**Universitatea Politehnica Bucuresti**

**Facultatea de Electronica, Telecomunicatii si Tehnologia Informatiei**

**Agile Software Developement**

**Studenți** **:** Taifas Ștefania

Lupe Bogdan

Haidu Marius

**Grupa :** 443 A

**Ianuarie 2014**

**Cuprins**

1. **Aspecte Generale – Taifas Stefania 443A**
   1. **Introducere**
   2. **Agile Manifesto**
   3. **Filozofie**
2. **Metode Agile – Lupe Bogdan 443A**
   1. **Adaptive Software Development**
   2. **Extreme Programming**
   3. **Scrum**
3. **Implementare – Haidu Marius 443A**
   1. **Implementare**
   2. **Avantaje si dezavantaje**
   3. **Concluzii**

**1. Aspecte Generale**

**1. Introducere**

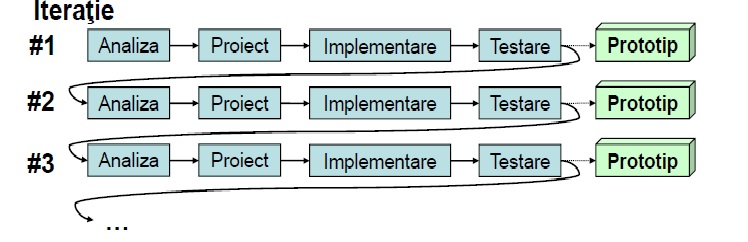
Metodele Agile sunt o reacție la mijloacele tradiționale de dezvoltare software. In vederea punerii în aplicare a metodelor tradiționale , munca începe cu documentația asupra unui set complet de cerințe , urmate de design arhitectural la nivel înalt, dezvoltare și de inspecție. Începând cu mijlocul anilor 1990 , unii practicanți au găsit aceste măsuri inițiale de dezvoltare aproape imposibile . Industria și tehnologia se mișca prea repede, iar clientii au dorit din ce în ce mai în mult ca software-ul sa faca fata nevoilor lor. Ca rezultat , mai mulți consultanți si-au dezvoltat independent metode și practici pentru a răspunde la schimbările inevitabile cu care au fost confruntati . Aceste metode Agile sunt de fapt o colectie de tehnici diferite ( sau practici ), care împărtășesc aceleași valori și principii de bază .Pe baza acestui accesoriu iterativ , o tehnică care a fost introdusă în 1975 .  
De fapt , cele mai multe dintre practicile Agile nu sunt ceva nou. In schimb accentul și munca din spatele metodelor Agile sunt cele care le diferențiază de metodele mai tradiționale . Îmbunătățirea procesului de software-ul este o evoluție în care procesele de noi se bazează pe eșecurile și succesele celor dinaintea lor , astfel încât să înțeleagă cu adevărat mișcarea Agile.

**Ce este Agile Software?**

Agile Software este o filisofie, nu este o metodologie sau proces. Este un mod de a gandi, un ghid organizational care sprijina un mediu in care exista sau pot exista prea multa schimbare.

Ca raspuns la schimbarea continua a mediului, Agile Software incurajeaza interactiunea si colaborarea frecventa cu clientii, livrare constanta a produselor si serviciilor.

Agile Software Development este o familie de metodologii de project management în ingineria software, bazată pe dezvoltarea incrementală și care îmbrățișează și promovează schimbările ce evoluează de-a lungul întregului ciclu de viață al unui proiect. Aceste metodologii se caracterizează prin divizarea problemei în subprobleme mici și planificarea lor pe durate scurte. Se evită planificarea în detaliu pe termen lung, deoarece inerent în dezvoltarea de software apar întârzieri frecvente din cauza schimbărilor și detalierii cerințelor clientului. Scopul principal este ca, la terminarea fiecărui ciclu de dezvoltare (denumit iterație, și a cărui durată este de obicei ordinul câtorva săptămâni) să existe o versiune cât de cât funcțională (deși incompletă) a software-ului dezvoltat (cu număr minim de buguri).



O altă caracteristică importantă este comunicarea frecventă între membrii echipei, care, în multe cazuri, se întâlnesc într-o scurtă ședință zilnică, denumită stand-up sau scrum (pronunțat [/scrʌm/](http://ro.wikipedia.org/wiki/Alfabetul_Fonetic_Interna%C8%9Bional)) în care fiecare prezintă pe scurt progresul său în ultima zi de lucru și problemele cu care s-a confruntat. Acestora li se alătură și un reprezentant al clientului, care trebuie să fie informat de aspectele dezvoltării, pentru a ști ce modificări este realist să ceară și cât de mult ar putea costa ele. Astfel, toată lumea are cunoștințe despre fiecare aspect al dezvoltării aplicației și poate prelua munca altuia sau ajuta pe altcineva.

De-a lungul timpului au apărut diverse metode la baza cărora stau unele dintre aceste valori şi principii. Ele au fost denumite metode agile, iar printre ele se numără:

* Programarea extremă (XP)
* Scrum
* Proces unificat agil (AUP)
* Modelare agilă
* Proces unificat esenţial (EssUP)
* Proces unificat deschis (OpenUP)
* Metoda de dezvoltare dinamică a sistemelor (DSDM)
* Dezvoltare bazată pe caracteristici (FDD)
* Crystal

Dezvoltarea Agile folosește feedback-ul pentru efectuarea într-o manieră colaborativă a unor ajustări periodice ale procesului de producție. Aceasta înseamnă să nu amânăm procesul de testare a aplicației pentru finalul proiectului, s-au să integrăm schimbările din codul aplicației doar la sfârșitul fiecărei luni, sau sa ne oprim din obținerea noilor cerințe imediat ce începem procesul de producție. Din contra, aceste activități se vor efectua pe tot ciclul de viața al proiectului. Cum un proiect software nu poate fi considerat ca şi „finalizat”, atât timp cât mai sunt persoane care îl folosesc, se impune un feedback continuu de la utilizatori precum şi o dezvoltare continua a produsului. Nu e nevoie sa aștepți șase luni ca sa-ți dai seama ca ceva merge prost, care e relativ simplu de reparat, îl repari imediat. Ideea de dezvoltare continuă este preponderenta in toate metodele agile. Aceasta se aplica la ciclul de producție în sine dar şi asimilarea noilor tehnologii, obținerea cerințelor, deployment-ul aplicației precum şi antrenamentul utilizatorilor. Principiul este valabil pentru toate fazele de dezvoltare ale unui proiect deoarece elaborarea unei aplicații software reprezintă o activitate foarte complexă. Orice detaliu substanțial amânat va fi implementat mai târziu prost, sau deloc , sau va evolua în ceva foarte dificil de controlat.

Exista mai multe metode de „dezvoltare agila” , toate acestea fiind expuse de o companie non-profit cunoscuta sub numele de „Agile Alliance”. Pentru orice proiect software exista o serie de factori de risc care pot periclita finalizarea cu succes a proiectului, cum ar fi: factori de experienta ( conducatorului de proiect, a echipei), factori de planificare, factori tehnologici, factori externi (modificarea cerintelor, modificarea interfetelor externe).

* Cele mai multe metode de dezvoltare agila incearca sa minimizeze riscul dezvoltand software-ul in intervale scurte de timp, numite “timeboxes”, constand din 1-4 saptamani. Data de sfarsit a unui “timebox” nu poate fi modificata. Daca echipa depaseste data, iteratia este anulata si replanificata.
* Fiecare iteratie este un proiect software in miniatura si include toate activitatile necesare livrarii mini-incrementului unei noi functionalitati: planificare, analiza cerintelor, proiectare, codificare, testare, documentare.
* Fiecare noua iteratie trebuie sa livreze un nou produs. La sfarsitul fiecarei iteratii echipa re-evalueaza prioritatile proiectului.

Timebox-urile sunt utilizate in metodele de dezvoltare rapida ca o forma de management al riscului in dezvoltarea unui soft.

Metodele “agile” pun in evidenta comunicarea directa intre participantii la elaborarea unei iteratii, in locul documentelor scrise. Acestia lucreaza in aceeasi locatie, intre ei fiind cel putin programatorii si “clientii” lor (cei care definesc produsul). Echipa poate include testori, proiectanti de interactiune, echipe de documentare tehnica si manageri. Principala masura a progresului este considerata software-ul functional. Se produce foarte putina documentatie, motiv pentru care metodele sunt criticate.

**1.1 Problemele metodelor agile**

• Dificil de menţinut interesul clienţilor implicaţi în procesul de dezvoltare

• Membrii echipei pot fi neadecvaţi pentru interesul intens în dezvoltarea proiectului cerut de

metodele agile

• Prioretizarea schimbărilor poate fi dificilă atunci când sunt mai mulţi acţionari implicaţi

• Menţinerea simplităţii necesită muncă suplimentară

• Contractele pot fi o problemă (similar altor metode de dezvoltare iterativă)

### 1.2 Predecesori

În 1974, o lucrare de la EA Edmonds a introdus un proces de adaptare de dezvoltare de software. În același timp și în mod independent aceleași metode au fost dezvoltate și implementate de dezvoltatorii sistemelor de la New York. La inceputul anilor 1970, [Tom Gilb](http://translate.googleusercontent.com/translate_c?depth=1&hl=ro&prev=/search%3Fq%3Dagile%2Bsoftware%2Bdevelopment%26biw%3D1280%26bih%3D641&rurl=translate.google.ro&sl=en&u=http://en.wikipedia.org/wiki/Tom_Gilb&usg=ALkJrhh0K6HPN0qdCXz3QffAczb5bymrlw) a început publicarea conceptelor de Evolutionary Project Management (EVO), care a evoluat în Inginerie competitiva. În timpul mijlocul anilor 1970 Gielan a pledat extensiv de-a lungul Statelor Unite pentru această metodologie, datorita practicilor sale si beneficiilor.

Așa-numitele metode *usoare* de dezvoltare software Agile au evoluat în mijlocul anilor 1990 ca o reacție împotriva metodelor orientate, care au fost caracterizate de criticii lor, ca fiind puternic reglementate, rigide și prea elementare pentru abordări de dezvoltare.

Sustinatorii de metode usoare Agile susțin că acestea sunt o revenire la practicile de dezvoltare care au fost prezente la începutul istoria de dezvoltare de software.

