Universidade de Aveiro, DETI

Padrões e Desenho de Software

Guião das aulas práticas

LEI – Licenciatura em Engenharia Informática

Ano: 2020/2021

Lab I.

Objetivos

Os objetivos deste trabalho são:

- Rever e aplicar conceitos de programação adquiridos anteriormente: arrays bidimensionais, genéricos, ciclos for-each, tipos enumerados.
- Rever e praticar técnicas de desenvolvimento de software: implementar uma especificação de classe, programa com múltiplos componentes, e ficheiros JAR

I.1 Word Search Solver

O objetivo deste trabalho é escrever um programa em JAVA para resolver *Sopas de Letras*. A entrada do programa é um único ficheiro de texto contendo o puzzle e as palavras a encontrar. Exemplo (poderá pesquisar outros online):

```
STACKJCPAXLF
YLKWUGGTESTL
LNJSUNCUXZPD
ETOFQIKICFNG
SENILMJFUMRK
ZBUUOMSBSKCY
SUMTRASARZIX
RBMWWRJDAXVF
JEJHQGSDRAIB
ACWEZOLMZOLT
VIUQVRAMDGWH
AGFTWPJZWUMH
programming;java;words lines civil
test;stack;
```

A saída é a lista de palavras, bem como a posição em que se encontram no puzzle.

(a) Requisitos de Entrada

O programa deve verificar se:

1. O puzzle é sempre quadrado, com o tamanho máximo de 40x40.
2. As letras do puzzle estão em maiúscula.
3. Na lista, as palavras podem estar só em minúsculas, ou misturadas.
4. As palavras são compostas por caracteres alfabéticos.
5. No puzzle e na lista de palavras, o ficheiro não pode conter linhas vazias.
6. Cada linha pode ter mais do que uma palavra, separadas por vírgula, espaço ou ponto e vírgula.
7. Todas as palavras da lista têm de estar no puzzle e apenas uma vez.
8. A lista de palavras pode conter palavras com partes iguais (por exemplo, pode conter BAG e RUTABAGA). Nestes casos a deteção das palavras mais pequenas não deve ser feita sobre a palavra maior.

(b) Requisitos de Saída

A lista de palavras do puzzle retornadas pelo WSSolver tem de estar na mesma ordem das palavras passadas na lista. As palavras têm de estar em maiúsculas.

(c) Exemplo de Execução

O programa deverá ser testado com vários ficheiros, verificando os requisitos. Abaixo, mostra-se um exemplo de execução com os dados anteriores:

```
$ java WSSolver sdl_01.txt

programming     11      12,6    Up
java            4       9,1     Down
words           5       11,11   UpLeft
lines           5       5,6     Left
civil           5       6,11    Down
test            4       2,8     Right
stack           5       1,1     Right

S T A C K . . . . . .
. . . . . G . T E S T .
. . . . . N . . . . . .
. . . . . I . . . . . .
. S E N I L . . . . . .
. . . . . M . . . . C .
. . . . . A S . . . I .
. . . . . R . D . . V .
J . . . . G . . R . I .
A . . . . O . . . O L .
V . . . . R . . . . W .
A . . . . P . . . . . .
```

Os resultados deverão ser entregues juntamente com o código, até um máximo de 5 (ficheiros *out1.txt, out2.txt,* ...).

I.2 Word Search Generator

Escreva o programa WSGenerator, que crie uma *Sopa de Letras* de acordo com o formato e requisitos anteriores. O programa deve receber como parâmetro de entrada um ficheiro com a lista de palavras, a dimensão da sopa de letras e o nome de um ficheiro para guardar a *Sopa de Letras*.

(a) Exemplo de Execução

Assumindo que o ficheiro "*wordlist_1.txt"* contém a lista de palavras (uma por linha, ou uma lista por linha).

```
$ java WSGenerator -i wordlist_1.txt -s 15
STACKJCPAXLF
YLKWUGGTESTL
LNJSUNCUXZPD
ETOFQIKICFNG
SENILMJFUMRK
ZBUUOMSBSKCY
SUMTRASARZIX
RBMWWRJDAXVF
JEJHQGSDRAIB
ACWEZOLMZOCT
VIUQVRAMDGWH
AGFTWPJZWUMH
programming;java;words lines civic
test;stack;
```

```
$ java WSGenerator –i wordlist_1.txt –s 15 –o sdl_01.txt
```

O resultado é o mesmo do anterior, mas guardado no ficheiro "*sdl_01.txt*".
Junto com o código, deve entregar 3 exemplos de wordlist (*wlist1.txt, wlist2.txt, wlist3.txt*) e respetivos resultados (*sopa1.txt, sopa2.txt, sopa3.txt*).

Nota importante: para cada guião prático, deverá ser usada no *git* uma nomenclatura uniforme (*lab01, lab02, lab03,…*) para permitir uma identificação mais fácil dos projetos.

Bom trabalho!

Lab II.

Objetivos

Os objetivos deste trabalho são:

- Avaliar código desenvolvido por outros colegas fazendo uma revisão crítica, de acordo com os princípios de POO.
- Utilizar o git (codeUA) para consolidar revisões em aplicações com múltiplos componentes.
- Realizar uma avaliação e autoavaliação da aula prática anterior.

II.1 Revisão de código do Lab 1

Para a realização desta tarefa, na pasta *lab02* do repositório, irá encontrar duas pastas distintas, T1 e T2 *(não se esqueça de efetuar um git pull localmente antes de continuar)*.

(a) Revisão

Para cada pasta, correspondente a dois grupos diferentes, deve ser verificado

- A organização do trabalho entregue
- Se o(s) programa(s) compila(m) sem erros
- A qualidade e organização do código
- Se os requisitos foram adequadamente alcançados

Pode/deve modificar o código se o resultado final for melhor que o original. Todas as alterações efetuadas devem ser colocadas no repositório *git* novamente. Verifique se existem diferenças entre as versões.

(b) Avaliação formal

Na pasta principal do repositório *Git* encontra também um ficheiro *'pds_<XXX>.txt'*, que contém o código que servirá para autenticação do seu grupo.

Aceda ao link https://forms.gle/vaBFDCKCjfK1Xtjo9 e preencha este formulário para cada um dos trabalhos (T1 e T2). Note que o objetivo deste exercício é treinar a capacidade de revisão de código, bem como avaliar a qualidade da solução. Assim, é muito importante que seja feita uma avaliação objetiva. Pode optar por fazer a revisão em grupo ou individualmente.

(c) Lab 1, Refactoring

Depois de realizadas as duas tarefas anteriores, deverá estar apto para melhorar o seu/vosso primeiro trabalho (lab1). Se este for o caso, faça o *push* da nova versão no git.

Nota importante: para cada guião prático, deverá ser usada no *git* uma nomenclatura uniforme (*lab01, lab02, lab03,…*) para permitir uma identificação mais fácil dos projetos.

Bom trabalho!

Lab III.

Objetivos

Os objetivos deste trabalho são:

- Aplicar conceitos de modulação de software necessários no desenvolvimento de uma solução
- Rever e consolidar competências de desenvolvimento de software

III.1 Jogo do Galo

Pretende-se desenvolver uma versão simples do "Jogo do Galo". Para tal são fornecidos os seguintes módulos (no dossier das aulas práticas):

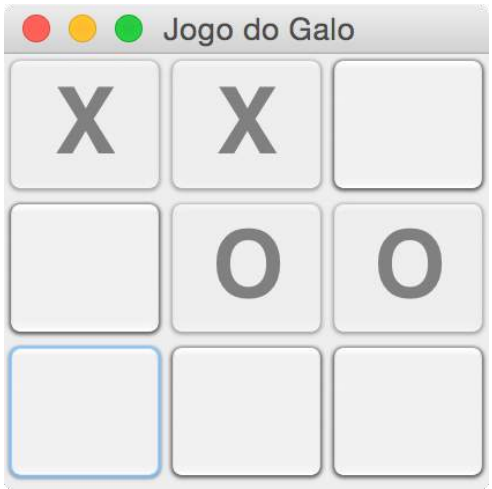
a) A aplicação visual do jogo, classe *JGalo*, desenvolvida sobre java Swing e que não precisa de ser modificada para este trabalho. Apesar disso, recomenda-se a sua análise cuidada.

b) A interface *JGaloInterface* que irá servir de ligação entre a classe *JGalo* e o módulo que terá de desenvolver.

```
public interface JGaloInterface {
    public abstract char getActualPlayer();
    public abstract boolean setJogada(int lin, int col);
    public abstract boolean isFinished();
    public abstract char checkResult();
}
```

Considere que o programa apenas executa o jogo uma vez, começando com cruzes (`X`) ou bolas (`O`) consoante o argumento inicial (por omissão, considere X).

