**ROMÂNIA**

**MINISTERUL APĂRĂRII NAŢIONALE**

**ACADEMIA TEHNICĂ MILITARĂ**

**„FERDINAND I”**

**FACULTATEA DE SISTEME ELECTRONICE ŞI INFORMATICE MILITARE**

**Specializarea: Calculatoare și sisteme informatice pentru apărare**

**și securitate națională**



**Implicații ale realocării resurselor asupra fezabilității ordonării activităților**

CONDUCATOR STIINFIC:

**CS. III Dana-Mihaela VÎLCU**

ABSOLVENT:

**Std. Sg. Maj. Mihai-Valentin BADEA**

Conține \_\_\_\_\_\_\_\_\_\_\_ file

Inventariat sub nr. \_\_\_\_\_\_

Poziția din indicator: \_\_\_\_

Termen de păstrare: \_\_\_\_\_

**București**

**-2022-**

Cuprins

[Abstract 4](#_Toc104740959)

[Rezumat 5](#_Toc104740960)

[Listă de figuri 6](#_Toc104740961)

[Listă de abrevieri 7](#_Toc104740962)

[1. Introducere 8](#_Toc104740963)

[1.1. Prezentare generala 8](#_Toc104740964)

[1.2. Importanța temei 8](#_Toc104740965)

[1.3. Utilitatea sistemului 9](#_Toc104740966)

[1.4. Aplicații de project management 9](#_Toc104740967)

[2. Stadiul actual 11](#_Toc104740968)

[2.1. Soluții similare 11](#_Toc104740969)

[2.1.0. Trello 11](#_Toc104740970)

[2.1.1. Asana 12](#_Toc104740971)

[2.2. Articole și cărți ce abordează problema propusă 13](#_Toc104740972)

[2.2.1. An updated survey of variants and extensions of the resource-constrained project scheduling problem 13](#_Toc104740973)

[*2.2.2.* Resource-Constrained Project Scheduling: *Models, Algorithms, Extensions and Applications* 14](#_Toc104740974)

[3. Ordonarea task-urilor în cadrul sistemelor de operare 16](#_Toc104740975)

[3.1. Definiții de bază 16](#_Toc104740976)

[3.2. Ordonarea în sisteme uni-procesor și multi-procesor 17](#_Toc104740977)

[3.2.1. Condiții suficiente pentru seturi de task-uri fără conflicte 22](#_Toc104740978)

[3.2.2. Task-uri periodice 28](#_Toc104740979)

[4. Definirea managementului de proiecte 30](#_Toc104740980)

[5. Structura aplicației 32](#_Toc104740981)

[5.1. Cerințele aplicației 32](#_Toc104740982)

[5.2. Definirea arhitecturii aplicației 33](#_Toc104740983)

[5.2.1. Proiecte Maven 33](#_Toc104740984)

[5.2.2. Modelul MVC 33](#_Toc104740985)

[5.2.3. Securitatea conexiunii Client-Server 35](#_Toc104740986)

[6. Implementarea Software 36](#_Toc104740987)

[6.1. Arhitectura bazei de date 36](#_Toc104740988)

[6.2. Tipurile de utilizatori 37](#_Toc104740989)

[6.4. Algoritmul de ordonare a activităților 40](#_Toc104740990)

[7. Interacțiune utilizator-aplicație 42](#_Toc104740991)

[7.1. Evaluarea euristica pe baza principiilor IOC 42](#_Toc104740992)

[7.1.1. Perceived Affordances 43](#_Toc104740993)

[7.1.2. Maparea 43](#_Toc104740994)

[7.1.3. Cauzalitatea 43](#_Toc104740995)

[7.2. Testarea 43](#_Toc104740996)

[8. Concluzii 46](#_Toc104740997)

[8.1. Probleme întâmpinate 46](#_Toc104740998)

[8.2. Rezultate obtinute 47](#_Toc104740999)

[8.3. Imbunatatiri propuse 47](#_Toc104741000)

[9. Bibliografie 48](#_Toc104741001)

[10. Anexe 49](#_Toc104741002)

[Anexa A 49](#_Toc104741003)

[Anexa B 50](#_Toc104741004)

# Abstract

This paper presents a software solution for the Resource-Constrained Project Scheduling Problem. Despite the various variants of the problem which were researched, I implemented the solution for the basic RCPSP with some updates which allowed me to create an application that can be used in real business environment. The main feature of this application is represented by the automated scheduling of the tasks if there is a feasible scheduler. The schedule is determined by an algorithm which considers the duration, the resource usage and the precedence relations between tasks. This make the difference between my application and other task management applications which allows users only to make a “to do” list without the posibility to offer a variant of scheduling.

# Rezumat

Această lucrare prezintă o soluție software realizată pentru a rezolva problema ordonării activităților din cadrul proiectelor cu constrângeri pe resurse. În procesul de dezvoltare al aplicației s-au studiat atât problema ordonării activităților în cadrul proiectelor cu constrângeri pe resurse, cât și modul în care task-urile sunt ordonate în sistemele de operare. Astfel, în definirea noțiunii de activitate, s-a realizat o paralelă cu noțiunea de task din cadrul teoriei sistemelor de operare. Aplicația implementează un algoritm ce testează fezabilitatea unui set de activități dat, algoritm propus pentru rezolvarea problemei RCPSP standard. Pe lângă acesta, am adus anumite modificări pentru ca aplicația să se muleze cat mai mult într-un mediu real de lucru. Diferența majoră dintre aplicația propusă și alte aplicații de Project management constă în faptul că acestea deseori nu oferă și o variantă de ordonare a activităților, ci oferă doar o vizualizare.

# Listă de figuri

|  |  |  |
| --- | --- | --- |
| **Nr. crt.** | **Figură** | **Pagină** |
| 1. | Fig. 1.0. Jira Roadmap – aplicație de project management | 6 |
| 2. | Fig. 1.1. Jira Board – aplicație de project management | 7 |
| 3. | Fig. 2.0. Trello – interfața grafică | 8 |
| 4. | Fig. 2.1. Asana – afișarea timeline-ului | 9 |
| 5. | Fig. 2.2. Asana – Caracteristicile unei activități | 9 |
| 6. | Fig. 3.0. – Reprezentarea a 3 task-uri in planul cartezian L-C | 14 |
| 7. | Fig. 3.1. – Demonstrarea caracterului optim al algoritmului Earliest Deadline | 15 |
| 8. | Fig. 3.2. – Exemplificarea nemenținerii caracterului optim al algoritmului Earliest Deadline in cadrul sistemelor multi-procesor | 16 |
| 9. | Fig. 3.3. – Reprezentarea unui scenariu cu 3 task-uri în care timpul de procesare nu este cunoscut | 17 |
| 10. | Fig. 3.4. – Scenarii de task-uri în care nu sunt cunoscute valorile de start-time | 19 |
| 11. | Fig. 3.5. – Împărțirea cadranului L-C in 3 subdiviziuni. Ilustrație ce ajută la demonstrarea funcției F(k) | 20 |
| 12. | Fig. 3.6. – Condiții necesare pentru ordonare | 21 |
| 13. | Fig. 3.7. – Cazul F(k, i+1) ≥ F(k, i) | 23 |
| 14. | Fig. 3.8. – Cazul F(k, j+1) ≥ F(k+1, i) | 24 |
| 15. | Fig. 3.9. – Scenariu cu 4 taskuri periodice | 26 |
| 16. | Fig. 4.0. Breșe organizatorice | 32 |
| 17. | Fig. 5.0. – Modelul arhitectural MVC | 34 |
| 18. | *Fig. 5.1. – TLS Exchange* | 35 |
| 19. | Fig. 5.2. – Captura de trafic TLS | 35 |
| 20. | Fig. 6.1. – Arhitectura bazei de date | 36 |
| 21. | Fig. 6.2. – Diagrama cazurilor de utilizare pentru modul administrator | 38 |
| 22. | Fig. 6.3. Diagrama cazurilor de utilizare pentru utilizatori fără rol de administrator | 39 |
| 23. | Fig. 6.4. – Pseudocod al algoritmului ce testează fezabilitatea unui scenariu de task-uri | 40 |
| 24. | Fig. 6.5. – Exemplu de vizualizare a utilizării unei resurse R1 pe un scenariu cu 10 activități | 41 |
|  |  |  |

# Listă de abrevieri

|  |  |  |
| --- | --- | --- |
| **Nr. crt.** | **Abreviere** | **Formă completă** |
| 1. | RCPSP | Resource-Constained Project Scheduling Problem |
| 2. | PMI | Project Management Institute |
| 3. | JDK | Java Development Kit |
| 4. | UI | User Interface |
| 5. | MVC | Model-View-Controller |
| 6. | JSON | JavaScript Object Notation |
| 7. | TLS | Transport Layer Security |
| 8. | IOC | Interactiune Om-Calculator |

# Introducere

Proiectul își propune crearea unei aplicații software desktop de ordonare a activităților destinată organizațiilor. Ordonarea activităților presupune implementarea unor algoritmi astfel încât resursele să fie folosite cât mai eficient.

## Prezentare generala

Proiectul constă într-o aplicație client-server capabilă să realizeze managementul diferitelor organizații. Astfel, in cadrul aplicației, utilizatorii își pot înregistra compania având astfel in mod implicit rolul de administrator al acesteia. Ca și administrator, utilizatorii pot vizualiza proiectele ce se afla in derulare în cadrul organizației, pot crea conturi angajaților si pot defini resursele companiei. Angajații au posibilitatea să vizualizeze activitățile pe care aceștia le au de realizat, să creeze noi proiecte, iar in cazul in care aceștia au dreptul, pot realiza realocări de resurse în proiectele respective.

Rolul principal al aplicației, acela de a stabili dacă există o ordonare fezabilă a activităților, se va concretiza în momentul în care un proiect este creat sau în cazul în care se realizează realocări de resurse.

## Importanța temei

Dezvoltarea oricărei organizații, instituții sau companii stă in modalitatea în care acestea își manageriază activitățile. Fie că vorbim despre proiecte ale multinaționalelor în care sunt angrenați sute sau chiar mii de angajați (de exemplu pentru realizarea jocului Grand Theft Auto V au lucrat peste 1000 de programatori[[1]](#footnote-1)) sau despre startup-uri ce au un număr restrâns de angajați, organizarea activității fiecărui membru în parte este extrem de importantă pentru o productivitate crescută. Într-o economie de piață extrem de competitivă cum este cea din prezent, o aplicație de management al activităților poate face diferența dintre o companie ce se dezvoltă si una ce se duce încet spre faliment.

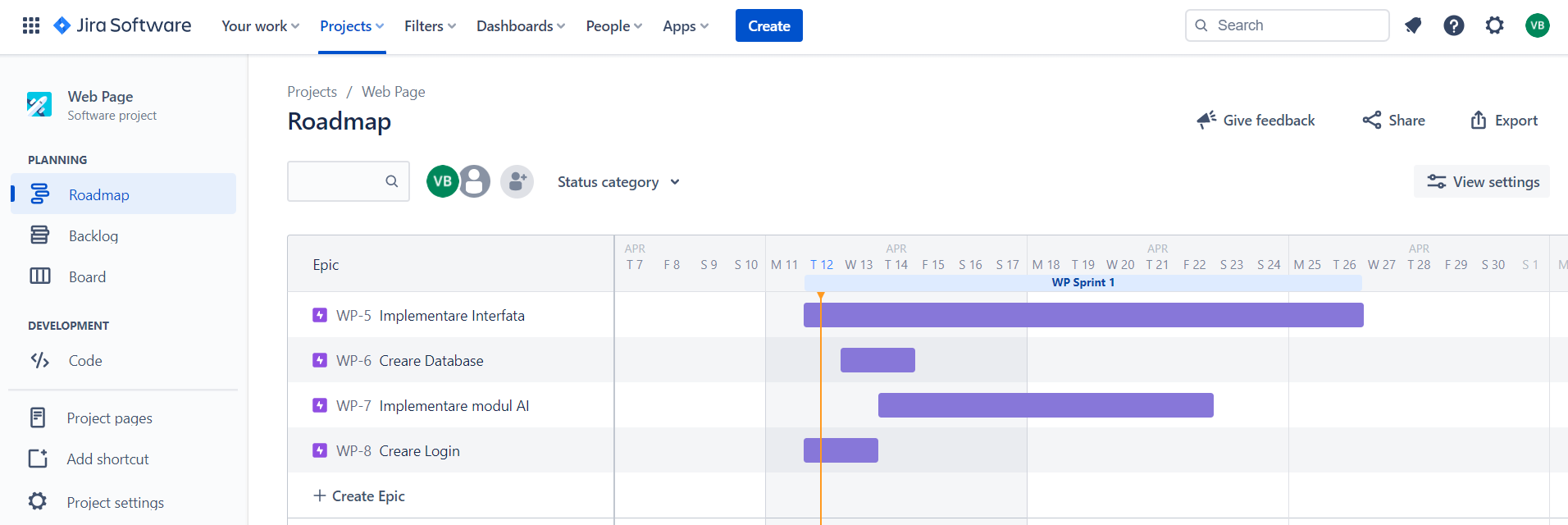
## Utilitatea sistemului

Soluția propusă în această lucrare este utilă pentru organizațiile ce își doresc să li se ofere o ordonare automată a activităților pe care acestea le au de îndeplinit, în funcție de resursele de care dispun. De asemenea, prin intermediul interfeței grafice, se poate monitoriza utilizarea resurselor si modalitatea în care sunt organizate activitățile fiecărui angajat.

