.getPhoneEntity("123456789");
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantGetEntityFromALessThenNineDigitPhone() {
        atm.getPhoneEntity("91234462");
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantGetEntityFromAMoreThenNineDigitPhone() {
        atm.getPhoneEntity("9123423465");
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantGetEntityFromNotANumber() {
        atm.getPhoneEntity("912b3c234");
    }
}
```

AccountTest.java

```
package atm.model;

import java.util.List;
import java.util.ArrayList;
import java.io.IOException;
import org.junit.rules.TemporaryFolder;
import org.junit.Rule;
import org.junit.Before;
import org.junit.Test;
import static org.junit.Assert.*;

public class AccountTest {

    @Rule public TemporaryFolder bucket = new TemporaryFolder();

    private final double PRECISION = 1e-6;
    private final String testAccountNum = "0010029289641272009";
    private final String testClientName = "Rui Filipe Tavares Melo";

    private AccountMapper mapper;
```

```
    private Account account;

    @Before
    public void setUp() throws IOException {
        mapper  = new AccountMapper(bucket.newFile("testFile.dat"));
        account = new Account(testAccountNum, testClientName, mapper);
        account.setBalance(600);
    }

    @Test
    public void accountLoadsBalance() {
        assertEquals(600.0, account.getBalance(), PRECISION);
    }

    @Test
    public void canMakeCredit() {
        account.setBalance(600.0);
        account.credit(200.0);
        assertEquals(800.0, account.getBalance(), PRECISION);
    }

    @Test
    public void negativeValuesCreditAPositiveAmount() {
        account.setBalance(100.0);
        account.credit(-50.0);
        assertEquals(150.0, account.getBalance(), PRECISION);
    }

    @Test
    public void canMakeDebit() {
        account.setBalance(600.0);
        account.debit(50.0);
        assertEquals(550.0, account.getBalance(), PRECISION);
    }

    @Test
    public void negativeValuesDebitAPositiveAmount() {
        account.setBalance(100.0);
        account.debit(-50.0);
        assertEquals(50.0, account.getBalance(), PRECISION);
    }

    @Test(expected=IllegalArgumentException.class)
    public void doesNotAllowDebitWhenNotEnoughFunds() {
```

```
        account.setBalance(50.0);
        account.debit(100.0);
    }

    @Test
    public void allowsDebitOfAllFunds() {
        account.setBalance(150.0);
        account.debit(150.0);
        assertEquals(0.0, account.getBalance(), PRECISION);
    }

    @Test
    public void canGetTenLastestTransactions() {
        List<Transaction> source = generateTransactions(15);
        List<Transaction> actual = account.getLatestTransactions(10);
        assertLatestTransactions(10, source, actual);
    }

    @Test
    public void canGetLastestTransactionsWhenLessThenTenAvailable() {
        List<Transaction> source = generateTransactions(5);
        List<Transaction> actual = account.getLatestTransactions(10);
        assertLatestTransactions(5, source, actual);
    }

    private void assertLatestTransactions(int expected,
                                          List<Transaction> source,
                                          List<Transaction> actual) {
        assertEquals(expected, actual.size());
        for (int i = 0; i < expected; i++) {
            assertEquals(
                source.get(source.size()-1-i).getAmount(),
                actual.get(i).getAmount(),
                1e-6
            );
        }
    }

    private List<Transaction> generateTransactions(int num) {
        Transaction transaction;
        List<Transaction> transactions = new ArrayList<Transaction>(num);
        for (int i = 1; i <= num; i++) {
            transaction = Transaction.newCredit("Crédito "+i, 100+2*i);
            transactions.add(transaction);
```

```
            account.addTransaction(transaction);
        }
        return transactions;
    }
}
```

AccountMapperTest.java

