METODE *AGILE* DE MANAGEMENT AL PROIECTELOR

lect. dr. Dan Mircea SUCIU

Cluj Napoca

2018

Metodologii Agile de Management al Proiectelor

Dan Mircea Suciu

This book is for sale at http://leanpub.com/Metode_Agile

This version was published on 2018-08-30

ISBN 978-973-0-27742-5

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Cuprins

Introducere

Cartea de față își propune să sintetizeze în paginile sale caracteristicile celor mai populare metodologii și practici utilizate în managementul proiectelor într-o manieră agilă. Primele trei capitole tratează acest subiect în ansamblu său, explicând necesitatea acestor metodologii în contextul actual, domeniile în care ele pot fi utilizate cu succes și evoluția lor de-a lungul timpului.

Următoarele capitole se concentrează pe descrierea unor metodologii și practici particulare, începând cu *Scrum* și *Extreme Programming* și continuând cu abordările *Lean* (insistând pe *Kanban*) și cele derivate din metodologiile de management predictiv cum sunt *Feature Driven Development* și *Agile Unified Process*. De asemenea, sunt amintite și alte două metodologii ce au fost utilizate pe o scară restrânsă în managementul proiectelor (*Crystal* și *Dynamic Systems Development Method*) dar care au avut o mare importanță în întreaga istorie a abordărilor Agile, în special în facilitarea tranziției de la predictiv la agil.

Capitolele finale discută capacitatea echipelor de proiect de a adopta metodologiile Agile și sunt analizate mai multe curente ce vor influența abordările Agile în viitor.

Înainte de toate, această carte se adresează celor care doresc să se familiarizeze cu domeniul managementului Agile al proiectelor. Prin urmare este utilă celor care doresc să înțeleagă care sunt avantajele și riscurile utilizării practicilor Agile, care sunt tipurile de proiecte în care acestea se recomandă să fie utilizate și ce alternative sunt disponibile.

În același timp, cartea este utilă și celor care au o experiență mai vastă în implementarea unor metodologii particulare în echipele lor de proiect însă care se confruntă periodic cu diverse provocări în ceea ce privește organizarea echipei, implicarea clientului sau respectarea anumitor constrângeri ce influențează hotărâtor succesul proiectului.

Cartea de față nu este însă recomandată ca suport principal în implementarea efectivă a unei metodologii particulare într-o echipă de proiect. Acest lucru ar fi presupus adăugarea unor detalii ce ar fi crescut complexitatea conținutului, lucru care ne-ar fi abătut de la scopul declarat inițial, și anume sinteza.

Nu în ultimul rând, merită adăugat aici faptul că această carte acoperă o lipsă cu care se confruntă astăzi literatura românească de specialitate. Subiectul metodologiilor de management Agile se regăsește în foarte puține cărți apărute în limba română (ca variantă originală sau tradusă), iar aria de răspândire a acestora este redusă. Este adevărat, majoritatea celor interesați de acest subiect își desfășoară activitatea în domenii în care, în general, limba engleză este foarte bine cunoscută, iar oferta de lucrări pe subiectul Agile este în acest caz foarte generoasă. Acest lucru nu justifică însă absența tratării subiectului aproape complet în limba română. Acesta este și principalul motiv pentru care a fost destul de dificil de găsit, în anumite momente, variantele românești ale unor termeni care, în echipele de proiect curente sunt utilizați exclusiv în limba engleză. De aceea se regăsesc în numeroase locuri pe parcursuri cărții, alături de termenii în limba română și corespondentul acestora în engleza, cum ar fi *backlog*, *user story*, *cycle time, release* sau chiar *best practice*.

Am construit această carte avându-i constant în minte pe studenții mei. În ultimii 3 ani am predat cursul de *Metodologii Agile de Dezvoltare a Softului* la patru secții de master al Facultății de Matematică și Informatică a Universității Babeș-Bolyai din Cluj Napoca și cursul de *Metode Alternative de Management al Proiectului* la grupa de master în Management a Școlii Nationale de Studii Politice și Administrative din București. O bună parte din capitolele acestei cărți respectă structura implementată în aceste două cursuri. De asemenea, foarte multe dintre comentariile și observațiile primite de la studenți și-au găsit locul în diverse forme în această carte, motiv pentru care le mulțumesc din suflet.

Patru dintre capitole (dispuse de început și final își au originile în tot atâtea articole publicate în revista lunară de IT *Today Software Magazin*, ele fiind adaptate și încadrate în contextul cărții. Îi mulțumesc fondatorului revistei, Ovidiu Mățan, pentru perseverența cu care m-a încurajat să le scriu și apoi să le prezint în diverse evenimente de lansare ale revistei în comunitatea IT clujeană și nu numai.

Această carte nu ar fi existat fără discuțiile intense și provocatoare pe tematici Agile și conexe avute cu Simona Bonghez și Bogdan Mureșan de-a lungul timpului și cărora le mulțumesc că mi-au relevat perspective cu totul aparte asupra acestor tematici. Cele mai interesante idei pe care le-am dezbătut împreună se află răsfirate printre rândurile acestei cărți.

Un gând final merge către familia fără de care nu aș fi fost capabil să învăț, să mă dezvăț și să re-învăț atât de multe lucruri. Mulțumesc!

1. Managementul cunoștințelor. Particularitățile proiectelor *software*

Metodologiile Agile de gestionare a proiectelor sunt utilizate într-o categorie aparte de proiecte, și anume în cele în care majoritatea activităților de proiect se bazează pe prelucrarea cunoștințelor membrilor echipei de proiect, a organizației din care fac parte aceștia dar și a celorlalte părți implicate în proiect (*stakeholders*).

Încă din anul 1950 Peter Drucker a introdus conceptul de "*knowledge workers*" pentru caracterizarea muncitorilor capabili să utilizeze cunoștințele unei organizații la realizarea unor produse intangibile [1]. Astăzi o categorie însemnată de proiecte se bazează pe resursa critică *cunoștința*. Alternativele de gestionare a acestor proiecte sunt orientate spre găsirea unor principii, metode și tehnici de analiză, planificare, organizare și transfer a cunoștințelor, toate acestea regăsindu-se astăzi sub denumirea generică de management al cunoștințelor.

Cea mai relevantă categorie de proiecte ce se bazează aproape exclusiv pe echipe de *knowledge workers* este cea a proiectelor software. Secțiunile următoare prezintă o analiză amănunțită a acestei categorii de proiecte, analiză în urma căreia se pot identifica o serie de nevoi pe care proiectele software în particular și cele orientate pe manipularea cunoștințelor în general, le au în ceea ce privește managementul.

Analiza succesului proiectelor software

Unul dintre primele lucruri pe care le aflăm atunci când citim o lucrare despre gestiunea proiectelor din domeniul IT (*Information Technology*) este acela că proiectele informatice se deosebesc foarte mult de toate celelalte

proiecte așa-zis tradiționale. Chiar și atunci când subiectul principal al lucrării este legat de managementul proiectelor în general, se fac frecvent referiri directe cu privire la ineficiența anumitor metode atunci când avem de-a face cu proiecte *software*. Din păcate sunt puține cazurile în se analizează sau explică în ce anume constau acele deosebiri și se propun mai degrabă soluții practice, mai mult sau mai puțin eficiente.

Unul dintre cele mai intens citate rapoarte statistice cu privire la succesul proiectelor informatice este *Standish Group Chaos Report* (disponibil la *www.standishgroup.com),* care an de an publică rezultate statistice ce evidențiază ponderea proiectelor finalizate cu succes, a celor eșuate precum și a celor care au întâmpinat dificultăți majore până la finalizare. Criteriile care stau la baza realizării acestui raport sunt se referă la gradul de satisfacere a componentelor triplei constrângeri a proiectelor - scop, timp și bani - sau mai precis la:

- respectarea și implementarea cerințelor specificate;
- încadrarea în bugetul estimat inițial;
- respectarea termenelor de livrare agreate.

Proiectele de succes sunt considerate acelea care respectă toate aceste criterii, cele eșuate sunt cele care s-au stopat fără a mai fi finalizate din cauza nerespectării unuia sau mai multor criterii, iar cele finalizate reprezentă proiectele care au fost până la urmă terminate, rezultatul proiectelor fiind implementat cu condiția reevaluării și ajustării în mod semnificativ a unora dintre criterii.

Tabelul 1.1 prezintă câteva dintre rezultatele raportului de-a lungul timpului, începând cu rezultatul "șocant" din 1994, ce releva o rată de succes extrem de scăzută, și până în 2015. Și această statistică vine să confirme existența unei diferențe evidente între proiectele informatice și celelalte proiecte, un procent de eșec de 20-30% fiind de neconceput în oricare dintre celelalte domenii, de la cel al construcțiilor de clădiri sau mașini până la servicii medicale.

Desigur că se poate discuta mult pe marginea acestor statistici și a relevanței lor. Pentru că "proiect informatic" este un termen care acoperă un spectru destul de larg și eterogen de proiecte, e neclar ce fel de proiecte au fost luate în considerare. Între anii 1994 și 2000 de exemplu, s-au studiat în jur de 30000 de proiecte informatice doar din Statele Unite ale Americii. Pentru 2004 raportul precizează că au fost luate în considerare peste 50000

de proiecte din întreaga lume (58% SUA, 27% Europa, 15% rest), cei de la *Standish Group International* asigurându-ne că s-a respectat un echilibru cantitativ între proiectele mici, medii și mari.

An	Succes	Finalizat	Eșec
1994	16%	53%	31%
1996	27%	33%	40%
1998	26%	46%	28%
2000	28%	49%	23%
2002	34%	51%	15%
2004	29%	53%	18%
2007	35%	46%	19%
2009	32%	44%	24%
2010	37%	42%	21%
2011	29%	49%	22%
2012	27%	56%	17%
2013	31%	50%	19%
2014	28%	55%	17%
2015	29%	52%	19%

Tabelul 1.1. Rezultatele raportului CHAOS al *Standish Group International* în perioada 1994 – 2015 ()

În altă ordine de idei, este destul de restrictiv să măsurăm succesul unui proiect luând în considerare doar criteriile scop, timp și ban. Calitatea produsului sau a serviciului rezultat reprezintă la rândul său un criteriu important. De asemenea, există proiecte care nu au respectat cele trei criterii, dar care au fost considerate de către organizațiile în care s-au desfășurat ca fiind de mare succes pentru că au atras ulterior noi proiecte importante, după cum există și proiecte ce au excelat în toate cele trei direcții însă presiunea exercitată a făcut ca echipa de proiect să se destrame rapid după terminarea proiectului pierzându-se experiența și expertiza câștigate în domeniul proiectului respectiv.

Începând cu anul 2012 raportul vine cu informații agregate pe diferite tipuri de metodologii utilizate în gestionarea proiectelor. După cum se poate observa și din tabelul 1.2 ce conține date privind succesul proiectelor software analizate în 2015, metodologiile Agile ameliorează simțitor numărul de proiecte eșuate, însă ele nu pot fi aplicate cu succes pentru toate tipurile de proiecte informatice.

Metodă	Succes	Finalizat	Eșec
Agile	39%	52%	9%
Waterfall	11%	60%	29%

Tabelul 1.2. Analiza succesului proiectelor software per metodologii în 2015, conform *Standish Group International* [2]

Ce am dorit să scoatem în evidență până la acest punct este că există diferențe notabile între proiectele informatice și proiecte din alte domenii de activitate și că aceste diferențe par a influența negativ succesul acestora. Vom detalia în secțiunea următoare principalele aspecte ce diferențiază proiectele informatice de proiectele tradiționale, așa cum au fost prezentate în [3].

Caracteristicile proiectelor software

Softul este nemăsurabil

Din fericire sau din păcate (în funcție de perspectiva din care privim lucrurile), activitatea de dezvoltare a softului este una creativă. Nu există, așa cum se întâmplă în cazul altor activități, un nomenclator care să precizeze care este timpul uzual necesar pentru implementarea unei ferestre ce conține, să zicem, două liste derulante, un tabel (*grid*) și două butoane. În ciuda dezvoltării de instrumente sofisticate de generare automată a codului sau de implementarea diverselor biblioteci de clase sau *framework*-uri, fiecare proiect nou vine cu provocările proprii în ceea ce privește creativitatea.

Un exemplu edificator în acest sens este dat în [4] în care se face un experiment chestionând un număr de 44 de experți în legătură cu numărul de instrucțiuni care apar în codul de mai jos:

```
#define      LOWER  0
#define      UPPER  300
#define      STEP   20
main()
{
  int fahr;
  for (fahr=LOWER; fahr<=UPPER; fahr=fahr+STEP)
    printf("%4d %6.1 f\n", fahr, (5.0/9.0*(fahr-32)))
}
```

Figura 1.1. Program scris în limbajul C
de conversie din grade Fahrenheit în grade Celsius [4]

Răspunsurile acestora cuprind toate numerele între 1 și 9 inclusiv (11 experți au răspuns 6, alții 11 au răspuns 9, restul alegând ca răspuns un alt număr din interval)! Concluzia este imediată: dacă mai mulți experți în domeniul *software* nu reușesc să cadă de acord asupra unei întrebări simple pe baza unui program de 9(!) linii de cod, înseamnă că este de așteptat ca estimările într-un proiect software cu mii de linii de cod să difere foarte mult de la o persoană la alta, aceste estimări având o relativ restrânsă legătură cu experiența acumulată.

O practică uzuală este aceea ca estimările date (atunci când este vorba de estimări de timp și nu în puncte de complexitate) să fie mărite cu 20% de către managerul de proiect înainte ca ele să fie transmise către client, pentru a se asigura un oarecare "confort" (deși sunt situații în care acest 20% este departe de a fi suficient).

O altă consecință directă a acestui aspect o reprezintă dificultatea gestionării schimbărilor într-o echipă de proiect, persoanele care vor înlocui anumiți membri ai echipei rareori respectând estimările date de aceștia.

Și pentru ca lucrurile să fie "complete", pentru multe dintre activitățile estimate este foarte dificil de controlat progresul. Cele mai reprezentative exemple aici sunt activitățile de analiză și proiectare unde putem știi când s-au terminat aceste activități, dar nu vom putea specifica dacă la un moment dat ne aflăm la 50% la 75% din activitatea respectivă.

Softul este invizibil și intangibil

Rezultatul muncii unei echipe ce dezvoltă un produs informatic este compus dintr-o colecție de "texte" de diferite tipuri: documente de analiză și proiectare, cod sursă, manuale de utilizare și operare etc. Doar o parte dintre acestea interesează sau au vreo semnificație pentru cei care vor utiliza produsul.

Clientul final tinde să evalueze un produs informatic după ceea ce vede, iar ceea ce vede este în general o interfață grafică cu utilizatorul.

Deoarece nu există nimic concret de arătat clientului în faza de analiză a cerințelor, acesta își formează o imagine proprie asupra rezultatului final care de cele mai multe ori nu se potrivește cu produsul dezvoltat. În plus, există tendința de a include în cadrul specificațiilor elemente care reprezintă mai degrabă dorințe decât nevoi reale de *business*. Se ajunge astfel într-un punct caracterizat de un grad ridicat de frustrare, în care echipa de proiect știe că a dezvoltat conform specificațiilor funcționale dar clientul nu poate utiliza aplicația finală pentru că nu este ceea ce și-a imaginat că va fi. Acest rezultat poate fi ameliorat prin prezentarea unuia sau mai multor prototipuri în fazele timpurii ale dezvoltării produsului, însă clientul nu va face întotdeauna diferența între acestea și produsul final crezând că nu s-a depus un efort semnificativ între timp, interfața cu utilizatorul rămânând aproape neschimbată.

Tot din cauza invizibilității softului și implicit a complexității acestuia, cerințele inițiale par foarte ușor de modificat și implementat în cadrul unei aplicații.

Softul are o complexitate ridicată

Fără doar și poate produsele *software* au o structură extrem de complexă. Într-o aplicație de dimensiuni medii există extrem de multe conexiuni între diverse elemente *software* care fac aproape imposibilă testarea și validarea completă a acestora. Un exemplu simplu în care folosim nouă condiții succesive intercalate poate rezulta într-un număr impresionant de căi posibile care trebuie testate fiecare în parte pentru o validare completă. Acest lucru însă ar putea să nu fie realizabil nici într-o săptămână, chiar dacă testul fiecărei căi ar dura o secundă. Mai mult, aceste teste ar trebui reluate de fiecare dată când se face o modificare. Prin urmare multe dintre

defecte persistă o vreme îndelungată (chiar ani de zile) într-o aplicație până să fie descoperite sau, mai rău, până să își arate efectele.

După cum se spunea în aceeași lucrare menționată anterior: "*...în cazul proiectelor software așa numitele proceduri de control al calității au uneori de-a face mai degrabă cu limitarea defecțiunilor decât cu garantarea calității produsului final.*" [4]. Și acest lucru a rămas valabil și în ziua de azi, în ciuda tuturor metodologiilor moderne de dezvoltare a softului apărute în ultimii ani.

