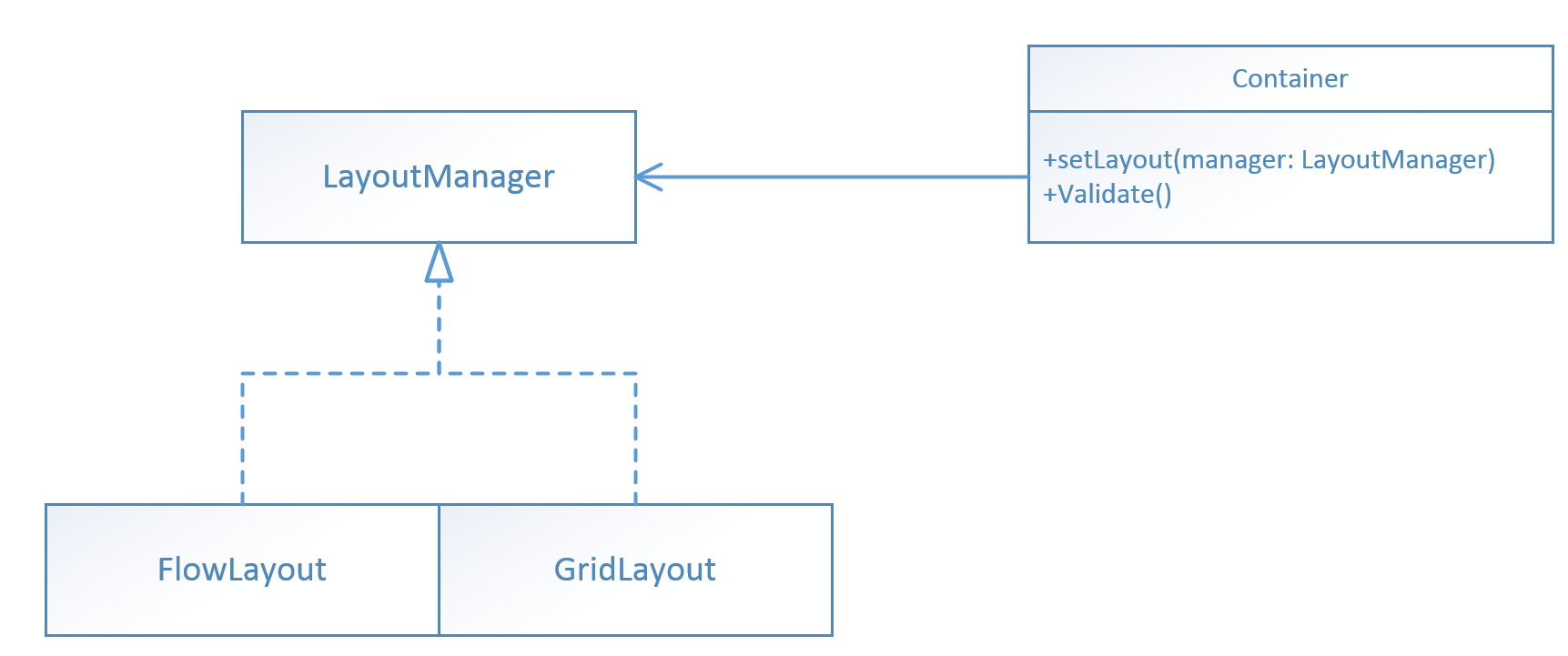
מגישים:

אופיר קרנגל 302621305

רומן אברמזון 306359001

# שאלה 1

1. ה design pattern שמומש כאן הוא מסוג strategy. סוג זה הוא behavioral design pattern שמגדיר משפחה של אלגוריתמים, מבצע אנקפלוציה ובכך מאפשר החלפה קלה בזמן הריצה. במקרה שלנו, ישנן שתי אופציות לסידור ב containers : FlowLayout או GridLayout (כאשר שניהם מממשות את הממשק LayoutManager). כאשר נוצר Container חדש בזמן ריצה, הסידור שלו מקבע לפי הפרמטר המוכנס ב setLayout (הפרמטר מסוג LayoutManager).
2. 

כפי שהוסבר בסעיף א, כאשר נוצר container חדש, נקבע סוג הסידור שלו במתודה setLayout כאשר סידור יכול להיות או FlowLayout או GridLayout.

