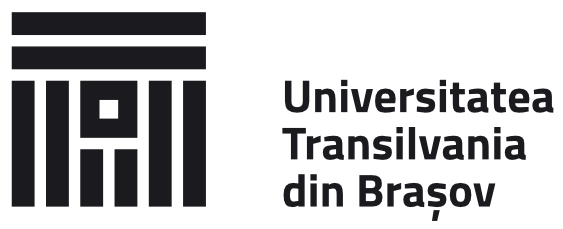
****

**Aplicație suport pentru realizarea orarului universității**

**Coordonator**

**Lect. Dr. Ioana Plajer**

**Student**

**Szabó Pál-Szabolcs**

**Brașov, 2022**

**CUPRINS**

**ÎNTRODUCERE ………………………………………………………………………………………………… 3**

**CAPITOLUL I. TEHNOLOGII FOLOSITE ………………………………………………………………. 5**

**I.1. JAVA ……………………………………………………………………………………………… 5**

**I.2. JavaFX …………………………………………………………………………………………… 8**

**I.3. MAVEN …………………………………………………………………………………………. 9**

**I.4. GSON ……………………………………………………………………………………………. 10**

**I.5. Apache POI …………………………………………………………………………………… 10**

**I.6. ControlsFX ……………………………………………………………………………………. 11**

**CAPITOLUL II. CLASELE APLICAȚIEI …………………………………………………………………. 12**

**II.1. Clasa Activity ………………………………………………………………………………. 12**

**II.2. Clasa Professor ……………………………………………………………………………. 14**

**II.3. Clasa Group …………………………………………………………………………………. 17**

**II.4. Clasa Room …………………………………………………………………………………. 19**

**II.5. Clasa MainMenu …………………………………………………………………………. 21**

**II.6. Clasa IndexedLabel ……………………………………………………………………… 23**

**II.7. Clasa Utility …………………………………………………………………………………. 24**

**II.7.1. Metoda readXls …………………………………………………………….. 24**

**II.7.2. Metoda readFile ……………………………………………………………. 30**

**II.7.3. Metoda writeXls ……………………………………………………………. 31**

**II.7.4. Metoda saveFile ……………………………………………………………. 31**

**II.7.5. Metoda addIfNotInGroups ……………………………………………. 32**

**II.7.6. Metoda addIfNotInProfs ……………………………………………….. 32**

**II.7.7. Metoda createActivities ……………………………………………….. 33**

**II.7.8. Metoda saveData …………………………………………………………. 35**

**II.7.9. Metodele load ……………………………………………………………… 36**

**II.8. Clasa Scenes ……………………………………………………………………………….. 37**

**II.8.1. Metoda professorsScheduleScene ................................... 38**

**II.8.2. Metoda addDropHandlingProfSchedule ………………………… 41**

**II.8.3. Metoda isMovableToProfSchedule ……………………………….. 42**

**II.8.4. Metoda moveToProfSchedule ………………………………………. 43**

**CAPITOLUL III. APLICAȚIA AJUTORUL ORARISTULUI - GHID DE UTILIZARE ………. 46**

**INTRODUCERE**

Realizarea orarului este una dintre cele mai grele sarcini care apar în cadrul organizării funcționării unei unități de învățământ. De obicei orarul se realizează la începutul anului de învățământ, respectiv la începutul unu semestru în cazul facultăților, de către unul sau mai mulți membrii ai corpului didactic și presupune un efort destul de mare ca rezultatul să fie satisfăcător pentru toți cei implicați, adică cadre didactice și studenți.

La realizarea orarului trebuie avute în vedere foarte multe restricții ca de exemplu: un profesor nu poate avea ore concomitent cu mai multe grupe de studenți, este de preferat ca nici profesorii nici studenții să nu aibă mai multe ferestre pe zi, ideal ar fi să nu existe ferestre deloc, dar de obicei este foarte greu să obținem un astfel de rezultat. În cazul școlilor ferestrele pentru elevi sunt de obicei evitate pentru că elevii fiind minori vor trebui supravegheați oricum de cineva (învățător, profesor, educator etc. ).

Aplicația ”Ajutorul oraristului” nu își propune să suplinească oraristul, ci va încerca să ofere un sprijin eficace pentru realizarea orarului facultăților din cadrul Universității Transilvania din Brașov. Aplicația citește datele necesare din fișierul excel al statelor de funcțiuni ale facultății în cauză și va genera activitățile care rezultă (adică cursuri, seminare, laboratoare, activități practice) din fișier. Aceste activități vor putea fi plasate, mutate de către orarist între diferite tabele, orare ale profesorilor, grupelor, anilor etc. Aplicația semnalează suprapunerile care apar pentru profesori sau grupe și nu permite mutarea unei activități dacă profesorul sau grupa (grupele) în cauză au deja alte activități atribuite pentru perioada de timp respectivă. După alcătuirea orarului, aplicația poate transcrie automat datele într-un fișier excel în format similar cu cel standard al Universității Transilvania din acest moment, putând fi copiate în tabelul orar al Universității. Am speranța că aplicația poate oferi un real ajutor celor care sunt însărcinați cu elaborarea orarului facultăților Universității Transilvania, fiind o aplicație personalizată care se folosește de fișierele deja existente care conțin toate informațiile necesare pentru a genera activitățile curente ce trebui să apară în orarul final al facultăților. În această primă variantă aplicația va putea citi din fișierul statelor de funcțiuni orele profesorilor aparținând unei anumite facultăți și va genera activitățile acestora, care se desfășoară în cadrul aceleași facultăți, de exemplu pentru Facultatea de Matematică și Informatică va citi toate orele profesorilor din cadrul facultății, pe care aceștia le au cu grupele acestei facultăți.

În mediul online există multe aplicații de realizare de orar, desigur nici unul adaptat necesităților specifice ale Universității Transilvania. Aceste aplicații sunt de mai multe feluri, cele mai multe sunt cele de editare a unui orar personal, care de fapt este un fel de calendar care trimite atenționări, remindere pentru titular, dar de fapt nu au nici o legătură cu elaborarea unui orar ai unei unități de învățământ. Dacă ne referim strict la aplicații de elaborare automată a unui orar am găsit și aici câteva alternative cum ar fi

* asc Orare - <https://ascorare.ro/>
* docendo Zen - <https://zen.docendo.co/>
* GHC School timetable maker - <https://www.penalara.com/en/RO>

Toate acestea oferă elaborarea automată a orarului și, desigur, și editarea manuală a acestuia însă chiar dacă aplicația noastră nu oferă elaborarea automată a orarului, ci doar oferă suport celui care se ocupă de elaborarea orarului, marele avantaj al aplicației Ajutorul Oraristului constă în faptul că, aceasta fiind personalizat, introducerea datelor este de fapt un proces automatizat și faptul că oraristul are libertate în alcătuirea orarelor profesorilor sau grupelor îi dă acestuia posibilitatea să țină cont de o multitudine de restricții, cerințe, doleanțe venite atât din partea profesorilor, cât și datorate specificului acestei unități de învățământ. Totodată există și anumite situații speciale care pot fi luate în calcul de către orarist în momentul în care elaborează orarul și chiar dacă toate aplicațiile enumerate mai sus acceptă condiționări în ceea ce privește elaborarea automată a orarului, probabil că introducerea manuală a tuturor acestor restricții și condiții reprezintă un efort destul de mare și nici nu este garantată introducerea tuturor acestora. În cazul elaborării manuale a orarului cu ajutorul aplicației prezentate în această lucrare, restricțiile și condițiile impuse pot fi mult mai bine gestionate de către orarist.

Aplicația este concepută pentru a oferi asistență și nu pentru a elabora orarul în mod automat, cum se întâmplă în cazul aplicațiilor enumerate anterior dar, așa cum am mai precizat, sunt convins că poate oferi un real sprijin și sper că va putea fi chiar folosit de cei care vor elabora orarele facultăților de la Universitate. Acest real folos și speranța de a crea ceva util justifică alegerea temei lucrării. Această primă variantă nu este perfectă și, în urma utilizării practice, poate perfecționată și adaptată în funcție de necesități.

Capitolul I descrie tehnologiile folosite în elaborarea aplicației, limbajul de programare Java și Maven, platforma pentru interfața grafică JavaFX, bibliotecile utilizate pentru salvarea datelor (GSON), pentru citirea/scrierea fișierelor Excel (POI) etc.

Capitolul II prezintă o descriere a aplicației, a claselor folosite, metodelor implementate etc.

Capitolul III reprezintă un manual de utilizare a aplicației.

**CAPITOLUL I.**

**TEHNOLOGII FOLOSITE**

Capitolul oferă o descriere a tehnologiilor, limbajelor de programare și bibliotecilor folosite pentru a asigura cele mai bune soluții pentru realizarea aplicației. Codul este scris 100% în Java 1.8, folosește JavaFX pentru interfața grafică, Maven pentru bibliotecile GSON, Apache POI și ControlsFX folosite pentru salvare/citire din baza de date proprie Json, respectiv din fișierele Excel ale Universității, și o bibliotecă de completare pentru JavaFX care oferă clase folositoare pentru realizarea unei aplicații cât mai ”user-friendly”.

**I.1. JAVA**

James Gosling a fost pionierul Java în iunie 1991 cu proiectul numit „Oak”. Gosling și-a propus să dezvolte o mașină virtuală și un limbaj cu o notație bine-cunoscută precum C, dar cu mai multă precizie și simplitate decât C/C++. În 1995, Java 1.0 a fost prima versiune publică. Era foarte sigură și configurabilă, cu securitate care restricționa accesul la rețea și la fișiere. Într-o configurație „applet” stabilă, browserele web semnificative l-au implementat în curând în setările lor standard. În 1997 Sun a contactat ISO/IEC JTC1 și apoi Ecma International pentru a oficializa Java, dar s-au retras rapid. Java continuă să fie un standard proprietar de facto reglementat de Procesul comunității Java. Cu veniturile generate de noua viziune, cum ar fi Java Enterprise Framework, Sun a realizat mai multe implementări Java gratuite. Diferența critică este că compilatorul nu este prezent în JRE, ceea ce face diferența între Kitul de dezvoltare software (SDK) și JRE (JRE). Pe 13 noiembrie 2006, Sun a lansat o cantitate considerabilă de Java în Licența publică generală GNU ca software gratuit și open-source (GPL) (GPL). La 8 mai 2007 Sun a finalizat procesul prin lansarea unui cod de bază Java complet accesibil, gratuit și open-source, cu excepția unei mici porțiuni a codului pentru care Sun nu a deținut drepturi de autor.

Dezvoltarea limbajului Java a avut cinci obiective principale:

* Abordare orientată pe obiect.
* Independență de platformă
* Suport încorporat pentru rețele de calculator.
* Executarea codului din surse îndepărtate în siguranță.
* Simplitate de utilizat.

Orientarea pe obiecte

Principiul distinctiv critic pentru limbajele orientate pe obiecte este proiectarea programului pentru a combina diferite tipuri de date, pe care le manipulează cu operațiunile lor proprii. Datele și codul sunt apoi combinate în entități cunoscute sub numele de obiecte.

O entitate poate fi considerată ca entitate (cod) și stare de conduită autonomă (date). Ideea este de a separa elementele care se schimbă de lucrurile care rămân constante, o modificare a oricărei structuri de date înseamnă determină de asemenea, o modificare corespunzătoare a codului care rulează pe acele date sau invers. Această împărțire în obiecte compatibile oferă o bază mai fiabilă pentru proiectarea unui sistem software. Scopul este de a promova managementul proiectelor software mari, de a crește calitatea și de a reduce numărul de proiecte eșuate.

Un alt obiectiv principal al OO este de a crea artefacte comune, astfel încât codul să poată fi reutilizat între proiecte. De exemplu, un obiect generic „client” ar trebui să aibă în esență același set de bază de comportamente, mai ales atunci când aceste proiecte se suprapun la un anumit nivel fundamental, așa cum fac adesea în cadrul organizațiilor mai mari. Sperăm că, în acest sens, obiectele software pot fi tratate mai mult ca niște componente plug-in, astfel încât industria software-ului să poată construi în primul rând proiecte bazate pe piese actuale, bine testate, reducând astfel semnificativ timpul de dezvoltare. Reutilizarea codului a obținut rezultate funcționale mixte, cu două dificultăți principale: conceptul și abordarea de comunicare a obiectelor cu adevărat generice este prost înțeleasă.

Independența de platformă

A doua caracteristică, independența de platformă, înseamnă că programele scrise în Java trebuie să ruleze în mod similar pe hardware diferit, astfel încât să fie posibil de a le executa de oriunde. Majoritatea compilatoarelor Java îndeplinesc această sarcină compilând „la jumătatea drumului” Java către codul bytecode, în special bytecode Java, instrucțiuni simplificate ale mașinii specifice platformei Java. Codul rulează apoi pe o mașină virtuală care interpretează și execută cod generic de octeți Java în cod nativ pe hardware-ul gazdă. De asemenea, biblioteci standardizate sunt disponibile pentru a permite accesul unificat la caracteristicile mașinilor gazdă, cum ar fi grafica, fire de execuție și rețea, pentru a enumera câteva dintre acestea. Se poate observa că, deși etapa de compilare este precisă, undeva bytecode-ul Java este interpretat sau convertit de compilatorul JIT în instrucțiuni native ale mașinii. Compilatoarele Java sunt, de asemenea, implementate pentru a compila codul obiect nativ, cum ar fi GCJ, pentru a elimina nivelul intermediar de bytecode.

Licența Java a Sun necesită toate implementările „compatibile”. Acest fapt a dus la o dispută legală cu Microsoft după ce Sun a susținut că interfețele RMI și JNI nu au acceptat implementarea Microsoft și au introdus caracteristici specifice platformei. Ca urmare Microsoft nu mai livrează Java cu Windows, iar Internet Explorer nu mai suportă applet-uri Java fără un plugin în versiunile recente de Windows. Cu toate acestea, Sun și mai târziu Oracle au pus la dispoziție sistemele de rulare Java gratuit. Primele implementări de limbaj au folosit o mașină virtuală codificată pentru portabilitate. Aceste inovații au determinat crearea unor programe care au funcționat puțin mai lent decât cele compilate pe executabile native, cum ar fi C sau C++, astfel încât limbajul a fost considerat lent și nefolositor. Implementările JVM mai recente generează programe cu mai multe tehnici care funcționează mult mai rapid decât înainte. Prima tehnică este de a compila codul nativ direct într-un compilator mai convențional, care omite complet codurile de octet. Acest lucru are duce la rezultate bune, dar cu prețul portabilității. Codurile de octet Java sunt convertite în cod nativ ca o tehnică diferită cunoscută sub numele de compilare just-in-time (JIT) atunci când programul rulează. Programul rulează mai repede decât codul interpretat și este, de asemenea, compilat în timpul execuției. VM-urile mai avansate folosesc recompilarea dinamică în care VM-ul poate să recompileze și să optimizeze selectiv componentele esențiale ale software-ului, să evalueze comportamentele. Recompilarea dinamică poate face mai mult decât optimizări statice, deoarece compilatorul dinamic poate folosi mediul de rulare și colecția de clase încărcate pentru a construi optimizări. Reconstrucția dinamică și compilarea JIT permit programelor Java să beneficieze de viteza codului nativ fără a pierde portabilitatea.

Gestionarea automată a memoriei – *Garbage collector*

Un concept din spatele modelului de gestionare automată a memoriei Java este acela de a scuti programatorii de stresul gestionării manuale a memoriei. În anumite limbaje, programatorul alocă memorie pentru a construi orice obiect stocat la heap și apoi gestionează memoria manual pentru a elimina acele obiecte. O scurgere de memorie poate apărea atunci când un programator uită să mute memoria și nu introduce codul prompt: consumă o cantitate potențial mare de memorie. De altfel există și o anumită suprasarcină și sofisticare a codului de utilizator pentru a urmări și finaliza alocările în medii care nu dispun de garbage collector. În Java garabge collector-ul evită această posibilă problemă a eliberării memoriei care nu mai este folosită. Crearea obiectelor este decizia programatorului, iar Java este responsabil pentru gestionarea ciclului de viață al obiectului. Garbage collector-ul Java șterge automat un obiect inaccesibil dacă nu rămâne nici o referință acest obiect, eliberând memorie și evitând pierderea memoriei. Pot exista, de asemenea, scurgeri de memorie dacă în codul unui programator se găsește o referință la un element care nu mai este necesar - cu alte cuvinte, poate apărea în continuare la un nivel conceptual superior.

În prezent, Java face parte din primele cinci limbaje de programare cele mai frecvent utilizate din lume, potrivit Statista. Unii dezvoltatori tind să folosească Java pentru proiectarea aplicațiilor GUI, în timp ce alții folosesc Java pentru a construi aplicații web. Java este, de asemenea, utilizat pe scară largă în producția de jocuri pentru dispozitive mobile și Android, cea mai extinsă platformă mobilă. Iată câteva dintre motivele pentru care Java este și va rămâne important pe termen lung.

Avantaje ale limbajului Java

Java este un limbaj sofisticat și stabil pentru programare. Cu toate acestea, Oracle Corporation actualizează limbajul de programare în mod regulat cu ajutorul unei comunități dinamice. Fiecare versiune Java nouă are multe caracteristici noi și îmbunătățiri de performanță.

Programatorii trebuie să scrie aplicații folosind mai multe dispozitive și platforme. Prin urmare, ei caută un limbaj de programare care să le permită să scrie codul aplicației odată și să utilizeze codul aplicației pe mai multe platforme fără nici un efort suplimentar. Programatorii pot compila pur și simplu codul Java odată și îl pot implementa pe diverse platforme fără a recompila codul. Bytecode-ul permite codul aplicației implementat de programatori pe orice platformă care suportă Java. De asemenea, programatorii pot transporta rapid aplicația de pe o platformă pe alta fără a compila constant codul.

Regulile de sintaxă ale Java sunt similare cu sintaxa C și C++. Astfel, este mai ușor pentru începători să învețe și să folosească Java în timp destul de scurt. În același timp, Java este un limbaj de programare bazat pe clase, deci orientat pe obiecte. Deoarece Java îmbrățișează principiile populare de programare orientată pe obiecte, cum ar fi moștenirea, polimorfismul și abstracția, aplicațiile sunt făcute modulare, extensibile și scalabile, mai ușor accesibile pentru programatori. Dezvoltatorii pot beneficia, de asemenea, de astfel de biblioteci Java pentru a încorpora mai eficient conceptele de design orientate pe obiecte.

În timpul dezvoltării aplicațiilor mobile, dispozitivul mobil cu cea mai extinsă bază de instalare nu poate fi trecut cu vederea de niciun dezvoltator. Programatorii pot scrie aplicații Android în C, C++ și Java. Google însă le recomandă dezvoltatorilor de aplicații mobile să scrie aplicații Android în Java, îmbunătățind astfel în mod semnificativ performanța și compatibilitatea aplicațiilor Android. Dezvoltatorii au, de asemenea, opțiunea de a scrie aplicații Android robuste în Java într-un timp mai scurt, folosind diverse instrumente și biblioteci.

