**6. TÉTEL**

**Módszertanok**

A módszertanok feladata, hogy meghatározzák, hogy a szoftver életciklus egyes lépései milyen sorrendben követik egymást, milyen dokumentumokat, szoftver termékeket kell előállítani és hogyan. Egy nagy szabálykönyvre emlékeztetnek, ami pontosan leírja, hogyan kell szoftvert „főzni”.

A módszertanokat több szempontból is osztályozhatjuk. Az első szempont, hogy milyen sorrendben követik egymást az életciklus fázisai. E szerint van:

• lineáris,

• spirális

• iteratív (vagy inkrementális).

A második szempont, hogy milyen implementációs nyelvet részesít előnyben a módszertan. E szerint van:

• folyamat orientált,

• adat központú,

• strukturált (ezen belül lehet top-down és button-up),

• objektum orientált,

• szerviz orientált,

• esetleg ezek keveréke.

A harmadik szempont, hogy milyen megközelítést használ a módszertan a célja eléréséhez. Mivel a cél a sikeres szoftverprojekt, ezért a megközelítés alatt általában egy olyan technikát értünk, ami a félreértések elkerülésére szolgál. Ugyanakkor a megközelítés alatt a programozásra tett ajánlást is érthetjük. E szerint van:

• jól dokumentált,

• prototípus alapú,

• rapid,

• agilis,

• extrém,

• esetleg ezek keveréke.

A negyedik szempont, hogy mennyire szigorúan követelik meg a jól dokumentáltságot. E szerint van:

• könnyűsúlyú (lightweight) módszertan,

• nehézsúlyú (heavyweight) módszertan.

Az ötödik szempont, hogy mit helyez a modell középpontjába a módszertan. E szerint van:

• adat központú,

• folyamat központú,

• követelmény központú,

• használati eset központú,

• teszt központú,

• felhasználó központú,

• ember központú,

• csapat központú,

• esetleg ezek keveréke.

Sok módszertan nem sorolható be egyetlen kategória alá, mert például az adatbázis tervezéshez adatközpontú módszereket ír elő, egyébként használati eset központú. 28 Általában minden módszertanra jellemzőek a következőek:

• Az elemzés és tervezés szétválasztása.

• A logikai és fizikai tervezés szétválasztása.

A strukturális módszertanok egy gyűjtő fogalom. Jelentőségüket az adja, hogy ezek jelentek meg elsőnek és terjedtek el széles körben. Megjelenésük idején azt gondolták, ezek a módszertanok lesznek a szoftverkrízis megoldói. Sok ma is népszerű technika, pl. adatfolyam modellezés, ezekben a módszertanokban jelent meg. Strukturális módszertanok: • Vízesés modell,

• SSADM,

• V modell.

**Vízesés modell**

A vízesés modell (waterfall model) volt az első módszertan, ami széles körben elterjedt. Ez egy strukturális módszertan, ezek közül is a legismertebb. Nagy megrendelők nagy projektjeihez alakították ki. Mivel a nagy megrendelők általában rugalmatlanok, ezért előnyős, hogy a módszertan kevés döntési pontot definiál. A modell lineáris, azaz az életciklus lépései egymás után, átfedés nélkül következnek. A módszertan ipari termelésből ered, ahol nincs lehetőség a követelmények változtatására, ha már egyszer azokat meghatároztuk. Ezért a vízesés modell eredeti változata nem engedi, hogy visszatérjünk egy már lezárt fázisba. Ezt azért teheti meg, mert csak akkor léphetünk a következő fázisba, ha az előzőt már tökéletesen sikerült lezárni. A modell rengeteg dokumentum elkészültét írja elő. Mivel ezeket el kell fogadtatni a megrendelővel, ezzel véli biztosítani a módszertan, hogy nincs félreértés a megrendelő és a szoftvercég közt. 29 Sajnos a sok dokumentumot ritkán olvassa el tüzetesen a megrendelő. Ezért ez a technika nem csak itt, hanem más módszertanok esetén se alkalmas a félreértések elkerülésére. Ugyanakkor a modell feltételezi, hogy precíz mérnökökkel lesz dolgunk, akik kritikusan olvassák a dokumentumokat. Ez a feltételezés jogos. Tehát a vízesés modell igazi kritikája, hogy nem engedi a követelmények menet közbeni változtatását, ami a mai felgyorsult világunkban nem jogos megszorítás. Tehát a vízesés modell:

• lineáris,

• strukturált,

• jól dokumentált,

• nehézsúlyú,

• nem agilis.

A vízesés modell nem határoz meg technikákat, de általában adat és folyamat központú technikákkal együtt alkalmazták. Mostanában használati eset központú technikákkal is szokták használni. A modell a következő fázisokat írja elő:

• szükségletek felmérése (Requirements Analysis),

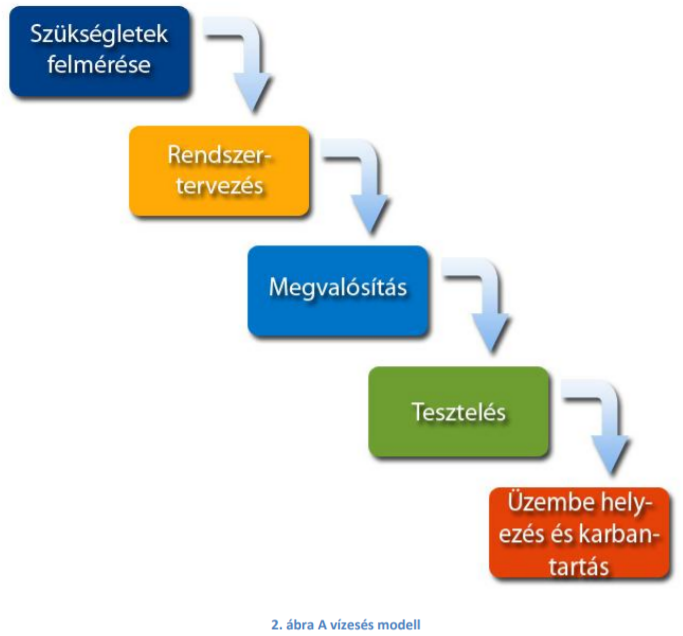
• rendszertervezés (System Design),

• megvalósítás (Implementation),

• tesztelés (Testing),

• üzembe helyezés és a karbantartás (Delivery).

Ezek a folyamatok egymást követik, fel nem cserélhetőek, valamint akkor következik az egyik fázis után a másik, ha az előző teljesen befejeződött, a kitűzött célokat elértük. Ezért is hívják vízesés modellnek.



A vízesés modell előnye az erős kontroll a folyamatok felett. Ütemtervet lehet készíteni, hogy melyik fázis mikorra készül el. A hátránya, hogy nem engedi meg az újragondolást, áttervezést. A fejlesztés végig megy az egyes fázisokon, szigorúan, csak lineáris sorrendben, nincsenek ismétlődő lépések.

A vízesés modell előnye az erős kontroll a folyamatok felett. Ütemtervet lehet készíteni, hogy melyik fázis mikorra készül el. A hátránya, hogy nem engedi meg az újragondolást, áttervezést. A fejlesztés végig megy az egyes fázisokon, szigorúan, csak lineáris sorrendben, nincsenek ismétlődő lépések.

**V-modell**

A V-modell a vízesés modell kiegészítése teszteléssel. Ez azt jelenti, hogy először végre kell hajtani a fejlesztés lépéseit, ezután jönnek a tesztelés lépései. Ha valamelyik teszt hibát talál, akkor vissza kell menni a megfelelő fejlesztési lépésre.