Complexitatea softului este dată și de numărul de tranziții și "traduceri" ce trebuie realizate între fazele ciclului de viață al unui produs. Specificațiile funcționale (redactate în limbaj natural) sunt translatate într-un model de analiză (diagrame vizuale care modelează toate soluțiile posibile ale problemei enunțate în specificații), care ulterior este translatat într-un model de proiectare (unde se particularizează modelul de analiză pentru o soluție specifică), care mai apoi este translatat în cod sursă, acesta din urmă fiind translatat într-un model executabil (prin compilare sau interpretare). Tot acest lanț de translatări, în ciuda faptului că anumite tranziții sunt automatizate, face procesul de dezvoltare a softului mult mai vulnerabil la erori umane. Acest lucru este cauzat de greșeli de "traducere" sau prin persistarea erorilor nedescoperită la timp în fazele de specificare sau analiză până în modelul executabil.

Softul este dinamic

Dinamismul cu care se confruntă proiectele software se manifestă în trei direcții relevante. Pe de o parte există atracția noutății tehnologice. Știm că tehnologia avansează foarte rapid și an de an apar biblioteci și *framework*-uri noi, și versiuni îmbunătățite ale limbajelor și mediilor de dezvoltare. Din punct de vedere al unui manager de proiect, păstrarea *framewok*-urilor inițiale în care a fost dezvoltat un anumit soft este o condiție elementară de păstrare a stabilității acestui soft. Pe de altă parte nu trebuie ignorată apetența echipei pentru a lucra cu ultimele tehnologii (inerent mai instabile și cu neajunsuri încă nerezolvate sau nediscutate pe forumurile de specialitate). Există un efort constant și care nu trebuie neglijat, de păstrare a unui echilibru între stabilitate și eliminarea unui sentiment de plafonare a echipei de dezvoltare.

A doua direcție ce conferă dinamism proiectelor software este fluctuația personalului, care în domeniul IT este foarte ridicată. O fluctuație "sănătoasă" pentru o organizație se situează undeva în jurul a 10 procente. Un studiu realizat de Ziarul Financiar arată ca în domeniul IT în România fluctuația de personal este la nivelul de 20%, și ea nu se datorează în principal nivelului de salarizare (cum este în cazul altor domenii) ci diferențelor de cultură și a sentimentului de nerealizare profesională. Efectele acestei fluctuații, după cum subliniam și mai devreme, constau în reconsiderarea planificărilor anterioare și de adaptare a noilor veniți la proiect. Nu trebuie însă neglijată tendința noilor veniți de a respinge codul scris de predecesorii lor, încercând uneori chiar să își asume responsabilitatea înlocuirii complete a acestui cod, acțiune ce conduce la anularea validărilor anterioare.

În fine, o dinamică aparte față de alte proiecte o au cererile de modificare frecvente venite din partea clientului în diverse faze ale ciclului de viață a dezvoltării unui proiect software. Această caracteristică a proiectelor software a condus la dezvoltarea metodologiilor Agile care includ acest aspect ca pe o componentă activă a procesului de dezvoltare.

Concluzii

Prin urmare softul este nemăsurabil, invizibil, intangibil, dinamic și are o nivel de complexitate ridicat. Unele dintre aceste atribute sunt specifice proiectelor bazate pe gestionarea cunoștințelor (nemăsurabil, intangibil) altele sunt comune și altor tipuri de proiecte (complex, dinamic) însă în cazul proiectelor software pare ca au un impact mai mare.

Toate acestea vin în sprijinul ideii că proiectele software în particular (și cele bazate preponderent pe gestionarea cunoștințelor în general) necesită practici și instrumente de lucru diferite în comparație cu celelalte tipuri de proiecte. Aceste considerente au influențat dezvoltarea unor metodologii noi, care inițial au fost utilizate pe o scară redusă dar care au „pavat" încet un drum ce urma să ducă, în 2001, la nașterea unei noi filozofii de abordarea a proiectelor: filozofia Agile.

2. Filozofia Agile

Am numit acest capitol „Filozofia Agile" pentru a sublinia faptul că Agile nu este o metodologie de gestionare a proiectelor, așa cum este perceput în mod greșit azi de mulți dintre cei ce lucrează în industrie. Este, mai degrabă, o modalitate de a vedea și aborda proiectele și constrângerile acestora, o mentalitate. E adevărat că, sub umbrela acestei mentalități au fost propuse și implementate în echipele de proiect mai multe metodologii, instrumente sau practici. Agile însă este mult mai mult: este o colecție de valori și principii care trebuie să guverneze întregul ciclu de viață al unui proiect. În cele ce urmează vor fi detaliate toate aceste valori și principii.

Manifestul Agile

Particularitățile proiectelor software descrise în capitolul precedent au condus în timp la dezvoltarea și implementarea unor practici care să asigure obținerea unei rate mai mari de succes ale acestor tipuri de proiecte. Astfel, începând cu anii 1990 s-au dezvoltat mai multe metode independente de gestionare a proiectelor software. Aceste metode nu foloseau în mod necesar tehnici radical diferite față de cele utilizate în managementul tradițional însă se bazau pe un set de valori și principii derivate din particularitățile proiectelor software. Pe măsură ce astfel de metode independente se înmulțeau a apărut ca evidentă necesitatea găsirii unui numitor comun sau, de ce nu, a definirii unor standarde.

În 2001 un număr de 17 consultați independenți din industria software s-au întâlnit la un resort de schi din Utah, Statele Unite ale Americii, unde au pus bazele a ceea ce urma să se numească ulterior mișcarea Agile. Ei au elaborat următorul manifest ce se poate găsi tradus în peste 65 de limbi la adresa http://agilemanifesto.org și care sună astfel:

„Noi descoperim modalități mai bune de dezvoltare de software din experiența proprie și ajutându-i pe cei din jurul nostru în această activitate. Astfel am ajuns să apreciem:

***Indivizii și interacțiunea** înaintea proceselor și instrumentelor*

***Software funcțional** înaintea documentației vaste*

***Colaborarea cu clientul** înaintea negocierii contractuale*

***Receptivitatea la schimbare** înaintea urmăririi unui plan*

Cu alte cuvinte, deși există valoare în elementele din dreapta, le apreciem mai mult pe cele din stânga." [5]

Chiar dacă textul acestui manifest este orientat evident către proiecte din industria software, el este aplicabil oricărui tip de proiect a cărui activitate se bazează pe managementul cunoștințelor și a cărui echipă de proiect este formată din *knowledge workers*.

Metodologiile care au îmbrățișat aceste valori s-au numit inițial metodologii de "categorie ușoară" (*lightweight methodologies*) urmând ca ulterior aceste să fie cunoscute ca metodologii Agile.

Principiile Agile

Deși clare și relevante, valorile enunțate în Manifestul Agile nu au fost suficiente pentru a ajuta echipele de proiect să coordoneze proiecte într-o manieră agilă, acestea având nevoie de câteva direcții specifice. Ca urmare a acestui fapt au fost enunțate 12 principii care, la rândul lor, se regăsesc la http://agilemanifesto.org. Aceste principii sunt:

1. *Prioritatea noastră este satisfacția clientului prin livrarea rapidă și continuă de software valoros.*
2. *Schimbarea cerințelor este binevenită chiar și într-o fază avansată a dezvoltării. Procesele agile valorifică schimbarea în avantajul competitiv al clientului.*
3. *Livrarea de software funcțional se face frecvent, de preferință la intervale de timp cât mai mici, de la câteva săptămâni la câteva luni.*
4. *Oamenii de afaceri și dezvoltatorii trebuie să colaborez zilnic pe parcursul proiectului.*

5. *Construiește proiecte în jurul oamenilor motivați. Oferă-le mediul propice și suportul necesar și ai încredere că obiectivele vor fi atinse.*
6. *Cea mai eficientă metodă de a transmite informații înspre și în interiorul echipei de dezvoltare este comunicarea față în față.*
7. *Software funcțional este principala măsură a progresului.*
8. *Procesele agile promovează dezvoltarea durabilă. Sponsorii, dezvoltatorii și utilizatorii trebuie să poată menține un ritm constant pe termen nedefinit.*
9. *Atenția continuă pentru excelență tehnică și design bun îmbunătățește agilitatea.*
10. *Simplitatea - arta de a maximiza cantitatea de muncă nerealizată - este esențială.*
11. *Cele mai bune arhitecturi, cerințe și design emerg din echipe care se auto-organizează.*
12. *La intervale regulate, echipa reflectă la cum să devină mai eficientă, apoi își adaptează și ajustează comportamentul în consecință.*

La o primă citire principiile de mai sus pot părea greoaie și iar circumstanțele în care se aplică sunt greu de înțeles. Un mod eficient de a percepe aceste principii este asocierea lor cu cuvinte sau expresii simple ce reflectă esența principiilor, cum ar fi:

1.Prioritatea noastră este satisfacția clientului prin livrarea rapidă și continuă de software de valoare.	Produs valoros pentru client
2. Schimbarea cerințelor este binevenită chiar și într-o fază avansată a dezvoltării. Procesele agile valorifică schimbarea în avantajul competitiv al clientului.	Schimbările sunt binevenite
3. Livrarea de software funcțional se face frecvent, de preferință la intervale de timp cât mai mici, de la câteva săptămâni la câteva luni.	Livrare frecventă
4. Oamenii de afaceri și dezvoltatorii trebuie să colaborez zilnic pe parcursul proiectului.	Aproape de business
5. Construiește proiecte în jurul oamenilor motivați. Oferă-le mediul propice și suportul necesar și ai încredere că obiectivele vor fi atinse.	Încredere în oameni
6. Cea mai eficientă metodă de a transmite informații înspre și în interiorul echipei de dezvoltare este comunicarea față în față.	Comunicare față-în-față

7. Software funcțional este principala măsură a progresului.	Măsoară terminare produs
8. Procesele agile promovează dezvoltarea durabilă. Sponsorii, dezvoltatorii și utilizatorii trebuie să poată menține un ritm constant pe termen nedefinit.	Menține un ritm sustenabil
9. Atenția continuă pentru excelență tehnică și design bun îmbunătățește agilitatea.	Întreține designul
10. Simplitatea - arta de a maximiza cantitatea de muncă nerealizată - este esențială.	Nu te complica
11. Cele mai bune arhitecturi, cerințe și design emerg din echipe care se auto-organizează.	Echipa creează arhitectura
12. La intervale regulate, echipa reflectă la cum să devină mai eficientă, apoi își adaptează și ajustează comportamentul în consecință.	Analizează și ajustează

Tabel 2.1. Esența principiilor Agile

Management tradițional și management Agile

Principiile și valorile enunțate în Manifestul Agile vizează schimbarea modului de abordare și percepție a mai multor activități ce fac parte din ciclul de viață al unui proiect.

Denumirea Agile în sine exprimă adaptabilitate și flexibilitate și prin urmare indică o manieră mai puțin rigidă de management al proiectului. În consecință reacția de răspuns la evenimente neașteptate apărute în proiect este mult mai rapidă dar, în același timp, ținându-se cont de menținerea echilibrului între elementele ce definesc succesul proiectului.

Această abordare contrastează cu activitățile caracteristice managementului tradițional, care se bazează pe aplicarea rigidă și riguroasă a unor pași dinainte stabiliți și pe controlul și abordarea modificările astfel încât să afecteze cât mai puțin acești pași.

Aspectele cheie care fac diferența între managementul tradițional și cel Agile prin felul în care sunt abordate sunt:

- tratarea schimbările (de specificații/scop, de echipă, de resurse etc),
- planificarea,
- comunicarea, și

- învățarea.

Metodologiile tradiționale privesc schimbarea ca un „inamic" al proiectului. În cazul în care nu este posibilă eliminarea lor se depune un efort semnificativ în controlarea și reducerea impactului acestora asupra proiectului. Schimbările sunt privite ca excepții și sunt documentate, analizate și aprobate individual ca urmare a unui plan de gestionare a schimbărilor foarte bine stabilit. În Agile schimbările nu sunt tratate în mod special față de alte elemente ale proiectului ci sunt privite ca o parte integrantă (chiar de dorit) a acestuia. Orice schimbare reprezintă o oportunitate de creștere a valorii produsului/serviciului final.

Evident, acest lucru afectează și abordarea planificării. În timp ce în metodologiile tradiționale diferențele dintre planificare și realitate sunt rezolvate mai degrabă prin aplicarea de presiuni „realității", în Agile planurile suferă modificări dese pe toată durata de viață a proiectelor adaptându-se modificărilor apărute. Acesta este și motivul pentru care planurile în Agile sunt detaliate doar pe perioade foarte scurte de timp (1-4 săptămâni) iar planurile generale sunt doar schițate.

În ceea ce privește comunicarea, așa cum se subliniază și în principiul 6, comunicarea față în față este de dorit utilizării unor documente stufoase și formale. Acest lucru nu înseamnă că o serie de aspecte nu sunt documentate, însă acestea reprezintă minimul necesar util pentru a continua în bune condiții munca pe proiecte. Pe de altă parte, canalele de comunicare în managementul tradițional sunt impuse într-o oarecare măsură de ierarhii, lucru care în majoritatea metodologiilor Agile nu se întâmplă, comunicarea profesională putându-se face oricând cu oricine.

Nu în ultimul rând, modul în care sunt asimilate și implementate îmbunătățirile de proces de lucru în managementul tradițional face ca aceste îmbunătățiri să afecteze rareori proiectul curent și să fie mai degrabă aplicate în proiecte ulterioare în care vor fi implicați membrii echipei de proiect. În abordarea Agile sesiunile de discutare a disfuncționalităților de proces sunt foarte dese și tratarea lor are loc imediat prin decizii de modificare a unor practici sau tehnici de lucru ce sunt evaluate ulterior de către întreaga echipă.

3. Evoluția metodologiilor Agile

Această secțiune își propune pe de o parte să explice de ce valorile și principiile enunțate în Manifestul Agile reprezintă un mod de abordare potrivit al proiectelor ce au caracteristicile amintite în primul capitol al acestei cărți: nemăsurabile, invizibile, intangibile, complexe și dinamice. Acest lucru îl vom realiza prezentând un excelent sistem de clasificare și abordare a problemelor propus de Dave Snowden în [6], și anume sistemul Cynefin.

De asemenea, în partea a doua a capitolului, vor enumerate cele mai importante (și populare) metodologii Agile dezvoltate până în prezent și câteva considerente privind modul în care acestea au evoluat.

Sistemul de clasificare a problemelor *Cynefin*

În 2002 Dave Snowden propunea un sistem de abordare a problemelor numit Cynefin Framework [6]. Cynefin este un instrument de luare a deciziilor care descrie modalitățile prin care putem aborda diversele probleme pe care le întâmpinăm prin detectarea domeniului de care acestea aparțin. Este important de observat că Cynefin (pronunțat [*ku-nev-in*]) nu descrie și nu propune soluții de probleme ci căi de detectare a soluțiilor.

Dave Snowden pornește la definirea Cynefin Framework de la descrierea a trei tipuri de sisteme:

- sisteme **ordonate**,
- sisteme **complexe**, și
- sisteme **haotice**.

Aceste sisteme diferă între ele prin modul în care este percepută în cadrul lor relația dintre cauză și efect. Astfel, sistemele **ordonate** sunt sistemele în care există o relație clară între cauză și efect, iar aceasta este repetabilă și predictibilă. Prin urmare, în aceste sisteme vom ști exact ce anume vom obține ca urmare a realizării unor acțiuni înainte de a întreprinde acele acțiuni.

Spre deosebire de sistemele ordonate, în cazul sistemelor **complexe** relația dintre cauză și efect nu este predictibilă. Înțelegem că fiecare efect are o cauză clară însă determinarea cauzei se poate face doar inspectând în ordine inversă pașii ce au condus la un anumit efect. Un lucru foarte interesant ce caracterizează sistemele complexe este acela că nu putem fi siguri că în urma repetării unei acțiuni aceasta va avea același efect cu acțiunea precedentă. Acest lucru se întâmplă deoarece apar diverși factori care nu sunt cunoscuți sau percepuți inițial, atunci când s-a realizat acțiunea.

În fine, în cazul sistemelor **haotice** nu se poate determina existența unei relații între cauză și efect indiferent dacă parcurgem pașii ce au dus la un efect într-un sens sau altul (acest lucru se poate întâmpla și pentru că scopul problemei este complet necunoscut).

