Manifestul pentru dezvoltarea agilă de software

Noi scoatem la iveală modalități mai bune de dezvoltare de software prin experiență proprie și ajutându-i pe ceilalți.
Prin această activitate am ajuns să apreciem:
Indivizii şi interacţiunea înaintea proceselor şi instrumentelor
Softwareul funcţional înaintea documentaţiei vaste
Colaborarea cu clientul înaintea negocierii contractuale
Receptivitatea la schimbare înaintea urmăririi unui plan
Cu alte cuvinte, deşi există valoare în elementele din dreapta, le apreciem mai mult pe cele din stânga.

Principiile manifestului Agil (12)

Noi urmăm aceste principii:

1. Prioritatea noastră este satisfacţia clientului prin livrarea rapidă şi continuă de software valoros.

2. Schimbarea cerinţelor este binevenită chiar şi într-o fază avansată a dezvoltării. Procesele agile valorifică schimbarea în avantajul competitiv al clientului.

3. Livrarea de software funcţional se face frecvent, de preferinţă la intervale de timp cât mai mici, de la câteva săptămâni la câteva luni.

4. Oamenii de afaceri şi dezvoltatorii trebuie să colaboreze zilnic pe parcursul proiectului.

5. Construieşte proiecte în jurul oamenilor motivaţi.
Oferă-le mediul propice şi suportul necesar
şi ai încredere că obiectivele vor fi atinse.

6. Cea mai eficientă metodă de a
transmite informaţii înspre şi în interiorul
echipei de dezvoltare este comunicarea faţă în faţă.

7. Software funcţional este principala măsură a progresului.

8. Procesele Agile promovează dezvoltarea durabilă.

9. Sponsorii, dezvoltatorii şi utilizatorii trebuie să poată
menţine un ritm constant pe termen nedefinit. Atenţia
continuă pentru excelenţă tehnică şi design bun
îmbunătăţeşte agilitatea.

10. Simplitatea--arta de a maximiza cantitatea
de muncă nerealizată--este esenţială.

11. Cele mai bune arhitecturi, cerinţe şi design
emerg din echipe care se auto-organizează.

12. La intervale regulate, echipa reflectă la cum
să devină mai eficientă, apoi îşi adaptează şi ajustează
comportamentul în consecinţă.

Principii de design Agile

Dumitrița Munteanu

Software engineer

@Arobs

PROGRAMARE

Programarea *agile* se bazează pe dezvoltarea produsului software în blocuri mici și incrementale, în care cerințele clienților și soluțiile oferite de programatori evoluează simultan.

Programarea *agile* are la bază o strânsă legătură între calitatea finală a produsului și livrările frecvente ale funcționalităților dezvoltate incremental. Cu cât se realizează mai multe livrări, cu atât calitatea produsului final va fi mai ridicată. Într-un proces de implementare *agile* , cerințele de modificare sunt privite ca un lucru pozitiv, indiferent de faza de dezvoltare în care se află proiectul. Aceasta deoarece cerințele de modificare evidențiază faptul că echipa a înțeles ceea ce este necesar pentru ca produsul software să satisfacă necesitățile pieței. Din acest motiv este necesar ca o echipă *agil* e să mențină structura codului cât mai flexibilă, astfel încât noile cerințe ale clienților să aibă un impact cât mai redus asupra arhitecturii existente. Aceasta nu înseamnă însă că echipa va face un efort suplimentar luând in considerare viitoarele cerințe și necesități ale clienților sau că va investi mai mult timp pentru a implementa o infrastructură care să suporte posibile cerințe necesare în viitor. Dar evidențiază faptul că *accentul este pus de dezvoltarea produsului curent* cât mai eficient. În acest acest scop, vom investiga câteva dintre principiile de *software design* care se impun aplicate de la o iterație la alta de către un programator *agile* , pentru a menține cât mai curat și flexibil posibil codul și designul proiectului. Principiile au fost propuse de către Robert Martin în lucrarea *Agile Software Development: Principles, Patterns, and Practices [1].*

Principiul Singurei Responsabilități (Single Responsability Principle : SRP)

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

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.

Exemplu:

Să presupunem că aveam o clasă care încapsulează conceptul de telefon și funcționalitățile aferente.

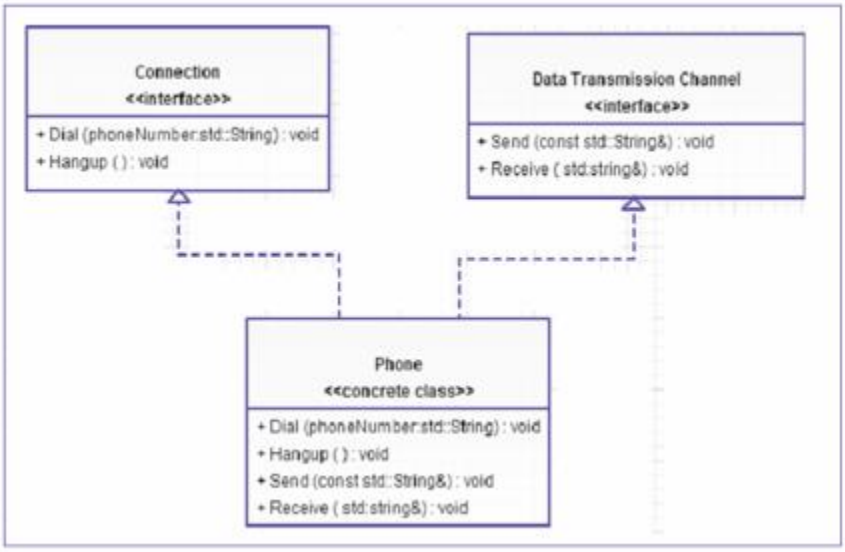
```
class Phone
{
    public void Dial(const string&
    phoneNumber);

    public void Hangup();
   public void Send(const string&
    message);

   public  Receive(const string&
    message);
   };
```

Acesta clasă ar putea fi considerată rezonabilă. Toate cele patru metode definite reprezintă funcționalități referitoare la conceptul de telefon. Și totuși această clasă are două responsabilități. Metodele *Dial* si *Hangup* sunt responsabile pentru realizarea conexiunii, în timp ce metodele *Send* și *Receive* sunt responsabile pentru transmiterea datelor.

În cazul în care signatura metodelor responsabile pentru realizarea conexiunii ar fi supuse schimbărilor, acest design ar fi rigid, deoarece toate clasele care apelează metodele *Dial* și *Hangup* ar trebui recompilate. Pentru a evita această situație este necesar un *re-design* care să separe cele două responsabilități.



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.

Concluzii

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 principiile de *agile software* design pe care le vom analiza în continuare, vom vedea că vom reveni, într-un fel sau altul, asupra acestui principiu.

Principiul Deschis-Închis (Open Closed Principle: OCP)

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

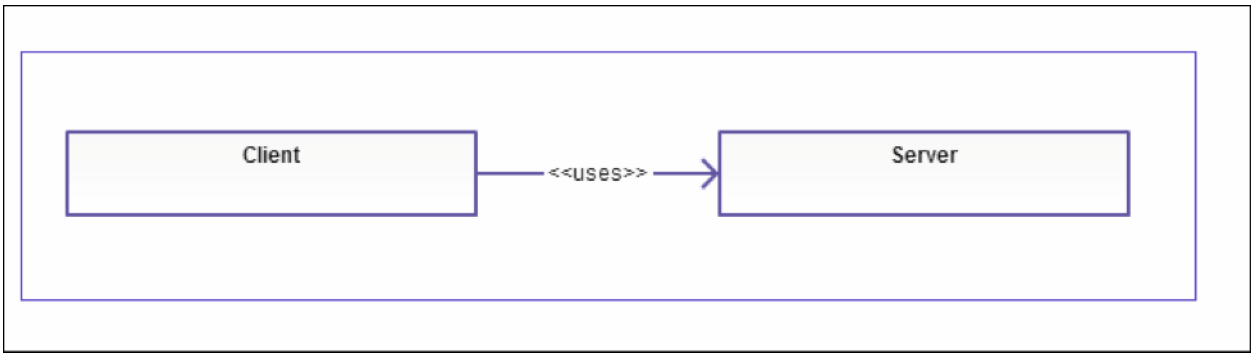
Atunci când o singură modificare asupra unui modul *software* rezultă în necesitatea de a modifica o serie de alte module, atunci designul suferă de*rigiditate* . Principiul OCP susține refactorizarea designului astfel încât modificări ulterioare de același tip, nu vor mai produce modificări asupra codului 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:

