## Az agilis fejlesztés

Az agilis fejlesztés a szoftverfejlesztési módszertanok egy olyan csoportja, ahol a követelmények és a program folyamatosan és együttesen változnak. A fejlesztőcsapatok önszerveződőek és képesek egymás felváltására. Abból indul ki, hogy a fejlesztés korai szakaszában nem lehet pontos becslést, megbízható terveket készíteni. Ezért minden részletes, nagyszabású terv üzletileg veszteséget termel, mert folyamatosan módosításokra szorul, hiszen a megalapozottságuk nem megfelelő az elkészítésük pillanatában. Mivel nem lehet a fejlesztés korai szakaszában pontosan tervezni, ezért dinamikusan kell a tervezést végrehajtani. Ennek megfelelően az agilis fejlesztési módszertanok adaptívak, iteratívak. Nem baj, ha hiba történik a fejlesztés közben, mert hibázni jó, elősegíti pontos és megfelelő módszer kialakítását. Folyamatosan kísérletezni, tapasztalatokat gyűjteni kell, a hibákból pedig le kell vonni a tanulságokat, és fejleszteni a módszert, hogy elkerülhetők legyenek a későbbiekben a hibák.

Az agilis fejlesztés alapját a 2001-ben megjelent **Agilis Szoftverfejlesztési Kiáltvány** jelenti.

Manifesto for Agile Software Development

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

Individuals and interactions over processes and tools
Working software over comprehensive documentation
Customer collaboration over contract negotiation
Responding to change over following a plan

That is, while there is value in the items on
the right, we value the items on the left more.

Kent Beck
Mike Beedle
Arie van Bennekum
Alistair Cockburn
Ward Cunningham
Martin Fowler
James Grenning
Jim Highsmith
Andrew Hunt
Ron Jeffries
Jon Kern
Brian Marick
Robert C. Martin
Steve Mellor
Ken Schwaber
Jeff Sutherland
Dave Thomas

Azaz körülbelüli fordításban:

Úgy tárjuk a szoftverfejlesztés jobb módjait, hogy csináljuk és segítünk másoknak is ezt tenni. Ezen munkán keresztül a következő értékekhez jutunk el:

Az **egyének és kölcsönhatások** fontosabbak, mint a folyamatok és eszközök

A **működő szoftver** fontosabb, mint az átfogó dokumentáció.

Az **ügyféllel való folyamatos együttműködés** fontosabb, mint a szerződéses megállapodás helyett

A **változásokra adandó válasz** fontosabb, mint egy terv követése.

Habár a jobb oldali dolgokban is vannak fontos dolgok, mi mégis fontosabbnak tartjuk a baloldali elemeket.

Ennek megfelelően az agilis fejlesztés 12 alapelve:

- A legfontosabb az elégedett ügyfél a használható szoftver gyors és folyamatos szállításával
- A változáskérelmek (change requirements) a fejlesztés késői szakaszában is elfogadottak.
- Működő szoftververziók gyakori szállítása, és törekedni kell a minél gyakoribb szállításokra.
- Szoros együttműködés az üzleti folyamatban résztvevők és a fejlesztők között.
- A projektet motivált egyénekre kell építeni. Adj meg nekik mindent, amire szükségük van, és bízz bennük, hogy jól végzik a dolgukat!
- A legjobb és leghatékonyabb módja az információcserének a fejlesztőcsapattal és a fejlesztőcsapatban a szemtől-szembeni kommunikáció.
- A működő szoftver a haladás legfontosabb mérőeszköze.
- Az agilis módszerek elősegítik a fenntartható fejlesztést. A szponzoroknak, fejlesztőknek és felhasználóknak képesnek kell lenniük egy állandó sebesség fenntartásában.
- A folyamatos figyelem a technikai kiválóságra és a jó tervezésre fokozza az agilitást.
- Az egyszerűség, az el nem végzett munka maximalizálásának művészete alapvetően fontos.
- A legjobb architektúrák, követelmények és tervek az önszerveződő csapatoknak köszönhetőek.

