6. Estudo de Caso: Caixa Automático

Neste capítulo é desenvolvido passo a passo um exemplo completo de aplicação em Java, utilizando os conceitos de objeto, classe, instância, mensagem, encapsulamento, operação, interface pública, referência de objeto, estado, método e método construtor.

Ao final deste capítulo o estudante deverá ser capaz de, a partir de um modelo de objetos e das especificações das operações a serem realizadas, desenvolver aplicações simples em Java.

6.1. Descrição do Problema

Nosso objetivo é desenvolver uma aplicação para um banco que mantém uma rede de caixas automáticos, onde os clientes podem consultar seus saldos e efetuar saques.

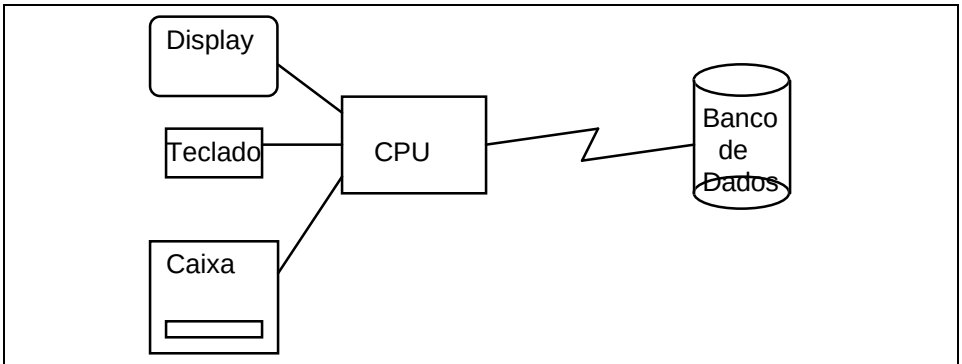

Fig. 6.1(a) - Arquitetura Física do Sistema Caixa Automático

A Figura 6.1(a) apresenta os principais componentes de um caixa automático. Os caixas são operados através de um teclado numérico e um pequeno display, que substitui o vídeo. Toda a operação é controlada por uma CPU local, onde será instalada a aplicação. Há ainda uma ligação com o banco de dados que armazena as informações sobre as contas correntes e com o caixa propriamente dito, que fornece as cédulas aos clientes.

Os caixas automáticos operam em dois modos distintos: supervisor e cliente. No modo supervisor, que é o modo inicial de operação, só é possível realizar uma operação de recarga de numerário, efetuada por um supervisor que possua uma senha apropriada. Após uma operação de recarga o caixa passa automaticamente para o modo cliente, iniciando com R$5.000,00 em notas de R$10,00. No modo cliente, o caixa pode efetuar consultas de saldo e saques, exigindo a identificação do cliente através do número da conta e senha correspondente. Só são permitidos saques de valores múltiplos de R$10,00, até um máximo de R$200,00 e desde que haja saldo suficiente na conta do cliente. Sempre que o saldo disponível cair abaixo de R$200,00 o caixa volta automaticamente ao modo supervisor, exigindo uma nova recarga.

Na Figura 6.1(b) estão apresentados os vários casos de uso previstos para o sistema, através da notação UML para diagramas de casos de uso (*use case diagrams*).

