תכנות מתקדם 1

אבני דרך 1,2 בפרויקט

ד"ר אליהו חלסצ'י
Eliahu.khalastchi.biu@gmali.com

רקע

פרויקט זה בקורס תכנות מתקדם הוא חלון הראווה שלכם כשתרצו להציג את הניסיון התכנותי שצברתם. פרויקט זה מכיל את האלמנטים הבאים:

- שימוש בתבניות עיצוב וארכיטקטורה
- תקשורת וארכיטקטורת שרת-לקוח
- שימוש במבני נתונים ובמסד נתונים
- הזרמת נתונים (קבצים ותקשורת)
- השוואה, בחירה והטמעה של אלגוריתמים בתוך המערכת שניצור
- תכנות מקבילי באמצעות ת'רדים
- תכנות מוכוון אירועים, אפליקציית desktop עם GUI
- תכנות מוכוון אירועים, אפליקציית Web בסגנון REST
- אפליקציית מובייל (אנדרואיד)

בסמסטר זה עליכם להגיש 2 אבני דרך:

1. מפרש קוד המאפשר שליטה מרחוק בסימולטור טיסה
2. מימוש של מספר אלגוריתמי חיפוש, השוואה ביניהם מי הכי מוצלח, והטמעת המנצח כפותר הבעיות בצד השרת. כך נטמיע את צורת העבודה המלאה של בוגר מדעי המחשב.

בהצלחה!

אבן דרך 1 – מפרש קוד (interpreter) השולט מרחוק במל"ט

היכרות עם סימולטור הטיסה

ברצוננו לכתוב מפרש לקוד שליטה במל"ט (מטוס ללא טייס). המטוס שלנו יטוס במרחב הווירטואלי של סימולטור הטיסה FlightGear. את סימולטור הטיסה תוכלו להוריד מ http://home.flightgear.org/ .

בין היתר, סימולטור זה מהווה גם שרת שאפשר להתחבר אליו כלקוח (ולהיפך). כך נוכל בקלות לשלוף מידע אודות הפרמטרים השונים של הטיסה בזמן אמת ואף להזריק לו פקודות שינהגו את המטוס.

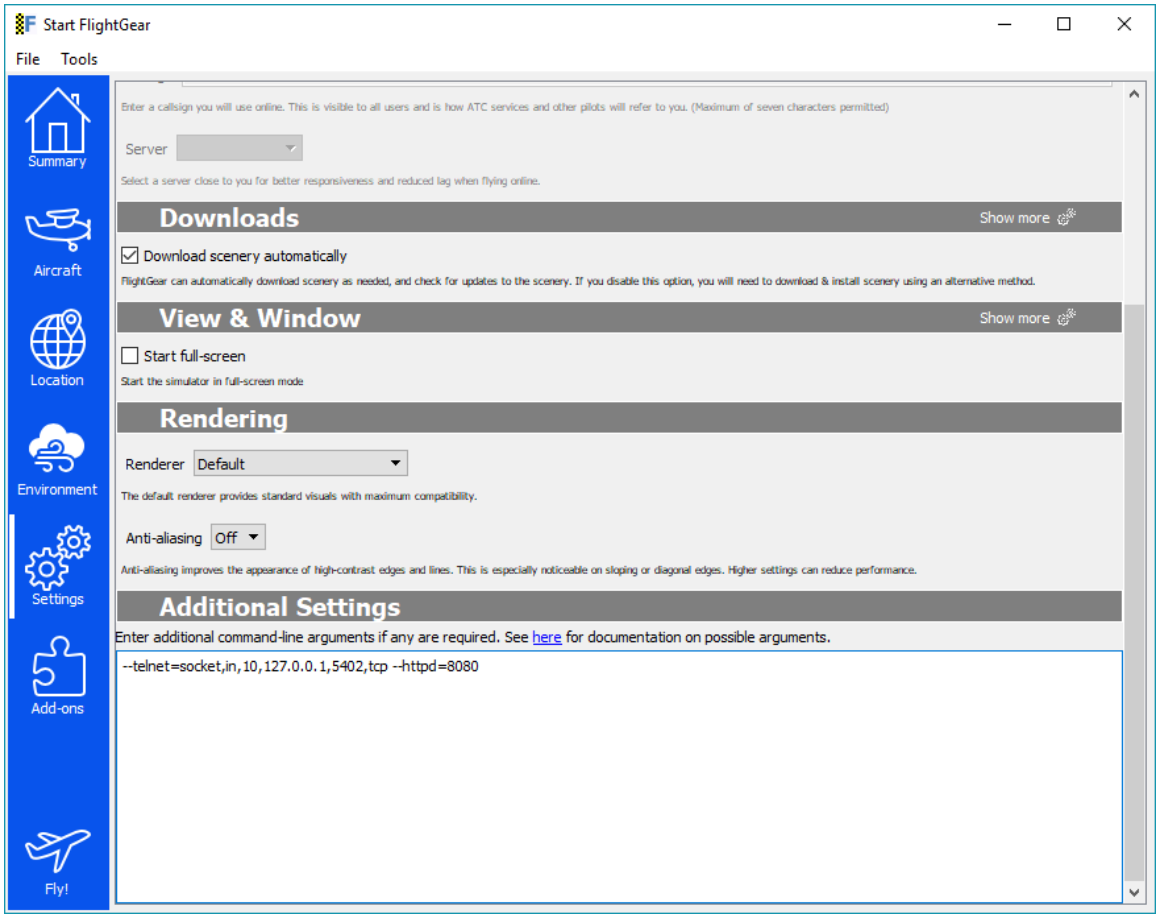
במסך הפתיחה ניתן לגשת ל settings ולהוסיף הגדרות שבד"כ נכתבות ב command line בסעיף של Additional Settings (ראו צילום מסך בהמשך).

למשל ההגדרה --telnet=socket,in,10,127.0.0.1,5402,tcp

אומרת לסימולטור לפתוח ברקע שרת שניתן להתחבר אליו באמצעות כל telnet client. השרת מבוסס על socket, והוא נועד לקרוא מידע שיגיע מהלקוח (in) בקצב של 10 פעמים בשנייה, ב local host כלומר באותו המחשב (ip 127.0.0.1) על port 5402 מעל פרוטוקול tcp/ip.

למשל ההגדרה --httpd=8080 – תפתח web server על פורט 8080.

הריצו את הסימולטור עם ההגדרות לעיל (Fly!).



לאחר שהסימולטור פעל, פיתחו את הדפדפן בכתובת http://localhost:8080 ותוכלו לראות את האפליקציה ה web-ית שבאה עם הסימולטור.

