SZOFTVERTECHNOLÓGIA ÉS -TECHNIKÁK
II. HÁZI FELADAT
2020

1. A FELADAT

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

Neptun kód **harmadik** karaktere:

- A-G: 1. AUTÓ GYÁRTÓSOR SZIMULÁCIÓ
- H-N: 2. ZENESZÁM NYILVÁNTARTÓ
- O-U: 3. FÉNYKÉPADAT NYILVÁNTARTÓ
- V-Z, 0-2: 4. FÁJLRENDSZER
- 3-9: 5. RIASZTÓRENDSZER

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.

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

Jelenléti oktatási forma esetén a bemutatás történhet saját eszközről (laptop) vagy laborgépről. Vírushelyzetben bevezetett távoktatás esetén a bemutatás online formában (Teams hívás) történik, a hallgató saját gépének megosztott képernyőjével. 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).

A **házi feladatot az utolsó heti gyakorlat előtti nap éjfélig fel kell tölteni a tárgy honlapjára a „Házi feladat II." számonkéréshez** 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. ZENESZÁM NYILVÁNTARTÓ

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

- Az egyes zeneszámokhoz a következő adatokat kell tárolni:
  - Cím
  - Szerzőlista, melyben a szerzők egyszerű sztringek

  A zeneszámok nem szükséges, hogy a címen felül egyéb, pl. számazonosítóval rendelkezzenek (de bevezethető)
- A megvalósítandó funkcionalitás:
  - Új zeneszám felvétele (címmel és szerzőkkel, egy lépésben)
  - Zeneszám törlése cím alapján
  - Minden zeneszám 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 zeneszám létezik, így valamennyi zeneszám adata egyben betölthető a memóriába, és egyben el is menthető. Így az adathozzáférési rétegben elég, ha valamennyi zeneszám mentése, illetve betöltése egyben történik. De választhat olyan megoldást is, ahol az adathozzáférési réteg műveletei a zeneszámok egyesével való manipulációjára biztosít lehetőséget.
- A logikai réteg cache-elheti (tagváltozóban tárolhatja) a zeneszámokat, 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 zeneszám felvételekor:
    - Cím nem lehet null és üres string
    - A szerzőlista nem lehet null, legalább egy eleme kell legyen, az elemek nem lehetnek null-ok, és legalább négy karakter hosszúak kell legyenek

  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 zeneszám felvétele és zeneszám törlése) esetén naplózzon ki egy sor a konzolra: ebben szerepeljen az érintett zeneszám címe, a manipuláció típusa („Új", vagy „Eltávolítás"), 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 zeneszámok perszitenciájáért 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 zeneszám felvétele, beégetett címmel és szerzőkkel. A négy közül az utolsó legyen érvénytelen.
    - Adott, beégetett című zeneszám törlése.

  Mindegyik módosítás után a felület osztály kérdezze le az aktuális zeneszám listát az alatta levő rétegtől és listázza ki a konzolra valamilyen egyszerű formában (pl. számonként egy sor, cím és szerzők pontosvesszővel 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.

2. FÉNYKÉPADAT NYILVÁNTARTÓ

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

- Az egyes képekhez a következő adatokat kell tárolni:
  - Név
  - Szélesség pixelben
  - Magasság pixelben
  - Formátum (ez egy egyszerű sztring, pl. „jpeg”

  A képek nem szükséges, hogy a néven felül egyéb, pl. számazonosítóval rendelkezzenek (de bevezethető)
- A megvalósítandó funkcionalitás:
  - Új kép felvétele (névvel és metaadatokkal, egy lépésben)
  - Kép törlése név alapján
  - Minden kép 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 kép létezik, így valamennyi kép adata egyben betölthető a memóriába, és egyben el is menthető. Így az adathozzáférési rétegben elég, ha valamennyi kép mentése, illetve betöltése egyben történik. De választhat olyan megoldást is, ahol az adathozzáférési réteg műveletei a képek egyesével való manipulációjára biztosít lehetőséget.
- A logikai réteg cache-elheti (tagváltozóban tárolhatja) a képeket, 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 kép felvételekor:
  - Név nem lehet null és üres string
  - A két méret egyike sem lehet 0, negatív szám vagy 10.000-nél nagyobb érték

