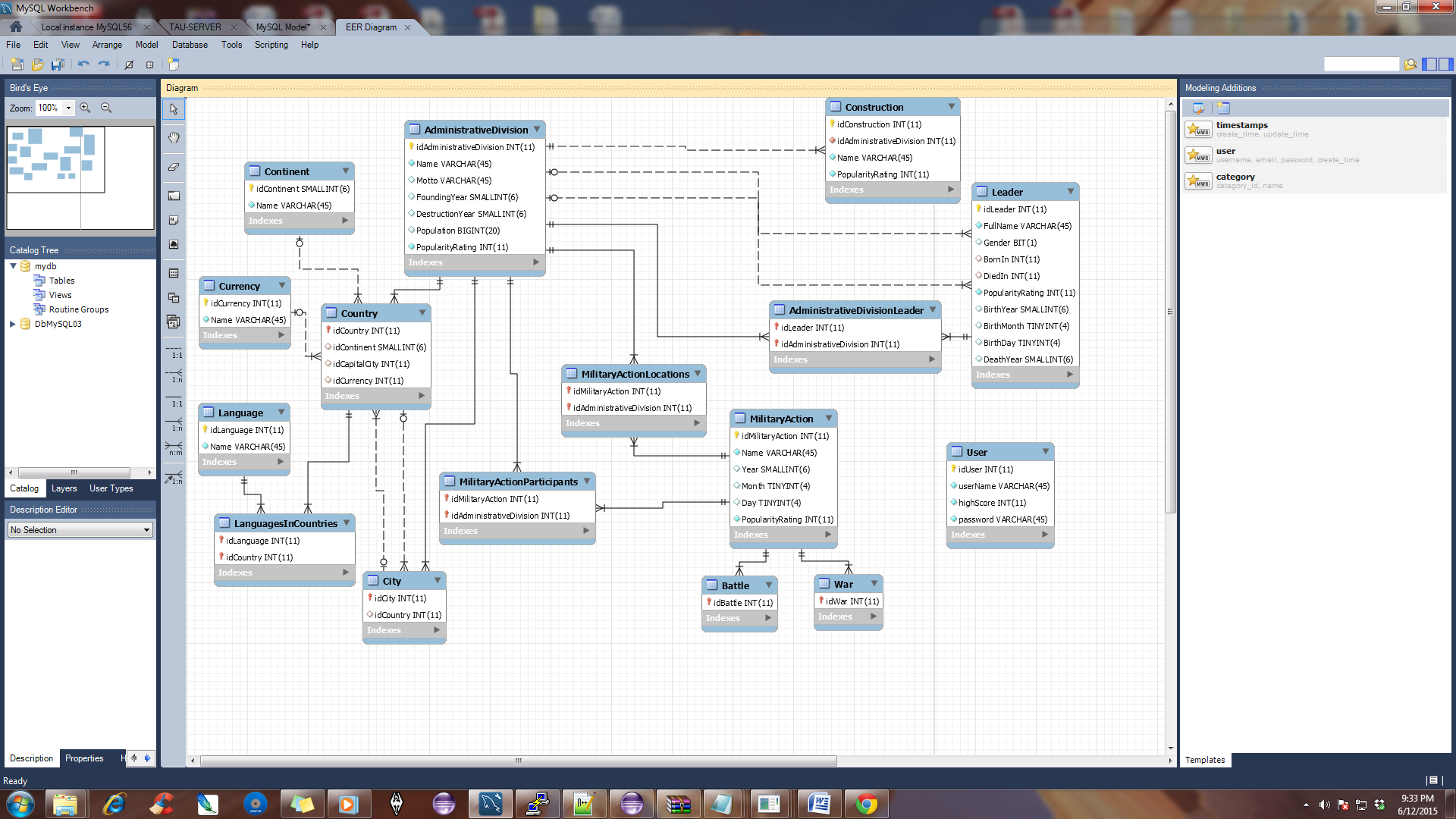
Software documentation

**1. מבנה מסד הנתונים**

**1.1 תרשים הסכימה**

..

**1.2 תיאור הרלציות**

**הערות כלליות**

1. **Indexes**: בנוסף לאינדקסים הרגילים שהקיום שלהם מחויב על ידי primary and foreign keys, רלציות שכוללות שדה שם, מגדירות אינדקס עבור שדה זה על מנת לייעל ביצועי שאילתות SELECT לפי השם כחלק מהאפשרות שניתנת לשחקן לערוך רמזים הניתנים לו במהלך המשחק. בנוסף לכך, האינדקסים הללו הם מסוג UNIQUE, על מנת למנוע כפילויות בשמות.

הרלציות הללו הן: Construction, User, Language, Currency, Continent, MilitaryAction, AdministrativeDivision .

הערה: גם User מופיע ברשימה. מוגדר אינדקס UNIQUE עבור השדה userName, כלומר לא יתכנו שני משתמשים\שחקנים בעלי אותו שם משתמש.

בנוסף לכך, עבור הרלציות Leader ו-AdministrativeDivision מוגדר אינדקס רגיל (כלומר מטיפוס INDEX) עבור השדה שם (מאותה סיבה בדיוק, על מנת לייעל שאילתות SELECT לפי שם). עבור Leader, הסיבה שהאינדקס אינו UNIQUE ברורה: יתכנו בני אדם עם אותו השם. עבור AdministrativeDivision, הסיבה שהאינדקס אינו UNIQUE, היא שיתכנו עיר ומדינה בעלי אותו שם, ולמעשה יתכנו גם ערים שונות בעלי אותו שם ומדינות\states שונות בעלות אותו השם.

כמו כן, ברלציות בהן קיים השדה `PopularityRating`, השדה הוא בהכרח שדה חובה (NOT NULL) ומוגדר עבורו אינדקס (מסוג INDEX רגיל) על מנת לשפר ביצועי שאילתות שמחפשות וממיינות רשומות לפי הפופולאריות שלהן. לדוגמה ברמות קושי נמוכות, הרמזים שהאפליקציה תציג יתבססו על נתונים בעלי רמת פופולאריות גבוהה ולהפך, ברמות קושי גבוהות הרמזים יתבססו על נתונים פחות פופולאריים. הערה: הפופולאריות היא אורך עמוד ה- Wikipedia המתאים ל-entity, ערך שYAGO מספק (article עשיר יותר, entity מפורסם יותר).

הרלציות הללו הן: Construction, Leader, MilitaryAction, AdministrativeDivision.

2. **ייצוג תאריכים**: ברלציות Leader ו- AdministrativeDivision בחרנו לייצג תאריך על ידי פירוקו למרכיביו: month, day ו-year. הסיבה לכך היא שהטיפוסים הקיימים ב-mySQL עבור תאריכים מגבילים את טווחי התאריכים האפשריים, ולא מסוגלים לייצג תאריכים ישנים (שכן מופיעים ב-YAGO) ולכן כדי לתמוך בתאריכים הללו החלטנו לשמור את המרכיבים בנפרד (אם כי הקפדנו לא לבחור טיפוסים בזבזניים עבור כל אחד מהם).