A V-modell hasonlóan a vízesés modellhez nagyon merev, de alkalmazói kevésbé ragaszkodnak ehhez a merevséghez, mint a vízesés modell alkalmazói. Ennek megfelelően jobban elterjedt. Fő jellemzője a teszt központi szerepe.

Tehát a vízesés modell:

• lineáris,

• strukturált,

• jól dokumentált,

• nehézsúlyú,

• teszt központú.

Egy tipikus V-modell változatban először felmérjük az igényeket és elkészítjük a követelményspecifikációt. Ezt üzleti elemzők végzik, akik a megrendelő és a fejlesztők fejével is képesek gondolkozni. A követelményspecifikációban jól meghatározott átvételi kritériumokat fogalmaznak meg, amik lehetnek funkcionális és nemfunkcionális igények is. Ez lesz majd az alapja a felhasználói átvételi tesztnek (User Acceptance Test, UAT). Magát a követelményspecifikációt is tesztelik. A felhasználók tüzetesen átnézik az üzleti elemzők segítségével, hogy ténylegesen minden igényüket lefedi-e a dokumentum. Ez lényeges része a modellnek, mert a folyamatban visszafelé haladni nem lehet, és ha rossz a követelményspecifikáció, akkor nem az igényeknek megfelelő szoftver fog elkészülni. Ezzel szemben például a prototípus modellben lehet pongyola az igényfelmérés, mert az a prototípusok során úgyis pontosításra kerül.

Ezután következik a funkcionális specifikáció elkészítése, amely leírja, hogyan kell majd működnie a szoftvernek. Ez lesz a rendszerteszt alapja. Ha a funkcionális specifikáció azt írja, hogy a „Vásárol gomb megnyomására ki kell írni a kosárban lévő áruk értékét”, akkor a rendszertesztben lesz egy vagy több teszteset, amely ezt teszteli. Például, ha üres a kosár, akkor az árnak nullának kell lennie.

Ezután következik a rendszerterv, amely leírja, hogy az egyes funkciókat hogyan, milyen komponensekkel, osztályokkal, metódusokkal, adatbázissal fogjuk megvalósítani. Ez lesz a komponens teszt egyik alapja. A rendszerterv leírja továbbá, hogy a komponensek hogyan működnek együtt. Ez lesz az integrációs teszt alapja.

Ezután a rendszertervnek megfelelően következik az implementáció. Minden metódushoz egy vagy több unit-tesztet kell készíteni. Ezek alapja nem csak az implementáció, hanem a rendszerterv is. A nagyobb egységeket, osztályokat, al- és főfunkciókat is komponens teszt alá kell vetni az implementáció és a rendszerterv alapján.

Ha ezen sikeresen túl vagyunk, akkor az integrációs teszt következik a rendszerterv alapján. Ha itt problémák merülnek fel, akkor visszamegyünk a V betű másik szárára a rendszertervhez. Megnézzük, hogy a hiba a rendszertervben vagy az implementációban van-e. Ha kell, megváltozatjuk a rendszertervet, majd az implementációt is.

Az integrációs teszt után jön a rendszerteszt a funkcionális specifikáció alapján. Hasonlóan, hiba esetén a V betű másik szárára megyünk, azaz visszalépünk a funkcionális specifikáció elkészítésére. Majd jön az átvételi teszt a követelményspecifikáció alapján. Remélhetőleg itt már nem lesz hiba, mert kezdhetnénk az egészet elölről, ami egyenlő a sikertelen projekttel.

Ha a fejlesztés és tesztelés alatt nem változnak a követelmények, akkor ez egy nagyon jó, kiforrott, támogatott módszertan. Ha valószínű a követelmények változása, akkor inkább iteratív, vagy még inkább agilis módszert válasszunk.