- A fejlesztői csapat rendszeresen időközönként megfontolja, hogyan válhat hatékonyabbá, majd ennek megfelelően finomítja működését.

## Extrém programozás (XP)

Az extrém programozás, mint az agilis fejlesztés egy módszertanának alapelve, hogy folyamatosan változó követelmények mellett kell a lehető leghatékonyabban a lehető legjobb alkalmazást elkészíteni. A hagyományos módszertanok abból indulnak ki, hogy a követelmények már a projekt elején ismertek, azon nem nagyon változnak. Bármilyen változás a követelményekben azt jelenti, hogy növekedni fognak a költségek, és minél később történik meg egy igénymódosítás a kivitelezés során, annál nagyobb lesz ennek a költsége. Ezért az XP arra törekszik, hogy ezeket a költségeket minimalizálja, ezért új módszertant vezet be.

Az extrém programozás alapvető eleme a **kommunikáció**. A legfontosabb mindig az, hogy a lehető leghamarabb jusson a fejlesztők tudomására minden olyan információ, amire szükségük lesz a munkájuk folyamán. Fontos, hogy minden fejlesztő rendelkezzen minden információval, és ugyanúgy tekintsen a rendszerre, ahogy a leendő felhasználó tenné. Ezért nem szabad hosszadalmas munkával bonyolult terveket készíteni, sokkal inkább épít a megrendelő és a fejlesztők együttműködésére, a közöttük zajló folyamatos egyeztetésre.

Az XP másik alapeleme az **egyszerűség**. Ez alapján a fejlesztést a lehető legegyszerűbb módon kezdjük el, és később bővítjük az igények szerint. Tehát nem a későbbi igényeknek próbálunk megfelelni ma, hanem a jelenlegi igényeket implementáljuk, és ha van is róla tudomásunk, hogy a funkciót a későbbiekben majd bővíteni szeretnék, azzal most nem foglalkozunk. Ez a felfogás később megdrágíthatja az esetleges továbbfejlesztést, viszont elkerülhető vele a túlbonyolított kód, amikor olyan kódrészletek vannak a kódban, ami nincs használva, és csak azért kerül bele, mert később majd szükség lehet rá. A bonyolult kód karbantartása, módosítása szintén megdrágítja a fejlesztést, így nem dönthető el könnyen, hogy melyik irány okozta esetleges többletköltség drágítaná meg jobban a fejlesztést.

Ax XP erőteljesen épít a **visszajelzések**re is. Az ügyféltől a folyamatos kommunikáció miatt a visszajelzések is folyamatosak, kevesebb időbe kerül az új igények megfogalmazása. A megrendelő folyamatosan kapja az alkalmazás aktuális verzióit, így mindig tisztában lehet annak tényleges állapotával, ezáltal könnyebben tud beavatkozni a fejlesztés esetleges irányába. Az XP folyamán a fejlesztők is folyamatosan adják a visszajelzéseket az ügyfelek igényeire, ezáltal könnyebben lehet arra idő és költségbecslést adni, és akár módosítani a fejlesztés menetén is ezek miatt.  De maguk a fejlesztők is folyamatosan kapják a visszajelzéseket az

ügyféltől valamint a belső tesztektől, így jobban tisztában tudnak lenni az alkalmazás aktuális állapotával, valamint a várható igényekkel is.

A negyedik alapelv a **bátorság**. Elsőre nem tűnik fontosnak, pedig igenis annak kell lenni, hogy a fejlesztők csakis arra törekedjenek, hogy az éppen aktuális igényeknek megfelelő alkalmazást készítsék el, és tudomásul vegyék, hogy egy-egy funkciót adott esetben sokszor is át kell írni a módosuló igényeknek megfelelően. Ez jelenti azt is, hogy adott esetben nagyobb kódrészleteket kell átírni, vagy éppen kidobni, amit sokan nem szeretnek, „hiszen egyszer még jó lehet valamire”. Azonban az egyszerűségre törekvés miatt a kódot a lehető legegyszerűbbre kell készíteni, és ez néha maga után vonja a meglévő kódrészek kidobását is.