# שאלה 2

במימוש הצאט השתמשנו במחלקות הבאות:

* Chat – בונה את הטופס (frame) שבו מתנהלים הצאטים של ארבעת הסטודנטים. מכיל ארבעה פאנלים. כל פאנל מכיל מקום להכנסת טקסט חדש ואת תוכן הצאט הקיים.
* Observable – מממש instance של משהו שניתן "לעקוב אחריו". במחלקת chat קיימים ארבעה מופעים עבור כל תיבת טקסט של הכנסת טקסט חדש.
* Observer – מממש ממשק של "עוקב". מכיל מתודה בשם update שמופעלת כאשר observable מפעיל את notifyObservers.
* ChatUpdater – מממש את הממשק Observer. המתודה update מעדכנת את הטקסט של הסטודנט הרלוונטי. במחלקת chat ישנם ארבעה מופעים של ChatUpdaters.
* ChatFont – מממש ממשק של גופן הטקסט. מכיל מתודה בשם setFont שמחזירה גופן כלשהו.
* ChatRegularFont, ChatBoldFont, ChatItalicFont – מממשות את הממשק ChatFont. כל אחת מהן מחזירה גופן בסגנון Plain, Bold, Italic בהתאמה.

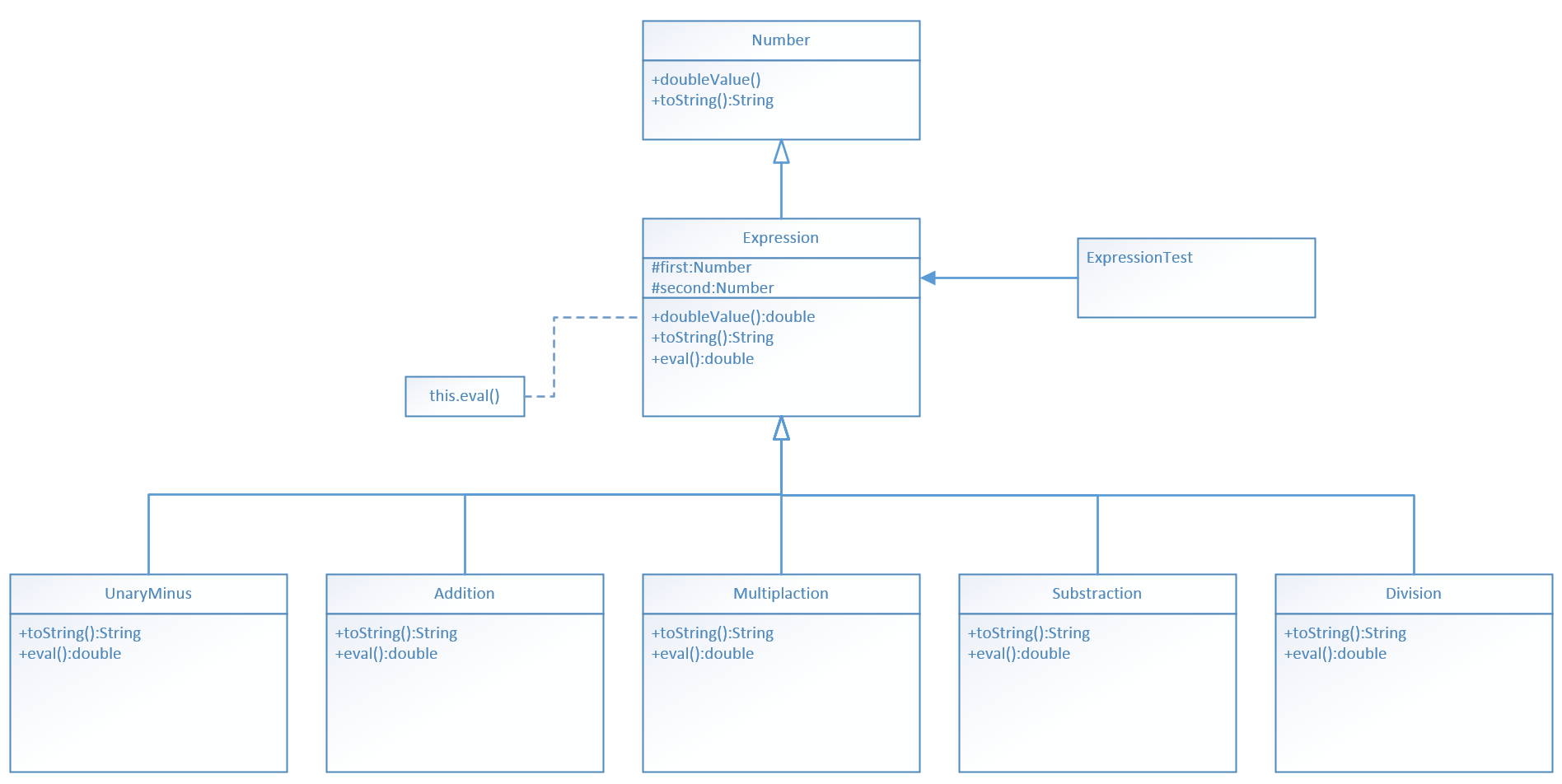
מהלך הצאט:

ברגע שסטודנט מעדכן משהו בתיבת הכנסת הטקסט ולוחץ enter, ה observable המתאים מעדכן שמצבו השתנה ומדווח לכל ChatUpdater (ע"י המתודה notifyObservers). המתודה מעבירה כפרמטר את המחזורת עם שם הסטודנט והטקסט שהוכנס. כתגובה לכן, כל ChatUpdater מעדכן את תוכן הצאט שלו ומכניס את הטקסט החדש. ה design pattern שמומש הוא observer.