הערה: עבור תאריכים לפנה"ס אנו שומרים שנה שלילית

3. **is a:** ב-design שלנו השתמשנו ביחס **‘is a’** כפי שנלמד בכיתה:

* City ‘is an’ AdministrativeDivision
* Country ‘is an’ AdministrativeDivision
* Battle ‘is a’ MilitaryAction
* War ‘is a’ MilitaryAction

הסיבות שבחרנו לעצב את הסכימה בצורה זו:

1. מאפשר לתאר עמודות משותפות ברלציה אחת בלבד וכן לבצע שינויים של עמודות משותפות ברלציה אחת בלבד.
2. מאפשר להתייחס לסוגים שונים של entities (בעלי הורה משותף) בצורה אחידה. לדוגמה Leader יכול להנהיג גם ערים וגם מדינות, ואנו מסוגלים לשמור מידע זה בטבלה אחת AdministrativeDivisionLeader.

**תיאור הרלציות עצמן:**

* **Currency**– שומרת סוגי מטבע (שם Name ומזהה ייחודי idCurrency).
* **Language**– שומרת שפות (שם Name ומזהה ייחודי idLanguage).
* **Continent**– שומרת יבשות (שם Name ומזהה ייחודי idContinent).
* **AdministrativeDivision** – הכללה של מדינות, ערים וכל מסגרת טריטוריאלית אדמיניסטרטיבית אחרת. שומרת מזהה ייחודי (idAdministrativeDivision) השם של המקום (Name), Motto (אם קיים), שנה בה נוסדה (FoundingYear, אם המידע קיים), שנה בה הושמדה\הפסיקה להתקיים (DestructionYear, אם המידע קיים), גודל אוכלוסייה (Population, אם המידע קיים) ולבסוף את "דירוג הפופולאריות" (PopularityRating).

ישות מסוג זה עשויה להשתתף במלחמות וקרבות, מלחמות וקרבות עשויים להתרחש בשטח הגיאוגרפי שלה, עשויים להיות לה מנהיגים ומבנים מפורסמים עשויים להימצא בתוכה. באפליקציה שלנו הישויות שאת שמם צריך לנחש הן כולם מסוג AdministrativeDivision (כלומר ערים ומדינות).

* **Country**– שומרת מדינות. כאמור Country ‘is an’ AdministrativeDivision, לכן idCountry (שהוא ה-primary key) הוא foreign key ל- AdministrativeDivision.idAdministrativeDivision.

הרלציה שומרת את היבשת בה המדינה נמצאת (idContinent מפתח זר ל-Continent.idContinent) אם המידע קיים, את עיר הבירה של המדינה (idCapitalCity מפתח זר ל-City.idCity) אם המידע קיים ואת המטבע בו משתמשים במדינה (idCurrency מפתח זר ל- Currency.idCurreny) אם המידע קיים.

* **City**– שומרת ערים. כאמור City ‘is an’ AdminstrativeDivision, לכן idCity (שהוא ה-primary key) הוא foreign key ל- AdministrativeDivision.idAdministrativeDivision.

כמו כן, הרלציה שומרת את המדינה בה העיר נמצאת (idCountry מפתח זר ל-Country.idCountry), אם המידע קיים.

* **LanguagesInCountries** – טבלה מקשרת בין מדינות לשפות, אילו שפות רשמיות יש בכל מדינה (יתכן כי למדינה יש יותר משפה רשמית אחת). idLanguage מפתח זר ל-Language.idLanguage ו-idCountry מפתח זר ל-Country.idCountry.
* **Construction** – מבנים מפורסמים שיש בערים או מדינות. הרלציה שומרת מזהה ייחודי (idConstruction), את ה-administrativeDivision בו המבנה נמצא (idAdministrativeDivision מפתח זר ל- AdministrativeDivision.idAdministrativeDivision, בנוסף זהו שדה חובה not null), השם של המבנה (Name), ומדד הפופולאריות (PopularityRating).
* **Leader** – מנהיגים של ערים ומדינות. הרלציה שומרת מזהה ייחודי (idLeader), את השם המלא של המנהיג (FullName), מקום לידה (BornIn מפתח זר ל - AdministrativeDivision.idAdministrativeDivision) אם המידע קיים, מקום מיתה (DiedIn מפתח זר ל- AdministrativeDivision.idAdministrativeDivision) אם המידע קיים, תאריך לידה (BirthYear, BirthMonth, BirthDay) אם המידע קיים, שנת מיתה (DeathYear) אם המידע קיים, ואת מדד הפופולאריות (PopularityRating).
* **AdministrativeDivisionLeader** – טבלה מקשרת בין ה-Leaders וה-AdministrativeDivisions, אילו מנהיגים הינם מנהיגים של (או הנהיגו בעבר) אילו ערים\מדינות (ייתכן שמנהיג אחד הנהיג\מנהיג יותר מעיר אחת, יותר ממדינה אחת, או גם וגם). idLeader מפתח זר ל- Leader.idLeader, idAdminstrativeDivision מפתח זר ל-AdministrativeDivision.idAdminstrativeDivision.
* **MilitaryAction** – הכללה של פעולה צבאית (מלחמה, קרב וכד'). הרלציה שומרת מזהה ייחודי (idMilitaryAction), שם (Name), תאריך התרחשות (Year, Month, Day – תאריך בו הפעילות החלה, אם המידע קיים) ומדד פופולאריות.
* **Battle** – רלציה שומרת קרבות. כאמור Battle is a MilitaryAction ולכן idBattle (שהוא ה-primary key) הוא מפתח זר ל-MilitaryAction.idMilitaryAction
* **War** – רלציה שומרת מלחמות. כאמור War is a MilitaryAction ולכן idWar (שהוא ה-primary key) הוא מפתח זר ל- MilitaryAction.idMilitaryAction.
* **MilitaryActionLocations** – טבלה מקשרת בין MilitaryAction לבין AdministrativeDivision. שומרת היכן הפעולות הצבאיות התרחשו (פעילות צבאית עשויה להתרחש ביותר מישות גיאוגרפית אחת)
* **MilitaryActionParticipants** – טבלה מקשרת בין MilitaryAction לבין AdministrativeDivision. שומרת אילו ערים או מדינות השתתפו באילו פעילויות צבאיות.
* **User –** טבלת משתמשים (שחקנים) רשומים. הרלציה שומרת לכל משתמש רשום מזהה ייחודי (idUser), שם משתמש שהוא unique (userName, לא יכול להופיע בטבלה יותר מפעם אחת), את הניקוד הגבוה ביותר של המשתמש (highScore) ואת הסיסמא (password). שם משתמש וסיסמא הם שדות חובה (not null).

**2. מבנה הקוד**

**Main**

זו המחלקה ממנה מתחילה פעולת התוכנית (מהפונקציה main).

מה שהפונקציה main עושה זה לקרוא את קובץ הקונפיגורציה (עם הפונקציה loadConfig, שגם היא נמצאת בMain) ואז לקרוא לפונקציה שמאתחלת את החלון ההתחלתי של התוכנית בUIMain.

משם הפונקציות של UIMain משתלטות על העניינים.

**2.1 חבילת core**

**GameManager**

מחלקה האחראית על ניהול המשחק (מאחורי הUI ) , היא מחזיקה את השדות השונים הנחוצים לניהול והצגת מידע רציף לשחק. מחלקה זו היא הקשר בין הUI לCore של הפרויקט, היא מאפשרת גישה לרמזים הנאספים על המקום אותו מנסים לגלות , מספר הטעויות שהשחקן ביצע ומידע על ניצחון או הפסד. כמו כן היא גם מנהלת את ניחוש האותיות של השחקן ומחזירה את האותיות שהשחקן גילה בהצלחה עד כה.

