Universidade do Porto
Faculdade de Engenharia
FEUP

MESTRADO EM REDES E SISTEMAS DE COMUNICAÇÕES

PROGRAMAÇÃO EM COMUNICAÇÕES

Resolução do Exame de 23 de Novembro de 2001

I. Conceitos de Orientação por Objectos

Para cada uma das seguintes perguntas classifique quanto ao seu valor lógico (Verdadeiro e Falso) as afirmações das suas alíneas.

I.1. Os princípios fundamentais do modelo objecto são: abstracção, encapsulamento, modularidade, hierarquia, tipos, concorrência e persistência. O encapsulamento:

a. salienta as características dum objecto consideradas essenciais e que o distinguem de todos os outros tipos de objectos, fornecendo, portanto, contornos muito bem definidos relativamente à perspectiva do utilizador.

FALSO: o texto refere-se à abstracção e não ao encapsulamento.

b. é a forma de ligar uma classe a um objecto, de tal modo que objectos de diferentes tipos não possam ser trocados entre si, ou no máximo, possam ser trocados apenas de forma controlada.

FALSO: o texto refere-se aos tipos e não ao encapsulamento.

c. é o processo de esconder todos os detalhes dum objecto que não contribuem para a definição das suas características essenciais.

VERDADEIRO.

I.2. Um objecto possui estado, comportamento e identidade. O estado de um objecto:

a. é a propriedade que o permite distinguir de todos os outros.

FALSO: o texto refere-se à identidade e não ao estado.

b. é a forma como ele age e reage, em termos do seu comportamento e em termos de passagem de mensagens.

FALSO: o texto refere-se ao comportamento e não ao estado.

c. engloba todas as propriedades do objecto, mais os valores correntes de cada uma dessas propriedades.

VERDADEIRO.

I.3. Uma linguagem de programação orientada aos objectos deve possuir características que permitam usar técnicas de programação orientada aos objectos.

a. Na programação orientada aos objectos os sistemas são modularizados com base nas funções ou métodos.

FALSO: o texto refere-se à programação estruturada.

b. Na programação tradicional os sistemas são modularizados com base nas estruturas de dados.

VERDADEIRO.

c. Uma das técnicas da programação orientada aos objectos consiste em definir uma classe como extensão ou restrição de outra.

VERDADEIRO.

I.4. Considerando os cinco atributos de um sistema complexo:

a. Os sistemas hierárquicos são normalmente compostos por poucos tipos de subsistemas diferentes, mas dispostos em combinações várias.

VERDADEIRO.

b. A complexidade toma a forma de uma hierarquia de vários subsistemas interrelacionados, que por sua vez têm os seus próprios subsistemas, e assim sucessivamente, até aos componentes elementares.

VERDADEIRO.

c. A escolha dos componentes primitivos dum sistema não é arbitrária, e é independente do observador do sistema.

FALSO: a escolha dos componentes primitivos é de facto arbitrária.

I.5. Tendo em consideração o modelo objecto:

a. O modelo objecto orienta a construção de sistemas no sentido de torná-los muito complexos.

FALSO: os sistemas resultantes são menos complexos.

b. O modelo objecto produz sistemas sobre formas intermédias estáveis, logo mais flexíveis e reusáveis.

VERDADEIRO.

c. O modelo objecto reduz o risco de desenvolvimento de sistemas complexos.

VERDADEIRO.

I.6. Tendo em consideração as noções de classes e de objectos:

a. Cada classe é uma instância de algum objecto e cada instância tem zero ou mais classes.

FALSO: os termos classe e objecto estão trocados.

b. A estrutura e comportamento de objectos similares são definidos na classe comum.

VERDADEIRO.

c. Do vocabulário do domínio do problema obtêm-se as abstracções chave que por sua vez originam as classes e objectos.

VERDADEIRO.

II. Modelos UML

A empresa **YAO** (YET ANOTHER OPERATOR) pretende desenvolver um sistema de software denominado **BESTCALL** para disponibilizar aos seus clientes sob a forma de um serviço web, um serviço telefónico ou de uma aplicação Java cliente.

