SZOFTVERTECHNOLÓGIA ÉS -TECHNIKÁK
II. HÁZI FELADAT
2023

1. A FELADAT

Mindenkinek egy, a jelen dokumentum 2. fejezetében szereplő valamelyik feladatot kell megvalósítania az alábbi táblázat szerint:

Neptun kód **harmadik** karaktere:

- 0-9: KÖNYVTÁR
- A-G: KATONAI VEZÉRLŐKÖZPONT
- H-N: CÉGRÉSZLEGEK
- O-U: ÁLLATOS KÁRTYA KATALÓGUS
- V-Z: BÚTORGYÁR SZIMULÁCIÓ

A megoldásban többek között az alábbi források nyújthatnak segítséget:

- Előadás anyagok, ezen belül kiemelten az előadáshoz készített, GitHub-on elérhető, az egyes minták működését demonstráló mintakódok
- Gyakorlat anyagok

A házi feladat célja, hogy a feladat által érintett minták a tárgyhoz kapcsolódó anyagok segítségével mélyebb feldolgozásra, egy-egy egyszerű példaalkalmazás segítségével alkalmazásra és gyakorlásra kerüljenek.

A megoldásban pontszám csak akkor kapható az adott részfeladatra, ha az követi a feladatleírásban megjelölt minta/minták elveit. Ha egy feladat több részből áll (pl. A és B), akkor ezek mindegyike a feladat része.

A feladatok megoldását a gyakorlatvezetőnek személyesen be kell mutatni az utolsó gyakorlaton. Ennek során szükséges:

- A kód felépítésének és a megoldás működésének magyarázata
- Az alkalmazás helyes működésének demonstrálása a kód futtatásával beépített tesztadatok által generált kimenet alapán
- Annak magyarázata, hogy a megvalósítás során a feladatleírásban szereplő tervezési mintáknak mi a szerepe a megoldásban, illetve mely osztályok feleltethetők meg a megoldásban a tervezési minta egyes szereplőinek.
- Kérésre a funkcionalitás/kód kismértékű módosítása

A bemutatás történhet saját eszközről (laptop) vagy laborgépről. Az utóbbi esetben a forráskód olyan formában kell rendelkezésre álljon, hogy a laborban rendelkezésre álló eszközök (Visual Studio, Eclipse) segítségével megnyitható, fordítható és futtatható legyen.

A megoldáshoz nem szükséges UML diagramot, illetve unit teszteket készíteni (de természetesen készíthetők). A megoldás során törekedjen az egyszerűségre, csak olyan szinten kell a feladatot megvalósítania, ahogy azt a feladatleírás kéri. A megoldásához dokumentációt nem kell készítenie. A forráskódot csak minimális mértékben kell kommentezni, a fontosabb osztályokat lássa el rövid megjegyzésekkel (pl. mi a szerepe az adott osztálynak).

Valamennyi házi feladatot **a bemutatás előtti nap délig** fel kell tölteni a tárgy honlapjára egy .zip csomagban. Ennek a .zip állománynak tartalmaznia kell a megoldás forráskódját, mást nem. További szabályok:

- .NET esetén a .zip állomány nem tartalmazhatja a kimeneti („.exe") és köztes állományokat! Ezen állományok törléséhez a Solution mappából kézzel törölni kell a „bin" és „obj" mappákat teljes tartalmukkal együtt. A .zip állományba ne tegyük bele a „.git" és „.vs" könyvtárat. Az viszont fontos, hogy ne csak a .cs forrásfájlok legyenek feltöltve, hanem az egész solution mappa a .sln/.csproj/stb. fájlokkal, hogy Visual Studioban könnyen meg lehessen nyitni.
- Java esetén pedig olyan formában/tartalommal kell bezippelni, hogy utána könnyedén lehessen importálni Eclipse-ben. Tehát pl. Eclipse-ben: File -> Import -> Existing Projects into Workspace működjön: nem elég csak az src mappát, vagyis a .java forrásállományokat feltölteni.

