תכנות
מונחה עצמים
מתקדם

מסוכם משיעורי מרן הרב

ד"ר שחר גולן שליט"א

עם פירוש

"רי"ח טוב"

מאיתי הצב"י יוחנן חאיק

שנת 1011000110000

"כשם שאי אפשר לבר בלא תבן, כך אי אפשר לתוכנית בלי שגיאות של הבודק האוטומטי"

להערות, הארות ותיקונים:
yohananha@gmail.com
yohanan@ - בטלגרם
ניתן להשתמש בסיכום באופן חופשי לכולם!!

תוכן עניינים

תכנות מונחה עצמים מתקדם 3
הקדמה 3
שלושת חלקי הקורס – סקירה כללית 3
UML 4
עקרונות עיצוב תכנה 7
Extreme Programming 8
מחלקות וקשרים בין מחלקות 12
מבנה המחלקה ב-UML 12
קשרים בין מחלקות 13
ריבוי קשרים 13
סוגי קשרים בין מחלקות 14
Dependency (תלות) 14
Directed Association (קישור ישיר) 15
Aggregation 20
Composition 20
Generalization (ירושה) 23
Realization (מימוש) 24
Composite 24
Sequence Diagrams (תרשימי רצף) 25
סוגי הודעות העוברות בין האובייקטים 26
תרשימי מצבים (Statecharts) 27
מחלקה תגובתית 28
ירושת תרשימי מצבים 30
SOLID 32
פרמטרי איכות לקוד 32
עקרונות SOLID 33
SRP – Single Responsibility Principle 34
יתרונות SRP 34
OPC – Open-Close Principle 36
DIP Dependency Inversion Principle 39
תבניות עיצוב 40
Adapter 41

Iterator 42
LSP – Liskov Substitution Principle 42
Design by Contract 45
ISP – Interface Segregation Principle 46
לסיכום 50
Extreme Programming 51
Refactoring 51
Extract Method 52
Renaming 54
Inline Method 56
Variable Extraction 57
Inline temp variable 57
Extract/Inline class 58
Split temporary variable 58
Remove assignment to parameters 58
Replace “magic number” 59
Consolidate duplicate conditional fragments 59
Parametrize method 59
TDD – Test Driven Development תכנות מונחה בדיקות 60

תכנות מונחה עצמים מתקדם

הקדמה

בחלק זה נדבר קצת על הקורס, מה הוא בא ללמד אותנו, ומה הוא בא לחדש לנו.

תכנות-מונחה-עצמים (OOP) זה טכניקה ופרדיגמת תכנות. בקורס זה אנחנו נלמד להבין את המרכיבים השונים בצורה מלאה – מהם עצמים, אובייקטים, איך לעשות עיצוב של קוד, לדעת איך לתכנן את הקוד, שזה הרבה מעבר לכתיבה של הקוד עצמו, ומתמקד בהבנה של איפה צריך להיות כל דבר ואיך לשים אותו בצורה נכונה.

נלמד איך לייצג כל דבר שאנחנו רוצים בעזרת שפת UML, נלמד את השפה עצמה, ונראה כיצד אנחנו מיישים בה עקרונות תכנות, ובסוף נדבר על התכנון עצמו ועל שיטות שונות שהן קשורות לתוכנית עצמה ופחות לדיזיין.

ראשית נעשה סקירה זריזה על שלושת הנושאים העיקריים בקורס, ומה כל אחד מהם כולל, ואחרי זה נרד לרזולוציות מפורטות יותר.

שלושת חלקי הקורס – סקירה כללית

UML

שפה גרפית של דיאגרמות שנותנת לנו דרך לתאר קוד בצורה ויזואלית. ברמה הפשוטה אנחנו נלמד כיצד לתאר מחלקות ואת החלקים המרכיבים אותם, ואז את הקשרים השונים ביניהם – ירושה, תלות, הכרה ועוד), כאשר אלו יהיו התרשימים הסטטיים, ולעומתם יהיו לנו כל התרשימים הדינאמיים (Statecharts) שייבטאו לנו מעברים שונים בין מצבים תחת תנאים שונים.

נוכל לראות גם כיצד כל התרשימים האלו יכולים להתרגם לצורת קוד (אנחנו נעבוד בתוכנת Rhapsody של IBM, שהיא מאוד מרגיזה ולא משתמשים בה בשום מקום נורמלי מעבר לקורס הזה, וקורס ההמשך שלו), הקוד יהיה בשפת C++, ונבין כיצד כל דבר הכי קטן שאנחנו מגדירים בתרשים יותא אחר כך בשורות הקוד.

עקרונות תיכון תכנה

אנחנו נדבר על ה-SOLID. חמישה עקרונות שהזכרנו כבר בקורסים קודמים, ונזכיר גם בקורסים בהמשך, המנחים אותנו כיצד לתכנן את כתיבת הקוד, באופן כזה שהוא יהיה יעיל ונוח. העקרונות יתמקדו בחלוקת המחלקות עצמן, ריכוז העבודה ועוד, כאשר המטרה היא לתת לנו קווים מנחים, שייתנו לנו את היכולת להסתכל על קוד ולהעריך את הנכונות שלו מעבר למבחן הפשוט (והחשוב) של "עובד/לא עובד", אלא לדעת איך הקודו יהיה "נכון/לא נכון". המדדים שאנחנו ניתן יהיו אובייקטיביים, ולא רק הערכה סתמית.

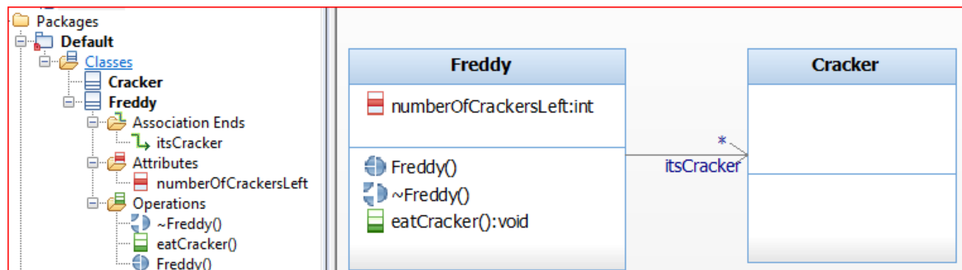
Extreme Programming

נושא זה מתייחס ממש לכתיבת הקוד, ומציג גישה מסוימת לכתיבה בעצמה. אנחנו נתמקד בשלושה עקרונות הנלמדים תחת מתודה זה – **Test First Programming** – בדיקת קוד לפני כתיבה. **Refactoring** – כתיבה ושיפור תוך כדי כתיבה של הקוד, באופן שייתן לנו את האפשרות ליצור להוסיף על הקיים באופן בטוח ונוח. **Pair Programming** – גישה שהיום כבר פחות פופולרית, אבל כרעיון אנחנו מדברים על חשיבות המעבר הנוסף והבלתי-משוחד על הקוד, על ידי זה שבנוסף למתכנת שכותב את הקוד בעצמו, מעבירים את הכתוב למישהו אחר שייעבור וייראה אם יש בעיות בכתיבה או במימוש.

ניכנס עכשיו לרזולוציה קצת יותר מפורטת של כל חלק ומה הוא כולל –

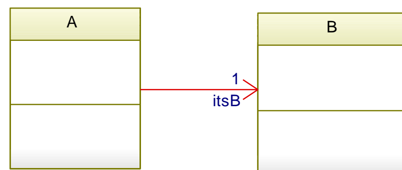
UML

Unified Modeling Language מייצג לנו שפה, ולא רק תרשימים, אלא ממש שפה בפני עצמה של הדרכים השונות שאנחנו יכולים להציג את התוכנית שאנחנו בונים.



ברמה הפשוטה ביותר, אנחנו יכולים לראות כאן שתי תצוגות שונות: האחת, תצוגת על של כל התוכנית עצמה (מצד שמאל של התמונה), עם תיקייה שמכילה את המחלקות השונות, את המאפיינים והיכולות של כל מחלקה, והקשרים למחלקות אחרות. החלק השני, מימין, מראה לנו את כל מה שכתוב משמאל, רק בצורה ויזואלית יותר – יש לנו שתי מחלקות "פרדי" ו"קרקר"[1], ביניהם יש איזה קשר מסוים (יש חשיבות גם לסוג החץ, המציין את סוג הקשר), וכן במחלקה פרדי אנחנו יכולים לראות שיש חלוקה לשני אזורים – החלק העליון מכיל את המאפיינים של המחלקה, והחלק השני את הפעולות שפרדי יכול לעשות, במקרה שלנו לאכול קרקר.

עד כאן הכל טוב ויפה, אך מעבר לזה, דיברנו על כך שהתכנה יכולה לייצא את כל זה לקוד. נראה דוגמא פשוטה ביותר בלי יותר מידי פרטים:



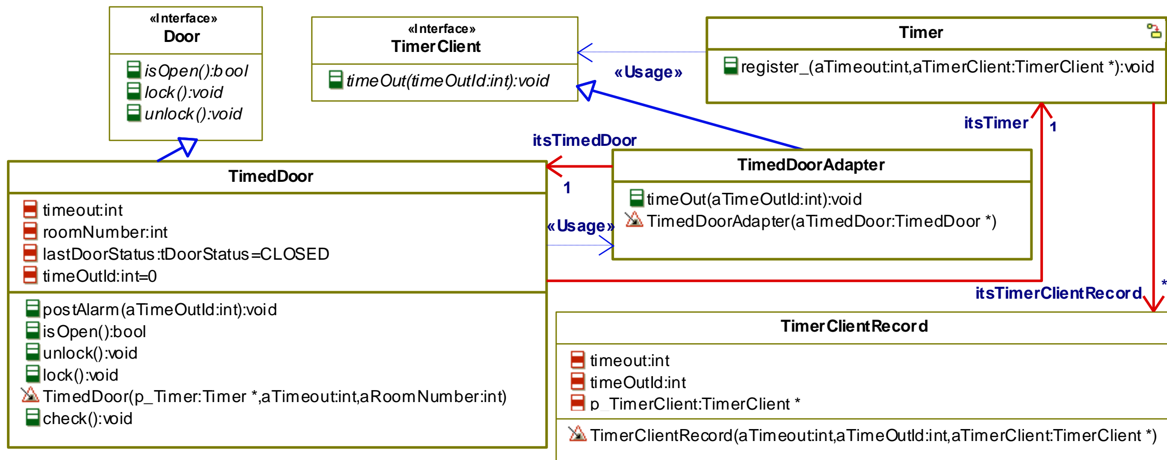
אם ניקח את שתי המחלקות הללו ונבקש מהרפסודי ליצור את הקוד, מה שנקבל יהיה הדבר הבא –

```
class A {
public :

#ifdef _OMINSTRUMENT
    friend class OMAnimatedA;
#endif // _OMINSTRUMENT

    ////    Constructors and destructors    ////

    //## auto_generated
    A();

    //## auto_generated
    ~A();

    ////    Additional operations    ////

    //## auto_generated
    B* getItsB() const;

    //## auto_generated
    void setItsB(B* p_B);

protected :

    //## auto_generated
    void cleanUpRelations();

    ////    Relations and components    ////

    B* itsB;        //## link itsB
```

[1] יש בקורס איזושהי אובססיה מוזרה לזוג צ'ינצ'ילות שגידל הבן של מי שכתב את הקורס במקור ד"ר גלנט, הבן עצמו כבר למוד רפואה או משהו ומזמן לא מתעניין בצ'ינצ'ילות, שמן הסתם גם הם מתו בשיבה טובה.
צמד החיות נקראו "פרדי" ו"פרדריקה" ומה שהם עשו רוב היום, היה התעסקות עם קרקרים שונים.

כל ההערות והמסביב, מגיעים ביחד עם הקוד עצמו, ולא נתעסק בפרטים, אלא רק נראה שזה עושה ויופי לנו.

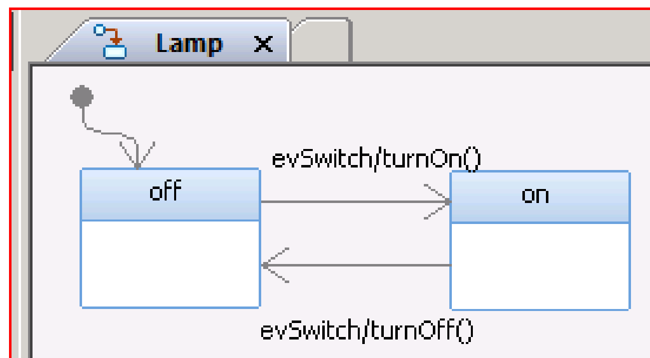
ברמה ה"מונחית עצמים" יש לנו מספר הכרות שונות של מחלקות, ואנחנו יכולים לראות שבעבור כל סוג של עצם וקישור יש לנו חץ מיוחד –



שוב, כרגע לא נתעסק בדיוק בסוג הקשרים, אלא רק מעיפים מבט מלמעלה.

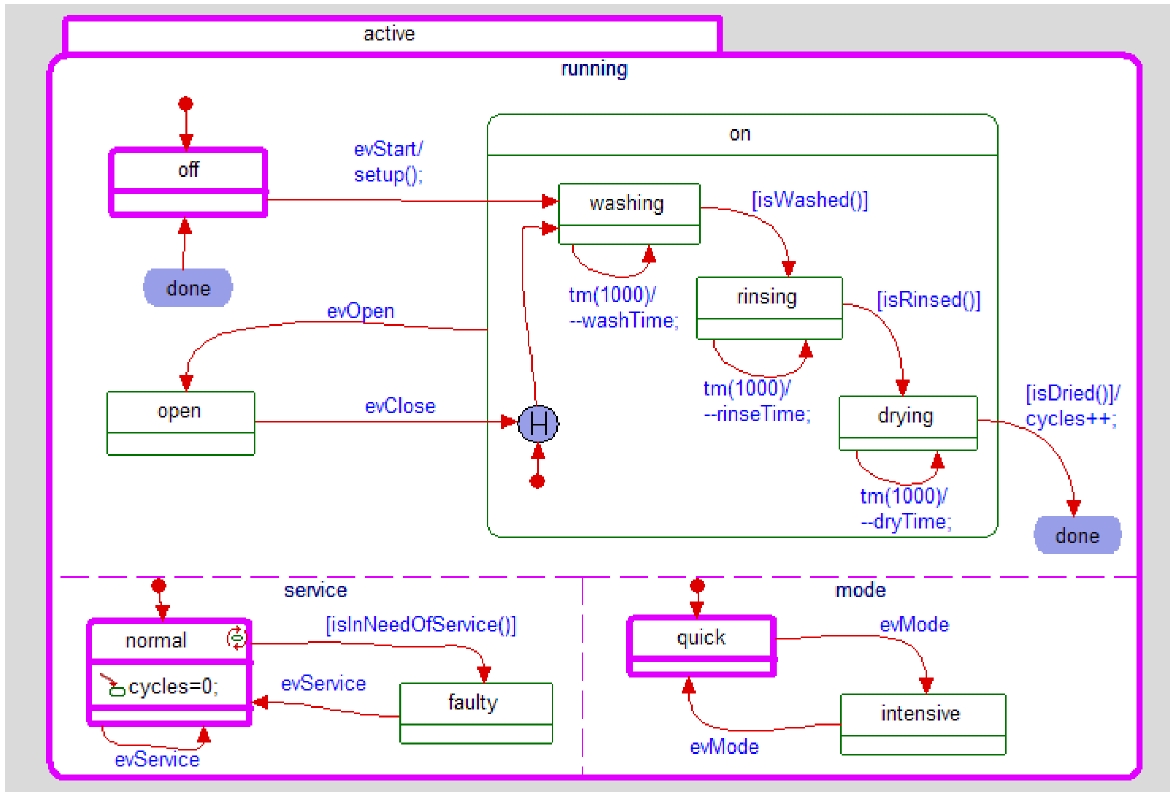
החלק השני של סוגי התרשימים, אלו התרשימים הדינאמיים. כל התרשימים של המחלקות הסטטיות מייצגות לנו מבט (כמה מפתיע) סטטי, ולא מציג לנו שום רצף פעולות של מה שקורה בפועל. בשביל כל אלו מגיעים ה-Statecharts השונים.

המייחד את התרשימים הדינאמים, הוא בכך שיש לנו נקודת התחלה (לפעמים גם סוף), של תוכנית, והמעברים בין המצבים השונים (מצבים ולא מחלקות), מתבצעים תחת תנאים מסויימים. ניתן לראות את הדיאגרמה הפשוטה הבאה של המנורה -



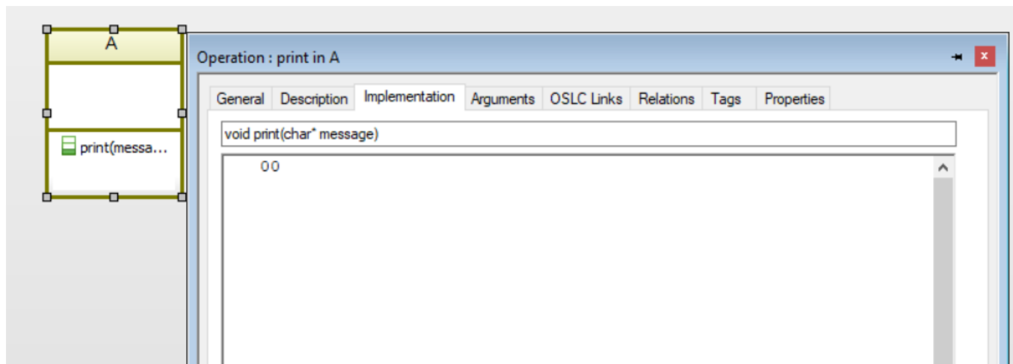
יש לנו נקודת התחלה פשוטה של המנורה, בה היא כמובן כבויה כברירת מחדל, וכאשר מדליקים את המתג, המנורה עוברת למצב "דלוק", וחוזר חלילה – כאשר המתג חוזר למצב הראשוני, המנורה כבויה. מדהים.

זה כמובן רמה מאוד פשוטה, שאנחנו לא באמת צריכים לשרטט, אבל אנחנו נסתכל על משהו קצת יותר מורכב, כמו מדיח כלים, כאן התרשים יהיה הרבה יותר מפורט. התרשים יכיל את הפעולות כאשר המדיח פעיל, את המעבר לביצוע פעול שטיפה, המצבים השונים של השטיפה עצמה, הייבוש, כל הבלאגן וההמתנה ברקע, וכו את מצבי המשנה האפשריים. כל אלו מופיעים בתרשים שמופיע כאן, וכרגע לא אומר לנו כלום –



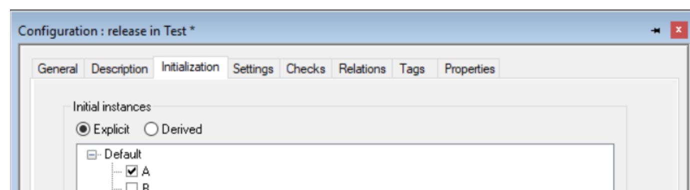
כמו שראינו כבר, יש לנו את היכולת להמיר את התרשימים השונים לקוד מסודר. אבל מה קורה אם אנחנו רוצים להוסיף פונקציות נוספות? או שאנחנו רוצים להוסיף איזה פוונקציונליות מסוימות שלא תוגדר לנו בצורה הבסיסית של התכנה?

לכל זה יש לנו את האפשרות להוסיף בחלון של הכתיבה העצמית את כל מה שנרצה באופן חופשי (בכפוף לשפת הקוד כמובן).

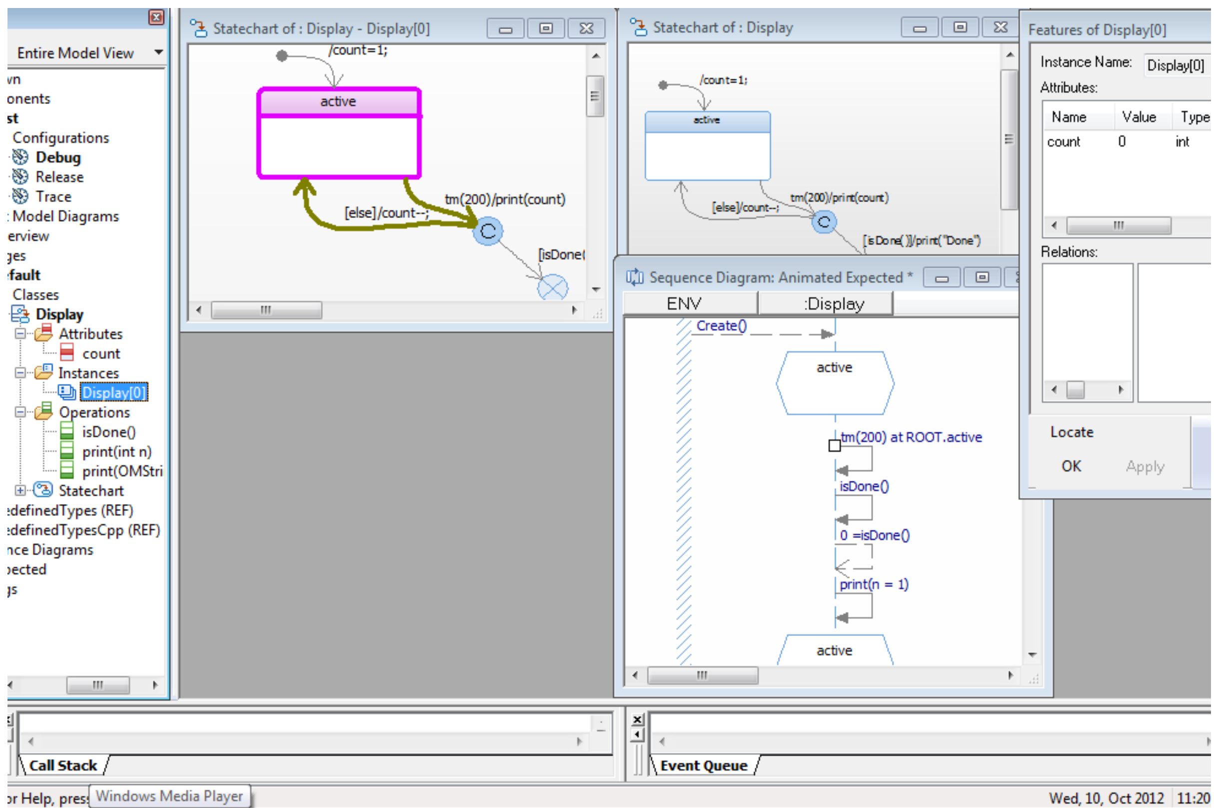


כאן אנחנו מגדירים פונקציית הדפסה ויכולים לכתוב שם איזה הדפסה שנרצה.

בנוסף, כאשר נגדיר מסר מחלקות, נוכל לעשות קונפיגורציה של התוכנית שתקבע לנו את סדר הפעולות. אם יש לנו למשל שתי מחלקות שהגדרנו, אך אנחנו יודעים שמספיק לנו אם רק מחלקה אחת מהן תאותחל כברירת מחדל, אנחנו יכולים בחלון הקונפיגורציה לסמן רק את המחלקה הזאת, ולראות מה יצא שם –



הדבר האחרון שנראה בנוגע לרפסודי כרגע, הוא מגוון האפשרויות לתרשימים דינאמיים –



התרשימים כאן מתארים החל מהמצב הראשוני של התרשים ששירטטנו, ועד לתרשים שנקרא Sequence Diagram, שמראה לנו תוך כדי עבודה אילו מחלקות נוצרות ואת הקישור והעבודה הנוצרות תוך כדי שהתכנית פועלת.

עקרונות עיצוב תכנה

למה בכלל אנחנו לומדים עקרונות לעיצוב תכנה? למה אנחנו פשוט לא כותבים את מה שאנחנו צריכים וזהו?

אחת הבעיות הגדולות שיש לנו כשאנחנו בלימודים, הוא שהתוכניות ואפילו הפרוייקטים שאנחנו עושים, הם בהיקף מאוד מוגבל. הן ברצה של כמות הקוד שאנחנו כותבים בפועל, והן בזה שבסוף, אנחנו כותבים משהו לבד, מקסימום עם עוד חבר, והדבר הזה לא מייצג את העולם האמיתי.

אם ניקח פרוייקט סביר מבחינת גודל, אנחנו נעבוד בתור חלק מצוות שמונה עד עשרה אנשים. הצוות הזה הוא בדרך כלל חלק מממספר צוותים שונה, וככל שאנחנו מרחיבים את המבט, אנחנו מבינים כמה שאנחנו בורג קטן במערכת. הכוונה כמובן אינה לבאס אותנו אל מול המערכת, אלא המחשבה על כך שאם אנחנו נבוא ונכתוב פונקציות שלמות בלי שמות משתנים משמעותיים (סתם כדוגמא), יבוא מתכנת אחר שעובד על הקוד, או במקרה הנפוץ לא פחות, אנחנו בעצמנו נסתכל על מה שכתבנו אחרי שבוע, וננסה להבין מה זה ה-X וה-Y הזה ומה הם עושים שם. מעבר לזה, תוכנה היא דבר חי ונושם. אנחנו לא כותבים את התכנה פעם אחת, משחררים אותה לאוויר העולם, וסופרים את הבוכטות באי שקנינו. אנחנו צריכים כל הזמן לתחזק, להוסיף פיצ'רים שונים, לתקן באגים, לעדכן גרסאות וכו'. אם אנחנו לא נכתוב ונחשוב על כל הדברים האלה מראש, יהיה לנו הרבה יותר קשה לתחזק את התוכנה.

על מנת שבעיות כאלה לא יתעוררו לנו, או לפחות יהיו יותר נוחים לשליטה, אנחנו מדברים על אוסף של עקרונות המהווים כללי אצבע ליצירת תכנה גמישה וניתנת לשינויים על ידינו ועל ידי מתכנתים אחרים. דבר זה נעשה על ידי הפשטה של הקוד , שבירידה לפרטים עצמם אנחנו נשארים בהירים ומובנים. עקרונות אלו, המכונים SOLID, מציגים חמישה עקרונות המהווים אקרוסטיכון של המילה וכל אחד מהם הוא עיקרון וביטוי בפני עצמו-

Single Responsibility Principle (SRP)

עקרון האחריות היחידה מדבר על כך שלכל מחלקה או מודול אמור להיות אחריות אחת ויחידה. כלומר, יכול להיות שפונקציה מסוימת תעבוד על מטרת על, ועל הדרך תגרוף כמה דברים, אבל אנחנו לא אמורים לייצר פונקציה שאמורה גם להוסיף משתמש למערכת וגם לעדכן אותו, אלא ניצור פונקציה אחת ליצירת משתמש, ופונקציה נוספת לעדכון משתמש. דבר זה יהיה הרבה יותר קל לתפעול, כאשר נחפש פשוט את החלק שאנחנו רוצים, ולא נצטרך לעבור בכל המודולים השונים לבדוק אם מופיע שם דברים נוספים.

