UNIVERSITATEA POLITEHNICA BUCUREȘTI

FACULTATEA DE ELECTRONICĂ, TELECOMUNICAȚII ȘI TEHNOLOGIA INFORMAȚIEI

 INGINERIA SISTEMELOR

 **Agile Modeling**

 **443 A Petrescu Maria**

**Cuprins**

1. Introducere

2. Comparație cu ingineria software tradițională

 2.1 Metoda tip adaptivă sau predictivă

 2.2 Practica iterativă sau cascadă

 2.3 Cod sau documentație

 3. Tipuri de metodologii Agile

 3.1 Scrum

 3.2 Programare extremă (XP)

 3.3 Proces rațional unificat (RUP)

 4. Principii și exemple de design Agile

 4.1 Principiul Singurei Responsabilități (SRP)

##  4.2 Principiul Inversării Dependenței

##  4.3 Principiul Deschis-Închis (OCP)

1. **Introducere**

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

O altă caracteristică importantă este comunicarea frecventă între membrii echipei, care, în multe cazuri, se întâlnesc într-o scurtă ședință zilnică, denumită *stand-up* sau *scrum* (în care fiecare prezintă pe scurt progresul său în ultima zi de lucru și problemele cu care s-a confruntat. Acestora li se alătură și un reprezentant al clientului, care trebuie să fie informat de aspectele dezvoltării, pentru a ști ce modificări este realist să ceară și cât de mult ar putea costa ele. Astfel, toată lumea are cunoștințe despre fiecare aspect al dezvoltării aplicației și poate prelua munca altuia sau ajuta pe altcineva. [[1]](https://ro.wikipedia.org/wiki/Dezvoltarea_agil%C4%83_de_programe)

Majoritatea metodelor "agile" încearcă să minimizeze riscurile prin dezvoltarea de software în perioade scurte de timp, denumite iteraţii care durează, de obicei, de la o săptămână până la patru săptămâni. Fiecare iteraţie poate fi considerată un proiect mic de dezvoltare software de sine-stătător şi conţine toate sarcinile necesare dezvoltării unei funcţionalităţi noi: planificarea, analiza cerinţelor, proiectarea, codificarea, testarea şi elaborarea documentaţiei. Deoarece este posibil ca o iteraţie să nu adauge suficientă funcţionalitate pentru garantarea realizării produsului, un proiect "agil" de dezvoltare software are ca obiectiv oferirea unui software nou, la sfârşitul fiecărei iteraţii. În cele mai multe cazuri, la sfârşitul fiecărei iteraţii, este furnizat un software. Acest fapt este, în mod special, adevărat atunci când software-ul este bazat pe Web şi poate fi produs şi lansat cu uşurinţă. Indiferent de situaţie, la sfârşitul fiecărei iteraţii, echipa proiectului reevaluează priorităţile proiectului.

Metodele "agile" consideră versiunea de software (working software) ca fiind măsura principală a progresului obţinut. [[2]](http://rria.ici.ro/ria2007_3/art01.html)

Dezvoltarea agilă este un supliment pentru celelalte metodologii agile, cum ar fi Scrum, programarea extremă (XP), și procesul rațional unificat (RUP).

**2. Comparație cu ingineria software tradițională**

Comparativ cu ingineria software tradițională, dezvoltarea agilă se adresează în principal la sistemelor complexe și proiectelor cu caracteristici dinamice, nedeterministe și non-lineare, unde estimările exacte, planurile stabile și previziunile sunt de multe ori greu de obţinut în stadii incipiente, si marile proiecte şi aranjamente prestabilite pot provoca, probabil, o mulțime de pierderi, nefiind economice. Aceste argumente de bază și experiențele prețioase din industrie învățate în ani de succese și eșecuri au ajutat la favoarizarea modelului agil de adaptare, iterativ și dezvoltare evolutivă. [[3]](http://www.sitweb.ro/filozofia-dezvoltarii-web-agile/)

**2.1 Metoda tip adaptivă sau predictivă**

Metodele de dezvoltare se găsesc într-un continuum de la *adaptiv* la *predictiv*. Metodele agile se află pe partea *adaptivă* a acestui continuum. O idee de metodă de dezvoltare adaptivă este abordarea de tip „Rolling Wave” în planificarea programului, care identifică repere, dar lasă flexibilitatea căilor de a le atinge, și, de asemenea, permite chiar schimbarea etapelor. Metodele adaptive se concentrează pe adaptarea rapidă la realitățile în schimbare. În cazul în care este nevoie de modificarea proiectului, o echipă de adaptare îl schimbă. O echipă de adaptare va avea dificultati în a descrie exact ce se va întâmpla în viitor. Cu cât este mai îndepărtat termenul, cu atât mai vagă va fi metoda de adaptare va fi cu privire la ce se va întâmpla la acea dată. O echipă de adaptare nu se poate raporta exact la ce sarcini se vor îndeplini săptămâna viitoare, ci doar ceea ce se intenţionează pentru luna viitoare. Când e întrebată despre o lansare de peste șase luni de acum încolo, o echipă de adaptare ar putea să vorbească doar despre angajamentul de lansare, sau privind valoarea așteptată față de cost.