Cynefin împarte aceste 3 sisteme în 5 domenii diferite astfel (v. figura 3.1.):

- domeniile ***Evident*** și ***Complicat*** (derivate din sistemele ordonate)
- domeniul ***Complex***
- domeniul ***Haotic***
- domeniul ***Confuz*** (când o problemă nu se află încă în nici una din domeniile mai sus o vom asocia domeniului *Confuz*)

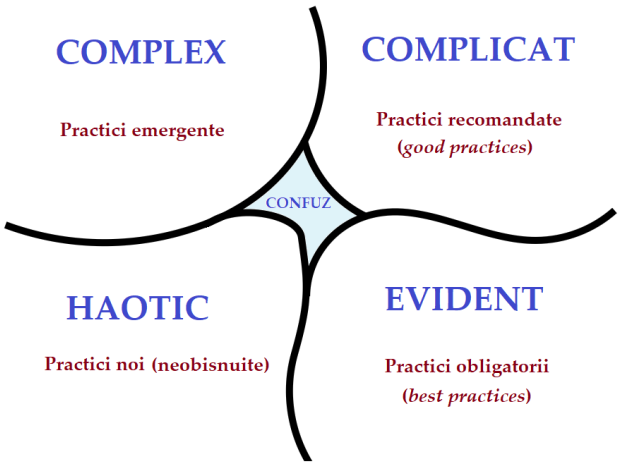

Figura 3.1. Reprezentare schematică a domeniilor sistemului *Cynefin*

Domeniul problemelor ***Evidente*** este domeniul în care există o conexiune foarte clară între cauză și efect (ca parte a sistemelor ordonate) și aceasta este vizibilă și ușor de observat de către oricine (este predictibilă de către orice persoană). În acest caz trebuie doar observată problema și determinată categoria din care face parte pentru a putea răspunde corespunzător (abordare *sense – categorize - respond*). Fiecare problemă dintr-o categorie poate fi rezolvată acționând într-un anumit fel (descris printr-o secvență clară de pași), sau cu alte cuvinte urmând un *best practice* (practică obligatorie). Un exemplu de problemă din domeniul *Evident* este utilizarea pentru prima dată unei aplicații de complexitate redusă (cu sau fără manual de utilizare). După o perioadă scurtă de timp vom putea determina ce anume trebuie să facem pentru a lista la imprimantă un raport sau pentru a modifica numele unei persoane.

În cazul problemelor ce fac parte din domeniul ***Complicat*** conexiunea dintre cauză și efect, deși există, nu este evidentă pentru toată lumea. Ea este predictibilă doar de către persoane specializate în domeniul problemei (experți) iar soluțiile pot fi diferite în funcție de experiența expertului sau condițiile externe. De aceea modul de abordare a problemelor complicate constă în observarea și analizarea de către un specialist și apoi se va alege răspunsul potrivit (*sense - analyze - respond*). Deoarece soluția în acest caz nu este unică, pentru problemele din acest domeniu vom putea face apel la *bune practici* (sau practici recomandate). Un exemplu de problemă din domeniul *Complicat* este funcționarea defectuoasă a unui calculator sau laptop. Pentru a le repara vom avea

nevoie de un specialist iar soluțiile adoptate de aceștia, deși duc la rezolvarea problemei, pot fi total diferite (reparare, înlocuire, evitare etc).

Domeniul de probleme ***Complex*** are, așa cum deja am văzut, proprietatea că relațiile dintre cauză și efect nu sunt predictibile. Ele pot fi determinate doar prin inspectarea rezultatului și observarea acțiunilor care au dus la acel rezultat. De aceea existența un set de practici nu este utilă în acest caz, deoarece nu vom putea fi siguri că aplicându-le vom rezolva problema. Nici un grup de experți nu este neapărat util, deoarece problema le va oferi acestora elemente ce le sunt necunoscute. Prin urmare soluția este găsită prin încercări succesive ce sondează problema până când se ajunge la o soluția acceptabilă (*probe – sense – respond*). Practicile utilizate sunt practici emergente, sau, cu alte cuvinte, practici care scot soluția la iveală după o serie de căutări și experimentări. Un element esențial în aplicarea acestor practici emergente este un mediu în care eșecul nu reprezintă o nouă problemă ci o oportunitate de învățare, un mediu în care este în regulă să eșuezi. O situație ce aparține domeniului ***Complex*** este participarea la un interviu de angajare. Cu siguranță există fraza sau frazele perfecte care îți vor permite să obții o ofertă de angajare, însă aceasta nu poate fi determinată înainte (nici studiind zeci de alte interviuri, nici consultând pagina web a organizației ce angajează). Ce poți afla sunt tehnici emergente, care să ajute la găsirea răspunsului în timpul conversației avute la interviu.

În fine, domeniul ***Haotic*** este guvernat de „legile" sistemelor haotice în care, așa cum am văzut deja, nu există corelație între cauză și efect. Pentru problemele ce aparțin acestui domeniu cel mai important lucru este luarea unei decizii. În astfel de situații nu este timp pentru consultarea manualului, apelarea experților, pentru experimentare sau analizarea problemei. Trebuie reacționat imediat pentru a stabiliza problema, apoi pe baza observațiilor asupra efectele reacției se răspunde în consecință (*act - sense - respond*). În acest caz practicile sunt noi, dependente de situația existentă și singurul scop este acela de a scoate cât mai repede posibil problema din acest domeniu. Un exemplu de situație din domeniul ***Haotic*** este descoperirea unui defect într-o aplicație software ce aduce prejudicii financiare majore clientului. Prima reacție în acest caz este oprirea sistemului înainte de a face orice alte investigații. Abia după ce sistemul este oprit vom putea aborda problema după cum ea este una evidentă, complicată sau complexă.

Cel de-al cincilea domeniu, ***Confuz***, este cel în care se află problemele pe care nu am reușit să le asociem încă uneia dintre domeniile anterioare. Este bine ca o problemă să nu stea mult timp în această zonă și să determinăm modul în care o vom aborda într-un timp cât mai scurt.

În concluzie *Cynefin Framework* ne ajută să identificăm modul potrivit prin care putem aborda o problemă. Atunci când vom clasifica problema trebuie să ținem cont de faptul că analiza noastră poate fi influențată de problemele cu care am fost obișnuiți până acum sau de domeniul în care noi ne simțim mai confortabili. De aceea, uneori suntem tentați să clasificăm o problemă complexă ca fiind complicată sau o problemă complicată ca fiind haotică pentru că simțim că vom face față mai bine într-un astfel de context.

Proiectele software în particular și proiectelor bazate pe managementul cunoștințelor în general sunt proiecte complexe. Soluțiile nu sunt predictibile și se pot descoperi în urma unor experimente iterative și incrementale. Pe de altă parte, managementul tradițional este un excelent mod de abordare a problemelor complicate, unde relațiile cauză – efect sunt predictibile și repetabile. Una din principalele cauze pentru care rata de eșec a proiectelor informatice este atât de mare este aceea că s-a încercat abordarea lor cu instrumente (bune practici) specifice problemelor complicate. De cele mai multe ori eșecul este privit ca o consecință a unei slabe implementări a procesului. În același timp, valorile și principiile Agile conduc către o abordare iterativă și incrementală a descoperirii soluției, cu evaluări si validări intermediare ce caracterizează practicile emergente și, prin urmare, sunt potrivite pentru abordarea problemelor din domeniul Complex.

O serie de probleme se deplasează, în timp, dintr-un domeniu în altul. Acest lucru este normal și, în general, nu este costisitor. Excepția constă în deplasarea unei probleme din ***Evident*** în ***Haotic***. Această tranziție se poate face relativ ușor atunci când, nefiind atenți la modificările de context ale problemei, aplicăm aceleași practici care au funcționat de nenumărate ori. La primul eșec de aplicare a unei astfel de practici problema poate trece în domeniul ***Haotic***. Tranziția inversă, de a readuce problema în domeniul ***Evident***, este foarte costisitoare și se face traversând domeniile ***Complex*** și ***Complicat***.

Evoluția metodologiilor Agile

După cum am văzut și în capitolul anterior, metodologii ce conțineau elemente de agilitate au apărut înainte ca manifestul Agile să fie enunțat. Ideea de bază a acestor metodologii consta în relaxarea unora dintre procesele mai rigide ale managementului tradițional și adaptarea într-un ritm cât mai rapid la modificările frecvente ce apăreau în proiecte. Manifestul Agile nu a făcut altceva decât să structureze această idee și să o „îmbrace" cu 4 valori fundamentale pentru abordarea proiectelor executate de echipe de *knowledge workers*.

A urmat apoi enunțarea celor 12 principii care, pe baza valorilor, evidențiază cu mai multă claritate direcția de abordare a proiectelor. Cu toate acestea exista în continuare nevoia de identificarea unor instrumente sau tehnici care să implementeze în practică ideile cuprinse în valorile și principiile Agile. Acestea au format un al 3-lea „strat" în jurul ideii Agile (vezi figura 3.2), mult mai generos decât celelalte prin prisma numărului de elemente. Aceste instrumente și practici general acceptate, au fost adoptate în fiecare organizație sau echipă împreună cu anumite practici specifice organizației și echipei.

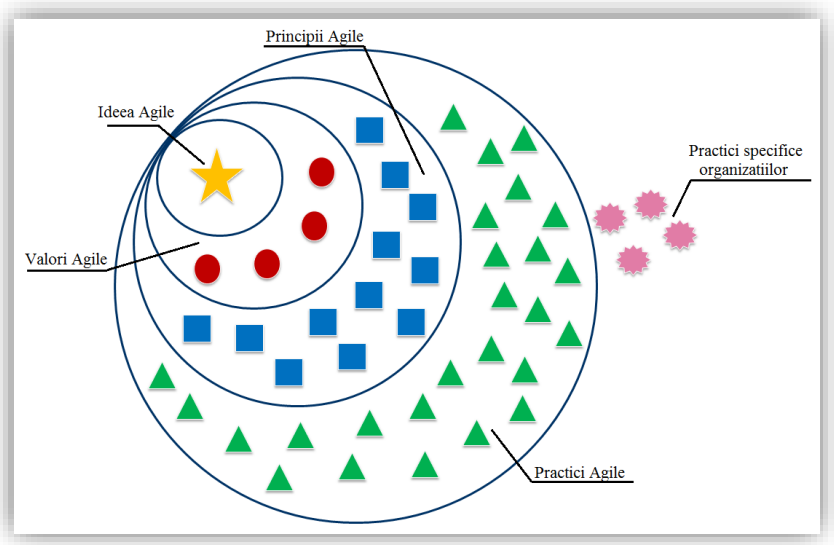

Figura 3.2. Drumul de la idee la practici în universul Agile

In figura 3.3 sunt prezentate grafic metodologiile Agile dezvoltate ce pot fi considerate ca o mulțime de tehnici și instrumente legate între ele.

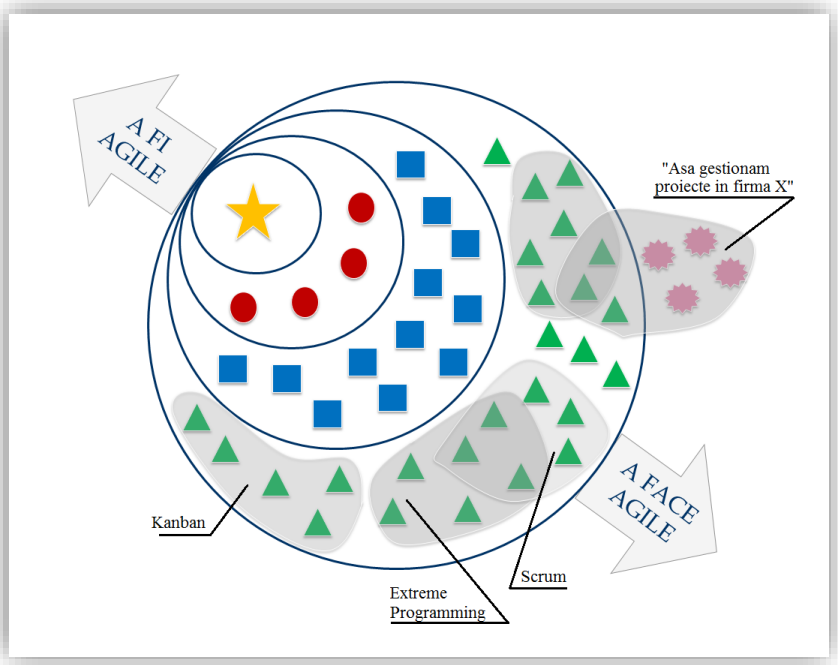

Figura 3.3. De-a lungul evoluției metodologiilor Agile comunitatea a făcut o tranziție graduală de la practici emergente la *best practices*

Unele metodologii au elemente comune (cum ar fi *Extreme Programming* și *Scrum*), altele sunt ortogonale putând fi aplicate împreună cu alte metodologii (cum sunt, de exemplu, *Kanban* și *Scrum*). De asemene, o serie de organizații (în special cele mari cu sute de angajați grupați în echipe de proiect) și-au dezvoltat propriile metodologii ca derivate ale metodologiilor cunoscute.

Acesta este de bună seamă un semn de progres și de maturizare a metodologiilor alternative de management al proiectelor. În același timp însă atenția generală s-a mutat pe aceste practici și instrumente, în special pe implementarea lor „corectă". Și acest lucru se întâmplă în multe echipe fără ca acestea să își pună întrebarea dacă respectivele practici sunt într-adevăr utile sau benefice pentru proiectul lor. Cu alte cuvinte asistăm la o paradoxală abordare rigidă a agilității, ceea ce face ca multe echipe să trateze instrumentele și tehnicile Agile ca *best practices* și să le

implementeze ca atare deturnând astfel respectarea unor valori și principii prin urmarea strictă a unui proces. Acest lucru este reprezentat în figura 3.3 prin cele două tendințe opuse ce pot influența negativ succesul proiectelor: „a fi" și „a face" Agile. Astfel scopul principal al unui manager de echipă Agile este acela de a păstra constant un echilibru între cele două tendințe.

Metodologii Agile relevante

În ultimii 20 de ani au fost dezvoltate zeci de metodologii Agile sau mixte (Agile + tradițional). O parte din cele mai populare sunt prezentate în figura 3.4. Metodologia *Scrum* este lider detașat în această statistică (58% din proiecte urmează *Scrum* iar 17% folosesc o metodologie bazată pe *Scrum* combinată cu *Extreme Programming*, respectiv *Kanban*) și acesta este motivul pentru care în multe situații Agile și Scrum sunt confundate. Popularitatea *Scrum* este în continuă creștere (în 2011 un raport similar al VersionOne arăta că Scrum e folosit in 52% din proiectele Agile), dar și numărul proiectelor ce folosesc metodologii Agile este în continuă creștere. Interesant este procentul de persoane care nu au idee ce metodologie se folosește în proiectelor lor (este adevărat că acest procent este în scădere).

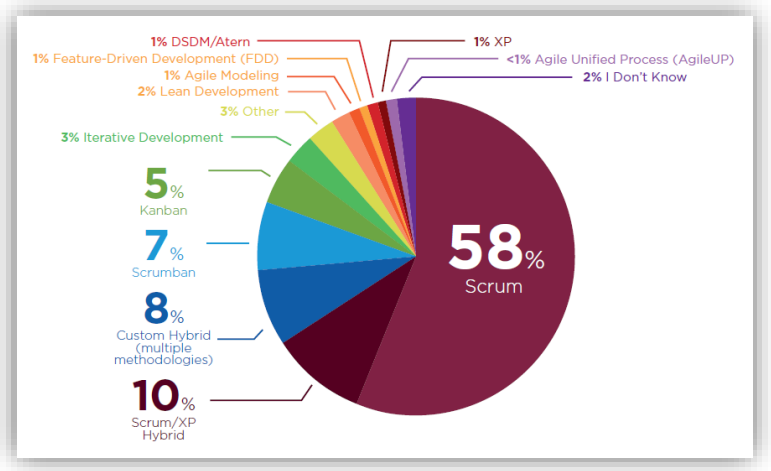

Figura 3.4. Rata de adoptare a metodologiilor Agile
în proiecte IT în 2015 [7]

Din punct de vedere al tehnicilor utilizate același raport al VersionOne identifică un top al celor mai importante 5 practici (figura 3.5).

Figura 3.5. Top 5 practici Agile
în 2015 [7]

Tabelul 3.1 conține o listă a tuturor metodologiilor ce vor fi descrise în continuare, alături de anul apariției și numele inițiatorilor.

Metodologie	Inițiere	Descriere
Scrum	Jeff Sutherland, Ken Schwaber, 1996	Capitolul 4
Extreme Programming (XP)	Kent Beck, 1999	Capitolul 5
Lean Software Development	Mary Poppendieck; Tom Poppendieck, 2003	Capitolul 6
Kanban	Taiichi Ohno, 1988 (industria manufacturieră) David Anderson 2010 (software)	Capitolul 7
Feature Driven Development (FDD)	Jeff De Luca, 1997	Capitolul 8
Agile Unified Process (AUP)	Scott Ambler, 2010	Capitolul 8
Crystal	Alistair Cockburn, 2004	Capitolul 8
Dynamic Systems Development Method (DSDM)	DSDM Consortium, 1994	Capitolul 8

4. *Scrum*: roluri, ceremonii și artefacte

Scrum este o metodologie Agile ce utilizează cicluri iterative și incrementale de dezvoltare a produselor. Dacă inițial Scrum a fost utilizat pentru dezvoltarea de software recent a fost adoptat și în alte sfere cum ar fi companii medicale și farmaceutice sau de construcții.