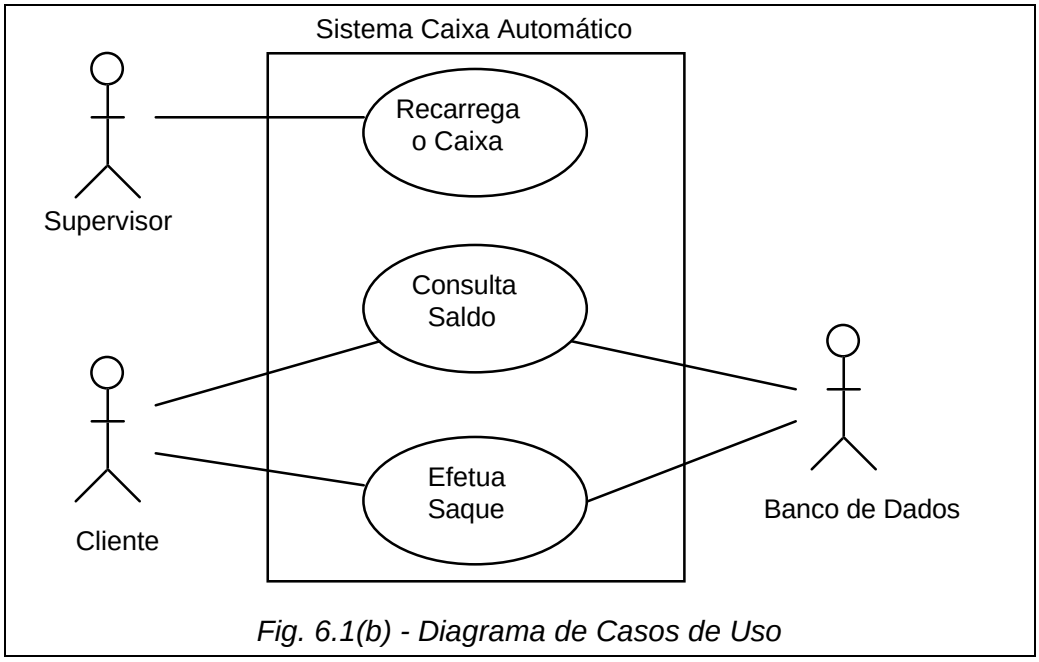

Fig. 6.1(b) - Diagrama de Casos de Uso

6.2. Análise e Projeto do Sistema Caixa Automático

No projeto do sistema definimos as seguintes classes:

`TrmCxAut`

Os objetos dessa classe são os terminais de caixas automáticos, compostos pelo teclado, display e caixa. Essa classe encapsula toda a interface com os usuários. Com isso, a aplicação pode ser adaptada para outros tipos de terminais que usem cartões magnéticos ou telas sensíveis ao toque, por exemplo, substituindo-se apenas essa classe, de forma transparente para o restante da aplicação. Como há uma CPU em cada caixa automático, haverá um único objeto desse tipo em cada instância do programa.

`Caixa`

Os objetos dessa classe são os caixas propriamente ditos, que armazenam as notas para pagamento dos saques efetuados pelos clientes. Essa classe encapsula a política do banco em relação aos saques em caixas automáticos e as interfaces entre um objeto da classe `TrmCxAut` com o restante do sistema. Como em cada caixa automático há somente um caixa, haverá um objeto desse tipo para cada objeto da classe `TrmCxAut`, portanto haverá também um único objeto dessa classe em cada instância do programa.

`ContaCor`

Os objetos dessa classe são as contas mantidas pelos clientes do banco. Essa classe encapsula as políticas do banco para manutenção das contas correntes. Os objetos dessa classe são independentes dos objetos das classes anteriores.

`CadastroContas`

Essa classe contém um único objeto, que é o banco de dados que armazena as informações sobre as contas correntes.

A Figura 6.2(a) apresenta o diagrama de classes correspondente, na notação UML.

