Swedish store (C#)

Óbudai Egyetem, Programozás 2
Mérnök Informatikus szak, BSc
Labor

Bedők Dávid
2013.03.05.
v0.1

Feladat

A **Swedish store** alkalmazás egy bútoráruházat szeretne modellezni*. Az alkalmazás azért jöhet létre, mert:

- az áruház vezetése szeretne sok pénzt keresni
- a vevőket ki kell tudni szolgálni annak érdekében, hogy a vezetők sok pénzt kereshessenek

* szimulációs alkalmazás, melynek kiemelt célja, hogy valós üzleti problémát járjon körbe, azonban számos ponton egyszerűsítéseket tartalmaz, hogy az elméleti anyag maradhasson hangsúlyos

Kezdeti követelmények

Három különféle típusú bútort szeretnénk eladni a vevőknek. A vevők pontos kiszolgálása érdekében a következő adatokat szeretnénk tárolni minden egyes bútorról:

- milyen **szobá**ba való elsődlegesen (pl. Konyha, Fürdőszoba, Kamra, Előszoba, stb.)
- milyen **anyag**ból készült elsődlegesen (pl. Tölgyfa, Bükkfa, Műanyag, Fém, stb.)
- milyen **dimenziók**kal rendelkezik (szélesség, magasság, mélység/hosszúság)
- **mennyibe kerül** EUR valutában
- bútor **fántázia nev**e, mely a könnyebb azonosíthatóságot segíti (egyediségét nem kell ellenőrizni, de érdemes különböző neveket választani tesztelésnél)

Asztalokról tárolt további adatok

- hozzá adott **székek száma** (tételezzük fel, hogy ezek benne vannak az árban, vagyis ha nincsenek benne az árban, akkor értéke 0)
- **karcálló** az asztal, avagy nem?
- **összecsukható** kisebb méretűre, avagy nem?

Ágyakról tárolt további adatok

- milyen **matrac** tartozik hozzá (pl. Antiallergén, Komfortos, Egészséges, stb.)
- egyszemélyes avagy **franciaágy**?
- **összecsukható** kisebb méretűre, avagy nem?
- **beépített lámpá**val rendelkezik-e, avagy sem?