כעת, תפתחו telnet client בשורת הפקודה, על local host ופורט 5402 (בהתאם להגדרות שראינו בסימולטור)

הערה: בלינוקס ניתן ישר לפתוח מהטרמינל, בחלונות יש לוודא שהתקנתם telnet client, מי שלא התקין וצריך עזרה יכול לחפש בגוגל "install telnet client windows 10 command line " ולמצוא מדריכים.

כאמור, ב CMD של חלונות הקלידו telnet 127.0.0.1 5402 ובעצם התחברתם כלקוח לשרת שפתח הסימולטור על המחשב שלכם. הממשק הזה נוח מאד מכיוון שהוא בנוי בצורה של file system (קבצים ותיקיות). כתבו ls כדי לראות את "התיקיות והקבצים" במיקום הנוכחי שלכם. תוכלו להיכנס לתיקיה באמצעות הפקודה cd (שזה change directory) למשל cd controls. תטיילו קצת בין אינספור ההגדרות השונות כדי לקבל תחושה מה יש שם.

כעת הביטו בו זמנית בסימולטור וב telnet. בסימולטור אתם יכולים לשנות זוויות צפייה ע"י לחיצה על V ואף משחק עם העכבר. הביטו על המטוס ממבט אחורי. כעת בואו נזין פקודה להזזת מייצב הכיוון של המטוס (rudder):

לא משנה היכן אתם ב telnet, כתבו:

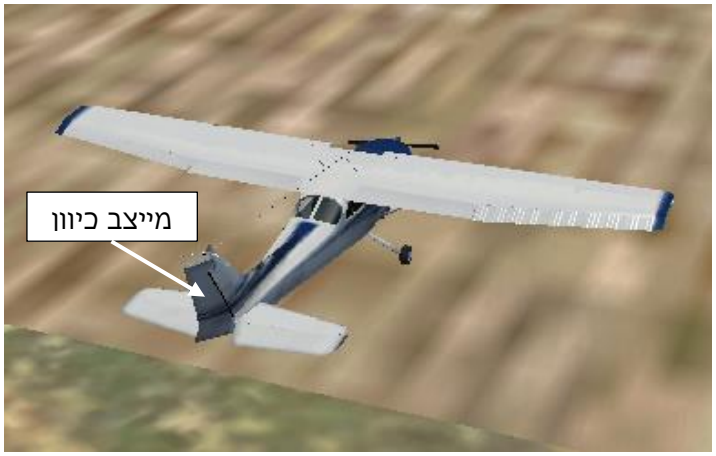
set controls/flight/rudder 1

ראו בסימולטור כיצד מייצב הכיוון זז עד הסוף ימינה.

באופן דומה כתבו

set controls/flight/rudder -1

וראו כיצד מייצב הכיוון של המטוס זז עד הסוף שמאלה. כל ערך בין 1- ל 1 יסובב את מייצב הכיוון בהתאמה.



חוץ מלתת פקודות להגאים של המטוס, ניתן גם לדגום את הערכים השונים שנמדדו ע"י מכשירי הטיסה, כמו כיוון, מהירות, גובה וכו'. כתבו ב telnet את השורה הבאה:

```
get /instrumentation/altimeter/indicated-altitude-ft
```

ניתן לראות בתגובת השרת את הערך של הגובה הנוכחי כפי שנמדד במכשיר טיסה שנקרא `altimeter`.

אבל, את הערכים של הטיסה אנו נדגום בצורה מחוכמת יותר. יחד עם ההפעלה של הסימולטור נוסיף את ההגדרה:

```
--generic=socket,out,10,127.0.0.1,5400,tcp,generic_small
```

משמעותה היא שהפעם הסימולטור יתחבר כלקוח לשרת (שאנו נבנה) באמצעות socket, לצורך פלט (out) בתדירות של 10 פעמים בשנייה, על ה local host בפורט 5400 מעל tcp/ip. הערכים שנדגום מוגדרים בקובץ שנקרא generic_small.xml המצורף כנספח לפרויקט. את הקובץ הזה עליכם לשתול במיקום בו התקנתם את FlightGear בתיקייה data/protocol. פתחו את קובץ ה XML כדי להתרשם אלו נתונים נדגום.

עם הגדרה זו נצטרך לפתוח את השרת שלנו **לפני** שנפתח את הסימולטור כדי שהוא יוכל להתחבר אלינו כלקוח. הסימולטור ישלח 10 פעמים בשנייה את הערכים שדגם מופרדים בפסיק (בדיוק כמו ב CSV) ולפי הסדר שהוגדרו ב XML. בהמשך אבן הדרך תכתבו שרת קטנטן שיאזין לערכים האלו.

מי שרוצה לקבל עוד קצת רקע לגבי שליטה במטוס יוכל לקרוא (בוויקפדיה למשל) אודות

- ההגאים: aileron, elevators, rudder (מייצב כיוון, מייצב גובה, מאזנות בהתאמה)
- וכיצד הם משפיעים בעת טיסה על ה roll, pitch, yaw בהתאמה.
  - ובעברית סבסוב, עלרוד, גלגול בהתאמה.

אין כל חובה לדעת להטיס מטוס בפרויקט זה, אך מעט רקע בהחלט יכול לעזור.

מפרש קוד לשליטה בטיסה

כאמור, ברצוננו לכתוב מפרש לשפת תכנות חדשה שמטרתה להטיס את המטוס שבסימולטור. נתחיל מלהגדיר קוד לדוגמא שמטרתו לגרום למטוס להמריא בצורה ישרה. בהמשך נסביר את המשמעות של שורות אלו וכיצד נכתוב מפרש שיריץ אותן.

קוד לדוגמא:

```
1.  openDataServer 5400 10
2.  connect 127.0.0.1 5402
3.  var breaks = bind "/controls/flight/speedbrake"
4.  var throttle = bind "/controls/engines/engine/throttle"
5.  var heading = bind "/instrumentation/heading-indicator/offset-deg"
6.  var airspeed = bind "/instrumentation/airspeed-indicator/indicated-speed-kt"
7.  var roll = bind "/instrumentation/attitude-indicator/indicated-roll-deg"
8.  var pitch = bind "/instrumentation/attitude-indicator/internal-pitch-deg"
9.  var rudder = bind "/controls/flight/rudder"
10. var aileron = bind "/controls/flight/aileron"
11. var elevator = bind "/controls/flight/elevator"
12. var alt = bind "/instrumentation/altimeter/indicated-altitude-ft"
13. breaks = 0
14. throttle = 1
15. var h0 = heading
16. while alt < 1000 {
17.    rudder = (h0 – heading)/20
18.    aileron = - roll / 70
19.    elevator = pitch / 50
20.    print alt
21.    sleep 250
22. }
23. print "done"
```

הסבר:

נרצה ששורה 1 תגרום לפתיחה של ת'רד ברקע, שפותח **שרת** המאזין על פורט 5400 וקורא שורה שורה בקצב של 10 פעמים בשנייה. את הערכים הנדגמים יש לאכסן במבנה נתונים שבאמצעותו נוכל לשלוף ב O(1) את הערך העדכני של משתנה כלשהו שבחרנו.