Nota: No dossier existe ainda um ficheiro executável (`JogoDoGalo.jar`) que permite verificar o comportamento desejado para este programa.



III.2 Voos

O objetivo é desenvolver um programa para gerir voos e reservas. Analise os requisitos e planeie cuidadosamente as interfaces, classes, e estruturas de dados mais adequadas.

Requisitos iniciais

1. Um voo é identificado por um código alfanumérico e tem associado um avião com determinada configuração;
2. Cada avião tem lugares para a classe turística e, opcionalmente, lugares para a classe executiva;
3. O número de lugares é especificado, para cada classe, pelo número de filas e número de lugares por fila. Por exemplo '3x2' corresponde a 3 filas com 2 lugares em cada fila;
4. A classe executiva, se existir, ocupa as primeiras filas, começando na fila 1; a numeração das filas da classe turística continua esta numeração;
5. Os bancos em cada fila são identificados por letras, começando na letra A;
6. Uma reserva de lugares indica a classe (**T**uristíca / **E**xecutiva) e o número de passageiros;
7. Caso não existam lugares suficientes para uma reserva, esta não deve ser efetuada;
8. Na atribuição dos lugares para uma reserva deve primeiro procurar-se uma fila vazia (na classe correspondente), atribuindo os lugares de forma sequencial e continuando na fila seguinte caso necessário; caso não haja filas vazias, deve distribuir-se os lugares vagos sequencialmente, começando na primeira fila (da classe correspondente).

Comandos

O programa deve permitir comandos lidos da consola, conforme indicado:

- **H**: apresenta as opções do menu.

- **I *filename***: Lê um ficheiro de texto contento informação sobre um voo. A primeira linha do ficheiro deve começar com o caracter ">" e indicar o código de voo, o número de filas e lugares por fila em classe executiva (caso exista) e o número de filas e lugares por fila em classe turística. As linhas seguintes, caso existam, contêm reservas já efetuadas, no formato classe, número de lugares, como se vê nos exemplos.

Exemplos de ficheiros:

(flight1.txt)

```
>TP1920 3x2 15x3
T 2
T 4
E 3
T 1
E 2
E 2
E 1
```

(flight2.txt)

```
>TP1930 20x4
E 4
T 6
```

Exemplos de execução:

```
Escolha uma opção: (H para ajuda)
I flight1.txt
Código de voo TP1920. Lugares disponíveis: 6 lugares em
classe Executiva; 45 lugares em classe Turística.
Não foi possível obter lugares para a reserva: E 2

Escolha uma opção: (H para ajuda)
I flight2.txt
Código de voo TP1930. Lugares disponíveis: 80 lugares em
classe Turística.
Classe executiva não disponível neste voo.
Não foi possível obter lugares para a reserva: E 4
```

- **M *flight_code***: exibe o mapa das reservas de um voo, conforme mostra o exemplo. Os lugares reservados são identificados pelo número sequencial da reserva; os lugares livres são identificados pelo número 0.

  Exemplo de execução:

```
Escolha uma opção: (H para ajuda)
M TP1920
   1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18
A  3  3  5  1  2  2  4  0  0  0  0  0  0  0  0  0  0  0
B  3  6  5  1  2  0  0  0  0  0  0  0  0  0  0  0  0  0
C           0  2  0  0  0  0  0  0  0  0  0  0  0  0  0
```

- **F *flight_code num_seats_executive num_seats_tourist*:** acrescenta um novo voo, com código, lugares em executiva (p.ex. 4x3, representando 4 filas com 3 lugares por fila), e lugares em turística. Os lugares em classe executiva são opcionais, podendo existir apenas lugares em turística.

- **R *flight_code class number_seats***: acrescenta uma nova reserva a um voo, com indicação do código do voo, da classe (T / E), e do número de lugares. O programa deve verificar se há lugares disponíveis na classe pretendida. Caso a reserva seja efetuada deve ser apresentado no ecrã o código da reserva no formato *flight_code:sequential_reservation_number* e os lugares atribuídos.

  Exemplo de execução:

```
Escolha uma opção: (H para ajuda)
R TP1930 T 3
TP1930:2 = 3A | 3B | 3C
```

- **C *reservation_code***: cancela uma reserva. O código de reserva tem o formato *flight_code:sequential_reservation_number*.

- **Q**: termina o programa.

Deve permitir que o programa possa ser executado usando um ficheiro de comandos como argumento de entrada. Por exemplo, "`java lab3 ficheiro_de_comandos`".
No dossier da disciplina são fornecidos alguns ficheiros de exemplos de voos e reservas.

Nota importante: para cada guião prático, deverá ser usada no *git* uma nomenclatura uniforme (*lab01, lab02, lab03,…*) para permitir uma identificação mais fácil dos projetos.

Bom trabalho!

Lab IV.

Objetivos

Os objetivos deste trabalho são:

- Analisar e rever de forma crítica, de acordo com os princípios de POO, o código desenvolvido por outros colegas nas duas aulas anteriores.
- Elaborar um conjunto de princípios e boas práticas de POO

IV.1 Revisão de código do projeto anterior

Na pasta *lab04* do repositório irá encontrar duas pastas distintas, T1 e T2 *(não se esqueça de efetuar um git pull localmente antes de continuar).*

(a) Revisão

Para cada pasta, correspondente a dois grupos diferentes, deve ser verificado

- A organização do trabalho entregue
- Se o(s) programa(s) compila(m) sem erros
- A qualidade e organização do código
- Quais os requisitos que foram adequadamente alcançados

Pode modificar o código livremente (para executar, testar, anotar, etc.).

(b) Avaliação formal

Aceda ao link https://forms.gle/9Tx2vRXkZ3xdYZFP8 e preencha o formulário para cada um dos trabalhos (T1 e T2). Note que o objetivo deste trabalho é treinar a capacidade de revisão de código bem como avaliar a qualidade da solução. Assim, é muito importante que seja feita uma avaliação honesta e objetiva. Pode optar por fazer a revisão em grupo ou individualmente.

(c) Lab III - Refactoring

Depois de realizadas as duas tarefas anteriores (T1 e T2), poderá igualmente melhorar alguns aspetos do seu próprio trabalho. Se este for o caso, faça o *push* da nova versão no git.

Nota importante: para cada guião prático, deverá ser usada no *git* uma nomenclatura uniforme (*lab01, lab02, lab03,…*) para permitir uma identificação mais fácil dos projetos.

Bom trabalho!

Lab V.