Java domină alte limbaje la categoria interfețe de programare bogate în aplicații (API). Programatorii au opțiunea de a crea proiecte de dezvoltare populare folosind o varietate de API-uri Java fără a adăuga niciun cod suplimentar. Unele dintre aceste API-uri sunt partajate de corporații importante, iar altele sunt încărcate de membrii comunității. Dezvoltatorii pot folosi API-uri pentru a lega baze de date, intrări și ieșiri, rețele, utilități, protecție și analiza XML în funcție de necesitățile lor. Ei pot combina aceste API-uri cu alte biblioteci Java Open Source pentru a îmbunătăți funcționalitatea și eficiența aplicațiilor, fără a adăuga timp și efort.

În general, un dezvoltator poate folosi Java pentru a scrie aplicații GUI, mobile și web pentru aplicații desktop. Simplitatea și versatilitatea Java îl fac, în sens real, un limbaj de programare general. Simultan, cadrele și instrumentele de programare Java fac din Java unul dintre cele mai utilizate limbaje de programare.

**I.2. JavaFX**

JavaFX este proiectat pentru a oferi dezvoltatorilor Java o nouă platformă grafică ușoară și performantă. Intenția este ca noile aplicații să utilizeze JavaFX mai degrabă decât Swing pentru a construi interfața grafică (GUI) a aplicației. Dat fiind faptul că foarte multe aplicații sunt concepute pe Swing JavaFX este conceput astfel încât cele două API-uri grafice se execută una lângă cealaltă. JavaFX poate fi folosit pentru a crea interfețe grafice pentru utilizatori pentru orice platformă (de ex. Desktop, web, mobil etc.).

Inițial, accentul pentru platforma JavaFX a fost în principal pentru aplicațiile de internet bogate (RIA). A existat un limbaj de scripting JavaFX destinat să faciliteze crearea unei interfețe web. În timpul vieții timpurii a JavaFX nu a fost niciodată foarte clar dacă JavaFX va înlocui în cele din urmă Swing. După ce Oracle a preluat conducerea Java de la Sun, focusul a fost mutat pentru a face JavaFX platforma grafică preferată pentru toate tipurile de aplicații Java. Versiunile JavaFX 1.x nu mai sunt disponibile din 2013 și este recomandat ca orice aplicații de producție JavaFX 1.x să fie migrată spre JavaFX 2.0, care a fost lansat în octombrie 2011. Acest lucru a semnalat sfârșitul limbajului de scripting JavaFX și mutarea funcției JavaFX într-un Java API. Astfel dezvoltatorii Java nu trebuie să învețe un nou limbaj grafic ci pot să creeze aplicații JavaFX utilizând sintaxa obișnuită Java. API-ul JavaFX conține tot ce se așteptă de la o platformă grafică - controale UI, animații, efecte etc. Principala diferență pentru dezvoltatorii care trec de la Swing la JavaFX va fi obișnuită cu modul în care sunt prezentate componentele grafice și noua terminologie. O interfață de utilizator este încă construită folosind o serie de straturi care sunt cuprinse într-un grafic de scenă.

**I.3. MAVEN**

Maven este un instrument de gestionare și compilare a proiectelor software utilizat în principal cu proiecte bazate pe Java, dar care poate fi folosit și pentru gestionarea proiectelor în alte limbaje de programare precum C # și Ruby. Maven ajută la gestionarea compilărilor, documentației, raportării, dependințelor, gestionării configurației software (SCM), lansărilor și distribuției. Multe medii de dezvoltare integrate (IDE) furnizează plug-in-uri sau suplimente pentru Maven, ceea ce permite Maven să compileze proiecte din cadrul IDE. Denumirea de Maven este un cuvânt în limba idiș și înseamnă ”culegător de cunoștințe”.

Maven a fost creat de Jason Van Zyl în 2002 ca parte a proiectului Apache Turbine. A devenit un proiect Apache Software Foundation în 2003. După aceea, au fost lansate mai multe versiuni ale Maven, inclusiv Maven v1.0, v2.0 și v3.0.

Unitatea fundamentală în Maven este POM (Project Object Model), un fișier XML care include informații despre proiectul software, detalii de configurare pe care Maven le folosește la construirea acestui proiect, orice dependințe de componente sau module externe și ordinea de construire. Funcționalitatea Maven depinde și de plug-in-uri, care oferă un set de obiective care pot fi executate. De fapt, toate lucrările sunt gestionate de pluginuri. Există numeroase plug-in-uri Maven pentru construire, testare, SCM, rularea unui server Web etc. Plug-in-urile sunt configurate în fișierul POM, unde unele plugin-uri de bază sunt incluse în mod implicit.

Caracteristicile cheie ale Maven includ:

* Un mod simplu și ușor de a construi proiecte în care sunt ascunse detalii inutile
* Un sistem de construire uniform, în care este urmată o strategie standard la construirea oricărui proiect
* Informații de proiect de calitate, cum ar fi listele de dependință, surse cu referință încrucișată și rapoarte de unity testing
* Managementul dependințelor, inclusiv actualizarea automată și închiderea dependenței
* Posibilitatea de a gestiona mai multe proiecte simultan
* Descărcarea dinamică a bibliotecilor și plug-in-urilor Java necesare din depozitele Maven

În cazul nostru Maven este folosit în primul rând pentru gestionarea dependințelor, plug-in-urilor GSON și POI pe care le folosim pentru operațiunile de salvare/citire de date intermediare și finale.

**I.4. GSON**

Gson este o bibliotecă Java care poate fi folosită pentru a converti obiectele Java în reprezentarea lor JSON. Poate fi folosită și pentru a converti un șir JSON într-un obiect Java echivalent. Gson poate lucra cu obiecte Java arbitrare, inclusiv obiecte preexistente pentru care nu există cod sursă.

Gson furnizează mecanisme ușor de utilizat, cum ar fi toString() și constructor pentru a converti Java în JSON și invers. Permite ca obiectele preexistente nemodificabile să fie convertite în și din JSON. Permite reprezentări personalizate pentru obiecte. Sprijină obiecte arbitrar complexe. Generează o ieșire JSON compactă și lizibilă.

Biblioteca Gson a fost creată inițial pentru a fi utilizată în interiorul Google, unde este în prezent folosită într-o serie de proiecte. Este de asemenea folosită de o serie de proiecte publice și companii.

În proiectul de față biblioteca Gson este folosită pentru a salva obiectele claselor de bază ale aplicației în format JSON, în fazele intermediare a elaborării orarului, pentru a se putea relua lucrul într-o altă sesiune, când tot cu ajutorul Gson obiectele se vor citi și se vor reconverti din JSON în obiectele claselor aplicației.

**I.5. Apache POI**

POI este cea de a doua biblioteca de conversie folosită pentru a face legătura cu fișierele excel ale universității, un fișier de intrare care conține toate informațiile necesare privind profesorii, grupele, activitățile din cadrul unei facultăți și fișierele de ieșire care vor conține orarul finalizat.

Misiunea proiectului Apache POI este de a crea și menține API-uri Java pentru manipularea diferitelor formate de fișiere bazate pe standardele Office Open XML (OOXML) și formatul Microsoft OLE 2 Compound Document (OLE2). Pe scurt, puteți citi și scrie fișiere MS Excel folosind Java. În plus, puteți citi și scrie fișiere MS Word și MS PowerPoint folosind Java. Apache POI este soluția Java Excel (pentru Excel 97-2008).

POI are o interfață API completă pentru portarea altor formate OOXML și OLE2. Fișierele OLE2 includ majoritatea fișierelor Microsoft Office, cum ar fi XLS, DOC și PPT, precum și formate de fișiere bazate pe API de serializare MFC. Proiectul oferă API-uri pentru sistemul de fișiere OLE2 (POIFS) și proprietățile documentului OLE2 (HPSF).

Office OpenXML Format este noul format de fișier XML bazat pe standarde găsit în Microsoft Office 2007 și 2008. Acesta include XLSX, DOCX și PPTX. Proiectul oferă un API de nivel scăzut pentru a sprijini convențiile de ambalare deschisă folosind openxml4j. Pentru fiecare aplicație MS Office există un modul de componentă care încearcă să ofere un API Java comun de nivel înalt atât pentru formatele de document OLE2, cât și pentru OOXML.

Acesta este cel mai dezvoltat pentru registrele de lucru Excel (SS=HSSF+XSSF) și este folosit și în cazul aplicației prezentate în lucrarea de față.

**I.6. ControlsFX**

ControlsFX este o bibliotecă de instrumente grafice pentru JavaFX. Biblioteca conține numeroase clase, cele mai multe foarte folositoare și binevenite.

În cazul de față este folosită clasa Searchable Combobox, o variantă modificată/completată a comboboxului din JavaFX, care permite căutarea în lista aferentă comboboxului, prin introducerea unui text în câmpul editabil. În momentul introducerii textului se face o filtrare a elementelor din listă și se vor afișa elementele care conțin șirul de caractere introdus în câmpul editabil.

**CAPITOLUL II**

**CLASELE APLICAȚIEI**

După cum se poate deduce din capitolul anterior, aplicația este realizată în Java, fiind o aplicație JavaFX cu suport Maven, folosind bibliotecile Gson și Apache POI pentru realizarea operațiunilor de citire/scriere a diverselor documente.

**II.1. Clasa Activity**

Clasa ”Activity.java” poate fi considerată clasa de bază a aplicației, obiectele instanțiate ale clasei reprezintă activitățile care trebuie să se desfășoare în cadrul facultății în anul și semestrul pentru care se va elabora orarul, adică cursuri, seminare, ore de laborator sau activități practice. Obiectele se vor instanția pe baza fișierului excel care conține statul de funcțiuni al facultății în cauză sau se vor citi dintr-un fișier salvat de către orarist într-o anumită fază de construcție a orarului.

Clasa are următoarele **atribute**:

* int **idActivity** – este indicele, ID-ul unic al activității, prin care activitatea poate fi identificată cel mai rapid și simplu. Vom vedea în continuare că activitățile sunt identificate de alte clase prin acest ID
* String **subject** – titlul activității cum apare în statul de funcțiuni, de fapt materia care constituie subiectul activității
* String **codeSubject** – este o variantă prescurtată a subiectului activității, se citește de asemenea din același fișier. Această prescurtare se folosește de obicei în afișarea grafică a activităților, din cauză că subiectul unei activități poate fi foarte lung și nu poate fi afișat într-o formă grafică care să permită afișarea grafică a cât mai multor activități în niște tabele de orar global sau personal, de exemplu materia ”Algebră liniară, geometrie analitică și diferențială” are codul ”AG03”. Aceste coduri apar în statul de funcțiuni și pe baza lor materiile sunt ușor de identificat.
* int **professorId** – reprezintă ID-ul profesorului căruia îi este atribuită activitatea, adică cel care va ”ține ora” respectivă. Ca o mică paranteză vreau să precizez că inițial aici am avut chiar un obiect Professor, însă din această cauză a apărut o relație bidirecțională între cele două obiecte, adică clasa Activity avea un atribut Professor, iar obiectul Professor avea o matrice și un vector de obiecte Activity, ceea ce în momentul stocării prin GSON a dus la o cantitate de date nefondat de mare, astfel am ajuns la concluzia că mai bine introduc aceste atribute de identificare, adică ID-urile, prin care obiectele pot fi identificate mult mai simplu și rapid
* int **type** – tipul activității. Am identificat 4 tipuri de activități în statele de funcțiuni ale Universității și anume Curs, Seminar, Laborator, Practică. Tipul activității este o variabilă de tip număr întreg care este în concordanță cu tipul adică: 1 – CURS, 2 – SEMINAR, 3 – LABORATOR, 4 – ACTIVITATE PRACTICĂ.
* int[] **groupsId** – este un array de numere de identificare a grupelor cărora li se adresează activitatea respectivă. Aici este nevoie de o listă pentru că, de obicei, cursurile sunt pentru mai multe grupe, iar celelalte activități se desfășoară de obicei pe grupe, dar în primul caz o singură activitate trebuie să aibă mai multe grupe. Acest ID, ca și în cazul professorId, există un ID unic pentru fiecare grupă din cadrul facultății
* int **semester** – reprezintă numărul semestrului în care se va desfășura activitatea
* int **yearOfStudy** – anul de studiu căruia i se adresează activitatea. Deși acesta s-ar putea deduce și din grupele arondate am considerat necesar un atribut separat pentru o mai ușoară și mai directă identificare a anului de studiu ale activităților
* int **time** – durata activității, este de obicei de două ore, dar apar și activități de o oră sau trei ore sau activități bisăptămânale care sunt tratate de fapt ca activități de o oră putând fi alterate cu altă activitate în același interval de orar de două ore ca și câte o activitate de o oră sau câte o activitate de două ore bisăptămânale, care alternează din două în două săptămâni
* boolean **weekly** – se referă exact la cele amintite despre activitățile bisăptămânale, acesta este un atribut boolean care are valoarea true dacă activitatea se repetă în mod normal, adică săptămânal și este false dacă activitatea se repetă din două în două săptămâni
* int **classRoomId** – este ID-ul sălii de clasă în care se va desfășura activitatea, un ID unic. Sălile de cursuri nu apar în fișierul de state de funcțiuni, deci acestea vor trebui introduse manual de către orarist, prin selectarea meniului de adăugare sală de curs. Sălile de curs vor fi stocate într-o listă și alegerea acestora se va face în momentul în care o activitate va fi plasată în orar. Este de fapt singurul atribut care nu este final, toate celelalte sunt atribuite odată cu instanțierea fiecărui obiect prin constructorul unic despre care vorbim în continuare. De fapt mai există un atribut care poate fi modificat, este vorba de lista de grupe, însă de fapt acesta se va modifica în anumite cazuri în faza de citire a datelor din fișierul de sursă (state de funcțiuni) dar după această fază nu se mai modifică.

Clasa are un singur **constructor** care are forma:

**public** Activity(**int** idActivity, String subject, String codeSubject, **int** professorId, **int** tip, **int**[] groupsId, **int** semester, **int** yearOfStudy, **int** time, **boolean** weekly){  
 **this**.**idActivity**=idActivity;  
 **this**.**subject** = subject;  
 **this**.**codeSubject** = codeSubject;  
 **this**.**professorId** = professorId;  
 **this**.**type** = tip;  
 **this**.**groupsId** = groupsId;  
 **this**.**semester** = semester;  
 **this**.**yearOfStudy** = yearOfStudy;  
 **this**.**time** = time;  
 **this**.**weekly** = weekly;  
 **this**.**classRoomId** =-1;  
}

Se poate observa că prin acest constructor sunt determinate toate atributele, inclusiv classRoomId care primește valoarea negativă care va însemna de fapt că această activitate nu are încă o sală alocată.

**Metodele** clasei:

* **public void** addGroups(Group[] newGroups){  
   ArrayList<Integer> groupsToAdd=**new** ArrayList<>();  
   **for** (**int** idGroup : **groupsId**) {  
   groupsToAdd.add(idGroup);  
   **for** (Group newGroup : newGroups) {  
   **if** (newGroup.getIdGroup()==idGroup) {  
   newGroup.setGroupName(**""**);  
   }  
   }  
   }  
   **for** (Group group : newGroups) {  
   **if** (!group.getGroupName().equals(**""**)) {  
   groupsToAdd.add(group.getIdGroup());  
   }  
   }  
   **int**[] newGroup=**new int**[groupsToAdd.size()];  
   **for** (**int** i=0;i<newGroup.**length**;i++)  
   newGroup[i] = groupsToAdd.get(i);  
   **groupsId** = newGroup;  
  }

– o metodă prin care se adaugă grupe la lista de grupe ale activității. Această metodă se folosește pe parcursul operației de citire date / creare obiecte în cazul unor activități care au grupe din diverse specializări, în cazul acesta grupele din a doua (sau treia) specializare vor fi alocate mai târziu, după instanțierea obiectului.

* **public** String getTypeChar() {  
   String[] typeChar={**"C"**,**"S"**,**"L"**,**"P"**};  
   **try** {  
   **return** typeChar[**this**.**type**-1];  
   }  
   **catch** (Exception ex) {  
   **return "X"**;  
   }  
  }

– returnează tipul activității în format de caracter, adică ”C” pentru curs, ”S” pentru seminar, ”L” pentru laborator și ”P” în cazul activităților practice. În cazul în care tipul activității nu este nici unul dintre cele enumerate, caz teoretic inexistent, metoda returnează ”X”. Metoda se folosește în momentul creării etichetelor indexate (IndexedLabel) care constituie reprezentarea grafică a obiectelor Activity.

**II.2. Clasa Professor**

Clasa ”Professor.java” este clasa obiectelor de tip profesor. Membrii acestei clase sunt profesorii care sunt titularii activităților din cadrul facultății. Ca și în cazul activităților, obiectele se vor instanția pe baza fișierului excel care conține statul de funcțiuni ale facultății în cauză sau se vor citi dintr-un fișier salvat de către orarist într-o anumită fază de construcție a orarului.

Clasa are următoarele **atribute**:

* int **idProfessor** – este un identificator unic pentru fiecare obiect profesor instanțiat, acest ID este folosit de toate celelalte clase sau metode pentru identificarea profesorilor.
* String **name** – conține numele profesorului așa cum apare în fișierul sursă
* String **shortName** – este numele prescurtat al profesorului care se generează de către constructor. Această prescurtare se folosește de obicei în afișarea grafică a activităților, din cauză că numele complet poate fi prea lung și nu poate fi afișat într-o formă grafică care să permită afișarea grafică a cât mai multor activități în niște tabele de orar global sau personal.
* int[][][] **scheduleProfessor** – este o matrice tridimensională care conține, de fapt, orarul profesorului. Este matrice tridimensională deoarece conține orarul pe semestru / oră / zi. Avem 2 semestre, 7 intervale orare și 12 zile. Avem 12 zile pentru că de fapt avem câte o zi pentru săptămânile pare și cele impare. Numerele conținute sunt de fapt numere de identificare pentru activități, iar dacă numărul este -1 înseamnă că în ziua și ora respectivă profesorul nu are alocată nici o activitate.
* int[] **activitiesOfProfessor** - este un array care conține toate activitățile profesorului, activitățile fiind reprezentate și aici de ID-ul fiecăruia.

Unicul **constructor** al clasei arată în felul următor:

**public** Professor(**int** id, String name){  
 **final int** HOURS=7,DAYS=12;  
 **this**.**idProfessor** = id;  
 **this**.**name** = name;  
 String[] names=name.split(**" "**);  
 **this**.**shortName** = names[0].substring(0,1)+names[0].substring(1).toLowerCase();  
 **for** (**int** i=1;i<names.**length**;i++){  
 **this**.**shortName** += **"\_"** + names[i].substring(0, 1);  
 }  
 **this**.**scheduleProfessor** = **new int**[2][HOURS][DAYS];  
 **for**(**int** i=0;i<2;i++)  
 **for**(**int** j=0;j<HOURS;j++)  
 **for**(**int** k=0;k<DAYS;k++)  
 **scheduleProfessor**[i][j][k]=-1;  
 **activitiesOfProfessor** = **new int**[0];  
}

După cum se poate observa constructorul are doi parametri, iar valorile restului de atribute sunt generate de către constructor. **shortName** este generat din **name**, se generează orarul fără nici o activitate, adică avem numai valori de -1 în toate câmpurile matricei, iar array-ul de activități se inițiază fără elemente, acestea se vor adăuga în momentul citirii datelor de intrare din fișierul statelor de funcțiuni.