Open Close Principle (OPC)

עקרון זה אומר, כי התכנה צריכה להיות פתוחה להרחבות, וסגורה לשינויים. מה זה אומר? במסגרת כל התחזוק של הקוד, אנחנו עלולים להידרש לא מעט להוספה של אפשרויות – למשל, אנחנו רוצים שסיום של פעולה מסוימת ישלח לנו חיווי גם למייל וגם ל-SMS. עיקרון ה-OPC אומר שבניית התכנה באופן נכון ייתן לנו את האפשרות לעשות את כל זה בלי לחשוש מכך שמשהו יישבר.

מה שכן, אנחנו צריכים להיות סגורים לשינויים – כלומר אנחנו לא יכולים להחליט פתאום שאנחנו סוגרים איזור בקוד כי הוא פחות שמיש או משהו בסגנון, אלא להזהר בחלקים כאלה.

Liskov Substitution Principle (LSP)

עיקרון ההחלפה של ליסקוב, מדבר על ענייני ירושות. בגדול, התנהגות הבנים אמורה להיות נגזרת של התנהגות האב, וזה בא לידי ביטוי בכך שאנחנו יכולים להגדיר בן ולהתייחס אליו בתור האבא, אבל לא להיפך. מבולבלים? כולם ככה, נקווה שיוסבר יותר טוב בהמשך.

Interface Segregation Principle (ISP)

עקרון הפרדת הממשקים מגדיר שכאשר אנחנו מממשים ממשק במחלקה מסוימת, אנחנו שמים לב שאנחנו לא סתם מעמיסים ממשקים מיותרים, אלא מממשים רק את מה שדרוש. אנחנו לא ניקח ממשק שלם רק בשביל שתי פונקציות שנמצאות בו, אלא פשוט נטמיע רק אותם בתוך המחלקה.

Dependency Inversion Principle (DIP)

עיקרון היפוך התלות מדבר על כך שכאשר אנחנו דורשים משכבה גבוהה לגשת לשכבה נמוכה יותר, אנחנו נשתדל לנתק עד כמה שאפשר את התלות. אנחנו לא נבקש מאיזה מחלקה מופשטת לגשת לפונקציה שעושה חישוב פשוט וקטן, אלא נעבור דרך משהו מופשט יותר שהמודולים המורכבים תלויים בו. שוב, כל זה יוסבר בהמשך, ואנחנו רק מציגים ראשי פרקים.

Extreme Programming

Agile

שיטה זו היא פיתוח משך לשיטת ה"אג'ייל", ונוצרה על ידי אחד מהיוזמים וההוגים של האג'ייל. אבל לפני שנדבר על מה שהוא הוסיף לשיטה, נדבר על ארבע הערכים של שיטת האג'ייל כפי שנוסחו[2]-

אנו חושפים דרכים טובות יותר לפיתוח תוכנה תוך עבודה ועזרה לאחרים לעשות זאת.
אלו הם ערכינו:
אנשים ויחסי גומלין על פני תהליכים וכלים
תוכנה עובדת על פני תיעוד מפורט
שיתוף פעולה עם הלקוחות על פני משא ומתן חוזי
תגובה לשינויים על פני מעקב אחרי תוכנית
כלומר, בעוד שיש ערך לפריטים בצד שמאל, אנחנו מעריכים יותר את הפריטים **בצד ימין**

מה המשמעות של המנשר הזה, ומה הערכים שהובלו בו? השיטות השונות עד לאותה תקופה, היו שיטות מאוד סדורות וברורות מההתחלה ועד הסוף – תעשה את שלב 1, תעבור לשלב 2 וכו'. עבור כל שלב היו את עשרות הפרוטוקולים והטפסים שצריך למלא – תחשבו רק על הפרוייקט המדומיין שאנחנו יוצרים בתרגול – כמה כתיבה וחירטוטים רק בשביל פרוייקט שהוא אפילו לא יגיע לכדי כתיבת שורת קוד בודדת! בנוסף, החוזים היו מאוד סגורים לשינויים מול הלקוח – אנחנו סוגרים איתו את מה שנעשה, ואז פעם הבאה שאנחנו נפגשים, זה כאשר הלקוח מקבל מאיתנו את התכנה בצורה המוגמרת. והדבר הנוראי ביותר – אם אנחנו עוקבים אחרי כל התכנית בצורה מושלמת, כל שינוי הכי קטן יכול להיות הרה-אסון, עשינו הכל לפי התכנית אבל טעינו באיזה דרישה, אז בסוף אנחנו עלולים להגיע למצב בו אנחנו בכלל במקום אחר.

כנגד כל זה, אומרים אנשי האג'ייל את הסעיפים הבאים:

אנשים ויחסי גומלין על פני תהליכים וכלים

השאיפה שלנו היא להגיע למצב בו אנחנו עובדים בצורה מסודרת ומתאימה, כלומר, כל התהליכים שלמדנו וכל הכלים היקרים ביותר, לא יועילו לנו אם אנחנו לא מסוגלים לעבוד כצוות – אנחנו צריכים להשקיע מאמץ ביצירת צוות שעובד נכון ובצורה נכונה, ודבר כזה יפצה אפילו על כלים שהם לא ברמה הכי גבוהה.

בגישות המסורתיות נתנו את עיקר החשיבות לתהליך המובנה, הדרך הברורה והמסודרת שצריך לעבור. בנוסף, עיקר ההתמקדות היתה בכלים השונים שיהיו ברמה הגבוהה ביותר.
בגישה האג'ילית לעומת זאת, התמקדו באנשים על פני התהליך – כלומר, ההצלחה של כל התהליך לא תלויה דווקא בכלים הגדולים ביותר. אפשר לקחת כלים שהם סבירים, ואנשים שהם מתכנתים שהם ברמה טובה, אבל הם מסוגלים לעבוד ביחד בצורה טובה, וזה יותר נוח, מאשר איש אחד גאון שלא מסוגל לעבוד עם אף אחד אחר.

תוכנה עובדת על פני תיעוד מפורט

בסופו של דבר, אנחנו אנשי תכנה, התיעוד המפורט של כל דבר שנעשה אולי נראה נכון, אבל הוא מבזבז לנו זמן יקר. אם נכתוב את התכנה כמו שצריך, והיא תעבוד, התיעוד יהיה משני להכל.

בגישות המסורתיות על מנת להבין את התכנה ואיך היא עובדת, היית צריך לקרוא את כל המסמכים, והתיעוד שעל גבי הקוד. כל מסמך שכזה יכול להיות ארוך מאוד – עבור כל פונקציה צריך להוציא בסופו של דבר מסמך. כאשר מגיע איש צוות חדש, והא צריך לעבור על כל הטופסיאדה ורק אז הוא יוכל להתחיל לעבוד
בגישות האג'יליות מתמקדים בהעברת מסר דרך הקוד. אם הקוד כתוב היטב (שמות משתנים ופונקציות

[2] החלק שמסביר פה את האג'ייל זה העתק-הדבק מהסיכום שלי בקורס הנדסת תכנה. אני לא רואה צורך לנסח את זה מחדש כרגע.

משמעותיים למשל), אפשר לקרוא את הקוד ממש כמו סיפור, ולהבין ישר מה קורה ומה הולך לאן. כמויות המסמכים הם מאוד נמוכות יחסית, מאחר ואם אנחנו צמודים למסמכים, כל שינוי קל בתכנה דורש גם לשנות כמויות של ניירת, אם יש חוסר סנכרון קטן, כל הטפסים לא רלוונטים כי הם מתייחסים לדבר לא קיים. כאשר יגיע עובד חדש, פשוט ישב איתו מישהו ויסביר לו את המערכת, ברגע שגם היא תהיה בנויה נכון, אז לא יהיה צורך להתעכב יותר מידי על כל דבר.

שיתוף פעולה עם הלקוחות על פני משא ומתן חוזי

ברגע שהלקוח הוא חלק מתהליך העבודה, ולא רק מטרד של חוזים וכסף, העבודה שלנו תצליח יותר, ותקלע יותר לטעם הלקוח – לקוח מרוצה, אומר שאנחנו מרוצים.

בגישות המסורתיות הזמנת תוכנה היא הזמנת מוצר. אתה יושב בבית ומחפש מה שאתה רוצה, ומחכה שהמוצר יגיע אליך – לכן, הרבה יותר קל ליצור חוזה, כי הכל חקוק באבן. כמו כן, אין סיבה לייצר אינטראקציה בין שני הקצוות של הלקוח-מתכנת, בדיוק כמו שאנחנו לא מזמינים ארוחה במסעדה ואז נדחפים לטבח ומתחילים לשאול שאלות.

בגישה האג'ילית אנחנו מתקשים לנסח חוזים בשלבים הראשונים. התכנית עוברת הרבה איטרציות ושינויים, כך שהמיר חייב להיות גמיש. החוזה לא יכול לציין עלויות ולוחות זמנים מסודרים, פשוט כי אין דבר כזה. מסיבה זו, אנחנו שואפים גם ליצור קשר עם הלקוח לכל אורך העבודה, לוודא שהלקוח יקבל את המוצר שהזמין (ושהוא יבין אחר כך את החשבון שיגיע אליו).

תגובה לשינויים על פני מעקב אחרי תוכנית

כל תכנית היא בסיס לשינויים, יכול להיות שבמהלך העבודה, הלקוח מבין פתאום שצריך לעשות שינוי, הוא מבין את הדרישות שלו יותר טוב, אבל אם אנחנו עוקבים אחרי התכנית, יהיה לנו מאוד קשה להגיב לבקשה שלו בצורה נכונה. לעומת זאת, אם אנחנו חיים על שינויים, אז נאפס את עצמנו לטובת הדרישה החדשה ונמשיך הלאה.
בגישה המסורתית התכנה היא קבועה ואחידה. התכנון עבור שינויים נעשה מראש ומוגדר בצורה ברורה כמה שנים קדימה.
בגישה האג'יילית בונים את התוכנית בצורה גמישה וכבסיס עבור השינויים שיבואו. כל התכנונים ייעשו לזמן הקרוב בלבד, ותכניות מאוד כלליות וגולמיות יתוכננו עבור זמן רחוק יותר, אך לא בצורה ספציפית.

חשוב לזכור, שהאג'ייל לא שורף את כל מה שהיה, ואומר "יאללה בלאגן!" ומוביל אותנו לעולם בלי נהלים וסדר עבודה, אלא מתמקד בפרופורציות של החשיבות וההעדפה שלנו על צורה עבודה "אנושית" על פני קיבעון מחשבתי.

Refactoring

כאשר אנחנו כותבים את התוכנית בצורה אג'ילית, אנחנו דואגים לכך שהקוד עצמו יהיה קריא – במקום לכתוב על מסמך מה הפונקציה אמורה לעשות, אנחנו דואגים ששם הפונקציה יסביר את עצמו, הערכים שהוא יקבל יהיו בעלי שמות משמעותיים וכו'. ברגע שאנחנו עושים דבר כזה, אנחנו מסוגלים לעשות כתיבה מחדש של הקוד על מנת

להוסיף ולשנות פיצ'רים בצורה יחסית זורמת. למשל, אם אנחנו עושים פונקציה שתשלח לנו אישור במייל בסיום פעולה, ואחרי זה נרצה שיהיה לנו אישור נוסף שיישלח ב-SMS, אז אנחנו נגדיר למשל ממשק של "עדכון בסיום פעולה" ואז נממש אותו כל אחד באיזור שלו לפי סוג האישור במייל או בהודעת SMS. כמובן שזה רק דוגמא, ונראה בדיוק איך לעשות את זה בהמשך.

Test Driven Development

הרבה לפני שאנחנו ניגשים לכתיבת התוכנית, אנחנו אוספים את הדרישות של התוכנית. מגיע הלקוח ומבקש תוכנית מסוימת שעושה כך וכך. הגישה המונחית בדיקות אומרת שעוד לפני שאנחנו מתחילים ליישם את האלגוריתמיקה של הקוד עצמו, אנחנו כבר צריכים לחשוב על הטסטים הרלוונטים. דוגמא פשוטה לזה הוא חישוב של חילוק – אנחנו נצטרך לבדוק חילוק שלם, חילוק עם שארית, חילוק באפס ולמצוא את כל מקרי הקצה ואת המקרין הפשוטים שאנחנו אמורים לעבור עליהם. ברגע שנכיר את המגוון שאנחנו אמורים להכיר ולעבוד, יהיה לנו יותר קל לכתוב את התוכנית שתיישם את הדרוש. באופן הזה, אנחנו גם נמנע מעצמנו הכנסה של פונקציונליות מיותרת – אם אנחנו הגדרנו נכון את הדרישות, אין שום סיבה להוסיף דברים שייראו לנו נחמדים, אבל הם רק מכבירים על התוצר הסופי.

Pair Programming

החלק הזה הוא בעצם ליבת השיטה של ה"תכנות אקסטרים". הגישה סוברת שעל כל שורה שתיכתב אנחנו צריכים שני מתכנתים. אחד שיכתוב, והשני שיבדוק שזה באמת נעשה באופן נכון. ברמה התיאורטית, ברור שזה דבר נכון ומועיל, אבל בפועל, אף אחד לא רוצה לשלם כפול על כל שורת קוד.

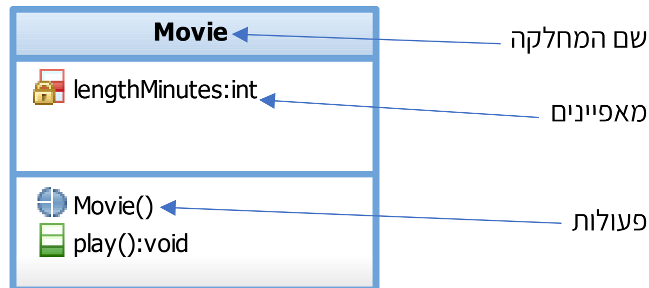
מה שכן נעשה מגישה זאת, הוא השמירה על עיקרון ה- Code Review שמעבירים את הקוד למישהו אחר שיוודא אותו שהוא לא אותו אחד שכתב הכל.

לסיכום ההקדמה: כל מה שנלמד כאן יוביל אותנו להבנה של הדרך לכתוב קוד נכון, אמין ושלא יגרום לתקלות. או לפחות, לא הרבה כאלה.

מחלקות וקשרים בין מחלקות

נכנס לתוך סימוני ה-UML השונים (לאו דווקא ברפסודי, אלא בכלל). נסביר את משמעויות הסרטוט השונים, ומה ההבדל בפועל בין כל סוג קשר מבחינת מימוש.

מבנה המחלקה ב-UML



מחלקה מבוטאת בדרך כלל על ידי מלבן, כאשר הוא מחולק לשלושה חלקים – שם המחלקה, המאפיינים והפעולות.

שם המחלקה הוא החלק שאין הרבה מה להסביר.

המאפיינים (Attributes) מכילים את כל המשתנים המוגדרים במחלקה. הסימון של כל משתנה בפני עצמו משתנה בין התוכנות השונות, אך בגדול – בצד שמאל של שם המשתנה תצויין רמת הפרטיות של המשתנה (private, protected, public), ומצד ימין, לאחר הנקודותיים, יופיע הטיפוס של המשתנה.

הפעולות (Operation) הם כל הפונקציות של המחלקה. במקרה שלפנינו אנחנו יכוים לראות את הבנאי של המחלקה (חלקי הפאזל הכחולים), כאשר ה-Destructor של מחלקה, אם יוגדר, יופיע באופן דומה מאוד רק קצת יותר מפורק ועם ~ ליד שם הפונקציה.

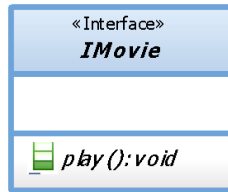
הסוג השני של הפעולות, זה הפונקציות ושאר המתודות, כאשר מצד שמאל יופיע רמת הפרטיות, כאשר ככל שלב בסולם יהיה פומבי יותר ככל שהוא נמוך. הטיפוסים המתקבלים בפונקציה יופיעו בתוך הסוגריים, ולאחר הנקודותיים יוגדר הערך המוחזר של כל פונקציה.

ברמת הקוד, כאן אין הרבה מה לראות מעבר, מאחר והפעולות וכל המאפיינים הם פשוטים וללא כל קשרים וחיבורים למחלקות אחרות. יש זכרו שכל מימוש מדויק של הקוד גם הוא משתנה מתוכנה לתכנה – אם נשתמש ברפסודי או בסטארUML, אנחנו נקבל קוד שהוא קצת שונה בפרטים, אך הוא מממש את אותם עקרונות. הבסיס של בניית מחלקה ברמת הרפסודי, יתחלק לשני קבצים – ה-Header וה-cpp, ויייראה פחות או יותר כך-

```
// A.h
#ifndef A_H
#define A_H
class A {
public :
   A();

   ~A();
};
#endif
```

```
// A.cpp

#include" A.h"

A::A(){}

A::~A(){}
```

כמו שניתן לראות אין פה הרבה, כמובן זה כי לא הגדרנו כלום. אנחנו יכולים לראות בהמשך איך מכניסים כל מיני מימושים לתוך הבנאי, מה שכמובן ישנה את הנראות של הפונקציות – קובץ ה-h יכיל את הפרמטרים המוכנסים לבנאי, וב-cpp יופיע לנו המימוש.



אם נרצה להגדיר ממשק, הוא יצוייר בדיוק באותו אופן, רק שהמילה <<interface>> תופיע מעל שם המחלקה. ה-I שמופיעה לפני שם המחלקה הוא רק מוסכמה ולא מוכרח מעצמו.

אם גם נרצה להכניס פונקציה לתוך הממשק (מה שמן-הסתם נעשה), הפונקציה תופיע כפונקציה וירטואלית באופן הבא:

```
virtual void play() = 0;
```

קשרים בין מחלקות

לצורך תחילת הדיון, רק נגדיר את הקווים הכלליים של ציור הקשרים בין המחלקות. לרוב ידובר בסוג של חץ היוצא ממחלקה אחת לאחרת – שימו לב – המחלקה "סרט" מקושרת למחלקה של "שחקן".

בקצה הקישור יופיע שם הקישור או המצביע למחלקה, שברפסודו מוגדר בברירת מחדל בתור “itsXXXX” לפי שם המחלקה אליה מקשרים, והמספר שמעל החץ מבטא את מספר המופעים שאנחנו מרשים לקשר.

ריבוי קשרים

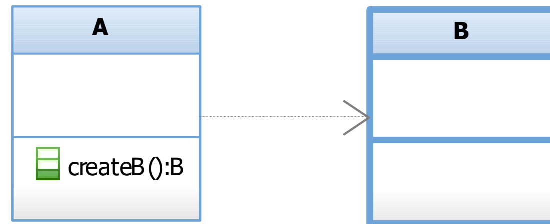
המספר מעל החץ מבטא את מספר הקשרים המותר, כאשר האפשרויות הם כדלקמן:

- 1 – ברירת המחדל. אם לא נגדיר כלום, יהיה מדובר על מופע בודד שיקושר למחלקה.
- X – מספר קבוע. כאשר יוגדר מספר קבוע המשמעות היא למספר מופעים סגורך לא פחות ולא יותר (למעשה, 1 הוא מקרה פרטי של זה).
- X..Y – טווח מופעים. כאשר יוגדר טווח, מספר המופעים יכול לגדול או לקטון בטווח המוגדר.
- * - אפס או יותר. בדיוק כמו בהגדרת שפה באוטומטים, אנחנו יכולים להגדיר שאולי נשתמש במחלקה אליה אנחנו מקשרים, ואולי לא. ואם נשתמש אין לנו הגבלה לכמות המופעים שאנחנו עלולים ליצור מאותה המחלקה.
- X..* - מקבוע ומעלה. אנחנו מגדירים טווח, האומר לנו מה המספר המינימלי של המופעים שיכולים להיות לנו, אך אין לנו שם תקרה על מספר המופעים אותם אנחנו יכולים ליצור.

סוגי קשרים בין מחלקות

כל סוג חץ מבטא סוג קשר אחר. עלינו לדעת מה אומר כל סוג קשר הן מבחינה רעיונית, כלומר מתי להשתמש בו, והן מבחינת המימוש, כלומר מה יועיל לנו בקוד ואיפה.

Dependency (תלות)

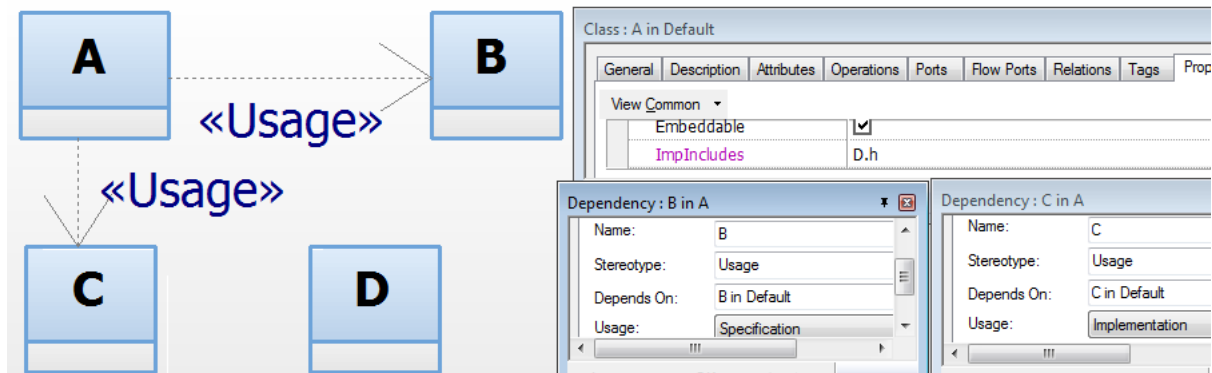


מחלקה A מכירה/תלויה במחלקה B. כלומר, הקשר הוא כזה שמתישהו במחלקה A נשתמש ביכולות של B. למשל, בתור חלק משלב הבניה, או שיש לנו פונקציה שמקבלת מופע של מחלקה B. במקרים האלו, אנחנו לא באמת חייבים להחזיק מראש מופע של B או משהו בסגנון, אלא אנחנו אמורים להכיר בקיומה של המחלקה ולדעת איך לעבוד איתה בעת הצורך.

ברמת הקוד – מאחר ואנחנו מדברים רק על הכרה של המחלקה B, המחלקה A תכיל בתוכה "include B.h", בדרך כלל בקובץ header שלה, למרות שישנם מימושים שונים בהם ה-include יהיה דווקא בקובץ ה-cpp. נראה זאת בהמשך.

```
// file A.h
#ifndef A_H
#define A_H
#include "B.h"
class A {
public :
A();
~A();
};
#endif
```

המימוש הזה מתייחס למימוש הפשוט ביותר של התלות הזאת. ברפסודי יש לנו עוד מספר שיטות להגדיר הכרה שמשפיעים במקצת על הקוד. נעבור על זה גם בתרגולים, אבל כדאי להסתכל ולזכור את ההבדלים-



הקוד המתאי של המחלקה A ייראה פה כך, ונסביר אותו –

```
//file A.h
#ifndef A_H
#define A_H
#include "B.h"
class C;
class A {
public :
A();
~A();
};
#endif
```

כאן אנחנו רואים את שלושת הדרכים. המחלקה B היא כמו שראינו קודם – ההגדרה שלה היא מסוג -Usage Specification, כלומר רק הכרזה, ולכן יש לנו include למחלקה.

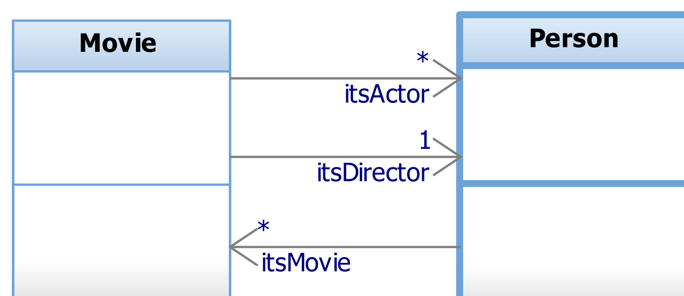
המחלקה C לעומת זאת, מוגדרת גם היא עם חץ, אבל התלות שלה היא מסוג implementation, אנחנו ממשים את C ולכן כמו שאנחנו יכולים לראות יש לנו הכרזה על מופע ריק של המחלקה C.

המחלקה D היא המימוש היור מעניין, שמגיע בכלל בלי חיצים. בשביל לממש אותו אנחנו צריכים לעשות את זה דרך האפשרויות של המחלקה תחת ImpInclude שם אנחנו נרשום את שם קובץ ה-Header שאנחנו רוצים להשתמש בו. איך הוא יבוא לידי ביטוי?

```
//file A.cpp
#include "A.h"
#include "C.h"
#include "D.h"
A::A() {
}
A::~A() {
}
```

בקובץ ה-cpp אנחנו נחזור ונעשה include לכל המחלקות, כולל המחלקה שלא מקושרת עם חיצים, וכך נוכל לקבל אפשרות למימוש של מה שקיים בה.

Directed Association (קישור ישיר)



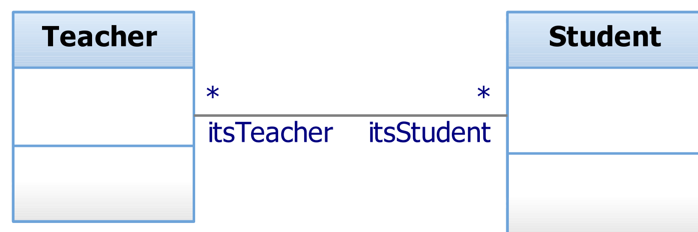
כאן מדובר ברמת הכרה יותר עמוקה – המחלקה A משתמשת במחלקה B. אם אנחנו רוצים ממש לקבל מופעים חיים של המחלקה B. אם נסתכל על הדוגמה שהביאו לנו עם אותן שתי מחלקות של סרט ו"איש", אנחנו יכולים כבר לראות בצורה מאוד פשוטה דברים שקורים פה – לכל מופע של המחלקה סרט יש מספר שחקנים (או אף שחקן בכלל,

במקרים מאוד קיצוניים), ובמאי אחד. כאשר לכל איש, שחקן או במאי, יכולים להיות מספר סרטים בהם הוא משתתף, או אף סרט בכלל אם הם עדיין בשלב של למלצר בבתי קפה ולחכות לפריצה הגדולה.

מבחינת הקוד – אם ניקח דוגמה פשוטה של A שמקושר למחלקה B, אזי הקובץ header של המחלקה A יכיל מופע כלשהו של המחלקה B, ואז יפעיל אותו בתור פוינטר לכל שאר המופעים שיגיעו.