Metoda *predictivă*, în schimb, se concentrează pe analiza și planificarea viitorului în detaliu și luarea în considerare a riscurilor cunoscute. La extreme, o echipă de predicție poate raporta exact ce opţiuni și sarcini sunt planificate pentru întreaga durată a procesului de dezvoltare. Metodele predictive se bazează pe o analiză eficientă a fazei incipiente și, dacă acest lucru merge foarte rău, proiectul ar putea avea dificultăți în schimbarea direcției. Echipele de predicție vor institui de multe ori un panou de control cu schimbările pentru a se asigura că numai modificările cele mai valoroase sunt luate în considerare. [[3]](http://www.sitweb.ro/filozofia-dezvoltarii-web-agile/)

**2.2 Practica iterativă sau cascadă**

Una dintre diferențele dintre agil și cascadă este că testarea software este realizată la diferite stadii ale ciclului de viață de dezvoltare a software. În modelul cascadă, există întotdeauna o fază de testare separată, aproape de finalizarea unei faze de implementare. Cu toate acestea, în metoda agilă și mai ales în programarea extremă, testarea se face de obicei în același timp cu codarea, sau, cel puțin activitatea de testare începe în primele etape ale iteraţiei.

Pentru că faza de testare se face la fiecare mică iterație – care dezvoltă o mică bucată de software – utilizatorii pot folosi frecvent aceste piese noi de software și a valida valoarea.

După ce utilizatorii cunosc valoarea reală a piesei actualizate de software, ei pot lua decizii mai bune cu privire la viitorul software. Cu o retrospectivă a valorii și sesiunii de re-planificare software la fiecare iterație – modelul Scrum are maximum o lună ca durată de iteraţie, – va ajuta astfel echipa să adapteze continuu planurile sale astfel încât să maximizeze valoarea pe care o oferă.

Această practică iterativă introduce, de asemenea, o „mentalitatea de produs”, mai mult decât „mentalitatea de proiect” de tip cascadă. Software poate fi văzut ca un organism viu, care se schimbă în mod activ ca urmare a schimbărilor de mediu. Atâta timp cât software este utilizat, în special atunci când are concurenţi, iterațiile în dezvoltarea de software agilă va conduce schimbarea.

Din cauza stilului scurt de iteraţie în dezvoltarea de software agilă, există de asemenea, legături puternice cu conceptul de „lean startup”. [[3]](http://www.sitweb.ro/filozofia-dezvoltarii-web-agile/)

**2.3 Cod sau documentație**

Susținătorii dezvoltării de software agilă afirmă că dezvoltatorii ar trebui să scrie documentație în cazul în care acesta este cel mai bun mod de a îndeplini obiectivele în cauză, dar că există modalități de multe ori mai bune de a atinge aceste obiective decât scrierea documentației statice. Scott Ambler afirmă că documentația ar trebui să fie „doar destul de bună”, că documentația prea multă sau completă ar provoca, de obicei, pierderi, iar dezvoltatorii rareori au încredere în documentația detaliată, pentru că este, de obicei, desincronizată cu codurile, în timp ce documentația prea puțină poate provoca, de asemenea, probleme de întreținere , comunicare, învățare și schimb de cunoștințe. [[3]](http://www.sitweb.ro/filozofia-dezvoltarii-web-agile/)

**3. Tipuri de metodologii Agile**

 **3.1 Scrum**

Principiul metodologiei SCRUM constă în dezvoltarea incrementală a unei aplicaţii software, implicând totodată păstrarea unei liste transparente cu cererile de upgradare sau de corectare care trebuie implementate (backlog). Datorită livrărilor frecvente, care de regulă au loc o dată la patru săptămâni, clientul primeşte de fiecare dată o aplicaţie care conţine un număr tot mai mare de funcţionalităţi şi care se află în perfectă stare de funcţionare. Astfel, metoda se bazează pe o dezvoltare iterativă cu frecvenţă regulată, cuprinsă între două şi patru săptămâni. Prin urmare, upgradările pot fi integrate cu mai multă uşurinţă atunci când se utilizează [ciclul în V](http://www.pentalog.ro/abordare/metodologia-ciclul-in-v.htm). [[4]](http://www.pentalog.ro/abordare/metodologia-agile-scrum.htm)