**Metodele** clasei:

* **public int** getActivityProfessor(**int** semester, **int** hour, **int** day) {  
   **try** {  
   **return scheduleProfessor**[semester-1][hour][day];  
   }  
   **catch** (Exception ex) {  
   **return** -1;  
   }  
  }

– returnează ID-ul activității din orarul profesorului din semestrul, ora, ziua primite ca parametri de intrare. În cazul în care valoarea returnată este -1 înseamnă că în semestrul dat, în ziua și la ora primite ca parametri nu există activitate, profesorul este liber.

* **public boolean** setActivityProfessor(**int** semester, **int** hour, **int** day, **int** activity) {  
   **try** {  
   **scheduleProfessor**[semester-1][hour][day]=activity;  
   **return true**;  
   }  
   **catch** (Exception ex){  
   Utility.*errorMessage*(**"Activitatea nu a fost adăugată"**);  
   **return false**;  
   }  
  }

– această metodă setează/schimbă activitatea profesorului din semestrul, ora, ziua, cu activitatea cu ID-ul primit ca parametru de intrare. Metoda returnează ”true” dacă această setare a fost reușită și returnează ”false” dacă, din orice motiv, operația a eșuat.

* **public boolean** addActivity (**int** activity) {  
   **for** (**int** act:**activitiesOfProfessor**) {  
   **if** (activity==act)  
   **return false**;  
   }  
   **int** size= **activitiesOfProfessor**.**length**;  
   **int**[] newActivites=**new int**[size+1];  
   System.*arraycopy*(**activitiesOfProfessor**, 0, newActivites, 0, size);  
   newActivites[size]=activity;  
   **activitiesOfProfessor** =**new int**[size+1];  
   System.*arraycopy*(newActivites, 0, **activitiesOfProfessor**, 0, size + 1);  
   **return true**;  
  }

– adaugă o activitate la array-ul de activități ale profesorului. Se returnează ”true” dacă elementul a fost adăugat și ”false” dacă nu s-a reușit adăugarea noului element. De exemplu, dacă o activitate este deja în lista profesorului, acesta nu se va adăuga din nou, în acest caz operațiunea nu reușește și se returnează ”false”, lista/array-ul activităților rămâne neschimbată.

* **public boolean** removeActivity (**int** activity) {  
   **int** size= **activitiesOfProfessor**.**length**;  
   **int**[] newActivities=**new int**[size-1];  
   **int** j=0;  
   **for** (**int** i=0;i<size;i++) {  
   **try** {  
   **if** (**activitiesOfProfessor**[i]!=activity) {  
   newActivities[j]= **activitiesOfProfessor**[i];  
   j++;  
   }  
   }  
   **catch** (Exception ex) {  
   Utility.*errorMessage*(**"Activitatea nu a fost ștearsă"**);  
   **return false**;  
   }  
   }  
   **activitiesOfProfessor** = **new int**[size-1];  
   **if** (size - 1 >= 0) System.*arraycopy*(newActivities, 0, **activitiesOfProfessor**, 0, size - 1);  
   **return true**;  
  }

– șterge o anumită activitate din array-ul activităților profesorului. Metoda returnează ”true” dacă operațiunea a reușit și ”false” în caz contrar.

**II.3. Clasa Group**

Clasa grupelor este clasa a căror obiecte instanțiate reprezintă grupele de la facultatea a cărei orar se elaborează. Grupele sunt instanțiate, ca și activitățile și profesorii, în momentul citirii și prelucrării datelor din statul de funcțiuni al facultății în cauză. Clasa Group este asemănătoare cu clasa Professor, însă sunt și anumite diferențe față de aceasta, dat fiind particularitățile celor două entități.

**Atributele** clasei Group sunt următoarele:

* int **idGroup** – numărul unic de identificare al grupei, număr care se folosește pentru identificarea grupei de către alte obiecte, metode etc.
* private String **groupName**  - numele grupei, care în primă instanță este generată în momentul citirii datelor și a generării obiectelor de tip Activity, Professor și Group. Numele generat al grupei se compune din codul specialității, anul de studiu și numărul grupei. De exemplu pentru grupa a doua din anul I. la specializarea (programul de studii) Matematică-Informatică numele generat este ”MI12”, adică Matematică-Informatică, anul 1, grupa 2.
* private int[][][] **scheduleGroup** – matricea orarului grupei. Este o matrice similară cu aceea din cazul profesorilor, care conține activitățile din orar pe pozițiile setate și -1 unde grupa nu are alocată nici o activitate
* private int[] **activitiesOfGroup** – array care conține toate activitățile alocate grupei.
* private String **speciality** – specializarea sau programul de studii al grupei
* private int **year** – anul de studiu
* private int **groupNumber** – numărul grupei din specializarea și anul date

**Constructorul** unic al clasei:

**public** Group(**int** id, String speciality, **int** year, **int** groupNumber){  
 **final int** HOURS = 7,DAYS = 12;  
 **this**.**idGroup**=id;  
 **this**.**groupName**=speciality+year+groupNumber;  
 **this**.**scheduleGroup** =**new int**[2][HOURS][DAYS];  
 **for**(**int** i=0;i<2;i++)  
 **for**(**int** j=0;j<HOURS;j++)  
 **for**(**int** k=0;k<DAYS;k++)  
 **scheduleGroup**[i][j][k]=-1;  
 **activitiesOfGroup** = **new int**[0];  
 **this**.**speciality**=speciality;  
 **this**.**year**=year;  
 **this**.**groupNumber**=groupNumber;  
}

Constructorul prezintă similarități cu cel al clasei Professor. Se generează numele grupei, nume care poate fi modificat de către orarist la un moment dat, dacă se dorește acest lucru. Matricea asemănătoare de orar se inițiază cu valorile de -1, însemnând că în semestrul, ziua și ora respectivă în orar nu este nici o activitate. Se inițializează array-ul d activități a grupei, activitățile se vor adăuga în cadrul procesului de citire/inițializare date.

**Metodele** clasei sunt următoarele:

* **public int** getActivityGroup(**int** semester,**int** hour,**int** day){  
   **try** {  
   **return scheduleGroup**[semester-1][hour][day];  
   }  
   **catch** (Exception ex) {  
   **return** -1;  
   }  
  }

– returnează ID-ul activității din orarul grupei din semestrul, ora și ziua specificate de parametrii transmiși. Dacă pe poziția respectivă nu este nici o activitate, atunci se va returna -1, valoare standard pentru orele fără activități

* **public boolean** setActivityGroup(**int** semester,**int** hour,**int** day, **int** activity) {  
   **try** {  
   **scheduleGroup**[semester-1][hour][day]=activity;  
   **return true**;  
   }  
   **catch** (Exception ex){  
   Utility.*errorMessage*(**"Activitatea nu a fost adăugată"**);  
   **return false**;  
   }  
  }

– setează o activitate pentru un anumit moment din orar. Metoda returnează ”true” dacă setarea s-a reușit și va returna false dacă operațiunea a eșuat

* **public boolean** addActivity (**int** activity) {  
   **for** (**int** act:**activitiesOfGroup**) {  
   **if** (activity==act)  
   **return false**;  
   }  
   **int** size=**activitiesOfGroup**.**length**;  
   **int**[] newActivities=**new int**[size+1];  
   System.*arraycopy*(**activitiesOfGroup**, 0, newActivities, 0, size);  
   newActivities[size]=activity;  
   **activitiesOfGroup**=**new int**[size+1];  
   System.*arraycopy*(newActivities, 0, **activitiesOfGroup**, 0, size + 1);  
   **return true**;  
  }

– adaugă un ID de activitate la array-ul de activități a grupei. Metoda returnează ”true” dacă operațiunea s-a realizat cu succes și returnează ”false” dacă nu s-a reușit adăugarea activității din diverse motive.

* **public boolean** removeActivity (**int** activity) {  
   **int** size=**activitiesOfGroup**.**length**;  
   **int**[] newActivities=**new int**[size-1];  
   **int** j=0;  
   **for** (**int** i=0;i<size;i++) {  
   **try** {  
   **if** (**activitiesOfGroup**[i]!=activity) {  
   newActivities[j]=**activitiesOfGroup**[i];  
   j++;  
   }  
   }  
   **catch** (Exception ex) {  
   Utility.*errorMessage*(**"Activitatea nu a fost ștearsă"**);  
   **return false**;  
   }  
   }  
   **activitiesOfGroup** = **new int**[size-1];  
   **if** (size - 1 >= 0) System.*arraycopy*(newActivities, 0, **activitiesOfGroup**, 0, size - 1);  
   **return true**;  
  }

- șterge o anumită activitate din array-ul activităților grupei. Metoda returnează ”true” dacă operațiunea a reușit și ”false” în caz contrar.

**II.4. Clasa Room**

Aceasta este clasa sălilor disponibile pentru desfășurarea activităților. Este de datoria oraristului să încarce manual lista sălilor disponibile înainte să se apuce de orar, pentru că va fi nevoie să aleagă și sala de desfășurare a activităților, în momentul când plasează o activitate pe orarul unui profesor sau pe orarul complet al grupelor.

**Atributele** claseisunt:

* int **roomId** – numărul unic de identificare al sălii, se folosește în relațiile cu alte clase
* String **roomName** – denumirea sălii, se va introduce de către orarist înainte de începerea elaborării efective a orarului
* Int[][][] **scheduleRoom** – este matricea orarului sălii, similară cu aceea din cazul profesorilor sau al grupelor, care conține activitățile din orar pe pozițiile setate și -1 unde sala nu are alocată nici o activitate

**Constructorul** clasei este:

**public** Room(**int** id, String name) {  
 **final int** HOURS = 7,DAYS = 12;  
 **this**.**roomId** = id;  
 **this**.**roomName** = name;  
 **for**(**int** i=0;i<2;i++)  
 **for**(**int** j=0;j<HOURS;j++)  
 **for**(**int** k=0;k<DAYS;k++)  
 **scheduleRoom**[i][j][k]=-1;  
}

Constructorul are parametri de intrare id-ul sălii și denumirea acesteia. În momentul instanțierii unui obiect Room constructorul construiește matricea de orar a sălii și o inițializează cu valori de -1, însemnând că nu există nici o activitate în sală.

**Metodele** clasei sunt

* **public int** getActivityRoom(**int** semester,**int** hour,**int** day){  
   **try** {  
   **return scheduleRoom**[semester-1][hour][day];  
   }  
   **catch** (Exception ex) {  
   **return** -1;  
   }  
  }

– returnează ID-ul activității din orarul grupei din semestrul, ora și ziua specificate de parametrii transmiși. Dacă pe poziția respectivă nu este nici o activitate atunci se va returna -1, valoare standard pentru orele fără activități

* **public boolean** setActivityRoom(**int** semester,**int** hour,**int** day, **int** activity) {  
   **try** {  
   **scheduleRoom**[semester-1][hour][day]=activity;  
   **return true**;  
   }  
   **catch** (Exception ex){  
   Utility.*message*(**"Activitatea nu a fost adăugată"**);  
   **return false**;  
   }  
  }

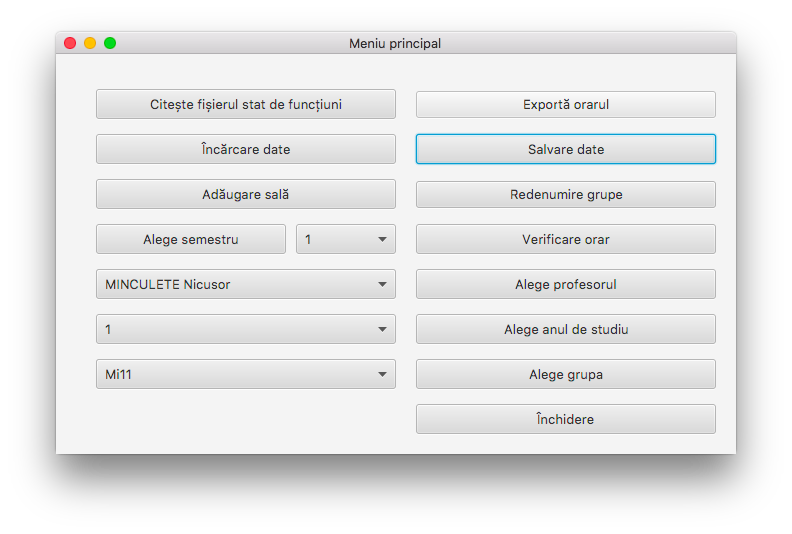
– setează o activitate pentru un anumit moment din orar. Metoda returnează ”true” dacă setarea s-a reușit și va returna false dacă operațiunea a eșuat

**II.5. Clasa MainMenu**

Este clasa meniului principal și din care se apelează toate funcționalitățile aplicației. Atributele clasei sunt:

* ArrayList<Activity> **activities** – lista tuturor activităților
* ArrayList<Professor> **professors** – lista profesorilor
* ArrayList<Group> **groups** – lista grupelor
* ArrayList<Room> **rooms** – lista sălilor de curs
* int **years** – reprezintă numărul anilor de studii
* Scenes **scenes** – instanțierea clasei Scenes, care conține elementele de interfață grafică

Constructorul clasei este cel implicit, fără parametri și instanțiază un obiect MainMenu. Singura metodă este **createMainMenu(),** care este răspunzătoare pentru crearea ferestrei meniului principal al aplicației și a funcționalităților aferente. Clasa este una d JavaFX și conține toate butoanele, comboboxurile, textele etc., care apar în meniul principal, care arată ca în figura de mai jos:



Codul acestei metode este:

**public void** createMainMenu() {  
  
 Stage mainStage=**new** Stage();  
 GridPane mainGrid=**new** GridPane();  
 mainGrid.setPadding(**new** Insets(20,20,20,20));  
 mainGrid.setAlignment(Pos.***CENTER***);  
 mainGrid.setVgap(15);  
 mainGrid.setHgap(20);

Scene mainScene=**new** Scene(mainGrid);  
  
 Button readFile=**new** Button(**"Citește fișierul stat de funcțiuni"**);  
 readFile.setPrefSize(300,30);  
  
 ComboBox<Integer> semesterCombo=**new** ComboBox<>();  
 semesterCombo.getItems().add(1);  
 semesterCombo.getItems().add(2);  
 semesterCombo.setPrefSize(100,30);  
  
 SearchableComboBox<String> profCombo=**new** SearchableComboBox<>();  
 profCombo.setPrefSize(300,30);  
  
 Button chooseProfesor=**new** Button(**"Alege profesorul"**);  
 chooseProfesor.setPrefSize(300,30);  
  
 SearchableComboBox<String> groupCombo=**new** SearchableComboBox<>();  
 groupCombo.setPrefSize(300,30);  
  
 Button chooseGroup=**new** Button(**"Alege grupa"**);  
 chooseGroup.setPrefSize(300,30);  
  
 Button saveData=**new** Button(**"Salvare date"**);  
 saveData.setPrefSize(300,30);  
  
 Button loadData=**new** Button(**"Încărcare date"**);  
 loadData.setPrefSize(300,30);  
  
 Button exit=**new** Button(**"Închidere"**);  
 exit.setPrefSize(300,30);  
  
 ComboBox<Integer> yearCombo=**new** ComboBox<>();  
 yearCombo.getItems().add(1);  
 yearCombo.setPrefSize(300,30);  
  
 Button chooseYear =**new** Button(**"Alege anul de studiu"**);  
 chooseYear.setPrefSize(300,30);  
  
 HBox semesterChooser = **new** HBox();  
 semesterChooser.setSpacing(10);  
  
 Button semesterText = **new** Button(**"Alege semestru"**);  
 semesterText.setPrefSize(190,30);  
  
 Button verificareOrar=**new** Button(**"Verificare orar"**);  
 verificareOrar.setPrefSize(300,30);  
  
 Button addRoom=**new** Button(**"Adăugare sală"**);  
 addRoom.setPrefSize(300,30);  
  
 Button nameGroups=**new** Button(**"Redenumire grupe"**);  
 nameGroups.setPrefSize(300,20);  
  
 Button writeFile=**new** Button(**"Exportă orarul"**);  
 writeFile.setPrefSize(300,20);

readFile.setOnAction(event -> { … });  
  
 writeFile.setOnAction(event -> { … });  
  
 saveData.setOnAction(event -> { … });  
  
 addRoom.setOnAction(event -> { …});  
 nameGroups.setOnAction(event -> { … });  
  
 chooseProfesor.setOnAction(event -> { … });  
  
 chooseGroup.setOnAction(event -> { …});  
  
 exit.setOnAction(event -> { … });  
  
 mainGrid.add(readFile,1,1);  
 mainGrid.add(writeFile,2,1);  
 mainGrid.add(loadData,1,2);  
 mainGrid.add(saveData,2,2);  
 mainGrid.add(addRoom,1,3);  
 mainGrid.add(nameGroups,2,3);  
 mainGrid.add(semesterChooser,1,4);  
 semesterChooser.getChildren().addAll(semesterText,semesterCombo);  
 mainGrid.add(verificareOrar,2,4);  
 mainGrid.add(profCombo,1,5);  
 mainGrid.add(chooseProfesor,2,5);  
 mainGrid.add(yearCombo,1,6);  
 mainGrid.add(chooseYear,2,6);  
 mainGrid.add(groupCombo,1,7);  
 mainGrid.add(chooseGroup,2,7);  
 mainGrid.add(exit,2,8);  
 mainStage.setScene(mainScene);  
 mainStage.setOnCloseRequest(Event::consume);  
 mainStage.setTitle(**"Meniu principal"**);  
 mainStage.show();  
}

După crearea elementelor de control JavaFX, cum ar fi butoanele, ComboBox-urile, SearchableComboBox-urile care vor fi afisate, în meniu sunt implementate comportamentele acestora în cazul apelării. Acestea vor apela metode din clasele Utility sau Scenes, sau o suită de metode pentru a obține rezultatul scontat.

**II.6. Clasa IndexedLabel**

Aceasta este o clasă care extinde clasa **Label** a JavaFX și este o etichetă (Label) care față de obiectele din clasa extinsă are 3 atribute și anume:

* int **activityId** – conține ID-ul activității a cărei reprezentare grafică este acest obiect
* int **professorId** – este ID-ul profesorului atașat activității
* int[] **groupsId** – un array cu ID-urile grupelor activității

**Constructorul** este unul simplu:

**public** IndexedLabel(**int** activityId, **int** professorId, **int**[] groupsId) {  
 **this**.**activityId** = activityId;  
 **this**.**professorId** = professorId;  
 **this**.**groupsId** = groupsId;  
}

De asemenea este adăugat și un constructor fără parametri.

**Metodele** sunt doar gettere și settere pentru cei trei parametri și, desigur, toate metodele moștenite de la clasa Label.

**II.7. Clasa Utility**

Este o clasă de utilități care are multe metode statice utilitare, adică de ajutor, și metodele de input/output ale aplicației. Clasa nu este folosită ca și clasă, de fapt nici nu se poate instanția, este mai mult o colecție de metode de care ne folosim în aplicație fără să fie necesar instanțierea unui obiect. În continuare sunt prezentate metodele acestei clase.

**II.7.1.** Metoda **readXls**

Această metodă este metoda prin care se citește fișierul stat de funcțiuni și care creează listele de activități, lista profesorilor și cea al grupelor.

**public static** ArrayList<Activity> readXls(String fileName, ArrayList<Professor> professors, ArrayList<Group> groups, String faculty){  
 ArrayList<Activity> activities=**new** ArrayList<>();  
 File file=**new** File(fileName);  
 professors.clear();  
 groups.clear();  
 *profIndex*=0;  
 *groupIndex*=0;  
 *activityIndex*=0;

Această secvență de cod determină inițializarea listelor de profesori și grupe și variabilele de indice ale celor trei entități.

