Universidade de Aveiro, DETI

Padrões e Desenho de Software

Guião das aulas práticas

LEI – Licenciatura em Engenharia Informática

Ano: 2019/2020

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
ACWEZOLMZOCT
VIUQVRAMDGWH
AGFTWPJZWUMH
programming;java;words lines civic
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:

- O puzzle é sempre quadrado, com o tamanho máximo de 50x50.
- As letras do puzzle estão em maiúscula.
- Na lista, as palavras podem estar só em minúsculas, ou misturadas.
- As palavras são compostas por caracteres alfabéticos.
- As palavras têm de ter pelo menos 4 caracteres.
- A lista de palavras pode conter linhas em branco.
- Cada linha pode ter mais do que uma palavra, separadas por vírgula, espaço ou ponto e vírgula.
- Todas as palavras da lista têm de estar no puzzle e apenas uma vez. O programa deve permitir a utilização de palíndromos (palavras que podem ser lidas tanto da esquerda para a direita como no sentido inverso, como é o caso da palavra CIVIC).
- A lista de palavras não pode conter palavras duplicadas ou frases redundantes (por exemplo, não pode conter BAG e RUTABAGA ao mesmo tempo).

(b) Requisitos de Saída

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

(c) Exemplo de Execução

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       5,5     left
CIVIC           5       6,11    down
CIVIC           5       10,11   up
TEST            4       2,8     right
STACK           5       1,1     right

$ java WSSolver –timing sdl_01.txt
Elapsed time (secs): 0.047
PROGRAMMING     11      12,6    up
JAVA            4       9,1     down
WORDS           5       11,11   upleft
LINES           5       5,5     left
CIVIC           5       6,11    down
CIVIC           5       10,11   up
TEST            4       2,8     right
STACK           5       1,1     right
```

O tempo é medido entre a leitura do ficheiro e a listagem de resultados.

I.2 Word Search Generator

Escreva o programa WSGenerator, que uma *Sopa de Letras* de acordo com o formato apresentado no exercício anterior, e assumindo os mesmos requisitos de entrada. 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*".

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_g<XX>.txt'*, em que *<XX>* é o seu número de grupo, contendo um código de grupo. Não partilhe com ninguém porque será usado como código de autenticação do grupo.

Aceda ao link https://forms.gle/DoJ56EYqaUVCihs18 e preencha o formulário. 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 objetiva.

(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):

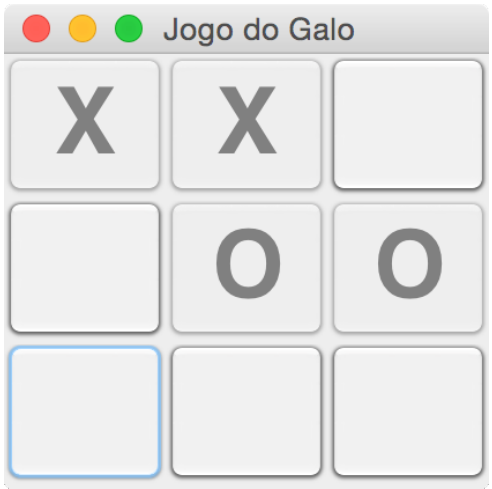
a) A aplicação visual do jogo, classe *JGalo*, desenvolvida sobre java Swing e que não precisa de ser modificada para este trabalho. Apesar disso, recomenda-se a sua análise cuidada.

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 Rua

O objetivo é desenvolver um programa para gerir a localização de famílias e seus membros ao longo de uma rua. Analise os requisitos e planeie cuidadosamente as interfaces, classes, e estruturas de dados mais adequadas.

Requisitos iniciais

1. A numeração de porta começa em 1.
2. Cada membro pode ocupar um ou mais números [x1, x2], em que 0 <x1<=x2.
3. Dois membros com o mesmo intervalo de localização são considerados membros da mesma família.
4. Cada família tem um ou mais membros nomeados da seguinte forma:
    a. O nome é único
    b. Um nome contém apenas letras (maiúsculas e minúsculas), dígitos numéricos e estes símbolos: _.@
    c. O nome deve começar com uma letra.
    d. As letras podem ser maiúsculas ou minúsculas.
    e. O nome não pode acabar um símbolo.
    f. O comprimento máximo de um nome é de 40 caracteres.

Comandos

O programa deve permitir os seguintes comandos, lidos da consola (*case insensitive*):