נרצה ששורה 2 תתחבר כלקוח לשרת שנמצא ב 127.0.0.1 ומאזין על פורט 5402.

בשורות 3-12 אנו מגדירים את המשתנים שאיתם נעבוד במהלך התוכנית. המשמעות של bind היא כריכה בין ערך המשתנה בתוכנית שלנו לבין מיקומו בסימולטור הטיסה.

כך למשל בשורה 14 כשביצענו השמה throttle = 1 שלחנו למעשה לסימולטור את הפקודה:

```
set /controls/engines/engine/throttle 1
```

וגרמנו למצערת להיפתח עד הסוף (כוח מלא למנוע כדי שהמטוס יתחיל לנוע)

נשים לב שהכריכה היא דו-כיוונית, למשל המשתנה `heading` כרוך למכשיר `heading indicator` שנמצא ב:

```
/instrumentation/heading-indicator/offset-deg
```

בכל פעם בתוכנית שנשתמש ב `heading` נקבל את הערך הנוכחי של הכיוון מסימולטור הטיסה. למשל, בשורה 15 המשתנה `h0` מקבל את ערכו הנוכחי של `heading` - כיוון הטיסה, כפי שנגדם באותו הרגע ע"י ת'רד שרץ ברקע שפתחנו בשורה 1.

בשורה 16 פתחנו לולאת while כל עוד המשתנה alt (הכרוך לגובה הטיסה כפי שנמדד ע"י ה altimeter) קטן מ 1000 רגל. כאמור הערך של alt מתעדכן באופן אוטומטי בזכות הכריכה שהגדרנו ובאמצעות הת'רד שפתחנו בשורה 1.

בתוך הלולאה אנו מעדכנים את הערכים

- של ה rudder כפונקציה של הכיוון
- של ה aileron כפונקציה של הגלגול
- ושל ה elevator כפונקציה של ה pitch

הערה: ערכים אלו ניתנו עבור המטוס הדיפולטיבי בסימולטור – Cessna C172p ובהחלט יכולים להיות שינויים בין גרסאות שונות כדי שהמטוס באמת יתייצב בהמראה.

כמו כן, בכל איטרציה אנו מדפיסים את הערך של alt וממתינים 250 מילישניות לפני המעבר לאיטרציה הבאה. בסוף הרוטינה אנו כותבים done.

אז איך ניגשים למשימה מפלצתית שכזו? ☺

תחילה נכיר תבנית עיצוב חשובה ופשוטה בשם Command Pattern. התבנית אומרת לנו להגדיר ממשק בשם Command עם מתודה doCommand()\. כל פקודה במערכת שלנו (אצלנו זה פקודות שיש לפרש) תהיה מחלקה מהסוג של Command. כך Command פולימורפי יכול להיות פקודה ספציפית כלשהי, ונפעיל את כולן באותו האופן. לצרכים שלנו doCommand יכולה לקבל כפרמטר מערך של מחרוזות שיש לפרש.

הטריק התכנותי שנבצע הוא שנכניס את כל הפקודות למפה מבוססת hash כך שהמפתח הוא מחרוזת, והערך הוא אובייקט ספציפי מסוג Command. כך בהינתן המחרוזת נוכל לשלוף מידית את ה Command שצריך לפעול.

מנגנון העבודה:

צרו פונקציה בשם lexer שתפקידה לקרוא את הסקריפט שצריך לפרש (שורה בודדת מה console או קובץ שלם של פקודות) והיא תחזיר מערך של מחרוזות. כל מחרוזת היא מילה בתוכנית שיש לפרש.

כעת, כתבו פונקציה בשם parser שעוברת (כמעט) על כל מחרוזת במערך שיצר ה lexer. בהינתן מחרוזת, היא תשמש אותנו כמפתח שבאמצעותו נשלוף את אובייקט הפקודה המתאים מהמפה, ונזין לו את המחרוזות שהוא צריך כדי שיפרש את הפקודה ויבצע אותה.

לדוגמא:

המחרוזת הראשונה בתוכנית היא openDataServer. מפתח זה יגרום לשליפה של אובייקט שמימש את הממשק Command. נניח שקוראים לו OpenServerCommand. למתודה doCommand שלו נזין את שארית השורה כמערך של מחרוזות. בתורה, doCommand תודא שגודל המערך הוא 2 (שני פרמטרים בסקריפט) ושהמחרוזות מהוות ערך מספרי תקין. אחרת, נדפיס הודעת שגיאת סינטקס מתאימה. אם הערכים תקינים נפעיל את השרת שלנו ברקע על פרמטרים אלה. כמובן, כדאי שהשרת הזה יוגדר במחלקה אחרת, קראו לה DataReaderServer.

כעת, ה parser יזין את המחרוזת הבאה שיש לפרש (connect) ישלוף אובייקט פקודה מתאים, יריץ אותו, וחוזר חלילה עד לסוף הסקריפט.

<u>הגדרת משתנים:</u>

כפי שניתן לראות המילה var היא הטריגר להגדרת משתנים. הפקודה המתאימה צריכה לתחזק מפה מבוססת hash שבו המפתח הוא שם המשתנה והערך (מסוג double) הוא ערך המשתנה. תקראו למפה הזו symbolTable.

המשמעות של bind היא כפולה, מצד אחד יש לגרום לכך שאם המשתנה נדגם, כלומר מצד ימין של אופרטור ההשמה, אז ערכו יישלף מהמפה שיצר הת'רד שקיבל את הנתונים מהסימולטור (השרת הקטן שלנו). לעומת זאת אם המשתנה נמצא מצד שמאל של אופרטור ההשמה אז עלינו לשלוח לסימולטור כלקוח את הפקודה set למיקום המתאים עם הערך שנמצא מימין לאופרטור ההשמה. כפי שאתם מבינים אופרטור ההשמה "=" גם הוא אובייקט מסוג Command.