**try** {  
 FileInputStream fileInputStream = **new** FileInputStream(file);  
 HSSFWorkbook hssfWorkbook = **new** HSSFWorkbook(fileInputStream);  
 **int** numberOfSheets=hssfWorkbook.getNumberOfSheets();  
 **int** sheetToRead=-1;  
 **for** (**int** i=0;i<numberOfSheets;i++){  
 **if** (hssfWorkbook.getSheetName(i).equals(**"Centr"**)) {  
 sheetToRead=i;  
 i=numberOfSheets;  
 }  
 }  
 **if** (sheetToRead==-1) {  
 **throw new** NoSheetException(**"Sheet inexistent"**);  
 }  
 HSSFSheet hssfSheet=hssfWorkbook.getSheetAt(sheetToRead);

Aici se deschide fișierul ”file”, parametru de intrare al metodei, și se caută foaia (sheet) ”Centr” care conține datele necesare creării obiectelor de tip Activity, Professor și Group. Dacă fișierul nu conține această foaie metoda aruncă o excepție denumită NoSheetException, excepție creată special pentru această eroare.

Datele care trebuie citite încep din rândul 10 al foii. De aici se începe citirea datelor, până în momentul în care nu mai există date de citit în celule. Fișierul conține date pentru fiecare materie în parte. Datele unei materii sunt stocate în două rânduri concomitente, prima materie este poziționată în rândurile 10-11. Mai jos un exemplu de date pentru o materie:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Nr crt | DISCIPLINA | Fac. | | Dep. | Specializarea | An de studiu | Codul disciplinei | SEMESTRUL I | | | |  | SEMESTRUL II | | | |
|
|
| C | S | L | P | C | S | L | P |
| 1.0 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |  | 14 | 15 | 16 | 17 |
| 157 | Algoritmi fundamentali | MI |  | MIN | ITR | 1 | IT11 | 0 | 0 | 0 | 0 |  | 0 | 0 | 0 | 0 |
|  | IT11,1,4,4,0 |  |  |  | 175 |  |  | 2 | 4 | 8 |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Verif.atrib. StatF | Discipline cuplate interfacultati | | Posturile din statul de funcții în care se afla disciplina | | |
|
| la curs | la sem. |
|
| 18 | 21 | 22 | 28 | | |
| **ok** | 1 |  | 48 ; B ; 1 | 71 ; B ; 1 | 81 ; PO ; 1 |
|  |  |  | BAICOIANU Alexandra;2;2;0;0 | MAJERCSIK Luciana;0;0;8;0 | MAJERCSIK Luciana;0;2;0;0 |

Pe primul rând avem următoarele informații folositoare: disciplina ”Algoritmi fundamentali”, de la facultatea de Matematică și Informatică (”MI”), departamentul de Matematică și Informatică (”MIN”), anul de studiu 1, codul disciplinei ”IT11”. Pe al doilea rând avem informații legate de activitățile și profesorii disciplinei. În coloana 2 din rândul 2 apare ”IT11,1,4,4,0” ceea ce înseamnă că disciplina cu codul IT11 este predată în următoarele formațiuni: la curs 1 formațiune, adică toate grupele la un singur curs, la seminar 4 formațiuni, la laborator 4 formațiuni iar la practică 0, ceea ce înseamnă că nu există practică la această disciplină. În coloanele 9-17 apare durata totală a activităților disciplinei, adică în cazul de față 2 ore de curs, 4 ore de seminar și 8 ore de laborator. În coloanele cu indexul 28 apar profesorii cu activitățile atribuite la această disciplină în exemplul de față avem în felul următor: BAICOIANU Alexandra are 2 ore de curs și 2 ore de seminar iar MAJERCSIK Luciana are 8 ore de laborator și 2 ore de seminar. În ceea ce urmează în metodă avem o primă fază de citire a datelor din fișier cu tot felul de verificări, conversii etc.

**int** r=10;  
 **while** (!hssfSheet.getRow(r).getCell(1).toString().equals(**""**)){  
 Row firstRow=hssfSheet.getRow(r);  
 **if** (faculty.equals(firstRow.getCell(2).toString())){  
 String ok=firstRow.getCell(17).toString();  
 **if** (ok.equals(**"ok"**)) {  
 Row secondRow = hssfSheet.getRow(r + 1);  
 String subject = firstRow.getCell(1).toString().trim();  
 String departament = firstRow.getCell(4).toString().trim();  
 String[] speciality = firstRow.getCell(5).toString().split(**"\\+"**);  
 **for** (String spec : speciality)  
 System.***out***.println(spec);  
 **int** year = (**int**) firstRow.getCell(6).getNumericCellValue();  
 String[] codeFormation = secondRow.getCell(1).toString().split(**","**);  
 String codeSubject = codeFormation[0];  
 **int** fCrs = Integer.*parseInt*(codeFormation[1]);  
 **int** fSem = Integer.*parseInt*(codeFormation[2]);  
 **int** fLab = Integer.*parseInt*(codeFormation[3]);  
 **int** fPrc = Integer.*parseInt*(codeFormation[4]);  
 **int** s1C, s1S, s1L, s1P, s2C, s2S, s2L, s2P;  
 **if** (secondRow.getCell(8, Row.MissingCellPolicy.

***RETURN\_BLANK\_AS\_NULL***) != **null**)  
 s1C = (**int**) secondRow.getCell(8).getNumericCellValue();  
 **else** s1C = 0;  
 **if** (secondRow.getCell(9, Row.MissingCellPolicy.

***RETURN\_BLANK\_AS\_NULL***) != **null**)  
 s1S = (**int**) secondRow.getCell(9).getNumericCellValue();  
 **else** s1S = 0;  
 **if** (secondRow.getCell(10, Row.MissingCellPolicy.

***RETURN\_BLANK\_AS\_NULL***) != **null**)  
 s1L = (**int**)secondRow.getCell(10).getNumericCellValue();  
 **else** s1L = 0;  
 **if** (secondRow.getCell(11, Row.MissingCellPolicy.

***RETURN\_BLANK\_AS\_NULL***) != **null**)  
 s1P = (**int**)secondRow.getCell(11).getNumericCellValue();  
 **else** s1P = 0;  
 **if** (secondRow.getCell(13, Row.MissingCellPolicy.

***RETURN\_BLANK\_AS\_NULL***) != **null**)  
 s2C = (**int**)secondRow.getCell(13).getNumericCellValue();  
 **else** s2C = 0;  
 **if** (secondRow.getCell(14, Row.MissingCellPolicy.

***RETURN\_BLANK\_AS\_NULL***) != **null**)  
 s2S = (**int**)secondRow.getCell(14).getNumericCellValue();  
 **else** s2S = 0;  
 **if** (secondRow.getCell(15, Row.MissingCellPolicy.

***RETURN\_BLANK\_AS\_NULL***) != **null**)  
 s2L = (**int**)secondRow.getCell(15).getNumericCellValue();  
 **else** s2L = 0;  
 **if** (secondRow.getCell(16, Row.MissingCellPolicy.

***RETURN\_BLANK\_AS\_NULL***) != **null**)  
 s2P = (**int**)secondRow.getCell(16).getNumericCellValue();  
 **else** s2P = 0;

**int** semester = 0;  
 **if** (s1C + s1S + s1L + s1P > 0) semester = 1;  
 **else if** (s2C + s2S + s2L + s2P > 0) semester = 2;

În secțiunea de mai sus sunt citite datele cu privire la disciplină, adică denumire disciplină, departament, specializare, codul disciplinei, numărul de formațiuni, numărul de ore. Ajungem apoi la faza următoare în care se analizează datele despre profesori și aplicația ”distribuie” orele la profesori, ajungând în final la crearea obiectelor de Activity, adică a activităților. După cum se observă în exemplu, aici avem doi profesori care au diverse activități.

În primă fază metoda numără profesorii și rulează un algoritm de distribuire a activităților. Avem un prim ciclu pe numărul de profesori, în care se analizează datele privind numărul de ore alocate fiecărui profesor și se stochează în array-urile aferente (crsTime[x], semTime[x], labTime[x], prcTime[x]). Totodată se adună aceste valori în variabile care conțin numărul de ore pentru fiecare tip de activitate, adică crsTotal, semTotal, labTotal, prcTotal, însemnând numărul de ore de cursuri/ seminare/ laboratoare/practică totale alocate profesorilor arondați disciplinei. Se apelează metoda ***addIfNotInProfs*** în felul următor

actualProfs[i] = *addIfNotInProfs*(actualProfs[i], professors);

De fapt aici se verifică dacă obiectul profesor a fost deja creat sau trebuie creat acuma, acesta fiind prima disciplină unde apare profesorul în cauză.

**if** (semester > 0) {  
 **int** c = 27;  
 **while** (!secondRow.getCell(c,Row.MissingCellPolicy.***CREATE\_NULL\_AS\_BLANK***).

toString().equals(**""**)) {  
 c++;  
 }  
 **int** nrProfs = c - 27;  
 **if** (nrProfs > 0) {  
 Professor[] actualProfs = **new** Professor[nrProfs];  
 String[] profNames = **new** String[nrProfs];  
 **int**[] crsTime = **new int**[nrProfs];  
 **int**[] semTime = **new int**[nrProfs];  
 **int**[] labTime = **new int**[nrProfs];  
 **int**[] prcTime = **new int**[nrProfs];  
 **int** crsTotal = 0;  
 **int** semTotal = 0;  
 **int** labTotal = 0;  
 **int** prcTotal = 0;  
 **for** (**int** i = 0; i < nrProfs; i++) {  
 String[] profData = secondRow.getCell(27 +

i).toString().split(**";"**);  
 profNames[i] = profData[0];  
 crsTime[i] = Integer.*parseInt*(profData[1]);  
 crsTotal += crsTime[i];  
 semTime[i] = Integer.*parseInt*(profData[2]);  
 semTotal += semTime[i];  
 labTime[i] = Integer.*parseInt*(profData[3]);  
 labTotal += labTime[i];  
 prcTime[i] = Integer.*parseInt*(profData[4]);  
 prcTotal += prcTime[i];  
 actualProfs[i] = **new** Professor(0, profNames[i]);  
 actualProfs[i] = *addIfNotInProfs*(actualProfs[i],

professors);  
 }

Un al doilea ciclu verifică dacă nu cumva avem de-a face cu o dublare în fișierul excel a vreunuia dintre profesorii disciplinei. În exemplul nostru avem de-a face cu această situație. Unul dintre profesori apare în două coloane, astfel încât informația se poate comasa, iar orele alocate la a doua instanță vor fi transferate la prima instanță a profesorului și a doua instanță va fi ștearsă din lista profesorilor alocați disciplinei.

**for** (**int** i = 0; i < nrProfs; i++) {  
 **for** (**int** j = i + 1; j < nrProfs; j++) {  
 **if** (actualProfs[i] != **null** && actualProfs[j] !=

**null** && actualProfs[i].equals(actualProfs[j])) {  
 crsTime[i] += crsTime[j];  
 crsTime[j] = 0;  
 semTime[i] += semTime[j];  
 semTime[j] = 0;  
 labTime[i] += labTime[j];  
 labTime[j] = 0;  
 prcTime[i] += prcTime[j];  
 prcTime[j] = 0;  
 actualProfs[j] = **null**;  
 }  
 }

}

În faza următoare se vor crea sau actualiza grupele care vor avea această disciplină în orar. În ceea ce urmează, vedem cum se creează 4 array-uri de array-uri pentru fiecare tip de activitate posibilă. În acestea vom stoca grupele care vor avea activități legate de disciplină. Aici se folosește o metodă asemănătoare de creare/alocare grupă ***addIfNotInGroups***. Metoda creează o nouă grupă și o adaugă listei de grupe sau returnează o grupă existentă deja în lista grupelor.

**int** numberOfGroups = *max*(fCrs, fSem, fLab, fPrc);  
Group[][] actualGroups = **new** Group[4][numberOfGroups \* nrProfs];  
**for** (**int** i = 0; i < numberOfGroups; i++) {  
 actualGroups[0][i] = **new** Group(0, speciality[0], year, i + 1);  
 actualGroups[0][i] = *addIfNotInGroup*(actualGroups[0][i], groups);  
 **for** (**int** j = 1; j < 4; j++) {  
 actualGroups[j][i] = actualGroups[0][i];  
 }  
}  
**for** (**int** i = 1; i < nrProfs; i++) {  
 **for** (**int** k = 0; k < numberOfGroups; k++) {  
 **for** (**int** j = 0; j < 4; j++) {  
 actualGroups[j][i \* numberOfGroups + k] = actualGroups[j][k];  
 }  
 }  
}

După această fază ajungem la crearea efectivă a activităților, a obiectelor de tip Activity care se adaugă la lista ***activities*** care va conține toate activitățile din anul universitar și pe care le vom putea manipula sub forma unor etichete între tabele și în interiorul tabelelor de orar. Activitățile se creează de o metodă denumită ***createActivities*** care, de fapt, creează mai multe activități deodată. Metoda se apelează pentru fiecare tip de activitate care urmează să fie creată și se vor crea una sau mai multe activități, depinde de câte trebuie, câte sunt prevăzute pentru fiecare tip de activitate în parte (curs, seminar, laborator, practică). Metoda ***createActivities*** va fi prezentată puțin mai târziu.

**for** (**int** i = 0; i < nrProfs; i++) {  
 **if** (crsTime[i] > 0) {  
 **int** type = 1;  
 *createActivities*(activities,professors,groups,i,

numberOfGroups,fCrs,actualGroups,subject,codeSubject,

actualProfs,type,semester,year,crsTotal,crsTime,s1C+s2C);

}  
 **if** (semTime[i] > 0) {  
 **int** type = 2;

*createActivities*(activities,professors,groups,i,

numberOfGroups,fSem,actualGroups,subject,codeSubject,

actualProfs,type,semester,year,semTotal,semTime,s1C+s2C};

}  
 **if** (labTime[i] > 0) {  
 **int** type = 3;  
 *createActivities*(activities,professors,groups,i,

numberOfGroups,fLab,actualGroups,subject,codeSubject,

actualProfs,type,semester,year,labTotal,labTime,s1C+s2C+1);

}  
 **if** (prcTime[i] > 0) {  
 **int** type = 4;  
 *createActivities*(activities,professors,groups,i,

numberOfGroups,fPrc,actualGroups,subject,codeSubject,

actualProfs,type,semester,year,prcTotal,prcTime,s1C+s2C+1);

}  
 }

Ultima secțiune a metodei tratează un caz special și anume cazul în care o activitate, de obicei un curs, este destinată mai multor grupe din cadrul mai multor specializări. În acest caz cursul ca și activitate alocată unui profesor apare în fișa de funcțiuni numai în cazul uneia dintre specializări (de obicei prima), ci și la specializarea 2, aici însă cursul nefiind alocat niciunui profesor pentru că a fost alocat deja la specializarea 1. În tabel aici apar celelalte activități, laboratoare, seminare, care se referă la grupele cu această specializare. Următoarea secțiune de cod tratează această situație specială, adăugând grupele din specializarea la care nu mai apare cursul, la activitatea deja creată și adăugată la lista de activități, ***activities***.

**if** (s1C + s2C == 0) {  
 List<Group> groupsToAddToCourse=**new** ArrayList<Group>();  
 **boolean** isIn;  
 **for** (Group newGroup:actualGroups[0]) {  
 isIn=**false**;  
 **for**(Group oldGroup:groupsToAddToCourse) {  
 **if** (oldGroup==newGroup) {  
 isIn=**true**;  
 }  
 }  
 **if** (!isIn) {  
 groupsToAddToCourse.add(newGroup);  
 }  
 }  
 Group[] groupsToAdd=groupsToAddToCourse.toArray(**new**

Group[0]);  
 **for** (Activity completable:activities){  
 **if** (completable.getCodeSubject().equals(codeSubject)) {  
 **if** (completable.getType()==1) {  
 completable.addGroups(groupsToAdd);  
 }  
 }  
 }  
 }  
 }  
 }  
 }

}

În final se trece la următoarea disciplină, variabila r (de la rând) este mărită astfel încât să se citească rândul unu al disciplinei următoare.

r=r+2;  
 }

}

Dacă în cursul citirii/prelucrării datelor se aruncă orice excepție, metoda va afișa excepția respectivă și va returna **null,** meniul principal va afișa că există o greșeală de citire/prelucrare a datelor.

**catch** (Exception ex){  
 System.***out***.println(ex.toString());  
 **return null**;  
 }  
 **return** activities;  
 }

**II.7.2.** Metoda **readFile**

Metoda *readFile* este o metodă auxiliară metodei readXls care se folosește de clasa FileChooser prin care se deschide o fereastră grafică de alegere a unui fișier de state de funcțiuni. FileChooser-ul este setat în așa fel încât nu acceptă ca și fișier de intrare decât fișiere de tip\*.xls, adică Excel 97 – Excel 2003 Workbook.

**public static** Pair<String, String> readFile() {  
  
 Stage chooserStage=**new** Stage();  
 chooserStage.setTitle(**"Deschidere fișier stat de funcțiuni"**);  
 FileChooser fileChooser = **new** FileChooser();  
 fileChooser.setTitle(**"Open File"**);  
 fileChooser.getExtensionFilters().addAll(  
 **new** FileChooser.ExtensionFilter(**"Excel files"**, **"\*.xls"**));  
 File selectedFile = fileChooser.showOpenDialog(chooserStage);  
 **if** (selectedFile == **null**) {  
 chooserStage.close();  
 **return null**;  
 }  
  
 String faculty;  
 String fileName=selectedFile.getAbsolutePath();  
  
 **try** {  
 FileInputStream fileInputStream = **new** FileInputStream(fileName);  
 HSSFWorkbook hssfWorkbook = **new** HSSFWorkbook(fileInputStream);  
 **int** numberOfSheets = hssfWorkbook.getNumberOfSheets();  
 **int** sheetToRead = -1;  
 **for** (**int** i = 0; i < numberOfSheets; i++) {  
 **if** (hssfWorkbook.getSheetName(i).equals(**"DateC"**)) {  
 sheetToRead = i;  
 i = numberOfSheets;  
 }  
 }  
 **if** (sheetToRead == -1) {  
 **throw new** NoSheetException(**"Sheet inexistent"**);  
 }  
 HSSFSheet hssfSheet = hssfWorkbook.getSheetAt(sheetToRead);  
 Row row=hssfSheet.getRow(2);  
 faculty=row.getCell(12).toString();  
 **if** (faculty.length()<1) {  
 **return null**;  
 }  
 }  
 **catch** (Exception ex) {  
 **return null**;  
 }  
 **return new** Pair<>(fileName,faculty);  
}

Când utilizatorul încearcă să deschidă un fișier pentru a citi datele statelor de funcțiuni, în cadrul acestei metode se determină dacă fișierul conține pagina ”DateC” care conține date privind colectivul de cadre didactice din cadrul facultății în cauză și dacă pagina există, va căuta în celula M3 abrevierea denumirii facultății. Dacă această celulă conține date, nu este nulă, atunci metoda returnează numele fișierului și denumirea facultății ale cărei state de funcțiuni sunt conținute în fișier. Aceste date, adică numele fișierului și cel al facultății, vor fi folosite apoi de metoda readXls care, după cum am văzut, va citi și va procesa datele din statul de funcțiuni. În cazul în care pagina ”DateC” nu există sau celula M3 din pagină nu conține nici un fel de date, sau în cazul apariției de excepții cauzate de erori de citire, conversie etc., metoda va returna **null**, astfel că readXls nu va fi apelat ci se va afișa un mesaj de eroare.