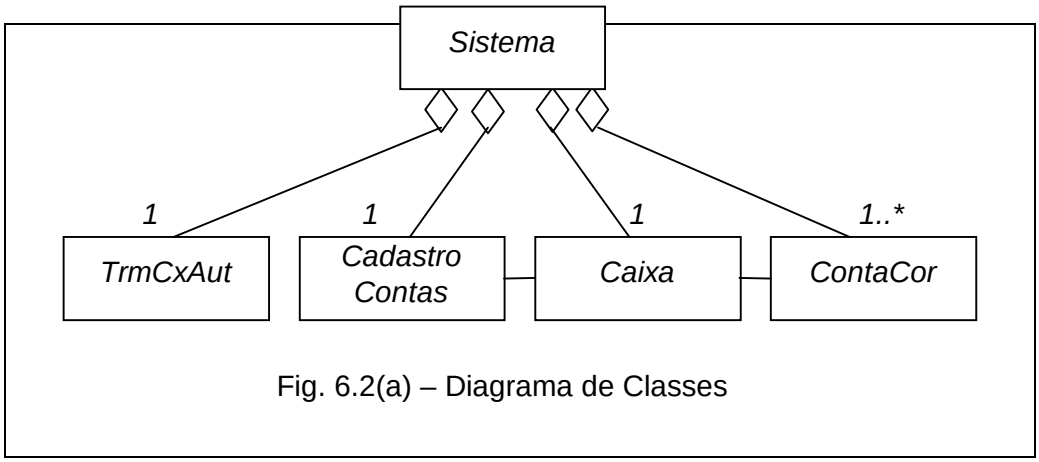

Fig. 6.2(a) – Diagrama de Classes

Interface da Classe `TrmCxAut`

Um objeto da classe `TrmCxAut` possui duas operações: uma para executar um ciclo de operações do caixa e outra que permite mudar o seu modo de operação. A primeira operação é implementada através do método `iniciaOperacao` , que tem como parâmetro uma referência para o banco de dados onde estão armazenadas as contas, e a segunda através do método `setModo`. Esse último método é ativado por uma mensagem com um único parâmetro, do tipo inteiro, que especifica o novo modo de operação: 0 para modo supervisor ou 1 para modo cliente. Ambos métodos não retornam nenhum resultado.

Na especificação da classe `TrmCxAut`, as definições desses métodos têm as seguintes assinaturas:

```
void iniciaOperacao (CadastroContas db)
void setModo (int modo)
```

Interface da Classe `Caixa`

Um objeto da classe `Caixa` possui operações para efetuar uma consulta de saldo, efetuar um saque e recarregar o caixa. Essas operações devem ser implementadas através de métodos com as seguintes assinaturas:

```
float consultaSaldo (int num, int pwd)
boolean efetuaSaque (int num, int pwd, float val)
void recarrega(int pwd )
```

O método `consultaSaldo` deve retornar o valor do saldo da conta se o número da conta e a senha estiverem corretas, ou -1 em caso contrário.

O método `efetuaSaque` deve retornar `true`, se o pedido de saque foi atendido, ou `false` em caso contrário.

Há também um método `liberaNota` que controla o dispositivo que efetua o pagamento, com a seguinte assinatura:

```
private void liberaNota (int qtd)
```

A visibilidade `private` garante que o método só pode ser ativado por objetos da classe onde o método é definido. Essa é uma restrição desejável para o método `liberaNota`, para assegurar que o dispositivo não será acionado por outros objetos.

Interface da Classe `ContaCor`

Um objeto da classe `ContaCor` possui operações para se obter o saldo da conta e debitar valores da conta, além de outras que não interessam à nossa aplicação. Essas operações são implementadas através de métodos com as seguintes assinaturas:

```
float getSaldo  (int pwd)
boolean debitaValor (String hist, float val, int pwd)
```

Interface da Classe `CadastroContas`

Para nossa aplicação a única operação relevante nessa classe é recuperar a referência de uma conta já existente, a partir do número da conta. Essa operação é implementada através de um método com a seguinte assinatura:

```
ContaCor buscaConta (int numcta)
```

Interação entre os Objetos

As figuras 6.2(b), 6.2(c) e 6.2(d) apresentam, através de diagramas de seqüência, a interação entre os objetos das classes da aplicação para realizar as ações definidas no diagrama de casos de uso.

