



UNIVERSITATEA  
DIN BUCUREŞTI

# Metode de dezvoltare software

---

## Procese de dezvoltare software

27.02.2019

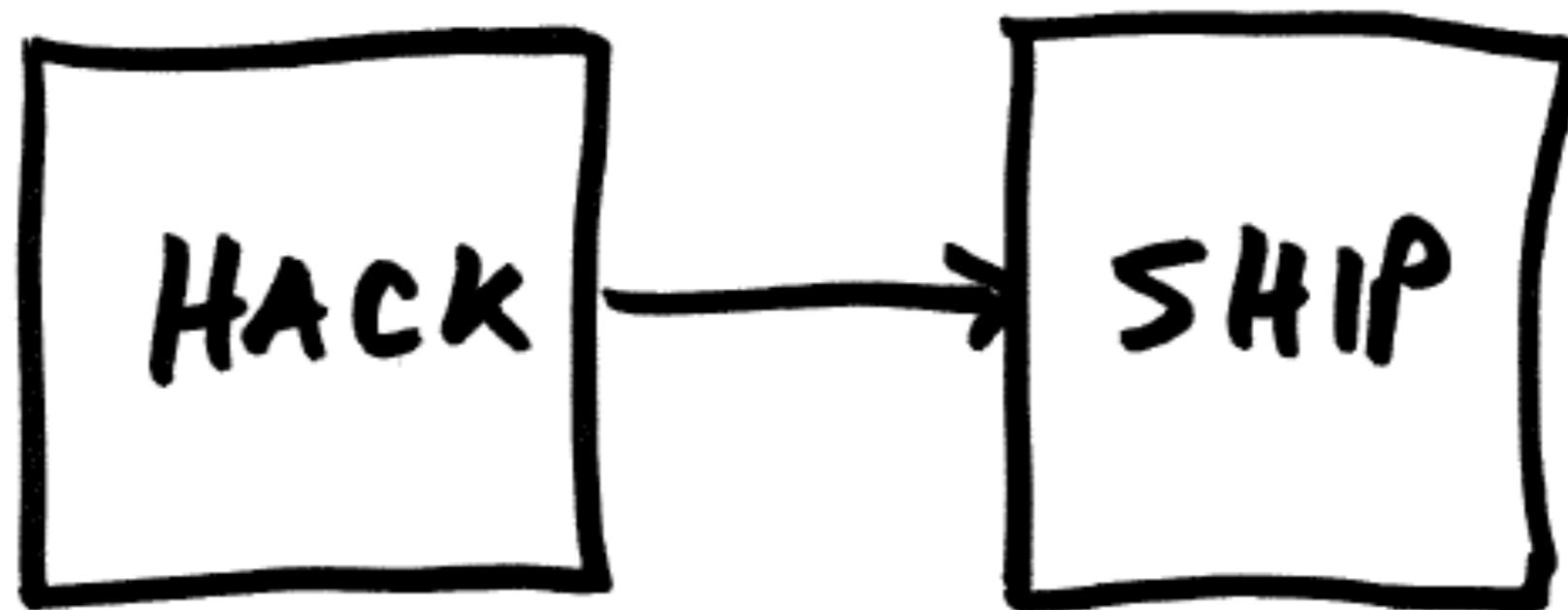
Alin Ţăfărescu



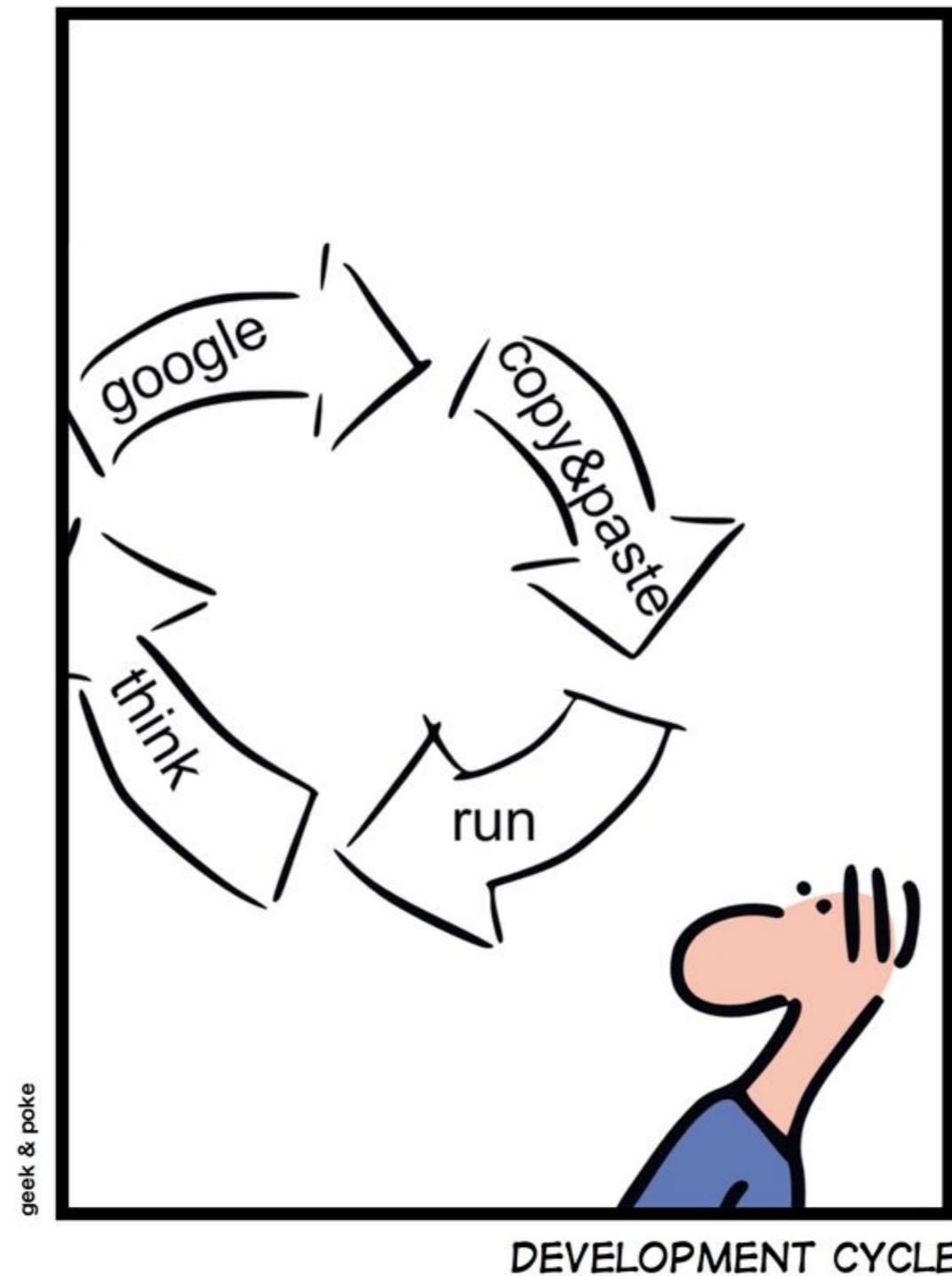


Prezentare bazată pe materiale de: Florentin Ipate (FMI UniBuc) - note de curs  
și Ian Sommerville: Software Engineering 10th edition