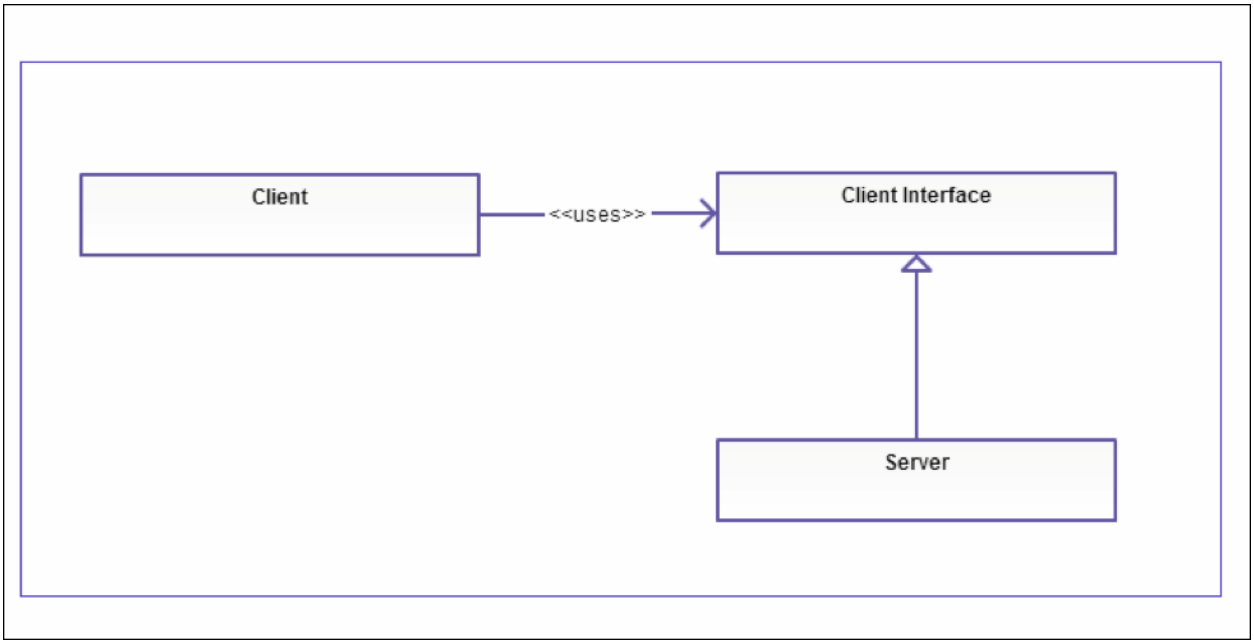
- **"Deschis pentru extensii." Acesta înseamnă că acel comportament al codului poate fi extins. Atunci când cerințele proiectului se modifică, codul poate fi extins cu implementarea noilor cerințe, adică se poate modifica comportamentul modulului deja existent.**
- **"Închis pentru modificări." Implementarea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

Abstractizareaeste 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 [2].

În *Fig. 2* este prezentată o diagramă de clase care nu respectă principiul deschis-închis. Atât clasa Client cât și clasa Server sunt clase concrete. Clasa Client folosește clasa Server. Dacă mai târziu se dorește ca această clasă Client să folosească un alt tip de server, va fi nevoie să se modifice clasa Client astfel încât să utilizeze noul server. Figure 2 Exemplu care nu respecta principiul OCP 1

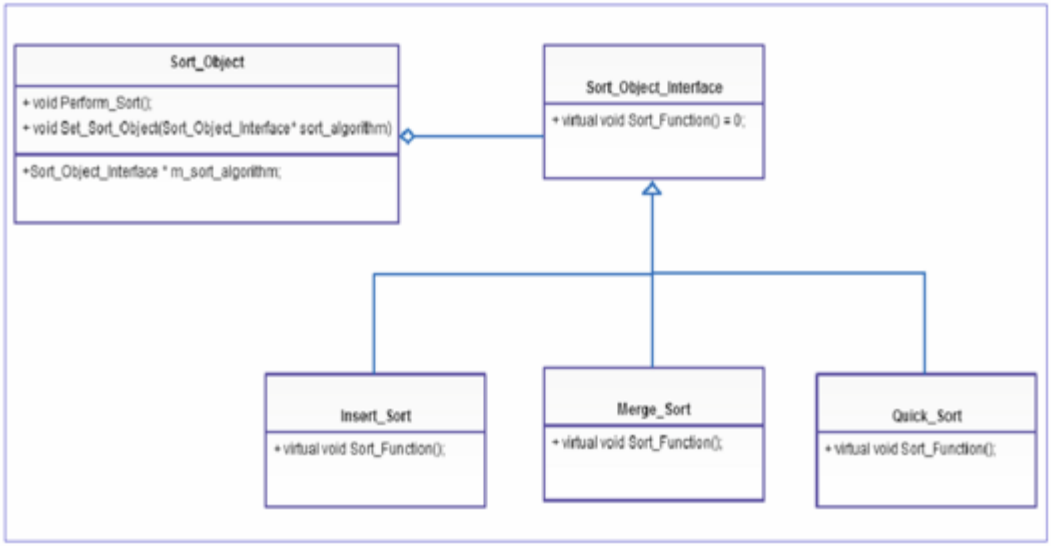


În *Fig. 3* se prezintă același design ca și în *Fig. 2* , dar de această dată principiul deschis-închis este respectat. În acest caz a fost introdusă clasa abstractă AbstractServer, iar clasa Client folosește această abstractizare. Totuși clasa Client va folosi de fapt clasa Server care implementează clasa ClientInterface. Dacă în viitor se dorește folosirea unui alt tip de server tot ce trebuie făcut va fi să se implementeze o nouă clasă derivată din clasa ClientInterface, dar de această dată clientul nu mai trebuie modificat.



Un aspect particular în acest exemplu, este modul în care am denumit clasa abstractă *ClientInterface* și nu *ServerInterface* , spre exemplu. Motivul pentru această alegere este faptul că clasele abstracte sunt mai apropiate de clasele client pe care le folosesc, decât de clasele concrete pe care le implementează.

Principiul Deschis -Închis este utilizat și în *design pattern* -urile Strategy și Plugin [3]. Spre exemplu, *Fig.4* prezintă designul corespunzător care respectă principiul deschis-închis.



Clasa *Sort_Object* efectuează o operație de sortare a obiectelor, operație care poate fi descrisă în interfața abstractă *Sort_Object_Interface* . Clasele derivate din clasa abstractă *Sort_Object_Interface* sunt obligate să implementeze metoda Sort_Function(), dar au în același timp libertatea de a oferi orice implementare doresc pentru această interfață. Astfel comportamentul specificat de interfața metodei *void Sort_Function()* , poate fi extins și modificat prin crearea de noi subtipuri ale clasei abstracte Sort_Object_Interface.

În definiția clasei Sort_Object vom avea următoarele metode:

```
void Sort_Object::Sort_Function()
{
       m_sort_algorithm->sortFunction();
}
void Sort_Object::Set_Sort_Algorithm(const Sort_Object_Interface*
sort_algorithm)
{
     std::cout << „Setting a new sorting algorithm...” << std::endl;
     m_sort_algorithm = sort_algorithm;
}
```

Concluzii

Principalele mecanisme în spatele acestui principiu sunt *abstractizarea* și*polimorfismul* . Ori de câte ori codul trebuie modificat pentru implementarea unei noi funcționalități, trebuie să se ia în considerare și crearea unei abstracții care să furnizeze o interfață pentru comportamentul dorit și care să ofere în același timp posibilitatea de a adăuga în viitor noi comportamente pentru aceeași interfață. Desigur nu întotdeauna este necesară crearea unei abstractizări. Acestea metodă este utilă în general acolo unde apar modificări frecvente. Conformarea la principiul deschis-închis este costisitoare. Aceasta necesită timp de dezvoltare și efort pentru a crea abstracțiile necesare. Aceste abstracții incrementează complexitatea *designului software* .

În schimb, principiul deschis-inchis reprezintă din multe puncte de vedere *un element central din programarea orientată obiect* . Conformarea la acest principiu conduce la cele mai mari beneficii ale programării orientate obiect, acestea fiind flexibilitatea, reutilizarea și mentenanța codului.

Principiu de substituire Liskov (LSP)

Tipurile de bază trebuie să poată fi substituite de tipurile derivate.

În limbaje precum C++ sau Java, mecanismul principal prin care se realizează abstractizarea și polimorfismul este moștenirea. Pentru a crea o ierarhie de moștenire corectă trebuie să ne asigurăm că clasele derivate, extind fără a înlocui funcționalitatea claselor de bază. Cu alte cuvinte, funcțiile care utilizează *pointer* -i sau referințe la clasele de bază trebuie să poată