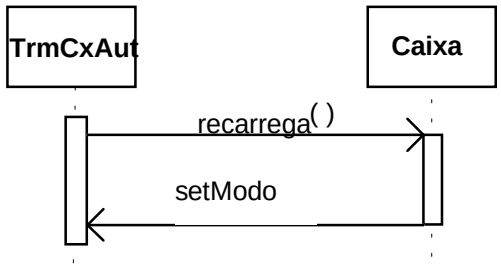

Fig. 6.2(b) - Diagrama de Seqüência para Recarga do Caixa

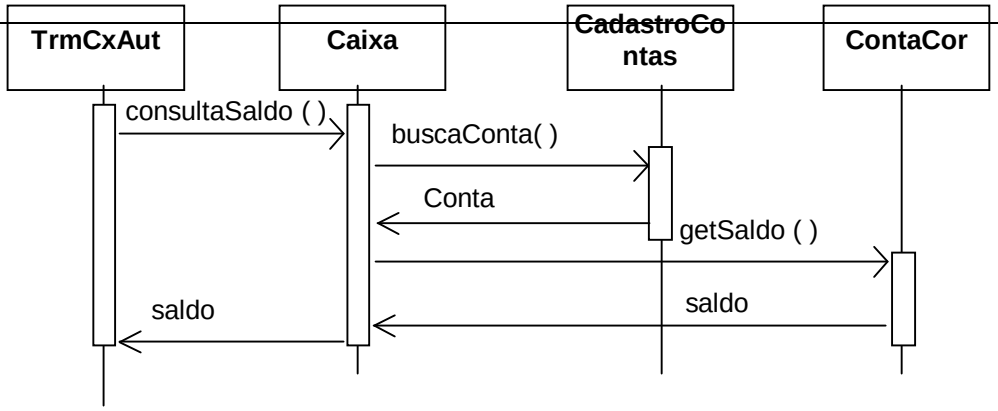

Fig. 6.2(c) – Diagrama de Seqüência para Consulta de Saldo

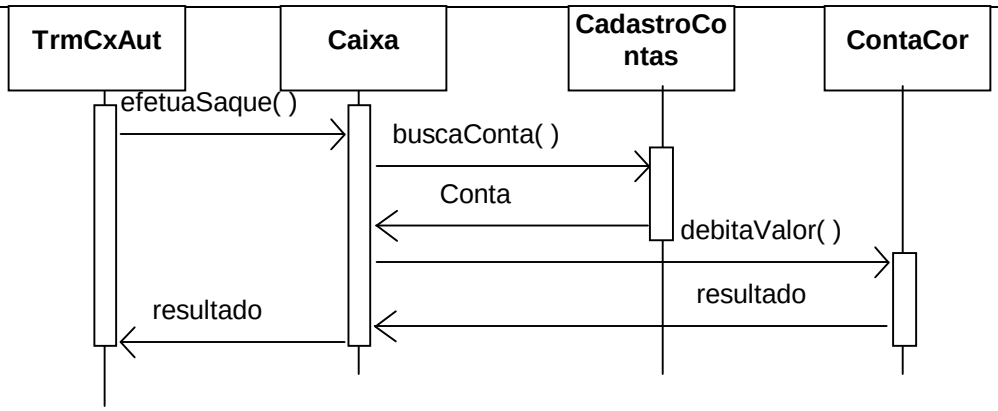

Fig. 6.2(d) – Diagrama de Seqüência para Saque

6.3. Implementação do Sistema

Para cada classe da aplicação iremos criar um arquivo com sua definição. Os arquivos devem ser gravados em formato texto, sem qualquer formatação, com o mesmo nome da classe e extensão `.java` .

6.3.1. Definição da Classe `Caixa`

Com as informações de projeto que definem a interface da classe, podemos iniciar a codificação com o seguinte esqueleto:

```
class Caixa {
   // aqui entram as definições de variáveis que armazenam os atributos da
   // classe
   float consultaSaldo (int num, int pwd){
   // aqui entra a definição do método consultaSaldo
   }
   boolean efetuaSaque (int num, int pwd, float val){
   // aqui entra a definição do método efetuaSaque
   }
   void recarrega(int pwd ){
   // aqui entra a definição do método recarrega
   }
}
```

O estado de um caixa pode ser definido através de três atributos: a que caixa automático está ligado, qual o banco de dados a ser utilizado e o saldo atual em caixa. O primeiro atributo pode ser representado por uma referência ao objeto da classe `TrmCxAut` que representa o caixa automático no sistema. Para isso definir uma variável cujo tipo será a classe `TrmCxAut`, conforme abaixo:

```
private TrmCxAut meuCaixaAut;  // caixa automático ao qual
                               //    está ligado
```

O banco de dados a ser utilizado é um objeto da classe `CadastroContas`, definido como:

```
private CadastroContas dbContas;  // Banco de dados das contas
```

O saldo em caixa poderia ser expresso pela quantidade de notas de R$10,00 ou pelo seu valor total em R$. Optamos por essa última forma, por ser mais genérica. Definiremos,

portanto, uma variável do tipo `float` que chamaremos de `saldoCaixa` e que inicia com zero, conforme abaixo:

```
private float saldoCaixa=0; // saldo no caixa, em R$
```

O método `recarrega()`

Esse método, que é o mais simples, implementa a operação de recarga do caixa, realizando as seguinte ações: elevar o saldo em caixa para R$5.000,00 e mudar o modo de operação do caixa automático para "modo cliente".