Această metodă necesită patru tipuri de şedinţe:

* Şedinţele zilnice
* Şedinţele de planificare
* Şedinţele de revizuire a activităţii
* Şedinţele retrospective

 **3.2 Programare extremă (XP)**

Programarea extremă este cea mai faimoasă dintre metodele active. Acesta este formată din metode simple şi inter-dependente. Se enumeră mai jos principiile de bază ale programării extreme:

**Clientul este membru al echipei** - Clientul şi dezvoltatorii lucrează împreună pentru a rezolva problemele ce pot apărea într-un proiect (**Customer team member**). Clientul unei echipe XP este o persoană sau un grup de persoane care definesc priorităţile proiectului. Cazul cel mai fericit este clientul să fie localizat în aceeaşi cameră/clădire cu dezvoltatorii proiectului.

**Cicluri scurte** - În general, în cazul unui proiect XP, dezvoltatorii trimit la fiecare două săptămâni (părţi de) soft care rulează corect. La sfârşitul fiecărei iteraţii (de două săptămâni), sistemul este verificat de clienţi pentru a obţine răspunsul lor.

**Planul de iteraţie** este o trimitere minoră care ar putea fi adăugată în producţie. Este o colecţie de specificaţii utilizator selectate de client în funcţie de bugetul stabilit de dezvoltatori. În momentul începerii unei iteraţii, clientul este de acord să nu mai schimbe definiţia sau priorităţile specificaţiilor din acea iteraţie. Astfel, dezvoltatorii sunt liberi să împartă problema în subprobleme.

**Planul de distribuire** se referă la iteraţiile procesului, de obicei şase. Acesta reprezintă o trimitere majoră care de obicei este parte a producţiei. Din nou, clienţii selectează un număr de specificaţii în funcţie de bugetul propus. Clientul poate schimba conţinutul planului de distribuire în fiecare moment, poate rezilia anumite specificaţii, sau schimba prioritatea unei specificări.

**Programare în pereche** - Codul de producţie este scris în perechi de programatori lucrând împreună la acelaşi calculator (**pair programming**). Unul din ei scrie codul, iar celalalt caută erori şi îmbunătăţiri.

**Dezvoltare dirijată de teste** - Întreg codul de producţie este scris pentru a testa modulele care nu sunt implementate corect (**Test-driven development**). [[5]](http://www3.agora.ro/index.php?qs_sect_id=2071)

 **3.3 Proces rațional unificat (RUP)**

**IBM Rational Unified Process (RUP) este o concepţie de dezvoltare software, gândită, dezvoltată, livrată şi întreţinută ca un framework. RUP este creat şi documentat folosind UML şi este livrat on-line de către Rational Software Corporation, o divizie a IBM. Se fac upgrade-uri în mod regulat de către Rational Software la aproximativ 2 ani. Fiind modular şi în format electronic, el poate fi configurat şi personalizat pentru a răspunde cerinţelor specifice fiecărei organizaţii. Este integrat cu foarte multe unelte de dezvoltare software în suitele Rational, astfel încât dezvoltatorii pot să acceseze fie asistenţa proceselor în cadrul uneltei pe care o folosesc, fie să acorde o asistenţă a proceselor, consistentă şi personalizată pentru echipa de realizare. Ciclul de dezvoltare RUP este alcătuit din patru faze secvenţiale (partea de sus a diagramei) ce modelează aspecte din derularea proiectului software financiare, strategice, comerciale şi umane. Nouă activităţi (partea stângă a diagramei) modelează aspectele tehnice ale dezvoltării proiectului, incluzând modelarea proceselor de business, implementare, testare şi altele. Fiecare fază a unui proiect RUP este împărţită în iteraţii (partea de jos a diagramei).** [[6]](http://andrei.clubcisco.ro/cursuri/f/f-sym/4idp/3_Instrumente_CASE_doc.pdf)



 FIGURA 1 [6]

**4. Principii și exemple de design Agile**

**4.1 Principiul Singurei Responsabilități (SRP)**

* **O clasă trebuie să aibă un singur motiv pentru a fi modificată.**

**Principiul singurei responsabilități este unul dintre cele mai simple și cu toate acestea unul dintre cele mai dificil de aplicat. Identificarea și separarea responsabilităților este unul dintre aspectele fundamentale ale designului *softwar*e.**

**În contextul SRP, responsabilitatea poate fi definită ca "*un motiv pentru modificare*". Atunci când cerințele proiectului se modifică, modificările vor fi vizibile prin modificarea responsabilității claselor. Dacă o clasă are mai multe responsabilități, atunci va avea mai multe motive de schimbare. Având mai multe responsabilități cuplate, modificări asupra unei responsabilități vor implica modificări asupra celorlalte responsabilități ale clasei. Acesta corelație conduce către un design fragil.*Fragilitatea* însemnă că o modificare adusă sistemului produce o ruptură în design, în locuri care nu au nici o legătură conceptuală cu partea care a fost modificată.** [[7]](http://www.todaysoftmag.ro/article/980/principii-de-design-agile)