```
package atm.model;

import org.junit.Rule;
import org.junit.rules.TemporaryFolder;
import org.junit.Before;
import java.text.ParseException;
import java.text.SimpleDateFormat;
import java.text.DateFormat;
import java.io.IOException;
import java.util.ArrayList;
import org.junit.Test;
import static org.junit.Assert.*;

public class AccountMapperTest {

    @Rule public TemporaryFolder bucket = new TemporaryFolder();

    private Account account;
    private AccountMapper mapper;
    private ArrayList<Transaction> transactions = new ArrayList<Transaction>();

    private void setUpTransactions() throws ParseException {
        DateFormat df = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
        transactions.add(new Transaction(
            df.parse("15/11/2010 17:03:33"),
            "Depósito MB", Transaction.Type.CREDIT, 500
        ));
        transactions.add(new Transaction(
            df.parse("17/11/2010 14:20:12"),
            "Levantamento MB", Transaction.Type.DEBIT, 150
        ));
        transactions.add(new Transaction(
            df.parse("20/11/2010 20:13:44"),
            "Depósito MB", Transaction.Type.CREDIT, 250
```

```
        ));
        transactions.add(new Transaction(
            df.parse("22/11/2010 14:20:12"),
            "Pagamento de serviços Electricidade", Transaction.Type.DEBIT, 250
        ));
    }

    private void setUpAccount(double startingBalance) {
        account = new Account("123456789", "Dummy User", mapper);
        account.setBalance(startingBalance);
    }

    @Before
    public void setUp() throws IOException, ParseException {
        mapper = new AccountMapper(bucket.newFile("testFile.dat"));
        setUpTransactions();
        setUpAccount(600);
        for (Transaction transaction : transactions) {
            account.addTransaction(transaction);
        }
        mapper.save(account);
    }

    @Test
    public void canLoadDataFromMapper() {
        setUpAccount(0);
        mapper.load(account);
        assertEquals(600, account.getBalance(), 1e-6);
        assertTrue(account.getTransactions().containsAll(transactions));
    }
}
```

TransactionTest.java

```
package atm.model;

import org.junit.Test;
import static org.junit.Assert.*;

public class TransactionTest {

    private final Transaction credit =
```

```
            Transaction.newCredit("Test credit", 200);

    private final Transaction debit =
            Transaction.newDebit("Test debit", 200);

    /** Verifica se dois movimentos são iguais, ignorando a data */
    public static boolean equalTransactions(
                        Transaction expected, Transaction actual) {
        return actual.getDescription().equals(expected.getDescription())
            && (Double.compare(expected.getAmount(), actual.getAmount()) == 0)
            && (expected.getType() == actual.getType());
    }

    @Test
    public void canInstanciateNewCredit() {
        assertEquals(Transaction.Type.CREDIT, credit.getType());
    }

    @Test
    public void canInstanciateNewDebit() {
        assertEquals(Transaction.Type.DEBIT, debit.getType());
    }

}
```

PaymentTest.java

```
package atm.model;

import org.junit.Test;
import static org.junit.Assert.*;

public class PaymentTest {

    @Test(expected=IllegalArgumentException.class)
    public void cantUseNonDigitsReference() {
        new Payment("12345", "1234a4567", 20);
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantUseReferenceWithLessThenNineDigits() {
        new Payment("12345", "12345", 20);
```

```
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantUseReferenceWithMoreThenNineDigits() {
        new Payment("12345", "1234567891", 20);
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantUseNonDigitEntity() {
        new Payment("12a45", "123456789", 20);
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantUseEntityWithLessThenFiveDigits() {
        new Payment("123", "123456789", 20);
    }

    @Test(expected=IllegalArgumentException.class)
    public void cantUseEntityWithMoreThenFiveDigits() {
        new Payment("123456", "123456789", 20);
    }

    @Test
    public void paymentWithNegativeAmountShouldBeSetToPositive() {
        Payment payment = new Payment("12345", "123456789", -20);
        assertEquals(20.0, payment.getAmount(), 1e-6);
    }
}
```