האוניברסיטה הפתוחה

המחלקה למתמטיקה ולמדעי המחשב

דפוסי תכן (תבניות עיצוב) בהנדסת תוכנה: מתיאוריה לשימוש בתעשייה

Design Patterns in Software Engineering: From Theory to Production

עבודה מסכמת זו הוגשה כחלק מהדרישות לקבלת תואר

"מוסמך למדעים" M.Sc. במדעי המחשב

באוניברסיטה הפתוחה

החטיבה למדעי המחשב

על-ידי

אלי איזק

ת.ז. 029475175

דוא"ל: eisaak123@gmail.com

העבודה הוכנה בהדרכתו של ד"ר דן אהרוני

הצעה מאושרת הוכנה בהדרכתה של ד"ר ציפי ארליך

תאריך: 24/01/2013

תודות

- תודה רבה לד״ר דן אהרוני שהנחה אותי במסירות בהכנת העבודה המסכמת.
- תודה רבה לד״ר ציפי ארליך שהנחתה אותי במסירות בהכנת ההצעה לעבודה המסכמת.
- תודה רבה לפרופסור דוד לורנץ על הערותיו המקצועיות.

תוכן העניינים

רשימת טבלאות ואיורים

טבלאות

איורים

תקציר

"דפוס תֶּכֶן" (design patterns) הוא תבנית (או דפוס) כללית המשמשת פתרון כללי לבעיה שכיחה בעיצוב תוכנה שעשויה להיות שימושית לעיצוב תוכנה ביישומים רבים.

העבודה מתמקדת ב"דפוסי תכן מונחי עצמים" (object oriented design patterns).
ל"דפוסי תכן מונחי עצמים" יש כיום משקל רב בתעשיית התוכנה, למשל נעשה בהם שימוש רב במסגרות-עבודה (frameworks). העבודה מתארת איך זה קרה ומה ההצדקה לכך.

העבודה מציגה את הצורך המעשי בדפוסי תכן וכיצד כדאי ללמוד ולעבוד איתם באופן נכון. מוסבר כיצד דפוסי תכן תורמים לשיפור יישום עקרונות הנדסת תוכנה תוך שימוש בדוגמאות מעשיות ודוגמאות מתעשיית התוכנה. חלק מעקרונות הנדסת התוכנה שמוצגים הם: שימוש בהרכבה והאצלה במקום בהורשה, Single Responsibility Principle (SRP), Open Close Principle (OCP), Inversion of Control (IoC), Liskov Substitustion Principle (LSP).

כמו כן, העבודה סוקרת בקצרה נושאים קרובים כמו:

- דפוסי תכן של ארכיטקטורת תוכנה שהם דפוסי תכן שמספקים תבניות לארכיטקטים של תוכנה לעיצוב מערכות מורכבות של תוכנה מקצה לקצה.
- דפוסי-נגד מונחי עצמים (Object Oriented Anti Patterns) שהם דפוסי תכן שמלמדים אותנו להימנע מעיצוב מונחה עצמים שגוי.

1 מבוא

בפרק זה אביא הסבר מקדים על "דפוסי תכן" ועל "דפוסי תכן מונחי עצמים" ואז אתאר בקצרה את מטרת העבודה ומה צפוי בהמשך העבודה.

1.1 דפוסי תֶּכֶן

נתחיל במשמעות הלשונית של המושג "דפוסי תֶּכֶן" (design patterns). התרגום של המילה "תֶּכֶן" באנגלית הינו Design. במקום "תכן", ניתן גם להשתמש במילה "עיצוב" במשמעות של design ולא styling. במקום המונח "דפוס" (pattern), ניתן להשתמש במונח "תבנית" ואכן דפוסי תכן נקראים גם בשמם הנפוץ "תבניות עיצוב".

כתיאור ראשוני של "דפוס תכן" (הגדרה תובא בעמוד 15) בהקשר של הנדסת תוכנה, נאמר שזאת תבנית כללית המשמשת פתרון כללי לבעיה שכיחה בעיצוב תוכנה שעשויה להיות שימושית ביישומים רבים. אבל דפוסי תכן עוזרים לשיפור הנדסי גם בתחומים רבים אחרים. דפוסי תכן שימשו בבנייה ועיצוב ערים [Alex], [Ais] ורק בהמשך הנושא תפס תאוצה בעולם התוכנה כפי שנראה בהמשך. דפוסי תכן משמשים אפילו בעיצוב פדגוגי של שיעור [Berg].

שימוש בדפוסי תכן תורם ליישום של עקרונות בהנדסת תוכנה בכך שהוא מספק תבניות הנדסיות כלליות הבנויות על עקרונות של הנדסת תוכנה שמפתח התוכנה יכול להשתמש בהן. וזאת כדי למנוע טעויות בעיצוב ההנדסי של התוכנה [Weis], [Mha] ולמנוע פיתוח חוזר של תבניות שכבר פותחו.

תוכנה צריכה להימדד לא רק על פי ההתנהגות שמוכתבת ע"י האלגוריתם שלה, אלא גם לפי היבטים של עיצוב כמו עמידות לשינוי וגמישות להרחבה של בסיס הקוד ושל של מספר המשתמשים במקביל (Scalability). כאן באים לעזור דפוסי התכן [Mgj].

1.2 דפוסי תכן מונחי עצמים

בדרך כלל דפוסי התכן לא תלויים בשפת מחשב מסוימת, אבל לרוב דפוסי התכן נעזרים במתודולוגיה של עיצוב מונחה עצמים [Sid]. לכן, בעבודה זו אתמקד בעיקר בדפוסי תכן מונחי עצמים. כפי שאציג בהמשך, דפוסי תכן מונחי עצמים מציגים שילוב מיטבי של יסודות הנדסיים חשובים של עיצוב מונחה עצמים לצורך יצירת כל דפוס תכן.

דפוסי תכן לא חייבים להיות מבוססים על עיצוב מונחה עצמים אבל אם לא היה נעשה שימוש בעיצוב מונחה עצמים ליצירת דפוסי תכן רבים אז פשוט היו קוראים ליסודות של עיצוב מונחה עצמים בשם דפוסי תכן – למשל הורשה ופולימורפיזם. ואז שוב אפשר היה להשתמש ביסודות אלו כדפוסי תכן בסיסיים לצורך יצירת דפוסי התכן האחרים [Gof]. נאמר גם שעיצוב מונחה עצמים נותן את אבני היסוד ללימוד דפוסי תכן וכדאי גם שלא להפריד בין נושאים אלו וללמוד אותם ביחד [Ppp].

שפת UML , שתוצג בקצרה בהמשך, עוזרת לנו להציג את דפוסי התכן מונחי העצמים.

1.3 מטרת העבודה

בעבודה זאת אסקור באופן מקיף את הנושא החשוב בהנדסת תוכנה שנקרא "דפוסי תכן מונחי עצמים" (Object Oriented Design Patterns).
מטרתי היא להציג את הנושא לעומק בהיבטים של הנדסת תוכנה ולא להסתפק בסקירה שיטחית של דוגמאות. שהרי, לימוד שיטחי של דוגמאות, בסופו של דבר מוביל ליצירת קוד תוכנה גרוע (קשה להבנה ולתחזוקה). זה קורה בגלל שימוש לא ראוי בכלים שנעשה בהם שימוש מבלי להבין אותם היטב.

1.4 מה בהמשך העבודה

בפרק 2 אציג רקע היסטורי, שיעזור לתת מוטיבציה ללימוד הנושא.

בפרקים 3 ו-4 אתן הגדרה והצדקה.

בפרק 5 נדון בסוגיות שונות לצורך הרחבה בנושא. נראה כיצד "דפוסי תכן" תורמים לשיפור יישום עקרונות הנדסת תוכנה נראה שיטות הקטלוג של "דפוסי התכן" ואת הקשרים ביניהם ואתאר את המורכבות של "דפוסי תכן" שונים ואת הגישות ללימוד שלהם ותחילת עבודה איתם.

בפרק 6, כהכנה לנושא "דפוסי תכן מונחי עצמים" בו מתמקדת עבודה זאת, אתן סקירת ריענון קצרה על השימוש בתרשימי UML שמציגים בצורה גראפית זוויות ראיה שונות על כל עיצוב מונחה עצמים – למשל תרשים מחלקות (class diagram) הינו תרשים סטטי שמתאר את המבנה של העיצוב וקוד התוכנה, לעומת תרשים רצף (sequence diagram) שהינו תרשים דינאמי שמציג את המסרים הנשלחים בין עצמים של אותן מחלקות.

בפרק 7 אציג דפוסי תכן נבחרים.

בפרק 8 אציג שימוש בדפוסי תכן במסגרות-עבודה ובסביבות פיתוח בתעשייה. למשל, אביא דוגמאות מעשיות מהתעשייה בה אני עובד כמפתח תוכנה – למשל כיצד מסגרת-עבודה (framework) משתמשת בדפוסי תכן.

בפרק 9 נראה דוגמאות כיצד להשתמש בדפוסי תכן במהלך פיתוח תוכנה.

לבסוף, בפרק 10, אציג את הנושא המשלים שנקרא "דפוסי-נגד מונחי עצמים" (object oriented anti patterns). אלו הם דפוסי תכן שמלמדים אותנו להימנע מעיצוב מונחה עצמים שגוי.

עבודה זאת תלווה בפרויקט מתקדם מעשי (המתבצע במסגרת הקורס "פרויקט מתקדם במדעי המחשב") שידגים שימוש מעשי ביסודות התיאורטיים של הנדסת התוכנה שיתוארו כאן.
הפרויקט המעשי יהיה תוכנת ציור. מטרת הפרויקט לתת דוגמה של שימוש נכון בדפוסי תכן בקוד התוכנה ובמסמכי העיצוב. כלומר, הפרויקט יעוצב בעזרת מסמך עיצוב שיעשה שימוש בתרשימי UML. במסמך העיצוב יודגם ויינתן הסבר על דפוסי תכן שאיעזר בהם לצורך פיתוח הפרויקט. הפרויקט יפותח בשפת התכנות מונחית העצמים שנפוצה מאוד בתעשייה – שפת Java. שפה זאת הורחבה בעצמה כך שתתמוך בדפוסי תכן – כמו למשל תמיכה בדפוס התכן Observer שאביא כדוגמא בהמשך.

2 רקע היסטורי

2.1 מאיזה שורשים צמחו דפוסי התכן

נושא דפוסי התכן התפתח בתחומי הנדסה שונים עוד הרבה לפני השימוש בהם בתעשיית התוכנה, למשל באדריכלות. קיימים דפוסי תכן לבניית בתים שנבנים לפי דפוס מסוים (למשל, חלונות נבנים בגובה מסוים וחדר שינה בדרך כלל לא ממוקם ליד דלת הכניסה לבית).

בספר [Part] ניתנת סקירה היסטורית שמספרת על האדריכל כריסטופר אלכסנדר (שמוכר כאחד מאבות תורת השימוש בדפוסי תכן) שחקר מבנים שנבנו באיכות גבוהה ואסף מהם שיטות בנייה שנועדו לפתור בעיה דומה ואז מצא את הדברים שחוזרים על עצמם בשיטות בנייה אלו. בספרים שפרסם כריסטופר אלכסנדר הוא קרא לדברים הדומים האלו בשם דפוסים (patterns). כאמור, אלכסנדר עסק בדפוסים שקשורים לאדריכלות של מבנים כמו בניינים, גנים וכבישים.