**2. Agile Manifesto**

Dezvoltarea de software agilă este denumirea unui grup de metodologii de dezvoltare software ce sunt bazate pe principii comune. Astfel de metodologii propun un anumit stil al procesului de management al proiectelor ce pune accent pe inspecţie frecventă si adaptare, muncă de echipă (ce trebuie incurajată de liderul acesteia), auto-organizare şi responsabilitate, favorizând astfel dezvoltarea rapidă a unui produs software de mare calitate, precum şi coordonarea procesului de dezvoltare cu cererile clientului şi obiectivele companiei. Cele mai multe metode agile propun dezvoltarea iterată (o iteraţie produce unul sau mai multe pachete complete ce împlinesc o funcţie precisă în cadrul proiectului, iar totalitatea iteraţiilor oferă produsul final), muncă în echipă, colaborare şi adaptabilitate pe întreg parcusul de evoluţie al proiectului In februarie 2001, un grup de programatori care împărțeau aceleași probleme legate de procesul greoi de dezvoltare software s-a reunit în Utah, SUA pentru a discuta tehnicile noi numite procese ușoare. În urma acestor discuții s-a adoptat termenul agile, ce desemna o noua abordare, presupunând revizuirea metodologiilor tradiționale de dezvoltare software prin prioritizarea activităților, concentrarea asupra oamenilor, mediului colaborativ, scopurilor concrete, precum şi asupra elaborării unor aplicații care sunt cu adevărat utile. Toate aceste idei au fost încadrate în Agile Manifesto 1, care sta la baza noțiunii din industria software cunoscuta sub numele de Tehnici Agile de dezvoltare software.

**2.1 Principiile Agile**

* Satisfacția clientului prin livrarea rapidă şi continuă de soluții software utile.
* Sunt lansate frecvent versiuni funcționale ale aplicației (săptămâni mai degrabă decât luni)
* Unitatea de măsura a progresului este dată de către funcționalitatea aplicației.
* Chiar si schimbările târzii ale cerințelor aplicației sunt binevenite.
* Colaborarea strânsă dintre dezvoltatori si clienți, pe bază zilnică.
* Convorbirile fața în față sunt cea mai buna modalitate de comunicare.
* Proiectele sunt construite de către indivizi motivați, care merită încredere.
* O atenție deosebită asupra arhitecturii aplicației și asupra perfecționării tehnicilor de programare.
* Simplitatea
* Echipe dinamice bine organizate.
* Adaptarea periodica la circumstanțele noi.

**Manifestul Agile** cuprinde si urmatoarele valori si concepte:

* Indivizii si interacţiunea mai important decât procesele şi instrumentele
* Software funcţionabil mai important decât o documentaţie foarte amplă
* Colaborarea cu clientul mai important decât negocierea contractului
* Receptivitate la schimbare mai important decât urmărirea unui plan

Indiferent de poziția ocupata in cadrul unui proiect software, fiecare poate aplica aceste practici personal, sau împreună cu toata echipa începând sa producă aplicații stabile ușor de realizat într-un ritm dinamic. Ron Jeffrey’s : Nu iei nici un premiu dacă ești agile, adevăratul beneficiu este alegerea corecta a combinației de metodologii care te vor duce la succes. Trebuie sa ne concentram asupra rezultatelor, in loc sa urmam orbește metodologiile fie ele şi agile.

**3. Filozofie**

Comparativ cu ingineria software tradiționala, dezvoltare Agile se adresează în principal sistemelor complexe și proiectelor cu caracteristici dinamice și non-liniare, în cazul în care estimările exacte, planurile stabile și previziunile sunt de multe ori dificile pentru a fi obținute în stadii incipiente, și modele up-front mari și aranjamente vor provoca, probabil, o mulțime de deșeuri. Aceste argumente de bază și experiențele din industrie au ajutat la modelarea de dezvoltare Agile.

**3.1 Comparatie cu alte metode**

Considerand ca metodele de dezvoltare pot fi plasate pe o scara continua de la cele adaptive la cele predictice, metodele agile se situeaza la extremitatea celor adaptive.

<--Agile--> <--Iterative/Incrementale--> <--Cascada-->

<-------------|------------------------------------|----------------|->

Adaptive Predictive

**Metodele adaptive** sunt focalizate pe adaptarea rapida la schimbari. Nu se au in vedere cu exactitate ce se va intampla in viitor. O echipa adaptiva poate raporta exact ce sarcini vor fi realizate saptamana urmatoare si ce este planificat pentru luna urmatoare.

**Metodele predictive** sunt focalizate pe planificarea in detaliu a activitatilor, in timp. O echipa predictiva poate raporta exact ce este planificat pentru intreagul proces de dezvoltare. Echipa predictiva are dificultati in schimbarea directiei. Planul este optimizat pentru desinatia originala iar schimbarea directiei poate necesita renuntarea la rezultatele curente si replanificarea activitatilor. Numai schimbarile considerate importante sunt luate in considerare.

In comparatie cu metodele incrementale:

**Deosebiri**

- perioadele de timp sunt masurate in saptamani si nu luni

-perioadele de timp sunt stricte (time-box-uri)

-in metodele incrementale procesul este ghidat de analiza si masurare a caracteristicilor produsului. Ele sunt suportul pentru determinarea eficientei procesului si a calitatii produsului

**Asemanari**

- creaza produse livrabil in perioade de timp scurte

In comparatie cu **modelul cascada** dezvoltarea agila are foarte putine in comun cu dezvoltarea cascada. Modelul cascada are si in prezent o utilizare larga.. Modelul cascada este cel mai predictiv, activitatile sunt pre-planificate intr-o secventa. Progresul este masurat prin produse intermediare (specificatii, doc de proiectare, planuri de testare, revizii ale codului, etc). Efortul de integrare si testare poate fi foarte mare (cateva luni – cativa anui!), fiind una dintre cauzele de esec ale unei dezvoltari cascada. Prin dezv agila se produc produse executabile intervale de cateva saptamani sau luni. Se pune accentul pe obtinerea de executabile cat mai repede apoi imbunatatirea lor continua!Extreme Programming, este o metoda de dezvoltare agila care a crescut popularitatea metodelor agile. Extreme Programming (XP) este un model incremental bazat pe iteratii scurte, testare intensiva si simplitate. Este adecvata pentru echipe mici si proiecte caracterizate prin schimbari rapide ale cerintelor.

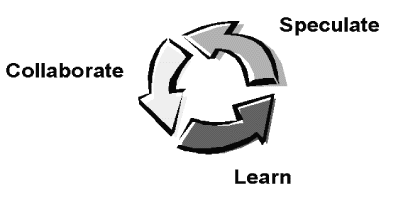
In concluzie, filosofie Agile a ajutat la o colaborare mai buna cu clientii/utilizatorii fiind o parghie intre procesele invatate si cele aplicate.

**2. Metode Agile**

Conceptul de proces agile a câștigat o mare popularitate în comunitatea software (SW) în ultimii ani. Modelele agile promoveză dezvoltarea rapidă. Această proprietate are anumite dezavantaje, cum ar fi documentație de slabă calitate sau de calitate proastă. Dezvoltarea rapidă promovează utilizarea de modele de procese agile în proiecte la scară mică. Modele de procese agile pun accent pe „agilitate” pentru dezvoltarea de software. Agilitate înseamnă a răspunde la schimbări rapid și eficient. Posibilele modificări necesare în proiectele software sunt în cerințe, buget, program, resurse, tehnologie și echipă. Acestea sunt schimbări „reactive”, modificări pe care modele agile pun accent pentru un proiect software de succes. Principiile de aur agile definite în „agile alliance meeting” în 2001 sunt :

1. Prioritatea noastră este satisfacţia clientului prin livrarea rapidă şi continuă de software valoros.
2. Schimbarea cerinţelor este binevenită chiar şi într-o fază avansată a dezvoltării. Procesele agile valorifică schimbarea în avantajul competitiv al clientului.
3. Livrarea de software funcţional se face frecvent, de preferinţă la intervale de timp cât mai mici, de la câteva săptămâni la câteva luni.
4. Oamenii de afaceri şi dezvoltatorii trebuie să colaboreze zilnic pe parcursul proiectului.
5. Construieşte proiecte în jurul oamenilor motivaţi. Oferă-le mediul propice şi suportul necesar şi ai încredere că obiectivele vor fi atinse.
6. Cea mai eficientă metodă de a transmite informaţii înspre şi în interiorul echipei de dezvoltare este comunicarea faţă în faţă.
7. Software funcţional este principala măsură a progresului.
8. Procesele agile promovează dezvoltarea durabilă. Sponsorii, dezvoltatorii şi utilizatorii trebuie să poată menţine un ritm constant pe termen nedefinit.
9. Atenţia continuă pentru excelenţă tehnică şi un design bun îmbunătăţeşte agilitatea.
10. Simplitatea - arta de a maximiza cantitatea de muncă nerealizată - este esenţială.
11. Cele mai bune arhitecturi, cerinţe şi design emerg din echipe care se auto-organizează.
12. La intervale regulate, echipa reflectă la cum să devină mai eficientă, apoi îşi adaptează şi ajustează comportamentul în consecinţă.
13. **Adaptive Software Development (ASD)**

Practicile ASD-ului sunt bazate pe încrederea în adaptare continuă - o filozofie diferită și un alt ciclu de viață adaptate pentru a accepta schimbarea continuă ca o regulă. În ASD, ciclul de static de viață Planificare-Proiectare-Construcție (Plan-Design-Build) se înlocuiește cu o dinamică Speculare-Colaborare- Învățare (Speculate-Collaborate-Learn). Este un ciclu de viață dedicat învățării continue și orientate la schimbări, reevaluare, previziunea unui viitor incert, și o colaborare intensă între dezvoltatori, management, și clienți.