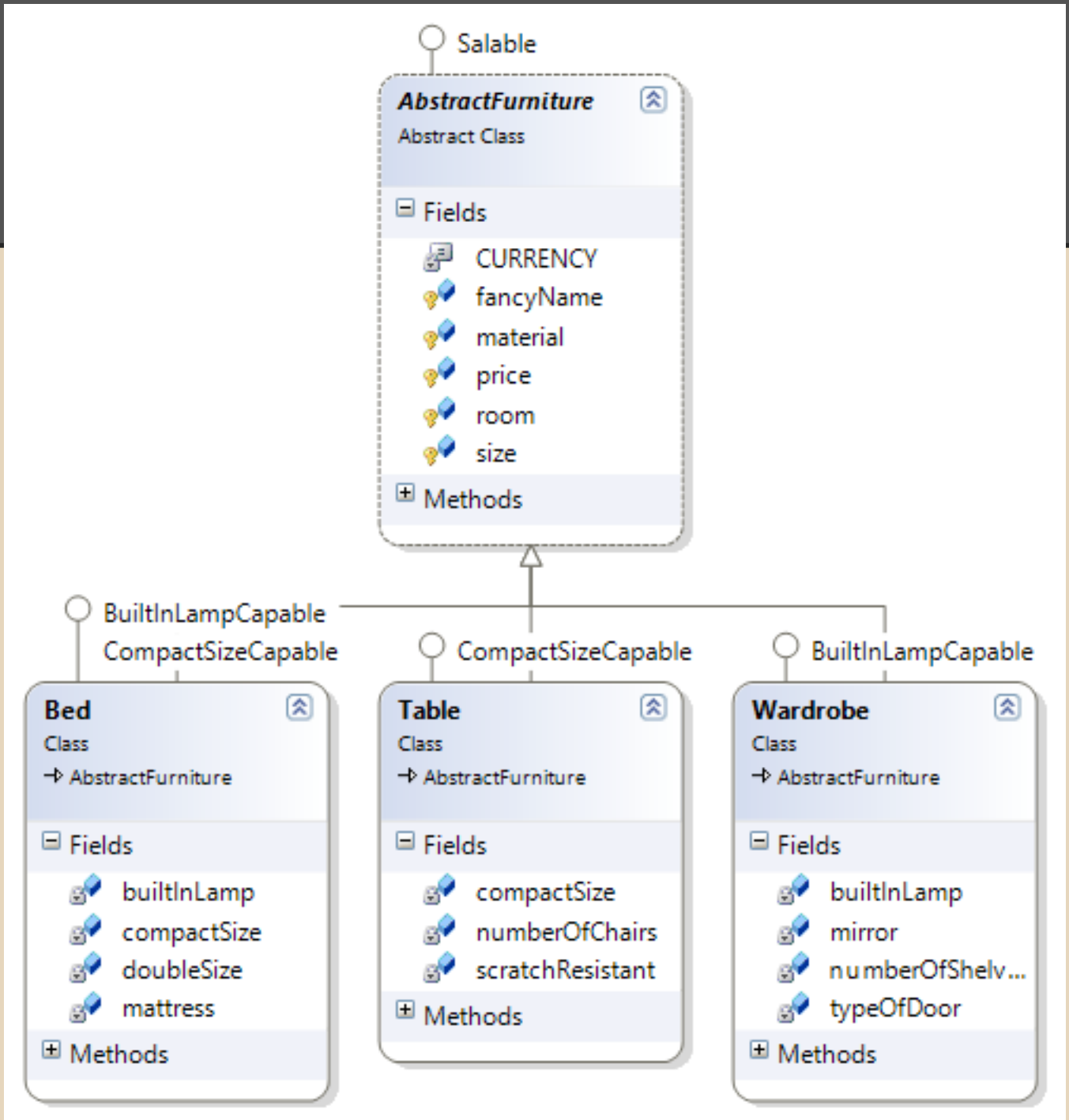
Szekrényekről tárolt további adatok

- hány darab **polc**cal rendelkezik?
- milyen típusú **ajtó** van rajta (pl. Kétszárnyú, Toló, Redőny, stb.)
- van-e rajta **tükör**, avagy nincs?
- **beépített lámpá**val rendelkezik-e, avagy sem?

Kezdeti modell

A követelmények elég jól vezetik azt, hogy a bútorokról tárolt közös adatokat egy **közös ős** osztályban érdemes tárolni, hiszen a **redundancia csökkentése** kiemelten fontos szempont, és természetesen a cég vezetése szempontjából csak "bútorok" vannak, őket különösebben nem érdekli hogy milyen tételek eladásából van a profit (ez nyilván azért nem így van a valóságban, de a modellben igen!)

Osztály diagram



***Megj.**: a jelölt interface-ekkel most ne foglalkozzunk (később visszatérünk majd ehhez)*

Probléma

Az ősosztály nem tud megoldást adni minden redundancia eltűntetésére. Vannak olyan tulajdonságok, melyek újabb köztes ős osztály létrehozását sugallják, azonban többszörös öröklés nélkül ezt nem tudjuk megvalósítani minden irányból!

Ezért a középső osztályszintet nem vezetjük be, és a redundancia alkalmazásánál maradunk.

Szükséges elméleti ismeretek

- Programozás 1...
- Nem virtuális metódusok kontra virtuális metódusok
- Korai-késői kötés
- Virtuális metódus tábla
- Absztrakt osztályok
- Absztrakt metódusok

Áruház

Egy halom bútor és elérhető raktármennyiség tárolása egységesen egy áruház osztály példányaként reprezentálva.

Eleddig nem tanult generikus típus alapvető bevezetése itt előnyös:

Dictionary<Key, Value>

A generikus típusok elméletét későbbre tartogatjuk (ez komolyabb anyag), egyelőre csak a fenti típus felhasználásával ismerkedjünk meg!

Követelmények az áruházzal kapcs.

- Új árucikk felvétele
- Név alapján árucikk keresése
- Új árucikk érkeztetése (név alapján meglévő kikeresése, és új beérkezett mennyiség raktárkészlethez csatolása)
- Számos keresési feltétel szerint bútorok listájának előállítása az érdeklődő vevők számára
- Árucikk eladása

Probléma

Egy Dictionary-ban minden egyes Kulcs csak egyszer szerepelhet, azonban az csak "félmegoldás", ha a név alapján egyezőnek tekintünk két bútort, nem beszélve arról, hogy mindez az alkalmazásnak nem lesz két azonos bútor (két azonos névvel (sőt adatokkal) rendelkező bútor duplikálva is megjelenhet jelenleg a szótárban!

Megoldás: overload Equals(), GetHashCode() Object-ből, esetleg overload == és != operátor

Feladat

- Size osztályban Equals() és GetHashCode() metódusok felülírása (általános eset)
- AbstractFurniture osztályban Equals és GetHashCode() metódusok felülírása (speciális eset: egy objektum sosem lehet "csak AbstractFurniture")
- Leszármazott bútor osztályokban Equals() és GetHashCode() metódusok felülírása

Equals: reflexív, szimmetrikus, tranzitív, konzisztens, null biztos

GetHashCode: konzisztens, Equals true esetén a hashCode-ok azonosak (fordítva nem igaz!)

Probléma

Vevői igény: melyik bútorok csukhatók össze kisebb méretűre?

Erre jelenleg nem tudunk válaszolni.. az ágyak és az asztalok lehetnek "compact"-ok, viszont ezt csak ronda beégetéssel tudánk lekezelni, ami karbantarthatatlan kódot eredményez

Megoldás: interface-ek bevezetése

Feladat

- interface **CompactSizeCapable** (Bed, Table)
- interface **BuiltInLampCapable** (Bed, Wardrobe)
- Generikus List<Item> használata
- List<AbstractFurniture> listAllCompactSizeCapableFurniture() metódus a Store osztályban

Új üzleti igény

A profit növelése érdekében a cég vezetése mostantól kiegészítőket is szeretne árusítani, de a jelenlegi osztályokat a már megvalósult integráció miatt (bla, bla... sosem késő refaktorálni..) nem módosíthatjuk!

Az új termékeknek is egy céljuk van: profit termelése, vagyis hogy el lehessen őket adni.

Vezessük be az új osztályt, és a **közös viselkedés**t a bútorokkal!

Feladat

- interface **Salable**
- class **AbstractMarketGood** : Salable
- class **Lamp** : AbstractMarketGood
- class **DinnerwareSet** : AbstractMarketGood

Áruház refaktorálása

Új típus: Dictionary<**Salable**, Int32>

Gyakorlatilag mindenhol csak az **AbstractFurniture**-t kell kicserélni **Salable**-re.

A Salable interface által definiált viselkedés valójában nem más, mint a Store osztályból hívott nyílvános AbstractFurniture metódusok listája!

Kérdések

?