# Procese de dezvoltare software... în practică



# Varianta detaliată



Cerinte

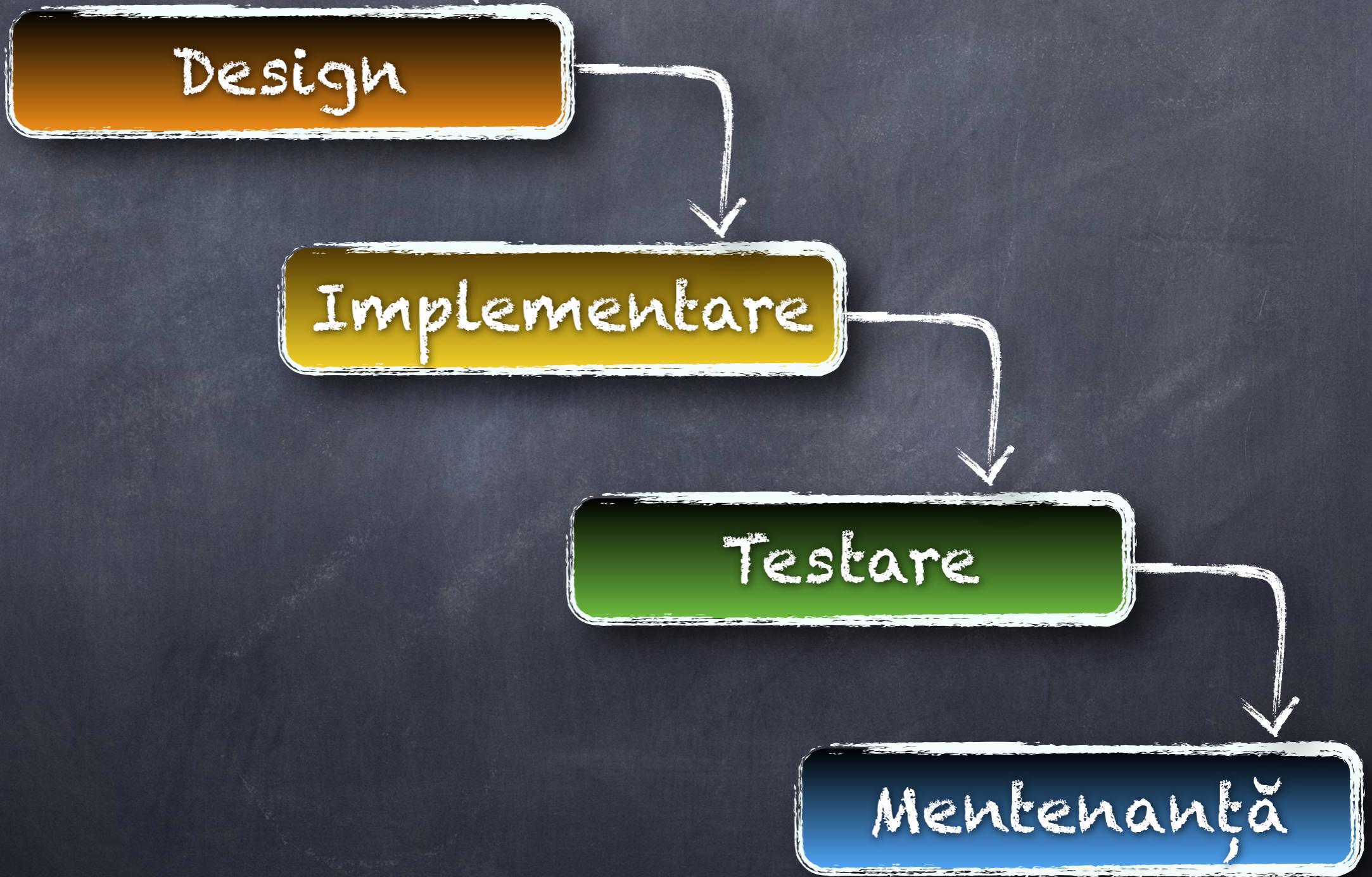
Procesul de dezvoltare  
"cascadă" (waterfall)

Design

Implementare

Testare

Mențenanță



# Etapele procesului “cascadă”

- **analiza și definirea cerințelor:** Sunt stabilite serviciile, constrângerile și scopurile sistemului prin consultare cu utilizatorul. (**ce** trebuie să facă sistemul).
- **design:** Se stabilește o arhitectură de ansamblu și funcțiile sistemului software pornind de la cerințe. (**cum** trebuie să se comporte sistemul).
- **implementare și testare unitară:** Designul sistemului este transformat într-o mulțime de programe (unități de program); testarea unităților de program verifică faptul că fiecare unitate de program este conformă cu specificația.
- **integrare și testare sistem.** Unitățile de program sunt integrate și testate ca un sistem complet; apoi acesta este livrat clientului.
- **operare și mențenanță.** Sistemul este folosit în practică; mențenanța include: corectarea erorilor, îmbunătățirea unor servicii, adăugarea de noi funcționalități.

# Avantaje și dezavantaje

- fiecare etapă nu trebuie să înceapă înainte ca precedenta să fie încheiată
- fiecare fază are ca rezultat unul sau mai multe documente care trebuie “aprobată”
- bazat pe modele de proces folosite pentru productia de hardware

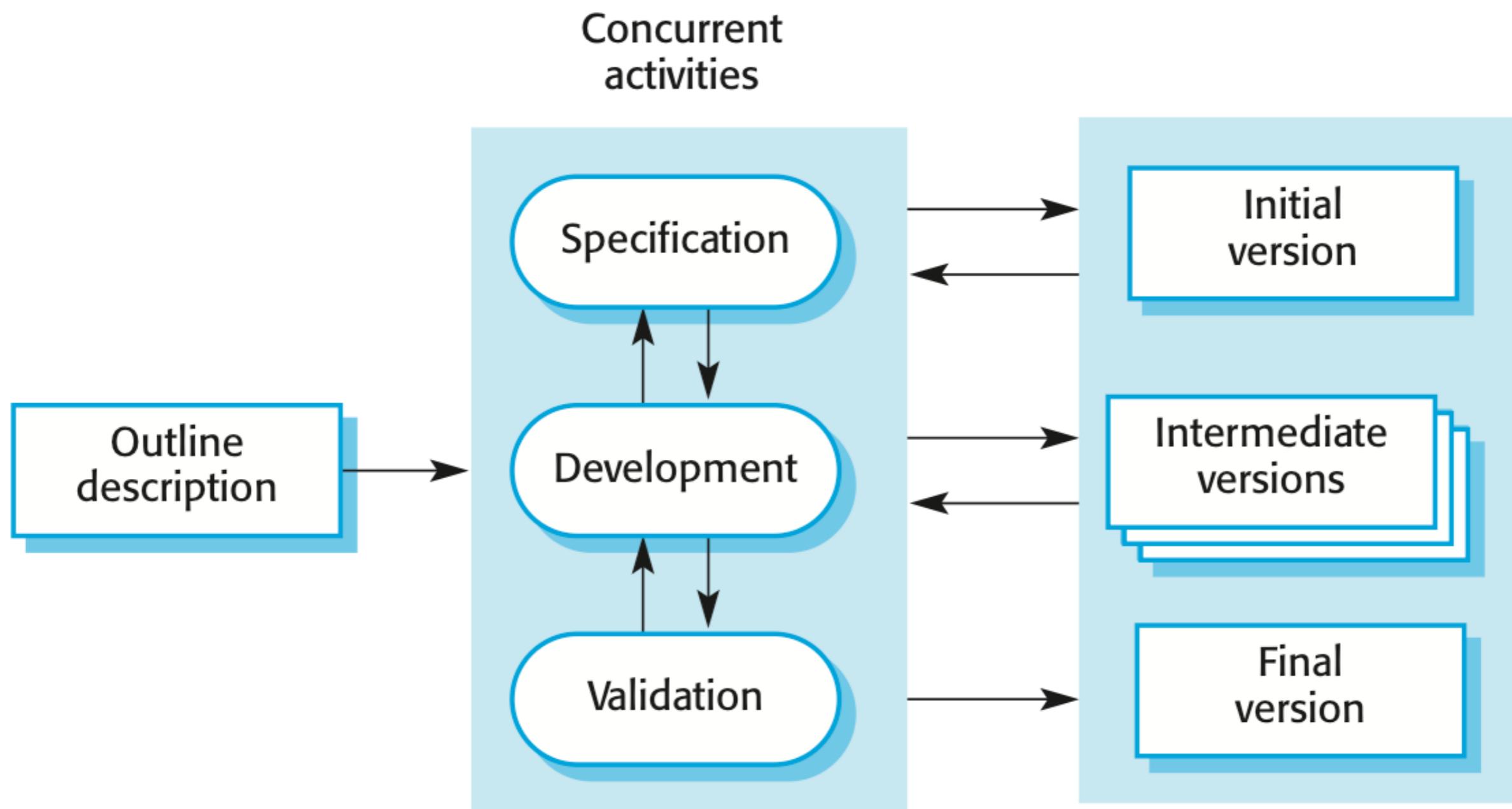
**Avantaj:** proces bine structurat, riguros, clar; produce sisteme robuste

- Probleme:
  - dezvoltarea unui sistem software nu este de obicei un proces liniar; etapele se întrepătrund
  - metoda oferă un punct de vedere **static** asupra cerințelor
  - schimbarile cerințelor nu pot fi luate în considerare după aprobarea specificației
  - nu permite implicarea utilizatorului după aprobarea specificației

**Concluzie:** Modelul cascadă trebuie folosit atunci cand cerințele sunt bine înțelese și când este necesar un proces de dezvoltare clar și riguros.