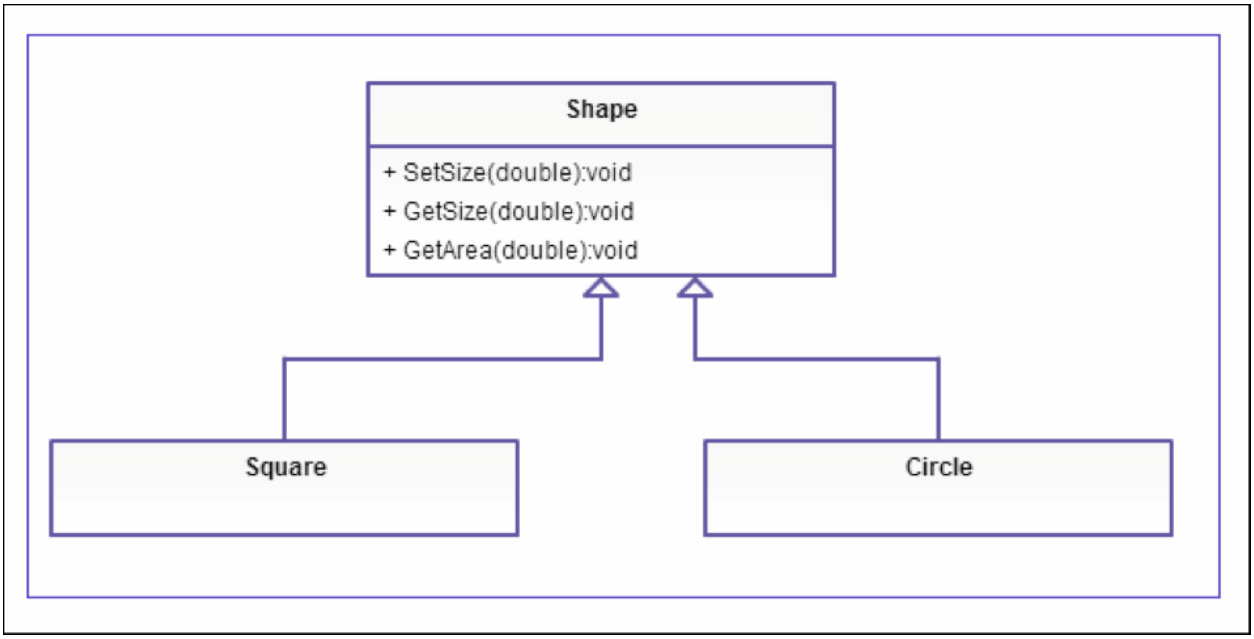
folosi instanțe ale claselor derivate fară să își dea seama de acest lucru. În caz contrar, noile clase pot produce efecte nedorite când sunt utilizate în entitațile programului deja existent. Importanța principiului LSP devine evidentă în momentul în care este incălcat.

Exemplu:

Să presupunem că avem o clasă Shape, a cărei obiecte sunt deja folosite undeva în aplicație și care are o metoda *SetSize* , care proprietatea *mSize* ce poate fi folosită ca latură sau diametru, în funcție de figura reprezentată.

```
class Shape
{
  public:
        void SetSize(double size);
        void GetSize(double& size);
 private:
        double mSize;
};
```

Mai târziu extindem aplicația și adăugăm clasele Square și Circle. Având în vedere faptul că moștenirea modelează o relație "este de tipul" (IS_A relationship) noile clase Square si Circle, pot fi derivate din clasa Shape.



Să presupunem în continuare că obiectele Shape sunt returnate de o metodă*factory* , pe baza unor condiții stabilite la *run time* , astfel încât nu cunoaștem exact tipul obiectului returnat.

Dar știm că este Shape. Obținem obiectul Shape, îi setăm proprietatea size de 10 unități și îi calculăm aria. Pentru un obiect *square*aria va fi 100.

```
void f(Shape& shape,  float& area)
{
  shape.SetSize(10);
  shape.GetArea(area);
  assert(area == 100);  // Oups!
// for circle area = 314.15927!
 }
```

În acest exemplu, atunci când funcția *f* , primește ca parametru *r* , o instanță a clasei Circle va avea un *comportament* greșit. Deoarece, în funcția *f,* obiectele de tip *Square* nu pot substitui obiecte de tip *Rectangle* , principiul LSP este violat. Funcția *f,* este fragilă în raport cu ierarhia Square/Circle.Design by Contract

Desigur că mulți programatori, se vor simți inconfortabil cu noțiunea de "comportament" care trebuie să fie "corect". Cum am putea presupune ceea ce așteaptă utilizatorii/clienții claselor pe care le implementăm?În acest scop, ne vine în ajutor tehnica designului prin contract (*design by contract* - DBC). Contractul unei metode îl informează pe autorul unei clase client despre comportamentele pe care se poate baza cu siguranță. Contractul este specificat prin declararea *precondițiilor* și a *postcondițiilor* pentru fiecare metodă. Precondițiile trebuie să fie adevarate pentru ca metoda să se execute. Iar în final, după execuția metodei, aceasta garantează că postcondițiile sunt adevărate. Anumite limbaje, precum Eiffel oferă suport direct pentru precondiții și postcondiții. Acestea trebuie doar declarate, iar în timpul rulării sunt verificate automat. În C++ sau Java, această funcționalitate lipsește. Contractele pot fi în schimb specificate prin teste unitare (*unit test*). Prin testarea comportamentului unei clase, testele unitare clarifică comportamentul unei clase. Programatorii care vor folosi clasele pentru care s-au implementat teste unitare, pot folosi aceste teste pentru a ști care este comportamentul pe care îl oferă claselor client.

Concluzii

Principiul LSP este doar o extensie a principiului Deschis-Închis și înseamnă că, atunci când adăugăm o nouă clasă derivată într-o ierarhie de moștenire, trebuie să ne asigurăm că clasa nou adăugată extinde comportamentul clasei de bază, fară a-l modifica.

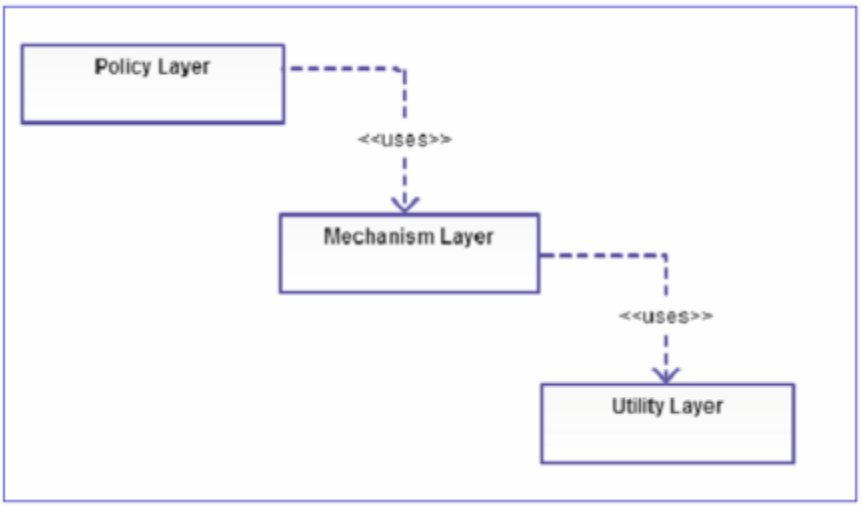
Principiul Inversării Dependenței (Dependency Inversion Principle)

A. 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.

B. Abstractizările nu trebuie să depindă de detalii. Detaliile trebuie să depindă de abstractizări.

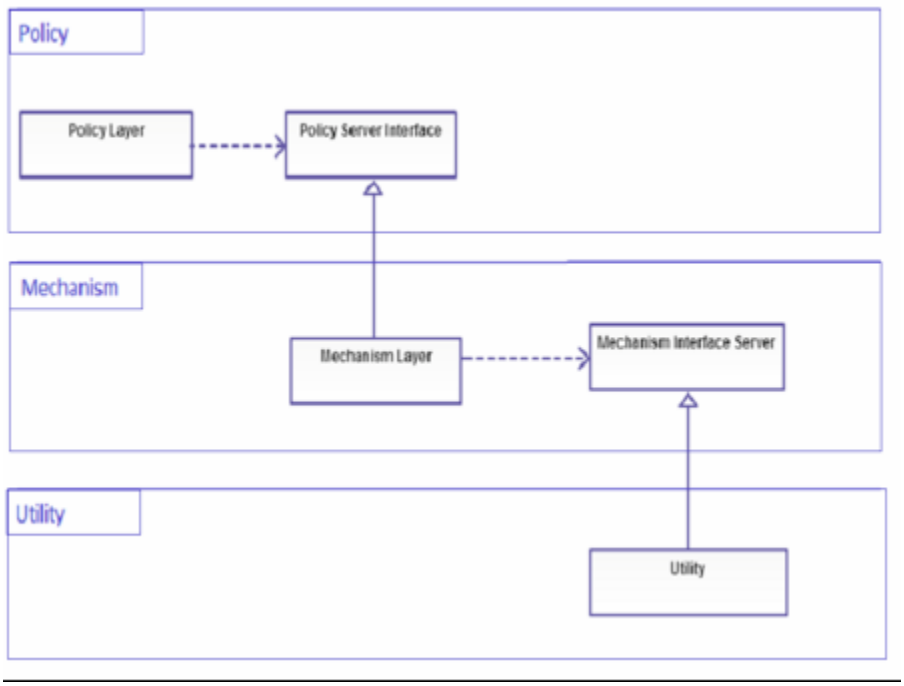
Acest principiu enunță faptul că modulele de pe nivelul ierarhic superior trebuie să fie decuplate de cele de pe nivelurile ierarhice inferioare. Această decuplare se realizează prin introducerea unui nivel de abstractizare între clasele care formează nivelul ierarhic superior și cele care formează nivelurile ierarhice inferioare. În plus principiul spune și faptul că abstractizarea nu trebuie să depindă de detalii, ci detaliile trebuie să depindă de abstractizare. Acest principiu este foarte important pentru reutilizarea componentelor software. De asemenea, aplicarea corectă a acestui principiu face ca întreținerea codului să fie mult mai ușor de realizat.

