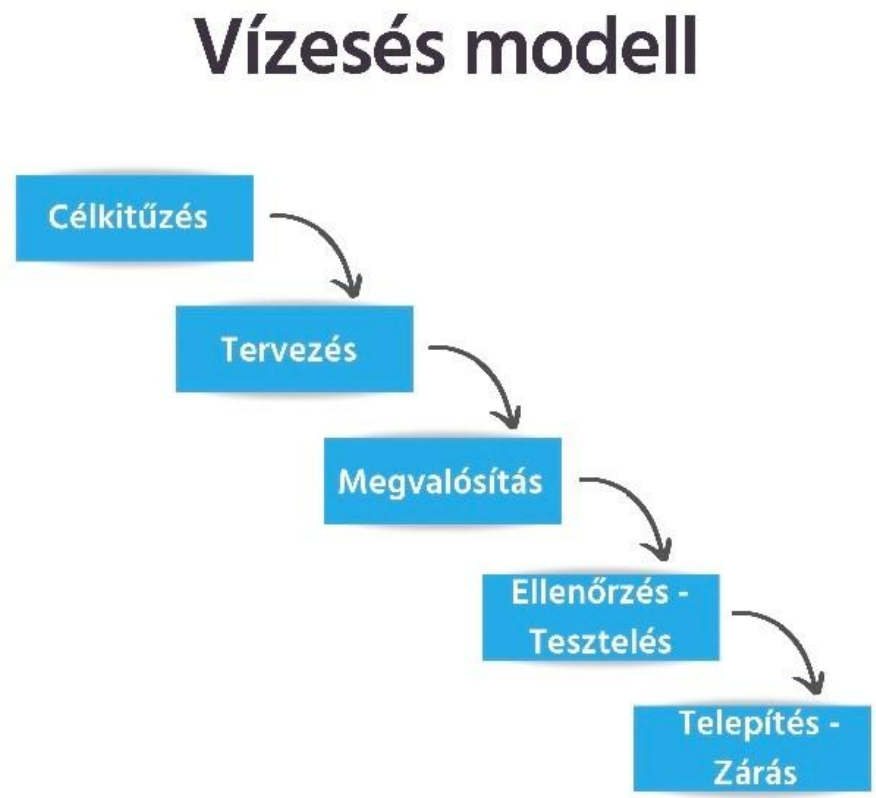
Programfejlesztési Módszertanok

Vízesésmodell

A szoftverfejlesztés folyamatának első publikált modellje, amely más tervezői modellekből származik. Az elnevezése onnan fakad, hogy a modellben az egyes fázisok lépcsősen kapcsolódnak egymáshoz, ami alapján vízesésmodellként vált ismertté.



1. Követelmények azonosítása, célkitűzés

Az első részben egy adott határidőn belül megvalósítandó projekt tömören megfogalmazott célkitűzéséről esik szó. Alapvetően a célkitűzés megoldást jelenthet egy meglévő problémára vagy új lehetőséget kínálhat az üzletmenetben. Ezen fázis részeként egy felmérési dokumentumban vagy

esettanulmányban elemzésre kerül: a projekt megvalósíthatósága, elvárt nyeresége, szükséges ráfordítások, lehetséges kockázatok. A megvalósíthatósági tanulmány elkészültét követő, megfelelő szintű jóváhagyás után következik a tényleges tervezés.

2. Tervezés

Ebben a szakaszban részletes projektterv készül, megfogalmazott célkitűzések, elemi munkacsomagok kerülnek lebontásra (WBS). A fentről lefelé történő tervezési eljárás a komplex és bonyolult projektekben is segíti a konkrétan elvégzendő tevékenységek, részfeladatok meghatározását. Ez a terv a feladatokat, szükséges felhasználni kívánt erőforrásokat, ütemterveket, költségeket tartalmazza, ami a megállapított értékek a projekt kontroll alapjául szolgálnak.

3. Megvalósítás

A megvalósítási szakasz sikerességét nagyban befolyásolja az alapos és körültekintő tervezés. Minden tényező figyelembe lett-e véve? Megfelelő mérési rendszer lett-e kialakítva?

Ebben az életciklusban a szükségszerű változáskezelés, kockázat kezelés és intenzív kommunikáció szükséges a projekt team, a projektmenedzsment és a vezetőség - ügyfél között. Ekkor kerül sor a tényleges fejlesztésre, a tesztelésre és telepítésre.

4. Ellenőrzés - tesztelés

A leszállítandó termék létrehozását követően kerül sor annak minőségi ellenőrzésére, tesztelésére. A tesztelés – hasonlóan a megvalósításhoz – előre rögzített követelmények és tervek alapján történik, melyek végrehajtási módja és sorrendje kötött. A tesztelés során feltárt hibák a projekt lezárását megelőzően javítandók.

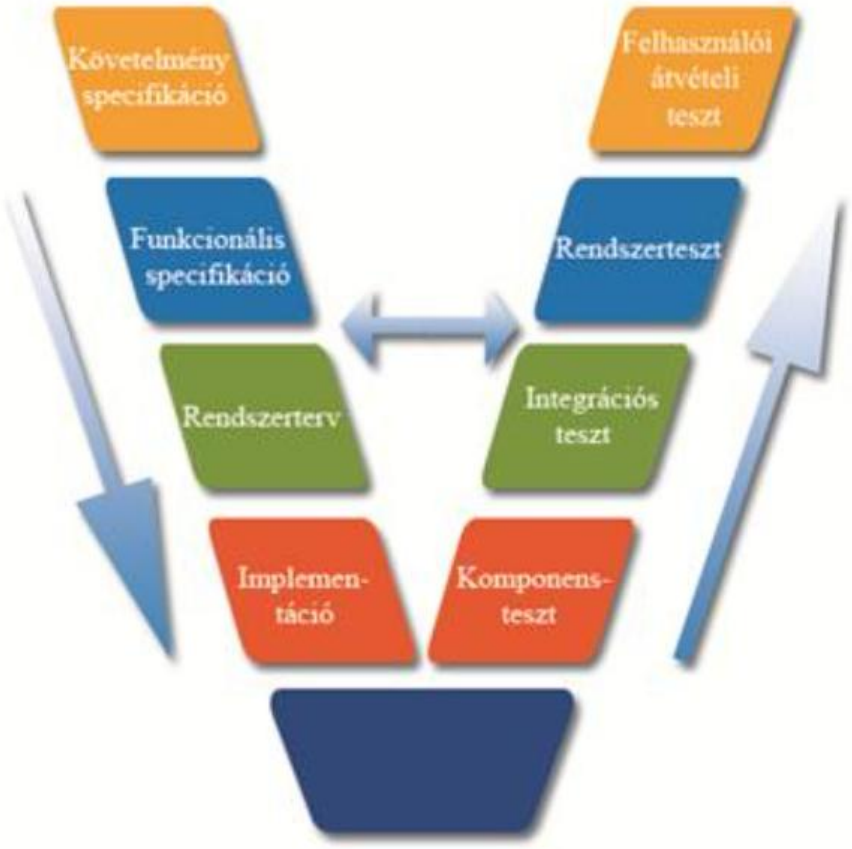
5. Telepítés - Zárás

A projekt zárását az összes teljesítés igazolás elfogadása után lehet megkezdeni. A zárás része a projektcsapat tagjainak új projektre allokálása, a fizetési zárások megkezdése és a projekt tanulságainak levonása, valamint az elkészített termék üzemeltetésre történő átadása.

