קישור לגיטהאב:

https://github.com/GuySananes/S-Emulator-Project

הפרוייקט שהוגש בפועל נמצא בענף:

daniel/littleTest

מגישים:

דניאל פרנקל, 206090466

גיא סאנאנס, 318693017

מייל:

danielfr3@mta.ac.il

תיאור הפרוייקט:

במהלך המימוש של הפרויקט שמנו דגש על יצירת ארכיטקטורה ברורה, יציבה, ניתנת להרחבה, תוך הקפדה על עקרונות של הפרדת רשויות (Separation of Concerns), שימוש בממשקים (Interfaces), ו־הימנעות משכפול קוד ככל שניתן. תיארנו כאן כמה מהבחירות המימושיות והסגנוניות שעיצבו את הקוד שלנו, אך חשוב לציין כי מדובר במערכת רחבה הכוללת עוד מחלקות, מודולים ורכיבים, והרוח הכללית המובאת כאן משקפת את הגישה שלנו בכל חלקי המערכת.

אחד המנגנונים המרכזיים שבנינו הוא מנגנון ההרחבה (Expansion). הוראות סינתטיות מממשות את הממשק Expandable, אשר מגדיר את ההתנהגות expand(ExpansionContext). בעת הרחבת תוכנית, אנו סורקים את ההוראות ומרחיבים רק את אלו שמממשות את ההתנהגות הזו. הדבר מתבצע דרך מחלקת Expansion באופן רקורסיבי לפי דרגת ההרחבה. הרחבות מתבצעות תוך עטיפה ב־RootedInstruction, כך שכל הוראה חדשה שומרת קשר ישיר להוראת האב ממנה נוצרה. מנגנון זה מאפשר לנו לזהות אילו הוראות בתוכנית נוצרו מאיזו הוראה סינתטית, דבר שמועיל מאוד גם לצורכי Debug, גם להצגה, וגם לניתוח ביצועים.

כדי להבטיח הרחבה תקינה שאינה יוצרת התנגשויות, השתמשנו במחוללי שמות פנימיים – LabelGenerator ו־ZGenerator – אשר ניגשים לתוכנית המקורית ובודקים את הערכים הגבוהים ביותר הקיימים, כדי להקצות שמות חדשים באופן בטוח ומבוקר. הגנרטורים עצמם שומרים state וממשים את ההיגיון מתוך הקשר ההרחבה (ExpansionContext).

במימוש שלנו הקפדנו מאוד על שימוש בממשקים ככלי לתכנון התנהגות. כמעט כל רכיב משמעותי במערכת ממומש מול ממשק: לדוגמה, הוראות מממשות את SInstruction, סטטיסטיקה מממשת את SingleRunStatistic, ניהול סטטיסטיקות מתבצע דרך StatisticManager, והרצה דרך ProgramExecutor. גישה זו איפשרה לנו להגדיר את חוזי ההתנהגות שלנו בנפרד מהמימושים הקונקרטיים, מה שתורם להפרדה טובה יותר בין רכיבים, לאפשרות החלפה, ולהבנה נקייה יותר של המבנה הכללי.

גם הימנעות משכפול קוד הייתה עקרון מכוון. דוגמה מובהקת לכך היא היררכיית ההוראות: AbstractInstruction מהווה בסיס משותף עם מתודות אחידות, ואנו מרחיבים ממנה למחלקות ספציפיות לפי הצורך – למשל הוראות עם שני משתנים (AbstractInstructionTwoVariables) או עם שתי תוויות (AbstractInstructionTwoLabels). בצורה זו, קוד כמו getVariables() או getRepresentation() לא משוכפל בין הוראות אלא ממומש פעם אחת ומורחב רק היכן שצריך. באופן דומה, מתודות עזר כלליות אוגדו למחלקת Util כך שלא נידרש לממש מחדש לוגיקה כמו יצירת IndexedInstructionList.

כחלק מהדרישות להצגה מסודרת, יישמנו גם Comparator שממיין תוויות (Label) בסדר עולה לפי ערכן, כדי לאפשר החזרה עקבית של רשימות תוויות, הן בעת הצגה והן בעת הדפסת התוכנית. הדבר בא לידי ביטוי בפונקציות כמו getOrderedLabels() ובשימושים פנימיים במחוללי שמות.

