-avem nevoie de inginerie software ca sa gestionam complexitatea sistemelor software
-avem constrangeri de buget si de costuri
-partea de conceptie + partea de proiectie/realizare
-software-ul e inginerie, nu manufactura -> lipseste costul material -> nu e vorba de utilaje, nu e vb de tabla, sticla , siliciul pe care il folosesti etc.
**-de ex a scazut volumul productiei de chip-uri -> nu se mai pot construi atat de multe masini**
**-concluzie : de asta e mai profitabil SW-ul**
-costurile din SW vin din partea de engineering, mai exact de design, de concept -> pana sa fie dat in productie
-un sistem SW nu se schimba cand se actualizeaza un algoritm, de ex un sistem SW pt calculul salariilor -> daca se schimba legea trebuie doar sa se actualizeze algoritmul care face calculul salariilor
-SW-ul poate imbatrani!!! De ce? Din cauza schimbarilor repetate
-daca dedicam timp pt mentenanta asta va influenta cat timp vom putea folosi sistemele SW respective
-un sistem SW functioneaza pe principiul “daca nu te schimbi, ai sa mori” -> sistemul trebuie sa se schimbe pt ca altfel devine nesatisfacator, problematic si inutil
-dar si daca te schimbi -> mori : orice schimbare a sistemului ii va creste complexitatea structurii si ii va scadea calitatea interna -> tot mai greu de intretinut si de lucrat cu el -> - curba/caldarea sistemului SW e in functie de masurile pe care noi le luam pt a imbunatatii structura interna
-nevoie sistemelor de a se schimba->caldarea cu rata erorilor in timp -> nu merge asymptotic la 0 din cauza ca trebuie sa schimbam sistemul din cand in cand -> in timp sistemele se erodeaza -> nu se mai pot folosi
-sistemele SW sunt informatie, nu materie!!! -> nu e vb de manufacturing
-la avion, fiecare piesa are un timp de uzura -> se inlocuieste dupa un timp -> curba erorilor creste dupa un anumit timp
-principiul inversarii constrolului -> nu tu apelezi biblioteca, framework-urile te apeleaza pe tine -> principiul Hollywood -> call back : controlul e mereu la framewrok
-care sunt costurile? Trebuie sa construim sisteme SW intr-o lume reala -> intrebarea aceasta care sunt consturile ne da framework-ul de constrangeri care trebuie indeplinit ca sa folosim sistemul nostrum SW in lumea reala
-costurile de mentenanta/intretinere a sistemului SW sunt mult mai mari decat cele de dezvoltare al lui
-cel mai mare cost nu e codarea, ci testarea de sistem!!!! -> are cele mai mari constrangeri secventiale
-testarea are mai multe componente -> ele pot fi testate si individual -> daca una din piese intarzie, intarzie si testarea -> costuri mari
-planing -> captarea cerintelor si modelarea : toata partea de pregatire a task-urilor de codare
-pe ultima parte vine testarea
-nu dureaza sa scriem cod, ci citind prin cod!!!
-trebuie sa petrecem cat mai mult timp pt testare ca sa pregatim sistemul SW pt client -> daca nu petrecem mult timp pe testare -> avem banana soft -> clientul incepe sa descopere multe bug-uri, ceea ce nu e ok
-de ce daca adaugam mai multi oameni ca sat ermine un proiect mai repede este gresit ? Pt ca exista diferite tipuri de task-uri : cu interdependente (
-Legea lui Brooks : se formeaza din task-uri cu interdependente complexe : tu trebuie sa termini un modul pt ca eu sa-l incep pe urmatorul si sa le putem lega, tu trebuie sa testezi un modul ca sa putem testa apoi integrat cu un alt modul etc. , trebuie sa se faca interfetele/API-urile ca sa le pot testa

-daca tot adaug oameni, pot scadea timpul de dezvoltare a SW-ului -> e foarte periculos!!!!
-ideea de comunicare intr-o echipa nu consta doar in vorbitul intre ei -> cu cat ai o echipa mai mare de oameni, cu atat e mai greu de gestionat -> daca stau unii dupa altii e foarte complicat
**-Legea lui Brooks : daca adaugi pe parcursul unui proiect SW in dezvoltare mai multi oameni si proiectul e deja in intarziere, il faci sa intarzie si mai mult**
-se creeaza mai multe interconexiuni -> implica un effort mult mai mare, ceicare intra mai tarziu trebuie sa depuna un effort mai mare ca sa inteleaga conceptul
-parametrii dupa care se diseaca legea lui Brooks : timpul limitat, oamenii -> timpul depinde de constrangerile secventiale -> sunt anumite lucruri pe care nu le poti face in paralel, ci trebuie sa se execute secvential : de ex, avem a, b , c etc. task-uri -> task-ul b nu poate fi indeplinit pan ace task-ul a nu e indeplinit samd.
-nr de oameni depinde de nr de task-uri independe pe care le poate gasi seful
-legea lui Brooks se leaga de miturile despre SW (mythical man fact)
-un program se transforma intr-un sistem SW cand e facut de mai multi oameni pt mai multi oameni/client
-trebuie sa avem documente pt SW -> documentam interfete, cod etc. -> ca sa putem sa scalam dezvoltarea SW-ului -> ca sa se faca fata codului -> inceput de a,b si c si sa fie preluat de x,y,z mai usor
- cum asiguri continuitatea sistemului SW? fix descris ca mai sus

---

---

PROCESUL PRIN CARE AJUNGEM SA CONSTRUIM/DEZVOLTAM UN SISTEM SW
-SCRUM : cel mai celebru process de dezvoltare
-implementarea si operationalizarea -> aducerea unui sistem SW in productie -> sa fie deja in mana celor care il folosesc
-fundamental, avem 2 procese de dezvoltare -> waterfall si procese iterative
-avem nevoie de procese de dezvoltare ca sa invingem complexitatea -> inginerie SW = complexitate
-cum invingi/gestionezi ca un task monumental sa poata fi realizat? De ex, cum ajung inginer? E ceva monumental initial -> ce facem? Descompunem problema mare in bucati mai mici -> luam recursiv problemele mici si vedem daca nu sunt gestionabile -> le descompunem in probleme mai mici pana ajung sa fie gestionabile
-ce inseamna ca ajung inginer? Intru la facultate si trebuie sa termin 4 ani? Cum ii termin? Dand examene . Cum ma pregatesc de examene? Invatand si luand o nota peste 5 , la fel ca la lab
-a face facultatea -> trebuie sa indeplinim o serie de task-uri mai mari si mai mici -> task-uri gestionabile
-DESCOMPUNERE : descompun activitatea de construire a programului/proiectului in bucati gestionabile
-inteleg cerinta -> proiectez ->implementez/programez -> il validez -> intretin sistemul (prima modalitate)
-waterfall -> modalitate de a invinge complexitatea -> activitatile de mai sus
-am diferite features-uri/functionalitati -> incep sa implementez unul, dar care? -> luam un feature si executam activitatile de mai sus pe el -> apoi la sfarsitul ciclului asta, incep cu feature-ul 2 -> descompunere iterativa -> implementare feature cu feature
-trebuie omogenizate feature-urile intre ele ca sa am un singur sistem -> overhead
-incercam sa optimizam iterative, nu waterfall!

**Waterfall**

-descompunere bazata pe activitati : avem 2 dimensiuni : feature + activitati
-de unde cascada? Pornim de la definirea cerintelor -> proiectarea de sistem -> implementare si testare unitara -> testare de sistem si de integrare -> intrarea in productie, operationalizarea, mentenanta si evolutia
-avem o bucla de feedback / de reglaj -> niciun sistem nu poate functiona fara ea, repr reactia
-problema fundamentala la waterfall e ca reglajul/feedback-ul/schimbarea nu pot fi accommodate decat cand ajungem cu sistemul in productie
-toate activitatile dureaza mult -> de la prima activitate (definirea cerintelor) pana la ultima (ajungerea in productie) NU avem bucla de reglaj (feedback-loop)
- sistemul fiind intr-o lume dinamica, trebuie sa se schimbe frecvent -> singurul moment in care poti interveni asupra cerintelor/proiectarii etc. e mom in care sistemul ajunge operational
-timpul e extrem de important -> avem nevoia de a putea sa reglam , sa ne adaptam cel mai rapid
-problema cea mai mare din Waterfall e ca NU pot sa acomodez schimbarea, nu pot sa accept schimbarea in sistem sufficient de frecvent
**-in momentul in care incepe procesul trebuie sa se incheie -> n-ai ce face**
-activitatile sunt separate in mod rigit , secvential, fara sa ma pot intoarce inapoi -> nepractic
Dezavantaje :

- ➔ inflexibilitatea (ne oblige sa intelegem cerintele perfect de la inceput -> nu se poate) , schimbarile intervin(nu avem cum sa prevedem)
- ➔ ignora tot de la inceput pana la sfarsit
- ➔ **waterfall uraste cuvantul schimbare**

Avantaje :

- ➔ lucrurile sunt foarte predictibile/clare -> ex:3 luni avem pt cerinte -> avem documentul de cerinte, parafat, nu se mai schimba nimic -> avem o tinta bine stabilita, incepem proiectarea -> e foarte clean, riguros si usor de urmarit -> stii mereu unde esti si ce urmeaza

-procesul de dezvoltare in V (validare) : nu e complet waterfall

**Modelul Iterativ**

-procesul iterativ e ca si cum ai face un waterfall pe un nano-sistem
-care e diferenta ? timpul
-secventierea activitatilor e necesara -> ne place aceasta cronologie de la waterfall, dar ceea ce ne deranjeaza **e pretul pe care trebuie sa-l platim ,si anume rigiditatea** -> o perioada lunga de timp nu mai avem voie sa schimbam nimic, ex: un proiect pe 2 ani (nu se schimba nimic, decat dupa finalizarea proiectului, adica dupa acesti 2 ani)
-sparg sistemul in functionalitati si tratez fiecare functionalitate ca pe un nano-proiect pe care il dezvolt
-buclele de reglaj se intampla mai repede : de mai multe ori intr-un proiect -> de aceea e mai repede
-procesele iterative : fiecare iteratie construieste o parte din sistemul real
-NU vorbim despre prototipuri cand vine vorba de procesele iterative !!! -> sistemul are un subsistem de features-uri
-ideea de iterativ si incremental merg impreuna -> la sf fiecarei iteratii, sistemul meu creste
-ex: am 100 de features , iau primul feature -> ce inseamna primul feature? -> unul din cele mai importante avantaje e de a prioritize cerintele -> primul feature inseamna cel mai important feature -> el ajunge sa fie implementat primul