כפי שהוא מצוטט במקורות רבים (בתעשיית התוכנה), אלכסנדר כתב את המשפט המפורסם הבא [Ais]:

"The elements of this language are entities called patterns. Each pattern describes a problem that occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice." (p. 10)

2.2 התפתחות דפוסי התכן בהנדסת תוכנה

התפתחות דפוסי התכן בהנדסת תוכנה היא התפתחות טבעית והכרחית. למשל, אחד מהיסודות החשובים של הנדסת תוכנה שממומש ע״י דפוסי תכן הוא כימוס (encapsulation) של קוד תוכנה שמועד לשינוי. למשל, בעיצוב מונחה עצמים, מחלקה היא ישות בסיסית שמחביאה את המימוש שלה (עושה לו כימוס) מהלקוחות שלה שנעזרים בשירותים שהיא מפרסמת. שהרי זמן פיתוח תוכנה נחשב לזמן קצר מאוד יחסית לזמן התמיכה והתיקונים, שהוא הזמן הארוך (שזה משפיע גם על עלות התוכנה). לכן אם נעשה כימוס של אספקטים של התוכנה שמועדים לשינוי (ארחיב על כך בהמשך עבודה זאת) אז ברגע שנרצה לשנות את הקוד, נוכל לצמצם את השינוי לתחום קטן (ובכך נחסוך במשאבים: זמן, כסף, מיקוד הארגון, כוח אדם ועוד). לקחים מפרויקט דומה לדוגמה שפותח בעבר ניתן למצוא במאמר [Dgd].

נושא דפוסי התכן קיבל זריקת מרץ גדולה עם פרסום הספר [Gof] (כפי שאתאר בהמשך), אך כפי שספר זה מעיד בעצמו, דפוסי תכן רבים התפתחו במהלך העבודה של אנשי פיתוח בתעשייה והרבה פעמים פורסמו במבוזר ובאופן לא רשמי (למשל בפורומים באינטרנט). דפוסי התכן המקובלים הם אלה ששרדו את מבחני הקבלה של תעשיית התוכנה לאורך השנים ובמשך הזמן הם שופרו ולוטשו (כלומר, דפוס תכן שהומצא עכשיו לא יוכל להתקבל כדפוס מקובל בתעשייה באופן מיידי).

בשנת 1994 פרסמו ארבעה חוקרים שידועים ביחד בקיצור בשם החיבה המפורסם Gang of Four או בקצרה Gof את הספר [Gof] שנחשב תנ״ך בתחום.
עבודת המחקר שלהם ארכה יותר מ-4 שנים והם קיטלגו 23 דפוסי תכן בספר שלהם.

אבל כפי שנאמר לעיל, דפוסי התכן התפתחו בתעשיית התכנה הרבה לפני פירסום הספר [Gof]. למשל, בשנת 1987, אנשי התוכנה קנט בק ו- וארד קנינגהם שהושפעו מספריו של האדריכל אלכסנדר פיתחו דפוסי תכן לעיצוב תוכנה לצורך בניית מנשקי משתמש והציגו מצגת בכנס OOPSLA (כנס שעוסק בעיצוב מונחה עצמים) ומאז הרבה מאמרים ומצגות בנושא נתרמו ע״י אנשי עיצוב מונחה עצמים. לאחר פרסום הספר [Gof] נכתבו עוד מאמרים וספרים רבים בנושא. בתעשייה, נעשה שימוש נרחב בדפוסי תכן, למשל בתוך מסגרות עבודה שלרוב מממשות דפוסי תכן [Acd].

3 הגדרה

בפרק המבוא הצגתי את המשמעות הלשונית של "דפוס תכן". עתה נגדיר מה יוכל להיחשב כדפוס תכן.

נגדיר דפוס תכן כדרך מופשטת לפיתרון של בעיה כללית של עיצוב תוכנה בהקשר (context) נתון, כאשר דפוס התכן צריך לעמוד בדרישות הבאות [Alex, Gof]:

- ניתן לשימוש חוזר לפיתרון בעיות עיצוב תוכנה רבות ושונות
- צריך להשתמש באופן יעיל ואלגנטי בעקרונות עיצוב נכון של תוכנה כפי שנראה בהמשך
- צריך לתאר את היתרונות והחסרונות שנותן הפיתרון
- חייב לשרוד את מבחן הזמן והביקורות מצד אנשי המקצוע והקהילה האקדמית

3.1 הערות משלימות להגדרה

בדפוסי תכן נעשה שימוש חוזר בפרקטיקה טובה תוך העברת ידע בין אנשי המקצוע. דפוסי התכן משמשים דוגמאות טובות לדרכים יעילות ואלגנטיות לפתרון בעיות [Mull].

הוזכר בהגדרה לעיל מושג הַהֶקְשֵׁר (context). האדריכל כריסטופר אלכסנדר בספרו [Alex] מגדיר דפוס תכן כשלישייה שמגדירה את הַקֶּשֶׁר בין הֶקְשֵׁר מסוים, בעיה והפיתרון. חשוב מאוד לקחת בחשבון את נושא ההקשר כי הטמעת דפוס תכן בתוכנה מבלי להתחשב בהקשר גורמת להוספת סיבוך ללא צורך במקרה הטוב [Wick].

להלן הדרך המקובלת להציג דפוס תכן [Abcm]. כך גם אציג דפוסי תכן נבחרים בעבודה זאת:

1. שם הדפוס – משמש כנקודת אחיזה ומרחיב את אוצר המילים של המשתמש.
2. הצגת ההקשר והבעיה.
3. הפיתרון.
4. התוצאות של הפיתרון ותאור היתרונות והחסרונות.

נסייג את ההגדרה בכך שההחלטה מה לא יכול להיחשב כדפוס תכן תלויה בנקודת המבט של כל אחד – "מה שאדם אחד יראה כדפוס תכן, אדם אחר יראה כבלוק בנייה פרימיטיבי" [Gof]. למשל, מבנה הנתונים "רשימות מקושרות" לא נחשב לדפוס תכן על פי [Gof] ויש הטוענים ש-Singleton איננו דפוס תכן (בניגוד ל-Gof).

4 הצדקה לשימוש בדפוסי תכן

השימוש בדפוסי תכן בהנדסת תוכנה נותן יתרונות רבים. להלן חלק מהיתרונות שמתקבלים, תוך התייחסות לדפוסי תכן מונחי עצמים (אספתי יתרונות אלו ממגוון מקורות):

1. **שימוש בניסיון** - דפוסי תכן נועדו ללמוד מניסיונם של מפתחים אחרים [Gof].

2. **שימוש נכון בכלים הנדסיים** - קל מאוד להשתמש באופן שגוי ביסודות עיצוב מונחה עצמים (הורשה, אגריגציה, קומפוזיציה, פולימורפיזם, כימוס, מנשקים ומימושם) כך שנוצרת תוכנה קשה מאוד להבנה ולתחזוקה – דפוסי התכן עוזרים לנו לשלב יסודות אלו בצורה מיטבית בתוכנה שאנו מפתחים. דוגמא טובה נוספת היא עמידה **בעיקרון ה-Open-Close Principle** (OCP) - מחלקה צריכה להיות פתוחה להרחבות אך סגורה לשינויים [Bert]. כלומר, כאשר נרצה לשנות דברים אז הקוד של המחלקה צריך שלא להשתנות. מה שכן נשנה יהיה תוספת של מחלקות חדשות ע"י שימוש בהורשה ובהאצלה. את זה נשיג ע"י כך שכאשר נכתוב את המחלקה, נדאג לעשות לה כימוס (לחשוף ללקוח רק את המנשק ההכרחי אליו) ונדאג שהיא תיהיה רכיב מודולרי קטן. כלומר, המחלקה תיהיה אחראית על נושא אחד בודד בלבד וזאת על-פי **עיקרון ה-Single Responsibility Principle** (SRP).

3. **שימוש באוצר מילים משותף** - במקום שנצטרך לתאר חלקים ממערכת התוכנה (מונחית העצמים) שלנו באופן מפורט אנו יכולים בסה"כ לומר את שם דפוס התכן – בכך חסכנו למשל פירוט של כל המחלקות המשתתפות, המידע שהן מחזיקות, הפעולות שהן מבצעות, אופן התקשורת בין המחלקות, היחסים (למשל הורשה, הכלה, מימוש) בין המחלקות והמנשקים ועוד [Fbs].

4. **הרחבת השימוש החוזר** (reuse) - חלקים גדולים מהתוכנה שלנו יהיו בעלי מבנה דומה להרבה תוכנות אחרות שלנו ושל אחרים ולא רק ברמה של שימוש חוזר במחלקה (למשל ע"י הורשה) – דבר שיקל מאוד על תיעוד התוכנה, בדיקת התוכנה ותחזוקתה [Njga].

5. **הסתרה של אספקטים בעיצוב שסביר שישתנו** – יורחב על כך בפרק "עקרונות עיצוב נכון של מערכת תוכנה" בעמוד 19.

6. **צימוד חלש** - יש לשאוף לצימוד (coupling) חלש בין עצמים/מודולים שמשתפים פעולה. המושג "צימוד" כולל את האפשרויות הבאות שמובאות בסולם של רמות צימוד מהצימוד החלש ביותר (למעשה חוסר צימוד) ועד לצימוד החזק ביותר [Knob]:

   - חוסר צימוד – קטע קוד אחד אינו תלוי בקטע קוד אחר וניתן לשנות כל אחד מהם מבלי לשנות את האחר.

   - צימוד חתימה (signature coupling) - קוד תלוי בחתימה של מחלקה מסוימת, כלומר תלות בחתימה של המתודות של המחלקה ובזיהוי המלא של המחלקה, שניתן ע"י השם המלא שלה

[package+name] ובעץ המחלקות שהיא יורשת ממנו. ניתן לפתור זאת ע"י שימוש במנשק (interface).

- צימוד נתונים (data coupling) – מודולים שונים ניגשים יחד לנתונים מאותה קטגוריה. נרצה שכל מודול יטפל בנתונים שבתחום אחריותו בלבד. למשל, רק מודול "ניהול סטודנטים יטפל בעדכון נתוני הסטודנטים".

- צימוד בקרה (control coupling) – מודול אחד שולח נתוני בקרה להשפעה על פעולת מודול אחר. למשל, מודול X שולח למודול Y מספר בתחום [1..3] ואז מודול Y משנה את אופן פעולתו על פי מספר זה.

- צימוד חיצוני (external coupling) – תלות בתוכנת צד שלישי מסוימת או בחומרה מסוימת.

- צימוד משותף (common coupling) – מודולים ניגשים ישירות לאותו משאב (למשל תא זיכרון).

- צימוד תוכן (content coupling) – מודול ניגש ישירות לנתונים פנימיים של מודול אחר.

7. **הפשטה** – דפוסי התכן מאפשרים לנו לחשוב ברמה גבוהה יותר של דפוסים במקום לחשוב ברמה נמוכה יותר של מימוש שעוסקת בכל הפרטים הקטנים [Fbs].

5 סוגיות בדפוסי תכן — סקירת ספרות, הערות והארות

5.1 עקרונות עיצוב נכון של מערכת תוכנה

עיצוב נכון של מערכות תוכנה הוא דבר חשוב מאוד בתעשיית התוכנה. עיצוב נכון יעזור להפחית את העלות הגבוהה של תחזוקת המערכת. בעיקר מדובר על מערכות תוכנה גדולות ומורכבות – אם מדובר בתוכנה קטנה ופשוטה אז לפעמים כדאי דווקא להימנע משימוש בכלים שיובאו כאן, שהרי יש מחיר תקורה (overhead) לשימוש בכלים אלו.