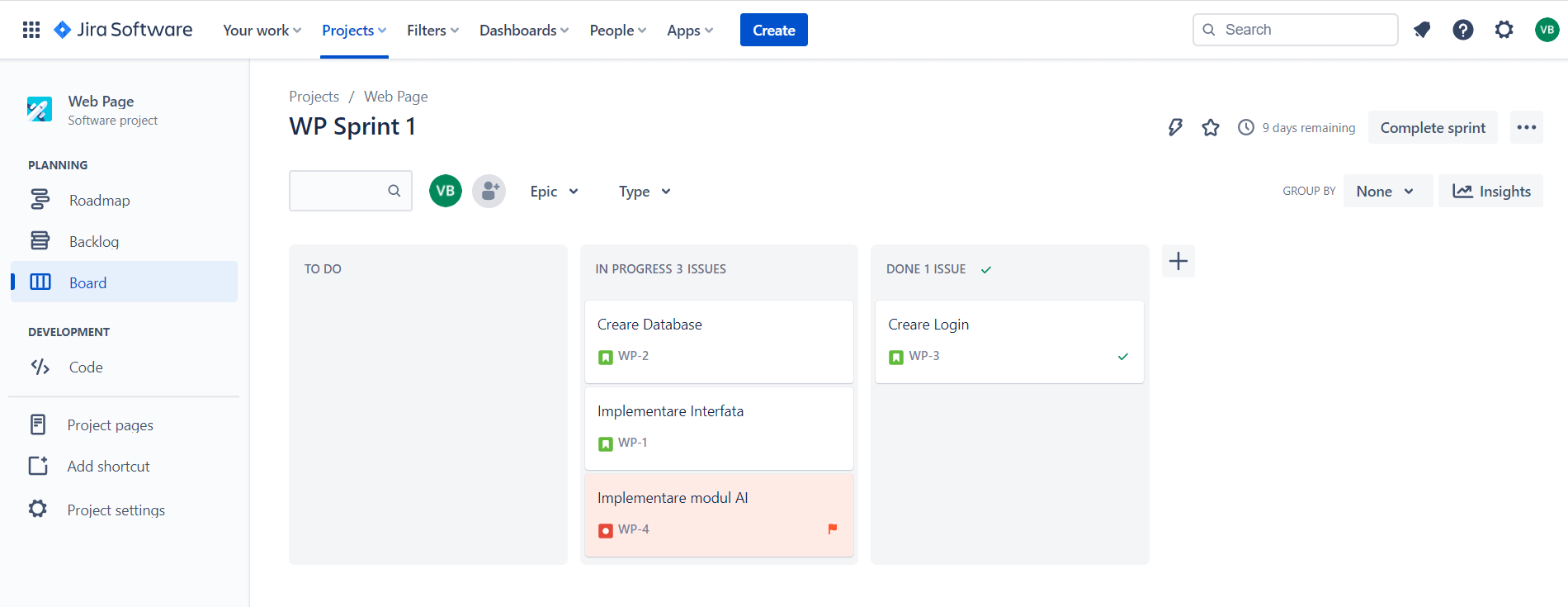
Fiecare angajat ce se autentifica în aplicație este capabil să vizualizeze activitățile pe care acesta le are de realizat in următoarea perioadă. În cazul în care se impune, acesta poate realiza realocări ale resurselor in cadrul proiectelor în care este deja asignat, lucru ce va duce în mod implicit la apelarea algoritmului de ordonare ce va determina dacă există o nouă soluție fezabilă pentru datele actuale.

## Aplicații de project management

În clipa de față există numeroase aplicații de project management ce pot fi folosite pentru gestionarea activităților, însă principala problemă pe care am întâlnit-o in analizarea acestora a fost reprezentata de faptul ca aceste aplicații nu realizează in mod automat o ordonare a activităților pe care utilizatorul le definește. Astfel, rolul acestora este mai degrabă acela de a fi un „to do list”. Desigur, o mare majoritate a acestora oferă alte facilitați precum accesul prin intermediul interfeței la Git-ul proiectului, făcând astfel mult mai ușoară actualizarea acestuia, însă pe partea de ordonare automata a activităților acestea sunt deficitare.



*Fig. 1.0. Jira Roadmap – aplicație de project management*



*Fig. 1.1. Jira Board – aplicație de project management*

Un exemplu de aplicație folosită foarte frecvent in industria IT este Jira. Aplicația are o interfață ce permite o foarte ușoară creare si vizualizare a activităților din cadrul proiectelor.

In cadrul tab-ului „Board” se pot vizualiza activitățile împărțite pe 3 categorii:

* „to do” – activități ce urmează a fi realizate
* „in progress” – activități curente ce se află in desfășurare
* „done” – activități ce au fost finalizate

În cadrul tab-ului „Roadmap” se pot vizualiza activitățile din punct de vedere temporal. Consider ca acest mod de vizualizare este mult mai intuitiv pentru utilizatori, dar în continuare problema ordinii în care se vor executa aceste activități este nedeterminată și rămâne astfel la latitudinea celui ce le execută.

# Stadiul actual

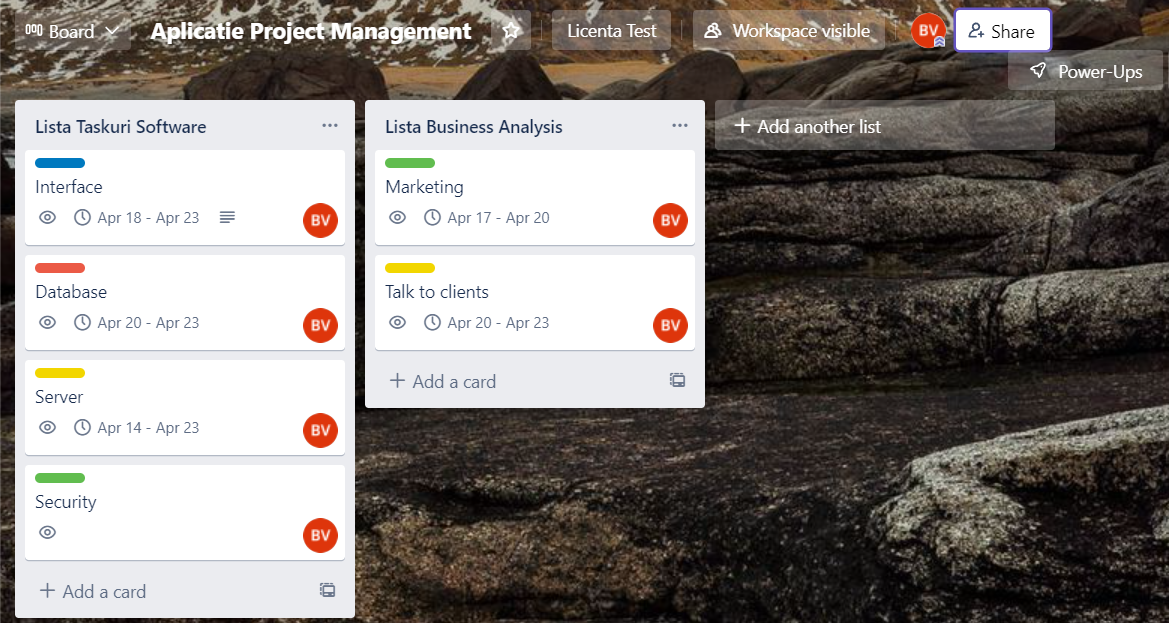
## Soluții similare

În clipa de față există numeroase soluții software ce pot fi folosite pentru managementul proiectelor. Printre acestea amintim: Jira, Trello si Asana.

Pe lângă soluțiile gratuite accesibile pe Internet, sau demo-urile celor ce necesită un abonament plătit, soluția prezentată în prezenta lucrare implementează și o sugestie de ordonare a activităților, lucru ce poate fi util organizațiilor în realizarea obiectului lor de activitate. În continuare, voi prezenta câteva soluții software similare ce pot fi găsite pe Internet, prezentând totodată modul acestora de utilizare.

### 2.1.0. Trello

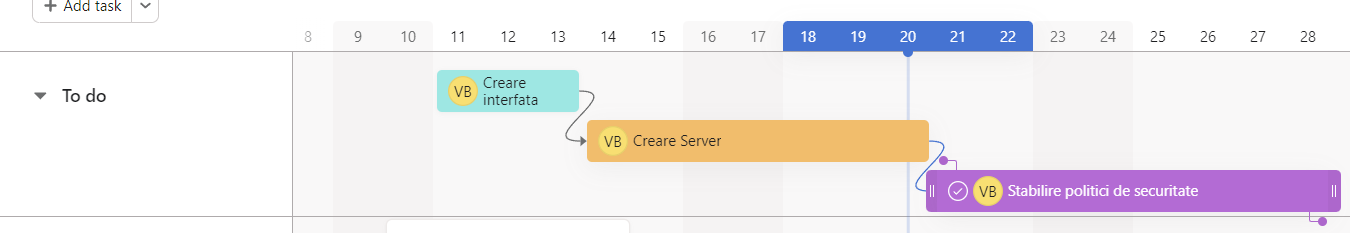
Trello reprezintă o aplicație web în care utilizatorii își pot organiza activitățile pe care le au de realizat în cadrul organizațiilor din care fac parte. Aplicația permite crearea de proiecte în care activitățile sunt împărțite pe liste de lucru. Ca și parametrii, fiecărei activități îi sunt atribuite o descriere, o perioadă în care aceasta trebuie realizată și un membru al echipei ce trebuie să o execute. De asemenea, se mai pot atașa link-uri sau fișiere necesare completării activității. În varianta demo gratuită se pot afișa activitățile doar sub forma unui Board. Pentru varianta plătită a aplicației există beneficii suplimentare precum: conectarea la Google Drive, Dropbox si altele; posibilitatea de vizualizare a activităților sub forma unui calendar, lucru ce face mult mai ușor de urmărit evoluția acestora.



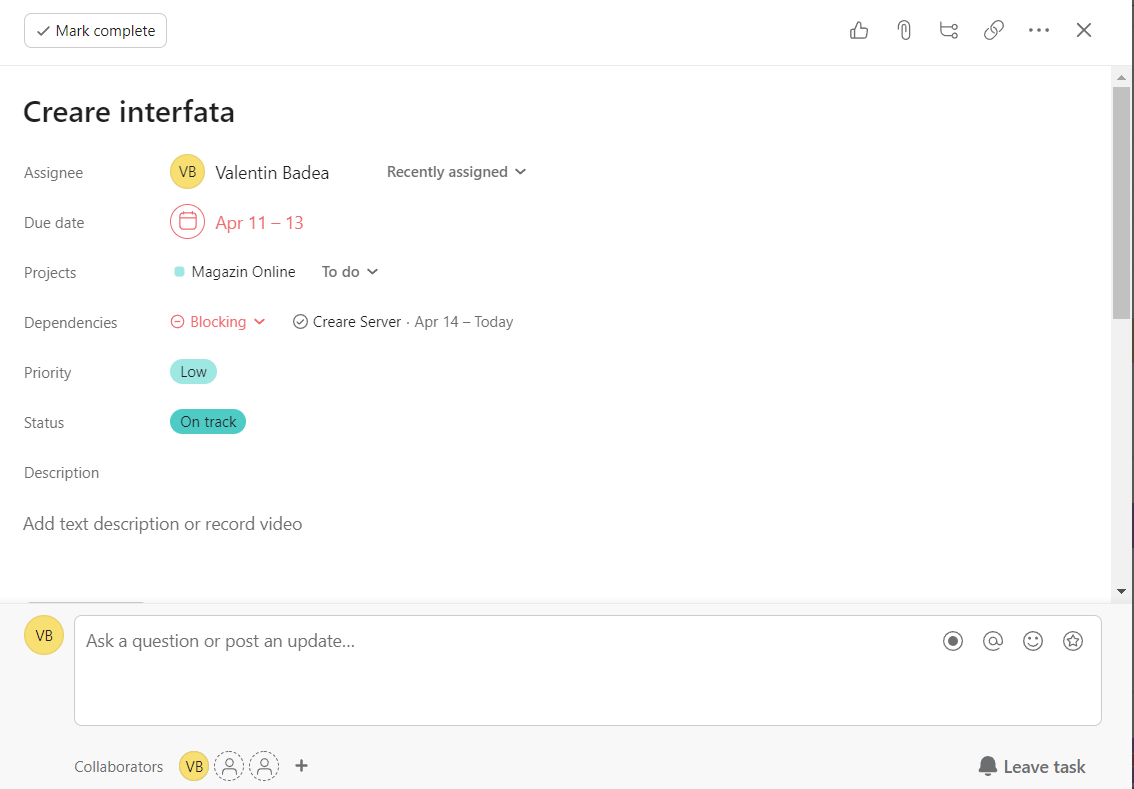
*Fig. 2.0. Trello – interfața grafică*

### Asana

O aplicație ce oferă mai multe module pentru utilizatori este Asana. Aceasta oferă în mod implicit 3 moduri de vizualizare a activităților: Calendar, Board si List. În ceea ce privește parametrii pe care o activitate îi are, aceștia sunt : utilizatorul ce trebuie sa îndeplinească task-ul, perioada de desfășurare a activității si prioritatea. În cadrul aplicației sunt definite 3 niveluri de prioritate: Low, Medium si High. Pe lângă acestea, o caracteristică foarte interesantă este reprezentată de posibilitatea de a defini relații de precedență, astfel putând arăta dacă execuția unei activități blochează sau este blocată de o alta. Săgețile ce se observă în imaginea următoare reprezintă relațiile de precedență definite de utilizator. În cazul în care se încearcă schimbarea perioadei de execuție a unei activități care nu ar respecta relațiile definite, se va realiza in mod automat o schimbare în perioada de execuție și pentru activitatea ce nu ar mai îndeplini condițiile impuse.



*Fig. 2.1. Asana – afișarea timeline-ului*



*Fig. 2.2. Asana – Caracteristicile unei activități*

## Articole și cărți ce abordează problema propusă

În secțiunea următoare voi prezenta baza științifică ce motivează alegerea temei curente și importanța pe care o are in cadrul desfășurării activităților de project management.

### An updated survey of variants and extensions of the resource-constrained project scheduling problem[[2]](#footnote-2)

Articolul precizat definește problema ordonării activităților având constrângeri de resurse, având scopul de a găsi o ordonare a activităților fără să fie încălcate constrângerile date de relațiile de precedență și de numărul de resurse disponibil, astfel încât timpul de îndeplinire al tuturor activităților să fie minim. Se specifică faptul că această problemă a devenit una standard in domeniul organizării proiectelor, atrăgând un număr mare de cercetători ce au venit cu diverse proceduri de rezolvare.

În cadrul articolului sunt prezentate pe scurt diferite variații ale unor soluții implementate de către cercetători in domeniu.

În Capitolul 2 se specifică faptul că problema de bază a ordonării activităților cu constrângeri pe resurse necesită procesarea task-urilor in mod non-preemptiv, ceea ce înseamnă ca odată începută o activitate, aceasta își continua execuția până în punctul în care este finalizată. Articolul precizează faptul că în lucrarea *„Handbook on Project Management and Scheduling Vol.1 ”*[[3]](#footnote-3) autorii oferă o soluție în care activitățile pot fi întrerupte fără nicio restricție. Tot in cadrul aceluiași capitol, este abordată problema numărului de resurse solicitat de către fiecare activitate în parte. Se specifica faptul că în cadrul problemei standard, orice activitate *Ai* care necesită o cantitate de *rik* din resursa *k*, nu își va schimba aceasta valoare pe parcursul procesării acesteia. Se oferă o referință către lucrarea redactata de *Hartmann(2013)*[[4]](#footnote-4), în care autorul abordează și situația în care activitatea *Ai* necesită o cantitate de *rikt* unități din resursa *k* la perioada *t* din timpul său total de procesare.