2. FELADATOK

1. KÖNYVTÁR

Készítsen egy egyszerű nyilvántartó **háromrétegű** alkalmazást C# vagy Java nyelven könyvtárhoz.
A) Készítse el a háromrétegű alkalmazást!

- Az egyes könyvekhez a következő adatokat kell tárolni:
  - Azonosítószám (16 karakter hosszú alfanumerikus azonosító)
  - Cím
  - Műfaj (sci-fi, romantikus, fantasy, klasszikus, vagy egyéb)
  - Szerzőlista, melyben a szerzők egyszerű sztringek

  A könyvekről csak a fenti adatokat tároljuk!
- A megvalósítandó funkcionalitás:
  - Új könyv felvétele (Azonosítószám, cím és szerzők megadásával, egy lépésben)
  - Könyv műfajának megadása (a könyv azonosítóját használva)
  - Az összes könyv esetén egy adott nevű szerző nevének átírása minden könyvben
  - Minden könyv listázása
- Rétegek kódszervezése
  - Amennyiben .NET környezetben dolgozik, az egyes rétegekhez nem szükséges külön projektet létrehoznia, de az eltérő rétegekbe tartozó osztályokat külön projekt mappákba és névterekbe szervezze (a feladat egyszerűsége miatt előfordulhat, hogy egy mappában/névtérben csak egy osztály lesz).
  - Amennyiben Java környezetben dolgozik, szervezze az egyes rétegeket külön package-ekbe.
- Adathozzáférési réteg. A logikai réteghez az adathozzáférési réteg legyen lazán csatolt, amit a **Strategy** és a **Dependency Injection** minták segítségével valósítson meg, és ennek során két adathozzáférési megvalósítást is készítsen elő!
  - Memória alapú. Ennek során az adatokat tárolhatja egyszerű listában (.NET esetén List<…> vagy pl. Dictionary-ben).

    - Adatbázis alapú. Az adatbázis alapúnál a tényleges adattárolást nem kell megvalósítania, a függvények törzse üresen maradhat, illetve visszatérhetnek tetszőleges beégetett adatokkal.

  A két megvalósítást .NET környezetben két külön mappába/névtérbe, Java esetében két külön package-be szervezze.

  Segítség: A laza csatolás esetünkben azt jelenti, hogy a logikai réteg osztályaiba ne legyen beégetve, milyen adathozzáférés rétegbeli (konkrétan repository) implementációs osztállyal/osztályokkal dolgozik. Vagyis a logikai rétegbeli osztály/osztályok kódját egyáltalán ne kelljen módosítani új adathozzáférés (repository) implementációk bevezetésekor!
- Azt feltételezheti, hogy a rendszerben viszonylag kevés könyv létezik, így azok adatai egyben betölthetőek a memóriába, és egyben el is menthető a teljes lista. Így az adathozzáférési rétegben valamennyi könyv mentése, illetve betöltése egyben történik (nem könyvenként meghívva a megfelelő függvényt).
- A logikai réteg cache-elheti (tagváltozóban tárolhatja) a könyveket, mint ahogy az kapcsolódó gyakorlat megoldásában is szerepelt, de ez nem kötelező.
- A logikai validációkat a középső, logikai rétegben valósítsa meg. Ilyen logikai szabályok új könyv felvételekor:
    - Az azonosító nem szerepelhet a korábbi könyvek közt
    - A cím nem lehet null és legalább négy karakter hosszú legyen