În 1986 Hirotaka Takeuchi și Ikujiro Nonaka au descris o abordare de dezvoltare a produselor ce ducea la creșterea vitezei de dezvoltare și a flexibilității prin utilizarea de echipe inter-funcționale ce traversează fazele proiectului similar cu modul de avansare în teren a unei echipe de rugby. Termenul de „scrum" este împrumutat de fapt din rugby și reprezintă o metodă de re-începere a jocului prin strângerea tuturor jucătorilor într-o „grămadă". Acest termen a fost utilizat pentru prima dată Peter DeGrace și Leslie Stahl în 1990 în cartea lor „*Wicked Problems, Righteous Solutions: A Catalogue of Modern Software Engineering Paradigms*". Dar metodologia Scrum așa cum este ea cunoscută și implementată azi a fost descrisă în 1995 de către Jeff Sutherland și Ken Schwaber într-o prezentare publică.

Scrum este o metodologie simplă ce conține un set minimal de roluri, practici (*evenimente*) și instrumente (*artefacte*). Teoria pe care se bazează Scrum e compusă din trei piloni importanți:

- ***Transparență***: cei responsabili au vizibilitate asupra rezultatului. Un aspect al transparenței este dată de crearea unei definiții comune a ceea ce se înțelege prin terminarea unei activități (*definition of „done"*), definiție făcută cu participarea tuturor celor care sunt implicați în proiect.
- ***Inspectare***: verificări periodice a progresului proiectului în ceea ce privește atingerea scopului și analiza deviațiilor critice sau ale diferențelor majore.

- ***Adaptare***: ajustarea procesului pentru a minimiza viitoare probleme generate de deviațiile observate la inspectare.

Acești trei piloni ghidează toate aspectele proiectelor Scrum, aspecte reprezentate schematic în figura 4.1.

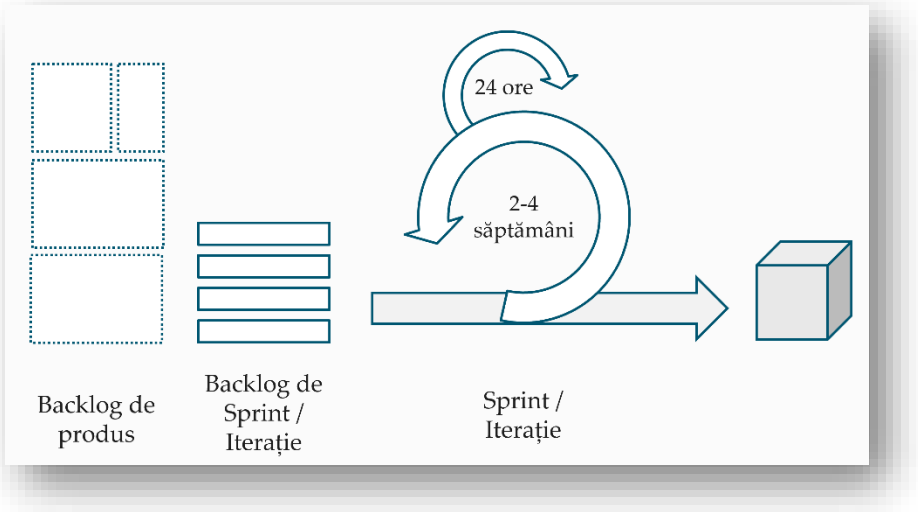

Figura 4.1. Reprezentare simplificată a principalelor ceremonii și artefacte Scrum

Roluri

În Scrum sunt definite doar trei rolul distincte: *echipa de dezvoltare*, *product owner* (proprietar produs) și *ScrumMaster*.

Echipa de dezvoltare este un grup de profesioniști ce realizează într-un mod incremental produsul final, în iterații succesive numite *sprint*-uri. Echipa de dezvoltare este multifuncțională și e formată din profesioniști ce acoperă toate rolurile necesare pentru executarea task-urilor proiectului. Aceștia se auto-organizează și sunt responsabili de gestionarea propriului timp de lucru.

Product Owner este responsabil cu maximizarea valorii produsului final prin organizarea (mai precis definire, clarificare, prioritizare etc.) a listei de activități ce urmează să fie executate (în Scrum această listă poartă numele de *product backlog*).

ScrumMaster-ul este responsabil de buna-înțelegere și implementare a practicilor Scrum. Persoana care îndeplinește acest rol va fi un lider asertiv ce se pune la dispoziția echipei de dezvoltare prin facilitarea evenimentelor ce au loc de-a lungul ciclului de viață a proiectului, în același timp eliminând sau coordonând eliminarea impedimentelor pe care membrii echipei de dezvoltare le întâmpină. De asemenea un *ScrumMaster* se va constitui într-un *coach* pentru cei care nu sunt familiarizați cu practicile și evenimentele Scrum. Nu în ultimul rând, *ScrumMaster*-ul îl va asista pe *product owner* în activitatea sa de organizare a *backlog*-ului, comunicând viziunea, scopul și descrierea elementelor din *backlog* echipei de dezvoltare.

Ceremonii (Practici)

Există 5 evenimente relevante care se petrec într-un proiect Scrum (evident, alături de dezvoltarea efectivă a produsului sau serviciului ce face obiectul proiectului): sprint, ședința de planificare a sprintului, ședința Scrum zilnică, recenzia sprintului și retrospectiva sprintului.

Un *sprint* reprezintă o perioadă de timp limitată (iterație) necesară pentru a realiza un produs ce poate fi utilizat (nu este vorba despre un produs final, ci de o variantă intermediară a acestuia ce poate fi testată și verificată). Această perioadă este fixă și în general are între 1 și 4 săptămâni. De-a lungul unui sprint se petrec toate celelalte 4 evenimente amintite anterior + dezvoltarea produsului.

Putem privi un sprint ca un mini-proiect deoarece are un scop bine definit, care nu se schimbă pe parcursul sprintului.

Ședința de planificare a sprintului are loc la începutul unui sprint și are ca scop determinarea scopului sprintului, a elementelor ce vor fi livrate la final de sprint și clarificarea modului în care acestea vor fi implementate. La această participă toți cei care îndeplinesc cele 3 roluri descrise anterior, *product owner*-ul având misiunea de a selecta din *backlog*-ul produsului

elementele cele mai importante pentru el în acel moment și de a explica și clarifica membrilor echipei de dezvoltare conținutul fiecărui element. Echipa de dezvoltare estimează câte și care dintre elementele prezentate vor putea fi dezvoltate în sprintul curent (ținând evident cont și de prioritățile specificate de *product owner*). Această estimare este făcută în funcție de capacitatea echipei și de nivelul anterior de performanță. Toate elementele selectate pentru a fi dezvoltate în sprint vor constitui așa-numitul *sprint backlog* și vor reprezenta scopul sprintului. În finalul ședinței echipa de dezvoltare pune la punct ultimele detalii legate de cum va implementa funcționalitățile și cum se va organiza astfel încât să fie îndeplinit scopul sprintului.

Ședința zilnică de Scrum (numită *Daily Scrum* sau *stand-up meeting*) este o ședință care are loc de obicei la începutul fiecărei zile de lucru și care durează 15 minute. Este recomandat ca ședința să înceapă de fiecare dată la aceeași oră și să se desfășoare în același loc. În această scurtă ședință membrii echipei se informează reciproc de ultimele aspecte relevante apărute în proiect, fiecare dintre ei răspunzând pe rând la trei întrebări:

- Ce a realizat de la ultima ședință?
- Ce urmează să realizeze până la următoarea ședință?
- Ce obstacole sunt?

În ședința zilnică de Scrum se poate aprecia progresul în vederea atingerii obiectivelor sprintului. ScrumMaster-ul va modera și facilita această ședință și ulterior va acționa pentru îndepărtarea obstacolelor identificate.

Recenzia sprintului (*sprint review*) are loc la finalul de sprint și are rolul de a permite *product owner*-ului și altor părți interesate de proiect să inspecteze funcționalitățile implementate de echipa de dezvoltare de-a lungul sprintului. Echipa de dezvoltare facilitează această sesiune iar *product owner*-ul este cel care decide dacă funcționalitățile identificate spre a fi implementate la ședința de planificare au fost finalizate sau nu. La finalul ședinței *product owner*-ul și echipa de dezvoltare identifică elementele de funcționalitate ce vor fi abordate în următorul sprint (și care urmează să fie discutate în detaliu și estimate în ședința de planificare următoare).

Ultimul eveniment important ce are loc într-un sprint este *retrospectiva sprintului* unde întreaga echipă reflectează asupra întregului mod de lucru și caută eventuale căi de îmbunătățire a acestuia. Echipa de proiect se va

concentra în această ședință pe oameni, relații, procese și instrumente însă vor fi luate în discuție doar elementele care vor putea fi îmbunătățite direct de către echipă. La finalul ședinței se vor identifica 1-2 acțiuni de îmbunătățire și persoanele responsabile de acestea în următorul sprint. Durata retrospectivei este de 1-2 ore (aproximativ 30 de minute pentru fiecare săptămână de sprint).

Artefacte (Instrumente)

Sunt trei instrumente importante utilizate în Scrum, fiecare dintre ele fiind deja menționate în descrierea evenimentelor: *backlog*-ul produsului, *backlog*-ul sprintului și definiția terminării (*definition of done*). În echipele de proiecte ce utilizează Scrum toate aceste instrumente sunt referite folosindu-se denumirea lor în limba engleză. Astfel în cadrul comunității IT și nu numai, utilizarea de termeni echivalenți în limba română îngreunează comunicarea pe aceste aspecte, termenii originali fiind foarte bine asimilați.

Backlog-ul de produs este o listă ordonată ce conține tot ceea ce este important pentru îndeplinirea scopului proiectului și a mai rămas de realizat, listă ce reprezintă unica sursă de descriere a cerințelor proiectului. Această listă este actualizată în permanență în funcție de nevoile *product owner*-ului dar și a rezultatului analizei celorlalte elemente din listă. În principiu, elementele ce se află în capul listei au o descriere mult mai detaliată, ceea ce permite echipei de proiect să le estimeze cu mai multă precizie. În timpul planificării sprintului se mai pot adăuga detalii sau schimba priorități tuturor elementelor din *product backlog*. (sunt variante ale Scrum-ului ce propun planificarea unui eveniment special pentru detalierea si prioritizarea acestui *backlog*, eveniment ce poartă numele de *sesiune de grooming*).

Cele mai importante elemente din *product backlog* sunt transferate în *sprint backlog*, care reprezintă lista tuturor funcționalităților și/sau livrabilelor ce urmează să fie dezvoltate în sprintul curent. După terminarea ședinței de planificare această listă va mai putea fi modificată doar de către echipa de dezvoltare, *product owner*-ul nemaiputând interveni decât cu clarificări până la finalul sprintului.

Definiția terminării reprezintă un set de reguli agreat de toți cei implicați în proiect și care precizează în detaliu când se poate considera un element din *backlog* ca fiind terminat. Este foarte important ca această definiție să fie enunțată la începutul proiectului și să fie transparentă pentru toți actorii implicați în proiect.

5. *Extreme Programming*: valori, principii și practici

Extreme Programming (XP) este o metodologie concepută exclusiv pentru dezvoltarea de aplicații software având ca principal scop creșterea calității *software*-ului și adaptarea rapidă la schimbările cerințelor impuse de clienți. Similar cu Scrum, dezvoltarea produselor se face incremental, în ciclurile scurte, și iterativ.

XP a generat un interes semnificativ în rândul comunităților IT la sfârșitul anilor 1990 și începutul anilor 2000 , fiind adoptat într-o mare diversitate de variante, unele dintre ele mult diferite față de conceptul original.

Denumirea metodologiei are la bază ideea că practicile considerate benefice pentru dezvoltarea de software de calitate trebuie duse la extrem pentru obținerea de beneficii maxime. De exemplu, evaluarea codului sursă este privită ca o practică benefică pentru creșterea calității softului. În XP această practică este dusă la extrem, codul fiind evaluat în mod continuu.

Inițiatorul XP este Kent Beck, unul dintre cei 17 care au enunțat manifestul Agile și care în 1999 a publicat cartea „*Extreme Programming Explained*" ce descria o serie de practici de dezvoltare a softului privite atunci ca fiind nonconformiste. Ulterior a revizuit aceste practici, clasificându-le în primare și secundare, în a doua ediție a cărții apărute în 2004. În continuarea acestei secțiuni vor fi descrise valorile, principiile și cele mai importante practici ale XP, toate acestea fiind dezvoltate pornind de la o paradigmă foarte simplă și puternică în același timp: „*Fii vigilent. Adaptează. Schimbă.*" Paradigmă enunțată de Kent Beck încă de la începutul cărții sale.

Valori XP

XP a avut la bază patru valori: comunicarea, simplitatea, *feedback*-ul și curajul. A cincea valoare, respectul, a fost adăugată de către Kent Beck ulterior.

Comunicarea

Construirea unui produs în general și a aplicațiilor software în special se bazează pe comunicarea clară și transparentă a cerințelor către echipa de proiect. În metodologiile tradiționale de management al proiectelor, acest lucru este realizat prin intermediul documentației. Tehnicile XP pot fi privite ca metode de a construi rapid și de a separa cunoștințele între membrii unei echipe de dezvoltare, scopul fiind acela de a oferi întregii echipe o viziune comună a sistemului, care se potrivește cu punctele de vedere deținute de către utilizatorii finali ai sistemului. În acest scop, XP favorizează comunicarea constantă, utilizarea de metafore/concepte comune, colaborarea între clienți/utilizatori și programatori, comunicarea verbală frecventă și *feedback*-ul continuu. În plus XP presupune că echipa este colocată, și toți membri își desfășoară activitatea în aceeași încăpere, într-un spațiu deschis, elemente ce facilitează comunicarea osmotică (ascultarea pasivă a discuțiilor legate de proiect purtate de ceilalți colegi în timp ce lucrezi).

Simplitatea

XP încurajează rezolvarea oricărei probleme în proiect prin identificarea celei mai simple soluții. Unul dintre promotorii XP, Stefan Roock, spunea la un moment dat că: „*Soluțiile complexe sunt greșite chiar dacă sunt corecte*", paradox ce descrie foarte bine conceptul de *simplitate* așa cum este el înțeles și aplicat în XP. Diferența dintre această abordare și metodele de dezvoltare tradiționale constă în concentrarea pe proiectarea și implementarea cerințelor de astăzi, și nu a celor de mâine, săptămâna viitoare sau luna viitoare. Cerințele viitoare nu trebuie să influențeze modul în care sunt implementate cerințele curente. Dezavantajul acestei abordări este acela că, ulterior este nevoie de un efort mai mare pentru a face modificările corespunzătoare cerințelor ce nu au fost luate în considerare la proiectarea inițială. Totuși, dinamica modificării cerințelor proiectelor software face ca avantajele abordării să compenseze acest dezavantaj și să se prevină astfel dezvoltarea unor cerințe ce ulterior sunt

modificate sau chiar eliminate din specificații. Codarea și proiectarea cerințelor viitoare incerte implică riscul de a cheltui efort pe ceva ce nu este necesar, slăbindu-se concentrarea pe finalizarea caracteristicilor esențiale.

În plus simplitatea facilitează comunicarea: un design simplu și un cod sursă simplu este ușor de înțeles de către cei mai mulți programatori din echipă.

Feedback-ul

În XP obținerea *feedback*-ului este privită din mai multe perspective.

***Feedback*-ul de la sistem**: prin programarea de teste de modul (*unit tests*) sau rularea frecventă de teste de integrare periodice, programatorii primesc *feedback* direct despre starea sistemului aproape imediat după implementarea schimbărilor.

***Feedback*-ul de la client**: prin utilizarea de teste funcționale (teste de acceptare), care sunt scrise de către client și testeri, echipa de dezvoltare obține informații concret despre starea actuală a sistemului lor. Astfel de revizuiri sunt planificate o dată la două sau trei săptămâni.

***Feedback*-ul de la echipă**: estimarea efortului necesar pentru implementarea cerințelor este transmisă clienților în scurt timp, aceștia înțelegând cu ușurință impactul modificărilor asupra întregului proiect..

Feedback-ul este strâns legat de comunicare și simplitate. XP insistă ca feedback-ul, indiferent de perspectivă, să fie dat frecvent și continuu pentru a se elimina orice defect într-o fază incipientă. Kent Beck spunea în cartea sa că optimismul este unul dintre cele mai mari pericole cu care se confruntă proiectele software iar *feedback*-ul este tratamentul potrivit.

