Arhitectura sistemelor software

Curs 1

Scopul principal al cursului este acela de a defini o legătură cuantificabilă între analiza și proiectarea arhitecturii aplicației, adică între resursele arhitecturale ale aplicației și soluțiile arhitecturale propuse.

- relațiile dintre componentele principale ale sistemului (care depind de complexitatea aplicației)
- relațiile sistemului cu mediul în care va fi utilizat
- modul în care ajungem la implementarea aplicației pe baza arhitecturii propuse (fezabilitatea arhitecturii)

Proiectarea:
1. arhitecturală (la nivel centralizat) - arhitect software
2. componentelor (la nivel de programator)

Analiza:
1. arhitecturală (la nivel centralizat) - analist tehnic
2. componentelor (la nivel de analist / programator)
3. funcțională (de business)
4. tehnică (calitățile aplicației)

Arhitectura unui sistem software reprezintă suma deciziilor argumentate, strategice și tactice importante luate pentru dezvoltarea sistemului.

Curs 2

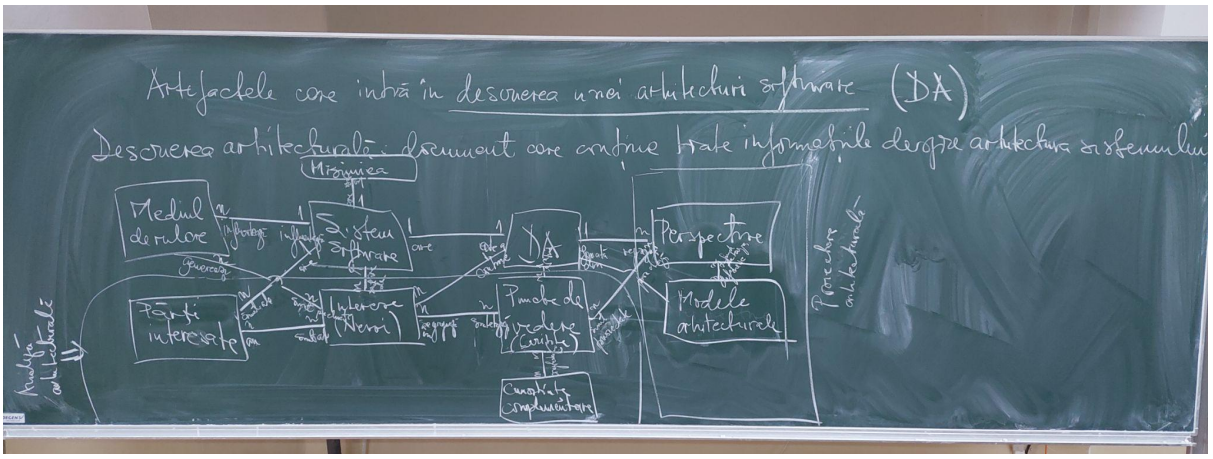
Recapitulare

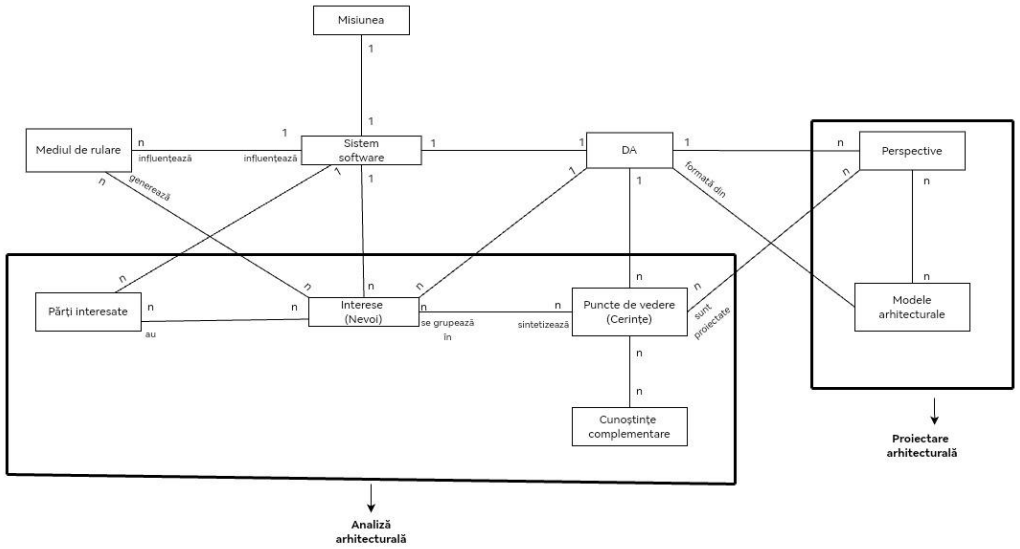
Def. 1: Arhitectura unui sistem software constă în componentele sale principale, relațiile dintre ele, relațiile cu mediul în care rulează aplicația și modul în care se ajunge la implementarea sistemului.

Def. 2: Arhitectura unui sistem software reprezintă suma deciziilor majore pe care le luăm în legătură cu implementarea sistemului.

Artefactele care intră în descrierea unei arhitecturi software (DA)

Descrierea arhitecturală : document care conține toate informațiile despre arhitectura sistemului.





Curs 3

Tipuri de cerințe:

1. funcționale (descriu funcționalitățile necesare)
2. non-funcționale (toate celelalte)
    a. cerințe de calitate
    b. constrângerile
        i. tehnologice
        ii. reglementate
            1. legale
            2. specifice

iii. de resurse
    1. umane
    2. timp
    3. buget
    4. altele

Analiza arhitecturală

Colectarea cerințelor (intereselor, nevoilor)

Principiul 1: Nicio aplicație software nu reprezintă un scop în sine. Orice aplicație software reprezintă soluția la o anumită problemă din lumea reală.

Principiul 2: Cerințele aplicației software nu se inventează (nu presupunem că ar fi într-un anumit fel). Cerințele se extrag de la persoanele sau grupurile interesate (Stakeholders)

Principiul 3: Cerințele colectate de la părțile interesate se curăță, se completează cu informații specifice colectate din literatura de specialitate, aplicații similare și se structurează în Puncte de Vedere (Viewprints)

Principiul 4: Nu toate cerințele au aceeași relevanță pentru dezvoltarea aplicației. Cerințele trebuie prioritizate

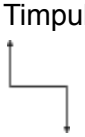
Studiu de caz (parțial). Motorul de căutare de la Google

Întrebarea: De ce este Google Search Engine una dintre cele mai de succes aplicații software din toate timpurile?

Misiunea aplicației: Oferă sugestii relevante privind locul de unde s-ar obține răspunsul căutat pentru orice subiect de care este interesat orice utilizator

Principale calități:

- Simplitatea => ușurința de utilizare
- Timpul de răspuns



- Relevanța răspunsurilor

Curs 4

Cerințe de calitate - Part 1

Tipuri de cerințe:
- funcționale
- non-funcționale
    - de calitate
    - constrângeri

Motorul de căutare Google:
- viteza răspunsului
- relevanța răspunsului
- simplitatea interfeței

Performanță

- acoperă orice comportament al aplicației în ceea ce privește unul sau mai mulți algoritmi => criteriu de evaluare al impactului acelor algoritmi în aplicație
- estimarea condițiilor în care trebuie atinși indicatorii de performanță doriți, evoluția acestor condiții în timp
- testarea performanței în condiții “reale”

Percepție și performanță

- care sunt așteptările utilizatorilor?
- tehnici de modificare a percepției (diversiuni)
- tehnici de anticipare a cererii

Disponibilitate

- proprietatea sistemului de a răspunde la solicitări

Indisponibilitate:
- nefuncționarea sistemului sau a unei anumite componente
- cu percepția unei lipse de performanță acute