Az ötödik, utolsó alapelv a **tisztelet**. Ha valaki nem tiszteli a csapattagokat, akkor nem tud velük rendesen együttdolgozni. Ha a megrendelő nem tiszteli a fejlesztőket, az elkészült programon ez látszani fog. Ha a fejlesztők nem tisztelik a megrendelőt, akkor nem az igényeinek megfelelő alkalmazást készítik el, ezáltal erősítik meg saját rossz hírüket, ami a későbbiekben esetleges munkák elvesztését jelenthetik.

Az extrém programozás extrém módon használja az iteratív fejlesztés módszertant is. Naponta akár több iteráció is lehetséges. Fontos, hogy minden elkészült kódot egységtesztelésnek vet alá, és minden újabb iteráció esetén is teljesülnie kell az egységtesztelésnek.

Az extrém programozás egy másik jellemzője, hogy erőteljesen támogatja a páros programozást. Ilyenkor egy munkaállomás mellett két fejlesztő – vagy egy fejlesztő és egy szervező – dolgozik. A párok összerendelése nem végleges, folyamatosan változik. Ennek a módszernek az egyik legnagyobb előnye, hogy elvileg jobb minőségű kódot készülhet, hiszen másik ember is átnézi, fele annyi hardver kell, a cserélődő párok miatt a fejlesztők jobban megismerik egymást, nincs monotónia, és könnyebb a meglévő tapasztalatot átadni a párnak.

## Scrum

A Scrum egy inkrementális, iteratív szoftverfejlesztési módszertan, az agilis fejlesztés legelterjedtebb fajtája.

A Scrum szó a rögbiből származik, jelentése viaskodik, összecsap. Hivatalosan 1996-ban publikálták először, de 1986-ban már Takeucsi Hirokata és Nonaka Ikudzsiro leírta a módszertan alapjait. Nézetük szerint a hagyományos vízesés modell, ahol az egyes fázisok egymást követik, olyan, mint egy váltófutás, ahol egyszerre egy ember fut, és a futók egymásnak adják át a stafétát. Ezzel szemben az ő módszerükben a fázisok erős átfedésben vannak egymással. Itt a résztvevők folyamatosan együttműködnek, úgy, mint a rögbiben, ahol a csapat egyszerre fut, és a labdát egymás között passzolgatják folyamatosan.

A Scrum módszertan pontos leírását végül 2001-ben Ken Schwaber és Mike Beedle írta le az Agile Software Development with Scrum című munkájukban. Olyan iteratív és inkrementális módszertan, ami az extrém programozással ellentétben nem definiálja pontosan a módszert, sokkal inkább egy keretrendszert ad meg, aminek köszönhetően akár a szoftverkészítéstől eltérő területeken is használható. A működő szoftvert tekinti a sikeres teljesítés mértékének, nem a részletes dokumentációt.

A Scrumban a feladatokat mindig sprintekre bontják. Ez jellemzően 2-4 hetes intervallumot jelent. Mivel a Scrum a kommunikációra helyezi a fő hangsúlyt, ezért nem meglepően kiemelten fontos szerep jut a megbeszéléseknek is. Minden spint előtt van egy megbeszélés (sprint planning, sprint tervezés), amelyen meghatározzák a következő sprintben elvégzendő feladatokat, megbecslik a várható időigényt, és felmérik a szükséges tevékenységeket, amelyek az adott sprinten belül szükségesek a feladatok teljesítéséhez.

Az adott sprinten belül naponta tartanak megbeszélést (daily scrum), ahol a scrum master vezetésével a csapat jellemzően három kérdésre ad választ:

- Mit csináltál tegnap?
- Mit tervezel mára?
- Vannak-e akadályok, problémák, amelyek akadályozzák a feladat teljesítését.

Ezeket a megbeszéléseket szokták általában napi scrum néven emlegetni.

A sprint végén tartanak egy sprint értékelő (**retrospective**) megbeszélést. Itt a csapattagok véleményt mondhatnak az éppen lezárult sprintről, javaslatokat tehetnek a későbbi sprintek teljesítését illetően a folyamat továbbfejlesztésére. Lényegében azt értékelik, hogy mi ment jól, és mi az, amin változtatni kell a sikeresebb működés miatt.