יש לשים לב – אם נגדיר את המחלקה כמו בדוגמה למעלה, תהיה לנו קצת בעיה, מאחר ואמרנו שאנחנו מגדירים בכל מחלקה מופע של המחלקה השניה, ואז אנחנו עלולים להתקע בתוך פרדוקס שכל הדרה של מחלקה תלויה בהגדרה של המחלקה השניה, שגם היא בתורה לא יכולה להיווצר וניתקע כך לנצח.

על מנת שזה לא יקרה, אם אנחנו מגדירים מחלקה אם קישור דו-כיווני, נציג את זה בצורה הבאה –



וברמת הקוד, אנחנו נעבוד במספר שלבים:

1. בכל מחלקה נגדיר "include" של המחלקה השניה.
2. הבנאי של כל מחלקה יגדיר מופע NULL של המחלקה השניה
3. בזמן ההגדרה האמיתית של המחלקה השניה, נבדוק קודם כל אם המופע ריק, ואז נעביר אותו, ואם לא נעביר את המופע הקיים אליו.

נראה דוגמת קוד פשוטה ונסביר את המימושים השונים –

```
//file A.h
#ifndef A_H
#define A_H
class B;                    // הגדרת מופע של המחלקה אליה אנחנו מקשרים
class A {
public :
 A();
 ~A();
 B* getItsB() const;
 void setItsB(B* p_B);
protected :
 void cleanUpRelations();
 B* itsB;                   // הגדרת משתנה שהוא רפרנס למחלקה אליה אנחנו מקשרים
};
#endif

//file A.cpp
#include "A.h"
#include "B.h"                // את המחלקה אליה אנחנו מקשרים אנחנו מכניסים בקובץ המימוש
A::A() {
  itsB = NULL;                // בבנייה של המחלקה אנחנו מתחילים עם פוינטר ריק
```

```
}
A::~A() {
   cleanUpRelations();             // ההורס מוודא שאנחנו לא משאירים פה פוינטרים באוויר
}
B* A::getItsB() const {
   return itsB;
}
void A::setItsB(B* p_B) {
   itsB = p_B;
}

void A::cleanUpRelations() {
   if(itsB != NULL)
      {
         itsB = NULL;
      }
}
```

עד כאן ראינו איך מחברים מחלקה אחת למחלקה אחרת. וזה נחמד. אבל מה קורה כשאנחנו רוצים לעשות קישור דו כיווני? אמרנו קודם שאנחנו עלולים להתקע במעגל לא ממש יעיל – בכל קובץ של אחת מהמחלקות, אנחנו דורשים לייצר מופע של המחלקה האחרת שהיא עדיין לא נוצרה, ובשביל שהיא תיווצר, אנחנו צריכים לחזור למחלקה הראשונה שלא יכולה להיבנות וחוזר חלילה. בשביל לצאת מהבלאגן הזה, המימוש פה הוא קצת שונה. אנחנו רוצים שהחיבור והניתוק בין המחלקות יהיה סימולטני ויקרה כאחד –

```
//file A.h
#ifndef A_H
#define A_H
class B;
class A {
public :
  A();
  ~A();
  B* getItsB() const;
  void setItsB(B* p_B);
protected :
  void cleanUpRelations();
  B* itsB;
public :
  void __setItsB(B* p_B);          // זה השוני שאנחנו מגדירים פה מחלקות ביניים
  void _setItsB(B* p_B);
  void _clearItsB();
};
#endif


//file A.cpp
#include "A.h"
#include "B.h"
A::A() {
```

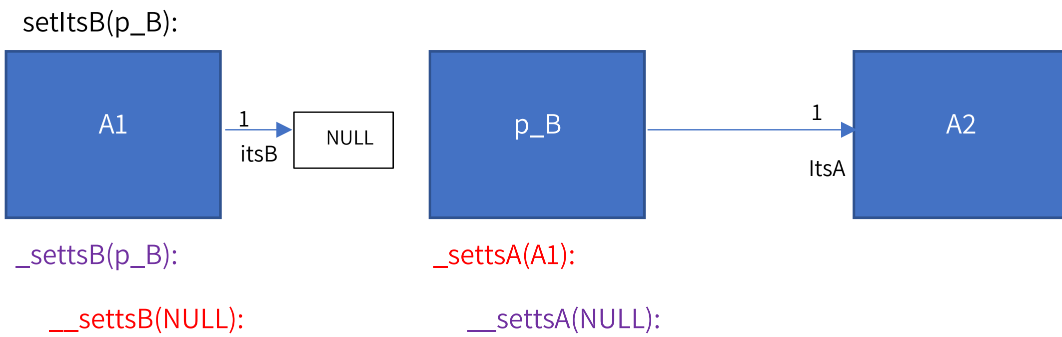
```
  itsB = NULL;
}
A::~A() {
  cleanUpRelations();
}
B* A::getItsB() const {
  return itsB;
}
// עד כאן הכל היה בדיוק כמו בקישור יחיד, עכשיו אנחנו מתחילים להתעסק עם הקישור הדו-כיווני

void A::setItsB(B* p_B) {
  if(p_B != NULL)             // במקרה ויש לנו כבר מחלקה שאנחנו מצביעים אליה, והיא לא סתם אוויר
    {
      p_B->_setItsA(this);    // אנחנו מגדירים את הקישור הדו כיווני על ידי המלקה הנוכחית
    }
  _setItsB(p_B);              // כאן אנחנו שולחים לאחת הפונקציות שהוספנו בדו כיווני
}
void A::cleanUpRelations() {
  if(itsB != NULL)
    {
      A* p_A = itsB->getItsA();
      if(p_A != NULL)
        {
          itsB->__setItsA(NULL);
        }
      itsB = NULL;
    }
}
void A::__setItsB(B* p_B) {   // פה זה ההגדרה הסופית של הרפרנס המקומי
  itsB = p_B;
}
void A::_setItsB(B* p_B) {    // אם היה כבר רפרנס בתוך המחלקה הפנימית אנחנו מוחקים אותו בשביל שנוכל
                              // להכניס בו את הרפרנס החדש
  if(itsB != NULL)
    {
      itsB->__setItsA(NULL);
    }
  __setItsB(p_B);
}
void A::_clearItsB() {
  itsB = NULL;
}
```

כמה נקודות שחשוב לשים לב אליהן, בשביל להבין את הבלאגן של השליחות הכפולות האלה –

- אנחנו מדברים על קשר של אחד לאחד. לכן, אם האחת המחלקות שאנחנו מנסים לקשר יש כבר קישור למופע של המחלקה האחרת, אנחנו ניאלץ לנתק אותו לפני שנמשיך. מהסיבה הזאת אנחנו קודם כל ניכנס פנימה ונשם שם NULL.

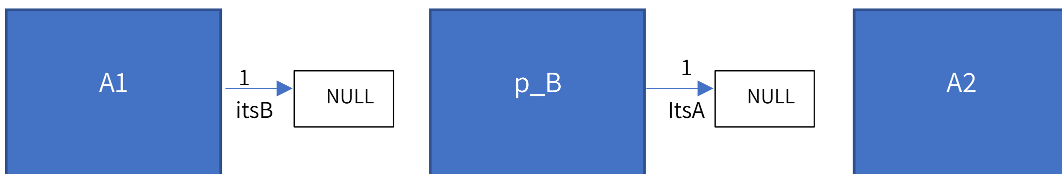
לצורך העניין, נניח שזה המצב הראשוני שלנו, לפחות לאחר הבנייה הראשונית של A1 שאנחנו מגדירים את ה-itsB להיות NULL:

ועכשיו אנחנו קוראים לפונקציה setItsB עם רפרנס ל-B שיש לנו, עכשיו מאחר ו-B אינו ריק, אנחנו חייבים לטפל גם במופע שקיים אצלו, לכן כל מה שאנחנו מבצעים עובד בשני הצדדים בו זמנית:

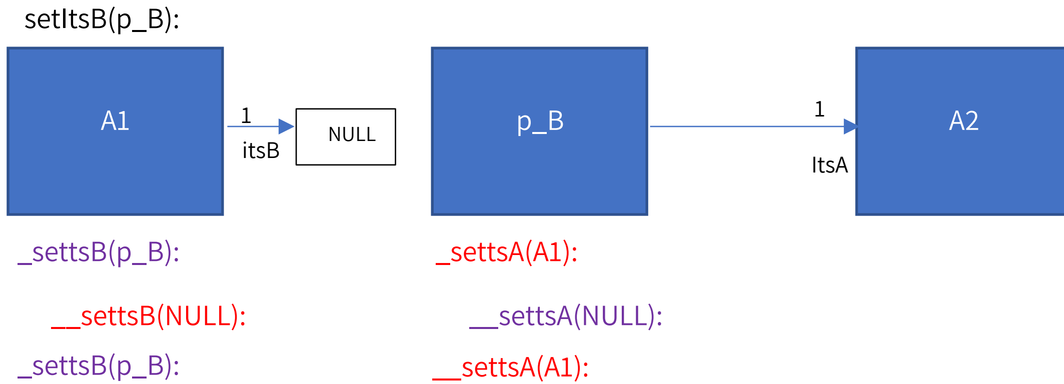


מה קרה כאן?

המחלקה A1 הפעילה את SetItsB, ומאחר ש-p_B לא היה ריק, הפעלנו במקביל גם את p_B::_setItsA עם המופע של A1, וגם את A1::_setItsB עם p_B, כלומר נכנסנו לרמה אחת פנימית יותר בהשמה.
לצורך הנוחות גם צבעתי כל אחד בצבע אחר כדי לראות מי הפעיל מה.
עכשיו, נסתכל על A1:: _settsB(p_B) הוא רואה ש-p_B אינו ריק, כלומר אנחנו צריכים לנתק את הקישור שקיים בו. לכן אנחנו ניגש אליו ונשלח בתוכו מחיקה לרפרנס שקיים בו, כלומר p_B::__setItsA(NULL).
במקביל גם p_B התחיל לעבוד בצורה עצמאית לגמרי, ונכנס לפונקציה _setItsA אליה הוא שלח את הרפרנס של A1. הוא שוב מוודא שלא מדובר בפוינטר ריק, למרות שאנחנו יודעים שלא, אבל יש פה אמצעי זהירות כפול. וגם הוא שולח אותו למחיקה דומה עם A1::__setItsB(NULL).
בעצם המצב שלנו אחרי כל השלבים האלה ייראה כך:



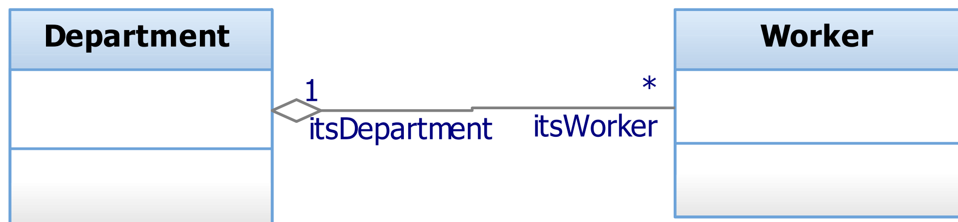
ואז נשארה לנו פקודה אחת בכל אחד מהמופעים שיקשר אותם אחד לשני:



כאשר המימוש של __setItsX הוא פשוט לקחת את הרפרנס שנשלח בכל אחת מהפונקציות ולהגדיר את הפוינטר בתוך המחלקה ישירות אליו, ועכשיו המצב נראה כך:

וכולם שמחים ומאושרים. חוץ מ-A2 שאףאחד כבר לא מצביע עליו.

Aggregation

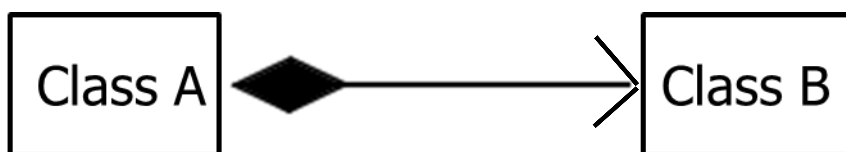


Aggregation הוא סוג מסוים של קישור בין מחלקות, כאשר אנחנו מגדירים באופן יותר ברור את המחלקה המקושרת. באופן ברור ניתן לראות את הדוגמא שהביאו – יכולים להיות מספר אינסופי של מופעים של עובד, אך לכל אחד מהם מקושר רק מופע אחד ויחיד של מחלקה (Department). כדאי לשים לב, שמדובר פה בקשר שמכונה "רבים ליחיד". כלומר, אם יש מאתיים סטודנטים במחלקת מחשבים, ועוד 200 בתעשייה וניהול, אז נחזיק רק שתי מחלקות של department ואליהם נקשר את כל הסטודנטים השונים.

כדאי לשים לב, שלא מדובר אבל בקשר מוכרח - יכול להיות שהמחלקה B אליה מצביעים תימחק, ואז כל מי שהצביע אליו, פשוט יצביע ל-NULL, ומבחינתנו אין בזה בעיה.

מבחינת הקוד מדובר בדיוק באותו קוד של Association, ולכן אין מה להתעכב עליו.

Composition

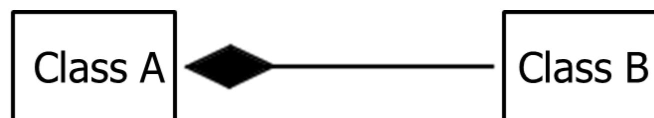


רמה נוספת של קישור מעבר לאגרגציה, בה אנחנו מגדירים קשר מוכרח בין המחלקות. כאשר אנחנו נגדיר מחלקה מסוג A אנחנו נהיה חייבים ליצור במקביל גם את המחלקה B שתהיה מוכלת בתוכו באופן שלם, כך שאם המחלקה A תימחק, ביחד איתה יימחקו כל המחלקות שנוצרו במיוחד בשבילה.

דוגמאות לאירועים בהם נשתמש בקומפוזיציה הוא יצירה של מחלקה שמורכבת ממספר רכיבים שונים שאין ביניהם קשר או סיבה שיהיו במחלקה בודדת. למשל – נראה בתרגיל שבשביל ליצור טלפון, אנחנו נשתמש במחלקה של חייגן, צג וכו' ורק כאשר ניצור את כולם, נוכל לומר שהמחלקה של הטלפון בנויה ומוכנה.

ברמת הקוד, מה שניצור ייראה כך:

```
//file A.h
#ifndef A_H
#define A_H
#include "B.h"              // הכללת המחלקה אותה אנחנו מרכיבים
class A {
public :
  A();
  ~A();
  B* getItsB() const;
protected :
  B itsB;                   // כאן אנחנו ניצור מופע בודד של המחלקה בתור משתנה פנימי
};
#endif
// file A.cpp
#include "A.h"
A::A() {
}
A::~A() {
}
B* A::getItsB() const {
   return (B*) &itsB;       // כאן אנחנו נחזיר רפרנס של המחלקה שקיימת אצלנו
}
```



גם כאן, כמו ב-Association יש לנו אפשרות לעשות את זה באופן דו כיווני, רק שפה יש משהו מאוד שונה – כאן תהיה לנו בעצם מחלקה ראשית, שתעבוד בצורה פשוטה, פחות או יותר כמו מימוש חד כיווני, והמחלקה השניה תצטרך לעשות את כל העבודה של הקישור הכפול. הדרך הטובה ביותר היא לראות את הקוד של שתי המחלקות אחד מול השני ולעמוד על ההבדלים ביניהם:

```
//file B.h
#ifndef B_H
#define B_H
class A;
class B {
public :
B();
~B();
A* getItsA() const;
void setItsA(A* p_A);
protected :
void cleanUpRelations();
A* itsA;
public :
void __setItsA(A* p_A);
void _setItsA(A* p_A);
void _clearItsA();
};
#endif
```

```
//file A.h
#ifndef A_H
#define A_H
#include "B.h"
class A {
public :
A();
~A();
B* getItsB() const;
protected :
void initRelations();
B itsB;
};
#endif
```

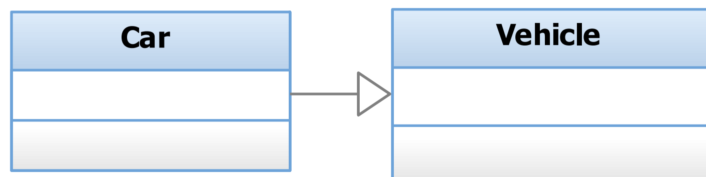
ההבדל הראשון והפשוט ביותר הוא צורת ההכללה של כל מחלקה באחרת. המחלקה A שהיא המקור לקשירה (לפחות לפי התרשים), עושה include למחלקה B, בדיוק כמו שאנחנו עושים ב-Association. לעומתו, המחלקה B מקבלת מופע ריק של A, וככה לגשת לתוך הפונקציונליות של המחלקה.

קובץ ה-cpp של המחלקות ייראה שונה לגמרי. קודם כל אנחנו שוב נכליל את שתי המחלקות האחת בתוך השניה (אין לנו חשש ללולאה אינסופית). ואם נסתכל על המימוש הפנימי של כל דבר נראה הבדלים תהומיים.
נתחיל דווקא במחלקה B – הבנאי שלו קודם כל מגדיר את ה-A שלו להיות NULL, מה שבטוח.
אחרי זה, כשנבקש להגדיר את itsA בערך אמיתי יותר, אנחנו ניכנס בצורה "בטוחה יותר" דרך _setItsA כמו שראינו קודם, ורק ברמה השלישית אנחנו נגדיר אותו לגמרי
ועכשיו המחלקה A – הבנאי שלו פשוט קורא לפונקציה שמאתחלת את הקשרים של ה-B שבתוכו.0, אבל הוא מתחיל את זה מהרמה השניה, ולא העליונה ביותר.

```cpp
//file B.cpp
#include "B.h"
#include "A.h"
B::B() {
   itsA = NULL;
}
B::~B() {
   cleanUpRelations();
}
A* B::getItsA() const {
   return itsA;
}
void B::setItsA(A* p_A) {
   _setItsA(p_A);
}
void B::cleanUpRelations() {
   if(itsA != NULL)
      {
         itsA = NULL;
      }
}
void B::__setItsA(A* p_A) {
   itsA = p_A;
}
void B::_setItsA(A* p_A) {
   __setItsA(p_A);
}
void B::_clearItsA() {
   itsA = NULL;
}
```

```cpp
// file A.cpp
#include "A.h"
A::A() {
   initRelations();
}
A::~A() {
}
B* A::getItsB() const {
  return (B*) &itsB;
}
void A::initRelations() {
itsB._setItsA(this);
}
```

Generalization (ירושה)



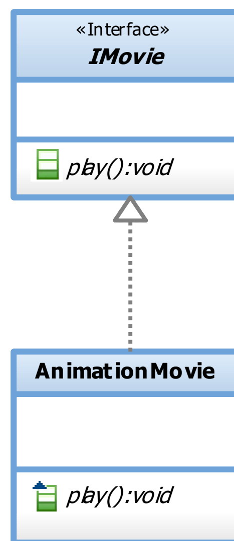
זה דבר שאנחנו מכירים ולא אמור להיות לנו בעיה להבין, כאשר אנחנו מגדירים ירושה, למשל A יורש מ-B, אנחנו למעשה אומרים שהמחלקה A היא סוג של B, עם שינויים כלשהם. נראה בהמשך, מה היא עילה להפריד בין מחלקות ולומר שהחל מנקודה זו, המחלקות האלו צריכות להפרד לשתיים שונות.

הדוגמא שלפנינו גם היא מובנת וחזרנו עליה יותר מפעם אחת – מכונית היא סוג של כלי רכב. גם אופניים הוא סוג של כלי רכב, אך ההבדלים בי שתי המחלקות האלה הוא מספיק רחב בשביל לקום ולומר שאנחנו נצטרך ליצור את מחלקת הרכבים כמשהו חדש.

הירושה תסומן בקוד על ידי include לקובץ ה-header של מחלקת-האב וברגע הגדרת המחלקה עם שם מחלקת האב לאחר נקודותיים –

```
//file Car.h
#ifndef Car_H
#define Car _H
#include "Vehicle.h"
class Car: public Vehicle {
public :
Car ();
~ Car ();
};
#endif
```

Realization (מימוש)



מימוש ממשק, מופיע עם ראש חץ דומה לירושה, אך החץ עצמו מגיע מקווקו, על מנת לסמן את השוני. כבר אמרנו קודם על מימוש ה"ממשק" שנראה דומה לגמרי למחלקה רגילה מלבד הכותרת השונה. כאשר אנחנו "יורשים" ממשק, או יותר נכון, מממשים אותו, אנחנו נראה בדיוק את אותן פונקציות שמופיעות בממשק, שיופיעו גם במחלקה המממשת (שזה בעצם כל הרעיון מאחורה).

בגדול, יצירת ממשק היא מעין חוזה שאנחנו מכריחים את כל מי שלוקח עליו לממש אותו, שימלא אותו במלואו. כלומר כל פעולה שמוכרזת באינטרפייס חייבת להיות ממומשת בכל מי שמכריז עליו.

Composite

בגדול ה-Sequence Diagram מראה לנו את המחלקות השונות ואת ההודעות העוברות בין המחלקות, כלומר אם מחלקה אחת קוראת לפונקציה שמתבצעת בה או במחלקה אחת, אנחנו נוכל לראות את סדר הקריאות. באופן כזה נוכל לראות תרחישים שונים שעלולים לקרות תוך כדי ריצת התוכנית -עבור קלט מסוים יהיה לנו תרשים מסוים, ועבור קלט אחר, נקבל תרשים שונה לגמרי.

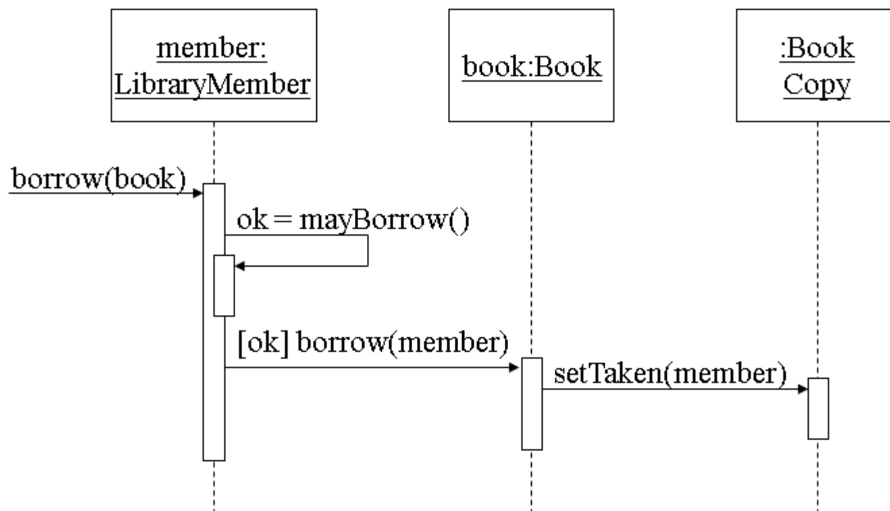
Sequence Diagrams (תרשימי רצף)

בכל פעם שנעבור על ה-UML, ונעבור עליהם עוד כמה פעמים במשך התואר, אנחנו תמיד נפריד בין התרשימים הסטטיים לדינאמיים. מה הכוונה? כל התרשימים שראינו עד עכשיו מתארים לנו מצב סטטי – קפוא. אנחנו יכולים לתאר ממשקים, ירושות, הכלות ועוד מגוון סוגי קשרים, אך לא נדע לומר באמת מה קורה כאשר התוכנית רצה. האם אנחנו בכלל מפעילים את כל מה שבנינו, או שסתם בנינו אלפי מחלקות ובסוף פשוט הדפסנו למסך "Hello World".

בשביל לדעת את כל זה ועוד, אנחנו יוצרים תרשימי רצף – Sequence Diagram.

תרשימי הרצף יתארו לנו מה קורה החל מהרגע שאנחנו מפעילים את התוכנית (GMR) ואת כל שלבי הביניים של העבודה. כמובן שלעבודה הדינאמית של תוכנית יש לא מעט משתנים, וזה הדבר המיוחד בתרשימים הדינאמיים. יש לנו אפשרות לראות עבור כל תרחיש מה יקרה ומה התוכנית תוציא לבסוף.

בתור דוגמה ראשונית נראה את ה-Sequence Diagram הבא וננסה לקרוא מה קורה שם –



שלושת המלבנים העליונים הם מחלקות אותם הגדרנו. יש לנו מופע של מנוי בספריה בשם member שהוא מופע של המחלקה LibraryMember, מופע של ספר (מסוים, כלשהו), ומופע נוסף של עותק של הספר. כלומר, אם יש לנו בספריה שלושה עותקים של "ההוביט", אז "הוביט" (book לפי הדיאגרמה הנוכחית) יירש מספר, ולו יהיה עותק אותו ניתן להשאיל.

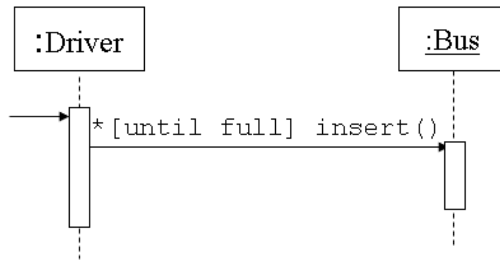
ה-Sequence Diagram למעשה מבוטא בצורה של גרף/ציר זמן רגיל שמתחיל משמאל לימין ומייצג אובייקטים שונים על ציר ה-X, כאשר הוא מתחיל מלמעלה, ואז יורד מטה עם התקדמות הפעולה על ציר ה-Y.

במקרה שלנו, member רוצה לשאול ספר. הוא שולח הודעה הודע שברצונו לעשות זאת. קודם כל, נעשית בדיקה (עדיין במתחם המנוי) שבכלל מותר לו לשאול ספרים. במידה וכן, ההודעה/בקשה לשאילת הספר עוברת הלאה בצירוף המידע שהמנוי רשאי לשאול את שהוא רוצה. ברגע שאנחנו עברנו את השלבים האלה, אנחנו שולחים הודעה לעותק שנלקח שהוא כבר לא קיים במלאי הנוכחי.

הערה: הקו המקווקו היורד מכל אובייקט מבטא בעצם ציר זמן פרטי של כל אובייקט. ברגע שהאובייקט פעיל המלבן מתחיל לעלות לסמן את החיות שלו ע שהוא נסגר.

סוגי הודעות העוברות בין האובייקטים

- הודעה רגילה – יצויין עם חץ רגיל ועם שם הפעולה
- ערך מוחזר – קו מקווקו עם הערך החוזר
- יצירת מופע חדש – יצוייר כשליחת הודעה עם חץ רגיל, ועליו תופיע המילה <<create>>.
- הריסת מופע – שליחה של <<destroy>> שלאחריו קו החיים יתנתק.
- בקרת מידע – הודעה שעוברת רק עם תנאי מסוים, תעבור כמובן רק כאשר התנאי מתקיים, אך בשביל לבטא את זה, יופיע לנו כמו שראינו למעלה [condition]message(value).
- איטרציה – אם יש הודעה שחוזרת מספר פעמים, כמו לולאות למיניהם, הוא יופיע באופן הבא message [expression]*. למשל עבור לולאת while יופיע לנו השרטוט הבא –



-

שמבטא שעד שהאוטובוס לא יהיה מלא, הנהג ימשיך להכניס נוסעים.