**1.1 Un ciclu de viață orientat pe schimbare**

Dezvoltarea pe ciclul de viață în cascadă , bazat pe o presupunere a unui mediu de afaceri relativ stabil, devine copleșită de schimbări majore. Planificarea este una dintre cele mai dificile concepte pentru ingineri și manageri. Pentru cei educați pe știința de reducționism (reducerea întregului la părțile sale componente) și pe credința aproape religioasă că o planificare atentă, urmată de executare ingineriească riguroasă produce rezultatele dorite ( suntem în control) - ideea că nu există nicio modalitate de a o „face bine de prima dată” rămâne străină. " Planul", cuvânt folosit în cele mai multe organizații, indică un grad destul de ridicat de certitudine cu privire la rezultatul dorit. Scopul implicit și explicit de „respectare a planului” restricționează capacitatea unui manager de a conduce proiectul în direcții inovatoare. „Specularea” ne oferă loc pentru a explora, pentru a ne face clar faptul că suntem nesiguri de rezultat, să ne putem abate de la plan fără teamă. Aceasta nu înseamnă că planificarea este învechită, doar că planificarea nu este consistentă. Aceasta înseamnă că trebuie să avem iterații pentru livrare scurte și să se încurajeze iterația. O echipă care „speculează” nu abandonează planificarea, ea recunoaște incertitudinea. Speculațiile recunosc caracterul incert al problemelor complexe și încurajează explorarea și experimentarea. Putem admite în cele din urmă că nu știm totul. Al doilea concept al ASD este de colaborare. Aplicațiile complexe nu sunt construite, acestea evoluează. Aplicațiile complexe necesită un volum mare de informații să fie colectate, analizate, și aplicate problemei - un volum mult mai mare pentru orice persoană să se poate ocupa de el. Deși există întotdeauna loc pentru îmbunătățire, cei mai mulți dezvoltatori de software sunt destul de competenți în analiză, programare, testare, și abilități similare. Dar mediile turbulente sunt definite parțial de ratele ridicate ale fluxului de informații și diverse cerințe de cunoștințe. Construirea unui site de comerț electronic de exemplu, necesită o mai mare diversitate atât de tehnologie și de cunoștințe de afaceri decât proiectul tipic de acum cinci sau zece ani. În acest mediu cu flux mare de informații, în care o persoană sau un grup mic nu pot „să le știe pe toate”, sunt extrem de importante aptitudinile de colaborare (capacitatea de a lucra împreună pentru a produce rezultate, împărtășirea cunoștiințelor cu ceilalți, sau luarea deciziilor).

Odată ce admitem noi înșine că suntem supuși greșelii, partea de „Learn” („Învățare”) a ciclului de viață devine vitală pentru succes. Trebuie să testăm cunoștințele noastre în mod constant, prin practici cum ar fi retrospective de proiect și focus-grupuri cu clienții. Mai mult decât atât, trebuie să se facă revizuiri după fiecare iterație, mai degrabă decât să se aștepte până la sfârșitul proiectului.

Ciclul de viață ASD are șase caracteristici de bază :

1 . Concetrare pe misiune  
2 . Bazat pe trăsături  
3 . Iterativ  
4 . Respectarea timpului alocat (Time-boxing)  
5 . Atenția la risc  
6 . Toleranța la schimbări

Pentru multe proiecte, cerințele pot fi neclare la început, dar misiunea generală care ghidează echipa este bine articulată. Declarațiile de misiune actionează ca ghizi care să încurajeze explorarea la început, dar îngustarea în centru pe parcursul unui proiect. O misiune prevede limite, mai degrabă decât o destinație fixă. Fără o misiune bună și o practică constantă de rafinament a misiunii, ciclurile de viață iterative pot deveni cicluri viață oscilante - swinging înainte și înapoi, fără niciun progres . Declarațiile de misiune (și a discuțiilor care au condus la aceste declarații) oferă îndrumare și criterii pentru luarea deciziilor importante în proiect. Ciclul de viață ASD se concentrează pe rezultate, nu pe sarcini, iar rezultatele sunt identificate ca și caracteristici ale aplicației. Caracteristicile sunt funcționalitățile pentru client, care urmează să fie dezvoltate în timpul unei iterații. Documentele (de exemplu , un model de date) pot fi definite ca rezultate, dar ele sunt întotdeauna secundare la o caracteristică software care furnizează rezultate directe unui client . (Un document orientat către client, cum ar fi un manual de utilizare este, de asemenea, o caracteristică.) Caracteristicile pot evolua pe mai multe iterații astfel încât clienții să ofere feedback. Practica de time - box , sau practici de stabilire a datelor fixe de livrare pentru iterații și proiecte, au fost abuzate de către mulți dintre cei care folosesc termenele de timp incorect. Termenele de timp folosite pentru a forța personalul să livreze în intervale de timp sau trierea cu privire la calitate sunt o formă de tiranie, ele subminează un mediu de colaborare. A fost nevoie de mai mulți ani de gestionare a proiectelor de ASD înainte să se realizeze că în time – boxing trebuie să fie vorba mai puțin despre timp și mai mult despre concentrare și luarea deciziilor de compromis dificile. Într-un mediu nesigur, în care rata de schimbare este mare, trebuie să existe o funcție periodică de motivare pentru a realiza munca în parametrii propuși.

**1.2 Ciclul de viață ASD**

Specularea : Inițierea și Planificarea

Există cinci etape generale în „speculații”, deși cuvântul „pași” este oarecum un termen impropriu, deoarece fiecare pas poate fi revizuit de mai multe ori în timpul fazei de inițiere și planificare. În primul rând, inițierea proiectului presupune stabilirea misiunii și obiectivelor proiectului, înțelegerea constrângerilor privind organizarea de proiect, identificarea și conturarea cerințelor, executarea estimărilor de dimensiune și domeniu de aplicabilitate, și identificarea riscurilor cheie ale proiectului. Pentru că viteza este, de obicei, un aspect major în utilizarea ASD, multe dintre datelele de inițiere ale proiectului ar trebui să fie colectate în ședința JAD (Joint Application Design) preliminară. Inițierea poate fi finalizată într-un efort concentrat de două până la cinci zile pentru un proiect mic-mijlociu sau de două sau trei săptămâni pentru proiecte mai mari. În timpul ședințelor JAD, cerințele sunt exprimate în suficiente detalii pentru a identifica caracteristici și pentru a stabili un obiect schelet, de date, sau un alt model arhitectural.

Apoi, time -boxingul pentru întregul proiect este stabilit pe baza domeniului de aplicare, setul de caracteristici cerut, estimări, și pe disponibilitatea resurselor care rezultă din inițierea proiectului. Speculațiile nu abandonează estimările, ci doar subliniază faptul că estimările sunt superficiale.

Al treilea pas este de a decide cu privire la numărul de iterații și asignarea unui time-box pentru fiecare. Pentru aplicațiile mici-mijlocii, iterațiile variază , de obicei, de la patru la opt săptămâni. Unele proiecte funcționează cel mai bine cu iterații de două săptămâni, în timp ce altele ar putea necesita mai mult de opt săptămâni ( deși acest lucru este rar ). În general dimensiunea proiectului și gradul de incertitudine sunt doi factori care determină lungimile individuale ale iterațiilor.

După stabilirea numărului de iterații și un program pentru fiecare, membrii echipei dezvoltă o temă sau un obiectiv pentru fiecare dintre iterații. La fel ca stabilirea unui obiectiv de ansamblu al proiectului, fiecare repetare ar trebui să aibă propria temă ( acest lucru este similar cu Scopul Sprint în Scrum ). Fiecare iterație oferă un set de caracteristici funcționale pentru un proces de review al clientului, lucru care face produsul vizibil clientului. În cadrul iterațiilor, „construiește”(„builds”) livrează caracteristici funcționale pentru un proces de integrare de zi cu zi ( sau mai frecvent ), ceea ce face produsul vizibil pentru întreaga echipă de dezvoltare. Testarea este o parte în curs de desfășurare, integrantă a dezvoltării nu o activitate stabilită pe la sfârșit.

Dezvoltatorii și clienții atribuie caracteristici la fiecare iterație. Cel mai important criteriu de atribuire a caracteristicilor este că fiecare iterație trebuie să livreze un set vizibil, tangibil de facilități (features) clientului. În procesul de atribuire, clienții decid cu privire la caracteristicile prioritare, folosind estimări, riscuri, precum și informații de dependență furnizate de echipa de dezvoltare . O foaie de calcul este un instrument eficient pentru planificarea iterațiilor pe baza de facilități. Experiența a arătat că acest tip de abordare, făcută de echipă mai degrabă decât de managerul de proiect oferă o mai bună înțelegere a proiectului decât o abordare tradițional bazată pe sarcină. Planificarea pe bază de caracteristică reflectă unicitatea fiecărui proiect .

**1.3 Colaborarea: Dezvoltarea concurentă a caractersiticilor**

În timp ce echipa tehnică livrează software-ul funcțional, managerii de proiect trebuie să faciliteze activitățile concurente de dezvoltare și colaborare. Pentru proiecte care implică echipe distribuite, variind de partenerii de alianță, și cunoștințe ample, modul în care oamenii interacționează și modul în care acestea gestionează interdependențele sunt probleme vitale. Pentru proiecte mai mici, în care membrii echipei lucrează în proximitatea fizică, colaborarea poate consta în discuții informale pe hol și notații pe tablă. Proiectele mai mari, cu toate acestea, necesită practici suplimentare, instrumente de colaborare și interacțiunea cu managerul de proiect. Colaborarea, un act de creație comun, este stimulată de încredere și respect. Crearea în comun cuprinde echipa de dezvoltare, clienții, consultanți externi, și vânzătorii. Echipele trebuie să colaboreze pe probleme tehnice, cerințele de afaceri, și pe luarea deciziilor rapide.