Strategii de răspuns

- detecția + înregistrare
- notificare
- restart (LOL)

revenirea - înlocuirea componentei afectate -> ne interesează doar ceea ce ține de arhitectura aplicației

redundanță

- pasivă - backup - întreruperile programate
- activă

revenire (shadowing)

Curs 5

Cerințe de calitate - Part 2

Securitate

- trebuie adaptata la nevoile reale ale app:
  - atractivitatea pt infractori
  - posibile neajunsuri ale unui atac reusit:
    - date sensibile (bancare, personale)
    - actiuni sensibile (care interfereaza cu viata umana)
- nicio metoda de protectie nu e garantata 100%
- metode avansate de securitate genereaza costuri proportionale

Strategii

- preventia (implementata gradual si adecvat) -> dificultatea consta in diversitatea numeroasa a atacurilor posibile
- detectia / inregistrare / notificare
- revenirea din atac:
  - recuperarea starii sistemului din copii salvate anterior
  - shodowing (sincronizarea starilor diverselor componente ale sistemelor)

Experienta de utilizare (Usability) UX/UI

- atractivitate
- aplicatii prietenoase cu utilizatorul
- usurinta de utilizare
- eficienta de utilizare (Effciency)
- eficacitate de utilizare (Effctiviness)
- usurinta de invatare

- experienta de utilizare trebuie adaptata grupului tinta (profilarea utilizatorilor):
    - utilizatori experimentati
    - neexperimentati
    - cutume de utilizare

Securitate vs Experienta de utilizare

- parole complexe
- reintroducerea periodica a parolei
- certificate de securitate
- niveluri de securitate

Problema principala este generata de interconectarea sistemelor software.

Curs 6

Constrangeri

Recap:
Cerinte de calitate:
- performanta
- disponibilitate
- experienta de utilizare
- securitate

Exisa si altele destul de multe:
- Mentenabilitate:
    - upgrade-ul mediului de rulare
    - upgrade la app
- Modificabilitate :
    - - raportul dintre modificari si costuri
    - riscuri:
        - costuri: - provine din aplictatiile unor modificari
        - alterarea arhitecturii in timp (alterarea modificarilor de calitate)
- Testabilitate
- Scalabilitate:
    - in volum (utilizari, date etc)
    - in complexitate (functionalitatile noi)
- Portabilitatea : - medii diferite de rulare (hardware, SO, SGDD etc)

Constrangerile reprez acele cerinte non-functionale care nu contribuie la imbunatatirea arhitecturii si implicit a aplicatiei: dar care se aplica din diverse cauze si care pot influenta decizia arhitecturii aplicatiei.

Ceea ce intereseaza este: Sa gasim o solutie cat mai bune, atat dpdv functional, cat si calitativ, in conditiile impuse de constrangeri.

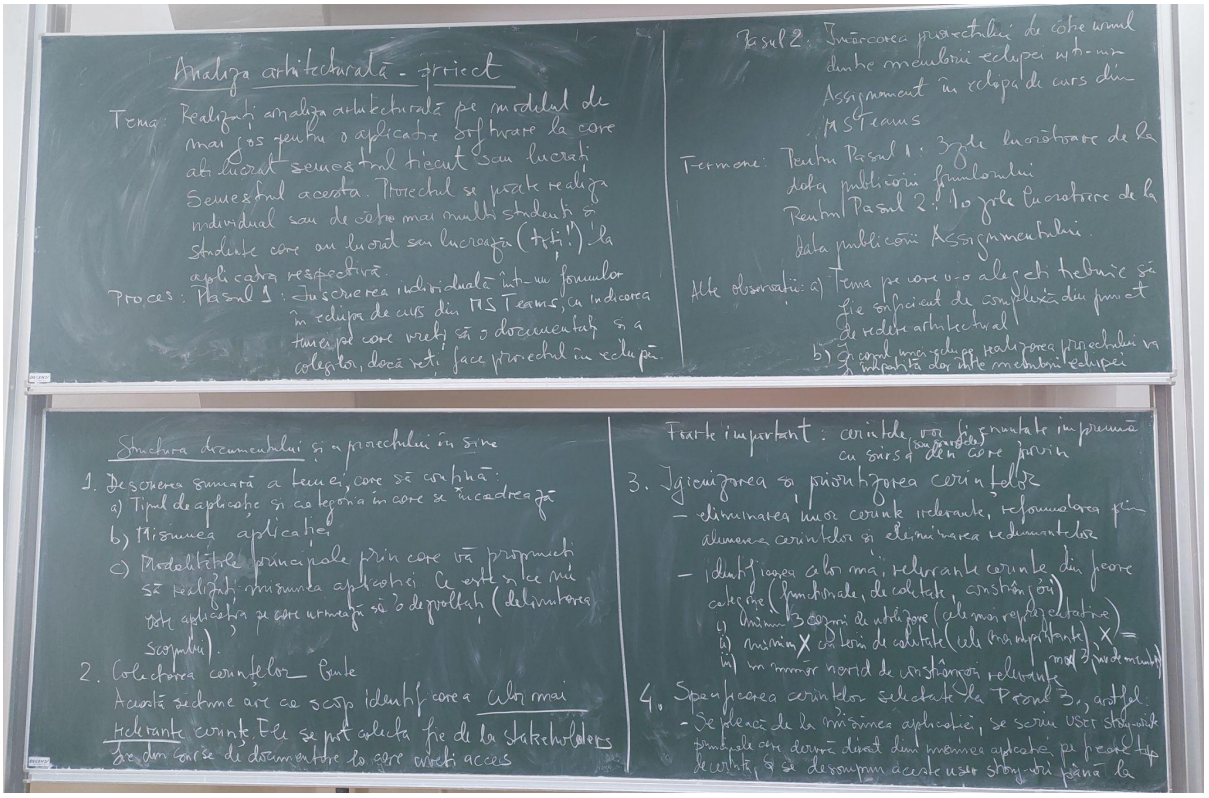
Tipuri de Constrangeri

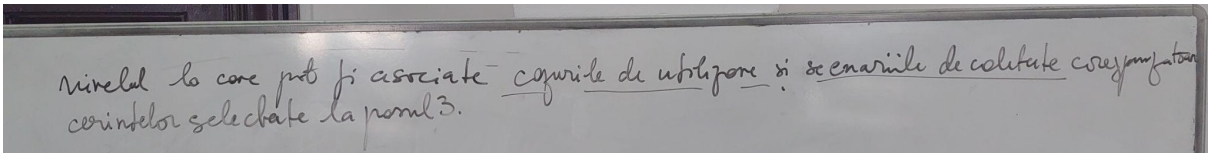
A. Fezabilitatea solutiei -> Resurse:
   - timp:
     - termene -> relevant si pt proiectele din facultate (deadline driven projects)
     - ferestre de oportunitate
   - oameni :
     - volm -> { relevant si pt proiectele din facultate } -> disponibilitate limita
     - calitate -> { relevant si pt proiectele din facultate } -> experienta limitata
   - buget -> acopera timp + oameni + altele …)