### Resource-Constrained Project Scheduling: *Models, Algorithms, Extensions and Applications[[5]](#footnote-5)*

În capitolul 1 al cărții este definită problema ordonării activităților cu constrângeri pe resurse ca fiind o problemă de optimizare combinatorială. O problemă de optimizare combinatorială este definită de o soluție discretă în spațiul X sau care poate fi redusa la un set discret, si printr-un subset de soluții fezabile asociat unei functii obiectiv . Astfel, o problemă de optimizare combinatorială presupune găsirea unei soluții fezabile astfel încât este minimizată sau maximizată.

Având la bază informația teoretică despre probleme de optimizare combinatorială, autorul definește problema curenta de ordonare a activităților cu constrângeri asupra resurselor ca fiind dată de următorul tuplu :

Un set este folosit pentru a identifica activitățile ce constituie baza proiectului. Prin convenție, activitatea *A0* indică începerea procesului de ordonare, iar activitatea *An+1* indică, în mod simetric, finalizarea acestuia. Mulțimea rămasă prin eliminarea celor doua activități, , reprezintă activitățile propriu-zise ce trebuie sa fie executate.

Duratele de execuție a activităților sunt reprezentate printr-un vector *p* in , unde *pi* reprezintă durata activității *Ai*. O caracteristică ce trebuie menționată este legată de valorile speciale folosite pentru activitățile *A0* si *An+1* , acestea având duratele asociate *p0* = *pn+1* = 0.

Relațiile de precedență sunt definite de o colecție E având perechi de forma (*Ai* , *Aj*) care denotă faptul că activitatea *Ai* va preceda activitatea *Aj* . Prin definirea acestor relații de precedență se poate crea un graf G(V,E) în care fiecărui nod îi este asociat o activitate. Pentru ca relațiile de precedență să aibă sens, în cadrul grafului nu vor exista cicluri. Deoarece precedența reprezintă o relație binară tranzitivă, prezența unei rute în cadrul grafului G de la nodul *Ai* către nodul *Aj* denotă faptul că *Ai* va începe prima. Având in vedere cele doua activități ce reprezintă începutul, respectiv finalizarea ordonării, rezultă faptul că activitatea *A0* va preceda toate celelalte activități, iar activitatea *An+1* va fi succesorul acestora.

Resursele sunt organizate sub forma unei mulțimi .

Disponibilitatea resurselor este reprezentată de un vector *B* in , unde *Bk* reprezintă numărul disponibil din resursa *Rk* .

Cererile de resurse sunt reținute într-o matrice de dimensiuni notată cu *b*. Astfel, *bik* reprezintă cantitatea din resursa *Rk* folosită în fiecare perioadă de timp pentru executarea activității *Ai* .

O ordonare reprezintă un punct *S* in unde *Si* reprezintă punctul de început al activității *Ai* . *Ci* reprezintă timpul de completare al activității *Ci* , unde . *S0* reprezintă un punct de referință pentru începutul proiectului, astfel, având in vedere mențiunile anterioare, *S0* = 0. Pentru ca o ordonare să fie considerată fezabilă, aceasta trebuie să respecte simultan atât relațiile de precedență (1.0), cât și constrângerile rezultate din numărul limitat de resurse disponibile (1.1). Mulțimea *Åt* reprezintă totalitatea activităților ce se află în execuție la momentul de timp *t* : .

(1.0)

(1.1)

Timpul de finalizare al ordonării este egal cu timpul de început al activității *Sn+1*. Autorul folosește termenul de *„makespan”* pentru a reprezenta timpul necesar finalizării tuturor activităților. Astfel, având in vedere toate cele prezentate mai sus, autorul oferă următoarea definiție:

DEFINITIA 1.0. - RCPSP[[6]](#footnote-6) reprezintă problema găsirii unei ordonări non-preemptive S având makespan-ul minim Sn+1 care respectă constrângerile date de relațiile de precedenta (1.0) si de resurse (1.1).

# Ordonarea task-urilor în cadrul sistemelor de operare

În capitolul acesta vor fi prezentate noțiuni teoretice ce stau la baza ordonării task-urilor în cadrul sistemelor de operare, prezentându-se astfel principiile și algoritmii folosiți. Următoarele informații teoretice au fost selectate din cartea *„Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks”[[7]](#footnote-7)*.

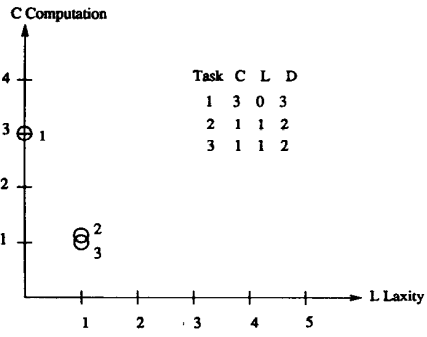
## Definiții de bază

În general, un task poate fi caracterizat prin următorii parametri întregi: S (momentul la care task-ul își începe execuția), C (numărul de unități CPU) și D (deadline-ul). De asemenea, mai pot fi adăugate și constrângeri adiționale precum relații de precedență sau periodicitate. Starea fiecărui task care și-a început execuția poate fi caracterizat prin doi parametrii: timpul de procesare rămas *C(i)* la momentul *t* = i și deadline-ul *D(i)* când acesta își termină execuția.

Laxitatea unui task la momentul *t* = i este definită de relația:

(2.0)

Această caracteristică a taskului definește prioritatea cu care trebuie executat task-ul respectiv . În mod evident, un task cu laxitate zero trebuie executat imediat și fără întreruperi. Laxitatea negativă a unui task arată faptul că deadline-ul a fost depășit. Problema de organizare a taskurilor poate fi modelată prin configurarea de *„token-uri”* în primul cadran al unui plan cartezian ce are pe axa verticală *C*, iar pe cea orizontală *L*. Astfel, un task *j* având parametrii *Cj(i)* și *Lj(i)* la momentul *t* = i, va fi poziționat la *L = Lj(i)* și *C = Cj(i)* în planul *L-C*. Un astfel de exemplu de reprezentare a unui scenariu în care avem un număr de 3 task-uri este prezentat in *Fig. 3.0*. De reținut faptul că o poziție din acest plan poate fi ocupată de mai mult de 1 task în același moment de timp, în cazul de față observându-se faptul că task-urile 2 si 3 ocupă aceeași poziție. De asemenea, în reprezentarea planului nu este trecut numărul de unități CPU ce pot fi folosite, dar acesta va fi folosit în momentul în care se realizează execuția unui task.



*Fig. 3.0. – Reprezentarea a 3 task-uri in planul cartezian L-C*

În cazul în care avem la dispoziție un număr de n procesoare și m task-uri (*n* < *m*) ce se pot executa la momentul *t* = i, atunci se vor putea executa un număr de *n* din totalul de *m* task-uri posibile. Reprezentând acest lucru tot în planul *L-C* menționat anterior, execuția task-urilor se traduce prin deplasarea în jos cu o unitate pe axa paralelă cu *C* și, simultan, spre stânga tot cu o unitate pe axa paralelă cu *L*. Astfel, ecuațiile ce prezintă execuția unui task sunt următoarele:

(3.1)

(3.2)

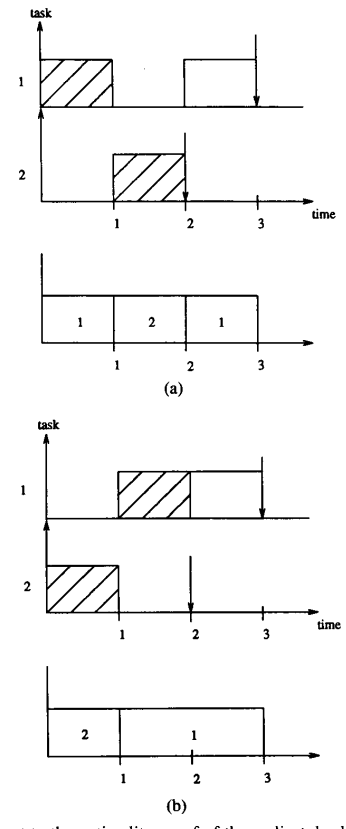
Pentru un task ce nu este executat, ecuațiile sunt următoarele : si . Algoritmul de ordonare este cel ce impune ce set de task-uri se va executa la un moment de timp. În cazul în care un task ajunge în al doilea cadran al planului, atunci acesta și-a ratat deadline-ul și, drept urmare, ordonarea nu este una fezabilă. Contrar acestui caz, dacă un task ajunge la valoarea 0 pe axa C, iar valoarea laxității este pozitivă, atunci task-ul a fost executat cu succes înainte de deadline.

## Ordonarea în sisteme uni-procesor și multi-procesor

În cazul sistemelor cu un singur procesor există algoritmi de ordonare ce pot fi considerați optimi, contrar sistemelor multi-procesor unde studiile au arătat că problema găsirii unui algoritm optim nu este rezolvabilă.

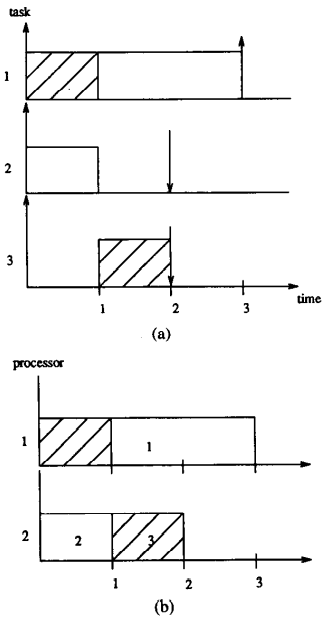
În continuare, voi prezenta 2 algoritmi ce sunt folosiți pentru ordonarea task-urilor: *Earliest Deadline* si *Least Laxity.* **Earliest Deadline** presupune execuția task-ului cu cel mai apropiat deadline, iar **Least Laxity** presupune execuția task-ului ce are cea mai mică valoarea a laxității. În ambele cazuri, dacă există task-uri situate pe aceeași poziție, se va alege unul in mod arbitrar.

Dertouzos[[8]](#footnote-8) a demonstrat caracterul optim al algoritmului Earliest Deadline prin faptul că întotdeauna este posibil ca o ordonare fezabilă să fie transformată într-o ordonare ce implementează algoritmul Earliest Deadline. O astfel de situație este prezentată in *Fig. 3.1.* unde avem 2 task-uri: primul are deadline-ul la momentul *t*=3 și are nevoie de 2 unități de procesare, iar al doilea are deadline-ul la momentul *t*=2 si are nevoie de o singură unitate de procesare. În cazul *a)* se prezintă o ordonare fezabilă în care prima dată este executat task-ul cu deadline-ul mai mare, în locul task-ului cu deadline-ul mai mic. Având in vedere cele menționate mai sus, dacă înlocuim varianta de la punctul *a)* cu implementarea algoritmului Earliest Deadline, vom obține de asemenea o soluție fezabilă, așa cum se poate observa în figură la subpunctul *b)*.



*Fig. 3.1. – Demonstrarea caracterului optim al algoritmului Earliest Deadline*

Din nefericire, caracterul optim al algoritmului nu se menține și în cazul sistemelor multi-procesor. Pentru a demonstra acest fapt, vom considera problema de ordonare reprezentată în *Fig. 3.2.a)* în care avem 3 task-uri ce trebuie executate si un număr n=2 procesoare. Task-urile 2 si 3 au ambele deadline-ul peste 2 unități de timp din momentul în care se începe ordonarea, iar task-ul numărul 1 are deadline-ul peste 3 unități de timp. Pentru a își finaliza execuția cu succes, task-ul 1 are nevoie de 3 unități de procesare, iar celelalte 2 au nevoie de 1 unitate de procesare. Algoritmul Earliest Deadline ar presupune prima dată executarea task-urilor 2 si 3, lucru ce ar rezulta în mod imediat trecerea task-ului 1 în al doilea cadran al planului *L-C* prezentat la începutul capitolului. Astfel, nu există o ordonare fezabilă ce se poate realiza pentru scenariul dat în cazul în care se dorește folosirea algoritmului Earliest Deadline.



*Fig. 3.2. – Exemplificarea nemenținerii caracterului optim al algoritmului Earliest Deadline in cadrul sistemelor multi-procesor*

Totuși, scenariul prezentat în figura de mai sus poate avea o ordonare fezabilă în cazul în care este folosit algoritmul Least Laxity. Implementarea acestuia este reprezentată prin diagrama din *Fig. 3.2.b).*

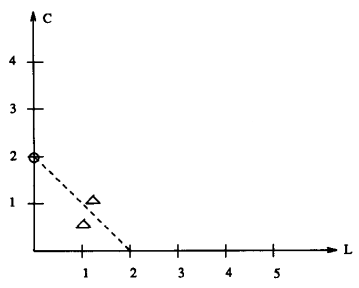
Am demonstrat mai sus caracterul optim al algoritmului Earliest Deadline în cazul sistemelor uni-procesor și modul în care acesta realizează ordonarea task-urilor. O caracteristică ce face ca prezentul algoritm să fie considerat „best effort” este reprezentată de atributele task-urilor pe care acesta le necesită: algoritmul va realiza ordonarea acestora doar în funcție de deadline-ul precizat, necontând timpul de procesare și nici măcar start-time-ul. Astfel, ar fi foarte util dacă am putea găsi un algoritm ce are această proprietate pentru sistemele multi-procesor. Din păcate, un astfel de algoritm **nu exista**.

În mod particular, în cazul în care nu cunoaștem *a priori* niciunul din următorii parametri: 1) deadline-uri, 2) timpii de procesare sau 3) start-time-uri, atunci pentru orice algoritm propus există un scenariu de task-uri pentru care nu s-ar putea găsi o soluție fezabilă.