```
Load, Map, Add, Remove, List, Lookup, Clear, Quit
Command:
```

- **Load *filename***: Lê um ficheiro de texto para construir uma representação interna da rua. Este arquivo pode ter uma linha de descrição inicial que começa com um caracter ">". As linhas seguintes contêm um membro por linha na forma "*Número Inicial-NúmeroFinal Nome*". Esta lista pode aparecer em qualquer ordem.

Exemplo de ficheiro:

```
> Exemplo de teste
10-12 nola
11-14 mikey@ua.pt
4-7 Ana_98
4-7 mikey
10-17 rui
10-17 wx1
16-18 pete
5-6 falB
16-18 silvia
16-18 sergi
```

- **Map**: mostra o mapa da rua no formato seguinte. Habitantes com intervalos sobrepostos são exibidas em colunas paralelas.
Se duas, ou mais famílias, começarem no mesmo valor, a que tem o maior alcance aparece no lado esquerdo. Cada local ocupado por uma família deve ser separado por dois pontos.

Exemplo de Mapa:

```
 1
 2
 3
 4 : mikey Ana_98
 5 : mikey Ana_98 : falB
 6 : mikey Ana_98 : falB
 7 : mikey Ana_98
 8
 9
10 : rui wx1 : nola
11 : rui wx1 : nola : mikey@ua.pt
12 : rui wx1 : nola : mikey@ua.pt
13 : rui wx1       : mikey@ua.pt
14 : rui wx1       : mikey@ua.pt
15 : rui wx1
16 : rui wx1 : pete silvia sergi
17 : rui wx1 : pete silvia sergi
18           : pete silvia sergi
```

- **Add *nome x1 x2***: cria um novo membro na localização x1, x2. Se o membro já existir deve ser reportado um erro.

- **Remove *name***: remove um membro da família. Se o membro não existir deve ser reportado um erro.

(Os seguintes comandos são opcionais)

- **List**: apresenta todos os nomes e números de porta, ordenados alfabeticamente.

```
a 1 2
b 1 2
cnd 1 2
sm7_a 5 6
```

- **Lookup *name***: procura um habitante e apresenta a informação da família, na forma:

```
x1 x2 : name1 name2
```

- **Clear**: limpa toda a informação da rua.

- **Quit**: 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 distribuição de habitantes.

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

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

(b) Avaliação formal

Aceda ao link - https://goo.gl/forms/3imCpbSRlolkv0c13 - e preencha o formulário. 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.

(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.

Rever:
Mudar V3 para ser diferente dos slides. Criar problema(s) novo(s)

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 comidas Take Away

Pretende-se criar um pequeno programa que dado um conjunto conhecido de comidas escolha o recipiente mais adequado ao seu transporte. Considere que os diferentes tipos de comidas são definidos segundo a seguinte interface.

```
public interface Commodity {
        public Temperature getTemperature();
        public State getState();
}
public enum State {
        Solid, Liquid;
}
public enum Temperature {
        WARM, COLD;
}
```

Deve criar um conjunto de classes que modele as seguintes comidas:

Classe	Estado	Temperatura	Outros
Milk	Liquid	Warm	
FruitJuice	Liquid	Cold	FruitName
Tuna	Solid	Cold	
Pork	Solid	Warm	

Analogamente, a descrição dos recipientes deve seguir a seguinte interface. Devem também ser criadas as classes para modelar os seguintes recipientes.

```
public abstract class Container {
        protected Commodity commodity;

        public boolean placeCommodity(Commodity c){
                this.commodity = c;
                return true;
        }
}
```

Classe	Adequado a:	
	Estado	Temperatura
PlasticBottle	Liquid	Cold
TermicBottle	Liquid	Warm, Cold
Tupperware	Solid	Warm, Cold
PlasticBag	Solid	Cold

Modele as classes e construa o código necessário para que o cliente possa executar

pedidos como os apresentados no método *main* seguinte. Deverá criar os dois métodos fábrica tendo em conta os requisitos de temperatura de estado do alimento de maneira a condicionar os alimentos de forma perfeita.