# Procesul incremental

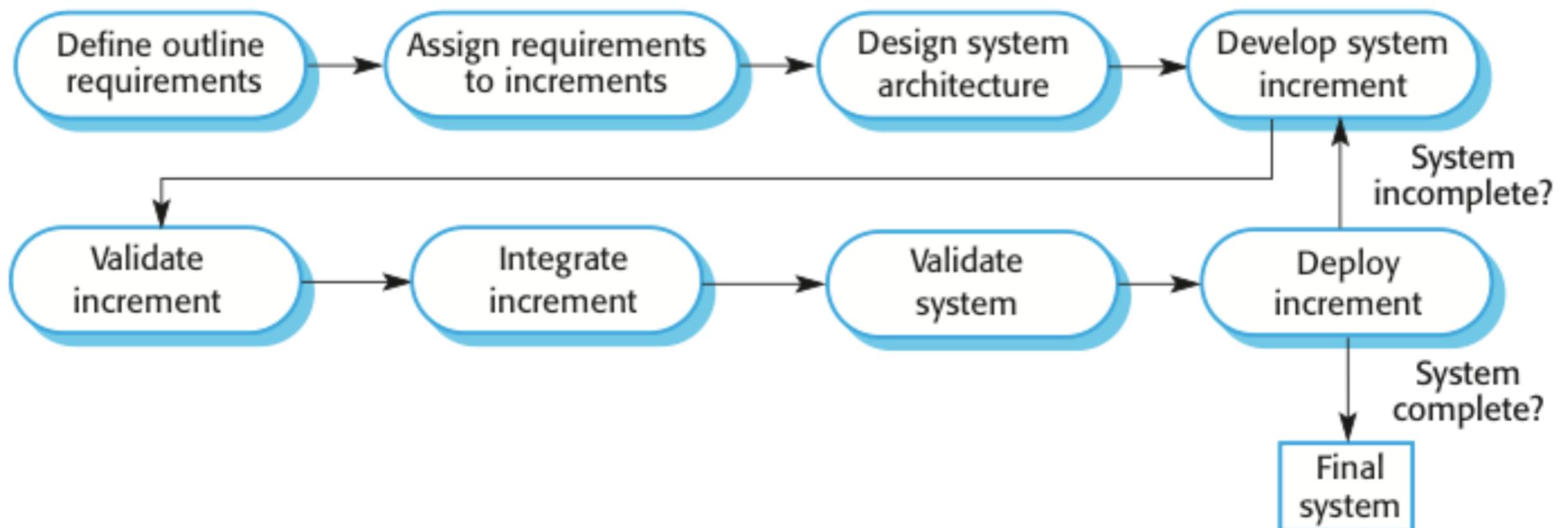
**Idee**: "un sistem de succes de mare dimensiune începe cu un sistem de succes de mică dimensiune care apoi crește puțin câte puțin" (Gilb, 1988)



# Procesul incremental

- sunt identificate cerințele sistemului la nivel înalt, dar, în loc de a dezvolta și livra un sistem dintr-o dată, dezvoltarea și livrarea este realizată în părți (**incremente**), fiecare increment încorporând o parte de funcționalitate.
- cerințele sunt ordonate după **priorități**, astfel încât cele cu prioritatea cea mai mare fac parte din primul increment, etc.
- după ce dezvoltarea unui increment a început, cerințele pentru acel increment sunt înghețate, dar cerințele pentru noile incremente pot fi modificate.

# Procesul incremental (detaliat)



# Avantajele procesului incremental

## Avantaje

- clienții nu trebuie să aștepte până ce întreg sistemul a fost livrat pentru a beneficia de el. Primul increment include cele mai importante cerințe, deci sistemul poate fi folosit imediat.
- primele incremente pot fi prototipuri din care se pot stabili cerințele pentru următoarele incremente.
- se micșorează riscul ca proiectul să fie un eșec deoarece părțile cele mai importante sunt livrate la început.
- deoarece cerințele cele mai importante fac parte din primele incremente, acestea vor fi testate cel mai mult.

## Probleme

- dificultăți în transformarea cerințelor utilizatorului în incremente de mărime potrivită.
- procesul nu este foarte vizibil pentru utilizator (nu e suficientă documentație între iterații).
- codul se poate degrada în decursul ciclurilor.

# Exemplu de procese incrementale

Există multe variante de procese de dezvoltare incrementale:

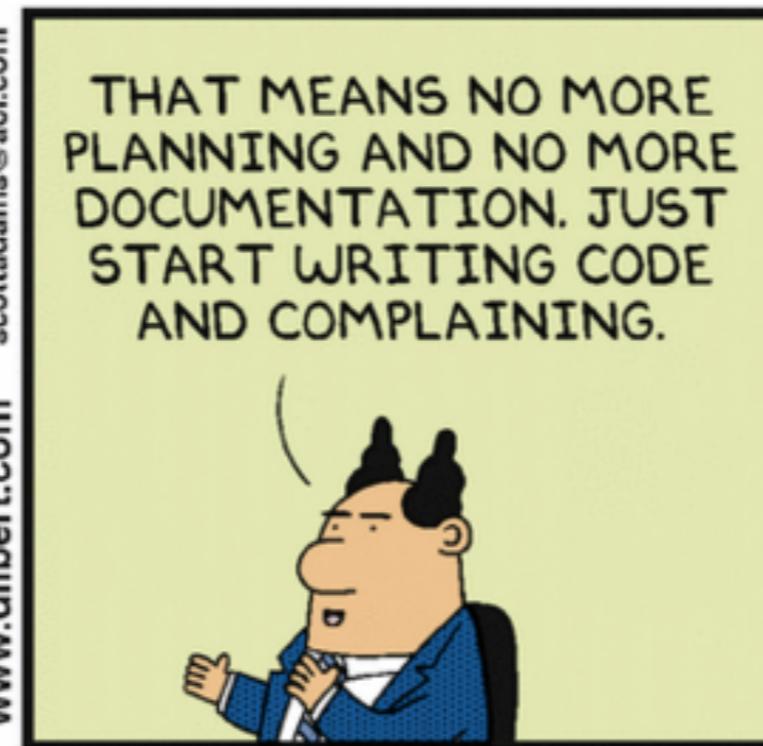
- Unified Process cu varianta Rational Unified Process (RUP)
- procese de dezvoltare agile. Exemple:
  - programare extremă (XP - extreme programming)
  - Scrum



## Metodologii agile

Prezentare bazată pe “Ian Sommerville: Software Engineering 10th edition”

# Procese agile... în practică



# Metodologii “agile”

- scopul metodelor agile este de a reduce cheltuielile în procesul de dezvoltare a software-ului (de exemplu, prin limitarea documentației) și de a răspune rapid cerințelor în schimbare.
- aceste metode:
  - se concentrează mai mult pe cod decât pe proiectare
  - se bazează pe o abordare iterativă de dezvoltare de software
  - produc rapid versiuni care funcționează, acestea evoluând repede pentru a satisface cerințe în schimbare.
- În 2001, este publicat și semnat de mai mulți practicieni **“Manifestul Agil”**, care exprimă succint principiile metodelor agile.

# Agile Manifesto

<http://agilemanifesto.org/iso/ro/>

*“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 și interacțiunea** înaintea proceselor și uneltelor
- **software-ul 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.”*

# Cele 12 principii ale manifestului agil

<http://agilemanifesto.org/iso/ro/principles.html>

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. Clientii și dezvoltatorii trebuie să colaboreze zilnic pe parcursul proiectului.

