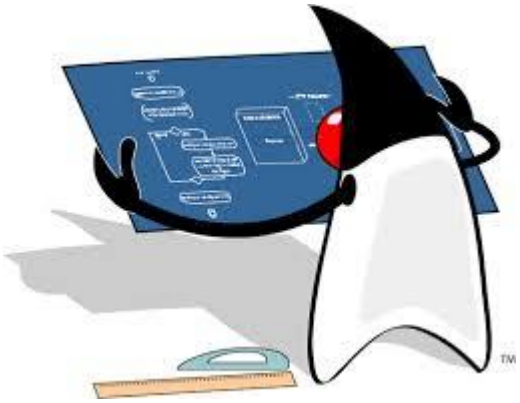
אוניברסיטת בן-גוריון בנגב
Ben-Gurion University of the Negev

משחק מבוך

סמסטר ב' תשע"ז

תקציר

- חלק א' - יצירת ספריית קוד
  - מפסאודו-קוד לתכנות מונחה עצמים
- חלק ב' – שרת-לקוח
  - Streaming, קבצים, ו decorator
  - תכנות מקבילי עם java threads
- חלק ג'
  - MVVM
  - תכנות מונחה אירועים
  - GUI בטכנולוגיית JavaFX



ד"ר אליהו חלסצ'י
khalastc@post.bgu.ac.il

Table of Contents

מנהלות

בקורס זה יינתן תרגיל חימום אחד עם חובת הגשה וללא ציון על מנת להכין אתכם לפרויקט. התרגולים במעבדה נועדו ללמד אתכם את יסודות שפת ***Java***, את מה שיש לה להציע ואת דרכי העבודה המומלצות יחד עם הדגמת החומר הנלמד בהרצאה. הכלים והידע שתלמדו במעבדה נדרשים לכתיבת הפרויקט וימשו אתכם רבות בהמשך דרככם.

לפרויקט שלוש אבני דרך, מועדי הגשתם המעודכנים רשומים באתר ה-Moodle של הקורס.

משקל כל חלק בפרויקט 33% ממשקל הפרויקט הכולל.

כל חלקי העבודה הינם להגשה בזוגות.

כל חלק בפרויקט נבנה בהתבסס על החלק שקדם לו לכן חשוב לעמוד בדרישות בצורה הטובה ביותר. עמידה בדרישות בחלק מסוים בפרויקט יקל עליכם בחלקים שיבואו אחריו.

הפרויקט מאורגן בצורה מרווחת מאוד כך שיש מספיק זמן לביצוע כל מטלה. כבר ניתנו הזמנים המקסימליים עבור כל מטלה, כל בקשה להארכה קולקטיבית משמעותה לבוא על חשבון המטלות האחרות. להארכות זמן פרטניות מסיבות מוצדקות בלבד יש לפנות למתרגל הקורס.

ציון "התרגול" יורכב מממוצע המטלות ומשקלו 30% מהציון הסופי בקורס. משקל הבחינה 70%.

לקורס זה קיים מבחן תכנותי. מועדי א' ו ב' מתקיימים בתקופת הבחינות המתאימה. **קיימת חובת מעבר מבחן כדי לעבור את הקורס.** רק מי שהשקיע בעצמו בפרויקט והפנים את החומר הנלמד בקורס יוכל לעבור את המבחן. בכלל זה אוסיף שהעתקה אינה משתלמת; מעבר לסיכון להיתפס ולהישלח (המעתיק והמועתק) לוועדת משמעת, כל צורה של העתקה (גם מפרויקטים דומים קודמים) אינה שווה כלל למימוש קוד עצמאי, ובכך היא פוגעת ביכולת שלכם להפנים את החומר ותוביל לכישלון במבחן.

בהצלחה,

אלי ואביעד.

הנחיות כלליות

אנו ממליצים בחום לקרוא את מסמך זה מתחילתו ועד סופו ולהבין את הכיוון הכללי של הפרויקט עוד בטרם התחלתם לממש. הבנת הדרישות ומחשבה מספקת לפני תחילת המימוש תחסוך לכם זמן עקב טעויות מיותרות. *"סוֹף מַעֲשֶׂה בְּמַחְשָׁבָה תְּחִלָּה"*.

מומלץ שתעבדו עם ***IntelliJ*** בגירסה העדכנית ביותר, כפי שמותקן במעבדות בהן מועבר בתרגול. את גירסת ה-Community החינמית, תוכלו להוריד מכאן:

https://www.jetbrains.com/idea/download/

עבור כל חלק בפרויקט זה, שעליו אתם מתחילים לעבוד אנו ממליצים:

1. לקרוא את כל הדרישות של החלק מתחילתם ועד סופם.
2. לדון על הדרישות עם בן/בת הזוג למטלה.
3. לחשוב איך אתם הולכים לממש את הדרישות.
4. לתכנן את חלוקת העבודה ביניכם.
5. להתחיל לממש.
6. לשמור ולגבות את הקוד שלכם במהלך העבודה במספר מוקדים שונים: ***Dropbox***, ***Email*** וכו'.
7. לבצע בדיקות עבור כל קוד שמימשתם. בדקו כל מתודה על מנת להבטיח שהיא מבצעת כראוי מה שהיא אמורה לבצע. זכרו שאתם יכולים לדבג (***Debug***) את הקוד שלכם ולראות מה קורה בזמן ריצה.
8. לשמור את התוצאה הסופית שאותה אתם הולכים להגיש במספר מוקדים שונים.

דגשים לכתיבת קוד:

- הפונקציות צריכות להיות קצרות, עד 30 שורות ולעסוק בעניין אחד בלבד. פונקציות ארוכות ומסובכות שעוסקות בכמה עניינים הם דוגמא לתכנות גרוע.
- הפונקציות צריכות להיות גנריות (כלליות), שאינן תפורות למקרה ספציפי.
- שמות משתנים ברורים ובעלי משמעות.
- שמות שיטות ברורים ובעלי משמעות.
- מתן הרשאות מתאימות למשתנים ולמתודות (***public***, ***protected***, ***private***). כימוס (***Encapsulation***)[1].
- שימוש נכון בתבניות העיצוב שנלמדו בכיתה, בירושה ובממשקים.
- תיעוד הקוד:
  - תיעוד מעל מחלקות, שיטות וממשקים.
  - יש לתעד שורות חשובות בתוך המימוש של השיטות.
  - הסבר על תיעוד Javadoc ניתן למצוא כאן: https://www.tutorialspoint.com/java/java_documentation.htm

בסיום כתיבת הפתרון:

1. הריצו את הפרויקט ובדקו אותו ע"פ הדרישות של החלק אותו מימשתם.
2. עברו שוב על דרישות המטלה ובדקו שלא פספסתם אף דרישה.
3. חשבו על מצבי קצה שאולי עלולים לגרום לאפליקציה שלכם לקרוס וטפלו בהם.

קחו בחשבון שהפרויקט שאתם מגישים נבדק על מחשב אחר מהמחשב שבו כתבתם את הקוד שלכם. לכן, אין להניח שקיים כונן מסוים (לדוגמא :D) או תיקיות אחרות בעת ביצוע קריאה וכתיבה מהדיסק.

[1] https://he.wikipedia.org/wiki/%D7%9B%D7%99%D7%9E%D7%95%D7%A1

בנוסף, כאשר אתם כותבים ממשק משתמש, בין אם זה ***Console*** או ***GUI***, קחו בחשבון שהמשתמש (או הבודק) אינו תמיד מבין מה עליו לעשות ולכן עלול לבצע מהלכים "לא הגיוניים" בנסיונו להבין את הממשק שלכם. מהלכים אלו יכולים לגרום לקריסה של הקוד אם לא עשיתם בדיקות על הקלט שהמשתמש הכניס או לא לקחתם בחשבון תרחישים מסוימים. לכן, חשוב שתעזרו למשתמש/בודק לעבוד מול הממשק שלכם ע"י הדפסה של הוראות ברורות מה עליו להקליד, על איזה כפתור ללחוץ (ומתי) ומה האופציות שעומדות בפניו. הנחיות אלו יפחיתו את הסיכויים לטעויות משתמש שעלולות לגרום לקריסה של האפליקציה ולהורדת נקודות.

חלקי הפרויקט שאתם מגישים נבדקים אוטומאטית (פרט לחלק ג' שיבדק גם ידנית) ע"י קוד בדיקה לכן חשוב ביותר להצמד להוראות ולוודא ששמות המחלקות והממשקים שהגדרתם הם <u>בדיוק</u> מה שהתבקשתם. הבדיקה האוטומאטית אינה סלחנית לכן השתדלו להמנע מטעויות מצערות.

לפני הגשת המטלות, בדקו את עצמכם שוב! פעמים רבות שינויים פזיזים של הרגע האחרון שנעשים ללא מחשבה מספקת, גורמים לשגיאות ולהורדת נקודות. חבל לאבד נקודות בגלל טעויות של הרגע האחרון.