A primeira ação é simplesmente uma atribuição de valor, conforme abaixo:

```
saldoCaixa=50000;  // caixa recarregado
```

O modo de operação, tal como especificado anteriormente, é parte do estado da interface com o usuário, que é responsabilidade do objeto que representa o caixa automático. Para mudar o modo de operação devemos, portanto, enviar uma mensagem para ativar o método `setModo`, do objeto referenciado pela variável `meuCaixaAut`, passando como argumento o inteiro 1, que corresponde ao modo de operação "cliente".

O comando que realiza essa ação é:

```
meuCaixaAut.setModo(1); // muda o modo do caixa
                        // automático para "cliente"
```

O método `consultaSaldo()`

Para realizar uma operação de consulta de saldo, em `consultaSaldo`, é necessária a colaboração do objeto da classe `ContaCor` que representa a conta que está sendo consultada. O primeiro passo, portanto, será obter uma referência para esse objeto, a partir do número da conta recebido através do parâmetro `num`. Conforme especificado no projeto, será utilizado o método `buscaConta` do banco de dados.

Devemos definir, portanto, uma variável auxiliar `cta`, de tipo `ContaCor`, que receberá o resultado do método `buscaConta`, passando como argumento o valor do parâmetro `num`, conforme abaixo:

```
ContaCor cta;
cta=dbContas.buscaConta(num); // obtém referencia para o
                      // objeto que representa a conta 'num'
```

Conforme especificado no projeto, caso o número informado não seja um número de conta válido, é retornada uma referência nula (valor `null`). Nesse caso o método `consultaSaldo` deve retornar o valor -1. Caso contrário devemos simplesmente repassar a consulta para o objeto que representa a conta e retornar o resultado obtido.

Podemos, portanto, concluir esse método com:

```
if (cta==null)   // se número de conta inválido ...
    return (-1); // ... retorna -1
else             // caso contrário ...
    return (cta.getSaldo(pwd)); // efetua consulta
```

O método `efetuaSaque()`

A operação de saque possui alguma similaridade com a consulta de saldo, necessitando também de uma referência para o objeto que representa a conta no sistema. Para isso definiremos uma variável auxiliar, como no método `consultaSaldo`:

```
ContaCor cta;
```

Antes de buscarmos a conta, porém, podemos verificar se o valor do saque está de acordo com os critérios exigidos pelo sistema: um valor positivo, múltiplo de R$10,00 e não superior a R$200,00. Podemos também, por segurança, verificar se não é superior ao saldo em caixa. Se alguma dessas condições for violada o método deve retornar o valor `false`.

O código abaixo efetua essa verificação:

```
if (val<0 || (val%10)!=0 || val>200 || val>saldoCaixa)
    return (false);
```

Podemos agora incluir o código para buscar a conta, retornando o valor `false` caso não seja encontrada:

```
cta=dbContas.buscaConta(num);  // obtém a referencia para
                   // o objeto que representa a conta 'num'
if (cta==null)  // se número de conta inválido ...
    return(false);  // ... retorna false
```

Caso exista a conta podemos repassar o pedido de saque para o objeto que a representa. Se o saque não for aceito, encerramos a operação. Se for aceito, fornecemos uma quantidade de notas correspondente ao valor do saque.

O código seguinte implementa essas ações:

```
if (cta.debitaValor("Saque Automatico", val, pwd)==false)
                    // se saque recusado ...
   return (false);  // retorna false
liberaNota((int)(val/10));   // libera pagamento
```

Restam ainda duas ações importantes: atualizar o saldo do caixa, abatendo o valor pago, e efetuar a mudança para o modo supervisor caso o caixa tenha ficado abaixo do mínimo exigido. Feito isso devemos retornar o valor `true`, indicando que a operação foi concluída com êxito.