  Logikai szabály sérülése esetén dobjon kivételt, és a kivétel szövegét jelenítse meg a felületen (írja ki a konzolra a felhasználói felület rétegben).
- A logikai rétegben vezessen be a **Singleton** minta alapján egy osztályt, melytől (egy többfelhasználós környezetet “szimulálva”) le lehet kérdezni az aktuálisan bejelentkezett felhasználó azonosítóját. Az azonosító egy egész szám. A megoldásban a művelet egy fix, beégetett számmal térjen vissza. A logikai réteg minden adatmanipuláció (új könyv felvétele, kategória rögzítése, ill. szerző módosítása) esetén naplózzon ki egy sort a konzolra: ebben szerepeljen az érintett könyv címe, a manipuláció típusa (pl. „Létrehozás”), illetve az aktuálisan bejelentkezett felhasználó **neve**. A logikai réteg a név megszerzéséhez a Singleton segítségével szerezze meg a felhasználó azonosítót, majd az adathozzáférési rétegtől szerezze meg ezen azonosító alapján a bejelentkezett felhasználó nevét (ez utóbbi egy fix, beégetett névvel térjen vissza). Az adathozzáférési réteg azon osztálya, mely a felhasználóazonosító alapján visszaadja a felhasználó nevét, ne a könyvek perzisztenciájáért felelős osztály legyen (vagyis ehhez vezessen be egy új repository osztályt, illetve interfészt).
- **Felhasználói felület** egy egyszerű, konzolra író osztály legyen. Ebben nem szükséges felhasználótól parancsokat bekérnie, helyette az alábbi, felhasználói parancsokat szimuláló műveleteket vezesse be és tesztelés céllal hívja is meg:
    - Négy új könyv felvétele, beégetett adatokkal. A négy közül az utolsóelőtti legyen érvénytelen.
    - Adott, beégetett nevű szerző módosítása valamennyi könyvben.

  Mindegyik módosítás után a felület osztály kérdezze le az aktuális könyvlistát az alatta levő rétegtől és listázza ki a konzolra valamilyen egyszerű formában.
- A Dependency Injection bevezetése során egyszerű „manuális” megoldásra törekedjen (ahogy előadáson is szerepelt), nem kell használnia az ASP.NET Core vagy más keretrendszer Dependency Injection konténer szolgáltatását. Azt, hogy a logikai réteg milyen adathozzáférési implementációval dolgozzon, a felhasználói felület réteg injektálja be számára.
- Tesztadatok segítségével illusztrálja megoldásának működését.
- Megoldásában nem szükséges semmilyen adatot fájlba menteni, vagy onnan betölteni.
- A megoldáshoz **javasolt** többek között a „10. gy - Szoftverarchitektúrák: többrétegű alkalmazások” gyakorlat és a „07. ea. Tervezési minták 2” előadáson belül a Dependency Injection témakör (illetve kiemelten az utóbbihoz kapcsolódó GitHub-on elérhető mintakód) mélyebb feldolgozása.

2. KATONAI VEZÉRLŐKÖZPONT

A feladata egy katonai vezérlőközpont megvalósítása C# nyelven.

A) A megoldását az **Observer** minta alapján valósítsa meg a következők szerint:

- A katonai vezérlőközpontban egy vezérlőpanel kapott helyet. Ezen a panelen a következő információkat követhetjük: az általános harci fokozat (alapeset/készültség/éles helyzet), a rakétakilövő aktuális fokozata (inaktív/készültség/éles helyzet), naplók. A naplók egy-egy listát jelentenek az általános harci fokozat, ill. a rakétakilövő fokozatának változásairól, melyek egy időbélyeggel rendelkeznek. A két napló külön listában legyen kezelve. Tipp: .NET környezetben idő kezelésére a DateTime típus használható, a DateTime.Now adja vissza az aktuális időt.
- A vezérlőpanel lehetőséget (műveletet) biztosít az általános harci fokozat, ill. a rakétakilövő fokozatának megváltoztatására, valamint vészhelyzet esetén a kettő egyszerre történő aktiválására (éles helyzet állapotba kapcsolva mindkét rendszert). Éles helyzet állapotban a rendszer a beregisztrált kezelőknek 4 másodpercen keresztül másodpercenként értesítést küld, majd az idő lejárta után automatikusan visszakapcsol a rendszer készültség állapotba. Ha a vezérlők eleve éles helyzetben voltak, akkor nem történik semmi (nem kezdünk el újra riasztásokat küldeni).
- Vezérlőből egyszerre több is létezhet a rendszerben (vagyis megvalósítására nem használhatja a Singleton mintát).
- A rendszerben kétfajta kezelőt valósítson meg:
    - Felirat. Inaktív állapotban a „[ ]”, készenlét állapotban a „[****]”, míg éles helyzet esetén a „[!!!!]” szöveget írja ki az adott kurzorpozícióba! A készenlét állapot szövege sárga, az éles helyzeté piros színnel kerüljön kiírásra!
      Mivel a két rendszer egymástól függetlenül is aktiválható, két független, eltérő kurzorpozícióban megjelenített szöveggel dolgozzon!
    - Sziréna. Harci készültség esetén másodpercenként villogjon egy csillag karakter piros színnel adott kurzorpozícióban! A villogás során a karakter egy másodpercre jelenjen meg, majd egy másodpercre tűnjön el!
      Mivel a két rendszer egymástól függetlenül is aktiválható, két független, eltérő kurzorpozícióban megjelenített szöveggel dolgozzon.
    - A rendszer könnyen, a vezérlő és a meglévő riasztási kezelők módosítása nélkül lehessen bővíthető újabb kezelőkkel!
- Az alkalmazás az indulásakor állítson össze egy általános, ill. egy rakétakilövő állomást kezelő konfigurációt, és indítson megfelelő aktiválásokat, melyek demonstrálják az vezérlő és a kezelők működését.
- Megoldásában nem szükséges semmilyen adatot fájlba menteni, vagy onnan betölteni.