# Cele 12 principii ale manifestului agil

<http://agilemanifesto.org/iso/ro/principles.html>

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 directă, 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.

# Cele 12 principii ale manifestului agil

<http://agilemanifesto.org/iso/ro/principles.html>

9. Atenția continuă pentru excelență tehnică și design bun îmbunătășește agilitatea.
10. Simplitatea — arta de a maximiza cantitatea de muncă nerealizată — este esențială.
11. Cele mai bune arhitecturi, cerințe și design se obțin de către echipe care se auto-organizează.
12. La intervale regulate, echipa reflectează la cum să devină mai eficientă, apoi își adaptează și ajustează comportamentul.

# Aplicabilitatea metodelor agile

- În companiile care dezvoltă **produse software de dimensiuni mici sau mijlocii**.
- În cadrul companiilor unde se dezvoltă **software pentru uz intern (proprietary software)**, deoarece există un angajament clar din partea clientului (intern) de a se implica în procesul de dezvoltare și deoarece nu există o mulțime de reguli și reglementări externe care afectează software-ul.
- Datorită orientării lor pe echipe mici și bine integrate, există probleme în scalarea metodelor agile pentru sistemele mari (deși și în acest caz se pot realiza anumite componente ale sistemului în mod agil).

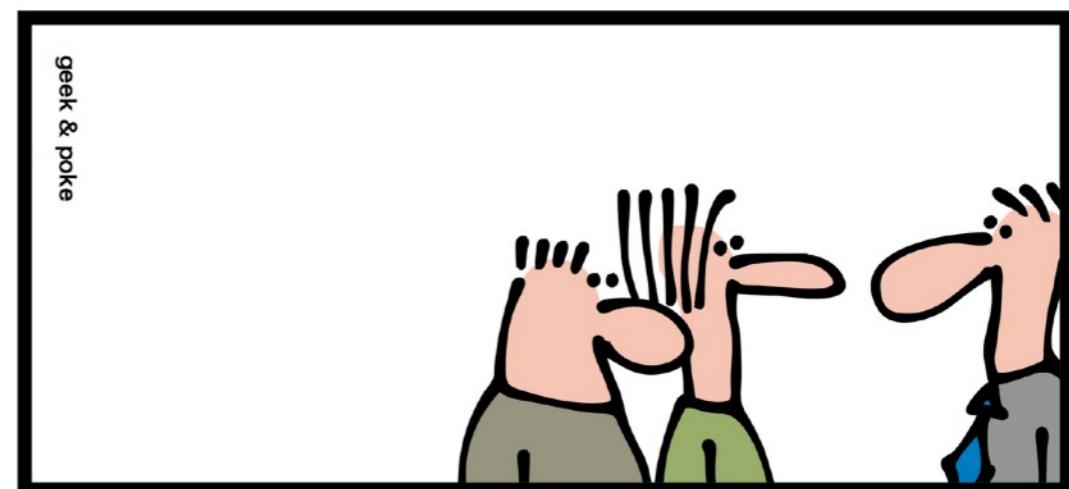
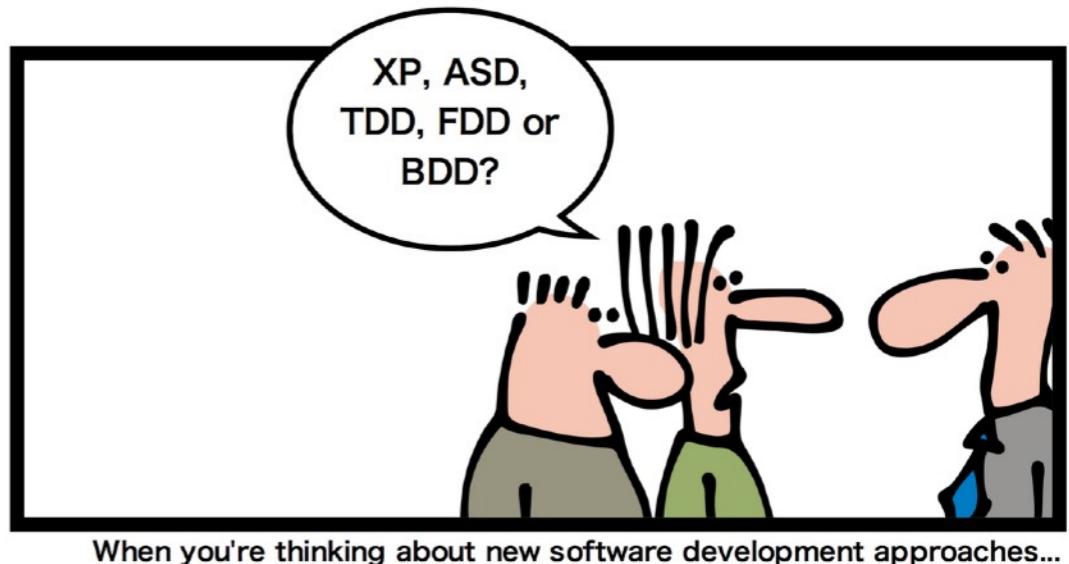
# Posibile probleme cu metodologiile agile

- dificultatea de a păstra interesul clienților implicați în acest procesul de dezvoltare pentru perioade lungi
- membrii echipei nu sunt întotdeauna potriviti pentru implicarea intensă care caracterizează metodele agile
- prioritizarea modificărilor poate fi dificilă atunci când există mai multe părți interesate
- menținerea simplității nu e ceva simplu :-)
- contractele pot fi o problemă ca și în alte metode de dezvoltare incrementală

# Metodologii “agile”

Exemple de **metodologii agile**:

- **Extreme Programming (XP)** - 1996
- Adaptive Software Development (ASD)
- Test-Driven Development (TDD)
- Feature Driven Development (FDD)
- Behavior Driven Development (BDD)
- Crystal Clear
- **Scrum - 1995**
- etc.



# Programare extremă



# Programare extrema (XP)

- poate cea mai cunoscută metodă agilă.
- **Extreme Programming (XP)** are o abordare "extremă" de dezvoltare incrementală:
  - noile versiuni pot fi construite de mai multe ori pe zi;
  - acestea sunt livrate clienților la fiecare 2 săptămâni;
  - toate testele trebuie să fie executate pentru fiecare versiune și o versiune e livrabilă doar în cazul în care testele au rulat cu succes.

Informații: <http://www.extremeprogramming.org>

# XP și principiile “agile”

- “dezvoltarea incrementală” este susținută prin intermediul livrării de software în mod frecvent cu mici incremente.
- “implicarea clientului” înseamnă angajamentul “full-time” al clientului cu echipa de dezvoltare.
- “oameni, nu procese” prin programare în doi (pereche de programatori), proprietatea colectivă și un proces care să evite orele lungi de lucru.
- “receptivitate la schimbare” prin livrări frecvente.
- “menținerea simplității” prin refactoring constant de cod.

# Valorile XP

- Simplitate (Simplicity)
- Comunicare (Communication)
- Reacție (Feedback)
- Curaj (Courage)
- Respect (Respect)

# Practici XP

- procesul de planificare (The Planning Game)
- client disponibil pe tot parcursul proiectului (On-Site Customer)
- implementare treptată (Small Releases)
- limbaj comun (Metaphor)
- integrare continuă (Continuous Integration)
- proiectare simplă (Simple Design)
- testare (Testing)
- rescriere de cod pentru îmbunătățire (Refactoring)
- programare în pereche (Pair Programming)
- drepturi colective (Collective Ownership)
- 40 ore/săptămână (40-Hour Week)
- standarde de scriere a codului (Coding Standard)

# Planificare

- “Joc” de planificare a livrărilor: clienți și dezvoltatori.
- “Joc” de planificare a iterățiilor: doar dezvoltatorii
- Clientul înțelege domeniul de aplicare, prioritățile, nevoile business ale versiunilor care trebuie livrate:
  - sortează “cartonașele” cu sarcini după priorități
- Dezvoltatorii estimează riscurile și eforturile:
  - sortează “cartonașele” după risc
  - dacă o sarcină ia mai mult de 2-4 săptămâni, e distribuită pe mai multe “cartonașe”



# Livrări frecvente

- livrări cât de des este posibil, cu condiția să existe o plus-valoare pentru client
- acest lucru asigură feedback-ul rapid
- clientul are funcționalitatea esențială cât mai curând posibil
- timp între livrări de la o săptămână până la o lună  
(proiectele non-XP au termene de jumătate de an sau mai mult)

# Metafora

- “metafora” este de fapt cuvântul-cheie XP pentru ceea ce alți ingineri numesc “arhitectura sistemului”
- se evită cuvântul “arhitectură” pentru a sublinia faptul că nu avem de-a face doar cu structura generală a sistemului
- “metafora” sugerează o coerență generală, ușor de comunicat, plus mai multă maleabilitate

# Integrare continuă

- codul este integrat și testat în cel mult câteva ore sau o zi de când este scris.
- de exemplu, atunci când dezvoltatorii au terminat o parte din implementare:
  - o integrează cu codul existent
  - rulează teste și corectează eventualele probleme
  - dacă toate testele sunt pozitive, adaugă modificările în sistemul care se ocupă cu managementul codului sursă.