תרשימי מצבים (Statecharts)

כמו שתיארנו קודם לגבי ה-sequence diagram, גם כאן אנחנו מדברים על תיאור מצב של אובייקט דינאמי. אך לעומת תרשימי הרצף, המראים לנו מה מבוצע, אך אין לנו שום לוגיקה המקושרת למה שקורה. ברגע שאנחנו מכניסים גם סטייטצ'ארט, אנחנו נדבר על לוגיקה, הלוגיקה תתאר לנו את המחלקה ואת כל המצבים השונים בה היא יכולה להיות.

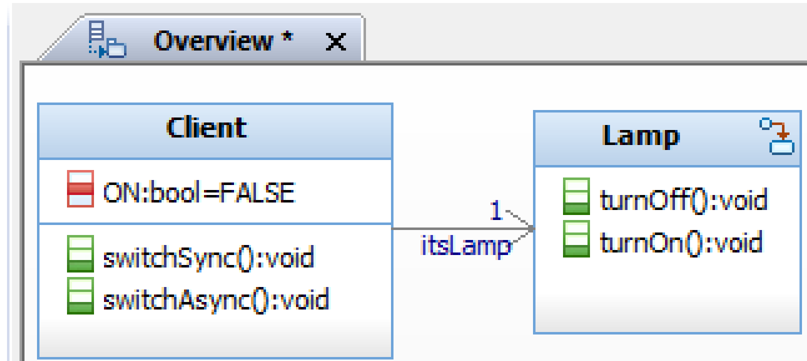
כאשר אנחנו מדברים על תרשימי המצבים השונים, אנחנו צריכים להבדיל בין שני סוגי מצבים שנים – סינכרוני ואסינכרוני.

מה ההבדל ביניהם?

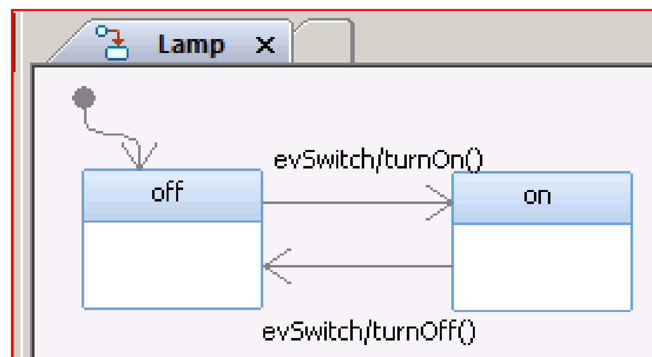
מצב סינכרוני, הוא המצב בו אנחנו עובדים לפי סדר מסוים. אם יש לנו פונקציה שקוראת לפונקציה חדשה, אזי הפונקציה הנוכחית עוצרת עד שהפונקציה הנקראת תסיים את פעולתה ותחזיר או לא תחזיר ערכים. כל עוד הפונקציה החדשה לא תסיים כולם עוצרים ומחכים לו, בדיוק כשם שיכול להיות שיש לנו עוד פונקציית אב שבכלל לא ידענו שהיא קיימת.

במצב אסינכרוני, אנחנו מדברים על "הודעות" שעוברות בין הפונקציות השונות ותגובות לאירועים. אם פונקציה שלחה הודעה למקום מסוים, היא לא תחכה לקבל תשובה, אלא תסמוך על זה שהתשובה תגיע, וגם אם לא, אז שיהיה איזה מנגנון שיידע לטפל גם עם מחסור בתשובות. אבל מבחינת פונקציית העל זה לא באמת משנה אם נעשתה פעולה, אלא מבחינתה היא התבקשה לשלוח הודעה, והיא שלחה וזהו.

בדרך כלל כאשר מדברים על סינכרוניות ואסינכרוניות אנחנו מגדירים לקוח-שרת. מי שמבקש פעולה, מי שמבצע. הדוגמא במצדת מתייחסת לשרת בתור מנורה, והשרטוט נראה כך:

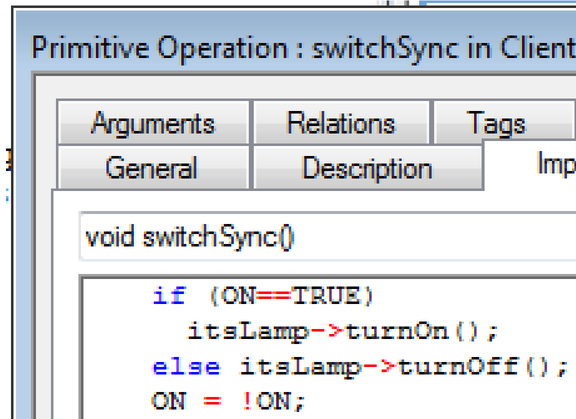


האייקון מימין לשם המחלקה “Lamp” מציין לנו כי מדובר במחלקה תגובתית, כלומר מחלקה שמוצמד אליה סטייטצ'ארט מסוים. והנה התרשים שלו:



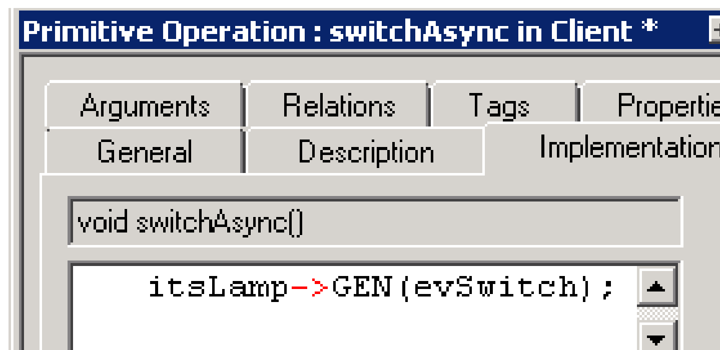
ניתן להבין באופן די פשוט שאם נתייחס לשני המצבים on\off אז יש לנו מעבר עם ההדלקה בין שני המצבים. עכשיו, מחלקת הלקוח Client מחזיקה אצלה שתי פונקציות שמטפלות במתג (לצורך העניין מתג של מנורה), אחת סינכרונית והשניה – אסינכרונית, יפה.

נסתכל קודם על המימוש הסינכרוני כי הוא יותר פשוט –



המצבים של המנורה נמצאים כרגע באחד משני מצבים דלוק/כבוי. והוא נשאר שם ללא כל שינוי, עד אשר פתאום מישהו ילחץ על המתג. ברגע שהמתג ילחץ, נבדוק באיזה מצב אנחנו, נעבור למצב השני, ואז נחכה שם עד הקריאה הבאה.

לעומת זאת המצב האסינכרוני, מטפל בהכל בצורה פשוטה יותר:

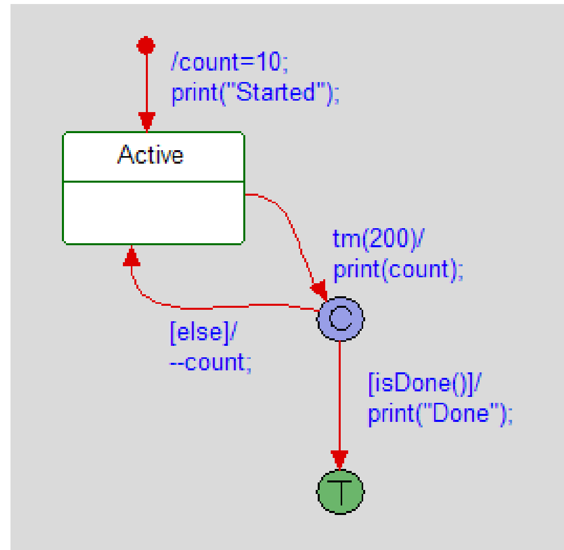


אנחנו מגדירים אירוע על ידי הקריאה GEN, וברגע שקורה משהו אנחנו מפעילים את האירוע evSwitch. אפשר לשים לב שבתרשים, אירוע המתג מעביר אותנו בין שני המצבים. כלומר, כל פעם שיופעל לנו אירוע מתג, אנחנו נעבור מצב.

בסופו של דבר, אנחנו נקבל את אותו הדבר - כך או כך הנורה תדלק, אבל הפונקציה הסינכרונית תוכל לעבוד רק כתלות בשאר הפונקציות שעובדות במחסנית, והאסינכרונית תפעיל ישר אירוע.

מחלקה תגובתית

כמו שכבר ציינו, יש לנו את האייקון הקטן המסמל לנו "מחלקה תגובתית", כלומר מחלקה שמקושר אליה תרשים מצבים. יש לציין מספר מרכיבים שונים של הסטייטצ'ארט שנשתמש בהם לא מעט. נראה דוגמה של תרשים פשוט מתרגילי הבית ונפרט כל אחד מהמרכיבים –



<u>מצבים</u> – כמו שיש לנו שרטוט למחלקות שהם המרכיב העיקרי בתרשים רגיל, גם כאן יש לנו את הריבועים שמציינים את המצבים השונים, כאשר בראש כל מלבן יהיה רשום שם המצב.

<u>מעברים</u> – החיצים המבטאים את הדרך בין שני מצבים שונים. ברוב המקרים יהיה מדובר במעבר למצב חדש, אך לא מן הנמנע שהוא יעבור לאותו מצב בעצמו, בדיוק כמו בשרטוט שלנו.

<u>מעבר התחלתי</u> – מעבר ברירת המחדל של כל התרשים. ברגע שהתרשים יתחיל לעבוד, הוא לא יתחיל כל פעם איפה שבא לו, אלא הוא מתחיל רק בזב שבראש החץ שלו יש עיגול.

<u>תנאים</u> – המעברים השונים בין המצבים קורים בעזרת תנאים ואירועים מסוימים אותם אנחנו מפרטים על המעבר עצמו. התנאים מחולקים לשלושה מרכיבים trigger[Guard]/action.

- Trigger – אירוע שגורם לתנאי להתחיל לעבוד (לחיצה על כפתור, מעבר זמן , זיהוי חיישן וכדו').
- Guard – תנאי נוסף שאנחנו מגדירים במידה ואנחנו רוצים לוודא שאירוע מסוים כבר קרה , או שאנחנו רוצים לוודא שמשתנה כלשהו מחזיק איזה ערך מסוים או דברים דומים – התנאי/שומר יסומן בתוך סוגריים מרובעות (במקרה שלנו אפשר לראות את IsDone, שאנחנו לא יודעים מה הוא ושה, אבל אפשר להבין שיש פה בדיקת סיום למשהו.
- Action הפעולה אותה צריך לעשות. להפעיל, להדפיס וכו'.

מעבר בין המצבים לא חייב להכיל את כל המרכיבים האלה, ראינו למשך את המעבר ההתחלתי שאין לו אף אחד מהמרכיבים (למרות שאפשר להגדיר לו Action, אבל לא מגדירים לו שום תנאי). מעבר שמכיל אירוע נקרא מעבר רגיל Regular Transition, וניתן גם להגדיר מעבר ללא שום פעולה שייקרא Null Transition.

הערה: טריגר שנשתמש בו הרבה הוא Tm שבתוך הסוגריים נגדיר כמה מילי שניות התוכנית צריכה לחכות לפני המעבר הבא.

<u>קונקטור</u> – קישור בין מעברים. כדאי לשים לב – קונקטור הוא לא מצב! הוא, כשמו, רק מקשר.

יש שני סוגי קונקטורים עיקריים שאנחנו מדברים עליהם –

קונקטור תנאי – מופיע עם C בתוך העיגול (כמו זה של כל הזכויות שמורות), שנועד לאחד מספר מעברים מיציאה מסוימת. במידה ויש לנו מעברים שהם בעצם מופעים שונים של אותו תנאי, אז אפשר לאחד הכל לקונקטור, ואז לבדוק ב-guard איזה תנאי מתקיים.

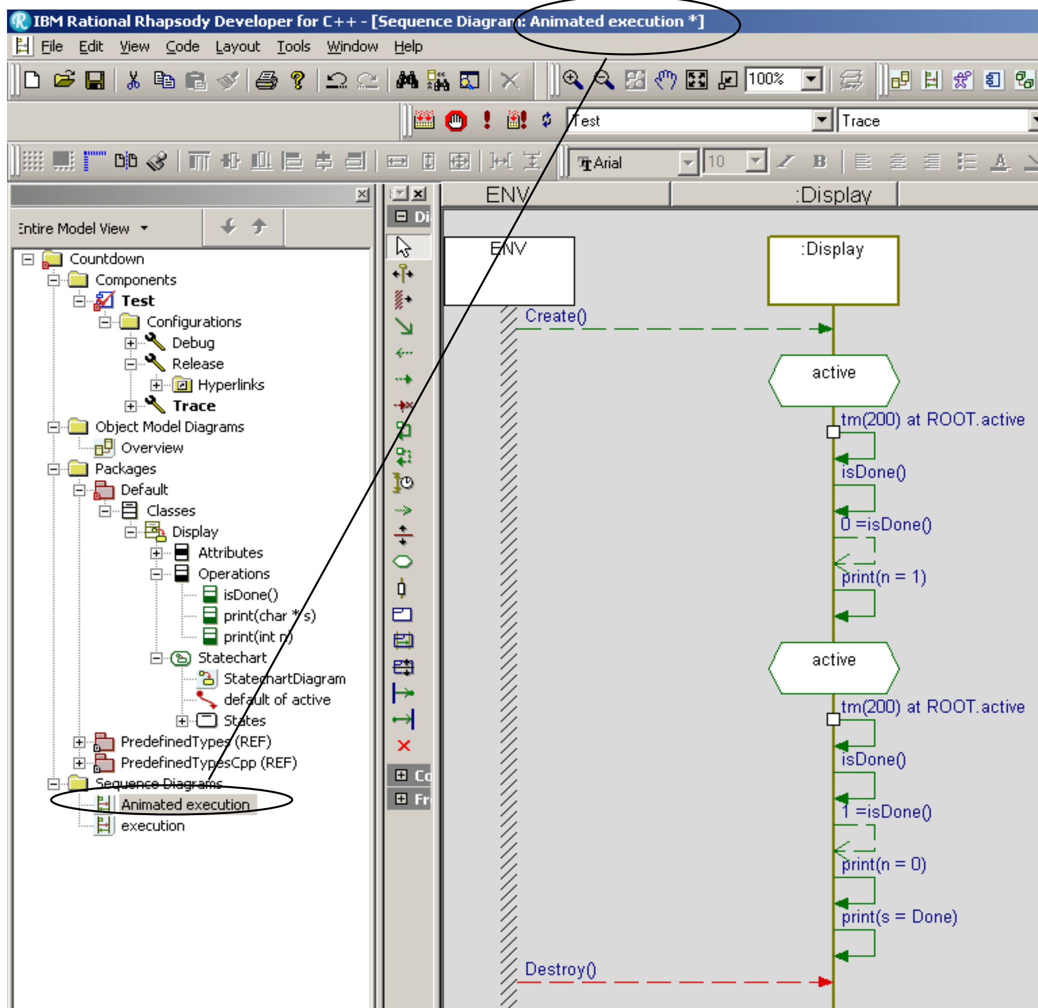
קונקטור סיום – פשוט מבטא סיום פעולה.נראה כמו עיגול עם איקס בתוכו.

חשוב לשים לב: כשאנחנו מגדירים קונקטור בתרשים, אנחנו לא קובעים סדר בדיקה, אלא רק בקוד סדר הבדיקה ייקבע. כך שאם יש סדר שהוא חשוב ואנחנו רוצים לבדוק דברים אחד אחרי השני, אז יש לוודא בקוד שזה מסודר נכון.

כמו כן, כמובן שלשל תרשים שמכיל קונקטור תנאי, יש תרשים מקביל ללא הקונקטור ר עם הרבה יותר מצבים.

מה סדר הפעולות כשמגיעים לקונקטור? ניקח למשל את ה-Statechart מלמעלה, ונשאל הפעולה או בדיקת התנאי?

כמובן שהתשובה מונח בעצם השאלה – אנחנו בודקים אם מתקיים לנו תנאי מסוים, אז אנחנו נעבור לכל ההסתעפויות ואם נמצא שם את אחד התנאים שמתאימים לנו, רק אז נעשה את הפעולה הנדרשת ממנו (המעבר המביא אותנו לקונקטור), ורק אחרי זה נבדוק מה הפעולה המתאימה מצד המעבר החדש בעצמו.



ביצוע הסטייטצ'ארט ייראה ב-sequence diagram שונה לגמרי. כל אירוע ומעבר בין המצבים יסומן עם חץ שעובר למקום הנכון ובודק שם את התנאים הנדרשים.

ירושת תרשימי מצבים

כאשר יצרנו מחלקה תגובתית, ועכשיו יש מחלקה אחרת שיורשת ממנה, היא יורשת גם את תרשים המצבים. דיברנו על כך שירושה צריכה להתנהג במחלקה-הבן באותו אופן כמו במחלקת-האב. מה זה אומר מבחינת הסטייטצ'ארטים השונים?

על מנת לרשת את תרשים המצבים יש מספר דברים שאנחנו **חייבים** שיעברו הלאה, מספר דברים ש**מותר** לנו שיעברו הלאה, ומספר אלמנטים ש**אסור** שיעברו.

ברגע שנעביר תרשים אנחנו חייבים להמשיך ולשמור על המהות שלו. מהי המהות? קודם כל, המצבים השונים והמעברים בין המצבים. כשאנחנו מדברים על המצב שצריך להישאר כמו שהוא אנחנו מדברים אפילו על שם המצב, ובדומה לזה גם לגבי המעברים – המאפיין החשוב ביותר במעבר הוא הטריגר ולכן אותו לא נשנה. כל דבר אחר נוכל להוסיף ולשנות באופן חופשי.

SOLID

עכשיו אנחנו בחלק השני, בוא אנחנו מדברים על עקרונות תיכון תכנה, ובראשם כמובן SOLID. אבל לפני שנבין מה זה בכלל תיכון תכנה טוב, כדאי להבין מה זה תיכון תכנה לא טוב. יש דברים שברור לנו שהם לא טובים ולא יעילים. אם אנחנו רוצים לעשות חישוב וכותבים אותו בזמן ריצה גבוה מידי, ברור לכולם שזה לא יעיל ולא טוב. אבל אם כתבנו את התוכנית הכי יעילה ברמת זמן ריצה זה מספיק? אם אנחנו שואלים, אז כנראה שלא.

יש "מושג" שנקרא TNTWI-WHDI, או "טינטי ווהדי" בעברית. זה לא באמת מושג, כמו שזה ראשי תיבות של “That’s Not The Way I Would Have Done It”– לא ככה הייתי עושה את זה. אבל כמובן, שאמירה כזאת היא מאוד תלוית הקשר ואנשים. יש אנשים שאוהבים לכתוב תנאי If-else בצורה רגילה, ויש אנשים שאוהבים לראות את השמים בוערים וכותבים את זה בצורה ה"קצרה" והנוראית x?2;1 או איך שזה.

הבעיה היא שאנחנו רוצים לתת מדד סובייקטיבי שיוכל להגיד לנו האם מה שכתבנו הוא אכן טוב. לא רק "הקוד נראה טוב" או ברים דומים. עבור דברים אלו, נכתבו עקרונות ה-SOLID, שעמידה בהם תוכל לנתח לנו את הנכונות של מה שכתבנו.

פרמטרי איכות לקוד

בסופו של דבר, לפני שנגיע לעקרונות עצמם, אנחנו נחפש פרמטרים שנוכל לקבוע לפיהם את האיכות אליה אנחנו שואפים או שואפים שלא תהיה. נתחיל עם הפרמטרים הרעים:

- **קשיחות** (Rigidity) – כשנכתוב קוד, נצא מנקודת הנחה שאנחנו תמיד נשנה אותו. לטוב או לרע, תמיד יהיו שינויים ונגיעות שונות בתוכנית. קוד קשוח הוא קוד שלא נותן לשנות אותו. אם נרצה לשנות משהו קטן בקוד, נצטרך לשנות את כל המופעים השונים וכל מיני מקומות לא קשורים. זה לא טוב.
- **שבירות** (Fragility) – שינוי במקום מסויים, יגרום לליקויים במקומות בלתי צפויים. כדאי לשים לב שזה נשמע דומה לקשיחות, אבל המהות פה היא שונה – זה שאנחנו צריכים לעשות שינויים במקומות שונים, זה מילא. אבל אם יש לנו כל מיני תלויות מסתוריות בין הפונקציות והמודולים השונים, אנחנו עלולים לשנות פונקציה קטנה שנמצאת בקובץ מסוים, ופתאום במקום אחר לגמרי דברים יקרו לא כמו שתכננו.
- **אי ניידות** (Immobility) – קשה להעביר מיישום מסוים ליישום אחר בגלל שהוא תפור עבור היישום הראשון. אמנם הכתיבה שלנו בקוד היא עבור המשימה שאנחנו מבצעים כרגע. אבל אם בעוד חודשיים נצטרך לבצע את אותה משימה, רק עם שינוי קטן. אם נכתוב את הקוד בצורה לא נכונה אנחנו לא נוכל לעשות את זה, כי הפונקציה שכתבנו מוציאה לנו פלט מאוד מדויק.

הפרמטרים הטובי לעומת זאת, הם כאלה שעוזרים לנו לתחזק את התוכנה, ולבדוק את המודולים השונים בה בצורה קלה ופשוטה:

- **גמישות (Flexibility) –** היכולת להוסיף ולשנות חלקים בקוד בצורה מקומית וללא תופעות לוואי של שינויים מרחיקי לכת בקוד.
- **חסינות** (Robust) – חסינות מגיעה כנד השבירות – נגענו במשהו ושינינו אותו, ואז מעבר לתיקונים המקומיים לא עשינו הרבה.
- **שימוש חוזר** (Reusable) – היכולת לקחת קטעי קוד ולהשתמש בהם במקומות נוספים ללא שום שינוי והתאמות בקוד.

בגדול, אנחנו טוענים, כי אחת החלות הרועות שמתחיל את כל הבלאגן בקוד, הוא השימוש בתלויות לא-ראויות שגורמים לתוכנה ללכת ולהדרדר. ככל שהתוכנה מלאה ביותר תלויות שכאלה, כך יותר קשה גם לתחזק את התוכנה לטווח ארוך. הפיתרון הוא לנהל את התלויות בצורה שרמות מסוימות לא יפריעו לרמות האחרות וכו'.

תכנות מונחה עצמים מכיל עקרונות כתיבה שיצמצמו לנו את התלויות למינימום ההכרחי ויוביל אותנו לתיכון מוצלח. מה נגדיר כתיכון מוצלח? טוב ששאלתם!

- **Maintainablity** ניתן לתחזוקה לאורך זמן. אין לנו אלמנטים של שבירות וקשיחות, כך שאם נרצה לחזק את התכנה נוכל לעשות את זה בצורה פשוטה ובטוחה.
- **Testability** – ניתן לבדיקה. גם זה נובע מ"הפרדת הרשויות" האמורה – אם אנחנו עובדים על מודול ספציפי ונרצה לבדוק אותו, נוכל לעשות את זה מבלי להטריד את כל התוכנה.
- **Flexibility and Extensibility** – גמיש וניתן להרחבה.
- **Parallel development** – מאפשר עבודה במקביל – אם תכנה בנויה היטב, שני מתכנתים יכולים לעבוד על חלקים שונים של התוכנה במקביל, ואז כל מה שיישאר להם הוא לאחד את מה שהם כתבו ולקוות שלא יהיו התנגשויות.
- **Loose coupling** – צימוד רופף. מכיל מעט תלויות בין המודולים השונים, ביחס הכללי בין המחלקות אנחנו נשאף לשחרר כמה שאפשר תלויות מבין החלקים השונים.

לעיצוב גרוע יש כמה סימנים שמרמזים על בעיה בעיצוב, כאשר הסימנים מרכזיים והבולאים שבהם מכונים "Code Smells" – קוד מסריח. נדבר על מספר דוגמאות של סרחונות כאלה –

אם למשל אנחנו נתקלים בפיסת קוד שחוזרת על עצמה מספר פעמים, אנחנו יכולים לדעת בוודאות שמשהו פה לא נעשה נכון. גם אם מדובר בשינוי קטן של פרמטרים בכל פעם, יהיה יותר נכון לעשות את זה בצורה שתהיה לנו רק פונקציה אחת ורק הפרמטרים השונים ישתנו מבין השליחות בחלקים האחרים.

אפשרות נוספת היא להיתקל במה שנקרא "Shotgun Surgery" , בהשאלה זה ביטוי לסוג של שבירות שניתוח קטן בקוד גורם לכל כך הרבה בעיות, כך שבסוף יוצא שיש לנו תכנה ש"חוררו" אותה בשוטגאן.

מחלקות או פונקציות גדולות מאוד גם הן עדות למשהו לא סביר. כמובן שיכול להיות שתהיה לנו מחלקה גדולה, בהנחה שהיא מאוד מרכזית בקוד, או חישוב ארוך שנעשה בתוך פונקציה. אבל אם אנחנו נעבוד לפי עקרונות ה-SOLID גם חישוב גדול וארוך, יתקצר למספר פונקציות שונות, מהסיבה הפשוטה שאם נרצה להבין טעות בחישוב לא נצטרך לעבוד על 5000 שורות שהם פשוט בלוק ארוך, אלא נוכל לעבור חלק חלק בחישוב ולראות איפה בדיוק הבעיה.

עקרונות SOLID

כדאי להקדים רק ולומר לגבי העקרונות הללו – חלק מהעקרונות נועדו להעלות דיון וכדאי לשים אותם על השולחן. אך יש לזכור – מדובר קודם כל בעקרונות, לא ב"הלכה למשה סיני". ולכן אין צורך לקחת את העקרונות האלו לרמה הקיצונית ולשרוף את כל מי שלא עומד בהם. כמו כן, הדוגמאות שמובאות לנו כאן הן קטנות ומלאכותיות, כלומר בנויות בצורה שתכריח אותנו להשתמש בעקרונות האלה, ובסוף, הכל תלוי בשיקול דעת של המתכנת, וגם אם הוא מחליט שהוא לא עומד בהם, כל עוד יש לזה הצדקה זה בסדר מבחינתנו.

SRP – Single Responsibility Principle

"כל מחלקה/מודול אמורה להיות אחראית לחלק יחיד מהפונקציונליות שהתכנה מספקת. אחריות זו אמורה להיות מוכמסת (encapsulated) על ידי המחלקה, וכל שירותי המחלקה אמורים להיות מותאמים לאחריות יחידה זו"[3].

Single Responsibility Principle
Just because you *can* doesn't mean you *should*.