În *Fig. 6* este prezentată o diagramă de clase organizată pe trei niveluri. Astfel clasa PolicyLayer reprezintă nivelul ierarhic superior, ea accesează funcționalitatea din clasa MechanismLayer aflată pe un nivel ierarhic inferior. La rândul ei, clasa MechanismLayer accesează funcționalitatea din clasa UtilityLayer care de asemenea se află pe un nivel ierarhic inferior. În concluzie, este evident faptul că în diagrama de clase prezentată nivelurile superioare depind de nivelurile inferioare. Acest lucru înseamnă că dacă apare o modificare la unul din nivelurile inferioare există șanse destul de mari ca modificarea să se propage în sus spre nivelurile ierarhice superioare. Ceea ce înseamnă că nivelurile superioare mai abstracte depind de nivelurile inferioare care sunt mai concrete. Așadar se încalcă principiul inversării dependenței.



În *Fig. 7* este prezentată aceeași diagramă de clase ca și în *Fig. 6* , dar de această dată 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. 7* respectă principiul inversării dependenței.



Concluzii

Programarea tradițională procedurală creează polițe de dependențe în care modulele ierarhice superioare depind de detaliile modulelor ierarhice inferioare . Acestă metodă de programare este ineficientă deoarece modificari ale detaliilor conduc către modificări și în modulele superioare.Programarea orientată obiect inversează acest mecanism de dependență, astfel încât atat detaliile cât și nivelurile superioare depind de abstractizări, iar serviciile aparțin deseori clienților.

Indiferent de limbajul de programare folosit, dacă dependențele sunt inversate atunci designul codului este orientat obiect. Dacă dependențele nu sunt inversate, atunci designul este procedural.Principiul dependenței inversate reprezintă mecanismul fundamental de nivel scăzut (*low level*) ce stă la baza multor beneficii oferite de programarea orientată obiect. Respectarea acestui principiu este fundamentală pentru crearea de module reutilizabile. Este de asemenea esențială pentru a scrie cod rezistent la modificari. Atât timp cât abstracțiile și detaliile sunt izolate reciproc, codul este mult mai ușor de menținut.

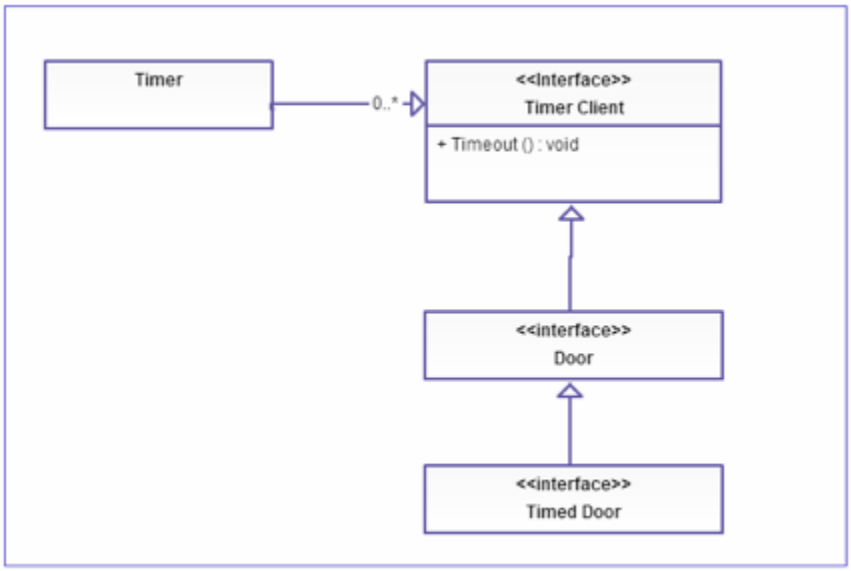
Principiul Segregării Interfeței (The Interface Segregation Principle)

Clienții nu trebuie să depindă de interfețe pe care nu le folosesc.

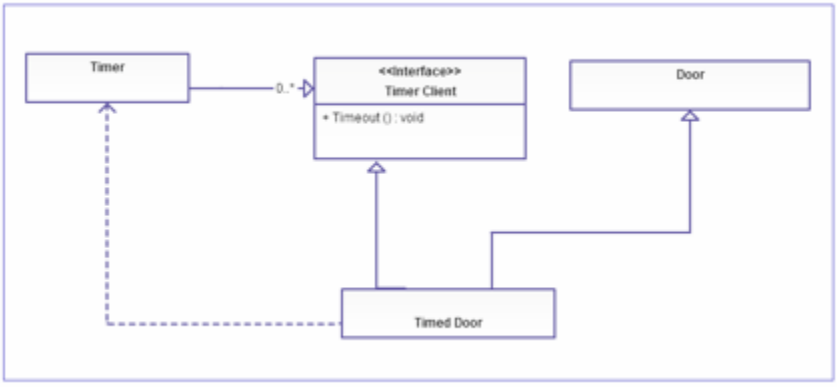
Acest principiu pune în evidență faptul că atunci când se definește o interfață trebuie avut grija ca doar acele metode care sunt specifice clientului să fie puse în interfață. Dacă într-o interfață sunt adăugate metode care nu au ce căuta acolo, atunci clasele care implementează interfața vor trebui să implementeze și acele metode. Spre exemplu, dacă se consideră interfața *Angajat* care are metoda*Mănâncă* , atunci toate clasele care implementează această interfața vor trebui să implementeze și metoda *Mănâncă* . Ce se întâmplă însă dacă Angajatul este un robot? Interfețele care conțin metode nespecifice se numesc interfețe poluate sau grase.În *Fig. 8* este prezentată o diagramă de clase care conține: interfața*TimerClient* , interfața *Door* și clasa *TimedDoor* . Interfața *TimerClient* trebuie implementată de orice clasă care are nevoie să intercepteze evenimente generate de un Timer. Interfața Door trebuie să fie implementată de orice clasă care implementează o ușă. Având în vedere că a fost nevoie de modelarea unei uși care se închide automat după un anumit interval de timp în *Fig. 3.7* este prezentată o soluție în care a fost introdusă clasa *TimedDoor* derivată din interfața*Door* , iar pentru a dispune și de funcționalitatea din *TimerClient* a fost modificată interfața *Door* astfel încât să moștenească interfața *TimerClient* . Aceasta soluție însă poluează interfața *Door* deoarece toate clasele care vor moșteni această interfața vor trebui să implementeze și funcționalitatea din TimerClient [4].

```
class Timer
{
public:
     void Register(int timeout, TimerClient* client);
} ;
class TimerClient
{
public:
    virtual void TimeOut();
};
```

```
class Door
{
 public:
       virtual void Lock() = 0;
    virtual void Unlock() = 0;
    virtual bool IsDoorOpen() = 0;
} ;
```



Separarea interfețelor se poate realiza prin mecanismul moștenirii multiple. În *Fig 9* se poate observa cum moștenirea multiplă poate fi folosită pentru a respecta în design principiul segregării interfețelor. În acest model, interfața *TimeDoor* , moștenește din ambele interfețe *Door* și *TimerClient* .



Concluzii

Clasele poluate sau ,,grase" conduc către cuplări greșite pentru clienții lor. Când un client efectuează o modificare asupra unei clase poluate, toți ceilalți clienți ai clasei poluate sunt afectați. De aceea, clienții trebuie să depindă numai de metodele pe care le apelează efectiv. Acest lucru poate fi realizat prin separarea interfețelor poluate, în mai multe interfețe specifice clienților. Fiecare interfața specifică unui client va conține numai metodele care sunt efectiv apelate de către client. Clasele ,,grase" pot apoi să moștenească toate interfețele specifice clienților și să le implementeze. Acest mecanism decuplează dependența clienților

de metode pe care nu le apelează niciodată și permite clienților să fie independenți unii de alții.

Concluzii Agile Design

Prin aplicarea repetată - de la o iterație la alta, a principiilor mai sus enunțate, se evită cele trei caracteristici care definesc o arhitectură software de slabă calitate:*rigiditatea* - sistemul este greu de modificat pentru că fiecare modificare afectează prea multe părți ale sistemului, *fragilitatea* - dacă se modifică ceva, apar tot felul de erori neașteptate și*imobilitatea* - este dificil să se reutilizeze părți dintr-o aplicație pentru că nu pot fi separate de aplicația pentru care au fost dezvoltate inițial.

Designul *agile* reprezintă un proces care se bazează pe aplicarea *continuă* a principiilor agile și a design *pattern* -urilor astfel încât designul aplicației rămâne în mod constant simplu, curat și cât se poate de expresiv.

Pentru o aprofundare a principiilor enunțate, recomand cu încredere cartea lui Robert C. Martin*, Agile Software Development - Principles, Patterns, and Practices.*

Bibliografie

[1] Robert Martin. - Agile Software Development: Principles, Patterns, and Practices. Editura Prentice Hall. 2012.

[2] Gamma, et al. - Design patterns. reading Ma: Addison-Wesley, 1998.