הוראות הגשה

את המטלות יש להגיש ל:

1. למערכת הבדיקה האוטומאטית:

http://subsys.ise.bgu.ac.il/submission/login.aspx

2. ל-***FTP*** של הקורס. שם משתמש: ***P***, סיסמה: ***1***.

ftp://ftp.ise.bgu.ac.il/ATP2017/

3. לאתר ה-Moodle של הקורס:

https://moodle2.bgu.ac.il/moodle/course/view.php?id=16292

* מספיק שסטודנט אחד מתוך הצוות יעלה את העבודה למערכת הבדיקה האוטומאטית ול-Moodle.

אתם מגישים למקורות לעי"ל את התוצרים כקובץ ***Zip*** המכיל:

1. תיקיית הפרוייקט שלכם שמכילה:
   a. תיקיית ***src*** קוד המקור.
   b. תיקיות נוספות כגון resources.
   c. תיקיית ה-idea.
   d. וכו'..
2. קובץ ה-***JAR*** של הפרויקט שלכם.
   a. הוראות יצירה ב-IntelliJ: https://www.youtube.com/watch?v=3Xo6zSBgdgk

- בתיקיה שאתם מגישים מחקו קבצים ותיקיות מיותרות כגון קבצים זמניים או קבצים השייכים ל-Git. התיקיה המוגשת צריכה להיות רזה יחסית.

קובץ ה-***Zip*** ישא את שמות המגישים בפורמט: ***ID1_ID2.zip***. לדוגמא: ***012345678_123456789.zip***. במידה וברצונכם להעלות גרסה משופרת לפני תום מועד ההגשה תוכלו להעלות קובץ נוסף עם התוספת ***updateX*** כאשר ***X*** הינו מספר העדכון. לדוגמא ***012345678_123456789_update3.zip*** אם העליתם קובץ עדכון שלישי. בעת איסוף העבודות לבדיקה, רק הגרסה האחרונה והעדכנית ביותר תילקח לבדיקה.

חשוב להגיש את המטלות בזמן, מטלות שיוגשו לאחר המועד ללא הצדקה לא יבדקו.

הפרויקט המוגש לבדיקה צריך:

- להתקמפל ללא שגיאות. **פתרון שאינו מתקמפל כלל לא ייבדק וציונו יהיה 0.**
- לרוץ ולבצע את מה שהתבקש ללא שגיאות בזמן ריצה. **כל חריגה (Exception) שתגרום לקריסה של האפליקציה בזמן ריצה תגרור הורדת נקודות.**

בדיקת הפרויקט

המתרגל הוא גם בודק התרגילים בקורס, אביעד כהן (aviadjo@gmail.com). עם פרסום הציונים יפורסם גם מפתח בדיקה שיפרט את הניקוד עבור כל חלק שנבדק.

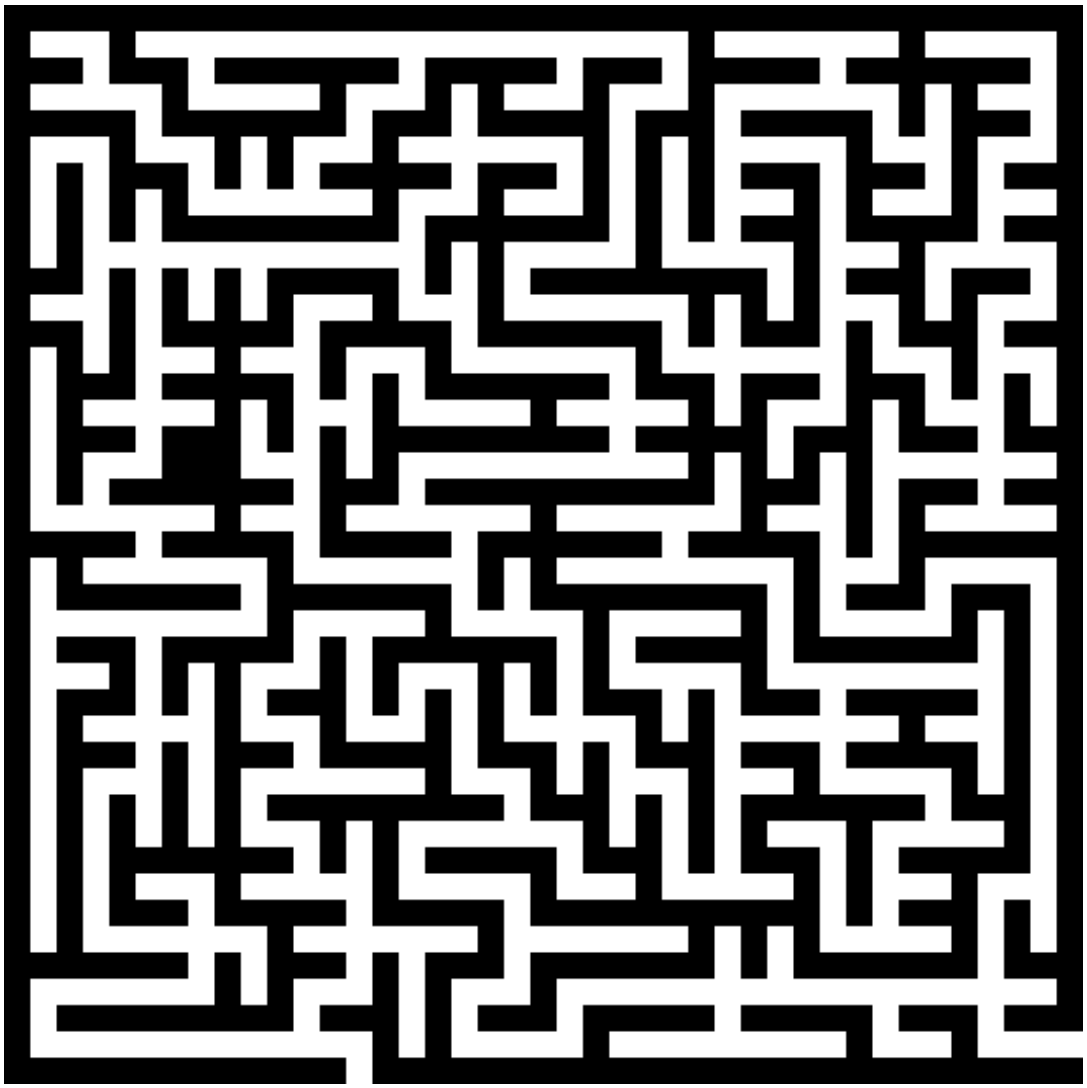
כפי שפורט קודם לכן, העבודות נבדקות אוטומאטית ע"י קוד בדיקה כך שאין מקום לערעורים מאחר והבדיקה זהה לכולם. **במקרים מיוחדים, ניתן להגיש ערעור עד 3 ימים ממועד פרסום הציונים.** פניות שיוגשו לאחר מכן לא יבדקו. הגשת ערעור תתבצע במייל בלבד ותכיל בכותרת המייל את השמות ות"ז של הסטודנטים. **שימו לב: הגשת ערעור תגרור בדיקה מחודשת ויסודית של העבודה ועלולה אף להוביל להורדה נוספת של נקודות, לכן אל תקלו ראש ותגישו ערעורים על זוטות.**

אין אפשרות לבדיקה חוזרת של מטלות. במקרים מיוחדים של טעויות קריטיות שהובילו להורדת ניקוד נרחבת עקב אי-יכולת לבדוק את המטלה תתאפשר בדיקה חוזרת אך יורדו 20 נקודות.

בהצלחה!

הקדמה - מבוך

מבוך הוא חידה הבנויה ממעברים מתפצלים, אשר על הפותר למצוא נתיב דרכה, מנקודת הכניסה לנקודת היציאה. הדמות יכולה לנוע ימינה, שמאלה, למעלה או למטה, במידה והיעד פנוי מקיר כמובן.



דוגמא למבוך

ייצוג המבוך

את המבוך עליכם לייצג כמערך דו-מימדי של int. הערך 1 מסמן תא מלא (קיר) ואילו 0 מסמן תא ריק.
לדוגמא:

```
int[][] maze={
                    {0,0,1,0,1,0,0,0,1},
                    {1,0,1,0,1,0,1,0,0},
                    {1,0,0,0,0,0,0,1,1},
                    {1,0,1,1,0,1,0,1,1},
                    {0,0,1,1,0,1,0,1,1},
                    {1,1,0,0,0,1,0,0,0},
              };
```

המבוך בעל שני מימדים, נקרא להם rows ו-columns המייצגים את מספר השורות והעמודות במבוך.
אין חובה ש-rows=columns, כלומר שהמבוך אינו בהכרח ריבוע, הוא יכול להיות גם מלבן.

חלק א': מפסאודו-קוד לתכנות מונחה עצמים

בשיעורים האחרונים למדנו כיצד לתרגם פסאודו-קוד של אלגוריתם לתכנות מונחה עצמים.

למדנו שני כללים חשובים:

כלל ראשון: להפריד את האלגוריתם מהבעיה שאותה הוא פותר.

כשנתבונן בפסאודו-קוד של האלגוריתם נסמן את השורות שהן תלויות בבעיה. שורות אלה יגדירו לנו את הפונקציונליות הנדרשת מהגדרת הבעיה. את הפונקציונליות הזו נגדיר בממשק מיוחד עבור הבעיה הכללית. מאוחר יותר מחלקות קונקרטיות יממשו את הממשק הזה ובכך יגדירו בעיות ספציפיות שונות.

