5. tétel

**Verziókezelés:**

**Verziókezelés** alatt több verzióval rendelkező adatok kezelését értjük. Leggyakrabban a mérnöki tudományokban és a [szoftverfejlesztésben](https://hu.wikipedia.org/wiki/Szoftverfejleszt%C3%A9s) használnak verziókezelő rendszereket fejlesztés alatt álló dokumentumok, tervek, forráskódok és egyéb olyan adatok verzióinak kezelésére, amelyeken több ember dolgozik egyidejűleg. Az egyes változtatásokat verziószámokkal vagy verzióbetűkkel követik nyomon.

**Kezelési modellek:**

A hagyományos verziókezelők központosított modellel dolgoznak, ahol minden verziókezelési művelet egy közösen használt szerveren történik. Ha két fejlesztő egyidejűleg próbálja meg módosítani valamelyik fájlt, akkor valami módon el kell kerülni azt, hogy a két személy felülírja egymás munkáját. Az ilyen (centralizált) rendszerek kétféleképpen oldják meg ezt a problémát: zárolással és összefésüléssel.

**Zárolás (Lock):**

A konkurens hozzáférés kezelésének legegyszerűbb módja, ha megtiltjuk a konkurens hozzáférést, azaz ha egy valaki már elkezd módosítani egy fájlt, akkor azt már más felhasználó nem nyithatja meg írásra. Ezt hívják elterjedt kifejezéssel *lock*-olásnak, a magyarosabb, de kevésbé elterjedt zárolás szó helyett. Ha egy felhasználó kivesz *(kicsekkel)* egy fájlt, akkor a többi felhasználó már csak olvasásra nyithatja meg azt egészen addig, amíg a kicsekkelő felhasználó visszateszi (becsekkeli) a módosított változatot (vagy elveti a módosítást).

Ennek a módszernek előnyei és hátrányai is vannak. A nagyobb vagy sok fájlt érintő változtatásoknál célszerű ezt választani, mert bonyolult összefésülési műveleteket lehet megtakarítani vele. Ha azonban egy fájl túl sokáig zárolt állapotban marad, akkor a többi fejlesztő esetleg arra kényszerülhet, hogy a verziókezelést megkerülve a fájl lokális másolatát módosítsa, ami nagyobb bonyodalmakhoz vezethet.

**Összefésülés (merge):**

Itt is az angol szóhasználat az elterjedtebb a magyarosabb *összefésülés* helyett. A legtöbb verziókezelő, például a [CVS](https://hu.wikipedia.org/w/index.php?title=CVS&action=edit&redlink=1) is, lehetővé teszi, hogy több felhasználó dolgozzon egyidejűleg ugyanazon a fájlon. Ekkor a saját változtatását elsőként becsekkelő felhasználó mindenképpen sikerrel fog járni. A rendszer a többi felhasználónak összefésülési lehetőséget ad, mellyel a különböző módosítások összeolvaszthatóak, így a felhasználók nem írják felül egymás munkáját. Az összefésülés lehet automatikus vagy kézi.

Általában az összefésülésre képes verziókezelők is adnak lehetőséget fájlok egyfelhasználós, kizárólagos szerkesztésére *reserved edit* néven.

**Elosztott verziókezelő rendszerek:**

Szemben a kliens-szerver modellel, az elosztott verziókezelők decentralizált rendszerek. Itt egy központi tároló (angolul *repository*) helyett minden felhasználó gépe egy-egy külön tárolóként jelenik meg.[[2]](https://hu.wikipedia.org/wiki/Verzi%C3%B3kezel%C3%A9s#cite_note-Wheeler-2) A szinkronizáció az egyes gépek között küldött [patch](https://hu.wikipedia.org/wiki/Patch)-ek (módosításcsomagok) által valósul meg. Ez a megközelítés jelentős változásokat okoz:

* Nincs nagy központi adatbázis, csak munkamásolatok vannak.[[2]](https://hu.wikipedia.org/wiki/Verzi%C3%B3kezel%C3%A9s#cite_note-Wheeler-2)
* A gyakori műveletek, mint a becsekkelés, verziótörténet böngészés és a változtatások visszaállítása gyorsak, mert nem kell központi szerverrel kommunikálni.[[3]](https://hu.wikipedia.org/wiki/Verzi%C3%B3kezel%C3%A9s#cite_note-OSullivan-3)
* Minden munkamásolat egy-egy backup, ami természetes védelmet ad az adatvesztés ellen.[[3]](https://hu.wikipedia.org/wiki/Verzi%C3%B3kezel%C3%A9s#cite_note-OSullivan-3)

Két fajta elosztott verziókezelő létezik, a nyitott és a zárt. A nyitott rendszereket inkább nyílt forráskódú termékeknél használják, a zártakat inkább a nem nyilvános forráskódú termékeknél.

**Nyitott rendszerek:**

A nyitott, elosztott verziókezelők támogatják különböző ágak létezését, és erősen függenek a fent tárgyalt összefésülés (merge) művelettől. Általános jellemzőik a következők:

* Minden munkamásolat gyakorlatilag egy ág.
* Minden ág egy-egy munkamásolatként implementálódik. Az ágak összefésülése patch-ek küldözgetésével történik.
* Lehet válogatni az egyes változtatások között, nem kell feltétlenül minden változtatást letölteni.
* Új tagok bármikor csatlakozhatnak a rendszerhez, nincs szükség szerveroldali regisztrációra.