בתחתית הצאט ישנה רשימה לבחירת גופן. ברגע שגופן משתנה, הצאט מקבל את האינדקס ברשימה, יוצר מופע מתאים של ChatFont (ChatRegularFont, ChatBoldFont, ChatItalicFont) ואז מעדכן את הצאט של הסטודנט הרלוונטי. ה design pattern שמומש הוא strategy.

# שאלה 3

הקוד מומש באופן הבא:



ה design pattern שמומש כאן הוא adapter מסוג מחלקה.

בשאלה נתבקשו לממש חישוב ביטויים אריתמטים. הביטויים הנ"ל הם חיבור, חיסור, כפל, חילוק ומינוס לפני מספר. חלק מהביטויים מקבלים איבר אחד וחלק שניים. כל אחד מהם יכול לקבל או מספר או ביטוי.

לכן, כדי לא להסתבך במימוש constructors כפולים, השתמשנו במתאם ל Number:

* המחלקה הראשית Expression היא מחלקה המבטאת ביטוי אריתמטי כלשהו. היא מכילה שתי מתודות אבסטרקטיות שאותן כל ביטוי שירש מ Expression יאלץ לממש:
  + eval() המחשבת את ערך הביטוי.
  + toString() המחזירה מחרוזת של הביטוי.

כדי שהמחלקה תוכל גם לקבל מספר מסוג Number וגם ביטוי, קבענו ש Expression יורשת ממחלקת Number וב constructor איפשרנו לקבל פרמטר אחד או שניים מסוג Number. כחלק מהגדרת Number נדרשנו לממש את doubleValue() ולמעשה קבענו שהערך שיוחזר הוא אותו ערך של eval() (זוהי למעשה מתודת התיאום).

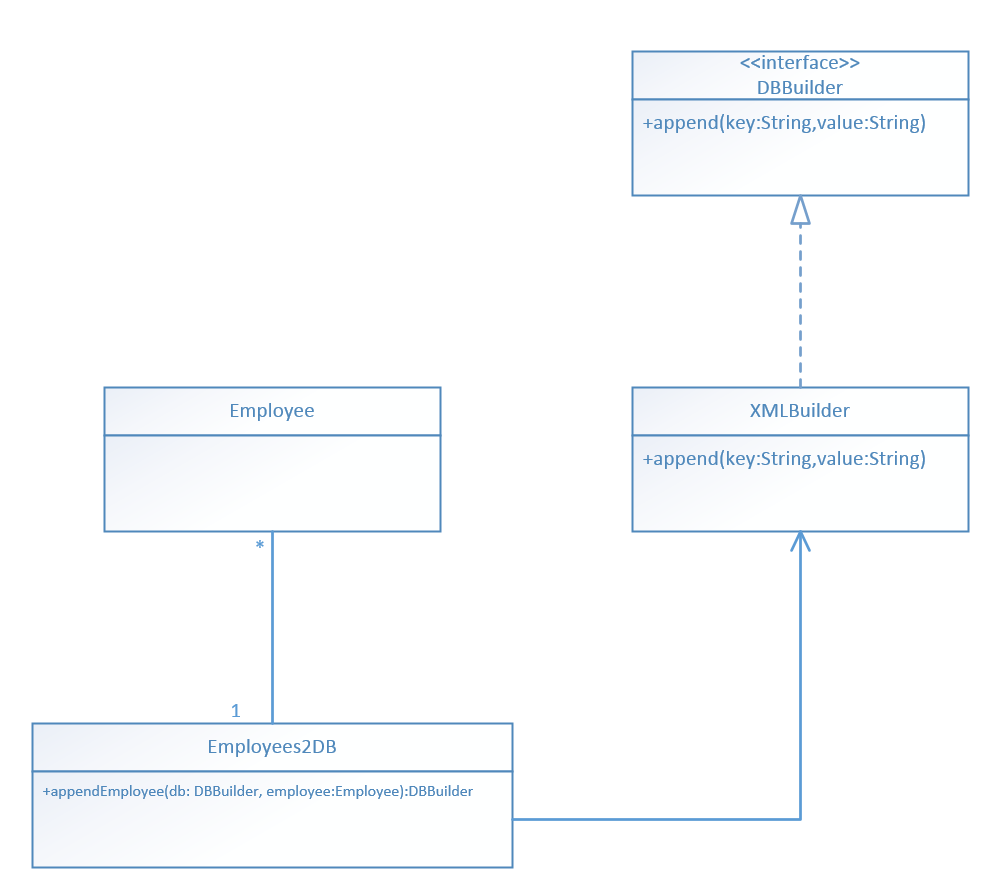
* כל אחת מהביטויים האריתמטים (חיבור,חיסור, כפל, חילוק ומינוס לפני מספר) מימשה את המתודות eval(), toString() בהתאם לביטוי הרצוי. ה constructor של כל אחד מהן קיבל מספר אחד מסוג Number או משני סוגים (תלוי באופי הביטוי). אם ביטוי מסויים קיבל מספר מסוג Number אז doubleValue() היה פשוט המספר עצמו ו toString היה הטקסט של המספר. אם הביטוי קיבל מספר מסוג ביטוי כלשהו, doubleValue() היה הערך של הביטוי (כי הוא קושר ל eval()) ו toString() דרס את המתודה הבסיסית והחזיר מחרוזת שמתארת את הביטוי.