להלן עקרונות עיצוב בסידור עולה של מורכבות ורמת מקצועיות, כפי שאני רואה אותם:

1. **ללא עיצוב** – קוד לא מודולרי שמוביל ל"קוד ספגטי" (בהמשך נראה שזהו שמו של Anti-Pattern מפורסם) ולשימוש במשתנים גלובליים לצורך תקשורת בין חלקי הקוד. במשך הזמן יהיה קשה יותר ויותר לתחזק את הקוד (כלומר לשנות, להרחיב, לתקן, לתעד, לבדוק).

2. **חלוקת מערכת תוכנה לרכיבים (components) שמתקשרים ביניהם** – המודולריות גדלה ע"י צמצום השימוש במשתנים גלובליים ושימוש במנשקים מוגדרים היטב בין הרכיבים.

3. **עיצוב פרוצדוראלי** – המודולריות גדלה גם בתוך כל רכיב/תת-מערכת ע"י צמצום התחום (scope) של המשתנים ושל זמן החיים שלהם, שימוש חוזר בקוד, שימוש במנשקים מוגדרים היטב בתוך תת-המערכת.

4. **מודולים של קוד** – יוצרים מודולים של קוד בתוך תת-מערכת תוכנה. המודולריות גדלה בתוך תת-המערכת בכך שהתחום של המשתנים הגלובליים וגם של הפרוצדורות מצטמצם לתוך המודול.

5. **עיצוב מונחה עצמים** - אנחנו מקבלים הרחבת רמת ההפשטה של הקוד (מערכת התוכנה היא עתה מודל של מציאות מסוימת שמורכבת מקבוצה של עצמים שעובדים יחד להשגת מטרות מסוימות), מודולריות (למשל ע"י הרחבת השימוש בכימוס ובמנשקים), הורשה (הרחבת אופן השימוש החוזר לעומת שימוש רק בפרוצדורות), פולימורפיזם.

6. **שימוש בעקרונות עיצוב עיצוב מונחה עצמים נכון** -

למשל, נשתמש בעקרונות העיצוב החשובים הבאים [Fbs]:

- עיצוב/תכנות מול מנשק ולא מול מימוש – עוזר לפיתוח של קוד מודולארי.
- העדפת הרכבה (composition) והאצלה (delegation) על פני שימוש בהורשה.

נגדיר האצלה כפי שמתואר ב-[GoF]: האצלה נוצרת כאשר עצם של מחלקה X מקבל בקשה לביצוע פעולה מסוימת. אז במקום שנשתמש בהורשה של המחלקה X למימוש ספציפי, העצם X יאציל את הטיפול בבקשה זאת ע"י כך שהוא מבקש את העזרה של עצם ממחלקה Y. וזאת כאשר עצם X מכיל את עצם Y ביחס של הרכבה (נממש זאת ע"י כך ש-X יצביע על Y). ראה יחס של הרכבה ב-"טבלה 1: היחסים בין המחלקות [Pete] (עריכה ותרגום — א.א.)".

להאצלה יש יש יתרון על יחס של הורשה בכך שניתן בזמן ריצה לגרום לעצם X להפסיק להצביע על העצם Y ולהצביע על עצם ממחלקה אחרת. כך, גרמנו בזמן ריצה שהבקשה ל-X תקבל טיפול שונה מבלי שהלקוח שעובד מול העצם X יהיה מודע לכך. כאשר אנו משתמשים בהורשה, לא נוכל לעשות שינוי זה בזמן ריצה.

הדינמיות הרבה שנותנת לנו ההאצלה היא גם חיסרון שלה. במערכות גדולות יהיה קשה יותר להבין ולתחזק את הקוד, לעומת קוד סטטי יותר.

7. **הסתרה של אספקטים בעיצוב שסביר שישתנו (ארחיב על כך מיד בהמשך):**

- Open-Close Principle (OCP) - מחלקה צריכה להיות פתוחה להרחבות אך סגורה לשינויים.
- יש לשאוף לצימוד חלש בין עצמים שמשתפים פעולה.
- Single Responsibility Principle (SRP) - צריכה להיות רק סיבה אחת לשינוי של מחלקה.

עוד עיקרון חשוב ומפורסם הוא עיקרון ה – LSP (ארחיב עליו מיד בהמשך).

8. **שימוש בדפוסי תכן** – זה נותן לנו את היתרונות שמובאים בעבודה זאת.

9. **שיתוף פעולה בין דפוסי תכן** – ארחיב על כך כמה פעמים בהמשך העבודה.

10. **שימוש במסגרת-עבודה:**

הגדרת מסגרת-עבודה [GoF]:

מדובר על מסגרת-עבודה על בסיס תוכנה מונחית עצמים (OOD software framework). מסגרת-העבודה מספקת מראש את העיצוב של הקוד שזה כולל את המחלקות ואיך הן משתפות פעולה אחת עם חברתה ואפילו את הקונפיגורציה. מפתח התוכנה נדרש רק לשתול את קטעי הקוד שלו (למשל מחלקות חדשות) לתוך הקוד של מסגרת-העבודה וזה יגרום למסגרת-העבודה לקרוא לקוד החדש במהלך ריצתה. כך, מסגרת-עבודה מדגישה שימוש חוזר בעיצוב תוכנה לעומת שימוש חוזר רק בקוד תוכנה (כאשר אנו משתמשים בספריות של קוד תוכנה אז אנחנו עושים שימוש חוזר רק בקוד של תוכנה). מסגרת-עבודה משתמשת בעיקרון ידוע בתעשיית התוכנה שנקרא "היפוך שליטה" (IoC = Inversion of Control) או בשם החיבה "העיקרון ההוליוודי" שאומר "Don't call us we will call you". היא עושה

זאת ע״י כך שבמקום שהקוד של המפתח יקרא לקוד של מסגרת-התוכנה, זה נעשה להיפך. כלומר, הקוד של מסגרת התוכנה קורא לקוד של המפתח.

כאשר אנחנו משתמשים במסגרת-עבודה איכותית אז אנחנו ״יורשים״ עקרונות עיצוב טובים וגם מסתגלים לצורת עבודה נכונה יותר תוך שימוש בעקרונות אלו (פעמים רבות מסגרת העבודה תיכפה עלינו אופן עבודה נכון יותר). מוצגת דוגמא במאמר שכתבתי במקום עבודתי [Eli] וצירפתי כנספח א׳ של עבודה זאת בעמוד 109.

אעיר כאן שוב, ששימוש בדפוסי תכן לא מחייב עיצוב מונחה עצמים. אבל יסודות העיצוב מונחה עצמים יכולים להיחשב כדפוסי תכן בסיסיים שניתן להיעזר בהם לבניית דפוסי תכן אחרים [GoF].

נרחיב על העיקרון המפורסם LSP (Liskov Substitution Principle) שנכתב ע״י ברברה ליסקוב וג׳נט וינג [Liskov].
עיקרון ה-LSP בצורתו הפורמאלית נכתב כך [Lisk]:

“Let q(x) be a property provable about objects x of type T.

Then q(y) should be true for objects y of type S where S is a subtype of T.״ [p. 1811]

או במילים פשוטות, מחלקה יורשת חייבת שלא לשנות את **ההתנהגות המקורית לה ציפה הלקוח** ממחלקת האב (אחרת, כאשר הלקוח יעשה שימוש ב-Polymorphism הוא יקבל תוצאות שהוא לא ציפה להן – כלומר תוצאות לא נכונות מבחינתו).
מפתחים רבים נוטים להשתמש באופן לא נכון ביחס של הורשה בין מחלקות, כפי שמסבירה הדוגמא הבאה שמציגה הפרה של עיקרון ה-LSP.
הדוגמא המפורסמת, שמובאת בדרך כלל בהצגת עיקרון זה, מראה כיצד ריבוע הוא אמנם סוג של מרובע **מבחינה מתמטית** אבל לא מבחינת **ההתנהגות לה מצפה הלקוח** שמשתמש במחלקה שמייצגת את הריבוע. הבעיה היא שמחלקת הריבוע יורשת את מחלקת המרובע ולכן צריכה לחשוף מנשק שמכיל מתודה לעדכון גובה הריבוע וגם מתודה לעידכון של רוחב הריבוע. אבל בפועל, המימוש של מחלקת הריבוע מעדכן בכל מתודה את אותו ערך יחיד שהינו צלע הריבוע שהרי כל צלעות הריבוע שוות. עד כאן הכל נראה בסדר, אבל אז נניח שהלקוח מקבל מערך של מרובעים מוכנים (שחלקם הם ריבועים) ומעדכן את הגובה שלהם לערך X ולאחר מכן את הרוחב שלהם לערך Y ואז הוא מדפיס את השטח שלהם. אבל השטח של הריבועים שמוצגים ללקוח כמרובעים יודפס כ-S=Y*Y למרות שהוא מצפה לקבל S=X*Y.

עתה ארחיב מעט בנושא שהזכרתי לעיל, הסתרה של אספקטים בעיצוב שסביר שישתנו. דפוסי תכן שונים עוזרים לנו להחביא אספקטים שונים בתוכנה שלנו, שמועדים לשינוי [Gof]. באופן בסיסי בעיצוב מונחה עצמים מחביאים מימוש של מחלקה אבל ניתן להחביא עוד אספקטים רבים. להלן דוגמאות המציעות דפוס תכן לכל אחד מהאספקטים שרוצים להחביא [Gof]:

1. החבאת אופן יצירת עצמים – Factory.
2. החבאת מימוש – Bridge.

3. החבאת מנשקים – Facade.

4. החבאת אופן האחסון של העצמים – Flyweight.

5. החבאת מיקום של עצם (למשל העצם יכול ליהיות במחשב אחר) – Proxy.

6. החבאה של אופן הגישה לסדרה של עצמים – Iterator.

7. החבאה של המצב הפנימי של עצם במהלך חייו – State.

8. החבאה של האלגוריתם הספציפי – Strategy.

9. החבאה של אופן התקשורת בין עצמים ועם איזה עצמים הם מתקשרים – Mediator.

5.2 קטגוריות של דפוסי תכן

נוהגים לקטלג את דפוסי התכן.למשל, נוכל לחלק אותם לקטגוריות הבאות [GoF]:

- Creational Patterns – יצירת עצמים.
  למשל דפוסי התכן: Singleton ו – Factory Method
- Structural Patterns – הרכבת עצמים או מחלקות (Composition).
  למשל דפסוי התכן: Adapter ו – Proxy.
- Behavioral Patterns – אפיון הדרך שבה המחלקות והעצמים משתפים פעולה.
  למשל דפוסי התכן: Observer ו – Mediator.

בנוסף, נוכל לתת מימד נוסף של טווח (Scope) שבעזרתו דפוסי התכן מחולקים לשתי הקטגוריות האחרות הבאות [GoF]:

1. מחלקה – דפוסים שעוסקים במחלקות ובמחלקות שיורשות מהן. הן עוסקות בקשרים סטטיים שלא משתנים לאחר שלב הקומפילציה.
   למשל: Factory Method ו - Adapter
2. עצם – דפוסים שעוסקים בקשרים שבין עצמים. קשרים אלו יכולים להשתנות במהלך זמן הריצה והם יותר דינאמיים.
   למשל: Builder ו – Flyweight.

ניתן גם לארגן את לימוד דפוסי התכן לפי אופן השימוש בהם [GoF].
למשל משתמשים בד"כ ביחד בדפוסי התכן: Composite, Iterator, Visitor.
וכן ניתן לארגן את דפוסי התכן לפי הקשרים ביניהם [GoF]. למשל, יש קשר ברור בין Composite ו-Decorator והם גם מאוד דומים (אסביר זאת כאשר אציג בהמשך את Composite).