Az egyik első nyitott rendszer a [BitKeeper](https://hu.wikipedia.org/w/index.php?title=BitKeeper&action=edit&redlink=1) volt, mely azért is nevezetes, mert a [Linux-rendszermag](https://hu.wikipedia.org/wiki/Linux-rendszermag) fejlesztéséhez is használták. Később a BitKeeper fejlesztői megváltoztatták a licencet, így a Linux fejlesztők más, szabad verziókezelő után kezdtek nézni.[[4]](https://hu.wikipedia.org/wiki/Verzi%C3%B3kezel%C3%A9s#cite_note-4) Néhány szabadon használható, nyitott verziókezelő rendszer:

* [Cvs](https://hu.wikipedia.org/wiki/Cvs)
* [Git](https://hu.wikipedia.org/wiki/Git)
* [Apache Subversion](https://hu.wikipedia.org/wiki/Apache_Subversion)

**Zárt rendszerek:**

A zárt, eloszott verziókezelők [adatbázis replikáción](https://hu.wikipedia.org/w/index.php?title=Adatb%C3%A1zis_replik%C3%A1ci%C3%B3&action=edit&redlink=1) alapulnak. Csak egy *baseline* van, minden becsekkelt változás ebbe kerül bele. Egyik ilyen szoftver a [Code Co-op](https://hu.wikipedia.org/w/index.php?title=Code_Co-op&action=edit&redlink=1).

**Integráció:**

A fejlettebb verziókezelők további lehetőségeket is kínálnak, melyek lehetővé teszik az integrációt más eszközökkel. Különböző [IDE](https://hu.wikipedia.org/wiki/Integr%C3%A1lt_fejleszt%C5%91i_k%C3%B6rnyezet)-khez ([IntelliJ IDEA](https://hu.wikipedia.org/wiki/IntelliJ_IDEA), [Eclipse](https://hu.wikipedia.org/wiki/Eclipse) és [Visual Studio](https://hu.wikipedia.org/wiki/Visual_Studio)) gyakran letölthetőek különböző verziókezelői [pluginok](https://hu.wikipedia.org/w/index.php?title=Plugin&action=edit&redlink=1). A [NetBeans](https://hu.wikipedia.org/wiki/NetBeans) IDE-t beépített verziókezelővel szállítják.

**Fogalmak:**

* **Baseline**

Egy dokumentum vagy fájl jóváhagyott verziója, melyhez az azt követő változtatásokat viszonyítják.

* **Branch (ág)**

A verziókezelt fájlok egy részhalmaza elágazhat, így azoknak több aktuális változatuk lesz egyidejűleg, melyeket akár különböző sebességgel és különböző irányokba is fejleszthetnek.

* **Check-out**

Lokális másolat készítése valamely verziókezelt fájlról. Alapértelmezésben ilyenkor a legfrissebb verziót kapja a felhasználó, de általában van lehetőség konkrét verzió kikérésére is verziószám alapján.

* **Check-in, Commit vagy Submit**

Az a művelet, amikor a lokális példány változtatásai beíródnak (vagy egyszerű másolás vagy összefésülés eredményeként) a szerveren tárolt változatba.

* **Conflict**

Konfliktusról akkor beszélünk, ha ketten akarnak megváltoztatni egy dokumentumot vagy fájlt és a rendszer nem képes összeépíteni a változásokat. A felhasználónak ekkor fel kell oldania a konfliktust, amit vagy úgy tehet meg, hogy a változtatásokat összekombinálja vagy úgy, hogy kiválasztja az egyik változtatást és csak azt juttatja érvényre.

* **Change**

Egy változtatás (**change**, **diff** vagy **delta**) mindig egy verziókezelt dokumentumon vagy fájlon tett változtatást jelenti. Rendszerfüggő, hogy milyen mértékű módosítások számítanak *change*-nek.

* **Change list**

Egy **change list** vagy **change set** egy check-in művelet során bevitt változtatások listája, olyan rendszereken, melyek támogatják [atomi műveletként](https://hu.wikipedia.org/w/index.php?title=Atomi_m%C5%B1velet&action=edit&redlink=1) több változás egyidejű becsekkelését.

* **Dynamic stream**

Egy olyan adatszerkezet, amely egy adott tárolón lévő elemek konfigurációját reprezentálja, és időben változik.

* **Export**

Az export a checkouthoz hasonlít azzal a különbséggel, hogy tiszta könyvtárat csinál a verziókezeléshez szükséges metaadatok nélkül. Ezt a műveletet általában közvetlenül a tartalom publikálása előtt szokták használni.

* **Head**

A legutóbbi checkin.

* **Import**

Az import művelettel lehet egy lokálisan tárolt adathalmazt, amely még nem munkamásolat, felmásolni a tárolóra és verziókontroll alá helyezni.