# Proiectare simplă

- principiul de bază: “proiecteză cel mai simplu lucru care funcționează acum. Nu proiecta și pentru mâine, pentru că s-ar putea să nu fie nevoie”
- deci, proiectează doar pentru a satisface cerințele și nimic în plus

# Testare

- se testează tot ceea ce ar putea fi problematic.
- programatorii scriu **teste unitare** folosind un cadru de testare automatizată (de exemplu, JUnit) pentru a minimiza efortul de scriere și verificare a testelor.
- clienții, cu ajutorul dezvoltatorilor, scriu **teste funcționale**.
- de multe ori, se aplică metoda “**test-driven development**”:
  - se scriu teste înaintea codului pentru a clarifica cerințele.
  - testele sunt scrise ca programe în loc de date, astfel încât acestea să poată fi executate automat.
  - fiecare test include o condiție de corectitudine.
  - toate testele anterioare și cele noi sunt rulate automat atunci când sunt adăugate noi funcționalități, verificând astfel că noua funcționalitate nu a introdus erori.

# Îmbunătățirea codului

- Îmbunătățirea codului prin “refactoring” este foarte importantă deoarece XP recomandă începerea implementării foarte repede.
- simplificarea codului duce la o mai bună înțelegere a lui, astfel compensând lipsa documentației în anumite situații.
- exemplu (“*three strikes and you refactor*”) eliminarea duplicării: dacă o bucată de cod similară apare în trei locuri, se scrie codul respectiv într-o metodă care e folosită în cele trei locuri.

# Programarea în echipe de doi

- tot codul este scris de două persoane folosind un singur calculator
- sunt **două roluri** în această echipă:
  - ◆ unul scrie cod și
  - ◆ celălalt îl ajută gândindu-se la diverse posibilități de îmbunătățire:
    - merge abordarea curentă?
    - care sunt testele care s-ar putea să nu funcționeze?
    - există simplificări posibile?

# Avantajele programării “în doi”

- susține ideea de proprietate și responsabilitate în echipă pentru sistemul colectiv
- proces de revizuire îmbunătățit, deoarece fiecare linie de cod este privită de cel puțin două persoane (“*four eyes principle*”)
- ajută la îmbunătățirea codului
- transfer de cunoștințe și training implicit (important când membrii echipei se schimbă)
- “more fun”?

# Pair programming... în practică



PetraCross.com

# Drepturi colective asupra codului

- nu există situația în care cineva are anumite “module” pe care nimeni altcineva nu le poate atinge.
- dacă cineva vede o modalitate de a îmbunătăți ceva, va face toate modificările necesare în sistem (desigur e nevoie de un sistem bun de versionare și management al codului).

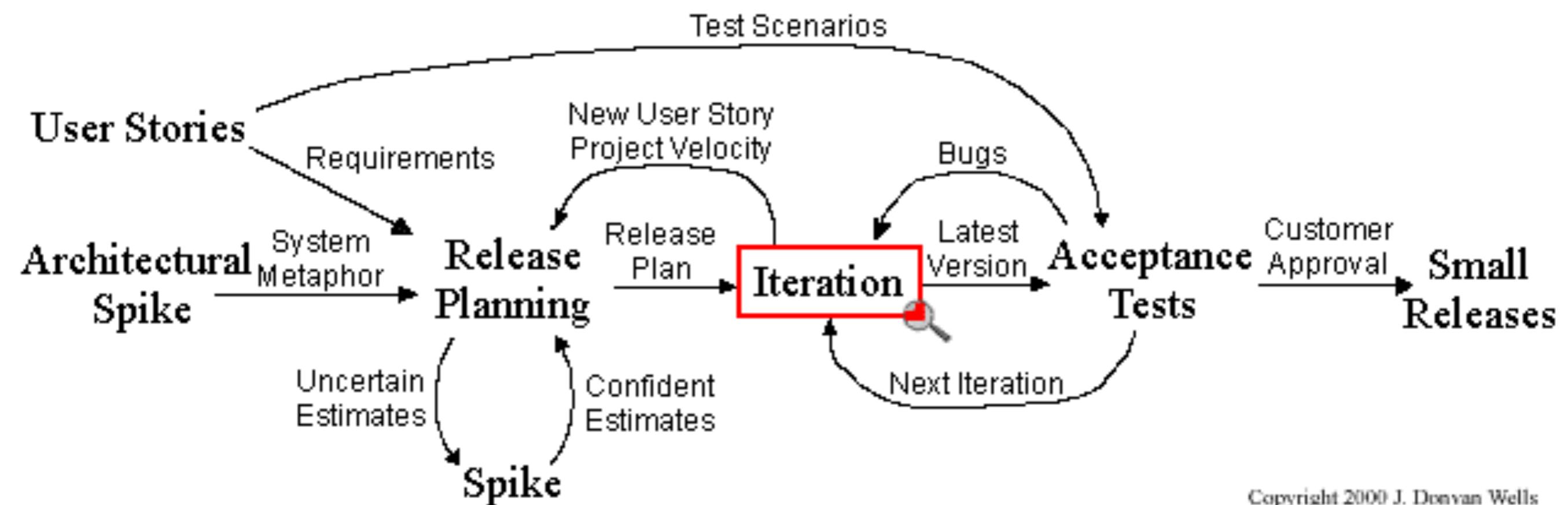
# 40 de ore pe săptămână

- săptămâna de lucru de maxim 40 de ore
- oamenii au nevoie de odihnă necesară pentru a munci eficient și a produce cod de calitate înaltă
- există săptămâni când e necesar să se lucreze mai mult, dar aceasta trebuie să fie o excepție