Avantaje:

- ➔ clientul primeste cea mai importanta functionalitate la inceput
- ➔ aceasta fiind primul feature, ajunge sis a fie testate cel mai mult -> ramane in fiecare noua iteratie
- ➔ clientul primeste cel mai import lucru prima oara -> acesta ajunge sa fie cel mai bine testat pt ca e implementat la adaugarea treptata a celorlalte feature-uri
- ➔ cerintele sunt prioritizate -> cand? Dupa fiecare iteratie
- ➔ aceasta abilitate de a schimba/ de a te adapta -> forta principala a proceselor iterativ
- ➔ pe durata unei iteratii -> cerintele nu se schimba -> ex:daca facem o iteratie de 6 luni, e rau si atunci intrebarea e : cat e durata unei iteratii ? cat de frecvente sunt livrarile unor incrementi? Intrebarile esentiale in dezvoltarea unui process

-ideea centrala la procesele iterative : TIME BOXING
-pt fiecare iteratie avem o durata fixa
-ce se intampla daca nu terminam ? am planificat ca facem F1, dar el e complex si atunci nu prelungim durata iteratiei -> una spune trebuie sa termin tot, cealalta forta e timpul, idea de feedback si spune sa nu amani momentul in care apare reactia la ceea ce ai implementat deja
-la inceput planifici cat dureaza iteratia -> bun, dureaza mai mult -> nu vreau sa ratez bucla de feedback -> inchiderea iteratiei imi spune ca pot sa reactionez
-faptul ca se intarzie are legatura cu feature-ul in sine
-orice proiect are un risc -> riscul eseculi -> cu cat se lungeste bucla de feedback -> riscul de esec creste
-pe langa avantajele legate de prioritizare, avem si reglarea/optimizarea procesului in sine
-avantaj : livrare accelerata -> sistemul se livreaza repede la client intr-o forma incipienta -> poate fi modelat si transformat repede
-ex pt waterfall : aviatie , constructie etc.
**-iteratiile trebuie sa fie cat mai scurte**
Dezavantaje :

- ➔ e foarte greu sa gestionezi proiectele de genul pt ca nu stii tot : avem un proiect cu feature-uri F1, F2, F3, F4, F5 -> poate aparea F7, care nu era pe lista -> nu stiu de la inceput tot
- ➔ **imaginea de ansamblu va fi disponibila doar la final**
- ➔ cum faci un contract? Ce sistem construim? Se mai schimba in timp -> cum faci contractul? Ce pret pui in contract? Cum stabilim pretul? Nu se mai fac contracte cu pret fix in general -> nu e in avantaul clientului sa specifice ce e in fiecare feature pt ca mereu se poate schimba -> daca punem ceva fix -> se transforma in waterfall -> abandonam pretul fix in contracte si abordam time and materials , cu accentul pe time-> clientul plateste volumul de efort -> nu il putem tepui pe client datorita procesului iterativ

-depinde cat e iteratia : apare forta de a reduce iteratia
**-presiunea de a face waterfall vine din conceptul de fix price !!!!**
-partea de validare se poate rezolva acceptand idea de dynamic, la fel si la mentenanta
-critica de la procesele iterative -> cand integrez F1 cu F2 s-ar putea sa rescriu anumite bucati din F1 sau sa arunc niste parti etc. -> refacerea pare ca o pierdere : de ex, motorul nu se potriveste cu caroseria, trebuie sa refac motorul , sa ajustez ceva, sa remodelez usa etc. -> se pierde material sau altceva, deci pierdere materiala-> exista si pierdere la nivelul informatiei -> pierdere de timp -> in sine, nu e o pierdere, ci exact perfectionarea conceptului
-avantajul e ca ne putem adapta daca pe parcursul dezvoltarii proiectului daca apar noi cerinte

-acesta e totodata si dezavantaj -> in vederea conceperii unui contract pt client -> clientul nu stie cum va arata proiectul la final -> ceea ce e un lucru bun totusi -> trebuie sa lasi flexibila evolutia proiectului -> nu stii care va fi exact forma lui finala

---

---

-deosebire intre planificare predictive si planificare adaptiva : in predictiva (fac un plan si-l urmez, dar e foarte greu sa prezici -> fix price, fix scope), in adaptive(nu mergem rigid dupa un plan, faci mai mult planning decat in predictiva -> fix price , variable scope -> ceea ce platesti e time and materials -> sa fim cat mai rapid adaptabili -> livrare de SW cat mai repede-> mai multe iteratii si mai scurte -> cu cat sunt mai scurte, cat atat bucla de reglaj se manifesta mai rapid)
-preferam sa punem accentul pe indivizi, pe capacitatea lor si pe interactiunile dintre ei pt a dezvolta un sistem SW
-apreciem un SW functional si livrarea constanta de functionalitate fata de documentatii stufoase
-contractele -> cum se negociaza? E mai importanta colaborarea vie cu clientul -> opus cu waterfall (clientul vine, cere si apoi vine la final sa ia proiectul)
-nu e important ca design-ul sa fie perfect, ci e important sa avem abilitatea de a ne schimba, de a fi agili
-nu putem executa teste pt ca nu avem cod , dar putem scrie teste fara sa avem cod -> in cazul asta, testele devin un mod formal prin care definim cerintele
-clientul trebuie sa se implice in partea de dezvoltarea proiectului -> se exprima bine in SCRUM
Dezavantaje metode agile :

- ➔ sunt foarte mult centrate pe individ si pe echipe coezive -> e foarte greu sa scalezi

**SCRUM**
-e un proces agil -> toate lucrurile de mai sus sunt mainfestate explicit si apoi duse la extrem
-ce e un sistem agil? E un sistem care favorizeaza schimbarea, se refera la planificarea a ce se va construe in urmatoarele saptamani, apoi se adapteaza de la acel punct
-cascada : planificam totul in avans apoi urmam planul pe o durata fixa stabilita la inceput (ex:1 an, 2 ani etc.)
-exista 3 roluri principale :

-> **PRODUCT OWNER** : cumva aici vine vorba de client

-> persoana careia ii pasa cel mai mult de produs, e mereu atent ca lucrurile care se construiesc si feature-urile care se adauga sa fie mereu cele mai valoroase -> in asa fel incat sa fie la orice moment maximizam valorile pe care sistemul SW le are

-> la fiecare moment, scopul product owner-ului e sa se asigure ca lucrurile care se construiesc maximizeaza valoarea produsului -> adica : ca se aleg mereu feature-urile cele mai importante, ca se revizuiesc prioritatile astfel incat la fiecare noua iteratie sa se obtina cea mai mare valoare posibila

**-> pe scurt, el se ocupa ca prioritatile produsului sa fie respectate de echipa care le va implementa**

-> **DEVELOPMENT TEAM** (echipa de dezvoltare) :

-> self organizing group : asta e idea centrala din procesele agile : echipele au autonomie , adica ele decid cum sa livreze acele bucati de functionalitate, respectand procesul, respectand

prioritatile clientului (prioritatile product owner-ului) -> mai apoi echipa se organizeaza ea intern cum sa faca livrarea asta cel mai bine

**-> pe scurt, echipa actioneaza autonom, adica liber, tinand cont doar de prioritatile date de product owner**

-> **SCRUM MASTER** : e cumva intre primele 2 roluri

-> e o forma voalata de a spune project manager

-> reprezinta aparatorul procesului : se asigura ca atat clientul cat si echipa de dezvoltare urmeaza procesul -> ca nu vor avea gandirea asta de waterfall

**-> pe scurt, are grija ca procesul sa fie urmat intocmai , urmareste evolutia task-urilor si a efortului depus de echipa pe un burndown chart**

- procesul de SCRUM : product owner-ul primeste toate cerintele/constrangerile care pot exista si pe baza lor se face o lista cu prioritati : cu ceea ce ar trebui sa se faca
- cu alte cuvinte, product owner-ul trebuie sa spuna care sunt feature-urile care trebuiesc implementate la fiecare moment (la sfarsitul fiecarei iteratii , adica foarte des) si apoi sa le prioritizeze -> sa actualizeze de fiecare data lista cu prioritati
- apoi echipa (care e autonoma) sa-si calculeze o viteza/capacitate de lucru si isi preia, in functie de asta (de viteza de lucru), niste item-uri din lista prioritizata (se mai numeste si product backlog) -> product backlog = stiva in cvare sunt puse lucrurile care trebuiesc facute, care sunt de facut -> aceasta e mereu revizuita
- echipa isi allege cele mai prioritare item-uri, task-uri din acest product backlog si formeaza un sprint backlog
- cum face asta? Printr-un meeting de planificare la inceputul unui sprint-> in prima zi a unui sprint se face planning-ul
- planning-ul inseamna : ne uitam la ce avem in product backlog impreuna cu product owner-ul si stabilim un subset de feature-uri de aici in ordinea prioritatilor avand in vedere capacitatea echipei la momentul ala sa faca -> se construieste astfel un sprint backlog
- sprint-ul e numele pe care SCRUM l-a dat unei iteratii -> **SPRINT = O ITERATIE**
- de ce ii spun sprint? Ca sa iti sugereze idea de rapid (sprintul dureaza intre 1-4 saptamani)
- despart feature-urile in niste task-uri -> se impart apoi task-urile intre membrii echipei
- din dorinta de a reduce riscurile -> avem o iteratie in iteratie -> adica echipa face o revizitare asupra lucrurilor pe care le-au facut cu o zi in urma (prin daily stand-up meeting) -> ca sa evitam blocajele -> avem o bucla de reglaj in interiorul unei bucle de reglaj
- dupa ce se termina cele 1-4 saptamani si daily-uri -> se termina un increment(sprint)
- cand se termina aceasta bucata de functionalitate se fac 2 lucruri -> se face un demo al incrementului/sprintului respectiv (product owner-ul impreuna cu echipa fac sprint review) -> se da feedback -> se mai face si o retrospectiva a sprintului (ce a mers bine/gresit , unde s-a incetinit procesul etc. -> pot fi destul de tensionate)
- de ex: daca am zis ca putem sa facem n feature-uri in x timp, de ce am facut doar 80% din ele? Unde ne-am inselat? Pt ca ori s-a intamplat ceva neprevazut, ori am asteptat dupa altii, ori a fost un rist tehnologic pe care nu l-am anticipat -> se fac aceste retrospective ca sa imbunatatim procesul de dezvoltare al feature-ului urmator