* **Mainline**

Hasonlít a *trunk*-hoz, de minden ágnak lehet saját *mainline*-ja.

* **Merge**

A *merge* művelettel két változtatáslistát lehet összefésülni, s ezáltal egy közös verziót létrehozni. Erre a következő esetekben lehet szükség:

Ha egy felhasználó módosítja a saját munkamásolatát, majd letölt a szerverről egy másik módosított változatot. Ekkor a szerveren lévő változásokat össze kell fésülni a lokális munkapéldány változásaival a kliensen.

Ha a fejlesztésben elágazás történt, majd egy hibát kijavítottak valamely ágban, s a javítást alkalmazni kell a másik ágra is.

Ha a fejlesztésben elágazás történt, majd az ágakat különböző irányba fejlesztettek tovább, s a különböző fejlesztéseket össze kell vonni egy közös változatba (*trunk*-ba).

* **Repository**

A **repository**, **depot** vagy **tároló** az a hely (tipikusan egy szerver), ahol az aktuális és a korábbi verziók tárolódnak.

* **Reverse integration**

Az egyes ágak összedolgozása és bedolgozása a verziókezelő fő *trunk*-jába.

* **Revision**

A **revision** szó ugyanazt jelenti, mint a **version**. Egy verzió.

* **Tag**

A **tag**, **label** vagy **címke** egy fontos időpillanatot jelöl. Egy adott fájlcsoporthoz hozzárendelhető egy címke, amely beszédes, felhasználóbarát nevet vagy verziószámot adhat a csoportnak.

* **Trunk**

A fejlesztés egyik olyan vonala, amely nem *branch*.

* **Resolve**

Változási konfliktusok feloldására irányuló felhasználói tevékenység.

* **Update**

Az **update** vagy **sync** a repositoryban lévő változtatásokat dolgozza bele a felhasználó munkamásolatába.

* **Working copy**

Magyarul munkamásolat. A repository fájljainak másolata a felhasználó lokális gépén. Minden olyan munka, ami bekerül a repository-ba, először mindig egy munkamásolatban történik meg, innen a neve. Fogalmilag a munkamásolat egy [homokozó](https://hu.wikipedia.org/wiki/Homokoz%C3%B3_(biztons%C3%A1g)).

**Szoftvertesztelési alapfogalmak**

**A tesztelés alapelvei**

A tesztelés alapjait a következő alapelvekben foglalhatjuk össze:

1. A tesztelés hibák jelenlétét jelzi: A tesztelés képes felfedni a hibákat, de azt nem, hogy nincs hiba. Ugyanakkor a szoftver minőségét és megbízhatóságát növeli.
2. Nem lehetséges kimerítő teszt: Minden bemeneti kombinációt nem lehet letesztelni (csak egy 10 hosszú karakterláncnak 256^10 lehetséges értéke van) és nem is érdemes. Általában csak a magas kockázatú és magas prioritású részeket teszteljük.
3. Korai teszt: Érdemes a tesztelést az életciklus minél korábbi szakaszában elkezdeni, mert minél hamar találunk meg egy hibát (mondjuk a specifikációban), annál olcsóbb javítani. Ez azt is jelenti, hogy nemcsak programot, hanem dokumentumokat is lehet tesztelni.
4. Hibák csoportosulása: A tesztelésre csak véges időnk van, ezért a tesztelést azokra a modulokra kell koncentrálni, ahol a hibák a legvalószínűbbek, illetve azokra a bemenetekre kell tesztelnünk, amelyre valószínűleg hibás a szoftver (pl. szélsőértékek).
5. A féregirtó paradoxon: Ha az újratesztelés során (lásd később a regressziós tesztet) mindig ugyanazokat a teszteseteket futtatjuk, akkor egy idő után ezek már nem találnak több hibát (mintha a férgek alkalmazkodnának a teszthez). Ezért a tesztjeinket néha bővíteni kell.
6. A tesztelés függ a körülményektől: Másképp tesztelünk egy atomerőműnek szánt programot és egy beadandót. Másképp tesztelünk, ha a tesztre 10 napunk vagy csak egy éjszakánk van.
7. A hibátlan rendszer téveszméje: Hiába javítjuk ki a hibákat a szoftverben, azzal nem lesz elégedett a megrendelő, ha nem felel meg az igényeinek. Azaz használhatatlan szoftvert nem érdemes tesztelni.

**Tesztelési szintek:**

**Egységtesztek**

A komponensteszt csak a rendszer egy komponensét teszteli önmagában. Az integrációs teszt kettő vagy több komponens együttműködési tesztje. A rendszerteszt az egész rendszert, tehát minden komponenst együtt, teszteli. Ez első három teszt szintet együttesen fejlesztői tesztnek hívjuk, mert ezeket a fejlesztő cég alkalmazottai vagy megbízottjai végzik. Az átvételi teszt során a felhasználók a kész rendszert tesztelik. Ezek általában időrendben is így követik egymást.

A komponensteszt a rendszer önálló részeit teszteli általában a forráskód ismeretében (fehér dobozos tesztelés). Gyakori fajtái:

1. unit-teszt,
2. modulteszt.

