# Analýza a návrh informačných systémov II 8

## objektovo orientovaný návrh

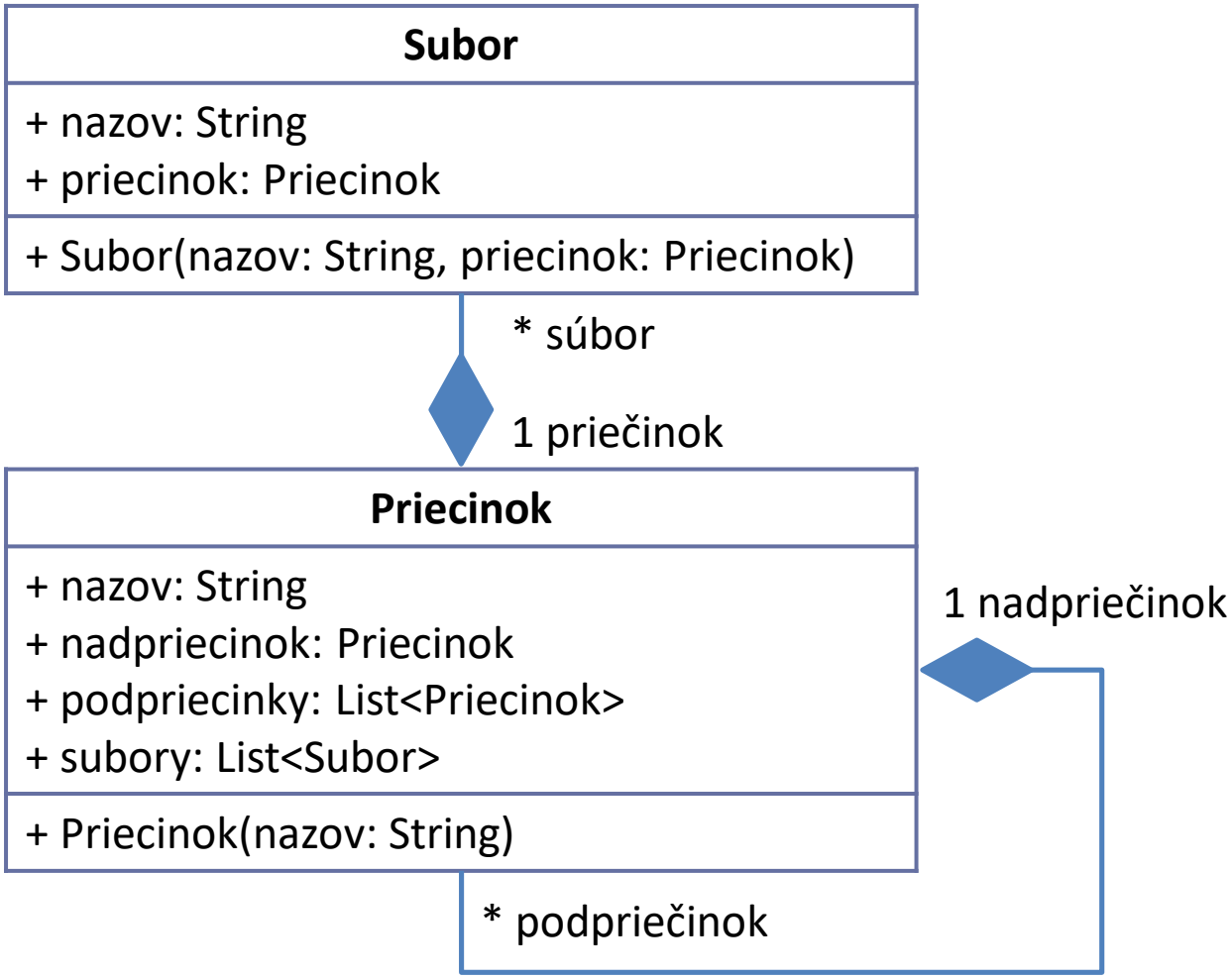
Peter Bednár

# Štrukturálne vzory II

# Composite

- Základný vzor objektového programovania, kedy sa zložitejší objekt skladá z častí rôznych typov.
- Príklady:
  - `Univerzita` – `Fakulta`
  - `Priecinok` – `Subor`
  - `Automobil` – `Motor`, `Prevodovka`, `Klimatizacia`, ...
- Objekt môže byť aj rekurzívne zložený z objektov toho istého typu
  - Napr. `Priecinok` môže mať aj podadresáre typu `Priecinok`