**SCRUM BURNDOWN CHART**

- ce este? E un graphic in care noi am spus ca facem x task-uri -> e BURNDOWN pt ca vrem sa ardem efortul ala, sa consumam task-urile pe care trebuie sa le facem din sprint backlog
- initial avem de ex 23 de task-uri de facut in 21 de zile-> dupa o zi cate avem?
- se poate si sa urcam dupa ce am coborat dupa un task -> asta vine dupa niste teste -> daca echipa de testare a spus ca task-ul nu e inca finalizat, crestem inapoi in graphic de unde am coborat initial
- tot ceea ce conteaza este sa avem un feedback zilnic -> trebuie sa vezi constant ce se intampla, nu doar la sf unei iteratii

CONCLUZII :

- sprint backlog-ul e ca un inventar al task-urilor care urmeaza sa fie implementate in respectivul sprint, iar task-urile se iau din product backlog, care e o stiva
- daily standup -> sunt folosite pt a avea o bucla de reglaj in interiorul buclei de reglaj care e deja sprintul, pt a vedea daca sunt blocaje ca sa poata fi rezolvate etc.
- un sistem SW nu e predictibil, dar poate fi planificabil
- clientul poate modifica oricand product backlog-ul, poate sa reprioritizeze, poate sa stearga sau sa adauge task-uri/feature-uri -> acest product backlog-ul nu e static -> e dynamic
- cea mai problematica activitate -> **estimarea efortului**
- cat incape intr-un sprint? Trebuie sa avem o unitate de masura a efortului -> pt ca un task poate fi mai mare sau mai mic, un feature poate fi mai complex sau nu samd. -> nu se poate spune un nr de task-uri -> cand stim unde sa ne oprim? Cat putem pune? -> unitatea de masura va fi ora => cate ore va petrece un membru al echipei pt task-ul respective -> trebuie sa ne gandim la mult mai multe aspecte
- avem 8 oameni in 10 zile -> 80 de parti in base * 8 ore -> 640 ore capacitate de lucru pt echipa
- daca nu suntem toti full time? -> trebuie sa fim mai realisti, poate avem 500 de ore capacitatea de lucru
- din cele 10 zile -> de ex 1 zi e pt planning meeting -> trebuie sa tinem cont de mai multe aspecte
- daca vrem sa luam un task din product backlog, cand cate person hours ar costa? Nu avem de unde sa stim -> avantajul e ca feature-urile pe care le punem in product backlog sunt definite cu granularitate a. i. sa se poata estima timpul de lucru -> exista riscul sa gresim volumul de effort necesar pt a implementa un task
- daca nu se poate estima -> task-ul e prea mare si trebuie spart
- cum noi facem estimarile pe un sprint (iteratie de 1-4 sapt maxim) -> maxim ce putem gresi e un sprint
- story points = descrierea unui anumit feature -> se ia un feature/story, il spargem in task-uri (sunt mai usor de implementat, de ex intr-o zi) -> story-urile se spartg in task-uri -> planificam task-urile si le estimam, le implementam, le evaluam si luam urmatorul story samd.
- Unde se planifica task-urile? Intr-un task management system, de ex JIRA
- Pt fiecare task avem prioritatea din perspectiva user-ului (clientului) si din perspectiva tehnica(echipa de dezvoltare)
- De ce trebuie sa avem si prioritate din perspectiva tehnica? Pt ca si ea trebuie sa aiba o prioritizare a task-urilor ca sa construiasca sistemul cat mai efficient si logic dpdv tehnic
- Pe baza prioritatilor se mai estimeaza un risc -> ca ei sa esueze -> se ia ca unitate de masura virtuala points-urile -> T-shirt sizes : XS, S, M, L, XL (se folosesc pt estimare)

- De ex, nu dau estimarea in ore, zic ca e un T-shirt size L , ceea ce inseamna 7 story points
- La sf unui sprint, nu zicem chiar nr de task-uri facute, spunem cate story-points a facut echipa, de ex 21 de story points -> velocity (viteza echipei -> volumul de effort pe care echipa il poate face intr-un sprint)
- De ex, cand se negociaza un contract, spunem ca echipa poate face 20 de story points -> estimez pt primele 6 sprint-uri/iteratii ca voi face 20 de story points cand negociez -> apoi reglam/stabilizam (poate la o echipa story point-ul e mai mare, la alta mai mic etc.) -> stabilizam velocity-ul la 20 story points
- Deci, pe un burndowm chart stabilim atat nr de task-uri cat si efortul si il urmarim zilnic-> de ex, am terminat un task de 5 story points , mai avem 17 si tot asa -> idea e sa te indrepti spre tinta
- Daca nu ajungi -> afli repede ca nu ajungi si poti sa revizuiesti velocity-ul, poti sa redefinesti L, S, XS etc. -> ca sa avem acuratete (sa fie cat mai apropiat de realitate)
- Procese iterative -> **TIME BOXING** -> **daca ceva e fix, atunci asta e durata iteratiei!! ->** nu inseamna ca esuam, ci ca intelegem cat putem face -> la sf iteratiei constatam ca nu am facut niste story point-uri -> nimeni nu spune ca la sf unui sprint D si E vor fi ceea ce intra in sprint-ul urmator : de ex reusim sa facem A,B si C si era planificat sa facem si D si E -> **probabil se vor face in sprintul urmator, dar probabil nu e si CERT!!!**
- Clientul are posibilitate sa reevalueze prioritatile task-urilor la sf fiecarui sprint -> atunci in meetingul de planning pt urmatorul sprint se ia de la 0 tot, adica D si E **nu vor fi automat incluse in noul sprint si nici in product backlog**, ele se vor intoarce in product backlog si bineinteles, va fi revizuita prioritatea lor de catre client
- **Atentie, nu se prelungeste niciodata durata de viata a unei iteratii/sprint pt ca durata fixa si scurta a acestuia creeaza mai multe oportunitati de reglaj, de reevaluare, de reestimare si de reprioritizare**

---

---

Prototyping

- POC I se mai spune si I se asociaza cuv. Explorare/descoperire : nu facem inca produsul, vedem ce se poate face orientativ, sa vedem algortimii ce se pot folosi etc.
- **Prototipizarea nu e o dezvoltare agila!!** De ce? La sf fiecarei iteratii se face o bucatica mica si apoi se face o alta bucatica mica si se integreaza cu restul -> iar in felul asta putem reduce riscurile dupa fiecare iteratie , doar ca **elem essential e ca tot ce se construieste intr-o iteratie este produsul REAL** -> sistemul dupa prima iteratie e sistemul real, nu e ceva ce exploram, e cee ace construim real
- In prototyping nu facem ceva real -> De ce? -> Datorita sau din cauza nevoii de explorare, de a descoperi -> nu vrei sa investesti in ceva in care nu esti sigur si atunci vrei sa cercetezi “terenul”
- Acest teren poate fi vazut atat din punct de vedere functional (domeniul , feature wise, user experience) , dar si din punct de vedere tehnologic(de ex,n –am lucrat niciodata in machine learning si acum vrem sa facem asta, stim sa facem? Etc.)
- Ce se intampla cu prototipul dupa explorare? Tentatia ft mare dupa ce vezi directia in care se va merge e sa iei prototipul si sa spui ok, asta e prima versiune a produsului -> GRESEALA MARE pt ca atunci cand prototipizezi nu esti atent la performanta, la alegerea limbajului de programare

- **Idea de a porni de la un experiment si a transforma baza de cod in ceva permanent este gresit (sa transformi ceva gandit termporar in ceva permanent e gresit)**
- **Idea ca iei schitele si aproximarile pe care le-ai gandit la inceput si le lasi sa ajunga in sistemul final -> mare eroare -> trebuie sa invatam sa ne educam clientii**

---

---

**Procese menite sa estompeze rsicurile**

- Si procesele agile sunt risk aware -> ele estompeaza riscurile prin faptul ca se poate intampla orice intr-un sprint, dar apoi oricat de catastrofal ar fi fost sprintul-> e doar un sprint
- Unul din procesele risk aware e RUP -> process de dezvoltare mai birocratic -> are avanajul ca e mai usor de contractat, de planificat apparent
- Cel mai mare risc pe care si-l asuma waterfall-ul e ca nu am ajuns unde trebuie, chiar daca procesul a fost minunat, rezultatul e groaznic, dezamagitor pt client
- **RUP nu e waterfall** -> incearca sa aduca aici sentimentul de progress de la waterfall
- De ce spun sentiment de progress? Pt ca de ex, la procesele iterative nu putem pune progress bar : de ex, uite asta e sistemul si am ajuns la 20%, dar nu putem face un progress bar pt ca nu stim cat e tot proiectul , pe cand la waterfall stim cat e proiectul
- Sigur, ca progress la procesele iterative pot spune ca avem progress, dar nu il pot manifesta numeric, sub forma de procente pt ca nu stim cat mai avem pana la final pt ca backlog-ul se modifica constant dupa fiecare iteratie
- Un domeniu in care se poate folosi progress bar-ul si in care avem waterfall sunt de ex in constructii -> de ex, construim o casa -> tot ce foloseste materie si construieste produse poate avea progress bar -> fundatie e 5% etc.
- La RUP se incearca un model hybrid, iar pt asta avem 4 faze, nu se intretesc :
    - Inception : se defineste la ce urmeaza la fie folosit sistemul
    - Elaborare : se specifica riguros cerintele, se vine cu o arhitectura, se face un concept general
    - Constructie : se face implementarea, testarea etc.
    - Tranzition : produsul trece in zona operationala si incepe sa functioneze in productie -> produsul devine operational
- Pare ca e waterfall 100% -> dar in interiorul fiecarei faze se fac multe iteratii , se lucreaza iterativ
- **Faza nu e totuna cu activitatea !!!!** -> la waterfall prima activitate facuta e captarea cerintelor -> ce activitate facem? Captarea cerintelor -> diferenta e ca **intr-o faza se executa mai multe activitati -> in fiecare faza se executa mai mult sau mai putin un subset din activitatile tipice de dezvoltare SW**
- Fazele nu implica un waterfall, ci doar o secventa de milestone-uri, dar inteteserea activitatilor se intampla ca la procesele iterative
- **Ideea de matrice de activitati/faza -> intr-o faza pot sa am, cu ponderi diferite , mai multe activitati si o faza nu e ceva ce se executa unitary in bloc, ci se executa iterativ, iar in fiecare iteratie se reevalueaza etc., ca in procesee iterative**
- De la RUP (rational unify process -> mai invechit, dar inca folosit) am invatat ca el a impus dezvoltarea iterativa, am invatat sa captam mai bine cerintele (usecase-uri), am invatat sa

modelam SW folosind diagrame UML (unify process -> au facut un model unficat, rational -> asa se numea firma)