A unit-teszt, vagy más néven egységteszt, a metódusokat teszteli. Adott paraméterekre ismerjük a metódus visszatérési értékét (vagy mellékhatását). A unit-teszt megvizsgálja, hogy a tényleges visszatérési érték megegyezik-e az elvárttal. Ha igen, sikeres a teszt, egyébként sikertelen. Elvárás, hogy magának a unit-tesztnek ne legyen mellékhatása.

A unit-tesztelést minden fejlett programozási környezet (integrated development environment, IDE) támogatja, azaz egyszerű ilyen teszteket írni. A jelentőségüket az adja, hogy a futtatásukat is támogatják, így egy változtatás után csak lefuttatjuk az összes unit-tesztet, ezzel biztosítjuk magunkat, hogy a változás nem okozott hibát. Ezt nevezzük regressziós tesztnek.

A modulteszt általában a modul nem-funkcionális tulajdonságát teszteli, pl. sebességét, vagy, hogy van-e memóriaszivárgás (memory lake), van-e szűk keresztmetszet (bottleneck).

A KÖVETEKEZŐKET NEM MUSZÁJ MONDANI, CSAK HA OLYAT KÉRDEZNEK

• Legalacsonyabb szint, legkisebb önállóan tesztelhető egység - osztály, függvény, stb.

• Elszigetelten fut

• "Stub" és "Driver"

• Automatikus coverage mérés toolokkal

• Funkcionális és nem funkc.

• TDD

**Példák:**

• Strlen metódus tesztelése ismert hosszú sztringekkel

• Áfa számolása az adott termekre (preparált adatok, nem valódi adatbázis hozzáférés)

**Integrációs tesztek (– alacsony szint)**

Az integrációs teszt során a komponensek közti interfészeket, az operációs rendszer és a rendszer közti interfészt, illetve más rendszerek felé nyújtott interfészeket tesztelik. Az integrációs teszt legismertebb típusai:

1. Komponens – komponens integrációs teszt: A komponensek közötti kölcsönhatások tesztje a komponensteszt után.
2. Rendszer – rendszer integrációs teszt: A rendszer és más rendszerek közötti kölcsönhatásokat tesztje a rendszerteszt után.

Az integrációs teszt az összeillesztés során keletkező hibákat keresi. Mivel a részeket más-más programozók, csapatok fejlesztették, ezért az elégtelen kommunikációból súlyos hibák keletkezhetnek. Gyakori hiba, hogy az egyik programozó valamit feltételez (pl. a metódus csak pozitív számokat kap a paraméterében), amiről a másik nem tud (és meghívja a metódust egy negatív értékkel). Ezek a hibák kontraktus alapú tervezéssel (design by contract) elkerülhetőek.

Az integrációs teszteket érdemes minél hamarabb elvégezni, mert minél nagyobb az integráció mértéke, annál nehezebb meghatározni, hogy a fellelt hiba (általában egy futási hiba) honnan származik. Ellenkező esetben, azaz amikor már minden komponens kész és csak akkor tesztelünk, akkor ezt a „nagy bumm tesztnek” (big bang tesztnek) nevezzük, ami rendkívül kockázatos.

A KÖVETKEZŐKET NEM MUSZÁJ MONDANI, CSAK HA OLYAT KÉRDEZNEK

• Integration testing in small

• Interfészek vizsgálata (kompatibilitás)

• Adatok áramlása (normál flow, hiba kezelés(null, stb.))

• Inkrementális megközelítés (előnyök, hátrányok)

• Big-bang megközelítés

• Fentről lefele megközelítés (stubs)

• Lentről felfele megközelítés (drivers)

**Példák:**

• UI – UI -> Egy új „doboz” a már meglevő felületen (pl. kupon használat)

• UI - BE -> Bejelentkezés működik

• BE – BE -> Kupon alkalmazása a kosárban levő termékekre

• DB – BE -> Áfa számolása valódi adatokkal

NEM MUSZÁJ MONDANI, CSAK HA KÉRDEZIK **🡪** **Integrációs tesztek - magas szint**

• Integration testing in large

• Sok komponens integrációja

• Lényegében ugyanazok elmondhatók, mint az alacsony szintnél

• Komplett rendszerek integrációja (ilyen értelemben ez akár a rendszer teszt után is jöhet – rendszer integrációs tesztek terminológia)

**Példák:**

• Rendelés visszaigazoló email küldése

• Tetszikelők megjelenítése a webshopunk oldalsávján

• Szerződött partner termekéinek listázása pillanatnyi árakkal

• A webshopomat megnyitom egy új böngészőben

• Az asztali alkalmazásomat elindítom egy új platformon

**Rendszer tesztek**

• A tesztelők területe

• A teljes rendszert vizsgálja

• Alapja a specifikáció -> verifikáció

• Az utolsó fázis hibakeresésre

• Tesztelői függetlenség elvárható

• Funkcionális és nem funkcionális

• Valódi környezet es működés szimulálása elengedhetetlen

• Smoke teszt – honlap átalakítása

• Sanity teszt – bank kártyás fizetési opciók bővítése

**Példák:**

• Teljes megrendelés leadása fizetéssel, email megérkezik

• Szabályosan leadott rendelés törlése, email megérkezik

**Átvételi tesztek**

Az átvételi teszt hasonlóan a rendszerteszthez az egész rendszert teszteli, de ezt már a végfelhasználók végzik. Az átvételi teszt legismertebb fajtái a következők:

1. alfa teszt,
2. béta teszt,
3. felhasználói átvételi teszt,
4. üzemeltetői átvételi teszt.

