Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar

Automatizálási és Alkalmazott Informatikai Tanszék
Alkalmazott Informatika Csoport

Szoftvertechnikák laborgyakorlat

oktatói segédlet

6. mérés

Document-View architektúra

A gyakorlatot kidolgozta: Benedek Zoltán (benedek.zoltan@aut.bme.hu)
Utolsó módosítás ideje: 2020.04.18.

A gyakorlat célja

A kapcsolódó előadások anyaga:

- UML alapú modellezés
- Windows Forms alkalmazásfejlesztés
- Szoftver architektúrák (document-view)

A gyakorlat célja:

- UML alapú tervezés és néhány tervezési minta alkalmazása
- A Document-View architektúra alkalmazása a gyakorlatban
- UserControl szerepének bemutatása Window Forms alkalmazásokban, Document-View architektúra esetén
- A grafikus megjelenítés elveinek gyakorlása Window Forms alkalmazásokban (Paint esemény, Invalidate, Graphics használata)

A gyakorlat elején töltsük le a kész alkalmazást. A hallgatók ekkor még ne töltsék le, ne ezt kattintgassák, majd csak a mérés második részében. A mérésvezetőnek viszont szüksége lesz rá, mert ennek segítségével történik az feladat bemutatása.

A gyakorlat lépéseinek rövid ismertetése

Röviden mondjuk el a hallgatóknak, hogy az alábbiak szerint fogunk dolgozni:

- A feladat/célok rövid ismertetése: egy interaktív fonteditor (betűtípus-szerkesztő) megtervezése.
- A kész alkalmazást futtatva a feladat (a kész alkalmazás működésének) ismertetése.
- Az alkalmazás architektúrájának megtervezése (osztálydiagram elkészítése)
- A kész alkalmazás forráskódjának alapján néhány fontosabb forgatókönyv megvalósításának áttekintése

I. A feladat ismertetése

Interaktív FontEditor készítése, amelyben lehet szerkeszteni a karaktereket, és a karakterekből álló fontokból tetszőleges példaszöveg jeleníthető meg. A felhasználói felülete futás közben:

A következő funkciókat kell támogatnia:

- Több betűtípus egyidejű szerkesztése. Ez egyes betűtípusok külön tab oldalakon szerkeszthetők (MDI – Multiple Document Interface).
- Új betűtípus a File/New menüelem kiválasztásával hozható létre (meg kell adni a nevét).
- Ez egyes betűtípusok elmenthetők (File/Save), betölthetők (File/Open), és az aktuális dokumentum bezárható (File/Close). Ezek helye megvan az alkalmazásban, de nincsenek részleteiben implementálva (a függvények törzse nincs kitöltve – opcionális HF).
- A felhasználói felület felépítése
    - Az oldalak tetején egy mintaszöveg adható meg, melyet az aktuális betűtípussal az alkalmazás megjelenít.
    - Az oldalak közepén egy karaktersáv található. Egy adott karakteren duplán kattintva alatta megjelenik egy az adott karakterhez tartozó szerkesztőnézet.
    - Az oldal alján egymás mellett az eddig szerkesztésre megnyitott karakterek szerkesztőnézetei láthatók. Egy karakter többször is megnyitható szerkesztésre, ez esetben több szerkesztőnézet jön létre hozzá. Ennek az az értelme, hogy ugyanazt a karaktert különböző nagyítással is láthatjuk/szerkeszthetjük.
- A szerkesztőnézetek felépítése
    - Nagy része (eltekintve a felső sáv) a szerkesztőfelület, ahol fekete háttéren sárgával jelennek meg az aktív pixelek. Egy adott pixelen az egérrel kattintva a pixel invertálódik.
    - Bal felső sarokban a megjelenített karakter látható

- ’c’ gomb: Clear, minden aktív pixelt töröl
- ’+’ gomb: nagyítás
- ’-’ gomb: kicsinyítés

Futtassuk az alkalmazást, és ismertessük működését a fentieknek megfelelően. Azt mindenképpen mutassuk meg, hogy ha egy karakter szerepel a mintaszövegben, valamint többször megnyitjuk szerkesztésre, akkor az egyik nézetben változtatva (egy pixelt invertálva) valamennyi nézete frissül.