Objetivos

Os objetivos deste trabalho são:

- Identificar e utilizar padrões relacionados com a construção de objetos
- Aplicar boas práticas de programação por padrões em casos práticos

V.1 Serviço de transportes

Pretende-se criar um pequeno programa que escolha o veículo mais adequado para um transporte, de acordo com um conjunto de requisitos. Considere que os veículos obedecem à seguinte interface.

```
public interface Vehicle {
        public int getMaxVolume();
        puclic int getMaxPassangers();
}
```

Deve criar um conjunto de classes que modelem os seguintes tipos de veículos:

Classe	Max passageiros	Volume carga	Outros
Scooter	1	0	
Micro	1	250	
City	3	250	
Family	4	600	
Van	4	1000	Cadeira de rodas

Modele as classes e construa o código necessário para que o cliente possa executar pedidos como os apresentados no método *main* seguinte. A função VehicleFactory.*getVehicle* deverá devolver o veículo mais pequeno que responde às necessidades (argumentos).

```
public static void main(String[] args) {

      int[] luggage;
      Vehicle v;

      // Get vehicle for 1 passenger without luggage
      v = VehicleFactory.getVehicle(1);

      // Get vehicle for 1 passenger with two items of luggage
      luggage = new int[]{100, 140}; // two bags with a total volume of 240
      v = VehicleFactory.getVehicle(1, luggage);

      // Get vehicle for 3 passengers with two items of luggage
      luggage = new int[]{50, 200, 240}; // three bags with a total volume of 490
      v = VehicleFactory.getVehicle(3, luggage);

      // Get vehicle for 2 passengers and wheelchair
```

```
        v = VehicleFactory.getVehicle(2, true);

            // you should add other examples here
    }
```

Output:

```
Vehicle for 1 passengers: Use a Scooter
Vehicle for 1 passengers with 2 items of luggage: Use a Micro car
Vehicle for 3 passengers with 3 items of luggage: Use a Family car
Vehicle for 2 passengers and wheelchair: Use a Van
```

V.2 Pastelaria

Pretende-se criar um conjunto de classes que modele a elaboração de bolos numa pastelaria. Para tal, considere que um bolo é representado pela classe *Cake*. Por omissão, os bolos são circulares, mas podem ser quadrados ou retangulares (não é necessário definir a dimensão), e podem ter um diferente número de camadas, com uma camada intermédia de creme.

```
class Cake {
    private Shape shape;
    private String cakeLayer;
    private int numCakeLayers;
    private Cream midLayerCream;
    private Cream topLayerCream;
    private Topping topping;
    private String message;

        //.. restantes métodos
}
```

Considere ainda que todos os bolos são construídos seguindo um padrão *Builder* que usa a interface *CakeBuilder*.

```
interface CakeBuilder {
    public void setCakeShape(Shape shape);
    public void addCakeLayer();
    public void addCreamLayer();
    public void addTopLayer();
    public void addTopping();
    public void addMessage(String m);
    public void createCake();
    public Cake getCake();
}
```

Modele as classes e construa o código necessário para que o cliente possa executar pedidos como os apresentados no método *main* seguinte. Note que o código necessário para construir cada bolo é sempre o mesmo, apenas variando o *CakeBuilder* passado em *CakeMaster*.

```
public static void main(String[] args) {
    CakeMaster cakeMaster = new CakeMaster();

    CakeBuilder chocolate = new ChocolateCakeBuilder();
    cakeMaster.setCakeBuilder(chocolate);
```

```
        cakeMaster.createCake("Congratulations");              // 1 cake layer
        Cake cake = cakeMaster.getCake();
        System.out.println("Your cake is ready: " + cake);

        CakeBuilder sponge = new SpongeCakeBuilder();
        cakeMaster.setCakeBuilder(sponge);
        cakeMaster.createCake(Shape.Square, 2, "Well done");   // squared, 2 layers
        cake = cakeMaster.getCake();
        System.out.println("Your cake is ready: " + cake);

        CakeBuilder yogurt = new YogurtCakeBuilder();
        cakeMaster.setCakeBuilder(yogurt);
        cakeMaster.createCake(3, "The best");                  // 3 cake layers
        cake = cakeMaster.getCake();
        System.out.println("Your cake is ready: " + cake);

        // you should add here other example(s) of CakeBuilder

    }
```

Output:

```
Your cake is ready: Soft chocolate cake with 1 layers, topped with Whipped_Cream cream
and Fruit. Message says: "Congratulations".

Your cake is ready: Sponge cake with 2 layers and Red_Berries cream, topped with
Whipped_Cream cream and Fruit. Message says: "Well done".

Your cake is ready: Yogurt cake with 3 layers and Vanilla cream, topped with Red_Berries
cream and Chocolate. Message says: "The best".
```

V.3 Construtor com demasiados parâmetros

Considere a classe seguinte. Reescreva-a usando o padrão *builder*.

```
public class Movie {
   private final String title;
   private final int year;
   private final Person director;
   private final Person writer;
   private final String series;
   private final List<Person> cast;
   private final List<Place> locations;
   private final List<String> languages;
   private final List<String> genres;
   private final boolean isTelevision;
   private final boolean isNetflix;
   private final boolean isIndependent;

   public Movie(
      final String movieTitle,
      final int movieYear,
      final Person movieDirector,
      final Person movieWriter,
      final String movieSeries,
      final List<Person> movieCast,
      final List<Place> movieLocations,
      final List<String> movieLanguages,
      final List<String> movieGenres,
      final boolean television,
      final boolean netflix,
      final boolean independent)    {
```

```
                this.title = movieTitle;
                this.year = movieYear;
                this.director = movieDirector;
                this.writer = movieWriter;
                this.series = movieSeries;
                this.cast = movieCast;
                this.locations = movieLocations;
                this.languages = movieLanguages;
                this.genres = movieGenres;
                this.isTelevision = television;
                this.isNetflix = netflix;
                this.isIndependent = independent;
    }
}
```

V.4 Classe Calendar

Analise a implementação da classe *java.util.Calendar* e identifique padrões de construção usados nesta classe. *Nota*: pode consultar este código em
http://www.docjar.com/html/api/java/util/Calendar.java.html
ou em
https://github.com/openjdk-mirror/jdk7u-jdk/blob/master/src/share/classes/java/util/Calendar.java
Reporte as suas observações no ficheiro *lab05/calendar.txt*.

Lab VI.

Objetivos

Os objetivos deste trabalho são:

- Identificar e utilizar padrões relacionados com a construção e estrutura de objetos e classes
- Aplicar boas práticas de programação por padrões em casos práticos

Nota: Para além do código no codeUA, apresente o diagrama de classes da solução final (pode usar o UMLet, por exemplo, ou um plugin Eclipse/Netbeans).

VI.1 Empresa Pst (Petiscos e Sweets)

As empresas *Sweets* e *Petiscos* estão em processo de fusão (Pst) e precisam de integrar os seus registos de pessoal. A empresa *Sweets* usa 2 classes *Employee* e *Database*, enquanto que a empresa *Petiscos* usa *Empregado* e *Registos*.

```
// Sweets
class Employee {
     private String name;
     private long emp_num;
     private double salary;

     public Employee(String name, long emp_num, double salary) {
          this.name = name;
          this.emp_num = emp_num;
          this.salary = salary;
     }
     public String getName() {
          return name;
     }
     public long getEmpNum() {
          return emp_num;
     }
     public double getSalary() {
          return salary;
     }
}

class Database {        // Data elements
     private Vector<Employee> employees; // Stores the employees

     public Database() {
          employees = new Vector<>();
     }
     public boolean addEmployee(Employee employee) {
          // Code to add employee
     }
     public void deleteEmployee(long emp_num) {
          // Code to delete employee
     }
     public Employee[] getAllEmployees() {
          // Code to retrieve collection
     }
}
```

```java
// Petiscos
class Empregado {
    private String nome;
    private String apelido;
    private int codigo;
    private double salario;

    public Empregado(String nome, String apelido, int codigo, double salario) {
        this.nome = nome;
        this.apelido = apelido;
        this.codigo = codigo;
        this.salario = salario;
    }
    public String apelido() {
        return apelido;
    }
    public String nome() {
        return nome;
    }
    public int codigo() {
        return codigo;
    }
    public double salario() {
        return salario;
    }
}

class Registos {
    // Data elements
    private ArrayList<Empregado> empregados; // Stores the employees
    public Registos() {
        empregados = new ArrayList<>();
    }
    public void insere(Empregado emp) {
        // Code to insert employee
    }
    public void remove(int codigo) {
        // Code to remove employee
    }
    public boolean isEmpregado(int codigo) {
        // Code to find employee
    }
    public List<Empregado> listaDeEmpregados() {
        // Code to retrieve collection
    }

}
```

1) Complete o código omisso nos métodos indicados nas classes *Database* e *Registos* e escreva uma função *main* para testar cada conjunto.
2) Escreva um programa que usa ambos os conjuntos de funcionários (*Database* e *Registos*) sem alterar o código legado em qualquer uma dessas classes. O programa deve implementar os seguintes métodos:
   - Um método para adicionar um empregado.
   - Um método para remover um empregado, dado o número de funcionário
   - Um método para verificar se um empregado existe na empresa, dado o número do empregado.
   - Um método para imprimir os registos de todos os funcionários.