[3] Liskov, Barbara. - Data Abstraction and Hierarchy. SIGPLAN Notices, 23.5 (May 1988)

[4] Meyer, Bertrand. - Object oriented software construction, 2nd ed. upper Saddle River, Nj: Prentice Hall, 1997.

Procesul SCRUM de dezvoltare agilă a produselor software

Scrum este un proces iterativ şi incremental ce se foloseşte în dezvoltarea produselor *software*. Face parte din categoria *metodelor agile de dezvoltare software*.

Cuprins

[ascunde]

Metodologii de dezvoltare software

De-a lungul anilor au apărut şi evoluat diverse *framework*-uri ce sunt folosite pentru a planifica structura şi procesul de dezvolare a unui sistem informatic (de exemplu dezvolare de produse*software*), denumite metodologii de dezvoltare *software*. În funcţie de fiecare proiect în parte şi de condiţiile de lucru, vom putea aplica o diferită metodologie, cea pe care o vom considera potrivită luând în calcul aspecte tehnice, organizaţionale, sau privitoare la echipa ce se ocupă de respectivul proiect. Câteva dintre metodologiile apărute de-a lungul anilor sunt: programare structurată (anii 1970), ingineria informaţiei (anii 1980), programare orientată obiect, metode de dezvoltare dinamică a sistemelor (anii 1990), Scrum, programare extremă, proces unificat raţional, proces unificat agil (anii 2000).

Dezvoltare de software agilă

Dezvoltarea de software agilă este denumirea unui grup de metodologii de dezvoltare *software* ce sunt bazate pe principii comune. Astfel de metodologii propun un anumit stil al procesului de*management* al proiectelor ce pune accent pe inspecţie frecventă si adaptare, muncă de echipă (ce trebuie incurajată de liderul acesteia), auto-organizare şi responsabilitate, favorizând astfel dezvoltarea rapidă a unui produs

software de mare calitate, precum şi coordonarea procesului de dezvoltare cu cererile clientului şi obiectivele companiei. Cele mai multe metode agile propun dezvoltarea iterată (o iteraţie produce unul sau mai multe pachete complete ce împlinesc o funcţie precisă în cadrul proiectului, iar totalitatea iteraţiilor oferă produsul final), muncă în echipă, colaborare şi adaptabilitate pe întreg parcusul de evoluţie al proiectului. În 2001, 17 persoane cu contribuţii în acest domeniu s-au întâlnit pestru a discuta modalităţi de a produce software într-un mod simplu, rapid, centrat în jurul omului. Astfel au conceput Manifestul Agil ce cuprinde următoarele valori:

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

Cei 17 consideră că, atât timp cât elementele din dreapta au o valoarea, cele din stânga sunt mult mai importante. Pe langă aceste principii, membri Alianţei Agile au propus si 12 principii.

Metode agile

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

- **Programarea extremă (XP)**
- **Scrum**
- **Proces unificat agil (AUP)**
- **Modelare agilă**
- **Proces unificat esenţial (EssUP)**
- **Proces unificat deschis (OpenUP)**
- **Metoda de dezvoltare dinamică a sistemelor (DSDM)**

- **Dezvoltare bazată pe caracteristici (FDD)**
- **Crystal**

Procesul SCRUM

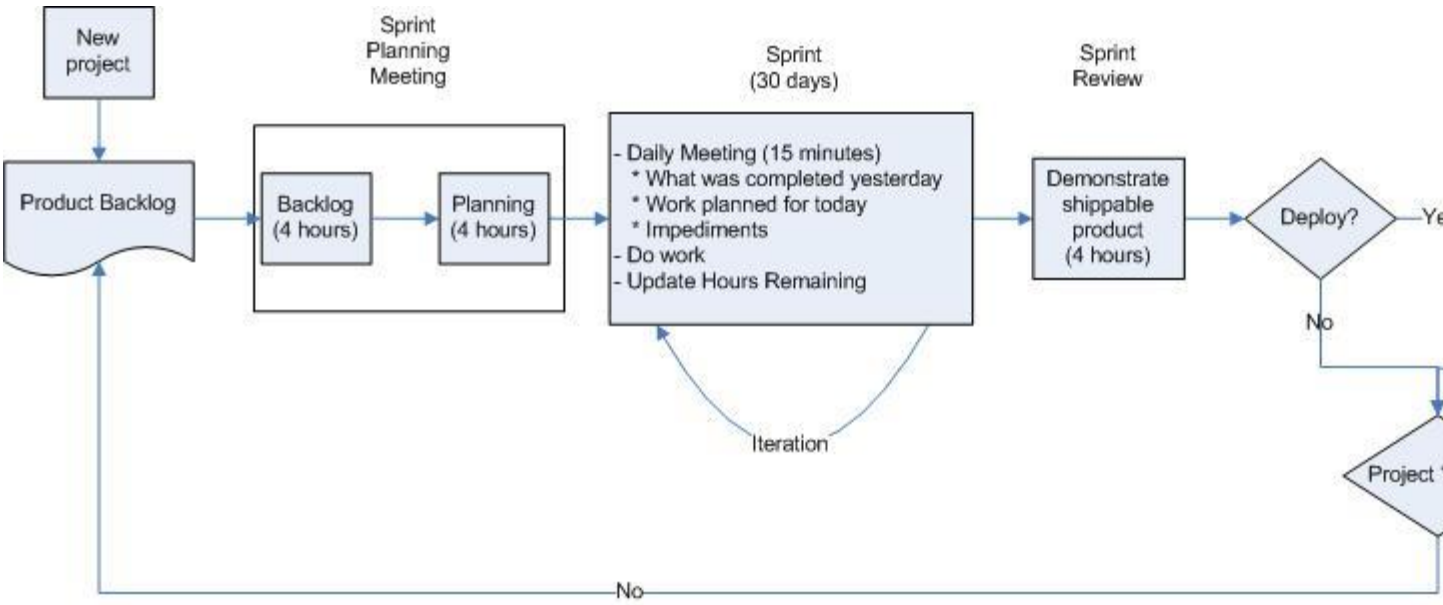
Scrum este o metodă iterativă şi incrementală al cărei scop este cel de a ajuta echipele de dezvoltare să işi concentreze atenţia asupra obiectivelor stabilite şi minimizarea muncii depunse de aceştia pentru rezolvarea sarcinilor mai puţin importante. Scrum doreşte să pastreze simplicitate intr-un mediu de afaceri complicat. Termenul provine din rugby unde reprezintă o strategie de a readuce o minge “pierdută” înapoi în joc folosind munca de echipă. Scrum nu oferă tehnici la nivel de implementare, ci se axează pe modul în care membrii unei echipe de dezvoltare ar trebui să interacţioneze pentru a produce un sistem flexibil, adaptabil şi productiv într-un mediu ce este permanent în schimbare. Cei care au prezentat în detaliu această metodă sunt Schwaber şi Beedle. Scrum se bazează pe două elemente: **autonomia echipei** şi **adaptabilitate**. Autonomia echipei se referă la faptul că cei care conduc proiectul stabilesc sarcinile care trebuie rezolvate de către echipă, însă acesta are libertatea de a-şi stabili propriul mod de lucru în cadrul fiecărei iteraţii, scopul fiind cel de a spori productivitatea echipei.

Caracteristici ale procesui Scrum

Scrum este un proces iterative şi incremental de dezvoltare *software* ce reprezintă planul unui proiect care include un set de activităţi şi roluri predefinite. Principalele roluri sunt cele de*Conducător Scrum* care întreţine procesele şi se comportă ca un project manager, *Deţinător de produs* care reprezintă vocea clientului şi *Echipa* care include dezvoltatorii de *software*. Produsul *software* evoluează de-alungul mai multor etape numite *sprint*-uri. În timpul unui *sprint* Echipa crează o unitate de *software* funţionabil. Elementele de care trebuie să se ţină cont într-un *sprint* se preiau dintr-o mulţime de sarcini nerezolvate. Sarcinile care intră în *sprint* sunt stabilite în urma *întâlnirii de planificare sprint*. Pe parcursul acestei întâlniri Deţinătorul de proiect informează Echipa cu privire la sarcinile nerezolvate pe care doreşte să le abordeze. Echipa stabileşte câte astfet de sarcini poate îndeplini până la următorul sprint. În timpul unui sprint nu se pot schimba sarcinile alese. La sfârşit Echipa demonstrează cum se utilizează produsul intermediar obţinut.

Scrum încurajează crearea echipelor cu auto-organizare şi cominicarea verbală între toţi membrii echipei şi între diferitele departamente care au legătură cu proiectul. Un principiu important în această abordare o reprezintă acceptarea faptului că, pe parcursul dezvoltării unui proiect, clientul se va răzgândi de multe ori cu privire la ce doreşte şi are nevoie să ofere produsul *software*. Astfel de schimbări neprevizibile nu sunt

uşor de adaptat la proiect folosind metodele tradiţionale de dezvoltare software. Scrum adoptă o abordare empirică susţinând că o problemă nu poate fi pe deplin inţeleasă sau definită şi punând accent pe dezvoltarea abilităţii unei echipe de a rezolva rapid noi cerinţe.