Hátrányok: A vízesésmodell legfőbb problémáját a projekt szakaszainak különálló részekké történő nem flexibilis partícionálása okozza. Egy komplex, bonyolult probléma megoldása nem végezhető el hatékonyan ezzel a megközelítéssel. A vízesésmodell csak akkor használható jól, ha már előre jól ismerjük a követelményeket, melyeket részletes és pontos specifikáció követ

V-modell

A V-modell (angolul: V-Model vagy Vee Model) a nevét onnan kapta, hogy két szára van és így egy V betűhöz hasonlít. Az egyik szára megegyezik a vízesés modellel. Ez a fejlesztési szár. A másik szára a létrejövő termékek tesztjeit tartalmazza. Ez a tesztelési szár. Az egy szinten lévő fejlesztési és tesztelési lépések összetartoznak, azaz a tesztelési lépés a fejlesztési lépés során létrejött dokumentumokat használja, vagy a létrejött terméket teszteli. Ennek megfelelően az előírt fejlesztési és tesztelési lépések a következők:



Evolúciós fejlesztés

Az evolúciós fejlesztés alapötlete az, hogy a fejlesztőcsapat kifejleszt egy kezdeti implementációt, majd azt a felhasználókkal véleményezteti, majd sok-sok verzión keresztül addig finomítja, amíg a megfelelő rendszert el nem érjük. A szétválasztott specifikációs, fejlesztési és validációs tevékenységekhez képest ez a megközelítési mód sokkal jobban érvényesíti a tevékenységek közötti párhuzamosságot és a gyors visszacsatolásokat.

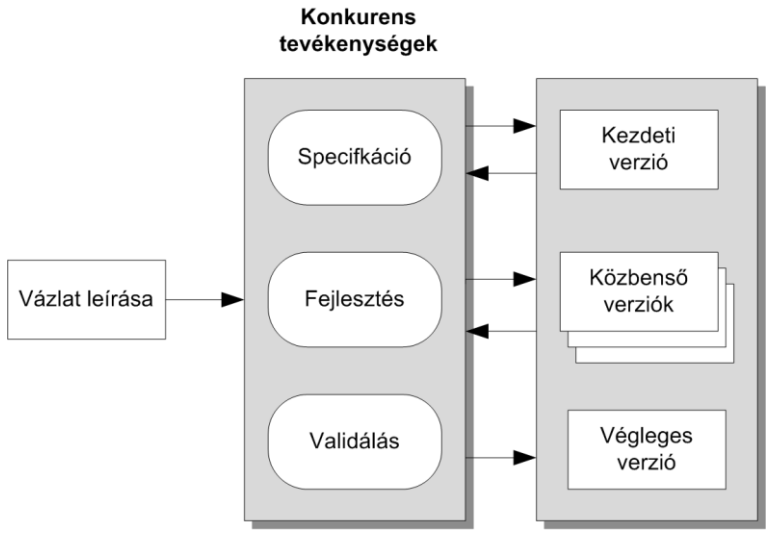
Az evolúciós fejlesztésnek két különböző típusa ismert:

1. Feltáró fejlesztés: célja egy működőképes rendszer átadása a végfelhasználóknak. Ezért elsősorban a legjobban megértett és előtérbe helyezett követelményekkel kezdik a fejlesztés menetét. Ennek érdekében a megrendelővel együtt tárjuk fel a követelményeket, és alakítják ki a végleges rendszert, amely úgy alakul ki, hogy egyre több, az ügyfél által kért tulajdonságot társítunk a már meglévőkhöz. A kevésbé fontos és körvonalazatlanabb követelmények akkor kerülnek megvalósításra, amikor a felhasználók kérik.
2. Eldobható prototípus készítés: A fejlesztés célja ekkor az, hogy a lehető legjobban megértsük az ügyfél követelményeit, amelyekre alapozva pontosan definiáljuk azokat. A prototípusnak pedig azon részekre kell koncentrálni, amelyek kevésbé érthetők.

Próbáljuk meg összevetni az evolúciós megközelítést a vízesésmodellel. A fentiek alapján láttuk, hogy a vízesésmodell kevésbé rugalmas a menetközben szükséges változásokra, így érvelhetünk azzal, hogy az evolúciós megközelítés hatékonyabb a vízesésmodellnél, ha olyan rendszert kell fejleszteni, amely közvetlenül megfelel az ügyfél kívánságainak. További előnye, hogy a rendszerspecifikáció inkrementálisan fejleszthető. Mindezek ellenére a vezetőség szemszögéből két probléma merülhet fel:

1. A folyamat nem látható: a menedzsereknek rendszeresen szükségük van leszállítható részeredményekre, hogy mérhessék a fejlődést.
2. A rendszerek gyakran szegényesen strukturáltak: a folyamatos változtatások lerontják a rendszer struktúráját, így kevésbé érthetővé válik. A szoftver változásainak összevonása pedig egyre nehezebbé és költségesebbé válhat.

Felmerülhet akkor a kérdés, hogy mikor és kinek érdemes használni az evolúciós fejlesztési modellt? Nos a válasz természetesen nem lehet egyértelmű, de a gyakorlati tapasztalatok alapján a várhatóan rövid élettartalmú kis vagy közepes rendszerek esetén célszerű az alkalmazása. Körülbelül 500.000 programsorig terjedően. Ugyanis nagy, hosszú élettartalmú rendszerek esetén az evolúciós fejlesztés válságossá válhat pontosan az evolúciós jellege miatt.



Prototípus modell

A prototípus modell válasz a vízesés modell sikertelenségére. A fejlesztő cégek rájöttek, hogy tarthatatlan a vízesés modell megközelítése, hogy a rendszerrel a felhasználó csak a projekt végén találkozik. Gyakran csak ekkor derült ki, hogy az életciklus elején félreértették egymást a felek és nem a valós követelményeknek megfelelő rendszer született. Ezt elkerülendő a prototípus modell azt mondja, hogy a végső átadás előtt több prototípust is szállítsunk le, hogy mihamarabb kiderüljenek a félreértések, illetve a megrendelő lássa, mit várhat a rendszertől.

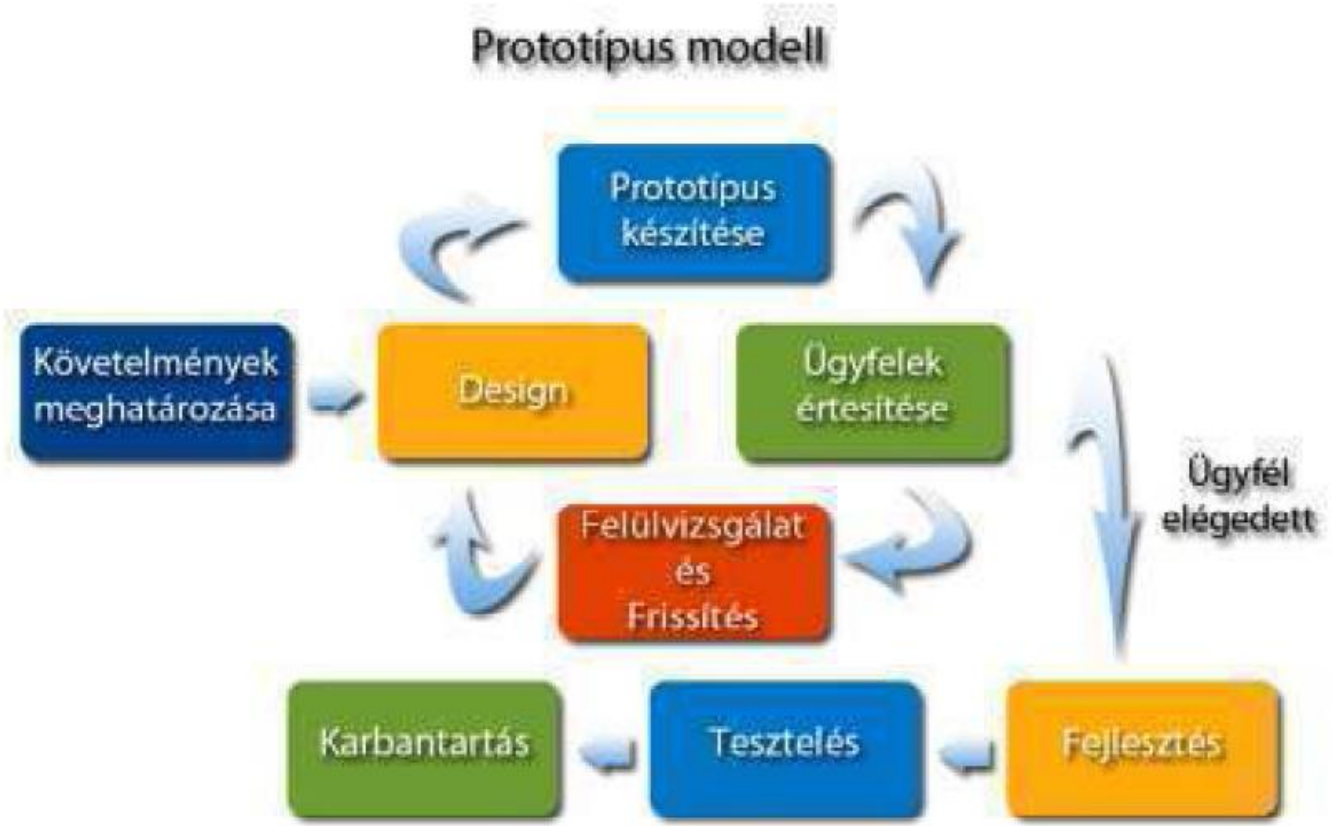
A prototípus alapú megközelítése a fejlesztésnek azon alapszik, hogy a megrendelő üzleti folyamatai, követelményei nem ismerhetők meg teljesen. Már csak azért sem, mert ezek az idővel változnak. A követelményeket érdemes finomítani prototípusok segítségével. Ha a felhasználó használatba vesz egy prototípust, akkor képes megfogalmazni, hogy az miért nem felel meg az elvárásainak és hogyan kellene megváltoztatni. Ebben a megközelítésben a leszállított rendszer is egy prototípus.