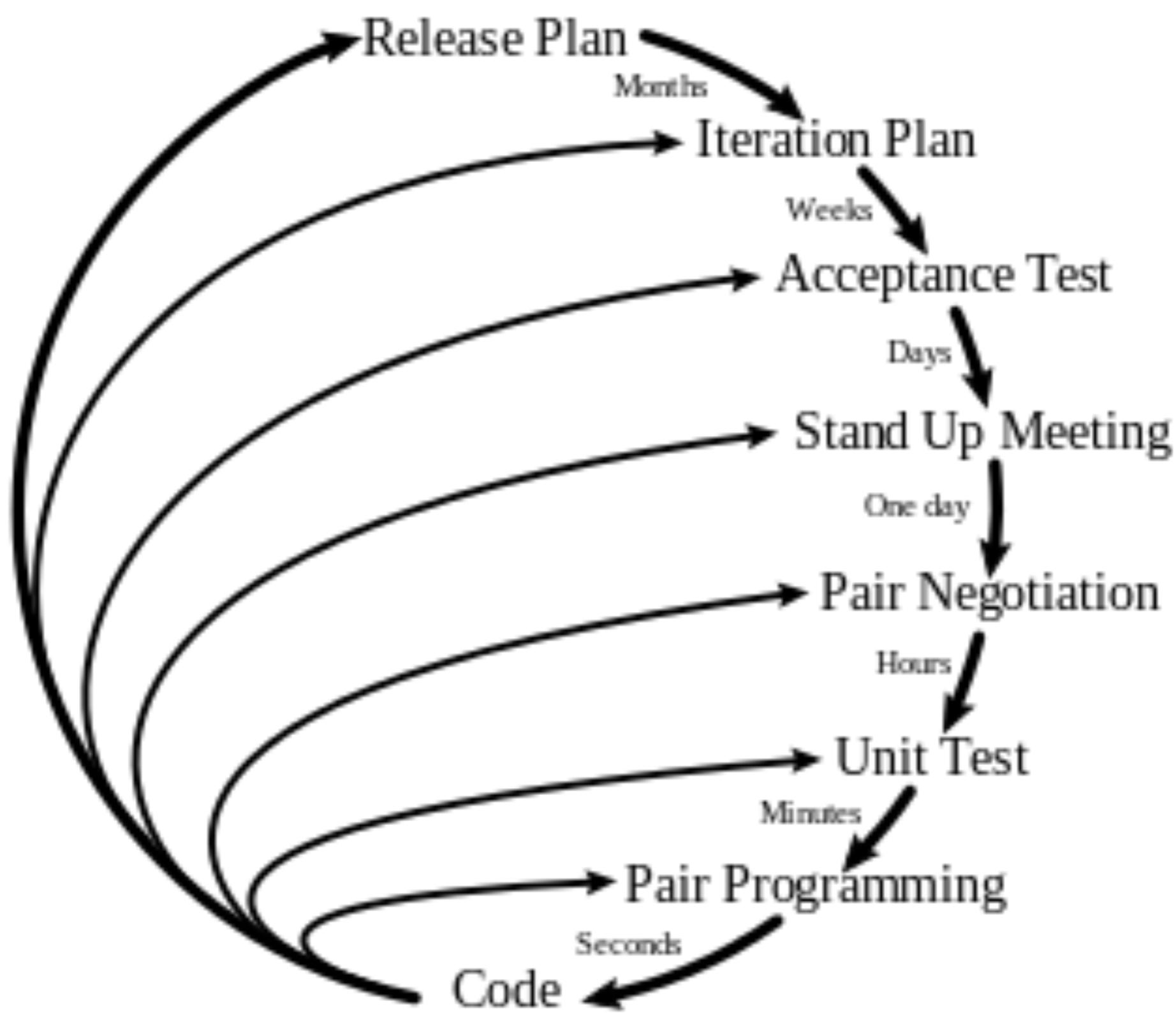
# Codul scris în mod standardizat

- Întreaga echipă aderă la un set unic de convenții cu privire la modul în care codul este scris
- scopul este de a facilita programarea “în doi” și proprietatea colectivă a codului

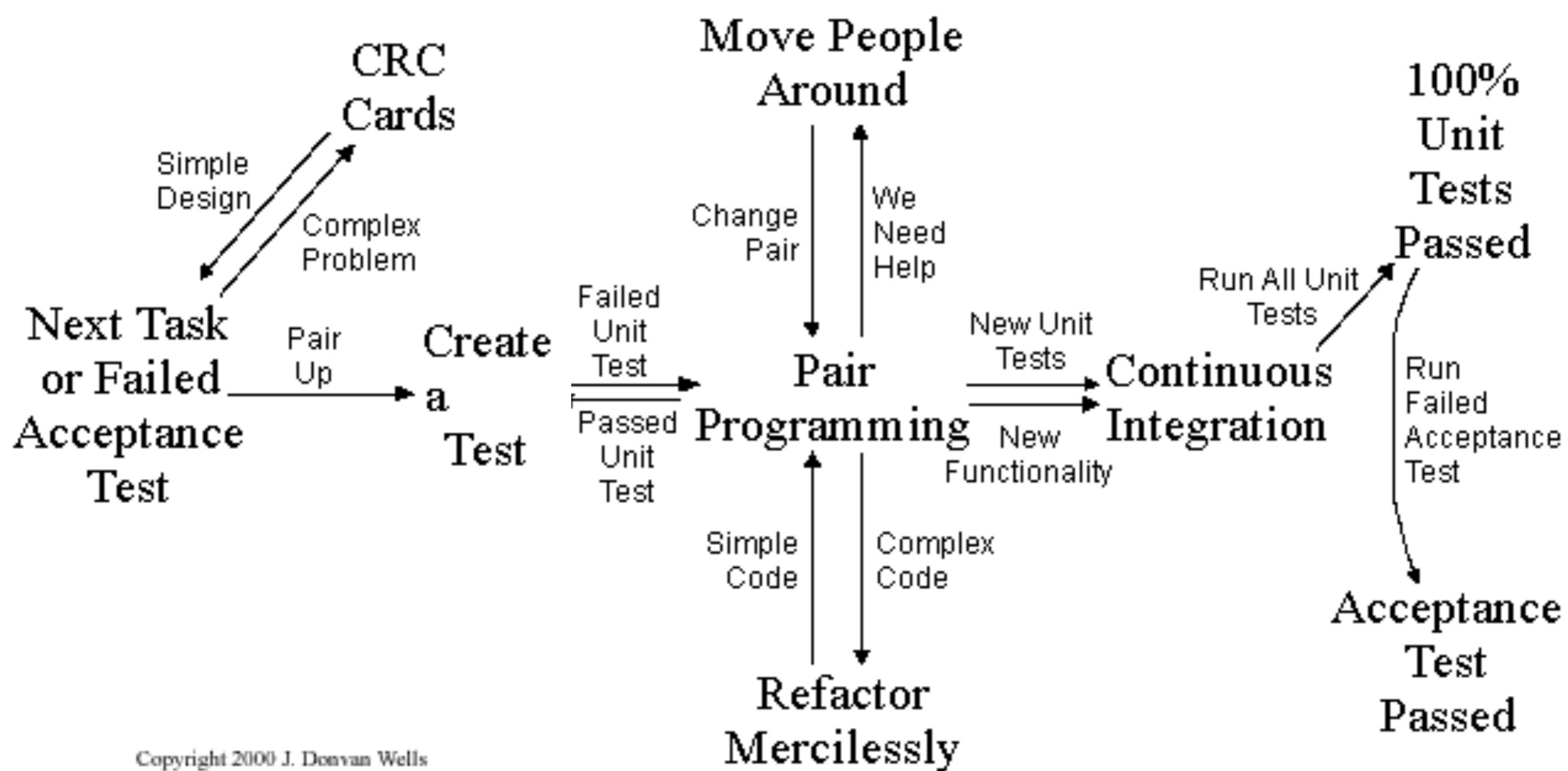
# Ciclul de dezvoltare XP



# Planificare și iterății XP



# Lucrul cu codul în XP



# Avantaje XP

- soluție bună pentru proiecte mici
- programare organizată
- reducerea numărului de greșeli
- clientul are control (de fapt, toată lumea are control, pentru că toți sunt implicați în mod direct)
- dispoziție la schimbare chiar în cursul dezvoltării

# Dezavantaje XP

- nu este scalabilă
- necesită mai multe resurse umane "pe linie de cod" (datorită d.ex. programării în doi)
- implicarea clientului în dezvoltare (costuri suplimentare și schimbări prea multe)
- lipsa documentelor "oficiale"
- necesită experiență în domeniu ("senior level" developers)
- poate deveni uneori o metodă ineficientă (rescriere masivă de cod)