  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 kép felvétele és kép törlése) esetén naplózzon ki egy sor a konzolra: ebben szerepeljen az érintett kép neve, a manipuláció típusa („Új", vagy „Eltávolítás"), 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 képek perszitenciájáért 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 kép felvétele, beégetett névvel,méretekkel és formátummal. A négy közül az utolsólegyen érvénytelen.
    - Adott, beégetett nevű kép törlése.

  Mindegyik módosítás után a felület osztály kérdezze le az aktuális kép listát az alatta levő rétegtől és listázza ki a konzolra valamilyen egyszerű formában (pl. képenként egy sor, név és méretek pontosvesszővel 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.

3. Fájlrendszer

Készítsen egy fájlrendszerbeli könyvtár- és fájlstruktúra reprezentálására alkalmas osztálystruktúrát C# vagy Java nyelven. Ennek során vezesse be a File és Folder osztályokat. A fájlok vonatkozásában a fájl nevét és méretét kell nyilvántartani (tagváltozóban), a könyvtáraknál nevét.

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

- Egy könyvtárhierarchia memóriabeli reprezentálása
- A név és a méret egységes formában legyen lekérdezhető a fájlokra és a könyvtárakra vonatkozóan! A könyvtárak esetében a méret a könyvtárakban és az alkönyvtárakban rekurzívan levő valamennyi fájlt méretének összegét adja vissza. Fontos: a könyvtárak nem tárolhatják a méretet, azt minden lekérdezés során ki kell frissen számolni.
- A könyvtárhoz vezessen be egy közös műveletet fájl/könyvtár felvételére (paraméter egy fájl vagy könyvtár objektum).
- A könyvtárhoz vezessen be egy közös műveletet adott fájl/könyvtár eltávolítására (paraméter egy fájl vagy könyvtár objektum).
- Írjon egy műveletet, mely egy könyvtárra vonatkozóan kilistázza az adatait, majd a tartalmát, minden alkönyvtárával, rekurzívan. Minden könyvtár/fájl információ új sorban jelenjen meg, minden sorban a következő mezőkkel, az egyes mezők tabulátorral elválasztva:
    - "Dir" vagy "File" sztring, annak függvényében, hogy az adott elem könyvtár vagy fájl
    - Adott fájl/könyvtár neve
    - Adott fájl/könyvtár mérete

A megjelenítés során elég az egyszerű lista, nem szükséges behúzással vagy egyéb módon a hierarchiát vizualizálni.
- A megjelenítés során tesztadatokkal illusztrálja a megoldásának működőképességét. Megoldásában NEM szükséges valódi fájlrendszerbeli könyvtárakkal és fájlokkal dolgoznia: egyszerűen olyan tesztadatokkal inicializálja az adatstruktúráit, melyekkel a működés jól demonstrálható.
- Megoldásában nem szükséges semmilyen adatot fájlba menteni, vagy onnan betölteni.

B) Vezessen be a könyvtárakra vonatkozóan műveleteket fájl/könyvtár könyvtárba való felvételéhez és törlésére (ennek megoldása során szabadon dönthet a paraméterezésről). Az **Observer** minta segítségével oldja meg a következőket:
- Az egyes könyvtárak változásaira (amikor fájlt vagy könyvtárat vesznek fel bele, vagy törölnek belőle) fel lehessen iratkozni. Az alkönyvtárak változása nem számít az adott könyvtár vonatkozásában változásnak.
- Mutasson is példát két eltérő típusú feliratkozóra:
  - Az egyik egy könyvtár megváltozásakor a konzolra pontosvesszővel elválasztva írja ki a könyvtárban levő fájlok és könyvtárak nevét (az almappákat nem kell bejárni és kiírni).
  - Egy másik pedig írja ki a konzolra a könyvtár tartalmának méretét (a méretbe beleszámítva az összes alkönyvtárát, teljes mélységben).