Ez a megközelítés annyira sikeres volt, hogy a modern módszertanok majd mindegyike prototípus alapú. Az iteratív módszerek általában minden mérföldkőhöz kötnek egy prototípust. Az agilis módszertanok akár minden nap új prototípust állítanak elő.

A kezdeti prototípus fejlesztése általában a következő lépésekből áll:

1. lépés: Az alap követelmények meghatározása: Olyan alap követelmények meghatározása, mint a bemeneti és kimeneti adatok. Általában a teljesítményre vagy a biztonságra vonatkozó követelményekkel nem foglalkozunk.
2. lépés: Kezdeti prototípus kifejlesztése: Csak a felhasználói felületeket fejlesztjük le egy erre alkalmas CASE eszközzel. A mögötte lévő funkciókat nem, kivéve az új ablakok nyitását.
3. lépés: Bemutatás: Ez egyfajta felhasználói átvételi teszt. A végfelhasználók megvizsgálják a prototípust, és jelzik, hogy mit gondolnak másként, illetve mit tennének még hozzá.
4. lépés. A követelmények pontosítása: A visszajelzéseket felhasználva pontosítjuk a követelmény specifikációt. Ha még mindig nem elég pontos a specifikáció, akkor a prototípust továbbfejlesztjük és ugrunk a 3. lépésre. Ha elég pontos képet kaptunk arról, hogy mit is akar a megrendelő, akkor az egyes módszertanok mást és mást írnak elő.

A prototípus készítést akkor a legcélszerűbb használni, ha a rendszer és a felhasználó között sok lesz a párbeszéd. A modell on-line rendszerek elemzésében és tervezésében nagyon hatékony, különösen a tranzakció feldolgozásnál. Olyan rendszereknél, ahol kevés interakció zajlik a rendszer és a felhasználó között, ott kevésbé éri meg a prototípus modell használata, ilyenek például a számítás igényes feladatok. Különösen jól használható a felhasználói felület kialakításánál.



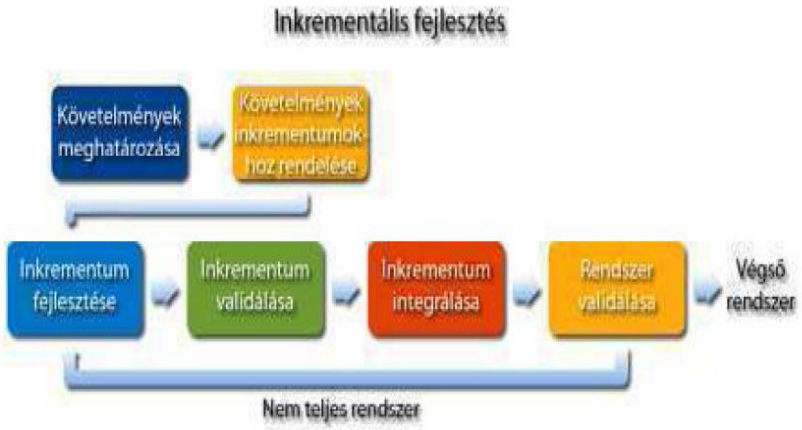
A prototípus modell nagyban épít a tesztelésre. Minden prototípust felhasználói átvételi tesztnek vetnek alá, ami során könnyen kiderül, hogy milyen funkcionális és nemfunkcionális követelményt nem tart be a prototípus. A korai szakaszban sok unit-tesztet alkalmazunk. Amikor befejezünk egy újabb prototípust, akkor regressziós teszttel vizsgáljuk meg, hogy ami az előző prototípusban működött, az továbbiakban is működik-e. Ha az új prototípusban van új komponens is, akkor a régi és az új komponensek között, illetve az új – új komponensek között integrációs tesztet kell végrehajtani. A modell későbbi szakaszában, miután már a követelmény és a funkcionális specifikáció letisztult, egy vízesés modellre hasonlít. Azaz az implementáció után jön a tesztelés. Ekkor elvégezzük újból komponens és integrációs teszteket is. Rendszertesztet általában csak a végső prototípus átadás előtt végzünk.

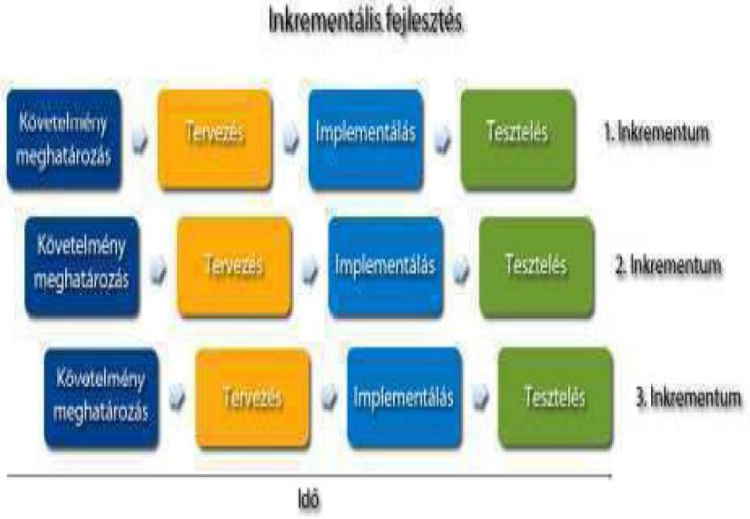
Iteratív és inkrementális módszertanok

Az iteratív módszertan előírja, hogy a fejlesztést, kezdve az igényfelméréstől az üzemeltetésig, kisebb iterációk sorozatára bontsuk. Eltérően a vízesés modelltől, amelyben például a tervezés teljesen megelőzni az implementációt, itt minden iterációban van tervezés és implementációi is. Lehet, hogy valamelyik iterációban az egyik sokkal hangsúlyosabb, mint a másik, de ez természetes.

A folyamatos finomítás lehetővé teszi, hogy mélyen megértsük a feladatot és felderítsük az ellentmondásokat. Minden iteráció kiegészíti a már kifejlesztett prototípust. A kiegészítést inkrementumnak is nevezzük. Azok a módszertanok, amik a folyamatra teszik a hangsúlyt, azaz az iterációra, azokat iteratív módszertanoknak nevezzük. Azokat, amelyek az iteráció termékére, az inkrementumra teszik a hangsúlyt, azokat inkrementális módszertanoknak hívjuk. A mai módszertanok nagy része, kezdve a prototípus modelltől egészen az agilis modellekig, ebbe a családba tartoznak.