```
public static void main(String[] args) {
        Commodity[] menu = new Commodity[2];
        menu[0] = BeverageFactory.createBeverage(Temperature.COLD);
        menu[1] = MeatFactory.createMeat(Temperature.WARM);

        Container[] containers = new Container[2];
        containers[0] = ContainerFactory.createContainerFor(menu[0]);
        containers[1] = ContainerFactory.createContainerFor(menu[1]);

        containers[0].placeCommodity(menu[0]);
        containers[1].placeCommodity(menu[1]);

        System.out.println("Thank you for choosing your meal!");
        for(Container c : containers){
                System.out.println(c);
        }
}
```

Output:

```
Thank you for choosing your meal!
PlasticBottle [commodity=FruitJuice [fruit=Orange Temperature()=COLD, State()=Liquid]]
Tupperware [commodity=Pork [Temperatura()=WARM, State()=Solid]]
```

V.2 Fornecedor de almoços no campus universitário

Pretende-se criar um conjunto de classes que modele a elaboração de ementas no campus universitário. Para tal, considere que um almoço é representado pela classe *Lunch*.

```
class Lunch {
    private String drink;
    private String mainCourse;
    private String side;

       //.. restantes métodos
}
```

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

```
interface LunchBuilder {
    public void buildDrink();
    public void buildMainCourse();
    public void buildSide();
    public Lunch getMeal();
}
```

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 almoço é sempre o mesmo, apenas variando o *LunchBuilder* passado em *LunchDirector*.

```
public static void main(String[] args) {
    LunchBuilder lunch = new CrastoLunchBuilder();
```

```
        LunchDirector mealDirector = new LunchDirector(lunch);
        mealDirector.constructMeal();
        Lunch meal = mealDirector.getMeal();
        System.out.println("Ana's meal is: " + meal);

        mealDirector = new LunchDirector(new SnackLunchBuilder());
        mealDirector.constructMeal();
        meal = mealDirector.getMeal();
        System.out.println("Rui's meal is: " + meal);

        mealDirector = new LunchDirector(new CentralCantineLunchBuilder());
        mealDirector.constructMeal();
        meal = mealDirector.getMeal();
        System.out.println("My meal is: " + meal);
    }
```

Output:

```
Ana's meal is: [ drink: Vinho Tinto, main course: Bacalhau à lagareiro, side: Broa ]
Rui's meal is: [ drink: Sumo, main course: Pão com Panado, side: Rissol ]
My meal is: [ drink: Água, main course: Grelhada mista, side: Queijo fresco ]
```

V.3 Construtor com demasiados parâmetros

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

```
public class Person
{
   private final String lastName;
   private final String firstName;
   private final String middleName;
   private final String salutation;
   private final String suffix;
   private final String streetAddress;
   private final String city;
   private final String state;
   private final boolean isFemale;
   private final boolean isEmployed;
   private final boolean isHomewOwner;

   public Person(
      final String newLastName,
      final String newFirstName,
      final String newMiddleName,
      final String newSalutation,
      final String newSuffix,
      final String newStreetAddress,
      final String newCity,
      final String newState,
      final boolean newIsFemale,
      final boolean newIsEmployed,
      final boolean newIsHomeOwner)    {
             this.lastName = newLastName;
             this.firstName = newFirstName;
             this.middleName = newMiddleName;
             this.salutation = newSalutation;
             this.suffix = newSuffix;
             this.streetAddress = newStreetAddress;
             this.city = newCity;
             this.state = newState;
             this.isFemale = newIsFemale;
             this.isEmployed = newIsEmployed;
             this.isHomewOwner = newIsHomeOwner;
```

```
        }
    }
```

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
http://grepcode.com/file/repository.grepcode.com/java/root/jdk/openjdk/6-b14/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, ...).

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.

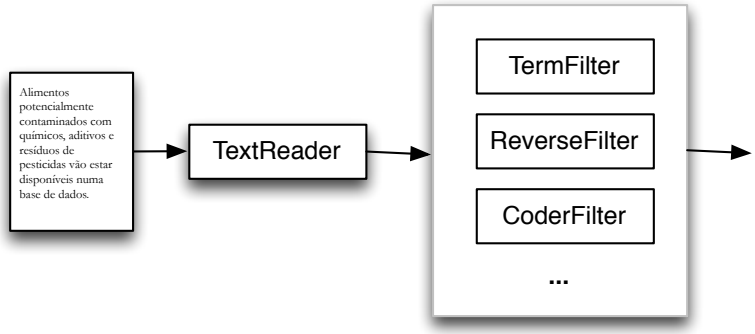
b) Crie um programa principal para testar a solução. Por simplicidade pode implementar os métodos apenas com mensagens na consola.