Curajul

Această valoare este cea care, pusă în practică, ne ajută să respectăm și celelalte valori enunțate anterior. Este nevoie de curaj pentru a proiecta și implementa cerințele curente fără a fi influențat de cerințele „de mâine”. Acceptarea revizuirii și restructurării codului în urma feedback-ului primit necesită de asemenea curaj. Un alt exemplu de curaj este acela de a accepta eliminarea anumitor fragmente de cod care sunt învechite, indiferent de efortul cheltuit pentru a le crea. Curajul de a persista este de asemenea foarte important (este un fapt comun pentru un programator să fie blocat pe o problemă complexă o perioada lungă de timp, iar apoi să

descopere soluția brusc a în ziua următoare – fără persistență acest lucru nu ar fi posibil).

Respectul

Aceasta valoare include atât respectul pentru ceilalți cât și respectul de sine. Programatorii nu ar trebui să se angajeze în modificări ce introduc defecte în sistem, care fac testele de modul să eșueze, sau care întârzie activitatea colegilor lor. Membrii echipei respectă propria lor muncă luptând întotdeauna pentru calitate și căutând cel mai bun design pentru soluția cea mai simplă.

Urmarea celor patru valori anterioare conduce la câștigarea respectului colegilor de echipă. Nici o persoană din echipă nu ar trebui să se simtă neapreciată sau ignorată. Acest lucru asigură un nivel ridicat de motivație și încurajează loialitatea față de echipă și de scopul proiectului.

Principii

Principiile care stau la baza XP se bazează pe valorile descrise mai sus și sunt destinate susținerii deciziilor luate într-un proiect de dezvoltare software. Principiile sunt mai concrete decât valorile și mai ușor de tradus într-o situație practică.

Feedback rapid

XP consideră *feedback*-ul fiind util doar dacă se face în mod frecvent și cu promptitudine. Eliminarea oricărei întârzieri, oricât de mici, între o acțiune și feedback este esențială pentru o învățare rapidă și aplicare eficientă a modificărilor necesare. Spre deosebire de metodele tradiționale de dezvoltare *software*, contactele cu clientul sunt mai frecvente în iterații succesive. Clientul are o perspectivă clară asupra sistemului dezvoltat și poate oferi din timp un feedback pentru a orienta dezvoltarea după cum consideră necesar. În condițiile unui *feedback* frecvent obținut de la client, o decizie de design greșită va fi observată și corectată rapid, cu un efort mai mic înainte ca această decizie să influențeze designul altor componente.

Testele de modul, de asemenea, contribuie la un feedback rapid. Automatizarea acestor teste face ca o bună parte din defecte să fie

descoperite în timp scrierii codului, lucru ce influențează pozitiv calitatea aplicației dezvoltate.

Asumarea simplității

Acest principiu presupune abordarea fiecărei probleme având în minte găsirea celei mai simple soluții posibile. Metodele tradiționale de dezvoltare software pun pe primul plan planificarea pentru viitor și codificarea pentru reutilizare însă XP respinge vehement aceste idei considerându-le ca fiind principala sursă a eșecului proiectelor software.

Ca urmare a aplicării principiului de asumare a simplității au fost create o serie de sub-principii care susțin simplitatea cum ar fi:

- principiul DRY („*Don't Repeat Yourself*") se referă la eliminarea redundanței în implementarea codului;
- principiul YAGNI („*You Aren't Going to Need It*") care se referă la punerea sub semnul întrebării a necesității fiecărui element al aplicației;
- principiul „*Fă cel mai simplu lucru ce ar putea funcționa*".

Modificare incrementală

Principiul modificării incrementale pornește de la ideea că schimbările de anvergură aplicate unui sistem software poate afecta calitatea acestuia. De aceea dezvoltarea aplicațiilor software se realizează în pași succesivi, fiecare pas încheindu-se cu colectarea de feedback de la client și utilizatorii finali. XP sugerează aplicarea de schimbări incrementale șî dezvoltarea de versiuni intermediare la fiecare una-trei săptămâni. Atunci când se fac mulți pași mici, clientul va aveaun control sporit asupra procesului de dezvoltare și a rezultatului final.

Figura 5.1 prezintă cele 7 cicluri importante de planificare și obținere a feedback-ului în XP.

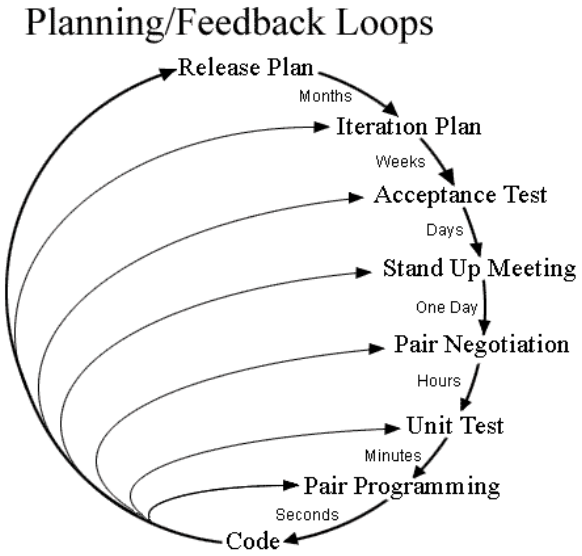

Figura 5.1. Cicluri de planificare și obținere a feedback-ului în XP

Îmbrățișarea schimbării

Acest principiu nu se referă la încurajarea schimbărilor atunci când ele nu sunt necesare, ci la acceptarea cerințelor de schimbare (din partea clientului) ca fiind ceva firesc și parte a procesului de dezvoltare produsului final. Una din strategiile utilizate în XP este aceea de a găsi soluții la cerințele curente care să fie deschise la mai multe opțiuni de abordare ulterioară

Muncă de calitate

Acest principiu este în majoritatea metodologiilor, Agile sau tradiționale, de la sine înțeles. În XP el este enunțat explicit deoarece apare pericolul ca unele dintre principiile anterioare (ca asumarea simplității sau îmbrățișarea schimbării) să încurajeze dezvoltarea unui produs cu o calitate precară. Pe de altă parte sunt aduse aici în discuție aspecte legate de mediul de lucru și atmosfera în echipă ce contribuie hotărâtor la creșterea satisfacției legată de munca depusă și, implicit, de calitatea acesteia.

Practici XP

XP propune un număr de 12 practici importante ce sunt derivate din respectarea valorilor și principiilor enunțate anterior:

1. *Jocul planificării*: Planificarea în XP se bazează pe identificarea unor descrieri scurte a modului în care clientul dorește să folosească aplicația finală, alături de prioritizarea și estimarea lor. Aceste descrieri poartă numele de *user stories* și sunt definite împreună cu clientul pe baza caracteristicilor dorite de el în aplicație. Acestea trebuie să fie păstrate scurte, dar cu suficiente detalii pentru a oferi o înțelegere a cererii. Echipa de proiect va folosi aceste *user stories* pentru estimarea costurilor și managementul proiectului dar și ca unitate de urmărire a progresului în proiect.

2. *Release-uri mici*: Dezvoltarea și livrarea aplicației se realizează într-o serie de versiuni intermediare, actualizate frecvent. La fiecare iterație se adaugă într-o versiune nouă cerințele discutate împreună cu clientul, se testează și apoi această versiune este transmisă clientului pentru validare și feedback.

3. *Metafore*: această practică se referă la formarea unui limbaj comun, de termeni specifici proiectului, folosit de echipa de dezvoltare și toate celelalte persoane implicate în proiect (*stakeholders*). „Metaforele" sunt deprinse de fiecare persoană la începutul proiectului, atunci când clientul încearcă să înțeleagă termenii tehnici folosiți de echipa de dezvoltare, iar membrii echipei descifrează principalele concepte ale domeniului de *business* al clientului. Angajarea în acest sistem de nume ar trebui să permită înțelegerea intuitivă a fiecărui element.

4. *Proprietate colectivă*: Nicio persoană nu deține exclusiv proprietatea codului pe care îl scrie. Acest cod este proprietatea întregii echipe de dezvoltare și poate fi revizuit și actualizat de către orice altă persoană din echipă. Ca o consecință imediată, fiecare persoană este responsabilă în orice moment de întreg codul sursă scris de echipă. Aplicarea acestei practici trebuie să fie făcută în corespondență cu aplicarea practicilor testării, a standardelor de codare și a programării în pereche, fără de care totul ar putea degenera într-un haos.

5. *Standard de codare*: Toți membrii echipei scriu cod în același mod, folosind aceleași stiluri și formatare. Acest lucru permite partajarea rapidă de cod și reduce curba de învățare pentru alți dezvoltatori.

6. *Design simplu*: Cel mai bun design este designul cel mai simplu care funcționează. Un design corect din punct de vedere al XP este cel care trece toate testele de modul și testele funcționale, întrunește așteptările clientului în ceea ce privește valoarea de *business* adusă, și nu conține redundanțe. În același timp, deciziile importante de proiectare sunt luate în cel mai târziu moment posibil astfel încât să nu fie eliminate prematur anumite căi de extindere a aplicației.

7. *Refactorizarea*: Refactorizarea este procesul de revizuire și îmbunătățire a codului (prin simplificare sau reproiectare) fără a se modifica funcționalitățile existente și fără a se adăuga funcționalități noi în aplicație. Fiecare membru al echipei trebuie să înțeleagă cerințele funcționale și să depună un efort constant pentru adaptarea și îmbunătățirea corespunzătoare a codului.

8. *Testarea*: Fiecare versiune intermediară a produsului trebuie să fie testată temeinic înainte de a fi transferată spre validare clientului. Modul de testare propus de XP este unul special deoarece testele sunt codificate înainte de activitatea de dezvoltare a funcționalității ce urmează să fie testate, funcționalitatea fiind implementată astfel încât să treacă testul (practica poartă numele de *Test Driven Development – TDD*). Acest lucru permite dezvoltarea unei aplicații cu un nivel calitativ ridicat prin identificarea defectelor înainte ca acestea să „se piardă" într-o aplicație de dimensiuni mari.

9. *Programarea în pereche* (*pair programming*): În XP codul sursă este dezvoltat de o pereche de programatori care lucrează împreună la o singur calculator. Scopul principal al implementării unui astfel de practici este producerea unui cod de calitate ridicată la același cost sau la unul mai mic decât în cazul programării clasice.

10. *Integrarea continuă*: Codul scris de echipa de programare este integrat într-o aplicație executabilă de mai multe ori pe zi. În acest fel toți membri echipei vor fi la curent cu cele mai recente schimbări implementate.

11. *40 de ore de lucru pe săptămână*: Pentru ca practicile XP să fie eficiente, dezvoltatorii trebuie să fie odihniți. Experiența a arătat că dezvoltatorii obosiți fac mai multe greșeli și sunt mai expuși la extenuare rezultând cod de calitate mai scăzută.

12. *Client colocat*: Una dintre cele mai importante practici ale XP este aceea de a menține clientul ca parte integrantă a echipei și efortului de

dezvoltare. Clientul trebuie să fie disponibil în orice moment pentru a stabili priorități, pentru a stabili și clarifica cerințe și pentru a răspunde la întrebări.

6. Lean Software Development

Conceptul *Lean* își are originile la începutul secolului XIX când a apărut nevoia producției în masă și reprezintă o practică de producție și manufacturare ce consideră pierdere orice utilizare a resurselor pentru altceva decât a produce ***valoare pentru utilizatorul final***. Lean definește ca *valoare* orice acțiune sau proces pentru care clientul este de acord/dorește să plătească.

Scopul principal al *Lean* este acela de a elimina toate pierderile ce apar într-un proces de producție iar atenția este orientată către minimizarea efortului pentru obținerea aceleași valori. Căile prin care Lean propune atingerea scopului sunt:

- optimizarea fluxului,
- creșterea eficienței,
- diminuarea pierderilor,
- utilizarea de metode empirice pentru a decide ceea ce contează, provocând concepțiile preexistente despre cum trebuie realizat un anumit lucru.

Practicile *Lean* au fost implementate cu succes de către Toyota în anii 1980 la inițiativa unui inginer de producție pe nume Taiichi Ohno. Acesta nu a vrut să continue procesul de producție în masă a mașinilor Toyota, ci a conceput un sistem prin care un produs era creat și livrat imediat după ce un client a făcut o comandă. Credința lui fermă era că pierderile sunt mult mai mici dacă se așteaptă cererea de livrare a următoarei comenzi decât dacă se producea în avans un stoc considerabil de produse ce urmau să anticipeze și să satisfacă posibile comenzi viitoare. În același timp credea în posibilitatea de a produce și livra un nou produs imediat. Toate aceste idei s-au concretizat în ceea ce s-a numit ulterior *Sistemul de Producție*

Toyota și care a devenit în scurt timp un model de urmat de către alte companii de mașini și nu numai.

Mult mai târziu acest model a fost adaptat și pentru industria software care, la rândul său, trece printr-o etapă similară producției de masă cunoscută sub numele de *offshore outsourcing*.

Principii *Lean*

În această secțiune vor fi prezentate cele 5 principii ale Lean așa cum au fost ele descrise în [8] pentru a ghida managerii în demersul lor de introducere a filozofiei Lean în producție:

1. ***Definirea valorii***: presupune specificarea valorii pentru fiecare categorie de produse, din punctul de vedere al clientului final.

2. ***Maparea secvenței de valoare***: constă în identificarea tuturor activităților aflate pe secvența de valoare (*value stream*) pentru fiecare categorie de produse, eliminând pe cât posibil acele activități care generează pierderi.

3. ***Crearea fluxului***: se concretizează prin ordonarea activităților generatoare de valoare într-un flux de pași clar identificați, astfel încât produsul să fie livrat clientului final parcurgând o secvență de activități continue, fără multe întreruperi, opriri sau așteptări intermediare.

4. ***Pornire sistem „pull"***: crearea secvenței de activități generatoare de valoare se finalizează cu activarea unui sistem de tip „pull" prin care orice client poate obține un produs „trăgându-l" din amonte, pe fluxul de producție

5. ***Perfecționarea fluxului***: are loc după ce valoarea a fost identificată și definită, activitățile generatoare de valoare au fost determinate iar cele generatoare de pierderi au fost eliminate, fluxul de valoare s-a determinat și implementat. Activitatea de perfecționare continuă până când se atinge un nivel optim, în care valoarea adăugată este maximă iar majoritatea pierderilor au fost eliminate.

Principii *Lean* pentru dezvoltarea de software

Principiile descrise anterior au fost adaptate și extinse corespunzător contextului proiectelor software. În [9] s-au identificat un număr de 7 principii care vor fi detaliate în cele ce urmează:

1. ***Eliminare pierderi***. După cum am văzut în secțiunea anterioară pierderea este dată de orice activitate care nu produce valoare pentru client. Există 7 categorii de pierderi în abordarea Lean:

- *Supraproducția* (fabricare de produse ce stau pe stoc sau procesare de informații ce nu sunt necesare)
- *Așteptare neproductivă* (lipsă instrumente sau informații la momentul necesar)
- *Transport inutil* (mutări sau transferuri inutile ale produsului, persoanelor sau informațiilor între diverse locații sau între diverse procese)
- *Procesare inutilă* (producere de calitate inutilă, utilizarea de instrumente sau procese sofisticate atunci când cele simple erau suficiente, depășirea timpului alocat ședințelor etc)
- *Stocuri inutile* (menținerea îndelungată de stocuri de materiale, păstrarea unor funcționalități neterminate timp îndelungat, neutilizarea întregii capacități a echipei de dezvoltare)
- *Mișcări inutile* (lipsă ergonomie, neglijență în realizarea pașilor necesari pentru efectuarea unei activități)
- *Defecte* (orice activitate de corectare a greșelii este o pierdere)

În industria IT tot ceea ce se realizează pentru dezvoltarea unui produs și care nu este analiză sau implementare de cod este considerat pierdere. O modalitate eficientă de eliminare a pierderilor constă în identificarea celor mai importante 10-15 activități ce sunt realizate în cadrul echipei de proiect. Echipa va trebui sa evalueze toate aceste activități cu note între 1 și 5 în funcție de proporția în care activitățile respective sunt considerate importante/valoroase de către client. Se vor stabili acțiuni concrete pentru eliminarea acelor activități care au fost evaluate cu 1 sau 2 puncte.

2. ***Amplificarea învățării***. Acest principiu este specific proiectelor ce se bazează pe managementul cunoștințelor. Așa cum am văzut, aceste proiecte corespund într-o mare măsură domeniului ***Complex*** al *Cynefin Framework* și presupun utilizarea de practici emergente care să ducă la

descoperirea soluției finale. Principiul de amplificare a învățării este determinat de analiza diverselor versiuni intermediare obținute iterativ dar și de dorința de accelerare a procesului de descoperire a soluției.

3. ***Decide cât mai târziu posibil***. Acest principiu se regăsește și printre principiile XP și presupune amânarea cât mai mult posibil a deciziilor ireversibile. Această amânare persistă până se identifică alternativa cea mai bună sau atunci când costul neluării unei decizii depășește costul luării unei decizii mai puțin bune.