עוד קטגוריות של דפוסי תכן [Part]:

1. Basic – למשל דפוסי התכן Monitor ו-Immutable Object
2. Concurrency – למשל דפוסי התכן Critical Section ו – Read-Write Lock

כמן כן, קיימת הקטגוריה של דפוסי תכן של ארכיטקטורות תוכנה (אגע עוד בנושא זה בהמשך בפרק " קשרים בין דפוסי התכן" שבעמוד 25, אך עבודה זאת לא עוסקת בנושא). כלומר, דפוסי תכן שבמקום להתייחס למחלקות בתת-מערכת מסויימת, הם מציגים ארכיטקטורה של מערכת תוכנה שמורכבת מתת-מערכות וכיצד הן יוצרות קשר ביניהן. בקטגוריה זאת נוכל למצוא את דפוסי התכן הארכיטקטוניים הנפוצים הבאים [Phd]:

1. MVC – דפוס תכן ארכיטקטוני שנפוץ מאוד במערכות תוכנה שכוללות GUI. ארחיב מעט על דפוס תכן זה בהמשך, בתת-הפרק הבא "5.3 קשרים בין דפוסי התכן".

2. Layers – הפרדה של מערכות תוכנה ל-n שכבות (N-Tiers) כאשר הארכיטקטורה הנפוצה ביותר היא Three-Tiers, שכוללת את השכבות הבאות:

- שכבת המידע (information tier/bottom tier/data tier) – שכבה זאת מטפלת ישירות בנתונים. בדרך כלל היא מטפלת בנתונים שנמצאים בבסיס נתונים (בתוכנות שאני מפתח במקום העבודה שלי אני קורא לשכבה זאת בשם DbLayer).

- שכבת האמצע (middle tier/business tier) – שכבה זאת נקראת גם Business Tier כי היא נותנת שירות ללקוח ע"י שימוש בנתונים שהיא מקבלת משכבת המידע ותוך כדי התחשבות במדיניות העיסקית. למשל, עובד בחברה יוכל לצפות בנתונים בלבד ואילו מנהל יוכל גם לעדכן אותם.

- שכבת הלקוח (client tier/top tier) - שכבה זאת נותנת שירותים ישירות ללקוח, למשל ע"י הצגת מסך ה-GUI ללקוח. דוגמא נפוצה לתוכנה שמשמשת בשכבה זאת היא דפדפני האינטרנט.

כך, אנו יכולים לשנות כל שכבה (שהינה תת-מערכת תוכנה) ללא תלות בשכבות האחרות כל עוד אנו לא משנים את פרוטוקול ההתקשרות בין השכבות.

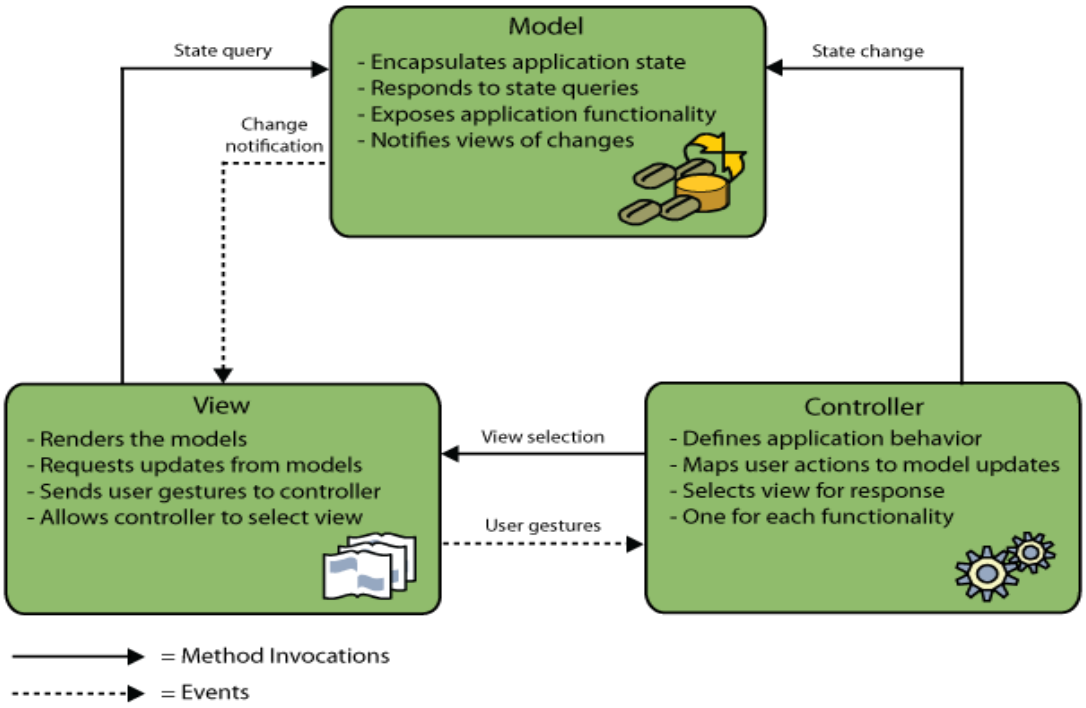
5.3 קשרים בין דפוסי התכן

יש לנתח מערכת תוכנה ממבט על ואז התעמקות בפרטים (top-down) ותוך כדי כך למצוא את דפוסי התכן המתאימים והקשרים ביניהם [Weis]. בספר [Gof] נכתב על הצורך להבין את הקשרים בין דפוסי התכן השונים ואף מובא שם תרשים של קשרים בין דפוסי תכן. אין זה המקום להביא את התרשים במלואו, אבל אציין כדוגמא, את הקשרים הבאים המוצגים בתרשים הזה:

- דפוס התכן Chain of Respobsibility יכול להגדיר את השרשור עבור דפוס התכן Composite.
- דפוס התכן Composite יכול להיעזר בדפוס התכן Iterator בכדי לסרוק את העצמים שמשתתפים במבנה העץ הרקורסיבי.
- דפוס התכן Facade משתמש במחלקה אחת שלרוב נרצה שיהיה לה מופע אחד בלבד ולכן נשתמש גם בדפוס התכן Singleton.

גם דפוסי תכן של ארכיטקטורת תוכנה יוצרים קשרים בין דפוסי תכן. עתה אציג בקצרה את דפוס התכן הארכיטקטוני MVC ואז נראה איך ניתן לשלב בתוכו דפוסי תכן מונחי-עצמים (בספר [Fbs] דפוס תכן ארכיטקטוני זה נקרא "Compound Pattern"). דפוס התכן הארכיטקטוני MVC נקרא כך על שם שלוש תת-המערכות שמרכיבים אותו: Model, View, Controller.

איור 1 להלן, מציג את התפקידים של כל תת-מערכת ב - MVC .



איור 1: דפוס התכן הארכיטקטוני MVC [bean]

דפוס התכן הארכיטקטוני MVC יכול לשלב בתוכו את דפוסי התכן מונחי העצמים Observer, Strategy ו-Composite באופן הבא [Fbs]:

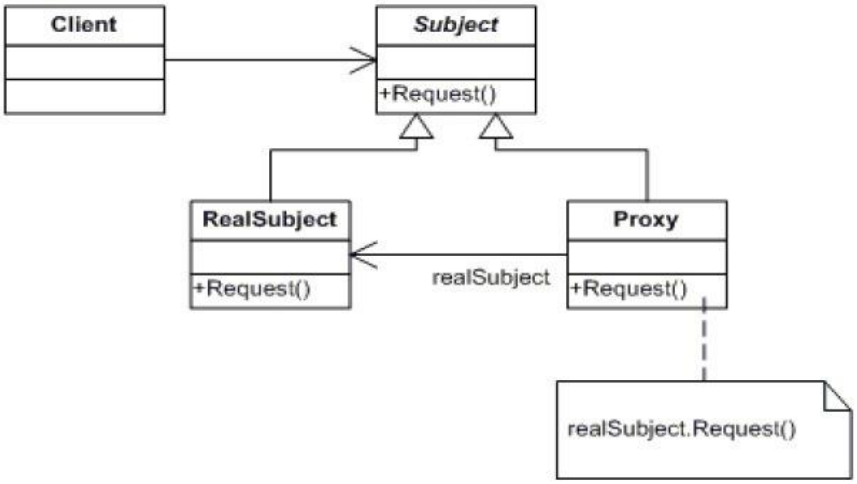
- ה-Model משתמש בדפוס התכן Observer בכדי לעדכן את ה-Views ועם זאת להיות בצימוד חלש איתם.
- ה-Controller משמש כ-Strategy עבור ה-View – ה-View יכול להשתמש במימושים שונים של ה-Controller.
- ה-View משתמש בדפוס התכן Composite בכדי לממש את המנשק הגראפי ללקוח (GUI). למשל, חלון שמכיל תיבת דיאלוג ובתוכה פקדים כמו כפתורים ותיבות טקסט.

כמו כן ניתן להשתמש בדפוס התכן Adapter לצורך התאמת Model חדש ל-View ו-Controller שכבר קיימים.

MVC עוזר גם לבניית שירותי אינטרנט כך שיהיו קלים יותר לבדיקה, לתחזוקה ולהרחבה. כאשר מפתחים שירותי אינטרנט, נוהגים להוסיף עוד רכיב תוכנה בין ה-Model לבין ה-View. רכיב זה נקרא Dispatcher והוא מפחית עוד יותר את הצימוד החלש שיוצר ה-Observer בין שכבות אלו [Sbhs].

לימוד הקשרים בין דפוסי התכן גם עוזר לזכור אותם ע״י כך שאנו נזכרים בדפוס תכן מסוים ואז אנו משחזרים דפוסי תכן אחרים שקשורים אליו.

בנוסף, כדאי ללמוד את ההבדלים בין דפוסי תכן דומים. הבדלים אלו יכולים לתרום להבנה טובה יותר של כל דפוס תכן ולהתאמה טובה יותר של כל דפוס תכן לבעית עיצוב מסוימת בה נתקלנו. למשל, דפוס התכן Adapter דומה בפעולתו לדפוס התכן המוכר Proxy אבל ההבדל ביניהם הוא ש-Adapter מספק ללקוח מנשק שונה לעצם שהוא מתאים את הלקוח לעבודה מולו ולעומת זאת Proxy מספק את אותו המנשק ללקוח כפי שמספק העצם שהלקוח רוצה לגשת אליו [Gof]. הזכרתי את דפוס התכן Proxy ולכן אתאר אותו כאן בקצרה. דפוס התכן Proxy מאפשר לשלוט בגישה של הלקוח לעצם מסוים, שנקרא לו RealSubject וזאת מבלי שהלקוח יידע זאת, כפי שמוצג באיור 2. למשל, העצם RealSubject יכול לפעול על מחשב מרוחק ואז הלקוח עובד מול העצם Proxy שמקשר את הלקוח ללא ידיעתו לעצם RealSubject.



איור 2: תרשים מחלקות של דפוס התכן Proxy [Huy]

5.4 דרגות קושי ומורכבות

מומלץ להתחיל ללמוד לפתח תוכנת מחשב ללא שימוש בדפוסי תכן ואז בהדרגה ללמוד לשפר את התוכנה בעזרת דפוסי תכן [Mgj].