בנוסף מחלקה זו שומרת ומעדכנת את הניקוד של השחקן, הניקוד מחושב ביחס הפוך ממספר הטעויות שהשחקן עשה עבור כל מקום בנפרד () ואחראית להשיג לעצמה מקום חדש עבור המשחק למקרה והשחקן יצליח לגלות את המקום הנוכחי.

**Location**

מחלקה המייצגת מיקום אותו מנסים לגלות (רשומות ב-AdministrativeDivision). Instance של המחלקה מחזיק פרטים על המיקום:

* name- השם של המיקום אותו צריך לגלות.
* Id- הid של המיקום הDB לצורך עדכון נתונים בDB.
* List of clues- רשימת רמזים שיעזרו לגלות את המיקום.

בנוסף המחלקה מספקת factory method המייצרת מיקום חדש שהשחקן יצטרף לגלות, המקום מוגרל מתוך טבלת המקומות בDB (AdministrativeDivision ) לפי רמת המשחק , כאשר במשחק קל יותר נגריל מקומות "מוכרים יותר" (בעלי popularity rating –שדה שנאסף לפי כמות המידע הקיים עליו ב Wikipedia – גבוה יותר) ובמשחק קשה מקומות "מוכרים פחות".

**Player**

מחלקה המאפשרת לקבל מידע הנוגע לשחקן מה-DB. בעזרת הפונקציות של המחלקה מבצעים registration של שחקנים חדשים ובודקים התחברות של שחקנים קיימים. בנוסף מאפשרת לקבל מידע על טבלת ניקוד של השחקנים המובילים והניקוד של השחקן המחובר כעת.

**Clues**

לכל מיקום במשחק אנחנו מוציאים מהDB רשימה של רמזים שיעזרו לשחקן לגלות את השם של המיקום , אותם אנחנו מאפשרים לשנות (ברוב המקרים) במקרה והשחקן חושב שהם שגויים.

הרמזים מתחלקים למספר סוגים לפי הדרך בה הם נשמרים בDB, המידע המתלווה לאותו רמז ואיך מקבלים אותו מה-DB.

רמזים מיוצרים רק אם המידע הנחוץ על מנת לייצר אותם קיים עבור המיקום שנבחר.

**ClueType-** enum המחזיק את סוגי הרמזים השונים והשאילתות המתאימות על מנת להשיג את המידע הדרוש לבנייתם מה-DB.

**ClueFactory-** "מפעל" המייצר את הרמזים השונים כל אחד לפי המידע המתקבל מה-DB בהתאם לשאילתת הייצור שנמצאת ב-ClueType.

**AbstractClue**- מייצג רמז כללי הכולל:

* + - *Clue*- הרמז שיוצג למשתמש.
    - *locationId*- הID של המיקום עליו הרמז מבוסס.
    - *ID*-הID של המידע בDB ממנו הוצאנו את הרמז. (לדוגמא : עבור רמז של מנהיג נשמור את הID של אותו מנהיג בDB )

**ClueField**- רמז המבוסס על מידע הנמצא בטבלה AdministrativeDivision (ראה DB לפירוט) ומייצג מידע בסיסי על המיקום. את המידע הזה אנחנו נותנים לשחקן לערוך במקרה והוא חושב והוא שגוי.

* + - **PopulationClue** –מספר התושבים באותו מיקום.
    - **MottoClue**- המוטו של אותו מיקום.
    - **DestructionClue**- שנת ההריסה\השמדה של המקום.
    - **FoundingClue**- שנת הקמה\ייסוד של המקום.

**ReplaceableClue-** רמז המציג מידע שנאסף דרך קשרים many to one עם המיקומים השונים.במידה והשחקן חושב שמידע זה שגוי אנחנו מאפשרים לשחקן להחליף אותו במידע הנכון *הקיים* ב-DB, כלומר אנחנו מאפשרים לשחקן לחפש בDB (בטבלה המתאימה לפי כל רמז) עבור התשובה הנכונה לדעתו והוא צריך לבחור מתוך האופציות שמתאימות למה שחיפש מתוך הDB.

* + - **CurrencyClue**- מייצג מידע על המטבע בו משתמשים באותו מקום (רק עבור מדינות).עריכה מאפשרת להחליף את המטבע.
    - **CountryClue**- מייצג מידע על המדינה בה מיקום זה נמצא (עבור ערים).עריכה מאפשרת להחליף את המדינה.
    - **ContinentClue**- מייצג מידע על היבשת בה מיקום זה נמצא (רק עבור מדינות)**.** עריכה מאפשרת להחליף את היבשת.
    - **ConstructionClue**- מייצג מידע על מבנה הנמצא במיקום זה (הערה: זהו מקרה יוצא דופן, משום שבניגוד לשאר הרמזים, הטבלה שנמצאת בצד ה-many ביחס היא טבלת המבנים Construction. עריכה של רמז זה מאפשרת לשנות את המיקום בוא *המבנה* נמצא)
    - **CapitalClue**- מייצג מידע על עיר הבירה של אותו מיקום (עבור מדינות).עריכה מאפשרת להחליף את עיר הבירה.

**ExtensibleClue-** רמז המציגמידע שנאסף דרך טבלאות הקישור השונות בDB- ונמצא בקשר של many-to-many עם המיקום. במידה והשחקן חושב שמידע זה שגוי ניתנת לו האפשרות להסיר קשרים קיימים ולהוסיף קשרים חדשים עבור ישויות קיימות ב-DB, כלומר אנחנו מאפשרים לשחקן להסיר ישויות שקשורות לאותו רמז (מתוך אלו המקושרות אליו כבר) ואף להוסיף ע"י חיפוש בDB (כל רמז בטבלה המתאימה לו) אופציות לקשר למידע של רמז זה.

* + - **WarClue-** מייצג מידע על מלחמה שהתרחשה במיקום זה.עריכה מאפשרת למחוק ולהוסיף מלחמות שהתרחשו למיקום הנוכחי.
    - **ParticipantsMilitaryActionClue-** מייצג מידע על פעולה צבאית שמיקום זה *השתתף*\*לחם* בה, ומציג גם מקומות אחרים שהשתתפו באותו פעולה.עריכה מאפשרת להסיר או להוסיף משתתפים מהפעולה הצבאית שהרמז מתייחס אליה.
    - **LeaderClue**- מייצג מידע על מנהיג שפעל המיקום זה.עריכה מאפשרת להוסיף\להסיר מנהיגים מהמיקום המדובר ברמז.
    - **LanguageClue**- מייצג מידע על שפה רשמית המדוברת במיקום זה.עריכה מאפשרת להוסיף\להסיר שפות מהמיקום המדובר.
    - **BattleClue**- מייצג מידע על קרב שהתרחש במיקום זה.עריכה מאפשרת להוסיף\להסיר קרבות שהתחרשו במיקום המדובר ברמז.

**NECLue-** רמז המציג מידע על המיקום שנאסף דרך קשרים עקיפים בDB ולכן לא ניתן לעריכה שכן ישפיע גם על מקומות אחרים (וגם על נכונות רמזים אחרים). מידע זה נאסף דרך טבלאות הקישור השונות, זה מידע שניתן להסיק מהכלל אל הפרט (בדרך כלל מידע על מדינה שמקרין על ערים הנמצאות בתוכו).