- RUP e mai lent decat SCRUM -> nu e waterfall nici el, in ciuda cascadarii de faze care aduce a waterfall, faza nu se suprapune cu activitatea -> in interiorul unei faze pot sa ai mai multe iteratii si mai multe activitati, cu ponderi diferite -> RUP e un process iterativ si incremental, pt ca sistemul se construieste incremental -> se spune ca e risc meet aware process din felul in care se fac iteratiile sau verificare la sf fiecarei iteratii se estompeaza riscurile

---

---

CAPTAREA CERINTELOR (requirements engineering)

- Cum sa construim un sistem? Avem nevoie de o pereche de roluri : nevoie de intelegere a domeniului aplicatiei (un business man care sa puna in ordine viziunea clientului, sa preia elementele care reprezinta acele goluri din viziunea lui si sa le defineasca intr-un mod coerent), dar si de o nevoie de intelegere a partii tehnice (un specialist cre sa inteleaga cum se pot transforma aceste lucruri intr-un sistem SW)
- Ce sa faca sistemul si cum sa faca sistemul? **E nevoie in prima faza sa vedem care vor fi interactiunile dintre utilizatori si sistemul in sine -> SUPER IMPORTANT**
- Pentru intrebarea cum se va manifesta sistemul : ex, am un editor de poze -> features : poate mofica parametrii de imagine, poate reduce zgomote (bucati mari de functionalitae -> usecase-uri) etc. ; mereu am un CUM pentru un CE
- Putem avea mai multe scenarii pt un usecase -> urmatorul pas e sa scriem cod -> pt fiecare usecase sa identificam clasele si obiectele, responsabilitatile si interactiunile intre clase
- Cand pornim sa facem usecase-uri, avem un mod systematic : cine sunt utilizatorii sistemului (actorii) ? -> un usecase serveste un actor al sistemului -> cine interactioneaza cu sistemul? -> care sunt interactiunile fiecaruia dintre actorii sistemului? -> pt fiecare usecase, cum se va desfasura el? Sunt mai multe moduri/scenarii.
- Avantaje : este sistematic si am un process ierarhic(am putini actori -> roluri diferite , fiecare are un nr limitat de usecase-uri -> pot sa vad cum interactioneaza cu sistemul)
- Pt fiecare usecase defines un set de scenarii -> ca sa imi spuna un mod de utilizare al sistemului
- Actorii sunt entitati externe sistemului care comunica cu sistemul-> nu trebuie sa fie o pers fizica, un om -> poate fi un om sau un alt sistem care comunica cu acest sistem
- Daca un sistem comunica cu alt sistem si doar schimba date si nu ii poate vedea codul celuilalt sistem spunem ca acel sistem este extern
- **Actorii nu fac parte din sistem -> actorul poate sa fie alt sistem, dar nu poate fi o parte din sistemul pe care deja il modelezi**
- Actorul e mereu cineva care interactioneaza cu sistemul : poate ori introduce,ori primi informatii din sistem ori ambele
- Spre ex, am doua persoane Ale si Ioana care lucreaza la primarie care interactioneaza cu sistemul si vor sa inregistreze masini -> nu voi avea 2 actori , ci voi avea unul singur, pentru ca ambele persoane au acelasi rol -> Acela de a introduce masini (chiar daca ele sunt entitati diferite, pt sistem ele vor reprezenta doar o entitate) -> daca era vreo diferenta legata de modul in care acestia introduce masini, atunci da, se considera ca erau 2 actori

- De ex, am un actor professor si am un actor student -> am un al treilea actor doctorand? -> nu, pt ca el poate juca si rolul de professor(preda si el), dar si rolul de student(inca e la doctorat)
- Usecase-ul e o bucata mare de functionalitate -> are multe scenarii , dar unul e default
- Cum identific un Usecase? Produce ceva util pentru un actor?

---

---

**MODELAREA**

- Avem usecase-urile care contin o multime de scenarii
- Scenariile sunt formate din : clase si obiecte, metodele claselor si colaboratorii care reprezinta alte clase -> ele interactioneaza
- Pt fiecare scenariu avem cate o **diagrama de secventa** si o **diagrama de clasa**
- Diagr. De secventa sau un set de diagrame de secventa -> putem construi o diagram de clase din ce in ce mai bogata
- Asta de mai sus reprezinta idea de CRC cards -> ne ajuta sa identificam clasele si obiectele
- Cum gasim clasele? Ascundem datele si orferim servicii
- Am captat cerintele -> aflam care sunt usecase-urile -> am modelat clasele -> am facut relatia dintre ele -> am gasit digrama de clasa si de secventa -> nu mergem mai in detaliu

EURISTICI PENTRU MODELAREA ORIENTATA PE OBIECTE :

- Problema proliferarii claselor : programarea goto e foarte dificila -> apare spaghetti code -> Solutia ar fi programarea structurata -> am inlocuit spaghetti code cu ravioli code- > am niste bucati mici de paste, dar toata farfuria e plina de paste -> control flow e foarte clar, stiu de unde pana unde se duce, stiu cine apeleaza pe cine , dar nu stiu daca am o mie de clasa, pe care clase ar trebui sa le modific ca sa implementez eu stiu ce feature -> te trezesti pierdut in miile de clase -> cum se poate Evita? Sa evitam entitatile care sunt in afara sistemului (ne putem prinde din ex cu calatorul -> calatorul e de fapt actorul aici)
- **O clasa e cu adevarat clasa atunci cand pe ea o apeleaza alte clase -> faptul ca ea apeleaza pe altele nu garanteaza ca e o clasa, dar daca e apelata de alte clase atunci ea e sigur o clasa**
- O alta situatie care ar putea duce la proliferarea claselor ar fi tentatia de a modela fiecare element/substantiv ca fiind o clasa distincta
    - Am 4 clase : family, mother, father si child -> mother si father ar putea fi scrise sub forma aceleiasi clase person/adult? Da -> elementul decisiv care ne arata daca trebuie sa avem clase diferite sau nu este comportamentul -> daca 2 obiecte ofera aceleasi servicii , atunci nu are sens sa modelez cu 2 clase distincte -> O clasa inseamna : am un comportament distinct pe care altcineva nu il are sau nu poate fi executat de alta clasa

**DIAGRAME DE CLASE:**

- **Repr potenialele relatii intre clase**
- Avem clase cu nume, attribute si metode
- Avem si semnatura
- Obiectele pot fi reprezentate si ele -> obiectele sunt instante ale claselor
- Un obiect e ceva real -> o clasa e reprezentarea oricarui obiect de tipul clasei respective, nu e ceva real, palpabil
- Cand avem * -> un obiect TarifShedule poate avea oricate obiecte Trip -> ma intereseaza multiplicitatea obiectelor, nu clasele

- Elem essential e nr de relatii posibile intre instantele acelor clase , relatia dintre obiectele acelor clase
- Elem essential e practice lucrurile care se vad cel mai putin -> astea se leaga cel mai tare de relatii
- Relatie 1 la 1 : 1 obiect de tip tara e asociat cu 1 obiect de tip oras -> relatie speciala
- Relatie 1 la mai multi : 1 obiect de tip polygon e asociat cu mai multe obiecte de tip Punct -> * poate sa insemne 0 sau mai multe puncta -> spunem mai correct 1 obiect de tip polygon e asociat cu 3 sau mai multe obiecte tip punct
- O diagram banala -> conteaza enorm multiplicitatea relatiilor
- Agregarea -> e ft importanta pt felul in care percepe lumea in general -> specifica clar cate obiecte se afla in relatia cu obiectele clasei parinte
- Ex: avem o clasa parinte masina si 2 subclase motor si usa -> avem relatie de agregare intre clasa masina si subclasa motor in care 1 obiect de tip masina poate avea 1 singur obiect de tip motor -> avem relatie de agregare intre 1 obiect de tip masina si subclasa usa : 1 obiect de tip masina poate avea intre 2 sau 4 obiecte tip usa si doar atat
- Avem un caz si mai special care se numeste compozitie : ne da o forma mai stricta a relatie de agregare
- Intrebare : obiectele motor si usa se construiesc special pt a construi acest obiect masina si daca obiectul masina se distruge, se vor distruge si ele? Sau ele sunt independente de obiectul masina?
- Diferenta dintre compozitie si agregare : cand ai o relatie de agregare, relatia e mai slaba, obiectele motor si usa nu trebuie sa fie special create pentru a crea un obiect masina, ele pot exista inainte si pot fi folosite si de masina asta noua, iar cand se distruge obiectul masina nu e neaparat sa se distruga si aceste obiecte de tip motor si usa -> in schimb, la relatia de compozitie e fix invers, cand se distruge obiectul ticketmachine automat se distruge si obiectul zonebutton
- Asemanare : ambele functioneaza pe principiul “are un” , “ are o” sau “are mai multe” -> la ambele nu poti sa construiesti obiectul intreg daca nu am acel nr indicat in diagrama
- Relatia de generalizare : mostenirea, dar nu in totalitate
- **Cum reduc proliferarea claselor? Solutie : mai multe obiecte pot fi instante ale aceleiasi clase**
- Procesul de generalizare : pornim de la 2 clase oarecare pe care le gasim cancelButton si zoneButton -> ele au butonul comun, deci se poate generaliza sau se poate da factor comun si putem construi o clasa Button
- Generalizarea simplifica modelarea prin eliminarea redundantei