**„Învățarea” : Review calitate**  
  
Învățarea devine din ce în ce mai dificilă în medii în care se merge pe ideea “fă-l bine de prima dată” -dezvoltarea progresează liniar în stilul ciclului de viață în cascadă. În cazul în care oamenii sunt mereu obligați să facă lucrul bine din prima, ei nu vor experimenta și/sau învăța. În dezvoltarea cascadă, fiecare fază finalizată descurajează revenirea la ea, deoarece nu ar trebui să existe nicio greșeală în acea fază. Experimentarea și învățarea din greșeli cere ca membrii echipei să împărtășească codul parțial și artefactele devreme, în scopul de a găsi greșeli, să învețe din ele, și să reducă cantitatea totală de rework de a găsi mici probleme înainte ca acestea să devină mari. Echipele trebuie să învețe să diferențieze între muncă de calitate inferioară și muncă pe jumătate făcută. Există patru categorii generale de lucruri care trebuie învățate, la sfârșitul fiecărei iterații de dezvoltare:

* Calitatea rezultatului din punctul de vedere al clientului
* Calitatea rezultatului din punct de vedere tehnic
* Modul de funcționare a echipei de livrare și practicile pe care membrii echipei le utilizează
* Statusul proiectului

Obținerea feedback-ului proiectului de la clienți este prima prioritate în proiectele adaptive. Practica recomandată de ASD-uri pentru aceasta este un focus-grup de clienți. Derivată din conceptul de focus grupuri de marketing, aceste ședințe sunt concepute pentru a explora un model de lucru al aplicației și înregistrează cererile de schimbare a clientului. Ele sunt ședințe facilitate, similare cu ședințele JAD , dar mai degrabă decât generarea de cerințe sau definirea planurile de proiect , focus grupurile de client sunt concepute pentru a revizui aplicația în sine . Clienții își pot da cel mai bine cu părerea cu privire la un software funcțional , nu la documente sau diagrame .

Al doilea domeniu de revizuire este cel de calitate tehnică. O practică standard pentru evaluarea calității este folosirea comentariilor tehnice periodice; programarea în perechi realizează un rezultat similar. Deși comentariile de cod sau de programarea în perechi ar trebui să fie continue, alte comentarii, cum ar fi o revizuire generală tehnică a arhitecturii, pot fi efectuate săptămânal sau la sfârșitul unei iterații.

Al treilea domeniu de feedback este pentru echipă pentru a monitoriza propria performanță. Acest lucru ar putea fi numit revizuirea oameniilor și procesului. Mini – retrospectivele de la sfârșitul unei iterații ajută la a determina ceea ce nu merge, la ceea ce echipa trebuie să lucreze mai mult, și ceea ce echipa trebuie să facă mai puțin. Retrospective încurajează echipele să învețe despre ei înșiși și cum funcționează împreună ca un întreg.

A patra categorie de revizuire este stadiul proiectului. Acest lucru duce la efort de replanificare a urmatoarei iterații. Întrebările de bază la revizuire stării sunt: Unde se află proiectul ? Unde este el față de planuri ? În ce stadiu ar trebui să fie ? Determinarea statutului unui proiect este diferită într-o abordare bazată pe caracteristică (facilitate). Într- un ciclu de viață în cascadă, produsele finalizate și livrabile marchează sfârșitul fiecărei etape majore (un document de specificații complete de exemplu, marchează sfârșitul fazei în caietul de sarcini). Într- o abordare bazată pe caracteristici, caracteristicile(facilitățile) complete și software-ul funcțional marchează sfârșitul fiecărei iterații.

Ultima dintre întrebările de status este deosebit de importantă: Unde „ar trebui” să fie proiectul? Deoarece planurile sunt înțelese a fi speculative, măsurile împotriva lor sunt insuficiente pentru a exista progres. Echipa de proiect și clienții trebuie să se întrebe continuu, „Ce am învățat până acum; dacă acest lucru nu schimbă perspectiva noastră spre ce direcție trebuie să mergem?”

**2. Extreme Programming (XP)**

Extreme Programming ( XP ) este o metodologie de dezvoltare de software , care este folosită pentru îmbunătățirea calitatății software-ului și pentru capacitatea de reacție și schimbare la cerințele clienților. Ca un tip de dezvoltare de software agil, se scot frecvent „releasuri”, în ciclurile de dezvoltare scurte, care sunt destinate îmbunătățirii productivității și introducerii punctelor de control (checkpoints) în care pot fi adoptate noile cerințe ale clienților .  
  
Alte elemente ale Extreme Programming includ : programarea în perechi sau examinări aprofundate ale codului, testarea unitară (unit testing) a întregului cod, evitând programarea caracteristicilor până când acestea sunt cerute, o structură de management plat, simplitate și claritate în cod, o mai bună înțelegere a așteptărilor clientului asupra cerințelor pe parcursul timpului, și o mai bună înțelegere a problemei, precum și o comunicare frecventă cu clientul și între programatori. Metodologia își ia numele de la ideea că elementele benefice ale practicilor de inginerie software tradiționale sunt duse la niveluri extreme. De exemplu, evaluările de cod sunt considerate o practică benefică, și duse la extrem, codul poate fi evaluat în mod continuu.   
  
Criticii au remarcat mai multe neajunsuri potențiale, inclusiv probleme cu cerințe instabile, fără compromisuri documentate ale conflictelor de utilizator, și o lipsă de o specificație sau documentație a designului de ansamblu.

XP a generat un interes semnificativ în rândul comunităților de software la sfârșitul anilor 1990 și începutul anilor 2000 , văzând adoptarea sa într-un număr de medii de radical diferite de originile sale.  
  
Disciplina mare cerută de practicile originale de multe ori a mers la extreme, cauzând unele dintre aceste practici, cum ar fi cele crezute prea rigide, să fie depășite sau reduse, sau chiar lăsate de izbeliște. De exemplu, practica de teste de integrare de sfârșitul zilei, pentru un anumit proiect, ar putea fi schimbată într-un program de sfârșit de săptămână, sau pur și simplu redusă la date stabilite de comun acord. Un astfel de program mai rigid ar putea face oamenii să simtă că s-au grabit pentru a genera “cioturi artificiale” de cod doar pentru a trece testarea de sfârșit de zi. Un program mai puțin rigid permite, în schimb, ca unele caracteristici complexe să fie dezvoltate mai complet pe o perioadă de mai multe zile. Cu toate acestea, un anumit nivel de testare de integrare periodică poate detecta grupurile de oameni care lucrează non - compatibil, tangențial cu scopul, înainte să fie prea multă muncă investită în direcții divergente sau greșite .  
  
Între timp, nici celelalte practici de dezvoltare agile nu au rămas fixe, XP fiind încă în evoluție, asimilând mai multă experiență în domeniu prin folosirea altor practici. În cea de a doua editie a „Extreme Programming Explained” ( noiembrie 2004) , la cinci ani de la prima editie, Beck a adăugat mai multe valori și practici, diferențiate între practicile primare și secundare.

**2.1 Concept**

### Obiective

„Xtreme Programare Explained” descrie Extreme Programming ca o metodologie de dezvoltare de software, care organizează oamenii pentru a face software de calitate superioară într-un mod productiv.   
  
XP încearcă să reducă costurile de schimbări în cerințe prin executarea mai multor cicluri scurte de dezvoltare, mai degrabă decât unul lung. În această doctrină, schimbările sunt un aspect natural, inevitabile și de dorit de proiecte software, și ar trebui să fie planificate, în loc de încercarea de a defini la început un set stabil de cerințe. Extreme Programming introduce, de asemenea, o serie de valori de bază, principii și practici desupra cadrului de programare agil.

**2.2 Activități**  
  
XP descrie patru activități de bază, care sunt efectuate în cadrul procesului de dezvoltare software : codare, testare, ascultare, și proiectare.  
  
Avocații XP susțin că cel mai important lucru al procesului de dezvoltare a sistemului este codul de instrucțiuni software care un calculator îl poate interpreta. Fără cod, nu există nici un produs funcțional.  
  
Codarea poate fi de asemenea utilizată pentru a afla cea mai potrivită soluție. Codarea poate ajuta, de asemenea, comunicarea ideilor despre problemele de programare. Un programator care are de-a face cu o problemă de programare complexă, și îi este greu să explice soluția colegilor programatori, ar putea coda într-un mod simplificat și ar putea folosi codul pentru a arăta colegilor la ce s-a gândit. Codul, spun susținătorii acestei poziții, este întotdeauna clar și concis și nu poate fi interpretat în mai multe feluri. Alți programatori pot oferi feedback cu privire la acest cod, prin codarea ideilor lor de asemenea.

**Testarea**

Abordarea Extreme Programming este că, dacă un puțină de testare poate elimina câteva defecte, o mulțime de teste pot elimina mult mai multe defecte.  
  
    Testarea unitară determină dacă o anumită caracteristică funcționează așa cum se dorește. Un programator scrie cât mai multe teste automate, în idee de a „sparge” codul. În cazul în care toate testele au rulat cu succes, atunci codarea este completă. Fiecare bucată de cod care este scrisă este testată înainte de a trece la facilitatea următoare .  
    Testele de acceptare verifică dacă cerințele de înțelese de către programatori satisfac cerințele reale ale clientului .  
  
Testarea de integrare la nivel de sistem a fost încurajată, inițial, ca o activitate de sfârșit de zi, pentru depistarea precoce a interfețelor incompatibile, pentru a le reconecta înainte ca secțiunile separate să devieze foarte mult de la functionalitățile coerente. Cu toate acestea, testarea de integrare la nivel de sistem a fost redusă, la o săptămă, sau mai rar, în funcție de stabilitatea interfețelor generale din sistem.

**Ascultarea**  
  
Programatorii trebuie să asculte la ceea ce clienții au nevoie să facă sistemul, la ceea ce este necesar în logica de afaceri. Ei trebuie să înțeleagă aceste nevoi suficient de bine pentru a da feedback-ul clientului cu privire la aspectele tehnice ale modul în care problema ar putea fi rezolvată, sau nu poate fi rezolvată. Comunicarea între client și programator este abordată în continuare în jocul de planificare. – „the planning game”

**Proiectarea**  
  
Din punctul de vedere al simplității, desigur, s-ar putea spune că dezvoltarea sistemului nu are nevoie de mai mult decât de codare, testare și ascultare. În cazul în care aceste activități sunt efectuate bine, rezultatul ar trebui să fie întotdeauna un sistem care funcționează. În practică , acest lucru nu va funcționa. Se poate parcurge un drum lung fără proiectare, dar la un moment dat proiectul se va bloca. Sistemul devine prea complex și dependențele în cadrul sistemului încetează să mai fie clare. Se poate evita acest lucru prin crearea unei structuri care organizeaza logica în sistem. Un design bun va evita o mulțime de dependențe în cadrul unui sistem, ceea ce înseamnă că schimbarea unei părți a sistemului nu va afecta alte părți ale sistemului.

**2.3 Valorile**  
  
Extreme Programming a avut inițial patru valori în 1999 : comunicarea, simplitatea, feedback-ul, și curajul. O nouă valoare, respectul, a fost adăugat în cea de a doua editie a „Extreme Programming Explained”. Aceste cinci valori sunt descrise mai jos .