שמירה על כלל זה תאפשר לאלגוריתם לעבוד מול טיפוס הממשק במקום מול טיפוס של מחלקה ספציפית. תכונת הפולימורפיזם תאפשר לנו להחליף מימושים שונים לבעיה מבלי שנצטרך לשנות דבר בקוד של האלגוריתם. **על אילו מעקרונות SOLID שמרנו כאן?**

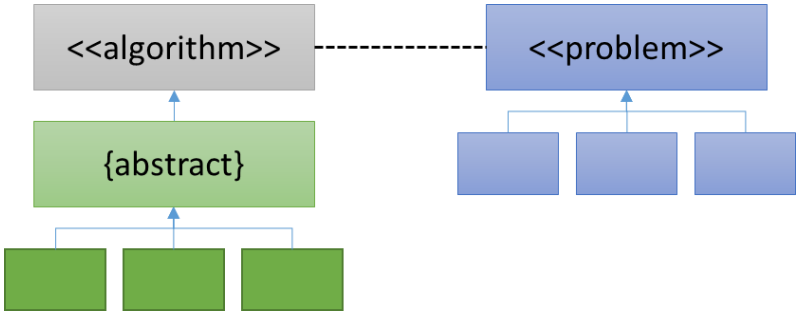
כלל שני: לממש את האלגוריתם באמצעות היררכיית מחלקות.

ייתכנו אלגוריתמים נוספים שנצטרך לממש בעתיד, או מימושים שונים לאותו האלגוריתם שלנו. לכן כבר עכשיו ניצור היררכיית מחלקות שבה

- את הפונקציונלית של האלגוריתם נגדיר בממשק משלו.
- את מה שמשותף למימושים השונים נממש במחלקה אבסטרקטית.
  - את מה שלא משותף – נשאיר אבסטרקטי.
- את המימושים השונים ניצור במחלקות שירשו את המחלקה האבסטרקטית.
  - הם יצטרכו לממש רק את מה ששונה בין האלגוריתמים.

את ההיררכיה הזו ניתן כמובן להרחיב ע"פ הצורך.

קבלנו את המבנה הבא:



על אילו מעקרונות SOLID שמרנו כאן?

משימה א' – אלגוריתם ליצירת מבוך

צרו פרויקט בשם ATP2017 – Project - PartA ובתוכו package בשם algorithms.mazeGenerators.

בפנים צרו מחלקה בשם Maze המייצגת מבוך כבתיאור לעיל. הוסיפו מתודות למחלקה זו כרצונכם ע"פ הצורך והדרישות בהמשך. מי שהולך ליצור מופעים של Maze יהיה הטיפוס MazeGenerator.

במשימה זו נתרגל את כתיבת ההיררכיה של המחלקות עבור אלגוריתם. את החלק של הבעיה נתרגל בחלק הבא של הפרויקט.

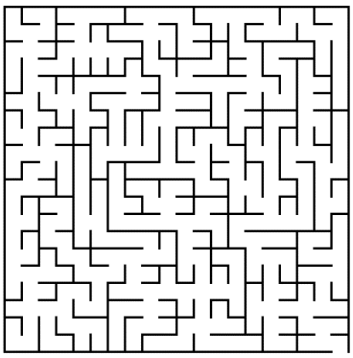
1. הגדירו ממשק בשם IMazeGenerator שמגדיר:
   a. מתודה בשם generate שמחזירה מופע של Maze. המתודה מקבלת שני פרמטרים, מס' שורות ומס' עמודות (כ-int).
   b. מתודה בשם measureAlgorithmTimeMillis מקבלת שני פרמטרים, מס' שורות ומס' עמודות (כ-int), ליצירת מבוך, ומחזירה long.

2. ממשו מחלקה אבסטרקטית כסוג של IMazeGenerator, קראו לה AMazeGenerator.
   a. היא תשאיר את המתודה generate כאבסטרקטית. כל אלג' יממש זאת בעצמו.
   b. לעומת זאת, פעילות מדידת הזמן זהה לכל האלגוריתמים ולמעשה אינה תלויה באלגוריתם עצמו. לכן אותה דווקא כן נממש כאן (במקום לממש אותה כקוד כפול בכל אחת מהמחלקות הקונקרטית).
   c. המתודה measureAlgorithmTimeMillis תדגום את שעון המערכת ע"י System.currentTimeMillis(), תפעיל את generate עם פרמטרים ליצירת מבוך שקיבלה ותדגום את הזמן שוב מיד לאחר מכן. הפרש הזמנים מתאר את הזמן שלקח להפעיל את generate. החזירו את זמן זה כ-long.

3. ממשו מחלקה בשם SimpleMazeGenerator (שתירש את המחלקה האבסטרקטית) שפשוט מפזרת קירות בצורה אקראית. צרו מסלול אקראי של חללים ריקים (ערך 0) מאיזושהי כניסה אקראית לאיזושהי יציאה אקראית כדי להבטיח שלמבוך יש פתרון.

4. למידה עצמית – עיקר התרגיל. היכנסו לעמוד הבא בוויקפדיה:
   https://en.wikipedia.org/wiki/Maze_generation_algorithm
   בחרו את אחד האלגוריתמים שאתם מתחברים אליו יותר. באמצעותו ממשו מחלקה בשם MyMazeGenerator, היורשת גם היא את המחלקה האבסטרקטית ומייצרת מבוך ע"פ הייצוג לעיל. **דאגו שהאלגוריתם שלכם מייצר מבוכים מעניינים עם מבויים סתומים והתפצלויות כפי הדוגמא לעי"ל.**

***** ישנם אלגוריתמים המתייחסים לכל תא במבוך כמוקף ב-4 קירות ובתהליך היצירה הם שוברים את הקירות באופן שמייצר מבוך. מבוכים הנוצרים נראים כך:***



***** אל תשתמשו באלגוריתמים כאלו! לחילופין בחרו אלגוריתמים הרואים בכל תא קיר, או מקום פנוי בלבד, כפי הייצוג שהודגם למעלה.***

בדיקות

הוסיפו לפרויקט שלכם Package חדש בשם test. ה-Package יכיל מספר מחלקות הניתנות להרצאה (כוללות פונקציית main) וכל מחלקה תבדוק קטעי קוד אחרים. הוסיפו מחלקה בשם RunMazeGenerator המכילה את הקוד הבא:

```
package test;

import algorithms.mazeGenerators.*;

public class RunMazeGenerator {
    public static void main(String[] args) {
        testMazeGenerator(new SimpleMazeGenerator());
        testMazeGenerator(new MyMazeGenerator());
    }

    private static void testMazeGenerator(IMazeGenerator mazeGenerator) {
        // prints the time it takes the algorithm to run
        System.out.println(String.format("Maze generation time(ms): %s",
mazeGenerator.measureAlgorithmTimeMillis(100/*rows*/,100/*columns*/)));
        // generate another maze
        Maze maze = mazeGenerator.generate(100/*rows*/, 100/*columns*/);

        // prints the maze
        maze.print();

        // get the maze entrance
        Position startPosition = maze.getStartPosition();

        // print the position
        System.out.println(String.format("Start Position: %s",
startPosition)); // format "{row,column}"

        // prints the maze exit position
        System.out.println(String.format("Goal Position: %s",
maze.getGoalPosition()));
    }
}
```

שימו לב ש:

- הקוד נדרש לרוץ במלואו ללא שגיאות.
- המחלקה maze מכילה את השיטות הבאות:
  - getStartPosition – מחזיר את נקודת ההתחלה של המבוך (טיפוס מסוג Position).
  - getGoalPosition – מחזיר את נקודת הסיום של המבוך (טיפוס מסוג Position).
  - Print – מדפיסה את המבוך למסך. סמנו את נקודת הכניסה למבוך בתו **S** ואת נקודת היציאה בתו **E**.
- כחלק ממימוש האלגוריתם שמייצר מבוך, תצטרכו לקבוע למבוך מהי נקודת ההתחלה ומהי נקודת הסיום של המבוך.
- עליכם ליצור מחלקה בשם Position המייצגת מיקום בתוך המבוך. מקמו את המחלקה לצד המחלקה Maze (תחת אותו ה-Package). למחלקה יהיו שני Data Members שייצגו את השורה והעמודה. ההדפסה של Position בקריאה מתוך System.out.println צריכה להחזיר את המיקום בפורמט {row,column}. המחלקה תכיל את המתודות הבאות:
  - getRowIndex
  - getColumnIndex
- ה-main בוחן את שני האלגוריתמים, קוד ה test לא היה צריך להשתנות!