**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.

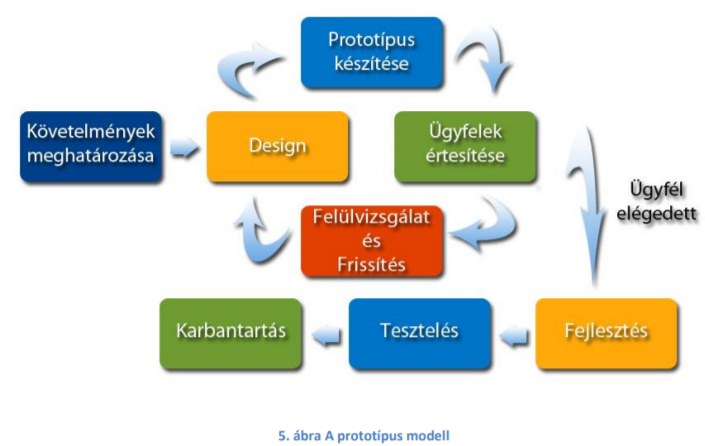
A prototípus alapú megközelítése a fejlesztésnek azon alapszik, hogy a megrendelő üzleti folyamatai, követelményei nem ismerhetők meg teljesen. Már csak azért sem, mert ezek az idővel változnak (lásd az agilis módszertanokat). A követelményeket érdemes finomítani prototípusok segítségével. Ha a felhasználó 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 (lásd napi fordítás) 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: 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 követelményspecifikációt. 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 prototípuskészítést 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. Tehát a prototípus modell:

• általában iteratív,

• általában nem strukturált

• prototípus alapú,

• gyakran rapid, agilis, esetleg extrém,

• általában könnyűsúlyú, és

• követelmény központú.

**Fő változatai:**

**Eldobható prototípus (Throwaway prototyping):** Az elnevezés arra utal, hogy az elkészült modellt úgymond eldobják az elemzés után, vagyis nem lesz része a végső kiadásnak. Arra használjuk ezt a prototípust, hogy a követelményeket minél gyorsabban, minél jobban fel tudjuk mérni, így a felhasználók is láthatják, hogy mire számíthatnak. Miután ez megtörtént, a prototípus eldobható, és a rendszer a prototípussal felderített követelményekre fog épülni. Ennek a változatnak a legnagyobb előnye, hogy a felhasználók gyorsan kipróbálhatják az első modellt, így újragondolhatják az előzetesen megadott követelményeket. Ezzel nagyon sok költséget és munkát meg lehet takarítani, mert rögtön az életciklus elején kiderülhet, hogy valamit máshogyan kellene csinálni. Ugyanakkor a gyors fejlesztés miatt nem szabad csodákat várni az eldobandó példánytól. A másik előnye, hogy a felhasználói felületet is hamar kialakíthatjuk, és azzal együtt tesztelhetik.

Ki kell hangsúlyozni, hogy ennek a változatnak csak akkor van értelme, ha az eldobható prototípus az életciklus elején gyorsan elkészül. A lényeg a gyorsaságon van. Ennek legszebb példája az extrém programozás (lásd később).

A felület kialakításának legismertebb módja:

• Kicsit valósághű eldobható prototípus vagy más néven papír prototípus: Papíron tollal rajzoljuk meg, hogy hogyan képzeljük el mégis a felületet.

• Nagyon valósághű eldobható prototípus: Egy grafikus felhasználói felület szerkesztővel hozzuk létre a felületet, ami teljesen úgy néz ki, mint a végső kiadás, csak a funkciókból még semmi sem működik.