```
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 termo (por exemplo: *Alimentos*)
- ReverseFilter – inverte a entrada. Inclui os métodos:
  - boolean hasNext()
  - String next() – devolve texto invertido (por exemplo: *sotnemilA*)
- CoderFilter – cifra a entrada (usando uma técnica simples). Inclui os métodos:
  - boolean hasNext()
  - String next() – devolve texto cifrado (por exemplo: *Bl3m2nt4s*)

Exemplos de utilização:

```
...
reader = new TextReader("someFileName");
reader = new CoderFilter(new TextReader("someFileName"));
reader = new ReverseFilter(new TermFilter(new TextReader("someFileName"));
...
```

VII.3 Composição gráfica

Construa uma solução que permita criar as seguintes figuras geométricas:

- Quadrado
- Rectângulo
- Círculo
- Bloco (que é um agregado de figuras geométricas)

Use o programa seguinte para testar a solução:

```
public class GeometricFigures {
    public static void main(String[] args) {
        Bloco principal = new Bloco("Main");
        Bloco top = new Bloco("Top");
        Bloco bot = new Bloco("Bottom");
        top.add(new Rectangulo("jogo"));
        principal.add(top);
        principal.add(bot);
        bot.add(new Circulo("rosa"));
        bot.add(new Quadrado("verde"));
```

```
            top.add(new Bloco("Outra área"));
            principal.draw();
        }
    }
```

Output possível:

```
Window Main
   Window Top
      Rectangulo jogo
      Window Outra área
   Window Bottom
      Circulo rosa
      Quadrado verde
```

Lab VIII.

Melhorar, Criar + um problema

Objetivos

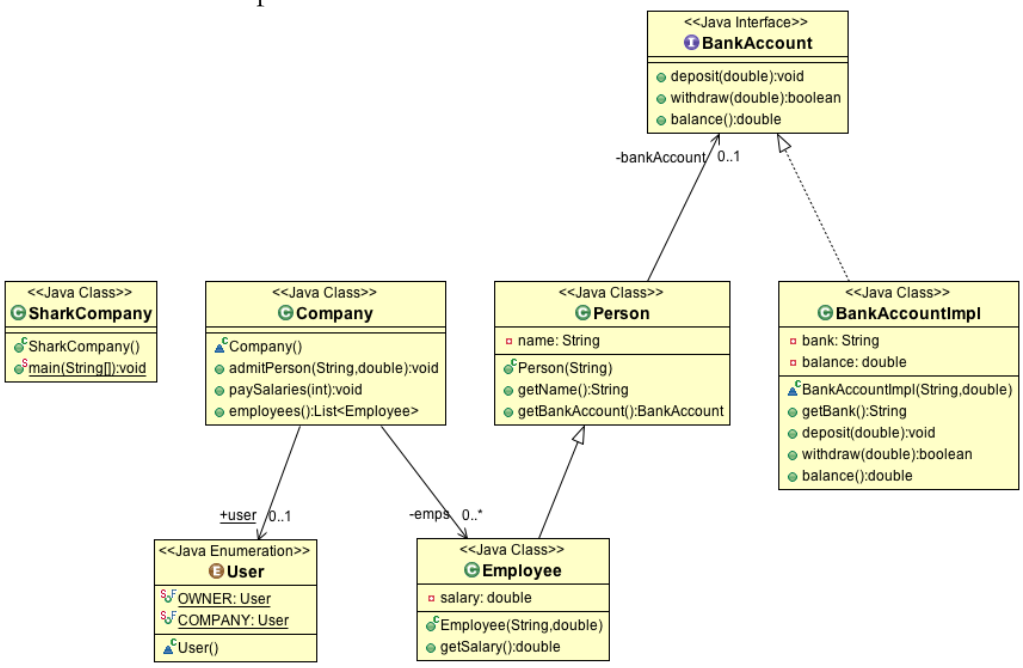
Os objetivos deste trabalho são:

- 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.

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 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*).

IX.3 Chain of Responsability

Reveja o padrão *Chain of Responsability* e proponha/construa um exemplo de aplicação que não repita a solução apresentada nos slides.

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

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

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).