  Tipp: ahhoz, hogy ezeket meg lehessen oldani, az értesítés során át kell adni a megváltozott könyvtár objektumot a feliratkozónak.
- Tesztadatok segítségével illusztrálja megoldásának működését. Megoldásában NEM szükséges valódi fájlrendszerbeli könyvtárakkal és fájlokkal dolgoznia: egyszerűen olyan tesztadatokkal inicializálja az adatstruktúráit, melyekkel a működés jól demonstrálható.

4. Riasztórendszer

A feladata egy riasztórendszer megvalósítása C# nyelven.
A) A megoldását az **Observer** minta alapján valósítsa meg a következők szerint:
- A riasztórendszer központi eleme egy vezérlőpanel, mely a következő információkat tárolja: aktuális egyszerű riasztási állapot (aktív/inaktív), aktuális tűzjelzés riasztási állapot (aktív/inaktív), riasztási történetiség. A riasztás történetiség egy közös történeti lista a riasztási és tűzjelzési eseményekről, melyek egy időbélyeggel és egy szöveges leírással rendelkeznek (az utóbbi csak azt tartalmazza, hogy egyszerű vagy tűzjelzési riasztás történt). Tipp: .NET környezetben idő kezelésére a DateTime típus használható, a DateTime.Now adja vissza az aktuális időt.
- A vezérlőpanel lehetőséget (műveletet) biztosít egyszerű riasztás aktiválására, tűzjelzési riasztás aktiválására, valamint a kettő egyszerre történő aktiválására. Aktiválás indítás után a megfelelő riasztási állapotok aktívak lesznek, a riasztó a beregisztrált kezelőknek 5 másodpercen keresztül másodpercenként értesítést küld, majd az idő lejárta után a riasztási állapotok inaktívak lesznek. A vezérlő riasztási állapotban (legyen az egyszerű, tűzjelző, vagy mindkettő) újabb riasztás indítást nem kell tudjon fogadni.
- Vezérlőből egyszerre több is létezhet a rendszerben (vagyis megvalósítására nem használhatja a Singleton mintát).
- A rendszerben kétfajta riasztási kezelőt valósítson meg:
  - Fényvillogó. Az alkalmazásban ez egyszerű riasztás esetén másodpercenként egy másodpercre megjelenít egy szöveget adott kurzorpozícióban, majd egy másodpercre elrejti azt. A szövege egyszerű riasztás esetén "Riasztás!", tűzjelző riasztás esetén "Tűzjelző!". A szöveg színe egyszerű riasztás esetén sárga, tűz esetén piros. Lényeges: akár egyszerre is lehet sima és tűzriasztás! Így a kettő jelzésére két független, eltérő kurzorpozícióban megjelenített szöveggel dolgozzon.
  - Sziréna. Az alkalmazásban egyszerű riasztás esetén másodpercenként felváltva a "Né" és a "Nó" szöveget jelenítse meg sárga színnel adott kurzorpozícióban. Tűzriasztás esetén

egy másik pozícióban másodpercenként váltogassa a "Né" és a "Nó" szöveget piros színnel megjelenítve.
    - A rendszer könnyen, a vezérlő és a meglévő riasztási kezelők módosítása nélkül lehessen bővíthető újabb kezelőkkel.
- Az alkalmazás az indulásakor állítson össze egy vezérlő és riasztási kezelő konfigurációt, és indítson megfelelő riasztásokat, melyek demonstrálják az vezérlő és a riasztási kezelők működését.
- Megoldásában nem szükséges semmilyen adatot fájlba menteni, vagy onnan betölteni.