**DIAGRAME DE SECVENTA**

- **Interactiuni intre obiecte !!!!**
- Aici se reprezinta obiectele, nu clasele -> putem avea mai multe obiecte instanta ale aceleiasi clase
- Cum reprezentam interactiunea intre obiecte? Prin sageti
- **Aici e foarte importanta ordinea!!!! -> ordinea sagetilor imi da ordinea interactiunilor intre obiecte**
- Ex de diagrama de secventa : construirea unui executabil dintr-o sursa

- Ce se intampla cand o metoda compile() se apeleaza pe un obiect Compiler? Obiectul Compiler interactioneaza cu obiectul fileSystem . De ce? Ca sa incarc toate fisierele sursa (se indica pe sageata)
- De unde stiu cand se termina un apel? Uneori, cand pe return se construieste un obiect nou, cu care mai apoi voi interactiona -> e ft important sa pui sageata de return ca sa stim ca de acolo ai obiectul ala
- Ce vrea sa ne arate diagram asta de secventa e ca un obiect poate exista sau poate fi creat si mai tarziu in program, nu e neaparat la inceput, si se si pot distruge mai repede
- Branching flow : avem conditii de salt in secventa
- Se mapeaza ideea de scenariu pe ideea de secventa -> o diagrama de secventa se naste de obiceu dupa captarea cerintelor, dupa modelare, aplicand tehncia CRC cards si identificand relatia de obiecte

---

---

**SW DESIGN**

- Intr-o app de mari dimensiuni -> ne trebuie mecanisme de gestionare a complexitatii unui sistem
- De ex, am folosit CRC cards si am gasit toate clasele din sistem -> nr de clase e 1500 -> cum le gestionam? (astea sunt rezultate doar din modelare)
- Pt asta avem nevoie de proiectarea SW-ului -> SW design (design in sensul de proiectare)
- Ca sa avem o ordine intre aceste dependente si ca sa specificam clar ce poate sa foloseasca o componenta dintr-o alta component folosim INTERFATA -> ea trebuie sa fie ft bine delimitate ca sa delimitam ft bine interactiunile dintre componente
- O clasa e o component? Da , clasa in sine e un elem de proiectare -> e o component -> dar clasa e o forma de abstarctizare, dar nu e suficienta
- **Elem cheie de aici e ideea de descompunere -> mai exact component care rezulta in urma descompunerii**
- Pornim de la o problema mare si prin metoda “divide and conquer” , descoperim componentele de la nivelul de la care facem descompunerea, scriem interactiunile dintre componente si repetam ciclul asta pt ajungem la un nivel de rafinare destul de bun
- Ce e o componenta? Se mai poate spune ca sunt entitati, subsisteme etc. -> ele ascund un volum de complexitate functionala si interactioneaza cu alte componente -> o alta descrier ear fi ca o component e o entitate SW care incapsuleaza o abstractiune
- Abstractizarea e un fel de a modela
- O componenta poate fi si un set bine delimitat de task-uri -> un grup de programatori -> un SCRUM team ar face
- Putem face un system sw care pare perfect, dar de fapt sa nu fie asa-> cum ne dam seama? In timp
- **Un sistem e modular atunci cand el este format din componente autonome(libere,independente) conectate intre ele printr-o structura coerenta si simpla (relatiile sunt dependente)**

- Caracteristici principale : autonomia sa fie cat mai mare (coeziunea unui element sa fie cat mai mare-> cat de bine definit e el) -> de ce sa fie mai mare? Pt ca dependetele cu celelalte module sa fie mai putine, mai simple

**5 CRITERII DUPA CARE NE PUTEM DA SEAMA CA UN SISTEM E MODULAR**

- Un sistem e **decompozabil**? Da -> indifferent la ce nivel sunt, daca pot sa descompun component in componente mai mici -> sistemul e decompozabil -> nu e ceva binary, avem mai multe nivele, pe un anumit nivel poate fi decompozabil, iar pe nivelele de mai joss a nu si invers
  - **ex bun**: **programare TOPDOWN** -> am un input, il citesc, il scriu la output
  - **un contraexemplu** ar fi atunci cand undeva **exista un modul de initializare (la un counter)** : mereu cand avem ceva care face de toate pt toti avem decompozabilitate (cand ceva face prea multe) -> pt ca nu mai avem ierarhia de descompunere
- Un sistem e **compozabil**? Da -> componentele mai mari se formeaza din componente mai mici? Nu neaparat -> se dau niste componente si din ele se pot construi mai multe sisteme
  - **Contraexemplu** : ai mai multe piese pt a construi un dulap -> daca le iei pas cu pas, va duce la produsul final, un dulap -> piesele respective (care sunt pt dulap) vor construi doar un dulap -> e modular, dar respectand regula de compozabilitate -> mobila a fost gandita modular ca sa se poata descompune (folosind criteriul decompozabilitatii) ca mai apoi tu sa o poti compune (criteriul compozabilitatii) -> **dar e decompozabilitate** -> de ce nu e compozabilitate ? pt ca din piesele astea pot face doar un lucru, nu mai multe -> ca sa poata fi compozabilitate, trebuie sa avem piese care sa poata sa faca mai multe lucruri
  - **Un exemplu** : ar fi piesele de lego, dar nu intotdeauan -> cand de ex avem turnul Eiffel construit din piese de lego, il descompui in piese mai mici care **sunt specifice(nu se pot refolosi pt altceva), NU GENERICE** , dar cand il recompui tot turnul Eiffel ti-l vei construe -> ceea ce duce la criteriul de decompozabilitate -> in schimb, daca folosesc niste piese de lego clasice, GENERICE , pot construi orice cu ele (dim mai mici sau mai mari) -> doar in cazul asta avem compozabilitate
  - **Compozabilitatea e un super obiectiv** : daca fac decompozabilitate in loc de compozabilitate, adica daca am un sistem modular care e proiectat in asa fel incat sa “break down in multiple levels” , adica sa pot vedea sistemul ala descompus succesiv in mai multe piese -> atunci sistemul meu e in continuare modular -> **nu e neaparat sa fie compozabil ca sa fie modular , e sufficient sa fie decompozabil ca sa fie modular**
- Un sistem e **usor de inteles**? Aici e clar vorba de autonomia modulului -> pot sa inteleg un modul in izolare? **Ideea e sa fie modulul sufficient de autonomy ca s ail pot intelege in izolare** --->
  - Ex: o functie calculeaza radical , apoi vreau sa calculez log din ceva , fac o functie noua? Nu -> fac o functie numita calculeaza care primeste un flag (ca un enum, o valoare) -> si atunci, daca val aia e radical -> face x lucru , daca val aia e de ex 7 atunci face logaritmul -> tu nu stii exact ce va face functia aia -> sensul unui modul e dat de cine il utilizeaza -> se numesc **dependente secventiale** -> daca il apelez pe b dupa a , va face altceva decat daca l-as apela pe c inainte de b sau altceva daca l-as apela pe b si apoi pe a -> sensul

modulului b nu e independent de context -> **noi vrem sa construim module cat mai independente de context**

- Un sistem are **continuitate**? Pot sa modific intr-un loc fara sa trebuiasca sa modific in tot sistemul? Va afecta o componenta sau toate componentele? Continuitatea se refera doar la schimbari mici? Nu. Se refera la orice schimbare. -> exista o proportionalitate intre cerintele de schimbare si impactul schimbarii in cod -> adica daca avem niste cerinte mici, vom avea de facut si schimbari mici? Sau daca avem cerinte medii, pot sa deduc ca vor fi afectate cateva module, dar nu ft multe? Si doar daca cerintele/schimbarile sunt extraordinar de mari, doar atunci va fi impactat tot sistemul? La intrebarile astea ne gandim cand vine vorba de continuitate
  - Daca vreau sa vad ca o functie e continua, la functia aia am un argument x -> f(x) -> ma astept ca daca valorile lui x sunt apropiate pe axa Ox (care e ordonata) , atunci si valorile lui f(x) sa fie apropiate pe axa Oy -> asta inseamna continuitate
  - La continuitate -> trebuie sa existe o proportionalitate intre schimbarea cerintelor, amplitudinea schimbarii codului si amplitudinea schimbarii cerintelor
  - **Contraexemplu** : avem constante : val 6 in 100 de locuri de program -> vreau sa modific pe 6 in 7 -> daca 6 are mai multe semnificatii in program? De ex 6 repr intr-o parte a codului luna a 6-a dintr-un an, in loc repr a 6-a zi din sapt , in alt loc repr nr 6 de pe un zar samd. -> vreau sa schimb a 6-a zi cu a 7-a zi -> din cauza ca nu am folosit constante simbolice, ci am lucrat cu valori hardcodate -> te costa ft mult pt ca trebuie sa revizitezi toate cele 100 de instante in care am folosit 6
  - **Exemplu** : constantele simbolice
- Ce presupune protectia? Porneste de la ideea de izolare, dar nu izolare de impactul schimbarii, ci izolare de impactul erorii -> daca se defecteaza un modul, va crapa tot sistemul sau va afecta doar un nr mic de alte module? Ex: un vapor are mai multe cabine, ele sunt compartimentate -> daca se sparge un compartiment, apa nu se va putea duce in toate celelalte compartimente -> nu se va scufunda vaporul -> regula e ca compartimentele sa fie ft bine delimitate -> etans delimitate -> **ca o eroare sa nu poata afecta tot sistemul**
  - E ultimul criteriu de modularitate
  - **Exemplu** : Cea mai buna metoda de protectie e sa verificam inainte de a incepe executia ca inputul este cel bun -> adica nu pornim executia decat daca verificam ca inputul dat este acceptabil de modulul respectiv -> de ex, daca primesc un nr negativ nu execut pt ca e posibil sa crape ceva -> raportez undeva ca am lucrat pe un input eronat -> cand validez inputul inainte de excutie introduc un factor de protective ca sa evit un rezultat eronat
  - **Contraexemplu** : mecanismul de exceptii -> mai exact, o tartare nedisciplinata a exceptiilor -> de ex, las toate exceptiile sa urce pana in main, nu conteaza sau nu prind niciun fel de exceptii sau le tratez aiurea -> risc sa se propage o exceptie/eroare dintr-un loc in care a aparut catre nivelele superioare care pot fi in cu totul alte module -> nu vrem asta , acest lucru tine de protective