כמו כן, מומלץ ללמוד בהתחלה דפוסי תכן פשוטים. למשל:

- Singleton
- Proxy
- Iterator

בהמשך אפשר ללמוד בהדרגה דפוסי תכן מורכבים יותר ויותר. למשל:

- Composite – לא כל כך מסובך אבל מבוסס על מבנה רקורסיבי שלעיתים קשה יותר להבנה.
- Factory Method – הוא נחשב לפשוט מאוד כי מבלבלים אותו עם Static Factory או עם SimpleFactory שאינם דפוסי תכן כלל. נראה זאת כשאציג בהמשך את דפוס התכן הזה.
- Abstract Factory – מורכב יותר מדפוס התכן Factory Method. משמש ליצירת משפחות של סוגי עצמים (נקראים Products). ניתן להשתמש ב-ConcreteFactory שלו ב-Factory Method.

אם אתה לא מעצב מנוסה של מערכות מונחות עצמים אז כדאי שתתחיל ללמוד את התבניות הפשוטות ביותר והנפוצות ביותר הבאות [Gof]:

- Adapter
- Composite
- Decorator

בין דפוסי התכן שכדאי לדחות את הלימוד שלהם יכול ליהיות Visitor. מנגנון ה-Double Dispatch בו משתמש דפוס תכן זה הוא די מורכב ללימוד ולזכירה. נתאר זאת במילה "נגישות" (Accessibility) [Acd]. כלומר, אם דפוס התכן הינו מורכב או אם נדגים דפוס תכן ע"י שימוש בדוגמא מורכבת מידי אז זה יהיה קשה ללמוד באופן יעיל את דפוס התכן.

בהמשך ניתן ללמוד פתרונות שמורכבים משילוב בין דפוסי תכן. למשל שימוש בדפוס התכן Composite בשילוב עם דפוס תכן Visitor בבואנו לבנות עץ של עצמים שנוצרים ממחלקות שונות ולעשות פעולות אחידות שונות על המידע הנשמר בצמתים שלו ללא צורך בהעמסת אותן פעולות כמתודות נוספות. למשל, בקורס עיצוב

תוכנה בנו הסטודנטים מנשק גראפי ללקוח. לצורך כך הם השתמשו בשילוב של דפוסי התכן הבאים שלמדו: Command, Proxy, Composite ו – Mediator [Abcm].

עוד שלב מתקדם יכול להיות לימוד של דפוסי תכן ארכיטקטוניים כמו MVC שהוצג בפרק "קשרים בין דפוסי התכן" בעמוד 25.

5.5 שימוש ברדוקציה

רדוקציה מאפשרת לפתור בעיה ע״י תרגום שלה לקבוצה של בעיות שקל לפתור אותן [Baat].

יש דפוסי תכן פשוטים שקל להבין אותם במהירות, למשל Factory או Singleton. אך קיימים גם הרבה דפוסי תכן מורכבים יותר שאותם ייקח יותר זמן ללמוד ויהיה קשה יותר לזכור אותם מאוחר יותר, למשל Visitor.

כאן יכולה לבוא לעזרתנו הרדוקציה שמעניקה לנו את היתרונות הבאים:

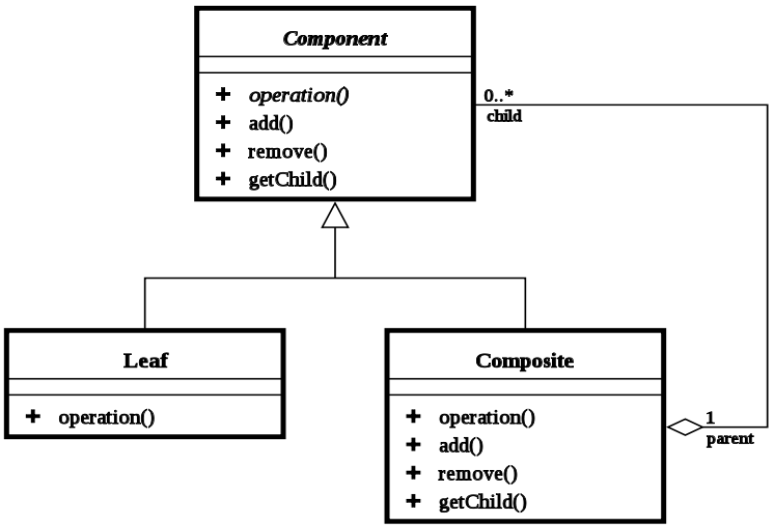
- הצגה הדרגתית ופשוטה יותר של החומר.
- עזרה לשחזר בזיכרוננו את דפוס התכן ע״י זה שנזכרים בשלבים הפשוטים ביותר.

דוגמא נוספת לשימוש ברדוקציה בנושא דפוסי תכן היא בעיה מורכבת שאנו נדרשים לפתור ואנו לא מוצאים דפוס תכן שמתאים לפיתרון הבעיה. אבל יכול להיות שנוכל לשלב כמה דפוסי תכן שנותנים כל אחד פיתרון פשוט יותר. ואולי נצטרך אפילו לשנות קלות חלק מדפוסי התכן בכדי לשלבם בצורה מיטבית לפתרון כולל לבעיה המוצגת.

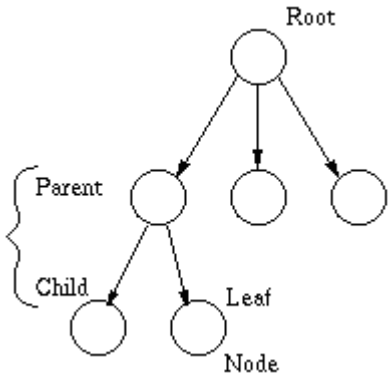
מחקרים מדווחים על קשיים של סטודנטים מתחילים וגם של סטודנטים מתקדמים להשתמש ברדוקציה ונמנים סיבות אפשריות לקשיים אלו [Baat], ביניהם קשיים פסיכולוגיים (למשל רצון לתפירת פיתרון מקורי שלא נסמך על פתרונות קיימים). עם זאת מדווח שסטודנטים מתקדמים נטו להיעזר יותר ברדוקציה – אבל עדיין גם הם נתקלו בקשיים בשימוש בכלי חשוב זה. עוד מסופר בעבודת הסמינר הזאת שהשימוש ברדוקציה, בבתי הספר התיכוניים ואפילו באוניברסיטה, נלמד באופן עקיף בלבד. נטען שם שזה מחדל שכדאי למצוא לו פיתרון.

דוגמא טובה היא לימוד של דפוס התכן Composite ע״י רדוקציה ללימוד של מבנה נתונים - עץ.
לאחר שאנו מבינים שעץ הינו מבנה נתונים רקורסיבי נוכל להבין באופן קל יותר את דפוס התכן Composite.
ניתן להסביר את דפוס התכן Composite ע״י מבנה הנתונים עץ [Gof].

להלן איור 3 מציג תרשים מחלקות של דפוס התכן Composite שעלול להראות בהתחלה קשה להבנה אבל ניתן להיעזר ברדוקציה ולגשת לאיור של עץ נתונים שהבאתי באיור 4 שלאחריו. ואז ניתן להסביר למשל איך כל עצם Composite מתפקד כצומת בעץ שיש לה בנים.



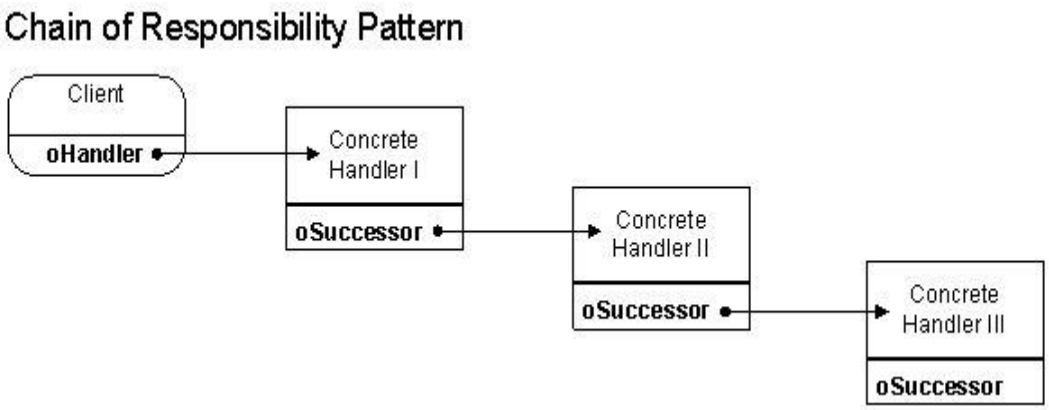
איור 3: תרשים מחלקות של הדפוס תכן Composite [Comm]



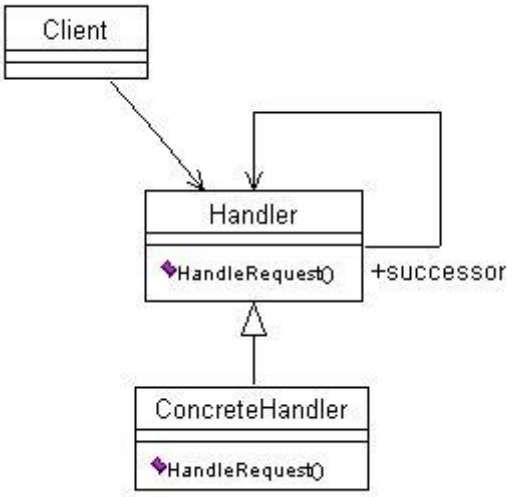
איור 4: איור של מבנה הנתונים הרקורסיבי עץ [Comm]

אעיר, שלצורך הבנה של דפוס התכן Composite ע"י רדוקציה ניתן גם ללמוד קודם את דפוס התכן Chain of Respobsibility ואז קל יותר להבין בצורה הדרגתית את דפוס התכן Composite. למשל, ניתן להשתמש בשני האיורים הבאים שממחישים את דפוס התכן Chain of Respobsibility.

האיורים הבאים, איור 5 ואיור 6 מציגים את דפוס התכן Chain of Respobsibility בצורה גרפית.



איור 5: מבנה עצמים טיפוסי של דפוס התכן [Comm] Chain Of Responsibility



איור 6: תרשים מחלקות של דפוס התכן [Comm] Chain Of Responsibility

5.6 גישות לתחילת עבודה עם דפוסי תכן

5.6.1 הקושי לעצב תוכנה

עיצוב טוב של תוכנה מונחית עצמים שניתנת לשימוש חוזר היא משימה קשה. זה כמעט בלתי אפשרי ליצור עיצוב טוב כזה בפעם הראשונה [Gof]. מפתחים מנוסים של תוכנה מונחית עצמים נעזרים בניסיונם תוך שימוש חוזר שוב ושוב בתבניות שהוכחו כנותנות פתרון טוב בעבר. זאת לעומת מפתחי תוכנה פחות מנוסים שנוטים לפתור כל בעיה ע״י פיתרון חדש שלהם. המפתחים המנוסים משתמשים בתבניות כמו עיטור (decorate) של עצמים בכדי להוסיף או למחוק בקלות יכולות לתוכנה או ייצוג של מצבים בתוכנה (states) ע״י עצמים. כלל שיכול לעזור לעיצוב מוצלח הוא שדפוסי התכן שכדאי לבחור הם דפוסי תכן ששולבו בעבר במערכות תוכנה בהצלחה.