Az alfa teszt a kész termék tesztje a fejlesztő cégnél, de nem a fejlesztő csapat által. Ennek része, amikor egy kis segédprogram több millió véletlen egérkattintással ellenőrzi, hogy össze-vissza kattintgatva sem lehet kifektetni a programot.

Ezután következik a béta teszt. A béta tesztet a végfelhasználók egy szűk csoportja végzi. Játékoknál gyakori, hogy a kiadás előtt néhány fanatikus játékosnak elküldik a játékot, akik rövid alatt sokat játszanak vele. Cserébe csak azt kérik, hogy a felfedezett hibákat jelentsék.

Ezután jön egy sokkal szélesebb béta teszt, amit felhasználói átvételi tesztnek nevezünk. Ekkor már az összes, vagy majdnem az összes felhasználó, megkapja a szoftvert és az éles környezetben használatba veszi. Azaz installálják és használják, de még nem a termelésben. Ennek a tesztnek a célja, hogy a felhasználók meggyőződjenek, hogy a termék biztonságosan használható lesz majd éles körülmények között is. Itt már elvárt, hogy a fő funkciók mind működjenek, de előfordulhat, hogy az éles színhelyen előjön olyan környezet függő hiba, ami a teszt környezetben nem jött elő. Lehet ez pl. egy funkció lassúsága.

Ezután már csak az üzemeltetői átvételi teszt van hátra. Ekkor a rendszergazdák ellenőrzik, hogy a biztonsági funkciók, pl. a biztonsági mentés és a helyreállítás, helyesen működnek-e.

A KÖVETKEZŐKET NEM MUSZÁJ MONDANI, CSAK HA OLYAT KÉRDEZNEK

• Azt gondoljuk, hogy a szoftver készen van, vagy az ügyfél ismeri és elfogadta az imitációkat

• Célja a rendszer elfogadtatása - validáció

• Minden követelménynek megfelelünk? (megállapodás, jogi, etikai, stb.)

• Az ügyfél mindenképp jelen van, sőt, néha csak ő

• Nem funkcionális - Használhatóság

• Mindig élesben vagy azzal egyezőnek tekinthető környezeten zajlik

• Egyedi / COTS

• Alpha, Beta, RC

• UAT -> megfelelés a felhasználónak

• OAT -> üzemeltethetőség

**Példák:**

• Felhasználó leültetése, rendeljen egy terméket

• Az ügyfél termék támogatási szakemberei üzemeljék be a webshopot egy új szerveren

## Tesztelési technikák

A tesztelési technikákat csoportosíthatjuk a szerint, hogy a teszteseteket milyen információ alapján állítjuk elő. E szerint létezik:

1. Feketedobozos (black-box) vagy specifikáció alapú, amikor a specifikáció alapján készülnek a tesztesetek.
2. Fehérdobozos (white-box) vagy strukturális teszt, amikor a forráskód alapján készülnek a tesztesetek.

Tehát beszélünk feketedobozos tesztelésről, amikor a tesztelő nem látja a forráskódot, de a specifikációkat igen, fehérdobozos tesztelésről, amikor a forráskód rendelkezésre áll.

A feketedobozos tesztelést specifikáció alapúnak is nevezzük, mert a specifikáció alapján készül. Ugyanakkor a teszt futtatásához szükség van a lefordított szoftverre. Leggyakoribb formája, hogy egy adott bemenetre tudjuk, milyen kimenetet kellene adni a programnak. Lefuttatjuk a programot a bemenetre és összehasonlítjuk a kapott kimenetet az elvárttal. Ezt alkalmazzák pl. az ACM versenyeken is.

A fehérdobozos tesztelést strukturális tesztelésnek is nevezzük, mert mindig egy már kész struktúrát, pl. program kódot, tesztelünk. A strukturális teszt esetén értelmezhető a (struktúra) lefedettség. A lefedettség azt mutatja meg, hogy a struktúra hány százalékát tudjuk tesztelni a meglévő tesztesetekkel. Általában ezeket a struktúrákat teszteljük:

1. kódsorok,
2. elágazások,
3. metódusok,
4. osztályok,
5. funkciók,
6. modulok.

Például a gyakran használt unit-teszt a metódusok struktúra tesztje.