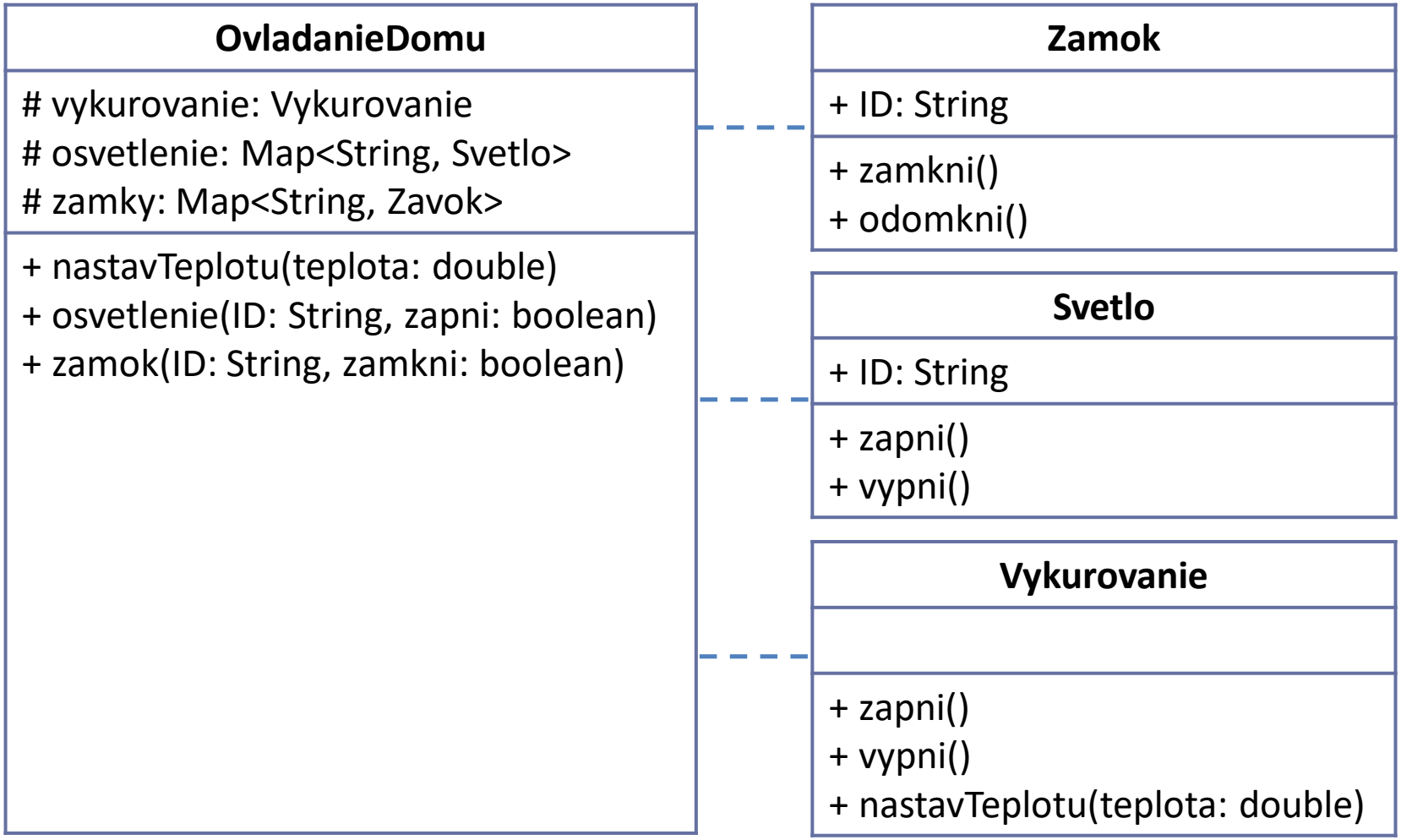
# Composite – príklad



# Facade

- Cieľom je poskytnúť jednotné zjednodušené rozhranie, ktoré spája viacero zložitejších rozhraní
- Implementácia nepridáva priamo novú funkčnosť, iba deleguje volanie na spájané objekty
- Na rozdiel od kompozície, cieľom nie je definovanie nového samostatného objektu, ale iba o implementácia rozhrania

# Facade – príklad (1)



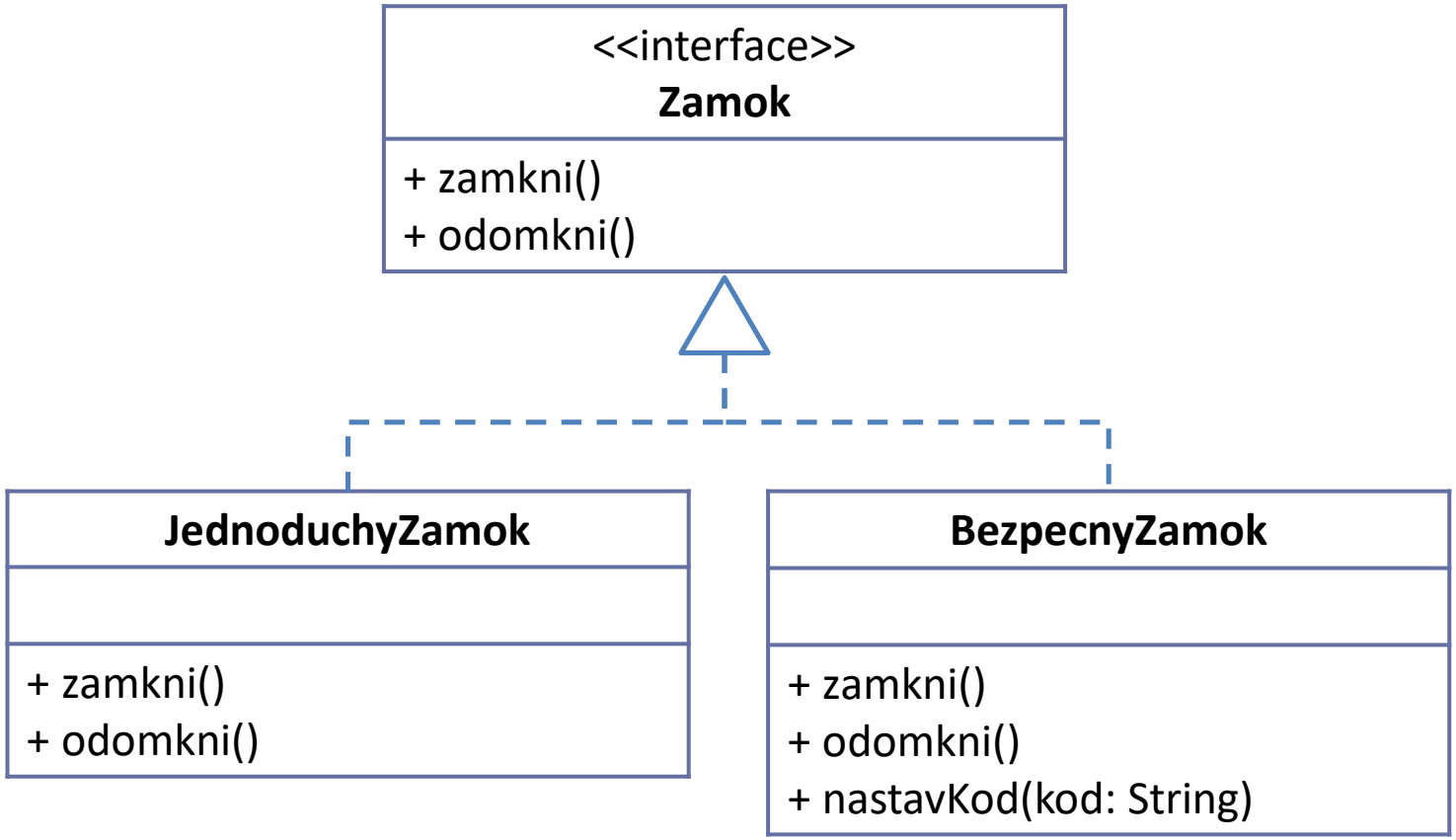
# Facade – príklad (2)

```
public class OvladanieDomu {
    protected Vykurovanie vykurovanie;
    protected Map<String, Osvetlenie> osvetletnie;
    ...
    public void nastavTeplotu(double teplota) {
        vykurovanie.zapni();
        vykurovanie.nastavTeplotu(teplota);
    }

    public void osvetlenie(String ID, boolean zapni) {
        if (zapni) {
            osvetlenie.get(ID).zapni();
        } else {
            osvetlenie.get(ID).vypni();
        }
    }
    ...
}
```