Roluri în cadrul unui procesui Scrum

În cadrul unui proces Scrum sunt definite 6 roluri (conform lui Schwaber şi Beedle), fiecare cu diferite sarcini şi scopuri. Aceste roluri sunt împărţite în două categorii:

- **Porcii** – cei direct implicaţi în procesul de dezvoltare, angajaţi să construiască proiectul şi care sunt traşi la răspundere.
  - **Conducătorul Scrum** – are un rol de *project manage*r (dar el nu este şeful echipei) ce trebuie să se asigure că procesul de dezvoltare evoluează în conformitate cu tehnicile, valorile şi regulile Scrum. Acesta interacţionează atât cu Echipa de dezvoltare, cât si cu clienţii şi conducerea organizaţiei. Este de asemenea responsabil să se asigure că orice impediment şi orice element care distrage atenţia echipei sunt înlăturate, astfel încât productivitatea echipei să fie permanent la un nivel ridicat.
  - **Deţinătorul de produs** – reprezintă vocea, interesele clientului. El este responsabil de proiectarea, administrarea, controlul şi prezentarea produsului nerezolvat; ia decizia finală cu privire la sarcinile din produsului nerezolvat şi le asociază priorităţi. Este ales de către Conducătorul Scrum, client şi conducere.

- **Echipa** – este responsabilă cu dezvoltarea produsului; are autoritatea de a decide ce măsuri trebuie luate pentru a rezolva sarcina asociată fiecărui sprint şi are dreptul de a se auto-organiza tot în acelaşi scop. În general o echipă Scrum este alcătuită din 5-9 persoane.

- **Puii** - cei care nu sunt implicaţi direct în dezvoltarea proiectului, dar de a căror părere trebuie să se ţină cont. În abordarea agilă un aspect foarte important îl reprezintă implicarea utilizatorilor, clienţilor, oamenilor de afaceri în procesul de dezvoltare. Aceştia trebuie să ofere *feed-back* cu privire la rezultatele fiecărui sprint pentru a adapta şi îmbunătăţi viitoarele procese de lucru.
  - **Utilizatorii** – cei care vor folosi produsul *software*
  - **Clienţii** – cei care stabilesc scopul proiectului; sunt implicaţi în procesul de dezvoltare doar când are loc evaluarea unui sprint
  - **Manager-ii** – cei responsabili de luarea deciziilor finale. Participă de asemenea în stabilirea obiectivelor şi a condiţiilor de lucru

Elemente Scrum

Scrum nu propune tehnici specifice de dezvoltare *software*, ci anumite metode şi instrumente referitoare la *management*, în diferite faze Scrum, pentru a evita confuzia creată de imprevizibilitatea şi complexitatea proiectelor. Elementele caracteristice metodei Scrum sunt:

- **Sprint-ul** – perioadă de 15-30 zile (durata exactă este stabilită de către Echipă). El include faze tradiţionale de dezvoltare software precum etape de cerinţe, analiză, design, evoluţie, livrare. Echipa Scrum se organizează astfel încât să producă o nouă unitate funţionabila a produsului la sfărşitul unui *sprint*. Cu ajutorul acestora, sistemul se adaptează mai uşor la schimbări.

- **Produsul nerezolvat** – mulţime de sarcini nerezolvate: descrieri ale funcţionalităţilor si serviciilor dorite – tot ce este nevoie pentru a atinge obiectivul final – aşa cum este el definit în prezent; poate fi modificat de către oricine. Sarcinile au asociate priorităţi de către Deţinătorul de produs în funcţie de valoarea lor din punct de vedere al afacerii (valoare stabilită de către Deţinătorul de produs), şi de efortul necesar dezvoltării acestora (stabilit de către Echipă). Este actualizat în mod constant prin adăugare, modificare, specificare, înlăturare, stabilire de priorităţi cu privile la elementele conţinute. Câteva dintre elementele ce pot face parte produsul nerozolvat sunt implementarea anumitor funcţii, remedierea *bug*-urilor, înlăturarea defectelor, îmbunătăţirea diferitelor componente. Printre cei care

pot participa la construirea acestui produs sunt clienţii, Echipa de dezvoltare, echipa de *marketing* şi vânzări. Deţinătorul de produs este cel responsabil cu administrarea produsului nerezolvat.

- **Sprint-ul nerezolvat** –reprezintă un document, detaliat, bazat pe elementele din produsul nerozolvat ce vor fi adresate în următorul *sprint*; conţine informaţii despre modul în care Echipa va implementa cerinţele stabilite. Sarcinile sunt împărţite pe ore astfel încât nici o sarcină să dureze mai mult de 16 ore (daca o sarcină ar dura mai mult, ea trebuie împărţită în sarcini mai mici). Sarcinile nu sunt repartizate angajaţilor – aceştia au libertatea de a-şi alege sarcinile dorite. Cei care stabilesc ce elemente vor face parte din *sprint*-ul nerezolvat sunt Conducătorul Scrum, Deţinătorul de produs şi Echipa în cadrul *întâlnirii de planificare sprint* pe baza priorităţilor şi a obiectivelor stabilite pentru acel *sprint*. *Sprint*-ul nerezolvat este stabil până când sprint-ul este terminat. Când toate sarcinile din *sprint*-ul nerezolvat sunt împlinite o nouă iteraţie a sistemului este finalizată.

- **Burn down** - este un grafic afişat la vedere ce reprezintă timpul şi munca rămasă până la terminarea proiectului. Este actualizat zilnic şi ajută la estimarea datei la care va gata produsul.

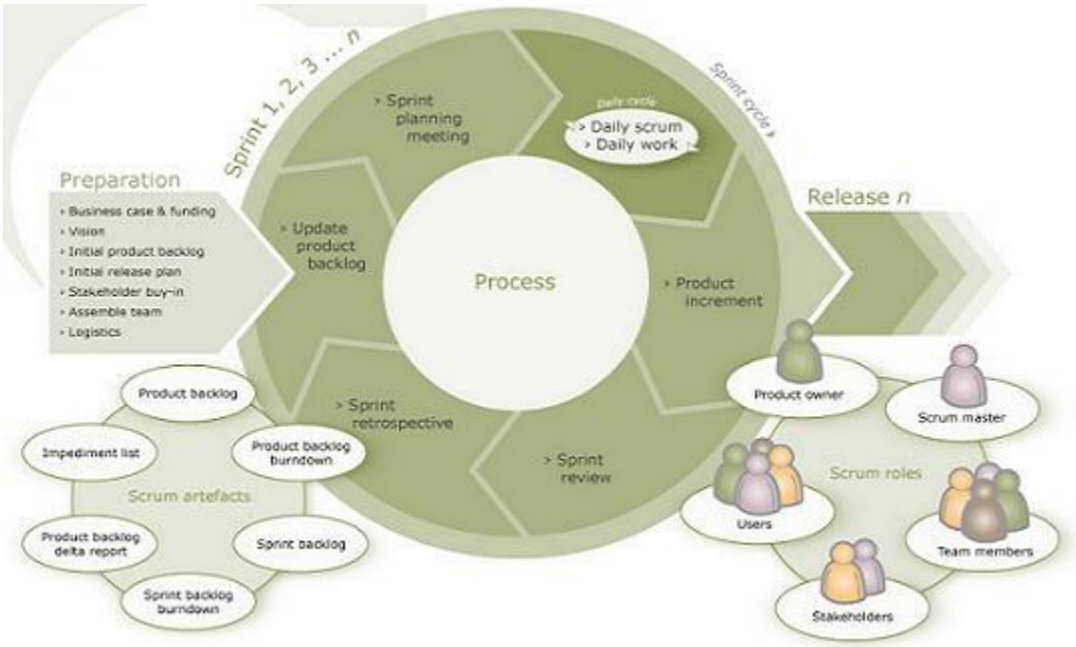
- **Estimarea efortului** – este proces iterativ în care se concentrează atenţia spre estimarea cât mai precisă a efortului depus pentru tratarea unei sarcini nerezolvate atunci când există mai multe informaţii despre acea sarcină.

- **Întâlnirea de planificare a unui sprint** – reprezintă o întâlnire în două etape organizată de Conducătorul Scum. La prima parte a întâlniri iau parte clienţii, utilizatorii, conducerea, Deţinătorul de produs şi Echipa, pentru a decide obiectivele şi funţionalitatea următorului *sprint*. La a doua parte a întâlnirii participă Conducătorul Scrum şi Echipa pentru a discuta despre cum se va implementa unitatea produsului reprezentată de acest *sprint*.