Evolúciós prototípus (Evolutionary prototyping): Az evolúciós prototípuskészítés jóval másabb, mint az előző változat. Itt már a fejlesztés elején egy nagyon robosztus prototípust kell kidolgozni, ami a lelke lesz a rendszernek, és a későbbiekben már csak finomítani kell rajta. Ugyanakkor itt is meg van a lehetősége a fejlesztőnek, hogy változtassa a beépített funkciókat. Az eldobható prototípussal szemben van egy előnye, mégpedig, hogy itt már az elején egy működő rendszert kap a megrendelő elemzésre, még ha az összes követelmény nincs is beépítve a szoftverbe. Még az is előfordulhat, hogy a felhasználó már az egyik prototípust elkezdi használni a gyakorlatban, mert még az is jobb, mint a semmi. Ennél a változatnál a fejlesztőknek nem kell az egész rendszert egyben fejleszteni. Megtehetik, hogy a rendszernek csak bizonyos részeire koncentrálnak. A kockázatok minimalizálása miatt a fejlesztők csak azt a részét fejlesztik a rendszernek, amit már tökéletesen megértettek, aztán a prototípust elküldik a megrendelőnek, aki dolgozik vele, és visszajelzések ad. Ha további követelmények tisztulnak le, akkor halad tovább a fejlesztés ezek megvalósításával.

**Előnyei:**

Minőségnövelés: A prototípus modell képes visszaszorítani a követelményspecifikációban lévő félreértések számát.

Költségcsökkenés: A prototípus modell segít minél hamarabb kideríteni, hogy a megrendelő valójában mit is akar, és így csökkenti a költségeket, mert minél később derül fény egy félreértésre, annál többe kerülnek a változtatások.

Erősíti a felhasználó bevonását a fejlesztésbe: A prototípus modellnél szükség van a felhasználók bevonására a fejlesztésbe. A prototípusok elemzésével már a fejlesztés közben jelzéseket tudnak adnia a fejlesztők felé. Ezáltal kiderülhetnek az esetleges félreértések a fejlesztők és a megrendelő között, ugyanis a végfelhasználók tudják a legjobban, hogy mit is kellene csinálnia szoftvernek. Így végül a termék magasabb minőségi szintet érhet el a fejlesztés végére, és valószínűleg jobban kielégíti a felhasználók igényeit is mindenféle tekintetben.

**Hátrányai:**

Probléma az elemzés miatt: Ha a fejlesztők csak a prototípusra figyelnek, az elterelheti a figyelmüket a rendszer részletes analíziséről. Ezáltal lehet, hogy nem találják meg a legjobb megoldásokat és így a végső termék:

• teljesítménye gyenge lesz,

• nehezen lesz karbantartható,

• nehezen lesz skálázható.

Ezek a veszélyek csökkennek, ha már az első prototípusban egy rugalmas architektúrát alkalmazunk. Ennek ellentmond, hogy az architektúrát a nemfunkcionális követelmények alapján kell kialakítani, amiket lehet, hogy még nem ismerünk az első prototípus elkészülténél. Erre megoldás lehet keretrendszerek alkalmazása, mint pl. a Spring keretrendszer vagy az ASP.NET MVC csomagja.

A prototípus és a befejezett rendszer összekeveredése a felhasználók fejében: A felhasználók egy prototípus tesztelésekor azt hihetik, hogy ennek már közel úgy kéne működnie, mint a végleges rendszernek, és ezért rossz visszajelzéseket adhatnak a fejlesztőknek.

A fejlesztők ragaszkodása a prototípushoz: Előfordulhat, hogy miután a fejlesztők sok energiát fektettek egy prototípus elkészítésébe túlságosan is ragaszkodnak hozzá, és abból akarnak egy végső kiadást készíteni. Még akkor is, ha nem is megfelelő a prototípus architektúrája. Ilyen szempontból jobb az eldobható prototípus alkalmazása az evolúcióssal szemben.