B) Adott az alábbi forráskódú, a parancsnokságra bekötött távfelügyeleti modul. Az **Adapter** minta segítségével illessze be egy új kezelőként a rendszerbe. Ennek során a ParancsnoksagiFelugyelet forráskódját az alábbi, változatlan formában építse be a megoldásába:

```
class ParancsnoksagiFelugyelet
{
   public void HarciAllapotAktivalva()
   {
      Console.ForegroundColor = ConsoleColor.Red;
      Console.SetCursorPosition(0, 0);
      Console.Write("Figyelem, ÉLES HARCI ÁLLAPOT aktív!   ");
   }

   public void HarciAllapotDeaktivalva()
   {
      Console.ForegroundColor = ConsoleColor.Green;
```

```
            Console.SetCursorPosition(0, 0);
            Console.Write("Figyelem, ÉLES HARCI ÁLLAPOT inaktív!   !   ");
        }
    }
```

A ParancsnoksagiFelugyelet HarciAllapotAktivalva műveletét akkor kell hívni, amikor az általános harci helyzet tekintetében éles fokozatba kapcsolunk (a rakétakilövő állomás aktiválása esetén nem!), illetve a HarciAllapotDeaktivalva()-t akkor, amikor az éles harci helyzet állapot megszűnik.
Kösse be a ParancsnoksagiFelugyelet-et a rendszerbe és demonstrálja működését!

3. CÉGRÉSZLEGEK

Készítsen egy alkalmas osztályhierarchiát C# vagy Java nyelven, amely képes cégek részlegeinek felépítését modellezni. A cég alkalmazottakból (Employee) és részlegekből (Department) áll. Az alkalmazottakról a nevüket és a munkaviszony kezdetét (dátum) tároljuk, a részlegekről pedig a nevüket és a bennük lévő maximális alkalmazottak számát. Egy részleg alkalmazottakból és további alrészlegekből is állhat. Minden alkalmazott 1 főnek számít, alrészlegek létszáma pedig rekurzívan a bennük lévő alkalmazottak és alrészlegek létszámainak összegét jelenti.

A) A **Composite** minta segítségével oldja meg a következőket:

- A részlegek hierarchiájának modellezése oszályokkal / interfészekkel.
- A név legyen lekérdezhető az alkalmazottakra és a részlegekre is, mindkét esetben egy string-el térjen vissza.
- A részlegekhez vezessen be egy műveletet alkalmazottak és alrészlegek felvételére. Egyszerre lehessen akár többet is hozzáadni (paraméter egy alkalmazott, vagy részleg objektumokat tároló lista). Ha az adott részlegben (amelyikhez éppen hozzá akarjuk adni az új elemeket) az új elemek létszámával együtt az összlétszám meghaladná az adott részleg maximális létszámát, akkor a hozzáadás legyen teljesen sikertelen, ne adja hozzá egyik kért elemet se! **Fontos**: a részlegek nem tárolhatják el, hogy pontosan hány alkalmazottuk van, azt minden hozzáadás során ki kell frissen számítani.
- A részlegekhez vezessen be egy műveletet adott alkalmazott vagy részleg eltávolítására is (paraméter egy alkalmazott vagy részleg objektum).
- Írjon egy műveletet, mely egy részlegre vonatkozóan kilistázza a nevét, és az összes tartalmazott alkalmazott nevét, minden alrészlegével, rekurzívan. Minden részleg/alkalmazott információ új sorban jelenjen meg, minden sorban a következő két mezővel, az egyes mezők 3 darab szóközzel elválasztva:
    - "Dept." vagy "Empl." sztring, attól függően, hogy az adott elem részleg vagy alkalmazott
    - Adott alkalmazott/részleg neve
- A megjelenítés során elég az egyszerű lista, nem szükséges behúzással vagy egyéb módon a hierarchiát vizualizálni.
- A megjelenítés során tesztadatokkal illusztrálja a megoldásának működőképességét. Mutasson példát sikertelen hozzáadásra, és az adatok listázására is.
- Megoldásában nem szükséges semmilyen adatot fájlba menteni, vagy onnan betölteni.

B) Az **Observer** minta segítségével oldja meg a következőket:

- Az egyes részlegek sikertelen hozzáadásaira (amikor alkalmazottakat vagy részlegeket szeretnénk felvenni bele, de a művelet sikertelen volt) fel lehessen iratkozni.
- Mutasson is példát két eltérő típusú feliratkozóra:

        - Az egyik írja ki a konzolra egy sorba, pontosvesszővel elválasztva a hozzáadni kívánt elemek neveit (a hozzáadás metódus lista paramétere). Az alrészlegeket nem kell rekurzívan bejárni és kiírni, csak a közvetlen elemeket.
        - Egy másik pedig írja ki a konzolra, hogy mennyivel lépte volna túl a hozzáadás az adott részleg maximális létszámát.
        - Tipp: ahhoz, hogy ezeket meg lehessen oldani, az értesítés során célszerű átadni a szükséges információkat. **Fontos**: a lista paraméter elemeinek összlétszámát ne adjuk át az értesítés során, azt a feliratkozó számolja ki frissen.
    - Tesztadatok segítségével illusztrálja megoldásának működését. Mutasson példát mindkét típusú feliratkozóra!

4. Állatos kártya katalógus

Készítsen egy egyszerű állatos kártyákat nyilvántartó **háromrétegű** alkalmazást C# vagy Java nyelven.
A) Készítse el a háromrétegű alkalmazást!

- Az egyes kártyákhoz a következő adatokat kell tárolni:
    - Név
    - Latin név
    - Fajta (jelenleg csak *madár*, *kétéltű*, *hal*, vagy *emlős*)
    - Rövid leíró szöveg

A kártyákról csak a fenti adatokat tároljuk - képeket tehát nem kell tárolni!

- A megvalósítandó funkcionalitás:
    - Új kártya felvétele (név, latin név, és rövid leíró szöveg megadásával, egy lépésben)
    - Kártya fajtájának megadása (a kártya neve alapján)
    - Kártya törlése név alapján
    - Minden kártya listázása
- Rétegek kódszervezése
    - Amennyiben .NET környezetben dolgozik, az egyes rétegekhez nem szükséges külön projektet létrehoznia, de az eltérő rétegekbe tartozó osztályokat külön projekt mappákba és névterekbe szervezze (a feladat egyszerűsége miatt előfordulhat, hogy egy mappában/névtérben csak egy osztály lesz).
    - Amennyiben Java környezetben dolgozik, szervezze az egyes rétegeket külön package-ekbe.
- Adathozzáférési réteg. A logikai réteghez az adathozzáférési réteg legyen lazán csatolt, amit a **Strategy** és a **Dependency Injection** minták segítségével valósítson meg, és ennek során két adathozzáférési megvalósítást is készítsen elő!
    - Memória alapú. Ennek során az adatokat tárolhatja egyszerű listában (.NET esetén List<…> vagy pl. Dictionary-ben, Java esetén ArrayList<…> vagy pl. HashMap-ben).
    - Adatbázis alapú. Az adatbázis alapúnál a tényleges adattárolást nem kell megvalósítania, a függvények törzse üresen maradhat, illetve visszatérhetnek tetszőleges beégetett adatokkal.