A kiegészítés hozzáadásával növekvő részrendszer jön létre, amelyet tesztelni kell. Az új kódot unit-teszttel teszteljük. Regressziós teszttel kell ellenőrizni, hogy a régi kód továbbra is működik-e az új kód hozzáadása és a változások után. Az új és a régi kód együttműködését integrációs teszttel teszteljük. Ha egy mérföldkőhöz vagy prototípus bemutatáshoz érkezünk, akkor van felhasználói átvételi teszt is. Egyébként csak egy belső átvételi teszt van az iteráció végén.





Ezt a megközelítést több módszertan is alkalmazza, például a prototípus modell, a gyors alkalmazásfejlesztés (RAD), a Rational Unified Process (RUP) és az agilis fejlesztési modellek. Itt ezeknek a módszertanoknak a közös részét, az iterációt ismertetjük. Egy iteráció a következő feladatokból áll:

1. Üzleti folyamatok elemzése
2. Követelményelemzés
3. Elemzés és tervezés
4. Implementáció
5. Tesztelés
6. Értékelés

Az iteratív modell fő ereje abban rejlik, hogy az életciklus lépései nem egymás után jönnek, mint a strukturált módszertanok esetén, hanem időben átfedik egymást. Minden iterációban van elemzés, tervezés, implementáció és tesztelés. Ezért, ha találunk egy félreértést, akkor nem kell visszalépni, hanem néhány iteráció segítségével oldjuk fel a félreértést. Ez az jelenti, hogy kevésbé tervezhető a fejlesztés ideje, de jól alkalmazkodik az igények változásához.

Mivel a fejlesztés lépéseit mindig ismételgetjük, ezért azt mondjuk, hogy ezek időben átfedik egymást, hiszen minden szakaszban minden lépést végre kell hajtani. A kezdeti iterációkban több az elemzés, a végéhez közeledve egyre több a tesztelés. Már a legelső szakaszban is van tesztelés, de ekkor még csak a teszttervet készítjük. Már a legelső szakaszban is van implementáció, de ekkor még csak az architektúra osztályait hozzuk létre. És így tovább.

A feladatot több iterációra bontjuk. Ezeket általában több kisebb csapat implementálja egymással versengve. Aki gyorsabb, az választhat iterációt a meglévők közül. A választás nem teljesen szabad, a legnagyobb prioritású feladatok közül kell választani. A prioritás meghatározása különböző lehet, általában a leggyorsabban megvalósítható és legnagyobb üzleti értékű, azaz a legnagyobb üzleti megtérüléssel (angolul: return of investment) bíró feladat a legnagyobb prioritású.

Üzleti folyamatok elemzése: Első lépésben meg kell ismerni a megrendelő üzleti folyamatait. Az üzleti folyamatok modellezése során fel kell állítani egy projekt fogalomtárat. A lemodellezett üzleti folyamatokat egyeztetni kell a megrendelővel, hogy ellenőrizzük jól értjük-e az üzleti logikát. Ezt üzleti elemzők végzik, akik a megrendelők és a fejlesztők fejével is képesek gondolkozni.

Követelményelemzés: A követelmény elemzés során meghatározzuk a rendszer funkcionális és nemfunkcionális követelményeit, majd ezekből funkciókat, képernyőterveket készítünk. Ez a lépés az egész fejlesztés elején nagyon hangsúlyos, hiszen a kezdeti iterációk célja a követelmények felállítása. Későbbiekben csak a funkcionális terv finomítása a feladata. **Fontos, hogy a követelményeket egyeztessük a megrendelőkkel**. Ha a finomítás során ellentmondást fedezünk fel, akkor érdemes tisztázni a kérdést a megrendelővel.

Elemzés és tervezés: Az elemzés és tervezés során a követelmény elemzés termékeiből megpróbáljuk elemezni a rendszert és megtervezni azt. A nemfunkcionális követelményekből lesz az architekturális terv. Az architekturális terv alapján tervezzük az alrendszereket és a köztük levő kapcsolatokat. Ez

a kezdeti iterációk feladata. A funkcionális követelmények alapján tervezzük meg az osztályokat, metódusokat és az adattáblákat. Ezek a későbbi iterációk feladatai.

Implementáció: Az implementációs szakaszra ritkán adnak megszorítást az iteratív módszertanok. Általában a bevett technikák alkalmazását ajánlják, illetve szerepköröket írnak elő. Pl.: a fejlesztők fejlesztik a rendszert, a fejlesztők szoros kapcsolatban vannak a tervezőkkel, továbbá van egy kód ellenőr, aki ellenőrzi, hogy a fejlesztők által írt programok megfelelnek-e a tervezők által kitalált tervezési és programozási irányelveknek. Ebben a szakaszban a programozók unit-teszttel biztosítják a kód minőségét.

Tesztelés: A tesztelési szakaszban különböző tesztelési eseteket találunk ki, ezeket unit-tesztként valósítjuk meg. Itt vizsgáljuk meg, hogy az elkészült kód képes-e együttműködni a program többi részével, azaz integrációs tesztet hajtunk végre. Regressziós tesztek segítségével ellenőrizzük, hogy ami eddig kész volt, az nem romlott el. Ehhez lefuttatjuk az összes unit-tesztet. Rendszerteszt csak a késői tesztelési fázisokban van.

Értékelés: A fejlesztés minden ciklusában el kell dönteni, hogy az elkészült verziót elfogadjuk-e, vagy sem. Ha nem, akkor újra indul ez az iteráció. Ha igen, vége ennek az iterációnak. Az így elkészült kódot feltöltjük a verziókövető rendszerbe, hogy a többi csapat is hozzáférjen. Az értékelés magában foglal egy átvételi tesztet is. Ha a megrendelő nem áll rendelkezésre, akkor általában a csoportok munkáját összefogó vezető programozó / tervező helyettesíti. Amennyiben a folyamat során elértünk egy mérföldkőhöz, akkor általában át kell adnunk egy köztes prototípust is. Ekkor mindig rendelkezésre áll a megrendelő, hogy elvégezzük a felhasználói átvételi tesztet.

Támogató tevékenységek, napi fordítás: Az iterációktól függetlenül úgynevezett támogató folyamatok is zajlanak a szoftver cégen belül. Ilyen például a rendszergazdák vagy a menedzsment tevékenysége. Az iterációk szemszögéből a legfontosabb az úgynevezett a napi fordítás (daily build). Ez azt jelenti, hogy minden nap végén a verziókövető rendszerben lévő forráskódot lefordítjuk. Minden csapat igyekszik a meglévő kódhoz igazítani a sajátját, hogy lehetséges legyen a fordítás. Aki elrontja a napi fordítást, és ezzel nehezíti az összes csapat következő napi munkáját, az büntetésre számíthat. Ez a cég hagyományaitól függ, általában egy hétig ő csinálja a napi fordítás és emiatt sokszor sokáig bent kell maradnia.

Végül vagy elérjük azt a pontot, ahol azt mondjuk, hogy ez így nem elkészíthető, vagy azt mondjuk, hogy minden felmerült igényt kielégít a szoftverünk és szállíthatjuk a megrendelőnek.

Gyors alkalmazásfejlesztés – RAD

A gyors alkalmazásfejlesztés vagy ismertebb nevén RAD (Rapid Application Development) egy olyan elgondolás, amelynek lényege a szoftver gyorsabb és jobb minőségű elkészítése. Ezt a következők által érhetjük el:

1. Korai prototípus készítés és ismétlődő felhasználói átvételi tesztek.
2. A csapat - megrendelő és a csapaton belüli kommunikációban kevésbé formális.
3. Szigorú ütemterv, így az újítások mindig csak a termék következő verziójában jelennek meg.
4. Követelmények összegyűjtése fókusz csoportok és munkaértekezletek használatával.
5. Komponensek újrahasznosítása.