משימה ב' – אלגוריתמי חיפוש

בספריית הקוד my algorithms, צרו package בשם algorithms.search

בהתאם לתשתית שראינו בהרצאה:

1. צרו את:
   a. מחלקות: ASearchingAlgorithm, AState, MazeState, Solution.
   b. ממשקים: ISearchable, ISearchingAlgorithm.
2. ממשו את שני אלגוריתמי החיפוש הבאים:
   a. חיפוש לרוחב Breadth First Search – קראו למחלקה BreadthFirstSearch.
   b. חיפוש לעומק Depth First Search – קראו למחלקה DepthFirstSearch.
   c. אלגוריתם Best First Search – קראו למחלקה BestFirstSearch.
3. צרו **Object Adapter** שמבצע אדפטציה ממבוך (מופע של Maze) לבעיית חיפוש (ISearchable), קראו לו SearchableMaze.
   a. במימוש המתודה getAllPossibleStates תוכלו להחליט אם ממיקום מסויים במבוך נתון להחזיר גם תנועות באלכסון (בנוסף לתנועה אפשריות כמו שמאלה, ימינה, למעלה ולמטה).
   b. תנועה אחת באלכסון מתא X לתא Y אפשרית רק אם ניתן להגיע מתא X ל-Y בשני צעדים רגילים ללא אלכסון (תנועה בצורת 'ר').

שימו לב ש Best First Search דומה מאד ל Breadth First Search פרט לעובדה שהראשון משתמש בתור עדיפויות. עדיף שתור העדיפויות יהיה ממומש כערמה (Heap) לשיפור הביצועים.

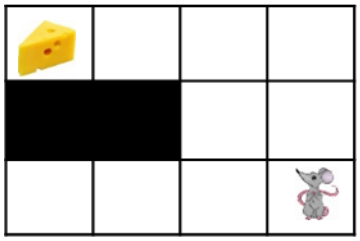
- ניתן לומר ש Best הוא סוג של ספציפי יותר של Breadth ולכן על המחלקה של Best לרשת את זו של Breadth ולדרוס את התור עם תור עדיפויות.
- מצד שני, ניתן לומר שמדובר באותו האלגוריתם ובאותו המימוש, פשוט ל Breadth First Search נזין שלכל הקודקודים עדיפות שווה.

שתי התשובות נכונות.

נשים לב שמספר הקודקודים שכל אלגוריתם ייפתח (מוציא מה open list) הוא שונה. ככל שמפתחים פחות קודקודים כך האלגוריתם יותר יעיל. כדי להבין זאת ולחוש את האלגוריתמים השונים, **לפני המימוש בקוד**, ענו כמה קודקודים ייפתח כל אלג' עבור הדוגמא הבאה:

נתון לנו מבוך שכל תנועה ישרה עולה 10 נקודות, ותנועה באלכסון עולה 15 נקודות (שימו לב שהיא יותר חסכונית משתי תנועות בעלות של 20 המביאות לאותה הנקודה).

נגדיר "מצב" (State) כמיקום של העבר במבוך (עמודה, שורה). במבוך העכבר נמצא ב (2,3). נגדיר שבהנתן מצב, סדר פיתוח השכנים הוא עם כיוון השעון כשמתחילים מלמעלה. כלומר, עבור המצב הנוכחי סדר פיתוח השכנים הוא 1. (1,3) 2. (1,4) 3. (2,4) וכך הלאה עד שנגיע ל (1,2). כמובן שלא נרצה לפתח מצבים שנמצאים מחוץ למבוך או מייצגים קירות. עבור כל אלגוריתם מצאו את מספר הקודקודים שהוא יפתח עד שימצא את המסלול הזול ביותר לגבינה.



בדיקות

תחת ה-Package לבדיקה שיצרתם (test) הוסיפו מחלקה בשם RunSearchOnMaze. צרו במחלקה מתודה Main ש:

a. יוצרת מבוך מורכב באמצעות MyMazeGenerator בגודל 30*30.
b. פותרת אותו באמצעות כל אחד משלושת מהאלגוריתמים.
   i. BreadthFirstSearch
   ii. DepthFirstSearch
   iii. BestFirstSearch
c. עבור כל אלגוריתם:
   i. מדפיסה למסך כמה מצבים פיתח. אם לא מורגש הבדל, הגדילו את המבוך.
   ii. מדפיסה למסך את רצף הצעדים מנקודת ההתחלה לנקודת הסיום.

הריצו את הקוד הבא:

```
package test;

import algorithms.mazeGenerators.IMazeGenerator;
import algorithms.mazeGenerators.Maze;
import algorithms.mazeGenerators.MyMazeGenerator;
import algorithms.search.*;

import java.util.ArrayList;

public class RunSearchOnMaze {
    public static void main(String[] args) {
        IMazeGenerator mg = new MyMazeGenerator();
        Maze maze = mg.generate(30, 30);
        SearchableMaze searchableMaze = new SearchableMaze(maze);

        solveProblem(searchableMaze, new BreadthFirstSearch());
        solveProblem(searchableMaze, new DepthFirstSearch());
        solveProblem(searchableMaze, new BestFirstSearch());
    }

    private static void solveProblem(ISearchable domain, ISearchingAlgorithm
searcher) {
        //Solve a searching problem with a searcher
        Solution solution = searcher.solve(domain);
        System.out.println(String.format("'%s' algorithm - nodes evaluated:
%s", searcher.getName(), searcher.getNumberOfNodesEvaluated()));
        //Printing Solution Path
        System.out.println("Solution path:");
        ArrayList<AState> solutionPath = solution.getSolutionPath();
        for (int i = 0; i < solutionPath.size(); i++) {
            System.out.println(String.format("%s.
%s",i,solutionPath.get(i)));
        }
    }
}
```

משימה ג' – Unit Testing (למידה עצמית)

עוד לפני שנתחיל לכתוב את ה GUI נכיר עוד כלי שמאפשר לנו לבדוק את הקוד בפרויקט – Unit Testing.

תפקידינו כמפתחים הוא גם לבדוק את המחלקות שהוא אחראי להן. הוא מתחייב שכל מחלקה שהוא מעלה ל repository היא בדוקה ונמצאה אמינה. מאוחר יותר ה QA בודק האם החלקים השונים של הפרויקט מדברים זה עם זה כמו שצריך ואין בעיות שנוצרות ביניהם.

אחד הכלים המוצלחים נקרא Junit. הרעיון הוא שלכל מחלקה חשובה שכתבנו תהיה לה גם מחלקת JUnit test שבודקת אותה. כך, לאחר שביצענו שינויים בקוד, נריץ תחילה את ריצת הבדיקה, ואם כל הבדיקות "עברו" אז נוכל להריץ את הפרויקט ולהעלות אותו ל repository. במידה ולא עברו, נוכל לפי הבדיקה שכשלה לבודד את התקלה שגרמנו בעקבות השינוי. כך נחסך זמן פיתוח רב.

להלן דוגמא לעבודה עם JUnit ב-Intellij:

https://www.youtube.com/watch?v=Bld3644bIAo

ישנם הגורסים שאת קוד הבדיקות יש לכתוב עוד לפני שכותבים את המחלקה הנבדקת עצמה. כך, הבדיקה תיעשה ללא ההשפעה של הלך המחשבה שהוביל לכתיבת המחלקה, ועלול להיות מוטעה.

עליכם ליצור מחלקת בדיקה לאלגוריתם Best First Search, קראו למחלקה JUnitTestingBestFirstSearch. מקמו את המחלקה בחבילת (package) בדיקה חדשה בשם JUnit. השתמשו ב-JUnit5.

מה בודקים? לא את נכונות האלגוריתם, בשביל זה יש מדעני מחשב. עליכם לבדוק את המימוש של האלגוריתם. כיצד הוא מתנהג עבור פרמטרים שגויים? Null? תבדקו מקרי קצה.

הכל עובד?

מצוין!

משימה ד' – עבודה עם מנהל גרסאות (למידה עצמית)

מעתה אתם עובדים בזוגות. אנו מדמים את המציאות בה אנו מתכנתים כחלק מצוות תכנות. אחד האתגרים הוא ניהול העבודה, ובפרט ניהול הקוד. לא נרצה שתדרסו את הקוד של אחד ע"י השני, או שתלכו לאיבוד בין אינסוף גרסאות ששלחתם במייל... כמו כן, נרצה לשמור גרסאות קודמות כדי שנוכל לחזור לגרסא עובדת במקרה של תקלות או כדי לתמוך במשתמשים בעלי גרסאות קודמות של המוצר שלנו.

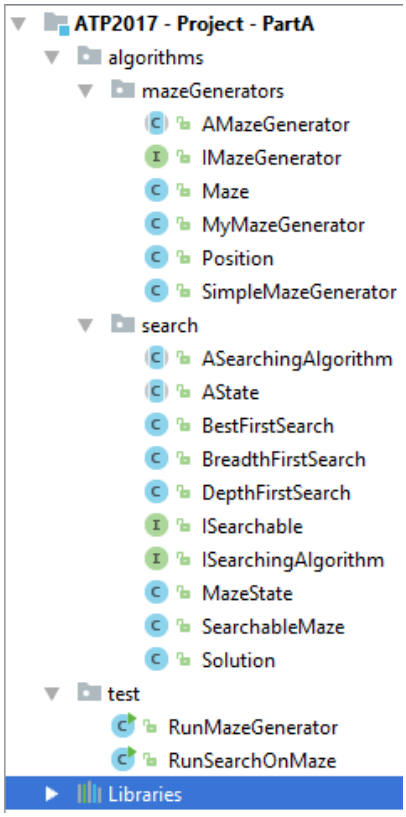
לשם צוותי פיתוח משתמשים במנהל גרסאות.