5.6.2 גישות להפנמת השימוש בדפוסי תכן

במאמר על הוראת מדעי המחשב שמשתמשת בלימוד שבא בעקבות עשיית טעויות [Gina], נכתבה האימרה הידועה – ״אתה לומד מהטעויות שאתה עושה״ וגם האמרה החזקה יותר ״אתה לומד טוב יותר מהטעויות שאתה עושה״. בהרבה תחומים אנו נעזרים באינטואיציה שלנו והרבה פעמים היא אכן עוזרת לנו לפחות בצורה חלקית. אבל כפי שנכתב במאמר, אנחנו משתמשים הרבה פעמים רק באינטואיציה שלנו לצורך הגשת פיתרון לבעיות שמטבען דורשות פיתרון מדויק יותר. והרי פיתרון מדויק יותר דורש שימוש בכלים מדויקים יותר, כמו למשל הצורך להוכיח שפיתרון תקף במקרה הכללי (למשל ע״י שימוש באינדוקציה מתמטית). מדווח במאמר, שסטודנטים טוענים שהפתרון שלהם הוא פתרון טוב לבעיה שנדרשו אליה בגלל שהם בדקו אותו על מספר דוגמאות או שפשוט ״ברור״ להם שהפיתרון הוא פיתרון טוב.
סטודנטים נוטים למדוד נכונות של תוכנה ע״י כך שהם מעריכים את ההתנהגות שלה ולא שמים לב להיבטים של עיצוב כמו מתן אפשרות להרחבה של בסיס הקוד ושל של מספר המשתמשים במקביל (Scalability) , עמידות לשינוי וגמישות להרחבה [Mgj].

אז כיצד ניתן ללמוד להתגבר על מגבלות האינטואיציה?
ניתן לנסות פתרונות עיצוב תוכנה מוטעים כאשר דפוס התכן לבסוף מחליף אותם כפיתרון הנכון . בנוסף אפשר לנסות שימוש מוטעה בדפוסי תכן קיימים. כלומר לקחת דפוס תכן ״נכון״ ואז לנסות לפתור איתו בעיה שדפוס התכן לא מתאים כפיתרון נכון עבורה. כלומר כאן אנחנו מתרגלים באופן מעשי את נושא ה״הקשר״ (context) שמופיע בהגדרת דפוס תכן [Mha].

במאמר [Weis] נכתב איך הועבר שיעור פשוט שבו הכותב נתן אפשרות לסטודנטים לטעות ואף עודד אותם לטעות ע״י הצגת בעיה פשוטה בהתחלה ולאחר מכן הוסיף הרחבות לבעיה והראה להם כיצד יהיה קשה לתקן את הפתרון שהציגו כך שיתאימו לפיתרון הבעיה המורחבת. במאמר הוא נותן דוגמא (מערכת לחישוב שכר) וסוקר רעיונות לפיתרונות מוטעים ואז מסביר את הבעייתיות בכל פיתרון. לבסוף הוא מציג כרעיון חדש את

הפיתרון הנכון אותו רצה ללמד מלכתחילה ומסביר למה פיתרון זה אכן פותר בצורה טובה את הבעיה שהוצגה. בדרך זאת, הוצגו דפוסי תכן כמו Singleton ו-Factory Method.

הסיבוך בעולם התוכנה הולך וגודל ומפתחי תוכנה נוטים לעבוד קשה יותר עם השקעה של פחות מחשבה קודמת. זה נותן להם יתרונות נתפסים רבים כמו נראות טובה יותר של עבודתם והערכה רבה יותר ע"י המנהלים שלהם. לכן הרבה פעמים נזנח שימוש מיטבי בדפוסי תכן [Ster].

מחקרים הראו שאין טעם ללמד דפוסי תכן כקורס למתחילים [Trt], [Czbd]. יש צורך שהסטודנטים ילמדו עיצוב מונחה עצמים בצורה טובה וחשוב מאוד שהם גם ירכשו ניסיון בתחום זה. אמנם דפוסי תכן עוזרים לשפר את העיצוב של מערכת תוכנה אבל יש צורך בהכרה טובה של עיצוב מונחה עצמים בכדי לדעת להשתמש היטב בדפוסי תכן. גם GoF כותבים, בהקדמה של ספרם, שהם מניחים ידע וניסיון בעיצוב מונחה עצמים [Gof].

לימוד דפוסי תכן הוא נושא קשה אפילו למפתחים מנוסים שהתנסו במבנים של תוכנה שחוזרים על עצמם [Weis]. אמנם יש הטוענים שכדאי ואפילו עדיף לשלב לימוד דפוסי תכן יחד עם לימוד של פיתוח מונחה עצמים [Ppp]. כאשר נטען שסטודנטים שלמדו דפוסי תכן בשלב מאוחר יותר התקשו יותר בשינוי התפיסות שהתגבשו אצלם במשך הזמן. ואמנם מדובר על קורסים שהועברו לסטודנטים ועל השלבים להעברה של החומר בצורה קלה יותר, אבל לעומת זאת מאמרחוקרים אחרים מסתמכים גסהם על קורסים אמיתיים וניתן להתרשם מהם [Trt] על מידת הבשלות הבעייתית של סטודנטים מתחילים ללמוד את הנושא המתקדם של דפוסי תכן כאשר הסטודנטים עדיין לא צברו מספיק ידע וניסיון בחומר הרקע, כמו יסודות עיצוב מונחה עצמים. הרעיון הוא לא לשנן שמות של דפוסי תכן ולדעת להגדיר אותם, אלא ללמוד כיצד לבנות עיצוב נכון – לצורך כך דרושה בשלות של אנשי הפיתוח [Mha].

לדעתי עלול להתקיים כאן מצב של "תפסת מרובה לא תפסת". גישת פשרה שאוכל להציע היא לשלב לימוד של דפוסי תכן בצורה מועטה ביותר בתוך החומר הבסיסי כך שזה יעניק מעין הכנה קצרה לסטודנט ללמוד נושא מתקדם זה כאשר הוא יהיה בשל יותר לכך.

6 שפת UML - כלי להצגת דפוסי תכן מונחי עצמים

6.1 מבוא

פרק זה מציג בקצרה את עיקרי UML (לצורך רענון) ואינו בא ללמד את הנושא. אני מניח שמי שמתעניין בנושא של דפוסי תכן כבר למד עיצוב מונחה עצמים ו-UML.

בכדי לתעד מבנה והתנהגות של עיצוב מונחה עצמים בצורה קלה סטנדרטית ומבנית משתמשים בשפת UML (Unified Modeling Language).
UML נוצרה ע״י שלושה חוקרים שיצרו בנפרד מתודולוגיות לניתוח של עיצוב מונחה עצמים ובהמשך איחדו את עבודתם [Jose].

שפת UML היא שפה לתיאור מודלים (כפי ששמה מעיד עליה).
נגדיר עתה מודל. מודל הינו אמצעי לתפיסת רעיונות, יחסים, החלטות ודרישות בעזרת סימונים מוגדרים היטב שיכול ליהיות מיושם להרבה תחומים שונים [Pilo]. מודלים משמשים לתיאור מופשט (abstract) ומפושט (simplified) של ישויות מורכבות [Tome]

- מודל מתמקד באלמנטים העקרוניים ללא ירידה לפרטים – זאת פשרה לעומת תיעוד מלא של כל פרט.
- מודל דורש ״תרגום״ לישות האמיתית
- במודל יש דרגות חופש לפרשנויות שונות

מודל יכול להיות גראפי, טקסטואלי או משולב.
השפה הגראפית בה משתמש המודל מוכתבת ע״י הכללים הבאים [Tome]

1. אוסף של סמלים חוקיים (אלף-בית) - אלו הן הצורות הגראפיות והטקסט
2. צירופים חוקיים (תחביר) - צירופים חוקיים של אוסף הסמלים
3. משמעויות (סמנטיקה) - המשמעות של כל צירוף חוקי
4. כושר ביטוי (expressiveness) - מה ניתן לבטא במודל

המודלים נחלקים לשתי קטגוריות עיקריות.

1. מודל סטטי / מבני - מודל המתאר ישויות וקשרים ביניהם.

2. מודל דינמי / התנהגותי / תפקודי - מודל המתאר זרימה של תהליך או שינויי מצב.

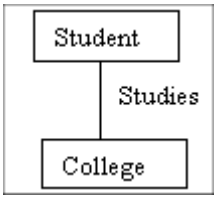
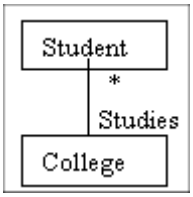
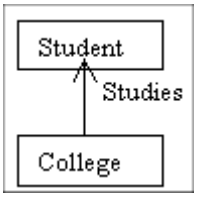
שתי קטגוריות אלו נותנות זוויות ראייה שונות שנועדו להבנה מעמיקה יותר של העיצוב וכן הן באות לתת מענה לבעלי עניין שונים. כל תרשים, שמציג המודל, מציג בצורה גראפית זווית ראיה שונה של העיצוב – למשל תרשים סטטי שמתאר את המבנה של הקוד ע"י הצגת המחלקות בתרשים מחלקות לעומת תרשים דינאמי שמראה את המסרים הנשלחים בין עצמים של אותן מחלקות בתרשים רצף.

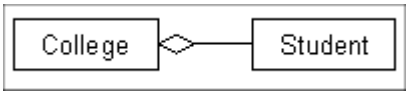
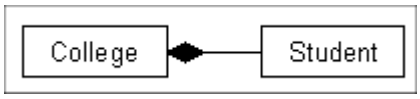
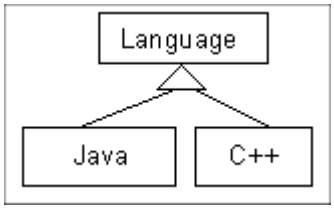
6.2 סקירת תרשימים סטטיים נבחרים

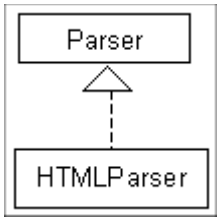
תרשים מחלקות (Class Diagram)

תאור המבנה של הקוד ע״י הצגת המחלקות והמנשקים (למשל שם המחלקה, השיטות/פונקציות הפרטיות והציבוריות, המאפיינים) והיחסים ביניהן.

הצגת היחסים בין המחלקות מפורטים בטבלה 1 הבאה.