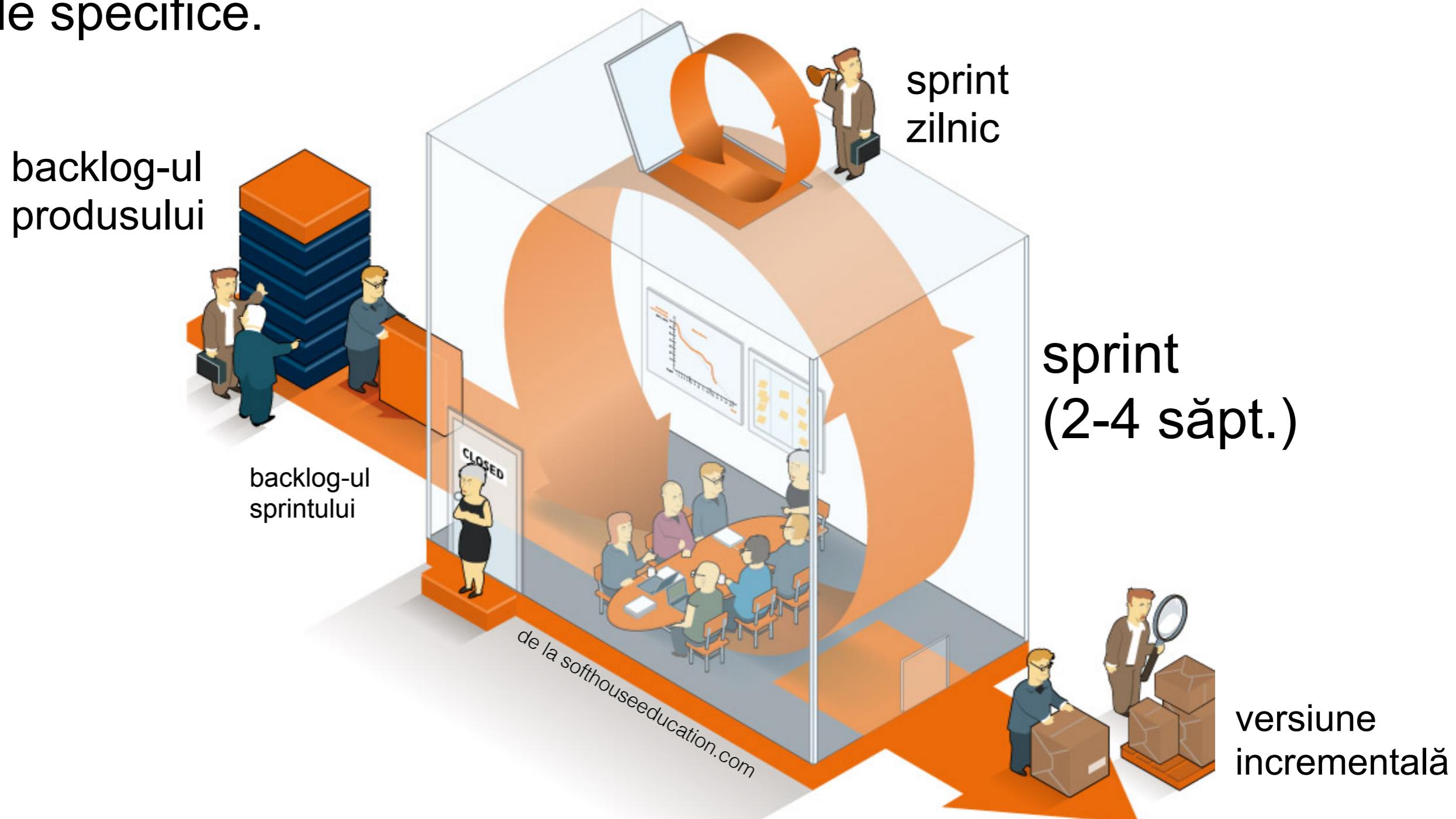


de la n-ix.com

# SCRUM

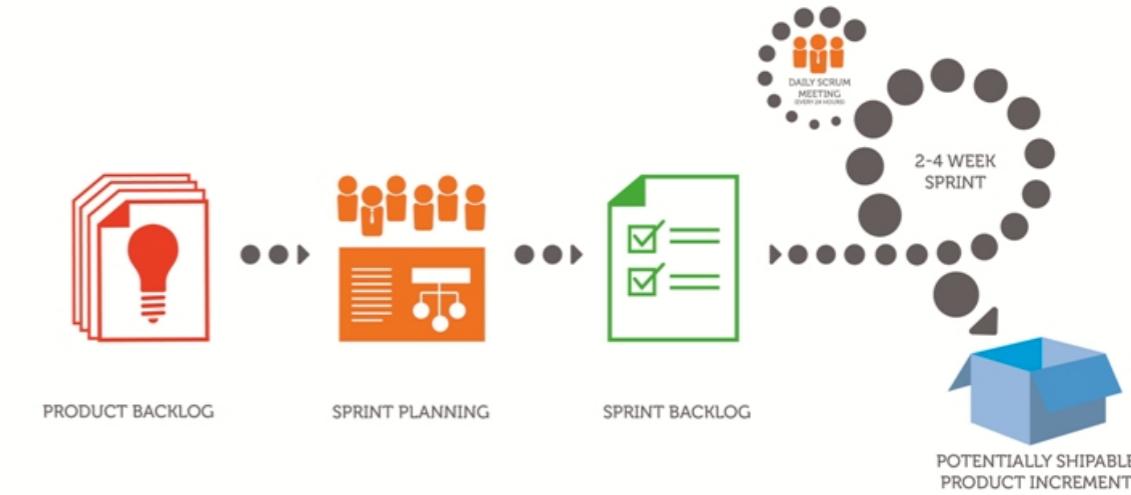
# Metoda Scrum

Scrum este o metodă “agilă”, care se axează mai mult pe **managementul dezvoltării incrementale**, decât pe practici agile specifice.



