**Osztályok közötti kapcsolatok.**

Absztrakciós folyamat:

**Általánosítás** (bottom-up megközelítés):

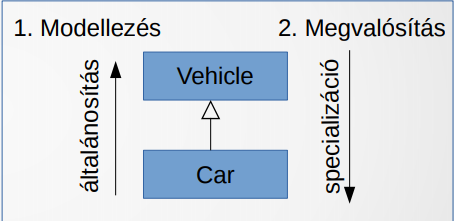
1. Tanulmányozzuk a valós objektumokat és megkeressük (kiemeljük) a probléma megoldása szempontjából fontos adataikat és viselkedésmintáikat.

2. Megkeressük azokat az objektumokat (objektumcsoportokat), amelyek azonos adatokkal és viselkedésmintákkal rendelkeznek és nevet adunk ennek az új csoportnak (absztrakt adattípus).

**Specializáció** (top-down megközelítés):

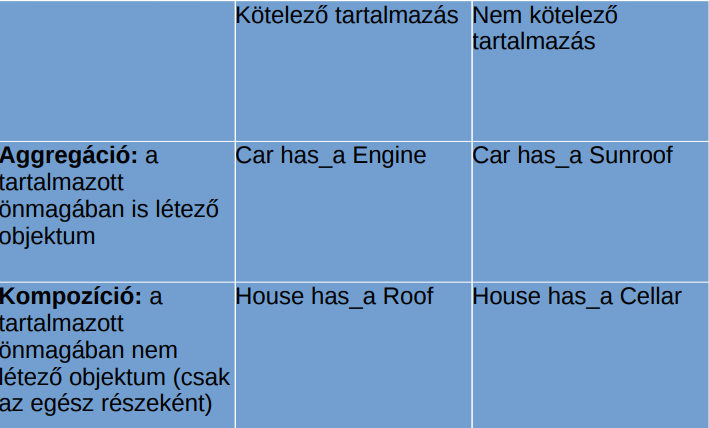
3. Az absztrakt adattípusokat leképezzük osztálydefinícióra / interfészre a legáltalánosabbtól kiindulva.

4. Objektumok létrehozása.

**Specializáció:** Pl: Car is\_a Vehicle

Megvalósítása: az általánosabb osztályból származtatással

Tartalmazás: A has\_a kapcsolat típusai:



**Aggregáció:**

Kötelező aggregáció: Car has\_a Engine, pl:

class Car { … private Engine engine; public Car(Engine engine) { this.engine = engine; }

/A motor adattag a konstruktor kötelező paramétere/

Nem kötelező aggregáció: Car has\_a Sunroof ,pl:

class Car { … private Sunroof sunroof; public Car() { this.sunroof = null; } …

/A napfénytető adattag nem kötelező paramétere a konstruktornak. Setter metódussal adhatunk neki értéket./

**Kompozíció:**

Kötelező kompozíció: House has\_a Roof ,pl:

class House {

private Roof roof;

public House(Roof roof) { this.roof = roof; … }

public static class Roof { private String type;

… public Roof() { … } } }

/A tető osztályszintű tagosztály./

Nem kötelező kompozíció: House has\_a Cellar pl:

class House { … public House() { … } class Cellar { private String walltype; … public Cellar() { … } } }

/A pince példányszintű tagosztály. Ha a háznak van pincéje, a ház példányosítása után hozható létre a pince példány. pl : House house = new House(); House.Cellar cellar = house.new Cellar();/

Tartalmazás kapcsolat számossága:

A lista adatszerkezet Java-ban generikus típusú. Interfészként van definiálva a java.util csomagban.

A lista elemeinek elérése az indexükkel lehetséges.

A tömbbel ellentétben előfordulhat egy listaelem ismétlődése, ahol igaz, hogy e1.equals(e2).

Az ArrayList generikus típusú, a java.util csomagban definiált osztály, amely implementálja a List interfészt. Az elemek hozzáadásával a mérete automatikusan nő. Iterálható.

Lista-műveletek:

Az ArrayList osztály fontosabb metódusai:

– add(E e), add(int index, E element): elem beszúrása

– clear(): lista kiürítése

– ensureCapacity(int minCapacity): lista min. hossza

– indexOf(Object o): listaelem indexe; -1 ha nincs a listában

– isEmpty(): igaz, ha a lista üres

– remove(int index), remove(Object o): listaelem törlése

– set(int index, E element): adott indexű elem értékének beállítása

– size(): lista mérete

– sort(Comparator c): a Comparator-ban megadott sorrend szerint rendezi a listát

– trimToSize(): lista hosszának (Capacity) aktuális méretre csökkentése

– toArray(): a lista tömbbé alakítása

Ha maximálni akarjuk a kapcsolatban résztvevő objektumok számát, tömb adattagot kell használni.

Specializáció vs tartalmazás:

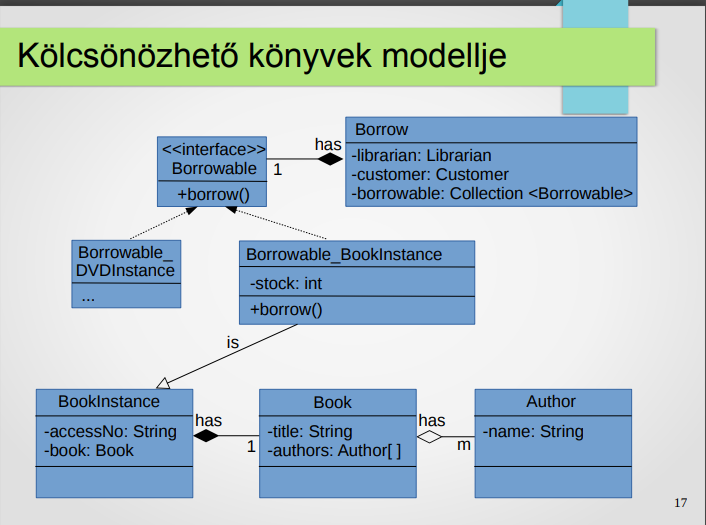
Mi legyen a kapcsolat a két osztály között?

Ha specializáció:

– BookInstance is\_a Book: megvalósítás származtatással

– BookInstance példányosításakor a könyv összes adatát meg kell adni (cím, szerző, …, leltári szám).

Ha tartalmazás:

– BookInstance has\_a Book: a könyv példánynak van könyv adattagja (aggregáció vagy kompozíció)

– Egy BookInstance létrehozásakor csak hivatkozunk egy már létező könyvre. → Kevesebb memória foglalás, nem lesznek felhasználói inputból származó anomáliák.

Az osztályokat (és azok kapcsolatát) úgy kell megtervezni, hogy biztosítsuk a polimorfikus viselkedést és a kód újrahasználhatóságát.

A viselkedés öröklése helyett olyan osztály példányai legyenek az adattagok, amelyek implementálják a kívánt viselkedést.