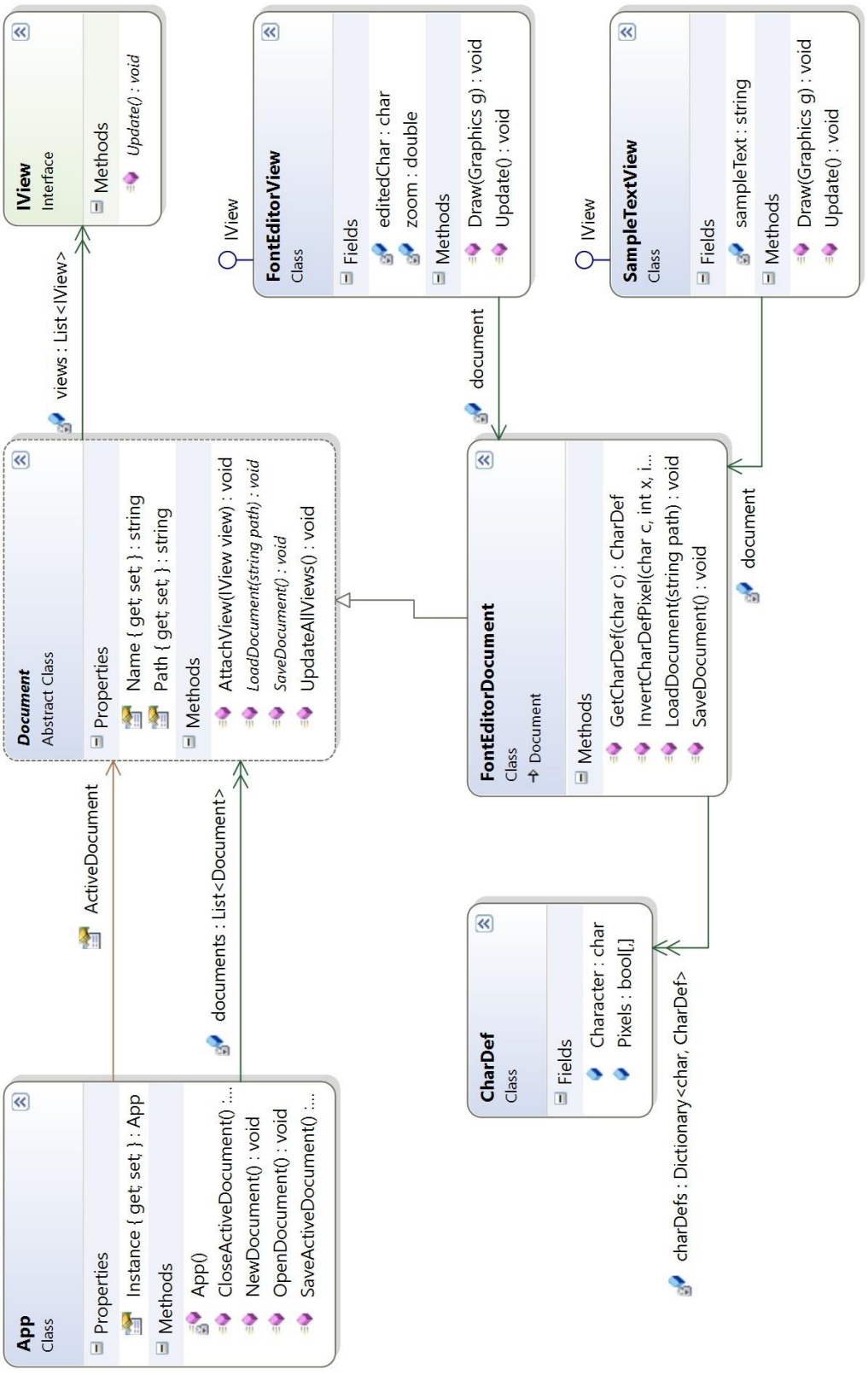
Az alkalmazás a kódmennyiség minimális értéken tartása érdekében minimalisztikus, pl. a hibakezelés nincs általánosságában kidolgozva, hiányoznak ellenőrzések. Ugyanakkor viszonylag jól kommentezett, ami segíti a kód utólagos megértését.