O sistema tem como objectivo indicar ao utilizador qual o operador de telecomunicações que cobra o menor valor para estabelecimento de uma ligação telefónica entre um número de origem e outro de destino, para uma determinada duração prevista. Adicionalmente, o sistema pode também estabelecer directamente a ligação através de um número verde da **YAO**.

Após uma análise do sistema a desenvolver, enumeraram-se os seguintes requisitos:

- A utilização do sistema de software **BESTCALL** inicia-se quando o utilizador introduz um número de origem, um número de destino (ambos com 9 digitos), e uma duração prevista (em segundos). Com base nestes dados de entrada e em informação disponibilizada por um sistema externo **PRICEEXPERT** (responsável pela gestão e actualização da informação sobre os tarifários praticados no mercado), o sistema calcula a melhor opção em termos de operador a utilizar e indica o resultado ao utilizador através de uma sequência de digitos a introduzir no telefone.
- O sistema destina-se a ser utilizado como um servidor e deve suportar diferentes tipos de clientes, que incluem uma aplicação Java cliente (**BESTCALLSOCKETCLIENT**), um cliente HTTP (**BESTCALLHTTPCLIENT**), e um cliente telefónico (**BESTCALLTONECLIENT**), às quais correspondem diferentes interfaces, respectivamente: interface gráfica interface textual e modelação em frequência
- A funcionalidade do sistema **BESTCALL** resume-se a suportar a interface com o utilizador nas suas três variantes e a efectuar um cálculo para encontrar o menor custo cobrado pelas operadoras, tendo como base informação disponibilizada pelo sistema **PRICEEXPERT**.
- O sistema **PRICEEXPERT** mantém actualizada informação sobre os diferentes operadores alternativos no mercado, os produtos que oferecem, as condições de utilização e os respectivos tarifários.
- Atendendo aos diferentes critérios encontrados para a tarifação das chamadas telefónicas, concluiu-se que os dados minimos a armazenar deveriam ser: tipo de cliente, operador origem, operador destino, distância, duração, tipo de arredondamento da duração (ao segundo ou ao minuto, por exemplo), desconto absoluto, desconto percentual. Assume-se que os dados de facturação acima mencionados são calculados *offline* com base na informação disponibilizada pelos operadores.
- O sistema **BESTCALL** é responsável ainda pela autenticação de utilizadores e gestão de sessões de utilização. Embora por defeito, o sistema não obrigue a autenticação de utilizadores, e cada sessão de utilização corresponder apenas a um pedido, o acesso a serviços adicionais poderá requerer autenticação e gestão de sessões de utilização do sistema.

II.1. Diagrama de Casos de Utilização

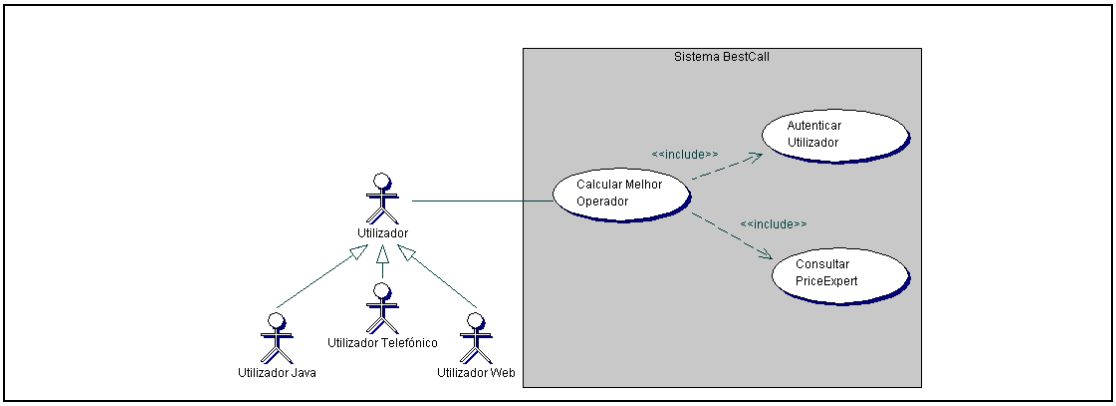
a. Identifique os actores intervenientes no sistema descrito.

Podem ser considerados actores intervenientes no sistema, os diversos tipos de utilizador (via web, via telefone, via aplicação Java) e o sistema externo **PRICEEXPERT**.

b. Enumere os principais casos de utilização do sistema.