הרעיון הוא פשוט: בתחילת יום עבודה מורידים ממאגר הקוד שלנו (היושב על שרת כלשהו) את הגרסא האחרונה. עושים את השינויים שלנו על עותק מקומי. לאחר שאנו בטוחים שהשינוי עובד כראוי אנו מעלים אותו חזרה לשרת ומעדכנים את כולם בקוד שלנו. עליכם להתחיל לעבוד כמו המקצוענים.

בפרויקט זה עליכם לעבוד מול GIT. לימדו עצמאית כיצד להגדיר מאגר. יש המון הדרכות וסרטוני הדרכה בנושא ברשת המדגימים את הנושא. כעת צרו פרויקט hello world פשוט ותתנהלו מולו עם שינויים בקוד עד שתבינו כיצד לנהוג כשותפים לאותו מאגר קוד. לאחר שהבנתם כיצד לעבוד יחד תוכלו להתחיל את העבודה על המטלה בפרויקט חדש.

דגשים להגשה

בדקו שהפרויקט שלכם מכיל את החבילות, המחלקות והממשקים בפורמט הבא במדוייק:



***** שימו לב שיש חשיבות לאותיות גדולות וקטנות בטקסט (Case Sensitive).***

**** וודאו שקוד הבדיקה שניתן לכם רץ אצלכם ועובד עם הקוד שלכם כמות שהוא בלי שערכתם בו שינויים.**

בהצלחה!

חלק ב': עבודה עם Streams ו-Threads

הקדמה

בהמשך הפרויקט אנו ניצור זוג שרתים שנותנים שרות לשלל לקוחות. תפקיד השרת הראשון לייצר מבוכים לפי דרישה. תפקיד השרת השני לפתור מבוכים. כאשר לקוח מתחבר לשרת שיוצר מבוכים הוא ישלח לו את הפרמטרים ליצירת המבוך ויקבל חזרה אובייקט מבוך. כאשר לקוח מתחבר לשרת שפותר מבוכים הוא ישלח לו מבוך קיים ויקבל חזרה מהשרת פתרון למבוך. כדי לקצר את זמן התקשורת יהיה עלינו **לדחוס** את המידע שעובר ביניהם טרם השליחה. הצד המקבל **יפתח** את הדחיסה וייהנה מהמידע. עוד דבר שנעשה בצד השרת זה לשמור פתרונות שכבר חישבנו, כך שאם נתבקש לפתור בעיה שכבר פתרו נשלוף את הפתרון מהקובץ במקום לחשב אותו מחדש. תהליך זה מכונה caching.

לשם כך, בחלק זה של המטלה אנו נתרגל עבודה עם קבצים, נממש אלג' דחיסה פשוט, ונכיר כמה מחלקות מוכנות. כדי שנוכל לעבוד עם הדחיסה שלנו גם מעל ערוץ תקשורת וגם מעל קבצים נצטרך לכתוב את הקוד שלנו כחלק מה decorator pattern של מחלקות ה streams ב-Java. זה ייתן לנו גמישות מרבית, שכן, המקור ממנו מגיע המידע לא קריטי לנו.

תוכלו להמשיך לפתח את חלק זה (חלק ב') על גבי הפרויקט שכבר שהגשתם (חלק א'). לפני שתמשיכו גבו את חלק א' שכבר הגשתם. שנו את שם הפרויקט שלכם ל-ATP2017 – Project – PartB.

משימה א' – דחיסה של Maze ו-Decorator design pattern

כמה מידע באמת מחזיק מופע של Maze?

הוא מחזיק את נק' הכניסה והיציאה מהמבוך, וכמובן את הגדרת המבוך. הגדרה זו די בזבזנית. כך שאם Maze תהיה serializable ונשלח מופע שלה בתקשורת או נשמור אותו בקובץ, בזבזנו המון מקום מיותר.

איך ניתן לכווץ את המידע?

דבר ראשון אנו משתמשים ב int (גודל של 4 בתים) כדי לייצג את הערכים 0 או 1, חבל. להשתמש ב byte בודד כבר יחסוך לנו פי 4 בתים.

כעת, דמיינו שאתם פורסים את המערך הדו-מימדי שמייצג את המבוך שלנו למערך חד-ממדי ארוך. יש בו המון רצפים שחוזרים על עצמם, לדוגמא:

1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,...

במקום לשמור את הרצף עצמו, נוכל לשמור את מספר ההופעות ברצף של כל ספרה. עלינו להגדיר עם איזו ספרה אנו מתחילים, למשל 0, ואז נרשום את מספר ההופעות ברצף של 0, המספר הבא יהיה מספר ההופעות ברצף של 1, ואחריו שוב עבור 0 וחוזר חלילה. עבור הדוגמא לעיל נקבל: 0,7,10,6. כך נדע ש 0 לא מופיע, 1 מופיע 7 פעמים ברצף, ואז 0 מופיע 10 פעמים ברצף, ואז 1 6 פעמים ברצף וכך הלאה...

נצטרך משמעותית פחות בתים כדי לייצג את אותו המידע בדיוק.

נשים לב שאם אנו משתמשים בבתים אז המקסימום שנוכל לכתוב הוא 255 (unsigned). כך שאם למשל "1" הופיע 300 פעם ברצף, נצטרך לכתוב 255,0,45 כלומר 1 הופיע 255 פעמים ברצף, 0 לא הופיע, ואז 1 הופיע 45 פעם ברצף. אם ניצור מחלקה שדוחסת את המידע של המבוך, שומרת את ממדיו הדו-ממדיים, וכן שומרת את מיקומי הכניסה והיציאה מהמבוך, אז נוכל באמצעותה לשמור \ להעביר את המבוך בכמות פחותה של בתים באופן משמעותי, ולאחר מכן לשחזר בדיוק את אותו המבוך.

איך נממש זאת נכון מבחינת design? נשתלב ב decorator pattern של ה streams ב java.

בספריית הקוד שלנו צרו חבילה בשם IO (מקבילה לחבילה algorithms) בתוכה, צרו מחלקה בשם MyCompressorOutputStream. מחלקה זו תירש את OutputStream ותקבל בבנאי שלה OutputStream. זה כמובן יחייב אתכם לממש את: `void write(int b)`.

כעת, צרו data member בשם out מהסוג OutputStream ואתחלו אותו בבנאי עם מופע של OutputStream שקיבלם. את write תממשו בהמשך באמצעותו.

נניח שמחלקה זו קבלה מערך של בתים לכתוב, היא תפעיל את write עבור כל אחד מבתים אלה. כל שעליכם לעשות הוא לבדוק האם b הוא בית חדש או שהוא חזר על עצמו מההפעלה הקודמת של write. כל עוד זו חזרה אז נצבור את הפעמים. ברגע שנקבל בית חדש, נשתמש ב out כדי לרשום גם את התו וגם את מספר ההופעות שלו, ונתחיל את הצבירה מחדש עבור הבית החדש שקבלנו זה עתה.

באופן דומה תוכלו לממש את הכיוון ההפוך במחלקה MyDecompressorInputStream, שתממש את InputStream באמצעות מופע של InputStream שתקבל בבנאי. תקראו לו in. באמצעות in נקרא מידע מכווץ ממקור המידע שלנו. "ננפח" את המידע בהתאם לשיטת הכיווץ לעיל ונזין את מי שקורא מידע ממחלקה זו במידע המנופח.

כעת, במחלקה Maze נוסיף שני דברים:

1. את המתודה toByteArray שתחזיר byte[] המייצג את כל המידע (הלא מכווץ) של המבוך: ממדי המבוך, תוכן המבוך, מיקום כניסה, מיקום יציאה. החליטו בעצמכם על הפורמט שבאמצעותו תייצגו את המבוך כ-byte[]. נסו להיות חסכוניים ככל האפשר.
2. בנאי שמקבל מערך של byte לא מכווץ (לפי הפורמט שאתם מחזירים מהסעיף הקודם) ובונה באמצעותו את Maze.