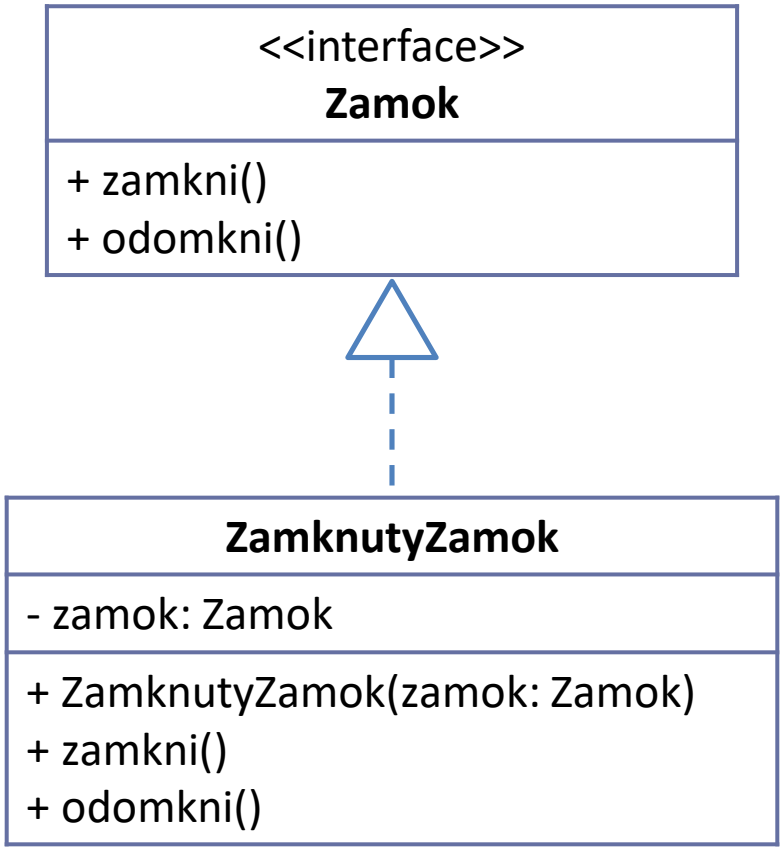
# Decorator

- Chceme upraviť funkčnosť existujúceho rozhrania dynamicky počas behu programu
- Na rozdiel od vzoru Adapter, Decorator poskytuje to isté rozhranie ako pôvodný objekt
  - Pre nezmenené metódy deleguje volanie na pôvodný objekt

# Decorator – príklad (1)



# Decorator – príklad (2)



# Decorator – príklad (3)

```
public class ZamknutyZamok implements Zamok {

    private Zamok zamok;

    public Zamok(Zamok zamok) {
        this.zamok = zamok;
    }

    @Override
    public void zamkni() {
        this.zamok.zamkni();
    }

    @Override
    public void odomkni() {
        throw new UnsupportedOperationException(
            "nie je možné odomknúť");
    }
}
```

Decorator implementuje pôvodné rozhranie

Nezmenené volanie sa deleguje na pôvodný objekt

# Nemenné kolekcie

- Príklad vzoru Decorator, ktorý vytvára nemenný pohľad na Java kolekcie (zoznam, množinu, mapu)

- Vytvárajú sa pomocou statických metód v triede `java.util.Collections` (vzor Factory)
  - `.unmodifiableList(zoznam)`
  - `.unmodifiableSet(mnozina)`
  - `.unmodifiableSortedSet(mnozina)`
  - `.unmodifiableMap(mapa)`
  - `.unmodifiableSortedMap(mapa)`

# Nemenné kolekcie – príklad

```
List<String> dni = new ArrayList<>();
dni.add("pondelok");
dni.add("utorok");
...
dni.add("nedeľa");

nemenneDni = Collections.unmodifiableList(dni);

nemenneDni.remove(1);
```

Všetky operácie ktoré menia zoznam vyvolajú výnimku UnsupportedOperationException