4.***Livrează cât mai devreme posibil***. Clienții apreciază livrările rapide. În plus, o livrare rapidă implică un timp mai scurt la dispoziție pentru client pentru a-și schimba percepția asupra cerințelor. Acest principiu este strâns legat de principiul precedent, deoarece cu cât livrarea este mai rapidă cu atât avem posibilitatea de a amâna mai mult anumite decizii (de exemplu, în cazul unei iterații de 3 săptămâni este nevoie să luăm decizii de proiectare la începutul iterației care să privească cerințele ce vor fi implementate până la finalul celei de-a treia săptămâni; în cazul unei iterații de o săptămână deciziile de proiectare pentru cerințele ce urmează să fie implementate în iterațiile ulterioare vor fi amânate cu cel puțin o săptămână.

5. ***Responsabilizează echipa***. Deoarece execuția într-un flux de valoare este rapidă iar deciziile se amână până la cel mai târziu moment posibil, membrii echipei trebuie să fie capabili să ia decizii rapide și să nu trebuiască să aștepte după direcții date de o autoritate centrală. Acest lucru implică faptul că managerii trebuie să fie în același timp și lideri abili care să seteze o direcție clară echipei, să se asigure că toți membrii echipei sunt aliniați înspre această direcție și că au și motivația necesară.

6. ***Demonstrarea integrității***. Un produs este integru dacă clientul sau utilizatorul final îl consideră perfect din perspectiva utilității. În ceea ce privește aplicațiile software, acestea sunt considerate integre nu doar dacă maximizează valoare pentru client dar și dacă sunt flexibile și extensibile fiind capabile să se adapteze cu ușurință la cerințe de dezvoltare ulterioare.

7. ***Viziunea de ansamblu***. Întotdeauna trebuie să se mențină un echilibru între forțele ce guvernează un proiect. Managerul proiectului trebuie să identifice care sunt efectele adoptării unei noi practici sau modificarea unei practici existente în toate economia proiectului. Optimizarea sau

eficientizarea unei activități particulare trebuie să se facă întotdeauna cu analiza efectelor secundare ale acestei acțiuni în restul proiectului.

7. Kanban

Kanban este cea mai populară implementare particulară a principiilor Lean prezentate în capitolul anterior. Ca și Lean, Kanban își are originile în *Sistemul de Producție Toyota* conceput de Taiichi Ohno și are ca principală caracteristică faptul că este un sistem **vizual** de control al producției și gestionare a stocurilor. Avantajul sistemului Kanban față de alte sisteme de gestiune a stocurilor este simplitatea și ușurința de folosire (figura 7.1).

Acest sistem utilizează carduri (de fapt *Kan-ban* provine din limba japoneză și înseamnă *card de semnalizare*), containere refolosibile sau spații/locuri goale pentru a facilita *„tragerea"* produselor din locurile de stocare sau furnizori în aria de producție (este de fapt o implementare de sistem *pull* ce a fost amintit printre principiile Lean în capitolul anterior). Scopul principal al Kanban este acela de a garanta 100% disponibilitatea materialelor și îmbunătățirea permanentă a nivelul stocurilor acestora.

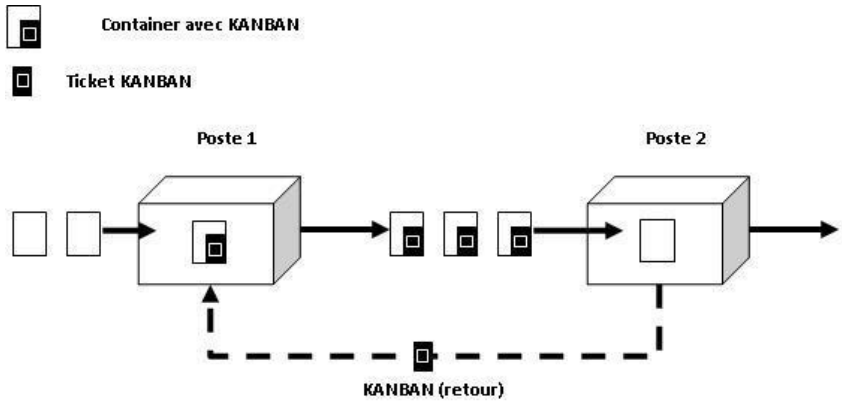

Figura 7.1. Fluxul cardurilor într-un sistem Kaban
(sursa http://www.cetice.u-psud.fr/aunege/gestion_flux/co/1_2_1.html)

Kanban controlează nivelul stocurilor în aria de producție și limitează producția la un nivel maxim admis pentru a evita supra-producția, ce constituie o categorie importantă de pierderi. În plus acest sistem contribuie permanent la îmbunătățirea calității produsului final, a timpului de execuție (numit în acest context *lead time*) și performanței livrărilor.

Modelul Kanban a fost utilizat pentru prima dată în dezvoltarea de software în 2004 de către o echipă din Microsoft. În 2010 David Anderson descrie în detaliu în [10] un proces de adopție a filozofiei Kanban într-un proiect software.

Diagrame de flux cumulativ

Pentru o mai bună înțelegere a modului în care utilizarea Kanban poate să ducă la limitarea pierderilor și creșterea calității produselor software este util să analizăm diagramele de flux cumulativ (*Cumultive Flow Diagram*) specifice acestor proiecte.

Diagrama de flux cumulativ (figura 7.2) este un instrument ce permite echipei de dezvoltare să vizualizeze efortul și progresul unui proiect. În același timp, este și principalul instrument de predicție, anticipând a unui impediment care e pe cale să aibă loc în cadrul procesului. Acest lucru este vizibil ori de câte ori panta graficului nu este lin crescătoare ci prezintă creșteri sau coborâri bruște.

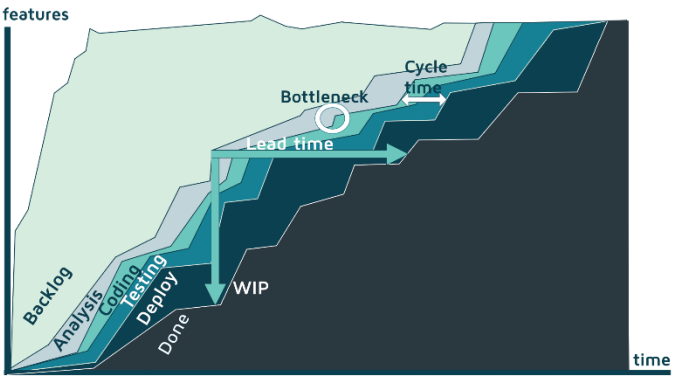

Figura 7.2. Diagrama de flux cumulativ al unui proiect software
(variantă pe baza diagramei din https://blog.avarteq.com/)

Modul de construcție a Diagramei de flux cumulativ este foarte simplu: axa verticală reprezintă un număr de funcționalități sau sarcini iar axa verticală reprezintă timpul scurs de la începutul proiectului. Curbele trasate într-un astfel de grafic reprezintă practic numărul de funcționalități ce se află într-o anumită stare (dependentă de proiect) prezentate într-o perspectivă temporală. Diferența față de alte grafice similare este aceea că informațiile sunt prezentate cumulativ (adică graficul nu va reprezenta câte taskuri se află în testare la un moment dat, ci toate taskurile care au fost în testare până la acel moment).

În figura 7.2 este prezentat un exemplu de Diagramă de flux cumulativ corespunzător unui proiect care consideră că o anumită funcționalitate se poate afla la un moment dat într-una din următoarele stări:

- inserat în *backlog* (*Backlog*),
- în faza de analiză (*Analysis*),
- în faza de dezvoltare (*Coding*),
- în faza de testare (*Testing*),
- în faza de instalare pe serverul clientului (*Deploy*),
- finalizat (*Done*).

Evident, zona reprezentând funcționalitățile finalizate este în continuă creștere. Funcționalitățile care sunt în curs de execuție (fie că sunt analizate, dezvoltate, testate sau în curs de instalare) vor fi reprezentate de diferența dintre începutul zonei de analiză și începutul zonei *Done* pe verticală. Aceasta este munca în desfășurare (notată pe grafic ca WIP – *Work In Progress*). Diferența între aceleași zone, de data aceasta calculată pe orizontală, ne va da timpul mediu petrecut de către o funcționalitate din momentul în care este extrasă din *Backlog* până în momentul în care ea este finalizată. Pe grafic această diferență este etichetată cu *Lead Time.*

În [10] David Anderson arată, luând ca studiu de caz Diagramele de flux cumulativ corespunzătoare a două echipe de proiect din Motorola, că o calitate precară a produsului final este asociată unui *lead time* generos. În același timp *lead time*-ul este influențat de numărul de funcționalități care se află în execuție la un moment dat. Concluzia acestei analize este aceea că reducerea sau limitarea funcționalităților aflate în execuție (*Work In Progress*) atrage după sine o calitate ridicată a produsului final.

Așa cum am văzut la începutul acestui capitol, Kanban este un instrument vizual util în controlul producției și gestionarea stocurilor, iar *Work In Progress* nu este altceva decât un stoc al funcționalităților aflate în execuție. Prin urmare, Kanban ar putea fi utilizat ca instrument vizual de limitare a funcționalităților aflate în lucru cu scopul îmbunătățirii calității produsului final.

Implementări Kanban în proiecte IT

În implementarea Kanban în proiecte IT un rol central îl ocupă *Kanban board*. Aceste este un instrument vizual, care grupează funcționalitățile în raport cu starea în care se află acestea. Un exemplu de stări posibile prin care poate trece o funcționalitate a fost dat la prezentarea Diagramei de flux cumulativ în secțiunea precedentă, însă aceste stări pot diferi de la un proiect la altul ele fiind definite în funcție de complexitatea proiectului și de numărul etapelor existente în procesul de dezvoltare. Funcționalitățile sunt scrise pe carduri (în practică se folosesc *post-it*-uri) colorate pentru a fi înțelese și procesate mai ușor. Utilizarea *Kanban board*-ului (figura 7.3) are două avantaje majore:

- membrii echipei pot vizualiza în orice moment starea în care se află proiectul,
- vizualizarea stării globale ajută în descoperirea rapidă a problemelor și/sau blocajelor în proiect

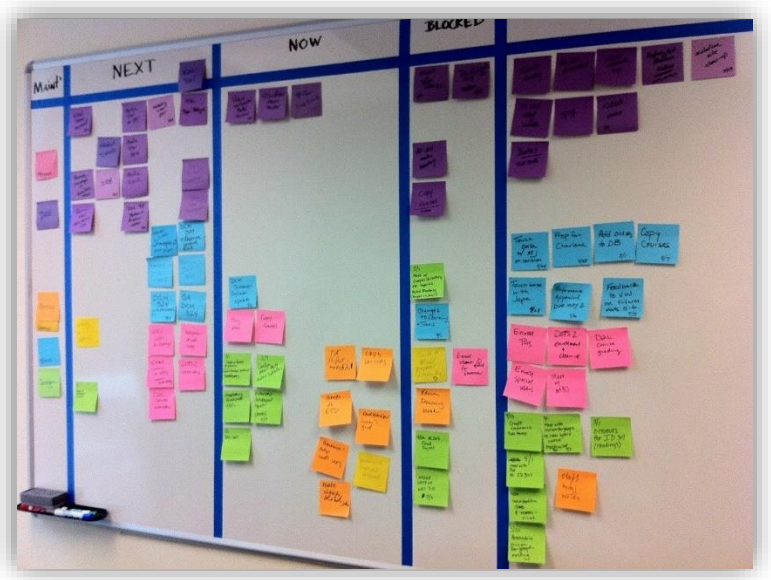

Figura 7.3. Exemplu de Kanban board
(sursa: http://blog.psoda.com/)

Structura de bază a unui *Kanban board* e compusă din trei părți distincte:

- ***De executat*** (ex. *Backlog*),
- ***În execuție*** (ex. *Analysis*, *Coding*, *Testing* , *Deploy*),
- ***Finalizate*** (ex. *Done*)

Pentru stările ce aparțin de partea de funcționalități aflate în execuție se pot defini limite . Aceste limite ajută la o concentrare mai bună și la creșterea performanțelor. Coloanele de tip "***În execuție***" din *Kanban board* au atribuite o astfel de limită, iar numărul de funcționalități procesate la un moment dat nu trebuie să depășească limita specificată. Prin reducerea *multitasking* -ului se reduce sau chiar timpul necesar pentru alternare (una dintre categoriile importante de pierderi). Prin executarea task -urilor în mod secvențial, rezultatele apar mai repede și timpul total de execuție a acestora este mai scurt.

Timpul și volumul de muncă în Kanban se măsoară în funcție de mai multe metrici. Două dintre aceste metrici au fost discutate și la prezentarea Diagramei de flux cumulativ:

- *Work In Progress* (*WIP*) care reprezintă numărul de funcționalități aflate în execuție

- *Lead time* - timpul măsurat de la intrarea unei funcționalități în zona „***În execuție***" până la momentul în care aceasta este finalizată. Este important de reținut că *lead time* măsoară timpul și nu efortul depus pentru dezvoltarea unei funcționalități și că este una dintre metricile relevante pentru client care va evalua performanța de lucru a echipei în funcție de această metrică.

O a treia metrică importantă este timpul de ciclare (*cycle time*) care reprezintă timpul măsurat de la momentul în care s-a început codificarea unei funcționalități până în momentul în care acesta este gata de instalare (vezi figura 7.2). În comparație cu *lead time*, aceasta este o măsură mecanică a capacității procesului și reflectă eficiența de muncă a echipei.

Prin urmare, pentru a mări performanța unei echipei vor trebui diminuate metricile *lead time* și *cycle time*.

Este evident că un *Kanban board* este complet configurabil în funcție de scopul pe care îl urmărim și în funcție de etapele procesului prin care se ajunge la produsul sau serviciul final, aceasta fiind o calitate ce face ca tehnica în sine să poată fi utilizată într-o varietate de domenii.

Existența unui *Kanban board* fizic, sub forma unei table pe care sunt desenate culoare corespunzătoare stărilor și sunt atașate carduri corespunzătoare funcționalităților, este o practică foarte populară și e considerat un prilej excelent de îmbunătățire a colaborării și comunicării între membrii echipei.

Cu toate acestea, interesul crescut în folosirea metodologiei Kanban a inspirat o serie de programe și instrumente online ce expun funcționalitățile similare unui *Kanban board* fizic. În plus, acestea au o serie de procedee și de avantaje adiționale cum ar fi: manipulare și actualizare simplă, arhivarea funcționalităților finalizate, editare, clasificare, temporizare, accesare la distanță, facilitatea colaborării între echipe distribuite etc.

Implementare Kanban

În afară de IT, sunt o serie de alte domenii bazate pe managementul cunoștințelor în care metodologia Kanban se poate integra cu ușurință:

- Marketing și PR,
- Resurse Umane,
- Logistică,
- Financiar,
- Legal/ Juridic.

Beneficiile pe termen scurt și lung sunt similare celor deja identificate în domeniul IT: vizibilitate mai bună a procesului de lucru, productivitate crescută și colaborare eficientă în echipă.

Alte avantaje ale implementării Kanban includ :

- adoptarea procesului prin modificări incrementale, fără a se renunța la metodologia utilizată curent de către echipă;
- adoptarea prin aplicarea unor schimbări cu risc politic (intern) redus;
- schimbările ce trebuie impuse întâmpină în general o rezistență minimă din partea echipei
- dezvăluie oportunități de îmbunătățire ce nu implică schimbări importante în aplicarea abilităților tehnice.

Trebuie reținut însă că schimbările de adoptare a unui sistem Kanban pot necesita un timp mai lung pentru a avea un efect deplin!

8. Alte metodologii Agile

După cum am arătat și în capitolul al treilea, există o mulțime semnificativă de metodologii Agile dezvoltate până în acest moment. Cele mai populare dintre ele (*Scrum*, *Extreme Programming*, *Lean Development*, *Kanban*) au fost tratate deja în capitole dedicate. În acest capitol vor fi trecute în revistă principalele caracteristici ale altor 4 metodologii care au reținut atenția unei părți din comunitatea Agile.

Feature Driven Development (FDD)

Metodologia FDD a fost dezvoltată de Jeff De Luca în 1997 și a fost influențată într-un mod hotărâtor de teoria modelării obiectuale a lui Peter Coad. FDD este exclusiv destinată dezvoltării de aplicații software și are ca punct central modelarea domeniului de business al produsului.