**Objektum orientált tervezési alapelvek (GoF, Solid) – PDF fájlként a kinti mappában**

**Függőség befecskendezés – PDF fájlként a kinti mappában**

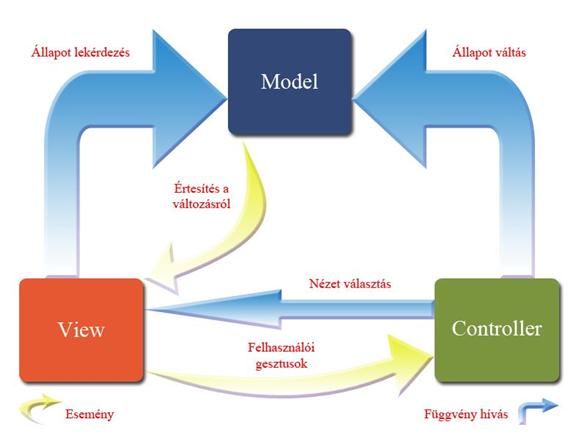
**Architekturális minták**

Az architektúra a program azon része, ami nem változik az idő során, vagy ha változik, akkor az nagyon nehezen kivitelezhető. Talán egy szívátültetéshez vagy agyműtéthez hasonlítható.

#### **MVC – Model-View-Controller**

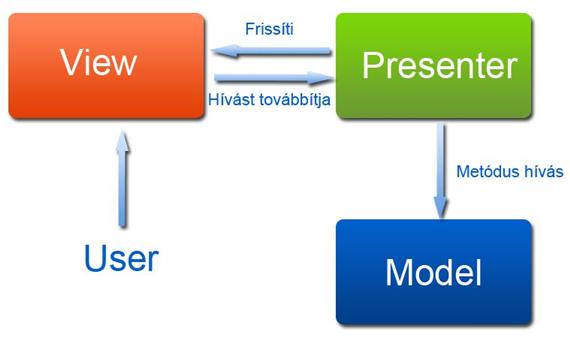
Az MVC minta talán az első tervezési minta. A nevét a három fő komponensének nevéből kapta:

1. Model (magyarul modell): Az adatokat kezelő, vagyis tulajdonképpen az üzleti logikát megvalósító réteg. Ez felel az adatok tárolásáért, visszaolvasásáért. Itt foglalnak helyet azok a függvények is, amik műveleteket végeznek az adatokon. Része az adatbázis is.
2. View (magyarul nézet): A felhasználói felület megjelenítéséért, a felhasználó különféle nyűgjeinek a Vezérlő felé továbbításáért felelős réteg. Itt jelennek meg a felhasználó számára a vezérlőelemek, a felhasználónak szánt adatok megfelelő formában való megjelenítése is itt történik.
3. Controller (magyarul vezérlő): Ez a réteg a vezérlőelemek eseményeinek alapján meghívja a modell megfelelő függvényeit, illetve ha a megjelenítésben érintett adatok változnak, akkor erről értesíti a Nézetet.



Az alkalmazást három egységre bontjuk. Külön egység felelős a megjelenítésért, az adatok kezeléséért valamint a felhasználói cselekedetek megfelelő kezeléséért. Ez több okból is jó nekünk, legelőször is, ha lecseréljük valamelyik részt, akkor a többi még maradhat, nem kell hozzányúlni, több időnk marad (munkaidőben játszani:). Könnyebben módosíthatjuk az egyes részeket.

Az MVC egyik fő újítása az volt, hogy lehetővé tetette, hogy egy modellhez több nézet is tartozzon. Minden nézet ugyanannak a modellnek a belső állapotát jeleníti meg. Bármelyik nézeten lenyomnak egy gombot, az az esemény eljut a kontrollernek. A kontroller meghívja a modell megfelelő metódusát. Ha e miatt a modell belső állapota megváltozik, akkor a modell a megfigyelő tervezési mintának megfelelően értesíti a nézeteket, hogy változás történt, nekik is meg kell változni.



Az MVC mintának több továbbfejlesztése is létezik. Ezek közül a két legismertebb:

1. MVP – Model View Presenter, magyarul Modell – Nézet – Megjelenítő: Ebben a változatban a modell nem a nézetet, hanem a megjelenítőt értesíti, ha változás történik. A megjelenítő lekéri az adatokat a modellből, feldolgozza, és megformázza a nézet számára.
2. MVVM – Model View View-Model, magyarul Modell – Nézet – Nézetmodell: Ez az MVP továbbfejlesztése, ahol a nézetben a lehető legkevesebb logika van. A nézetmodell elvégez minden feladatot a nézet helyett, csak a megjelenítés marad a nézetre.

**-----------------------------------------------------------------------------------------**

**A KÖVETKEZŐT NEM KELL MONDANI, CSAK HA KÉRDEZIK**

#### ASP.NET MVC Framework

Az ASP.NET MVC Framework az ASP.NET Web Forms alapú fejlesztésnek nyújt alternatívát MVC alapú web alkalmazások fejlesztéséhez. ASP.NET MVC Framework egy olyan könnyű és jól tesztelhető megjelenítő keretrendszer, amely (csakúgy, mint az ASP.NET Web Forms) integrálja a már meglévő ASP.NET lehetőségeit, mint például a master page-eket és a beépített felhasználó kezelést, azaz membership provider alapú azonosítást. Az MVC alapjait a System.Web.Mvc névtér definiálja, amely a System.Web névtér támogatott része.