II. Az alkalmazás megtervezése

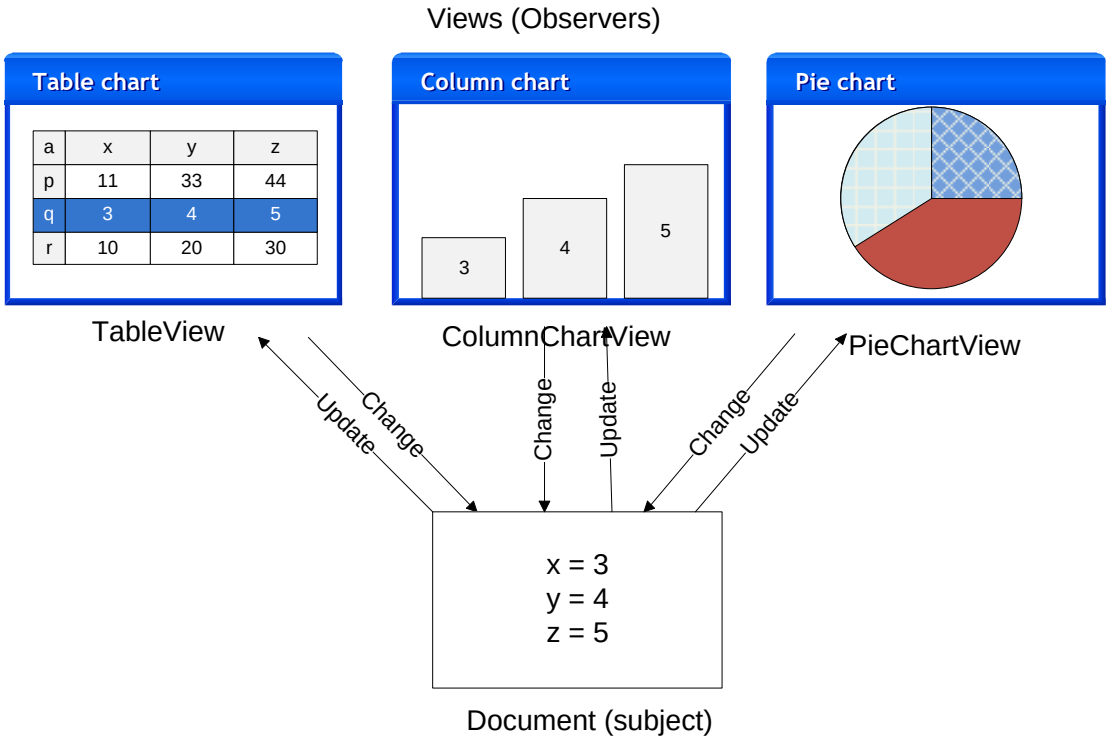
A cél az, hogy a hallgatók átlássák, milyen folyamatot követve, milyen lépésekben dolgozunk, mikor milyen tervezői lépéseket kell meghoznia. **Lényeges, hogy végig vonjuk be a hallgatókat, közösen hozzuk meg a döntéseket**.

Hozzunk létre egy új C# nyelvű Window Form projektet, legyen a neve FontEditor. Vegyünk fel egy osztálydiagramot: projekten jobb katt, Add/New Item, majd a megjelenő ablakban Class Diagram kiválasztása, a neve maradhat az alapértelmezett. Állítsuk be, hogy a diagram mutassa majd a műveletek szignatúráit is (pl. jobb katt a háttéren, Change Members Format/Display Full Signature. A gyakorlat nagy részében ezt a diagramot fogjuk szerkeszteni.