---

---

**5 REGULI DE MODULARITATE ALE LUI MAYER**

- De ce e mai profitabil SW-ul?
  - ✓ Pt ca nu avem costuri de material palpabil, de ex table, sticla etc., asta tine de manufactura, iar de ex : daca o fabrica de chip-uri nu va mai produce atat de multe chip-uri ca pana acum, adica va fi o criza de chip-uri, atunci automat va duce la producerea unui nr mai mic de masini -> de asta e mai

**Cap 5**

**SW DESIGN**

- Scopul proiectarii SW e sa invinga complexitatea -> ideea e sa identific un nr gestionabil de entitati/componente si apoi sa descriu relatiile dintre ele
- Proiectam SW -> detaliere succesiva pe mai multe nivele
- Un SW e complex -> e format la randul lui din componente complexe-> trebuiesc rafinate -> trebuie sa proiectez din nou facand zoom-in pe fiercare componenta-> rafinare succesiva si detaliere succesiva pe mai multe nivele-> **e decisiv in a invinge complexitatea si in a proiecta SW**
- O componenta poate ascunde o parte de complexitate -> da acces altor componente la acea functionalitate complexa printr-o interfata
- O componenta nu e doar o entitate de program -> ea e necesara si pt nevoile legate de comunicare intre echipele care vor implementa un sistem SW.
- Un sistem SW e bine proiectat at cand minimizeaza costurile pe toata durata a sistemului SW -> e ft greu pt ca confirmarea ca l-ai proiectat bine vine dupa ft mult timp
- Cand ma uit la un sistem SW -> cum imi dau seama daca e bine proiectat sau nu? : iti dai seama in timp -> e nevoie de niste criterii de evaluare
- Cand vine vorba de proiectare -> apare "**modularitatea**" (concept)
- Cand un sistem e modular? -> este at cand e format dintr-un set de componente care au niste caracteristici si identitate clara (e declara in interfata ft clar/in serviciile pe care le ofera) -> ele sunt conectate intre ele prin niste relatii simple si cat mai coerente/logice
- **5 criterii si 5 principii generale de proiectare SW**
- 5 criterii de modularitate (cum ne dam seama ca un sistem e modular) :
  - Decompozabilitate : un sistem intreg se descompune intr-o suma de subsisteme : fiecare subsistem/compon enta are o identitate ft clara si au o relatie de agregare ft clara (relatie parte-intreg, de subsumare a unui intreg) -> scopul decompozabilitatii e de a diviza munca intre diferite echipe ->
    - Exemplu **: fiecare modul sa fie cat mai autonom si dependentele sa fie cat mai divizate -> top down design :** am de ex un program care face x lucru, noi prima data ar trebui sa cream functionalitatea principala a lui, apoi de apucam de subprobleme/submodule

        - Contraexemplu : **initializarea unui modul** : sa initializam o variabila pt tot sistemul -> atunci ar disparea ideea asta de autonomie, de ex cand initializam o variabila globala care poate fi folosita de tot sistemul
    - Compozabilitate : am un set de module independente -> din ele pot sa construiesc oricate sisteme vreau -> combin modulele astea ca sa obtin sisteme noi -> ideea de **reutilizare** :
        - Exemplu si contraexemplu in acelasi timp : pot avea o figurina care se poate construi doar din piesele alea, cu ele nu se poate construi cu altceva!!!! -> nu e compozabilitate, ci decompozabilitate -> un exemplu pt compozabilitate ar fi niste piese de lego care sunt standard, poti construi mai multe figurine din ele -> sistemul e compozabil at cand folosesc piese reutilizabile
        - Alt exemplu : librariile matematice, comenzile si pipe-urile UNIX
    - Inteligibilitatea : un sist trebuie sa fie usor de inteles -> capacitatea unui modul de a fi inteles cand e singur/in izolare, cand se ia separat -> cand ceva nu are un inteles unitar, devine mai greu de inteles -> el trebuie sa aiba relatiile cu celelalte module cat mai limitate -> sa nu trebuiasca sa inteleg alte 20 de module ca sa vad ce face modulul respectiv + sa nu aiba flag-uri care daca ii pui un flag de ex sa faca ceva si daca un pui alt flag sa faca altceva
        - Contraex : dependenta secventiala : nu poti sa intelegi modulul B decat daca inaintea lui vine A : ideea e ca ai nevoie sa intelegi si alte module ca sa il intelegi pe cel curent
    - Continuitatea : cand avem  cerinta de schimbare -> impactul sa fie mic ; daca apar schimbari intr-un modul ma astept sa nu afecteze atat de mult celelalte module -> schimbarile sa se faca in modulul in care e necesar sa se faca schimbarea si apoi sa avem de modificat putin in celelalte module conexe -> sist e discontinuu at cand apare o cerinta de schimbare si trebuie sa fac enorm de multe modificari in multe module : **schimbarile necesare trebuie sa aiba un impact mic asupra codului/sistemului -> sistemul indeplineste criteriul de continuitate**
        - Exemplu : constantele simbolice (ex: pi etc.) : e putin probabil ca alte variabile sa aiba val 3.14, dar in schimb daca vreau sa caut 42 de ex, ea poate reprezenta multe lucruri in program, poate repr minutul, poate secunda, poate kilogramele etc.
    - Protectia : se refera la impactul erorilor -> cand apare o eroare/exceptie -> impactul ei trebuie sa fie cat mai mic asupra intregului sistem
        - Exemplu : Validarea inputului la sursa
        - Contra exemplu : tratarea gresita a exceptiilor : modulele nu sunt autonome dpdv al tratarii erorilor
- 5 reguli de modularitate (cum putem face ca un sistem sa fie cat mai modular):
    - Direct mapping : trebuie sa te asiguri ca avem o mapare simpla, directa intre structura solutiei si structura problemei modelate
        - Avantaje : impact asupra : continuitate (o schimbare afecteaza putin celelalte module) -> concepte unitar definite , decompozabilitate

- putine interfete : cu cat am mai putine legaturi intre module, cu atat imi va fi mai usor cand voi avea de schimbat ceva, nu va impacta grav tot sistemul -> un modul sa comunice cu cat mai putine alte module, dar nici cu 0 module
- Interfete mici : conteaza si cat de mare e latimea de banda a acestei comunicari intre module -> cat de mare sa fie interfata ?
- Interfata sa fie explicita : 2 module comunica explicit atunci cand dependenta lor e vizibila din descrierea ambelor module -> ma pot uita la un modul si pot sa vad cu cine comunica si ca acele relatii de comunicare sunt toate canalele de comunicare ale acelui modul cu restul lumii -> cum ar putea fi altfel? Avem date partajate : 2 module nu comunica direct, ci prin intermediul unor date globale la care au acces mai mult module si ele le pot modifica sau accesa -> cand comunicarea nu e explicita e o prob de intelegere -> comunicare implicita : ce am scris mai sus cu faptul ca ele comunica prin date
- Ascunderea informatiei : nu se refera doar la specificatorii de acces, ci si la module care expun interfete -> informez celalte module ce pot sa apelez din modulul respectiv prin intermediul interfetei sale : punctul cheie e echilibrul intre ce e public si ce e ascuns : principiul eisberg-ului -> mecanismele sa ascunda cat mai mult si sa expuna cat mai putin -> daca am mecanismul si 90% din metodele mele sunt publice -> nu vorbim despre ascunderea inform

- Concluzie :
  - Putine interfete : sa nu comunici cu multe module, ci cu cat mai putine
  - Interfete mici : sa faci comunicarea asta sa fie cat mai limitata : se refera atat la nr de canale de comunicare, cat si la nr de date schimbate
  - Interfete explicite : rel de comunicare sa fie publice si directe, explicite

## ARHITECTURA SW

- Exact ca la sistemele modulare -> entitati cat mai autonome care interactioneaza intre ele cat mai simplu si coerent (interfetele intre entitati si relatiile dintre ele)
- La nivelul cel mai exterior sistemului : partea de interactiune al userului cu sistemul (USER MODEL) : introduc datele etc.
- Apoi avem nivelul de requirement : cerintele (orientat spre ce face sistemul)
- Urm nivel : arhitectura
- Urm nivel : codul
- Urm nivel : executabilul
- User model + requirements : **ce e sistemul ?** -> Codul + executabilul : **ne spun cum e sistemul**, cum e el implementat -> **arhitectura e intre cele 2 categorii** -> arhitectul repr punctea dintre cele 2 maluri : cum e sistemul si ce face sistemul -> trebuie sa le armonizeze : intelege cerintele clientului, se asigura si ca ce construieste e posibil dpdv tehnic
- Arhitectura e despre lucrurile care sunt greu de schimbat!!! -> schimbare si agilitate ->

## STILURI ARHITECTURALE

- Repository / black board : avem in centrul repository-ul, care e la fel ca o baza de date -> contine toate datele de care au nevoie elementele componente -> componentele s. n. knowlegde

sources -> sunt niste entitati care care consuma date din acest repository, pot scrie, pot actualiza date etc.