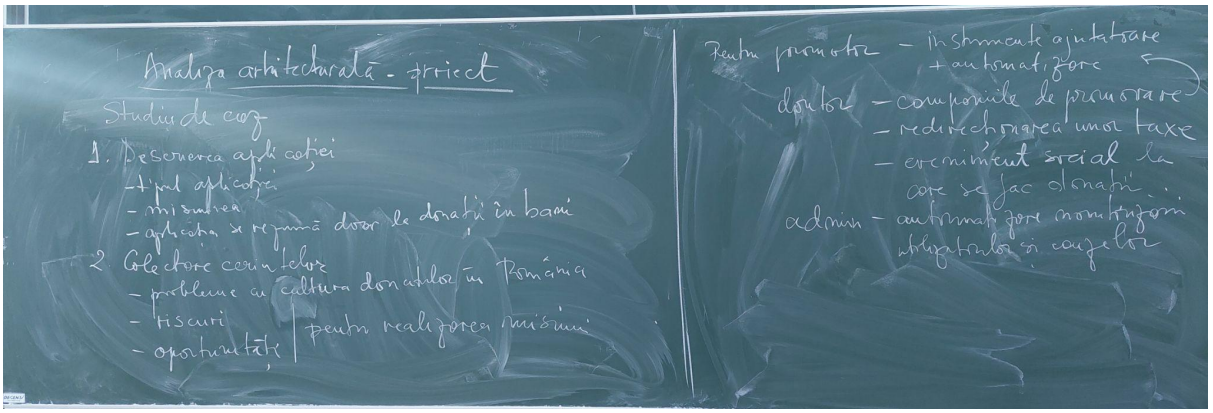
B. Reglementare la nivel de domeniu: (medical, bancar, etc)

C. Compatibilitati :
   - experienta de utilizare
   - tehnologie
   - cultura organizationala

Curs 7 - Proiect







Curs 8

Proiectarea arhitecturala

In scenariile de calitate:
- strategia de raspuns
- masura raspunsului

=> tactici arhitecturale (modele, solutii)

Descrierea arhitecturala =< Perspective (Views)
Toate perspectivele insumate ne dau o imagine a arhitecturii.
Impartirea descrierii arhitecturale in perspective ne ajuta in primul rand la claritate si comunicare.
Solutiile adoptate acopera anumite cerinte, dar nu trebuie sa inre intr-un comflict cu solutiile alese pt alte cerinte.

Alegerea tehnologiilor

- orice platforma tehnologica folosita pt dezvoltarea aplicatiei vine cu o arhitectura proprie numita arhitectura de referinta.
- platformele de dezvoltare sunt relativ inflexibile dpdv arhitectural
- alegerea tehnologiilor fixeaza cam 80% din artitectura.

Curs 9

Perspective arhitecturale

1. Perspectiva functionala (de utilizare) (cum va fi folosita app):
    - structura unei pagini
    - tranzițiile dintre pagini

        din ambele => Wireframes (diagrame de stari)

2. Perspectiva datelor
    - se alege modelul de stocare și intetogare
    - se fixeaza entitatile (principale) -> diagrama de date conceptuala
    - alte cerinte asupra datelor (criptare)

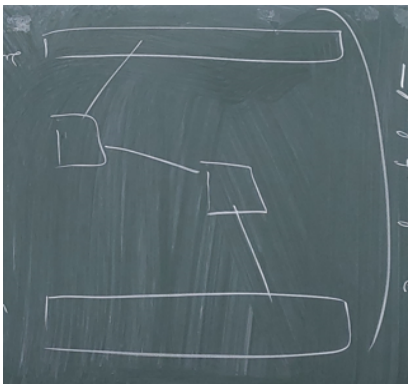
Perspectiva de utilizare |_____| interactiunea cu utilizatorii
Perspectiva datelor |_____| Modelul de date

3. Perspectiva structurala

- se definesc componentele principale ale app si relativ le
(modul in care comunica intre ele)

- Diagrama de componente
- Cand alegem o tehnologie și lucrăm pe arhitectura ei de referinta, atunci menționăm care este aceasta (eventual prezinta diagrama standard de structura a modului respectiv) și enumerăm (exemplificam) componentele specifice aplicației pe modelul dat.

Perspectiva de utilizare



Perspectiva structurala

Perspectiva datelor

4. Perspectiva comportamentala

- modul in care functioneaza app in interior prin reprezentarea unor procese tipice
- se deosebeste de Perspectiva de utilizare prin aceea ca interactiunea cu utilizatorul se face chiar la capetele procesului
- se deosebeste de Perspectiva datelor si de Perspectiva structurala prin aceea ca datele si componentele sunt primite in proces
- Diagrame comportamentale (diagrame de activitate, de stari, de secvente)
- Se insista pe aspectele de proces care compileaza anumite aspecte referitoare la cerintele evidentiate in forma forte (ex performanta)

5. Perspectiva fizica

- Diagrama de deployment
- stagiul software peste care ruleaza app
- resurse hardware
- elemente de comunicare in retea

Curs 10

Perspectiva testării

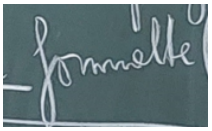
1. trebuie alese instrumentele de testare
2. momentele și frecvența utilizării lor
   1 și 2 => Strategia care este dictată de nevoia de testare

Dacă avem o cerință specifică de testabilitate, atunci soluția aleasă va trebui să țină seama de această cerință și va fi tratată.

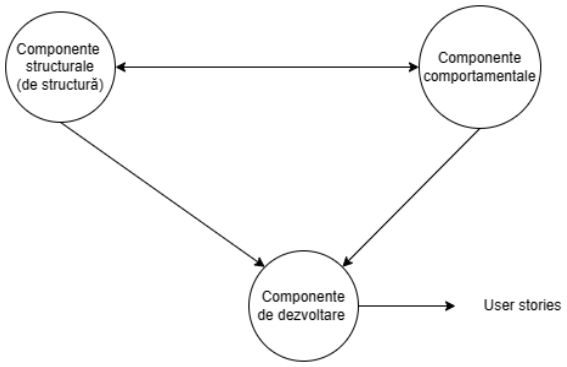
Activitate
- planificată:
    - când
    - cum
    - cu ce
- executată
- livrată
- validată

Perspectiva dezvoltării

Strategia de dezvoltare + tehnici specifice

Descompunerea aplicației în componente principale de dezvoltare și planificarea acestora conform unei strategii de dezvoltare, ținând cont de aspectele arhitecturale prezentate în celelalte perspective, dar și de toate constrângerile  (formulate?) (în etapa de analiză)

=> *Secvențializarea componentelor de dezvoltare*



Curs 11

Tactics

- **Tactic**: Design decision that influences the control of a quality attribute response
- Tactics can refine other tactics
    - Redundancy ⊆ data redundancy, code redundancy

- Example:
    - One availability tactic: introduce redundancy
    - Implication: we also need synchronization of replicas
        - To ensure the redundant copy can be used if the original fails

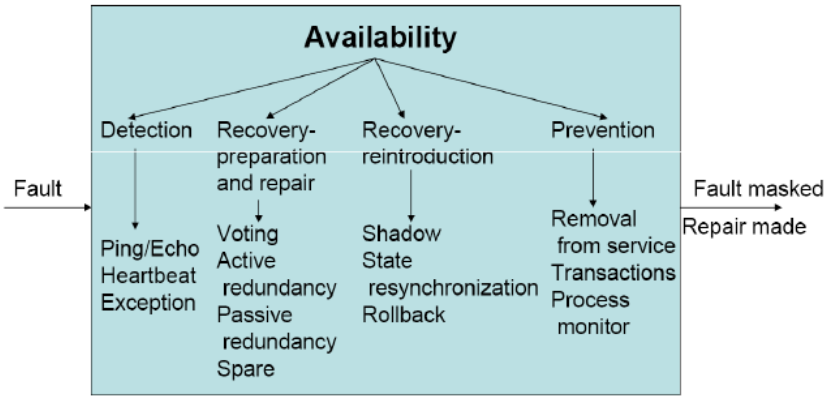
Tactics vs Strategy

OBAMA: They have done a brilliant job, and General Petraeus has done a brilliant job. But understand, that was a tactic designed to contain the damage of the previous four years of mismanagement of this war.
MCCAIN: I'm afraid Senator Obama doesn't understand the difference between a tactic and a strategy.