עקרון האחריות היחידה הוא מאוד פשוט – הקוד שלנו הוא לא אולר שוויצרי. אנחנו רוצים שלכל מחלקה/פונקציה יהיה תפקיד אחד בלבד ולא מספר תפקידים. אחריות המודול אמורה להיות מוכמסת בתוך הפונקציה (ולצורך הנוחות, בדרך כלל מתוארת בשם המודול), וכל מה שהמחלקה הזאת תעשה קשורה אך ורק לאחריות הזאת.

כדאי לשים לב לשני דגשים מתוך ההגדרה של העיקרון – חלק יחיד – אנחנו מדברים כאמור רק על חלק יחיד מהפונקציונליות – אם אנחנו דואגים לשני דברים כמו למשל: להכין קפה ולהקציף חלב, אנחנו יכולים לזהות פה שני פעילויות בעצם ההגדרה של המודול וכך נוכל לדעת שאנחנו צריכים לעבוד ולהפריד. כל שירותי המחלקה – אנחנו רותמים את כל מה שבתוך המחלקה בשביל לשרת את המטרה העיקרית. כך שיכול להיות שיהיו הרבה פונקציות המקושרות למחלקה, אך כל זאת בתנאי שכל מה שנמצא שם קשור לפונקציונליות הדרושה.

אם אנחנו לא נקיים את העיקרון הזה, אנחנו עלולים להגיע למצב של GodClass, מחלקה אלוהית שיכולה לבצע הכל, כולל לברוא אבן שהיא לא יכולה להרים. דבר כזה הוא כמובן לא טוב, כי אז המחלקה תפתח תודעה ותשלח את הטרמינייטור להרוג את כולנו וחבל.

דוגמא פשוטה אך מפורטת קצת יותר ממה שאמרנו קודם, מובאת במצגת:

Report
+ report: string
+ createReport: void + printReport: void

אם ניקח את מחלקת הדו"ח הזה, נראה שיש בה שני פעילויות – האחת יצירת דו"ח, והשניה הדפסת הדו"ח. עכשיו עלינו לחשוב מה יהיו המקרים בהם אנחנו נצטרך לשנות פרטים במחלקה – אירוע אחד אם צורת הדוח עצמו השתנתה, והמקרה השני הוא כאשר צורת ההדפסה תשתנה. כמובן שיצירת דוח והדפסתו הם שני אירועים שונים לגמרי, ולכן אנחנו נשאף להפריד את שתי הפעולות לשתי מחלקות שונות – האחת, מחלקה שקשורה רק לדוח בעצמו, כלומר יצירת הדוח, והמחלקה השניה תהיה מחלקה שקשורה להדפסות שונות – המחלקה הזאת תדפיס את הדוח, וכך כל דבר שנדרוש שיעבור הדפסה כל שהיא נוכל לקשר אותה למחלקה הזאת ולעבוד איתה.

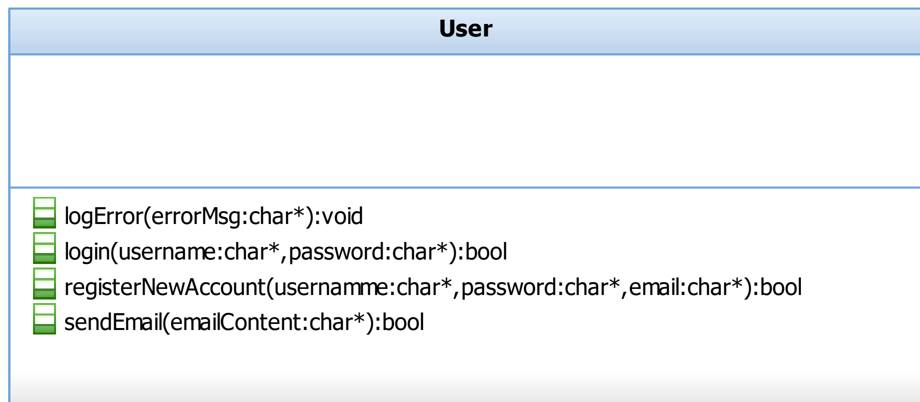
יתרונות SRP

- בדיקת המחלקות נעשית בצורה יותר פשוטה. ברגע שאנחנו יודעים שאנחנו מתרכזים סביב נושא אחד, אם במהלך הבדיקה נתקל באיזה מודול שמתחיל לעבוד על דברים אחרים יותר כלליים או שלא באמת קשורים למחלקה, נדע שיש לנו פה בעיה.
- בנוסף, הבדיקות עצמן יהיו יותר קלות ופשוטות – עבור נושא אחד אותו אנחנו אמורים לפתור, נוכל לבדוק את המקרים הפשוטים ואת מקרי הקצה, וכך לצאת ידי חובת הבדיקות בלי להתחיל לבדוק נושאים לא קשורים.

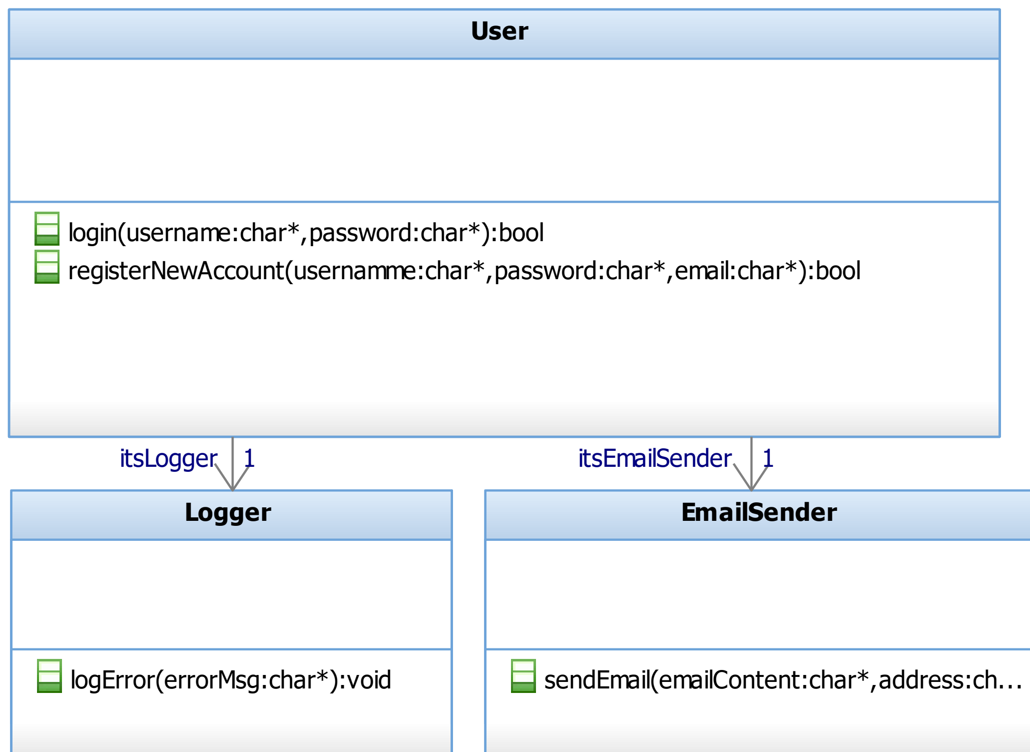
[3] עבור כל עיקרון אביא את הציטוט הרשמי מהמצגת ונסביר אותו.

- פחות פעולות שוות פחות תלויות שונות בין המודולים. ברגע שאנחנו עושים רק דבר אחד אנחנו לא אמורים להיות תלויים ביותר מידי דברים. כמובן שיכול להיות שיהיה לנו תלות מסוימת לדברים שאנחנו משתמשים, אבל אנחנו בטוח נצמצם את זה למינימום ההכרחי.
- קל יותר לחפש ולמצוא דברים ופונקציות בתוך מחלקה מאורגנת, ובכלל בתוך תכנית מאורגנת. אם אנחנו מחפשים נושא מסוים שמטופל בתכנית, אנחנו יודעים לנחש איה הוא יהיה והיכן להסתכל עליו, ולא נחשוש שפתאום הוא יופיע במקום צדדי ולא קשור.

דוגמא נוספת וקצת יותר מפורטת למימוש העיקרון –



יש לנו מחלקה של משתמש המכיל מספר פונקציות – לוג של שגיאה, התחברות, רישום ושליחת אימייל. ברור לנו שיש למחלקה הזאת יותר מידי משימות, וחלקן הגדול בכלל לא קשור למשימה העיקרית של המחלקה – טיפול במשתמש. איזה דברים אנחנו נצטרך לשנות במחלקה? נניח וצורת שליחת האימייל תשתנה, הלוג ייכתב אחרת, פרטי הרישום ישתנו ועוד. כל דבר כזה יגרור שינויים למרות שמדובר בתחומי אחריות שונות לגמרי. לכן אנחנו נפצל את המחלקה לניהול פרטי משתמש, טיפול בשגיאות וטיפול בשליחת מיילים.



כעת יש לנו את המחלקה של המשתמש, שמשתמש במודולים השונים ויכולה להדפיס שגיאות או לשלוח מיילים ללא בעיה. בנוסף, אם נרצה לשנות את הפונקצינליות של השגיאות, אנחנו נתעסק רק איתם ולא נחשוש לפגיעה במשתמש או בחלקים אחרים.

שיטה לבדיקת שייכות של הפונקציות למחלקה, היא כתיבה של כל המחלקות באופן הבא:

"The _____ ______ itself"

כאשר החלק הראשון יהיה שם המחלקה, והשני יהיה הפעלה/הפונקציה שהמחלקה מבצעת על עצמה. למשל: "המחלקה משתמש תרשום משתמש בעצמה" זה משפט הגיוני בהתחשב בזה שאנחנו מדברים על משתמשים, אבל "המחלקה משתמש תשלח מיילים" זה קצת פחות קשור.

OPC – Open-Close Principle

"תוכנה צריכה להיות פתוחה להרחבה וסגורה לשינויים"

העיקרון הזה הוא מאוד יסודי, וממנו נגזרים כמעט כל היסודות האחרים. כבר מהשם שלו, אפשר לראות שיש פה משהו קצת פרדוקסלי, ונראה מה הכוונה ואיך עומדים בו. הניסוח של הכלל מגדיר כי הקוד צריך להיות "פתוח להרחבה" – כלומר, היכולת להשתמש בתוכנה במקומות נוספים, על ידי התאמה מינימלית. אך "סגורה לשינויים" במובן שכאשר נרצה להשתמש בתכנה במקומות נוספים, אנחנו לא נוסיף דברים על הקיים אלא רק נרחיב אותו. תכלס זה עדיין לא מספיק ברור.

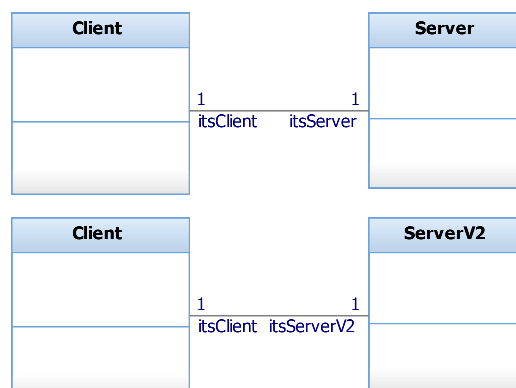
דוגמא בולטת לתכנה שכל הייעוד שלה עומד בעיקרון הזה, הוא תכונות כדוגמת Visual Studio או IntelliJ, שיש להם את המימוש הפשוט שלה, ובעזרת פלאגינים שונים – שאינם משנים שום דבר במהות התוכנית המקורית – אנחנו משיגים הרחבות פונקציונליות לאותו בסיס.

ניתן דוגמא פשוטה שמפירה את העיקרון הזה –

נניח ויש לנו משחק לוח בגודל 10*10. הנותן לנו אפשרות להזזת חיילים. עכשיו אנחנו רוצים ליצור מוד דומה למשחק הזה בגודל 20*20, ואנחנו רואים שמה שיש לנו בנוי בדיוק בגודל 10*10 ללא אפשרות לשינוי – עכשיו יש לנו כמה אפשרויות – ליצור מחלקה חדשה בגודל 20*20 זה סתם בזבוז זמן ושכפול קוד. לא טוב.

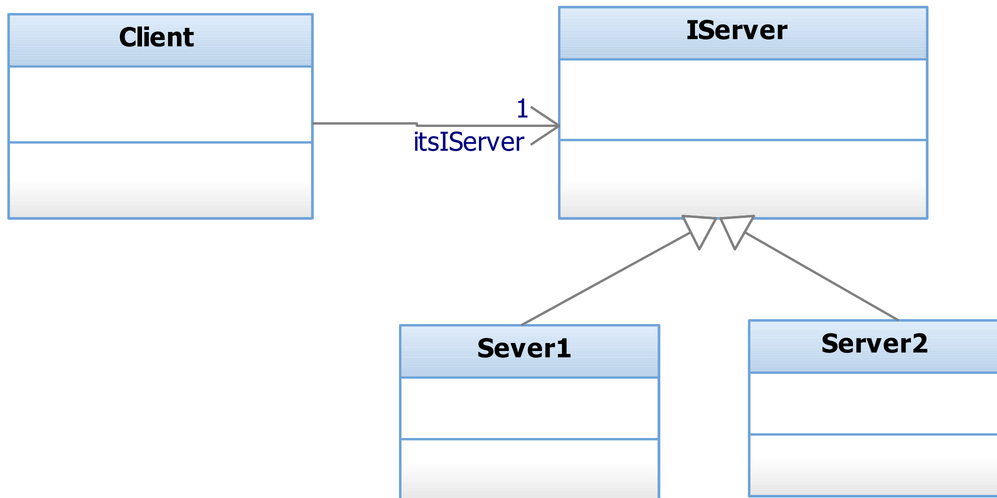
אנחנו יכולים לשכפל לוחות, וליצור את המחלקה מהרכבה של מספר לוחות כאלה, ואנחנו יכולים באופן הכי נכון פשוט ליצור את המחלקה באופן שהבנאי יגדיר את גודל הלוח הדרוש.

ניקח דוגמא נוספת –



יש לנו פה מחלקה של client, שיש לה הכלה דו כיוונית משתי מחלקות שונות של Server. תכלס שתי המחלקות האלה הן מבטאות את אותו רעיון, אז מימוש נכון של OCP יצמצם את הקשרים. למה? כי אם יש לנו רק שני סרברים, אנחנו יכולים להתמודד. אבל אם פתאום יגיע עוד אחד? או שאנחנו לא נדע כמה יגיעו בכלל, ובכל פעם נצטרך לממש את הסרברים האלה מחדש וחבל.

לכן אנחנו נרחיב את התוכנית מבלי לשנות בקוד –



אם הסרבר שלנו מגיע תמיד עם אותה פונקציונליות, או לפחות עם בסיס מספיק דומה, אנחנו יכולים ליצור ממשק של סרבר (להלן: IServer) ולדאוג שהקליינט ייגש אך ורק אליו. עכשיו כל שרת שנקים ונרצה שהלקוח יוכל להשתמש בו, יצטרך לרשת מהממשק ולממש את הדרוש, וכך הלקוח יוכל לגשת אליו בלי בעיות. שיו לב – הרחבנו את הקוד, אך בלי לשנות בו.

העקרון הזה מאפשר לנו גמישות בקוד, לאחר שנממש הכל באופן נכון אנחנו נוכל להוסיף דברים ומודולים בלי לחשוש לקשיחות או שבירות בקוד.

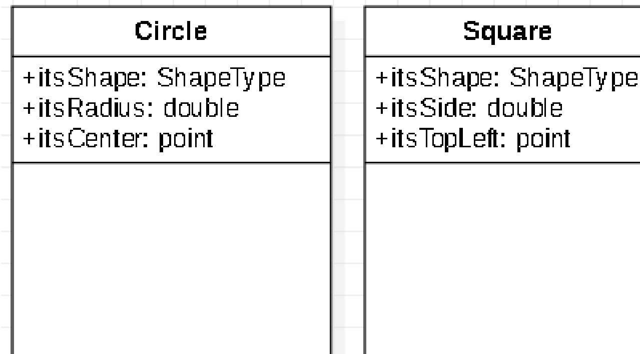
ישנה דוגמא קלאסית לעניין ה-OCP המתייחסת לצורות והדפסתן-

```
enum ShapeType {circle, square};
struct Point{…};
struct Shape
{
 ShapeType itsType;
};
struct Circle
{
 ShapeType itsType;
 double itsRadius;
 Point itsCenter;
};
struct Square
{
 ShapeType itsType;
 double itsSide;
 Point itsTopLeft;
};
```

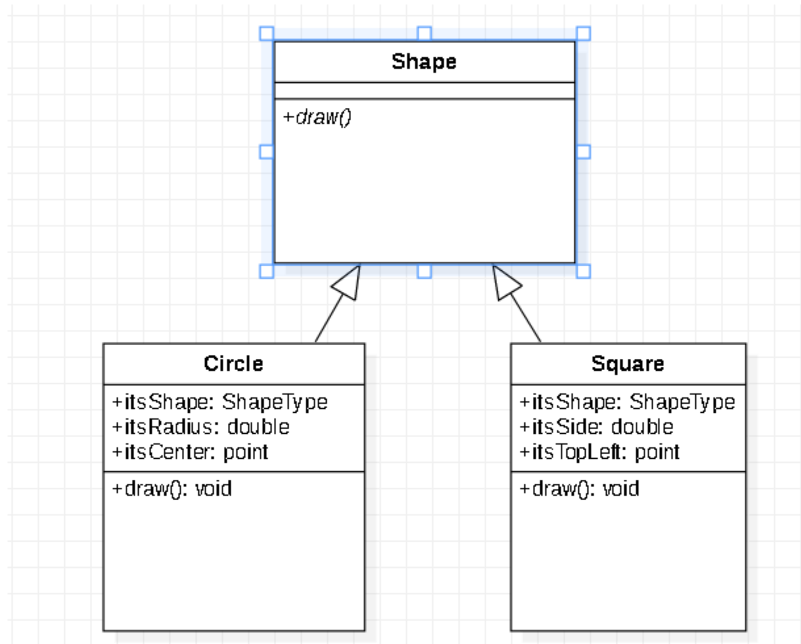
```
void drawSquare(Square* s){…};
void drawCircle(Circle* c){…};
void drawAllShapes(Shape* list[], int n) {
 int i;
 for (i=0; i<n; i++)
 {
  Shape *s = list[i];
  switch (s->itsType)
  {
   case square:
    drawSquare((Square*)s);
    break;
   case circle:
    drawCircle((Circle*)s);
    break;
  }
 }
}
```

נניח ואנחנו יוצרים מספר צורות – עיגול וריבוע, ואנחנו רוצים ליצור פונקציה שתדפיס כל אחד מהמופעים שאנחנו יוצרים. נתחיל מה דרך הכי בסיסית ו"פרימיטיבית". ניצור את הצורות, ובפונקציית ההדפסה נבדוק ברשימה שקיבלנו את המופעים הקיימים ובכל פעם נדפיס את המופע המתאים לו לפי הפרמטרים.

לפי כל מה שלמדנו עד עכשיו אנחנו יכולים לתאר את ה-UML של הצורות האלה באופן מאד פשוט –



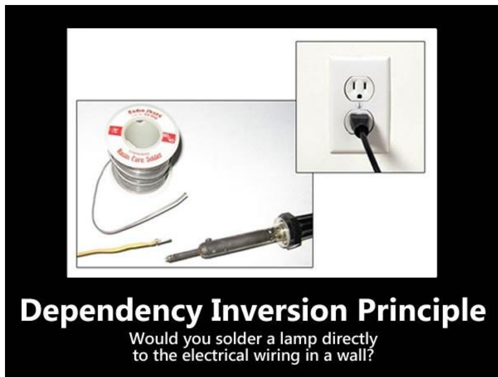
אבל ידוע לנו מקדמת דנא, שריבוע ועיגול, על אף השוני שלהם, שניהם צורות. לכן מה שנעשה, הוא ליצור ממשק של צורה שיכיל את הפונקציה הוירטואלית של הדפסה, וכך כל צורה תצטרך לממש את ההדפסה שלה בפני עצמה, ופונקציית ההדפסה תוכל לקרוא להדפסה ישירות מהצורה. למה זה חשוב? כי קודם היו לנו שתי פונקציות הדפסה שונות, אחת לריבוע ואחת לעיגול, ואנחנו קראנו לפונקציות האלה בצורה חיצונית. עכשיו אנחנו חוסכים את כל הבדיקות השונות וסומכים על כך שהמתכנת שיוסיף צורות יהיה חייב לממש הכל. מה שייצור לנו את הדבר הנפלא הבא –



וייתן לנו את האפשרות לממש את פונקציית ההדפסה בצורה הרבה יותר קצרה:

```
void drawAllShapes(Shape* list[], int n)
{
 int i;
 for (i=0; i<n; i++)
  list[i]->draw();
}
```

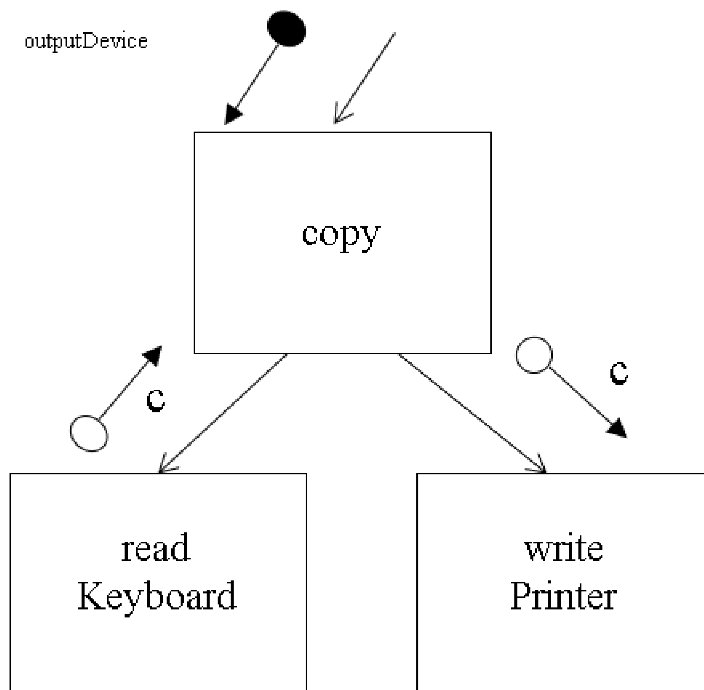
DIP Dependency Inversion Principle



" מודול בשכבה מסויימת לא יהיה תלוי ישירות בשכבה נמוכה יותר, התלות בניהם תהיה באמצעות הפשטה."

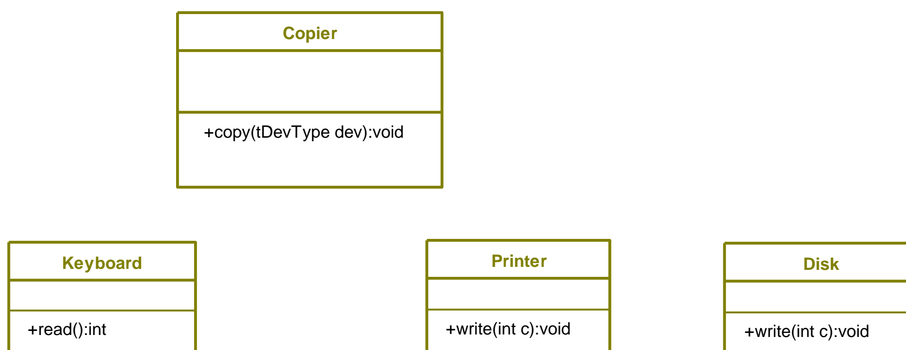
עקרון היפוך התלות מדבר על הקשר בין המחלקות השונות, לעומת OCP בו דיברנו על המחלקה הבודדת. באופן הפשוט ביותר, אנחנו דורשים שהמחלקה המפורטת יותר תהיה תלויה במחלקה המופשטת ולא להיפך – הפרט יהיה תלוי בכלל.

ניקח דוגמא שמסבירה את העניין ואיך משתמשים בו על ידי הפונקציה "העתקה":

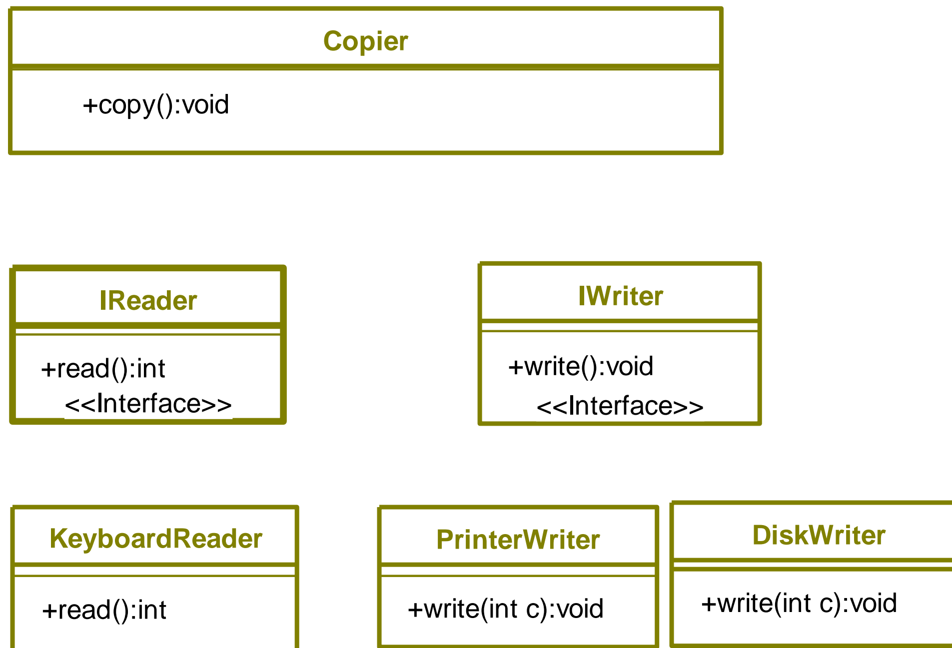


על מנת להעתיק קלט מה מקלדת ולהעביר אותו להיות פלט למדפסת, אנחנו משתמשים במין מחלקת העתקה שלוקחת וקוראת הכל. מה הבעיה בזה ברמת העיצוב? המקלדת והמדפסת שניהם מאוד קונקרטיים, ואילו פעולת ההעתקה היא תהליך מופשט שיכול להתיחס לכל כך הרבה דברים.

ברמת ה-UML אנחנו עלולים ליצור משהו כזה –



סתם ככה, היינו עלולים לשוב שזה בסדר, כי כל אחד יממש את ההעתקה אצלו וכו'. אבל אם אנחנו נרצה להוסיף פונקציונליות של העתקה ממקום למקום זה יהיה לנו מאוד בעייתי. עקרון ה-DIP מלמד אותנו כיצד לפשט את המחלקות לכיוון של ממשקים וכך להשתמש בדברים בצורה יותר פשוטה –



מה החידוש פה?