O código seguinte complementa a definição do método:

```
saldoCaixa=saldoCaixa-val;
if (saldoCaixa<200) meuCaixaAut.setModo(0);
return (true);
```

O método `liberaNota()`

Para que possamos testar o aplicativo vamos implementar o método `liberaNota` com o seguinte código:

```
private void liberaNota (int qtd) {
  while (qtd-->0)
     System.out.println("===/ R$10,00 /===>");
}
```

O construtor da classe

Para concluir a definição da classe Caixa, resta uma questão a resolver: como preencher as variáveis `meuCaixaAut` e `dbContas` com as referências do caixa automático e do banco de dados ? Essas referências só são conhecidas após a criação dos objetos, em tempo de execução. Por outro lado, a partir do momento em que um objeto da classe Caixa seja criado, é importante que essas variáveis estejam preenchidos corretamente, já que são utilizados em diversos métodos que podem ser ativados a qualquer instante.

A solução para isso é a definição de um construtor para os objetos da classe que atribua a essas variáveis as referências necessárias, como a seguir:

```
Caixa (TrmCxAut cxaut, CadastroContas db) { // método construtor
  meuCaixaAut=cxaut;  //guarda referencia do caixa
                      // automático
  dbContas=db; // guarda referencia do banco de dados
}
```

Compilando a classe `Caixa`

Após criado o arquivo `Caixa.java` contendo toda a definição da classe podemos compila-la usando o comando:

```
javac Caixa.java
```

Como a classe `Caixa` contém referências às classes `TrmCxAut`, `CadastroContas` e `ContaCor`, que ainda não estão definidas, o compilador irá enviar diversas mensagens de erro como as seguintes:

```
.\Caixa.java:5: Class ContaCor not found
.\Caixa.java:43: Undefined variable: dbContas
```

Essas mensagens iniciam com o nome do arquivo que está sendo compilado e o número da linha que gerou o erro. Mensagens como essas podem, por enquanto, ser ignoradas, verificando-se apenas se não existem outros tipos de erro a serem corrigidos, provenientes de alguma falha na digitação.

6.3.2. Definição da Classe `TrmCxAut`

Com as informações de projeto que definem a interface da classe, podemos iniciar a codificação com o seguinte esqueleto:

```
class TrmCxAut {
  // aqui entram as definições de variáveis que armazenam
  // os atributos da classe
  void iniciaOperacao () {
  // aqui entra a definição do método iniciaOperacao
}
  void setModo (int modo) {
  // aqui entra a definição do método setModo
  }
}
```

O estado de um objeto dessa classe pode ser descrito através de dois atributos: o seu caixa e o modo de operação atual. Para isso definiremos uma variável de nome `meuCaixa`, que irá conter uma referência para um objeto da classe Caixa, e outra de nome `modoAtual`, do tipo inteiro, para representar o modo de operação, podendo ser 0, para modo supervisor, ou 1, para modo cliente, conforme abaixo:

```
private Caixa meuCaixa; // caixa que processa as
                        //transações
private int modoAtual;  // modo de operação atual
                        // 0=supervisor, 1=cliente
```

O método `setModo()`

A única ação executada por esse método é a atribuição de um novo valor à variável `modoAtual`. O código seguinte implementa essa ação de forma a garantir a consistência do atributo, rejeitando atribuições inválidas.

```
if (modo==0 || modo==1) modoAtual=modo;
```

O método `iniciaOperacao()`

Nesse método deve ser executado um ciclo de diversas operações, que podem ser consultas de saldo, saques e recargas do caixa. No modo cliente só podem ser executadas operações dos dois primeiros tipos enquanto no modo supervisor somente operações de recarga. Iremos incluir uma opção para encerrar um ciclo de operações que poderá ser selecionada a qualquer momento, apenas para facilitar os testes da aplicação.

Podemos definir a seguinte estrutura geral, que utiliza um método auxiliar `getOp`, a ser definido mais adiante, para obter o código da operação solicitada pelo usuário, que poderá ser: 1 para consulta de saldo, 2 para saque, 3 para recarga do caixa e 4 para encerrar o ciclo de operações.

```
int op;// código da operação solicitada
op=getOp();
while (op!=4) {
  switch (op) {
    case 1:
    // aqui entra a implementação da consulta de saldo
    break;
    case 2:
    // aqui entra a implementação do pedido de saque
    break;
    case 3:
    // aqui entra a implementação da recarga do caixa
    break;
  }
  op=getOp();
}
```

Conforme especificado no projeto, uma consulta de saldo é feita através do método `consultaSaldo`, do objeto `meuCaixa`, fornecendo-se o número da conta e a senha, ambos números inteiros. Um resultado igual a -1 indica que os dados fornecidos não são válidos. Em qualquer caso deve ser enviada uma resposta para o usuário com o valor do saldo ou uma mensagem de erro.