Availability

- Failure
    - Deviation from intended functional behavior
    - Observable by system users
- Failure vs fault
    - Fault: event which may cause a failure
- Availability tactics
    - Keep faults from becoming failures
    - Make possible repaires

Availability tactics



Fault detection: Ping/Echo

- ☐ Comp. 1 issues a "ping" to comp. 2
- ☐ Comp. 1 expects an "echo" from comp. 2
- ☐ Answer within predefined time period
- ☐ Usable for a group of components

- [ ] Mutually responsible for one task
- [ ] Usable for client/server
    - [ ] Tests the server and the communication path
- [ ] Hierarchy of fault detectors improves bandwidth usage

Fault detection: Heartbeat

- [ ] Comp. 1 emits a ”heartbeat” message periodically
- [ ] Comp. 2 listens for it
- [ ] If heartbeat fails
    - [ ] Comp. 1 assumed failed
    - [ ] Fault correcting comp. 3 is notified
- [ ] Heartbeat can also carry data

Fault detection: Exceptions

- [ ] Fault classes: omission, crash, timing, response
- [ ] When a fault class recognized, exception is raised
    - [ ] A fault consequently is recognized
- [ ] Exception handler
    - [ ] Executes in same process that introduced the exception
    - [ ] Typically does a semantic transformation of fault into a executable form

Fault recovery: Voting

- [ ] Processes running on redundant processors take equivalent input and compute output
- [ ] Output sent to voter
- [ ] Voter detects deviant behavior from a single processor => it fails it
- [ ] Method used to correct
    - [ ] Faulty operation of algorithm
    - [ ] Failure of a processor

Fault recovery: Active redundancy

- [ ] All redundant components respond to events in parallel => all in same state
- [ ] Response from only one comp used
- [ ] Downtime: switching time to another up-to-date component (ms)
- [ ] Used in client-server configurations (database systems)
    - [ ] Quick responses are important
- [ ] Synchronization
    - [ ] All messages to any redundant component sent to all redundant components

Fault recovery: Passive redundancy

- ☐ Primary component
  - ☐ responds to events
  - ☐ informs standby components of state updates they must make
- ☐ Fault occurs:
  - ☐ System checks if backup sufficiently fresh before resuming services
- ☐ Often used in control systems
- ☐ Periodical switchovers increase availability

Fault recovery: spare

- ☐ Standby spare computing platform configured to replace many different failed components
  - ☐ Must be rebooted to appropriate SW configuration
  - ☐ Have its state initialized when failure occurs
- ☐ Checkpoint of system state and state changes to persistent device periodically
- ☐ Downtime: minutes

Fault reintroduction: Shadow operation

- ☐ Previously failed component may be run in ”shadow” mode
  - ☐ For a while
  - ☐ To make sure it mimics the behavior of the working components
  - ☐ Before restoring it to service

Fault reintroduction: State re-synchronization

- ☐ Passive and active redundancy
  - ☐ Restored component upgrades its state before return to service
- ☐ Update depends on
  - ☐ Downtime
  - ☐ Size of update
  - ☐ Number of messages required for the update
    - ☐ One preferable; more lead to complicated SW

Fault reintroduction: Checkpoint/Rollback

- ☐ Checkpoint
  - ☐ Record of a consistent state
  - ☐ Created periodically or in response to specific events
- ☐ Useful when a system fails unusually, with detectably inconsistent state: system restored using
  - ☐ A previous checkpoint of a consistent state

- [ ] Log of transactions occurred since

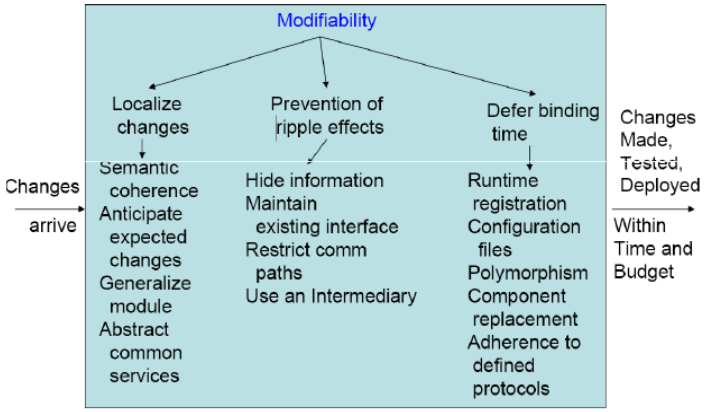
Fault prevention

- [ ] Removal from service
- [ ] Comp removed from operation to undergo some activities to prevent anticipated failures
- [ ] Ex.: rebooting comp to prevent memory leaks
- [ ] Automatic (design architecture) or manual (design system)
- [ ] Transactions
- [ ] Process monitor

Modifiability

- [ ] Goal: controlling time and cost to implement, test, modify and deploy changes
- [ ] Sets of tactics
    - [ ] Localize modifications
        - [ ] Reduce nr of modules affected by a change
    - [ ] Prevent ripple effects
        - [ ] Limiting modifications to localized modules
    - [ ] Defer binding time
        - [ ] Controlling deployment time and cost

Modifiability Tactics



Localize modifications: Maintain semantic coherence

- [ ] Semantic coherence: relationships among responsibilities in a module
- [ ] Goal: ensure all these responsibilities work together w/o excessive reliance on other modules
- [ ] Goal achievement: design modules with responsibilities in semantic coherence

Localize modifications: Abstract common services

- Subtactic of semantic coherence
- Provides common services through specialized modules
- Reuse and modifiability
    - Modifications to common services done only once rather than in every module using them
    - Modifications to modules using common services do not impact other users

Localize modifications: anticipate expected changes

- Considering the set of envisioned changes provides way to evaluate particular assignment of responsibilities
- Questions
    - For a change: does proposed decomposition limit the set of modules needing modifications?
    - Fundamentally diff. changes: do they affect the same modules?
- Goal: minimizing effects of changes

Localize modifications: generalize the module

- Generalize a module by making it compute a broader range of functions due to its input type
- Input → defining language for the module
    - Making constants input parameters
    - Implementing the module as an interpreter and making the input parameters be programs in that interpreter's language
- The more general the module
    - The most likely is that requested changes can be made by adjusting input language

Prevent ripple effects

- Localize modifications vs limit modifications to localized modules
    - There are modules directly affected
        - Whose responsibilities are adjusted to accomplish change
    - There are modules indirectly affected by a change
        - Whose responsibilities remain unchanged BUT implementation needs to be changed to accommodate the directly affected modules

Ripple effects

- Ripple effect from a modification
    - The necessity of making changes to modules not directly affected by it
    - This happens because said modules are SOMEHOW dependent on the modules directly dealing with the modification

Dependency types

- [ ] We assume
  - [ ] Module A changed to accomplish particular modification
  - [ ] Module B changed only because A changed
- [ ] There are several dependency types
  - [ ] Syntax, semantics, sequence, identity of interface, location of A, quality of service, existence of A, resource behavior of A

Syntax dependency

- [ ] Of data
  - [ ] B consumes data produced by A
  - [ ] Type and format of data in both A and B need to be consistent
- [ ] Of service
  - [ ] B invokes services of A
  - [ ] Signature of services produced by A need to be consistent with B's assumptions

Semantics dependency

- [ ] Of data
  - [ ] B consumes data produced by A
  - [ ] Semantics of data produced by A and consumed by B need to be consistent with B's assumptions
- [ ] Of service
  - [ ] B invokes services of A
  - [ ] Semantics of services produced by A need to be consistent with B's assumptions

Sequence dependency

- [ ] Of data
  - [ ] B consumes data produced by A
  - [ ] B must receive the data produced by A in a fixed sequence
- [ ] Of control
  - [ ] A must have executed previously within certain time constraints