**Comunicarea**  
  
Construirea unor sisteme software necesită comunicarea cerințelor de sistem pentru dezvoltatorii sistemului. În metodologiile formale de dezvoltare software, această sarcină este realizată prin documentație. Tehnicile de programare extreme pot fi privite ca metode de a construi rapid și de a separa cunoștințele între membrii unei echipe de dezvoltare. Scopul este de a oferi tuturor dezvoltatorilor o viziune comună a sistemului, care se potrivește cu punctele de vedere deținute de către utilizatorii sistemului. În acest scop, Extreme Programming favorizeaza modelele simple, metafore comune, colaborarea dintre utilizatori și programatori, comunicarea verbală frecventă , și feedback-ul.

**Simplitatea**  
  
Extreme Programming încurajează începutul cu cea mai simplă soluție. Funcționalitățile suplimentare pot fi adăugate mai târziu. Diferența dintre această abordare și metodele de dezvoltare mai conventionale este accentul pe proiectarea și codificarea pentru nevoile de astăzi, în loc de cele de mâine, săptămâna viitoare, sau luna viitoare. Acest lucru este uneori rezumat ca „Tu nu vei avea nevoie de asta”, abordare YAGNI. Susținătorii XP recunosc dezavantajul care acest lucru îl poate avea uneori, adică mai mult efort în ziua de mâine pentru a schimba sistemul. Ipoteza lor este că acest lucru este mai mult decât compensat de avantajul de a nu investi in posibile cerințe viitoare, care s-ar putea schimba inainte de a deveni relevante. Codarea și proiectarea pentru cerințele viitoare incerte implică riscul de a cheltuieli resurse pe ceva care nu este necesar, în timp ce, probabil, se întârzie cu termenul pentru finalizarea caracteristicilor esențiale. Legat de valoarea „comunicare” - simplitatea în design și codare ar trebui să îmbunătățească calitatea comunicării. Un design simplu, cu cod foarte simplu ar putea fi ușor de înțeles de către cei mai mulți programatori din echipă.

**Feedback-ul**  
  
În cadrul Extreme Programming , feedback-ul se referă la diferite dimensiuni ale dezvoltării sistemului :  
  
    Feedback-ul de sistem: scriere teste de unitate, sau rulare de teste de integrare periodice, programatorii primind feedback direct despre starea sistemului dupa implementarea schimbărilor.  
    Feedback-ul de la client : Testele funcționale (teste de acceptare), sunt scrise de către client și testeri. Ei vor primi feedback concret despre starea actuală a sistemului lor. Această revizuire este planificată o dată la fiecare două sau trei săptămâni, astfel încât clientul poate conduce cu ușurință de dezvoltare .  
    Feedback-ul de la echipa : Atunci când clienții vin cu noi cerințe în jocul de planificare, echipa le dă direct o estimare a timpului pe care le va lua pentru a le pune în aplicare.  
  
Feedback-ul este strâns legat de comunicare și simplitate. Defectele în sistem sunt ușor comunicate scriind un test de unitate care dovedește că o anumită bucată de cod se va rupe. Feedback-ul direct de la sistem le spune programatorilor să refacă această parte. Un client este capabil de a testa sistemul periodic în conformitate cu cerințele funcționale, cunoscut sub numele de relatări de la utilizator – user stories. Pentru a cita Kent Beck , " Optimismul este un pericol la locul de muncă în programare. Feedback-ul este tratamentul. "   
  
**Curajul**  
  
Mai multe practici întruchipează curajul. Una este ideea de a proiecta și coda mereu pentru ziua de azi și nu pentru ziua de mâine. Acesta este un efort de a evita împotmolirea în design care necesită un mare efort pentru a pune în aplicare a orice altceva. Curajul permite dezvoltatorilor să se simtă confortabil cu restructurarea codului lor, atunci când este necesar. Aceasta înseamnă revizuirea sistemului existent și modificarea sa, astfel încât schimbările viitoare pot fi puse în aplicare mai ușor. Un alt exemplu de curaj este de a ști când să se arunce departe cod: curajul de a elimina codul sursă, care este învechit, indiferent de cât de mult a fost folosit efort pentru a-l crea. De asemenea, curajul înseamnă persistență : Un programator poate fi blocat pe o problemă complexă pentru o zi întreagă, iar apoi rezolvă problema repede a doua zi, dar numai în cazul în care este persistent.

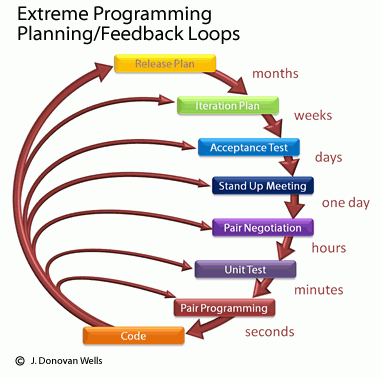
**Respectul**  
  
Valoarea respect include respect pentru alții, ca și respectul de sine. Programatorii nu ar trebui să se angajeze în modificări care strică compilarea, care fac testele unitare existente să pice, sau care întârzie activitatea colegilor lor. Membrii respectă propria lor muncă luptând întotdeauna pentru calitate și căutând cel mai bun design pentru soluția la îndemână prin refactorizare.  
  
Adoptarea celor patru valori anterioare duce la respect câștigat de la alții în echipă. Nimeni din echipă nu ar trebui să se simtă neapreciat sau ignorat. Acest lucru asigură un nivel ridicat de motivație și încurajează loialitatea față de echipă și de scopul proiectului. Această valoare este foarte dependentă de alte valori, și este foarte mult orientată spre oamenii dintr-o echipă.

**Reguli**  
  
Prima versiune de reguli pentru XP a fost publicată în 1999 de către Don Wells pe site-ul XP. 29 de reguli sunt date în categoriile de planificare, gestionare, proiectare, codificare și de testare. Planificarea, gestionarea și proiectarea sunt citate în mod explicit pentru a contracara susținerea că XP nu acceptă aceste activități.  
  
O altă versiune a regulilor XP a fost propusă de către Ken Auer în XP / Agile Univers 2003. El a simțit că XP a fost definită de regulile sale, nu practicile sale ( care sunt supuse la mai multe variații și ambiguitate ). El a definit două categorii : „regulile de angajament” , care dictează mediul în care poate avea loc dezvoltarea de software în mod eficient , și „regulile de joc”,care definesc activitățile și regulile minut cu minut în cadrul regulilor de implicare – Rules of Engagement .

**Principii**  
  
Principiile care stau la baza XP se bazează pe valorile descrise mai sus și sunt destinate să promoveze deciziile într-un proiect de dezvoltare a sistemului. Principiile sunt destinate a fi mai concrete decât valorile și mai ușor de tradus pentru orientare într-o situație practică.

**Feedback**  
  
Extreme Programming consideră feedback-ul drept cel mai utilă în cazul în care se face în mod frecvent și cu promptitudine. Acesta subliniază că întârzierea minimă între o acțiune și feedback este esențială pentru a învăța și de a face modificări. Spre deosebire de metodele tradiționale de dezvoltare ale sistemului, contactele cu clientul au loc mai frecvent în iterații. Clientul are o perspectivă clară despre sistemul care este în curs de dezvoltare. El sau ea poate oferi un feedback și poate orienta dezvoltarea după cum este necesar. Cu feedback frecvent de la client, o decizie de design greșit făcută de către dezvoltator va fi observată și corectată rapid, înainte ca dezvoltatorul să își petreacă mult timp cu punerea sa în aplicare.  
  
Testele de unitate, de asemenea, contribuie la principiul de feedback rapid. Când se scrie cod, rulând testul unitate oferă feedback direct cu privire la modul în care sistemul reacționează la schimbările care au fost făcute. Aceasta include nu nu numai testarea de unitate care testează codul de la dezvoltator, ci rularea în plus a tuturor testelor unitare pentru tot software-ul, folosind un proces automat care poate fi inițiat de către o singură comandă. În acest fel, în cazul în care modificările dezvoltatorului provocă un eșec în altă parte a sistemului, despre care dezvoltatorul știe puțin sau nimic, suite-ul automat de test va dezvălui eșecul imediat, va alerta dezvoltatorul despre incompatibilitatea schimbării sale cu alte părți ale sistemului, și necesitatea de a elimina sau de a modifica schimbarea acestuia. În conformitate cu practicile tradiționale de dezvoltare, lipsa unui suite automat testare unitară completă însemna că o astfel de schimbare de cod, asumată inofensivă de către dezvoltator, ar fi fost lăsată în loc, și va apărea doar în timpul testării de integrare sau, mai rău , numai în producție; iar apoi aflarea schimbării de cod ce a cauzat problema, dintre toate modificările făcute de către toți dezvoltatorii în săptămâni sau chiar luni anterioare de la testarea de integrare, este o sarcină formidabilă, aproape imposibilă.

**Presupunând simplitate**  
  
Acest lucru este tratarea fiecarei probleme ca și în cazul în care soluția a fost „extrem de simplă”. Metodele tradiționale de dezvoltare ale sistemului spun să a planifici pentru viitor și să codezi pentru reutilizare. Extreme Programming respinge aceste idei.  
  
Avocații programării extreme spun că a face schimbări mari dintr-o dată asupra programului nu funcționează. Programarea Extremă aplică schimbările incrementale : de exemplu, un sistem poate avea versiuni mici (release-uri) la fiecare trei săptămâni. Când se fac mulți pași mici, clientul are mai mult control asupra procesului de dezvoltare și a sistemului care este în curs de dezvoltare .



**Îmbrățișând schimbarea**  
  
Principiul de îmbrățișare al schimbării nu se referă la munca împotriva schimbărilor, ci la îmbrățișarea lor. De exemplu, dacă la una dintre reuniunile iterative se pare că cerințele clientului s-au schimbat dramatic, programatorii trebuie să îmbrățișeze acest lucru și să planifice noile cerințe pentru urmatoarea iterație.