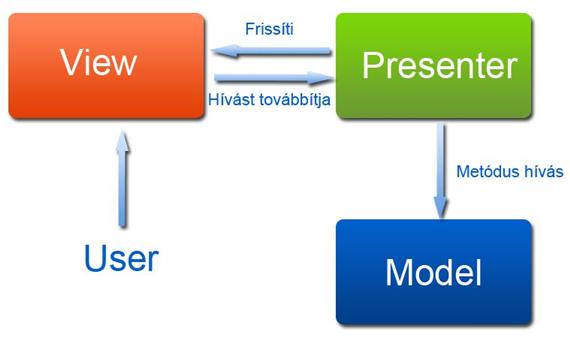
Az MVC egy alapvető programtervezési minta, amely számos fejlesztőnek már ismerős lehet. Néhány web alkalmazás már régóta használja az MVC keretrendszer előnyeit, míg mások továbbra is az ASP.NET hagyományos Web Forms-os postback alapú rendszert használják. Egyesek pedig ötvözik a két rendszer előnyeit. Azt később tárgyaljuk, hogy az MVC fejlesztési mód mikor előnyös.

**Az MVC framework három fő komponenst foglal magában:**

1. Modellek: A modell objektumok az alkalmazás azon részei, amelyek az adatokat "szállító" logikát implementálják. A modell objektumok gyakran fogadnak adatokat az adatbázisból és tárolják azokat magukban. Például egy Termék objektum lekérhet adatokat adatbázisból, dolgozhat vele, majd a módosított adatokat visszaírhatja a Termék táblába az SQL Szerveren.

Kisebb alkalmazások esetében a modellek inkább koncepcionálisak, mint fizikailag megvalósítottak. Például ha az alkalmazás kizárólag olvassa és megjeleníti az adatokat, akkor nincs konkrétan megvalósítva a modell réteg és a hozzá tartozó osztályszerkezet. Ebben az esetben a modell réteget csak az adattoló objektumok reprezentálják.

1. Nézetek (Views): A nézetek a felhasználói felület (User Inteface - UI) megjelenítő komponensei. A UI általában azokból az adatokból készül, amelyek a modell rétegből jönnek. Ilyen lehet például egy szerkesztő nézete a Termék táblának, amely állhat szövegdobozokból, gombokból, lenyíló menükből stb., melyek a Termék objektum aktuális állapotát mutatják.
2. Vezérlők (Controllers): A vezérlők azok a komponensek, melyek a felhasználói interakciót kezelik, dolgoznak a modell objektumokkal és kiválasztják a megfelelő nézetet a megjelenítéshez. Egy MVC alkalmazásban a nézet csak információt jelenít meg; a vezérlő kezeli és reagálja le a felhasználói interakciót. Például a vezérlő kezeli a query sztring értékeket, továbbítja a modell felé, melyek a megfelelő adatbázis lekérdezést állítják össze az átadott értékek alapján.



15. ábra

Az MVC minta olyan alkalmazások elkészítésében nyújt segítséget, melyek szétválasztják az alkalmazás különböző részeit (input logika, üzleti logika, megjelenítési logika), miközben egy laza csatolófelületet biztosít a szétválasztott részek között. A minta meghatározza azt is, hogy melyik logikai rétegnek hol kell elhelyezkednie az alkalmazásban. A megjelenítési vagy UI réteg a nézetekhez kötődik, az üzleti logika a vezérlőkhöz, az input logika pedig a modellekhez tartozik. Ez a szeparáció segít kezelni egy a komplexitást egy alkalmazás fejlesztésénél, mivel lehetővé teszi, hogy az implementáció során egy adott időben adott szemszögből vizsgáljuk a dolgokat. Például a megjelenítési réteg fejlesztésekor nem kell foglalkoznunk azzal, hogy az üzleti logikai rétegben milyen műveleteket kell végezni az adattal, hiszen a nézeteken keresztül csak megjelenítjük őket.

Ráadásul a komplexitás kezelésében az MVC minta könnyebbé teszi az alkalmazás tesztelését, mint egy Web Forms alapú fejlesztési modellben. Például Web Forms alapú web alkalmazásban egyetlen osztály felelhet a megjelenítésért és a felhasználói interakcióért is. Automata teszteket írni Web Forms alapú alkalmazásokhoz bonyolult lehet, mert egyedülálló oldal teszteléséhez példányosítani kell az oldal osztályát, az összes gyerekvezérlőt és további függő osztályokat is. Mivel az oldal futtatásához ennyi osztály példányosítására van szükség, nehéz olyan tesztet írni, amely az oldal egyes részeivel kizárólagosan foglalkozik. Kijelenthetjük tehát, hogy Web Forms alapú környezetbe sokkal nehezebb a tesztelést integrálni, mint egy MVC-t használó alkalmazásba. Továbbá Web Forms-os környezetben a teszteléshez szükségeltetik egy web szerver is. Mivel az MVC keretrendszer szétválasztja a komponenseket és ezek között interfészeket használ, könnyebb különálló komponensekhez teszteket gyártani az izoláció miatt.

A laza kötés az MVC alkalmazás három fő komponense között párhuzamos fejlesztést is lehetővé tesz. Ez azt jelenti, hogy egy fejlesztő dolgozhat a kinézeten, egy második a vezérlő logikán, egy harmadik pedig az üzleti logikára fókuszálhat egy időben.