* + - **NELanguageClue-** מייצג מידע על שפה המדוברת במיקום זה,מידע זה נאסף מהמדינה בה העיר נמצאת. (עבור ערים בלבד).
    - **NECurrencyClue-** מייצג מידע על המטבע בו משתמשים באותו מקום**,** מידע זה נאסף מהמדינה בה העיר נמצאת. (עבור ערים בלבד).
    - **NEContinentClue-** מייצג מידע על היבשת בה העיר נמצאת, מידע זה נאסף מהמדינה בה העיר נמצאת. (עבור ערים בלבד).
    - **IsCountryClue-** מייצג כי המיקום הוא מדינה.
    - **IsCityClue-** מייצג כי המיקום הוא עיר.

**תרשים של סוגי הרמזים השונים ויחס הירושה ביניהם:**

**2.2 חבילת db**

אחראית על כל התקשורת מול מסד הנתונים. החבילה מחולקת לשתי תתי חבילות

**2.2.1 תת חבילה hangman.db.utils**

מכילה מחלקות עזר ליצירת חיבור ותקשורת מול מסד הנתונים. החבילה כוללת את המחלקות הבאות:

**ConnectionPool 2.2.1.1**

מחלקה אחראית על יצירת חיבורים מול מסד הנתונים ומחזורם. המחלקה מגדירה מתודה configure, אשר מאתחלת את פרטי ההתחברות אל המסד (host, port, schema, username, password) וכן את הגודל המכסימלי של ה-pool הרצוי (poolSize). כפי הזכרנו מקודם, כל פרמטרים הללו נקבעים בקובץ הקונפיגורציה. הקריאה למתודה זאת מבצעת מתוך מחלקת Main, בתחילת ריצת האפליקציה.

ה-ConnectionPool עצמו הוא Singleton, גלובלי עבור כל האפליקציה, אותו ניתן להשיג באמצעות המתודה getPool(). כאשר המתודה getPool() נקראת לראשונה, אובייקט ה-singleton מאותחל לראשונה (על ידי קריאה לפונקציה init). חשוב לציין כי על מנת לשפר ביצועים, init עצמו אינו פותח את כל poolSize החיבורים, אלא רק טוען את ה-driver (מתודה registerJDBC). החיבורים עצמם, נפתחים רק בעת הצורך (ראה הסבר מתודה getConnection).

חשוב לציין כי רוב המתודות במחלקה הן **synchronized,** על מנת לתמוך בגישה (נכונה) של כמה threadים בו זמנית.

המתודות הפומביות שה-singleton חושף:

* getConnection() – מתודה מאפשרת לקבל חיבור מתוך ה-pool. אם כרגע יש חיבור בתוך ה-pool, אזי הוא מוחזר. אם אין חיבור בתוך ה-pool ועד כה פתחנו פחות מ-poolSize חיבורים, נפתח חיבור חדש ונחזיר אותו. אם אין חיבור בתוך ה-pool וכבר פתחנו poolSize חיבורים, המתודה נכנסת למצב המתנה עד אשר יוחזר חיבור לתוך ה-pool.
* returnConnection(Connection) – מתודה מחזירה את החיבור הנתון לתוך ה-pool. המתודה תעדכן את ה-threadים אחרים הממתינים לחיבור, כי התווסף חיבור לתוך ה-pool.
* קיימות מתודות זהות עבור קבלה\החזרה של *קבוצה* של חיבורים (getConnections, returnConnections).

בסוף ריצת האפליקציה, מתבצעת קריאה למתודה הסטטית closePool() על מנת לסגור את כל החיבורים.

הערה: poolSize חייב להיות לפחות 3, משום שבתהליך ההעלאה, שגם הוא מסתמך על ה-pool, אנחנו משתמשים בלפחות 3 חיבורים שונים.

**QueryExecutor 2.2.1.2**

מחלקה מספקת שירותי ביצוע שאילתות ליתר האפליקציה (בעיקר עבור ה-core). המטרה של המחלקה היא לספק ממשק נוח לביצוע שאילתות. המחלקה מאפשרת לבצע שאילתות בצורה כזו שהמשתמש שלה לא צריך להתעסק עם החיבורים למסד ישירות. המחלקה מנהלת את הגישה ל-ConnectionPool, החזרת החיבורים וסגירת משאבים.

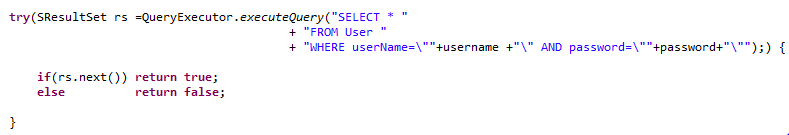
המתודות (הסטטיות) שהמחלקה מספקת:

* **executeUpdate**(query): מעטפת ל-executeUpdate של JDBC. השאילתא מתבצעת באמצעות חיבור שנלקח זמנית מה-connection pool. אם נזרק חריג, הוא נזרק הלאה לטיפול אצל מי שקרה למתודה. חשוב לציין כי המתודה דואגת לסגור את האובייקט Statement שנוצר ולהחזיר את החיבור לתוך ה-pool, בין אם נזרק חריג או לא.
* **executeUpdate**(Connection, query): כמו המתודה מעל, אך מקבלת גם חיבור, כאשר האחריות להחזיר אותו בחזרה ל-pool היא על הקורה למתודה.
* **executeQuery**(query): מעטפת ל-executeQuery של JDBC. לצורך ביצוע השאילתא, נקלח חיבור מתוך ה-connection pool. אם נזרק חריג, הוא נזרק הלאה לטיפול אצל מי שקרה למתודה. חשוב לציין כי במקרה של חריג, המשאבים שהוקצו מפונים והחיבור שנלקח מוחזר ל-pool. במקרה של הצלחה, לעומת זאת, מוחזר אובייקט מיוחד SResultSet (שיוסבר בפירוט בהמשך), שהוא מעטפת של ResultSet, אך שומר בתוכו את החיבור ואת ה-Statement המלווים ל-ResultSet. SResultSet הוא auto-closeable, וסגירה שלו סוגרת בצורה אוטומטית את ה-ResultSet, ה-Statement ומחזירה את החיבור ל-pool. חשוב לציין כי בדיוק כמו את האובייקט ResultSet, אובייקט מסוג SResultSet יש לסגור בסוף שימוש.
* **executeQuery**(connection, query, returnConnection): כמו המתודה מעל, אך מקבלת חיבור. הפרמטר האחרון מציין אם האובייקט SResultSet שחוזר, בסגירה שלו צריך להחזיר את החיבור ל-Pool, או שהאחריות להחזיר את החיבור היא על הקורה לפונקציה.
* **execute**(connection, query) – מעטפת פשוטה ל-execute של JDBC. המעטפת דואגת תמיד לסגור את האובייקט Statement שנוצר.

**SResultSet 2.2.1.3**

מעטפת של ResultSet (S מסמל Safe) המאפשרת את כל הפונקציונליות הנדרשת מ-ResultSet, אך שומרת גם את ה-Statement object שבהקשר שלו נוצר ה-ResultSet וכן את החיבור שבהקשר שלו נוצר ה-ResultSet (אם כי החיבור הוא אופציונלי).