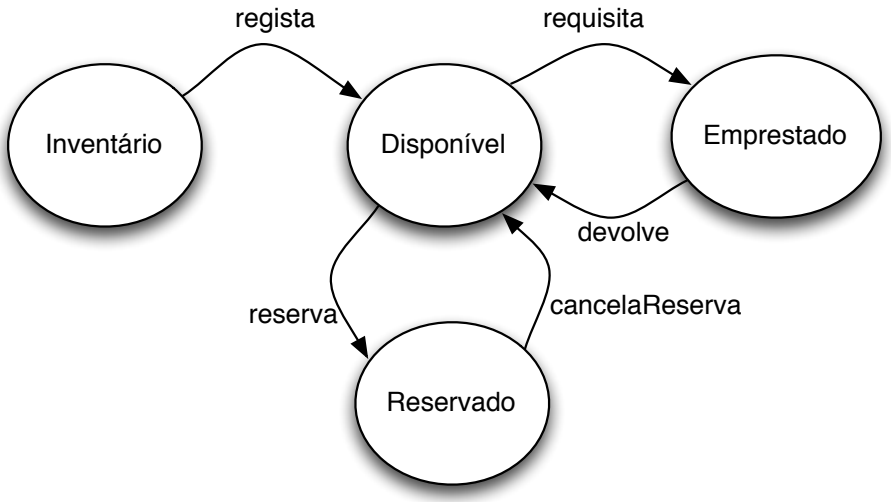
XI.2

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 (disponível, por exemplo, em *grepcode.com* ou *docjar.com*) 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* foi introduzido no Java 7 para 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.

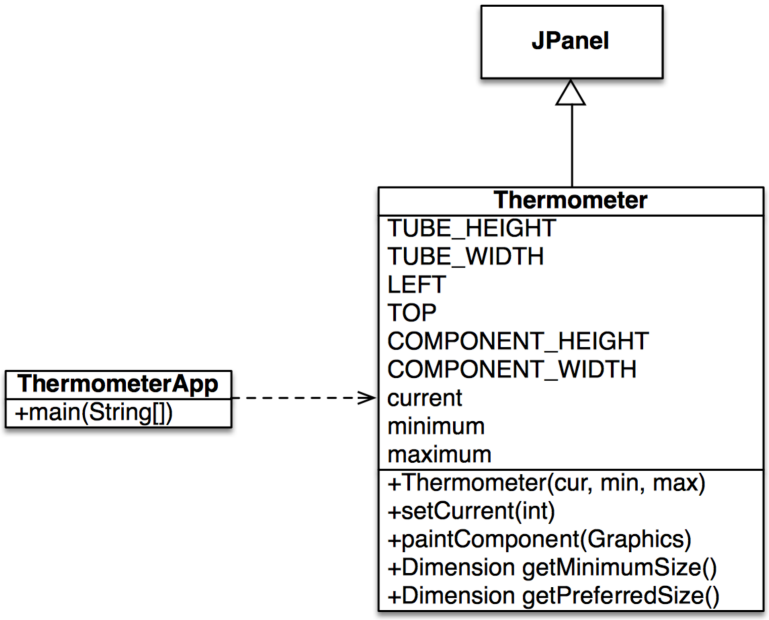
XII.1 Arquiteturas microkernel (plugins)

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.