A feladatokat a **Product backlog**ban tartják nyilván. Ide kell összegyűjteni minden olyan igényt, követelményt, amely fontos a program sikeres megvalósításához. Tehát nem csak a tényleges vevői igényeket tartalmazza, hanem a felfedezett hibákat, és minden nem-funkcionális követelményt is. Gyakorlatilag a product backlog írja le a fejlesztés célját. A feladatokat súlyozzák hasznosságuk szerint, és tartalmaznak egy becslést is a várható költségről (fejlesztési időről) is. A product owner feladata a feladatok súlyozása, a várható időigényt a fejlesztőcsapat határozza meg. Az itt leírt feladatok mindig a megrendelői oldal szempontjából vannak leírva.

A sprintben elvégzendő feladatokat a **Sprint Backlog** tartalmazza. Ilyenkor részletesen leírják az adott feladatokat, hogy mindenki pontosan tudja értelmezni. Szokás 4-8-16 órás részfeladatokra bontani, hogy jobban tervezhető legyen a sprint. Nem rendelik személyekhez, hanem az előrehaladás ütemében az egyes csapattagok választják ki, hogy ki melyik feladatot végzi el. Itt a feladatokat már a fejlesztők szemszögéből írják le.

Szokás még **Burndown chart**ot vezetni, amit naponta frissítenek, és az adott sprinten belüli előrehaladást lehet rajta látni.

A Scrum szoftverfejlesztési módszertanban három szerepkört határoznak meg, ezek a scrum master, a product owner, és a team, azaz maga a fejlesztő csapat. Ez utóbbi

létszáma jellemzően 5-9 fő.

A **scrum master (SM)** feladata egy vagy több Scrum csapat irányítása. Nem ő a vezető, sokkal inkább a mozgatórugó. Nem ő felelős a sikeres megvalósításért, hiszen a Scrum metodika szerint ezért az egész csapat együtt felelős.

A SM biztosítja a teamben a Scrum szabályok betartását. A napi Scrum meetingeken a SM teszi fel a szokásos kérdéseket: mit csináltál tegnap, mit fogsz csinálni ma, van-e valami, ami akadályoz ebben.

Vitás kérdésekben a SM feladata, hogy megegyezés szülessen. Ha valami akadályozza a fejlesztést, akkor neki kell ezt elhárítania. Segíti a csapatagokat, hogy a feladatukra tudjanak koncentrálni. Összefoglalva a scrum master az, aki biztosítja a team működését, megteremti a konszenzust a csapaton belül, elhárítja a külső és a belső akadályokat, ezért egyfajta ütközőzóna a megrendelő, a fejlesztő cég vezetői és pénzügyesei, valamint a BA és a fejlesztők között.

A **product owner** (PO) az a személy a Scrum metodikában, aki a megrendelői oldalt képviseli, ugyanakkor együttműködik a felhasználókkal, hogy milyen funkciók legyenek benne a következő verzióban. Ténylegesen nem vesz részt a fejlesztésben, viszont szorosan együttműködik mind a megrendelővel, mind a fejlesztőcsapattal. Emiatt jól kell ismernie a megrendelő üzleti folyamatait, és az egyes felhasználói csoportok igényeit is.

A PO határozza meg a megrendelővel egyeztetve, hogy a következő scrum sprintben milyen feladatok legyenek, és ő priorizálja a megvalósítandó feladatokat. Ha kérdés merül fel a fejlesztésben az üzleti folyamatokkal kapcsolatban, akkor a PO feladata ezeket tisztázni. A sprintek végén a PO feladata elfogadni vagy elutasítani a munka eredményét.

A Scrum arra törekszik, hogy ne legyen alá- és fölérendeltségi viszony a projekten belül, hanem a megrendelő képviselői és a fejlesztőcsapat is egymással szorosan együttműködve, azonos szinten, barátként dolgozzon együtt.