Identity of interface of A

- [ ] A may have multiple interfaces
- [ ] B uses one of them
- [ ] For B to compile and execute correctly, the identity (name or handle) of the interface must be consistent with B's assumptions

Other dependencies

- ☐ Runtime location of A
  - ☐ Must be consistent with B's assumptions
- ☐ Quality of service/data provided by A
  - ☐ Properties involving the above quality must be consistent with B's assumptions
- ☐ Existence of A
  - ☐ For B to execute, A must exist
- ☐ Resource behavior of A
  - ☐ Must be consistent with B's assumptions

Tactics for ripple effect prevention

- ☐ Information hiding
- ☐ Maintain existing interfaces
- ☐ Restrict communication paths
- ☐ Use intermediary

Information hiding

- ☐ Decomposition of responsibilities into smaller pieces and choosing which information to make private and which public
- ☐ Public information available through specified interfaces
- ☐ Goal: isolate changes within one module and prevent changes from propagating to others
  - ☐ Oldest technique from preventing changes from propagating
  - ☐ Strongly related to ”anticipate expected changes” (it uses those changes as basis for decomposition)

Maintain existing interfaces

- ☐ B syntax-depends on A's interface
  - ☐ Maintaining the interface lets B stay unchanged
- ☐ Interface stability
  - ☐ Separating interface from implementation
- ☐ How to implement the tactic
  - ☐ Adding interfaces
  - ☐ Adding adapter
  - ☐ Providing a stub for A

Restrict communication paths

- ☐ Restrict number of modules with which the given module shares data
  - ☐ Reduce nr of modules that consume data produced by given module
  - ☐ Reduce nr of modules that produce data consumed by given module

- [ ] Reduced ripple effect
    - [ ] Data production/consumption introduces dependencies

Use an intermediary

- [ ] B dependent on A in other ways than semantically
    - [ ] Possible to introduce an intermediary to manage the dependency
    - [ ] Data (syntax)
    - [ ] Service (syntax)
    - [ ] Location of A
    - [ ] Existence of A

Defer binding time

- [ ] Decision can be bound into executing system at various times
- [ ] Binding at runtime
    - [ ] System has been prepared for that binding
    - [ ] All testing and distribution steps already completed
    - [ ] Supports end user/administrator making settings or providing input that affects behavior

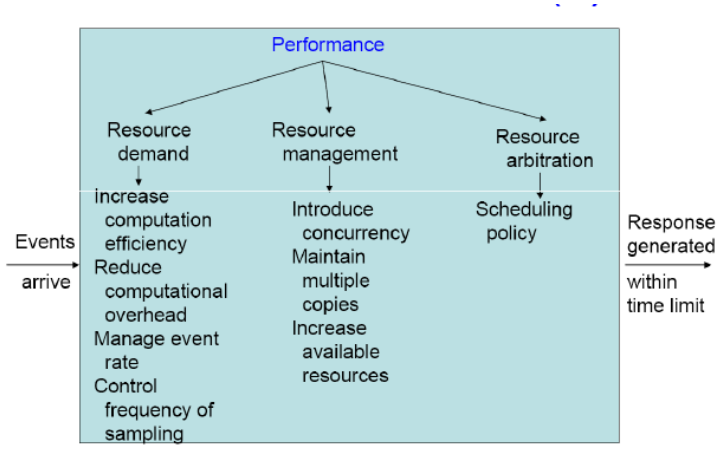
Tactics with impact at load/runtime

- [ ] Runtime registration
    - [ ] Plug-and-play operation, extra overhead to manage registration
- [ ] Configuration files - set parameters at start-up
- [ ] Polymorphism - late binding of method calls
- [ ] Component replacement – load time binding
- [ ] Adherence to defined protocols
    - [ ] Runtime binding of independent processes

Performance

- [ ] Goal: generate response to an event arriving at system within time constraint
- [ ] Event: single or stream
    - [ ] Message arrival, time expiration, significant state change etc
- [ ] Latency: time between the arrival of an event and the generation of a response to it
- [ ] Event arrives
    - [ ] System processes it or processing is blocked

Performance Tactics



Resource demand tactic

- ☐ Source of resource demand: event stream
- ☐ Demand characteristics
  - ☐ Time between events in resource stream (how often a request is made in a stream)
  - ☐ How much of a resource is consumed by each request
- ☐ Reducing latency tactic
  - ☐ Reduce required resources
  - ☐ Reduce nr of processed events

Reduce required resources

- ☐ Increase computational efficiency
  - ☐ Processing involves algorithms => improve algorithms
  - ☐ Resources may be traded for one another
- ☐ Reduce computational overhead
  - ☐ If no request for a resource => its processing needs are reduced
  - ☐ Intermediaries removed

Reduce no. of processed events

- ☐ Manage event rate
  - ☐ Reduce sampling frequency at which environmental variables are monitored
- ☐ Control frequency of samplings
  - ☐ If no control over arrival of externally generated events => queued requests can be sampled at a lower frequency (request loss)
- ☐ Bound execution times
  - ☐ Limit over how much execution time used for an event
- ☐ Bound queue sizes
  - ☐ Controls max nr of queued arrivals

Resource management

- ☐ Introduce concurrency
  - ☐ Process requests in parallel
    - ☐ Different event streams processed on different threads (create additional threads)
    - ☐ Load balancing
- ☐ Maintain multiple copies of data and computations
  - ☐ Caching and synchronization
- ☐ Increase available resources
  - ☐ Faster processors and networks, additional processors and memory

Resource arbitration

- ☐ Resource contention => resource must be scheduled
  - ☐ Architect's goal
  - ☐ Understand characteristics of each resource's use and choose compatible scheduling
- ☐ Scheduling policy
  - ☐ Priority assignment
  - ☐ Dispatching

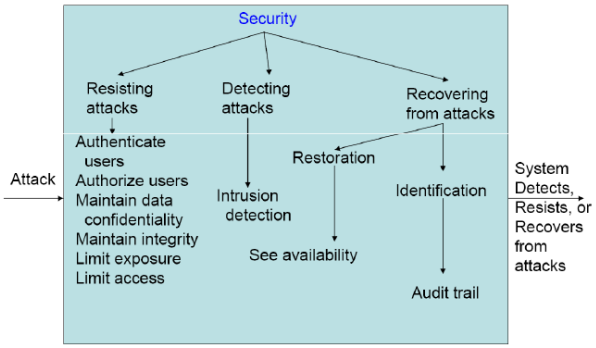
Scheduling policies

- ☐ First in/first out (FIFO)
- ☐ Fixed priority scheduling
- ☐ Dynamic priority scheduling
- ☐ Static scheduling

Security

- ☐ Goal: resisting attacks, detecting attacks, recovering from attacks
- ☐ House defense analogy
  - ☐ Door lock
  - ☐ Motion sensor
  - ☐ Insurance

Security Tactics



Resisting attacks

- [ ] Authenticate users
- [ ] Authorize users
- [ ] Maintain data confidentiality
- [ ] Maintain integrity
- [ ] Limit exposure
- [ ] Limit access

Detecting attacks

- [ ] Intrusion detection system
    - [ ] Compares network traffic patterns to database
    - [ ] Misuse => pattern compared to historical patterns of known attacks
    - [ ] Anomaly => pattern compared to historical baseline of itself
    - [ ] Filtering
        - [ ] Protocol, TCP flags, payload size, addresses, port numbers

Intrusion detectors

- [ ] Sensor to detect attacks
- [ ] Managers for sensor fusion
- [ ] Databases for storing events for later analysis
- [ ] Tools for offline reporting and analysis
- [ ] Control console
    - [ ] Analyst can modify intrusion detection actions

Recovering from attacks

- [ ] Tactics for restoring state

- Recovering a consistent state from an inconsistent one: availability tactics
- Redundant copies of system admin data
    - Passwords, access control lists, domain name services, user profile data: special attention
- Tactics for attacker identification
    - For preventive or punitive purposes
    - Maintain audit trail