A kész osztálydiagram a következő, eddig fogunk fokozatosan eljutni:


IView
Interface
Methods
Update() : void
views : List<IView>
IView
FontEditorView
Class
Fields
editedChar : char
zoom : double
Methods
Draw(Graphics g) : void
Update() : void
document
IView
SampleTextView
Class
Fields
sampleText : string
Methods
Draw(Graphics g) : void
Update() : void
Document
Abstract Class
Properties
Name { get; set; } : string
Path { get; set; } : string
Methods
AttachView(IView view) : void
LoadDocument(string path) : void
SaveDocument() : void
UpdateAllViews() : void
FontEditorDocument
Class
Document
Methods
GetCharDef(char c) : CharDef
InvertCharDefPixel(char c, int x, i...
LoadDocument(string path) : void
SaveDocument() : void
document
ActiveDocument
documents : List<Document>
CharDef
Class
Fields
Character : char
Pixels : bool[,]
App
Class
Properties
Instance { get; set; } : App
Methods
App()
CloseActiveDocument() :...
NewDocument() : void
OpenDocument() : void
SaveActiveDocument() :...
charDefs : Dictionary<char, CharDef>

Első tervezői döntés: **architektúrát kell választani**. A **D-V egyértelmű választás**: dokumentumokkal dolgozunk, több nézettel, melyeket szinkronban kell tartani. Az alábbi ábra ismertet a működést. Elvileg tudják a hallgatók, talán csak annyit mondjunk el, hogy a nézetek az observerek, a document pedig a subject, aminek változásaira az egyes nézetek fel vannak iratkozva.



A D-V architektúrából adódóan szükségünk lesz dokumentum osztályra, amely a dokumentum adatait tárolja (tagváltozókban), mint pl. a név, elérési út, pixelmátrix. T.f.h. a későbbiekben több dokumentum típust is támogatni kell majd: pl. megnyithatunk egy olyan tabfület, amin BKV járművekhez tudjuk rendelni a betűtípusokat (elektronikus kijelző). Vannak olyan dokumentum adatok, melyek minden dokumentum típusban megjelennek (pl. név, elérési út). **Az egyes dokumentum típusoknak a közös tulajdonságait/műveleteit célszerű egy Document ősosztályba kiszervezni**, hogy ne legyenek duplikálva az egyes dokumentumokat reprezentáló dokumentum osztályokban.

- Vegyük fel a Document osztályt (ő az absztrakt ős)
- Vegyünk fel bele egy Name: string property-t (ez jelenik meg a tabfüleken)
- Vegyünk fel bele egy Path: string property-t (ide fog menteni)

A D-V architektúrából adódóan szükség van egy **nézet interfészre** (egy Update művelettel a nézet értesítéséhez), valamint a dokumentumoknak nyilván kell tartaniuk egy listában a nézeteiket:

- Vegyük fel az IView interfészt
- Vegyünk fel bele egy Update műveletet
- A Document osztályba vegyünk fel egy views:List<IView> mezőt (a Fields-nél). Jobb gombbal kattintsunk a mező nevén a diagramon, és a menüből „Show as collection association” kiválasztása.

- A Document osztályba vegyünk fel az AttachView(view:View): void műveletet, amivel új nézetet lehet beregisztrálni. DetachView-t nem írunk, mert nézetet nem lehet bezárni.

Támogatnunk kell az egyes **dokumentumok tartalmának perzisztálását** (mentés/betöltés). Ezekhez vegyünk fel a Document ősbe a megfelelő műveleteket:

- Document-be a LoadDocument(path:string) felvétele.
- Document-be a SaveDocument() felvétele. Itt nem kell a path paraméter, a dokumentum a Path tagban ezt tárolja.
- Mindkettő legyen absztrakt, hiszen csak az egyes dokumentum leszármazottakban tudjuk megírni: szelektáljuk ki a két műveletet, és a Properties ablakban az Inheritence modifier legyen Abstract.

Az egyes **dokumentumoknak támogatniuk kell a nézeteik frissítését**, ez minden dokumentum típusra közös:

- A Document-be vegyük fel az UpdateAllViews()-t (ez felel meg az Observer minta Notify műveletének).

Szükség van egy **olyan dokumentum típusra, ami a betűtípusok szerkesztéséhez tartozik**, amely a tagváltozóiban nyilvántartja a szükséges adatokat: legyen a neve FontEditorDocument.