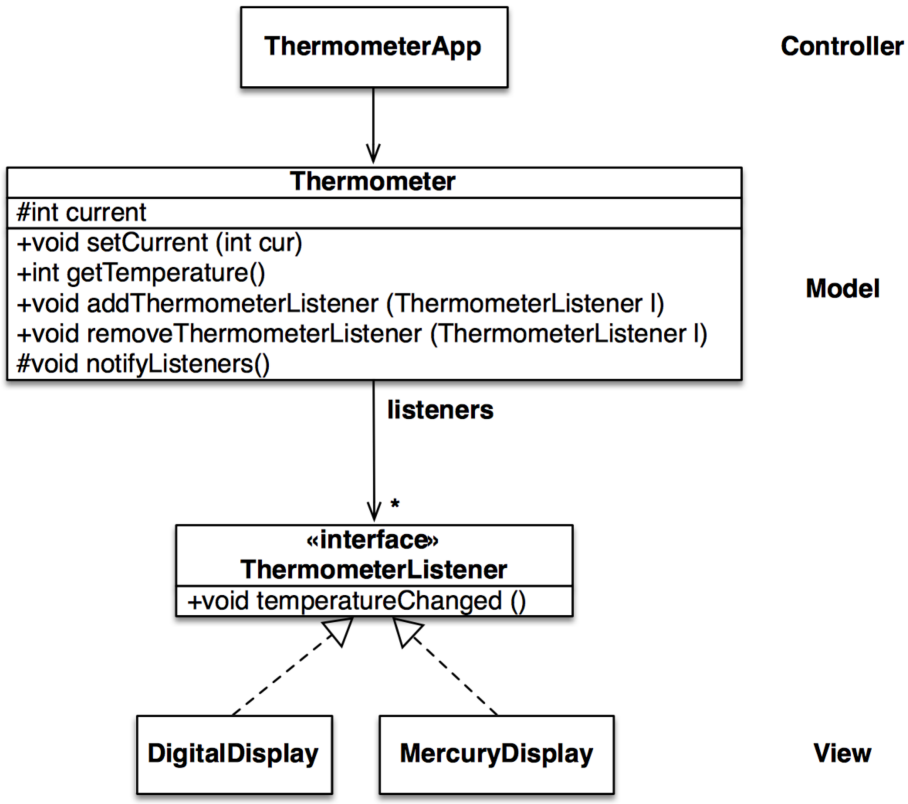
```
// IPlugin.java
package reflection;
public interface IPlugin {
        public void fazQualQuerCoisa();
}

// 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]
```

Lab XIII.

Objetivos

Os objetivos deste trabalho são:

- Aplicar conceitos de programação adquiridos em aulas anteriores:
- Aplicar conceitos de modulação de software necessários no desenvolvimento de uma solução
- Rever e consolidar competências de desenvolvimento de software

Condições

Este trabalho deve ser realizado no prazo de duas semanas. Na primeira semana, é necessário implementar a parte de leitura da configuração (XML) e o gerador de tabuleiro (com a distribuição dos barcos). O restante trabalho deverá ser realizado durante a segunda semana.

XIII.1 Jogo batalha naval

O objetivo deste projeto é desenvolver uma versão Java do popular jogo de tabuleiro "Batalha Naval". Deverá ser desenvolvida uma solução completa; preste atenção à modulação; decomponha a solução em diferentes módulos para tomar partido de todas as capacidades da programação orientada por objetos; e use padrões de desenho de software adequados à solução.

O jogo deverá iniciar com o carregamento das configurações de tabuleiro de um ficheiro de entradas. Neste ponto, é apenas requerido desenvolver uma versão para um único jogador. Nesta versão, o jogador irá tentar afundar todos os navios do computador no menor número de jogadas possível. O jogo deverá iniciar com a pergunta do nome do jogador. De seguida, o computador deverá colocar os seus navios aleatoriamente. Nesse momento, o jogador começa a disparar aos barcos do computador, um único tiro por jogada. O jogo termina quando todos os navios sejam "afundados". É atribuída uma classificação de acordo com a dificuldade do jogo, assim como o desempenho do jogador.

O programa deverá disponibilizar um menu onde é possível:
a) Iniciar novo jogo; b) Continuar o jogo anterior; e c)Ver a tabela de classificações

Não é necessário seguir uma interface predefinida. No entanto, o jogo deverá respeitar a configuração bem como as especificações do *logger*. Todas as jogadas deverão ficar registadas no *logger* e, ao fazê-lo, o jogo poderá ser interrompido e retomado mais tarde. Para além disso, o jogo terá de ser compatível com a versão do jogo de outros colegas. Os ficheiros de configuração e *logging* deverão ser chamados, config.xml e log.txt, respectivamente.

Para mais informações sobre este jogo, por favor visite: Battleship at Wikipedia

(a) Configurações

O projeto deverá ter um XML como ficheiro de configuração. Juntamente com esta proposta é disponibilizado um exemplo correto do que é pretendido. Deverá cumprir integralmente com as suas especificações.
De modo a carregar a configuração, é sugerida a utilização da Apache Commons Configuration Library. Esta biblioteca é a interface de configuração em Java mais utilizada. Disponibiliza um conjunto de ferramentas para lidar com os múltiplos formatos de configuração. No entanto, para este problemas, apenas é necessário a *Hierarchical Configuration* (*XMLConfiguration* class) – ver exemplo no user's guide. A secção "g" também disponibiliza um exemplo para ajudar no arranque do projeto.