קודם כל יש לנו את המחלקה המעתיקה הראשית והיא הרמה הכי מופשטת. ממנה יוצאים שני יורשים – האחת קריאה והשניה כתיבה, כי הרי לכל אחד מהם יש אלמנטים של העתקה – גם להעתיק את הקלט וגם להעתיק אל הפלט.

מכל אחת מהממשקים המשניים אנחנו יכולים להוציא את המחלקות שיממשו אותן באופן מקומי. כך שאם נרצה להוסיף אחר כך, אנחנו נוכל להוסיף כל אחד במקומו בלבד.

איך מימשנו כאן את העיקרון? באופן שעשינו, המימוש של ההעתקה לדיסק בהתחלה היה בו תלות מהמחלקה Copy לתוך המחלקה של הדיסק, אבל בעצם אין לנו תורך התלות הזאת. מה שאנחנו רוצים הוא שהדיסק יידע להעתיק ולכן את הרעיון המופשט של "העתקה" אנחנו מעלים למעלה, ברמה מתחת אנחנו מכניסים את סוגי העתקות השונים – קריאה וכתיבה – ורק לאחר מכן אנחנו מכניסים את המימושים הקונקרטיים שמתיחסים כל אחד לעצמו בהכרח האפשרות המופשטת של ההעתקה.

תבניות עיצוב

כאן אנחנו עוצרים לרגע את הדיון ב-SOLID בשביל להכנס לעניין תבניות העיצוב. התבניות הללו פותרות לנו כל מיני בעיות של תלויות ומימושים שעוזרים לנו עם עקרון ה-DIP. נעצור רגע להסביר זאת, ואז נוכל להמשיך הלאה.

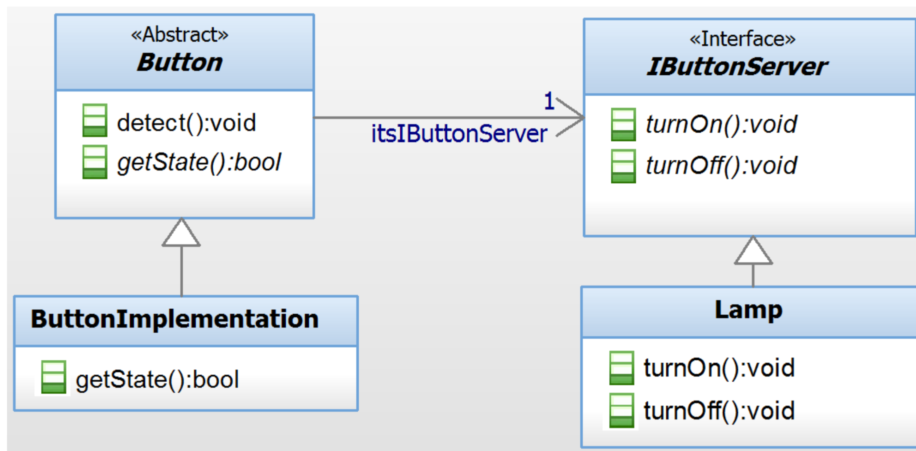
נציין קודם כל את הסוגים השונים של התבניות המקובלות –

- תבניות יצירה
- תבניות מבנה – (קשרים בין מחלקות)
- תבניות התנהגות – מתרכזים בהעברת שליטה ממקום למקום ובין מחלקות שונות.
- תבניות מקבול – עבודה על תהליכונים שונים.

עכשיו ננסה להרחיב חלק מהתבניות, ונראה כיצד אנחנו פותרים בעזרתם בעיות –

Adapter

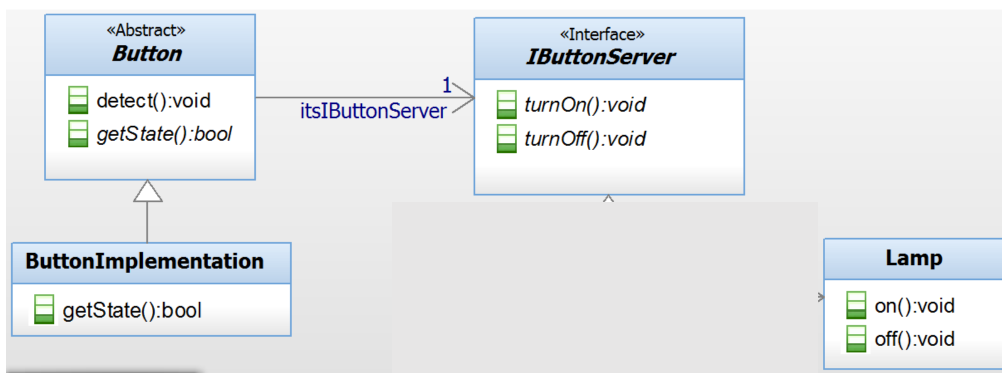
האדפטר (מסגל) הוא תבנית התנהגותית המשמשת מעין מתאם בין ממשק קיים לבין מחלקה המשתמשת בממשק אחר ללא צורך בקימפול.



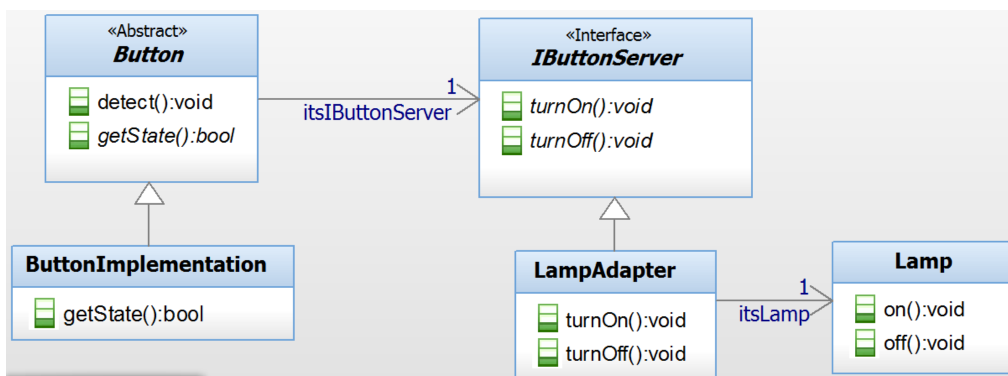
ניקח את הדוגמא הזאת בה מימשנו את הדלקת המנורה בעזרת הכפתור. המחלקה Lamp ירשה את הממשק של הכפתור ומדליקה או מכבה את עצמה על פי הנדרש.

אבל מה קורה כאשר היתה לנו מחלקת Lamp והפונקציות שלה לא היו turnOn, אלא רק on?

לא נראה כמו משהו מטורף מידי, אפשר פשוט לעשות Rename, ולת=התאים את השם. אבל אם אי אפשר? נגיד ויש לנו תלויות או סיבה מסוימת אחרת לעשות את זה דווקא ככה, ועכשיו אנחנו תקועים עם הדבר הבא –



מה עושים עכשיו? למקרה כזה נוצר האדפטר. מחלקה שנמצאת באמצע בין הממשק למנורה, ויודעת ברגע שהממשק מבקש turnOn אנחנו שולחים אותו לתוך Lamp->on.



החיסרון של האדפטר הזה, הוא שאם עכשיו יש לנו עוד מחלקות כאלה, נצטרך ליצור מחלקות כפולות רק בשביל לעשות את הגישור הזה כל הזמן, וזה כמובן לא כיף.

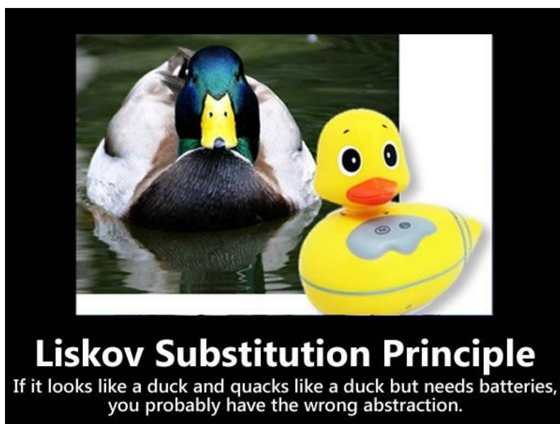
Iterator

את האיטרטור אנחנו כמובן מכירים כבר, והוא בעצם בא לממש לנו גם את עיקרון ה-DIP. ברגע שיש לנו פעולה מופשטת שאנחנו יודעים שהרבה מחלקות ממשות אותה, אנחנו יכולים להשתמש בזה לטובתינו. במה דברים אמורים?

אנחנו מגדירים ממשק של איטרטור המכיל שתי פונקציות – next(), hasNext().

כל מחלקה שתממש את האיטרטורציה הזאת תצטרך לעשות שני דברים, להחזיר האם יש עוד לאן לעבור, ובמידה ויש לאן, לעבור לאיבר הבא בקונטיינר (מערך, רשימה, מחרוזות, מה שבא לנו). כך שאנחנו יוצרים פה היפוך תלות, לא ניצור מחלקה של איטרציה ולתלות אותה לתוך רשימה למשל, ואז לעבור שם. אלא להכריז על הפעולות הדרושות המופשטות, וכל המחלקות האחרות שרוצות לממש את הפעולות האלה יהיו תלויות בזה.

LSP – Liskov Substitution Principle



"תת-טיפוסים חייבים להיות ברי החלפה (התנהגותית) עבור טיפוס הבסיס שלהם"

נחזור לדיון אודות עקרונות ה-SOLID עם עקרון ההחלפה של ליסקוב, המכונה כך על שם החוקרת שטבעה אותו.

לצורך הדיון, נתחיל בשאלה פרובוקטיבית – אמרנו שאם יש לנו מחלקות שמבצעות את אותו הדבר, אנחנו יכולים לנסות לשים אחת מהן בירושה תחת האחרת (כמובן, באופן כללי). נניח ויש לנו מחלקה של חשבון בנק, בעלת הפונקציה "Open". ולידו יש לנו תנין. התנין יכול לפתוח את הפה, ולכן אנחנו מסמנים את הפה הנפלא של התנין עם פונקציה, גם היא קרויה "Open".

האם התנין יכול לרשת מחשבון הבנק? האם הוא גם יכול להיות אח (כלומר, מחלקה אחרת שתירש מחשבון בנק) של המחלקה דלת? הרי גם היא נפתחת!

אם אנחנו מגדירים ירושה בתור פעולות דומות, אז האם אנחנו יכולים להגיד שתין הוא "סוג של" חשבון בנק?

לפני שנמשיך – לא.

עכשיו נמשיך.

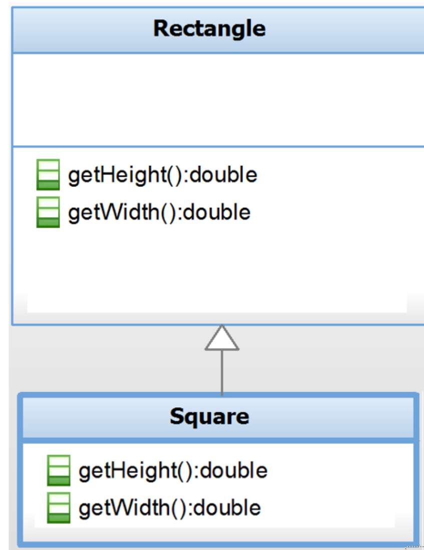
אנחנו רוצים לקבוע מספר חוקים על מנת לדעת האם ומתי אנחנו יכולים לעשות השמה של טיפוסים מסוגים שונים האחד על השני. נראה מספר הגדרות לעקרון הזה – מתי מותר לנו להגדיר ירושה ומתי אסור לנו. בסוף הדיון הזה, המטרה שלנו היא שכאשר נעשה את הרפרנס הזה, נעשה אוות נכן ברמה העיצובית ובאופן הראוי.

בבסיס העיקרון אנחנו מדברים על "תכונות-האב" לעומת "תכונות הבן" – כשמשתמשים באבא, ואנחנו רוצים מחלקה של אבא, אבל קיבלנו מחלקה של בן. השאיפה שלנו היא שנוכל להשתמש במחלקה היורשת באופן שלם בדיוק כמו המחלק המקורית, מבלי לחשוש להתנהגות הבן. אנחנו בעצם מתייחסים ל"סטיגמות" של כל המחלקות שיורשות מהמחלקה המקורית.

איך אנחנו יכולים לבדוק שאנחנו אכן עומדים בתנאים הללו? על מנת לזהות את זה, יש לנו את "מבחן הזיוף" שמסוגל לבדוק האם הקשר בין שתי המחלקות עובד בצורה נכונה. הבדיקה היא פשוטה – נגיד ורצינו מופע של T

אבל קיבלנו מופע של S. אם אנחנו לוקחים את המופע החדש ומריצים עליו את **כל** הפונקציות של T, ומקבלים את התוצאה לה ציפינו גם מ-T, נדע שהעיקרון עובד היטב. כלומר, אם זיהינו זיוף, אז בנינו משהו באופן לא נכון.

ניקח דוגמא פשוטה ונראה איך אנחנו מיישמים את העיקרון –



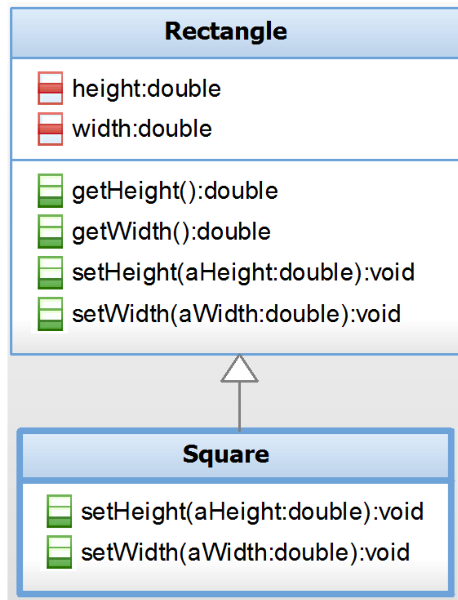
כידוע לכל – ריבוע הוא מקרה פרטי של מלבן בו כל הצלעות שוות, ולכן הגיוני שהריבוע יירש מהמלבן. מבחינת הקוד, לא אמור להיות פה בעיה מיוחדת – אנחנו מבקשים מהמלבן גובה או רוחב, והוא יחזיר, נבקש מהריבוע, ויחזור לנו אותו דבר.

```
class Rectangle
{
 public:
  double getHeight() const {return height;}
  double getWidth() const {return width;}
  double getArea() const {return hight*width;}
 private:
  double height;
  double width;
};
```

```
class Square : Rectangle
{
 public:
   double getHeight() const {return size;}
   double getWidth() const {return size;}
   double getArea() const {return size*size;}
  private:
   double size;
};
```

שימו לב שאכן הריבוע מכיל ערך משתנה יחיד עבור שתי הצלעות.

עכשיו נעבור לשלב הבא – מה יקרה אם נוסיף גם פונקציות של Set למחלקות האלה? האם אנחנו עדיין לא נגלה זיופים?



נראה גם את השינוי בקוד –

```
class Rectangle
{
 public:
  virtual void setWidth(double aWidth) {width= aWidth;}
  virtual void setHeight(double aHeight){height=aHeight;}
  double getHeight() const {return height;}
  double getWidth() const {return width;}
  double getArea() const {return hight*width;}
 private:
  double height;
  double width;
};
```

```
class Square : Rectangle
{
 public:
   void setWidth(double aWidth)
   {
     Rectangle::setWidth(aWidth);
     Rectangle::setHeight(aWidth);
   }

   void setHeight(double aHeight)
   {
     Rectangle::setWidth(aHeight);
     Rectangle::setHeight(aHeight);
   }

   double getHeight() const {return size;}
   double getWidth() const {return size;}
   double getArea() const {return size*size;}
 private:
  double size;
};
```

במבט ראשון – הריבוע עצמו ממומש בצורה הגיונית לגמרי – ברגע שנזין את האחד מערכי הפאות, אנחנו מצפים שהנתון הזה יעבור גם לחלק השני. אנחנו לא יכולים ליצור ריבוע של 4*5. אבל האם זה עובד בעקרון ההחלפה של ליסקוב? נריץ את הבדיקה הבאה –

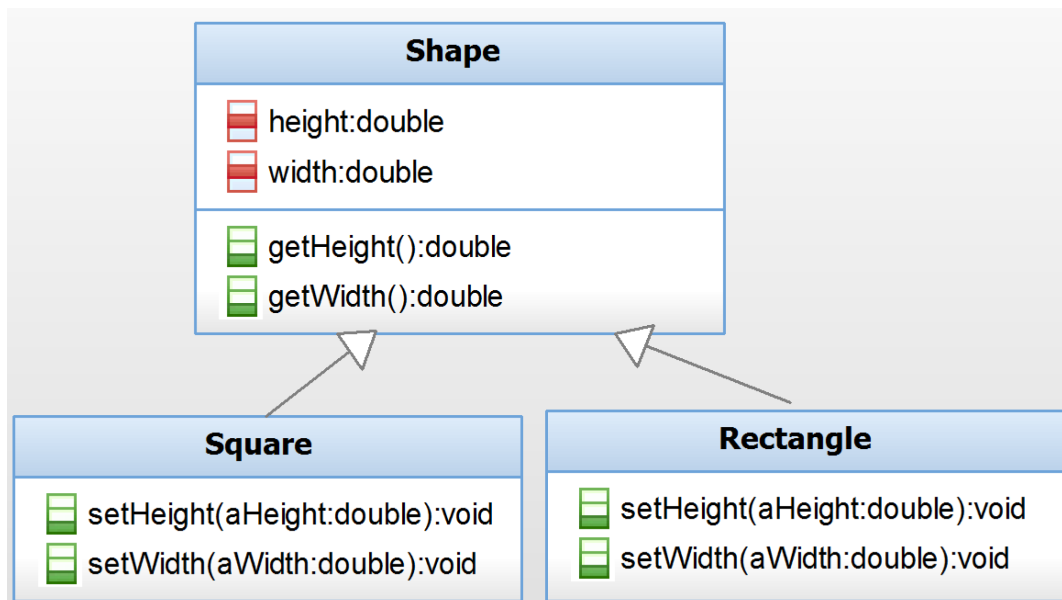
```
void g(Rectangle& r)
{
  r.setWidth(5);
```

```
 r.setHeight(4);
 assert(r.getArea()== 20);
}
```

כזכור, אנחנו מצפים לקבל **בדיוק** את אותה התנהגות של המחלקה ממנה אנחנו יורשים. אם נזין אורך ורוחב שונים למלבן, אנחנו מצפים לקבל את השטח באופן מתאים לאורכים שונים. אך האם הריבוע יוציא לנו את הפלט המתאים? לא. נכניס 5, ואז יוגדר לנו רק אורך 5, נכניס 4, ואז יוגדר לנו אורך 4, ומה שיחזור בשטח יהיה 16 ולא 20.

אם כן, מה מצופה מאיתנו? למה בכלל העקרון הזה חשוב כל כך? כשאנחנו בונים את המערכת שלנו, אנחנו רוצים שהממשק מול המערכת יהיה כמה שיותר פשוט. אם אנחנו ניצור מימוש שבו הלקוח או מתכנת אחר שירצה לעבוד על התוכנית יקבלו נתונים אחרים מאלו שציפו להם, תהיה לנו בעיה רצינית.

ובאופן קונקרטי יותר לבעיה שהעלינו, אמנם ריבוע הוא מקרה פרטי של מלבן, אבל מאחר והמימושים שלהם שונים ועלולים ליצור לנו זיופים, אנחנו לא נירש מהמלבן, אלא נדרוש שם המלבן וגם הריבוע יירשו שניהם ממחלקת "Shape".



הרחבה או הגדרה נוספת של העקרון של ליסקוב אומר כך - נניח ש $\phi(x)$ היא תכונה המתקיימת עבור אובייקטים x מסוג T. אז $\phi(y)$ צריכה להתקיים עבור אובייקטים y מסוג S, כאשר S יורש מ T.

ובעברית – אם יש לנו אוסף תכונות עבור אובייקט מסוים, נגיד – אף ארוך, חיבה לכסף, רצון לשלוט בעולם. אז גם כל הבנים והיורשים של זקני המחלקה בוודאי גם הם מקיימים את התנאים שאנחנו מפרסמים בפרוטוקול.

Design by Contract

ישנה גישה שמגדירים מחלקות ופונקציות באופן שאנחנו כותבים "חוזה", ואת החוזה הזה כל מי שרוצה להשתמש במחלקה חייב לבוא ולקיים. כלומר, אם אנחנו מכניסים לתוך פונקציה את הפריטים המסוימים, אנחנו יודעים שיצא לנו פלט מסוים, ואז דווקא פלט בתור ערך, אלא כטיפוס.

את העיצוב על פי חוזה, תכלס אנחנו כבר מכירים ומשתמשים בו מההתחלה – כאשר אנחנו כותבים כל פונקציה בשפת C, אנחנו מגדירים את הערך המוחזר, וכן את הערכים (הכללים) שאמורים להכנס לתוך הפונקציה.

למעשה אנחנו מדברים גם על תנאים שהם אפילו יותר מורכבים מזה, הנקראים תנאי קדם (Pre-Conditions) ותנאי בתר (Post-Condition).

בתנאי הקדם אנחנו נכניס עבור פעולת מסוימות, שהן יכולות להתבצע רק אם יש תנאים מסויימים שייקבעו – למשל, אם אנחנו רוצים להגדיר פונקציה של משיכה מחשבון בנק, אנחנו יכולים להגדיר כתנאי קדם שרק במידה והסכום לאחר המשיכה יהיה חיובי, אנחנו נשאר את הפעולה, ואם לא אז עדיף לנו שלא נבצע אותה.

תנאי הבתר פשוט מוודא שפעולה התבצעה נכון – לפני שנמשיך ונשחק עם החשבון, נוודא שפעולת המשיכה השאירה סכום הגיוני בתוך החשבון.

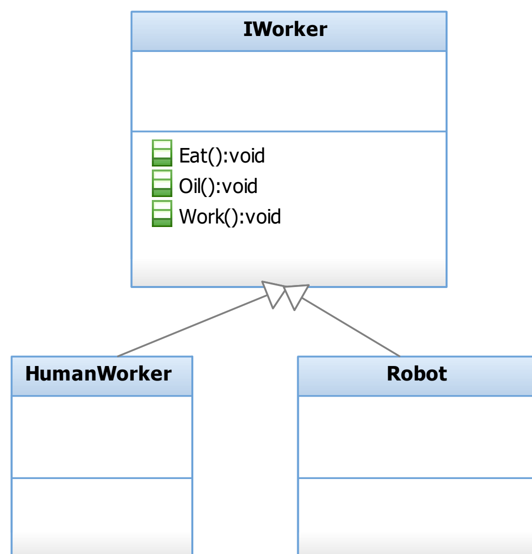
מה הקשר לפה?

אנחנו דורשים שהבנים יתנו לנו את הפונקציונליות של האב, אבל מה לגבי הדרישות השונות? הכלל אומר כך – עבור דרישות הקדם – אנחנו יכולים להחליש אותם ולתת טווח רחב יותר של אפשרויות. אם אנחנו לא מרחיבים פה כלום, אז אין לנו בכלל שום חידוש במה שאנחנו עושים. (כדוגמא- אם אנחנו רוצים להרחיב את הטווח שמדחום מסוים יציג לנו מעלות זה בסדר גמור)
מבחינת דרישות הבתר, אנחנו יכולים אך ורק לחזק אותם, כלומר לוודא שאנחנו מצמצים את התמונה (התוצאה של הפונקציה, מבחינתנו זה בסדר להוציא בסוף משהו מצומצם יותר, אבל לא משוחרר יותר. (אם יש לנו הגבלת סטיה של מעלה בודדת במדחום, אז לתת אפשרות לסטיה בשתי מעלות, זה כבר פסול)

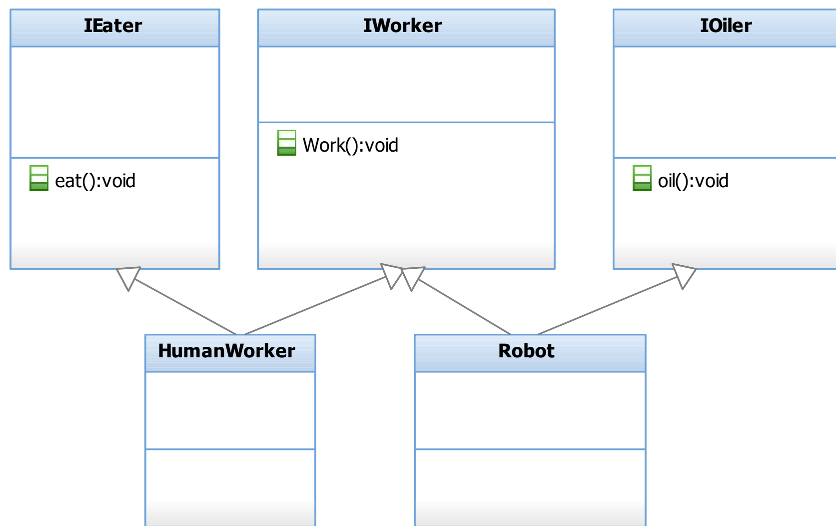
ISP – Interface Segregation Principle

"הקליינט יהיה חשוף אך ורק לממשק שהוא זקוק לו.
אסור להכריח קליינט להיות תלוי בממשק בו הוא לא משתמש"

עקרון הפרדת הממשקים לא מדבר על הקשר בין האב לבן, אלא בין הקליינט שמשתמש בממשק מסוים לאותה המחלקה אותה הוא מממש. אם יש פונקציות בממשק אותן המחלקה לא מממשת. יש פה הפרה של עקרון ה-ISP, מאחר שהקליינט לא אמור להיות חשוף לממשק בו הוא לא משתמש. סתם כדוגמא ראשונית נדבר על עובד, שמכיל פונקציות כמו ()eat, ו-()work. אם יבוא היום ואותו עובד יוחלף על ידי רובוט, כמובן שהפונקציה eat תהיה לא רלוונטית עבורו. במקרה כזה, אנחנו נרצה לממש את ממשק העובד בצורה שתהיה רלוונטית למשתמשים שונים. אם נתחיל מהמצב הזה -



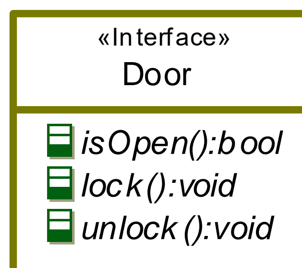
מה שנעשה, הוא להפריד את הממשק לקבוצות שונות. למשל יהיה לנו ממשק של "אוכלים", "נטענים" ועוד דומים להם. בנוסף יהיה לנו את ממשק העבודה הכללי, ואותו עובד אנושי או רובוטי שיוגדר וימשש את פונקציות העובד, יקושר גם לממשקים היותר רלוונטיים אליו. ככה –



איך נדע מתי להפריד? אם יש לנו קליינט חדש שאנחנו רוצים לתלות בממשק, ואנחנו נתקלים בפונקציה שאין סיבה שהוא יממש , במקום לעשות בלאגן ואדפטרים נפריד אותם לשני ממשקים שונים שיהיו רלוונטיים כל אחד בפני עצמו.