  A két megvalósítást .NET környezetben két külön mappába/névtérbe, Java esetében két külön package-be szervezze.

  Segítség: A laza csatolás esetünkben azt jelenti, hogy a logikai réteg osztályaiba ne legyen beégetve, milyen adathozzáférés rétegbeli (konkrétan repository) implementációs osztállyal/osztályokkal dolgozik. Vagyis a logikai rétegbeli osztály/osztályok kódját egyáltalán ne kelljen módosítani új adathozzáférés (repository) implementációk bevezetésekor!
- Azt feltételezheti, hogy a rendszerben viszonylag kevés kártya létezik, így azok adatai egyben betölthetőek a memóriába, és egyben el is menthető a teljes lista. Így az adathozzáférési rétegben valamennyi kártya mentése, illetve betöltése egyben történik (nem kártyánként meghívva a megfelelő függvényt).

- A logikai réteg cache-elheti (tagváltozóban tárolhatja) a kártyákat, mint ahogy az kapcsolódó gyakorlat megoldásában is szerepelt, de ez nem kötelező.
- A logikai validációkat a középső, logikai rétegben valósítsa meg. Ilyen logikai szabályok új kártya felvételekor:
    - Név nem lehet null és legalább négy karakter hosszú kell legyen
    - A névnek minden kártya között egyedinek kell lennie
    - A rövid leíró szöveg maximum 255 karakter hosszú lehet

  Logikai szabály sérülése esetén dobjon kivételt, és a kivétel szövegét jelenítse meg a felületen (írja ki a konzolra a felhasználói felület rétegben).
- A logikai rétegben vezessen be a **Singleton** minta alapján egy osztályt, melytől (egy többfelhasználós környezetet "szimulálva") le lehet kérdezni az aktuálisan bejelentkezett felhasználó azonosítóját. Az azonosító egy egész szám. A megoldásban a művelet egy fix, beégetett számmal térjen vissza. A logikai réteg minden adatmanipuláció (új kártya felvétele, kártya fajtájának megadása és kártya törlése) esetén naplózzon ki egy sor a konzolra: ebben szerepeljen az érintett kártyaneve, a manipuláció típusa („Létrehozás", „Módosítás", vagy „Megszüntetés"), illetve az aktuálisan bejelentkezett felhasználó **neve**. A logikai réteg a név megszerzéséhez a Singleton segítségével szerezze meg a felhasználó azonosítót, majd az adathozzáférési rétegtől szerezze meg ezen azonosító alapján a bejelentkezett felhasználó nevét (ez utóbbi egy fix, beégetett névvel térjen vissza). Az adathozzáférési réteg azon osztálya, mely a felhasználóazonosító alapján visszaadja a felhasználó nevét, ne a kártyák perzisztenciájáért felelős osztály legyen (vagyis ehhez vezessen be egy új repository osztályt, illetve interfészt).
- **Felhasználói felület** egy egyszerű, konzolra író osztály legyen. Ebben nem szükséges felhasználótól parancsokat bekérnie, helyette az alábbi, felhasználói parancsokat szimuláló műveleteket vezesse be és tesztelés céllal hívja is meg:
    - Három új kártya felvétele, beégetett névvel, latin névvel és leíró szöveggel. A három közül az utolsóelőtti legyen érvénytelen.
    - Adott, beégetett nevű kártya törlése.