Susținătorii XP identifică 12 reguli (sau practici), care trebuie urmate în timpul procesului de dezvoltare:  
  
1.    Povești de utilizator (de planificare) (User stories): poveștile de utilizator pot fi privite ca o versiune diluată a cazurilor de utilizare. Clientul definește caracteristicile dorite de noua aplicație și descrie valoare de afaceri și prioritatea pentru fiecare caracteristică. Acestea trebuie să fie păstrate scurte, cu suficiente detalii pentru a oferi o înțelegere a cererii. Echipa de proiect va folosi poveștile de utilizator pentru estimarea costurilor și managementul proiectului.  
2.    Release-uri mici (blocuri de construcție) (Small Releases) : Folosind XP, se dezvoltă și să livrează aplicația într-o serie de versiuni mici, actualizate frecvent. După cum se adaugă fiecare cerință nouă, se completează sistemul și se re -lansează .  
 3.   Metafora (Scheme de numire standardizate) : de dezvoltare a sistemelor XP cere aderă la un set de standarde pentru produse, cum ar fi nume de variabile, nume de clasă, și de metode. Angajarea în acest sistem de nume ar trebui să permită înțelegerea intuitivă a fiecărui element.  
4.    Proprietate colectivă : Nicio persoană nu deține sau nu este responsabilă pentru segmente de cod individuale. La rândul său, codul este revizuit și actualizat de către toată lumea din echipă, pentru a permite o mai bună colaborare.  
5.    Standard de codare : Toți membrii echipei scriu cod în același mod, folosind aceleași stiluri și formare. Acest lucru permite partajarea rapidă de cod și reduce curba de învățare pentru alți dezvoltatori.  
6.    Design simplu : Cel mai bun design este cel mai ușoar care funcționează. Un design corect pentru un sistem XP este unul care rulează toate testele unitare și funcționale, se potrivește cu valoarea de afaceri, și o face o singură dată pentru fiecare caracteristică.  
7.    Refactorizarea : Comunicarea dintre membrii echipei este esențială. Fiecare membru ar trebui să aibă o înțelegere sincronă a cererii și ar trebui să lucreze continuu pentru a ajusta și a îmbunătăți codul. Acest lucru permite sistemului să fie revizuit în mod constant, fără dublarea codului.  
8.    Testarea : Fiecare element de construcție (small release) trebuie să fie testată temeinic înainte de a se elibera. Se scriu testele întâi și se dezvoltă codul pentru a îndeplini cerințele testului. Acest lucru permite o aplicație curată pe termen lung prin scoaterea problemelor înainte ca acestea să se piardă într- o aplicație mare.  
9.    Programarea pe pereche : Programatorii XP lucreză în perechi. Tot codul este dezvoltat de doi programatori care lucrează împreună la o singură mașină. Așteptarea este ca programarea pe pereche să producă cod de calitate mai mare, la același cost sau la unul mai mic.  
10.    Integrarea continuă : Build-uri de software se execută de mai multe ori pe zi. Acest lucru ține toți dezvoltatorii la curent cu cerererea și cu cele mai recente modificări.  
11.    40 de ore pe saptamană de lucru : Pentru ca practicile XP să fie eficiente, dezvoltatorii trebuie să fie foarte odihiți. Experiența a arătat că dezvoltatorii obosiți sau lipsiți de somn fac mai multe greșeli și sunt mai supuși la extenuare rezultând cod de calitate mai scăzută.  
 12.   Clientul la fața locului : Una dintre cele mai importante concepte ale XP este nevoia de a avea clientul ca o parte integrantă a efortului de dezvoltare. Clientul trebuie să fie disponibil în orice moment pentru a stabili priorități, penru a livra și a stabili cerințe, și pentru a răspunde la întrebări .

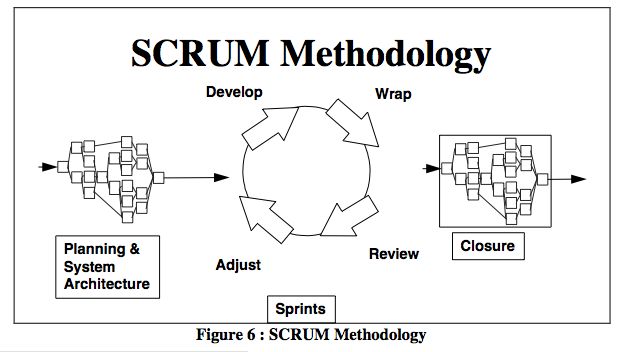
**3. SCRUM**

**Scrum** este o metodă Agile de management a proiectelor IT. În plus SCRUM este un proces iterativ și incremental pentru dezvoltarea software acolo unde cerințele se schimbă rapid. La sfîrșitul fiecărei iterații, echipa de proiect produce un produs software cu un set partial de functionalități, dar care se poate livra la client. Scrum este o abordare care stabilește principii simple de management de proiect. Se folosește de obicei cu practicile de Extreme Programming. Scrum este un proces care imbuntățeste comunicarea și cooperarea în echipă.

Această metodologie a fost prima dată descrisă în 1986 de Hirotaka Takeuchi și Ikujiro Nonaka, ca o metodă flexibilă și rapidă pentru dezvoltare de soft. Principiul acestei metodologii, este împărțirea proiectului în etape de lucru (sprint-uri), în urma carora se finalizează un set de functionalități care intră în exploatare. Această metodologie presupune ca oamenii implicați sa fie buni organizatori ai propriului timp.

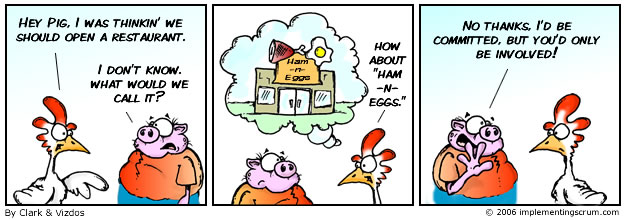
**Organizare**

* **Ședinte dedicate planificării sprinturilor** (în functie de anvergura proiectului, un sprint poate varia între 5 si 30 de zile). Scopul acestor ședinte este prioritizarea taskurilor și stabilirea celor incluse în urmatorul sprint. Din momentul stabilirii, funcționalitățile se “ingheață”
* **Sedințe zilnice** (cu o durată de max. 15 minute). Scopul acestor ședinte este identificarea taskurilor îndeplinite în ziua precedentă, a taskurilor planificate pentru ziua în curs și a problemelor aparute. Această ședință are un caracter mult mai informal



**Roluri**

* **Stakeholderi**
* **Product Owner** – reprezintă interesul stakeholderilor, asigură finantarea proiectului și dezvoltă lista de cerințe. Acesta se concentrează pe ROI și gestionează Product Backlog-ul.
* **Scrum Master –** acest rol poate fi asociat managerului de proiect, care are drept scop împărțirea și prioritizarea sarcinilor.
* **Echipa** – dezvoltă și implementează funcționalitățile.

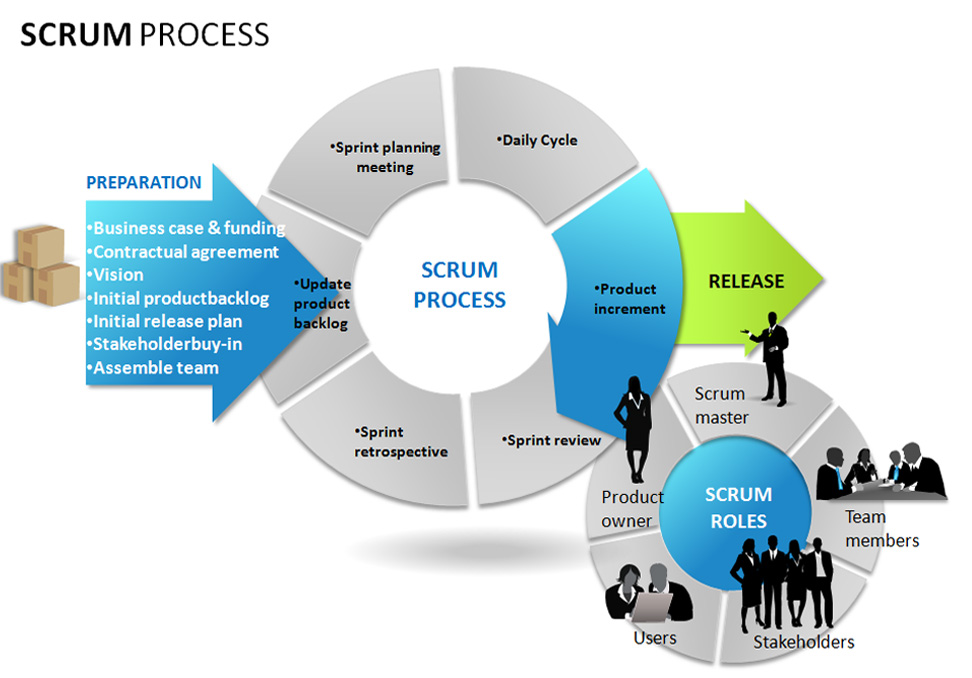
[](http://blog.pointit.ro/wp-content/uploads/2010/08/chickens_and_pigs_cartoon.jpg)

**Porcii** sunt acele persoane dedicate proiectului și implicate în mod direct în sarcinile sale. Porcii, conform cadrului Scrum sunt:

1. **Maestrul Scrum** – El este cel care acționează ca un intermediar între proprietar și echipă. Maestrul protejează echipa, are grijă ca toate sarcinile din perioada fulger să fie îndeplnite la timp și stabilește regurile proiectului.  
2. **Echipa** – Aceasta este implicată în mod direct în procesul de analiză, design, implemnetare și testare și toate sarcinile trebuie duse la capat corect de către membrii săi.  
3. **Proprietarul** – El este cel care stabilește cerințele proiectului și participă la întâlnirile organizate la finalul fiecăror perioade fulger. Din cauza faptului că implicarea proprietarului nu este constantă el poate face parte și din grupul găinilor.

**Găinile** sunt acele persoane care nu sunt implicate în mod direct în procesul Scrum, dar care au rol decizional. Aceste persoane sunt de fapt beneficiarii proiectului. Găinile sunt de fapt managerii companiei client, vendorii și compania client în sine. (fotografie si explicatie preluata de pe [enterprise-concept.com](http://www.enterprise-concept.com/index.php/en/community/blog/item/146-haos-umor-si-rugby))

Modelul Scrum sugerează că proiectele progresează printr-o serie de sprinturi. În conformitate cu o metodologia agilă, sprintarea este încadrată într-un timp (time-boxed) de nu mai mult de o lună, cel mai frecvent de două săptămâni.  
  