VI.2 Gestão dinâmica de lista de contactos

Neste problema pretende-se criar um conjunto de interfaces e classes que permitam a gestão ágil de uma lista de contactos (da classe Contact). Pretende-se nomeadamente que:

a) O armazenamento da lista possa ser feito em qualquer formato (por exemplo, TXT separado por tab, CVS, JSON, XLS, binário, etc.). Não sabemos à partida que formatos poderemos vir a criar. Para resolver este problema defina a seguinte interface que deverá ser respeitada por todas as implementações de armazenamento:

```
public interface ContactsStorageInterface {
    public List<Contact> loadContacts();
    public boolean saveContacts(List<Contact> list);
}
```

b) A utilização da lista de contacto poderá ser realizada por aplicações distintas pelo que, para separar funcionalidades, deve usar a seguinte interface:

```
public interface ContactsInterface {
    public void openAndLoad(ContactsStorageInterface store);
    public void saveAndClose();
    public void saveAndClose(ContactsStorageInterface store);
    public boolean exist(Contact contact);
    public Contact getByName(String name);
    public boolean add(Contact contact);
    public boolean remove(Contact contact);
}
```

Através desta interface deverá ser possível manipular os contactos sem saber o tipo de armazenamento usado.

Desenvolva uma solução para este problema de modo a permitir usar, pelo menos, os formatos texto e binário para armazenamento. Teste a solução com um conjunto de contactos (criados na função main de forma estática, introduzidos num ficheiro de texto, ...).

VI.3 Impressoras

(opcional)

A empresa *SóServiços* distribui dois tipos de impressoras:

a) *BasicPrinter* – aceitam um documento de cada vez e necessitam que sejam adicionados papel e tinta quando estes terminam.
b) *AdvancedPrinter* – são impressoras profissionais que incluem lista de trabalhos (*print queue*) e gestão autónoma da quantidade de papel e de tinta.

A empresa disponibiliza uma aplicação de software para gerir os pedidos para as impressoras profissionais, mas tornou-se necessário incluir também as restantes impressoras na mesma aplicação.

A interface das impressoras profissionais contém os seguintes métodos:

```
public int print(Document doc);
public List<Integer> print(List<Document> docs);
```

```
public void showQueuedJobs();
public boolean cancelJob(int jobId) ;
public void cancelAll();
```

As impressoras simples apresentam os métodos públicos indicados abaixo. O código da classe é disponibilizado para referência. O método *print(...)* recebe o conteúdo a imprimir (texto) como um array de Strings e devolve *false* se não for possível imprimir o texto. O método *refill()* simula a reposição manual de tinta e papel.

```
public boolean print(String[] content)
public void refill()
```

Proponha uma solução que não altere as interfaces das impressoras, mas que permita que as impressoras básicas possam ser usadas da mesma forma das profissionais.

O método *main* fornecido apenas funciona com *AdvancedPrinter*. Modifique-o para incluir e testar um conjunto de impressoras, de ambos os tipos, e envie diferentes pedidos para cada uma.
Com o código, entregue também um ficheiro de output com o nome *printer.txt*.

Output:

```
Spooling 1 documents.
Finished Job 0: "This is a great text..."
Spooling 3 documents.
Spooled jobs:
  * Job 2: "Natural language gen..."
  * Job 3: "You which to know ho..."

Finished Job 1: "This is a great text..."
Finished Job 2: "Natural language gen..."
Finished Job 3: "You which to know ho..."
Spooling 3 documents.
Cancelled Job 6: "You which to know ho..."
Spooled jobs:
  * Job 5: "Natural language gen..."

Finished Job 4: "This is a great text..."
Finished Job 5: "Natural language gen..."
Spooling 3 documents.
Job rejected by spool: service shutting down?
No spooled jobs.
```

Lab VII.

Objetivos

Os objetivos deste trabalho são:

- Identificar e utilizar padrões relacionados com a estrutura de objetos e classes
- Aplicar boas práticas de programação por padrões em casos práticos

Nota: Para além do código no codeUA, inclua também um ficheiro PDF ou PNG com o diagrama de classes da solução final (pode usar o UMLet, por exemplo, ou um plugin Eclipse/Netbeans).

VII.1 Atribuição dinâmica de responsabilidades

A empresa *TodosFazem* (TF) pretende fazer uma gestão dinâmica de funcionários de forma a poder atribuir responsabilidades diversas ao longo do ano.

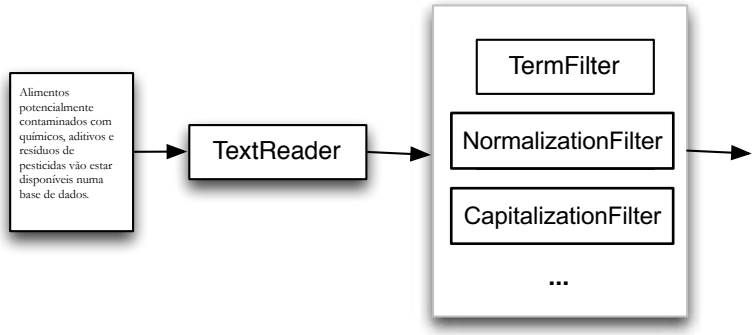
a) Usando como base as entidades seguintes, criando outras se necessário, construa uma solução que permita à empresa gerir empregados e poder atribuir dinamicamente, a cada um, competências distintas. Note que um empregado pode, eventualmente, ter ao mesmo tempo várias competências, por exemplo ser TeamMember e TeamLeader.

b) Crie um programa principal para testar a solução. Por simplicidade pode implementar os métodos apenas com mensagens na consola.

```
Employee
        start (Date)
        terminate(Date)
        work()
TeamMember
        start (Date)
        terminate(Date)
        work()
        colaborate()
TeamLeader
        start (Date)
        terminate(Date)
        work()
        plan()
Manager
        start (Date)
        terminate(Date)
        work()
        manage()
```

VII.2 Processador de texto

Construa uma solução geral que permita ler documentos de qualquer formato (mas na implementação restrinja a ficheiros TXT). O programa deverá permitir ler texto e aplicar um ou mais filtros sobre esse texto.
Tome como base as seguintes entidades, privilegiando a modulação do problema e só depois a implementação de funcionalidades:



- TextReader – lê um ficheiro. Inclui os métodos:
  - boolean hasNext()
  - String next() – devolve parágrafo. Por exemplo:
    Alimentos potencialmente contaminados com químicos, aditivos e resíduos de pesticidas vão estar disponíveis numa base de dados.
- TermFilter – Separa em palavras. Inclui os métodos:
  - boolean hasNext()
  - String next() – devolve palavra (por exemplo: *Alimentos*)
- NormalizationFilter – Remove acentuação e pontuação. Inclui os métodos:
  - boolean hasNext()
  - String next() – devolve texto sem acentuação e pontuação (por exemplo: *quimicos*)
- VowelFilter – Remove vogais. Inclui os métodos:
  - boolean hasNext()
  - String next() – devolve texto sem vogais (por exemplo: *lmnts*)
- CapitalizationFilter – coloca em maiúsculas o primeiro e último caracter do texto, e os restantes em minúsculas:
  - boolean hasNext()
  - String next() – devolve texto capitalizado (por exemplo: *AlimentoS*)

Exemplos de utilização:

```
...
reader = new TextReader("someFileName");
reader = new NormalizationFilter(new TextReader("someFileName"));
reader = new VowelFilter(new TermFilter(new TextReader("someFileName"));
...
```

VII.3 Distribuidor de Cabazes de compras

Construa uma solução que permita criar os seguintes produtos:

- Bebida
- Doce
- Conserva
- Caixa (que pode conter zero ou mais produtos)