- Vegyük fel a FontEditorDocument osztályt
- Származtassuk a Document-ből (Toolbox – Inheritence kapcsolat)
- Ekkor a LoadDocument és SaveDocument műveletekre automatikusan megszületik az override-oló művelet. Ha mégsem lenne így
    - Jelöljük ki az ősben a két műveletet
    - Copy
    - Jelöljük ki a FontEditorDocument osztályt
    - Paste
    - Jelöljük itt ki a két műveletet, és a Properties ablakban a Instance Modifier legyen override

A dokumentumunk tagváltozókban tárolja az adatokat. Gondoljuk át, hogy ezt hogyan célszerű megvalósítani. Lehetne egy háromdimenziós tömb (karakter – x – y), de inkább emeljük ki egy **külön osztályba az egy adott karakter pixeleinek tárolását/menedzselését: vezessük be a CharDef osztályt**. Megjegyzés: azért nem a pixeltömböt használjuk közvetlenül, mert csak egy új osztály bevezetésével van lehetőségünk kifejezetten ide tartozó műveletek bevezetésére.

- Vegyük fel a CharDef osztályt
- CharDef-be Pixels: bool[,] tagváltozó felvétele. Érdekességképpen elmondhatjuk, mi a különbség a kétdimenziós tömb, és a tömbök tömbje között.
- CharDef-be Character:char felvétele: az egyes CharDef osztályok tárolják magukról, hogy mely karakter pixeleit reprezentálják.

A dokumentumnak lesz egy gyűjteménye CharDef objektumokból: minden karakterhez pontosan egy darab. Gondoljuk át, hogy a legcélszerűbb ezt megvalósítani. Az egyes chardefeket a karakterkódjukkal akarjuk címezni: a

Dictionary ideális választás: a karakterkód a kulcs, az hozzá tartozó CharDef pedig az érték.

- CharDef-be: charDefs: Dictionary<char, CharDef> mező felvétele. Jobb katt, Show as collection association…

Az alkalmazásban nyilván kell tartani a megnyitott dokumentumok listáját. Mely osztály felelőssége legyen? Vezessünk be rá egy **alkalmazásszintű osztályt**: legyen a neve **App** (Windows Forms alatt már van Application, nem célszerű ezt választani). Ez lesz az alkalmazásunk „gyökérosztálya”.

- Vegyük fel az App osztályt
- App-ba documents: List<Document> mező felvétele, majd Show as collection association

Gondoljuk végig, hogyan történik majd **egy új dokumentum létrehozása** (mi történik a File/New menüelem kiválasztásakor): be kell kérni a felhasználótól a dokumentum nevét, létre kell hozni egy FontEditorDocument objektumot, fel kell venni a megnyitott dokumentumok listájába, stb.. Ezt a logikát ne tegyük a GUI-ba (menüelem click eseménykezelő): tegyük abba az osztályba, melynek a felelőssége a megnyitott dokumentumok menedzselése, amely tárolja a szükséges adatokat hozzá (dokumentum lista). Legyen ez az App osztályunk feladata, benne vegyük fel a szükséges műveleteket:

- App-ba NewDocument és OpenDocument műveletek felvétele

Most a **dokumentum mentést** gondoljuk végig: a File/Save mindig az aktív dokumentumra vonatkozik. Valakinek **nyilván kell tartani, melyik az aktív dokumentum**: legyen ez az App, hiszen ő tárolja a dokumentumok listáját is.

- A Toolbox-on válasszuk ki az Association kapcsolatot. Az App-ból húzzunk egy nyilat a Document-be. Válasszuk ki az újonnan létrehozott kapcsolatot, és nevezzük át ActiveDocument-re.
- App-ba SaveActiveDocument felvétele (nincs paraméter, void visszatérés)
- App-ba CloseActiveDocument felvétele (nincs paraméter, void visszatérés)

Az App objektumból **értelemszerűen csak egyet kell/szabad létrehozni**, amely a futó alkalmazást reprezentálja. Van még egy problémánk: a File/Save, stb. menüelem click eseménykezelőben el kell érjük ezt az egy objektumot. Meg majd több más helyen is. Jó lenne, ha nem kellene minden osztályban külön elérhetővé tenni (tagváltozó vagy függvényparaméter formájában), hanem **bárhonnan egyszerűen elérhető** lenne. Erre nyújt megoldást a **Singleton tervezési minta**. Egy osztályból csak egy objektumot enged létrehozni, és ahhoz globális hozzáférést biztosít, mégpedig az osztály nevén és egy statikus Instance property-n keresztül, így:
App.Instance.SaveDocument, stb. (célszerű felírni a táblára). Nem valósítjuk meg teljes értékűen, de tegyük meg az alábbiakat:

- App-ba Instance: App property felvétele. Properties ablakban static: true.
- App-ba App művelet felvétele, legyen private (privát ctor).