הדוגמא הבאה תהיה יותר מפורטת, והיא חשובה מאוד גם לתרגילי הבית וגם למבחן.

אנחנו רוצים ליצור תוכנית לדלת ביטחון. על מנת להתחיל את כל העבודה, אנחנו נתחיל ביצירת ממשק כללי של דלת. הדלת תכיל פונקציות פשוטות של "נעילה" ו"פתיחה", וכן בדיקה בוליאנית שתוכל לומר לנו האם הדלת פתוחה או סגורה.



מאחר שאנחנו מדברים על ממשק, אז כל הפונקציות שם יהיו וירטואליות טהורות, וימומשו באופן הבא –

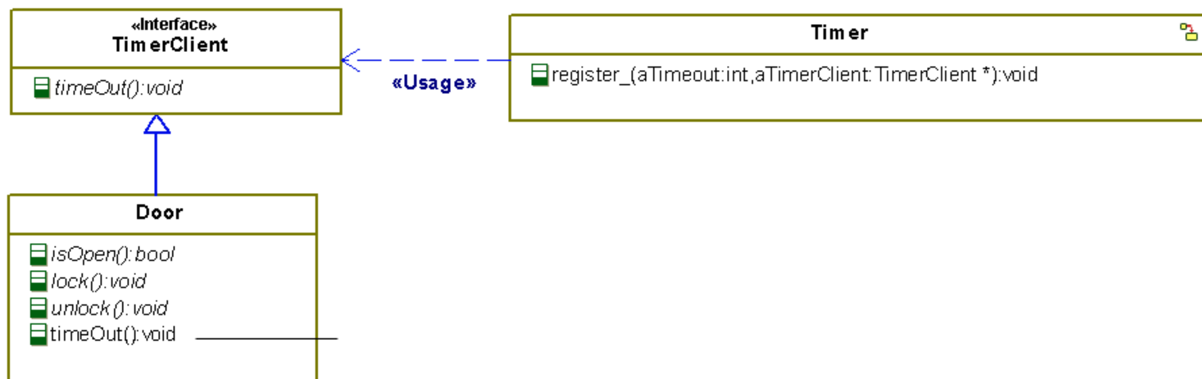
```
#ifndef Door_H
#define Door_H
class Door {
public :
    virtual bool isOpen()=0;
    virtual void lock()=0;
    virtual void unlock()=0;

};
#endif
```

בשלב הבא, אנחנו נרצה לממש דלת ביטחון, בצורה קצת יותר ספציפית. הדלת הזאת תיתן לנו התראה ברגע שהדלת פתוחה יותר מידי זמן. מה זה יותר מידי זמן? אנחנו נקבע.

בשביל ליצור את זה, אנחנו ניצור את המחלקה המתאימה שהיא TimedDoor. המחלקה תירש כמובן מהמחלקה Door, ותממש את הפונקציות בצורה הדרושה. עכשיו אנחנו צריכים להתמודד עם בדיקת הזמן בעצמו – ניצור אובייקט מסוג Timer, שישתמש בממשק מסוג TimerClient המכיל רק פונקציה וירטואלית אחת של timeOut. בתוך הטיימר אנחנו נכניס פונקציה של רישום אירוע, שתקבל את אורך הטייםאאוט, ומצביע לקליינט שנרשם לאירוע.

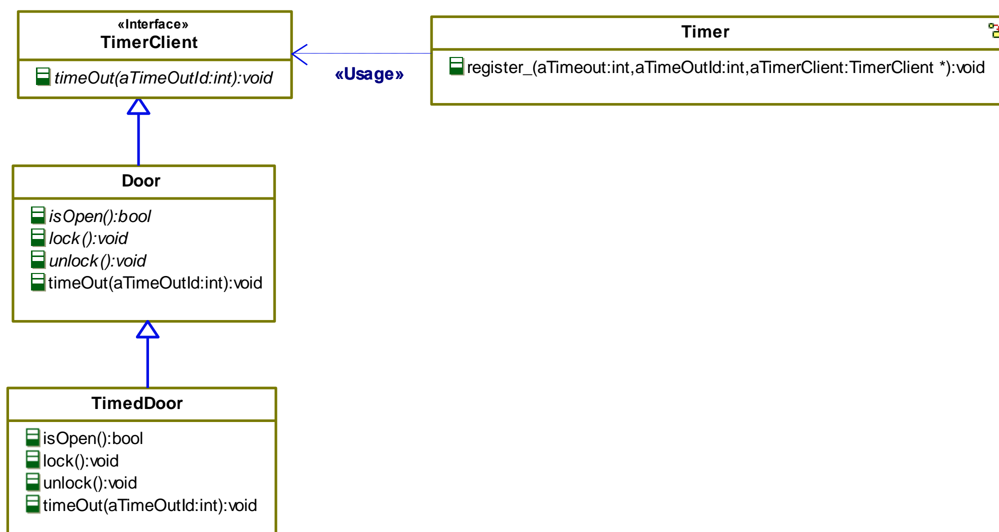
בנוסף ניצור מחלקה של רישומים לכל ההתראות של האירועים – כלומר פתיחות הדלת, וניצור קונטיינר של כל המופעים הללו, וכן נוסיף פונקציה שבודקת האם כעבור זמן מוגדר (בלעז timeout), הדלת עדיין פתוחה או סגורה.

אחרי כל זה, השאלה הנשאלת היא – איך נסדר את המחלקות? ההצעה הראשונית שלנו תיראה כך –



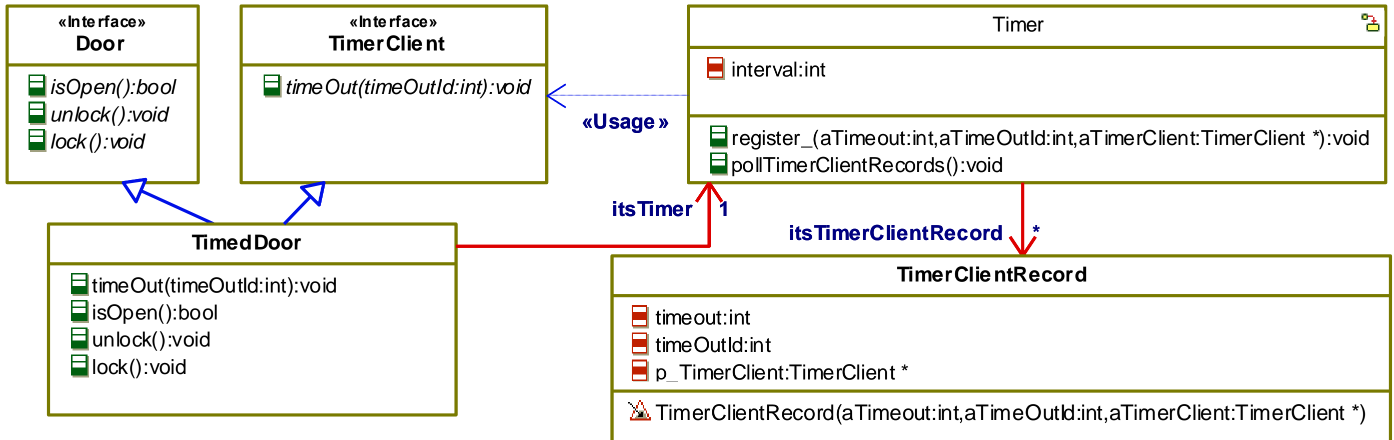
אבל כאן יש לנו הפרה של LSP. למה? כי הירושה הזאת מהממשק בהחלט לא מתנהג כמו טיימר. יותר מזה, עלול להיות לנו בעיה חמורה ברמת המימוש. אם אנחנו רושמים אירוע שבודק כל עשרים שניות האם הדלת פתוחה, ופתחנו את הדלת, אז הדלת רשמה אירוע, ואנחנו יודעים לבדוק בעוד 20 שניות האם הדלת פתוחה או סגורה. ועכשיו נניח שסגרנו אותה ופתחנו מחדש בטווח הזמן של אותם 20 שניות. לכאורה, יש פה פתיחת דלת חדשה, ואנחנו רושמים אירוע חדש, אבל מבחינת הפונקציה שבודקת האם הדלת פתוחה, היא עובר שוב, ורואה שהדלת פתוחה אז היא תעביר התראה. וזה לא טוב ועלינו לטפל בזה.

בשביל לטפל בזה, מעבר לרישום האירוע, אנחנו נוסיף משתנה של timeOutID, שיזהה לנו בכל פעם איזה אירוע עובד באותו רגע. כך שאם אירוע א' כבר נסגר, הוא לא בטעות ירים אזעקות על אירוע ב'. את כל זה אנחנו נוריש למחלקה של הדלת המתוזמנת ושם נממש את כל הסיפור הזה –



עכשיו שעשינו את כל זה זה נחמד, אבל הפרנו את ISP. למה? כי המחלקה דלת לא באמת צריכה להשתמש בפונקציה timeout, זאת פונקציה ששייכת למה שיורש ממנו, ה-TimedDoor. וכאן נכנס החוק שאמרנו קודם שמחלקה לא אמורה להיות חשופה לממשק אותו היא לא מממשת.

ולכן, אנחנו נפריד פה את הממשקים לשניים שונים, ונדאג ש-TimedDoor יממש את הממשקים הרלוונטיים אליו –

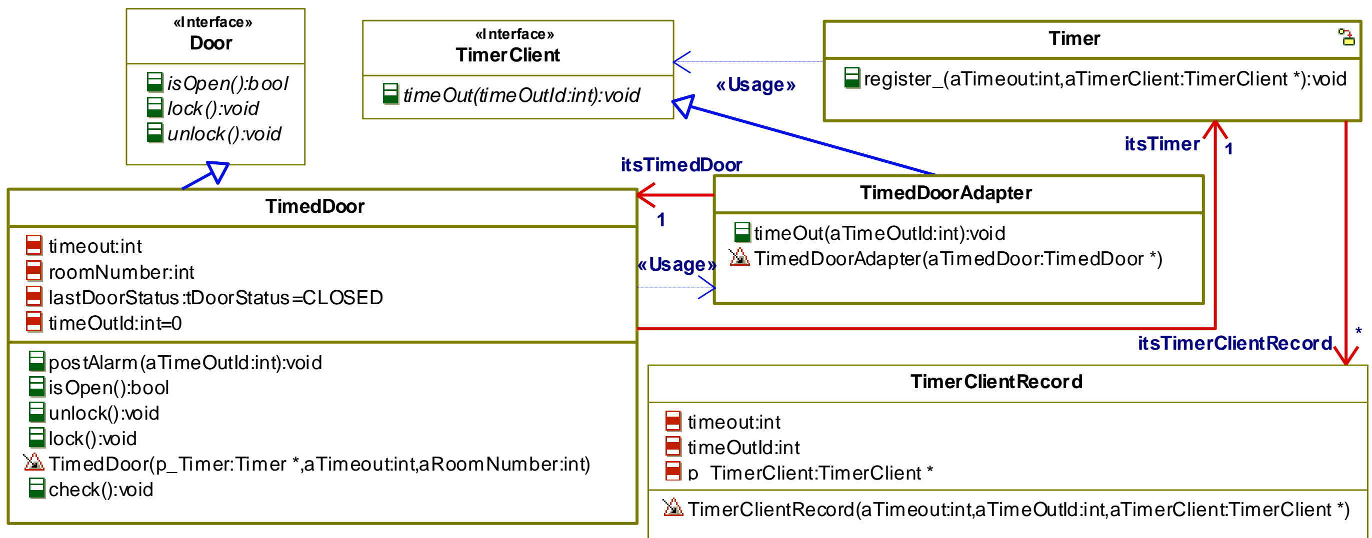


זה נראה קצת יותר מבולגן, אבל נשים לב מה קורה פה - המחלקה הרלוונטית עבורנו, שסביבה כל השאר סובבות היא ה-TimedDoor. המחלקה הזאת מממשת הן את הפונקציונליות של הדלת, והיא גם מוגדרת בתור קליינט-טיימר. בעבור כל מופע של כל דלת כזאת, אנחנו מצמידים גם מופע בודד של Timer, המסוגל להפעיל ולממש קונטיינר של אירועי הפעלת טיימר בעבור כל פתיחה שתהיה.

מבחינת הקוד נסתכל רק על זה של ה-TimedDoor, ונראה מה אנחנו מוציאים ממנו –

```
# ifndef TimedDoor_H
# Define TimedDoor_H
# include “Door.h”;
# include “TimerClient.h”;
class TimedDoor  : public Door, public TimerClient {
public :
    virtual void lock();
    virtual void unlock();
    virtual bool isOpen();
     virtual void timeOut(int timeout, int timeOutId);
};
#endif
```

עכשיו אנחנו נוסיף פה אדפטר. למה? מאחר ויכול להיות שיווצרו לנו מספר מופעים, הרי יש לנו יותר מדלת אחת, ובעבור כל דלת עלולים להיות לנו נתונים שונים. נראה את השרטוט וננסה להבין ממנו, מה אנחנו מרויחים פה –



השינוי הראשון הבולט לעין, המחלקה TimedDoor כבר אינה יורשת/מממשת את TimerClient, אלא רק האדפטר עושה זאת. האדפטר מקבל את השימוש בכל הפונקציונליות של הדלת, לה התווספו עוד נתונים נוספים ופונקציות שמטפלות בטיפול באזעקות וקביעה האם מדובר באזעקת אמת או שקר, וכן במשתנה ENUM-י בו כתוב לנו האם הדלת פתוחה או סגורה (לצרכי בקרה).

מה שיצרנו פה הוא מערכת שעובדת בצורה מלאה (פחות או יותר), ושום חלק מיותר לא קשור למחלקות שלא ישתמשו בו.

לסיכום

עקרונות ה-SOLID הוא בסיס לתבניות עיצוב רבות. אך חשוב לזכור – לא מדובר פה על הלכה למשה מסיני, ודברים שחייבים להיעשות רק בדרך אחת, מאחר ולפעמים אנחנו כן נדרוש במודע לעבור על עיקרון אחד, בתמורה לערון אחר. כך שיש תמיד להפעיל שיקול דעת.

אם נחלק בגדול את העקרונות לנושאים נוכל לומר שאנחנו מחפשים נכונות (LSP, ISP) באבסטרקציה (OCP, DIP) והגדרה נכונה של המחלקות השונות (SRP).

Extreme Programming

הבסיס לחלק זה מגיע מתפיסת האג'ייל עליה דיברנו קצת, השואפת לכתיבת קוד זריז וגמיש. את כל עקרונות האג'ייל עוד תלמדו בהמשך בקורס "הנדסת תכנה". עכשיו אנחנו מתמקדים בגישה שנגזרה מהאג'ייל, שגזרה כל מיני שיטות עבודה שונות ומשונות, שחלקן נתפסו כבר כמוסכמות כתיבת קוד. נעבור על חלק מהשיטות השונות.

Refactoring

הריפקטורינג היא שיטה של בקרת הקוד, ווידוא שרשמנו את הקוד נכון וברור. ישנן כל מיני פעולות שאנחנו יכולים לעשות תוך כדי כתיבת הקוד בעצמו, וכן לאחר שנסיים לכתוב את הקוד. השאיפה שאנחנו תמיד נהיה במין מעגל של איסוף דרישות ← הרחבת הקוד הקיים כך שיתמוך בדרישות ← ריפקטורינג לכל הקוד וחוזר חלילה.

הדרך שבה נלמד את הריפקטורינג הוא מעין Study-case שנכתב במאמר "The Craftsman", על ידי רוברט מרטין המכונה "הדוד בוב", ובו הוא מתאר פיתוח של כתיבת קוד על ידי מתלמד צעיר בשם אלפונס, אל מול המתכנת שעובד מולו ורודה בו – ג'רי.

המשימה שג'רי נותן לאלפונס היא לכתוב פונקציה שתחשב מספרים ראשוניים בטווח מסוים.

מה שאלפונס מביא הוא הדבר הבא –

נעבור בזריזות על הקוד רק כדי לראות שאנחנו מבחינים את הרעיון מאחורי מה שכתוב –

הרעיון מאחורי הפונקציה הוא הנפה של אריסתוטנס שפשוט בטווח הנתון, אנחנו מאתחלים את מערך המספרים שכולו יהיה true וכל מספר לא-ראשוני שנמצא יסומן כ-false. השיטה היא להתחיל ממספר 2 ולסמן את כל הכפולות שלו עד הגבול העליון, ואז לעבור ל-3 וכן הלאה, עד שנגיע לגבול העליון, ואז להחזיר את המספרים שנשארו לנו כראשוניים.

אלפונס דואג שתהיה הקדמה ארוכה בהערות המסבירה לנו מה הוא עושה, ועל מה הוא נסמך, מציג את המקרים הידועים ועושה הכל בצורה נכונה.

```java
/**
* This class Generates prime numbers up to a user specified
* maximum. The algorithm used is the Sieve of Eratosthenes.
* <p>

* Eratosthenes of Cyrene, b. c. 276 BC, Cyrene, Libya --
* d. c. 194, Alexandria. The first man to calculate the
* circumference of the Earth. Also known for working on
* calendars with leap years and ran the library at Alexandria.
* <p>
* The algorithm is quite simple. Given an array of integers
* starting at 2. Cross out all multiples of 2. Find the next
* uncrossed integer, and cross out all of its multiples.
* Repeat untilyou have passed the square root of the maximum
* value.
*
* @author Alphonse
* @version 13 Feb 2002 atp
*/
import java.util.*;
public class GeneratePrimes {
 /**
  * @param maxValue is the generation limit.
  */
public static int[] generatePrimes(int maxValue) {
 if (maxValue >= 2) // the only valid case
 {
  // declarations
  int s = maxValue + 1; // size of array
  boolean[] f = new boolean[s];
  int i;
  // initialize array to true.
  for (i = 0; i < s; i++)
   f[i] = true;
  // get rid of known non-primes
  f[0] = f[1] = false;
  // sieve
  int j;
  for (i = 2; i < Math.sqrt(s) + 1; i++) {
   if (f[i]) // if i is uncrossed, cross its multiples.
   {
    for (j = 2 * i; j < s; j += i)
     f[j] = false; // multiple is not prime
   }
  }
  // how many primes are there?
  int count = 0;
  for (i = 0; i < s; i++) {
   if (f[i])
    count++; // bump count.
  }
  int[] primes = new int[count];
  // move the primes into the result
  for (i = 0, j = 0; i < s; i++) {
   if (f[i]) // if prime
    primes[j++] = i;
  }
  return primes; // return the primes
 } else // maxValue < 2
  return new int[0]; // return null array if bad input.
 }
}
```

אבל, אם אנחנו מציגים את המקרה כנראה שיש לנו פה בעיות. ואם לא זיהינו את הבעיות, כנראה שאנחנו הבעיה. ראשית, יש לשים לב, שאם אנחנו ננסה לעבור על הקוד, אנחנו נצטרך לעצור כל שורה ולהבין מה היא עושה. אין לנו שום דרך לדעת מה אנחנו עושים רמה הרעיונית מלבד ההקדמה, ובמעקב אחרי השורות עצמן זה עלול להיות לנו לא קל.

בנוסף, אלפונס גם דאג לבדיקת הקוד –

```
import junit.framework.*;
import java.util.*;
public class TestGeneratePrimes extends TestCase
{
public static void main(String args[])
{
junit.swingui.TestRunner.main(
new String[] {"TestGeneratePrimes"});
}
public TestGeneratePrimes(String name)
{
super(name);
}
public void testPrimes()
{
int[] nullArray = GeneratePrimes.generatePrimes(0);

assertEquals(nullArray.length, 0);
int[] minArray = GeneratePrimes.generatePrimes(2);
assertEquals(minArray.length, 1);
assertEquals(minArray[0], 2);
int[] threeArray = GeneratePrimes.generatePrimes(3);
assertEquals(threeArray.length, 2);
assertEquals(threeArray[0], 2);
assertEquals(threeArray[1], 3);
int[] centArray = GeneratePrimes.generatePrimes(100);
assertEquals(centArray.length, 25);
assertEquals(centArray[24], 97);
}
}
```

הטסטים שאלפונס הכניס הם 0,2,3,100. הבדיקות שנעשות הן הבסיסיות ביותר, וגם די ברור לנו מה יהיו התוצאות כך שאין פה ממש אתגר לתוכנית. מצד שני, אם אנחנו מספיק בטוחים במה שאנחנו עושים זה קצת קשה לנו לבדוק את זה.

לאחר שאלפונס קרא לג'רי שיבדוק את העבודה שלו, ג'רי הסתכל על הקוד ואמר שקוד כזה יגרום לבוס לפטר את שניהם, והטיל על אלפונס לעשות ריפקטור לקוד. כאשר אנחנו כותבים קוד, או עושים ריפקטור לקוד, יש לנו כמה דברים שאנחנו צריכים לחשוב עליהם – קודם כל, אנחנו צריכים להית מסוגלי לראות מהקוד את התמונה המלאה, ולא רק את הדברים הקטנים שאנחנו מבצעים. בנוסף, אנחנו צריכים שהקוד יהיה קריא, ככה שכשמישהו אחר שהוא לא אנחנו יקרא את הקוד, הוא יבין מה קורה שם בלי להסתבך. מעבר לזה, תחזוקת הקוד בוודאי גם תיגזר מזה וכם היעילות הכללית. אנחנו גורסים ש"הקוד הוא התיעוד", ולכן במקום לתת את עלילות אריסתותנס ואיך הוא ניפה מספרים ראשוניים, אנחנו צריכים להתמקד יותר באופן שנכתוב את זה בצורה שהקורא יבין את סדר הפעולות.

Extract Method

ההוראות שג'רי אמר לאלפונס הן קודם כל, לקצר את ההערות בהתחלה, ואז לנסות לעשות extract לחלק מהפעולות כך שיהיו שלוש פונקציות עיקריות שיטפלו במה שאנחנו דורשים.

אלפונס ישב ימים ולילות, ללא מזון וללא אוכל וללא כלום. ואז חזר עם הקוד הבא –

```java
/**
 * This class Generates prime numbers up to a user specified
 * maximum. The algorithm used is the Sieve of Eratosthenes.
 * Given an array of integers starting at 2:
 * Find the first uncrossed integer, and cross out all its
 * multiples. Repeat until the first uncrossed integer exceeds
 * the square root of the maximum value.
 */
import java.util.*;
public class PrimeGenerator {
 private static int s;
 private static boolean[] f;
 private static int[] primes;
 public static int[] generatePrimes(int maxValue) {
  if (maxValue < 2)
   return new int[0];
  else {
   initializeSieve(maxValue);
   sieve();
   loadPrimes();
   return primes; // return the primes
  }
 }
 private static void loadPrimes() {
  int i;
  int j;
  // how many primes are there?
  int count = 0;
  for (i = 0; i < s; i++) {
   if (f[i])
    count++; // bump count.
  }
  primes = new int[count];
  // move the primes into the result
   for (i = 0, j = 0; i < s; i++) {
    if (f[i]) // if prime
     primes[j++] = i;
   }
  }
  private static void sieve() {
   int i;
   int j;
   for (i = 2; i < Math.sqrt(s) + 1; i++) {
    if (f[i]) // if i is uncrossed, cross out its multiples.
    {
     for (j = 2 * i; j < s; j += i)
      f[j] = false; // multiple is not prime
    }
   }
  }
  private static void initializeSieve(int maxValue) {
   // declarations
   s = maxValue + 1; // size of array
   f = new boolean[s];
   int i;
   // initialize array to true.
   for (i = 0; i < s; i++)
    f[i] = true;
   // get rid of known non-primes
   f[0] = f[1] = false;
  }
 }
```

האם אלפונס עשה נכון? מה הוא בכלל עשה?

קודם כל, הוא פירק באמת את כל העבודה לשלוש חלקים נפרדים – initializeSieve, sieve, loadPrimes. וההערות אכן יותר קצרות.

למה בכלל אנחנו עובדם בצורה כזאת? תכלס, מבחינת השורות בקוד, אנחנו רק הוספנו שורות, אבל לנו כבני אדם לפעמים קשה לראות יותר מידי שורות ופקודות ברצף. אם אנחנו רואים שכל התכנית עושה בעצם שלוש פעולות קצרות ולא אחת ארוכה זה יהיה לנו הרבה יותר קל לקרוא את זה – לא נועמס בפרטים מיותרים, ולא נאבד בכל השורות. יש לזכור, שלפעמים ההוצאה הזאת של פונקציות גורמת לנו צורך לעשות משתנים סטטיים בשביל שנוכל לעבוד בכל רמות ההכרה השונות של הפונקציות.

דבר נוסף שנעשה פה, הוא ברמת סידור התנאים. אם יש לנו קוד שבודק אחד מכמה תנאים, אז אנחנו נעדיף בדרך כלל לשים את התנאים הקצרים בהתחלה. גם זה מהסיבה שזה הרבה יותר קריא לנו – אנחנו עוברים קודם כל על הפעולות הפשוטות יותר, ורק אם יש צורך אנחנו מגיעים לחלקים המסובכים.

Renaming

דבר שאומרים לנו בערך מאז שהתחלנו ללמוד ולקודד, הוא הרעיון שעלינו לשים לב שאנחנו נותנים שמות משתנים בעלי ערך. למה זה חשוב? אם יוצא לכם לעבור על קוד ישן שלא אתם כתבתם, פתאום בתוך כל מאות השורות שאתם מנסים להבין, תקוע לכם משתנה xשאתם מנסים להבין מה הוא מבטא – מספר אנשים? נתון? ערך מוחזר? אם היה רשום לנו לעומת זאת numOfWizards זה היה לנו הרבה יותר ברור.

מספר כללי אצבע בבחירת שמות:

- לא לתת שמות חסרי משמעות בכלל x,y,foo,bar זה אחלה בתור דוגמאות בשיעורים, פחות בקוד של תכנית.
- שמות פונקציות בדרך כלל יהיו פעלים. הפעלים ינסו לתאר את המשימה של הפונקציה אליה אנחנו נכנסים – checkForAvailabeSpots אומר לנו באופן פשוט וברור מה הפונקציה עושה.
- להמנע משימוש בקיצורים. Acc יכול להיות קיצור של accelerator, ויכול להיות accumulator. לא כדאי לבנות על זה שנזכור מה זה אומר.
- אם אנחנו מתייחסים לאמות מידה, כדאי לציין אותם priceNIS יחסוך לנו הרבה זמן כאשר לא נבין למה חישוב הדולר לא עובד לנו.

