06 - Singleton, Visitor

Credits: Askar Gafurov

Uzitocne linky:

- Slajdy z prednasky
- Starsie poznamky: singleton, visitor
- Head First Design Patterns, chapter 5 (SIngleton)

Singleton

Pouzivame, ked chceme zaistit unikatnost instancie nejakej triedy.

Zakladna struktura:

```
public class Singleton {
    private static Singleton uniqueInstance = null;

    private Singleton() {
        // ...
    }

    public static Singleton getUniqueInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }
}
```

Pouzitie:

```
class Demo {
    void demo() {
        Singleton uniqueInstance = Singleton.getUniqueInstance();
        Singleton secondReference = Singleton.getUniqueInstance();
        assert (uniqueInstance == secondReference);
    }
}
```

Visitor

Pouzivame, ked chceme implementovat dodatocnu funkcionalitu pre specificke triedy pod spolocnym rozhranim, alebo v komplexnej strukture (napr. Composite). Vsimnite si, ze tymto

sposobom vieme rozlisovat objekty podla ich typov aj v pripade, ze su schovane pod nejake vseobecnejsie rozhranie.

Zakladna struktura:

```
public interface Component {
    // some functionality ...
    <T> T accept(ComponentVisitor<T> visitor);
}

public class ConcreteComponentA implements Component {
    // to allow visitor to access inner fields and methods, we need
    // to declare their access level as “package-level”, not “private”
    // this is achieved by omitting the access modifier
    // the visitor then should be added to the same package as Component

    int someField; // sic! no access modifier
    ArrayList<Object> anotherField;

    @Override
    public <T> T accept(ComponentVisitor<T> visitor) {
        return visitor.visit(this);
    }
}

public class ConcreteComponentB implements Component {
    @Override
    public <T> T accept(ComponentVisitor<T> visitor) {
        return visitor.visit(this);
    }
}

public class ConcreteComponentC implements Component {
    @Override
    public <T> T accept(ComponentVisitor<T> visitor) {
        return visitor.visit(this);
    }
}

public interface ComponentVisitor<T> {
    T visit(ConcreteComponentA component);
    T visit(ConcreteComponentB component);
    T visit(ConcreteComponentC component);
}

public class ConcreteVisitorX implements ComponentVisitor<Integer> {
    @Override
    public Integer visit(ConcreteComponentA component) {
        // do stuff
    }
```

```
    @Override
    public Integer visit(ConcreteComponentB component) {
        // do stuff
    }

    @Override
    public Integer visit(ConcreteComponentC component) {
        // do stuff
    }
}

// type `Void` is used when we don't want to return anything
public class ConcreteVisitorY implements ComponentVisitor<Void> {
    @Override
    public Void visit(ConcreteComponentA component) {
        // do stuff
        return null; // we need to return an object of type Void
    }

    @Override
    public Void visit(ConcreteComponentB component) {
        // do stuff
        return null;
    }

    @Override
    public Void visit(ConcreteComponentC component) {
        // do stuff
        return null;
    }
}
```

Pouzitie:

```
class Demo {
    void demo() {
        Component ca = new ConcreteComponentA();
        Component cb = new ConcreteComponentB();

        ComponentVisitor<Integer> v = new ConcreteVisitorX();
        final Integer result1 = ca.accept(v);
        final Integer result2 = cb.accept(v);

        ComponentVisitor<Void> v2 = new ConcreteVisitorY();
        ca.accept(v2);
        cb.accept(v2);
```

```
        // this may be shortened:
        ca.accept(new ConcreteVisitorY());
    }
}
```

Ulohy:

U: Implementujte triedy `Cat`, `Dog` pod spolocnym rozrhanim `Animal`. Pridajte vizitor `Sound` pre tieto dve triedy, ktory bude vypisovat na standardny vystup retazec so zvukom, ktore tieto zvieratka produkuju (napr. “Meow” a “Woof”). Pridajte taktiez visitor `Speed`, ktory vrati ocakavanu maximalnu rychlost zvieratka (psy - 32 km/h, macky - 48 km/h).

U: Pridajte stromovej strukture (vid Kostra implementacie stromu) pomocou navrhoveho vzoru Visitor operaciu `Size`, ktora vrati pocet vrcholov v danom strome.

U: Pridajte stromovej strukture (vid Kostra implementacie stromu) pomocou navrhoveho vzoru Visitor operaciu `Print`, ktora vypise stromovu strukturu na standardny vystup.

U*: Pridajte stromovej strukture pomocou navrhoveho vzoru Visitor operaciu `InOrder`, ktora vrati objekt s rozhranim `Iterable<TreeNode>`, ktory umoznuje preiterovat cez vsetky vrcholy stromu v in-order poradi.
Priklad pouzitia:

```
for (TreeNode n: treeNode.accept(new InOrder())) {
      System.out.println(n.getId());
}
```

U: Pridajte stromovej strukture pomocou navrhoveho vzoru Visitor operaciu `RemoveLeftmostLeaf`, ktora odstrani najlavejsi objekt typu `LeafNode` zo stromu.

U: Pridajte stromovej strukture pomocou navrhoveho vzoru Visitor operaciu `RemoveLeafById`, ktora odstrani listy stromu s danym identifikatorom.

U*: Pridajte stromovej strukture pomococu navrhoveho vzoru Visitor operaciu `ConvertToLeaves`, ktora premeni vsetky vrcholy typu `InternalNode` bez deti na vrcholy typu `LeafNode`. Da sa to implementovat pomocou visitora s rozhranim `TreeNodeVisitor<Void>`?

Pomocky:

Kostra implementacie stromu:

```
public interface TreeNode {
    int getId();
}
```

```
public class LeafNode implements TreeNode {
    private final int id;

    public LeafNode(int id) {
        this.id = id;
    }
    @Override
    public int getId() {
        return id;
    }
}

public class InternalNode implements TreeNode {
    private final TreeNode left, right;
    private final int id;

    public InternalNode(int id, TreeNode left, TreeNode right) {
        this.id = id;
        this.left = left;
        this.right = right;
    }

    @Override
    public int getId() {
        return id;
    }

    public TreeNode getLeftChild() {
        return left;
    }

    public TreeNode getRightChild() {
        return right;
    }
}
```