**Spirális fejlesztési modell**

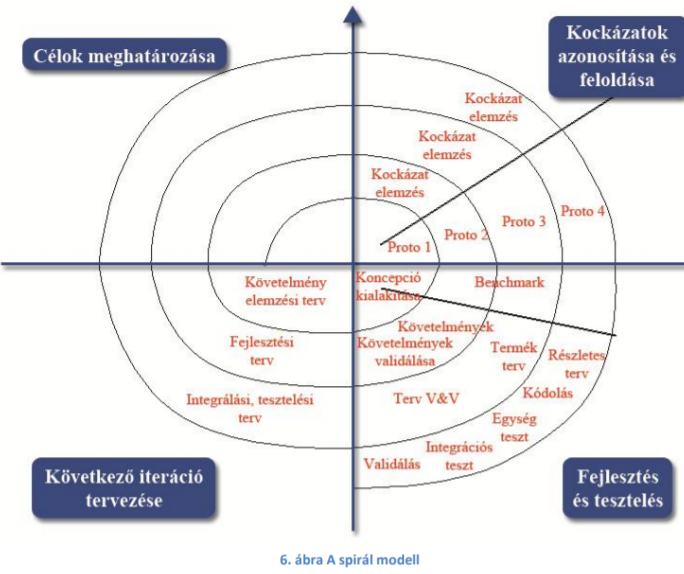
A spirál modellt Barry Boehm 1988. évben publikálta. A spirál modellt a gyakorlatban csak elvétve használják, inkább az elvi jelentősége nagy. A modell a prototípus modell és a vízesés modell egyes tulajdonságait kombinálja. Nagy, bonyolult, drága rendszerekhez ajánlott. A jelentőségét az adja, hogy ez az első modell, amely egyfajta iterációt használ. Habár a prototípus modellben is van ciklus, de vagy csak a követelmények felderítésére szolgál, vagy az egész életciklus megismétlését jelenti. A spirál modell esetén a spirál mindig ugyanazon a négy fázison halad át:

• célok meghatározása,

• kockázatok azonosítása és feloldása,

• fejlesztés és tesztelés

• következő iteráció tervezése.



A spirál modell A fázisok kezdetén meg kell határozni a célokat és a fázis végére egy működőképes prototípust kell előállítani, úgy hogy a prototípusok minden fázis után egyre inkább közelítsenek az elérendő végtermék felé, azaz evolúciós prototípus megközelítést használ. Nagyon fontos része a kockázati elemzés a modell minden fázisában, hiszen a megrendelő a prototípusra azt is mondhatja, hogy ez így nem jó és az elvégzett munkánk kárba veszhet. Az utolsó fázisban a spirál és a vízeséses módszer nagyon hasonló, ugyanis ekkorra már a prototípusok készítésével pontosan tudjuk, hogy hogyan kell kinéznie, működnie a szoftver végleges verziójának és bár az elnevezések nem egyeznek meg, maguk a tevékenységek közel azonosak.

**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 megelőzi az implementációt, itt minden iterációban van tervezés és implementáció 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.

Ezen módszertanok tulajdonságai:

• iteratív (vagy inkrementális),

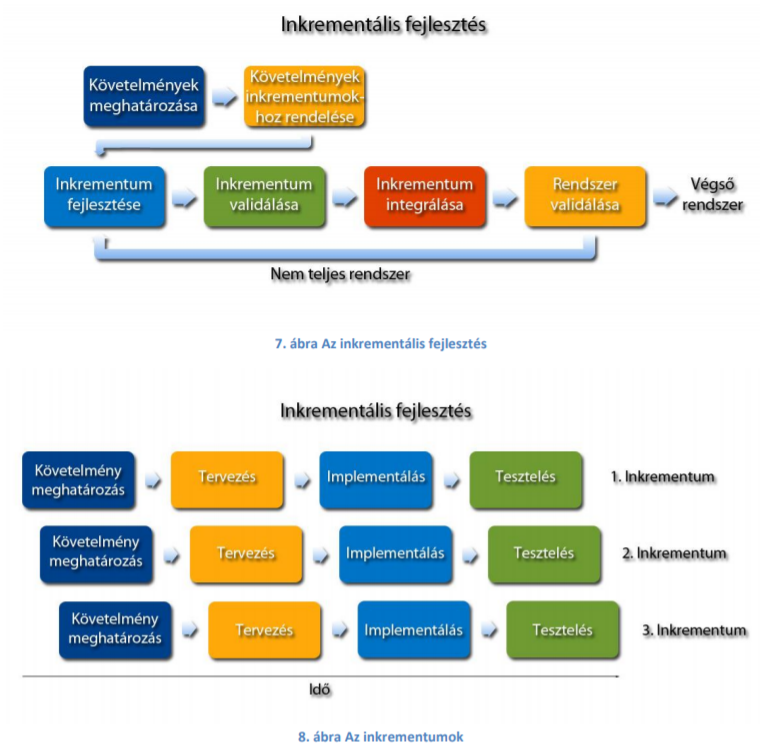
• általában objektum orientált,

• általában prototípus alapú, de van rapid, agilis, sőt extrém változata is,

• általában könnyűsúlyú,

• követelmény központú vagy használati eset központú.

A kiegészítés hozzáadásával növekvő részrendszer jön létre, amelyet tesztelni kell. Az új kódot unitteszttel 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:

• Üzleti folyamatok elemzése

• Követelményelemzés

• Elemzés és tervezés

• Implementáció

• Tesztelés

• 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, ezekre tesztelési osztályokat tervezünk. 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.

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.

Támogató tevékenységek, napi fordítás: Az iterációktól függetlenül úgynevezett támogató folyamatok is zajlanak a szoftvercégen 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 alkamazásfejlesztés 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:

• Korai prototípuskészítés és ismétlődő felhasználói átvételi tesztek.

• A csapat - megrendelő és a csapaton belüli kommunikációban kevésbé formális.

• Szigorú ütemterv, így az újítások mindig csak a termék következő verziójában jelennek meg.

• Követelmények összegyűjtése fókusz csoportok és munkaértekezletek használatával.

• Komponenesek újrahasznosítása.

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

• követelmény összegyűjtő eszközök,

• tervezést segítő eszközök,

• prototípus készítő eszközök,

• 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, és szigorúan tilos a visszalépés, a RAD sokkal rugalmasabb. 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- 45 hét hónap is eltelhet. Vagy ha menetközben 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 rövíd 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 komponesek lehetnek saját fejlesztésüek 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 termékeiket, illetve gyors hibajavítást vállalnak.

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 ugynevezett 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:

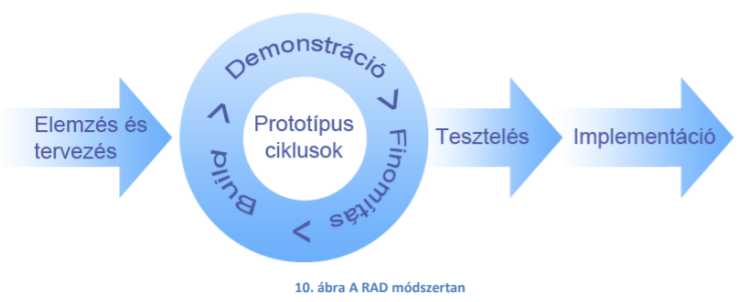
• Üzleti modellezés: Az üzleti funkiók közötti információ áramlást olyan kérdések feltevésével tudjuk felderíteni, mint hogy milyen információk keletkeznek, ezeket ki állítja elő, az üzleti folyamatot milyen információk irányítják, vagy hogy ki irányítja.

• Adat modellezés: Az üzleti modellezéssel összegyűjtöttük a szükséges adatokat, melyekből adat objektumokat hozunk létre. Beazonosítjuk az attribútumokat és a kapcsolatokat az adatok között.

• Folyamat modellezés: Az előzőleg létrehozott adatmodellhez szükséges műveletek (bővítés, törlés, módosítás) meghatározása, úgy hogy lértehozzuk a kellő információáramlást az üzleti funkiók számára.

• Alkalmazás előállítása: A szoftver előállításának megkönnyítése automatikus eszközökkel.