לאחר שג'רי בדק את הקוד של אלפונס, הוא אמר שכבר זה נראה הרבה יותר טוב, אבל יש יותר מידי משתנים שאולי אומרים לנו משהו כרגע, אבל בעוד שבוע או יומיים אנחנו לא נבין מה זה f. ולכן הוא שינה קצת את הקוד לדבר הבא –

```java
public class PrimeGenerator {
 private static boolean[] f;
 private static int[] result;

 public static int[] generatePrimes(int maxValue) {
  if (maxValue < 2)
   return new int[0];
  else {
   initializeArrayOfIntegers(maxValue);
   crossOutMultiples();
   putUncrossedIntegersIntoResult();
   return result;
  }
 }
 private static void initializeArrayOfIntegers(int maxValue) {
  f = new boolean[maxValue + 1];
  f[0] = f[1] = false; //neither primes nor multiples.
  for (int i = 2; i < f.length; i++)
   f[i] = true;
 }
```

אילו שינויים נעשו פה?

שמות הפונקציות שונו לכאלה שיהיו קריאות יותר, וכן המערך שונה מ-primes ל-result. לא יהפוך את העולם, אבל בהחלט יותר מובן.

כמובן, שכבר עברנו כברת דרך עצומה מהבלאגן שחגג לו אי אז, אבל האם זה מספיק? אם אנחנו שואלים, אז כנראה שלא –

```java
public class PrimeGenerator {
 private static boolean[] isCrossed;
 private static int[] result;
 public static int[] generatePrimes(int maxValue) {
  if (maxValue < 2)
   return new int[0];
  else {
   initializeArrayOfIntegers(maxValue);
   crossOutMultiples();
   putUncrossedIntegersIntoResult();
   return result;
  }
 }
 private static void initializeArrayOfIntegers(int maxValue) {
  isCrossed = new boolean[maxValue + 1];
  for (int i = 2; i < isCrossed.length; i++)
   isCrossed[i] = false;
 }
 private static void crossOutMultiples() {
  int maxPrimeFactor = calcMaxPrimeFactor();
  for (int i = 2; i <= maxPrimeFactor; i++)
   if (notCrossed(i))
    crossOutMultiplesOf(i);
 }
 private static int calcMaxPrimeFactor() {
  // We cross out all multiples of p, where p is prime.
  // Thus, all crossed out multiples have p and q for
  // factors. If p > sqrt of the size of the array, then
  // q will never be greater than 1. Thus p is the
  // largest prime factor in the array, and is also
  // the iteration limit.
  double maxPrimeFactor = Math.sqrt(isCrossed.length) + 1;
  return (int) maxPrimeFactor;
 }
private static void crossOutMultiplesOf(int i)
{
 for (int multiple = 2 * i; multiple < isCrossed.length; multiple += i)
  isCrossed[multiple] = true;
}
private static boolean notCrossed(int i) {
 return isCrossed[i] == false;
}
```

קודם כל, המערך f שינה את שמו ל-isCrossed, מה שמסתדר לנו עם הלוגיקה הכללית של כל התוכנית. בנוסף, יצרנו עוד כמה פונקציות משנה שמטפלות בראשוניים השונים באופן שנותן לנו את הגבול העליון לבדיקות הכפלות – שורש מספר לעולם יהיה המספר הגבוה ביותר המוכפל במספר נתון. וכך עבור כל ספר אנחנו בודקים את הכפולות הנתנות לנו.

```
private static void putUncrossedIntegersIntoResult() {
 result = new int[numberOfUncrossedIntegers()];
 for (int j = 0, i = 2; i < isCrossed.length; i++)
  if (notCrossed(i))
   result[j++] = i;
}
private static int numberOfUncrossedIntegers() {
 int count = 0;
 for (int i = 2; i < isCrossed.length; i++)
  if (notCrossed(i))
   count++;
 return count;
}
```

שינוי נוסף שאלפונס עשה, הוא הוצאה של פונקציה נוספת שבודקת כמה מספרים שעוד לא סומנו יש לנו.

תכלס, נעשו עוד שינויים בפונקציה הזאת אבל זה די ארוך ומתיש.

למעוניינים –

https://drive.google.com/file/d/0BwhCYaYDn8EgMDZkNTI5MzItYmQyNS00ZmFlLTgyMzItMjJiZGVhMTE2NGVm/view

https://drive.google.com/file/d/0BwhCYaYDn8EgYzY1ZTJhZmUtZDdlNy00Nzc5LTk4NjMtYzYzZGZhMWI0YTBj/view

https://drive.google.com/file/d/0BwhCYaYDn8EgMjc5ZGVjYTQtZmE4NS00MjM0LWIwMDMtMTE2M2NkNTUxNzgx/view

אלו שלושת החלקים שעברנו עד כה, כל השאר נמצא כאן - https://sites.google.com/site/unclebobconsultingllc/home/articles

אנחנו התחלנו ממש מהשלושה הראשונים.

שיטות נוספות לריפקטורינג שיש לציין

Inline Method

לפעמים אנחנו עלולים כל כך להנעל על פתיה של פונקציות חדשות, שאנחנו עלולים לסרבל קריאות לפונקציה שלא עושה הרבה. זה אמנם לא נראה קריטי, אבל יעילות הקוד פוחתת בזה.

אם למשל אנחנו כותבים פונקציה בצורה כמו שמתואר משמאל –

```java
class PizzaDelivery {
  // ...
  int getRating() {
    return moreThanFiveLateDeliveries() ? 2 : 1
  }
  boolean moreThanFiveLateDeliveries() {
    return numberOfLateDeliveries > 5;
  }
}
```

```java
class PizzaDelivery {
  // ...
  int getRating() {
    return numberOfLateDeliveries > 5 ? 2 : 1;
  }
}
```

יש לנו בעיה שאנחנו שולחים לפונקציה שבודקת תנאי מאוד פשוט שאין לנו סיבה מיוחדת שהוא לא ייכתב פשוט באותה שורה.

Variable Extraction

```java
void renderBanner() {
  if ((platform.toUpperCase().indexOf("MAC") > -1) &&
       (browser.toUpperCase().indexOf("IE") > -1) &&
        wasInitialized() && resize > 0 )
  {
    // do something
  }
}
```

האם יש לנו בעיה? לא. האם אנחנו יכולים יותר טוב מזה? כן.

התנאי פה מאוד ארוך וקצת קשה לקריאה. מה שאפשר לעשות הוא להוציא את כל הבלאגן הזה, ולהכניס לשם שמות משתנים קלים יותר –

```java
void renderBanner() {
  final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
  final boolean isIE = browser.toUpperCase().indexOf("IE") > -1;
  final boolean wasResized = resize > 0;

  if (isMacOs && isIE && wasInitialized() && wasResized) {
    // do something
  }
}
```

אין פה הרבה שינוי, אבל את התנאי אנחנו יכולים להבין בקלות רבה יותר, וזה מה שאנחנו מחפשים.

Inline temp variable

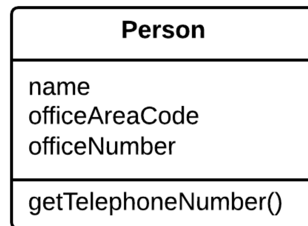
באופן הפוך לרעיון הקודם, ובדומה למה שאמרנו עם השליחה המיותרת לפונקציות, אין לנו צורך בכל פעם להעביר נתונים לשמות משתנים קצרים יותר. לפעמים אנחנו נעדיף ששם המשתנה יהיה טיפה יותר ארוך, אך התוכנית עצמה תהיה יותר קצרה. יש לחזור ולציין, לא מדובר פה הרבה פעמים בדברים ברורים מידי. קשה לפעמים להחליט מתי להוציא משתנה ומתי לא, אך זה בא עם הניסיון.

```java
boolean hasDiscount(Order order) {
  double basePrice = order.basePrice();
  return basePrice > 1000;
}
```

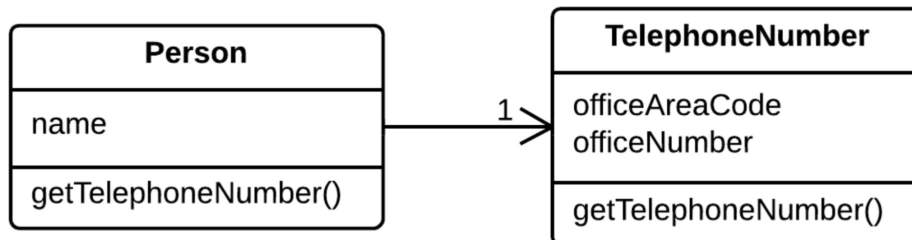
```java
boolean hasDiscount(Order order) {
  return order.basePrice() > 1000;
}
```

Extract/Inline class

אותו רעיון שדיברנו, רק עם מחלקות ונתונים. לפעמים נעדיף מחלקה שתכיל הרבה נתונים, ככה –



ולפעמים נעדיף שתי מחלקות, שכל אחת מהן תחזיק מספר נתונים שונה –



Split temporary variable

אם יש לנו שני משתנים בעלי אותו שם, אך הם מבטאים ערכים שונים, כדאי לנו לשנות אותם –

```
final double perimeter = 2 * (height + width);
System.out.println(perimeter);
final double area = height * width;
System.out.println(area);
```

```
double temp = 2 * (height + width);
System.out.println(temp);
temp = height * width;
System.out.println(temp);
```

Remove assignment to parameters

```
int discount(int inputVal, int quantity) {
  int result = inputVal;
  if (inputVal > 50) {
    result -= 2;
  }
  // ...
}
```

```
int discount(int inputVal, int quantity) {
  if (inputVal > 50) {
    inputVal -= 2;
  }
  // ...
}
```

אם יש לנו ערכים שנכנסים לפונקציה, ועלינו לשנות את הערכים שבה, עדיף לנו ברמה הבטיחותית לא לשנות את המשתנה שקיבלנו. אמנם בחלק מהשפות אנחנו מקבלים את הרפרנס בצורה מוגנת שלא תפגע במקור, אבל תמיד כדאי להיות בטוחים, ואם יש לנו ספק פגיעה במקום אחר עדיף לעשות השמה למשתנה מקומי.

Replace “magic number”

```
double potentialEnergy(double mass, double height) {
  return mass * height * 9.81;
}
```

אם יש לנו חישוב שאנחנו עושים עם איזה מספר קבוע, עדיף לנו לא לרשום את המספר בעצמו. גם אם אנחנו מכפילים בפאי, וברור לכולם מה זה 3.1416, הקוד יהיה יותר קריא אם פשוט נכתוב radius*PI.

```
static final double GRAVITATIONAL_CONSTANT = 9.81;

double potentialEnergy(double mass, double height) {
  return mass * height * GRAVITATIONAL_CONSTANT;
}
```

Consolidate duplicate conditional fragments

אם יש לנו כפילות פקודות תחת תנאים, אנחנו נעדיף את החלקים הכפולים לעשות מחוץ לתנאי. למשל –

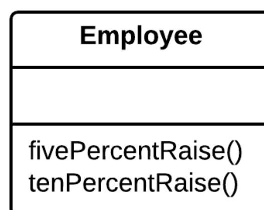
```
if (isSpecialDeal()) {
  total = price * 0.95;
  send();
}
else {
  total = price * 0.98;
  send();
}
```

```
if (isSpecialDeal()) {
  total = price * 0.95;
}
else {
  total = price * 0.98;
}
send();
```

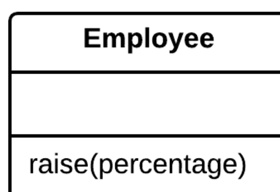
הפעולה send תתבצע בכל מקרה, אז אנחנו נעדיף להוציא אותה אל מחוץ למסגרות התנאים.

Parametrize method

אם יש לנו מספר פונקציות שעושות כמעט את אותו הדבר –



אין הבדל ממשי בחישוב בין העלאה של חמש או עשר אחוזים. החישוב עצמו יהיה בדיוק אותו דבר. לכן, עדיף לעשות הכל בפונקציה אחת, ופשוט לדאוג לשלוח את ההעלאה הרצויה בכל פעם –



אלו השיטות שעברנו בכיתה. יש כמובן עוד רבים וניתקל בהם בהמשך החיים.

TDD – Test Driven Development
תכנות מונחה בדיקות

נדבר כעת על תכנות מונחה בדיקות.

הגישה של TDD היא שונה ממה שעשינו עד עכשיו – היינו כותבים קוד, ואז (במקרה הטוב) היינו כותבים כמה טסטים שיוודאו שמה שכתבנו אכן עושה את מה שאנחנו רוצים. במקרה הפחות טוב, גם את זה לא היינו עושים. מה הבעיה בזה? כידו, תכנה היא דבר הנתון לשינויים, ואם אנחנו כותבים את הטסטים על פי מה שאנחנו כותבים, אזי בכל איטרציית שינויים, נצטרך לבדוק מחדש את הטסטים, כי אולי הם הותאמו דווקא למה שכתבנו ולא לדברים אחרים.

גישת ה-TDD גורסת, כי עוד לפני שנתחיל לכתוב את הקוד עצמו, נשב ונכתוב את הטסטים. במבט ראשון זה נשמע קצת מוזר, איך נכתוב טסטים אם אין לנו מה לבדוק? אבל ביסוד הגישה, אנחנו אומרים שמה שיש לנו לבדוק הוא לא אם הקוד הוא נכון, אלא אם הוא עומד בדרישות הלקוח. אם כתבנו תוכנית מאוד יפה לחישוב מאוד מתקדם, ואנחנו צריכים רק לעשות חיבור וחיסור זה קצת מיותר. וכן אם כתבנו תוכנית, וכשהגשנו אותה ללקוח הסתבר לנו שבכלל הוא התכוון למשהו אחר, בכל המקרים האלו לא יועיל לנוכל הטסטים שכתבנו.

יתרונות השיטה הם רבים:

- מוקד הפיתוח הוא עמידה בדרישות – ברגע שנתמקד בדרישות, נכתוב את הטסטים בהתאם, והפיתוח יהיה יותר מדויק.
- מוציא את המתכנת מעמדה של problem solver ומעביר אותו לעמדה של הלקוח – המתכנת כבר לא יצטרך כל פעם לגלות פגם קטן ואז לתקן, אלא ידע יותר מה הוא אמור לעשות ויעמוד בזה.
- מאפשר תקשורת ברורה בין חברי צוות – כשנכתוב קודם את הטסים, נוכל להסתכל על מה שכתבנו ולדעת בבירור האם הוא משרת את המטרת של הפרוייקט או לא.
- מייצר קוד אפקטיבי יותר – כותבים רק מה שצריך – ברגע שאנחנו מנסים לפתור דברים נקודתיים, אנחנו לא מתפזרים לכתיבה של קוד מסובך ומיותר.
- מוודא שהבדיקות אכן מבצעות פעולה משמעותית – כשנחשוב על הטסטים מראש, לא יווצר לנו מצב שאנחנו כותבים טסטים רק לצאת ידי חובה, ובמקררים כאלה נכתוב רק דברים שברור לנו שיעברו.

כמובן שיש גם חסרונות, אנחנו עובדים הרבה עוד לפני שהתחלנו לעבוד. אבל בגדול, השיטה הזאת חוסכת לנו את כל הבלאגן של לעבוד כפול ולתקן באגים בגלל שלא עמדנו בדרישות.

גם פה נעשה את הדיון דרך דוגמא של Pair Programming של ג'רי ואלפונס, רק שפחות נתמקד במערת היחסים ביניהם, ויותר נתמקד בהלוך-חזור של הקוד.

התוכנית שהם נתבקשו לכתוב היא תוכנית למציאת מחלקים – בעבור כל מספר, על התכנית להוציא מערך של מספרים ראשוניים המהווים את המחלקים של אותו מספר. למשל בעבור המספר 12, התוצאה תהיה [2,3,4] (כמובן ש-6 לא יהיה פה מאחר והוא לא ראשוני). הדרך שהתוכנית תעבוד בו היא די פשוטה – המספר שיוכנס, יעבור לתכנית שרשמנו בחלק הקודם, ומה שיחזור לנו יהיה רשימת כל הראשוניים בטווח הרלוונטי, ומשם נתחיל לבדוק.

הקוד הראשוני שלנו יהיה זה –

```
public static int[] findFactors(int multiple) {
  List factors = new LinkedList();                          // יצירת קונטיינר
  int[] primes = PrimeGenerator.generatePrimes(multiple);   // יצירת מערך ראשוניים בטווח הנתון
  for (int i = 0; i < primes.length; i++)                   // ריצה על כל המערך
    for (; multiple % primes[i] == 0; multiple /= primes[i])
      factors.add(new Integer(primes[i]));                  // הוספת המחלקים לרשימה
  return createFactorArray(factors);                        // שליחה לפונקציה שממירה למערך
}
```

```
private static int[] createFactorArray(List factors) {
  int factorArray[] = new int[factors.size()];
  int j = 0;
  for (Iterator fi = factors.iterator(); fi.hasNext();) {
    Integer factor = (Integer) fi.next();
    factorArray[j++] = factor.intValue();
  }
  return factorArray;
}
```

עכשיו אנחנו נכתוב את הטסט בצורה המינימלית ביותר. אנחנו לא נתחיל לבדוק ישר ממספרים אסטרונומיים, אלא נתחיל מהמספר הכי נמוך – 2. בעבורו נכתוב את הטסט הבא –

```
public void testTwo() throws Exception {
  int factors[] = PrimeFactorizer.factor(2);
  assertEquals(1, factors.length);
  assertEquals(2, factors[0]);
}
```

לצערנו זה לא עובד, ולכן אנחנו מתחילים לכתוב **הכל** מחדש, אך הפעם נעשה זאת בשיטת ה-TDD.

נתחיל מהדרישות באופן המינימלי ביותר – בעבור המספר 2, אנחנו אמורים להחזיר מערך עם המספר 2.

ולכן נכתוב את הקוד הבא –

```
public static int[] factor(int multiple) {
  return new int[] {2};
}
```

Tests passed

בעבור 2, יחזור לנו מערך עם המספר 2. הלאה.
המספר 3, שגם הוא בסיסי יחסית, יחזיר גם כן את עצמו. נכתוב את הטסט הבא על מנת לוודא את זה –

```
public void testThree() throws Exception {
  int factors[] = PrimeFactorizer.factor(3);
  assertEquals(1, factors.length);
  assertEquals(3, factors[0]);
}
```

ואז נעדכן את הקוד גם עבור הדרישה הזאת –

```
public static int[] factor(int multiple) {
  if (multiple == 2) return new int[] {2};
  else return new int[] {3};
}
```

עכשיו אנחנו חושבים לעצמנו כך – אם הדרישות בינתיים הם רק לבדוק את 2 ואת 3, ובכל פעם אנחנו מחזירים את אותו מספר, אז לא יותר טוב פשוט להחזיר מערך עם המספר? שמחחים וטובי לב אנחנו כותבים את הקוד הבא –

```
public static int[] factor(int multiple) {
    return new int[] {multiple};
}
```

עד כאן כיסינו את כל המקרים הדרושים. הלאה – 4. איך נעמוד בטסט הבא?

```
public void testFour() throws Exception {
    int factors[] = PrimeFactorizer.factor(4);
    assertEquals(2, factors.length);
    assertEquals(2, factors[0]);
    assertEquals(2, factors[1]);
}
```

פה אנחנו כבר מבקשים יותר – לא רק להחזיר מערך עם המספר, אלא להחזיר מערך בגודל 2, שבכל תא יוחזק הערך 2. בשביל זה, אנחנו נעבור על הקוד ונאלץ להאריך אותו בקצת –

```
public static int[] factor(int multiple) {
    int currentFactor = 0;
    int factorRegister[] = new int[2];
    for (; (multiple % 2) == 0; multiple /= 2)
      factorRegister[currentFactor++] = 2;
    if (multiple != 1)
      factorRegister[currentFactor++] = multiple;
    int factors[] = new int[currentFactor];
    for (int i = 0; i < currentFactor; i++)
      factors[i] = factorRegister[i];
    return factors;
}
```

בגדול, ברגע שיהיה לנו 4, אנחנו ניצור מערך מתאים, ואחרת (2 או 3) אנחנו נחזיר את המספר הנכון. הבעיה היא שיש פה קצת בלאגן, ולכן אנחנו עוברים ועושים ריפקטור קטן לקוד לפני שממשיכים. כדאי לשים לב שאת הריפקטור אנחנו עושים כבר עכשיו, ולא רק בסוף כתיבת התוכנית. מה שיצא לנו זה התוכנית הבאה –

```
public class PrimeFactorizer {
    private static int factorIndex;
    private static int[] factorRegister;

    public static int[] factor(int multiple) {
        initialize();
        findPrimeFactors(multiple);
        return copyToResult();
    }

    private static void initialize() {
        factorIndex = 0;
        factorRegister = new int[2];
    }

    private static void findPrimeFactors(int multiple) {
        for (; (multiple % 2) == 0; multiple /= 2)
            factorRegister[factorIndex++] = 2;
        if (multiple != 1)
```

```
            factorRegister[factorIndex++] = multiple;

    }

    private static int[] copyToResult() {
        int factors[] = new int[factorIndex];
        for (int i = 0; i < factorIndex; i++)
            factors[i] = factorRegister[i];
        return factors;
    }
}
```

עכשיו אנחנו עוברים לכתיבת הטסט הבא, אנחנו בודקים את הספרה הבאה – 5.

```
@Test
public void testFive() throws Exception {
    int factors[] = PrimeFactorizer.factor(5);
    assertEquals(1, factors.length);
    assertEquals(5, factors[0]);
}
```

במפתיע גם זה עובד. למה זה מפתיע? כי לא עשינו שום שינוי בקוד, אבל בגלל שמדובר במספר ראשוני, אנחנו מבינים שמה שחוזר לנו זה בדיוק אותו ערך, ובעצם אנחנו יכולים להסיק מזה שאנחנו כיסינו את כל המספרים הראשוניים, לכאורה.

עכשיו יש לנו לבדוק את 6. הוא לא ראשוני, אבל הוא גם לא 4 שברור לנו מה יחזור. אנחנו כותבים טסט שבודק גם את התוצאה וגם את תוכן ערך המוחזר –

```
@Test
public void testSix() throws Exception {
    int factors[] = PrimeFactorizer.factor(6);
    assertEquals(2, factors.length);
    assertContains(factors, 2);
    assertContains(factors, 3);
}

private void assertContains(int factors[], int n) {
    String error = "assertContains:" + n;
    for (int i = 0; i < factors.length; i++) {
        if (factors[i] == n)
            return;
    }
    fail(error);
}
```

להפתעתנו, זה עובד. אך האם זה באמת כיסה לנו את כל המקרים? 7 הוא מספר ראשוני, ואנחנו כבר ביססנו את ההצלחה של זה. עכשיו נעבור ל-8. המרכיבים הראשוניים של 8 הם 2,2,2.

נכתוב את הטסט הבא –

```
@Test
public void testEight() throws Exception {
    int factors[] = PrimeFactorizer.factor(8);
    assertEquals(3, factors.length);
    assertContainsMany(factors, 3, 2);
}
private void assertContainsMany(int factors[], int n, int f) {

    String error = "assertContains(" + n + "," + f +")";
    int count = 0;
```

```
    for (int i = 0; i < factors.length; i++) {
        if (factors[i] == f)
            count++;
    }
    if (count != n)
        fail(error);
}
```

כשנריץ את הטסט, אנחנו ניכשל.

```
java.lang.ArrayIndexOutOfBoundsException: 2
```

אנחנו מצפים שיחזור לנו מערך בגודל 3, אבל הגדרנו שהמערך יהיה בגודל 2, אז נזרקת לנו חריגה. אנחנו יכולים פשוט ליצור את המערך בגודל 100, ואז להיות בטוחים שגם אם יהיה לנו 2^{64} אנחנו נהיה בטוחים, כי המקסימום שנצטרך הוא לוג של המספר הזה, שהוא 64, אבל זה עדיין לא יספיק לנו. אם נבדוק את 9, למשל –

```
@Test
public void testNine() throws Exception {
    int factors[] = PrimeFactorizer.factor(9);
    assertEquals(2, factors.length);
    assertContainsMany(factors, 2, 3);
}
```

גם הטסט הזה ייכשל, כי מה שיחזור לנו הוא מערך הגודל 1 רק עם המספר 9, ואנחנו מצפים לקבל [3,3].

לכן נשנה את הקוד לדבר הבא –

```
private static void findPrimeFactors(int multiple) {
    for (int factor = 2; multiple != 1; factor++)
        for (; (multiple % factor) == 0; multiple /= factor)
            factorRegister[factorIndex++] = factor;
}
```

כמובן, שזה רק הפונקציה שמוציאה לנו את המחלקים הראשוניים, והשינוי שנעשה הכניס לנו את המספרים הנכונים, ועכשיו כל הטסטים עוברים בהצלחה.

עכשיו נבדוק 1000!!!

```
@Test
public void testThousand() throws Exception {
    int factors[] = PrimeFactorizer.factor(1000);
    assertEquals(6, factors.length);
    assertContainsMany(factors, 3, 2);
    assertContainsMany(factors, 3, 5);
}
```

גם זה עובד!

אפשר לומר שסיימנו, ועכשיו רק נותר לסכם מה למדנו היום –

תכנות מונחה בדיקות לוקח את הדרישה ומפרק אותה לרמה הכי מינימלית האפשרית. רק לאחר שאנחנו יודעים מה המינימום, אנחנו יכולים להתחיל לכתוב את הקוד המתאים. אנחנו כותבים קוד שיתייחס רק לטסט המינימלי שאותו אנחנו בודקים, עבר? מעולה. לא עבר? נתקן.

לאחר מכן, אנחנו מתחילים להרחיב את הטסטים, מוסיפים בכל פעם את הפונקציונליות המינימלית, ומוודאים שאנחנו מרוויחים מידע בכל פעם – אם אנחנו מגלים שאנחנו מצליחים במה שכתבנו לפתור תחום גדול יותר (למשל, את בדיקת כל הראשוניים) אנחנו לא צריכים להמשיך ולבדוק את זה.

חשוב מידי פעם לעשות ריפקטורינג לקוד תוך כדי סבבי הבדיקות, לפעמים זה יעזור גם בכך שזה ירחיב את הפונקציונליות של הקוד ויפתור לנו בעיות עתידיות, וגם כי אנחנו לא מתכוונים לכתוב בצורה הכי יבשה את כל המקרים בקוד, אלא אנחנו שואפים לאבסטרקציה.

לאחר שעשינו ריפקטורינג, אנחנו חוזרים ובודקים שהכל עובד, וממשיכים להרחיב את הטסטים ואת התכנית שלנו עד שהכל עובד ונראה טוב.

הידד.

יש בהמשך מקרה בוחן נוסף למשחק באולינג, אבל אני לא ממש שולט בחוקים אז לא הבנתי מה הולך שם.