Ezekhez a folyamatokhoz több szoftvergyártó is készített segédeszközöket, melyek részben vagy egészben lefedik a fejlesztés fázisait, mint például:

1. követelmény összegyűjtő eszközök,
2. tervezést segítő eszközök,
3. prototípus készítő eszközök,
4. csapatok kommunikációját segítő eszközök.

A RAD elsősorban az objektumorientált programozással kacsolódik össze, már csak a komponensek újrahasznosítása okán is. Összehasonlítva a hagyományos fejlesztési metódusokkal (pl.: vízesés modell), ahol az egyes fejlesztési fázisok jól elkülönülnek egymástól, a RAD sokkal rugalmasabban. Gyakori probléma, hogy a tervezésbe hiba csúszik, és az csak a megvalósítási vagy a tesztelési fázisban jön elő, ráadásul az elemzés és a tesztelési fázis között hat-hét hónap is eltelhet. Vagy ha menetközbe megváltoznak az üzleti körülmények, és már a megvalósítási fázisban járunk, vagy csak rájöttek a megrendelők, hogy valamit mégis másképpen szeretnének, akkor szintén gondban vagyunk. A RAD válasza ezekre a problémákra a gyorsaság. Ha gyorsan hozzuk létre a rendszert, akkor ezen rövid idő alatt nem változnak a követelmények, az elemzés és tesztelés között nem hat-hét hónap, hanem csak hat-hét hét telik el.

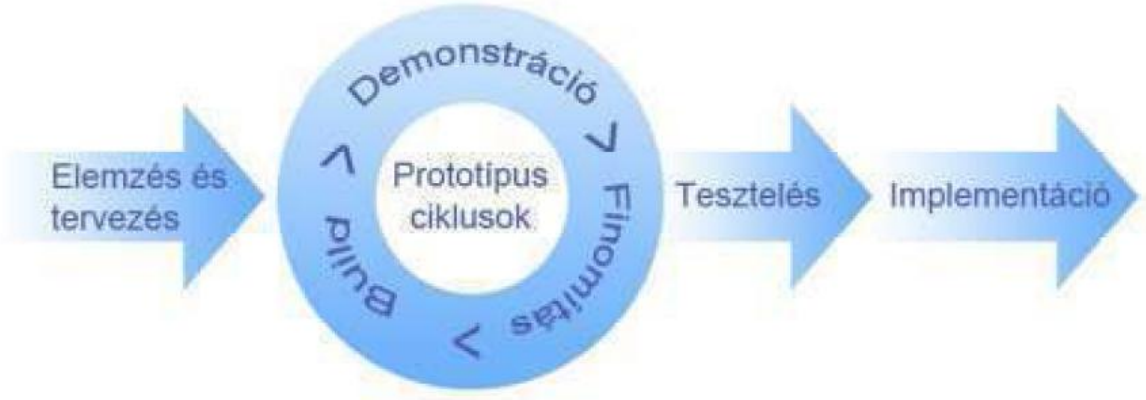
A gyorsaság eléréséhez sok meglévő komponenst kell felhasználni, amit a csapatnak jól kell ismernie. A komponensek lehetnek saját fejlesztésűek vagy megvásároltak. Komponenst vásárolni nagy kockázat, mert ha hiba van benne, azt nem tudjuk javítani, ha nem kapjuk meg a forrást, de még úgy is nagyon nehéz. Ezért a komponens gyártók nagyon alaposan tesztelik terméküket.

A RAD az elemzést, a tervezést, a megvalósítást, és a tesztelést rövid, ismétlődő ciklusok sorozatába tömöríti, és ennek sok előnye van a hagyományos modellekkel szemben. A fejlesztés során általában kis csoportokat hoznak létre fejlesztőkből, végfelhasználókból, ez az úgynevezett fókusz csoport. Ezek a csapatok az ismétlődő, rövid ciklusokkal vegyítve hatékonyabbá teszik a kommunikációt, optimalizálják a fejlesztési sebességet, egységesítik az elképzeléseket és célokat, valamint leegyszerűsítik a folyamat felügyeletét.

Öt fejlesztési lépés a RAD-ban:

1. Üzleti modellezés: Az üzleti funkciók közötti információ áramlást olyan kérdések feltevésével tudjuk felderíteni, mint hogy milyen információk keletkeznek, ezeket ki állítja elő, az üzleti folyamatot milyen információk irányítják, vagy hogy ki irányítja.
2. Adat modellezés: Az üzleti modellezéssel összegyűjtöttük a szükséges adatokat, melyekből adat objektumokat hozunk létre. Beazonosítjuk az attribútumokat és a kapcsolatokat az adatok között.
3. Folyamat modellezés: Az előzőleg létrehozott adatmodellhez szükséges műveletek (bővítés, törlés, módosítás) meghatározása, úgy hogy létrehozzuk a kellő információáramlást az üzleti funkciók számára.
4. Alkalmazás előállítása: A szoftver előállításának megkönnyítése automatikus eszközökkel.
5. Tesztelés: Az új programkomponensek tesztelése, a már korábban tesztelt komponenseket már nem szükséges újra vizsgálni. Ez gyorsítja a folyamatot.

Hátránya, hogy magasan képzett fejlesztőkre van szükség, emellett fontos a fejlesztők és a végfelhasználók elkötelezettsége a sikeres szoftver iránt. Ha a projekt nehezen bontható fel modulokra, akkor nem a legjobb választás a RAD. Nagyobb rendszerek fejlesztése ezzel a módszertannal kockázatos.



Agilis szoftverfejlesztés

Az agilis szoftverfejlesztés valójában iteratív szoftverfejlesztési módszerek egy csoportjára utal, amelyet 2001-ben az Agile Manifesto nevű kiadványban öntöttek formába. Az agilis fejlesztési módszerek (nevezik adaptívnak is) egyik fontos jellemzője, hogy a résztvevők, amennyire lehetséges megpróbálnak alkalmazkodni a projekthez. Ezért fontos például, hogy a fejlesztők folyamatosan tanuljanak.

Az agilis szoftverfejlesztés szerint értékesebbek:

1. az egyének és interaktivitás szemben a folyamatokkal és az eszközökkel,
2. a működő szoftver szemben a terjedelmes dokumentációval,
3. az együttműködés a megrendelővel szemben a szerződéses tárgyalásokkal,
4. az alkalmazkodás a változásokhoz szemben a terv követésével.

Az agilis szoftverfejlesztés alapelvei:

1. A legfontosabb a megrendelő kielégítése használható szoftver gyors és folyamatos átadásával.
2. Még a követelmények kései változtatása sem okoz problémát.
3. A működő szoftver / prototípus átadása rendszeresen, a lehető legrövidebb időn belül.
4. Napi együttműködés a megrendelő és a fejlesztők között.
5. A projektek motivált egyének köré épülnek, akik megkapják a szükséges eszközöket és támogatást a legjobb munkavégzéshez.
6. A leghatékonyabb kommunikáció a szemtől-szembeni megbeszélés.
7. Az előrehaladás alapja a működő szoftver.
8. Az agilis folyamatok általi fenntartható fejlesztés állandó ütemben.
9. Folyamatos figyelem a technikai kitűnőségnek.
10. Egyszerűség, a minél nagyobb hatékonyságért.
11. Önszervező csapatok készítik a legjobb terveket.
12. Rendszeres időközönként a csapatok reagálnak a változásokra, hogy még hatékonyabbak legyenek.

Az agilis szoftverfejlesztésnek nagyon sok fajtája van. Ebben a jegyzetben csak ezt a kettőt tárgyaljuk:

1. Scrum
2. Extrém Programozás (XP)

Ezek a következő közös jellemzőkkel bírnak:

1. Kevesebb dokumentáció.
2. Növekvő rugalmasság, csökkenő kockázat.
3. Könnyebb kommunikáció, javuló együttműködés.
4. A megrendelő bevonása a fejlesztésbe.