Az App-pal végeztünk.

A nézetekkel eddig nem foglalkoztunk, ez a következő lépés. Futtassuk (csak mi) a kész alkalmazást, és beszéljük meg, hogy **hány típusú nézetre van szükség**, melyikből hány darab lesz:

- Két TÍPUSÚ nézetre van szükség: az egyik a mintaszöveget jeleníti meg, a másik egy adott karakter szerkesztését teszi lehetővé
- Legyen az előző neve SampleTextView, az utóbbié FontEditorView
- SampleTextViewból mindig egy van (egy adott dokumentumra vonatkozóan), a FontEditorView objektumok igény szerint jönnek létre, 0..n példány létezhet.
- Vegyük fel a két osztályt
- Implementáltassuk velük az IView interfészt (Toolbox/Inheritence kapcsolat). Az Update művelet automatikusan implementálva lesz.

Az **egyes nézetek** a dokumentumaikból „táplálkoznak", a bennük tárolt adatokat jelenítik meg, azokat módosítják. Ehhez, a D-V architektúrának megfelelően **el kell érjék a dokumentumaikat**.

- A SampleTextView és FontEditorView-ban vegyünk fel egy document: FontEditorDocument mezőt (lehet egyikből copy-paste a másikba), majd Show as Association.
  Megjegyzés: azért nem célszerű általános Document típusút felvenni (és az interfészbe felvinni), mert a view-knak a konkrét dokumentum adatait (lásd lejjebb) el kell érniük.

Gondoljuk végig, **milyen adattagokkal rendelkeznek az egyes nézetek**. Ehhez futtassuk az alkalmazást, és nézzük meg ismét a felhasználói felületét.

- A SampleTextView tárolja a mintaszöveget, amit meg kell jeleníteni. Vegyünk fel egy sampleText:string mezőt. Mondjuk el, hogy ha el kellene menteni a mintaszöveget is, akkor a FontEditorDocument-ben kellene tárolni (és onnan mindig lekérdezni), mert az adatok mentéséért a dokumentum osztályunk a felelős.
- A FontEditorView két dolgot tárol
  - A karakter kódja, melynek pixeleit megjeleníti. Vegyünk fel egy editedChar: char mezőt.
  - A nagyítási tényezőt (zoom: double felvétele)

A nézetek maguk felelősek a kirajzolásukért.

- Draw(g:Grapics) felvétele mindkét nézetbe.

A FontEditorDocument-ben egy privát listában van egyelőre jelen a CharDef-ek listája. A nézetek így nem tudják elérni, pedig a megjelenítéshez szükségük lenne rá. **A dokumentumunkban be kell vezessünk olyan műveleteket, melyek a dokumentum által tárolt adatokat a nézetek számára elérhetővé teszik, és lehetőséget biztosítanak a módosításra** is.

- Mindkét nézet el kell érje a megjelenített karakterek pixeleit tároló CharDef objektumokat. Ehhez vezessük be a FontEditorDocument-ben a GetCharDef(c:char):CharDef műveletet. Ezt hosszú távon majd úgy lesz célszerű megvalósítani, hogy a GetCharDef nem az eredeti objektumot adja vissza, hanem annak egy másolatát (clone). Ha az eredetit adná vissza, akkor

a nézetek KÖZVETLENÜL tudnák módosítani a pixelek értékét, ezt mi nem akarjuk (bár a funkciók bővítésével rákényszerülhetünk).

- A FontEditorView-nak képesnek kell lennie egy adott CharDef adott koordinátában levő pixel értékét invertálni (egér kattintáskor). Ehhez vezessük be a FontEditorDocument-ben az InvertCharDefPixel(c:char, x: int, x: int) műveletet. Ha kevés az idő, hagyjuk a paraméterek felvételét, nem építünk rá.