Considerăm următoarea lema:

Lema 1: *„*Nu poate exista un algoritm de ordonare optim dacă timpii de procesare ai task-urilor nu sunt cunoscuți *a priori”.*

Pentru a realiza demonstrația lemei, vom considera un scenariu în care avem 3 task-uri si 2 procesoare. Informațiile ce se cunosc despre task-uri sunt următoarele: toate au deadline-ul peste 2 unități de timp, 2 dintre ele necesită o singură unitate de timp de procesare, iar ultimul va necesita 2 unități de timp de procesare. În *Fig. 3.3.* sunt reprezentate prin triunghiuri cele 2 task-uri cu același timp de procesare necesar, iar ultimul task este reprezentat printr-un cerc. În mod evident, pentru a obține o soluție fezabilă de ordonare, task-ul ce are valoarea laxității egală cu 0 trebuie să fie executat imediat. În cazul în care algoritmul de ordonare nu cunoaște nicio informație legată de timpul de procesare al task-urilor, acestea vor părea identice. Acestea fiind spuse, algoritmul de ordonare ar putea executa task-ul reprezentat printr-un triunghi, lucru ce ar presupune eșuarea găsirii unei ordonări fezabile.



*Fig. 3.3. – Reprezentarea unui scenariu cu 3 task-uri în care timpul de procesare nu este cunoscut*

Lema 2: *„*Nu poate exista un algoritm de ordonare optimdacă deadline-urile task-urilor nu sunt cunoscute *a priori”.*

Demonstrația acestei leme se realizează în mod identic cu cea prezentată mai sus.

Lema 3: *„*Nu poate exista un algoritm de ordonare optim dacă distribuția cererilor în timp (start time) nu este cunoscută *a priori”.*

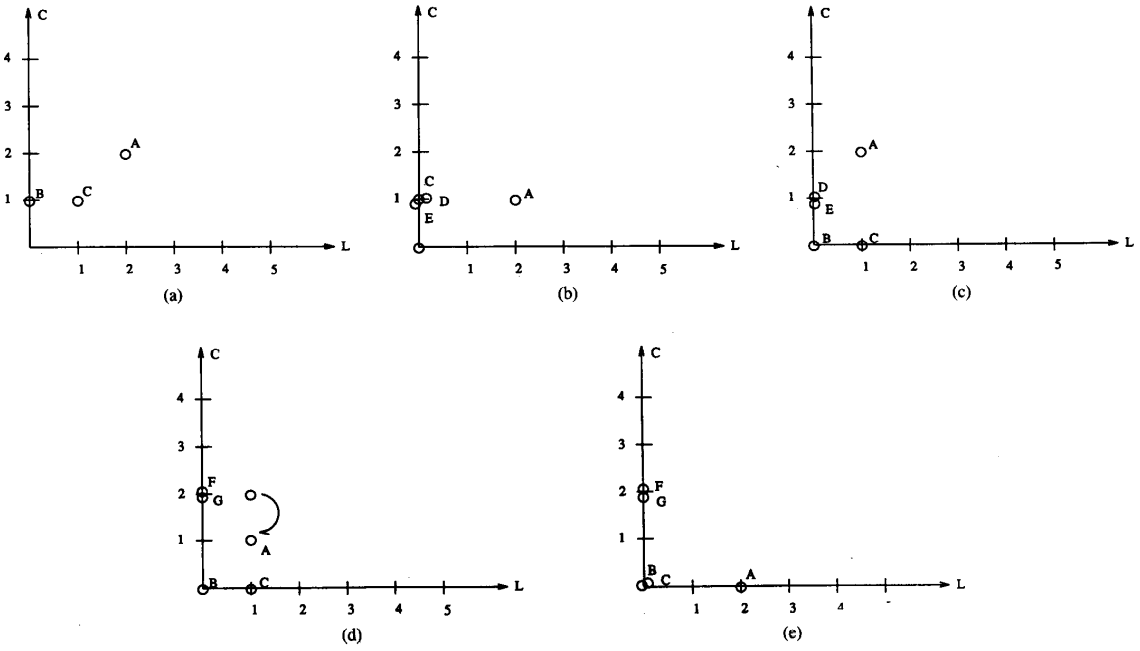
Pentru a realiza demonstrația acestei leme, vom considera un scenariu în care avem inițial 3 task-uri, reprezentate în *Fig. 3.4.a)*, și un număr de 2 procesoare disponibile. Deoarece se observă faptul că task-ul B are laxitatea 0, acesta va trebui executat imediat. În funcție de decizia pe care o ia algoritmul de ordonare, vor exista 3 cazuri posibile.

*Cazul 1:* La momentul *t* = 0 sunt mutate în jos pe axa verticală cu o unitate task-urile A si B. Considerăm că la momentul *t* = 1 mai apar 2 task-uri: C si D, ce au laxitatea egală cu 0, după cum este ilustrat în *Fig. 3.4.b)*. Acest lucru este permis deoarece, după cum precizează lema, nu cunoaștem momentele de timp la care vor fi începute task-urile. În mod evident, având 3 task-uri ce trebuie executate imediat și doar 2 procesoare disponibile, nu există o ordonare fezabilă pentru scenariul propus. Totuși, dacă algoritmul de ordonare ar fi executat prima dată task-urile B si C, scenariu ilustrat în *Fig. 3.4.c)*, atunci am fi putut avea o ordonare fezabilă.

*Cazul 2:* La momentul *t* = 0 sunt mutate în jos pe axa verticală cu o unitate task-urile B si C, astfel task-ul A se va deplasa cu o unitate spre stânga pe axa orizontală. La momentul *t* = 2, sunt introduse 2 noi task-uri: F si G, scenariul ilustrat în *Fig. 3.4.d)*. Având în vedere faptul că cele doua noi task-uri apărute vor avea o prioritate mai ridicată (având laxitatea 0), acestea ar trebui executate în următoarele 2 unități de timp. Problema ce apare este similară cu cea de la primul caz în care avem 3 task-uri ce trebuie executate imediat, dar doar 2 unități de procesare disponibile. Astfel, pentru acest caz nu există o ordonare fezabilă. Totuși, dacă la momentul de timp *t* = 0, ar fi fost executate task-uri A si B, scenariul propus ar fi avut o soluție fezabilă deoarece task-uri C si A și-ar fi încheiat execuția la momentul *t* =1. Acest caz este ilustrat în *Fig.3.4.e)*.

*Cazul 3:* Task-ul B este singurul ce va fi executat la momentul *t* = 0. Astfel, pentru oricare set de task-uri din cele de mai sus ce ar fi introduse ulterior, nu ar exista o ordonare fezabilă.

Prin urmare, pentru orice decizie pe care un algoritm de ordonare ar face-o la momentul inițial, ar exista un set de task-uri (D & E sau F & G) ce ar duce la un blocaj.

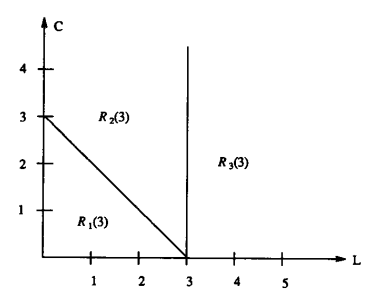


*Fig. 3.4. – Scenarii de task-uri în care nu sunt cunoscute valorile de start-time.*

Pentru a face anumite observații utile asupra ultimei leme menționate, ne vom reîntoarce la scenariul prezentat în *Fig. 3.4.a)*. Putem afirma că există doua alegeri simple pe care algoritmul de ordonare ar putea sa le facă: fie execută task-uri ce necesită un timp mai mic de procesare (B si C), asumându-și astfel riscul ca la momentul *t* = 1 un procesor să nu execute nimic (asemănător cu cazul prezentat în *Fig. 3.4.d)*), fie executa task-ul ce are un timp de procesare mai lung (A) și riscă apariția unor task-uri urgente ce pot apărea, cum se poate observa si în *Fig. 3.4.b)*. Primul caz corespunde unei strategii în care se dorește minimizarea numărului de task-uri ce nu pot fi finalizate, iar al doilea caz corespunde unei strategii în care se dorește minimizarea timpului de procesare per total.

### Condiții suficiente pentru seturi de task-uri fără conflicte

În acest subcapitol, vom continua să păstram notațiile folosite până în acest moment. Astfel, al *j*-lea task va fi notat cu *Jj*, iar parametrii acestuia vor fi, de asemenea, notați cu: *Cj*, *Dj* si *Lj*. Pentru o mai ușoară urmărire a task-urilor, planul *L-C* va fi împărțit în 3 diviziuni după cum se poate observa in *Fig. 3.5.*. Pentru orice valoare întreagă *k*, toate task-urile pot fi împărțite în trei seturi distincte, după cum urmează:



*Fig. 3.5. – Împărțirea cadranului L-C in 3 subdiviziuni. Ilustrație ce ajută la demonstrarea funcției F(k)*

În continuare, vom defini o funcție ce reprezintă surplusul puterii de calcul a sistemului pentru următoarele *k* unități de timp:

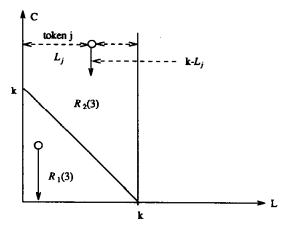
(3.3)

O mențiune foarte importantă este reprezentată de faptul că această ecuație definește o funcție de timp, astfel, de fiecare dată când va fi menționată, se va nota cu *F(k, i)* ceea ce va reprezenta valoarea funcției *F(k)* la momentul de timp *t=i.*

Ne propunem ca prima dată să demonstrăm o condiție necesară pentru a putea ordona un set de task-uri ce au același start-time. Dând o astfel de condiție necesară, va trebui mai apoi să demonstrăm existența unei ordonări fezabile în care valorile de deadline nu sunt depășite. Vor fi alese pentru execuție doar task-urile care și-au depășit start-time-ul, fără a se ține cont de start-time-urile task-urilor viitoare. Cu alte cuvinte, un set de task-uri nu va avea conflicte dacă deadline-urile acestora pot fi atinse atunci când valorile de start-time sunt cunoscute.

Lema 4: *„*O condiție necesară pentru ordonarea unui set de task-uri ale căror valori de start-time sunt egale (la momentul *t=0*) este ca pentru toate valorile *k* > 0, *F(k, 0) ≥ 0” .*

Vom demonstra lema de mai sus luând ca exemplu *Fig. 3.6.* Se observă faptul că orice task ce se afla in *R1* trebuie să fie executat în *k* pași, având în vedere faptul ca deadline-urile acestora nu vor fi mai mari de *k*. Acest lucru implică folosirea a unități de procesare pentru a putea executa toate task-urile din *R1*. Pentru un task ce se află în a doua subdiviziune *R2*, se va permite mutarea acestuia paralel cu axa *L* pentru cel mult *Lj* diviziuni în următorii *k* pași astfel încât acesta să rămână în primul cadran al planului. Pentru restul timpului, aceste task-uri vor fi mutate paralel cu axa *C* până în momentul în care vor atinge axa *L,* terminându-și astfel execuția. Pentru aceste task-uri, timpul de procesare rămas *Cj* poate fi scris sub următoarea formă:



*Fig. 3.6. – Condiții necesare pentru ordonare*

Prin urmare, în următorii *k* pași, fiecare task trebuie mutat cel puțin cu diviziuni pe axa paralelă cu *C.* Timpul minim de procesare necesar executării tuturor task-urilor din *R2* va fi egal cu: . Timpul maxim de procesare ce ar fi necesar unui sistem cu *n* procesoare va fi . Concluzionând, pentru a realiza execuția tuturor task-urilor din primele 2 subdiviziuni *R1* si *R2*, timpul minim de procesare necesar va fi dat de formula:

Întrucât această relație este adevărată pentru , lema este demonstrată.

Teoremă: *„*Dacă există o ordonare fezabilă a unui set de task-uri ce au același start-time, atunci același set de task-uri poate fi ordonat chiar dacă start-time-urile sunt diferite si nu sunt cunoscute *a priori”.*

Cunoașterea dinainte a deadline-urilor și a timpului de procesare este suficientă pentru a se putea realiza ordonarea. Un algoritm „run-time” de ordonare ce are succes este *Least Laxity*. Explicarea algoritmului s-a făcut la începutul acestui capitol.

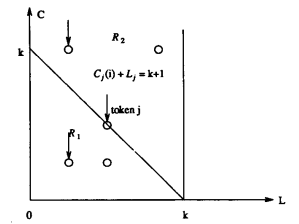
Demonstrația acestei teoreme se va realiza folosindu-se lema menționată anterior. Astfel, cunoaștem faptul că dacă există o ordonare care să îndeplinească deadline-urile task-urilor în situația în care start-time-urile acestora sunt identice. Având această premisă, trebuie să arătăm faptul că niciun deadline nu va fi omis și că după prima mișcare, această ipoteză de inducție se va menține. Astfel, în primul rând, implică ca numărul de task-uri ce se află pe axa *C* să nu depășească numărul de procesoare. În al doilea rând, pentru implică ca pentru dacă pentru ordonarea task-urilor este utilizat algoritmul Least Laxity.

Pentru a demonstra prima propoziție, vom lua ca exemplu . Acest lucru implică, conform ecuației (3.3), . Partea dreaptă a acestei inecuații reprezintă exact numărul de task-uri ce se afla pe axa *C*. Având în vedere faptul că aceste task-uri au laxitatea egală cu 0 și numărul lor este mai mic sau egal cu numărul de procesoare disponibile, acestea vor fi mutate cu o unitate în jos pe axa *C* potrivit algoritmului Least Laxity. Prin urmare, toate task-urile ce se află pe axa *C* vor fi fie mutate, fie vor staționa. Prin faptul că nu se poate ajunge la un eșec imediat al algoritmului, propoziția este demonstrară.

Pentru a demonstra a doua propoziție, va trebui să arătăm că pentru , există un *k’* astfel încât Întrucât din ipoteză, pentru , se poate concluziona că . Utilizând scenariile ilustrate in *Fig. 3.7.* si *Fig. 3.8.,* există 2 cazuri de analizat. De menționat faptul că săgețile din figură reprezintă mutarea task-urilor folosind algoritmul Least Laxity, iar cazurile în care acele săgeți lipsesc reprezintă staționarea task-urilor.

