Design Pattern em Java

Memento

Memento

- O *design pattern* ***Memento*** permite guardar o estado de um objeto para poder ser reposto mais tarde
  - Corresponde à criação de *snapshots* do estado do objeto e gestão dos mesmos
  - Mantendo um histórico dos *snapshots* do estado, será possível implementar operações de *undo* e, eventualmente, de *redo*

Memento

- Entidades que constituem o padrão
  - *Originator*
    - O objeto que possui os dados/estado que se pretende salvaguardar
  - *Memento*
    - *Snapshot* da informação/estado
    - Deve guardar a informação de forma a que apenas o *Originator* consiga aceder à informação
  - *CareTaker*
    - Entidade gestora dos *mementos*
      - É a partir desta entidade que são despoletadas as operações de criação de *mementos*
    - Mantém um histórico dos *mementos* e disponibiliza a funcionalidade de *undo*
      - Poderá possuir um histórico de *undos* para possibilitar operações de *redo*

Originator

- Deve possuir métodos que permitam
  - salvaguardar a situação atual, retornando um memento (*snapshot*) com toda a informação necessária
  - restaurar um *memento* guardado anteriormente, repondo a situação existente no momento em que o *snapshot* foi gerado

- A classe representativa do objeto deverá implementar a interface `IOriginator`

```
public interface IOriginator {
    IMemento save();
    void restore(IMemento memento);
}
```

Memento

- Objetos que permitem representar/guardar os *snapshots* de um determinado objeto/situação

```
public interface IMemento {
    default Object getSnapshot() { return null; }
}
```

  - A forma de armazenar o *snapshot* depende da informação do *Originator*
    - Não devem ser guardadas referências para os objetos originais
    - Devem ser guardadas cópias desses objetos

Memento

- Podem ser implementados de diversas formas, salientando-se duas formas
  - classe *nested* (privada) do *Originator*
    - Desta forma os dados no *memento* podem ser armazenados em variáveis privadas da classe *nested*, acessíveis pela classe *Originator*, mas não acessíveis de entidades externas
  - classe, eventualmente externa ao *Originator*, que armazena os "dados em bruto"
    - normalmente recorrendo a *array* de *bytes* e ao processo de serialização
    - método adequado quando estamos na presença de uma estrutura de dados mais complexa no *Originator*

Memento com classe *nested*

```
class MyOriginator implements IOriginator {
   MyObject data;
   // ...
   private static class MyMemento implements IMemento {
      MyObject data;

      MyMemento(MyOriginator base) {
         this.data = base.data.clone(); // or similar
      }
   }

   @Override
   public IMemento save() { return new MyMemento(this); }

   @Override
   public void restore(IMemento memento) {
      if (memento instanceof MyMemento m)
         data = m.data;
   }
}
```

Memento com serialização

```
class MyOriginator implements
                    Serializable, IOriginator {
   MyObject data;
   // ...
   @Override
   public IMemento save() {
      return new Memento(this);
   }

   @Override
   public void restore(IMemento memento) {
      Object obj = memento.getSnapshot();
      if (obj instanceof MyOriginator m)
         data = m.data;
   }
}
```

```
class Memento implements IMemento {
    byte[] snapshot;

    public Memento(Object obj) {
        try (ByteArrayOutputStream baos =
                              new ByteArrayOutputStream();
             ObjectOutputStream oos =
                          new ObjectOutputStream(baos)) {
            oos.writeObject(obj);
            snapshot = baos.toByteArray();
        } catch (Exception e) { snapshot = null; }
    }

    @Override
    public Object getSnapshot() {
        if (snapshot == null) return null;
        try (ByteArrayInputStream bais =
                    new ByteArrayInputStream(snapshot);
             ObjectInputStream ois =
                         new ObjectInputStream(bais)) {
            return ois.readObject();
        } catch (Exception e) { return null; }
    }
}
```

CareTaker

```
public class CareTaker {
    IOriginator originator;
    Deque<IMemento> history;
    Deque<IMemento> redoHist;

    public CareTaker(IOriginator originator) {
        this.originator = originator;
        history = new ArrayDeque<>();
        redoHist= new ArrayDeque<>();
    }

    public void save() {
        redoHist.clear();
        history.push(originator.save());
    }

    public void undo() {
        if (history.isEmpty())
            return;
        redoHist.push(originator.save());
        originator.restore(history.pop());
    }
//...... =>
```

```
// =>.......
    public void redo() {
        if (redoHist.isEmpty())
            return;
        history.push(originator.save());
        originator.restore(redoHist.pop());
    }

    public void reset() {
        history.clear();
        redoHist.clear();
    }

    public boolean hasUndo() {
        return !history.isEmpty();
    }

    public boolean hasRedo() {
        return !redoHist.isEmpty();
    }
}
```

CareTaker

- Utilização
  - Criar uma instância do `CareTaker`, associando-a ao *Originator*
  - Sempre que existir uma ação que altera os dados/estado do *Originator*, deve ser chamado o método `save()` do `CareTaker`
  - Quando for necessário realizar uma operação de *undo*, chamar o método `undo()`
  - Quando for necessário realizar uma operação de *redo*, chamar o método `redo()`
  - Se for executada uma ação que altera os dados/estado do *Originator*, mas para a qual não se pretende criar um *snapshot*, é aconselhado chamar o método `reset()` do `CareTaker` para limpar o histórico de *undos* e *redos*.

Exercício – Quatro-em-linha

- Aplique o padrão memento ao jogo *Quatro-em-Linha* de modo a permitir ações de *undo* e *redo*
- No Nónio está disponível uma versão básica do jogo, sem as ações referidas, que poderá utilizar neste exercício
  - "*Aula 16 (P) - Base para o exercício da aula*"
- Crie uma classe *Facade*, `FourInARowManager`, que permita esconder as opções de implementação interna do jogo e do padrão Memento