- Blackboard vine de la faptul ca n-ai o solutie imediata, mai multi oameni colaboreaza, poate fi ca un puzzzle complex si mai multi oameni lucreaza la el -> toata lumea se uita la aceeasi tabla
- Concluzie : avem mai multe creiere care toate interactioneaza cu datele din repo -> pot sa aceeseze repository-ul in care se afla date fie scriind, citind etc.
- Producatorii si consumatorii de date sunt total independenti -> subsistemele nu trebuie sa-si faca griji de responsabilitati auxiliare, cum ar fi backup-ul, securitatea etc.
- Se poate adauga oricand un nou modul, un nou knowlegde source, care poate sa inceapa sa consume si sa produca date in acel model -> modulele raman independente intre ele -> tot ce trebuie sa stie e cum sa se conecteze la acel repository
- **Avantaje** : oricand am de gestionat date, exista un overhead -> nu mai e necesar sa-l rezolvam pt ca fiecare modul foloseste modelul de date comun
  - E un mod eficient de a share-ui cantitati mari de date intre diverse subsisteme/componente/entitati
  - Producatorii si consumatorii de date sunt independenti
  - Subsistemele nu au responsabilitati auxiliare (ex: sa faca backup)
- **Dezavantaje** :
  - modelul cu date comune e un compromis printre subsisteme
  - translatarile catre un nou model sunt costisitoare
  - repository-ul forteaza o politica centralizata pentru toate subsistemele

- Arhitectura stratificata : arhitectura in 3 straturi : avem ideea de concentric : ca sa ajung la miez, in centru, trebuie sa trec prin toate straturile intermediare
  - Fiecare strat are un nivel de comunicare ft limitat
  - Stratul exterior : va putea fi accesat din afara sistemului prin UI -> nu foloseste decat stratul imediat inferior(mai din interior) -> apoi acest strat imediat inferior ofera o interfata catre stratul asta exterior si acceseaza informatii din stratul imediat inferior si tot asa -> de ex, stratul din exterior “Useful Systems” nu poate sa aceeseze core-ul (centru acestor straturi)
  - Orice cerere care vrea sa ajunga in core level trebuie neaparat sa treaca prin toate straturile
  - Fiecare strat are ft putin legaturi : ofera o interfata pt nivelul de deasupra si acceseaza interfata nivelului de dedesubt
  - Se poate lua un strat si se poate reutiliza pt alt sistem pe care il construiesc
  - **Avantaj mare** :
    - securitatea : fiecare strat ar putea avea avea un mecanism de protectie/securitate prin drepturile de acces
    - e o arhitectura portabila si schimbabila
    - se poate reutiliza un strat deja folosit pt un alt sistem
    - e avantajos pt dezvoltarea incrementala
  - Dezavantaje :
    - E greu de atins o structura atat de riguroasa
    - Reduce performanta prin cresterea comunicarii

- Arhitectura Client-Server :
  - Niste module care produc niste servicii -> fiecare are propriul modul de date complet independente intre ele
  - Am o gramada de module client care consuma servicii de la servere
  - Clientii sunt complet decuplati intre ei
  - Fiecare server prezinta o interfata publica -> clientii acceseaza serviciile din aceasta interfata
  - **Avantaje**:
    - Arhitectura e distribuita : e ft usor sa adaugi noi clienti si noi servere -> orice sistem se poate reconfigura dinamic (Cand zic client ma refer la un alt modul, nu la oameni) -> E ft usor pt clienti sa se mute/conecteze pe alt server -> serverele nu au habar de clienti, sunt decuplate , serverul e o entitate de sine statatoare
    - Ideea de compozabilitate e ft puternica : microservicii -> fiecare serviciu e complet independent si are propriul model de date si poate avea propriul limbaj de programare -> din microservicii compunem un serviciu mai mare
  - **Dezavantaje** :
    - Apar probleme de performanta( comunicare prin retea) cand apare un trafic mare de date -> fiecare server are propriul modul de date , iar un client are foarte rar nevoie de date de la un singur server
    - E greu sa anticipam probleme cand integram noi date de la un nou server
- Arhitectura pipes and filters :
  - Unde am nevoie? Avem ideea de compozabilitate : modulele s. n. filtre si au 2 capete : 1 intrare si 1 iesire -> la intrare primeste niste date -> si la iesire avem nsite date procesate -> avem nuclee de procesare -> fiecare nucleu are un sg task de calcul si transmite catre output un rezultat -> trebuie sa inlantuim mai multe nuclee de procesare
  - Construim un sistem din mai multe comportamente simple si individuale (sistemul if this then that)
  - **Avantaje** :
    - Utilizat in sisteme pentru tranzactii cu carduri
    - Pot compune si pot altera comportamente ft usor, e ft usor sa adaug sau sa imbunatatesc un comportament
    - Filtrele pot fi reutilizate ft bine, comportamentele pot fi practic customizate usor
  - **Dezavantaje** :
    - ca sa pot conecta 2 noduri, ele trebuie sa vb aceeasi limba -> outputul de la un filtru nu poate sa fie acelasi cu inputul de la altul garantat -> nu pot presupune cu 2 filtre au acelasi format de date -> fiecare filtru trebuie sa despacheteze ce primeste si apoi sa impacheteze la loc -> overhead de performanta si de implementare
    - nu e buna pt aplicatiile interactive
    - filtrele au nevoie de un format comun pt datele ce le transfera
    - avem overhead : fiecare filtru trebuie sa despacheteze ce primeste si apoi sa impacheteze la loc

- NFR (non-functional requirements) :
  - Ex : la providerul de net, de ex digi, spune : viteza pe care ti-o garantez e x (cerinta functionala)
    - Daca spun ca nu poate cadea conexiunea mai mult de 5 min pe an -> cerinta nefunctionala
  - Avem cerinte de performanta , de securitate si de mentenanta
  - Performanta :
    - Nr mic de subsisteme -> componente cu granulatie mare
    - Reducerea comunicarii -> comunicare rapida
    - Performanta de timp -> n-am voie sa am latenta -> nu adoptam un stil layered
  - Securitatea :
    - Structura stratificata
    - Cel mai critic strat in interior -> nucleu
    - Validarea securitatii la nivel inalt
    - Protectie impotriva erorilor
  - Mentenanta : intretinerea unui sistem : cat ma costa daca vreau sa adaug un nou feature
    - Sistemul trebuie sa fie usor de modificat
    - Stilurile arhitecturale sunt legate de cerintele functionale
    - Cod scris in asamblare -> greu de scris si de mentinut

CAP 6

**TESTAREA SW**

- Aici vedem daca sistemul face ce trebuie
- Descompunem in verificare si validare
- **Verificare** :
  - Vedem daca implementarea functioneaza corect -> anomalii de functionare
  - Tine de functionarea fiecarei bucati in parte
- **Validarea** :
  - Se refera la cerinte -> daca a fost implementat ce trebuie si functioneaza (dupa verificare)
- **Testarea imi arata doar prezenta erorilor -> nu imi arata si absenta lor** (daca de ex imi pica un test -> am eroare, dar daca nu imi pica niciun test, tot pot sa am eroare)
- Testarea e o activitate pesimista → nu iti da nicio garantie, pt ca nr de cazuri care trebuiesc testate e ft mare, infinit de mare
- **Verificare formala -> demonstrezi matematic ca un program este corect -> e diferit de testare**
- Verificarea formala vine cu o investitie -> de calcul si de specificare a cerintelor (trebuie ca specificarea cerintelor sa fie formala -> un cost suplimentar) -> implica costuri mari -> daca se merita, platesti, daca nu -> test
- Testarea e imperfecta !!
- Avem un nr maxim de incercari de a gasi cele mai rele probleme -> cine reuseste sa gaseasca cele mai multe erori cu resursele pe care le are -> succesul erorii e dat de nr de erori pe care le poate gasi si nu de garantarea absentei erorilor

- Strategie de testare : de la mic la mare (pornesc de la cel mai mic modul pe care il pot testa)
- Avem testare unitara (testarea fiecarui modul in parte : o functie, o clasa etc.) si testare de integrare (compun intregul, testez ,integrez , repet etc.
- Apar si aici cerintele nefunctionale, ele sunt valabile in general pt un sistem SW
- **Criterii nefunctionale** : perfomanta, securitatea, mentenanta -> le-am vazut mai sus
- Niciodata testarea nu va fi completa -> de fiecare data cand un user executa sistemul SW -> programul se testeaza -> trebuie sa divizam rezultatele de test in nivele de severitate -> consideram ca avem testarea “completa” cand nu mai avem nivele de erori
- Pasi de testare
    - 1 : testam fiecare componenta in izolare : testare unitara (nu inseamna ca nu are bug-uri)
    - 2 : testare de integrare -> merge gradual -> am un singur sistem
    - 3 : teste de nivel inalt (testare de performanta, de cerinta, de utilizare, de integrare cu alte sisteme pe care nu le-am implementat noi etc.)
    - 4 : testarea cu alte sisteme pe care nu le-am implementat noi
- Avem 2 tipuri de testare : white box (vedem codul) si black box(nu vedem codul, ci doar inputul si outputul pe care il promite)
- La nivelul cel mai de jos ma astept sa am codul -> ma uit la cod si scriu teste in functie de ce inteleg din codul ala -> s.n. testare unitara si se face prin testare white box(vad cod)
- Trec la nivelul urmator -> integrarea : am 1000 de unitati, fiecare trecuta de testarea unitara -> le privesc ca pe o cutie neagra, fiecare cu intrari si iesiri -> testare black box (vad specificatii) -> incep sa testez bazandu-ma strict pe interfete -> indiferent de nivelul la care fac testarea -> tot ce e peste testarea unitara s. n. testare black box
- Se poate face bineinteles si testare de integrare unde sa am si testare white box si black box -> pe masura ce creste nivelul de integrare, tot mai putin faci white boxing si tot mai mult black boxing -> testez pe baza de in/out
- **Trebuie sa generez date de test care sa genereze erori!!!!**

**MODELUL DE TESTARE IN V**

- De verificare si validare
- Are forma de V
- Pe partea stanga am requirements analysis, system design … coding (coboara de sus in jos)
- Repr toate fazele dintr-un waterfall in partea stanga
- Ideea principala aici e ca pt fiecare faza trebuie sa existe un nivel de testare -> el e derivat din nivelul la care esti
- La waterfall clasic, dupa coding te astepti sa fie verificare si validare -> ceea ce face modelul in V : partea de verificare si validare are 2 parti si e integrata la fiecare faza
- Are cum sa facem verificare si validare la requirements de ex? -> la mom in care am requirements, pot avea un caz de test de acceptanta -> ce e un caz de test? E un set de inputuri si outputuri (testele de performanta sunt mai diferite) -> poti sa le definesti, in schimb nu poti sa le rulezi, dar e ok si asa
- In fiecare faza -> avem in minte testarea -> fiecare faza produce un nivel de testare la nivelul la care e faza
- La requirements -> facem acceptance test