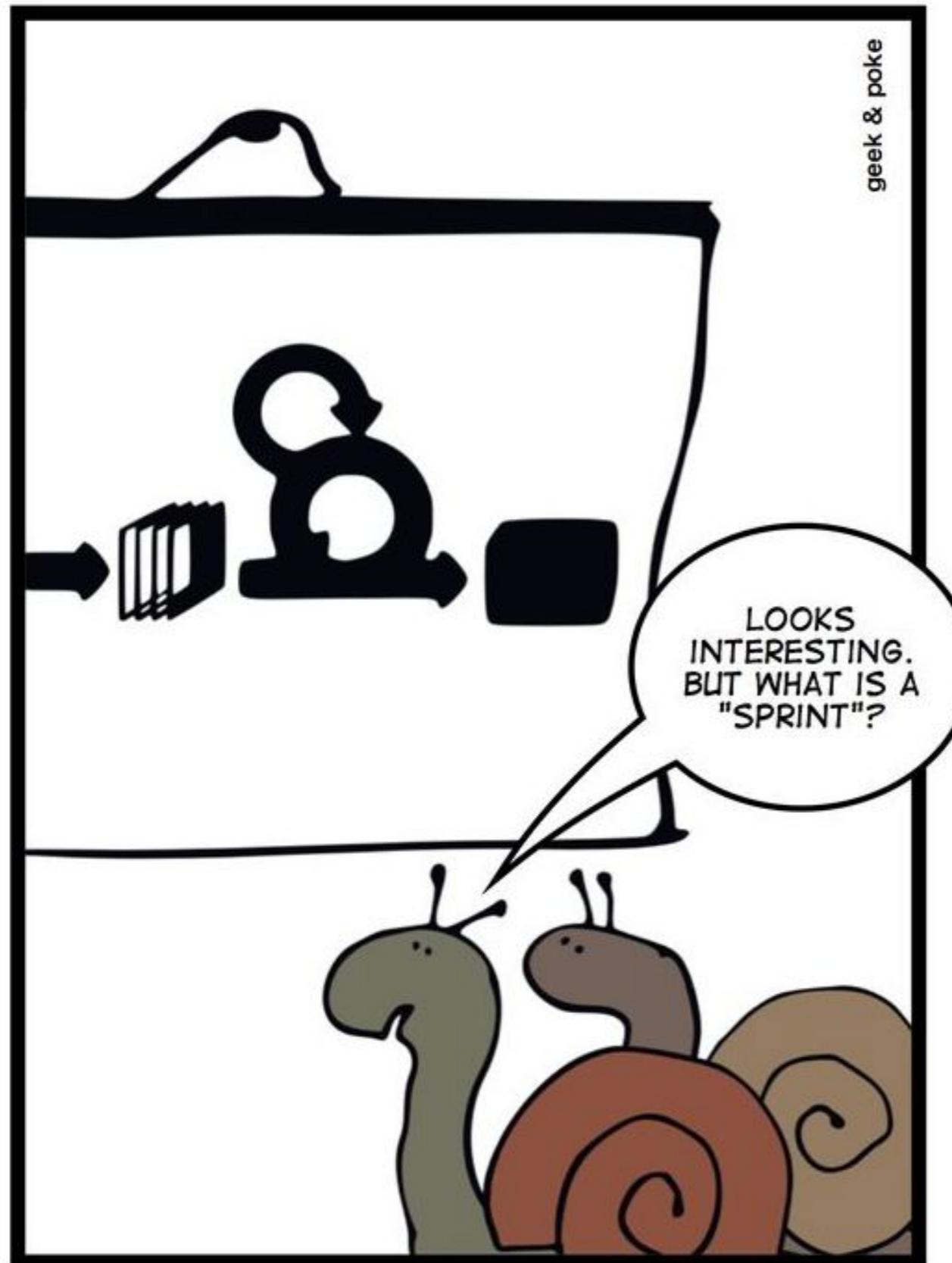
# Scrum pe scurt

- un proprietar de produs creează o listă de sarcini numită “**backlog**”.
- după aceasta, se planifică care dintre sarcini vor fi implementate în următoarea iteratie, numită “**sprint**”.
- această listă de sarcini se numește “**sprint backlog**”.
- sarcinile sunt rezolvate în decursul unui sprint care are rezervată o perioadă relativ scurtă de 2-4 săptămâni.
- echipa se întâlnește zilnic pentru a discuta progresul (“**daily scrum**”). Ceremoniile sunt conduse de un “**scrum master**”.
- la sfârșitului sprintului, rezultatul ar trebui să fie livrabil (adică folosit de client sau vândabil).
- după o analiză a sprintului, se reiterează.



ScrumAlliance®

# Scrum... în practică

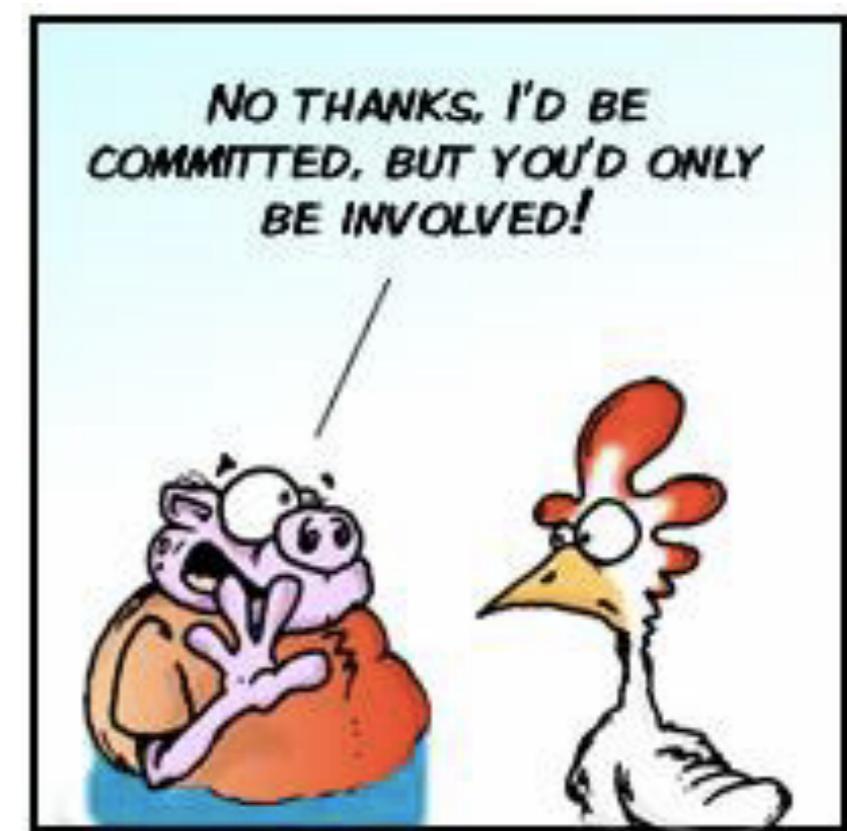
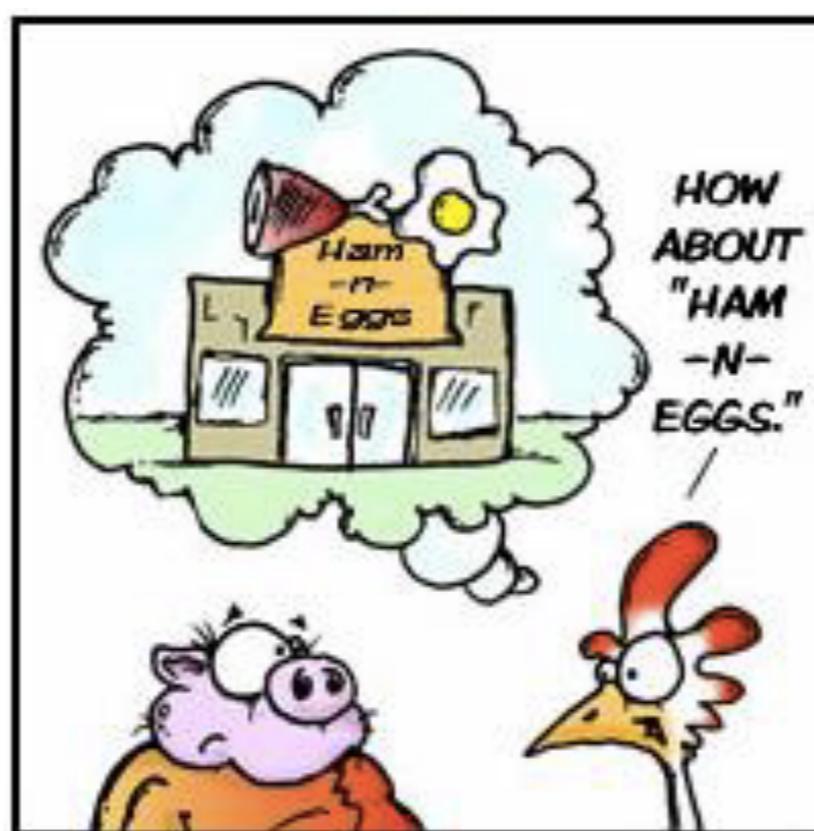


Learning Scrum

# Roluri în SCRUM

- product owner (proprietar de produs) - definește backlogul
- scrum master - are grija de întreg procesul
- membrii echipei
- utilizatori & stakeholders & manageri

pigs ← chickens



# Daily (deadly?)... Scrum



# Tool-uri pentru metode “agile” și Scrum

Există foarte multe tool-uri pentru metodele și practicile “agile”.

Câteva exemple:

<http://pivotatracker.com>

<http://agilescout.com/best-agile-scrum-tools/>

<https://www.scrumwise.com/scrum/#>

<http://www.acunote.com/how-it-works>

etc.

# Câteva exemple

Tehnicile “agile” sunt folosite în foarte multe companii, deși multe dintre ele nu folosesc explicit termenul de “agile”.

Câteva referințe:

- **Google:** <http://georgekontopidis.com/blog/2011/googles-agile-practices/>  
v. și <https://arxiv.org/pdf/1702.01715>
- **Facebook:** [http://www.facebook.com/note.php?note\\_id=409881258919](http://www.facebook.com/note.php?note_id=409881258919)
- **SAP:** <http://scn.sap.com/community/agile-software-engineering>
- **Microsoft:** <http://stories.visualstudio.com/scaling-agile-across-the-enterprise/>
- **IBM:** <http://www.ibm.com/software/rational/agile/>

# O comparație de final

<b>Metode agile</b>	<b>Metode cascădă</b>	<b>Metode formale</b>
criticalitate scăzută	criticalitate ridicată	criticalitate extremă
dezvoltatori seniori	dezvoltatori juniori	dezvoltatori seniori
cerințe în schimbare	cerințe relativ fixe	cerințe limitate
echipe mici	echipe mari	echipe mici
cultură orientată spre schimbare	cultură orientată spre ordine	cultură orientată spre calitate și precizie



UNIVERSITATEA  
DIN BUCUREŞTI

Primăvară frumoasă!