Cerințele sunt colectate și modelate într-o manieră ierarhică ce poate fi asemănată cu construcția unei structuri defalcate de lucrări (*Work Breakdown Structure – WBS*). Nivelele de modelare a cerințelor sunt:

- *Subject Area* (ce conține o descriere a practicilor de business generale)
- *Feature Set* (formate din mulțimi de activități de business)
- *Features* (activități distincte, funcționalități)

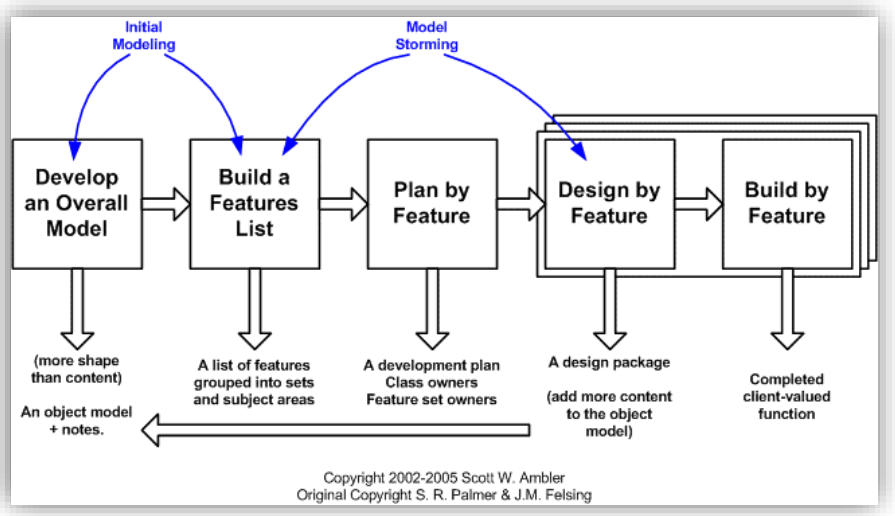

Figura 8.1. Ciclul de viață al proiectelor FDD
(sursa: http://www.agilemodeling.com/essays/fdd.htm)

Ultimul nivel, *Feature* (Funcționalitate), este foarte important deoarece toate activitățile ce se realizează la începutul unui proiect FDD sau în cadrul unei iterații FDD se referă la acest nivel: planificarea, monitorizarea și controlul, proiectarea și construcția se realizează la nivel de *Feature*.

Fiecare funcționalitate va avea un nume ce trebuie să urmeze șablonul:

< acțiune > < rezultat > [al | pe | pentru | din] < obiect >.

Un exemplu de funcționalitate este: *Calculează dobânda lunară pentru credit*.

FDD identifică o listă elaborată de roluri ce pot fi jucate în cadrul unui proiect, fiecare cu responsabilitățile sale bine definite. Aceste roluri sunt: manager de proiect, arhitect șef, manager de dezvoltare, programator șef, proprietar clasă/modul, expert de domeniu, tester, implementator, documentarist.

De remarcat proprietarul de clasă/modul care sugerează faptul că fiecare persoană este responsabil cu anumite părți (clase, module) de cod, lucru care intră în contradicție cu principiul proprietății colective a codului enunța de *Extreme Programming*.

Cel mai important instrument utilizat in FDD este raportul de progres pe mulțimi de funcționalități care are structura determinată direct de cele trei

nivele de modelare a cerințelor (figura 8.2). Starea exactă a fiecărei funcționalități în parte este documentată într-un tabel separat având 6 jaloane specifice: clarificarea domeniului, proiectare, inspectarea proiectării, codificare, inspectarea codificării, instalat.

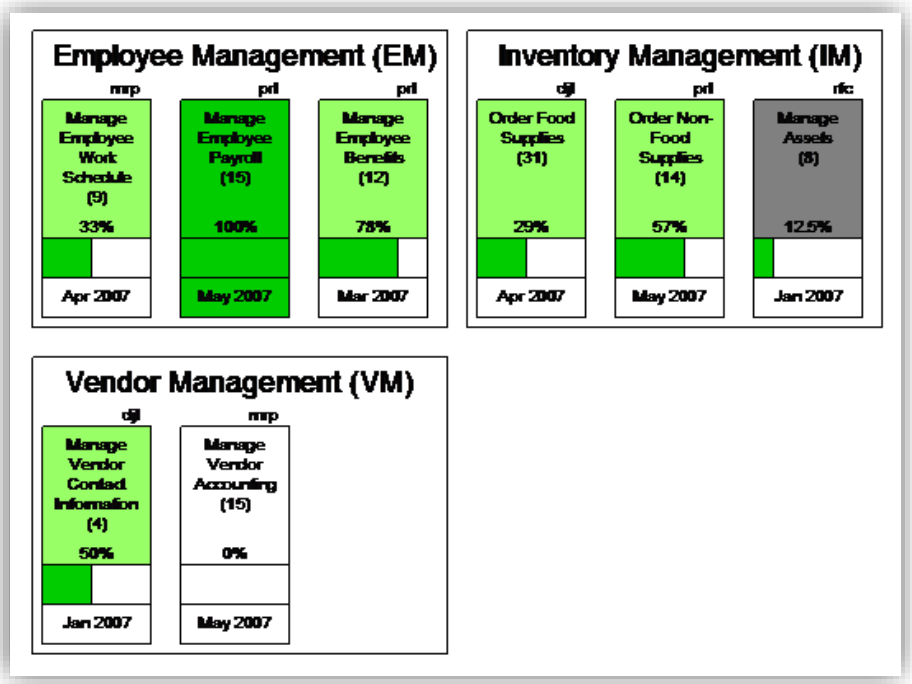

Figura 8.2. Raport de progres pe mulțimi de funcționalități
(sursa: http://accessplus.com.ph)

Deși practicile și rolurile FDD nu sunt 100% compatibile cu principiile Agile, FDD este considerată una dintre metodologiile importante și este recomandată proiectelor cu echipe mari sau chiar cu echipe multiple ce lucrează în paralel.

Agile Unified Process (AUP)

AUP este o metodologie bazată pe o documentație stufoasă. AUP este un descendent al Rational Unified Process, o metodologie de dezvoltare a proiectelor software ce era bazată pe metodologia tradițională Waterfall combinată cu o abordare iterativă și incrementală. AUP aduce în plus

integrarea mai multor concepte și tehnici Agile, dar contextul general rămâne totuși unul destul de rigid.

AUP privește un proiect ca fiind compus din 4 faze: inițierea, elaborarea, construcția și tranziția. De asemenea, echipa de proiect trebuie să își organizeze corespunzător în fiecare fază activități ce țin de 9 discipline diferite (vezi figura 8.3).

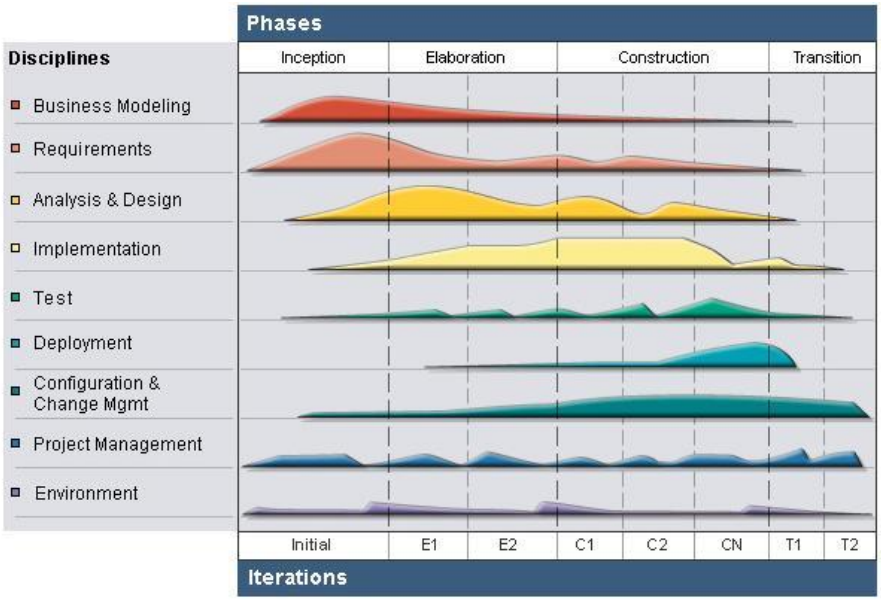

Figura 8.3. Fazele și disciplinele proiectelor AUP
(sursa: www.ibm.com)

AUP enunță și un set de 6 așa-numite filozofii ce trebuie să guverneze proiectele:

- *Competența* – Echipa își cunoaște bine rolul și obiectivele și este formată din profesioniști în cele 9 discipline ale proiectului
- *Simplitate* – Documentația este redactată concis, folosind un număr minim de pagini.
- *Agilitate* – Activitățile proiectului vor fi conforme cu valorile și principiile manifestului Agile.
- *Activitate* – Concentrare doar pe activitățile care au cea mai mare valoare pentru client și ignorarea „zgomotului" (elemente care distrag atenția de la ce este important pentru proiect)
- *Instrumente* – Instrumentele simple sunt cele mai potrivite.

- *Personalizare* – AUP funcționează cel mai bine atunci când practicile sale sunt personalizate pe baza nevoilor date de contextul proiectului.

Crystal

Familia de metodologii Crystal a fost concepută de Alistair Cockburn în 2004 și conține 20 de implementări diferite grupate în 5 categorii în funcție de dimensiunea echipei de proiect sau a criticității proiectului. Aceste 5 categorii sunt: Clear, Yellow, Orange, Red și Blue (figura 8.4).

Dezvoltarea prezintă șapte caracteristici: livrare frecventă, îmbunătățire reflexivă, comunicare osmotică, siguranță personală, concentrare, acces facil la utilizatorii experți și la cerințele pentru mediul tehnic.

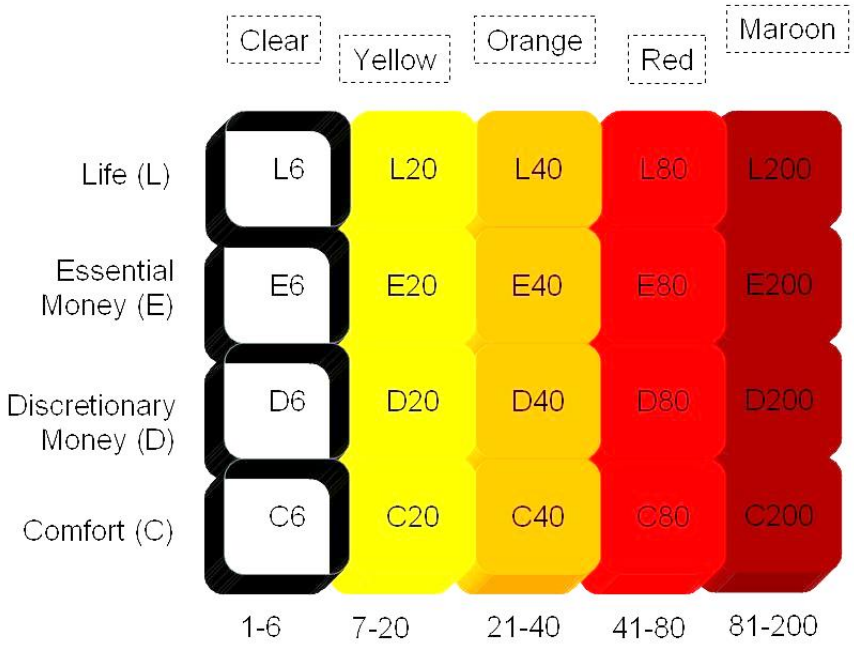

Figura 8.4. Familia de metodologii Crystal
(sursa: http://www.devx.com/architect/Article/32836/0/page/2)

Metoda "cea mai agilă", Crystal Clear, se concentrează pe comunicarea în echipe mici care dezvoltă software non-critic. Are definite doar 3 roluri: Sponsor, Proiectant Senior și Programator. Cei care joacă aceste roluri vor

acoperi toate activitățile bunului mers al proiectului (de exemplu Programatorul va fi responsabil și cu activitatea de testare, nu doar cu dezvoltarea de cod). Pentru proiectele Crystal Clear perioada medie de finalizare a proiectului este de 60-90 de zile iar documentația necesară este una minimă (se documentează doar jaloanele).

În cazul Crystal Orange sunt definite 7 roluri: Sponsor, Manager de proiect, Analist de business, Arhitect, Proiectant senior, Programator și tester. În plus sunt definite mai multe categorii de livrabile ce trebuie actualizate pe parcursul proiectului: document de cerințe, planificarea versiunilor intermediare, planificarea activităților, raportul de stare, documentul de proiectare a interfeței cu utilizatorul, modelul obiectual, manualul de utilizare, scenarii de teste. Durata medie de finalizare a proiectului este de 90-120 de zile.

Crystal este singura metodologie care formalizează nivelul de criticitate ale proiectului și este unul dintre cele mai flexibile sisteme având implementări specifice pentru aproape orice dimensiune de proiect. Cu toate acestea un proiect nu poate face tranziția de la o metodologie Crystal la alta, acestea nefiind compatibile.

Dynamic Systems Development Method (DSDM)

DSDM a fost dezvoltată în Marea Britanie de DSDM Consortium în 1990. Este sigura metodologie care este „standardizată" și ale cărei versiuni sunt foarte riguros documentate. DSDM este cea mai populară metodologie Agile în Marea Britanie deși este considerată ca fiind cea mai apropiată de metodologiile tradiționale. În plus specificația DSDM nu este gratuită, fiecare echipă ce dorește să implementeze această metodologie trebuind să cumpere documentația de la DSDM Consortium.

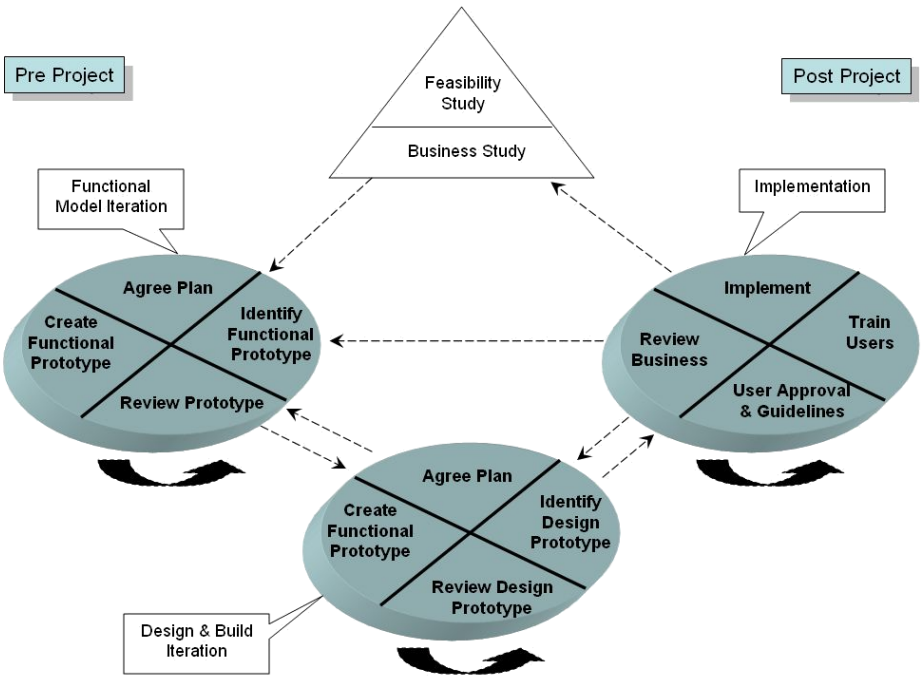

Figura 8.5. Cele trei faza ale proiectelor DSDM
(sursa: http://www.devx.com/)

DSDM împarte activitățile specifice fiecărui proiect în trei etape: pre-proiect, ciclul de viață a proiectului și post-proiect. La baza DSDM stau nouă principii:

- participarea utilizatorilor,
- responsabilizarea echipei de proiect,
- livrare frecventă,
- abordarea nevoile curente de business,
- dezvoltare iterativă și incrementală,
- permite inversarea schimbărilor,
- scopul de nivel înalt (viziunea) este stabilit înainte de începerea proiectului,
- testare de-a lungul ciclului de viață,
- comunicare eficientă și eficace.

Concluzii

Deși aflate sub aceeași umbrelă, a Manifestului Agile, metodologiile prezentate diferă între ele printr-o multitudine de aspecte. În același timp există contexte specifice anumitor proiecte care fac utilizarea anumitor metodologii mai potrivită decât altele. Tabelul 8.1 demonstrează că nu există o metodologie care să poată fi considerată potrivită pentru orice context particular și de aceea trebuie ca această să fie selectată cu atenție de către managerul de proiect înainte de începerea proiectului.