- La system design -> facem system test
- La nivel arhitectural (pt ca aici avem integrarea componentelor) -> facem testare de integrare
- La nivelul module design -> facem testare unitara (proiectam fiecare modul si el trebuie testat unitar)
- Pana la modelul in V nu s-a facut nicio testare -> s-a facut cealalta jumatate a testarii -> testarea presupune conceperea cazurilor de test -> se pregateste sistematic -> se merge apoi inapoi in V si vedem care sunt stadiile de testare -> pe partea dreapta vom avea (de jos in sus) prima data testarea unitara -> apoi testarea de integrare -> apoi avem testarea de sistem si abia apoi UAT
- Modelul in V nu e un model de dezvoltare neaparat, cat e un model care spune la fiecare faza/moment de timp verificarea si validarea sistemului SW : fac asta in prima faza de coborare (partea stanga) prin a construi cazuri de test (aici nu am nevoie de cod), iar in partea dreapta am testarea efectiva
- Vin apoi si iau in ordine inversa cazurile de test ca sa pot sa ma asigur ca am testat cuprinzator/riguros sistemul
- Prima chestie e definirea cazurilor de test pe care vreau sa le acopar pt fiecare faza -> apoi trebuie sa aleg datele de test dupa ce gasesc cazurile (ex: nr negative, subunitare etc. )
- Tu stii pt datele de intrare de test ce date de iesire trebuie sa obtii -> rulezi testele, vezi ce rez ai obtinut si le compari cu rez la care te asteptai

**TESTAREA UNITARA**

- Low level forma de testare -> dar ridica cele mai mari probleme
- Testez fiecare modul in parte, in izolare -> decuplez modulul de toate celelalte module
- Daca sistemul e prost proiectat -> ft greu sa testez unitar
- Un moc e ceva fals -> o pseudodependenta
- **Izolarea modulelor e cea mai mare problema aici!!!!**
- O unitate e o clasa in POO
- Ce testez?
    - Interfata
    - Structurile de date
    - Boundary conditions
    - Caile independente
    - Caile de exceptie
- Am un modul pe care vreau sa il testez -> acesta are module de care depinde si atunci le transformam in **STUB-URI** (repr pseudomodule care dau date modulului testat ca sa ii asigure functionarea)
- Driver-ul e ca un program principal -> ca un main care emuleaza ce e deasupra modulului pe care vreau sa il testez -> stub-urile inlocuiesc modulele de dedesubtul unui modul pe care vrem sa il testam -> ca sa scriem stub-uri si drivere avem costuri in plus -> doar pt testare, ele sunt necesare
- Cand avem new Ceva() -> acel new e hardcodat, nu poate fi inlocuit -> devine foarte greu de mocuit/de inlocuit cu un stub
- Orice referinta a unei subclase poate fi folosita in locul unde se cere sa avem o referinta a clasei principale -> ajuta ft mult in testare

- E ft greu sa mocui/ sa izolezi/sa faci unit testing adevarat pt module care au multe dependente statice

**TESTARE DE INTEGRARE**

- Urcam un nivel -> vrem sa testam sistemul in ansamblu -> testarea unitara nu e suficienta -> trebuie sa testam un modul impreuna cu alte module
- Cum o facem? BIG-BANG approach -> nu e o testare de integrare !!!! -> face parte din categoria asa nu : pui toate modulele impreuna si le dai drumul -> rulezi niste teste pe tot -> problema e localizarea erorilor : ce a cauzat problema? -> aceasta testare BIG-BANG nu e buna pt ca nu poti sa izolezi cauzele, nu se pot localiza exact problemele
- Modul logic si normal de a face lucrurile e o integrare incrementala : luam gradual modulele si le integram pe rand, apoi testam
    - TOP-DOWN integration :
        - Orice aplicatie are un graf de dependente : programul principal -> **e un graf aciclic orientat** -> dar daca e ciclic? Tot ce ce ciclic e intr-un singul modul
        - Sus inseamna modulul pe care nu-l apeleaza nimeni
        - De ex, am integrat modulul m1 impreuna cu modulul m2, pe m3 si m4 nu vreau inca sa le integrez -> folosesc stub-uri ca sa le substitui
        - **Avantaje** : pornind de sus, eu testez ft bine logica de nivel inalt -> punctele de control se testeaza repede, cu prioritate si des -> pe masura ce testez si urmatoarele, cele de sus se testeaza cel mai mult pt ca se testeaza si impreuna cu urmatoarele
        - **Dezavantaje** : trebuie sa creez stub-uri si datele cu care se lucreaza nu sunt date reale, sunt stub-uri
        - Integrarea se poate face si in latime(am acoperit un nivel si pot sa cobor gradual) si in adancime (cel putin pe o ramura exersez si data flow, nu doar control flow)
    - BOTTOM-UP integration :
        - Module care nu mai depind de nimeni
        - Avantaje : aici nu am nevoie de stub-uri, pt ca pornesc de la frunze
        - Dezavantaje : am nevoie de drivere (contin o logica de utilizare) -> driverele sunt ca un fake main module -> dezavantajul e ca business logicul cu care testezi nu e cel real, ci e unul dummy, fake -> pe de alta parte, data flow-ul e ft bine testat
        - Ca mod de integrare se merge clustere, nu se integreaza modul cu modul, practic pornesti de la mai multe frunze si faci un grup de frunze si pe ala il integrezi -> apoi deasupra lor pui un driver (un main fake) -> apoi in locul driverului se pune modulul real -> repet pana ajung in varf

**TESTARE SUPLIMENTARA**

1. Regression testing

- Am integrat, testat, totul merge -> NU! -> am tot sistemul testat cu testare de integrare , dar in continuare avem cazuri de test -> testez de ex o bucata/un slice de cod, dar nu este suficient -> apare aici testarea de regresie -> trebuie sa ne asiguram ca sistemul nostru nu a regresat (de ex, daca am introdus noi feature-uri, sa nu fi introdus niste bug-uri neasteptate

- Cum se face testarea de regresie : scriu teste noi pt integrarea noului feature, dar pe langa asta mai rulez niste teste care au fost deja rulate (un subset de teste, nu toate testele care au fost rulate)
- **O testare de regresie inseamna ca sa re-executi un subset de teste care au fost deja rulate** -> avem teste care acopera o zona mai larga, cumva si in exteriorul imediat apropiat al modulelor testate

2. Smoke testing

- E o testare superficiala, cam deloc -> ruleaza un subset minimal de teste, dar care acopera principalele chestii care ar putea sa dea fail ft naspa
- Dupa ce am terminat cu integrarea -> vedem daca trece de smoke test -> double check -> ma asigur ca dupa ce am facut un build, am rulat toate testele din build(nu sunt toate testele din sistem) -> ma asigur ca build-ul ala compileaza -> dupa ce trece de smoke test pot sa il pun undeva de unde poate fi utilizat

**DESIGNUL CAZURILOR DE TESTARE**

- Cum proiectez cazuri de test?
- Ce e un test bun? -> Are sanse bune sa gaseasca o eroare si un test e bun daca nu e redundant
- La black box vorbim de input/output, iar la white box de caile din program
- White box
    - Vrem sa ne asiguram ca am parcurs toate ramurile din program -> code coverage : am trecut cu cate un test prin fiecare bloc
    - Coverage ajuta cu un singur lucru : sa gasim erori stupide care se pot strecura oriunde in cod
    - Aici avem **complexitatea ciclomatica** : repr nr de cai independente pe care le are un program (nr de suprafete inchise) -> nr de decizii simple + 1 -> tinem cont ca o decizie poate fi si compusa ( ex: a sau b, a si b etc.)
    - Daca avem de ex o instructiune if-else -> complexitatea ciclomatica e 2 ,daca avem assign -> complexitatea e 1
    - Cum testam buclele? Ex : un for -> mai multe cazuri :
        - Simple loop :
            - 1. Dam skip la intregul loop
            - 2.facem o singura trecere prin loop
            - 3.2 treceri prin loop
            - 4.N-1 si N treceri prin loop -> N e nr maxim de pasi pe care ii putem face
        - Nested loops (ciclu compus : for in for)
            - 1. Testarea se incepe de la ciclul cel mai interior
            - 2.aplic cel mai simple reguli de testare pe cel mai interior ciclu, in timp ce tinem ciclurile in exterior la valorile lor minime
            - 3.fixez ciclul interior la o valoare tipica si ma concentrez asupra ciclului exterior : il rulez cu 0 intrari, 1 intrare etc.
            - 4.repet pasul 3 pana cand cel mai exterior ciclu a fost testat
            - Daca avem cicluri concatenate -> ai un for si apoi alt for -> un ciclu concated repr un ciclu simplu, unul dupa altul, un for dupa for ->

- Black box:
  - Am niste cerinte/evenimente -> in principal setez datele de intrare, gasesc valorile lor pe care vreau sa le dau -> vad rezultatele si apoi compar rez obtinute cu cele corecte
  - Am un vector de valori -> cate valori trebuie sa dau? Trebuie sa partitionez valorile pe care le am
  - **Partitiile echivalente** -> modulul meu accepta un vector de valori -> intre 0 si 100 valori de ex -> cum in testare am un nr limitat de incercari -> impart in niste grupe si apoi aleg din fiecare grupa -> impart toate inputurile in niste gramajoare -> verificam inputurile valide si invalide
    - Impartim inputurile valide in niste gramajoare -> consideram ca fiecare valoare din grup e echivalenta cu toate celelalte -> rezervam cateva teste si pt cazurile invalide
    - Pt fiecare partitie putem sa alegem 1 test sau 3 teste -> ex : am setul de valori 100 -> am 3 cazuri : putine valori, medii si multe valori -> spun de ex ca intre 1 si 5 am putine valori , intre 5 si 30 sunt medii si peste 30 sunt multe
    - Ex : avem un program care accepta in 4 si 10 inputuri -> vector intre 4 si 10 valori -> fiecare valoare repr un nr de 5 digits si trebuie sa fie valoare mai mare de 10.000 : aleg 3 cazuri de test pt nr de valori : daca am mai putin de 4 valori, daca am intre 4 si 10 valori si daca am peste 10 valori -> aleg 3 cazuri de teste pentru nr de 5 digits : pot sa am nr sub 10.000 (sub 5 digits -> 4 digits), nr cuprinse intre 10.000 si 99.999 si pot sa am nr peste 99.999 (peste 5 digits -> 100.000 -> 6 digits)
    - Trebuie mereu sa ma uit care ar fi invalid inputs si valid inputs -> ma uit mereu la margini si mijloc -> testarea nu se face exhaustiv, ci se face “good enough”
- Testarea colectiilor
  - Testez colectii care n-au nicio valoare, apoi o valoare etc.
  - Incerc colectii/secvente cu diferite dimensiuni