מערכת ניהול הסטטיסטיקה נשענת על מחלקת StatisticManagerImpl, שמומשה כ־Singleton, במטרה להבטיח אחידות בניהול ההיסטוריה של הרצות התוכנית. הסטטיסטיקה נשמרת במפת־מיפוי (Map) שבה המפתח הוא מופע SProgram והערך הוא רשימה של הרצות (<List<SingleRunStatistic>), כך שניתן בקלות לשלוף את כל הריצות של כל תוכנית. כל הרצה מתועדת עם פרטים כמו מספר ריצה, דרגת הרחבה, קלטים, תוצאה ומספר מחזורים.

גם מחלקת EngineImpl, המהווה את ה־API הראשי של המערכת מול ממשק המשתמש, מומשה כ־Singleton. היא מרוכזת ומפוקחת, ומרכזת את כל הפעולות האפשריות מול תוכנית: טעינה, הצגה, הרחבה, הרצה וסטטיסטיקות. כל פעולה כזו מנותבת למודול הרלוונטי, תוך הקפדה על הפרדת אחריות ברורה בין רכיבים: מחלקת Engine אינה מבצעת את ההרצה בעצמה אלא יוצרת RunProgramDTO שמנהל את ההרצה בפועל; הצגה מתבצעת דרך PresentProgramDTOCreator; והרחבה דרך Expansion.

גם RunProgramDTO עוצב כ־stateful object – ברגע שנוצר, הוא שומר את מצב הריצה, כולל הקלטים, דרגת ההרחבה, והתוכנית שהורצה (מורחבת או מקורית). לאחר הרצה, ניתן לגשת ממנו בקלות לתוצאה, למחזורי ההרצה ולערכי המשתנים. הגישה לנתונים הללו מתבצעת בצורה מבוקרת: אם לא בוצעה הרצה, נזרקת החריגה ProgramNotExecutedYetException. כך נשמר סדר לוגי ברור בין השלבים.

בהיבט הסגנוני, הקפדנו על Immutable בכל מקום שניתן: שדות final, העתקת רשימות ב־List.copyOf או new ArrayList, ומניעת שינוי מבני DTO מבחוץ. חריגות הוגדרו בצורה מדויקת וברורה, תוך שימוש בחריגות ייעודיות למקרי שגיאה ידועים מראש. כמו כן, במקומות רבים העדפנו את Objects.requireNonNullElse(...) כדי להימנע מ־if־ים מסורבלים ולשמור על קריאות גבוהה.

ולבסוף, אמנם המערכת מחולקת לארבעה מודולים עיקריים — engine, console-ui, dto, ו־exception — אך הקוד בכל מודול מאורגן בתוך פאקג'ים לפי תחומי אחריות, באופן שמאפשר תחזוקה נוחה, הרחבה קלה, והבנה מיידית של תחומי אחריות:

מודול engine, פאקאג'ים כמו:

core.logic – הלוגיקה המרכזית של התוכנית: הוראות, משתנים, תוויות, ותוכניות

expansion – מנגנון ההרחבה של הוראות סינתטיות, כולל ניהול הקשר, מחוללי שמות ותיעוד

statistic – ניהול סטטיסטיקות הרצה, כולל מבנה נתונים לריצה בודדת ומנהל סטטיסטיקות מרכזי

מודול dto, פאקאג'ים כמו:

present – יצירת DTOים להצגת תוכנית והוראותיה בממשק המשתמש

run – עטיפת הרצת תוכנית ספציפית כולל קלטים, דרגת הרחבה, ותיעוד התוצאה

מודול exception-

חריגות מותאמות למערכת, כולל NoProgramException, ProgramNotExecutedYetException, DegreeOutOfRangeException ועוד

מודול console-ui

ממשק שורת הפקודה שמאפשר הפעלה של פעולות דרך ה־API (EngineImpl)

הגישה שלנו לאורך כל הדרך הייתה לא רק לפתור את הבעיה, אלא לחשוב הגיונית מהו הדיזיין הנכון. ניסינו לבנות תשתית מסודרת, מודולרית וברורה, שתהיה נוחה להרחבה בעתיד, קלה להבנה, ועמידה לאורך זמן.