**II.7.3.** Metoda **writeXls**

Aceasta este metoda care exportă orarul în formatul Excel. Metoda se folosește, desigur, de aceeași bibliotecă Apache POI pentru conversia în formatul Excel dorit. Metoda va crea un fișier nou sau va suprascrie unul existent și va genera orarul pe ani al facultății în același format care se folosește la fișierele de orar ale Universității Transilvania din Brașov. Astfel aceste date pot fi cu ușurință copiate în fișierele de orar respective.

**public static boolean** writeXls(String file, ArrayList<Professor> professors, ArrayList<Group> groups, ArrayList<Activity> activities, ArrayList<Room> rooms, **int** semester)

Metoda returnează ”true” dacă exportarea a reușit și ”false” în caz contrar.

**II.7.4.** Metoda **saveFile**

Este o metodă auxiliară metodei writeXls care se folosește de clasa FileChooser pentru a crea o cale către și o denumire de fișier, care se va folosi de către metoda writeXls pentru a exporta orarul în formatul Excel care poate fi folosit de către orarist pentru fișierele Excel ale Universității.

**public static** String saveFile() {  
  
 Stage stage=**new** Stage();  
 FileChooser fileChooser = **new** FileChooser();  
 File selectedFile = fileChooser.showSaveDialog(stage);  
 **if** (selectedFile == **null**) {  
 stage.close();  
 **return null**;  
 }  
 String fileName=selectedFile.getAbsolutePath();  
 String[] fileNamePieces =fileName.split(**"\\."**);  
 fileName=**""**;  
 **if** (fileNamePieces.**length**>1) {  
 **for** (**int** i = 0; i < fileNamePieces.**length** - 1; i++) {  
 fileName = fileName + fileNamePieces[i];  
 }  
 }  
 **else** {  
 fileName=fileNamePieces[0];  
 }  
 **return** fileName;  
}

Metoda returnează un String, adică o cale, pe care metoda writeXls o folosește la crearea fișierului care va conține orarul în forma sa finală.

**II.7.5.** Metoda **addIfNotInGroup**

este o metodă de ajutor pentru readXls care verifică dacă o grupă există deja și este adăugată în lista grupelor, ***groups.***

**static** Group addIfNotInGroup(Group group, ArrayList<Group> groups){  
 **for** (Group nextGroup:groups){  
 **if** (group.getGroupName().equals(nextGroup.getGroupName())){  
 **return** nextGroup;  
 }  
 }  
 group.setIdGroup(*groupIndex*);  
 groups.add(group);  
 *groupIndex*++;  
 **return** group;  
}

Dacă grupa există deja metoda va returna grupa existentă pentru ca să se lucreze ca acesta în continuare, dacă grupa nu există atunci aceasta va fi creată și se returnează pentru a se lucra în continuare cu această grupă nou creată.

**II.7.6.** Metoda **addIfNotInProfs**

Aceasta este o metodă care verifică dacă un profesor care se creează în timpul citirii și prelucrării datelor este un profesor încă inexistent în lista profesorilor, ***professors***, sau deja există. Metoda funcționează similar cu metoda anterioară ***addIfNotInGroups.***

**static** Professor addIfNotInProfs(Professor professor, ArrayList<Professor> professors) {  
 **for** (Professor nextProfessor : professors) {  
 **if** (professor.getName().equals(nextProfessor.getName())) {  
 **return** nextProfessor;  
 }  
 }  
 professor.setIdProfessor(*profIndex*);  
 professors.add(professor);  
 *profIndex*++;  
 **return** professor;  
}

**II.7.7.** Metoda **createActivities**

Această metodă va crea și adăuga obiectele de tip Activity, adică activitățile, la lista de activități denumită ***activities.*** Acestă liustă conține toate activitățile din cadrul facultății care sunt prevăzute pentru anul de studiu în curs.

**static void** createActivities(ArrayList<Activity> activities, ArrayList<Professor> professors, ArrayList<Group> groups, **int** profNumber, **int** numberOfGroups, **int** fAct, Group[][] actualGroups, String subject, String codeSubject, Professor[] actualProfs, **int** type, **int** semester, **int** year, **int** actTotal, **int**[] actTime, **int** numberOfCourses) {

După cum se vede, metoda are foarte mulți parametri de intrare care sunt detaliați în continuare:

* ArrayList<Activity> activities – lista activităților create și adăugate până în momentul apelării metodei
* ArrayList<Professor> professors – lista profesorilor creați și adăugați până în momentul apelării metodei
* ArrayList<Group> groups – lista grupelor create și adăugate până în momentul apelării metodei
* int profNumber – numărul profesorilor din lista de profesori care sunt implicați în predare la disciplina curentă
* int numberOfGroups – numărul grupelor care participă la activitate
* int fAct – numărul de formațiuni
* Group[][] actualGroups – matricea grupelor care participă la activitățile disciplinei curente. Este vorba despre o matrice bidimensională unde prima dimensiune este pentru cele 4 tipuri de activități iar a doua pentru grupele aferente fiecărui tip de activitate (curs, seminar, laborator, practică)
* String subject – titlul disciplinei
* String codeSubject – codul disciplinei
* Professor[] actualProfessors – array-ul profesorilor care participă la activitățile disciplinei, parametrul profNumber se referă la această listă
* int type – tipul activității
* int semester – semestrul activității
* int year – anul de studiu
* int actTotal – durata totală săptămânală a activităților de tip ”type”
* int[] actTime – durata activităților în funcție de tip (de exemplu actTime[0] este durata unui curs la disciplină)
* int numberOfCourses – numărul cursurilor de la disciplina actuală

**float** nrAct=(**float**) fAct\*actTime[profNumber]/actTotal;  
 **for** (**int** j = 0; j < nrAct ; j++) {  
 **int** groupTeam = (numberOfGroups/fAct);  
 Group[] groupsToAdd = **new** Group[groupTeam];  
 **int**[] groupIdToAdd=**new int**[groupTeam];  
 **int** k = 0, l = 0;  
 **while** (k < groupTeam) {  
 **if** (actualGroups[type-1][j+k+l]!=**null**) {  
 groupsToAdd[k] = actualGroups[type-1][j+k+l];  
 groupIdToAdd[k] = actualGroups[type-1][j+k+l].getIdGroup();  
 actualGroups[type-1][j + k + l]=**null**;  
 k++;  
 }  
 **else** {  
 l++;  
 }  
 }

În secțiunea de cod de mai sus se calculează numărul de grupe care participă la activitatea care urmează să fie creată, se compun lista de grupe și lista cu ID-urile grupelor respective, care se vor adăuga la activitatea care se va crea.

**float** activityTime;  
 **if** (nrAct<1) {  
 activityTime=(**float**) fAct/actTotal;

}  
 **else {** activityTime=(**float**) actTotal/fAct/2;  
 }

activityTime\*=2;  
 **if** (activityTime<1) activityTime=1;

Se calculează durata activității în funcție de durata totală și numărul de formațiuni care participă la activitate.

Activity newActivity = **new** Activity(*activityIndex*, subject, codeSubject, actualProfs[profNumber].getIdProfessor(), type, groupIdToAdd, semester, year, (**int**) activityTime, (activityTime >=2));

Aici se creează activitatea după care se adaugă toate grupele care participă la activitate, dacă este o activitate la care participă mai multe grupe.

**if**(numberOfCourses==-1){  
 **for** (Activity nextActivity:activities){  
 **if**((newActivity.getSubject().equals(nextActivity.getSubject())

&&(nextActivity.getType()==1))){  
 nextActivity.addGroups(groupsToAdd);  
 }  
 }  
 }  
 activities.add(newActivity);

După ce se adaugă activitatea la lista activităților, care se va folosi apoi la elaborarea orarului, se va adăuga indicele activității la activitățile profesorului și a grupelor participante.

professors.get(newActivity.getProfessorId()).addActivity

(activities.indexOf(newActivity));  
 *activityIndex*++;  
 **for** (**int** group : newActivity.getGroupsId()) {  
 groups.get(group).addActivity

(activities.indexOf(newActivity));  
 }  
 }  
}

**II.7.8.** Metoda **saveData**

Aceasta este metoda folosită pentru salvarea datelor prelucrate într-un moment oarecare, adică este metoda prin care se salvează proiectul de elaborare a orarului într-o stare intermediară, nefinalizată. Metoda se folosește de biblioteca Gson care face conversia obiectelor Java în date de tip Json și invers. De fapt se salvează listele obiectelor Activity, Professor, Group și Room, adică ArrayList-urile activities, professors, groups și rooms. Acestea conțin toate informațiile necesare pentru a se putea relua activitatea după încărcarea datelor salvate în prealabil.

**public static boolean** saveData(String file, ArrayList<Professor>

professors, ArrayList<Group> groups, ArrayList<Activity>

activities) {  
  
 Gson gson = **new** GsonBuilder()  
 .setPrettyPrinting()  
 .create();  
 **try** {  
 FileWriter actWriter=**new** FileWriter(file+**".act"**);  
 gson.toJson(activities,actWriter);  
 actWriter.close();  
 }  
 **catch** (Exception ex) {  
 Utility.*message*(**"Salvare activități eșuată"**);  
 **return false**;  
 }  
  
 **try** {  
 FileWriter profWriter=**new** FileWriter(file+**".prf"**);  
 gson.toJson(professors,profWriter);  
 profWriter.close();  
 }  
 **catch** (Exception ex) {  
 Utility.*message*(**"Salvare profesori eșuată"**);  
 **return false**;  
 }  
  
 **try** {  
 FileWriter groupWriter=**new** FileWriter(file+**".grp"**);  
 gson.toJson(groups,groupWriter);  
 groupWriter.close();  
 }  
 **catch** (Exception ex) {  
 Utility.*message*(**"Salvare grupe eșuată"**);  
 **return false**;  
 }  
  
 **try** {  
 FileWriter roomWriter=**new** FileWriter(file+**".rm"**);  
 gson.toJson(groups,roomWriter);  
 roomWriter.close();  
 Utility.*message*(**"Salvare reușită"**);  
 }  
 **catch** (Exception ex) {  
 Utility.*message*(**"Salvare săli eșuată"**);  
 **return false**;  
 }  
  
 **return true**;  
}

Metoda salvează cele 4 liste de obiecte esențiale, adică Activity, Professor, Group și Room în fișiere separate, numele fișierului fiind comun, extensia diferită în funcție de ce conține act, prf, grp, rm. Aici se utilizeaază biblioteca Gson pentru conversia obiectelor în format JSON și se exportă aceste elemente JSON în fișiere text.

**II.7.9.** Metodele **load**

Pentru citirea datelor din fișierele salvate mai avem 4 metode diferite pentru fiecare tip de obiecte, dintre care vom exemplifica una singură, toate cele patru fiind similare. Metoda exemplificată returnează un array pentru elementele încărcate. Și aici este folosită biblioteca Gson pentru conversia (reconversia) elementelor Json în obiecte Java

**public static** ArrayList<Activity> loadActivities(String file) {  
  
 ArrayList<Activity> activities;  
 Gson gson=**new** Gson();  
 **try** {  
 Reader actReader = Files.*newBufferedReader*(Paths.*get*(file));  
 activities = gson.fromJson(actReader,

**new** TypeToken<ArrayList<Activity>>() {}.getType());  
 actReader.close();  
 Utility.*message*(**"Citire activități reușită"**);  
 }  
 **catch** (Exception ex) {  
 Utility.*message*(**"Citire activități eșuată"**+ex.toString());  
 **return null**;  
 }  
 **return** activities;  
}

Fiecare metodă separată va citi fișierul dedicat și va returna un ArrayList de tipul obiectelor citite. În momentul citirii datelor toate celelalte date de dinainte sunt șterse și înlocuite cu datele încărcate din fișiere.

**II.7.10** Alte metode ajutătoare

Clasa Utility mai conține câteva clase ajutătoare pe care le amintim aici, fără a comenta prea mult, fiind niște metode simple de afișare, calcul etc.

* **static int** max(**int** a,**int** b,**int** c,**int** d) - metoda returnează maximul dintre 4 numere întregi, se folosește ca ajutor pentru metoda readXls
* **public static int** maxYear(ArrayList<Activity> activities) – metoda returnează numărul de ani de studiu la facultate
* **public static** IndexedLabel createProfLabel(Activity currentActivity, Professor professor, ArrayList<Group> groups) – creează un **IndexedLabel** folosit în reprezentarea grafică a orarului unui profesor
* **public static** IndexedLabel createYearLabel(Activity currentActivity, Professor professor, ArrayList<Group> groups) – creează un **IndexedLabel** folosit în reprezentarea grafică a orarului complet al unui an și semestru
* **static void** message(String message) – afișează un mesaj într-o fereastră

**II.8. Clasa Scenes**

Clasa Scenes este clasa grafică a aplicației, aici se creează toate ferestrele în afară de meniul principal, care are o clasă separată. Totodată în clasa Scenes sunt elaborate elementele dinamice de mișcare, reprezentare grafică dar și partea logică a mișcărilor activităților, verificările etc. Clasa are câteva atribute și anume:

* final int **HOURS**=7, **DAYS**=12 – două constante pentru tabele de orar
* ArrayList<Activity> **activities** – lista tuturor activităților
* ArrayList<Professor> **professors** – lista profesorilor
* ArrayList<Group> **groups** – lista grupelor
* ArrayList<Room> **rooms** – lista sălilor de curs
* static IndexedLabel **draggingLabel** – eticheta care este mișcată cu ajutorul mouse-ului. Este o variabilă statică pentru că este nevoie de ea să se mute între diferite elemente/componente ale clasei Scenes
* final DataFormat **labelFormat** – este un DataFormat specific aplicației, care prin acest format evită ca nu cumva prin Drag and Drop eticheta *draggingLabel* să nu poată să fie mutată în ferestre aparținând altor aplicații

**Constructorul** este:

**public** Scenes(ArrayList<Professor> professors, ArrayList<Activity> activities, ArrayList<Group> groups, ArrayList<Room> rooms) {  
 **this**.**professors** = professors;  
 **this**.**groups**=groups;  
 **this**.**activities**=activities;  
 **this**.**rooms**=rooms;  
 DataFormat dataFormat = DataFormat.*lookupMimeType*(**"Unitbv"**);  
 **if** (dataFormat==**null**)  
 **labelFormat**=**new** DataFormat(**"Unitbv"**);  
 **else  
 labelFormat**=dataFormat;  
}

Prin constructor se inițiază cele patru liste de obiecte esențiale și se inițiază atributul *labelFormat.*

Metodele acestei clase pot fi împărțite în câteva categorii cum ar fi clase grafice, clase de verificare, clase de manipulare elemente, clase pentru tratarea evenimentelor, în special cele de Drag and Drop. Dintre clasele grafice voi prezenta în detaliu orarul profesorului

**II.8.1.** Metoda **professorsScheduleScene**

Metoda generează și afișează orarul unui profesor și un tabel cu activitățile profesorului. Activitățile care nu au fost încă plasate pe orar apar în acest tabel al activităților. Se poate face Drag and Drop între cele două tabele.

**public void** professorsScheduleScene(**int** professorId, **int** semester) {  
  
 Professor professor = **professors**.get(professorId);  
 Stage scheduleStage=**new** Stage();  
 HBox horizontalBox=**new** HBox();  
 GridPane classesGrid=**new** GridPane();  
 GridPane scheduleGrid=**new** GridPane();  
 StackPane[][] scheduleMatrix=**new** StackPane[**HOURS**+1][**DAYS**+1];

Se inițiază panelurile (Pane) care vor conține celelalte elemente grafice. Este vorba de un GridPane (un panel de tip tabel) care în fiecare celulă a ei are un StackPane care va avea proprietatea de a accepta Drag and Drop de elemente. Astfel că vom putea face Drag and Drop cu etichete (IndexedLabel) pentru putea folosi aplicația prin simpla tragere a activităților între orare și/sau ore și zile de desfășurare.

String[] ore={**"Zi \\ Ora"**,**"8-9,50"**,**"10-11,50"**,**"12-13,50"**,**"14-**

**15,50"**,**"16-17,50"**,**"18-19,50"**,**"20-21,50"**};  
String[] zile={**"Luni"**,**"Marti"**,**"Miercuri"**,**"Joi"**,**"Vineri"**,**"Sambata"**};  
  
**for** (**int** i=0;i<**HOURS**+1;i++) {  
 scheduleMatrix[i][0]=**new** StackPane();  
 scheduleMatrix[i][0].setPrefSize(80,40);  
 scheduleMatrix[i][0].setStyle(**"-fx-border-color:black;**

**-fx-background-color:beige; -fx-padding:5"**);  
 scheduleMatrix[i][0].getChildren().add(**new** Label((ore[i])));  
 scheduleGrid.add(scheduleMatrix[i][0], i, 0);  
}  
  
**for** (**int** j=1;j<**DAYS**/2+1;j++) {  
 scheduleMatrix[0][j]=**new** StackPane();  
 scheduleMatrix[0][j].setStyle(**"-fx-border-color:black;**

**-fx-background-color:beige; -fx-padding:5"**);  
 scheduleMatrix[0][j].getChildren().add(**new** Label((zile[j-1])));  
 scheduleGrid.add(scheduleMatrix[0][j], 0, j\*2-1,1,2);  
 System.***out***.println(j);  
}

Mai sus se crează headerul și coloana cu zilele pentru orarul profesorului.