Eljutottunk oda, hogy megterveztük az architektúrát, minden igazán lényeges döntést meghoztunk. Az UML diagram alapján megszületett az osztályok váza, amit természetesen jelentősen bővíteni kell, még születnek új osztályok is (pl. Form-ok, vezérlők).

III. A kész alkalmazás áttekintése

A tapasztalataim szerint az alábbiakra min. 15 perc kell maradjon, abba szűken bele lehet férni.

Idő hiányában nem valósítjuk meg az alkalmazást, hanem a kész megoldást nézzük át, annak is csak néhány lényeges használati esetét (forgatókönyv).

Most már a hallgatók is töltsék le a kész megoldást, nyissák meg a kész solution-t, futtassák/próbálják ki az alkalmazást.

Nézetek megvalósítása

Nyissuk meg a FontEditorView-t, először a kódot nézzük. A FontEditorView egyrészt implementálja az IView interfészt, de másrészt a UserControl-ból származik. Mégpedig azért, mert így a tervezőben (designer) tudjuk kialakítani a felhasználói felületét, pont úgy, mint egy űrlapnak. Nyissuk most meg tervezői nézetben, és mondjuk el, hogy a címkét és gombokat a Toolboxról tettük rá. Rendezzük is át egy kicsit őket, majd futtassuk az alkalmazást (elég, ha mi megtesszük, a hallgatók nem kell kövessék), hogy érezhető legyen, miről van szó.
A SampleTextView is így van megvalósítva, bár annak egyszerű a felülete, lehetett volna közönséges Control leszármazott is.

Vonjuk le a tanulságot: **Windows Forms környezetben a nézeteket tipikusan UserControl-ként (esetleg Control-ként) célszerű megvalósítani**.

Egy oldal (tab) elrendezése

Futtassuk az alkalmazást. Valahogy ki kell alakítsuk egy adott oldal (tabpage) elrendezését. Lehetőleg tervezői nézetben, és nem futás közben, kódból pozícionálva az elemeket (legalábbis ahol nem muszáj). A UserControl-ok alkalmazása jelenti számunkra a megoldást. Nyissuk meg a FontDocumentControl-t tervezői nézetben. Ez egy olyan control, amely egy taboldalra kerül fel, azt tölti ki teljesen. Az oldalt a már ismert layout technikákkal alakítottuk ki (Label, TextBox, Panel-ek Dock-kolva).

Ha van időnk, akkor nézzük meg a Document Outline ablakban. Az igazi „poén” pedig az, hogy a SampleTextView-t is a Toolbox-ról drag&drop-pal tettük fel. Annyit mutassunk meg, hogy tényleg ott van a Toolbox tetején.

Forgatókönyv 1 – Új dokumentum létrehozása

Ha kevés az idő, inkább hagyjuk ki, és a következő (és egyben utolsó) forgatókönyvet beszéljük át.
Azt nézzük meg, hogyan történik egy új dokumentum létrehozása, vagyis mi történik a File/New menüelem kiválasztásakor.

Nyissuk meg a MainForm-ot tervezői nézetben, válaszuk a File/New menüelemet, hogy ugorjunk el a Click eseménykezelőhöz. Arra látunk példát, hogy az App osztály, mint Singleton, hogy érhető el:

```
App.Instance.NewDocument();
```

Az összes többi menüelem eseménykezelője hasonló, nincs semmi logika a GUI-ban, csak egyszerű továbbhívás az App-ba.