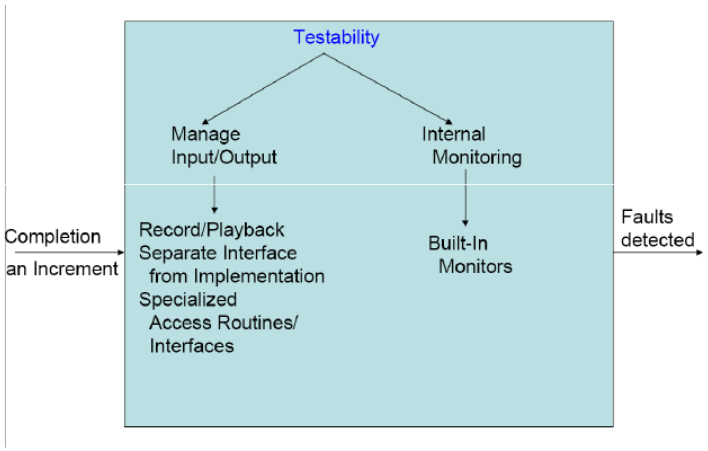
Audit trail

- Copy of each transaction applied to data in the system + identifying information
- Can be used to
    - Trace the actions of an attacker
    - Support non-repudiation
        - Provides evidence that a particular request was made
    - Support system recovery
- Often attack targets
    - Should be maintained in trusted fashion

Testability

- Goal: allow for easier testing when some increment of software development is completed
- Enhancing testability not so mature but very valuable
    - 40% of system development
- Testing a running system (not designs)
- Test harness
    - SW that provides input to the SW being tested and captures the output

Testability Tactics



I/O Tactics: Record/Playback

- Refers to

- Capturing info crossing an interface
- Using it as input to the test harness

- Info crossing an interface at normal operation
  - Output from one component, input to another
  - Saved in repository
    - Allows test input for one component
    - Gives test output for later comparisons

I/O Tactics: Interface vs. Implementation

- Separating interface from implementation
  - Allows substitution of implementations for various testing purposes
    - Stubbing implementations let system be tested without the component being stubbed
    - Substituting a specialized component lets the component being replaced to act as test harness for the remainder of the system

I/O Tactics: specialize access routes/interfaces

- Having specialized testing interfaces
  - Captures/specifies variable values for components
    - Via test harness
    - Independently from normal execution
- Specialized access routes/interfaces
  - Should be kept separate from required functions
- Hierarchy of test interfaces
  - Test cases can be applied at any arch. level

Internal monitoring tactic

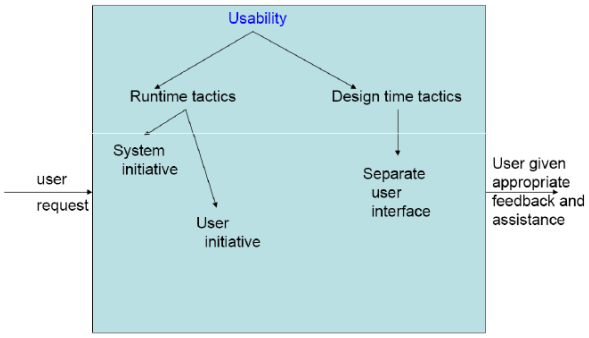
- Built-in monitors
  - Component can maintain state, performance load, capacity, security, etc accessible through interfaces
    - Permanent interface or temporarily introduced for testing
  - Record events when monitoring states activated
    - Additional testing cost/effort

Usability

- Usability concerns
  - How easy can a user accomplish desired task
  - What is the system support for the user
- Tactics
  - Runtime: support user during system execution
  - Design time: support interface developer
    - Iterative nature of interface design

- ☐ Related to modifiability tactics

Runtime Usability Tactics



Runtime tactics

- ☐ User feedback as to what is the system doing
- ☐ Providing user with ability to issue usability commands
  - ☐ Cancel
  - ☐ Undo
  - ☐ Aggregate
  - ☐ Show multiple views

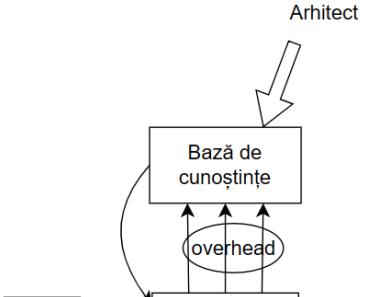
Curs 12

Proiectarea arhitecturală

- Caracteristica de proiect a dezvoltării oricărei aplicații software
- Specific pentru mediul industrial este procesul

Proiect	Proces
sunt formate din activități	
este unic	este repetabil
plan de proiect	linie de producție
importurile sunt diferite	importurile sunt identice
resursele sunt diferite	resursele sunt aprox. identice
riscuri necunoscute și imprevizibile	riscurile sunt cunoscute și controlate

- Metodologiile de dezvoltare software au un singur scop final: reducerea riscurilor.



- Există și linii de producție software care se aplică la produse din aceeași categorie și funcționează doar în organizațiile care dezvoltă foarte multe produse din acea categorie.

SEMINAR 1

Analiză arhitecturală

1. Funcțională

specificații funcționale - cazuri de utilizare(use cases) - user stories
user stories - are 3 părți:

- cine?
- ce?
- de ce?

2. Tehnică

specificatii tehnice - scenarii de calitate(detaliate)
o specificație se scrie într-un anumit format: sintetic și …?

Exemplu

Tipul aplicației:

- Platformă de donații pentru cazuri sociale care să îmbunătățească în raport cu soluțiile existente, atât numărul de cauze cât și suma care se donează. - Misiunea aplicației

Tipuri de utilizatori:

1. **Donator**
2. **Promotor**
3. **Admin**

	CINE?	**CE?**	**DE CE?**
US1	Donatorul	Susține unele cauze sociale	**Misiunea aplicației**
US2	Promotorul	Promovează cauze sociale	
US3	Admin	Asigură climatul de încredere în	

		aplicație	
US4	Sistemul	Furnizează în baza datelor caracteristici avansate	

Misiunea:

- creșterea numărului de cauze speciale susținute și sumele donate pentru aceste cauze

Nevoi (probleme):

1. Nu există o cultură a donației în societatea românească. Cei care donează sunt relativ puțini și o fac ocazional.
2. Există discrepanță între cauzele susținute de organizații sau personalități influente, care dispun de resurse de promovare.
3. În general, donatorii nu sunt incluși într-un parteneriat efectiv în care strang donații în sensul de a fi primite informații permanent despre evoluția cazurilor.
4. Există riscul unor înșelăciuni, adică cauze inventate publicate prin platformă care ar putea duce la pierderea reputației platformei.