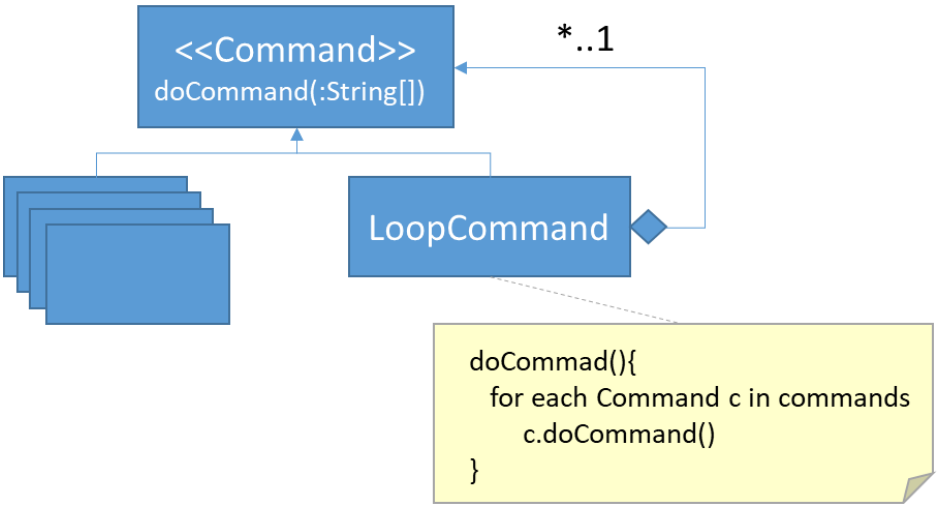
תנאים:

בשורה 16 המפתח הוא while לאחר מכן נצפה לסדרה של תנאים עד להופעת הסימן "}". לשם הקלה עליכם לצפות רק לתנאי אחד. אבל מי שרוצה להשקיע שיכניס גם סוגרים עגולים ופעולות AND ו OR.

התנאי יכול להיות מורכב מהאופרטורים <,>,<=,>=, ==,=! במשמעות הרגיל שאתם כבר מכירים.

לולאה:

היופי בתבנית העיצוב של Command זה שברגע שכל פקודה מוגדרת במחלקה, אין לנו בעיה ליצור פקודה שמורכבת מכמה פקודות בסיסיות יותר:



למשל, המחלקה LoopCommand יכולה להחזיק מערך דינאמי (ופולימורפי) של אובייקטי Command. כל אחד מהם יכול להיות אובייקט ספציפי של פקודה כלשהי או אפילו עוד אובייקט מהסוג של LoopCommand, כלומר עוד לולאה פנימית בסקריפט. המתודה doCommand של LoopCommand פשוט תפעיל את כל אובייקטי ה Command שנזין לה כל עוד התנאי מתקיים. נזין את כל אובייקטי ה Command שחזרו מהשורות ש"פירסרנו" עד להופעה של "}". בדוגמא לעיל מדובר בשורות 17 עד 21.

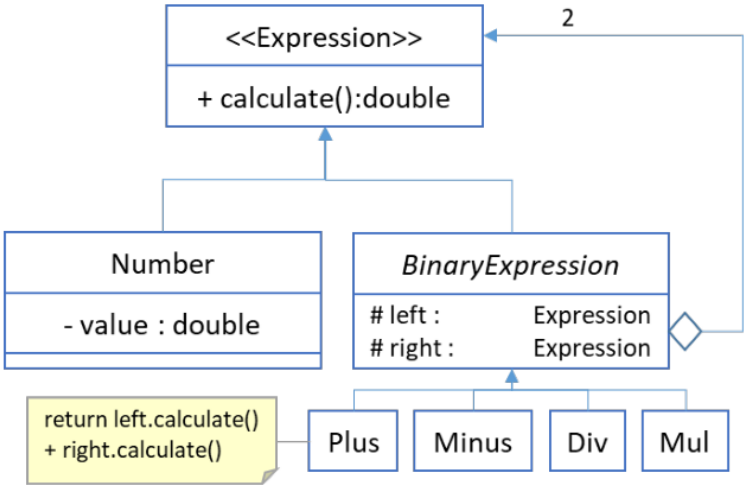
ביטויים:

ומה לגבי פיענוח ביטויים מתמטיים מורכבים כמו בשורות 17-19?

נשים לב שכל ביטוי יכול להיות ערך (קבוע או משתנה) או אופרטור אונרי הפועל על ביטוי, או אופרטור בינארי הפועל על שני ביטויים. כמובן שבתורם ביטויים אלה יכולים להיות שוב ערכים או אופרטורים... כלומר, מתקבל עץ של ביטויים בו כל קודקוד הוא או עלה (ערך בודד), או אופרטור עם בנים שהם קודקודים (עלים או אופרטורים). שוב מדובר בפולימורפיזם, יש לנו כמה סוגים של קודקודים, או ביטויים, כאשר הילדים של אופרטור כלשהו גם הם בעצמם ביטויים. כדי להתמודד עם זה אנו זקוקים לתבנית עיצוב בשם, איך לא, Interpreter.

הנה דוגמא:

ממשק בשם Expression מגדיר מתודה בשם Calculate המחזירה double. אובייקט מסוג Expression יכול להיות Number או BinaryExpression שלו יש שני משתנים בדיוק מהסוג של Expression: בשמות left, right. כמו כן ירשו אותו פעולות החישוב הבסיסיות כמו פלוס, מינוס, חילוק וכפל. לדוגמא, Plus יחזיר את left.calculate() + right.calculate() יהיה הביטוי שלהם עמוק כאשר יהיה...

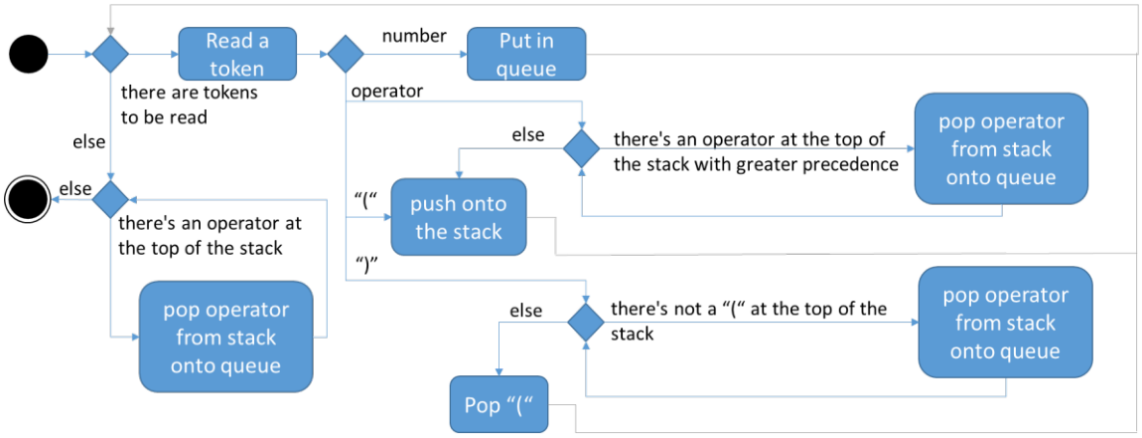


דוגמא לחישוב:

```
// 3+(4/2)*5
Expression* e=new Plus(new Number(3) , new Mul( new Div(new Number(4), new
Number(2)) , new Number(5)));
e->calculate();
```

איך מגיעים מביטוי בצורה של infix כמו 3+4 ליצירה של אובייקטים כמו Plus(Number(3),Number(4))?