Use o programa seguinte para testar a solução:

```
public class Cabazes {
   public static void main(String[] args) {
     Caixa principal = new Caixa("Principal", 4);
     Caixa top = new Caixa("Topo", 2);
     Caixa bot = new Caixa("Especialidades", 2);
     top.add(new Bebida("Vinho Reserva UA 2017", 6));
     top.add(new Bebida("Vinho Reserva UA 2018", 6));
     principal.add(top);
     principal.add(bot);
     bot.add(new Conserva("Atum à Algarvia", 3));
     bot.add(new Doce("Morango", 2));
     top.add(new Caixa("Interior", 1));
     top.add(new Conserva("Sardinhas em Azeite", 5));
     principal.draw();
   }
}
```

Output possível:

```
* Caixa 'Principal' [ Weight: 4.0 ; Total: 31.0]
   * Caixa 'Topo' [ Weight: 2.0 ; Total: 20.0]
      Bebida 'Vinho Reserva UA 2017' - Weight : 6.0
      Bebida 'Vinho Reserva UA 2018' - Weight : 6.0
      * Caixa 'Interior' [ Weight: 1.0 ; Total: 1.0]
      Conserva 'Sardinhas em Azeite' - Weight : 5.0
   * Caixa 'Especialidades' [ Weight: 2.0 ; Total: 7.0]
      Conserva 'Atum à Algarvia' - Weight : 3.0
      Doce 'Morango' - Weight : 2.0
```

Lab VIII.

Objetivos

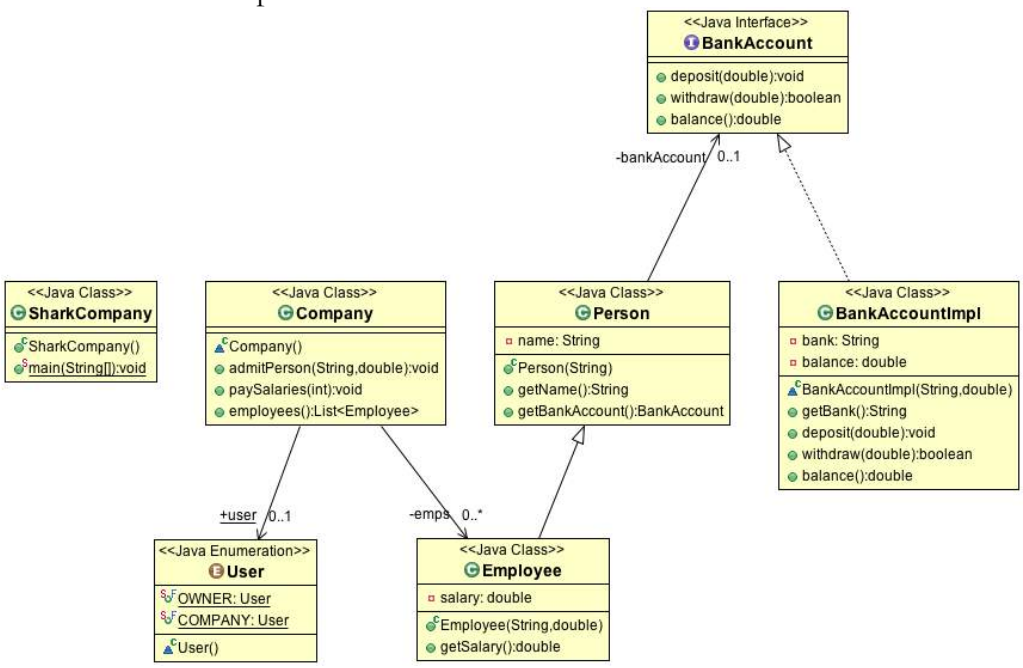
Os objetivos deste trabalho são:

- Utilizar padrões estruturais (i.e., Adapter, Facade, Proxy, Flyweight, etc.) para resolver casos práticos.
- Aplicar boas práticas de programação por padrões

Nota: Para além do código no codeUA, inclua também um ficheiro PDF ou PNG com o diagrama de classes da solução final (pode usar o UMLet, por exemplo, ou um plugin Eclipse/Netbeans).

VIII.1 Gestão de acesso a conta bancária

Considere o programa seguinte que pretende gerir os pagamentos dos salários de funcionários de uma empresa.




```
interface BankAccount {
     void deposit(double amount);
     boolean withdraw(double amount);
     double balance();
}
class BankAccountImpl implements BankAccount {
     private String bank;
     private double balance;
     BankAccountImpl(String bank, double initialDeposit) {
           this.bank = bank;
           balance = initialDeposit;
     }
     public String getBank() {
           return bank;
```

```
    }
    @Override public void deposit(double amount) {
          balance += amount;
    }
    @Override public boolean withdraw(double amount) {
          if (amount > balance )
              return false;
          balance -= amount;
          return true;
    }
    @Override public double balance() {
          return balance;
    }
}

class Person {
    private String name;
    private BankAccount bankAccount;

    public Person(String n) {
          name = n;
          bankAccount = new BankAccountImpl("PeDeMeia", 0);
    }
    public String getName() {
          return name;
    }
    public BankAccount getBankAccount() {
          return bankAccount;
    }
}

class Employee extends Person {
    private double salary;

    public Employee(String n, double s) {
          super(n);
          salary = s;
    }
    public double getSalary() {
          return salary;
    }
}

enum User { OWNER, COMPANY  }

class Company {
    public static User user;
    private List<Employee> emps = new ArrayList<>();

    public void admitPerson(String name, double salary) {
          Employee e = new Employee(name, salary);
          emps.add(e);
    }
    public void paySalaries(int month) {
          for (Employee e : emps) {
              BankAccount ba = e.getBankAccount();
              ba.deposit(e.getSalary());
          }
    }
    public List<Employee> employees() {
          return Collections.unmodifiableList(emps);
    }
}
```

```
public class SharkCompany {

    public static void main(String[] args) {
          Company shark = new Company();
          Company.user = User.COMPANY;
          shark.admitPerson("Maria Silva", 1000);
          shark.admitPerson("Manuel Pereira", 900);
          shark.admitPerson("Aurora Machado", 1200);
          shark.admitPerson("Augusto Lima", 1100);
          List<Employee> sharkEmps = shark.employees();
          for (Employee e : sharkEmps)
              // "talking to strangers", but this is not a normal case
              System.out.println(e.getBankAccount().balance());
          shark.paySalaries(1);
          for (Employee e : sharkEmps) {
              e.getBankAccount().withdraw(500);
              System.out.println(e.getBankAccount().balance());
    }
    }
}
```

Na implementação atual é possível que a empresa aceda aos dados privados da conta bancária de cada pessoa.

a) Construa uma solução que permita ao funcionário impedir a empresa de ter acesso aos métodos *withdraw* e *balance* da sua conta bancária, mantendo ao mesmo tempo a possibilidade de ele próprio aceder a tudo. Utilize a variável *Company.user* para simular o perfil do utilizador. Nesta solução não pode alterar as classes existentes (apenas modificar ligeiramente a classe *Person*).

b) Considerando que as classes *Person* e *Employee* fazem parte de domínios distintos crie uma nova versão do programa (*SharkCompany2*) onde *Employee* não herda de *Person* e o acesso à conta bancária fica limitado à classe *Employee*. Assim deixa de ser possível as funções cliente (*main* por exemplo) usarem expressões como *e.getBankAccount().balance()* (princípio *"Don't talk to stranger"*). Note que esta solução não resolve *per si* o problema identificado em a) uma vez que, se nada for feito, a classe *Employee* (classe da empresa) pode aceder a todos os métodos de *BankAccount*. Exemplo possível para a função *main*:

```
public class SharkCompany {

    public static void main(String[] args) {
          Person[] persons = { new Person("Maria Silva"),
                      new Person("Manuel Pereira"),
                      new Person("Aurora Machado"),
                      new Person("Augusto Lima")   };
          Company shark = new Company();
          Company.user = User.COMPANY;
          shark.admitEmployee(persons[0], 1000);
          shark.admitEmployee(persons[1], 900);
          shark.admitEmployee(persons[2], 1200);
          shark.admitEmployee(persons[3], 1100);
          List<Employee> sharkEmps = shark.employees();
          for (Employee e : sharkEmps)
                System.out.println(e.getSalary());
          shark.paySalaries(1);
    }
}
```

VIII.2 SharkCompany with a Facade

Com base na implementação anterior pretende-se agora criar uma Facade (pode usar a classe *Company* e o método *admitEmployee* para evitar criar uma nova classe) que garanta que quando um novo funcionário é admitido, para além do registo na empresa, são igualmente invocados os seguintes serviços:

1. Registo na segurança social (e.g. *SocialSecurity.regist(person)*)
2. Registo na seguradora (e.g. *Insurance.regist(person)*)
3. Criação de um cartão de funcionário
4. Autorização para use de parque automóvel caso o salário seja superior à média (e.g. *Parking.allow(person)*)

Construa entidades e métodos adequados a este problema. Note que o enfâse é na construção da *facade* e menos na construção de métodos nas novas classes que vai necessitar.