Cerințe:

1. Creșterea încrederii în cauze
2. Creșterea încrederii în cauze propriu zise
3. Creșterea încrederii în concretizarea donațiilor
4. Transformarea utilizatorilor în donatori constanți
5. Atragerea donațiilor prin mecanisme care crează oportunități de donare

SEMINAR 2

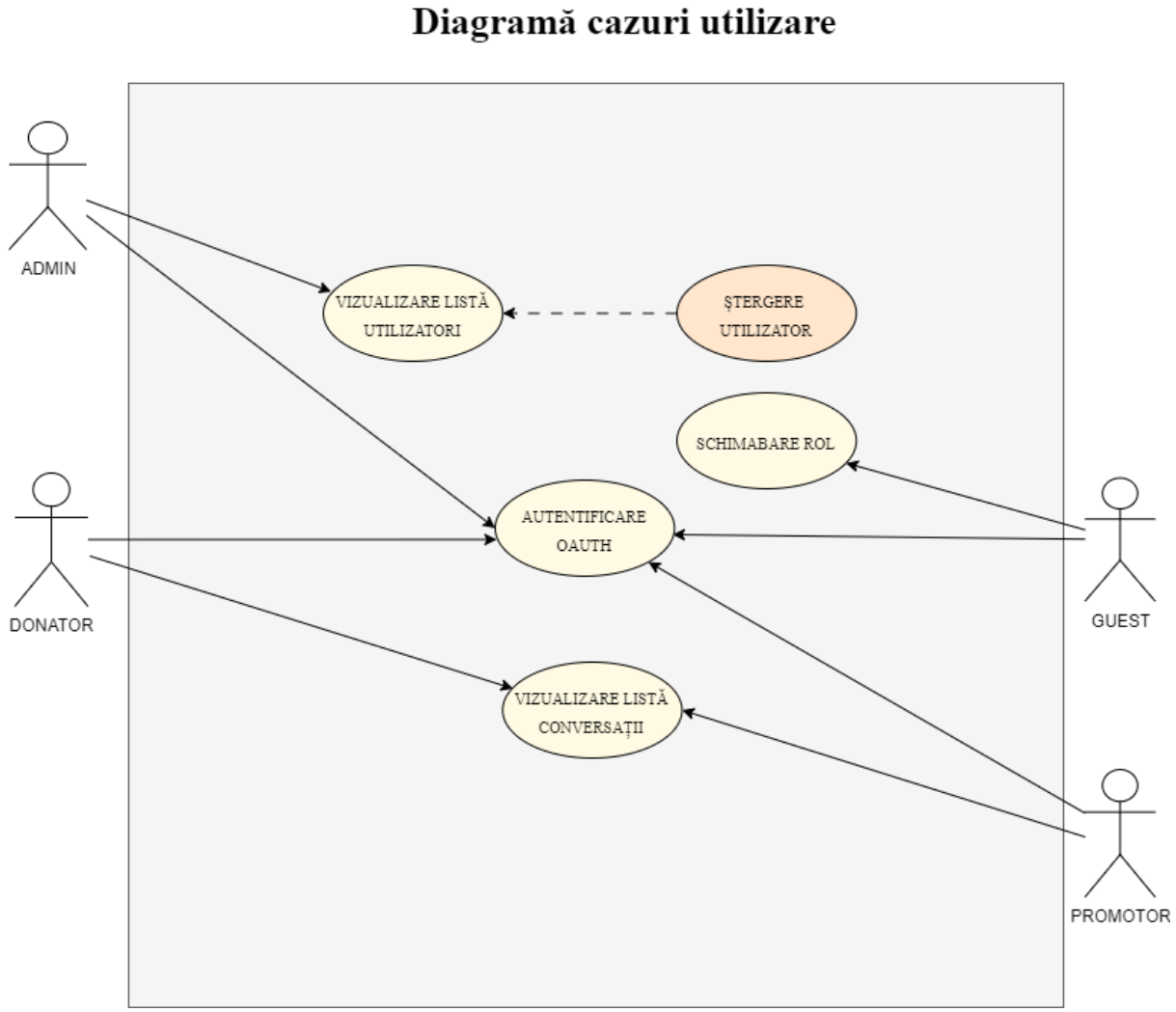
Cazuri de utilizare

Caz de utilizare = o secvență de pași în aplicație pe care **un anumit** tip de utilizator le execută pentru a-și îndeplini un **anumit scop** (trebuie să reflecte un anumit scop al respectivului utilizator în aplicație).

1. Diagrama cazurilor de utilizare (Use case diagram)

- cazuri de utilizare(elementare și/sau codificare)
- rolurile(actorii/tipurile de utilizatori)

- relații - între roluri și cazurile de utilizare
- relații de compunere (mai ales în cazuri de utilizare) sau moștenire (mai ales între roluri)



2. Fișa cazului de utilizare

Codificare	
Denumire	<scopul utilizatorului>
Rol principal	<cel care deține scopul>
Roluri secundare	<dacă există>
Precondiții	<în general presupune finalizarea cu succes a unor alte cazuri de utilizare precedente>
Declanșator	<acțiune care declanșează scopul>

Scenariu de success	1. …… 2. ….. 3. ……	Schemă logică sau diagrama de activități
Post condiție de success	<starea în care se ajunge după rularea scenariilor de succes>	
Scenarii de eșec	<toate variantele de eșec care se produc atunci când nu este completat scenariul de succes>	
Post condiție de eșec	<stările sistemului după fiecare variantă de eșec>	
Extensii	<dezvoltări ulterioare>	

Exemplu:

Codificare	UC1
Denumire	Publicarea unei cauze
Rol principal	Promotor
Roluri secundare	Admin
Precondiții	Utilizatorii sunt autentificați
Declanșator	Promotorul inițiază adăugarea cauză
Scenariu de success	1. Promotorul completează informația despre cauză 2. Promotorul nu publică cauza și iese 3. Promotorul inițiază publicarea cauzei 4. Adminul aprobă publicarea cauzei 5. Promotorul este notificat de publicarea cauzei
Post condiție de success	Cauza este publicată și poate fi accesată de donatori
Scenarii de eșec	2.1 Promotorul renunță la adăugarea cauzei și iese 2.2 Promotorul salvează cauza fără să o publice și iese 3.1 Sistemul identifică erori în completarea cauzei 3.2 Promotorul este atenționat să corecteze erorile 3.3 Promotorul revine la pasul 1 4.1 Adminul nu aprobă publicarea cauzei

	4.2 Promotorul este notificat cu motivul nepublicării
Post condiție de eșec	2.1 Cauza nu a fost adăugată 2.2 Cauza este adăugată, dar nu publicată 4.2 Cauza este adăugată, dar nu este publicată
Extensii	Pasul 1 se poate extinde prin specificarea tipurilor de informații și a caracterului obiectiv sau optimal al acestora Cauza se adaugă într-o listă de cauze

SEMINAR 3

Scenarii de calitate (Quality Scenarios)

User stories = orice fel de cerință, dar sunt sintetice
Cazuri de utilizare = cerințe funcționale, detaliate
Scenarii de calitate = echivalentul cazurilor de utilizare dar pentru cerințe de calitate, detaliate
Format general

Formatul general al unui scenariu de calitate se particularizează cel puțin în anumite aspecte la nivelul unui criteriu de celelalte -> scenariu general de calitate pentru acel criteriu.

Odată ce am stabilit că un anumit criteriu de calitate este important pentru aplicație, trebuie să vedem în ce mod (scenariu) sau moduri se manifestă în raport cu aplicația -> unul sau mai multe scenarii de calitate relevante.

Codificare	
Denumire	
Inițiator	<poate fi un rol/componentă în aplicație>
Declanșator	<acțiunea inițiatorului care declanșează scenariul>
Artifact	<sistemul sau o componentă a sa care este afectată de acțiune>
Starea	<normală sau particulară - artefact>

Răspuns	<tipul și modul de răspuns>
Măsurarea răspunsului	<modalitatea de măsurare + indicatorul numeric urmărit>

Pentru criteriu de calitate putem defini un **scenariu generic** care particularizează formatul general cu elemente specifice respectivului criteriu

Problemă: ONG-urile mici nu știu și nici nu au resurse pentru a-și promova cauzele la fel de bine ca ONG-urile mari.