לשם כך עליכם לממש אלגוריתם חביב בשם Shunting-yard של דייקסטרה. הנה Activity Diagram שמתארת את פעולתו:



בהינתן ביטוי infix , האלגוריתם מסדר את המספרים בתור, ומשתמש במחסנית כדי להכניס את האופרטורים לתור זה בסדר שמציג את הביטוי כ postfix. למשל עבור הדוגמא לעיל בסוף האלג' התור יראה כך: 342/5*+. כשנקרא את התור הפוך (כלומר, מימין לשמאל) נבין שעלינו לבצע חיבור של (הכפלה של 5 עם (חלוקה של 4 ב 2) עם 3. לפיכך, נוכל לייצר בהתאם לביטויים את המופעים של Number, Plus, Minus, Mul, Div ולחשב את תוצאות הביטוי.

:Main

למעשה פונקציית ה `main` צריכה לקרוא מהמשתמש שורה שורה, ועל כל שורה לשלוח ל `lexer` ואת הפלט שלו לשלוח ל `parser`. כך הקוד שהמשתמש כתב מתפרש וגורם לפעולות שונות בסימולטור.

אופציה שנייה, היא לפתוח קובץ טקסט ובתוכו הסקריפט המלא שאותו נפרש שורה שורה בדיוק באותו האופן.

למשקיעים: ה `main` תמיד תצפה לקלט של המשתמש, אך אחת הפקודות תהיה להריץ סקריפט שכתוב בקובץ מסוים. למשל:

```
> run “d:/scripts/takeoff.fgs”
```

בהצלחה!

אבן דרך 2 חלק א' – שמירה על עקרונות Solid

בכיתה למדנו אודות עקרונות עיצוב שונים, ובפרט על העקרונות single responsibility ו open/close של SOLID. בבואנו לכתוב את צד השרת נרצה לקיים את העקרונות האלה. מכיוון שזו הפעם הראשונה, נעשה זאת יחד באופן מודרך.

ברצוננו לכתוב שרת כללי, כלומר שרת שיוכל לשמש אותנו שוב ושוב בפרויקטים שונים. לשם כך, עליו לבצע הפרדה עקרונית וחשובה, מהי לדעתכם?

א. הפרדה בין צד השרת ללקוח
ב. הפרדה בין מה שמשתנה בין פרויקט לפרויקט, לבין מה שלא.
ג. הפרדה בין פרוטוקולים שונים לתקשורת

נסו לחשוב לבד – איזו הפרדה עקרונית אנו צריכים?

תשובה:

אל הסכנה הוא לקחת מהמשתמש יותר. נרצה לבנות מנגנון שיאפשר לו לבחור מה הוא רוצה לבצע – מהם מודולים לעשות בו שימוש. לכל זו הסכנה מעניקה לבנות מנגנון לבחירת פעולות, כגון לבחירת מודולים. כגון כל, דוגמא מעניינת לכך – מהם מודולים לעשות בו שימוש. כגון כל, דוגמא מעניינת לכך – הוא נרצה מהם מודולים לעשות בו שימוש. נרצה לבנות מנגנון לבחירת פעולות, כגון לבחירת מודולים. הוא הסכנה הוא.

את המנגנונים שאינם משתנים נוכל לממש כבר כעת. אולם, פעולות שעשויות להשתנות מפרויקט לפרויקט לא נרצה לממש, אלא נרצה רק לדעת להפעיל אותן בבוא העת כשנצטרך. לשם כך נוכל לנצל את הקיום של ממשקים (interfaces).

חישבו כיצד?

תשובה:

בוודאי הבנתם "הפרדה מהמימוש" דבר הכרחי לא מהמרד. הרעיון הבא הוא לממש lexer, לדוגמה, בעזרת מחלקה מהמרד; הרעיון הבא הוא לממש parser, לדוגמה, בעזרת מחלקה מהמרד; הרעיון הבא הוא לממש lexer() s.lexer() בעזרת מחלקה מהמרד. הרעיון הבא הוא לממש parser() s.parser() בעזרת מחלקה מהמרד. הרעיון הבא הוא לממש lexer() s.lexer() בעזרת מחלקה מהמרד.

אז בואו נתחיל.

אילו הינו מתחילים עם מחלקה שמממשת שרת, דהיינו מנגנון שמאזין וממתין ללקוחות שיתחברו ואז מטפל בבקשות שלהם, אז הינו כותבים מנגנון אחד שלא בהכרח היה מתאים לכל פרויקט. יותר נכון זה **להגדיר את הפונקציונאליות של השרת באמצעות ממשק**, ובכל פרויקט יכולה להיות מחלקה אחרת שתממש את אותה הפונקציונאליות בדרך שונה. לדוגמא, הסמסטר נממש מחלקה שתטפל בלקוחות אחד אחרי השני ואילו בפת"מ 2 תוכלו להוסיף מחלקה המהווה שרת שמטפל בכל הלקוחות במקביל.

נשים לב שמתקיימת כאן שמירה על עקרון open/ close שכן, הוספה של מחלקה שמימשה את אותו הממשק היא הרחבה של הפונקציונאליות הנדרשת (open) מבלי שהיינו צריכים לשנות קוד שכבר כתבנו (close) – פתוח להרחבה אך סגור לשינויים.

צרו פרויקט ובו namespace בשם server_side ובתוכו את הממשק Server. נתחיל מפונקציונאליות פשוטה:

- המתודה open תקבל פרמטר int port להאזנה ותפקידה יהיה לפתוח את השרת ולהמתין ללקוחות.
- המתודה stop תסגור את השרת.

חישבו האם צריך מתודות נוספות.

כעת, צרו את המחלקה MySerialServer מהסוג של Server. את הקוד תממשו בהמשך; כעת זו רק התשתית.

שיחה עם הלקוח

תארו לכם מצב שבו במחלקה MySerialServer גם מימשנו את פרוטוקול השיחה בין הלקוח לשרת. הרי בכל פרויקט תיתכן שיחה שונה בפורמט שונה ועם ציפיות שונות בין הלקוח לשרת. כך לא נוכל להשתמש במחלקה זו בפרויקטים אחרים. בין מה למה עלינו להפריד כדי לשמור על עקרונות single responsibility ו open close?