Context	XP	Scrum	Lean/ Kanban	FDD	AUP	Crystal	DSDM
Echipe mici	✓	✓	✓	X	X	-	✓
Cerințe în schimbare	✓	✓	✓	✓	-	-	X
Echipe distribuite	X	✓	✓	✓	✓	X	X
Documentație stufoasă	X	X	-	-	✓	-	✓
Sisteme critice	X	-	-	-	-	✓	X
Clienți multipli	X	✓	✓	-	-	-	X

Tabel 8.1. Utilitatea principalelor metodologii Agile în diverse contexte

9. Contracte Agile. Gestionarea riscurilor

Contracte Agile

Dacă în proiectele tradiționale, în general, scopul era considerat fix iar timpul și bugetul negociabile, într-un context Agile timpul și bugetul sunt considerate fixe iar scopul este negociat la începutul fiecărei iterații (figura 9.1).

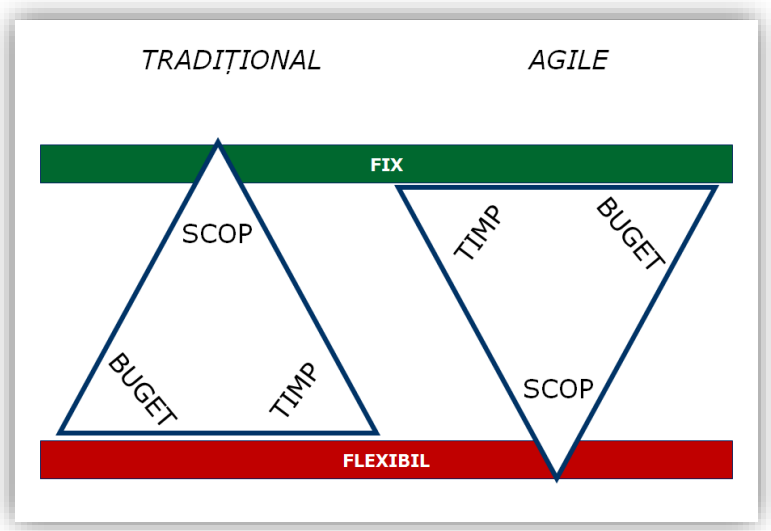

Figura 9.1. Abordarea principalilor factori de succes ai proiectelor în cazul managementului tradițional și Agile

Acest aspect ne duce la concluzia că tipurile de contracte ce funcționează în proiectele tradiționale nu mai sunt potrivite într-un context Agile. Prin urmare trebuie găsite variante care să se sincronizeze cu spiritul valorii a treia din Manifestul Agile: *„**Colaborarea cu clientul** este mai importantă decât negocierea contractuală"*. Acest lucru este necesar deoarece în mod contrar contractele ce se vor semna vor avea efortul supra estimat artificial pentru un set de cerințe ce conțin multe elemente lipsite de valoare pentru client.

Vom enumera în cele ce urmează câteva variante de contracte Agile uzuale.

Contract DSDM

Acest tip de contract este specific metodologie DSDM și structura sa este descrisă în specificațiile date de *DSDM Consortium*. Contractele DSDM se concentrează pe calitatea software-ului de a satisface nevoile de *business* ale clientului și pe trecerea testelor și mai puțin pe respectarea întocmai a specificațiilor.

Contracte tip *Money for Nothing* și *Changes for free*

În contractele *Money for Nothing* se stipulează că la un moment dat clientul poate decide oprirea proiectului dacă va considera că funcționalitățile rămase de implementat au o valoare mult prea mică pentru propria afacere pentru a merita continuarea (cu alte cuvinte valoarea întoarcerii investiției - *Return of Investment*, ROI – nu justifică implementarea funcționalităților rămase).

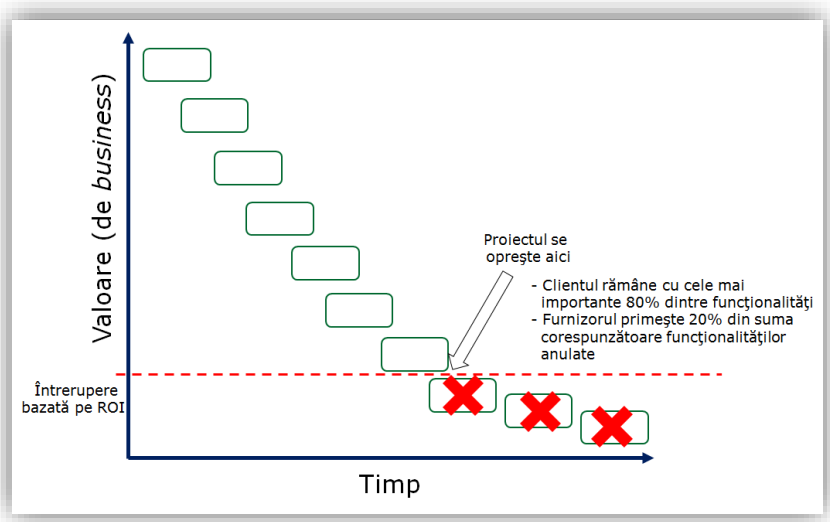

Figura 9.2. Schema de funcționare a clauzei *Money for Nothing*

În acel moment firma contractată va primi sumă de bani echivalentă cu 20% din valoare funcționalităților neimplementate. Prin această clauză clientul elimină pierderi generate din implementarea unor lucruri fără valoare pentru el iar firma de software va primi o sumă de bani care să compenseze parțial pierderile generate până toți membrii echipei sunt alocați altor proiecte.

Cea de-a doua posibilă clauză, *Changes for free*, poate fi adăugată într-un contract alături de clauza *Money for Nothing* pentru a permite clientului să aducă ulterior modificări în specificațiile inițiale, pe baza cărora s-au stabilit termenii contractuali. Aceste modificări vor avea ca efect eliminarea altor funcționalități cu prioritate scăzută din specificațiile inițiale.

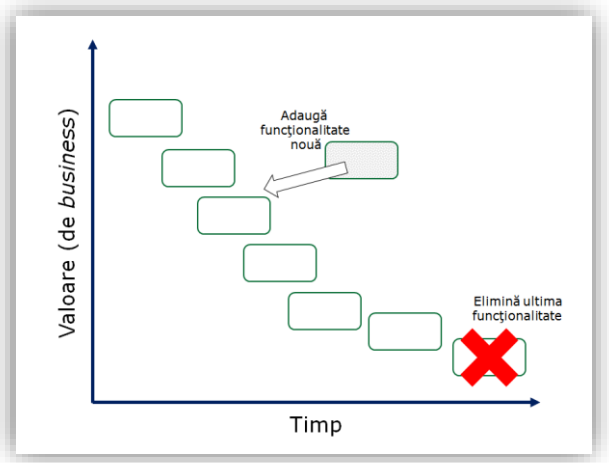

Figura 9.3. Schema de funcționare a clauzei *Changes for Free*

Contracte cu prețuri graduale fixe

Contractele cu prețuri graduale fixe specifică tarife diferite pentru trei situații distincte și anume: dezvoltare înainte de termen, dezvoltare la termen, respectiv dezvoltarea după termenul agreat (figura 9.4).

Project Completion	Graduated Rate	Total Fee
Finish Early	$110 / hour	$92000
Finish On Time	$ 100 / hour	$100000
Finish Late	$ 90 / hour	$112000

Figura 9.4. Clauză specifică contractelor gradate cu preț fix

Practic un astfel de contract conține un sistem de premiere și taxare în sine în funcție de data de finalizare a produsului final. Totuși, acest tip de contract nu schimbă foarte mult datele unui contract cu preț fix tradițional decât dacă data de finalizare este mult mai devreme sau mult mai târzie decât cea estimată inițial.

Contracte cu preț fix per pachet

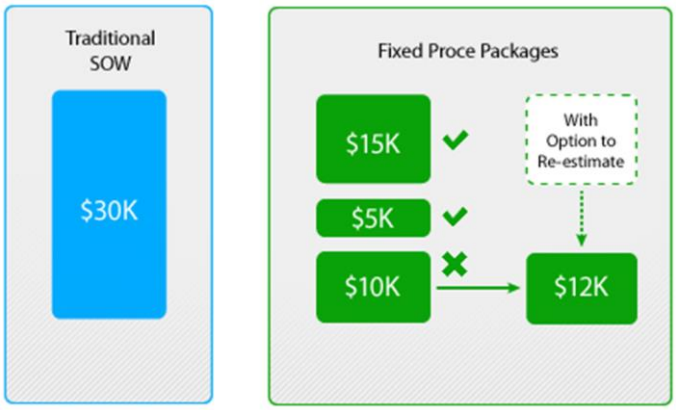

Figura 9.5. Schema de funcționare a contractelor cu preț fix per pachet (sursa: https://www.future-processing.pl/blog/agile-contracts-part-2/)

Contractele cu preț fix per pachete permit reestimarea efortului pentru pachetele a căror dezvoltare nu a fost încă începută (figura 9.5).

Gestionarea riscurilor în proiecte Agile

Deși nu reprezintă un subiect tratat special în nici una dintre metodologiile Agile, gestionarea riscurilor ocupă un rol foarte important. De fapt, o serie de principii și practici Agile au ca și efect eliminarea, anticiparea sau ameliorarea riscurilor (cu ar fi de exemplu livrare rapidă, integrare continuă, amânarea deciziilor până la ultimul moment posibil etc).

Gestionarea riscurilor se realizează într-un ciclu simplu format din 4 pași:

- Identificarea riscurilor,
- Analiza (calitativă/cantitativă) a riscurilor,
- Planificare acțiuni (dacă este cazul),
- Monitorizarea și controlului acțiuni.

Spre deosebire de managementul tradițional, aici lipsește intenționat un pas important: execuția riscului. Acest lucru se întâmplă deoarece, în momentul în care se planifică o acțiune, aceasta este automat inserată în *backlog* alături de celelalte funcționalități identificate. Ce mai trebuie făcut este doar să fie prioritizate la fel ca și celelalte funcționalități pentru a putea fi incluse în *backlog*-ul de sprint/iterație.

O variantă practică este cea prezentată schematic în figura 9.6.

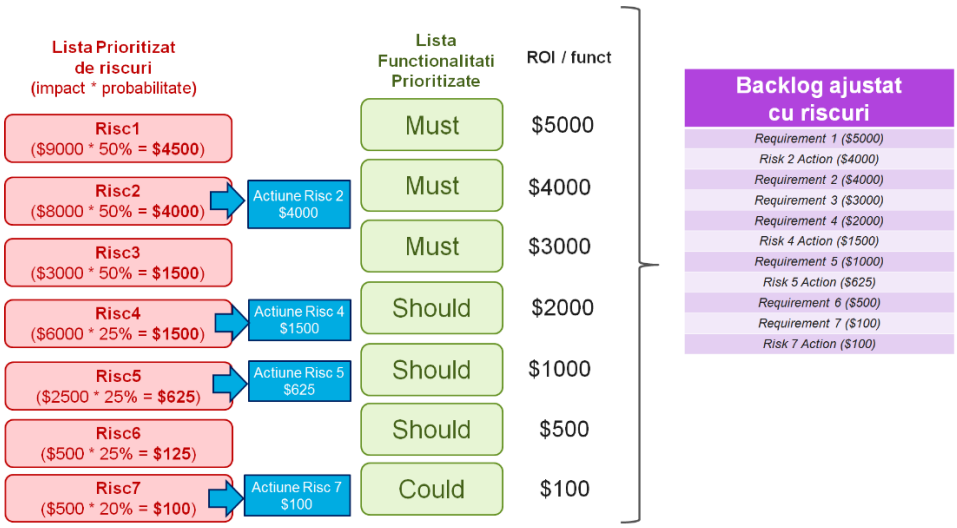

Figura 9.6. Gestionarea riscurilor într-un proiect Agile

Aici fiecare funcționalitate din backlog, în urma prioritizării (în exemplul din figură s-a ales un model de prioritizare MoSCoW: *Must have*, *Should have*, *Could have*, *Would like to have*) este asociată cu un procent din ROI-ul estimat, procent proporțional cu prioritatea sa. Distribuirea valorii ROI pe funcționalități este realizată de către client și nu este neapărat un proces exact (dar suficient pentru a putea prioritiza acțiunile de tratare a riscurilor).

În același timp, în urma analizei riscurilor, vom obține pentru fiecare risc un impact (în bani) și o probabilitate de apariție (în procente). Produsul acestor două valori ne va da valoarea riscului și, implicit, valoare acțiunii de tratare a riscului respectiv. În figura 9.6, riscurile 1, 3 și 6 nu au asociate un răspuns, însă în celelalte cazuri răspunsurile vor fi inserate în *backlog* și vor ocupa o poziție corespunzătoare valorii lor comparativă cu valoarea asociată funcționalităților.

Bibliografie

[1] P. F. Drucker, The Landmarks of Tomorrow: A Report on the New "post-modern" World, Transaction Publishers, 1996.

[2] Standish Group International, „The CHAOS Report 2015," Boston MA, 2015.

[3] D. M. Suciu, „Particularități ale proiectelor informatice," *Today Software Magazine,* nr. 8, pp. 11-13, 2012.

[4] M. Norris, M. Payne și P. Rigby, The Healthy Software Project: a guide to successful development, John Wiley & Sons, 1993.

[5] ***, „Manifesto for Agile Software Development," 2001. [Interactiv]. Available: http//agilemanifesto.org. [Accesat 2018].

[6] D. Snowden, „Complex Acts of Knowing: Paradox and Descriptive Self Awareness," *Journal of Knowledge Management,* vol. 2, nr. 6, pp. 100-111, 2002.

[7] VersionOne, „The 10th Annual State of Agile Report," 2016. [Interactiv]. Available: https://versionone.com/pdf/VersionOne-10th-Annual-State-of-Agile-Report.pdf. [Accesat 2018].

[8] B. Boehm și R. Turner, Balancing Agility and Discipline: A Guide for the Perplexed, Pearson Education Incorporated, 2003.

[9] M. Popendieck și T. Popendieck, Lean Software Development: An Agile Toolkit, Addison-Wesley Professional, 2003.

[10] D. J. Anderson, Kanban: Successful Evolutionary Change for Your Technology Business, Blue Hole Press , 2010.

[11] V. Duarte, NoEstimates: How To Measure Project Progress Without Estimating, OikosofySeries, 2016.

[12] T. Demarco, Waltzing with Bears - Managing Risk on Software Projects, Dorset House, 2003.

[13] A. Kelly, Continuous Digital - An agile alternative to projects for digital business, 2017.

[14] PMI, PMBOK® Guide – Sixth Edition, Project Management Institute, 2017.

[15] J. Humble, The Case for Continuous Delivery, ThoughtWorks, 2014.

[16] J. McKendrick, How Amazon handles a new software deployment every second, zdnet, 2015.

[17] K. Obermüller și J. Campbell, Mob Programming - the Good, the Bad and the Great, "under the hood" blog, 2016.

[18] R. C. Martin, My Lawn, www.cleancoder.com, 2014.

[19] D. M. Suciu, „Product Mindset," *Today Software Magazine,* nr. 13, pp. 15-17, 2013.

[20] D. M. Suciu, „Best Practices în Agile," *Today Software Magazine,* nr. 15, pp. 13-15, 2013.

[21] D. M. Suciu, „Agile Humanum Est," *TodaySoftware Magazine,* nr. 39, pp. 12-14, 2015.

[22] D. M. Suciu, „Agile si adaptarea la schimbare," *Today Software Magazine,* nr. 63, pp. 11-13, 2017.

[23] J. P. Womack și D. T. Jones, Lean Thinking: Banish Waste and Create Wealth in Your Corporation, 2nd edition, Productivity Press, 2003.

[24] D. Anderson, Kanban: Successful Evolutionary Change for Your Technology Business, Blue Hole Press, 2010.

[25] C. Cobb, Making Sense of Agile Project Management, Balancing Control and Agility, John Wiley and Sons, 2011.

[26] A. Cockburn, Crystal Clear: A Human-Powered Methodology for Small Teams, Addison-Wesley Professional, 2004.

[27] A. Cockburn, Agile Software Development: The Cooperative Game, 2nd edition, Upper Saddle River, NJ: Addison-Wesley, 2007.

[28] M. Cohn, Succeeding with Agile Software Development Using Scrum, Addison Wesley, 2010.

[29] M. Cohn, User Stories Applied: For Agile Software Development, Upper Saddle River, NJ: Addison-Wesley, 2004.

[30] S. Palmer și J. M. Felsing, A Practical Guide to Feature-Driven Development, Prentice Hall, 2002.

[31] E. Derby și D. Larsen, Agile Retrospectives, Making Good Teams Great, Pragmatic Bookshelf Publishing, 2006.

[32] A. S. Koch, Agile Software Development, Evaluating the Methods for Your Organization, Artech House, 2005.

[33] C. F. Kurtz și D. Snowden, „The new dynamics of strategy: Sense-making in a complex and complicated world," *IBM Systems Journal*, vol. 42, nr. 3, pp. 462-483, 2003.

[34] J. Langr și T. Ottinge, Agile in a Flash, Speed-Learning Agile Software Development, Pragmatic Bookshelf Publishing, 2011.

[35] C. Larman, Agile and Iterative Development, A Manager's Guide, Addison-Wesley, 2003.