VIII.3 Estrelas no céu

Pretende-se apresentar um grande número de estrelas numa representação do céu. As estrelas são de sete tipos (pasta *startypes*), apresentam diferentes características físicas e a suas posições no espaço são definidas pelas coordenadas *x* e *y*.
A solução que é disponibilizada (*Demo.java*) cria e repete múltiplas instâncias de cada tipo de estrela dando origem a um consumo de memória que poderia ser evitado. Desenvolva uma solução que melhore o código disponibilizado em termos de uso de recursos, i.e., que utilize menos de um 1/3 da memória da solução inicial.

Lab IX.

Objetivos

Os objetivos deste trabalho são:

- Utilizar padrões estruturais (i.e., *Chain of Responsibility, Command, Interpreter, Iterator*) para resolver casos práticos.
- Aplicar boas práticas de programação por padrões

Nota: Para além do código no codeUA, inclua também um ficheiro PDF ou PNG com o diagrama de classes da solução final.

IX.1 Implementação de iteradores sobre um conjunto

Considere o código seguinte:

```
public class VectorGeneric<T> {
   private T[] vec;
   private int nElem;
   private final static int ALLOC = 50;
   private int dimVec = ALLOC;

   @SuppressWarnings("unchecked")
   public VectorGeneric() {
      vec = (T[]) new Object[dimVec];
      nElem = 0;
   }

   public boolean addElem(T elem) {
      if (elem == null)
         return false;
      ensureSpace();
      vec[nElem++] = elem;
      return true;
   }

   private void ensureSpace() {
      if (nElem>=dimVec) {
         dimVec += ALLOC;
         @SuppressWarnings("unchecked")
         T[] newArray = (T[]) new Object[dimVec];
         System.arraycopy(vec, 0, newArray, 0, nElem );
         vec = newArray;
      }
   }

   public boolean removeElem(T elem) {
      for (int i = 0; i < nElem; i++) {
         if (vec[i].equals(elem)) {
            if (nElem-i-1 > 0) // not last element
               System.arraycopy(vec, i+1, vec, i, nElem-i-1 );
            vec[--nElem] = null; // libertar último objecto para o GC
            return true;
         }
      }
      return false;
   }
```

```
    public int totalElem() {
        return nElem;
    }

    public T getElem(int i) {
        return (T) vec[i];
    }
}
```

a) Construa o código necessário para que a classe passe a incluir os seguintes métodos:

```
public java.util.Iterator<E> Iterator()
public java.util.ListIterator<E> listIterator()
public java.util.ListIterator<E> listIterator(index) // start at index
```

Não implemente os métodos opcionais e respeite rigorosamente os contratos especificados na documentação java8.

b) Desenvolva uma classe de teste para verificar todas as operações criadas. Inclua a situação de usar vários iteradores em simultâneo sobre o mesmo conjunto.

IX.2 Chain of Responsibility

Um restaurante tem vários chefes de cozinha, cada um com a sua especialidade. Quando um pedido chega, o primeiro chefe confeciona o pedido se se tratar da sua especialidade; caso contrário, passa o pedido ao chefe seguinte e assim consecutivamente. Implemente e teste a solução, tentando replicar o seguinte resultado.

```
Can I please get a veggie burger?
SushiChef: I can't cook that.
PastaChef: I can't cook that.
BurgerChef: Starting to cook veggie burger. Out in 19 minutes!

Can I please get a Pasta Carbonara?
SushiChef: I can't cook that.
PastaChef: Starting to cook Pasta Carbonara. Out in 14 minutes!

Can I please get a PLAIN pizza, no toppings!?
SushiChef: I can't cook that.
PastaChef: I can't cook that.
BurgerChef: I can't cook that.
PizzaChef: Starting to cook PLAIN pizza, no toppings!. Out in 7 minutes!

Can I please get a sushi nigiri and sashimi?
SushiChef: Starting to cook sushi nigiri and sashimi. Out in 14 minutes!

Can I please get a salad with tuna?
SushiChef: I can't cook that.
PastaChef: I can't cook that.
BurgerChef: I can't cook that.
PizzaChef: I can't cook that.
DessertChef: I can't cook that.
We're sorry but that request can't be satisfied by our service!

Can I please get a strawberry ice cream and waffles dessert?
SushiChef: I can't cook that.
PastaChef: I can't cook that.
BurgerChef: I can't cook that.
PizzaChef: I can't cook that.
DessertChef: Starting to cook strawberry ice cream and waffles dessert. Out in 17 minutes!
```

IX.3 Command

Usando o padrão *Command*, construa uma classe para adicionar um elemento a uma coleção (*java.util.Collection*<*E*>) permitindo realizar a operação *undo*. Repita a metodologia para uma classe que remova um elemento de uma coleção (com possibilidade de *undo*).

Lab X.

Objetivos

Os objetivos deste trabalho são:

- Utilizar padrões de comportamento (i.e., *Mediator, Memento, Null Object, Observer*) para resolver casos práticos.
- Aplicar boas práticas de programação por padrões

Nota: Para além do código no codeUA, inclua também um ficheiro PDF ou PNG com o diagrama de classes da solução final.

X.1

A empresa *Lei Lao* pretende criar um sistema de leilões online. Simule este cenário criando as seguintes entidades, bem como outras que entenda serem fundamentais para uma boa modulação:

- *Produto*, caraterizado por um código único atribuído automaticamente (*int*), descrição (*String*) e preço base (*double*). Cada produto pode estar num dos seguintes estados: stock, leilão, vendas. A passagem a leilão deve incluir o tempo de duração neste estado. Caso não seja licitado deverá ser reposto no stock; caso vendido passará para a lista de vendas.
- *Cliente*, caraterizado por nome (*String*). Pode consultar os produtos em leilão. Pode licitar um (ou mais) produto por um determinado valor, passando a receber informação sempre que esse produto receba uma oferta mais elevada. Deverá ser informado quando a licitação termina e o produto é vendido.
- *Gestor*, caraterizado por nome (*String*). Tem acesso à lista de produtos em stock, em leilão e vendidos. Deve receber informação sempre que uma licitação é feita, ou um produto é vendido.

Para simular a informação trocada com cada cliente pode usar a linha de comando ou Java Swing. Crie um programa *main* de teste para simular uma situação real (por exemplo, com 5 produtos, 3 clientes e 1 gestor).