Os casos de utilização primários são simplesmente o "Calcular melhor operador" e o "Consultar PriceExpert".

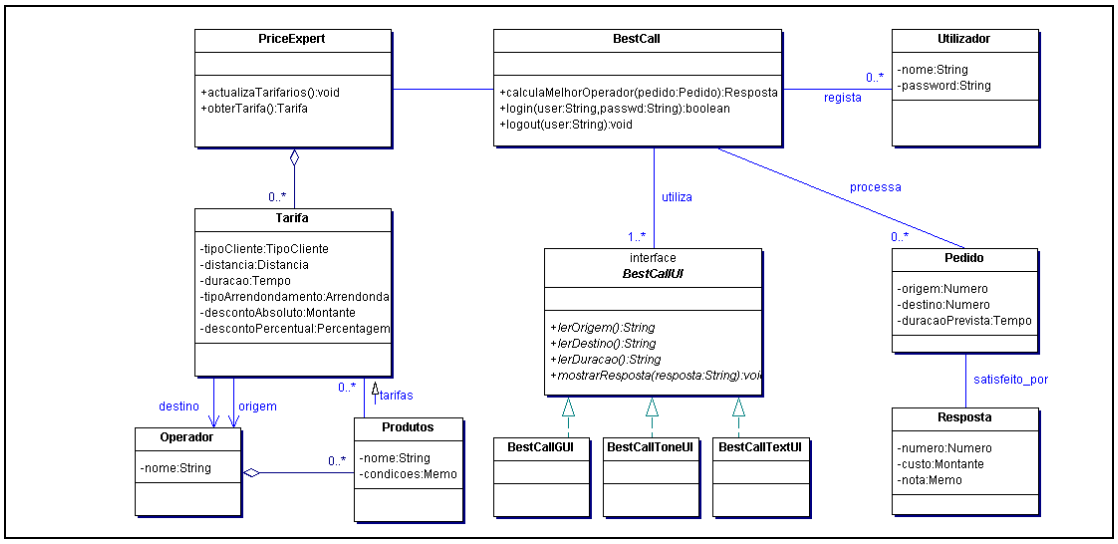
c. Desenhe o diagrama de casos de utilização para o sistema, representando apenas os actores que iniciam casos de utilização.



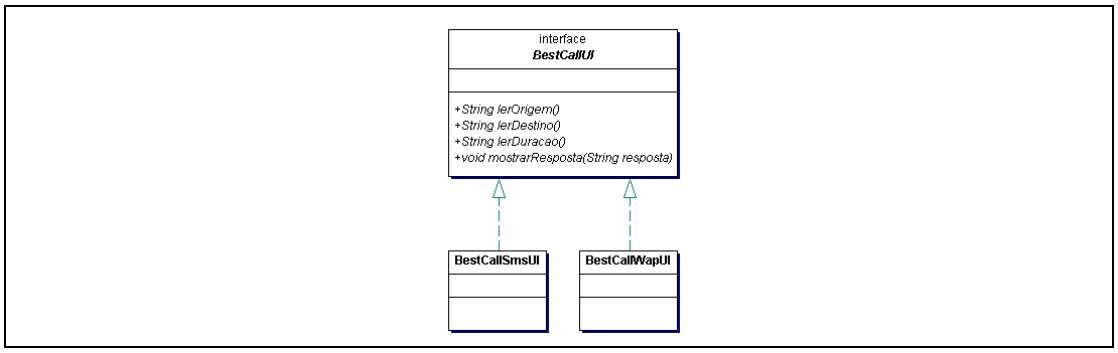
II.2. Diagrama de Classes

a. Elabore um diagrama de classes em UML para o sistema acima enunciado, criando atributos/métodos e completando as associações encontradas com a informação que achar conveniente.

Nota: sugere-se que elabore o diagrama começando por identificar as classes, os seus atributos e métodos, as associações (nomes e cardinalidades) entre classes, e depois as relações de herança.