Cazul 1): În acest caz (*Fig. 3.7.*), fiecare task din stânga dreptei , fie a staționat, fie a ajuns la poziția curentă la momentul de timp *i+1*. Alegem , pentru a demonstra că

Vom avea în vedere posibila mutare a unor task-uri din *R3* . Pentru aceste task-uri, astfel acestea nu vor influența Un alt număr de task-uri poate migra din *R2* spre *R1*. Acestea vor contribui în mod egal la si deoarece . Restul de task-uri ce rămân in *R2* își vor păstra aceeași valoare a laxității si nu vor afecta Toate celelalte task-uri ce se afla in *R1* fie vor contribui la scăzând valoarea din *Cj* cu o unitate, fie vor contribui în mod egal la si în cazul în care rămân staționare. Așadar, pentru acest caz, va reprezenta numărul de task-uri din *R1* ce s-au executat în ultima unitate de timp, echivalent cu

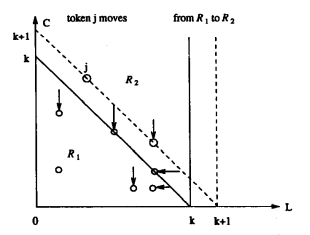


*Fig. 3.7. – Cazul F(k, i+1) ≥ F(k, i)*

Cazul 2) : În acest caz (*Fig. 3.8.*), fiecare task din stânga dreptei , a ajuns pe poziția curentă mișcându-se paralel cu axa *L*. Alegem , pentru a demonstra că

Considerăm task-urile ce vor fi mutate în jos paralel cu axa *C*. Dacă acestea se află în regiunea *R2*, atunci acestea contribuie cu la deoarece . Dacă un task se află pe dreapta , atunci acesta în mod sigur a fost la momentul anterior de timp pe dreapta , astfel acesta s-a situat in *R1* în ambele momente de timp. Deoarece timpul acestuia de procesare a scăzut, acesta contribuie în mod pozitiv la . În cazul în care task-ul se află deja în *R1*, atunci, în mod evident, contribuția acestuia va fi 1. Concluzionând, în toate cazurile, un task ce a fost mutat vertical în jos va contribui în mod pozitiv la valoarea lui .

În continuare, vom considera task-urile ce vor fi mutate pe axa orizontală. Dacă un asemenea task se află în *R2*, atunci acesta va contribui cu la . Dacă un task se află pe dreapta , atunci acesta a fost mutat la momentul anterior de pe dreapta . Contribuția acestuia la este zero deoarece acesta se află în *R1* în ambele momente de timp și timpul său de procesare nu a scăzut. Așadar, task-urile ce sunt mutate orizontal nu vor afecta valoarea . În mod similar, task-urile staționare vor avea o contribuție egală cu 0 sau 1 asupra dacă . Un lucru interesant de notat este reprezentat de faptul că task-uri staționare ce se află pe dreapta se vor muta din la momentul *i* în la momentul *i+1.* În orice caz, acestea vor contribui în mod pozitiv la din moment ce . Astfel, în final, valoarea lui va fi dată de . Această inegalitate este întotdeauna adevărată dacă toate task-urile staționare sunt poziționate în stânga dreptei . Pentru , unul din cere două cazuri prezentate va fi adevărat. Astfel, teorema este demonstrată deoarece în toate cazurile .



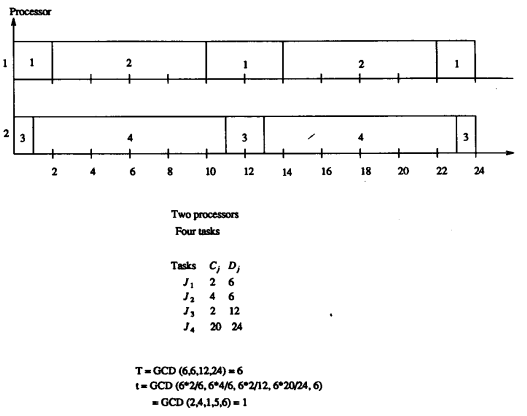
*Fig. 3.8. – Cazul F(k, j+1) ≥ F(k+1, i)*

### Task-uri periodice

Pentru a vorbi despre task-urile periodice, va trebui să introducem o nouă noțiune: factorul de utilizare (*U*). Factorul de utilizare al unui set de task-uri este egal cu . Un task poate solicita un nou timp de procesare de îndată ce deadline-ul acestuia expiră. Un aspect evident este reprezentat de faptul că reprezintă o condiție necesară pentru ordonarea unui set de task-uri, având la dispoziție *n* procesoare. Posibilitatea ca ar reprezenta, de asemenea, și o condiție suficientă pentru a se putea realiza o ordonare fezabilă reprezintă o problemă discutabilă. Vom încerca să dăm un răspuns parțial prin următoarea teoremă:

*Teoremă: „*Presupunem un set de *m* task-uri periodice având factorul de utilizare . Notăm cu si . Atunci, o condiție suficientă pentru a putea ordona având la dispoziție *n* procesoare este ca *t* să fie număr întreg.*”*

Pentru a demonstra această teoremă vom presupune că *t* este un număr întreg. Acest lucru ar rezulta în faptul că vor fi toate numere întregi. Strategia noastră este să executam taskul 1 pentru unități de timp, taskul 2 pentru unități de timp și așa mai departe. Pentru un task *j*, fiecare cerere va solicita un număr de *Cj* unități de timp și care va trebui să se execute in *Dj* unități de timp. Putem diviza *Dj* în părți, având fiecare dimensiunea T. Pentru fiecare parte vom aloca un număr de unități de procesare pentru taskul *j*. Așadar, pentru fiecare task în parte, timpul total de procesare va fi dat de urmarea formulă:. De aici putem deduce faptul că timpul total de procesare al întregului set va fi . Un astfel de exemplu este ilustrat în următoarea figură:



*Fig. 3.9. – Scenariu cu 4 taskuri periodice*

# Definirea managementului de proiecte

Pentru a înțelege managementul proiectelor, vom începe printr-o scurtă definiție a acestuia. Astfel, un proiect poate fi considerat ca având o serie de activități care:

* Au un obiectiv specific ce se concentrează pe adaosul de valoare asupra organizației
* Au definite intervale de timp în care trebuie realizate
* Au anumite limitări
* Necesită o cantitate de resurse pentru a putea fi îndeplinite (umane sau materiale precum: bani, echipament, licențe etc.)

Rezultatul proiectelor poate fi unic (ex. creăm o nouă aplicație ce nu există pe piață) sau repetitiv(ex. oferim servicii de mentenanță ce pot fi oferite și de alte companii), însă acestea trebuie să fie atinse într-o perioadă finită de timp. Deoarece companiile au la dispoziție un număr limitat de resurse, împărțirea acestora în cadrul proiectelor trebuie realizată într-un mod cât mai eficient. PMI[[9]](#footnote-9) identifică în lucrarea *PMBOK Guide[[10]](#footnote-10)* un număr de 5 grupuri de procese ce au loc în cadrul organizării proiectelor:

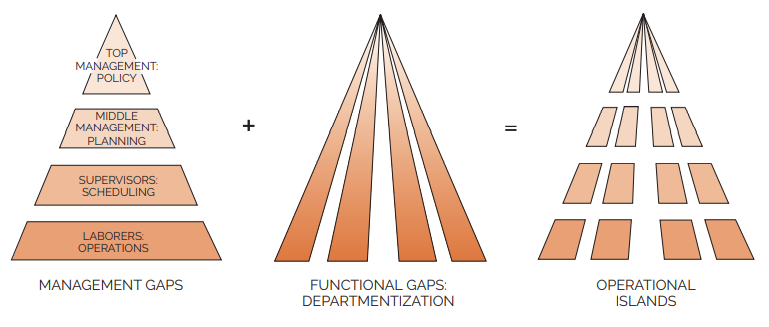
1. Inițierea proiectului
   1. Selectarea celor mai bune proiecte având în vedere limitele definite de numărul resurselor
   2. Listarea beneficiilor aduse de către proiect
   3. Pregătirea documentației proiectului
   4. Atribuirea unui Project manager
2. Planificarea proiectului
   1. Definirea cerințelor
   2. Definirea calității și cantității de muncă
   3. Definirea resurselor necesare
   4. Ordonarea activităților
   5. Evaluarea riscurilor
3. Executarea proiectului
   1. Atribuirea membrilor ce vor lua parte la proiect
   2. Coordonarea activităților
   3. Lucrul cu membrii echipei pentru îmbunătățirea muncii
4. Monitorizarea si controlul proiectului
   1. Urmărirea evoluției proiectului
   2. Compararea rezultatelor obținute cu ceea ce se dorea inițial
   3. Analiza variațiilor și a impactului rezultat
   4. Ajustarea detaliilor
5. Finalizarea proiectului
   1. Verificarea faptului că toate activitățile au fost completate
   2. Finalizarea contractului
   3. Remunerația
   4. Finalizarea documentelor aferente muncii depuse

Un management de succes al proiectelor se poate considera acela ce reușește să obțină o serie de obiective atinse în timp util, fără a depăși costul cerut, utilizând nivelul de tehnologie dorit în timp ce resursele sunt utilizate cât mai eficient, având ca scop final acceptarea de către client a produsului final. Deoarece fiecare proiect are diferite cerințe venite din partea clientului, activitățile incluse în fiecare grup de procese definit mai sus poate diferi de la proiect la proiect. *PMBOK Guide* identifică diferite metode de împărțire ale acestor activități în funcție de industria în care sunt utilizate. Astfel, acestea sunt considerate „best practice” pentru industriile respective și pot fi structurate pentru a crea metodologii de management de proiecte ce vor sta la baza oricărui proiect. Desigur, acestea pot fi personalizate pentru o varietate de proiecte.

O serie de beneficii pe care un management de proiecte eficient le are este următoarea:

* Identificarea clară a responsabilităților funcționale pentru a asigura faptul că toate activitățile sunt contabilizate
* Minimizarea necesității unui raport continuu
* Identificarea limitelor de timp pentru ordonarea activităților
* Măsurarea a ceea ce a fost realizat față de ceea ce a fost planificat
* Identificarea prealabilă a problemelor astfel încât acestea pot fi corectate
* Îmbunătățirea capacității de estimare pentru proiectele viitoare
* Cunoașterea dinainte a posibilității de a îndeplini anumite obiective propuse

În următoarea figură (*Fig. 4.0.*) vom ilustra modul în care majoritatea companiilor sunt organizate. Întotdeauna vor exista anumite breșe între diferitele nivele de management. Însă, totodată, astfel de breșe vor exista și între departamentele ce sunt situate pe același nivel, creându-se astfel o multitudine de mici grupuri operaționale ce refuză să comunice între ele datorită fricii de a divulga informații ce ar putea face adversarul mai puternic. Responsabilitatea managerului de proiect este de a realiza conexiunea acestor grupuri astfel încât acestea să comunice și să lucreze împreună pentru realizarea obiectivelor comune.



*Fig. 4.0. Breșe organizatorice*

# 5. Structura aplicației

## 5.1. Cerințele aplicației

Aplicația este capabilă să ruleze pe sistemele de operare Windows 10 și Windows 11, însă, datorita faptului ca JDK[[11]](#footnote-11) este disponibil pe majoritatea platformelor desktop (Windows, Mac OS X si Linux), aplicațiile JavaFX compilate cu JDK 7 sau o versiune ulterioară pot, de asemenea, rula pe majoritatea platformelor desktop. Compatibilitatea cross-platform permite o experiență run-time consistentă atât pentru dezvoltatori, cât și pentru utilizatori.

Aplicația va oferi o gamă largă de funcționalități pe lângă funcția de baza a acesteia, aceea de a realiza o ordonare automată a activităților în cadrul proiectelor. Pentru a realiza această ordonare s-a implementat algoritmul propus de [Christian Artigues, Sophie Demassey, Emmanuel Neron][[12]](#footnote-12) ce testează fezabilitatea ordonării unui set de activități dat. Desigur, pentru ca aplicația să se muleze cât mai mult pe un mediu real de Project management, s-a decis introducerea mai multor elemente ce se regăsesc in practică. Astfel, în cadrul aplicației noastre, un utilizator nu poate avea mai mult de o activitate atribuita la un moment de timp, lucru ce presupune mai multe restricții asupra ordonării. Totodată, în cazul în care un set de activități nu poate fi ordonat in termen de 24 h, atunci rezultă faptul că nu există o ordonare fezabilă, iar ca rezultat, doar o parte din activități vor fi ordonate.

Pentru interacțiunea cu utilizatorul se va implementa o interfață grafică folosindu-se UI[[13]](#footnote-13) framework-ul JavaFX prin intermediul căreia utilizatorul va putea accesa funcționalitățile implementate.

În implementarea proiectului, se va folosi o abordare orientată pe obiecte ce va oferi scalabilitate si posibilitatea adăugării simple de noi funcționalități.

## 5.2. Definirea arhitecturii aplicației

Aplicația este realizată folosindu-se bine-cunoscuta arhitectură Client-Server. Astfel, server-ul va reprezenta componenta ce manageriaza si livrează majoritatea datelor folosite de către client. Conexiunea dintre cele două entități este realizată prin intermediul socket-urilor, pe un port stabilit dinainte. Mesajele trimise intre cele doua entități vor implementa formatul de reprezentare JSON[[14]](#footnote-14).

### 5.2.1. Proiecte Maven

Maven este un sistem de build si administrare a proiectelor. Funcționalitățile sale principale sunt: descrierea procesului de build al software-ului și descrierea dependențelor acestuia. Aceste proiecte conțin un fișier denumit *pom.xml* (Project Object Model), ce are o structură specifică. POM-ul conține informații despre module, precum și despre dependențele proiectului. Utilizarea acestui tip de proiect ne-a permis o foarte ușoară descărcare dinamică a bibliotecilor Java utile pe parcursul desfășurării procesului de realizare a aplicației. Astfel, prin adăugarea acestor dependențe, am fost scutiți de descărcarea fiecărei biblioteci in parte.

### 5.2.2. Modelul MVC[[15]](#footnote-15)

În realizarea aplicației (Client) s-a folosit modelul arhitectural MVC (Model-View-Controller). Acest pattern realizează separarea aplicației în 3 părți principale: model-ul, view-ul și controller-ul. Fiecare dintre aceste 3 elemente sunt construite astfel încât să manipuleze anumite aspecte specifice de dezvoltare a aplicației. Acesta reprezintă, în momentul de fata, unul dintre cele mai frecvente Framework-uri folosite în industrie deoarece permite o ușoară scalabilitate proiectelor. În continuare vom detalia fiecare dintre cele 3 elemente.