המחלקה מממשת את הממשק של AutoCloseable (ולכן ניתן להשתמש ב-try with resource block על מנת לסגור אותה), על ידי מימוש מתודה close(). חובה לקרוא למתודה זו בסוף שימוש ב-SResultSet, מתודה זו תסגור את ה-ResultSet, את האובייקט Statement ואם סופק, תחזיר את החיבור ל-connection pool.

השילוב של שתי המחלקות האחרונות מאפשר ביצוע מאוד נוח של שאילתות (ניתן לראות זאת בחבילת core), לדוגמה, להלן מתודת בדיקת ההתחברות:

הקוד הפשוט הנ"ל כולל בתוכו את כל הלוגיקה של השאלת חיבור וההחזרה שלו לpool- (מיד במקרה של שגיאה, בעת סגירת ה-SResultSet במקרה של הצלחה), ביצוע שאילתה וסגירת אובייקטים של ResultSet ו-Statement (בין אם התרחשה שגיאה ובין אם לא).

**2.2.2 תת חבילה hangman.db.upload**

החבילה הזו מרכזת את כל המחלקות שלוקחות חלק בביצוע העלאה של כל הנתונים שהתקבלו בשלב ה-parsing למסד הנתונים. החבילה עצמה כוללת מספר מחלקות ושתי תתי חבילות נוספות.

**DBBuilder 2.2.2.1**

המחלקה הראשית, מחברת בין הפרסר לבין ה-upload. המתודה הראשית, buildDB(), אחראית על לאתחל את תהליך הפרסינג, ולאחר מכן לאתחל את תהליך ה-upload למסד הנתונים (על ידי שימוש במחלקה ParsedDataUploader הנמצאת באותה חבילה ומתוארת בהמשך).

חשוב לציין כי מוגדרים שלושה דגלים גלובליים במחלקה:

* DESERIALIZE – האם לבצע deserialization מתוך קבצים שהתקבלו ב-serialization או לבצע parsing מהתחלה. השדה נועד לצורכי debugging בלבד (הרבה יותר מהיר מאשר לבצע parsing כל פעם מחדש). ב-release version הערך הוא תמיד false, כלומר תמיד יתבצע parsing.
* SERIALIZE – האם לבצע serialization בסוף ה-parsing. שוב, נועד לצורכי debugging בלבד. ב-release version הערך הוא תמיד false.
* UPLOAD – האם לבצע upload בסוף שלב ה-parsing. שוב, נועד לצורכי debugging בלבד. ב-release הערך הוא תמיד true.

חשוב לציין כי המחלקה דואגת לעדכן את ה-UI עם הודעות progress מתאימות (כלומר, כמה אחוז מן התהליך המתאים הסתיים).

**AbstractBatchUploader 2.2.2.2**

מחלקת אב של היררכיה רחבה של uploaders. מטרת המחלקה היא לספק תשתית משותפת להעלאת טבלה יחידה.

העלאה מתבצעת על ידי batches, כל אחד בגודל BATCH\_SIZE (נקבע בקובץ config). הזרימה של האלגוריתם שמופעל על ידי המתודה upload היא כזו:

* מתבצעת קריאה ל-upload
* כל עוד נותרו רשומות שצריך לעלות, נבצע קריאה ל-uploadBatch
* מתודה זו אחראית לקרוא ל- constructPreparedStatement, על מנת ליצור אובייקט PreparedStatemement עם השאילתה שהמתודה getQueryString() מחזירה. לאחר מכן האובייקט ממולא עם הרשומות אותן אנו רוצים לעלות ב-batch הנוכחי על ידי המתודהprepareBatch . לאחר מכן, נבצע executeBatch ו-commit.
* לאחר ה-commit, מתבצעת קריאה ל-postCommit (לדוגמה ב-EntityUploader שמתואר בהמשך, תהיה אחראית על קביעת המפתחות לכל ה-entities שנשמרו על ידי prepareBatch).
* מחרוזת השאילתה עצמה, מושגת על ידי המתודה getQueryString (אבסטרקטית כמובן)

האלגוריתם הוא גנרי, וחלקיו הם abstract methods אשר יוחלפו בהתאם לנתונים אותם רוצים לעלות.

המחלקה מספקת גם factory method לבניית כל סוגי ה-uploaders.

**ParallelDBUploader 2.2.2.3**

על מנת למזער את הזמן שלוקח לעלות את הנתונים (לשרת מרוחק אמיתי), החלטנו למקבל פעילות של AbstractBatchUploader יחיד, בצורה כזו שכל batch יעלה כחלק מ-task נפרד. המחלקה ParallelDBUploader עונה בדיוק על הדרישה הזו.

בהינתן סוג uploader (enum שמוגדר במחלקה AbstractBatchUploader) ואוסף חיבורים פתוחים, המחלקה מחלקת את אוסף הנתונים שיש לעלות לתתי קבוצות של נתונים, כאשר כל קבוצה היא בגודל של BATCH\_SIZE בודד, ומעלה כל קבוצה כזו על ידי uploader נפרד עם חיבור נפרד (בצורה של משימות שמתוזמנות על ידי ה-ThreadPool שמוגדר בחבילת core), בהתאם למספר החיבורים שניתנו.

**MultipleParallelUpload 2.2.2.4**

מחלקה פשוטה שעוטפת את המחלקה הקודמת, ותפקידה להפעיל את ParallelDBUploader סדרתית עבור כמה טבלאות שונות (כלומר לבצע העלאה לכמה טבלאות, טבלה אחר טבלה, כאשר העלאה של כל טבלה ממוקבלת בצורה כזו שכל batch שלה עולה כחלק מ-task נפרד, כאשר כמות ה-taskים היא לכל היותר כמות החיבורים שניתנו לאובייקט)

**ParsedDataUploader 2.2.2.5**

מממש את כל תהליך ה-upload, שמאותחל על ידי קריאה למתודה begin().

בשלב הראשון, תתבצע קריאה ל-clearExistingRelations, מתודה שתנקה את כל הטבלאות הקיימות במסד הנתונים, פרט לטבלת המשתמשים, User ולכן הפרטים של כל השחקנים ישמרו.

בשלב שני, תתבצע קריאה ל-initiateUpload() שמבצעת את העלאה עצמה. בתחילת המתודה, כל החיבורים בתוך ה-ConnectionPool מחולצים ומחולקים ל-3 קבוצות (זו הסיבה שצריך לפחות 3 חיבורים קיימים), כל קבוצת חיבורים תמופה לקבוצה של טבלאות.

על מנת למזער את זמן ה-upload, הטבלאות אינן עולות בצורה סדרתית, אלא יש חלוקה למספר MultipleParallelUpload כאשר כל אובייקט כזה אחראי לעלות קבוצה מסוימת של טבלאות (יש לזכור, כי MultipleParallelUpload עצמו מעלה כל batch בטבלה כחלק מ-task אחר) ומקבל את אחת הקבוצות החיבורים שהוזכרו מקודם. בצורה כזו, הטבלאות עולות בצורה מקבילית (3 טבלאות לכל היותר עולות בו זמנית, לכן צריך לפחות 3 חיבורים) וכל טבלה בעצמה מפורקת ל-batchים, אשר כל batch עולה כחלק מ-task נפרד (בהתאם למספר החיבורים שניתנו לאובייקט MultiplePararellUpload שמעלה טבלה זו).

