GROEIPROJECT
MODULE 5: DESIGN PATTERNS

1. Voorbereiding

1.1. Open het groeiproject en maak daarin een vijfde module: `5_patterns`

1.2. Kopieer van module `1_herhaling` ENKEL de package `model` (met o.a. de oorspronkelijke basis- en multiklasse) naar de nieuwe module.

2. Observer pattern

2.1. Extraheer een interface uit je multiklasse, waarin alle methoden zitten. In ons voorbeeld extraheren we uit `Dictators` de interface `DictatorsInterface`. Ga daartoe als volgt te werk:

- Positioneer de cursos in het project-venster op je multiklasse, doe rechtermuisklik en kies de optie: "*Refactor* > *Extract* > *Interface*".
- Kies als naam voor de interface: `XxxInterface` (in ons voorbeeld: `DictatorsInterface`)
- Selecteer ALLE methoden als members voor de nieuwe interface.
- Zorg dat je multiklasse de nieuwe interface implementeert.

2.2. We willen de multiklasse **Observable** maken. Bij elke wijziging moeten alle observers verwittigd worden. Maar om de een of andere reden mogen we de multiklasse zelf niet aanpassen, maar moeten we het **Observer** pattern uitvoeren in een nieuwe klasse.

- Maak een nieuwe package `patterns` en daarin een nieuwe klasse die je `ObservableXxx` noemt (In ons voorbeeld: `ObservableDictators`)
- Erf over van `Observable` en implementeer ook de interface die je pas gemaakt hebt (zie 2.1)
- Voorzie een attribuut van je multiklasse en laat het als parameter via de constructor binnenkomen
- Pas in elke methode **delegatie** toe en delegeer alles naar de multiklasse
- Verwittig de observers telkens wanneer de data van de multiklasse gewijzigd worden.

2.3. We hebben ook minstens één **Observer** nodig, die op de hoogte wordt gebracht bij wijzigingen.

- Maak daarom in de package `patterns` een nieuwe klasse die je `XxxObserver` noemt. (In ons voorbeeld: `DictatorsObserver`)
- Implementeer hier de interface `Observer`
- Druk de informatie die je van de observable ontvangt hier af. Bijvoorbeeld:
  `Observer meldt: Toegevoegd: … (toString van toegevoegd Object) …`
  OF:
  `Observer meldt: Verwijderd: … (toString van toegevoegd Object) …`

2.4. Maak een klasse `Demo_5` om dit uit te proberen:

- Instantieer de observable-klasse en de observer-klasse en vergeet ze niet te koppelen!
- Voeg toe of verwijder data uit de observable-klasse en controleer of de observer-klasse daar gepast op reageert.

2.5. Mogelijke afdruk:

```
Observer meldt: Toegevoegd: Francisco Franco (°1892) Spanje   regime: Militaire dictatuur
5,0 mln doden
Observer meldt: Verwijderd: Francisco Franco (°1892) Spanje   regime: Militaire dictatuur
5,0 mln doden
```

3. Static Factory pattern

3.1. We willen dat het creëren van nieuwe basisobjecten door een **Factory** geregeld wordt. Dus maak ALLE constructors van de basisklasse (in ons geval: `Dictator`) package-private.

3.2. Maak in de package model een nieuwe klasse die je `XxxFactory` noemt (in ons geval: `DictatorFactory`):

- Blokkeer hier de constructor door hem private te maken.
- Voorzie een static factory-methode `newEmptyDictator` die de defaultconstructor van de basisklasse oproept en dus een "leeg" basisobject retourneert.
- Voorzie een static factory-methode `newFilledDictator` die de andere constructor van de basisklasse oproept. Alle benodigde gegevens komen hier via parameters binnen en worden doorgegeven. Een "gevuld" basisobject wordt geretourneerd.
- Voorzie een static factory-methode `newRandomDictator` die geen parameters heeft, maar alle gegevens at random genereert en vervolgens via parameters doorgeeft aan de constructor van de basisklasse.
  Probeer deze waarden zo realistisch mogelijk te maken:
  - Datumvelden correct binnen een bepaald tijdsinterval (bijvoorbeeld tussen 1900 en 2000)
  - Getalwaarden correct binnen een bepaald bereik (bijvoorbeeld percentages tussen 0.0 en 1.0)
  - Strings realistisch: een bepaald aantal woorden, maxlengte van een woord, beginhoofdletter of niet. Maak hiervoor liefst een hulpmethode `generateString`:

```
private static String generateString(int maxWordLength, int wordCount,
        boolean camelCase)
```

  Om een realistische string samen te stellen, hanteer je volgende simplistische regel: 1 kans op 3 voor een klinker, 2 kansen op 3 voor een medeklinker.

3.3. Gebruik de klasse `Demo_5` om daar de factory-methoden uit te testen.

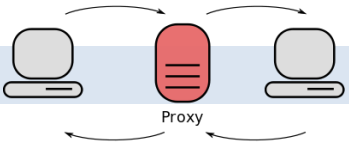
3.4. Voorbeeld van afdruk:

```
Empty dictator:
Anoniem                (°   0) Ongekend        regime: Ongekend              0,0 mln doden

30 random dictators:
1) Hsxviuy Ugtkpuhnggj    (°1909) Raagswhq        regime: nnyaexaooidlmj         3,9 mln doden
2) Xdadii Aajrgbu         (°1906) Tiudl           regime: sxiufetwljcqm         28,9 mln doden
3) Odmqflpq Mdexumuen     (°1933) Ueuu            regime: czmpulvfexkdn          6,4 mln doden
4) Dneyhb Hbuniz          (°1983) Mhqnqfd         regime: oefasuxfei            20,0 mln doden
5) Kbxfhhriy Upesxdylm    (°1995) Edevgyxxsee     regime: nrkr                  10,0 mln doden

… enz…
```

4. Proxy pattern



4.1. We willen een **Protection-proxy** maken voor onze multiklasse, die de toegang tot de multiklasse controleert en afschermt.

- Maak in de package `patterns` een nieuwe klasse die je `UnmodifiableXxx` noemt (in ons voorbeeld: `UnmodifiableDictators`)
- Laat deze klasse je multiklasse interface implementeren.
- Via de constructor komt het multi-object binnen dat moet beschermd worden.
- Alle methoden die iets willen wijzigen aan de data van de multiklasse worden afgeblokt. Doe daar een throw van een `UnsupportedOperationException`
- Bij alle methoden die een List of een andere collection retourneren bescherm je die door gebruik te maken van `Collections.unmodifiableXxx`-methoden

4.2. Test het Proxy pattern uit in de klasse `Demo_5`.
Controleer oa of de `UnsupportedOperationException` gegooid wordt bij wijzigingen aan de proxy of de gesorteerde collectie.