#### 5.2.2.1. Model

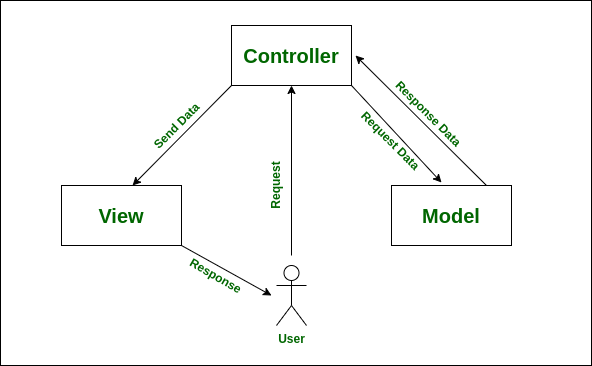
Componenta Model corespunde tuturor obiectelor cu care user-ul lucrează. În principiu, acesta va reprezenta datele ce sunt transferate între View si Controller.

#### 5.2.2.2. View

Componenta View va corespunde tuturor interfețelor grafice ale aplicației. De exemplu, Task View va include o componentă UI ce va conține elemente precum: text box, combo box, table view etc. cu care utilizatorul final va interacționa.

#### 5.2.2.3. Controller

Controller-ul acționează ca o interfață între componentele View si Model pentru a procesa toate cererile primite, manipulând datele prin intermediul componentei Model și interacționând cu componenta View pentru a reda rezultatul final. De exemplu, Task Controller va manipula toate inputurile venite de la Task View pentru a realiza update-uri în baza de date prin intermediul Task Model.

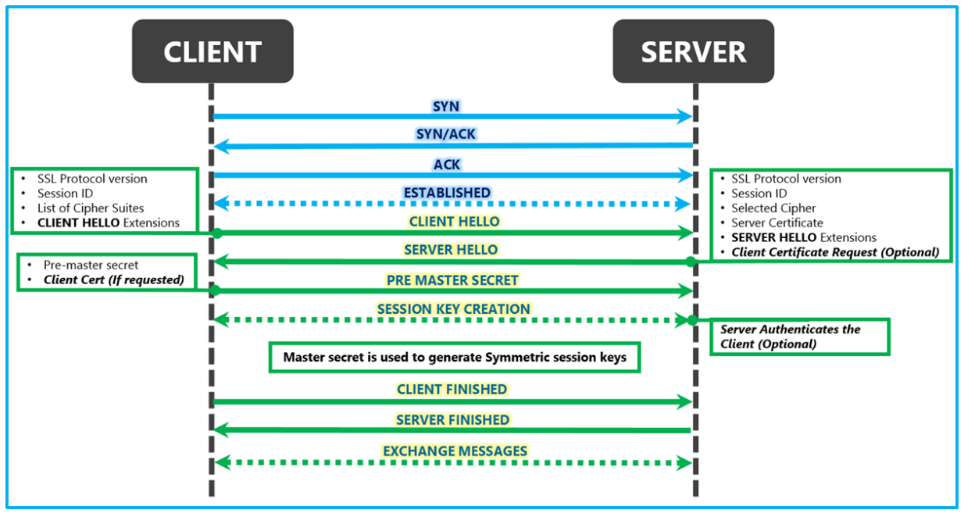


*Fig. 5.0. – Modelul arhitectural MVC*

## 5.2.3. Securitatea conexiunii Client-Server

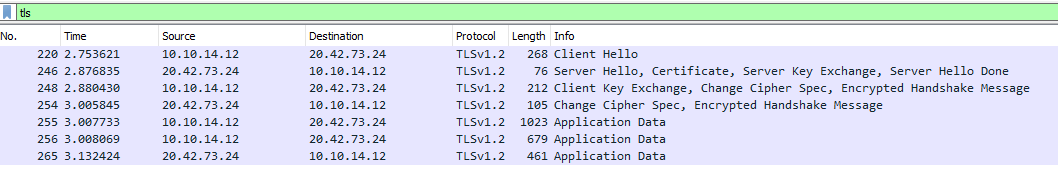
Așa cum a fost menționat și la începutul acestui capitol, conexiunea dintre cele doua entități Client-Server se va realiza prin intermediul socket-urilor. Astfel, pentru a realiza o conexiune sigură între cele doua entități a fost folosit TLS[[16]](#footnote-16).

Deoarece cele doua entități pot comunica între ele cu sau fără TLS, este necesar pentru client sa ceara server-ului stabilirea unei conexiune TLS. Odată ce clientul și serverul au stabilit folosirea TLS-ului, aceștia negociază o conexiune stateful folosind procedura de handshake. Protocoalele folosesc un handshake împreună cu o criptare asimetrică pentru a stabili și o cheie de sesiune ce va fi folosită mai apoi pentru a cripta comunicația folosind o criptare simetrică.



*Fig. 5.1. – TLS Exchange*

În următoarea imagine este prezentată o captură a traficului de date realizată cu Wireshark în care se poate observa folosirea TLS-ului în comunicația dintre client și server.



*Fig. 5.2. – Captura de trafic TLS*

# 6. Implementarea Software

Mediul folosit pentru dezvoltarea aplicației a fost sistemul de operare Windows 10. Pornind de la caracteristicile deținute de task-uri, cunoscute din teoria sistemelor de operare și prezentate în capitolele anterioare, s-a început conturarea aplicației prin realizarea bazei de date. Pentru stocarea informațiilor necesare aplicației s-a decis utilizarea SQL Server. In continuare vom prezenta arhitectura bazei de date:

## 6.1. Arhitectura bazei de date

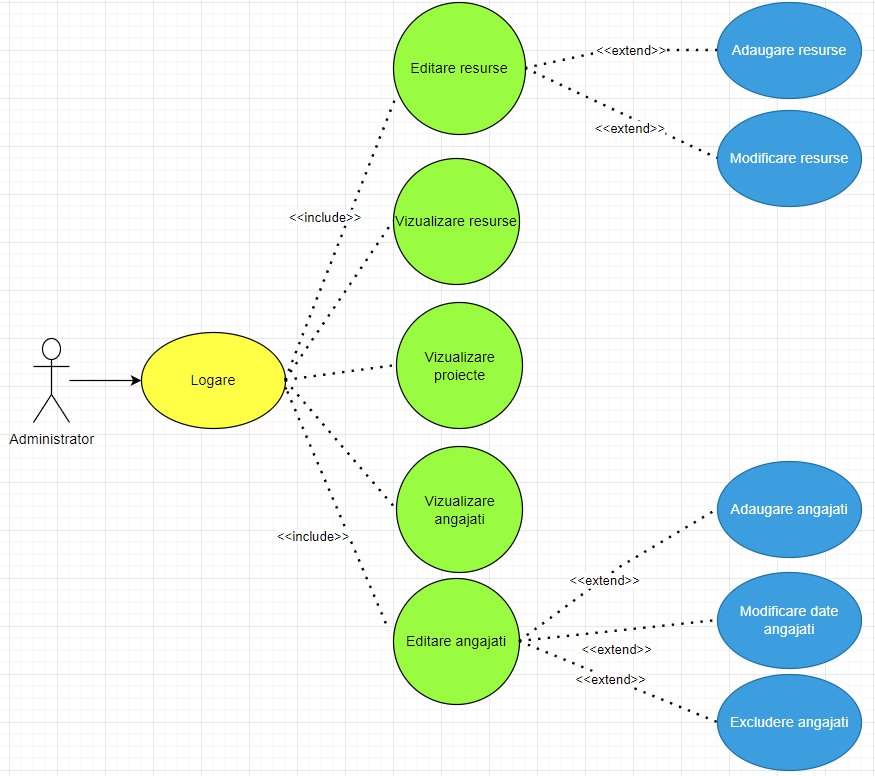
*Fig. 6.1. – Arhitectura bazei de date*

Se poate observa faptul că tabela Taskuri conține toate caracteristicile provenite din cadrul sistemelor de operare: start time, deadline și durata. Pe lângă acestea, a fost nevoie de adăugarea unor caracteristici suplimentare. Printre acestea menționam: relații de precedență definite prin intermediul unei coloane Foreign key către Primary key-ul aceleiași tabele, asignarea acestora către angajați prin intermediul unei tabele many-to-many ce face legătura între tabelele Taskuri și Useri. De asemenea, coloana „Periodicitate” va fi folosită în momentul în care inseram valori in tabela Taskuri\_Reale. Practic, tabela din urmă menționată va conține activitățile efective ce vor fi afișate în cadrul aplicației, iar tabela Taskuri va conține activitățile generale. De exemplu, daca vom defini o activitate ce va avea periodicitatea setata ca „Daily” pentru o perioada de 10 zile, în tabela Taskuri vom avea o singura intrare, iar tabela Taskuri\_Reale va conține 10 intrări specifice fiecărei activități, urmând ca apoi să putem face modificări pe fiecare în parte.

Această parte a fost cea specifică paralelei cu sistemele de operare. În continuare, pentru a realiza gestionarea unei organizații am avut nevoie de mai multe tabele. În primul rând, pentru a realiza autentificarea în cadrul aplicației, am avut nevoie de o tabela în care sa stocam conturile (Useri). În cazul în care contul este deținut de un administrator al companiei, ID-ul acestuia se va afla și în tabela Admini. Pentru că vorbim despre organizații reale, fiecare angajat va avea mai multe caracteristici personale: adresa, telefon, email etc. O practică des utilizată în realizarea bazelor de date este ca informațiile cu caracter personal să fie ținute separat de informațiile specifice contului, astfel, după cum se poate observa și în diagrama prezentată anterior, există o tabelă specifică informațiilor cu caracter personal. Un alt element principal al bazei de date, pe lângă activități, este reprezentat de resursele de care dispun companiile. Astfel, pentru definirea acestora a fost creată o tabelă în care sunt reținute detaliile despre acestea precum: denumire, descriere, cantitate. Legăturile dintre resurse și proiecte/activități sunt realizate prin intermediul a două tabele de tip many-to-many.

## 6.2. Tipurile de utilizatori

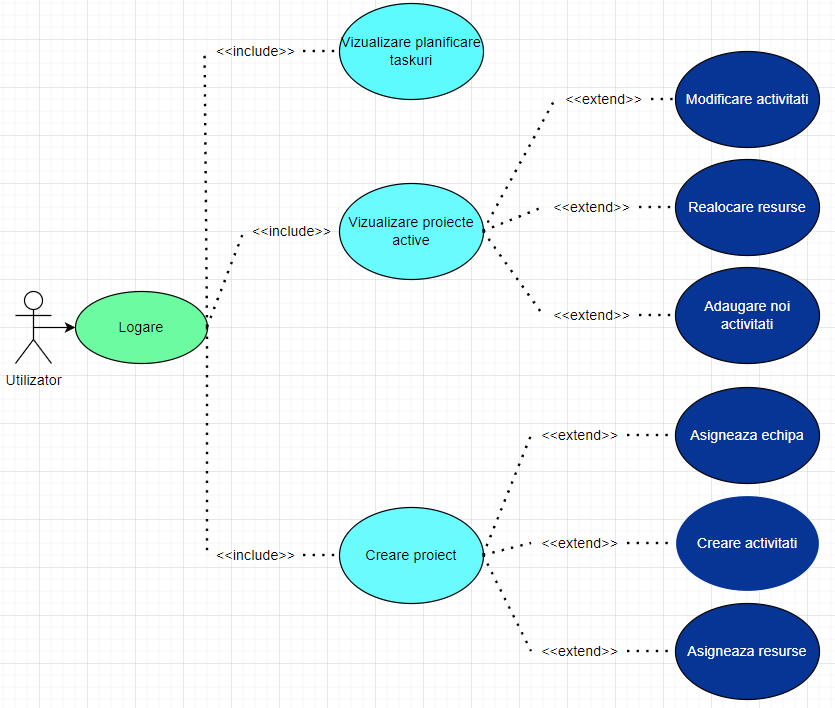
În funcție de rolul pe care o persoană o are în cadrul organizației din care face parte, au fost definite 2 roluri posibile: administrator sau angajat normal. Astfel, aplicația nu oferă posibilitatea creării unui nou cont de angajat, ci acestea vor putea fi create doar de către administrator. S-a ales această modalitate deoarece s-a luat în calcul faptul că o persoană ar putea crea conturi false în cadrul diverselor companii. Astfel, opțiunea de register a aplicației va oferi doar posibilitatea creării unui cont de administrator împreună cu organizația acestuia. În continuare, administratorul va putea defini conturile de utilizator pentru fiecare angajat in parte. O altă funcție pe care doar administratorul o deține este reprezentată de posibilitatea definirii resurselor companiei. Restul acțiunilor pe care un administrator le mai poate face sunt doar de vizualizare a datelor privind: angajații, proiectele si resursele. Diagrama din figura următoare are rolul de a clarifica acțiunile pe care un administrator le poate face:



*Fig. 6.2. – Diagrama cazurilor de utilizare pentru modul administrator*

Activități mai complexe sunt definite pentru modul de angajat normal deoarece prin intermediul acestor conturi se vor putea crea noi proiecte și studia modul de ordonare al activităților în cazul în care există realocări de resurse. Astfel, pe lângă acțiunile standard de vizualizare, un angajat poate inițializa un nou proiect. Inițializarea unui proiect va presupune 3 pași consecutivi pe care angajatul desemnat ca manager de proiect îi are de parcurs. În primul rând, utilizatorul va trebui să completeze informațiile de bază despre proiect precum: denumire, intervalul de timp în care se va realiza, descrierea și asignarea personalului. După ce sunt completate aceste câmpuri, managerul de proiect va desemna rolul și nivelul de prioritate al fiecărui angajat din cadrul proiectului (rolul va permite sau va bloca pe viitor încercarea angajaților de a realiza modificări în cadrul proiectului), va crea echipe (acestea au rol informativ și nu vor influenta ordonarea activităților) și va asigna resursele necesare executării acestuia. Ultimul pas pe care managerul îl are de făcut este reprezentat de crearea activităților ce vor trebui executate pe parcursul desfășurării proiectului. Ulterior se vor mai putea adăuga activități suplimentare dacă situația o impune. În momentul în care un proiect este creat, se realizează un request către Server care va realiza și ordonarea activaților.