**for** (**int** i=1;i<**HOURS**+1;i++) {  
 **for** (**int** j=1;j<**DAYS**+1;j++){  
 scheduleMatrix[i][j]=**new** StackPane();  
 scheduleMatrix[i][j].setStyle(**"-fx-border-color:black"**);  
 scheduleMatrix[i][j].setPrefSize(80,40);  
 scheduleMatrix[i][j].setAlignment(Pos.***TOP\_CENTER***);  
 addDropHandlingProfSchedule(scheduleMatrix[i][j]);  
 **int** presentActivityId=professor.getActivityProfessor(semester,i-1,j-1);  
 **if** (presentActivityId!=-1) {  
 Activity presentActivity=**activities**.get(presentActivityId);  
 IndexedLabel lbl=Utility.*createProfLabel*(presentActivity,

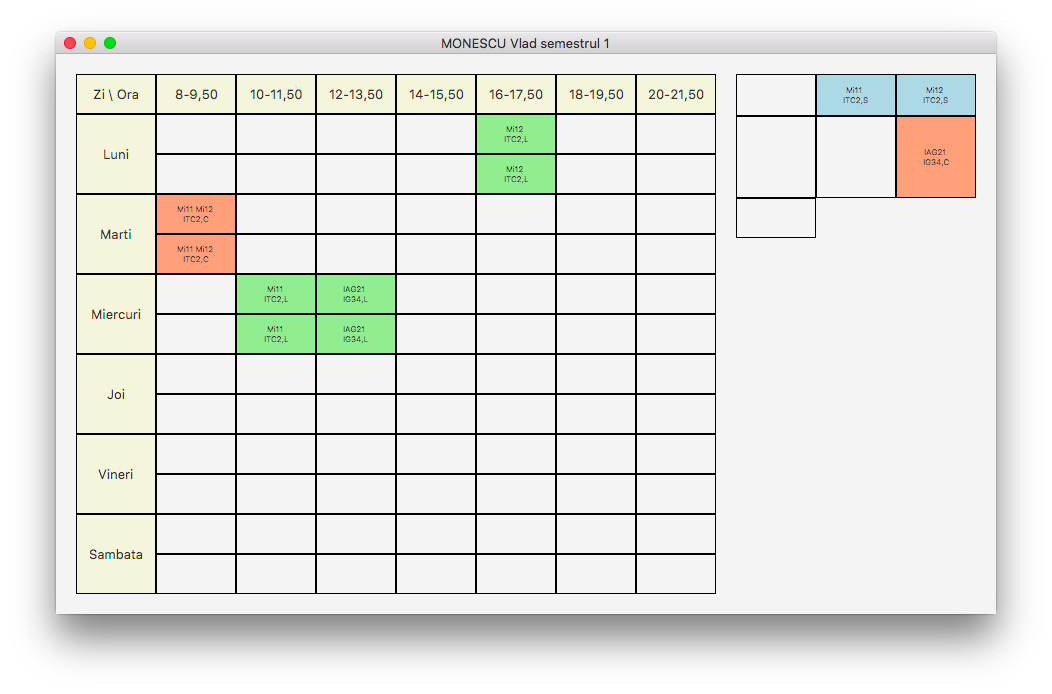
professor, **groups**);  
 scheduleMatrix[i][j].getChildren().add(lbl);  
 dragTextArea(lbl);  
 System.***out***.println(**"Add label "** + **activities**.get(professor.

getActivityProfessor(semester,i - 1, j - 1)).

getSubject() + **" "** + i + **","** + j);  
 }  
 scheduleGrid.add(scheduleMatrix[i][j], i, j);  
 }  
}

Mai sus avem secvența care încarcă în celulele panelului **scheduleGrid** de tip *GridPane* elementele din matricea **scheduleMatrix[][]** de tip *StackPane* personalizate, care prin aplicarea metodei **addDrophandlingProfSchedule** vor fi capabile să lucreze cu etichetele Drag and Drop, iar etichetele prin metoda **dragTextArea** devin ”dragabble”.

Mai jos se pregătește un alt GridPane care conține toate activitățile profesorului și din care putem trage activitățile pe orar și invers. Acest panel va apare lângă panelul de orar în aceeași fereastră pe care o prezint mai jos.



În stânga se poate vedea orarul profesorului din semestrul 1, iar în dreapta avem 3 activități care nu sunt încă poziționate pe orar. Desigur că orarul poate fi considerat terminat când toate activitățile și-au găsit locul în orar, nu există nici o activitate fără zi și oră de desfășurare.

**int** nrActivities=0;  
 **for** (**int** activity: professor.getActivitiesOfProfessor()) {  
 **if** (**activities**.get(activity).getSemester() == semester) {  
 nrActivities++;  
 }  
 }  
 StackPane[] classesArray=**new** StackPane[nrActivities];  
 **int** sqr,multiplier=0;  
 **if** (Math.*floor*((Math.*sqrt*(nrActivities)))=

=Math.*sqrt*(nrActivities)) {  
 sqr = (**int**) Math.*floor*(Math.*sqrt*(nrActivities));  
 }  
 **else** {  
 sqr = (**int**) Math.*floor*(Math.*sqrt*(nrActivities)) + 1;  
 }  
 **int** count=0;  
 **for** (**int** i = 0; i< professor.getActivitiesOfProfessor().**length**;

i++) {  
 Activity currentActivity=**activities**.get(professor.

getActivitiesOfProfessor()[i]);  
 **if** (currentActivity.getSemester() == semester) {  
 classesArray[count] = **new** StackPane();  
 classesArray[count].setPrefWidth(80);  
 classesArray[count].setMinHeight(40);  
 classesArray[count].setAlignment(Pos.***CENTER***);  
 addDropHandlingClasses(classesArray[count], professorId);  
 classesGrid.add(classesArray[count], count % sqr, count/sqr);  
 **if** (!onSchedule(professor.getActivitiesOfProfessor()[i],

professorId, semester)) {  
 IndexedLabel lbl = Utility.*createProfLabel*(currentActivity,

professor, **groups**);  
 classesArray[count].getChildren().add(lbl);  
 dragTextArea(lbl);  
 }  
 count++;  
 }  
 }  
 **for** (**int** i=sqr-1;i<sqr+1;i++) {  
 **if** (sqr\*i>count) {  
 multiplier=i;  
 **break**;  
 }  
 }  
 **if** (multiplier==0) multiplier=sqr;  
 **for** (**int** i=count;i<sqr\*multiplier;i++) {  
 StackPane pane = **new** StackPane();  
 pane.setPrefWidth(80);  
 pane.setMinHeight(40);  
 pane.setAlignment(Pos.***CENTER***);  
 addDropHandlingClasses(pane,professorId);  
 classesGrid.add(pane, i % sqr, i / sqr);  
 }

Mai sus se fac calcule pentru a se stabili câte activități are profesorul în semestrul activ și cum să se aranjeze din punct de vedere estetic acestea pe tabelul activităților încă nepuse pe orar.

horizontalBox.getChildren().addAll(scheduleGrid,classesGrid);  
 horizontalBox.setPadding(**new** Insets(20,20,20,20));  
 horizontalBox.setAlignment(Pos.***CENTER***);  
 horizontalBox.setSpacing(20);  
 Scene scheduleScene=**new** Scene(horizontalBox);  
 scheduleStage.setScene(scheduleScene);  
 scheduleStage.setTitle(professor.getName()+**"semestrul "**+semester);  
 scheduleStage.show();  
}

În final cele două tabele sunt puse într-un *HBox*, un alt element container al JavaFX, se dă titlul ferestrei și se vizualizează fereastra creată.

Metoda care generează și afișează orarul unei grupe este similară și se numește **groupScheduleScene,** iar metoda de generare a orarului unui an și semestru se numește **yearScheduleScene** și este deasemenea foarte asemănătoare cu acestea.

În afară de aceste metode de afișare a orarelor clasa mai are niște metode de configurare, calcule, ajutor pentru funcționarea corectă a acestor scene grafice. Voi prezenta în continuare aceste metode care sunt legate de metoda prezentată în detaliu *professorScheduleScene*.

**II.8.2.** Metoda **addDropHandlingProfSchedule**

Metoda se aplică pentru *StackPane-*urile din orarul profesorului și este răspunzător pentru funcționarea corectă a Drag and Dropului în cazul acestor tabele.

**private void** addDropHandlingProfSchedule(StackPane pane) {  
 pane.setOnDragOver(e -> {  
 Dragboard db = e.getDragboard();  
 **if** (db.hasContent(**labelFormat**)&&

*draggingLabel*!=**null**&&pane.getChildren().isEmpty()) {  
 e.acceptTransferModes(TransferMode.***MOVE***);  
 }  
 });  
 pane.setOnDragDropped(e -> {  
 Dragboard db = e.getDragboard();  
 Activity activity=**activities**.get(*draggingLabel*.getActivityId());  
 **if** (db.hasContent(**labelFormat**)) {  
 **int** row= GridPane.*getRowIndex*(pane),  
 col= GridPane.*getColumnIndex*(pane),  
 time = activity.getTime();  
 **if** (isMovableToProfSchedule(col,row,time,activity)){  
 Pane parentOfLabel=(Pane) *draggingLabel*.getParent();  
 parentOfLabel.getChildren().clear();  
 moveToProfSchedule(pane,col,row,activity);  
 *draggingLabel* = **null**;  
 e.setDropCompleted(**true**);  
 }  
 }  
 });  
}

În prima parte panelul *pane* este ”învățat” să accepte Drag and Drop, iar modul de Drag and Drop este de MOVE, mutare aceasta însemnând că, eticheta care se trage dintr-un alt tabel sau din altă poziție se va muta, se va șterge din poziția din care a fost trasă pe panelul în cauză.

În a doua parte vedem ce se întâmplă în momentul în care eticheta, care de fapt este reprezentarea grafică a unei activități, este aruncată (Drop) pe panelul nostru. Aici intervin două metode, una de verificare și una de mutare. Cea de verificare este:

**II.8.3.** Metoda **isMovableToProfSchedule**

verifică dacă activitatea în cauză se poate muta pe poziția dorită, adică dacă sunt îndeplinite simultan următoarele condiții:

* activitatea nu depășește orarul zilei
* la ora respectivă profesorul este liber, nu are alte activități
* la ora respectivă grupa (grupele) este liberă, nu are alte activități
* la ora respectivă sala, dacă a fost deja aleasă pentru activitate, este liberă

În continuare este prezentată această metodă

**private boolean** isMovableToProfSchedule(**int** col, **int** row, **int** time,

Activity activity) {  
 **int** semester = activity.getSemester();  
 Professor professor = **professors**.get(activity.getProfessorId());  
 **if** (row == 0 || col == 0) {  
 **return false**;  
 }  
 **if** (col + (time - 1) / 2 > 7) {  
 **return false**;  
 }  
 **int** add;  
 **int** X,Y;  
 **switch** (time%2) {  
 **case** 1:  
 **if** (row % 2 == 0)  
 add=-1;  
 **else** add=1;  
 **for** (**int** t=0;t<time;t++) {  
 X=row+(t%2)\*add;  
 Y=col+(t+1)/2;  
 **if** (professor.getActivityProfessor(semester,Y-1, X-1)!=-1) {  
 **return false**;  
 }  
 **for** (**int** j = 0; j<activity.getGroupsId().**length**; j++) {  
 **if** (**groups**.get(activity.getGroupsId()[j]).

getActivityGroup(semester, Y - 1, X - 1) != -1) {  
 **return false**;  
 }  
 }  
 }  
 **break**;  
 **case** 0:  
 **if** (row % 2 == 0)  
 row--;  
 **for** (**int** t=0;t<time;t++) {  
 X=row+t%2;  
 Y=col+t/2;  
 **if** (professor.getActivityProfessor(semester,Y-1, X-1)!=-1) {  
 **return false**;  
 }  
 **for** (**int** j = 0; j<activity.getGroupsId().**length**; j++) {  
 **if** (**groups**.get(activity.getGroupsId()[j]).

getActivityGroup(semester, Y - 1, X - 1) != -1) {  
 **return false**;  
 }  
 }  
 }  
 **break**;  
 **default**:  
 **break**;  
 }  
 **return true**;  
}

Dacă metoda returnează ”true” înseamnă că mutarea este posibilă și se trece la execuția metodei de mutare

**II.8.4.** Metoda **moveToProfSchedule**

Metoda este răspunzătoare nu numai de mutarea grafică a etichetei care corespunde activității care se dorește mutată, ci va face și ”mutările” logice în matricele de orar ale profesorului, grupelor și a orarului mare de an/semestru. Metoda se prezintă astfel:

**private void** moveToProfSchedule(StackPane pane,**int** col, **int** row,

Activity activity) {  
  
 StackPane secondPane,actualPane;  
 GridPane grid;  
 IndexedLabel actualLabel;  
 IndexedLabel[] labels;  
 **int** add;  
 ObservableList<Node> childrens;  
 **int** X,Y,nodeX,nodeY;  
  
 Professor professor=**professors**.get(activity.getProfessorId());  
 **int** semester=activity.getSemester();  
 **int** time=activity.getTime();  
 **switch** (time%2) {

**case** 1:

grid=(GridPane) pane.getParent();  
 labels=**new** IndexedLabel[time];  
 **for** (**int** i=0;i<time;i++) {  
 labels[i]=Utility.*createProfLabel*(activity,professor,**groups**);  
 dragTextArea(labels[i]);  
 }  
 childrens = grid.getChildren();  
 **for** (Node node:childrens){  
 **if** (node.getClass()==pane.getClass()) {  
 actualPane = (StackPane) node;  
 **if** (!actualPane.getChildren().isEmpty()) {  
 **if** (actualPane.getChildren().get(0).getClass()=

=*draggingLabel*.getClass()) {  
 actualLabel=(IndexedLabel)actualPane.

getChildren().get(0);  
 **if** (actualLabel.getActivityId()=

=*draggingLabel*.getActivityId()) {  
 ((Pane) actualLabel.getParent())

.getChildren().remove(actualLabel);  
 }  
 }  
 }  
 }  
 }  
 **for** (**int** i=0;i<**HOURS**;i++)  
 **for** (**int** j=0;j<**DAYS**;j++){  
 **if** (activity.getIdActivity()=

=professor.getActivityProfessor(semester,i,j))  
 professor.setActivityProfessor(semester,i,j,-1);  
 **for** (**int** k = 0; k<Objects.*requireNonNull*(activity)

.getGroupsId().**length**; k++)  
 **if** (activity.getIdActivity()==**groups**.get (activity.getGroupsId()[k]).getActivityGroup(semester,i,j))  
 **groups**.get(activity.getGroupsId()[k]).setActivityGroup(semester,i,j,-1);  
 }  
  
**if** (row % 2 == 0)  
 add=-1;  
**else** add=1;  
**for** (**int** t=0;t<time;t++) {  
 X=row+(t%2)\*add;  
 Y=col+(t+1)/2;  
 childrens = grid.getChildren();  
 **for** (Node node:childrens) {  
 nodeX=GridPane.*getRowIndex*(node);  
 nodeY=GridPane.*getColumnIndex*(node);  
 **if** ( nodeX == X && nodeY == Y ) {  
 secondPane = (StackPane) node;  
 secondPane.getChildren().add(labels[t]);  
 }  
 **if** ( nodeX >= X && nodeY >= Y ) {  
 **break**;  
 }  
 }  
 professor.setActivityProfessor(semester,Y-1,X-1,activity

.getIdActivity());  
 **for** (**int** j = 0; j<activity.getGroupsId().**length**; j++) **groups**.get(activity.getGroupsId()[j]).setActivityGroup(semester,Y-1, X-1, activity.getIdActivity());  
}

**break**;

**case** 0:  
 grid=(GridPane) pane.getParent();  
 labels=**new** IndexedLabel[time];  
 **for** (**int** i=0;i<time;i++) {  
 labels[i] = Utility.*createProfLabel*(activity,professor,**groups**);  
 dragTextArea(labels[i]);  
 }  
 childrens = grid.getChildren();  
 **for** (Node node:childrens){  
 **if** (node.getClass()==pane.getClass()) {  
 actualPane = (StackPane) node;  
 **if** (!actualPane.getChildren().isEmpty()) {  
 **if** (actualPane.getChildren().get(0).getClass()=

=*draggingLabel*.getClass()) {  
 actualLabel=(IndexedLabel) actualPane.getChildren().get(0);  
 **if** (actualLabel.getActivityId()=

=*draggingLabel*.getActivityId()) {  
 ((Pane) actualLabel.getParent()).

getChildren().remove(actualLabel);  
 }  
 }  
 }  
 }  
 }  
 **for** (**int** i=0;i<**HOURS**;i++)  
 **for** (**int** j=0;j<**DAYS**;j++){  
 **if** (activity.getIdActivity()=

=professor.getActivityProfessor(semester,i,j))  
 professor.setActivityProfessor(semester,i,j,-1);  
 **for** (**int** k = 0; k< Objects.*requireNonNull*(activity).

getGroupsId().**length**; k++)  
 **if** (activity.getIdActivity()==**groups**.get

(activity.getGroupsId()[k]).getActivityGroup(semester,i,j))  
 **groups**.get(activity.getGroupsId()[k]).setActivityGroup(semester,i,j,-1);  
 }  
 **if** (row % 2 == 0)  
 row--;  
 **for** (**int** t=0;t<time;t++) {  
 X=row+t%2;  
 Y=col+t/2;  
 childrens = grid.getChildren();  
 **for** (Node node:childrens) {  
 nodeX=GridPane.*getRowIndex*(node);  
 nodeY=GridPane.*getColumnIndex*(node);  
 **if** ( nodeX == X && nodeY == Y ) {  
 secondPane = (StackPane) node;  
 secondPane.getChildren().add(labels[t]);  
 }  
 **if** ( nodeX >= X && nodeY >= Y ) {  
 **break**;  
 }  
 }  
 professor.setActivityProfessor(semester,Y-1,X-1,

activity.getIdActivity());  
 **for** (**int** j = 0; j<activity.getGroupsId().**length**; j++)  
 **groups**.get(activity.getGroupsId()[j]).setActivityGroup

(semester,Y-1, X-1, activity.getIdActivity());  
 }  
 **break**;

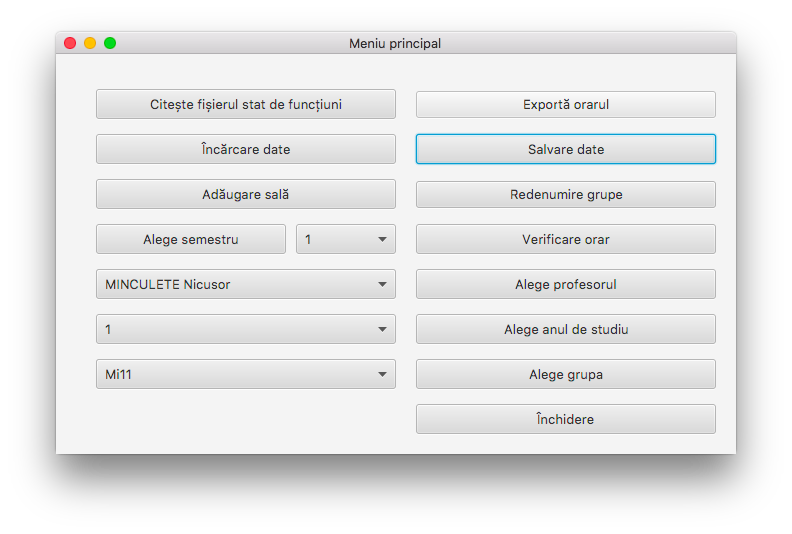
**default**:  
 **break**;  
 }  
}

Metoda în prima ei parte face mutarea grafică iar după aceea va face și modificările în matricele profesorilor, orelor. Din cauza reprezentării grafice și a logicii, activitățile cu durata de ore impare și cele cu durată de ore pare trebuie tratate diferit din punct de vedere logic și al draggingului. Din cauza asta metoda are două secțiuni apropiate din punctul de vedere al conținutului dar nu identice.

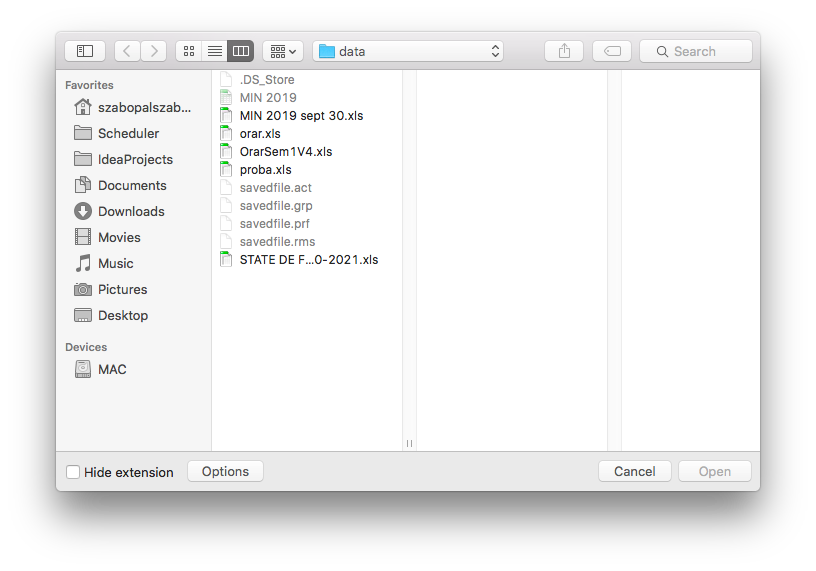
Metodele ajutătoare pentru mutarea în tabelele de orar a grupelor și cele pentru mutările din tabelul orar an/semestru sunt similare, apropiate.