X.2

Considere o código seguinte. Reescreva-o, usando o padrão *Null Object*, de modo a evitar os erros de execução.

```
abstract class Employee {
   protected String name;
   public abstract String getName();
}

class Programmer extends Employee {
   public Programmer(String name) {
        this.name = name;
   }
   @Override
```

```
    public String getName() {
        return name;
    }
}

class EmployeeFactory {
    public static final String[] names = { "Mac", "Linux", "Win" };

    public static Employee getEmployee(String name) {
        for (int i = 0; i < names.length; i++) {
            if (names[i].equalsIgnoreCase(name)) {
                return new Programmer(name);
            }
        }
        return null;
    }
}

public class NullDemo {
    public static void main(String[] args) {

        Employee emp = EmployeeFactory.getEmployee("Mac");
        Employee emp2 = EmployeeFactory.getEmployee("Janela");
        Employee emp3 = EmployeeFactory.getEmployee("Linux");
        Employee emp4 = EmployeeFactory.getEmployee("Mack");

        System.out.println(emp.getName());
        System.out.println(emp2.getName());
        System.out.println(emp3.getName());
        System.out.println(emp4.getName());
    }
}
```

X.3

Tendo por base o padrão *Mediator*, descreva um problema e apresente uma implementação em Java onde aplique este padrão.
Juntamente com o código entregue um ficheiro README onde descreva: 1) o problema; 2) a solução; 3) referências para recursos/fontes utilizados.

Lab XI.

Objetivos

Os objetivos deste trabalho são:

- Utilizar padrões de comportamento (i.e., *State, Strategy, Template Method, Visitor*) para resolver casos práticos.
- Aplicar boas práticas de programação por padrões
- Rever todos os padrões

Nota: Para além do código no codeUA, inclua também um ficheiro PDF ou PNG com o diagrama de classes da solução final.

XI.1

Uma revista eletrónica quer fornecer aos seus clientes um conjunto de propriedades sobre diferentes telemóveis, como por exemplo, processador, preço, memória, câmara, etc. Os resultados devem ser apresentados numa lista, ordenada por qualquer um dos atributos. Por outro lado, existem vários algoritmos de ordenação com diferentes desempenhos, relativamente ao tempo de processamento e ao espaço ocupado. Assim, é necessário podermos selecionar facilmente o melhor algoritmo (por exemplo, em tempo de execução).

a) Que padrão (padrões?) pode ser aplicado para cumprir estes requisitos?
b) Desenhe um diagrama de classes para responder a este problema.
c) Construa o código necessário para demonstrar o princípio. Inclua 3 algoritmos de ordenação distintos (não é relevante para este exemplo a sua implementação).

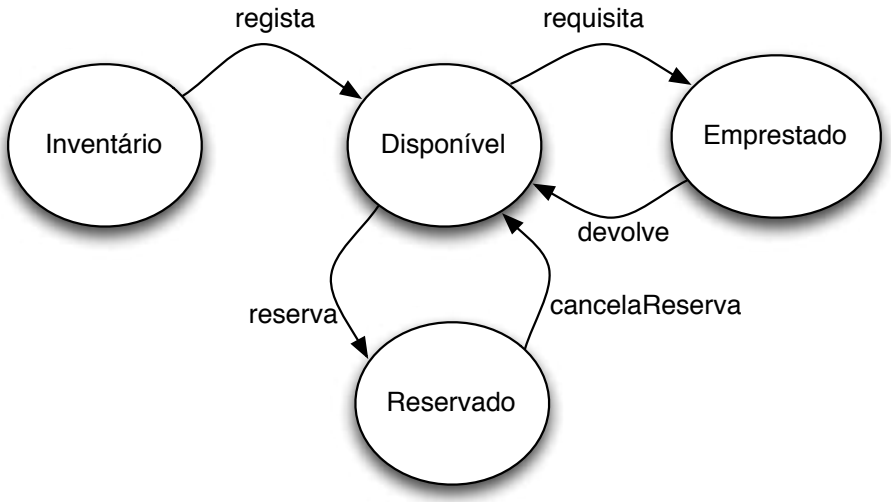
XI.2

O padrão *Template Method* é utilizado em várias classes da Java API. Por exemplo, nas classes abstratas *java.io.InputStream*, *java.io.OutputStream*, *java.io.Reader*, *java.io.Writer*, *java.util.AbstractList*, *java.util.AbstractSet* e *java.util.AbstractMap*.
Selecione 3 destas classes, analise o código fonte (e.g., em *http://openjdk.java.net*) e identifique todas as situações em que o padrão ocorre. Descreva as suas conclusões no ficheiro *Lab11_2.txt*.

XI.3

Numa biblioteca cada livro é caracterizado por um *título*, *ISBN*, *ano*, e o primeiro *autor*. A entidade livro permite operações tais como: regista, requisita, reserva, cancelaReserva, disponível, etc.). No entanto, cada uma destas operações depende da situação do livro na biblioteca: se está em situação de inventário, por exemplo, só permite a operação regista.

a) Considerando o diagrama de estados (círculos) e operações (setas) representados na figura seguinte, construa uma solução que represente adequadamente este problema. *Nota: os vários estados poderão não representar fielmente uma situação real.*



b) Teste a solução criando uma lista de 3 livros e permitindo ao funcionário da biblioteca simular a iteração com a lista da seguinte forma (*o texto escrito pelo utilizador está marcado a azul*):

```
*** Biblioteca ***
1    Java Anti-Stress       Omodionah      [Inventário]
2    A Guerra dos Padrões   Jorge Omel     [Inventário]
3    A Procura da Luz       Khumatkli      [Inventário]
>> <livro>, <operação: (1)regista; (2)requisita; (3)devolve; (4)reserva; (5)cancela

>> 1,1
*** Biblioteca ***
1    Java Anti-Stress       Omodionah      [Disponível]
2    A Guerra dos Padrões   Jorge Omel     [Inventário]
3    A Procura da Luz       Khumatkli      [Inventário]
>> <livro>, <operação: (1)regista; (2)requisita; (3)devolve; (4)reserva; (5)cancela

>> 1,3
Operação não disponível

>> 1,2
*** Biblioteca ***
1    Java Anti-Stress       Omodionah      [Emprestado]
2    A Guerra dos Padrões   Jorge Omel     [Inventário]
3    A Procura da Luz       Khumatkli      [Inventário]
>> <livro>, <operação: (1)regista; (2)requisita; (3)devolve; (4)reserva; (5)cancela

>> 3,1
*** Biblioteca ***
1    Java Anti-Stress       Omodionah      [Emprestado]
2    A Guerra dos Padrões   Jorge Omel     [Inventário]
3    A Procura da Luz       Khumatkli      [Disponível]
>> <livro>, <operação: (1)regista; (2)requisita; (3)devolve; (4)reserva; (5)cancela

>> ...
```

XI.4

Estude a estrutura e a finalidade de todos os padrões que foram apresentados em PDS. Responda ao seguinte questionário em linha, fazendo uma gestão da quantidade de acertos e falhas:

http://www.vincehuston.org/dp/patterns_quiz.html

XI.5

Um exemplo do padrão *Visitor* em Java visa mediar a interação com a estrutura de diretórios do sistema de ficheiros. Consulte a documentação do Java para perceber como usar as seguintes classes (*java.nio.file.FileVisitor*, *java.nio.file.SimpleFileVisitior*) e o método *java.nio.files.Files.walkFileTree*. Eles permitem aceder ao sistema de ficheiros utilizando o padrão *Visitor*.

a) Utilize o *grepcode.com* ou o *docjar.com* para ficar a conhecer a implementação destas classes.

b) Desenvolva um programa conceptual que devolva o tamanho de um diretório, i.e. a soma dos tamanhos de todos os ficheiros. Adicione a opção –r (recursiva), assim o programa contará com o tamanho dos subdiretórios dentro do diretório raiz.

```
$ java –jar sizeOf.jar root
A: 10 kB
C: 8  kB
Total: 18 kB
==================================
$ java –jar sizeOf.jar –r root
A: 10  kB
B: 100 kB
|->Test.file: 100kB
C: 8   kB
Total: 118 kB
```

Lab XII.