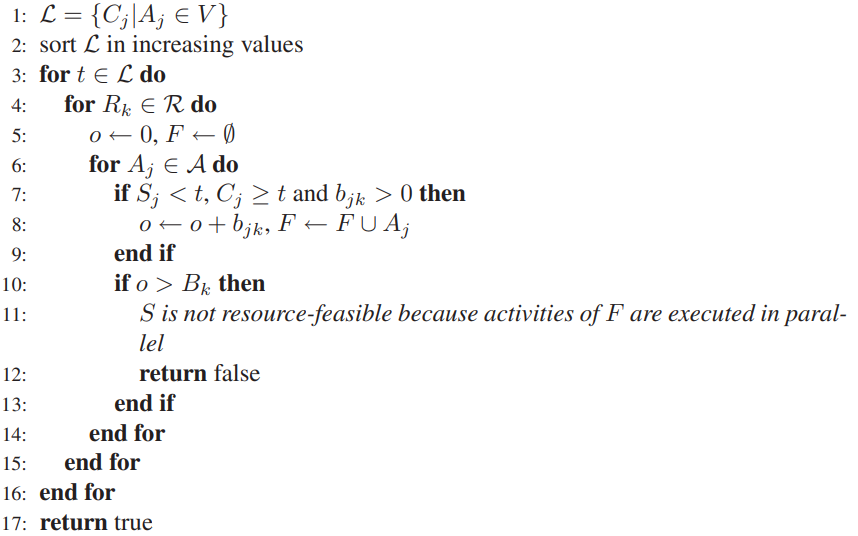
Cea mai importantă pagină din cadrul proiectului, cea care face și obiectul cercetării noastre, este reprezentată de pagina de vizualizare a proiectelor. În cadrul acesteia, angajații ce dețin un nivel de prioritate necesar modificării datelor curente pot realiza următoarele acțiuni: vizualizarea modului în care algoritmul a alocat resursele, adăugarea unei noi activități, modificarea numărului de resurse atribuit proiectului și acceptarea unor modificări propuse de aplicație, oferite pentru activitățile ce nu au putut fi incluse în timpul de lucru inițial. În figura următoare este reprezentată diagrama cazurilor de utilizare pentru un utilizator ce nu are rol de administrator:



*Fig. 6.3. Diagrama cazurilor de utilizare pentru utilizatori fără rol de administrator*

## 6.4. Algoritmul de ordonare a activităților

Principala funcționalitate a aplicației este reprezentată de funcția ce permite o ordonare automată a activităților în cadrul proiectelor. Astfel, s-a decis că algoritmul de la care vom pleca să fie cel propus de Christian Artigues, Sophie Demassey si Emmanuel Neron[[17]](#footnote-17). Notațiile pe care le vom vedea în următorul pseudocod sunt aceleași notații ce au fost prezentate in Capitolul 2.2.2. :



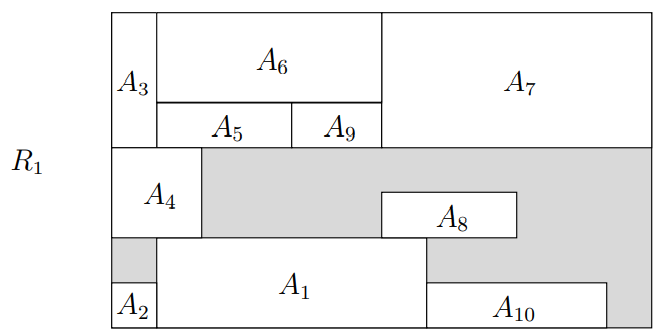
*Fig. 6.4. – Pseudocod al algoritmului ce testează fezabilitatea unui scenariu de task-uri*

Practic, acest algoritm nu ne oferă și o ordonare a activităților, așa cum am fi predispuși să credem, ci doar testează fezabilitatea scenariului dat. De la acest algoritm a trebuit să plecăm în procesul de implementare, iar primul pas pe care l-am avut de implementat a fost setarea inițială a momentului de timp la care fiecare activitate ar trebui să își înceapă execuția. Activitățile ce nu aveau relații de precedență definite față de alte activități vor avea acest punct de plecare setat cu valoarea 0, iar restul vor avea inițial acest punct la :

, unde Si și pi reprezintă punctul de început, respectiv durata activității față de care există relația de precedență. Următorul pas a presupus ca în momentul în care intrăm pe ramura din care rezultă că setul dat nu are o ordonare fezabilă, să incrementăm punctul de start al primei activități, apoi să recalculăm toate celelalte puncte de start pentru că, din cauza relațiilor de precedență, se pot modifica mai multe activități simultan. În final, vom relua execuția algoritmului până când toate activitățile pot fi ordonate.

Este important de menționat și faptul că până la începerea execuției algoritmului, se va realiza o translatare a angajaților în resurse, astfel încât unui angajat să nu i se poată atribui mai mult de 1 activitate la un moment de timp. Pentru a reține aceste informații, fiecărei activități i-a fost atribuită un HashMap în care am reținut ID-ul resursei și cantitatea necesară execuției. Algoritmul este prezentat in Anexa A.

Ulterior execuției algoritmului, în cadrul bazei de date, vor fi introduse valorile actuale de start pentru fiecare activitate în parte. Pentru a realiza o vizualizare asemănătoare cu cea prezentată în carte (Fig. 6.5.), a fost folosit un algoritm de backtracking pentru a reuși să poziționăm fiecare activitate. Nu ne interesează numărul resursei pe care o activitate o folosește, astfel, prima soluție oferită de algoritm va fi si cea aleasă pentru vizualizare. Algoritmul de backtracking este prezentat în Anexa B.

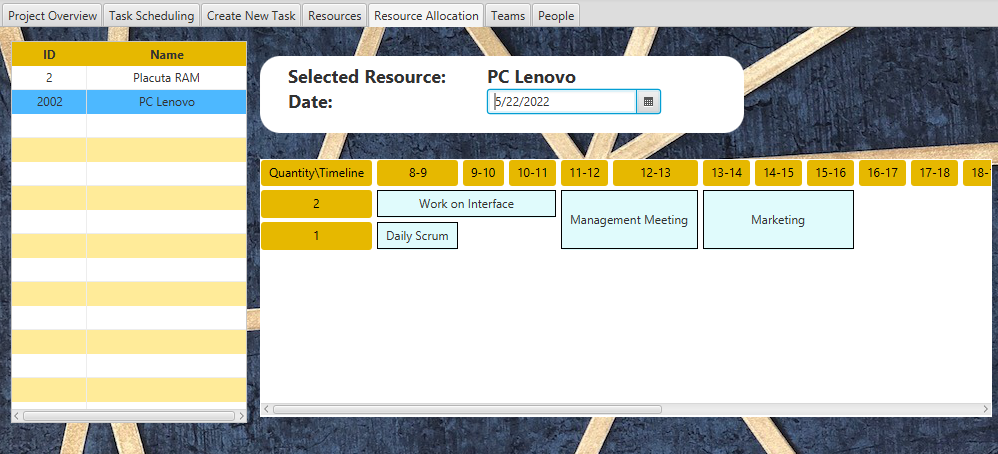


*Fig. 6.5. – Exemplu de vizualizare a utilizării unei resurse R1 pe un scenariu cu 10 activități*

# 7. Interacțiune utilizator-aplicație

În cadrul realizării interfeței grafice, s-a ținut cont de principiile de Interacțiune Om-Calculator pentru a ajunge la un rezultat final care să fie cât mai atractiv și ușor de utilizat de către potențialii clienți.

În imaginea următoare este ilustrat un exemplu de ordonare a activităților pentru un scenariu în care avem 4 activități si 2 resurse:



*Fig.7.1. – Ilustrare a unei ordonări efectuata de aplicație*

## 7.1. Evaluarea euristica pe baza principiilor IOC[[18]](#footnote-18)

În urma realizării primului prototip al aplicației, s-a realizat o evaluare euristică ce a avut rolul de a identifica probleme ce pot apărea în urma utilizării de către un utilizator obișnuit, dar și pentru a scoate în evidență punctele forte ale acesteia.

Sugestivitatea interfeței este redată prin îmbinarea diverselor componente utilizate: imagini, formulare, tabele etc. cu scopul final de a scoate în evidență caracteristicile de bază ale aplicației.

Consistența interfeței are un rol important deoarece ajută utilizatorul să observe diversele componente folosite și, în același timp, să intuiască utilitatea acestora.

În continuare, vom prezenta unele dintre principiile utilizate în evaluarea euristică:

### 7.1.1. Perceived Affordances

Acest principiu se referă la proprietățile percepute ale unui obiect care sugerează modul în care l-am putea folosi. Astfel, design-ul aplicației a trebuit implementat astfel încât să invite utilizatorul să ia acțiunile posibile, proprietățile obiectelor afișate fiind deduse din design. Acest principiu se regăsește în majoritatea paginilor implementate. De exemplu, butoanele ce au semnul „+” indică adăugarea diverselor informații sau butoanele colorate în roșu indică o acțiune negativă (ștergere/golire).

### 7.1.2. Maparea

Maparea reprezintă setul de relații posibile dintre controlul și afișarea obiectelor, altfel spus, compatibilitatea dintre cele două. Această caracteristică se regăsește în majoritatea paginilor create, astfel, la fiecare eveniment al mouse-ului peste un buton acesta se va modifica într-un mod intuitiv pentru a-i oferi un feedback utilizatorului și a-i confirma că acțiunea sa este realizată.

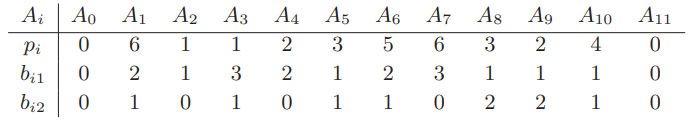
### 7.1.3. Cauzalitatea

Cauzalitatea reprezintă ceea ce se întâmpla imediat după ce o acțiune este asumată de către utilizator, sau altfel spus, reprezintă interpretarea feedback-ului. În cazul aplicației noastre, fiecare acțiune ce presupune o comunicare cu Serverul va întoarce un feedback pentru a confirma sau infirma ceea ce utilizatorul a încercat să facă. Un exemplu al acestei caracteristici poate fi considerat adăugarea unei noi activități în cadrul proiectului la care lucrăm. În momentul în care s-a realizat adăugarea noii activități și realizarea unei noi ordonări, utilizatorul va fi anunțat printr-un element de tip Alert.

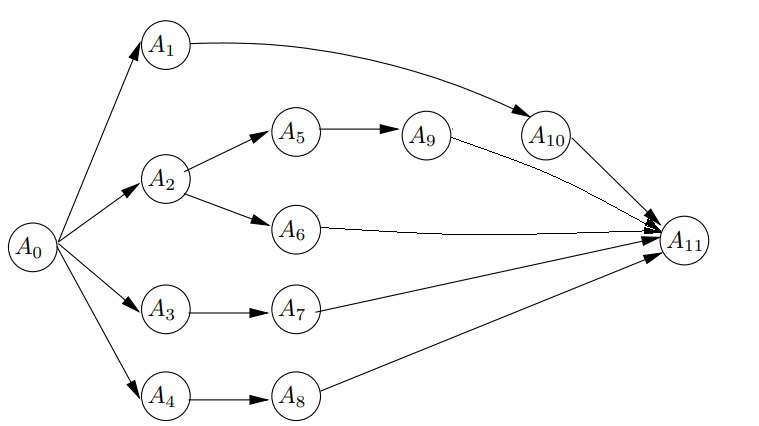
## 7.2. Testarea

Pentru a putea confirma faptul că algoritmul implementat de către noi oferă rezultate corecte, a fost creat un proiect pentru care se putea verifica ordinea în care sunt ordonate activitățile.

Astfel, pentru acest caz vom considera un scenariu de activități în care avem un număr de activitati si resurse având următoarele disponibilități . În următorul tabel putem observa durata și cantitățile de resurse necesare executării fiecărei activități în parte.

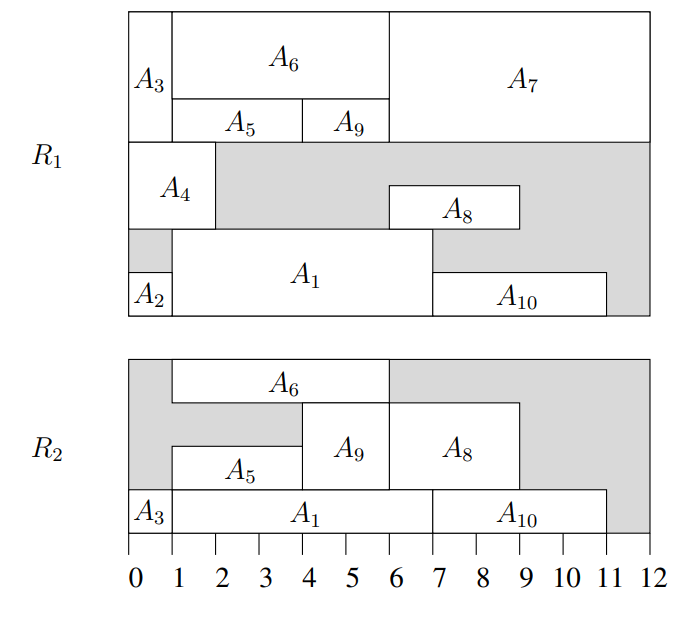


În următoarea imagine se poate observa graficul ce ilustrează relațiile de precedență definite între activități:



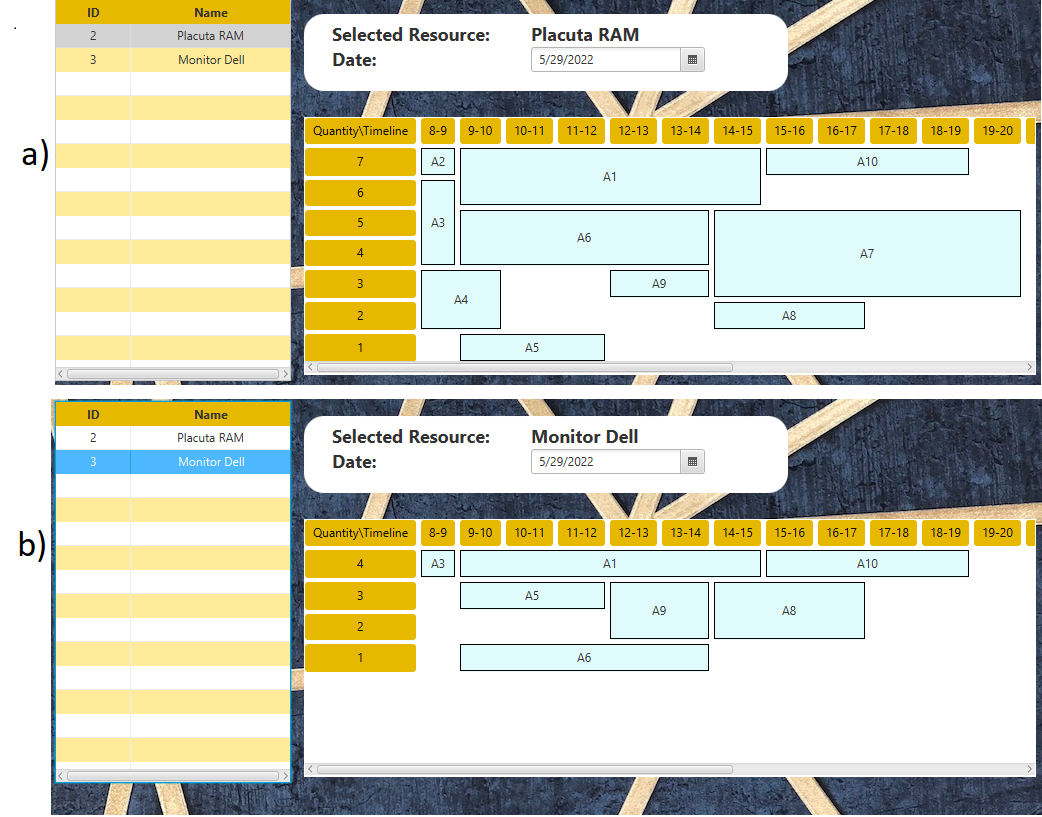
*Fig. 7.2. – Grafic ce ilustrează relațiile de precedenta*

În figura următoare este prezentată ordonarea activităților pentru scenariul dat:



*Fig. 7.3. – Ordonarea dată pentru scenariul dat*

După ce datele au fost introduse în aplicație și proiectul a fost creat, algoritmul implementat a realizat ordonarea activităților. Rezultatul este prezentat în următoarea figură:



*Fig. 7.4. – Ordonarea oferit*ă *de aplicație*

În fereastra de vizualizare a alocării resurselor se poate observa cantitatea și timpul pentru care o resursă este alocată unei activități anume. În cazul nostru, resursele folosite pentru exemplul dat au fost: „Plăcuță RAM” și „Monitor Dell”. În partea dreaptă a imaginilor se poate observa cantitățile deținute de proiect iar în partea de sus se poate observa o axă a timpului în care unitatea de timp este reprezentată de 1h.