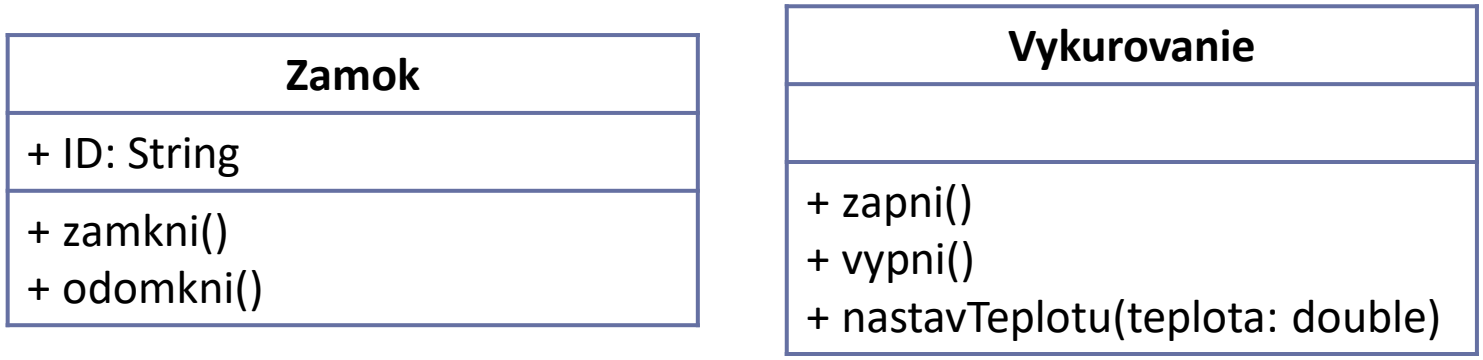
# Behaviorálne vzory

# Command

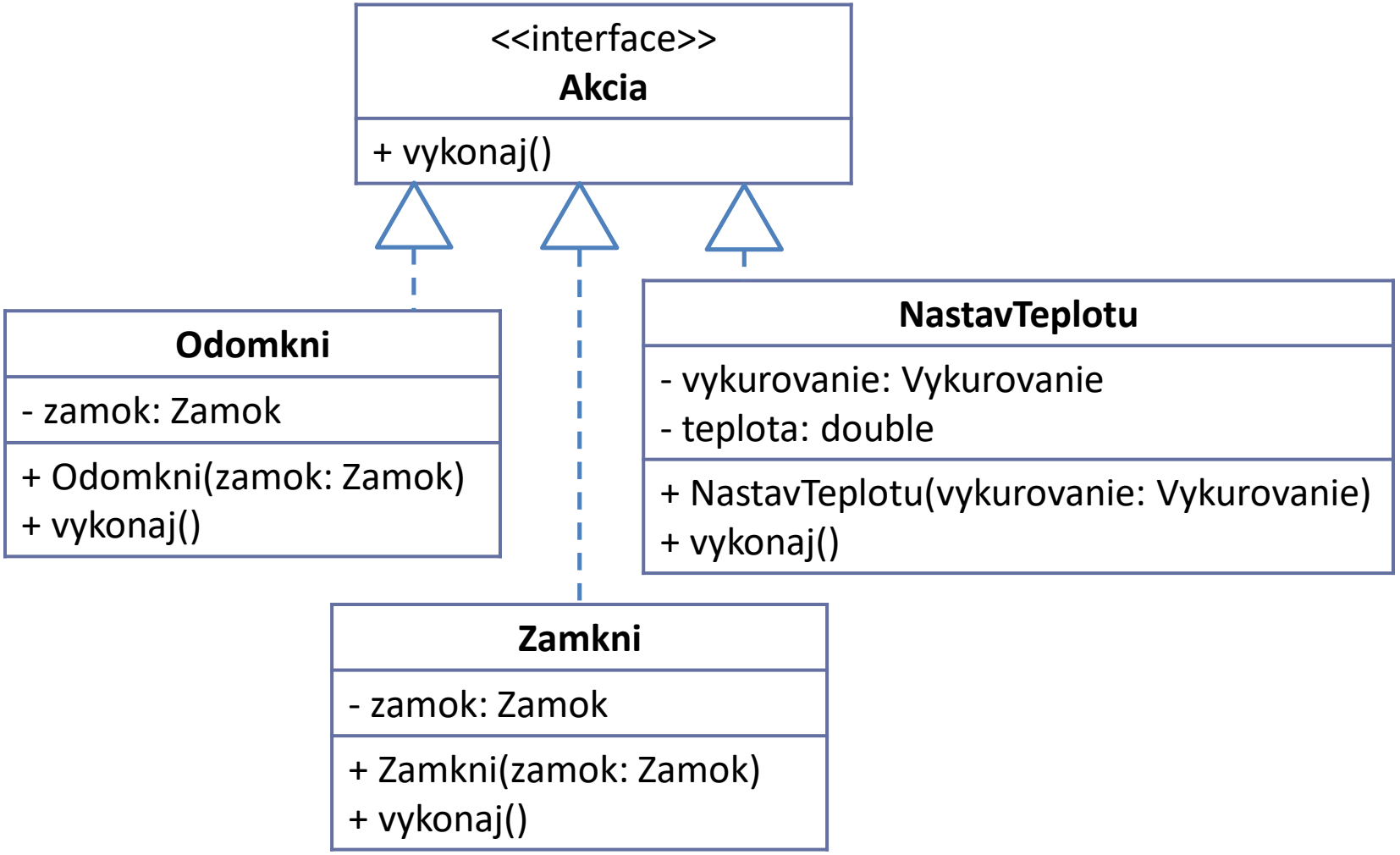
- Namiesto priameho volania nejakej metódy zabalíme volanie do objektu.
  - Objekt operácie bude uchovávať všetky argumenty potrebné na volanie metódy a odkaz nad ktorým objektom sa má operácia vykonať
  - Zabalenú operáciu môžeme vykonať aj neskôr (napr. ak dôjde k nejakej inej udalosti)

# Command – príklad (1)

- Triedy pre ktoré chceme zabaliť operácie – volania jednotlivých metód
- Chceme naprogramovať dynamické scenáre, ktoré si bude môcť používateľ meniť



# Command – príklad (2)



# Command – príklad (3)

```
public class Scenar {

    private Map<String, List<Akcia>> scenare = new HashMap<>();
    ...

    public void pridajAkciu(String scenar, Akcia akcia) {
        scenare.get(scenar).add(akcia);
    }

    public void vykonajScenar(String scenar) {
        for (Akcia akcia : scenare.get(scenar)) {
            akcia.vykonaj();
        }
    }
}
```