התרשים להלן מסכם את תהליך בניית המסד וה-upload בפרט:

1

…

FileParser

…

MultipleParallelUpload

…

MultipleParallelUpload

ParsedDataUploader

DBUploader

2

…

MultipleParallelUpload

…

ParallelDBUploader

ParallelDBUploader

**ThreadPool Tasks**

**…**

Languages

Continentss

Subtype of AbstractBatchUploader [Single Batch Upload]

Subtype of AbstractBatchUploader [Single Batch Upload]

**ThreadPool Tasks**: Each task uploads a batch

**…**

**2.2.3 תת החבילה hangman.db.upload.entities**

חבילה זו אחראית לעלות את כל הרשומות לטבלאות שמייצגות entities (כגון: מנהיג, מדינה)

לכל טבלה שמייצגת Entity, יש אובייקט uploader משלה. לדוגמה, ConstructionUploader תפקידו לעלות את כל ה-ConstructionEntityים לתוך הטבלה Construction.

כל המחלקות בתת חבילה זו, יורשות מ-AbstractBatchUploader. להלן הסבר על המחלקות הקיימות החבילה זו:

**EntityUploader 2.2.3.1**

המחלקה החשובה ביותר בחבילה זו. מחלקה אבסטרקטית, שמגדירה כיצד uploader שמעלה טבלת entity צריך לתפקד.

המחלקה מוגדרת על ידי iterator שרץ על אוסף של Entities (כלומר מסוג parsing.entities.Entity) מאותו סוג, האוסף אותו אנו מבקשים לעלות לתוך טבלה אחת במסד הנתונים. מחלקה הזו מגדירה כיצד המתודה prepareBatch צריכה להתנהג – לשמור את ה-entity batch שעומד לעלות כרגע, על מנת לאתחל מאוחר יותר את ה-idים כפי שנקבעו לכל entity על ידי מסד הנתונים . כמו כן, מגדירה את המתודה postCommit – שמבצעת השמה של המפתחות שחזרו בעקבות העלאת ה-batch, אל ה-Entities שנשמרו על ידי prepareBatch (כלומר מאתחלת את השדה dbID של כל entity כזה).

מחלקות concrete שיורשות מ-EntityUploader המוגדרות בתת חבילה זו, מגדירות את השאילתה שיש לבצע (getQueryString) בהתאם לסוג הטבלה וכיצד יש לתרגם Entity נתון לפרמטרים של אובייקט PreparedStatememt (מתודה setStatementArgs(statement, entity)).

**2.2.4 תת החבילה hangman.db.upload.relations**

החבילה הזו אחראי לעלות את כל הרשומות של הטבלאות המקשרות (שמייצגות קשרים many to many). לכל טבלה מקשרת במסד הנתונים, קיים אובייקט uploader מיוחד. לדוגמה, MilitaryParticipantsUploader אחראי לעלות את כל הרשומות של הטבלה MilitaryActionParticipants. כל ה-concrete uploaders בחבילה יורשים מ-RelationUploader, מחלקה אבסטרקטית שעצמה יורשת מ-AbstractBatchUploader.

**RelationUploader 2.2.4.1**

בדומה ל-EntityUploader, המחלקה מוגדרת על ידי איטרטור שרץ על אוסף Entities, שמשתתפים בקשר ה-"many to many" אותו רוצים לעלות. הקשרים הללו נשמרים כרשימות בתוך אחת מן מחלקות ה-Entity שמשתתפות בקשר (לדוגמה, Entity של מדינה, שומר רשימה של השפות רשמיות המדוברות במדינה), לכן המימוש של prepareBatch הוא להוסיף רשימות ל-batch לפי ה-entities ששמורים בתוך *הרשימות* הללו.

המימוש של prepareBatch כעת הוא לרוץ על פני הרשימה הקיימת, currentRelationList, ולמלא את הרשומות לפי ה-Entity הנוכחי עליו עומדים ברשימה. אם הרשימה נגמרת, עוברים לרשימה של ה-Entity הבא שמוחזר על ידי האיטרטור, על יד המתודה nextRelationList.

כל Uploader concrete שיורש מן RelationUploader צריך להגדיר את המתודה nextRelationList, שמאתחלת את הרשימה הבאה לתוך המשתנה currentRelationList.

כמו כן, יש להגדיר את המתודה setStatementArgs, שמאתחלת את ערכי הרשומה הנוכחית בתוך אובייקט preparedStatement, לפי ה-Entity שהנוכחי עליו עומדים ברשימה, וכן ה-Entity הנוכחי שחזר מהאיטרטור (אשר לו שייכת הרשימה).

לבסוף, יש להגדיר גם את המתודה getQueryString() שמחזירה את השאילתה שעמה האובייקט preparedStatement הנ"ל נבנה, כאשר השאילתה נקבעת לפי הטבלה אליה רוצים לעלות.

**2.2.5 הערות נוספות**

חשוב לציין כי כל המחרוזות שעולות למסד הנתונים עוברות sanitization (על ידי המתודה sanitizeString במחלקת EntityUploader), לדוגמה מבצעים escape לתווים מיוחדים כמו '.

בנוסף לכך, entities שסומנו כלא valid (בדיקה על ידי המתודה isValid במחלקה Entity) לא יעלו למסד הנתונים (ואם הישות השתתפה בקשר רבים לרבים, לא תיווצר רשומה מתאימה עבור הקשר. מבחינת ה-uploaderים, ישויות כאלה, והקשרים שבהם הן מעורבות, לא קיימים כלל).

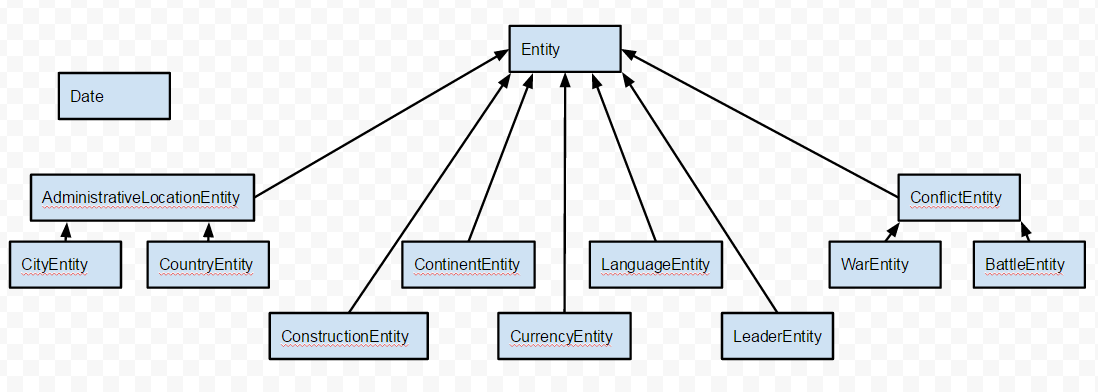
**2.3 חבילת parsing**

כל המחלקות הנמצאות בחבילת ה-parsing אלו מחלקות שלוקחות את קבצי הYAGO, ומוציאות מהם את המידע הרלוונטי עבור ה- DB. המידע המחולץ נשמר בקבוצה של מבני נתונים שיעזרו לנו לבצע את ההעלאה בצורה נוחה.

בתוך חבילת parsing יש שתי תתי חבילות: hangman.parsing.entities ו- hangman.parsing.parsers.