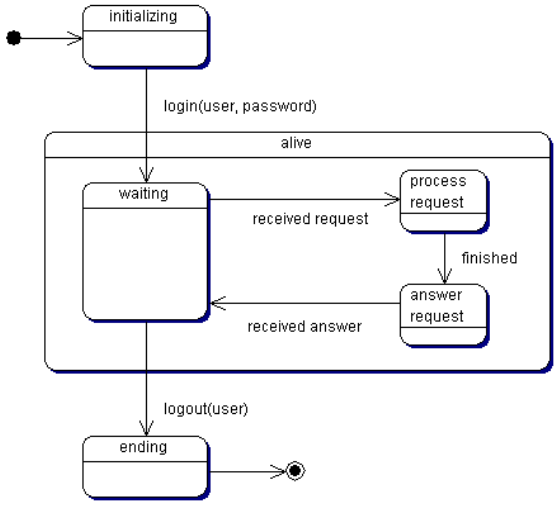


b. Supondo que durante a vida do sistema, surgiram dois novos tipos de interfaces a suportar (SMS e WAP), reveja o diagrama anterior e apresente num diagrama parcial as modificações que entendeu serem necessárias.



II.3. Diagrama de Estados

Considere o diagrama abaixo apresentado que descreve os vários estados possíveis de um objecto da classe **SESSION**.



a. Descreva de forma sucinta o diagrama acima e indique em que estado o objecto ficará depois de receber a sequência de eventos (login, received request, finished, received answer, received request, finished), sabendo que o estado inicial é o initializing.

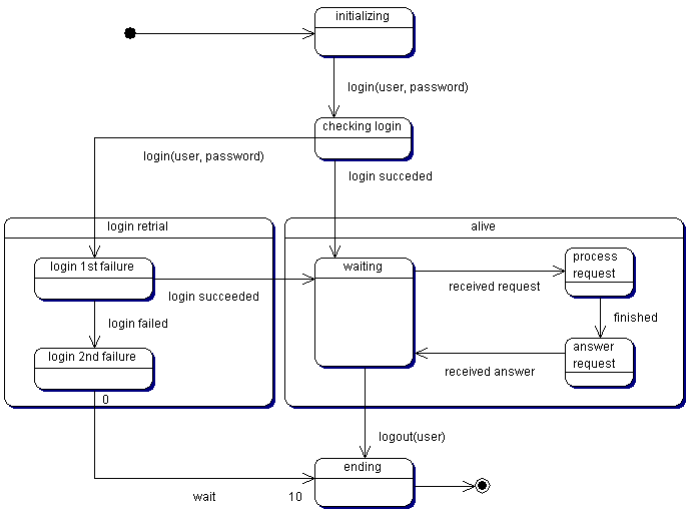
b. Que alterações sugere efectuar ao diagrama anterior para representar a possibilidade de o login não suceder (utilizador/password incorrecto) e apenas existir uma segunda possibilidade de login. Por motivos de segurança, se a segunda tentativa também falhar, o sistema aguarda 10 segundos antes de terminar a sessão.

Os objectos da classe **SESSION** podem estar em três estados principais:

- initializing, que é o estado inicial do objecto, e do qual transitam para o sub-estado waiting do estado alive após o evento login;
- alive, que é por sua vez composto pelos três sub-estados waiting, process request e answer request; deste estado o objecto transita para o estado ending após o evento logout; a transições internas a este super-estado são para o estado process request após a recepção do evento received request, deste para o estado process request após a recepção do evento finished, e finalmente deste para answer request quando o evento received answer é recebido;
- ending, é o estado final.

Após a sequência de eventos dada, o objecto encontrar-se-á no estado answer request.