# Iterator

- Iterator je objekt, ktorý umožňuje pristupovať k prvkom iných objektov (kolekciám)
- V Jave je implementovaný ako objekt implementujúci rozhranie `java.util.Iterator<E>`
- Metódy, ktoré je potrebné implementovať:
  - `boolean hasNext()` – vráti `true` ak existuje nasledujúci prvok, na konci iterácie vráti `false`
  - `E next()` – vráti aktuálny prvok a posunie sa na ďalší

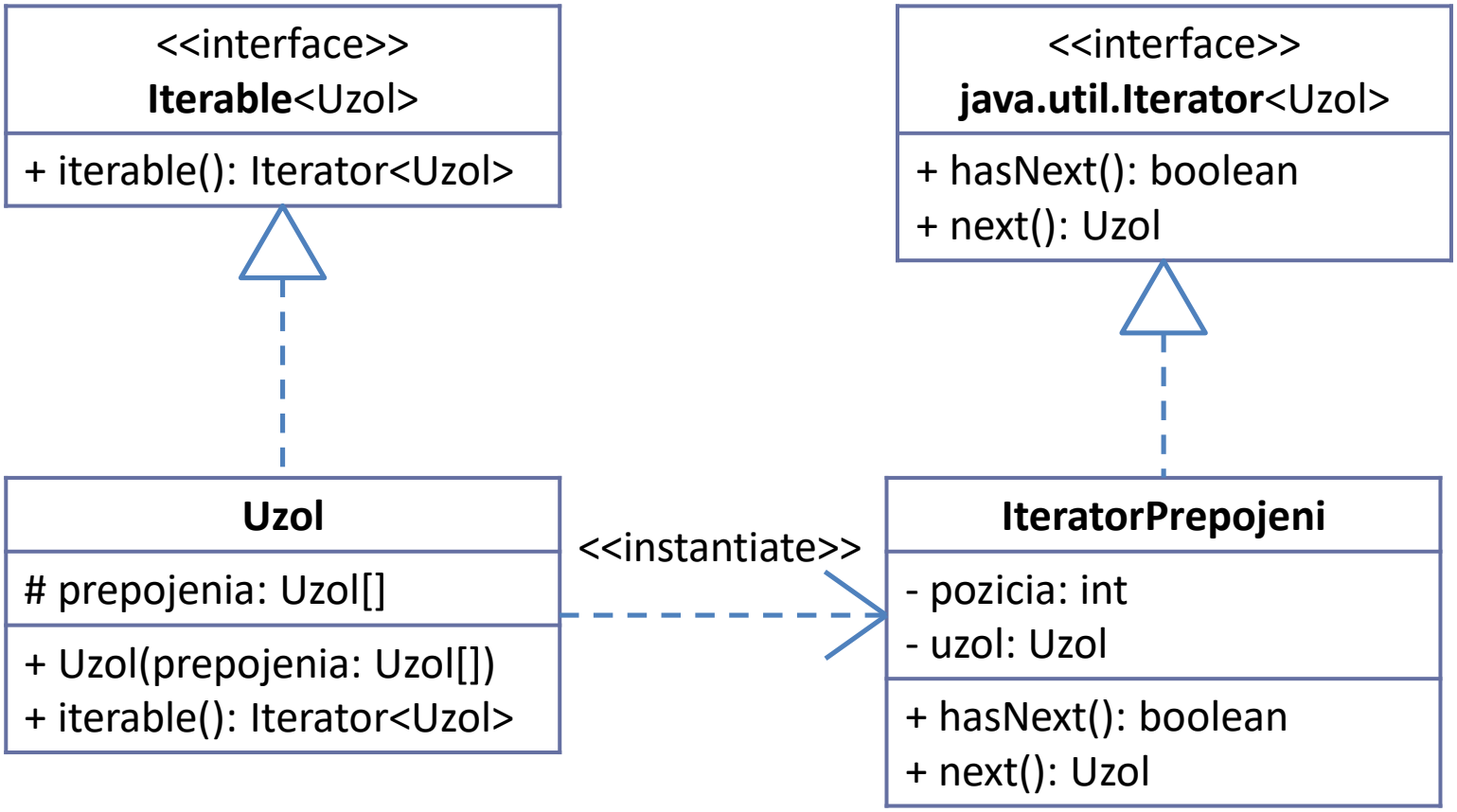
# Iterable

- Ak objekt poskytuje pre iterovanie svojich prvkov Iterator, mal by implementovať rozhranie `Iterable<E>`
  - `Iterator<E> iterator()`
- Takéto objekty potom môžeme jednoducho iterovať pomocou cyklu `for`:

```
for (E prvok : objekt) {
    ...
}
```

- Všetky štandardné kolekcie v Jave implementujú `Iterable`

# Iterator – príklad (1)



# Iterator – príklad (2)

```
public class IteratorPrepojeni implements Iterator<Uzol> {
    private int pozicia = 0;
    private Uzol uzol;

    public IteratorPrepojeni(Uzol Uzol) {
        this.uzol = uzol;
    }
    @Override
    public boolean hasNext() {
        return pozicia < uzol.prepojenia.length;
    }
    @Override
    public Uzol next() {
        Uzol aktualny = uzol.prepojenia[pozicia];
        pozicia++;
        return aktualny;
    }
}
```