- Tamanho do tabuleiro: Número de linhas e colunas no tabuleiro (<=25, cada). Ex. "8x10".
- Navios: (Nome, número, tamanho)
- Navios complexos: (nome, número, matriz). Estes navios não estão incluídos no jogo original. Podem assumir vários formatos, em vez de apenas um segmento com uma linha simples. A sua forma é mapeada como uma matriz no ficheiro de configuração. Cada dígito mapeia uma única célula, "0" são as células não ocupadas, enquanto "1" são as células ocupadas. Todas as linhas devem conter o mesmo número de células.
- Abaixo encontra-se um exemplo do ficheiro de configuração. Também é disponibilizado separadamente no repositório. O resto deste documento tem como referência esta configuração.

```
<configuration>
	<board>
		<rows>8</rows>
		<collumns>10</collumns>
	</board>
	<ships>
		<ship>
			<name>Aircraft carrier</name>
			<number>1</number>
			<size>5</size>
		</ship>
		<ship>
			<name>Battleship</name>
			<number>1</number>
			<size>4</size>
		</ship>
		<ship>
			<name>Submarine</name>
			<number>1</number>
			<size>3</size>
		</ship>
		<ship>
			<name>Destroyer</name>
			<number>1</number>
			<size>2</size>
		</ship>
		<ship>
			<name>Patrol boat</name>
			<number>2</number>
			<size>1</size>
		</ship>
		<complex-ship>
			<name>SuperShip</name>
			<number>1</number>
			<matrix>
				<row>0100</row>
				<row>1111</row>
				<row>0100</row>
			</matrix>
		</complex-ship>
	</ships>
</configuration>
```

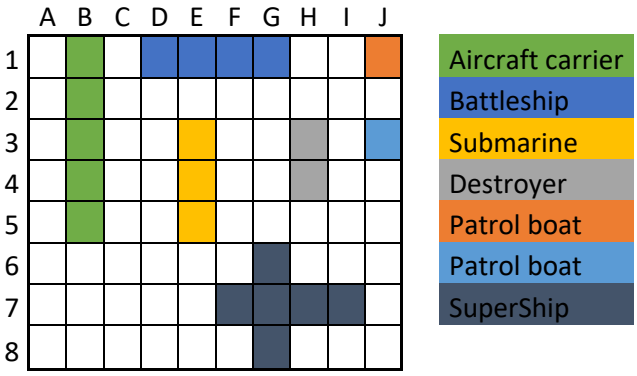
(b) Colocação de navios

Deve desenvolver um distribuidor aleatório dos navios. Quando o jogador inicia novo jogo, deve colocar os navios de acordo com as seguintes regras;

- Os tipos de navios devem ser descarregados do ficheiro de configuração. Note que é possível ter vários navios do mesmo tipo (*number*).
- O tamanho do navio indica o número de células necessárias.
- Navios normais devem ser colocados horizontal ou verticalmente (em qualquer direção). Navios complexos não podem ser rodados, devem ser colocados exatamente seguinte a matriz.
- Os navios não devem ser colocados em células contíguas.

(c) Tabuleiro

- O tamanho do tabuleiro deve ser descarregado das configurações. Os endereços das células devem seguir o exemplo de um tabuleiro "8x10".
- O sistema de coordenadas deve seguir o exemplo dado abaixo. As colunas são identificadas pelas letras (A a Z), enquanto as linhas são identificadas por números (1 a 25). Assim sendo, a primeira célula deverá ser "A1".



a)

(d) Logger/Registo

O ficheiro log é a interface de saída mais importante do projeto. Todos os jogos, e cada movimento, deverão ser guardados no log file. Ao fazer isso, deverá ser possível parar e continuar jogos, guardar estatísticas de jogos anteriores e fazer o mesmo para os jogos/projetos de colegas. Deverá respeitar integralmente o seguinte formato.