בדיקות

תחת ה-Package לבדיקה שיצרתם (test) הוסיפו מחלקה בשם RunCompressDecompressMaze. צרו במחלקה מתודה Main:

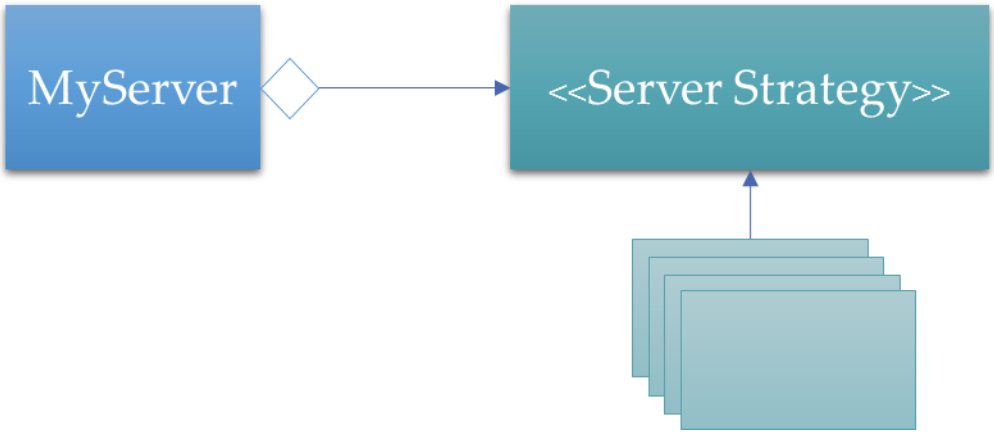
```
package test;

import IO.MyCompressorOutputStream;
import IO.MyDecompressorInputStream;
import algorithms.mazeGenerators.AMazeGenerator;
import algorithms.mazeGenerators.Maze;
import algorithms.mazeGenerators.MyMazeGenerator;

import java.io.*;

public class RunCompressDecompressMaze {
    public static void main(String[] args) {
        String mazeFileName = "savedMaze.maze";
        AMazeGenerator mazeGenerator = new MyMazeGenerator();
        Maze maze = mazeGenerator.generate(100, 100); //Generate new maze

        try {
            // save maze to a file
            OutputStream out = new MyCompressorOutputStream(new
FileOutputStream(mazeFileName));
            out.write(maze.toByteArray());
            out.flush();
            out.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        byte savedMazeBytes[] = new byte[0];
        try {
            //read maze from file
            InputStream in = new MyDecompressorInputStream(new
```

```
FileInputStream(mazeFileName));
            savedMazeBytes = new byte[maze.toByteArray().length];
            in.read(savedMazeBytes);
            in.close();
        } catch (IOException e) {
            e.printStackTrace();
        }

        Maze loadedMaze = new Maze(savedMazeBytes);
        boolean areMazesEquals =
Arrays.equals(loadedMaze.toByteArray(),maze.toByteArray());
        System.out.println(String.format("Mazes equal: %s",areMazesEquals));
//maze should be equal to loadedMaze
    }
}
```

בצעו את הניסוי הבא על מופע קיים של Maze, מה גודל המבוך בבתים? מה גודל המבוך ביצוג החסכוני? מה גודל הקובץ שנשמר בבתים? האם הקריאה מהקובץ הניבה מבוך זהה בדיוק?

משימה ב' - שרת-לקוח ו-Threads

בתרגול ראינו דוגמת שרת-לקוח המטפלת בלקוח אחד. העיצוב אפשר לנו ליצור שרת גנרי לשימוש חוזר בכל פרויקט באמצעות ה-Strategy Pattern.



בהרצאה ראינו דוגמת שרת-לקוח המטפלת בלקוחות במקביל, אך גם דיברנו על מס' מקרי קצה שניתן לפתור באמצעות thread pool. בתרגול, ראינו כיצד להשתמש ב-thread pool מוכן מהספרייה java.util.concurrent.

צרו חבילה חדשה בשם Server (לצד IO) ובתוכה צרו את המחלקה Server שראיתם בתרגול יחד עם ה-ServerStrategy (שם חדש במקום ClientHandler) הבאים:

1. ServerStrategyGenerateMaze – השרת מקבל מהלקוח מערך int[] בגודל 2 בלבד כאשר התא הראשון מחזיק את מספר השורות במבוך, התא השני את מספר העמודות. השרת מייצר מבוך ע"פ הפרמטרים, דוחס אותו בעזרת MyCompressorOutputStream ושולח חזרה ללקוח byte[] המייצג את המבוך שנוצר.
   a. הלקוח יקבל מהשרת את המערך, יפענח אותו, ויוכל בעזרתו מופע של מבוך תואם.
2. ServerStrategySolveSearchProblem – השרת מקבל מהלקוח אובייקט Maze המייצג מבוך. פותר אותו ומחזיר ללקוח אובייקט Solution המחזיק את הפתרון של המבוך.

משימות:

1. טפלו בקוד השרת הגנרי כך שיתמוך במספר לקוחות במקביל ע"י שימוש ב-thread pool, ויטפל במקרה הקצה של היציאה המסודרת. השרת והלקוח אינם מתחזקים קשר רציף אלא רק סשן של שאלה-תשובה. הלקוח שולח בקשה, השרת משיב בתשובה ואז הקשר נסגר. עבור בקשה חדשה יש לפתוח את התקשורת מחדש.
2. צרו שרת היוצר מבוכים ע"פ הפרוטוקול לעי"ל.
3. צרו שרת הפותר בעיות חיפוש ע"פ הפרוטוקול לעי"ל.
   a. השרת שומר את הפתרון למבוכים שהוא מקבל בדיסק, כל פתרון נשמר בקובץ נפרד. את המבוכים שהשרת פותר תוכלו לשמור לתיקיה זמנית. על מנת לקבל את התיקיה הזמנית במערכת שלכם השתמשו ב:

```
String tempDirectoryPath = System.getProperty("java.io.tmpdir");
```

   b. אם השרת נדרש לפתור בעיה שהוא כבר פתר בעבר, הוא ישלוף את הפתרון מהקובץ ויחזיר אותו ללקוח מבלי לפתור את הבעיה שוב.

צרו חבילה חדשה בשם Client (לצד IO) ובתוכה צרו את המחלקה Client שראינו בתרגול ואת הממשק IClientStrategy המגדיר את המתודה:

```
void clientStrategy(InputStream inFromServer, OutputStream outToServer);
```

השימוש במחלקה Client ובממשק clientStrategy הוא עבור בדיקת השרתים בלבד (מופיע בהמשך). אין חובה להשתמש במחלקה ו/או בממשק כאשר אתם נדרשים לפנות לשרת (שימו לב שהמתודה ClientStrategy של המחקלה Client אינה מחזירה ערך).

בדיקות

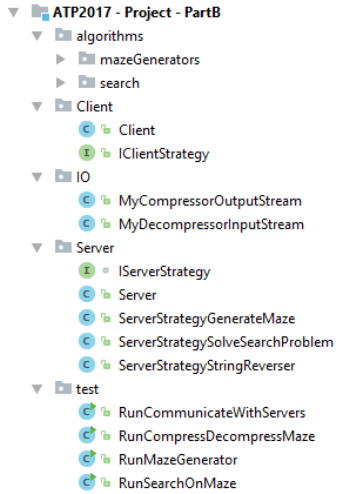
תחת ה-Package לבדיקה שיצרתם (test) הוסיפו מחלקה בשם RunCommunicateWithServers. צרו במחלקה מתודה Main:

```
public class RunCommunicateWithServers {
    public static void main(String[] args) {
        //Initializing servers
        Server mazeGeneratingServer = new Server(5400, 1000, new
ServerStrategyGenerateMaze());
        Server solveSearchProblemServer = new Server(5401, 1000, new
ServerStrategySolveSearchProblem());
        //Server stringReverserServer = new Server(5402, 1000, new
ServerStrategyStringReverser());

        //Starting  servers
        solveSearchProblemServer.start();
        mazeGeneratingServer.start();
        //stringReverserServer.start();

        //Communicating with servers
        CommunicateWithServer_MazeGenerating();
        CommunicateWithServer_SolveSearchProblem();
        //CommunicateWithServer_StringReverser();

        //Stopping all servers
        mazeGeneratingServer.stop();
        solveSearchProblemServer.stop();
        //stringReverserServer.stop();
    }

    private static void CommunicateWithServer_MazeGenerating() {
        try {
            Client client = new Client(InetAddress.getLocalHost(), 5400, new
IClientStrategy() {
                @Override
                public void clientStrategy(InputStream inFromServer,
OutputStream outToServer) {
                    try {
                        ObjectOutputStream toServer = new
ObjectOutputStream(outToServer);
                        ObjectInputStream fromServer = new
ObjectInputStream(inFromServer);
                        toServer.flush();
                        int[] mazeDimensions = new int[]{50, 50};
                        toServer.writeObject(mazeDimensions); //send maze
dimensions to server
                        toServer.flush();
                        byte[] compressedMaze = (byte[])
fromServer.readObject(); //read generated maze (compressed with
MyCompressor) from server
                        InputStream is = new MyDecompressorInputStream(new
ByteArrayInputStream(compressedMaze));
                        byte[] decompressedMaze = new byte[1000 /*CHANGE
SIZE ACCORDING TO YOU MAZE SIZE*/]; //allocating byte[] for the decompressed
maze -
                        is.read(decompressedMaze); //Fill decompressedMaze
```

```
with bytes
                        Maze maze = new Maze(decompressedMaze);
                        maze.print();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            client.communicateWithServer();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }

    private static void CommunicateWithServer_SolveSearchProblem() {
        try {
            Client client = new Client(InetAddress.getLocalHost(), 5401, new
IClientStrategy() {
                @Override
                public void clientStrategy(InputStream inFromServer,
OutputStream outToServer) {
                    try {
                        ObjectOutputStream toServer = new
ObjectOutputStream(outToServer);
                        ObjectInputStream fromServer = new
ObjectInputStream(inFromServer);
                        toServer.flush();
                        MyMazeGenerator mg = new MyMazeGenerator();
                        Maze maze = mg.generate(50, 50);
                        maze.print();
                        toServer.writeObject(maze); //send maze to server
                        toServer.flush();
                        Solution mazeSolution = (Solution)
fromServer.readObject(); //read generated maze (compressed with
MyCompressor) from server

                        //Print Maze Solution retrieved from the server
                        System.out.println(String.format("Solution steps:
%s", mazeSolution));
                        ArrayList<AState> mazeSolutionSteps =
mazeSolution.getSolutionPath();
                        for (int i = 0; i < mazeSolutionSteps.size(); i++) {
                            System.out.println(String.format("%s. %s", i,
mazeSolutionSteps.get(i).toString()));
                        }
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            client.communicateWithServer();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }

    private static void CommunicateWithServer_StringReverser() {
        try {
            Client client = new Client(InetAddress.getLocalHost(), 5402, new
IClientStrategy() {
                @Override
                public void clientStrategy(InputStream inFromServer,
OutputStream outToServer) {
                    try {
                        BufferedReader fromServer = new BufferedReader(new
InputStreamReader(inFromServer));
                        PrintWriter toServer = new PrintWriter(outToServer);
```

```
                        String message = "Client Message";
                        String serverResponse;
                        toServer.write(message + "\n");
                        toServer.flush();
                        serverResponse = fromServer.readLine();
                        System.out.println(String.format("Server response:
%s", serverResponse));
                        toServer.flush();
                        fromServer.close();
                        toServer.close();
                    } catch (Exception e) {
                        e.printStackTrace();
                    }
                }
            });
            client.communicateWithServer();
        } catch (UnknownHostException e) {
            e.printStackTrace();
        }
    }
}
```