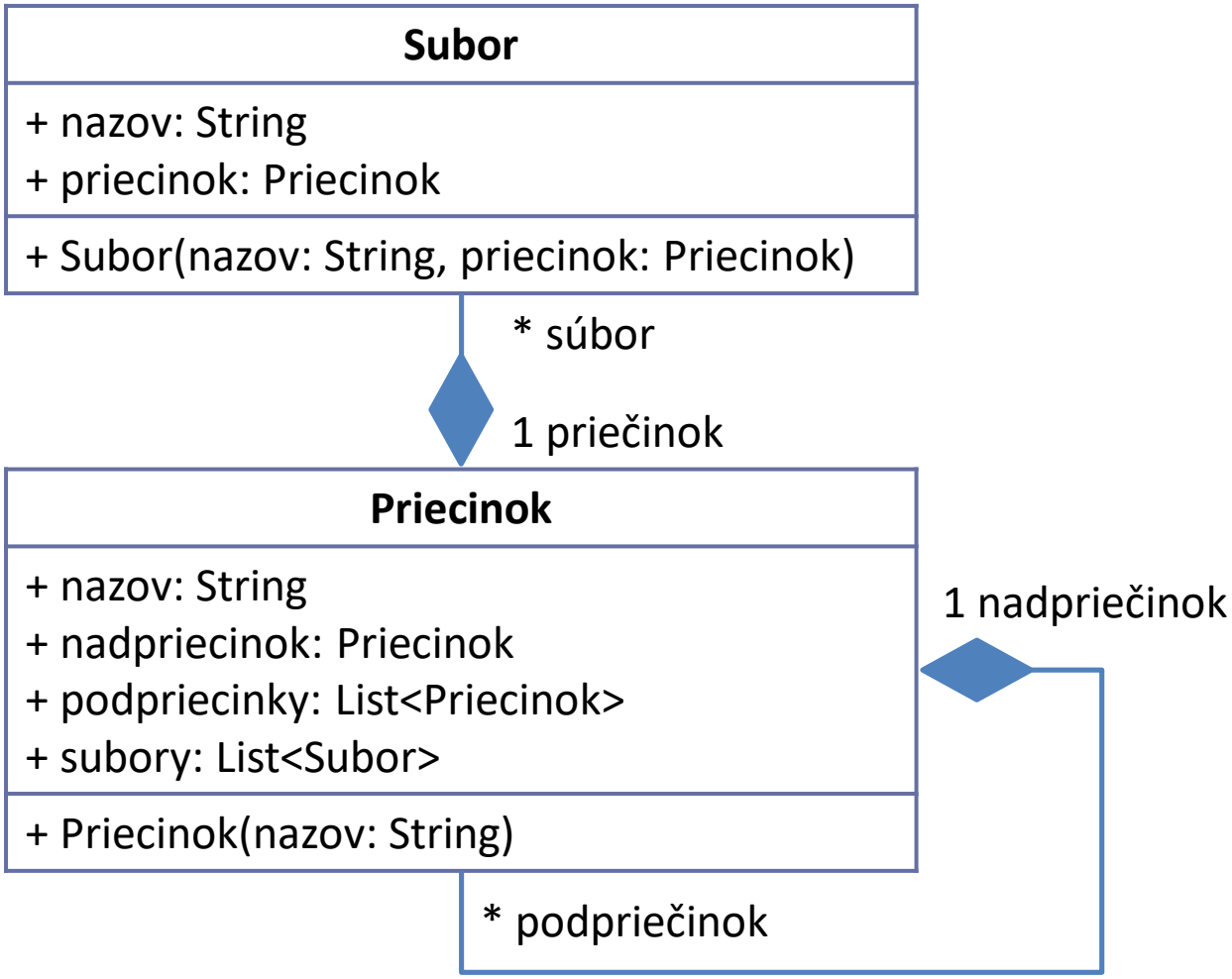
# Iterator – príklad (3)

```
public class Uzol implements Iterable<Uzol> {
    protected Uzol[] prepojenia;

    public Uzol(Uzol[] prepojenia) {
        this.prepojenia = prepojenia;
    }
    @Override
    public Iterator<Uzol> iterator() {
        return new IteratorPrepojeni(this);
    }
}

...

for (Uzol prepojenie : uzol) {
    ...
}
```

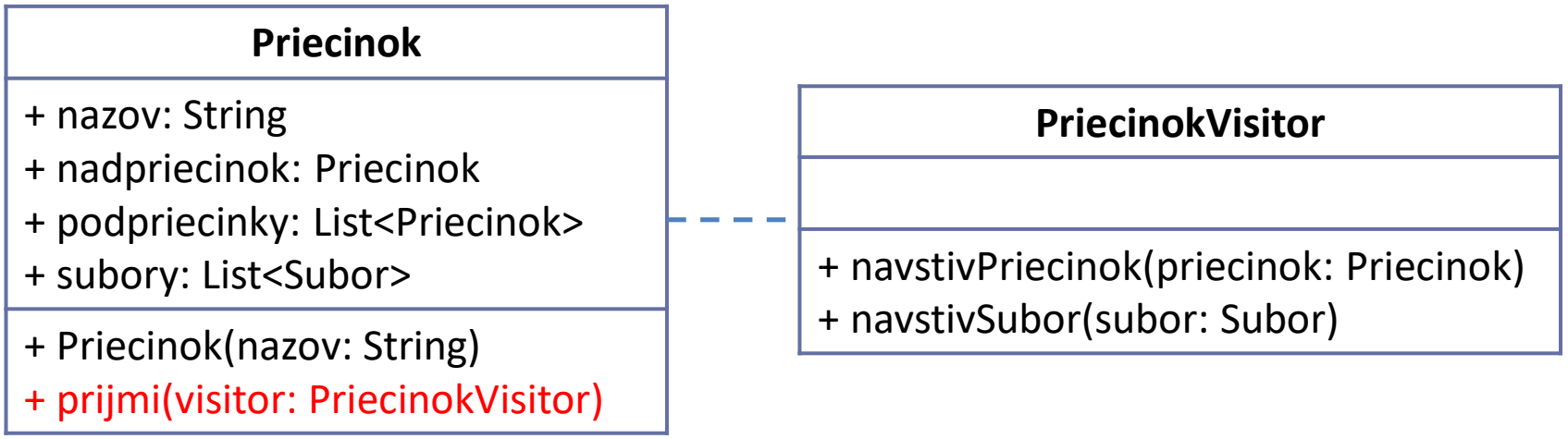
# Visitor

- Visitor podobne ako Iterator umožňuje pristupovať k prvkom zložených objektov
- Používa sa ak chceme pristupovať k zložitej štruktúre zloženej rekurzívne z rôznych typov
- Rozhranie Visitor definuje metódu "navštív(prvok)" pre jednotlivé typy prvkov štruktúry
  - Akcia ktorá sa má nad prvkom vykonať
- Objekt štruktúry definuje metódu "príjmi(Visitor)" ktorá prechádza prvkami a volá metódy "navštív"(prvok)"
  - Definuje, ktoré a v akom poradí sa prvky navštívia