- Cada entrada de jogo começa por logging: *"*`===GAME N===`*"*, em que "N" é o número de jogo.
- Depois de identificar o jogo, deve carregar as definições de jogo de acordo com o seguinte esquema:

a)

```
Board:ROWSxCOLUMNS
Player1:ThePlayerName
Boat Name:[CELL1,CELL2,...,CELL(n)]
```

b)

- Para fins de depuração, é obrigatório imprimir a matriz de tabuleiro. Primeiro imprimindo a tag "===Matrix===". De seguida, imprimindo cada linha numa linha separada. Use a mesma anotação do ficheiro de configuração, substituindo os zeros por espaços.
- Escrever **"**`===START===`**"** para marcar o início do registo de movimentos.
- Os movimentos do jogador devem ser registados da seguinte forma:

c)

```
MovementNumber:Player1:[CELL]:Custom Message!! as you like.
```

d)

e)

- Escrever **"**`===END===`**"** para marcar o fim do registo desse jogo. Esta linha final só deve aparecer se o jogo for terminado com sucesso. Se o jogo for interrompido com pelo menos um navio no tabuleiro, este marcador não deve ser escrito, para permitirá continuar o jogo mais tarde.
- Caso se inicie um novo jogo deverá ser anotado pela tag *Start*.
- Abaixo encontra-se um exemplo de um ficheiro log. A maioria dos movimentos dos jogadores está cortada por razões de simplicidade.

```
===GAME 0===
Board:8x10
Player:ThePlayerName
Aircraft carrier:[B1;B2,B3,B4,B5]
Battleship:[D1,E1,F1,G1]
Submarine:[E3,E4,E5]
Destroyer:[H3,H4]
Patrol boat:[J1]
Patrol boat:[J3]
SuperShip:[H6,F7,G7,H7,I7,J7,G8]
===MATRIX===
0101111001
0100000000
0100100101
0100100100
0100100000
0000001000
0000011110
0000001000
===START===
0:Player1:[A1]:Missed Shot
1:Player1:[J1]:Bull's eye!! Patrol Boat Sunk
2:Player1:[G7]:Enemy's SuperShip was hit, (1/7)
…
===END===
```

f)
g)

- Se o utilizador quiser continuar o jogo "N", deve iniciar fazendo registo da tag de jogo ("GAME2", por exemplo), seguido da tag de continuação ("RESUME").

h)

```
===GAME 2===
===RESUME===
25:Player1:[J5]:Bull's eye!! Patrol Boat Sunk.
Congratulations, all the ships have been sunk.
===END===
```

i)

(e) Estatísticas

- O projeto deve ser capaz de carregar jogos anteriores e disponibilizar algumas estatísticas.
- Sistema de ranking: Fornecer uma classificação contendo o nome e pontuação do jogador. Os pontos devem ser calculados usando a seguinte fórmula:

a)

```
NumberOfPlayedRound * DificultyIndex;
DificultyIndex = SizeBoard/TotalBoatSize + 1/AverageBoatSize;
```

b)
c)
d) Em que:
   - SizeBoard → número de células no tabuleiro
   - TotalBoatSize → número de células ocupadas pelos barcos
   - AverageBoatSize → média ponderada do tamanho dos barcos

(f) Características especiais

Incentivamos a inclusão de características especiais no jogo:

- Radar, com as tentativas anteriores do jogador. Tal como no jogo de tabuleiro, isto deverá ajudar os jogadores a relembrar os seus movimentos, evitando

disparar na mesma célula.

- Games's Hit Ratio: Fornecer uma estatística simples, com os tiros certeiros e os falhados de cada jogador.
- Navios restantes: dar informação sobre o número de navios restantes do adversário.
- Indicar o número da ronda
- Desenvolver uma interface gráfica de utilizador utilizando SWING, HTML ou outra ferramenta.

(g) Exemplo de configuração Apache XML

Abaixo encontra-se um pequeno exemplo da utilização de ficheiros XML com a biblioteca Apache Commons Configuration.

```
try {
     //Load the configuration from the xml file
     XMLConfiguration config = new XMLConfiguration("config.xml");

     //Retrieve a simple configuration: (int)
     int rows = config.getInt("board.rows");

     //Retrieve multiple nested configuration.
     // A: Using the configurationsAt function.
     List<HierarchicalConfiguration> shipCnfs = config.configurationsAt("ships.ship");
     for(HierarchicalConfiguration shipC : shipCnfs){
             String name = shipC.getString("name");
     }

     // B: Assembling manually the properties key.
     int maxIndex = config.getMaxIndex("ships.ship");
     for(int i = 0; i<= maxIndex; i++){
             String key = String.format("ships.ship(%d).", i);
             String name = config.getString(key+"name");
     }
} catch (ConfigurationException e) {}
```

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!