בדקו שהקוד שלכם מאורגן באופן הבא:



משימה ג' – קובץ הגדרות (לימוד עצמי)

כמה ת'רדים צריך לאפשר ב Thread Pool? באיזה אלגוריתם לפתור את המבוכים? ובאיזה ליצור אותם?

כל אלו הן הגדרות. אסור לנו לקבוע אותן באופן שרירותי בקוד. המשתמש אולי ירצה לשנות אותן. גם השתמשנו במשתנים הנמצאים בתוך הקוד עבור ההגדרות האלו, אז כדי לשנות אותן נצטרך לשנות את קוד המקור שלנו ולבנות (לקמפל) את הפרויקט מחדש! רעיון גרוע מאד...

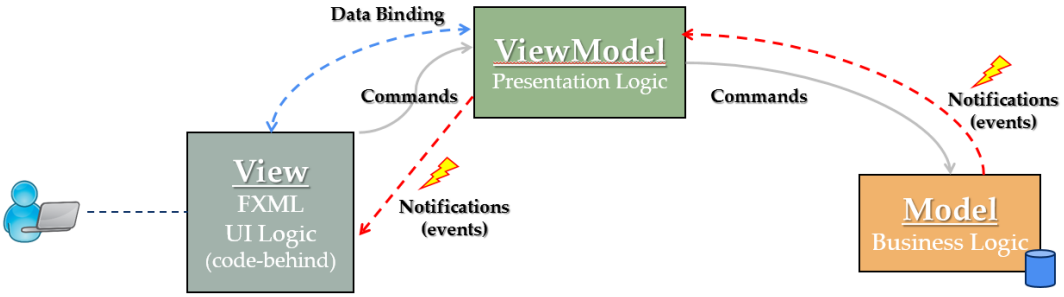
בתוך ה-Package שבו יצרתם את השרתים, צרו מחלקה סטאטית בשם Configurations המאפשרת השמה ושליפה של כל ההגדרות שנראה לכם שצריך בתוכנית שלנו. לקובץ ההגדרות קראו config.properties, מקמו את הקובץ בתוך הפרויקט בתיקיית Resources (לצד תיקיית ה-src). השרתים שתצרו ישלפו את ההגדרות מתוך המחלקת ההגדרות ויפעלו בהתאם. להסבר על קובץ קונפיגורציה ב-Java:

http://www.mkyong.com/java/java-properties-file-examples/

בהצלחה!

חלק ג': אפליקצית Desktop בארכיטקטורת MVVM, תכנות מונחה אירועים, ו-GUI

משימה א' – ארכיטקטורת MVVM



- פתחו פרויקט חדש מסוג JavaFX ובתוכו צרו את ה-Packages הבאים המהווים שכבות קוד שונות:
  - View
  - ViewModel
  - Model
- בשכבת ה-View:
  - מקמו את קובץ ה-fxml הראשי של הבפרויקט ושנו את שמו ל-MyView.
  - מקמו את ה-controller של קובץ ה-fxml ושנו את שמו ל-MyViewController.
  - צרו ממשק בשם IView והגדירו את המתודות הרלוונטיות לשכבה.
  - דאגו ש-MyViewController יממש את IView.
- בשכבת ה-Model:
  - צרו את המחלקה MyModel יחד עם הממשק IModel שאותו הוא יממש.
  - הגדירו בעצמכם מהן המתודות של IModel.
- בשכבת ה-ViewModel
  - צרו מחלקה בשם MyViewModel.
- צרפו לפרויקט את קבצי ה-JAR מחלקי הפרויקט הקודמים. זה יאפשר לכם לעשות שימוש במחלקות שכתבתם באבני הדרך הקודמים (Maze, MazeGenerator, Server, etc.).
- בחרו בעצמכם אילו תכונות ה-MyView נרצה לכרוך (Bind) ל-MyViewController.

שימו לב:

- בשכבת ה-View בלבד יהיו כל החלקים הקשורים בתצוגה:
  - קובץ או קבצי ה-fxml יחד עם ה-controllers שלהם.
  - קבצי עיצוב CSS.
  - פקדים שיצרתם בעצמכם.
- שכבת ה-Model בלבד אחראית על:
  - התקשורת לשרתים שיוצרים ופותרים מבוכים.
  - שימוש באלגוריתמים.
  - שמירת המבוך שכרגע המשתמש משחק בו.
  - שמירת מיקום הדמות במבוך הנוכחי.
- שכבת ה-ViewModel אחראית על החיבור בין שכבת ה-View לשכבת ה-Model.

משימה ב' – תכנות מונחה אירועים ו GUI

יצירת רכיב גרפי – תזכורת

א. כאשר אנו כותבים GUI כמו כל דבר אחר עלינו להקפיד על חלוקה נכונה למחלקות. לא הכל נכתב במחלקה אחת או ב-main אחד חלילה.

למשל כשאנחנו רוצים לממש לוח משחק או מבוך, מדובר באוסף של controls ו listeners שלא נרצה שיתערבבו יחד עם אלו שמגדירים את תוכן החלון (כפתורים תפריטים וכו'). כי אם נרצה להסיר את הרכיב הזה מהפרויקט או לחילופין להוסיף אותו לפרויקט אחר, תהיה לפנינו מלאכת תפירה קפדנית ומציקה.

תארו לכם לדוג' שאבקש להציג (רק) את המבוך בפרויקט אחר, או להחליף את המבוך בפרויקט שלכם במשחק לוח אחר... אם משחק הלוח שלכם אינו יחידה עצמאית אלא אוסף של מחלקות ו listeners מעורבבים יחדיו במחלקה גדולה אחרת יהיה לכם מאד קשה לבצע זאת.

במקום זאת, את כל אוסף הרכיבים השונים שמרכיבים את אותו לוח משחק נאגד בתוך מחלקה אחת משלו; הרי, בדומה למחלקת Button, מדובר באלמנט אחד שלם ונפרד מהחלון שמכיל אותו. נרצה שכל מתכנת יוכל להוסיף או להסיר את האלמנט הזה באותה הקלות שהוא מוסיף או מסיר כפתור.

ב. העובדה שהקוד של לוח המשחק נמצא במחלקה אחת אינה מספיקה לכך כדי שמתכנתים אחרים יוכלו להוסיף את הרכיב הזה בקלות בפרויקט שלהם ולהשתמש בו. לשם כך עלינו ליצור משהו שהם כבר מכירים ויודעים כיצד לנהוג בו. במילים אחרות, אנו צריכים ליצור widget.

בשיעור המעבדה השני אודות JavaFX למדנו כיצד ליצור control משלנו. ניתן לעשות זאת בקלות ע"י ירושה של Canvas שהוא כבר סוג של control שמימש לא מעט דברים ומאד גמיש לעבודה.

ג. שימו לב שה control שאותו אתם כותבים יודע ***מתי*** קורים דברים אך לא בהכרח יודע ***מה*** לעשות בנידון.

תארו לכם מצב שמחלקת Button היתה גם מגדירה מה קורה כשלוחצים על הכפתור. לא הינו יכולים להשתמש בכפתור הזה בפרויקטים אחרים כי לא הינו יכולים להגדיר לו מה לעשות כשהכפתור נלחץ. מצד שני, רק הכפתור יודע מתי הוא נלחץ ואז הוא זה שצריך להפעיל משהו. לכן, כפי שראיתם בשיעור, נעשה שימוש ב strategy pattern כדי להגדיר לכפתור **מה** הוא צריך ליזום **כש**הוא נלחץ.