**2.3.1 תת חבילה hangman.parsing.entities**

להלן תרשים עץ הירושה בין המחלקות בחבילה:



המחלקות בחבילה מייצגות *ישויות* שהנשמרות ב-DB, כמו מנהיגים, ערים ומלחמות. ה-Entities שמתקבלים בתהליך הפרסינג נשמרים כאובייקטים של המחלקות המתאימות בתרשים הנ"ל. מחלקות הללו נמצאות בשימוש רק בתהליך ה-parsing וה-upload.

המחלקה האבסטרקטית entity מייצגת סט של תכונות משותפות ולוגיקה שכל הישויות בDB- מקיימות.

המחלקות AdministrativeLocationEntity ו- ConflictEntityגם הן abstractיות ועוזרות לרכז את החלקים המשותפים של היורשים שלהם, שאינם באים לידי ביטוי במחלקת entity.

המחלקה date היא מחלקה שעוזרת בפרסינג ושמירה של התאריכים השונים, לשימושם של הישויות השונות.

רשימת המחלקות ותיאורן:

* AdministrativeLocationEntity - מחלקת אב אבסטרקטית עבור המחלקות CityEntity ו-CountryEntity. אובייקט מסוג AdministrativeLocationEntity מגדיר רשומה בטבלת AdministrativeDivision.
* CityEntity - מחלקה יורשת מ- AdministrativeLocationEntity. המחלקה מייצגת את ישויות הערים. המידע הנשמר באובייקט מסוג זה יכנס לטבלאות AdministrativeDivision (החלק שה- CityEntity יורש מ AdministrativeLocationEntity-) ו-City.
* CountryEntity - מחלקה יורשת מ AdministrativeLocationEntity-. המחלקה מייצגת את הישות country. המידע הנשמר באובייקט זה יכנס לטבלאות AdministrativeDevision (החלק שCountryEntity יורש מ AdministrativeLocationEntity) ו-Country.
* ConstructionEntity- המחלקה מייצגת את הישויות המבנים. המידע הנשמר באובייקט זה יכנס לטבלה Construction .
* ContinentEntity - המחלקה מייצגת את הישויות של יבשות. המידע הנשמר באובייקט זה יכנס לטבלה Continent.
* CurrencyEntity - המחלקה מייצגת את הישויות של סוגי מטבעות. המידע הנשמר באובייקט זה יכנס לטבלה Currency.
* LanguageEntity - המחלקה מייצגת את הישויות של שפות. המידע הנשמר באובייקט זה יכנס לטבלה Language.
* LeaderEntity - המחלקה מייצגת את הישויות של מנהיגים. המידע הנשמר באובייקט זה יכנס לטבלה Leader.
* ConflictEntity - מחלקת אבא אבסטרקטית עבור המחלקות BattleEntity ו- WarEntity. כל המידע בשדות של אובייקטים מסוג ConflictEntity יכנס לטבלה MilitaryActions.
* BattleEntity - מחלקה יורשת מ ConflictEntity. המחלקה מייצגת את הישות של קרב. המידע הנשמר באובייקט זה יכנס לטבלאות MilitaryActions (החלק ש BattleEntity יורש מ ConflictEntity-) ו- Battle.
* WarEntity - מחלקה יורשת מ ConflictEntity. המחלקה מייצגת הישויות של מלחמות. המידע הנשמר באובייקט זה יכנס לטבלאות MilitaryActions (החלק ש WarEntity- יורש מ ConflictEntity-) ו- War.
* Date - כפי שאמרו מקודם, זוהי מחלקה תומכת בשאר המחלקות בחבילה זו, ואינה מייצגת ישות מהDB. המחלקה משמשת עבור שמירה של התאריכים השונים של הישויות בDB, ופונקציות עזר הקשורות בהם. הפונקציה DateString\_to\_Date מממשת עבור המחלקה factory method, בעזרתה מייצרים אובייקטים מהטיפוס Date, בהינתן String המייצג תאריך שאנו משיגים בתהליך הפרסינג מקבצי YAGO.
* Entity - זוהי מחלקה ממנה יורשים כל המחלקות שמייצגות ישויות של הDB. המחלקה מעניקה את סט התכונות המשותף בין כל ישויות ה-DB, כמו שם (הנשמר אך ורק בתנאי שהוא עומד בתנאי שהטלנו על השמות בDB- מורכב מתווי ascii בלבד) ו- dbID (ה-primary key id כפי שנקבע עבור ה-entity לאחר שעלה למסד, לא נקבע בשלב ה-parsing כי לא ידוע בשלב זה).

**2.3.2 תת חבילה hangman.parsing.parsers**

בחבילה זו נמצאים כל המחלקות השונות המתעסקות בparsing של קבצי הYAGO ומחלקה בה נשמרים כל תוצאות הparsing (האובייקטים השונים המייצגים את הישויות נשמרים בתוך maps, בהם הkey הוא הYAGO tag של אותה ישות).

המחלקה FileParser מהווה אב אבסטרקטי משותף לכל המחלקות בחבילה למעט ParsedData (שבה נשמרים הנתונים, ולא מתבצע פרסור). מבנה המחלקה FileParser קובע למעשה כיצד parser צריך הראות ולהתנהג באופן כללי.

הקריאה לתהליך הפרסור שמחלקה מבצעת נעשית ע"י קריאה לפונקציה parseFile() של אותה המחלקה עם שם הקובץ המתאים (כלומר path מלא לקובץ הYAGO אותו בנויה המחלקה לפרסר).

כל מחלקות הפרסר דומות אחת לשנייה ומבצעות את תהליך הפרסור בצורה דומה:

1. קריאה לפונקציה parseFile().
2. הפונקציה parseFile() קורא לפונקציה init(), אשר מאתחלת את ה- Mapsהנדרשים בהם על מנת לשמור את הישויות המתקבלות.
3. הפונקציה parseFile() פותחת את הקובץ , ומפעילה את הפונקציה filter() על כל שורה שלו (חוץ מהראשונה, בה ניתן תיאור כללי לגבי תוכן הקובץ ולא מידע ממש).
4. הפונקציה filter() מוציאה מהשורה את המידע שהיא מכילה (אשר נראה באופן הבא: ישות\_א, קשר, חלק\_ב) . לאחר מכן היא עושה switch על גבי הקשר כדי לראות האם הוא מהסוג ש"מעניין" אותנו. במידה וכן מייצרים אובייקט מייצג לישות\עושים השמה מתאימה לשדה של הישות.

רשימת המחלקות ותיאורן:

* FileParser - אב אבסטרקטי משותף לכל המחלקות שעושות פרסור של קבצי YAGO. מכיל פונקציות עזר ואת המבנה שכל מחלקת פרסור צריכה לממש.
* DateFactsParser- מבצע את פרסור הקובץ **yagoDateFacts.tsv.**

מתוך הקובץ הזה אנחנו מוציאים את המידע הבא:

wasBornOnDate, ו-diedOnDate עבור ישויות leaders.

happenedOnDate עבור conflict entities.

wasCreatedOnDate, ו- wasDestroyedOnDate עבור AdministrativeLocation entity.

* FactsFileParserFirstRun - מבצע את המעבר הראשון על הקובץ **yagoFacts.tsv.**

מתוך מעבר זה הזה אנחנו מוציאים את המידע הבא:

בעזרת היחס isleaderof על ישויות של AdministrativeLocation, אנחנו משיגים את הישויות המנהיגים שיהיו בטבלת הleaders שלנו.