Kevesebb dokumentáció: Az agilis metódusok alapvető különbsége a hagyományosakhoz képest, hogy a projektet apró részekre bontják, és mindig egy kisebb darabot tesznek hozzá a termékhez, ezeket egytől négy hétig terjedő ciklusokban (más néven keretekben vagy idődobozokban) készítik el, és ezek a ciklusok ismétlődnek. Ezáltal nincs olyan jellegű részletes hosszú távú tervezés, mint például a vízeséses modellnél, csak az a minimális, amire az adott ciklusban szükség van. Ez abból az elvből indul ki, hogy nem lehet előre tökéletesen, minden részletre kiterjedően megtervezni egy szoftvert, mert vagy a tervben lesz hiba, vagy a megrendelő változtat valamit.

Növekvő rugalmasság, csökkenő kockázat: Az agilis módszerek a változásokhoz adaptálható technikákat helyezik előnybe a jól tervezhető technikákkal szemben. Ennek megfelelően iterációkat használnak. Egy iteráció olyan, mint egy hagyományos életciklus: tartalmazza a tervezést, a követelmények elemzését, a kódolást, és a tesztelést. Egy iteráció maximum egy hónap terjedelmű, így nő a rugalmasság, valamint csökken a kockázat, hiszen az iteráció végén átvételi teszt van, ami után megrendelő megváltoztathatja eddigi követelményeit. Minden iteráció végén futóképes változatot kell kiadniuk a csapatoknak a kezükből.

Könnyebb kommunikáció, javuló együttműködés: Jellemző, hogy a fejlesztő csoportok önszervezőek, és általában nem egy feladatra specializálódottak a tagok, hanem többféle szakterületről kerülnek egy csapatba, így például programozok és tesztelők. Ezek a csapatok ideális esetben egy helyen, egy irodában dolgoznak, a csapatok mérete ideális esetben 5-9 fő. Mindez leegyszerűsíti a tagok közötti kommunikációt és segíti a csapaton belüli együttműködést. Az agilis módszerek előnyben részesítik a szemtől szembe folytatott kommunikációt az írásban folytatott eszmecserével szemben.

A megrendelő bevonása a fejlesztésbe: Vagy személyesen a megrendelő vagy egy kijelölt személy, aki elkötelezi magát a termék elkészítése mellett, folyamatosan a fejlesztők rendelkezésére áll, hogy a menet közben felmerülő kérdéseket minél hamarabb meg tudja válaszolni. Ez a személy a ciklus végén is részt vesz az elkészült prototípus kiértékelésében. Fontos feladata az elkészítendő funkciók fontossági sorrendjének felállítása azok üzleti értéke alapján. Az üzleti értékből és a fejlesztő csapat által becsült fejlesztési időből számolható a befektetés megtérülése (Return of Investment, ROI). A befektetés megtérülése az üzleti érték és a fejlesztési idő hányadosa.

Az agilis módszertanok nagyon jól működnek, amíg a feladatot egy közepes méretű (5-9 fős) csapat képes megoldani. Nagyobb csoportok esetén nehéz a csapat szellem kialakítása. Ha több csoport dolgozik ugyanazon a célon, akkor köztük a kommunikáció nehézkes. Ha megrendelő nem hajlandó egy elkötelezett munkatársát a fejlesztő csapat rendelkezésére bocsátani, akkor az kiváltható egy üzleti elemzővel, aki átlátja a megrendelő üzleti folyamatait, de ez kockázatos.

Scrum

A Scrum egy agilis szoftverfejlesztési metódus. Jellegzetessége, hogy fogalmait az amerikai futballból, más néven rugby, meríti. Ilyen fogalom, maga a Scrum is, amely dulakodást jelent. A módszertan jelentős szerepet tulajdonít a csoporton belüli összetartásnak. A csoporton belül sok a találkozó, a kommunikáció, lehetőség van a gondok megbeszélésre is. Az ajánlás szerint jó, ha a csapat egy helyen dolgozik és szóban kommunikál.

A Scrum által előírt fejlesztési folyamat röviden így foglalható össze:

A Product Owner létrehoz egy Product Backlog-ot, amelyre a teendőket felhasználói sztoriként veszi fel. A sztorikat prioritással kell ellátni és megmondani, mi az üzleti értékük. Ez a Product Owner feladata.

A Sprint Planning Meetingen a csapat tagjai megbeszélik, hogy mely sztorik megvalósítását vállalják el, lehetőleg a legnagyobb prioritásúakat. Ehhez a sztorikat kisebb feladatokra bontják, hogy megbecsülhessék mennyi ideig tart megvalósítani azokat.

Ezután jön a sprint, ami 2-4 hétig tart. A sprint időtartamát az elején fixálja a csapat, ettől eltérni nem lehet. Ha nem sikerül befejezni az adott időtartam alatt, akkor sikertelen a sprint, ami büntetést, általában prémium megvonást, von maga után. A sprinten belül a csapat és a Scrum Master naponta megbeszélik a történteket a *Daily Meetingen*. Itt mindenki elmondja, hogy mit csinált, mi lesz a következő feladata, és milyen akadályokba (impediment) ütközött.

A sprint végén következik a Sprint Review, ahol a csapat bemutatja a sprint alatt elkészült sztorikat. Ezeket vagy elfogadják, vagy nem.

Majd a Sprint Retrospective találkozó következik, ahol a Sprint során felmerült problémákat tárgyalja át a csapat. A megoldásra konkrét javaslatokat kell tenni.

Ezek után újra a Sprint Planning Meeting következik. A fejlesztett termék az előtt piacra kerülhet, hogy minden sztorit megvalósítottak volna.

A csapatban minden szerepkör képviselője megtalálható, így van benne fejlesztő és tesztelő is. Téves azt gondolni, hogy a sprint elején a tesztelő is programot ír, hiszen, amíg nincs program, nincs mit tesztelni. Ezzel szemben a tesztelő a sprint elején a teszttervet készít, majd kidolgozza a teszteseteket, végül, amikor már vannak kész osztályok, unit-teszteket ír, a változásokat regressziós teszttel ellenőrzi.

A Scrum, mint minden agilis módszertan, arra épít, hogy a fejlesztés közben a megrendelő igényei változhatnak. A változásokhoz úgy alkalmazkodik, a Product Backlog folyamatosan változhat. Az erre épülő dokumentumok folyamatosan finomodnak, tehát könnyen változtathatók. A csapatok gyorsan megvalósítják a szükséges változásokat.

A Scrum tökélyre viszi az egy csapaton belüli hatékonyságot. Ha több csapat is dolgozik egy fejlesztésen, akkor köztük lehetnek kommunikációs zavarok, ami a módszertan egyik hátránya.

A Scrum két nagyon fontos fogalma a sprint és az akadály.

Sprint (vagy futam): Egy előre megbeszélt hosszúságú fejlesztési időszak, általában 2-4 hétig tart, kezdődik a Sprint Planning-gel, majd a Retrospective-vel zárul. Ez a Scrum úgynevezett iterációs ciklusa, addig kell ismételni, amíg a Product Backlog-ról el nem tűnnek a megoldásra váró felhasználói sztorik. Alapelv, hogy minden sprint végére egy potenciálisan leszállítható szoftvert kell előállítani a csapatnak, azaz egy prototípust. A sprint tekinthető két mérföldkő közti munkának.

Akadály (Impediment): Olyan gátló tényező, amely a munkát hátráltatja. Csak és kizárólag munkahelyi probléma tekinthető akadálynak. A csapattagok magánéleti problémái nem azok. Akadály például, hogy lejárt az egyik szoftver licence, vagy szükség lenne egy plusz gépre a gyorsabb haladáshoz, vagy több memóriára az egyik gépbe, vagy akár az is lehet, hogy 2 tag megsértődött egymásra. Ilyenkor kell a Scrum Masternek elhárítani az akadályokat, hogy a munka minél gördülékenyebb legyen.