Deși la prima vedere ar putea părea că cele două rezultate diferă, dacă observăm punctele de început al fiecărei activități în parte, vom observa faptul că rezultatele sunt identice. Această diferență de afișare apare deoarece, după cum am menționat și în capitolul 6.4., numărul resursei folosite nu contează pentru algoritm, iar afișarea este realizată prin intermediul unui algoritm de backtracking, astfel prima soluție în care blocurile corespunzătoare activităților nu se suprapun este și soluția aleasă pentru afișare.

# 8. Concluzii

## 8.1. Probleme întâmpinate

Pe parcursul realizării aplicației au fost întâmpinate și anumite probleme. O primă problemă a fost reprezentată de modul în care a trebuit realizată serializarea obiectelor pentru a putea fi transmise între cele 2 entități: Client și Server. Pentru a facilita această acțiune s-a decis utilizarea bibliotecii *Gson* ce are ca rol convertirea obiectelor Java într-o reprezentare JSON. De asemenea, biblioteca poate fi folosită și în sens invers, pentru a converti stringuri JSON în echivalentul obiectelor Java. Pentru a realiza această acțiune de serializare/deserializare a fost nevoie ca toți membri claselor de pe ambele componente să aibă aceeași denumire (clasa Proiect de pe Client are exact aceeași membri precum clasa Proiect prezentă pe Server). Deși această abordare a scutit timp prin prisma faptului că realiza serializarea întregului obiect imediat, a apărut o problemă în cazul în care obiectul ce se dorea a fi serializat avea în componența sa un membru de tip LocalDate deoarece biblioteca Gson nu poate realiza serializarea acestui tip de date. Astfel, a fost nevoie de crearea a două Adaptere speciale care să indice modul în care ar trebui serializat acest tip de date. Codul necesar creării unui astfel de Adaptor este următorul:

public class LocalDateSerializer implements JsonSerializer<LocalDate> {  
 private static final DateTimeFormatter *formatter* = DateTimeFormatter.*ofPattern*("d-MMM-yyyy");  
  
 @Override  
 public JsonElement serialize(LocalDate localDate, Type srcType, JsonSerializationContext context) {  
 return new JsonPrimitive(*formatter*.format(localDate));  
 }  
}

Se poate observa că o variabilă de tip LocalDate va fi transformată după următorul format: „d-MMM-yyyy”. Pentru Adaptorul ce realizează deserializarea codul este asemănător:

public class LocalDateDeserializer implements JsonDeserializer<LocalDate> {  
 @Override  
 public LocalDate deserialize(JsonElement json, Type typeOfT, JsonDeserializationContext context)  
 throws JsonParseException {  
 return LocalDate.*parse*(json.getAsString(),  
 DateTimeFormatter.*ofPattern*("d-MMM-yyyy").withLocale(Locale.*ENGLISH*));  
 }  
}

O altă problemă întâmpinată a fost reprezentată de translatarea personalului ce reprezintă resursa umană a organizației în resurse pentru a putea fi luați în calcul în momentul în care algoritmul realizează ordonarea activităților. S-a decis ca pentru memorarea numărul de resurse necesar unei activități să se folosească un HashMap în care erau memorate perechi de forma <ID resursa, Cantitate necesara>. Problema a apărut în momentul în care a trebuit să introducem și angajații în această listă deoarece s-ar fi putut ajunge la un caz de ambiguitate în care o resursă ar fi avut un ID identic cu angajatul ce avea de executat task-ul respectiv. Astfel s-a decis ca în HashMap-ul respectiv ID-ul angajaților va fi memorat cu semn negativ. Acest lucru elimină cazurile de ambiguitate deoarece atât resursele cât și angajații au un ID unic.

## 8.2. Rezultate obținute

Aplicația își atinge cu succes scopul principal, acela de a oferi ordonări fezabile pe seturile de activități date în funcție de alocarea resurselor avute. Pe lângă backend-ul avut, partea de frontend a aplicației se prezintă cu o interfață intuitivă pentru orice tip de utilizator. Deși se pot face optimizări atât pe partea de cod, cât și pe partea de interacțiune cu utilizatorul, aplicația reprezintă un bun punct de plecare pentru rezolvarea problemei propuse și pentru îmbunătățiri viitoare.

## 8.3. Îmbunătățiri propuse

După cum am menționat și în începutul prezentei lucrări, aplicația dezvoltată rezolvă problema standard RCPSP. Astfel, aplicația ar beneficia de un plus în cazul în care s-ar implementa și anumite variații ale problemei propuse. Astfel, modificări precum: posibilitatea întreruperii unei activități și reluarea acesteia într-un moment ulterior ar reprezenta un beneficiu important.

De asemenea, s-ar putea realiza modificări și pentru securitatea aplicației prin sanitizarea tuturor inputurilor pe care un utilizator le poate introduce.

# 9. Bibliografie

[1] Carlos Montoya, Odile Bellenguez-Morineau, Eric Pinson & David Rivreau, “Handbook on Project Management and Scheduling Vol. 1”, Springer, 2014

[2] Christian Artigues, Sophie Demassey & Emmanuel Neron, “Resource-Constrained Project Scheduling: Models, Algorithms, Extension and Applications”, ISTE, 2008

[3] M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," in IEEE Transactions on Software Engineering, vol. 15, no. 12, Dec. 1989

[4] Sonke Hartmann, Dirk Briskorn, “An updated survey of variants and extensions of the resource-constrained project scheduling problem”, European Journal of Operational Research, 2021

[5] Sonke Hartmann, “Project scheduling with resource capacities and requests varying with time: a case study”, Springer, 2013

# 10. Anexe

## Anexa A

public boolean algorithm() {  
  
 ArrayList<Integer> listaCompletion = new ArrayList<>();  
 for (int i = 0; i < listaTaskuri.size(); i++) {  
 listaCompletion.add(listaTaskuri.get(i).getCompletionTime());  
 }  
 Collections.*sort*(listaCompletion);  
  
  
 */\*\* Sort on starttime\*/* Collections.*sort*(listaTaskuri);  
  
 */\*\*list of completions times\*/* List<Integer> listWithoutDuplicates = listaCompletion.stream().distinct().collect(Collectors.*toList*());  
  
 int old\_value = listaTaskuri.get(0).getStartTime();  
 TaskReal tobeModified = listaTaskuri.get(0);  
  
 for (int i = 0; i < listWithoutDuplicates.size(); i++) {  
  
 int t = listWithoutDuplicates.get(i);  
 old\_value = listaTaskuri.get(0).getStartTime();  
 tobeModified = listaTaskuri.get(0);  
  
 for (int k = 0; k < listaResurse.size(); k++) {  
 int o = 0;  
 ArrayList<TaskReal> F = new ArrayList<>();  
  
 for (int j = 0; j < listaTaskuri.size(); j++) {  
  
 int start = listaTaskuri.get(j).getStartTime();  
 int completion = listaTaskuri.get(j).getCompletionTime();  
 int resRequest = listaTaskuri.get(j).getQuantityOfResourceRequest(listaResurse.get(k).getID());  
 if ((start < t) && (completion >= t) && (resRequest > 0)) {  
 o += resRequest;  
 F.add(listaTaskuri.get(j));  
  
 */\*\*TO BE MODIFIED\*/* if (listaTaskuri.get(j).getStartTime() > old\_value) {  
 tobeModified = listaTaskuri.get(j);  
 old\_value = listaTaskuri.get(j).getStartTime();  
 } else {  
 int area = tobeModified.getCompletionTime() \* tobeModified.getQuantityOfResourceRequest(listaResurse.get(k).getID());  
 int curentArea = listaTaskuri.get(j).getCompletionTime() \* listaTaskuri.get(j).getQuantityOfResourceRequest(listaResurse.get(k).getID());  
 if (curentArea > area) {  
 tobeModified = listaTaskuri.get(j);  
  
 }  
 }  
 }  
  
 if (o > listaResurse.get(k).getCantitate()) {  
 int currentStart = tobeModified.getStartTime();  
 currentStart++;  
 tobeModified.setStartTime(currentStart);  
 return false;  
 }  
 }  
 }  
 }  
 return true;  
  
 }

## Anexa B

public void RecursiveFunction(int timeslot, int placedTimeslot, String[][] m, int noTaskPut, ResourceModel res, ArrayList<TaskRealModel> listaTaskuri) {  
 if (found == 1)  
 return;  
 if (noTaskPut == listaTaskuri.size()) {  
 printMatrix(m, res.getCantitate(), getMakespan(listaTaskuri));  
 found = 1;  
 return;  
 }  
 if (timeslot > getMakespan(listaTaskuri))  
 return;  
 if (getNoTimeslot(timeslot, listaTaskuri) == 0)  
 RecursiveFunction(timeslot + 1, 0, m, noTaskPut, res, listaTaskuri);  
 else if (placedTimeslot == getNoTimeslot(timeslot, listaTaskuri))  
 RecursiveFunction(timeslot + 1, 0, m, noTaskPut, res, listaTaskuri);  
 else {  
 TaskRealModel task = getTaskNoTimeslot(timeslot, placedTimeslot, listaTaskuri);  
  
 int nrResurse = task.getQuantityOfResourceRequest(res.getId());  
  
 for (int rand = 0; rand < (res.getCantitate() - nrResurse + 1); rand++) {  
 int start = task.getStartTime();  
 int durata = task.getDuration();  
  
 boolean result = checkSpace(rand, start, nrResurse, durata, m);  
 if (result) {  
 for (int linie = rand; linie < rand + nrResurse; linie++) {  
 */\*\*\*/* this.positionMap.put(task.getID(), rand);  
 */\*\*\*/* for (int coloana = start; coloana < start + durata; coloana++) {  
 m[linie][coloana] = task.getName();  
 }  
 }  
  
  
 RecursiveFunction(timeslot, placedTimeslot + 1, m, noTaskPut + 1, res, listaTaskuri);  
 if (found == 1)  
 return;  
 for (int linie = rand; linie < rand + nrResurse; linie++) {  
 for (int coloana = start; coloana < start + durata; coloana++) {  
 m[linie][coloana] = "0";  
 }  
 }  
 }  
 }  
 }  
}

1. <https://www.mcvuk.com/business-news/over-1000-people-made-gta-v/> [↑](#footnote-ref-1)
2. Sonke Hartmann, Dirk Briskorn, “An updated survey of variants and extensions of the resource-constrained project scheduling problem”, European Journal of Operational Research, 2021 [↑](#footnote-ref-2)
3. Carlos Montoya, Odile Bellenguez-Morineau, Eric Pinson & David Rivreau, “Handbook on Project Management and Scheduling Vol. 1”, Springer, 2014 [↑](#footnote-ref-3)
4. Sonke Hartmann, “Project scheduling with resource capacities and requests varying with time: a case study”, Springer, 2013 [↑](#footnote-ref-4)
5. Christian Artigues, Sophie Demassey & Emmanuel Neron, “Resource-Constrained Project Scheduling: *Models, Algorithms, Extension and Applications*”, ISTE, 2008 [↑](#footnote-ref-5)
6. Resource-Constrained Project Scheduling Problem [↑](#footnote-ref-6)
7. M. L. Dertouzos and A. K. Mok, "Multiprocessor online scheduling of hard-real-time tasks," in IEEE Transactions on Software Engineering, vol. 15, no. 12, Dec. 1989 [↑](#footnote-ref-7)
8. M. Dertouzos, “Control robotics: The procedural control of physical processes”, Proc. IFIP Cong., 1974, pag. 807-813 [↑](#footnote-ref-8)
9. Project Management Institute [↑](#footnote-ref-9)
10. PMBOK reprezintă o marcă înregistrată de către PMI [↑](#footnote-ref-10)
11. Java Development Kit [↑](#footnote-ref-11)
12. Christian Artigues, Sophie Demassey & Emmanuel Neron, “Resource-Constrained Project Scheduling: *Models, Algorithms, Extension and Applications*”, ISTE, 2008, pag. 27 [↑](#footnote-ref-12)
13. User Interface [↑](#footnote-ref-13)
14. JavaScript Object Notation [↑](#footnote-ref-14)
15. Model-View-Controller [↑](#footnote-ref-15)
16. Transport Layer Security [↑](#footnote-ref-16)
17. Christian Artigues, Sophie Demassey & Emmanuel Neron, “Resource-Constrained Project Scheduling: *Models, Algorithms, Extension and Applications*”, ISTE, 2008, pag. 27 [↑](#footnote-ref-17)
18. Interactiune Om-Calculator [↑](#footnote-ref-18)