## Test Driven Development (TDD - Tesztvezérelt fejlesztés)

A TDD módszertan a 21. század elején gyorsan népszerűvé vált, az agilis fejlesztés egyik népszerű módszertana. Elve, hogy minden egyes új funkció megírása ELŐTT kell megírni a funkció automatikus tesztelését. Minden újabb funkció hozzáadása előtt újabb tesztet kell készíteni az új funkcióhoz, tehát egy iteratív, inkrementális módszertanról van szó.

A TDD keretén belül törekedni kell arra, hogy minél kevesebb új kódsort adjunk hozzá a meglévő kódbázishoz egy ciklusban, azaz a TDD a kis lépésekben történő fejlesztést preferálja. Hiszen ha sokezer kódsort adunk hozzá a meglévő kódhoz, biztosan számíthatunk rá, hogy az addigi automatikus tesztek közül jópár hibát fog

dobni. Egy ekkor módosítás esetén pedig sokkal több időt vesz igénybe a korábbi tesztekben jelentkező hibák megkeresése és javítása, mint ha csak pár tíz új sort adunk a kódhoz. A TDD szerint akár 10 soronként is érdemes új ciklust kezdeni.

Egy új funkció hozzáadása mindig úgy kezdődik, hogy létrehozunk egy új automatikus tesztet a meglévők mellé a még nem létező funkció tesztelésére. A tesztet fel kell készíteni minden olyan tesztesetre, amelyik az új funkció helyes működésének teszteléséhez szükséges. Ezzel pedig a TDD elősegíti az elvárt funkció működésének minél pontosabb megismerését, hiszen a fejlesztőnek a funkció tényleges megvalósítása ELŐTT már pontosan tisztában kell lennie a követelményekkel, ha valami nem világos, akkor már előtte tisztáznia kell. Emiatt a TDD tesztjei szolgálnak a funkciók kvázi specifikációiként is.

Ezután az új tesztet a régiekkel futtatni kell, és nézni, hogy hibát dob-e. Ha nem, akkor valószínű, hogy rossz tesztet írtunk, amelyik nem úgy működik, ahogy kellene. Sikertelen tesztfuttatás esetén el kell készíteni a kódot, ami alapján a teszt már hibátlanul lefut. Nem a szép kód ilyenkor az elvárás, hanem az, hogy a tesztet elégítse ki.

Ha a tesztek végre sikeresen lefutottak, akkor lehet nekilátni a kód refaktorálásának. Mivel kevés sor módosult, várhatóan a refaktoring is kevés kódot fog érinteni, ráadásul biztosan működő kódot kell szebbé és jobbá tenni.

Ezt a fajta módszertant szokták Red, Green, Refactor módszerként is hívni.

Fontos megemlíteni pár dolgot:

- A TDD fejlesztési módszertan nem pótolja a manuális tesztelést. Az automatikus tesztesetek nem fogják 100%-osan lefedni az összes lehetséges hibát, és nem is fedik fel a meglévők mindegyikét sem, sőt, gyakorlatban ez kb. a hibák egyötöde, amit felfednek. Tehát manuális tesztelés igenis szükséges. A Unit testek sokkal inkább a kód refaktorálását segítik elő, hiszen ha a módosított kódon lefutnak az automatikus tesztek, akkor a refaktorálás sikeres volt. Ennek ellenőrzése pedig pillanatok alatt megtörténhet, így csökkenteni lehet a refaktorálásra fordított időt.
- A TDD, mint agilis módszertan, nem hanyagolja el a tervezést, és a jó minőségű kód írását, hiszen egy új funkció hozzáadásakor a fejlesztőnek már pontosan tisztában kell lennie a pontos követelményekkel – lásd a teszt a funkció specifikációja. Csak annyi történik, hogy az adott iteráció megkezdése után még a teszt befejezéséig lehetséges a követelmények pontosítása, tehát az elkészült funkciónak ki kell elégítenie az elvárásokat. Ráadásul, mivel a teszt lefedi a követelményeket, a refaktorálás alatt pedig a kód minőségét javítja a fejlesztő, folyamatosan jó minőségű kódot biztosítva.