##### Mikor készítsünk MVC alkalmazást

Körültekintően kell megválasztanunk, mikor használunk ASP.NET MVC keretrendszert a fejlesztéshez az ASP.NET Web Forms helyett, ugyanis az ASP.NET MVC nem helyettesíti a Web Forms modellt; használhatjuk mindkettőt egyszerre egy alkalmazáson belül is akár.

Mielőtt az MVC keretrendszer használata mellett döntünk a Web Forms modell helyett, mérlegeljük mindkettő előnyeit.

##### Az MVC alapú web alkalmazás előnyei

Az ASP.NET MVC keretrendszer a következő előnyöket nyújtja:

1. Könnyebbé teszi komplex alkalmazások fejlesztését azzal, hogy három részre osztja az alkalmazást: modellre, nézetre és vezérlőre.
2. Nem használ állapottárolást (view state) és szerveroldali form-okat sem. Ez ideálissá teszi az MVC keretrendszert azok számára, akik teljes hatalmat szeretnének az alkalmazás viselkedése felett.
3. Egy fő vezérlőn mintán keresztül dolgozza fel a web alkalmazáshoz érkező kéréseket, innen továbbítja a megfelelő vezérlőknek tovább. (A fő vezérlőről az MSDN weboldalán lehet több információhoz hozzájutni a Front Controller szekció alatt)
4. Segítséget nyújt a teszt-vezérelt fejlesztéshez (test-driven development - TDD)
5. Jól működik olyan web alkalmazások esetében, amelyet fejlesztők csapata fejleszt és támogat, ahol a kinézet tervezőknek magas fokú ellenőrzésre van szüksége az alkalmazás viselkedése felett.

##### **A Web Forms alapú alkalmazás előnyei**

A Web Forms alapú keretrendszer előnyei:

1. Támogatja az eseménykezelés modellt és megőrzi az állapotokat HTTP protokoll felett, mely előnyös az un. "line-of-business" web alkalmazás fejlesztésnél. A Web Forms alapú alkalmazás tucatnyi eseménykezelőt biztosít, amit több száz szerverkontrollból elérhetünk.
2. Page Controller mintát használ melyek különálló tulajdonságokkal ruházzák fel az egyes oldalakat. További információ a Page Controller-ről az MSDN weboldalán található.
3. Állapottárolást (view state) és szerveroldali form-okat használ, melyek megkönnyítik az állapotkezelési információk menedzselését.
4. Jól működik kisebb fejlesztői csoportban, számos komponens felhasználható segítve a gyors alkalmazásfejlesztést.
5. Összességében kevésbé összetett alkalmazás fejlesztés szempontjából, mert a komponensek (a Page osztály, vezérlők stb.) szorosan integráltak így általában kevesebb kódolást igényel, mint az MVC modell.

##### **Az ASP.NET Framework tulajdonságai**

Az ASP.NET MVC Framework a következő funkciókat nyújtja:

1. Az alkalmazás feladatainak szeparálása (input logika, üzleti logika, megjelenítési logika), tesztelhetőség és teszt-vezérelt fejlesztés támogatása alapból. Az MVC összes mag eleme interfész alapú, mely lehetővé teszi az úgynevezett "mock" objektumokkal való tesztelést, amelyek olyan objektumok, amik imitálják az aktuális objektum viselkedését az alkalmazásban. Lehetővé teszi a unit-test alapú tesztelést anélkül, hogy a vezérlőket egy ASP.NET folyamaton keresztül futtatnunk kellene, így flexibilissé és gyorssá téve a unit-tesztelést. Bármelyik unit-test keretrendszert használhatjuk, amelyik kompatibilis a .NET keretrendszerrel.
2. Egy kiterjeszthető és bővíthető keretrendszer. Az ASP.NET MVC keretrendszer komponensei úgy lettek lefejlesztve, hogy azok könnyen testre szabhatóak, ill. lecserélhetőek legyenek.
3. Az URL-mapping (útvonal feltérképezés) komponens lehetővé teszi olyan alkalmazások fejlesztését, amelyek érthető és kereshető URL-ekkel rendelkeznek. Az URL-ek nem tartalmaznak fájlnév kiterjesztéseket, kereső (SEO) és felhasználóbarát.
4. Támogatja a meglévő ASP.NET oldalak (.aspx fájlok), felhasználói vezérlők (.ascx fájlok) és master page-ek (.master fájlok) használatát. Használhatjuk a meglévő ASP.NET lehetőségeit, mint a beágyazható (azaz nested) master page-ek használatát, valamint az ASP.NET jelölőnyelvén belüli szerveroldali kód (pl. C#) használatát a <%= %> kifejezés segítségével.
5. Meglévő ASP.NET funkciók támogatása. Az ASP.NET MVC lehetőséget ad a beépített lehetőségek használatára, mint a form autentikáció, Windows autentikáció, felhasználó kezelés (membership és roles), session kezelés, stb.

**---------------------------------------------------------------------------------------------------------**

**Tervezési minták – PDF a mappában**

**Szabad és nem szabad szoftverek – PDF a mappába**