Metodologia Scrum pledează pentru o întâlnire de planificare la începutul unui sprint, unde membrii echipei își dau seama la cât de multe elemente care pot angaja, și apoi creează un backlog - o listă a sarcinilor care urmează a fi efectuate în timpul sprintului.  
  
În timpul unei sprint agil Scrum, echipa Scrum preia un mic set de caracteristici și îl trece de la idee la funcționalitate codificată și testată. La sfârșit, aceste caracteristici sunt efectuate, adică codate, testate și integrate în produsul sau sistemul de evoluție .  
  
În fiecare zi de sprint, toți membrii echipei trebuie să participe la o întâlnire Scrum, inclusiv ScrumMaster-ul și proprietarul produsului . Această întâlnire este time-boxed la nu mai mult de 15 de minute. În acest timp , membrii echipei împărtășeasc ceea ce au lucrat în ziua anterioară, stabilesc ce se va lucra în acea zi, și identifică orice obstacole care stau în calea progresului .  
  
Modelul Scrum vede scrum-urile de zi cu zi ca o modalitate de a sincroniza munca membrilor echipei în timp ce discută despre activitatea de sprint.  
  
La sfârșitul unui sprint, echipa efectuează o revizuire a sprint-ului în care echipa arată noua funcționalitate la PO sau la orice stakeholder care dorește să ofere feedback-ul, care ar putea influența sprinturile următoare .  
  
Această buclă de feedback în dezvoltarea de software Scrum poate duce la modificări ale funcționalității proaspat livrate, dar poate duce la fel de probabil la revizuirea sau adăugarea de elemente la backlog-ul de produsului .  
  
O altă activitate în managementul de proiect Scrum este retrospectiva sprint de la sfârșitul fiecărui sprint. Întreaga echipă participă la această reuniune, inclusiv ScrumMaster și PO. Reuniunea este o oportunitate de a reflecta asupra sprint-ului care s-a încheiat, și de a identifica oportunități de îmbunătățire.



**Rolurile principale**

În procesul Scrum, un ScrumMaster diferă de un manager de proiect tradițional, în multe feluri, inclusiv prin faptul că acest rol nu oferă direcții de zi cu zi echipei și nu atribuie sarcini indivizilor.  
  
Un ScrumMaster bun adapostește echipa de distracții exterioare, fapt care permite membrilor echipei să se concentreze maxim în timpul sprint-ului pe obiectivul care le-au ales .  
  
În timp ce ScrumMaster-ul se concentrează pe a ajuta echipa să fie în cea mai bună stare, proprietarul produsului conduce echipa la obiectivul corect. Proprietarul produsului face acest lucru prin crearea unei viziuni convingătoare a produsulu , iar apoi transmite această viziune a echipei prin backlog-ul produsului.  
  
Proprietarul produsului este responsabil pentru prioritizarea backlog-ului în timpul dezvoltării Scrum, se asigură că tot ceea ce mai este învățat despre sistemul va ficonstruit , responsabil pentru utilizatorii săi, echipa și așa mai departe .  
  
Al treilea și ultimul rol în managementul de proiect Scrum este echipa Scrum în sine. Deși indivizii se pot alătura echipei cu diverse titluri de muncă , în Scrum, aceste titluri sunt nesemnificative. Metodologia Scrum prevede că fiecare persoană contribuie în orice fel posibil pentru a finaliza lucrările de fiecare sprint.  
  
Aceasta nu înseamnă că un tester va fi de pus să re - proiecteze sistemul; persoanele vor petrece cea mai mare ( și , uneori, tot ) din timpul lor de lucru în orice disciplină în care au lucrat înainte de a adopta modelul agile Scrum. Dar cu Scrum, individualitățile sunt de așteptat să lucreze dincolo de disciplinele lor preferate ori de câte ori acest lucru ar fi pentru binele echipei .  
  
Un mod de a privi aceste trei roluri în metodologia agilă este o mașină de curse.  
  
Echipa Scrum este mașina în sine, gata pentru a accelera în orice direcție indicată. Proprietarul produsului este șoferul, asigurându-se că mașina merge întotdeauna în direcția cea bună, iar ScrumMaster-ul este mecanicul șef, păstrând mașina bine reglată și performanțele cele mai bune.

**3. Implementare**

1. **Implementare si Exemple**

Datorita filozofiei sale, tehnicile agile se muleaza perfect pe companiile mici. Marea provocare o reprezeinta aplicarea sa la nivelul unei companii mari. Statisticile spun ca aproximativ 2000 de companii ca si IBM folosesc cu succes metoda agila. Datorita scalabilitatii proiectelor, companiile mari nu isi permit luxul de a folosi practici dezordonate. De altfel, implementarea metodelor agile tin foarte mult de specificul companiei, si de timpul alocat etapelor de dezvoltare, cum ar fi documentarea. O alta problema majora pentru pentru multinationale devine dificil sa organizeze intalniri si conferinte asa cum specifica Agile Manifesto.

In anumite situatii, se ajung la compromisuri si astfel se formeaza o metoda hibrida, care pastreaza in mare ideologia agila. Chiar daca dimensiunea proiectelor difera foarte mult de la IMM-uri la corporatii, fapt ce impiedica lucrul in echipa de dimensiuni reduse, Agile porneste de obicei in jurul proiectelor mari, cu o sponsorizare mare din partea clientului solicitant. Probleme exista de partea ambelor tabere, dar solutia companiilor mari a fost scalarea recomandarilor agile dupa propriile nevoie. Astfel, exista echipe cu mai mult de 10 persoane, aflate chiar in locatii diferite.

Conform unui studiu realizate s-a constatat ca doar 26% din intervievati folosesc exclusivi metode agile, 14% utilizeaza asa-zisele hibride. De asemenea, 27% au sustinut ca tehnicile agile intervin doar in proiectele in care este posbila sau este necesara implementarea lor. Astfel, se poate trage o concluzie ca aceste principii nu reprezinta o ideologica generala a unei companii, ci doar un mod de lucru si colaborare in anumite circumstante.

Companiile mare agreeaza conceptele agile de viteza si aliniere cu afacerea, dar exista si cazul aplicatiilor critice care au specificatii unice. Astfel, modificarile din mers nu pot fi efectuate fara un studiu in prealabil deoarece exista un plan care trebuie urmat cu strictete. De altfell, nu toate departamentele trebuie sa 100% agile, existant zone unde nu pot fi implementate concepte de auto-organizare.

Curtis Hibbs, inginer sef la Boeing Defense, Space & Security, unde exista cei mai multi dezvoltatori software de la Boeing declara : "Unele programe sunt destul de mari și scalarea intră în joc în aceste cazuri. Dar avem de a face mai ales cu punerea în aplicare de echipe Agile individuale sau doar scalarea la o mână de echipe." In cazul IBM, 70-80 % din dezvoltare este agila, in special in randul proiectelor mici sau la cele nou incepute. Pentru proiectele vechi, unde exista deja o baza mare de clienti inca se foloseste dezvoltarea iterativa. Motivul principal pentru aceasta problema este numarul foarte mare de utilizatori si implicit perioada de validare a produsului care intarzie ciclul de dezvoltare.

Nu este un set de cele mai bune practici agile care se potriveste tuturor echipelor de dezvoltare. In practica s-au definit trei cazuri de utilizare care dicteaza trei metode diferite priorități și metode. Echipele de dezvoltare a cazut in trei categorii:

* "Major Release Developers" lucreaza pe client-server și produse instalate la nivel local, cu release-uri la fiecare trei sau până la șase luni
* "Cloud Application Developers" construiesc aplicații web-based, care sunt actualizate frecvent, chiar si de mai multe ori pe zi.
* "Multi-Project Developers" lucreaza la mai multe proiecte concurente pentru mai multi clienții

Exista unui numar mare de locatii, uneori chiar si pe continente diferite a reprezentat a pus la indoiala utilizarea regulilor agile. In urma aceluias sondaj mentionat anterior s-a constatat ca marile companii din industria IT au o raspandire foarte mare: Numărul de companii cu personal de dezvoltare în:

* 7 + locații: 27%
* 3-5 locuri: 55%
* 1 Locul de amplasare: 18%

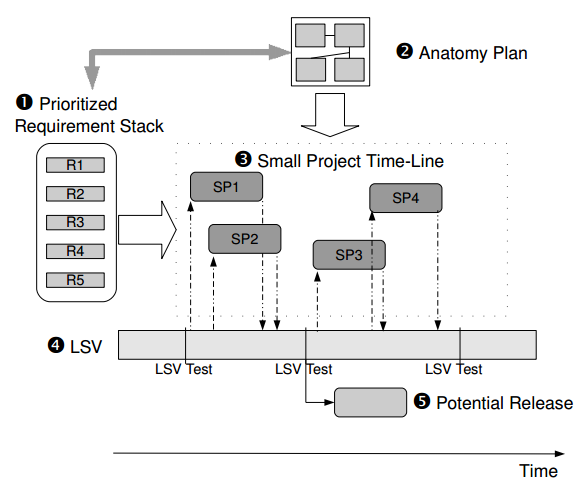
Și aproape toate aceste companii, de asemenea, au angajați care lucrează la domiciliu, persoane fizice, și partenerii de afaceri virtuale de la mai multe locații. Aici principiul interactiunii in detrimentul tehnologiei nu poate fi aplicat cu succes, iar folosirea post-it-urilor nu poate fi realizata in varianta reala. De asemenea, numărul companiilor care folosesc SCRUM este:

* 55%: tehnici de "scrumish" exclusiv Scrum sau
* Scrum și Kanban sau tehnicile Lean: 36%
* Exclusiv Kanban sau tehnicile Lean: 9%

Companiile încorporează conceptele Lean în procesele lor de Scrum. Aceasta lasa loc în planul de sprint de sarcini noi in ultimul minut. Aceste practici reduc o parte din inflexibilitatea pur Scrum, păstrând în același timp avantaje, cum ar fi ce comunicate definit la intervale regulate.