O código seguinte implementa essa ação utilizando outro método auxiliar, chamado `getInt`, para obter um número inteiro a ser fornecido pelo usuário, tendo como parâmetro a descrição do dado solicitado:

```
float saldo=meuCaixa.consultaSaldo
    (getInt("número da conta"), getInt("senha"));
if (saldo==-1)  // testa se consulta foi rejeitada
    System.out.println("conta/senha inválida");
else System.out.println("Saldo atual: "+saldo);
```

Um pedido de saque pode ser implementado de forma semelhante, através do método `efetuaSaque`, fornecendo-se, além do número da conta e senha, o valor solicitado pelo cliente:

```
boolean b=meuCaixa.efetuaSaque
  (getInt("número da conta"), getInt("senha"),
           getInt("valor"));
if (b)   // testa se saque foi aceito
    System.out.println("Pode retirar o dinheiro");
else System.out.println("Pedido de saque recusado");
```

Uma operação de recarga é ainda mais simples:

```
meuCaixa.recarrega(getInt("senha"));
```

Iremos definir agora os métodos auxiliares `getOp` e `getInt`.

O método `getOp()`

Esse método deve obter o código da operação solicitada pelo usuário, garantindo que seja compatível com o modo de operação atual. O código seguinte implementa essa ação, utilizando o método `getInt` para obter a opção do usuário:

```
private int getOp() {
  int op;
  do {
    if (modoAtual==1) {  // modo cliente
      op=getInt
        ("opcao: 1 = consulta saldo, 2 = saque, 4=sai");
        if (op!=1 && op!=2 && op!=4) op=0;
    }else {        // modo supervisor
        op=getInt
          ("opcao: 3 = recarrega, 4=sai");
        if (op!=3 && op!=4) op=0;
    }
  } while (op==0);
  return(op);
}
```

O método `getInt()`

A única ação desse método é obter um valor inteiro fornecido pelo usuário, após enviar uma mensagem solicitando o dado.

Para ler valores do teclado utilizando as classes das bibliotecas de Java precisamos definir mais duas variáveis que serão denominados `r` e `st`. A primeira é uma referência para um objeto de tipo `Reader`, associado à entrada padrão `System.in`, definida através do comando:

```
Reader r=new BufferedReader
        (new InputStreamReader (System.in));
```

A segunda variável é uma referência para um objeto da classe `StreamTokenizer` que faz o reconhecimento dos valores digitados pelo usuário, e é definida por:

```
StreamTokenizer st=new StreamTokenizer(r);
```

Feitas essas definições podemos agora codificar o método `getInt` da seguinte forma:

```
private int getInt(String str) {
  System.out.println("Entre com "+str);
  try {st.nextToken();}
  catch (IOException e) {
```

```
    System.out.println("Erro na leitura do teclado");
    return(0);
  }
  return((int)st.nval);
}
```

Como estão sendo utilizadas as classes `Reader` e `StreamTokenizer`, é necessário acrescentar, no início do arquivo e antes do comando `class`, a seguinte declaração, que especifica onde se encontram as definições dessas classes:

```
import java.io.*;
```

O construtor da classe

Conforme o projeto das classes, um objeto da classe `TrmCxAut` contém um objeto da classe `Caixa`. O construtor da classe `TrmCxAut` é, portanto, o método mais indicado para criar esse objeto. Como visto anteriormente, utilizaremos a variável `meuCaixa` para armazenar sua referência.

O código seguinte define um construtor para isso:

```
TrmCxAut (CadastroContas db) {
  meuCaixa=new Caixa(this, db);
}
```

Compilando a classe `TrmCxAut`

Após criado o arquivo `TrmCxAut.java` contendo toda a definição da classe podemos compilá-lo usando o comando:

```
javac TrmCxAut.java
```

Como a classe `TrmCxAut` contém referências para a classe `CadastroContas`, que ainda não existe, serão geradas diversas mensagens de erro de compilação, tal como ocorreu ao compilarmos a classe `Caixa`.