זה נעשה ע"י הזנה של אובייקט מסוג Listener כלשהו לכפתור, וכשהכפתור נלחץ הוא יפעיל את המתודה שאנחנו מימשנו. כך, **מתי** שהכפתור יודע שצריך להפעיל משהו, הוא מפעיל את **מה** שאנו הגדרנו לו.

באופן דומה, גם לוח המשחק שלכם כ control לא צריך להגדיר מה לעשות; הוא צריך לקבל זאת מבחוץ. וזאת כדי שיוכלו להשתמש בו גם בפרויקטים אחרים בהם **אותם האירועים גוררים פעולות אחרות.**

לדוגמא, במבוך נרצה לצייר איזושהי דמות. תהיה לנו פונקציית ציור הדמות, ונפעיל אותה בכל פעם שנצטרך לצייר את הדמות כשהגיע הזמן להזיז אותה. אם נממש את הפונקציה הזו במחלקת המבוך אז היא תהיה קבועה במחלקה זו. כלומר, בכל פרויקט תמיד מופיעה אותה הדמות. לעומת זאת, אם נקבל מבחוץ את פונקציית ציור הדמות אז כל פרויקט יוכל להגדיר בעצמו כיצד הדמות תיראה בדרך הרלוונטית לאותו הפרויקט. המבוך יודע מתי לצייר את הדמות, אך לא כיצד לצייר אותה, ולכן מקבל זאת מבחוץ.

דוגמא נוספת, נניח שהחלטתם שצריך להיות כפתור עזרה, והוא חלק בלתי נפרד מאובייקט לוח המשחק (זה לגיטימי). שמחים וטובי לב, בעת לחיצה על כפתור העזרה, ב listener של הכפתור לא עשיתם שום דבר מיוחד חוץ מלפנות למי שיודע מה לעשות בנידון, נניח איזשהו controller. לכאורה זה מצוין, הרי לא מימשתם בתוך ה listener מה לעשות, פשוט פניתם למישהו אחר.

אבל המימוש הזה ספציפי לפרויקט שלכם וכך הופך את לוח המשחק לפחות נייד. מי אמר שבפרויקט אחר יש לכם את אותו ה controller? הדרך הנכונה יותר היא לקבל את המופע של ה listener הזה מבחוץ, כך בכל פרויקט יוכלו להגדיר מה לעשות כשכפתור העזרה נלחץ. למשל לפתוח חלון נוסף או דפדפן אינטרנט, או פשוט לפנות ל controller שיעביר את הבקשה הלאה.

לסיכום, כשאתם מייצרים אלמנט גרפי, הקפידו על מימושו במחלקה משלו, מסוג widget, שמקבל מבחוץ את כל הפעולות שעשויות להיות שונות בפרויקטים שונים. כך, כולם יוכלו בקלות להסיר או להוסיף ואף להשתמש באלמנט הזה כרצונם בפרויקטים שלהם.

ד. הכנסת ה GUI שלכם לא צריכה לפגוע בשאר הקוד. שכבות ה presenter והמודל לא אמורות להיפגע מכך שמימשתם את הממשק של View בצורה שונה.

כתיבת ה-GUI

במשימה זו עליכם לכתוב את ה GUI של הפרויקט.

- תבדילו בין המרכיבים "הסביבתיים" של החלון (כפתורים, תפריטים וכו') לבין לוח המשחק שהוא רכיב עצמאי המצורף לפרויקט שלנו בהתאם לתזכורת לעיל. נרצה שתהיה לנו היכולת לשלוף את לוח המשחק כיחידה אחת לפרויקט אחר, וכן להחליף את לוח המשחק בקלות בפרויקט שלנו.
- כמו כן, עלינו להגדיר את הפונקציונאליות של לוח המשחק מבחוץ, כך שנוכל להתאים אותה כרצוננו בכל פרויקט.
- אפשרו למשתמש לבקש יצירה של מבוך ע"פ קריטריונים משתנים.
- המבוך כמשחק לוח – עליכם לאפשר למשתמש לנסות לפתור את המבוך בעצמו ע"י הזזת הדמות באמצעות חצי המקלדת (→←↓↑). תוכלו לתמוך במקביל גם באוספים נוספים של מקשים כגון 4,8,6,2 ב-Num Pad, או ב-a,s,w,d.
- הראו למשתמש בדרך יצירתית כלשהי שהוא הצליח לפתור את המבוך.
- אפשרו למשתמש לבקש פתרון למבוך שמוצג כרגע. הפתרון יוצג על גבי המבוך.
- אפשרו למשתמש לשמור את המבוך המוצג כרגע לקובץ בדיסק.
- אפשרו למשתמש לטעון מבוך ששמר בקובץ בעבר.

בונוס א' (5 נק'): הוספת מוזיקת רקע למשחק + מוזיקה מתאימה כאשר המבוך נפתר.

בונוס ב' (5 נק'): לחיצה על מקש Ctrl והזזת הגלגלת של העכבר תבצע zoom in \ out ללוח המשחק.

בונוס ג' (10 נק'): אפשרו למשתמש לגרור את הדמות על המסך באמצעות סמן העכבר (בנוסף לאפשרות של הזזתה ע"י מקשי המקלדת). הדמות חייבת לזוז בהתאם לתוואי הקירות (ולא לעבור דרכם) ואך ורק בדרכים האפשריות.

עיצוב ה-GUI

עצבו את חלון המשחק ואת כל מרכיביו כרצונכם, כל עוד אתם מקיימים את הדרישות הבאות:

1. אסור לקבע את גודל חלון המשחק. מותר למשתמש לשנות את גודלו. האלמנטים שבפנים צריכים להתייחס לגודל החלון.
2. צריך להיות תפריט עליון בחלון (menu). תוכלו להכניס בו איזה אלמנטים שאתם רוצים אך המינימום הוא תפריט עם האלמנטים הבאים. קחו אלמנטים מאפליקציות שאתם מכירים. סדרו אותם לפי סדר הגיוני.
    a. ***File***
        i. ***New*** – יצירת מבוך חדש.
        ii. ***Save*** – שמירת המבוך הנוכחי לקובץ בדיסק.
        iii. ***Load*** – טעינת מבוך שנשמר בדיסק והצגתו.
    b. ***Options***

i. ***Properties*** שפותח ומציג בצורה מסודרת את ההגדרות מתוך קובץ ההגדרות של האפליקציה.

c. ***Exit*** – יגרום ליציאה מסודרת מהתוכנית ללא קבצים או ת'רדים פתוחים. שימו לב שניתן לצאת גם ע"י לחיצה על הX-, גם אז היציאה צריכה להיות מסודרת. *הקפידו שאין קוד כפול.*

d. ***Help*** – נתוני עזר למשחק כגון ככלי המשחק, סימונים שונים על הלוח וכו'.

e. ***About*** – יכיל פרטים על המתכנתים, האלגוריתמים בהם אתם משתמשים ליצירת המבוך ולפתרונו וכו'.

3. בלוח המשחק שלכם צריך לעשות שימוש בתמונה. לדוגמא הדמות במבוך, או הרקע. אין בעיה להשתמש בגרפיקה וב-***Widgets*** לרוב הדברים אך לפחות אלמנט אחד צריך להיות תמונה.
4. הודעות שגיאה יוצגו מעתה בחלון (Alert).
5. תתפרעו! תוסיפו מה שבא לכם כל עוד לדעתכם זה ירשים את המשתמש, הבודק (או המראיין).

דגשים:

1. חשוב שלמשמש שעובד עם המשחק שלכם יהיה ברור איך להפעיל משחק מבוך ואיך לשחק. ממשק שמכיל המון כפתורים יגרום למשתמש ללחוץ על כפתורים כדי לנסות לשחק, לפעמים הוא יעשה זאת לא לפי הסדר שאתם חשבתם עליו מה שיכול לגרור שגיאות בזמן ריצה. תוכלו להשתמש ב-***Property***, ***disabled = true*** של הכפתורים כל מנת לנעול כפתורים שאתם לא רוצים שהמשתמש יקיש עליהם בשלב מסוים. תוכלו לפתוח את הכפתור ע"י ***disabled = false*** כרצונכם. לדוגמא, בטרם מוצג מבוך על המסך והמשתמש יכול לשחק, אין משמעות לכפתור ***Solve*** שפותר את המבוך. עדיף שהכפתור יהיה נעול מאשר שהוא יהיה פתוח ולחיצה עליו תביא לקריסה (במקרה הרע) או להודעה שלא ניתן לפתור את המבוך כי אין מבוך (במקרה הטוב).

2. כאשר אתם רוצים לקלוט מהמשתמש פרמטרים, לדוגמא עבור יצירת מבוך, תוכלו (אופציונאלי) לעשות זאת באמצעות חלון שיציג טופס ייעודי לכך. לאחר מילוי הטופס ולחיצה על כפתור ***OK*** תוכלו לשלוף מתוך הטופס את הפרטים שהמשתמש הקיש. בנוסף, חשוב שתעשו בדיקות על הקלט ואם יש בעיה עם הקלט הציגו הודעה מתאימה למשתמש.

3. בדקו שאין מקרי קצרה בהם האפליקציה קורסת.

בהצלחה!