א. עלינו להפריד בין מחלקות שונות שמימשו את Server, בכל מחלקה נממש פרוטוקול שונה
ב. עלינו להפריד בין המנגנון של השרת שמומש ב MySerialServer לבין צורות שיחה שונות עם הלקוח.

א' היא כמובן תשובה לא נכונה, מפני שגם עם אותו המנגנון נרצה לקיים שיחות שונות. למשל, תארו מצב שבו יש לנו שני מנגנונים של שרת – MySerialServer מטפל בלקוחות אחד אחרי השני ואילו MyParallelServer מטפל בהם במקביל. כמו כן, יש לנו שני פרוטוקולים של תקשורת – באחד הלקוח שולח מחרוזת לשרת והשרת מחזיר מחרוזת הפוכה, ואילו בפרוטוקול השני הלקוח שולח משוואות מתמטית והשרת מחזיר פתרון. לפי תשובה א' נצטרך 4 מחלקות (!) שרת טורי שהופך מחרוזות, שרת טורי שפותר משוואות, שרת מקבילי שהופך מחרוזות ושרת מקבילי שפותר משוואות. לא הגיוני.

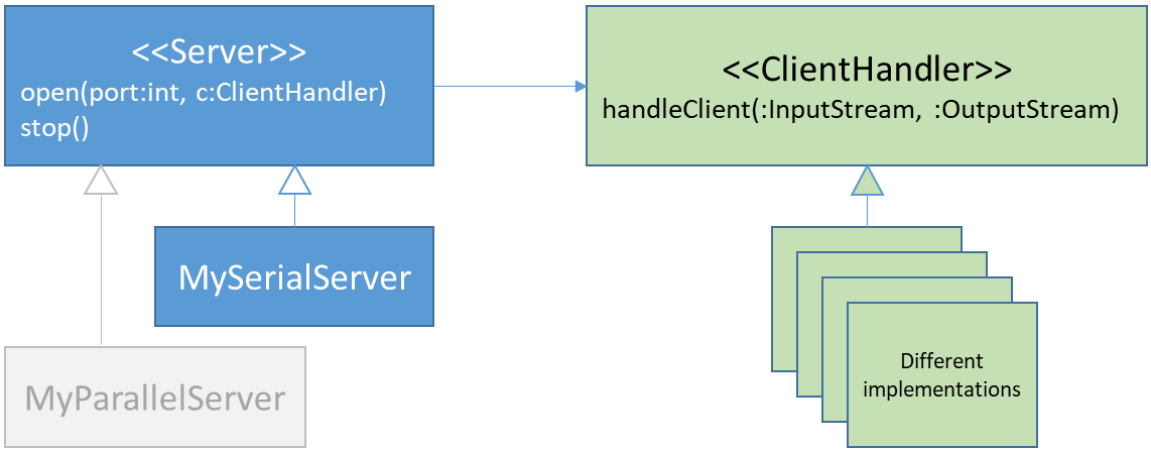
לפי ב', לעומת זאת, אם ניצור ממשק בשם ClientHandler שמטרתו לקבוע את סוג השיחה עם הלקוח והטיפול בה, אז נוכל להסתפק רק בשתי מחלקות עבור מנגנונים של שרת (MySerialServer, MyParallelServer) ולכל אחת מהן נוכל להזריק איזה מימוש שנרצה עבור ClientHandler. בפרט, לכל מימשו של Server נוכל להזריק שיחה של היפוך מחרוזות או פתרון משוואות. באותו האופן אם מחר נרצה לממש פרוטוקולים נוספים אז נצטרך רק להוסיף מימוש של ClientHandler מבלי לשנות או להעתיק שוב את הקוד של המנגנונים השונים לשרת.

נשים לב שבשיטה זו שמרנו גם על single responsibility וגם על open close.

הגדירו את הממשק ClientHandler עם המתודה handleClient שמקבלת stream של קלט (ממנו נקרא את הודעות הלקוח) ו stream של פלט (שאליו נכתוב את תשובת השרת).

היכן נטמיע את ההזרקה של ClientHandler?

הביטו בתרשים הבא ונסו לענות מדוע נכון יותר שזה יהיה כפרמטר של מתודה ב Server ולא data member באחת המחלקות שירשו אותו?



על איזה עיקרון של SOLID שומר הפרמטר c במתודה open של Server?

תשובה: Dependency Inversion – המחלקות שירשו את Server מקבלות תלות מסוג ClientHandler. הן מחליטות מתי להפעיל את ה ClientHandler ואילו הוא מחליט על המימוש.

כעת, ממשו את המחלקה MyTestClientHandler כסוג של ClientHandler שתשמש אותנו בהמשך לבדיקת התשתית.

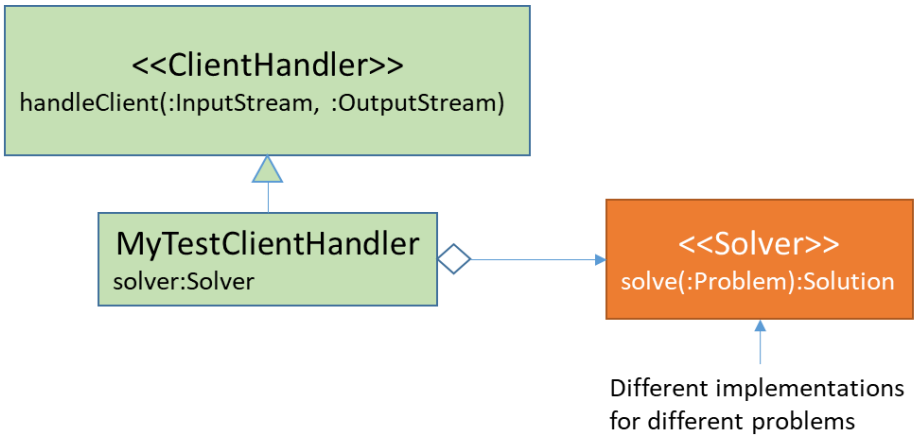
פתרון בעיות

בהמשך לדרך בה אנו שומרים על single responsibility ו open close נרצה להפריד גם בין פרוטוקול השיחה בין השרת ללקוח (שמומש אצל יורש כלשהו של ClientHandler) לבין האלגוריתם שפותר את הבעיה. אחרת, נצטרך מחלקה אחת עבור כל שילוב אפשרי של פיענוח פרוטוקול שיחה עם כל אלגוריתם...

לדוגמא, נניח שקיימים שני אלגוריתמים A ו B לפתרון של משוואות מתמטיות. ללא הפרדה, נצטרך שני מימושים של ClientHandler עבור אותו פרוטוקול תקשורת. בשני המימושים יהיה **קוד כפול** שקורא למשל מחרוזת שהגיע מהלקוח ושולף מתוכה את הנתונים. אך מימוש אחד מבצע את אלגוריתם A ואילו השני את B. מיותר.