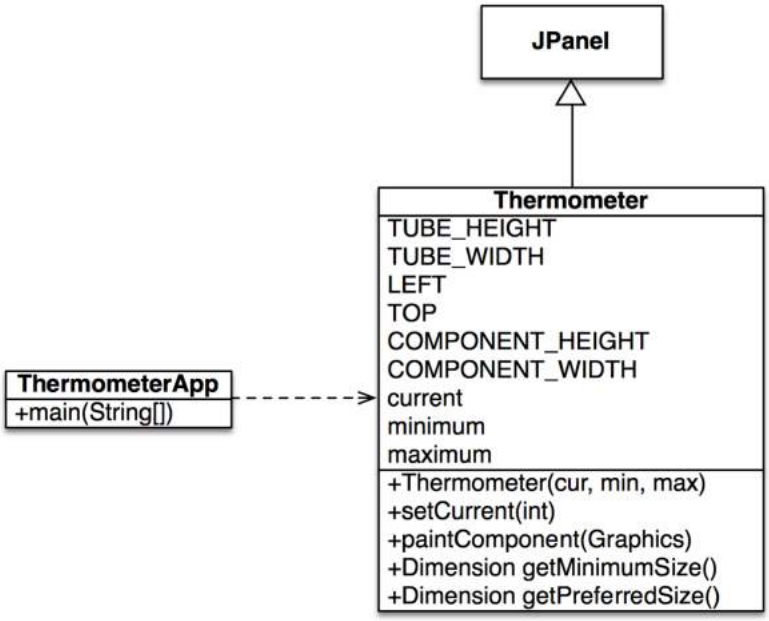
XII.1 Arquiteturas microkernel (plugins)

Tome como referência o seguinte código (*IPlugin.java* e *Plugin.java*). Construa um conjunto de classes que implementem a interface *IPlugin* e que permitam adicionar funcionalidades ao programa principal.

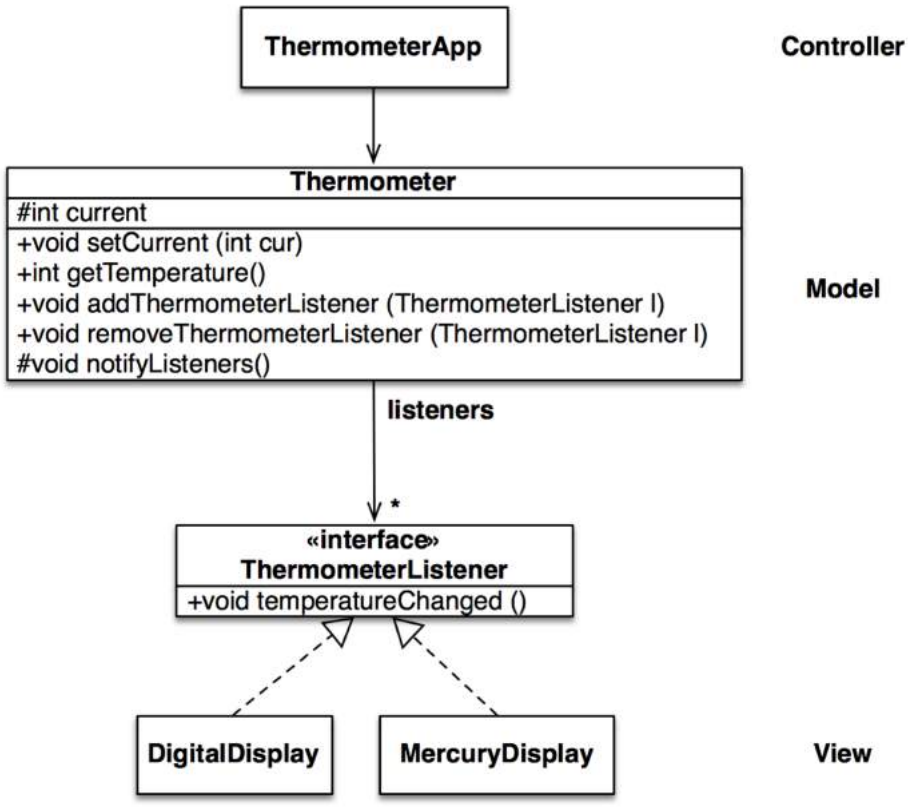
```
// IPlugin.java
package reflection;
public interface IPlugin {
        public void fazQualQuerCoisa();
}

// Plugin.java
package reflection;

import java.io.File;
import java.util.ArrayList;
import java.util.Iterator;

abstract class PluginManager {
   public static IPlugin load(String name) throws Exception {
      Class<?> c = Class.forName(name);
      return (IPlugin) c.getDeclaredConstructor().newInstance();
   }
}

public class Plugin {
   public static void main(String[] args) throws Exception {

       File proxyList = new File("bin/reflection");
       ArrayList<IPlugin> plgs = new ArrayList<IPlugin>();
       for (String f: proxyList.list()) {
         if (f.endsWith(".class")) {
            try {
               plgs.add(PluginManager.load("reflection."
                              + f.substring(0, f.lastIndexOf('.'))));
            } catch (Exception e) {
               System.out.println("\t" + f + ": Componente ignorado. Não é IPlugin.");
            }
         }
       }
       Iterator<IPlugin> it = plgs.iterator();
       while (it.hasNext()) {
          it.next().fazQualQuerCoisa();
       }
   }
}
```

XII.2 Arquitetura Model-View-Control

No ficheiro *thermo.zip* encontrará uma solução para representar um termómetro em Java Swing. Esta abordagem, numa única classe, torna difícil mudar a implementação e adicionar, por exemplo, uma segunda forma de visualização.



Numa segunda implementação, *thermoMVC.zip*, optou-se por organizar o código segundo um modelo *Model-View-Control*, que usa o padrão *Observer*.



O termómetro está agora dividido em um modelo e duas representações (observações) diferentes. Com esta arquitetura, podemos facilmente modificar a nossa *ThermometerApp* para utilizar a visualização que queremos, podemos usar ambos, ou poderemos usar múltiplas réplicas de uma ou mais visualizações. Podemos criar estas variações com pequenas alterações na *ThermometerApp* (controlo) e nenhuma para a classe termómetro

(modelo).

O Controlo neste exemplo é bastante simples, sendo representado por um campo de texto que permite inserir uma nova temperatura. Isso é implementado com um *ActionListener* – que está associado ao campo de texto. Este *listener* faz uma chamada direta ao Termómetro que regista a mudança no modelo e, em seguida, usa *notifyListeners* para informar todos os *observers* para que possam atualizar a interface (view).
O objetivo deste trabalho é analisar o código fornecido e criar um terceira representação do termómetro (usando Swing, a consola, um ficheiro, etc.).

XII.3 Serialização JSON de objetos arbitrários

(opcional, para quem queira estudar java reflection)

Um dos usos mais comuns das funcionalidades *Reflection* do java é encontrado na Serialização de objetos arbitrários, por exemplo para o formato JSON. De facto, a biblioteca JSON-Lib utilizada nesta cadeira utiliza precisamente esta estratégia. Explore as funcionalidades de *Reflection* do java de maneira a serializar objetos de forma recursiva. A sua implementação deverá ser capaz de exportar os atributos e métodos públicos, tendo em atenção o seu tipo de retorno (outros objetos, arrays, etc). Note que utilizando esta estratégia é possível criar um método *default* para serializar todo o tipo de objetos, independentemente da sua implementação.

```
public class Ship {

	private String name;
	private int size;
	private String[] passageiros;
	private Owner owner;

	public Ship(String name, int size) {
		super();
		this.name = name;
		this.size = size;
	}

	public String getName() {
		return name;
	}

	public int getSize() {
		return size;
	}

	public Owner getOwner() {
		return owner;
	}

	public void setOwner(Owner owner) {
		this.owner = owner;
	}

	public String[] getPassageiros() {
		return passageiros;
	}

}
```

```
public class PDSSerializer {

	public static String fromObject(Object o){
		//Class cl = o.getClass();
		//Explore os metodos
		//cl.getMethods();
		//cl.getFields();
		//Veja o javadoc das classes: Class, Method, Field, Modifier
	}
}


public static void main(String[] args) {
	Ship s = new Ship("BelaRia", 200);
	s.setOwner(new Owner("Manuel"));
	s.setPassageiros(new String[]{“Manuel”, “Amilcar”});

	System.out.println( PDSSerializer.fromObject(s) );
}
```

```
$ java –jar Serializer.jar
Name: BelaRia
Price: 200
Owner: {
Name: Manuel
}
Passageiros: [Manuel, Amilcar]
```