  Mindegyik módosítás után a felület osztály kérdezze le az aktuális kártya listát az alatta levő rétegtől és listázza ki a konzolra a következő formátumban: kártyánként egy sor, az adatok a # karakterrel elválasztva.
- A Dependency Injection bevezetése során egyszerű „manuális" megoldásra törekedjen (ahogy előadáson is szerepelt), nem kell használnia az ASP.NET Core vagy más keretrendszer Dependency Injection konténer szolgáltatását. Azt, hogy a logikai réteg milyen adathozzáférési implementációval dolgozzon, a felhasználói felület réteg injektálja be számára.
- Tesztadatok segítségével illusztrálja megoldásának működését.
- Megoldásában nem szükséges semmilyen adatot fájlba menteni, vagy onnan betölteni.
- A megoldáshoz **javasolt** többek között a „10. gy - Szoftverarchitektúrák: többrétegű alkalmazások" gyakorlat és a „07. ea. Tervezési minták 2" előadáson belül a Dependency Injection témakör (illetve kiemelten az utóbbihoz kapcsolódó GitHub-on elérhető mintakód) mélyebb feldolgozása.

5. Bútorgyár szimuláció

Egy automatizált bútorgyár különböző stílusú bútorkollekciók elkészítésére lehet felprogramozni. Készítse el a bútorgyár szimulációját C# vagy Java nyelven. A megoldását az **Abstract Factory** minta alapján valósítsa meg a következők szerint:

- A bútorgyár minden időpillanatban egyféle stílusú bútorkollekció elkészítésére van felkonfigurálva
- A bútorgyár műveletei:
    - Bútorgyár felkonfigurálása adott stílusú bútorok gyártására

    - Bútorgyár üzemeltetése, paramétere egy bútorkollekció és darabszáma. Hatására a gyártás addig fut, amíg le nem gyártja a megadott darabszámú és típusú bútort, utána leáll. A darabszámba a sikertelen autógyártás is beleszámít (ha diagnosztika során hiba lép fel, lásd alább).
- A támogatott bútor-kollekciók: nappali, előszoba
- A támogatott stílusok: art deco és classic
- A megvalósítás során csupán egy-egy osztályt vezessen be a bútorkollekciók reprezentálására, mely tagváltozókban tárolja az adott kollekció bútorait. Vagyis ne legyenek leszármazottjai az bútorkollekciókat reprezentáló osztályoknak: az ugyanazon típusú kollekciók abban különböznek, hogy a hozzájuk tartozó bútorok stílusa eltérő.
- A különféle típusú kollekciók gyártási folyamatai a következőképpen néznek ki:
    - Nappali bútor-kollekció esetén: asztal elkészítése, szekrény elkészítése, szék elkészítése, majd a végén az csomagolás.
    - Előszoba bútor-kollekció esetén: fogas, szekrény elkészítése, majd a végén az csomagolás.
- Valamennyi bútornak egységes módon támogatni kell a naplózási műveletet, amit a gyártás során meg is kell hívni.
- Lényeges, hogy a csomagolás lépés minden adott bútor-kollekció esetén ugyanaz! Vagyis a kódban bútor-kollekciónkként egyetlen csomagolást végző művelet lehet! Ennek bemenete a már legyártott bútorok, kimenete pedig egy új bútor-kollekció objektum.
- A megoldásában kerülje a kódduplikációt, és ügyeljen arra, hogy a gyártási folyamat (és ezen belül a csomagolás művelet) módosítása nélkül lehessen új stílust bevezetni!
- Az elkészített, azonos típusú bútor-kollekció objektumokat a memóriában egy közös gyűjteményben tárolja el.
- Tesztadatok segítségével illusztrálja megoldásának működését!
- Törekedjen a szép, objektumorientált megközelítésre!
- A megoldásában nem használhatja az object/Object típust!
- Új bútor stílus bevezetésével illusztrálja, hogy a gyártósor kódja módosítás nélkül képes új stílusú bútor-kollekciót (pl. szocreaál) is gyártani!
- Megoldásában nem szükséges semmilyen adatot fájlba menteni, vagy onnan betölteni.

A gyártási folyamat során keletkező naplóbejegyzések egy központi naplózórendszerben kerüljenek rögzítésre

- A naplózórendszert a szimuláció során a **Singleton** tervezési minta elvei szerint valósítsa meg.
- A naplózórendszer a Console-ra írja ki a naplóbejegyzéseket.
- Tesztadatok segítségével mutasson példát naplózásra is.