הביטו בתרשים הבא וענו על השאלות הבאות

א. הסבירו במילים שלכם כיצד ומדוע העיצוב בתרשים הבא נמנע מהקוד הכפול המתואר לעיל
ב. מדוע בחרנו לבצע את ההכלה ב MyTestClientHandler ולא בממשק ClientHandler עצמו?



רמז:

- האם משהו ישנה בקוד של MyTestClientHandler אם נרצה להשתמש באלגוריתם אחר כדי לפתור את אותה הבעיה?
- האם בהכרח כל ClientHandler פותר בעיות אלגוריתמיות?

שמירה של פתרונות (Caching)

ייתכן וחישוב הפתרון לוקח המון זמן. יהיה זה מיותר לחשב פתרון עבור בעיה שכבר פתרנו בעבר. במקום זאת, נוכל לשמור פתרונות שחישבנו בדיסק. במידה ומגיעה בעיה, נצטרך לבדוק במהירות האם כבר פתרנו אותה בעבר, אם כן, נשלוף את הפתרון מהדיסק במקום לחשב אותו. אחרת נפתור את הבעיה ונשמור את הפתרון בדיסק.

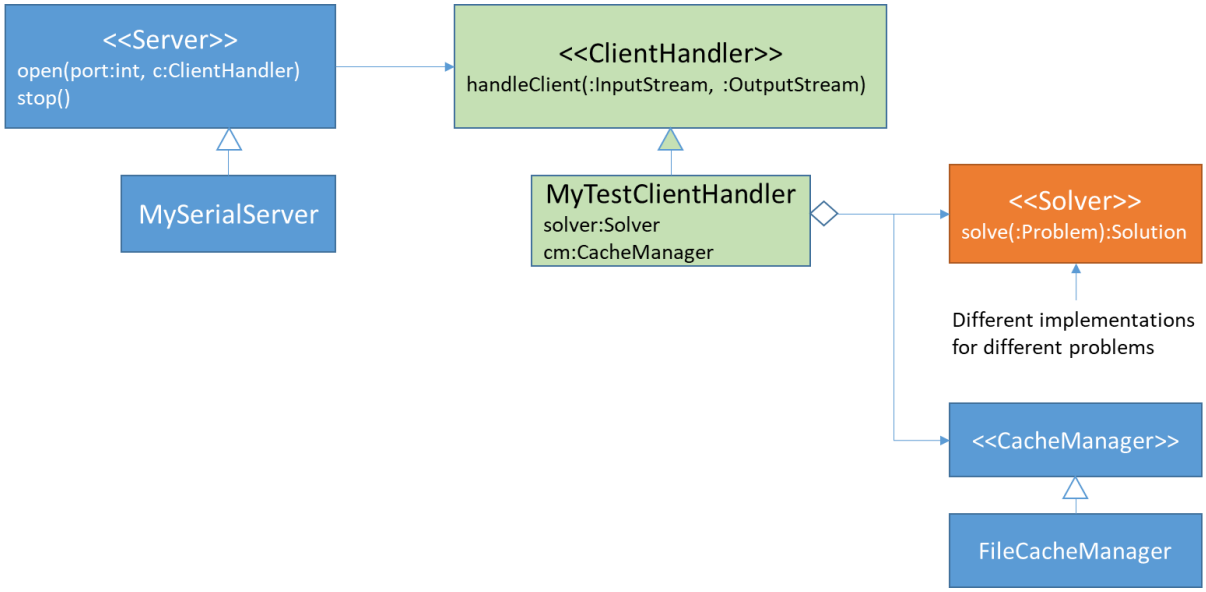
בשלב זה, אתם כבר מבינים שיתכנו מספר מימושים שונים לשמירת הפתרונות, לדוגמא בקבצים או במסד נתונים. לכן, ניישם שוב את אותה הטקטיקה של שימוש בממשק כדי לשמור על העקרונות השונים של SOLID. נגדיר את הממשק CacheManager שינהל עבורנו את ה cache (בעברית - מטמון).

חישבו, מהי הפונקציונאליות של CacheManager? אלו מתודות תגדירו לו?

- מתודה אחת שמחזירה true אם הפתרון נשמר בהצלחה
- מתודה אחת שמחזירה true אם הפתרון נמצא
- שאם מתודה אחת מחזירה true אז מתודה אחת מחזירה:

תגדירו את המתודות לממשק CacheManager. ממשו את הממשק במחלקה שנקראת FileCacheManager. מחלקה זו באבן הדרך הבאה תשמור פתרונות בתוך קבצים בדיסק.

בהצלחה.



אבן דרך 2 חלק ב' – streaming

רקע:

החל מחלק זה, בהדרגה מסמך זה יגדיר יותר **מה** לעשות (דרישות) ופחות **איך** לעשות זאת. הבחירה עוברת לידיים שלכם.

בהמשך לשיעורים אודות Streaming של מידע לקבצים ולערוצי תקשורת עליכם לממש את המחלקות השונות ע"פ העיצוב שלנו לצד השרת מחלק א'. אולם, הבעיה שנפתור תהיה לעת עתה בעיית צעצוע רק כדי שנוכל לבדוק את התשתית שיצרנו. בהינתן מחרוזת מהלקוח, השרת יצטרך להחזיר מחרוזת הפוכה. בהמשך, נעדכן את השרת שלנו כך שיפתור בעיות חיפוש.

MySerialServer

בהתאם לקוד שהדגמנו בשיעור, במתודה start ממשו לולאה שמאזינה ללקוח על ה port שקבלנו. לאחר שהלקוח מתחבר היא תפעיל את ה ClientHandler שהוזן לה כדי לשוחח עם הלקוח. בתום השיחה נחזור לתחילת הלולאה ונאזין ללקוח הבא. כך נטפל בלקוחות אחד אחרי השני באופן טורי.

דגשים:

- שימו לב שיש להריץ את הלולאה הזו בת'רד נפרד כפי שהודגם בשיעור, אחרת מי שמפעיל את start מפעיל לולאה שלעולם לא תסתיים ולכן לא יוכל להגיע לשורה הבאה ולקרוא ל stop.
- הקפידו להגדיר time out להמתנה ללקוח, אחרת גם הפעלה של stop לא תועיל מפני שנחכה לנצח ללקוח שלעולם לא יגיע, והשרת יישאר פתוח לעד.