A módszertan szerepköröket, megbeszéléseket és elkészítendő termékeket ír elő.

Scrum – Szerepkörök

A módszertan kétféle szerepkört különböztet meg, ezek a disznók és a csirkék. A megkülönböztetés alapja egy vicc:

A disznó és a csirke mennek az utcán. Egyszer csak a csirke megszólal: „Te, nyissunk egy éttermet!” Mire a disznó: „Jó ötlet, mi legyen a neve?” A csirke gondolkozik, majd rávágja: „Nevezzük Sonkástojásnak!” A disznó erre: „Nem tetszik valahogy, mert én biztosan mindent beleadnék, te meg éppen csak hogy részt vennél benne.”

A disznók azok, akik elkötelezettek a szoftver projekt sikerében. Ők azok, akik a „vérüket” adják a projekt sikeréért, azaz felelősséget vállalnak érte. A csirkék is érdekeltek a projekt sikerében, ők a haszonélvezői a sikernek, de ha esetleg mégse sikeres a projekt, akkor az nem az ő felelősségük.

Disznók:

1. Scrum mester (Scrum Master)
2. Terméktulajdonos (Product Owner)
3. Csapat (Team)

Csirkék:

1. Üzleti szereplők (Stakeholders)
2. Menedzsment (Managers)

Scrum mester (Scrum Master): A Scrum mester felügyeli és megkönnyíti a folyamat fenntartását, segíti a csapatot, ha problémába ütközik, illetve felügyeli, hogy mindenki betartja-e a Scrum alapvető szabályait. Ilyen például, hogy a Sprint időtartama nem térhet el az előre megbeszélttől, még akkor sem, ha az elvállalt munka nem lesz kész. Akkor is nemet kell mondania, ha a Product Owner a sprint közben azt találja ki, hogy az egyik sztorit, amit nem vállaltak be az adott időszakra, el kellene készíteni, mert mondjuk megváltoztak az üzleti körülmények. Lényegében ő a projekt menedzser.

Termék tulajdonos (Product Owner): A megrendelő szerepét tölti be, ő a felelős azért, hogy a csapat mindig azt a részét fejlessze a terméknek, amely éppen a legfontosabb, vagyis a felhasználói sztorik fontossági sorrendbe állítása a feladata a Product Backlog-ban. A Product Owner és a Scrum Master nem lehet ugyanaz a személy.

Csapat (Team): Ők a felelősek azért, hogy az aktuális sprintre bevállalt feladatokat elvégezzék, ideális esetben 5-9 fő alkot egy csapatot. A csapatban helyet kapnak a fejlesztők, tesztelők, elemzők. Így nem a váltófutásra jellemző stafétaváltás (mint a vízesés modellnél), hanem a futballra emlékeztető passzolgatás, azaz igazi csapatjáték jellemzi a csapatot.

Üzleti szereplők, pl.: megrendelők, forgalmazók (Stakeholders, i.e., customers, vendors): A megrendelő által jön létre a projekt, ő az, aki majd a hasznát látja a termék elkészítésének, a Sprint Review során kap szerepet a folyamatban.

Menedzsment (Managers): A menedzsment feladata a megfelelő környezet felállítása a csapatok számára. Általában a megfelelő környezeten túl a lehető legjobb környezet felállítására törekszenek.

Scrum – Megbeszélések

Sprint Planning Meeting (futamtervező megbeszélés): Ezen a találkozón kell megbeszélni, hogy ki mennyi munkát tud elvállalni, majd ennek tudatában dönti el a csapat, hogy mely sztorikat vállalja be a következő sprintre. Emellett a másik lényeges dolog, hogy a csapat a Product Owner-rel megbeszéli, majd teljes mértékben megérti, hogy a vevő mit szeretne az adott sztoritól, így elkerülhetőek az esetleges félreértésekből adódó problémák. Ha volt Backlog Grooming, akkor nem tart olyan sokáig a Planning, ugyanis a csapat ismeri a Backlog-ot, azon nem szükséges finomítani, hacsak a megrendelőtől nem érkezik ilyen igény. A harmadik dolog, amit meg kell vizsgálni, hogy a csapat hogyan teljesített az előző sprintben, vagyis túlvállalta-e magát vagy sem. Ha túl sok sztorit vállaltak el, akkor le kell vonni a következtetést, és a következő sprintre kevesebbet vállalni. Ez a probléma leginkább az új, kevéssé összeszokott csapatokra jellemző, ahol még nem tudni, hogy mennyi munkát bír elvégezni a csapat. Ellenkező esetben, ha alulvállalta magát egy csapat, akkor értelemszerűen többet vállaljon, illetve, ha ideális volt az előző sprint, akkor hasonló mennyiség a javasolt.

Backlog Grooming/Backlog Refinement: A Product Backlog finomítása a Teammel együtt, előfordulhat például, hogy egy taszk túl nagy, így story lesz belőle, és utána taszkokra bontva lesz feldolgozva. Ha elmarad, akkor a Sprint Planning hosszúra nyúlhat, valamint abban is nagy segítség, hogy a csapat tökéletesen megértse, hogy mit szeretne a megrendelő.

Daily Meeting/Daily Scrum: A sprint ideje alatt minden nap kell tartani egy rövid megbeszélést, ami maximum 15 perc, és egy előre megbeszélt időpontban, a csapattagok és a Scrum Master jelenlétében történik (mások is ott lehetnek, de nem szólhatnak bele). Érdekesség, hogy nem szabad leülni, mindenki áll, ezzel is jelezve, hogy ez egy rövid találkozó. Három kérdésre kell válaszolnia a csapat tagjainak, ezek a következőek:

1. Mit csináltál a tegnapi megbeszélés óta?
2. Mit fogsz csinálni a következő megbeszélésig?
3. Milyen akadályokba ütköztél az adott feladat megoldása során?

Sprint Review Meeting (Futam áttekintés): Minden sprint végén összeülnek a szereplők, és megnézik, hogy melyek azok a sztorik, amelyeket sikerült elkészíteni, illetve az megfelel-e a követelményeknek. Ekkor a sztori állapotát készre állítják. Fontos, hogy egy sztori csak akkor kerülhet ebbe az állapotba, ha minden taszkja elkészült, és a Review-on elfogadták. Ezen a megrendelő is jelen van.

Sprint Retrospective (Visszatekintés): Ez az egyik legfontosabb meeting. A Scrum egyik legfontosabb funkciója, hogy felszínre hozza azokat a problémákat, amelyek hátráltatják a fejlesztőket a feladatmegoldásban, így ha ezeket az akadályokat megoldjuk, a csapat jobban tud majd alkalmazkodni a következő sprint alatt a feladathoz. Problémák a Daily Meetingen is előkerülnek, de ott inkább a személyeket érintő kérdések vannak napirenden, míg itt a csapatmunka továbbfejlesztése az elsődleges.

Scrum – Termékek

Product Backlog (termék teendő lista): Ez az a dokumentum, ahol a Product Owner elhelyezi azokat az elemeket, más néven sztorikat, amelyeket el kell készíteni. Ez egyfajta kívánságlista. A Product Owner minden sztorihoz prioritást, fontossági sorrendet rendel, így tudja szabályozni, hogy melyeket kell elsősorban elkészíteni, így a Sprint Planning során a csapattagok láthatják, hogy ami a Backlog-ban legfelül van, azt szeretné a vevő leghamarabb készen látni, annak van a legnagyobb üzleti értéke. Emellett a csapatok súlyozzák az elemeket aszerint, hogy melynek az elkészítéséhez kell a kevesebb munka, így azonos prioritás mellett a kevesebb munkát igénylő elemnek nagyobb a befektetés megtérülése (Return of Investment, ROI). Az üzleti érték meghatározása a Product Owner, a munka megbecslése a csapat feladata. A kettő hányadosa a ROI.

