# 7. Támogatottság

Az együttes munka egyik legfontosabb eleme a csapattársak megértése. Biztosítani kell egy egyértelmű, jól használható kommunikációs rendszert, csatornát, melyet mindenki hamar el tud sajátítani. Az egyik nagyon fontos indirekt kommunikációs helyzet a kódbázisban írt sorok, kommentek, algoritmusok stb. gyors megértése, felismerése. Egy nagyobb kódbázis esetén ez még megfelelő szabályok, szabványok betartása során sem egy egyszerű feladat.  
Ha ránézünk egy szövegre, akkor rögtön nem fog feltűnni, hogy éppen miről is szól a szöveg, azt csak részletesebb olvasás során tudjuk eldönteni. ennek kiküszöbölésére a kódbázisban egyértelműen kell jelezni, hogy mi micsoda, mi hogyan kapcsolódik a többi elemhez, milyen funkciókat lát el egy adott egység stb. Lehetőleg a kódban rövid, lényegre törő megjegyzések, változó nevek, osztálynevek, algoritmusok szerepeljenek, hogy minimális idő alatt felismerhetőek legyenek a funkciói az elemeknek.  
Erre egy kifejezett módszer a konvenciók alkalmazása. Az évek során jó pár konvenció kialakult, általános és nyelvspecifikusok is, a csapatunkban ezek közül párat vegyesen fogunk használni. A konvencióink leginkább a standard C# kódolási szabályokat-konvenciókat fogják használni, enyhébb módosításokkal.

## 7.1.1 Elnevezési szabályok

Pascal case: Az első betű mindig nagy, illetve ha több szó egy név, akkor is egybe írjuk a szót, és a szavakat azzal különböztetjük meg egymástól, hogy azokat nagybetűvel kezdjük. Tehát például így: „EzEgyPascalCaseSzöveg” „EzIs” „Minta”

Ezt a konvenciót az alábbi helyeket használjuk:

* Osztályok nevei
* Struktúrák nevei
* Interfészek nevei – különlegesebb eset, itt az első betű egy nagy I betű
* Statikus változók nevei
* Enum osztály tagjainak nevei
* Metódusok nevei

Camel case: Az első betű mindig kicsi, illetve ha több szó egy név, akkor is egybe írjuk a szót, és a szavakat azzal különböztetjük meg egymástól, hogy azokat nagybetűvel kezdjük. Tehát például így: „ezEgyCamelCaseSzöveg” „ezIs” „minta”

Ezt a konvenciót az alábbi helyeket használjuk:

* Osztályok változóinak nevei, ha azok nem statikusak
* Struktúrák változóinak nevei
* Valamilyen osztály, vagy saját típus egyedének neve
* Általánosan azoknak a változóknak a nevei, amelyekre más szabályok nem vonatkoznak

Egyéb megemlítendő példák:

* Ha egy osztályon belül sok metódus, illetve változó van, valamint ezeknek a megkülönböztetése nem adott, akkor a változóknak a nevei elé egy „\_” karaktert írhatunk, hogy amikor az IDE automatikusan kitölti az adott osztályon belül publikusan elérhető tagoknak neveit, akkor a „\_” karakter beírásával elérhetőek legyenek a változók. Ez esetben másnak ne legyen a neve előtt a „\_” karakter.
* Ha kifejezetten hosszú egy adott név, akkor a szavakat el lehet választani „\_” karakterrel. Ezt lehetőleg mellőzzük, mert nem ez a megszokott elnevezési konvenció a jelenlegi esetben.
* Használhatunk „namespace”-eket, ha elakarjuk kerülni a kompatibilitási problémákat, vagy ha el akarunk különíteni egy egységet mégjobban.

## 7.1.2 Kinézet, strukturáltság konvenciók

Talán az egyik legfontosabb követendő konvenció. Egy kódbázisban egy kifejezetten jól átlátható rész szükséges ahhoz, hogy könnyedén megértsük azt, amit látunk. A jól strukturált kód segíthet abban, hogy pillanatok alatt javíthatóak legyenek egyes elemek, illetve abban, hogy hamar megláthassuk, hogy hova is akarunk nézni, mit is kellene keresnünk. Így lényegében mondható az is, hogy a szíve az egész konvenció szabályozásnak a strukturáltság. A jó hír az, hogy ezt egyszerűen be tudjuk tartani, nem kell megjegyezni sok mindent ahhoz, hogy igényesen legyen kidolgozva ez a rész.

Az erre vonatkozó szabályok a következők:

* Egy sorban egy deklarálás legyen, több változót több sorban deklaráljunk, inicializáljunk stb.
* Egy sornyi kód egy dolgot csináljon. Ne legyenek a sorok egybeírva, ne akarjunk több mindent egy sorban letudni. Erre tökéletes példa a .Net keretrendszerben a Linq használata, mellyel egy sorban rengeteg mindent el tudunk végezni. Ezt optimalizálás miatt sem érdemes elkövetni, illetve nehezen kivehető, hogy mit is csinál az adott sor. Illetve a Visual Studioban a C# nyelvet használva megtehetjük azt is, hogy egy sorban „;” vel elválasztva írunk több dolgot, ezt SZIGORÚAN KERÜLJÜK el. Tehát egy sor, egy „művelet” (ez nem feltétlenül jelenti azt, hogy minden összeadást stb. elemi műveletet külön sorba kell írni.).
* Ha egy sorban nem tudunk leírni egy adott parancsot, akkor írjuk inkább többe, ha lehetséges. Ha ez nem lehetséges, illetve a helyzet nem indokolja, akkor használjunk tördelést. A tördelés során figyeljünk arra, hogy a következő sorban lévő dolgok ott kezdődjenek, ahol az előző sorban is kezdődtek. Ehhez használjunk TAB-ot.
* Legyenek elkülöníthetőek a metódusok, és a változók helyei. Két változó közé ne deklaráljunk egy metódust, próbáljuk tartani a „top-down” szabályt, azaz a változók deklarálása legyen fentebb, mint a metódusoké.
* Ahol általában „{” és „}” zárójeleket használhatunk, ott használjuk is. Ennek módja az, hogy a nyitó zárójel nem az adott névvel egy sorban van, hanem egy külön sorban kezdődik. A csukó zárójel esetében is hasonlóan egy külön sorban legyen a blokk lezárásaként. A két zárójeles sorban lehetőleg más ne legyen.  
  Ez alól kivételt képeznek azon esetek, ahol több kis kódrészletet kell írni, például több „if” feltétel esetén, de itt is inkább törekedjünk a kapcsos zárójelek használatára.
* Ezeken kívül használjuk a Visual Studio által nyújtott formázási segítséget, melyet alapértelmezetten a CTRL + K + CTRL + D paranccsal érhetünk el, amely az egész dokumentumot a megszokott szabályok szerint formázza (ez a formázás nem feltétlenül fogja használni az előbb leírtakat, leginkább az adott sorokban lévő egybecsúszó szövegeket tördeli szét).

## 7.1.3 Kommentezés konvenciók

A kommentek is egy kulcs szerepet töltenek be az adott egység megértésében. Ezen eszköz segítségével tudjuk a dokumentum elején, illetve annak sorai közben jelezni másoknak, illetve magunknak, hogy miről is szól egy-egy rész. Így erre is vonatkozik pár szabály, a még tökéletesebb átláthatóság érdekében.

* A kommenteket lehetőleg külön sorokba írjuk, a leírni, magyarázni kívánt elemek előtt. Ha ez nem megfelelő az adott helyzetben, akkor az adott sorban egy jó távolságra írjuk a kommentet
* A kommenteket nagybetűvel kezdjük, azokat a standard írásos szabályoknak feleltessük meg.
* A kommentet a „//” karakterek után egy „space” karakter után kezdjük. Tehát pl. így: // Minta
* Használjunk kommenteket ott, ahol azt a helyzet megkívánja. Ne használjunk túl sok kommentet, mert az többet árthat, mint amit nyújt. Lehetőleg minden dokumentumot kezdjünk pár sornyi kommenttel, ami összefoglalja, hogy mit is csinál az adott egység.

## 7.2 Kódolási szabályok

Kódolási szabályok alatt értendőek azok az irányelvek, amelyeket a programozóknak ajánlott betartani. Nem lehet biztosan eldönteni általánosítva, hogy egy adott helyzet milyen kódolási szabályt kíván. Így ezekről a programozóknak kell dönteni és a lehető legjobb tudásuk alapján implementálni azokat.  
Pár említésre méltó irányelv:

* Ahol lehet használjuk fel az általunk már megírt kódokat. Ne írjunk pontosan ugyanazon algoritmus megvalósítására alkalmas kódot még egyszer, inkább próbáljuk az előzőket felhasználni.
* Ha a helyzet nem kívánja (itt értendő pl.: optimalizálás), akkor használjuk a beépített rutinokat, metódusokat, melyeket a nyelv biztosít. A könnyebb fenntarthatóság érdekében inkább használjunk könyvtárakat, általánosabban felhasználható kódokat, mintsem írjuk meg újra és újra ugyanazt.
* Használjuk a megfelelő módszereket az adott feladatok elvégzésére. Egy tipikus példa erre az „if” és a „switch-case” esete.
* A lehető legoptimálisabb algoritmust használjunk, főleg olyan esetekben, ahol erősen szükséges.

## 7.3 Karbantartás

A játék karbantartását segíteni fogja egy „log”, amely a játék futása során készül. Ezt a logot küldhetik majd el a játékosok a fejlesztőinknek, esetleges hibák, összeomlások esetén.

Ezen kívül elérhető lesz a fejlesztőknek az egész projekt verzió története a verziókezelő alkalmazásban (github). Minden esetleges fejlesztési komplikáltság ezen keresztül nyomon követkető, illetve vissza állítható.

Annak érdekében, hogy a legjobban izolálhatóak legyenek a problémák, a játék verzióiról biztonsági mentések készülnek, melyek a stabil verziókra való visszaállítás során lehet szükség. Ez egy központi szervergépen lesz elérhető a fejlesztők számára. Ennek a biztonsági mentésnek a frekvenciájáról a vezető programozó dönt.

* csak a karbantartóknak, illetve üzemeltetőknek szükséges funkciók,
* a működés során keletkező naplók (a későbbi hibakeresés vagy finomhangolás elősegítésére),
* a karbantartók munkáját segítő segédalkalmazások (például a naplók kiértékelésére).

# 8. Tervezési korlátozások

Design Constrainst

Itt kell felsorolni egy-egy alpontban minden olyan korlátozást, amely a tervezési, esetleg az implementációs munkát befolyásolja. Ilyenek lehetnek például:

* a kívánatos architektúrára vonatkozó előírások (például milyen részekből [„tier”] kell felépíteni az alkalmazást),
* milyen programozási nyelv(ek)et kell használni,
* milyen fejlesztőeszközt kell használni,
* milyen újra felhasználható elemek építhetők be az alkalmazásba (komponens- és/vagy osztálykönyvtárak, kész kódok stb.),
* milyen fejlesztési módszertant, hogyan kell használni.

# 9. On-line dokumentáció és Help rendszer

Online User Documentation and Help System Requirements

Kellenek-e ilyenek, ha igen, a vonatkozó információk.

Szótár:

namespace – névtér, a C# programnyelv egy szintaxisa