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

```
class RendorsegiErtesito
{
   public void Beriaszt()
   {
      Console.ForegroundColor = ConsoleColor.Red;
      Console.SetCursorPosition(0, 0);
      Console.Write("Rendőrségi riasztás aktív!   ");
   }

   public void Kiriaszt()
   {
      Console.ForegroundColor = ConsoleColor.Green;
      Console.SetCursorPosition(0, 0);
      Console.Write("Rendőrségi riasztás inaktív.  ");
   }
}
```

A RendorsegiErtesito RiasztasBe műveletét akkor kell hívni, amikor egyszerű riasztás indul (tűz esetén nem), illetve a RiasztasKi()-t akkor, amikor az egyszerű riasztási állapot megszűnik.
Kösse be a RendorsegiErtesito-t a rendszerbe és demonstrálja működését.

5. Autó gyártósor szimuláció

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

- A gyártósor minden időpillanatban egy adott autótípus elkészítésére van felkonfigurálva
- A gyártósor műveletei:
    - Gyártósor felkonfigurálása adott autótípus gyártására
    - Gyártósor futtatása, paramétere egy darabszám. Hatására a gyártósor addig fut, amíg le nem gyártja a megadott darabszámú autót, utána leáll. A darabszámba a sikertelen autógyártás is beleszámít (ha diagnosztika során hiba lép fel, lásd alább).
- A támogatott autó típusok: Tesla Model 3 és Tesla Model X.
- Minden autótípus esetén a következő alkatrészeket kell legyártani: alváz, karosszéria, motor. Ezen alkatrészek eltérő típusúak az egyes autó típusok esetén (vagyis a kódban minden alkatrészhez legyen külön autótípusfüggő leszármazott).
- A megvalósítás során egyetlen osztályt vezessen be egy autó reprezentálására, mely tagváltozókban tárolja az autó alkatrészeit. Vagyis ne legyenek leszármazottai az autót reprezentáló osztálynak: az egyes autó típusok abban különböznek, hogy más, típusfüggő alkatrészekből épülnek fel.
- A gyártás folyamata egy autó esetén a következőképpen néz ki: alváz elkészítése, karosszéria elkészítése, motor elkészítése, majd a végén az összeszerelés.

- Valamennyi alkatrészre egységes módon támogatni kell a diagnosztika műveletet (ez a kódban dobjon kivételt, ha problémát talál), amit a legyártott alkatrészre az összeszerelés során meg is kell hívni.
- Lényeges, hogy az összeszerelés lépés minden autótípus esetén ugyanaz! Vagyis a kódban egyetlen összeszerelést végző művelet lehet! Ennek bemenete egy már legyártott alváz, karosszéria és motor alkatrész, kimenete pedig egy új autó objektum. Az összeszerelés művelet az autó elkészítése előtt futtasson diagnosztikát a paraméterül kapott három alkatrészre.
- A megoldásában kerülje a kódduplikációt, és ügyeljen arra, hogy a gyártási folyamat (és ezen belül az összeszerelés művelet) módosítása nélkül lehessen új autótípust bevezetni!
- Az elkészített autó 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 autótípus bevezetésével illusztrálja, hogy a gyártósor kódja módosítás nélkül képes új típusú autót (pl. Tesla Roadster) is gyártani!
- Megoldásában nem szükséges semmilyen adatot fájlba menteni, vagy onnan betölteni.

Amennyiben a gyártási folyamat során diagnosztikai hiba lép fel (a szimuláció során a kódban kivétel keletkezik), egy központi riasztási rendszeren keresztül történjen annak kezelése.

- A riasztási rendszert a szimuláció során a **Singleton** tervezési minta elvei szerint valósítsa meg.
- A riasztórendszer a Console-ra írja ki a hiba paramétereit (a kivétel szövegét).
- Tesztadatok segítségével mutasson példát riasztásra.