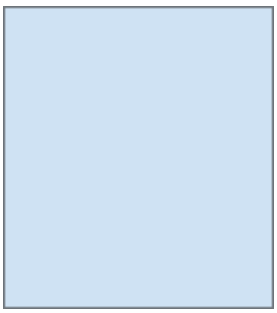
Codificare	QS1
Denumire	Îmbunătățirea promovării cauzelor
Inițiator	Promotorul
Declanșator	Inițierea adăugării unei cauze
Artifact	Interfața
Starea	Nivelul utilizatorilor este scăzut
Strategii de răspuns	Unelte de editare simple, dar de impact
Măsurarea răspunsului	Compararea sumele colectate de un eșantion de promotori cu și fără uneltele oferite - 20%

SEMINAR 4

Proiectare - completăm DA cu mai multe Perspective (Views) arhitecturale asupra aplicației

1. Perspectiva de utilizare
   Proiectarea ecranelor și a tranzițiilor dintre ele cu unelte de tip WIREFRAME

   Ecran 1 Ecran 2



—>

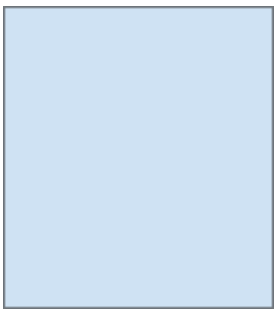
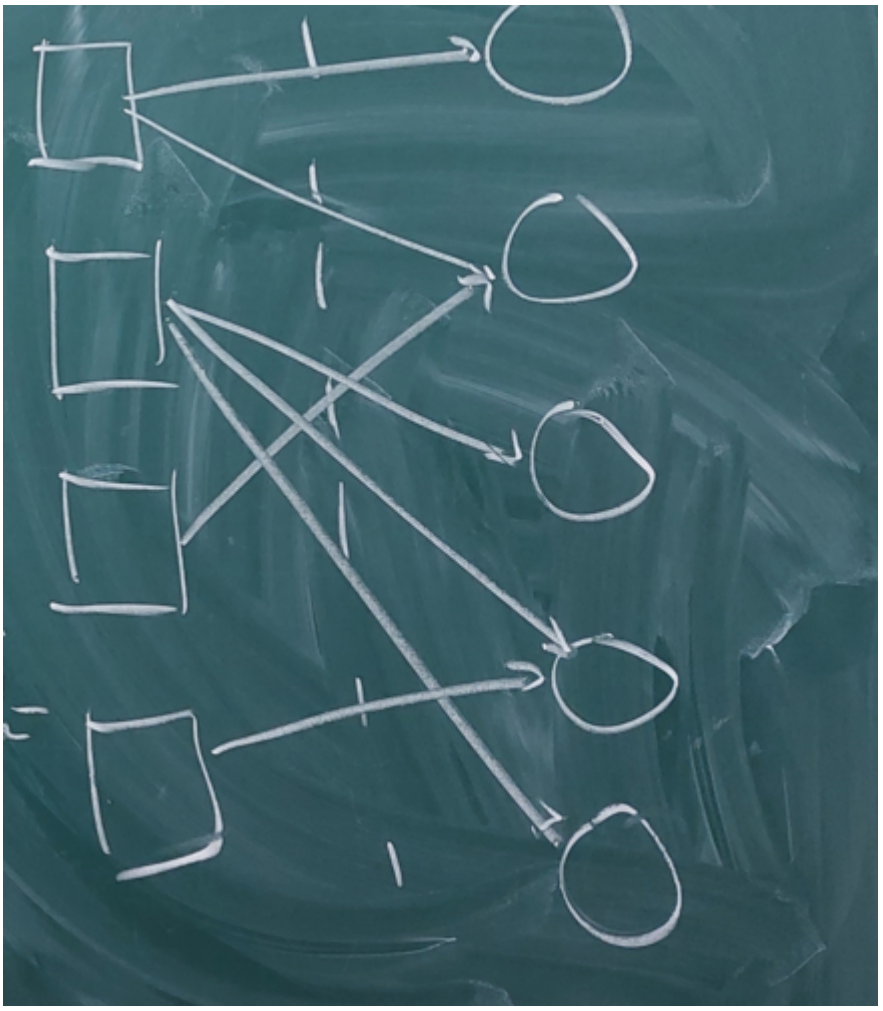


Diagrama de stări - se evidențiază doar acele elemente de proiectare care au relevanță arhitecturală în raport cu specificațiile de analiză arhitecturală.

Cerințele Elemente de proiectare



Sistemul oferă facilități avansate, ușor de utilizat, promotorilor pentru a crește numărul cauzelor finanțate prin platformă cu 20%.

Accentul pe vizual

- imagini - organizare

- grafică
- diagrame

Companii de marketing **predefinite**

- promovare pe site-uri externe
- organizare de concursuri/tombole
- anunțuri/notificări/remindere

Încadrează cauza în categorii relevante

- țintirea mai bună a potențialilor donatori

Rezultă funcționalități care au ca scop îndeplinirea unei cerințe/specificații de calitate.

1. Perspective de utilizare
   - se evidențiază modul de interacțiune a promotorului cu componenta de promovare a cauzei prin campanii de marketing:
     - specificație: use case (extensia use case-ului de adăugare a unei cauze)
     - ecranele și module efectiv de interacțiune care să demonstreze ușurința de utilizare în comparație cu restul formularului de adăugare a unei cauze

Facilitățile de bază pentru promotori (care sunt oferite de celelalte platforme de donații): formula online (text, imagini)

SEMINAR 5

Sistemul oferă facilități avansate, ușor de utilizat, donatorilor pentru a crește numărul donații realizate prin platformă cu 20%.

- Standardul este ceea ce oferă și alte platforme de donații.
- Avansat este peste standard

Brainstorming

Creare oportunități de donare către cauze

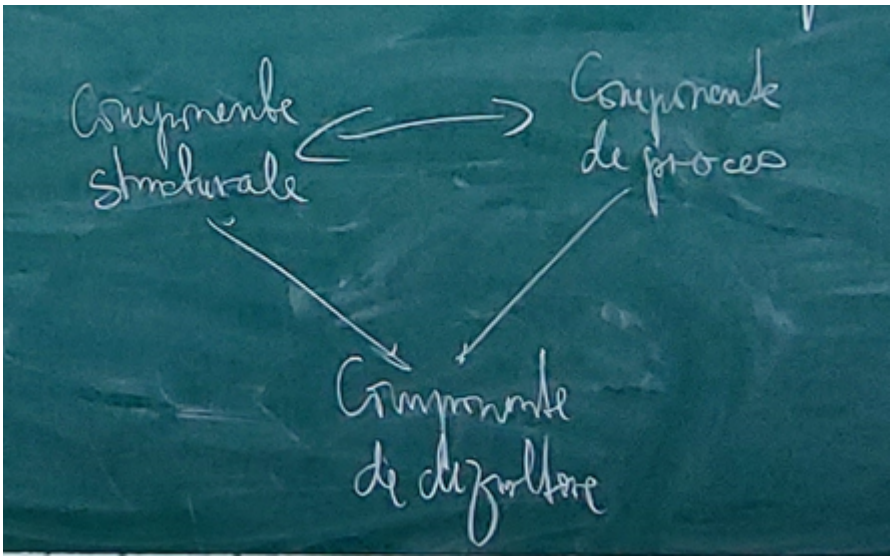
- turnee de gaming cu **taxă** care să meargă către cauză
- replicarea în aplicație a unor evenimente caritabile din offline
- sesiuni de gaming(jocuri simple) în care cel/cei care pierd donează către o cauză aleasă de cel care câștigă

Apar cazuri noi de utilizare/user stories(funcționale, calitative(durata unui joc), scenarii de calitate). În procesul de proiectare putem produce specificații noi, atât funcționale, cât și de calitate care nu mai sunt specificații de analiză ci de proiectare.

Proiectare

1. Perspectiva de utilizare
2. Perspectiva datelor
3. Perspectiva structurală
4. Perspectiva comportamentală
5. Perspectiva fizică
6. Perspectiva testării
7. Perspectiva dezvoltării - planificarea componentelor de dezvoltat

Componente structurale Componente de proces



Componente de dezvoltare