****

FIGURA 2 [[7]](http://www.todaysoftmag.ro/article/980/principii-de-design-agile)

**În acest exemplu cele două responsabilități sunt separate, astfel încât clasa care le utilizează - Phone, nu trebuie să le cupleze pe amândouă. Schimbările asupra conexiunii nu vor influența metodele responsabile cu transmisia datelor. Pe de altă parte în cazul în care cele două responsabilități nu prezintă motive de modificare în timp, nu este necesară nici separarea lor. Cu alte cuvinte responsabilitățile unei clase trebuie separate, numai dacă există șanse reale ca responsabilitățile să producă modificări, influențându-se reciproc.**

**4.2 Principiul Inversării Dependenței**

* **Modulele de pe nivelurile ierarhice superioare nu trebuie să depindă de modulele de pe nivelurile ierarhice inferioare. Toate ar trebui să depindă doar de module abstracte.**
* **Abstractizările nu trebuie să depindă de detalii. Detaliile trebuie să depindă de abstractizări.**

**În *Fig. 3* este prezentată o diagramă de clase în care este respectat principiul inversării dependenței. Astfel, fiecărui nivel care accesează funcționalitatea dintr-un nivel ierarhic inferior i s-a adăugat o interfață care va fi implementată de nivelul ierarhic inferior. În acest fel interfața prin care cele două niveluri comunică este definită în nivelul ierarhic superior astfel încât dependența a fost inversată și anume nivelul ierarhic inferior depinde de nivelul ierarhic superior. Modificări făcute la nivelurile inferioare nu mai afectează nivelurile superioare ci invers. În concluzie, diagrama de clase din *Fig. 3*  respectă principiul inversării dependenței.** [[7]](http://www.todaysoftmag.ro/article/980/principii-de-design-agile)

****

FIGURA 3 [[7]](http://www.todaysoftmag.ro/article/980/principii-de-design-agile)

**4.3 Principiul Deschis-Închis (OCP)**

**Entitățile software (clase, module, funcții, etc.) trebuie să fie deschise pentru extensii, dar închise pentru modificări.**

**Atunci când o singură modificare asupra unui modul software rezultă în necesitatea de a modifica o serie de alte module, atunci designul suferă derigiditate . Principiul OCP susține refactorizarea designului astfel încât modificări ulterioare de același tip, nu vor mai produce modificări asupra codului existent , care deja funcționează, în schimb va necesita doar adăugarea de noi module.Un modul software care respectă principiul Deschis-Închis are două caracteristici principale:**

* **"Deschis pentru extensii." Acesta înseamnă că acel comportament al codului poate fi extins. Atunci când cerințele proiectului se modifică, codul poate fi extins cu implementarea noilor cerințe, adică se poate modifica comportamentul modulului deja existent.**
* **"Închis pentru modificări." Implementarea noilor cerințe nu necesită modificări asupra codului deja existent.**

**Abstractizarea este metoda care permite modificarea comportamentului unui modul software , fără a modifica și codul deja existent al acestuia. În C++, Java sau oricare alt limbaj orientat obiect, este posibil să se creeze o abstractizare care oferă o interfață fixă și un număr nelimitat de implementări, adică de comportamente diferite.**

**Spre exemplu, figura 4 prezintă designul corespunzător care respectă principiul deschis-închis.** [[7]](http://www.todaysoftmag.ro/article/980/principii-de-design-agile)

****

FIGURA 4 [[7]](http://www.todaysoftmag.ro/article/980/principii-de-design-agile)

**Conformarea la acest principiu conduce la cele mai mari beneficii ale programării orientate obiect, acestea fiind flexibilitatea, reutilizarea și mentenanța codului.**

**Bibliografie**

[1]. <https://ro.wikipedia.org/wiki/Dezvoltarea_agil%C4%83_de_programe>

[2]. <http://rria.ici.ro/ria2007_3/art01.html>

[3]. <http://www.sitweb.ro/filozofia-dezvoltarii-web-agile/>

[4]. <http://www.pentalog.ro/abordare/metodologia-agile-scrum.htm>

[5]. <http://www3.agora.ro/index.php?qs_sect_id=2071>

[6]. <http://andrei.clubcisco.ro/cursuri/f/f-sym/4idp/3_Instrumente_CASE_doc.pdf>

[7]. <http://www.todaysoftmag.ro/article/980/principii-de-design-agile>