# Visitor – príklad (1)



# Visitor – príklad (2)



# Visitor – príklad (3)

```
public class PriecinokVisitor {

    public void navstivPriecinok(Priecinok priecinok) {
        // akcia ktorá sa má vykonať pre priečinok
        System.out.println("priečinok: " + priecinok.nazov);
        System.out.println("počet súborov: " +
            pricinok.subory.size());
    }

    public void navstivSubor(Subor subor) {
        // akcia ktorá sa má vykonať pre súbor
        System.out.println("súbor: " + subor.nazov);
    }
}
```

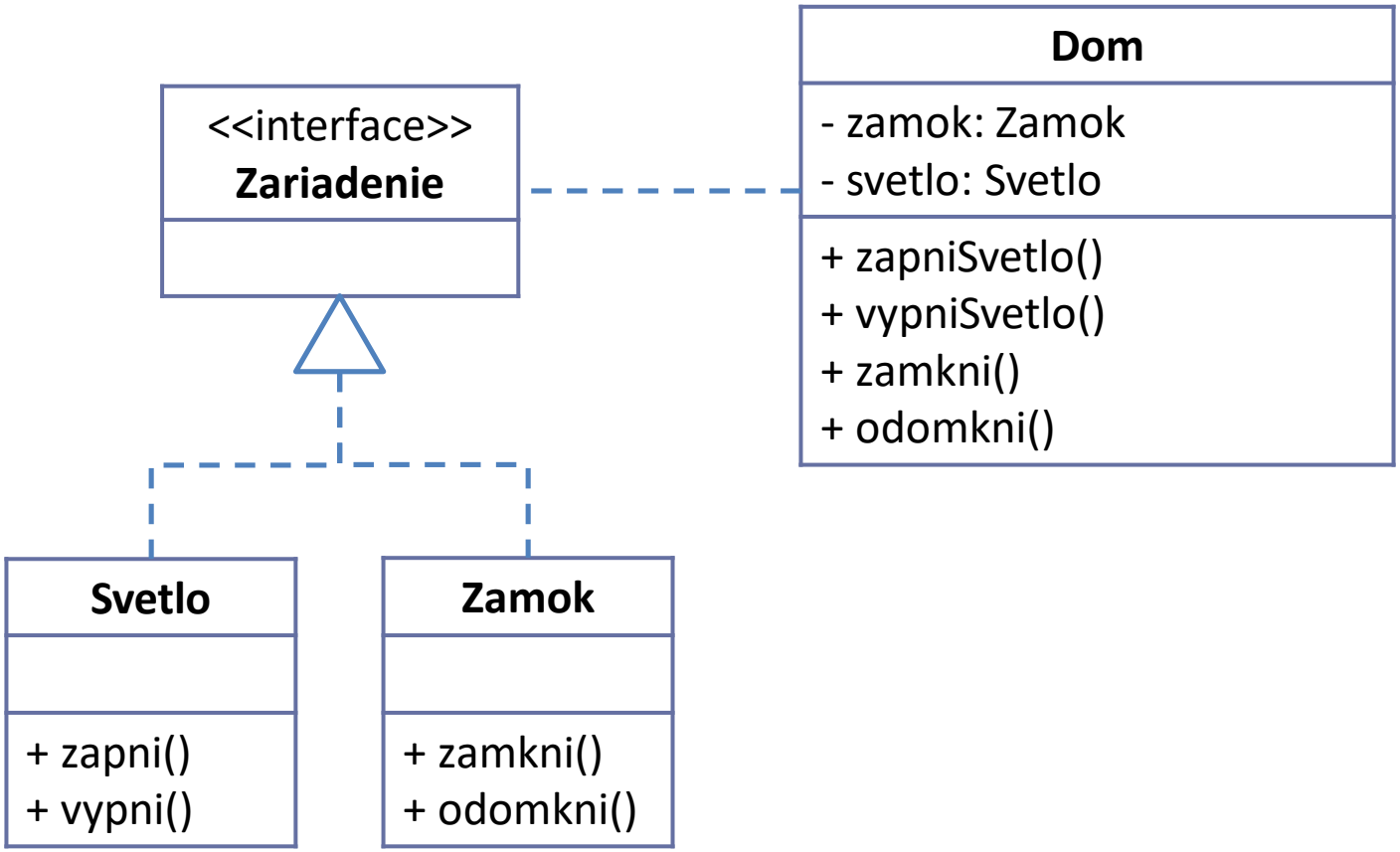
# Visitor – príklad (4)

```
public class Priecinok {
    ...
    public void prijmi(PriecinokVisitor visitor) {
        // najprv navštívime seba
        visitor.navstivPriecinok(this);
        // potom všetky súbory
        for (Subor subor : subory) {
            visitor.navstivSubor(subor);
        }
        // a nakoniec rekurzívne všetky podpriečinky
        for (Priecinok podpriecinok : podpriecinky) {
            priecinok.prijmi(visitor);
        }
    }
}
```