- **Întâlnirea Scrum zilnică** – este organizată pentru a urmări continuu progresul echipei; deserveşte de asemenea şi ca întâlnire de planificare: se vorbeşte depsre ce s-a făcut de la ultima întâlnire şi până acum şi ce se va face de acum până la următoarea întâlnire; se discută şi despre problemele şi impedimentele care au afectat membri echipei în ceea ce priveşte atingerea obiectivelor stabilite. Conducătorul Scrum este cel care conduce această întâlnire ce durează aproximativ 15 minute. Fiecare întâlnire urmăreşte anumite reguli:
  - Întâlnirea începe la timp; de multe ori întârzierile se pedepsesc
  - Toţi sunt bineveniţi, însă doar “porcii” au voie să vorbească
  - Întâlnirea durează 15 minute indiferent de numărul de participanţi
  - Toti participanţi ar trebui să stea jos

- Toate întâlnirile ar trebui să aibă loc în fiecare zi în acelaşi loc şi la aceeaşi oră

- **Întâlnirea de evaluare a unui sprint** – în ultima zi a *sprint*-ului Echipa împreună cu Conducătorul Scrum prezintă conducerii, clienţilor, utilizatorilor şi Deţinătorului de produs, rezultatul*sprint*-ului, într-o întâlnire neformală. Participanţii la întâlnire evaluează rezultatul şi iau decizii cu privire la direcţiile ce trebuie urmate în continuare. Durează cel mult 4 ore. Se pun două întrebări principale: “Ce a mers bine în timpul *sprint*-ului?” si „Ce se poate îmbunătăţi în următorul *sprint*?”



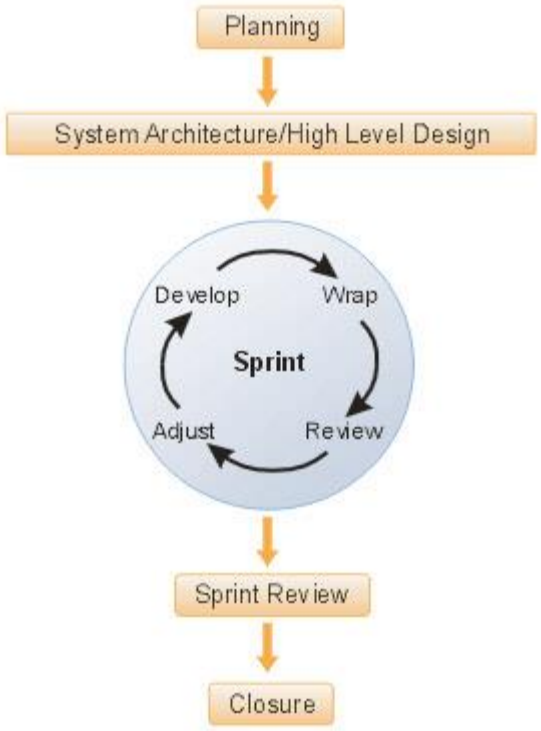
Procesul Scrum

Procesul Scrum include 3 faze (conform lui Schwaber şi Beedle): pre-joc, dezvoltare (sau joc), post-joc.

- **Pre-jocul** – include două sub-faze:
  - *Planificare*: presupune definirea sistemului ce se doreşte a fi construit. Este creat un produs nerezolvat ce conţine cerinţele cunoscute la momentul actual. Cerinţelor li se asociată priorităţi şi este evaluat efortul necersar pentru implementarea acestora. Tot acum se stabileşte şi echipa ce va lucra la proiect, intrumentele şi resursele necesare, ariile în care este nevoie de training.
  - *Arhitectura / Design la nivel înalt* – aici are loc design-ul de nivel înalt al sistemului; dacă sistemul deja există se discută schimbările necesare pentru implementarea cerinţelor, precum şi problemele pe care le ridică aceste schimbări.

- **Jocul** – reprezintă partea agilă a abordării Scrum. Această fază este tratată ca o cutie neagră unde unele elemente sunt imprevizibile. Variabilele identificate ce ţin de mediul de devoltare, care se pot schimba de-a lungul timpului, sunt observate şi controlate prin diverse practici Scrum. Scrum ia în considerare aceste variabile nu doar la început (în faza de pre-joc), ci pe tot parcursul procesului de dezvoltare pentru a se putea adapta uşor la schimbări. În această fază proiectul este realizat prin intermediul *sprint*-urilor.

- **Post-jocul** – este faza de finalizare a proiectului; toate cerinţele stabilite au fost îndeplinite. În acest stadiu nu mai sunt elemente sau probleme (în produsul nerezolvat) ce trebuie adresate şi nici nu se mai pot găsi altele noi. Produsul este gata pentru a fi livrat şi se pregăteşte această acţiune (prin integrare, testare, documentare).



Bibliografie

1. Agile Manifesto
2. Jonna Calermo and Jenni Rissanen, *Agile software development in theory and practice*
3. Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, Juhani Warsta, *Agile software development methods* (2002)

4. Ken Schwaber, *Scrum Development Process* (1995)
5. Ken Schwaber, Mike Beedle, *Agile software development with Scrum* (2002)
6. Wikipedia

Agilitatea este un set de valori și principii, o cultură. Agilitatea nu înseamnă Scrum sau Extreme Programming. Introducere la manifestul agil.

by Cornel Fatulescu on August 13, 2014 • 0 Comments

Recent am avut o discuție pe Facebook legată de aceleași neînțelegeri despre ce înseamnă să fii agil. Discuția roia în jurul articolului lui Robert Martin despre adevărata corupție a agilității. Robert susține că agilitatea este **o cultură exprimată printr-un set de practici**, contrazicându-l astfel pe Allen Holub, și nu numai, care scrisese că **agilitatea este o cultură, nu un set de practici**. Este important de reținut că până aici amândoi recunosc agilitatea ca fiind **o cultură, nu o metodologie**.

Termenul de **agil** și noțiunea de **dezvoltare agilă de software** au apărut împreună cu **manifestul agil**publicat în 11-13 februarie 2001 și neschimbat de atunci. Autorii, autointitulați anarhiștii organizaționali ai epocii, reprezentau diferitele cadre de lucru cunoscute deja la acea vreme, acum considerate ca fiind agile:Extreme Programming (XP), Scrum, DSDM, Adaptive Software Development, Crystal, Feature-Driven Development, Pragmatic Programming, sau simpatizau cu ideea unei alternative la dezvoltarea software caracterizată de documentație excesivă și de procese greoaie.

Eu am auzit de manifest la mult timp după apariția sa, mai precis prin 2005. La fel ca orice ținea de valori, cultură organizațională și leadership, le-am parcurs rapid, n-am înțeles nimic și am trecut mai departe. Nici acum nu sunt multe organizații în România care să-și fi oficializat un sistem de valori pe care să-l conștientizeze și să-l încarneze în activitățile zilnice. Lipsa de încredere a românilor într-o astfel de abordare este obstacolul major în adopția unui management bazat pe o cultură organizațională, implicit și în leadership sau în adopția agilității.

Despre practicile agile

Din punctul meu de vedere, agilitatea este o cultură **care se reflectă și** printr-un set de practici**.** Însă doar aplicarea acelor practici nu ne face automat agili. Motiv pentru care nu sunt de acord cu articolul unchiului Bob. Aceste practici evoluează odată cu trecerea timpului. În 2001 nu se vorbea despre **behavior-driven development,** practică ce-și găsește originile în TDD și care a fost îmbrățișată imediat de comunitatea agilă. Asta nu-i face pe cei care practicau XP, Scrum sau DSMD, mai puțin agili decât sunt astăzi. La fel :

- nici cei care estimează în valori absolute nu sunt mai puțini agili decât cei care estimează în valori relative,
- nici cei care folosesc o tablă virtuală de vizualizare a progresului nu sunt mai puțin agili decât cei care folosesc un tablou fizic lipit pe perete,
- nici cei care lucrează bazându-se pe cadrul Kanban nu sunt mai puțin agili decât cei care folosesc cadrul Scrum,
- nici cei care folosesc use case nu sunt mai puțin agili decât cei care folosesc user story,
- etc.

Lista cu exemple de practici de inginerie fără de care putem fi agili în continuare putea crește foarte mult. Înainte să folosim orice practică este bine să ne punem următoarele întrebări:

- de ce folosim practica X și nu practica Y?
- care sunt avantajele? dar dezavantajele?
- am încercat și …?

Însă pot fi de acord cu, Martin Fowler, care, mai puțin incisiv decât Robert Martin, ne aduce aminte că nu se poate Scrum fără valorile și principiile agile și că în timpul tranziției către Scrum n-ar trebui să neglijăm cel de-al nouălea principiu din manifestul agil: *Atenția continuă pentru excelentă tehnică și design bun îmbunătățește agilitatea*.

Nu este vorba doar de practici

Efectele asimilării corecte a valorilor și principiilor agile nu sunt reprezentate doar de practicile de inginerie utilizate de o echipă de dezvoltare. Nu degeaba în Scrum rolurile sunt definite precis. S-a constatat că organizațiile agile au suferit schimbări profunde

- în strategie,
- în structură,
- în procese,
- în sistemul de recompense și
- în politicile de resurse umane.

Astfel, mereu când se vorbește de tranziția către agilitate, Craig Larman, Mike Cohn, Jeff Sutherland – doar ca să dau câteva nume cu rezonanță, vorbesc de abordările concomitente:

- atât de sus în jos (top-down),
- cât și de jos în sus (bottom-up).

O echipă va fi mereu limitată în adopția agilității de organizația din care face parte.