הערך הסופי שהביטוי מחזיר מורכב משרשרת חישובי הביטויים והמחרוזת הסופית מורכבת משרשרת תווים שמייצג כל ביטוי באופן רקורסיבי.

# שאלה 4

1. עיקרון SOLID שמופר הוא Single Responsibility principle. המחלקה Employee מייצגת עובד בלבד. הוספת יכולות toXML() ו toDB() מוסיף אחריות שלא קשורה לעובד. באופן זה היכולות של העברה לבסיס נתונים של XML ו DB מוצמדים שלא לצורך לעובד והדבר מקשה על שינויים בעתיד.

תכן מוצלח יותר נראה כך:



מחלקה נפרדת בשם XMLBuilder בונה את ה xml (מממשקת ממשק כללי בשם DBBuilder). היא מכילה מתודות שונות ובניהם מתודה בשם append() המקבלת מפתח ואת ערכו (לצורך פשטות שניהם מסוג String). מחלקה ראשית בונה קובץ DB (שמקרה פרטי שלה הוא xml) לכלל העובדים. היא מכילה (או מקבלת) את רשימת העובדים ומוסיפה כל עובד ל db ע"י שימוש במתודה appendEmployee().

היתרון בתכן זה הוא הפרדת האחריות. המחלקה Employee מכילה את המידע של כל עובד. ישנה מחלקה אחרת שעוסקת בבניית xml כלשהו וישנה מתודה המקשרת בין employee ל xml בשם appendEmployee(). בדרך זו, ניתן בקלות לשפר\לבדוק\לתחזק כל קוד בנפרד. מן הסתם שינוי במבנה employ יגרור שינוי במתודה appendEmploy אבל שינוי ב Employee לא ישנה את התכן שיוצר את xml כלשהו. כך ניתן גם להשתמש במחלקה XMLBuilder בעוד מקומות שונים וגם להחליף את ה DB בסוג קובץ אחר (כי השתמשנו בממשק DBBuilder).

1. Chain of responsibility מממש רצף של אחריות כאשר כל יחידה לא מודעת לשאר היחידות. אם אחריות מסויימת יכולה לבצע את המוטל עליה, אז היא עושה זאת ואם לא אז היא "מגלגלת את האחריות הלאה", כלומר מעבירה את זה ליחידה הבאה. את רצף היחידות קובע הלקוח.

עקרון אחריות היחידה ממומש כאן מכיוון שישנה הפרדה מוחלטת בין תחומי האחריות כאשר את הרצף והעדיפות בין היחידות השונות קובע הלקוח.

1. עקרון CREATOR מספק כלים להחליט איזה class אחראי ליצירת אובייקט של class אחר, מספיק שאחד התנאים שראינו יתקיים בכדי ש class ייצר אובייקט של class אחר.

אחד התנאים הוא שה class היוצר משתמש במתודות של ה class הנוצר, ולכן ל class היוצר יש את הזכות לייצר אובייקט של class האחר

בדוגמה מתרגיל בית 3 , ניתן לראות למשל, ב class customer עושה שימוש במתודות של class payment.

ֿ

עקרון Information expert – מספק כלים למתן אחריות ל class, כלומר האחריות למימוש משימה מסוימת צריכה להיות של ה class המכיל את המידע הרלוונטי לביצוע המשימה.

בדוגמה מתרגיל 3, class stats מכיל את כל הנתונים הדרושים על מנת לעדכן סטטיסטיקות ולכן בכל בקשה לעדכון של סטטיסטיקה פונים אל class זה.