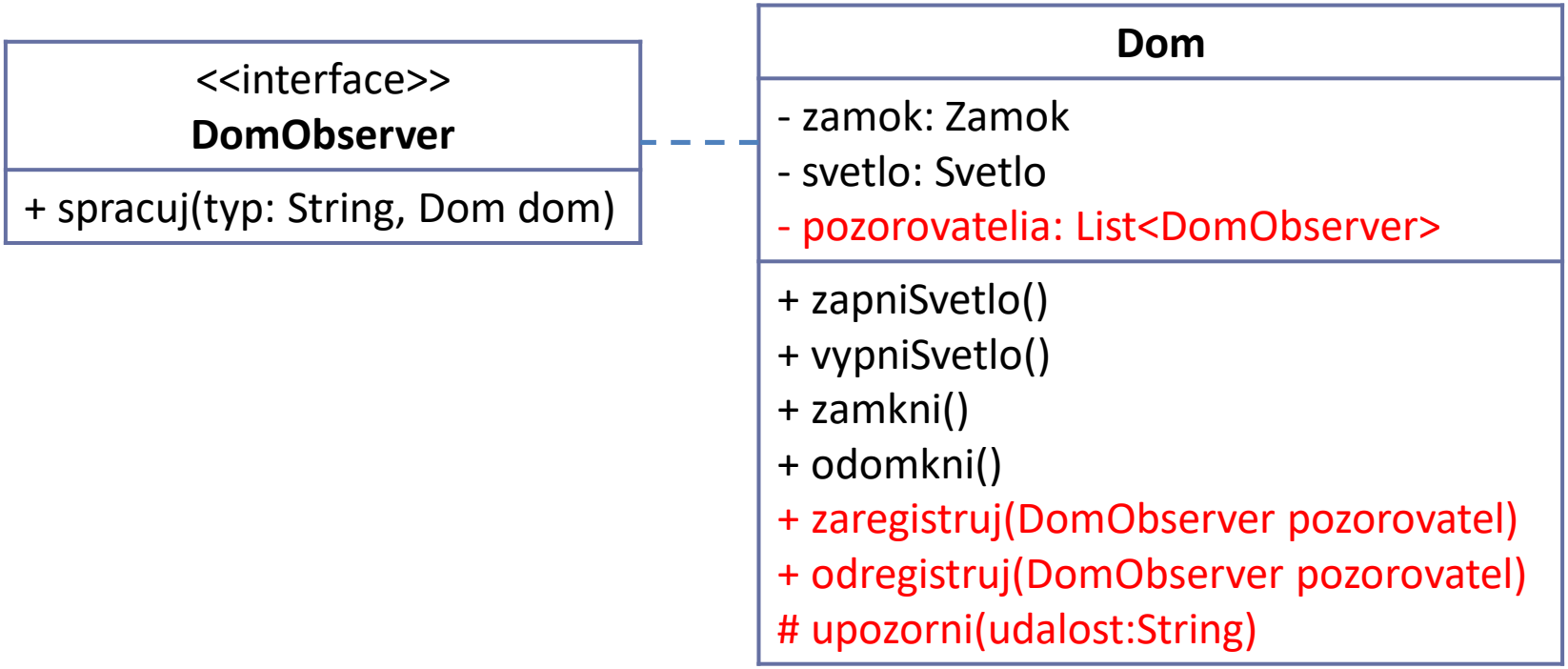
# Observe alebo Publish/subscribe

- Chceme upozorniť iný objekt (pozorovateľa), že sa zmenil stav pozorovaného objektu, alebo že došlo k nejakej udalosti
- Pozorovateľ sa môže dynamicky prihlásiť k odoberaniu udalostí počas behu programu
  - Jeden objekt môže byť pozorovaný viacerými pozorovateľmi
  - Jeden pozorovateľ môže pozorovať stav viacerých objektov

# Observe alebo Publish/subscribe – príklad (1)



# Observe alebo Publish/subscribe – príklad (2)



# Observe alebo Publish/subscribe – príklad (3)

```
public class Dom {
    ...
    private List<DomObserver> pozorovatelia = new LinkedList<>();

    public void zaregistruj(DomObserver pozorovatel) {
        pozorovatelia.add(pozorovatel);
    }

    public void odregistruj(DomObserver pozorovatel) {
        pozorovatelia.remove(pozorovatel);
    }

    protected void upozorni(String udalost) {
        // upozorníme pozorovateľov o udalosti
        for (DomObserver pozorovatel : pozorovatelia) {
            pozorovatel.spracuj(udalost, this);
        }
    }
}
```

# Observe alebo Publish/subscribe – príklad (4)

```
public class Dom {
    ...
    public void zapniSvetlo() {
        svetlo.zapni()
        upozorni("svetlo-zapnute");
    }

    public void vypniSvetlo() {
        svetlo.vypni()
        upozorni("svetlo-vypnute");
    }

    public void zamkni() {
        zamok.zamkni()
        upozorni("zamknute");
    }
    ...
}
```

# Observe alebo Publish/subscribe – príklad (5)

```
public class JednoduchyObserver implements DomObserver {
    @Override
    public void spracuj(String udalost, Dom dom) {
        System.out.println(udalost);
    }
}

public class PrichodOdchod implements DomObserver {
    @Override
    public void spracuj(String udalost, Dom dom) {
        if ("odomknute".equals(udalost)) {
            dom.zapniSvetlo();
        } else if ("zamknute".equals(udalost)) {
            dom.vypniSvetlo();
        }
    }
}
```

# Observe alebo Publish/subscribe – príklad (6)

```
Dom dom1 = new Dom();
Dom dom2 = new Dom();

DomObserver pozorovatel1 = new JednoduchyObserver();
DomObsercer pozorovatel2 = new PrichodOdchod();

dom1.zaregistruj(pozorovatel1);
dom1.zaregistruj(pozorovatel2);

// pre dom2 zaregistrujeme iba JednoduchyObserver
dom2.zaregistruj(pozorovatel1);
...

// ak už nechceme ovládanie po príchode a odchode
// odregistrujeme PrichodOdchod
dom1.odregistruj(pozorovatel2);
```

# Zhrnutie

- Štrukturálne vzory II
  - Composite
  - Facade
  - Decorator
- Behaviorálne vzory
  - Command
  - Iterator
  - Visitor
  - Observe alebo Publish/subscribe