Sprint Backlog (futam teendő lista): Ebben a dokumentumban az aktuális sprintre bevállalt munkák, storyk vannak felsorolva, ezeket kell adott időn belül a csapatnak megvalósítania. A sztorik tovább vannak bontva taszkokra, és ezeket a taszkokat vállalják el a tagok a Daily Meeting során. Ez a feldarabolása a feladatoknak a feladat minél jobb megértését segíti.
Burn down chart (Napi Eredmény Kimutatás): Ez egy diagram, amely segít megmutatni, hogy az ideális munkatempóhoz képest hogyan halad a csapat az aktuális sprinten belül. Könnyen leolvasható róla, hogy a csapat éppen elakadt-e egy ponton, akár arra is lehet következtetni, hogy ilyen iramban kész lesz-e minden a sprint végére. Vagy éppen ellenkezőleg, sikerült felgyorsítani az iramot, és időben, vagy akár kicsit hamarabb is kész lehet a bevállalt munka.

Extrém programozás

Az extrém programozás (angolul: Extreme Programming, vagy röviden: XP) egy agilis módszertan. A nevében az extrém szó onnan jön, hogy az eddigi módszertanokból átveszi a jól bevált technikákat és azokat nem csak jól, hanem extrém jól alkalmazza, minden mást feleslegesnek tekint. Gyakran összekeverik a „programozzunk összeesésig” módszerrel, amivel egy-két 24 órás vagy akár 48 órás programozó versenyen találkozhatunk.

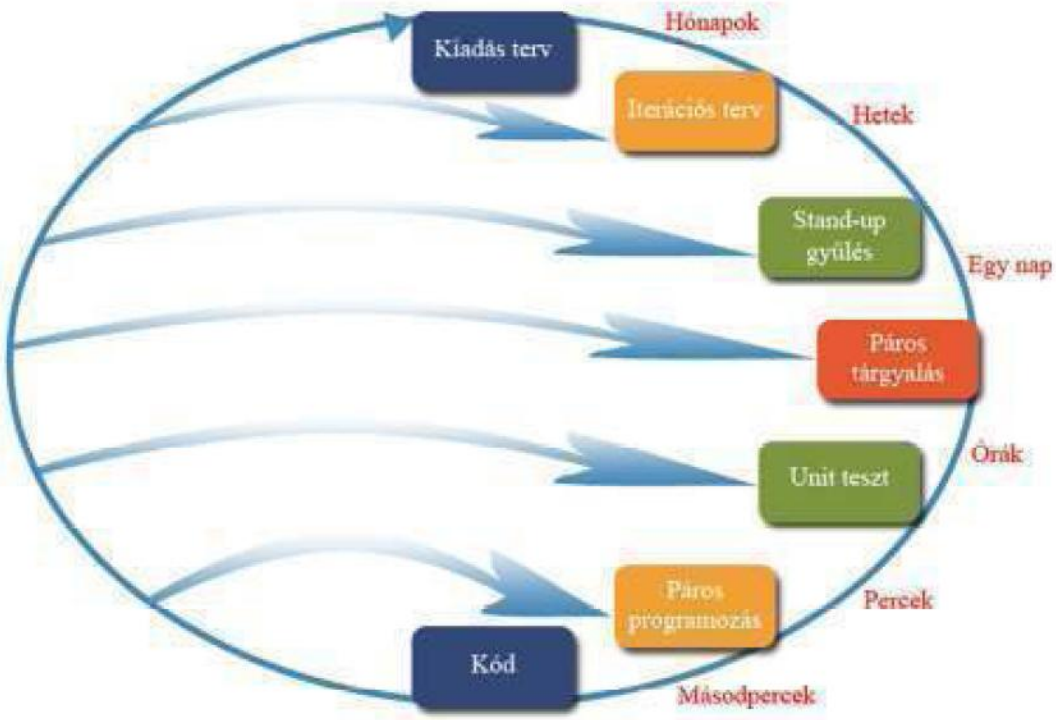
Az extrém programozás 4 tevékenységet ír elő. Ezek a következők:

1. Kódolás: A forráskód a projekt legfontosabb terméke, ezért a kódolásra kell a hangsúlyt helyezni. Igazán kódolás közben jönnek ki a feladat nehézségei, hiába gondoltuk azt át előtte. A kód a legalkalmasabb a két programozó közötti kommunikációra, mivel azt nem lehet kétféleképpen érteni. A kód alkalmas a programozó gondolatainak kifejezésére.
2. Tesztelés: Addig nem lehetünk benne biztosak, hogy egy funkció működik, amíg nem teszteltük. Az extrém felfogás szerint kevés tesztelés kevés hibát talál, extrém sok tesztelés megtalálja mind. A tesztelés játssza a dokumentáció szerepét. Nem dokumentáljuk a metódusokat, hanem unit-teszteket fejlesztünk hozzá. Nem készítünk követelmény specifikációt, hanem átvételi teszteseteket fejlesztünk a megértett követelményekből.
3. Odafigyelés: A fejlesztőknek oda kell figyelniük a megrendelőkre, meg kell érteniük az igényeiket. El kell magyarázni nekik, hogy hogyan lehet technikailag kivitelezni ezeket az igényeket, és ha egy igény kivitelezhetetlen, ezt meg kell értetni a megrendelővel.
4. Tervezés: Tervezés nélkül nem lehet szoftvert fejleszteni, mert az ad- hoc megoldások átláthatatlan struktúrához vezetnek. Mivel fel kell készülni az igények változására, ezért úgy kell megtervezni a szoftvert, hogy egyes komponensei amennyire csak lehet függetlenek legyenek a többitől. Ezért érdemes pl. objektum orientált tervezési alapelveket használni.

Néhány extrém programozásra jellemző technika:

1. Páros programozás (pair programming): Két programozó ír egy kódot, pontosabban az egyik írja, a másik figyeli. Ha hibát lát vagy nem érti, akkor azonnal szól. A két programozó folyamatosan megbeszélik hogyan érdemes megoldani az adott problémát.
2. Teszt vezérelt fejlesztés (test driven development): Már a metódus elkészítése előtt megírjuk a hozzá tartozó unit-teszteket. Ezt néha hívják először a teszt (test-first) megközelítésnek is.
3. Forráskód átnézés (code review): Az elkészült nagyobb modulokat, pl. osztályokat, egy vezető fejlesztő átnézi, hogy van-e benne hiba, nem érthető, nem dokumentált rész. A modul fejlesztői elmagyarázzák mit és miért csináltak. A vezető fejlesztő elmondja, hogyan lehet ezt jobban, szebben csinálni.
4. Folyamatos integráció (continuous integration): A nap (vagy a hét) végén, a verziókövető rendszerbe bekerült kódokat integrációs teszt alá vetjük, hogy kiderüljön, hogy azok képesek-e együttműködni. Így nagyon korán kiszűrhető a programozók közti félreértés.
5. Kódszépítés (refactoring): A már letesztelt, működő kódot lehet szépíteni, ami esetleg lassú, rugalmatlan, vagy egyszerűen csak csúnya. A kódszépítés előfeltétele, hogy legyen sok unit-teszt. A szépítés során nem szabad megváltoztatni a kód funkcionalitását, de a szerkezet, pl. egy

metódus törzse, szabadon változtatható. A szépítés után minden unit-tesztet le kell futtatni, nem csak a megváltozott kódhoz tartozókat, hogy lássuk, a változások okoztak-e hibát.



Az extrém programozás akkor működik jól, ha a megrendelő biztosítani tud egy munkatársat, aki átlátja a megrendelő folyamatait, tudja, mire van szükség. Ha a változó, vagy a menet közben kiderített követelmények miatt gyakran át kell írni már elkészült részeket, akkor az extrém programozás nagyon rossz választás. Kezdő programozók esetén az extrém programozás nem alkalmazható, mert nincs elég tapasztalatuk az extrém módszerek alkalmazásához.

Az extrém programozás legnagyobb erénye, hogy olyan fejlesztési módszereket hozott a felszínre, amik magas minőséget biztosítanak. Ezek, mint pl. a páros programozás, nagyon népszerűek lettek.