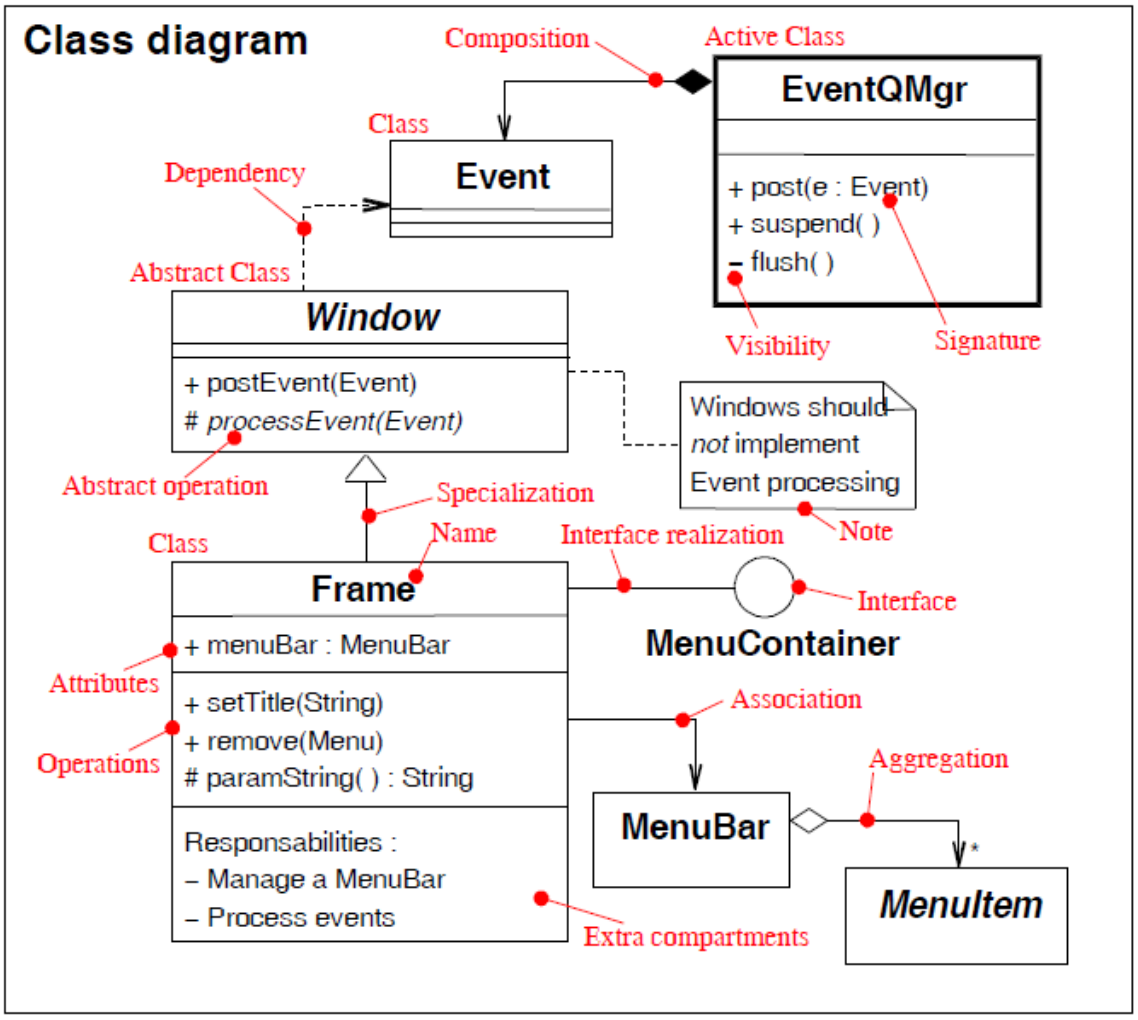
היחס בין המחלקות	סימון	תאור
אסוציאציה = התאחדות (Association)		כאשר שתי מחלקות קשורות אחת לשנייה בכל דרך, נוצר יחס של אסוציאציה בינהן.
ריבוי (Multiplicity)		יחס אסוציאציה של הרבה לאחד או הרבה להרבה. נסמן ״הרבה״ ע״י כוכבית.
אסוציאציה מכוונת (Directed Association)		יחס אסוציאציה בין מחלקות הוא דו כיווני כברירת מחדל. ע״י הוספת ראש חץ נקבע את כיוון יחס האסוציאציה. כלומר, החץ יוצא מהמחלקה המכילה

		לכיוון המחלקה המוכלת.
אסוציאציה משקפת (Reflexive Association)	אותו סימון של חץ כמו באסוציאציה מכוונת, אלא שהחץ יוצא מהמחלקה אל עצמה.	דוגמא ליחס אסוציאציה כזה היא כאשר למחלקה יש מגוון של סוגי אחריות. למשל, עובד באוניברסיטה יכול ליהיות פרופסור או דוקטור או אב הבית.
אגרגאציה = התקבצות (Aggregation/Has A)		כאשר מחלקה נוצרת כאוסף של מחלקות אחרות, אז נקרא ליחס אסוציאציה זה בשם יחס אגרגאציה. סימון המעויין יוצמד למחלקה המכילה.
קומפוזיציה = הרכב (Composition)		קומפוזיציה היא וריאציה על יחס האגריגאציה. יחס של קומפוזיציה מרמז שמחזור חיים חזק מקשר בין המחלקות. למשל למכללה אין זכות קיום ללא סטודנטים. סימון המעויין המלא יוצמד למחלקה המכילה.
הורשה/הכללה (Inheritance/Generalization/ Is A)		המחלקה היורשת היא מאותו סוג כמו מחלקת האב. יחס זה מאפשר ליצור אלמנטים לשימוש חוזר.

		המחלקה היורשת בעצם יורשת את התפקודיות המשותפת שהוגדרה במחלקת האב.
מימוש (Realization)		בדרך כלל, ביחס של מימוש, יישות אחת (שהיא לרוב מנשק אך יכולה ליהיות למשל גם מחלקה מופשטת כלומר מחלקה ללא מימוש) מגדירה קבוצה של תפקודיות בצורה של חוזה. ואז יישויות אחרות (לרוב אלו הן מחלקות) "מממשות" את החוזה.

טבלה 1: היחסים בין המחלקות [Pete] (עריכה ותרגום — א.א.)

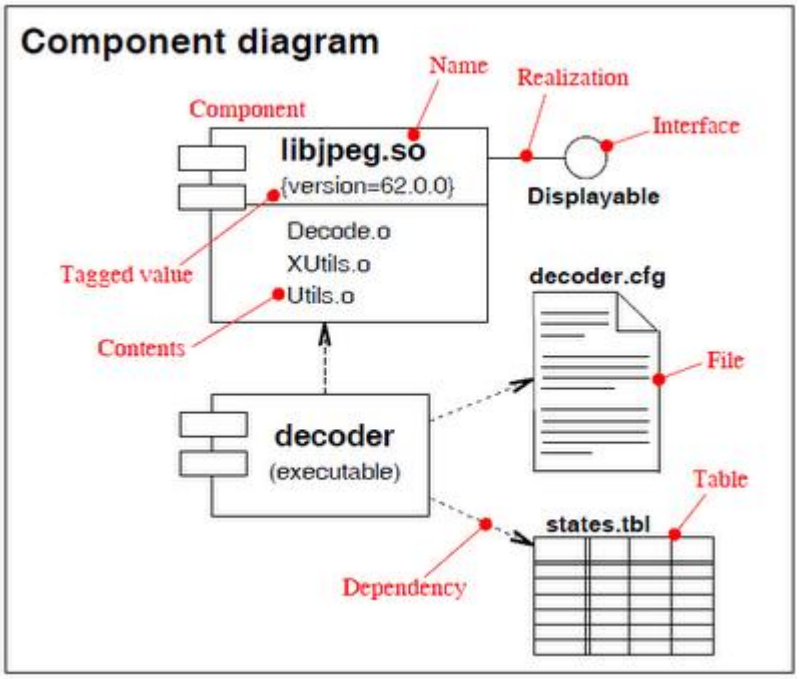
איור 7 הבא מציג דוגמא של תרשים מחלקות ומסביר על היחסים ביניהן.



איור 7: דוגמא של תרשים מחלקות [Sagg]

תרשים רכיבים (Component Diagram)

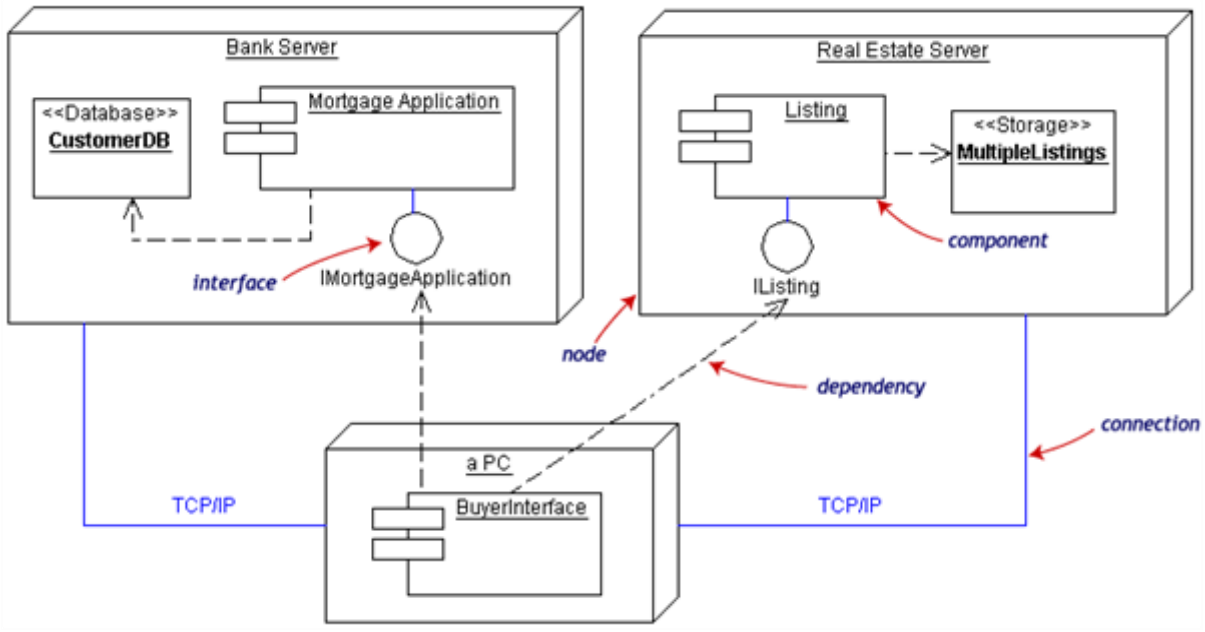
איור 8 הבא מציג תאור רכיבי תוכנה והמנשקים ביניהם.



איור 8: דוגמא של תרשים רכיבים [Sagg]

תרשים פריסה (Deployment Diagram)

תאור של מארזי החומרה והמנשקים ביניהם ושל רכיבי התוכנה המותקנים עליהם.
בעצם משלבים את התרשים של הרכיבים (Component Diagram) שהוצג לעיל בתוך תרשים זה.

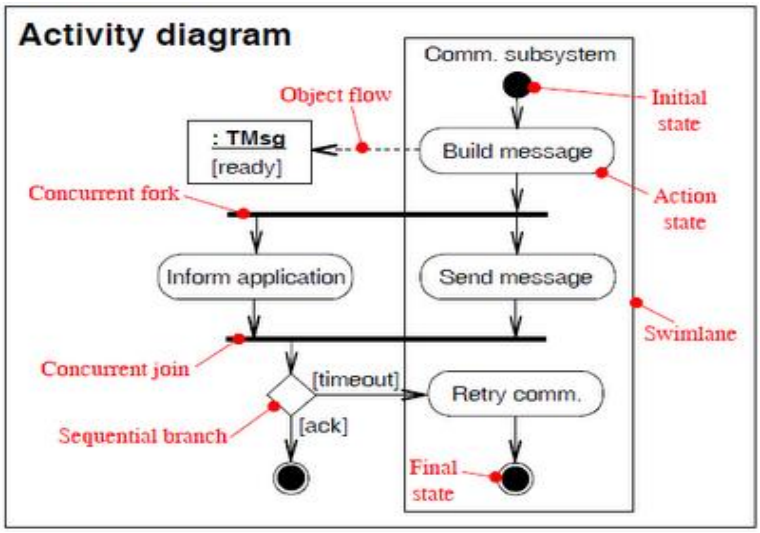


איור 9: דוגמא של תרשים פריסה [Mill]

6.3 סקירת תרשימים דינאמיים נבחרים

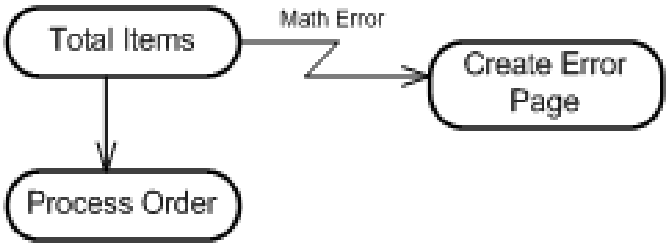
תרשים פעילות (Activity Diagram)

זהו תיאור של זרימת תהליכים. לתרשים זה יש כושר ביטוי (expressiveness) של תרשים הזרימה (flowchart) שלומדים בקורסי תכנות למתחילים [Jose], אך הוא מרחיב את כושר הביטוי ע"י תוספת של swimlanes (חלוקת התרשים לפי נושאים לוגיים), Guard conditions (כל מעבר בתהליך מקבל תנאי לביצועו), A fork and join (טיפול בתהליכים שקורים בו זמנית), טיפול במקרים חריגים (exceptions), סיגנלים.
ניתן להשתמש בתרשים זה בכדי לפרט ביצוע של פעילות בתרשים המצבים (יתואר בהמשך).