Tekintsük át az az App.NewDocument törzsét, és egy-egy mondatban tekintsük át (a gyakorlat során szóban ismertessük) a fontosabb lépéseket. Azt, hogy a TabControl-lal mit ügyeskedünk, nem kell elmondani, nem kell tudni.

```
/// <summary>
/// Létrehoz egy új dokumentumot.
/// </summary>
public void NewDocument()
{
    // Bekérdezzük az új font típus (dokumentum) nevét a
    // felhasználótól egy modális dialógs ablakban.
    NewDocForm form = new NewDocForm();
    if (form.ShowDialog() != DialogResult.OK)
        return;

    // Új dokumentum objektum létrehozása és felvétele a
    // dokumentum listába.
    Document doc = new FontEditorDocument(form.FontName);
    documents.Add(doc);
    // Az első paraméter egy kulcs, a második a tab felirata

    // Egy új tabra felteszi a dokumentumhoz tartozó felületelemeket.
    // Ezeket egy UserControl, a FontDocumentControl fogja össze.
    // Így csak ebből kell egy példányt az új tabpage-re feltenni.
    mainForm.TabControl.TabPages.Add(form.FontName, form.FontName);
    FontDocumentControl documentControl = new FontDocumentControl();
    TabPage tp = mainForm.TabControl.TabPages[form.FontName];
    tp.Controls.Add(documentControl);
    documentControl.Dock = DockStyle.Fill;

    // SampleTextView beregisztrálása a documentnál, hogy
    // értesüljön majd a dokumentum változásairól.
    documentControl.SampleTextView.AttachToDoc(doc);

    // Az új tab legyen a kiválasztott.
    mainForm.TabControl.SelectTab(tp);
```

```
    // Az új tab lesz az aktív, az activeDocument
    // tagváltozót erre kell állítani.
    UpdateActiveDocument();
}
```

Az App.OpenDocument nincs kitöltve, de a lépések be vannak írva, remek gyakorlási lehetőség a hallgatóknak otthon megírni.

Forgatókönyv 2 – egy pixel invertálása, nézetek szinkronizálása

Önálló feladat a hallgatóknak. Keressék meg azt a függvényt, ahol az egész folyamat elindul. A `FontEditorView.FontEditorView_MouseClick`-be kellene eljutni. Itt egy sor a lényeg:

```
private void FontEditorView_MouseClick(object sender, MouseEventArgs
e)
{
    int x = e.X/zoom;
    int y = (e.Y - offsetY)/zoom;
    if (x >= CharDef.FontSize.Width)
        return;

    document.InvertCharDefPixel(editedChar, x, y);
}
```

Nézzük meg a `FontEditorDocument.InvertCharDefPixel`-t. Invertálja a megfelelő CharDef pixelét, de a lényeg az utolsó sor:

```
public void InvertCharDefPixel(char c, int x, int y)
{
    CharDef fd = GetCharDef(c);
    fd.Pixels[x, y] = !fd.Pixels[x, y];
    UpdateAllViews();
}
```

A FontEditorDocument-ben vethetünk még egy pillantást a CharDef-ek szótárára:

```
Dictionary<char, CharDef> charDefs = new Dictionary<char, CharDef>();
```

Az UpdateAllViews a Document ősben van, Update-et hív minden nézetre. Ami érdekes, hogy az Update hogy van megírva az egyes nézetekben. Nézzük meg pl. a FontEditView-t:

```
/// <summary>
/// Az IView interfész Update műveletének implementációja.
/// </summary>
public void Update()
{
    Invalidate();
}
```

Az Update hatására a nézetek újra kell rajzolják magukat az aktuális dokumentum állapot alapján. De az Update-ben nem tudunk rajzolni, csak a „Paint”-ben. Így itt az Invalidate hívással kiváltjuk a Paint eseményt. Ez megint egy tanulság: **Windows Forms alkalmazásokban az Update függvényben tipikusan Invalidate hívás szokott lenni**.

Zárásképpen nézzük meg a `FontEditView.Paint` megvalósítását. Egyetlen lényeges dolog van itt: a megjelenítéshez le kell kérni a dokumentumtól az aktuális CharDef-et (mert a nézet a D-V architektúrának megfelelően nem tárolja).

```
/// <summary>
/// A UserControl.Paint felüldefiniálása, ebben rajzolunk.
/// </summary>
protected override void OnPaint(PaintEventArgs e)
{
    base.OnPaint(e);

    CharDef editedCharDef = document.GetCharDef(editedChar);

    for (int y = 0; y < CharDef.FontSize.Height; y++)
    {
        for (int x = 0; x < CharDef.FontSize.Width; x++)
        {
            e.Graphics.FillRectangle(
                editedCharDef.Pixels[x,y] ? Brushes.Yellow:
                  Brushes.Black,
                zoom * x, offsetY + zoom * y, zoom, zoom);
        }
    }
}
```