* FactsFileParserSecondRun - מבצע את המעבר השני על הקובץ **yagoFacts.tsv.**

מתוך מעבר זה הזה אנחנו מוציאים את המידע הבא:

diedIn , ו- wasBornIn עבור ישויות leaders.

hasCapital, hasCurrency, ו- hasOfficialLanguageעבור Country entities.

happenedOnDate עבור conflict entities.

isLocatedIn (ערים locatedIn במדינות, ומדינות locatedIn יבשות), participatedIn (אלו ישויות אדמיניסטרטיביות השתתפו באלו קונפליקטים) ו- owns (אילו מבנים נמצאים באלו ישויות אדמיניסטרטיביות) עבור AdministrativeLocation entity.

בעזרת היחס owns על ישויות של AdministrativeLocation, אנחנו משיגים את הישויות של Constructions.

* LabelsParser - מבצע את פרסור הקובץ **yagoLabels.tsv.**

מתוך הקובץ הזה אנחנו מוציאים את המידע הבא:

ה- prefLabel עבור הישויות השונות בDB. בו אנחנו משתמשים בתור שם לישות.

נסכים להעלות ישויות אם ורק אם השם שלהם קיים ומורכב אך ורק מתווי ascii. תנאים אלה בודקת הפונקציה setName() במחלקה Entity.

* LiteralFactsParser - מבצע את פרסור הקובץ **yagoLiteralFacts.tsv.**

מתוך הקובץ הזה אנחנו מוציאים את המידע הבא:

hasNumberOfPeople ו- hasMotto עבור AdministrativeLocation entity.

* TransitiveTypeParser - מבצע את פרסור הקובץ **yagoTransitiveType.tsv.**

מתוך הקובץ הזה אנחנו מוציאים את המידע הבא:

בעזרת הtype אנחנו מוציאים את הישויות City, Country, war, battle , Language ו-Currency. מאתחלים את האובייקטים המתאימים עבורם.

* WikipediaInfoParser - מבצע את פרסור הקובץ **yagoWikipediaInfo.tsv.**

מתוך הקובץ הזה אנחנו מוציאים את המידע הבא:

מדד הפופולריות עבור הישויות השונות (לא לכולן) בעזרת היחס hasWikipediaArticleLength.

* ParsedData - שומרת מפות עבור הישויות השונות עם הYAGO tag בתור המפתח והאובייקט המייצג את הישות בתור הvalue.

בנוסף במחלקה נמצאות פונקציות סריאליזציה ודסיארליזציה, המשמות אותנו עבור טסטים, כך שלא נצטרך לעבור תמיד על כל המידע בקבצים המקוריים של yago אלא לאתחל בקלות ובמהירות את המפות עם האובייקטים המתאימים.

**2.3.3 הערות נוספות**

אנחנו עושים 2 סוגי סינונים של הentities על גבי השם, בכדי לשמור על "נכונות" הנתונים בDB והיכולת להציג אותם נכון.

1. אנחנו מסננים את כל אותם הישויות, המכילות אותיות שאינם ascii. את רוב האותיות מהסוג הזה לא נוכל להציג כמו שצריך, ולכן אין טעם לשמור ישויות עם שם המכיל אותיות כאלו – לא נוכל להציג אותה. במקרה שנאתר שם כזה, נגדיר את הישות להיות invalid ובשלב ההלעאה לDB נתעלם ממנה.
2. אנחנו מסננים את כל אותם ישויות, שמבחינה הגיונית לא יתכנו שתיים שונות עם אותו השם ומאותו הסוג. ה"סוגים" הללו הוגדרו להיות: יבשות, סוגי מטבעות, שפות , קרבות ומלחמות (גם לא נרשה שלמלחמה ולקרב היה אותו השם).

כדי לוודא שהתנאי מתקיים, כלפ עם שנכניס ישות מהסוג המדובר נבדוק מול השמות הקודמים האם השם הזה כבר הופיע (נסתכל רק על ה45 תווים הראשונים, כי רק ה45 תווים הראשונים בשם ישמרו בDB). אם כן הישות תוגדר להיות invalid (בדומה למקרה והשם מכיל אותיות שהם 8UTF שאנם ascii).

**2.3 חבילת parsing**

**UIMain**

המחלקה הראשית של הUI, מכילה את כל הפונקציות הדרושות לבניית המסכים השונים בתוכנית (לא כולל מסכי עדכון מידע מהDB ) , מכילה מספר פונקציות public כדי לאפשר תקשורת בין הcore לUI בין השאר להצגת שגיאות שלא ניתן להתגבר עליהן.

**HangmanFormat**

מחלקה המכילה פונקציה סטאטית אחת לציור האיש התלוי במשחק.

**ExceptionHandler**

מחלקה המממשת Runnable, במקרה וישנה שגיאה חמורה כמו ניתוק אינטרנט, בעיות בDB וכו'  
תהיה קריאה ל asyncExec() עם instance של המחלקה, ותתבצע סדרה של פעולות כמו הקפצת הודעת שגיאה וסגירה של שאר החלונות של המשחק.

**UpdateLayout**

vמחלקה מכילה את הבנייה של חלון העריכה של 3 הסוגים השונים של רמזים שניתן לערוך כחלק מהמשחק:

* סוג ראשון: שדה יחיד המאפשר שינוי ישיר של המידע בDB
* סוג שני: השינוי מבוסס על מידע שקיים בDB ולכן תחילה יש לחפש אותו בDB
* סוג שלישי: שינוי המאפשר מחיקת מידע קיים והוספה של חדש בדומה לשינוי של הסוג השני

**WorkerThread**

מחלקה מממשת Runnable, ובעצם מאפשרת לUI לתקשר עם הCore- וה-DB באמצעות הThreadPool שמימשנו בתוכנית, על מנת למנוע מצב Blocking של ה-UI. כדי להימנע משימוש נרחב במחלקות אנונימיות, בעת יצירת האובייקט יש לתת לבנאי ארגומנט שיקבע איזו פעולה הUI מעוניין שתתבצע מתוך סדרה של פעולות אפשריות כמו: בדיקה האם שם משתמש וסיסמא תקינים, הרשמה של משתמש חדש, טעינת שיאים מה-DB, ובמהלך המשחק טעינה של מקומות חדשים בו ב**זמן** שהשחקן משחק עם מקום קיים (כדי שכשיסיים המקום הבא כבר יהיה מוכן).

**3. המידע שנלקח מYAGO**

אנחנו משתמשים בקבצים הבאים מתוך YAGO כדי למלא את הDB:

yagoDateFacts, yagoWikipediaInfo, yagoTransitiveType, yagoLabels, yagoLiteralFacts, yagoFacts.

כדי להוציא מהם את הקשרים\תכונות הבאות עבור הentities:

hasWikipediaArticleLength, hasNumberOfPeople, hasMotto, prefLabel, diedIn , wasBornIn, hasCapital,happenedOnDate , isLocatedIn, wasBornOnDate, diedOnDate, happenedOnDate, wasCreatedOnDate ו- wasDestroyedOnDate.

או כדי לזהות האם entity הוא מהסוג המתאים:

isleaderof, owns, "<wordnet\_currency\_113385913>", "<wordnet\_continent\_109254614>", "<wordnet\_language\_106282651>", "<wordnet\_country\_108544813>", "<wordnet\_city\_108524735>", "<wordnet\_battle\_100953559>", "<wordnet\_war\_100973077>".

**3.תרשים הזרימה של התוכנית**