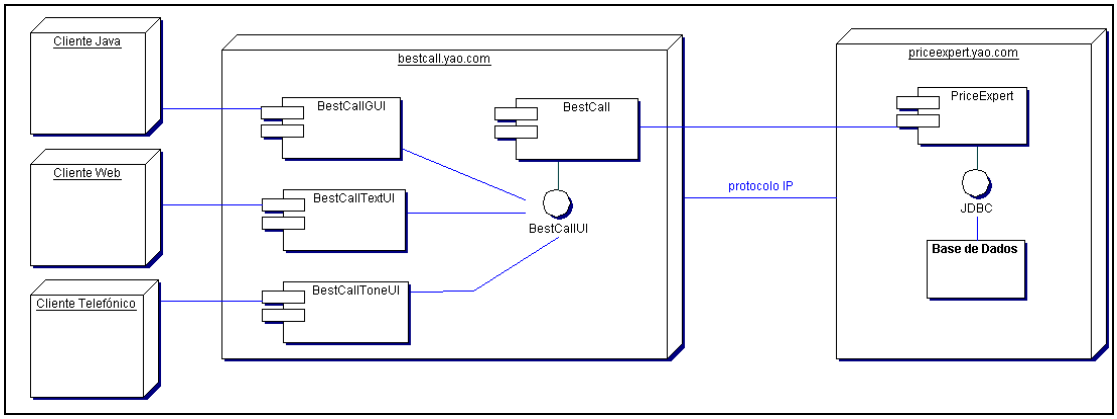
As alterações a efectuar por forma a suportar os novos requisitos em caso de falha de login, poderão ser as apresentadas abaixo no diagrama.



II.4. Diagrama de Distribuição (*Deployment*)

Os sistemas **BESTCALL** e **PRICEEXPERT** são utilizados em modo cliente/servidor. Os clientes utilizam uma interface que comunica com o servidor remoto em **bestcall.yao.com**. O sistema **PRICEEXPERT** residente na máquina **priceexpert.yao.com**, comunica com o sistema **BESTCALL** por ligação IP e acede a uma base de dados relacional nela armazenada através de **JDBC**.

Desenhe um diagrama de distribuição que inclua os sistemas **BESTCALL** e **PRICEEXPERT** e que represente a arquitectura anteriormente descrita (3 tipos de clientes e servidores) e a configuração acima descrita, completando com os detalhes que considerar convenientes.



III. Programação em Java

III.1. Linguagem Java

a. Implemente em linguagem Java uma declaração possível para os atributos e métodos das classes **BESTCALL**, **BESTCALLTEXTUI**, **BESTCALLTONEUI** e **PRICEEXPERT**.

```
public class BestCall {
    public Resposta calculaMelhorOperador(Pedido pedido) {    }

    public boolean login(String user, String passwd) {    }

    public void logout(String user) {    }

    private PriceExpert priceExpertSystem;
    private Vector utilizadores;

    private BestCallUI textUI;
    private BestCallUI toneUI;
    private BestCallUI graphicalUI;

    private Vector pedidos;
}

public class BestCallTextUI implements BestCallUI { }

public class BestCallToneUI implements BestCallUI { }

public class PriceExpert {
    public void actualizaTarifarios() { }

    public Tarifa obterTarifa() { }

    private Vector tarifas;
}
```

b. Apresente uma implementação para o diagrama de estados da classe **SESSION** anteriormente apresentado.

```
public class Session {

    private int state;

    // possible states
    final private int INITIALIZING = 0;
    final private int WAITING = 1;
    final private int PROCESS_REQUEST = 2;
    final private int ANSWER_REQUEST = 4;
    final private int ENDING = 8;

    final private int UNKNOWN_STATE = -1;

    // possible events
    final private int LOGIN = 16;
    final private int RECEIVED_REQUEST = 32;
    final private int FINISHED = 64;
    final private int RECEIVED_ANSWER = 128;
    final private int LOGOUT = 256;

    public Session() {
        setState(INITIALIZING);
        // initializing ...
    }

  public void receiveEvent(int event, Object eventMessage) {
        int finalState = getTransition(this.state, event);
        if (finalState != UNKNOWN_STATE) {
            processEvent(event, eventMessage);
            setState(finalState);
        }
    }

    public int getTransition(int fromState, int event) {
        switch(fromState) {
              case INITIALIZING:
                 if (event == LOGIN) { return WAITING; }
              case WAITING:
                 if (event == RECEIVED_REQUEST) { return PROCESS_REQUEST; }
                 if (event == LOGOUT) { return ENDING; }
              case PROCESS_REQUEST:
                 if (event == FINISHED) { return ANSWER_REQUEST; }
              case ANSWER_REQUEST:
                 if (event == RECEIVED_ANSWER) { return WAITING; }
              case ENDING:
                    break;
              default:
                    break;
        }
        return UNKNOWN_STATE;
    }

    public void processEvent(int event, Object eventMessage) {
        switch(event) {
              case LOGIN:
                    // processing login ...
                 break;
              case RECEIVED_REQUEST:
                // process request...
                 break;
              case FINISHED:
                    // answer request...
                 break;
              case RECEIVED_ANSWER:
                    // sending answer...
                 break;
              case LOGOUT:
                    // terminating...
                 break;
              default:
                    break;
        }
    }

    private void setState(int s) {
        this.state = s;
    }
}
```