• Tesztelés: Az új programkomponensek tesztelése, a már korábban tesztelt komponenseket már nem szükséges újra vizsgálni. Ez gyorsítja a folyamatot. Ugyanakkor ez természetesen nem vonatkozik az integrációs és a rendszertesztre. Az új komponens és a régi komponensek együttműködését integrációs tesztekkel kell vizsgálni. Valamint a rendszertesztet is meg kell ismételni, természetesen az új komponenssel kapcsolatos területekre helyezve a hangsúlyt.



Tehát a RAD módszertan:

• általában iteratív,

• objektum orientált,

• rapid,

• könnyűsúlyú

• követelmény központú sok megrendelő – csapat kommunikációval.

Hátránya, hogy magasan képzett fejlesztőkre van szükség, emellett fontos a fejlesztők és a végfelhasználók elkötelezettsége a sikeres szoftver iránt. Ha a projekt nehezen modularizálható, 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:

• az egyének és az interaktivitás szemben a folyamatokkal és az eszközökkel,

• a működő szoftver szemben a terjedelmes dokumentációval,

• az együttműködés a megrendelővel szemben a szerződéses tárgyalásokkal,

• az alkalmazkodás a változásokhoz szemben a terv követésével.

Az agilis szoftverfejlesztés alapelvei:

• A legfontosabb a megrendelő kielégítése használható szoftver gyors és folyamatos átadásával.

• Még a követelmények kései változtatása sem okoz problémát.

• A működő szoftver / prototípus átadása rendszeresen, a lehető legrövidebb időn belül.

• Napi együttműködés a megrendelő és a fejlesztők között.

• A projektek motivált egyének köré épülnek, akik megkapják a szükséges eszközöket és támogatást a legjobb munkavégzéshez.

• A leghatékonyabb kommunikáció a szemtől-szembeni megbeszélés.

• Az előrehaladás alapja a működő szoftver.

• Az agilis folyamatok általi fenntartható fejlesztés állandó ütemben.

• Folyamatos figyelem a technikai kitűnőségnek.

• Egyszerűség, a minél nagyobb hatékonyságért.

• Önszervező csapatok készítik a legjobb terveket.

• 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:

• Scrum

• Extrém Programozás (XP)

Ezek a következő közös jellemzőkkel bírnak:

• Kevesebb dokumentáció.

• Növekvő rugalmasság, csökkenő kockázat.

• Könnyebb kommunikáció, javuló együttműködés.

• 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 a 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 programozók 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.

Tehát az agilis módszertan:

• iteratív,

• gyakran objektum orientált,

• prototípus alapú,

• rapid,

• agilis,

• esetleg extrém,

• könnyűsúlyú,

• követelmény és csapat központú.

Az agilis módszertanok nagyon jól működnek, amíg a feladatot egy közepes méretű (5-9 fős) csapat képes megoldani. Nagyobb csoportok esetén nehéz a csapatszellem 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 a rugby nevű csapatjátékból 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 49 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 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, hogy 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.

Tehát a Scrum módszertan:

• iteratív (ahol az iteráció a sprint),

• objektum orientált vagy szerviz orientált,

• prototípus alapú, rapid és agilis,

• csapat központú,

• könnyűsúlyú (lightweight) módszertan.

A módszertan szerepköröket, megbeszéléseket és elkészítendő termékeket ír elő.

**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:

• Scrum mester (Scrum Master)

• Terméktulajdonos (Product Owner)

• Csapat (Team)

Csirkék:

• Üzleti szereplők (Stakeholders)

• 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, tulajdonosok (Stakeholders): 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.

**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 csapattal 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:

• Mit csináltál a tegnapi megbeszélés óta?

• Mit fogsz csinálni a következő megbeszélésig?

• 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 52 ott inkább a személyeket érintő kérdések vannak napirenden, míg itt a csapatmunka továbbfejlesztése az elsődleges.

**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 Backlogban 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 azok 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.