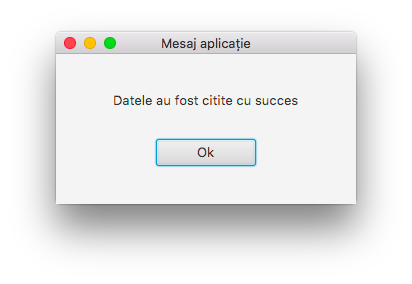
**CAPITOLUL III**

**APLICAȚIA AJUTORUL ORARISTULUI – GHID DE UTILIZARE**

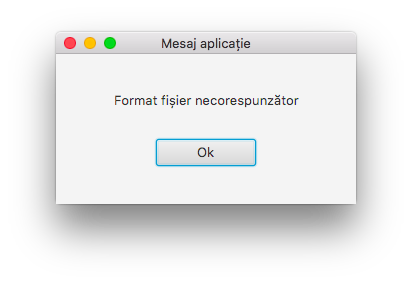
Aplicația ”Ajutorul Oraristului” este o aplicație Java care se pornește cu ajutorul fișierului ”Ajutorul oraristului.exe” și care ne afișează la pornire meniul principal.

Se poate vedea că aici găsim aproape toate funcțiile aplicației pe care le voi enumera acuma cu explicațiile de rigoare:

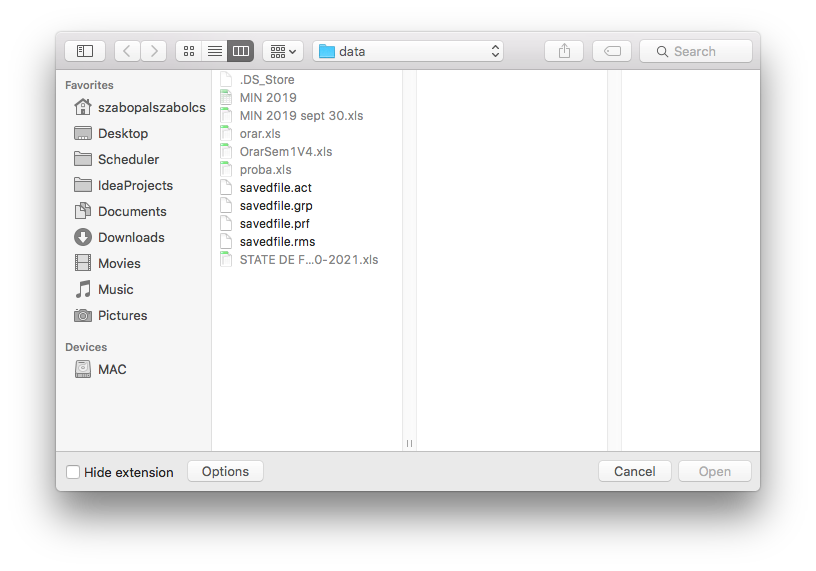
”Citește fișierul stat de funcții” va deschide o fereastră de citire fișier 

După cum se poate vedea, se deschide o fereastră în care apar accesibile doar fișierele excel 2007-2013 cu extensia .xls, cele care pot fi citite de către aplicație. Se accesează fișierul dorit și aplicația citește datele necesare. Dacă fișierul este potrivit apare mesajul de confirmare

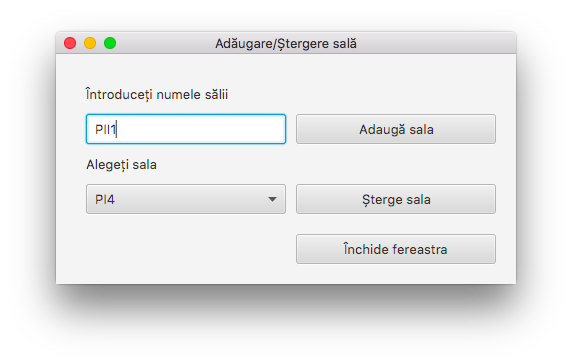
Dacă citirea datelor nu s-a reușit apare un alt mesaj



În cazul în care citirea datelor s-a efectuat cu succes putem continua lucrul cu datele citite.

”Încărcare date” – acest punct de meniu este pentru încărcarea de date din fișiere salvate în format json. Este vorba de date despre un orar în lucru, început la un moment dat dar nefinalizate și salvate. 

Salvarea datelor se face în format json în 4 fișiere ale căror nume sunt identice doar extensia variază. Se salvează de fapt toate obiectele de activități (*Activity*), profesori (*Professor*), grupe (*Group*) și sale de studiu (*Room*) în fișiere separate, acestea primind extensia de .act, .prf, .grp și .rms.

”Adăugare sală” – cu ajutorul acestui punct de meniu oraristul poate să adauge sălile de curs în baza de date a aplicației pentru a le putea folosi în la realizarea orarului.

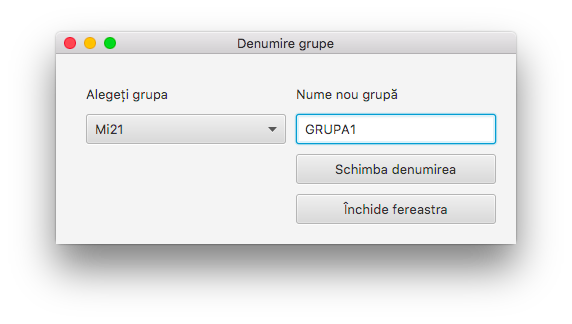
Se vede că se pot adăuga săli, dar se pot efectua și ștergeri la nevoie. Prin apăsarea butonului ”Șterge sala” se va șterge sala aleasă din lista sălilor existente.

”Alege semestru” – se alege semestrul de lucru 1 sau 2.

”Exportă orarul” – se exportă orarul în format excel, .xls, conform cu formatul din fișierele de orar folosite de Universitate. Se exportă orarul în formatul în care este pe pagina ”OrarSt”. Exportarea se poate face ori de câte ori oraristul crede de cuviință, însă pentru păstrarea datelor, pentru reluarea lucrului într-o altă sesiune în viitor, se recomandă ”Salvare date” – care salvează datele prelucrate în fișiere de tip json, care se pot după aceea deschide cu punctul de meniu deja prezentat ”Încărcare date”. Cele două tipuri de salvare a datelor deschid câte o fereastră de selecție/denumire fișier de salvat, după care se va trece la scrierea/salvarea datelor în formatul corespunzător. După cum am mai arătat, salvare în format json se face de fapt în patru fișiere distincte pentru obiectele diferite folosite de aplicație.

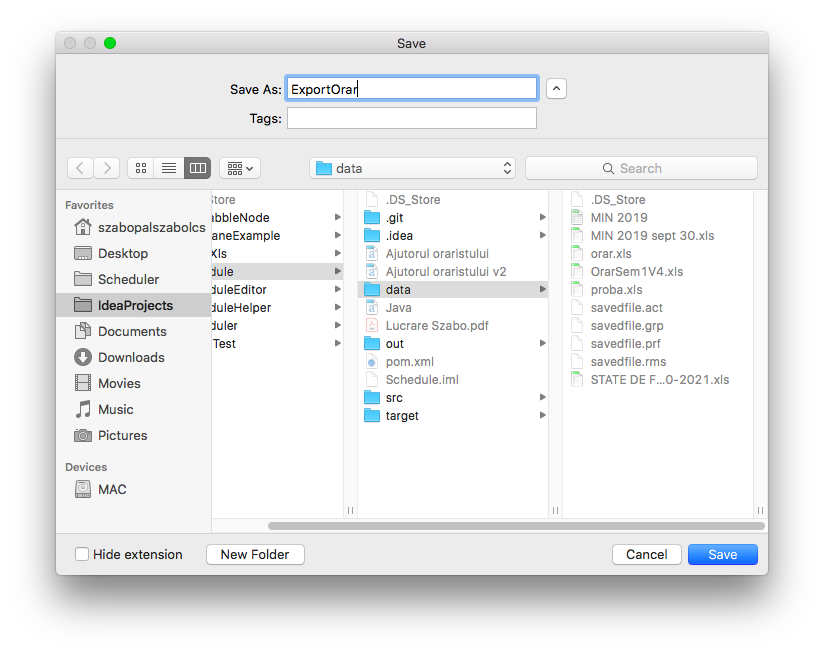
”Verificare orar” – este un punct de verificare a orarului în sensul în care acest buton de meniu va verifica dacă există activități care încă nu au fost plasate pe orar, câte și care sunt acestea, respectiv va verifica dacă sunt activități cărora nu le-au fost încă atribuite săli de curs. Efectiv vom avea două liste de activități pe care va trebui să le parcurgem și să le găsim loc și sală pentru ca orarul să poată fi considerat finalizat.

O altă funcție este cea a redenumirii grupelor, ”Redenumire grupe”, pentru a se putea redenumi grupele de către orarist, numele grupelor fiind date de către aplicație pe baza specializării, anului de studiu și numărul grupei depinde câte grupe sunt pe an). Aceste denumiri se pot modifica în funcție de codificarea fiecărei facultăți.

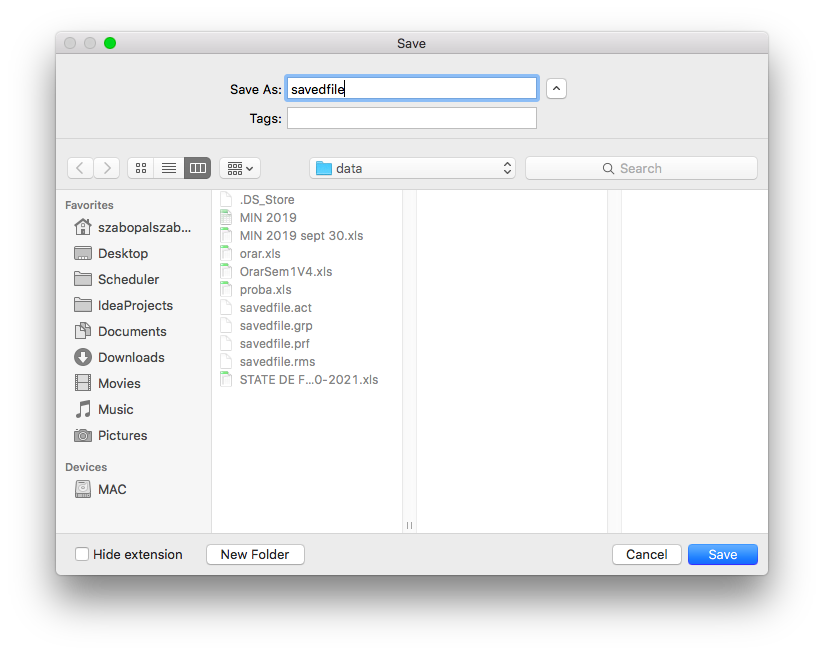


Pentru salvarea datelor și salvarea rezultatului final avem două puncte în meniu similare celor de citire și încărcare date.

”Exportă orarul” va exporta în format Excel orarul pe ani în stadiul actual într-un format care se va putea folosi în fișierele Excel ale Universității pentru prelucrare în continuare. Oraristul va putea pur și simplu copia din fișierele exportate datele în fișierul de lucru Excel.



"Salvare date” este opțiunea prin care se salvează datele prelucrate într-un anumit stadiu și care pot fi după aceea reîncărcate pentru continuarea lucrului. Aceste date se salvează în 4 fișiere cu aceeași denumire, dar extensii diferite, pentru cele 4 clase de date folosite, adică *Activity, Professor, Group* și *Room.* Utilizatorul va trece o singură denumire, iar aplicația scrie 4 fișiere. La încărcare se poate alege oricare dintre cele 4 fișiere, dar se citesc toate cele 4.

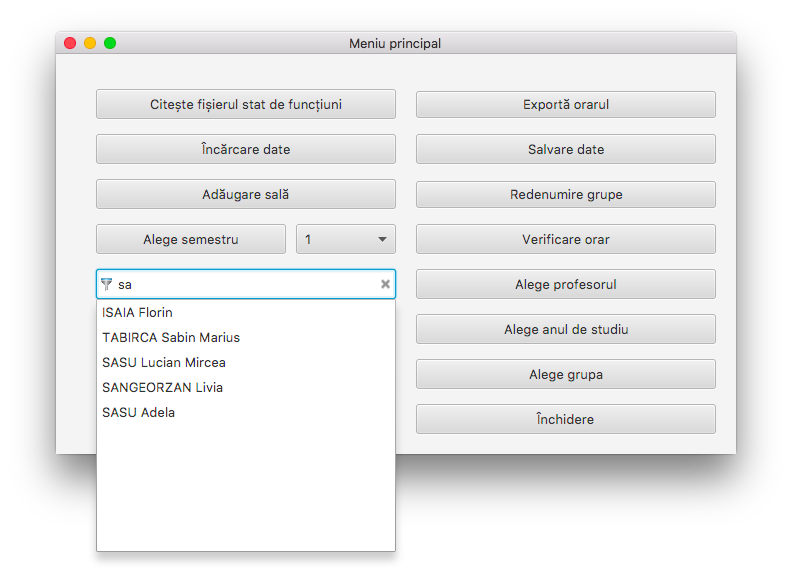


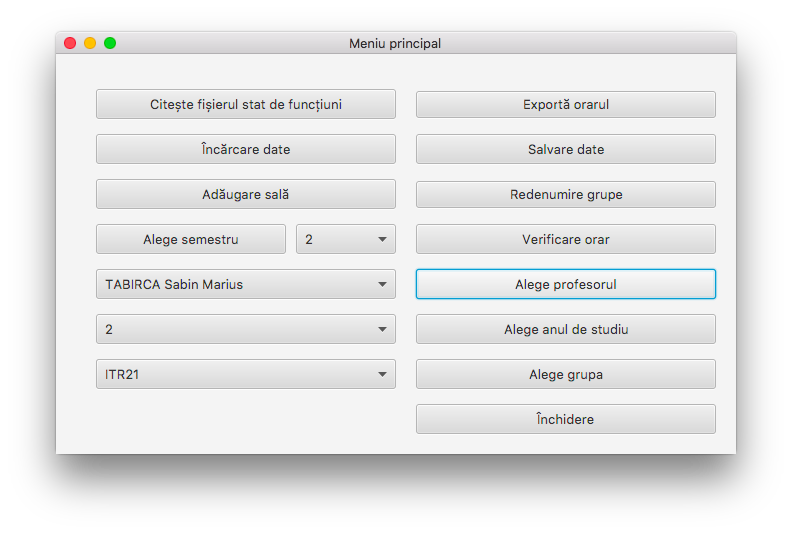
În figura de mai sus se pot observa cele 4 fișiere cu același nume și extensii diferite.

Urmează acuma cea mai importantă parte și anume mișcarea activităților pe și între tabelele de orar ale profesorilor, grupelor și anilor de studii. Acestea se vizualizează prin apăsarea butoanelor

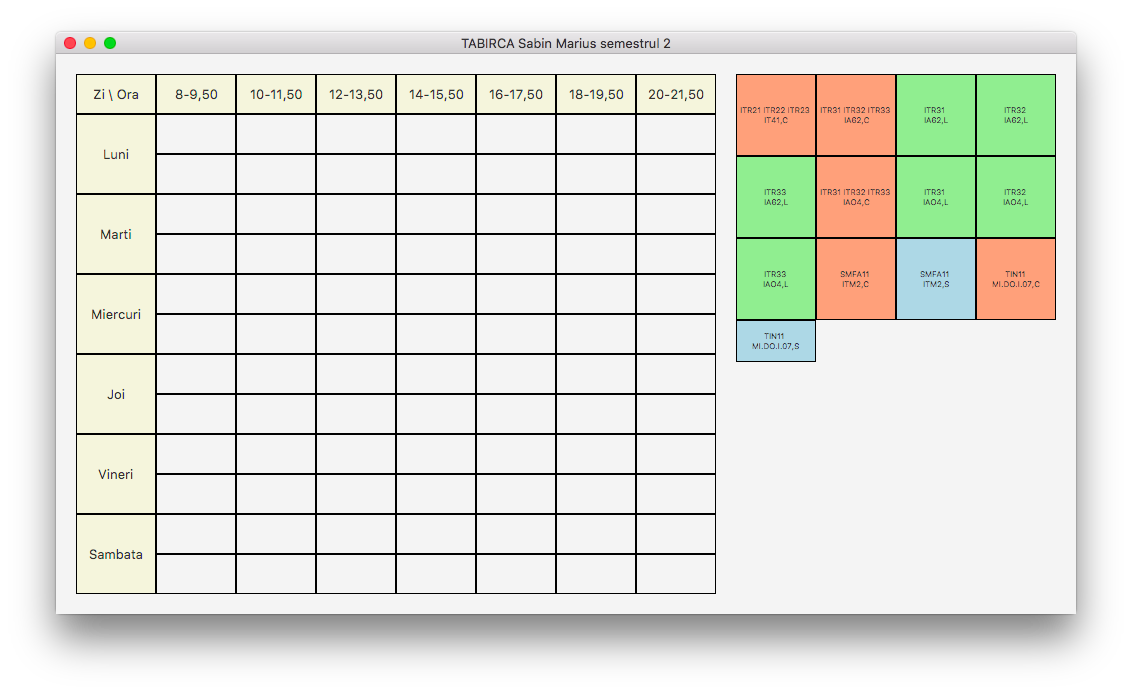
* ”Alege profesorul”
* ”Alege anul de studiu”
* ”Alege grupa”

Fiecare buton de alegere are în stânga un ComboBox unde se poate alege profesorul, anul de studiu sau grupa pe care urmează s-o alegem. În cazul profesorilor și grupelor avem un SearchableComboBox care are proprietatea de a filtra rezultatele în funcție de ceea ce tastăm în câmpul editabil al ComboBox-ului, de exemplu dacă trecem ”sa” vor fi listate în continuare profesorii care au în nume undeva, oriunde, combinația de litere ”sa”.