Observe também que, como a classe `TrmCxAut` contém referências à classe `Caixa`, que ainda não havia sido compilada com êxito, esta última será compilada automaticamente, sendo geradas mensagens de erro para cada uma delas, conforme abaixo:

```
.\TrmCxAut.java:62: Class CadastroContas not found
.\Caixa.java:43: Undefined variable: dbContas
```

Note que a primeira mensagem refere-se ao arquivo `TrmCxAut.java` e a segunda ao arquivo `Caixa.java`.

6.3.3. Definição da Classe `CadastroContas`

Nesse exemplo iremos simular a existência de um banco de dados que será implementado pela classe `CadastroContas`. Para este exemplo, essa classe define apenas um método, `buscaConta`, responsável por procurar um objeto do tipo `ContaCor`. Com o fim de facilitar a realização de testes, três contas são criadas pelo construtor da classe `CadastroContas` quando está é instanciada.

O código seguinte define essa classe:

```
class CadastroContas {
  private ContaCor c[];  // vetor de contas
  CadastroContas () {  // método construtor
    c=new ContaCor[4];
```

```
    c[1]=new ContaCor("Ursula",500,1,1);
    System.out.println
      ("Criada conta 1 senha 1 com 500,00");
    c[2]=new ContaCor("Mia",500,2,2);
    System.out.println
      ("Criada conta 2 senha 2 com 500,00");
    c[3]=new ContaCor("Alfredo",500,3,3);
    System.out.println
      ("Criada conta 3 senha 3 com 500,00");
  }
  ContaCor buscaConta (int numcta) {
    if (numcta<1 || numcta>3)  // apenas 3 contas no BD
      return(null);
    else
      return(c[numcta]);
  }
}
```

Compilando a classe `CadastroContas`

Uma vez que tenha sido criado o arquivo `CadastroContas.java` contendo a definição da classe, podemos compilá-la usando o comando:

```
javac CadastroContas.java
```

Não deverá haver erro na compilação. Caso haja alguma mensagem de erro confira o conteúdo do arquivo com o código apresentado ao longo do texto e compile-o novamente após efetuar as correções necessárias.

Devemos agora compilar novamente as classes `Caixa` e `TrmCxAut`, para verificar se os erros gerados anteriormente são todos eliminados com a definição da classe `CadastroContas`. Como as duas classes contém referências mútuas, basta compilarmos um dos arquivos `Caixa.java` ou `TrmCxAtu.java`, pois o outro será compilado automaticamente.

6.4. Teste da aplicação

Para testar a aplicação desenvolvida vamos utilizar um pequeno programa que cria o banco de dados e inicia a operação do terminal de caixa:

Crie um arquivo de nome `Cap6.java` contendo:

```
class Cap6 {
    static public void main (String argv[]) {
        (new TrmCxAut(new CadastroContas())).iniciaOperacao();
    }
}
```

Em seguida compile o programa usando o comando:

```
javac Cap6.java
```

Não havendo erros, a aplicação pode ser executada com:

```
java Cap6
```

A seguir encontram-se os resultado obtidos numa sessão de teste:

```
Criada conta 1 senha 1 com 500,00
Criada conta 2 senha 2 com 500,00
Criada conta 3 senha 3 com 500,00
Entre com opcao: 3 = recarrega, 4=sai
3
Entre com senha
123
Entre com opcao: 1 = consulta saldo, 2 = saque, 4=sai
1
Entre com numero da conta
1
Entre com senha
1
Saldo atual: 500.0
Entre com opcao: 1 = consulta saldo, 2 = saque, 4=sai
2
Entre com número da conta
1
Entre com senha
1
Entre com valor
50
===/ R$10,00 /===>
===/ R$10,00 /===>
===/ R$10,00 /===>
===/ R$10,00 /===>
===/ R$10,00 /===>
Pode retirar o dinheiro
Entre com opcao: 1 = consulta saldo, 2 = saque, 4=sai
1
Entre com numero da conta
1
Entre com senha
1
Saldo atual: 450.0
Entre com opcao: 1 = consulta saldo, 2 = saque, 4=sai
4
```

6.5. Exercícios de Fixação