MyTestClientHandler

מחלקה זו תקרא מה InputStream של הלקוח שהתחבר שורה אחר שורה. בהינתן שורה, נשאל את ה CahceManager שהוזן לנו האם יש לו פתרון שמור למחרוזת זו. אם כן, נשלוף באמצעותו את הפתרון ונכתוב אותו חזרה ללקוח. אם לא, נפעיל את ה Solver שניתן לנו על מנת לקבל את הפתרון (מחרוזת הפוכה לזו שהלקוח שלח). נבקש מה CacheManager לשמור את הפתרון ונשלח אותו חזרה ללקוח.

תהליך זה יחזור על עצמו עם כל שורה המגיעה מהלקוח.

שימו לב שהלקוח הוא זה שכותב ראשון, ושלאחר כל שורה הוא מצפה לקרוא תגובה לפני שהוא שולח את השורה הבאה שלו.

טיפ: flush

פרוטוקול התקשורת מסתיים כאשר הלקוח כותב את המילה "end" השרת לא יחזיר תשובה ויסיים את ההתקשרות עם אותו הלקוח.

Solver

שנו את הממשק כך שבמקום ש Problem ו Solution יהיו מחלקות, הם יהיו טיפוסים פרמטריים של Solver. כלומר Solver<Problem, Solution>. כך של מחלקה שתממש את Solver תוכל לבחור מהו הטיפוס המייצג את הבעיה ומהו הטיפוס שמייצג את הפתרון. מחקו את המחלקות המיותרות.

StringReverser

צרו את המחלקה StringReverser כמימוש של Solver המקבל מחרוזת ומחזיר מחרוזת הפוכה. תוכלו כמובן להשתמש במופע של StringBuilder לשם כך.

FileCacheManager

אתם מחליטים כיצד הכי נכון לממש אותו. בהינתן מחרוזת הוא צריך לדעת ב O(1) האם שמור לו פתרון. כמובן עליו לתמוך בשמירה ושליפה של פתרונות מהדיסק.

Main

בצד השרת צרו namespace בשם boot ובתוכו מחלקה בשם Main עם מתודת main. ה main תפעיל את השרת MySerialServer כ Server

- על port לפי האורגומנט הראשון של פונקציית ה main (args[0])
- עם StringReverser כ Solver
- עם FileCacheManager כ CacheManager

בהצלחה!

אבן דרך 2 חלק ג' – מימוש אלגוריתם

בהמשך לשיעור מפסאודו קוד של אלגוריתם ל OOP, עליכם להשתמש בתבנית העיצוב Bridge כדי להפריד בין האלגוריתם לבין הבעיה שאותה הוא פותר. בפרט, השתמשו בממשק Searchable כדי להגדיר מהי הפונקציונאליות של בעיית חיפוש, ובממשק Searcher עבור אלגוריתם חיפוש.

- השלימו את האלגוריתם BestFirstSearch
- באופן דומה ממשו צרו את האלגוריתמים הבאים:
  - DFS
  - BFS (breadth first search)
  - Hill Climbing
  - A* (A star)

כעת ערכו ניסוי אמפירי שבודק מי מהם עובד הכי טוב:

1. תגדירו אוסף של 10 מטריצות בגודל NxN עבור N הולך וגדל החל מ N=10 ועד N=50. בכל תא תגדירו איזשהו ערך שלם המבטא את המחיר לדרוך בתא זה. למשל 0 זה חינם או מישור, ככל הערך גדול יותר אז זה כמו עליה קשה יותר, ואינסוף זה קיר שלא ניתן לעבור דרכו. הכניסה לשטח תוגדר בתא 0,0 ואילו היציאה בתא N-1,N-1.
2. תריצו כל אחד מהאלגוריתמים 10 פעמים על כל אחת מ 10 המטריצות (בסך הכל 5 אלג' X 10 ריצות X 10 מטריצות) ובדקו כמה קודקודים פיתח כל אלגוריתם והאם הוא הגיע לפתרון – כלומר המסלול הזול ביותר
3. הציגו את הנתונים בגרף שבו ציר ה X הוא ערכי ה N וציר ה Y הוא מס' הקודקודים שפיתח כל אלגוריתם בממוצע על פני 10 הריצות.

לפי הקווים שנוצרו בגרף תוכלו לראות איזה מימוש הוא היעיל ביותר. זה המימוש שתבחרו להטמיע בצד השרת. דגש: כל המימושים קיימים בצד השרת, אולם ב main נחבר רק את הטוב ביותר.

הטמעת האלגוריתם בצד השרת

האלגוריתם שלנו מימש את הממשק Searcher ואילו ה ClientHandler שלנו מצפה לאובייקט מסוג Solver. אילו הממשק Searcher היה יורש את Solver הינו פותרים באופן טכני את הבעיה הזו אך הפתרון היה ממש לא נכון, שכן, כל המחלקות מסוג Searchable צריכות להיות כעת גם Solver בסתירה לעקרון ה Interface segregation. אילו הינו רוצים לקחת את כל ה searchers שלנו לפרויקט אחר שבו הם לא היו צריכים להיות גם solvers אז הינו מוצאים את עצמנו בבעיה.

מצד שני, אילו מחלקה מסוימת כמו למשל BFS היתה מממשת את Solver בנוסף ל Searcher אז יצרנו סתירה לעיקרון של Open/Close כי נאלצנו לשנות קוד קיים.

לכן, השתמשו ב Object Adapter כדי לפתור את הבעיה וראו כיצד הוא גורם לשמירה על כל עקרונות SOLID שלמדנו.

MyClientHandler

צרו את המחלקה MyClientHandler כך שתתאים לפרוטוקול התקשורת הבא:

- הלקוח שולח שורה אחר שורה עד שמתקבלת שורה עם הערך "end"
- כל שורה מחילה ערכים מספריים המופרדים ע"י פסיק. כך אוסף השורות יוצר מטריצה של ערכים.
- לאחר מכן הלקוח שולח שתי שורות נוספות. בכל שורה שני ערכים המופרדים ע"י פסיק: שורה ועמודה. Row,Col
    - הערכים בשורה הראשונה מציינים את הכניסה לשטח
    - הערכים בשורה השנייה מציינים את היציאה מהשטח
- כעת השרת יחזיר מחרוזת אחת בלבד, עם ערכים המופרדים בפסיק. הערכים יהיו מסוג המילים {Up, Down, Left, Right} שמציינות את הכיוון שיש לנוע לפיו על מנת לחצות את השטח במסלול הזול ביותר.

כעת ממשו את MyParallelServer כך שיטפל בכל הלקוחות במקביל.

בסמסטר הבא, נשתמש בשרת הזה כדי לחשב למטוס מסלול טיסה אידיאלי. נכתוב אפליקציות שונות שמטיסות את המטוס, ומתחברות כלקוח לשרת זה.

בהצלחה!