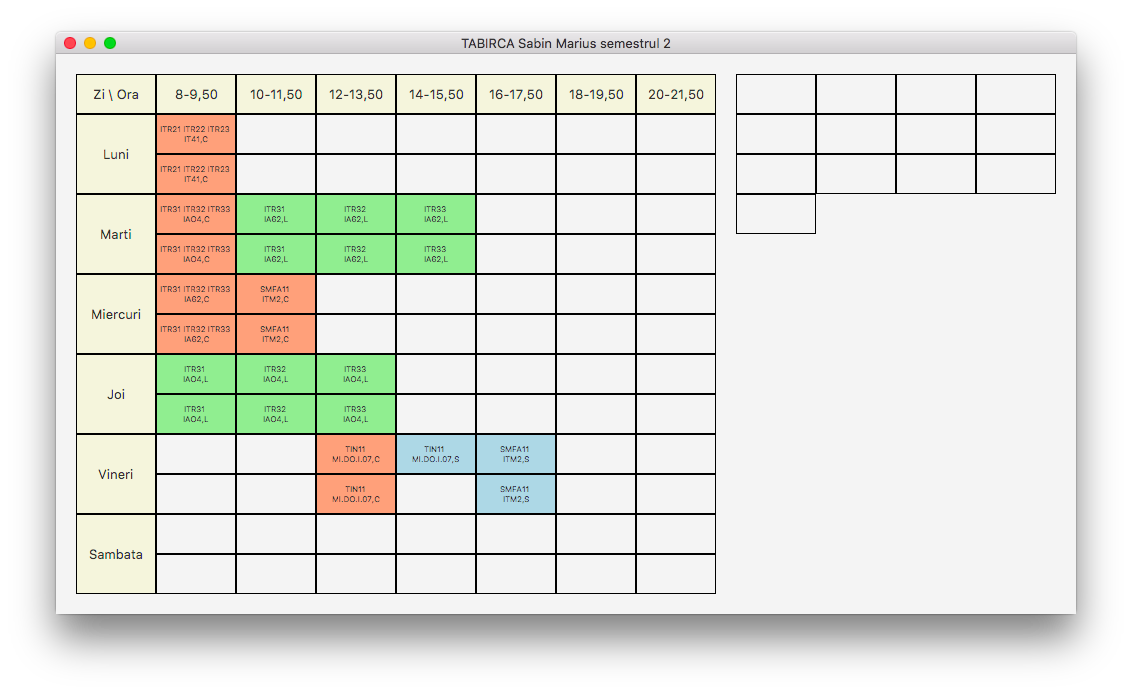
 Acest lucru facilitează căutarea mai ușoară în lista profesorilor, destul de lungă pentru o căutare fără ajutorul oferit de acest ComboBox.

După ce se alege unul dintre profesori și numele acestuia apare în ComboBox-ul dedicat se poate apăsa butonul din dreapta ”Alege profesorul” și se va afișa tabelul orar al profesorului respectiv pentru semestrul selectat mai sus.

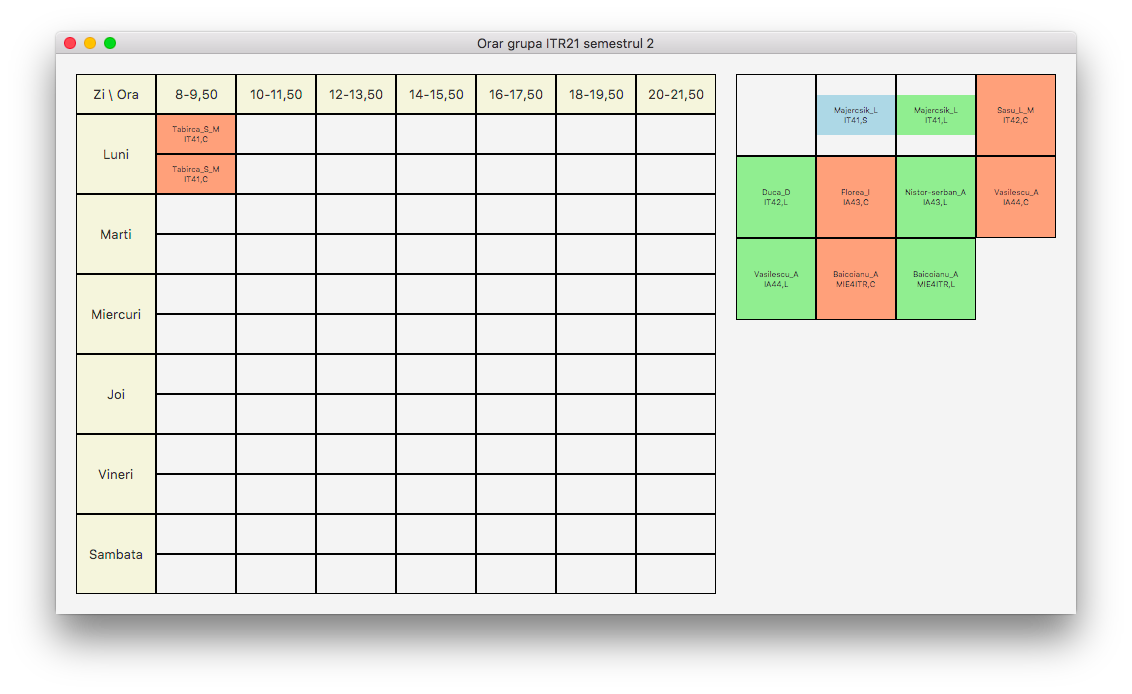
În figura de mai sus este reprezentat orarul unui profesor unde avem următoarele elemente:

* titlul acestei ferestre care are formatul Nume Profesor + Semestru + numărul semestrului
* tabelul orar al profesorului în partea stângă, tabel care este gol la începutul sesiunii de lucru
* tabelul cu activitățile profesorului din semestrul de lucru, acesta variază de la profesor la profesor, depinde câte activități are fiecare profesor în semestrul de lucru

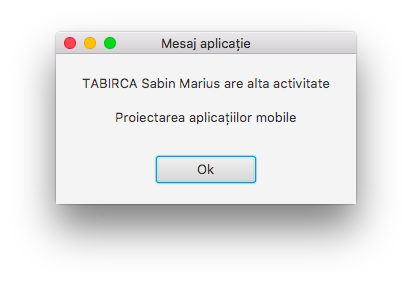
În momentul în care toate orele (activitățile) unui profesor au fost puse în orar, orarul acestuia poate arăta de exemplu în felul următor:



În același mod se poate alege una dintre grupe pentru a se edita orarul grupei respective. În ComboBox-ul din stânga se alege grupa, după care se apasă pe butonul ”Alege grupa” și se deschide tabelul orar al grupei alese.

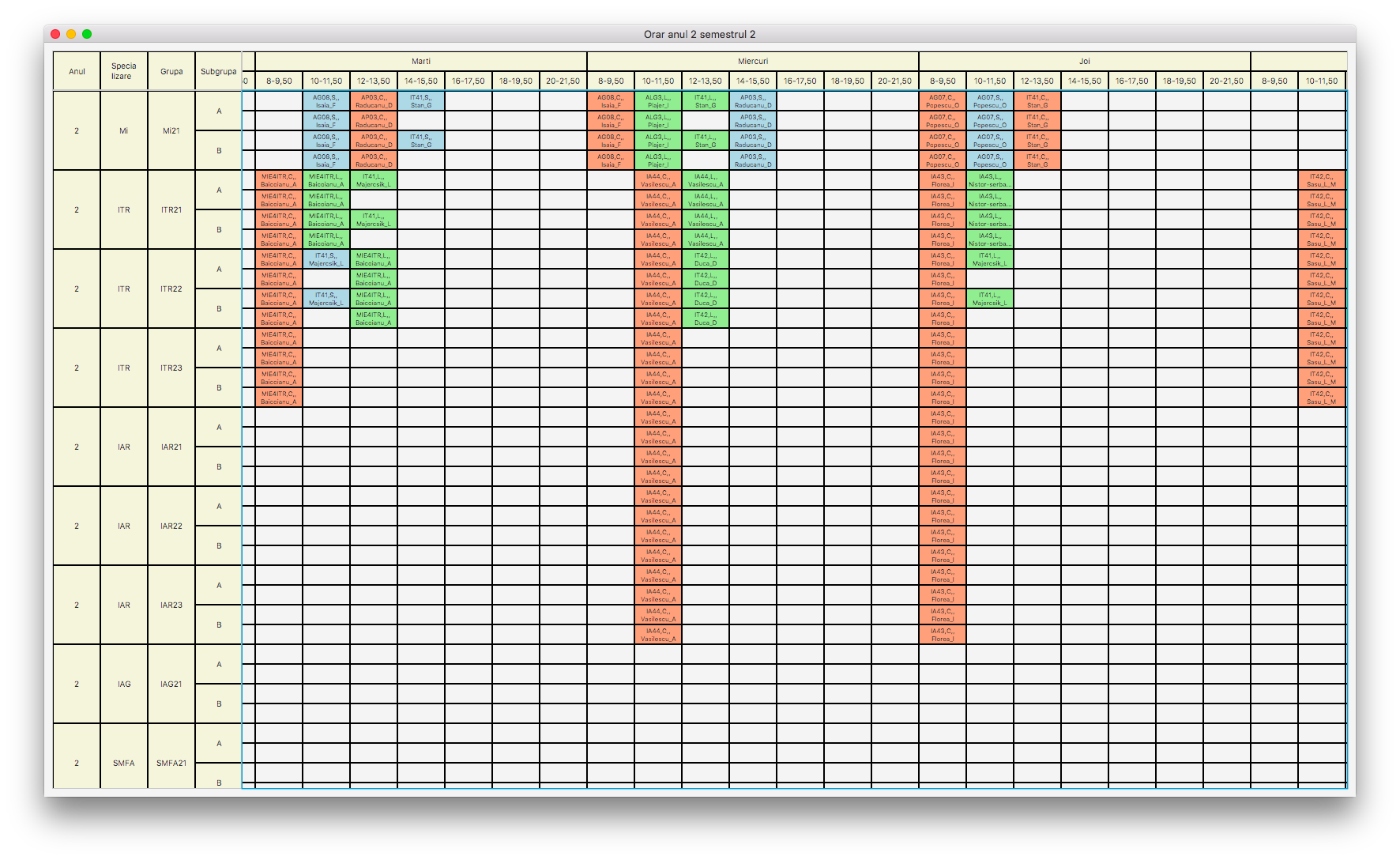


După cum se poate observa luni între orele 8-9,50 apare un curs cu Dl Tabarca, curs care apare ca rezultat al editării orarului acestuia. Acest curs se poate muta și în acest tabel sau în tabelul profesorului sau chiar în tabelul mare pe ani la, care urmează să fie prezentat. Desigur programul verifică la fiecare mutare, dacă nu cumva avem de-a face cu vreo coliziune, adică dacă nu cumva profesorul sau grupa au deja activități programate în ziua respectivă și la ora aleasă, sau dacă sala de curs unde dorim să se desfășoare activitatea nu este liberă în perioada respectivă. Dacă avem de a face cu vreuna dintre aceste situații, aplicația ne va atrage atenția și mutarea nu are loc. Dacă de exemplu încercăm să mutăm cursul de luni pe marți la ora 8,00 o să apară un mesaj de la aplicație sub forma:

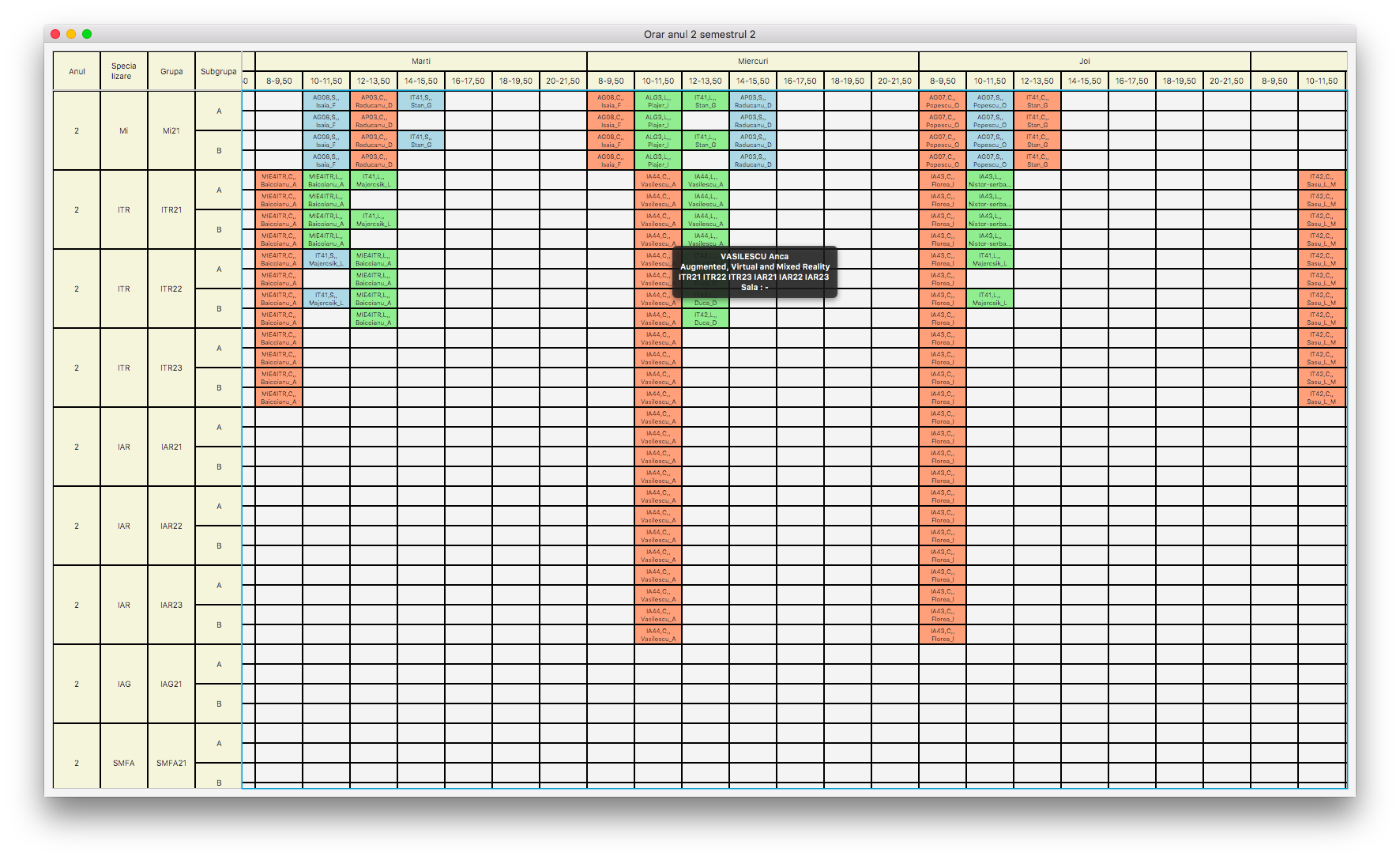


Aflăm, deci, că marți la ora 8,00 Tăbârcă Sabin Marius are altă activitate și anume Proiectarea Aplicațiilor Mobile. Prin apăsarea butonului ”Ok” fereastra de mesaj se închide și ne putem continua treaba, cursul nu este mutat, rămâne pe poziția inițială, până îi găsim un loc unde nu este în coliziune cu nici un alt curs, din niciun punct de vedere (profesor, grupă, sală).

Prin alegerea anului de studiu dorit, în ComboBox-ul din stânga și apăsarea butonului din dreapta ”Alege anul de studiu” se va deschide tabelul orar al anului de studiu selectat. Acesta este un tabel mai mare decât cele de până acuma, însă oferă o privire de ansamblu asupra orarului facultății. Cu cât este mai mare rezoluția monitorului folosit, cu atât se va afișa mai mult din orar. Acest tabel este Scrollable, adică derulabil, se poate derula stânga-dreapta și sus-jos. Mai jos un astfel de tabel:



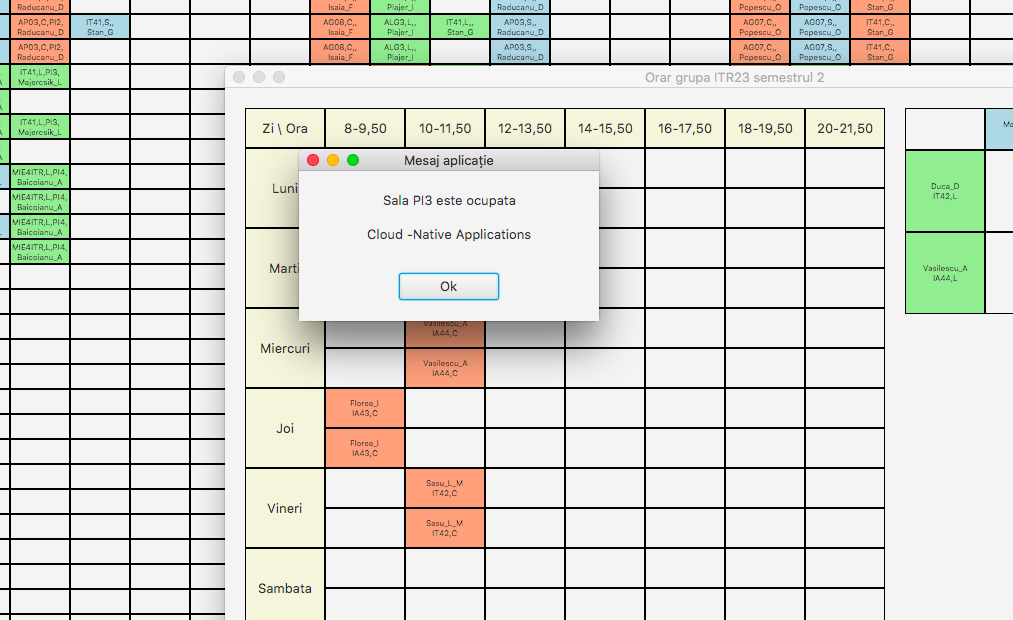
În exemplul de mai sus avem orarul anului II, în stânga avem grupele și subgrupele aferente iar în sus avem zilele/orele. Când se derulează tabelul se derulează astfel încât headerul și câmpurile din stânga care conțin explicațiile sunt fixate. Dat fiind faptul că etichetele activităților și textul afișat pe aceste etichete este destul de mic și sumar, în momentul în care se oprește cursorul mouseului deasupra vreunei activități, va apărea o fereastră temporară cu descrierea activității respective, după cum se vede în imaginea de mai jos



Și pe acest tabel se pot mișca activitățile și pot apărea situații în care avem coliziuni, moment în care primim mesaj de la aplicație cu descrierea problemei.

Alegerea sălii unei activități se poate face prin click dreapta pe activitate, acest click va duce la deschiderea unei ferestre cu alegerea sălii pentru activitatea în cauză. Aici avem două scenarii:

* Dacă activitatea este deja pe orar, deci este setată o zi și oră de desfășurare a activității atunci în fereastra de alegere a sălii ni se oferă o listă de săli disponibile în ziua și ora respectivă
* În cazul în care activitatea nu este pe orar, se poate selecta orice sală, iar în momentul în care activitatea este trasă într-un tabel orar se verifică dacă în ziua/ora respectivă sala nu este deja ocupată



Alte funcționalități:

* Cu dublu click pe orice activitate se va deschide orarul profesorului titular al activității respective
* Cu tasta ”esc” se accesează meniul principal
* Cu tasta ”tab” se poate naviga între ferestrele deschise ale aplicației
* Prin apăsarea butonului ”Închidere” din meniul principal se va închide aplicația.

Bibliografie

Eugen Rotariu – Limbajul Java, Editura Agora, 1996

https://docs.oracle.com/javase/8/javafx/get-started-tutorial/

https://maven.apache.org/guides/index.html

https://sites.google.com/site/gson/gson-user-guide

https://poi.apache.org/index.html