Poate că cea mai mare provocare în extinderea proiectelor agile este coordonarea echipelor și gestionarea dependențelor dintre echipe. Ca proiectele să crească, grupurile de dezvoltare recurg la runde din ce în ce mai complicate și consumatoare de timp, de întâlniri și apeluri. Acest lucru este în mod clar un subiect care necesita noi solutii si idei de inspiratie noua.

Modelul agil utilizat de companii este prezentată în figura . Procesul s-a bazat pe un set de practici specifice companiei, care au fost introduse.



1. Product Backlog: Setul de cerințe pentru proiecte a fost determinată de prioritățile cerințelor. Cerințele cu cele mai mari priorități au fost selectate pentru a fi puse în aplicare. Un alt criteriu de selecție a cerințelor a fost că acestea se potrivesc bine împreună și, prin urmare, ar putea fi puse în aplicare într-o singură proiect coerent.
2. Planul Anatomy : Planul de anatomie a fost creat pe baza dependențelor între părțile sistemului în curs de implementare. Dependențele sunt o arhitectură de sistem si probleme tehnice. Planul de anatomie a dus la o serie de linii de bază numite cele mai recente versiuni de sistem ( LSV ), care sunt necesare pentru a fi dezvoltate . De asemenea, a determinat conținutul fiecărui LSV și momentul în care o LSV ar fi trebuit sa fie finalizata . Planul de anatomie captureaza dependențele între caracteristici ( de exemplu, o caracteristică ce trebuit să fie gata înainte de alta ce trebuie implementata) și dependențele tehnice . Dependențele tehnice sunt critice în domeniul telecomunicațiilor cand platformele și protocoalele de comunicare sunt in permanenta schimbare. De exemplu , dacă un versiune a software-ului a fugit pe o versiune de protocol nu a putut fi integrat cu noua versiune de protocol . Prin urmare , în afară de prioritizarea în produsul restante planul de anatomie furnizat date importante cu privire la ordinea în care proiecte au fost executați , iar când incremente pot fi integrate și testate .
3. Echipe mici și Time-line: Steurile de cerințele au fost implementate de către echipe mici în proiecte de scurtă durată de aproximativ trei luni. Durata proiectului este determinat numărul de cerințe selectate pentru o a fi incluse in proiect. Fiecare proiect a inclus toate fazele de dezvoltare, de la pre-studiu pentru testare. După cum s-a subliniat în figură, atunci când este planificata ordinea în care proiectele sunt executate sunt luate in considerare prioritatile, precum și dependențele tehnice. Mai mult decât atât, figura arată că o interacțiune între cerințe și arhitectură a avut loc.
4. Utilizarea LSV : În cazul în care un proiect a fost integrat cu ultima bază a sistemului , a fost creat un nou sistem de referință ( denumit în continuare LSV ) . Prin urmare , numai un sistem de referință exista la un moment dat în timp, ajutand la reducerea efortului de întreținere a produsului. LSV poate fi, de asemenea, considerată ca un container în care rezultatele proiectelor ( inclusiv software și documentație ) sunt puse împreună . Când rezultatele proiectelor au fost integrate LSV este supus unui amplu proces de testare. Comparând activitatea desfășurată la nivel de echipă cu activitatea desfășurată în LSV poate spune că la nivel de proiect , scopul a fost de a se concentreze pe dezvoltarea seturilor de cerințele în timp ce LSV era axat pe sistemul global în care au fost integrate în rezultatele proiectelor. Odata cu finalizarea LSV, sistemul a fost gata de lansare .
5. Client Release: Daca fiecare versiune ar fi lansat pe piață, ar fi existat mult prea multe versiuni destinate utilizatorilor, care au nevoie de suport in timpul folosirii. Pentru a evita acest lucru, nu orice LSV este lansat, dar trebuie să fie de o calitate suficientă pentru a fi un posibil release.
6. **Avantaje si Dezavantaje**

**Avantaje**

* Satisfacerea clientilor prin livrare rapida, continua de produse software. Clienti sunt incurajati sa semnaleze probleme si sa ofere noi idei ce urmeaza a fi implementate de echipele de dezvoltare. Acest lucru devine un avantaj si mai important in cazul proiectelor by-request, atunci cand clientul care sponsorizeaza doreste sa vada un progresul rapid al cerintelor sale.
* Oamenii si interactiunile, inaintea proceselor si instrumentelor. In acest mod, dezvoltatori, clientii si testeri interactioneaza constant, comunicarea fiind esentiala in gasirea de solutii optime la probleme.
* Este adaptiv. Cum a fost mentionat in subcapitolul anterior, metodele agile pot fi utilizate atat de companii mici, cat si de firme mari, iar in ambele cazuri rezultatele au fost pozitive. Un alt lucru bun al acestui aspect tine de dezvoltarea produsului, astfel schimbarile aparute din cauza unor greseli de proiectare, sau mai mult, solutii noi, optime in detrimentul planului pot aduce schimbari in cursul dezvoltarii fara a creea mari probleme.
* Pune in legatura directa dezvoltatorii si oamenii de afaceri. Pe langa beneficiile directe aduse in dezvoltarea produsului, aceasta cooperare aduce avantaje si pe termen lung, fiind un schimb important de experienta. Astfel programatorii se acomodeaza cu cerintele si modul de lucru din mediul de afaceri, in schimb oamenii de afaceri primesc o educatie tehnica din care nu au decat de castigat pe termen lung.

**Dezavantaje**

* In cauzl proiectelor da mari dimensiuni este aproape imposibil sa definesti efortul necesar pentru task-urile din initierea proiectului, faza in care sunt necasare foarte multe aspecte de implementate si nu exista o forma clara prin care sa se poate defineasca timpul necesar executiei unui anumit task.
* Nu se pune accentul pe proiectare si documentare. Acest lucru poate fi un impediment pentru proiectele de scara mare, dar si pentru cele dedicate unui anumit domeniu de activitate, necunoscut in precedent echipei de dezvoltare. Astfel se pot pierde multe aspecte esentiale, iar implementarea poate avea de suferit. Adaptivitatea si stransa legatura persoanele interesate de produs pot veni in compensarea acestui punct slab.
* Produsul poate iesi usor din parametrii initial daca reprezentantul clientilor nu specifica foarte clar care trebuie sa fie forma finala a produsului, sau daca este slab pregatita si nu cunoaste exact specificatiile pe care le asteapta de la produsul software comandat.
* Doar programatorii seniori sunt capabili sa ia decizii majore in timpul procesului de dezvoltare. Acesta reprezinta si unul dintre motivele limitarii numarului de persoane intr-o echipa, cum majoritatea companiilor prefera sa aiba angajati din toate nivelele de experienta, este greu ca un junior sa poata avea acelasi randament ca un senior si aceeasi luciditate in momentele in care trebuie luata o decizie majora.

1. **Concluzii**

Nu exista un model perfect pentru a creste productivitatea si calitatea. Ideologia Agile este una dintre solutiile optime, care poate aduce schimbari majore in cadrul unei intreprinderi.

Este recomandat sa fie aplicat in urmatoarele cazuri :

* Atunci când sunt necesare noi modificări pentru a fi puse în aplicare. Este foarte importanta libertatea oferita de a schimba. Modificările noi pot fi implementate la costuri foarte mici, din cauza frecvenței mare cu care versiunile sunt lansate.
* Pentru a putea modifica o facilitate, dezvoltatori sunt fortati sa renunta doar la munca desfasurata in ultimele zile, maxim saptamani pentru a putea reveni la un punct unde sa poate implementa noua solutie.
* Spre deosebire de modelul waterfall, în model de Agile este nevoie de o planificare foarte limitata pentru a începe cu proiectul. Agile presupune că nevoile utilizatorilor sunt în continuă schimbare, într-o afacere dinamică din lumea IT. Modificările pot fi discutate și facilitatile pot fi imbunatatite sau eliminate în funcție de feedback-ul primit de la utilizatori. Acest lucru ofera clientului sistemul finit de care are nevoie.
* Atât dezvoltatorii și actionarii, descopera faptul ca au mai multă libertate de timp și opțiuni decât în cazul în care software-ul a fost dezvoltat într-un mod secvențial mai rigid. Opțiunile le oferă posibilitatea de a lăsa deciziile importante până sunt sunt suficiente date sau chiar programe întregi de hosting, ceea ce înseamnă că proiectul poate continua să avanseze, fără teama de a ajunge la un impas brusc.

Bibliografie

Capitolul 1

* 1. <http://en.wikipedia.org/wiki/Agile_software_development#Philosophy>
  2. Scrum in Action Agile Software Project Management and Development - Pham - Course Tech (2012).pdf
  3. <http://agilemanifesto.org/>
  4. <http://www.it-smc.com/Articles/Agile_Philosophy.pdf>
  5. <http://www.slideshare.net/psergiu/metodologii-agile-pentru-proiecte-it>

Capitolul 2

1. [www.wikipedia.com](http://www.wikipedia.com)
2. [www.extremeprogramming.org/](http://www.extremeprogramming.org/)
3. [www.scrummethodology.com/](http://www.scrummethodology.com/)
4. [www.agilemanifesto.org](http://www.agilemanifesto.org) ‎ ‎
5. [www.agilealliance.org](http://www.agilealliance.org)
6. Succeeding with Agile: Software Development Using Scrum (Addison-Wesley Signature)
7. [enterprise-concept.com](http://www.enterprise-concept.com/index.php/en/community/blog/item/146-haos-umor-si-rugby)
8. Beck, Kent - Extreme Programming Explained: Embrace Change, 2nd Edition (The XP Series)
9. [Beck, Kent](http://en.wikipedia.org/wiki/Kent_Beck) - (2001). ["Manifesto for Agile Software Development"](http://agilemanifesto.org/). Agile Alliance
10. Peter Lappo; Henry C.T. Andrew. ["Assessing Agility"](http://www.smr.co.uk/presentations/measure.pdf)

Capitolul 3

1. Implementing lean and AgIle Software development in industry, Kai Peterseon
2. Agile/Lean Product Development and Delivery – Mastering the Art of Change
3. [www.agilemodeling.com](http://www.agilemodeling.com)
4. [www.wikipedia.com](http://www.wikipedia.com)
5. ww.computerweekly.com/
6. Succeeding with Agile: Software Development Using Scrum (Addison-Wesley Signature)