Mai degrabă să fim agili decât să practicăm agilitatea

În cartea Scaling Lean & Agile Development, există un capitol numit „**să fii agil**”. Pornind de la motto-ul *„Be*agile rather than *do* agile”, Craig Larman explică:

„Agilitatea” nu înseamnă o practică. Este calitatea organizației și a oamenilor săi

- să se poată adapta,
- să fie receptivi,
- să învețe în continuu și
- să evolueze.

- să fie agili, având ca obiectiv succesul competitiv în afaceri și livrarea rapidă a produselor și cunoștințelor valoroase din punct de vedere economic. Agilitatea nu poate fi practicată de unul singur, deși este o neînțelegere des întâlnită cum că s-ar putea. Astfel de neînțelegeri sunt legate de ideea că agilitatea înseamnă mod de lucru iterativ sau XP.

…

Un grup care lucrează la un produs poate practica Scrum sau XP – metodologii concrete. Și pot să utilizeze practici care să încurajeze agilitatea – **practicile agile**. Dar grupul poate doar să fie agil sau doar să nu fie.

…

Agilitatea nu înseamnă să livrezi mai repede. Agilitatea nu înseamnă mai puține defecte sau calitate ridicată. Agilitatea nu înseamnă productivitate mai mare. Agilitatea înseamnă să fii agil – abilitatea de a te mișca cu grație, rapid și ușor, să fii sprinten și să te poți adapta. Să îmbrățișezi schimbarea și să devii maestru al schimbării – să întregești prin adaptabilitate, fiind capabil să schimbi mai repede decât pot concurenții.

…

În plus, este vital să înțelegem că agilitatea organizațională nu poate fi atinsă doar de o echipă de dezvoltare izolată. Este o provocare a sistemului de redefinire a organizației.

Și continuă așa până la manifestul agil și valorile și principiile definite în acesta.

Dezvoltarea agilă este bazată pe un set de valori – nu de practici – care suportă și încurajează ceea ce înseamnă să fii agil.

Craig Larman nu uită să menționeze valorile Scrum și principiile managementului agil, dar care nu fac parte din scopul acestui articol și despre care voi vorbi în altă zi.

Aceleași idei le-am dezbătut și eu în articolul „Gânduri despre agilitate”.

Manifestul pentru dezvoltarea agilă de software

Putem verifica cât suntem de agili revenind regulat la valorile și principiile descrise în acest manifest și întrebându-ne dacă suntem aliniați întocmai.

Cele patru valori din manifestul pentru dezvoltarea agilă de software

Noi scoatem la iveală modalități mai bune de dezvoltare de software prin experiență proprie și ajutându-i pe ceilalți.
Prin această activitate am ajuns să apreciem:

1. **Indivizii și interacțiunea** înaintea proceselor și uneltelor
2. **Software funcțional** înaintea documentației vaste
3. **Colaborarea cu clientul** înaintea negocierii contractuale
4. **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.

Cele 12 principii ale manifestului

Noi urmăm aceste principii:

1. Prioritatea noastră este satisfacţia clientului prin livrarea rapidă şi continuă de software valoros.
2. Schimbarea cerinţelor este binevenită chiar şi într-o fază avansată a dezvoltării. Procesele agile valorifică schimbarea în avantajul competitiv al clientului.
3. Livrarea de software funcţional se face frecvent, de preferinţă la intervale de timp cât mai mici, de la câteva săptămâni la câteva luni.
4. Oamenii de afaceri şi dezvoltatorii trebuie să colaboreze zilnic pe parcursul proiectului.
5. Construieşte proiecte în jurul oamenilor motivaţi. Oferă-le mediul propice şi suportul necesar şi ai încredere că obiectivele vor fi atinse.
6. Cea mai eficientă metodă de a transmite informaţii înspre şi în interiorul echipei de dezvoltare este comunicarea faţă în faţă.
7. Software funcţional este principala măsură a progresului.

8. Procesele agile promovează dezvoltarea durabilă. Sponsorii, dezvoltatorii şi utilizatorii trebuie să poată menţine un ritm constant pe termen nedefinit.
9. Atenţia continuă pentru excelenţă tehnică şi design bun îmbunătăţeşte agilitatea.
10. Simplitatea–arta de a maximiza cantitatea de muncă nerealizată–este esenţială.
11. Cele mai bune arhitecturi, cerinţe şi design emerg din echipe care se auto-organizează.
12. La intervale regulate, echipa reflectă la cum să devină mai eficientă, apoi îşi adaptează şi ajustează comportamentul în consecinţă.

Valorile și principiile de mai sus sunt extrase din manifestul pentru dezvoltarea agilă de software publicat la adresa http://agilemanifesto.org/iso/ro/

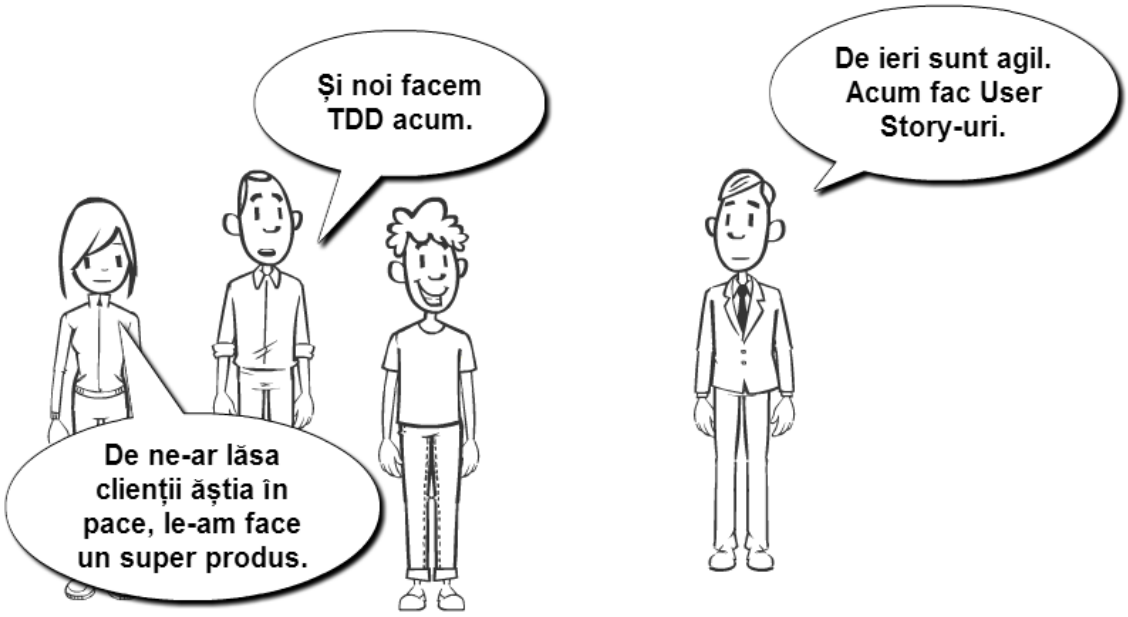
Tranziția către agilitate

Nu știu cine zicea că agilitatea este o călătorie nu o destinație, dar mare dreptate avea. Mike Cohn a scris recent un articol: adevărata cale să fii agil. El nu vine cu un răspuns ferm, dar crede că adevărata cale să fii agil ar putea fi cea care funcționează cel mai bine pentru echipă. În comentarii se abordează chiar și ideea compromisurilor din partea unei echipe din organizație pentru binele suprem: organizația să fie agilă.

Dacă totul era atât de simplu, toate organizațiile erau deja agile.

Concluzie

Deci, cum răspundem la întrebarea: Ce este agilitatea? Corect, este un set de valori și principii, o cultură.



Agilitatea este un set de valori și principii, o cultură. Agilitatea nu înseamnă Scrum sau Extreme Programming. Introducere la manifestul agil.

Articol scris de Cornel Fătulescu. Găsiți mai multe informații despre Cornel Fătulescu pe pagina membrilor AgileHub, în articolul despre mine sau la pagina de contact.

Joc monezi:

Multe monezi: 50, 10, 5 bani.

Se aleg 20 la intamplare (sau dupa dorinta)

Se intorc toate cap.

Se impart echipele astfel ca unul intoarce si altul determina timpul de intoarcere la fiecare membru in parte. Stau in jurul unei mese toti circular.

Unul masoara intreg timpul de intoarcere a tuturor monezilor.

Se noteaza toti timpii pe table la fiecare membru intr-un tabel si in final un timp de start altul de stop, cand se intorc toate monezile. Se face si o medie.

Intai fiecare membru intoarce toate monezile si se noteaza timpii la fiecare membru in parte.

Apoi se vede daca nu e bine sa fie monezile de acelas tip si se reiau masuratorile inca odata.

Apoi se intorc 10 care se paseaza mai departe, apoi celelalte 10 si se paseaza mai departe.

Se masoara timpul in care se termina (2 membrii) intoarcerea completa. Se pot face divizari de 5, 2 monezi, care intoarse se paseaza mai departe.

Se ajunge la o moneda cre intoarsa e pasata mai departe. Se repeta de 3 ori si se analizeaza timpii individuali si cei globali cand se intoarce ultima moneda.

Se trag concluzii la lucrul individual, in echipa cu taskuri mari (2 fac o echipa) la taskuri mici, 1 moneda.

Se urmaresc principiile Agile.