III.2. Package java.net

a. Apresente uma implementação possível para a classe **BESTCALLSOCKECTSERVER**, considerando que existem já implementados os métodos abaixo mencionados e que a comunicação entre clientes e servidor é realizada através do porto 1234.

i. **Strng readLine (InputStream in)**, que lê de **in** uma linha de argumentos e devolve-a em formato String;

ii. **Vector scanArguments(String argumentsLine)**, que processa os argumentos e devolve um vector com os argumentos a passar ao método de cálculo seguinte;

iii. **Result calculate(Vector arguments)**, que calcula um resultado com base nos argumentos passados em formato a enviar ao cliente através do método de escrita de resultados abaixo;

iv. **void writeResult(Result result, OutputStream out)** que permite escrever os resultados do cálculo no OutputStream passado;

```
public class BestCallSocketServer {
    public Result calculate(Arguments args) {…}
    public void writeResult(Result result, OutputStream out) {…}
    public String readLine(InputStream in) {…}
    public Arguments scanArguments(String s) {…}

    public static void main(String[] argv) {
        ServerSocket s;
        try {
            s = new ServerSocket(1234, 10);
        }catch (IOException e) {
            System.err.println("Unable to create socket");
            e.printStackTrace();
            return;
        }
        try {
                new BestCallSocketServer(s.accept());
        }catch (IOException e) {
        }
    }

    private Socket socket;

    private BestCall bestCallSystem;

    BestCallSocketServer(Socket s) {
        socket = s;
        execute();
    }

    public void execute() {
        InputStream in;
        PrintStream out = null;
        Result result;
        String argumentsLine;
        Arguments arguments;

        try {
            in = socket.getInputStream();
            out = new PrintStream(socket.getOutputStream());
            argumentsLine = readLine(in);
            arguments = scanArguments(argumentsLine);
            result = calculate(arguments);
            writeResult(result, out);
        }catch (IOException e) {
            if (out != null) out.print("Error\n");
            out.close();
            try {
                socket.close();
            }catch (IOException ie) {}
            return;
        }
        out.print("Good:\n");
        try {
            in.close();
            out.close();
            socket.close();
        }catch (IOException e) { }
    }
}
```

b. Atendendo a que a operação **calculate()** pode consumir demasiado tempo, sugira modificações ao programa para suportar uma utilização em diversos threads.

```
public class BestCallSocketServer extends Thread {
    public Result calculate(Arguments args) {…}
    public void writeResult(Result result, OutputStream out) {…}
    public String readLine(InputStream in) {…}
    public Arguments scanArguments(String s) {…}

    public static void main(String[] argv) {
        ServerSocket s;
        try {
            s = new ServerSocket(1234, 10);
        }catch (IOException e) {
            System.err.println("Unable to create socket");
            e.printStackTrace();
            return;
        }
        try {
                while(true) new BestCallSocketServer(s.accept());
        }catch (IOException e) {
        }
    }

    private Socket socket;

    private BestCall bestCallSystem;

    BestCallSocketServer(Socket s) {
        socket = s;
        start();
    }

    public void run() {
        InputStream in;
        PrintStream out = null;
        Result result;
        String argumentsLine;
        Arguments arguments;

        try {
            in = socket.getInputStream();
            out = new PrintStream(socket.getOutputStream());
            argumentsLine = readLine(in);
            arguments = scanArguments(argumentsLine);
            result = calculate(arguments);
            writeResult(result, out);
        }catch (IOException e) {
            if (out != null) out.print("Error\n");
           out.close();
            try {
                socket.close();
            }catch (IOException ie) {}
            return;
        }
        out.print("Good:\n");
        try {
            in.close();
            out.close();
            socket.close();
        }catch (IOException e) { }
    }
}
```