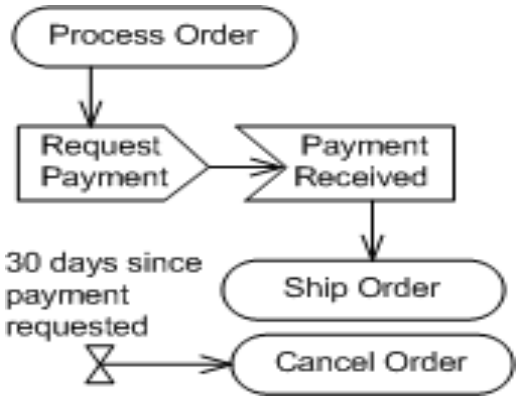
איור 10: דוגמא של תרשים פעילות [Sagg]

איור 11 מציג טיפול במצב שגיאה.



איור 11: הצגת טיפול במצב שגיאה בתרשים פעילות [Alle].

איור 12 מציג טיפול בסיגנלים: שליחת סיגנל, קבלת סיגנל, סיגנל שתלוי בזמן.



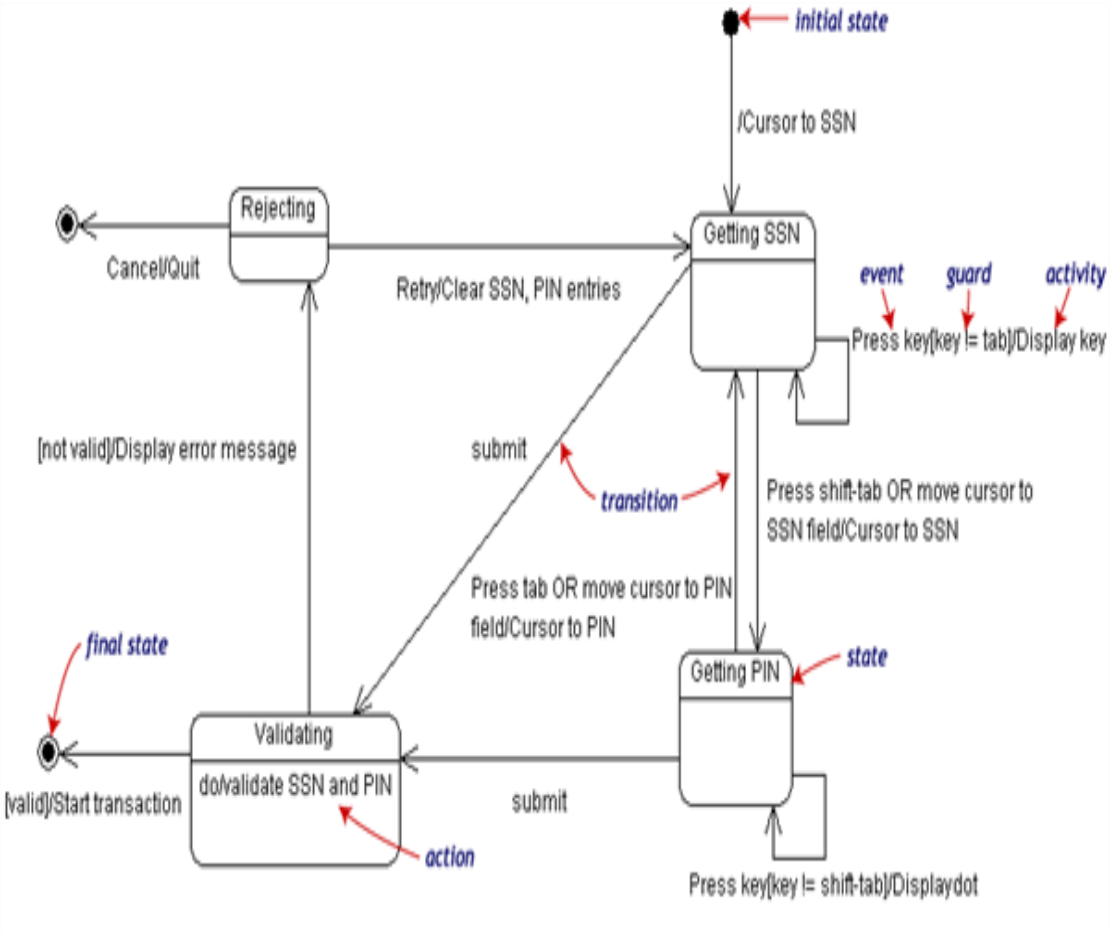
איור 12: הצגת טיפול בסיגנלים בתרשים פעילות [Alle]

תרשים מצבים (Statechart Diagram)

המעברים בין מצבים במערכת (לרוב מוצגים מצבים פנימיים של עצמים) כתגובה לאירועים.

על כל חץ שמציג את המעבר בין מצבים (המצבים מיוצגים ע"י מרובעים) נוכל לכתוב את האירוע שקרה (event), את התנאי (guard) לביצוע המעבר ואת הפעולה (activity) שמתבצעת בזמן המעבר (הערה: במערכות זמן אמת מודגש שזמן המעבר הינו אפסי). נכתוב זאת באופן הבא: event [guard] / activity .

בתוך כל מצב נוכל לכתוב באופן דומה את הפעולות שמתבצעות בו (למשל אירוע של כניסה או יציאה מהמצב).

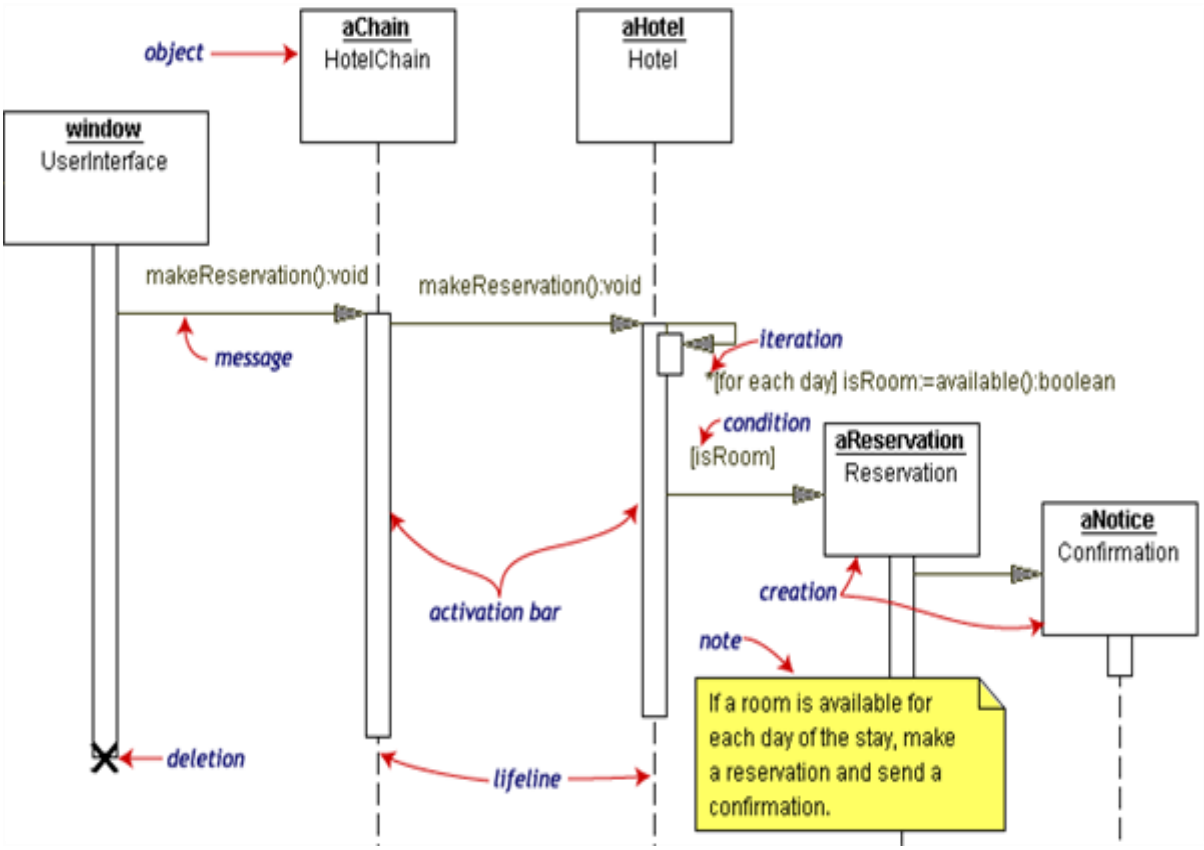


איור 13: דוגמא של תרשים מצבים [Mill]

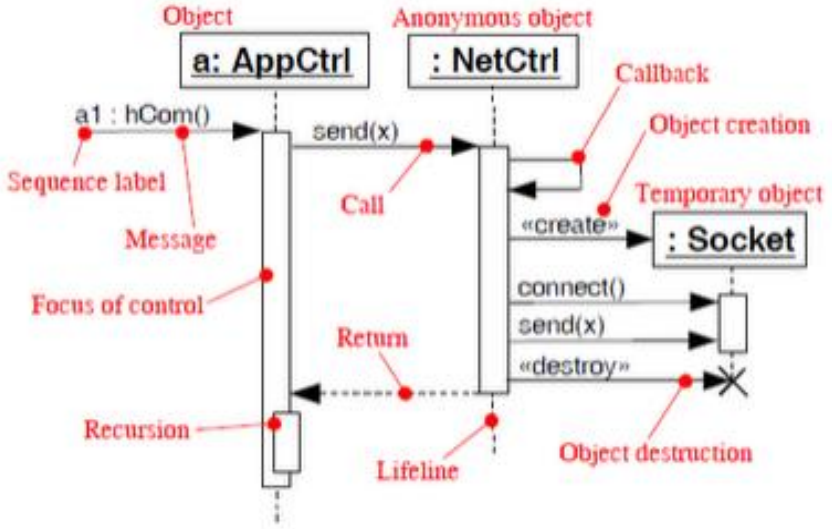
תרשים רצף (Sequence Diagram)

תאור המסרים הנשלחים בין העצמים וכן יצירה ומחיקה של עצמים (ניתן להתייחס לזה כסוג של מסר).
זמן הפעילות בין העצמים מתקדם משמאל לימין ואילו זמן חייו ופעילותו של עצם יחיד מתקדם מלמעלה למטה. כל עצם מיוצג ע"י מרובע שמכיל את שמו וכן קו מקווקו שיורד ממנו (קו זה נקרא lifeline).

באופן רשמי נציג את המסרים כקריאה למתודות של כל עצם ואף נציין את סוג התשובה המוחזרת (ראה איור) אבל נהוג בתעשייה לציין רק שם שיסביר את מהות המסר (לצורך הפרדה בין עיצוב לבין מימוש שמשתנה כל הזמן) .



איור 14: דוגמא של תרשים רצף [Mill]

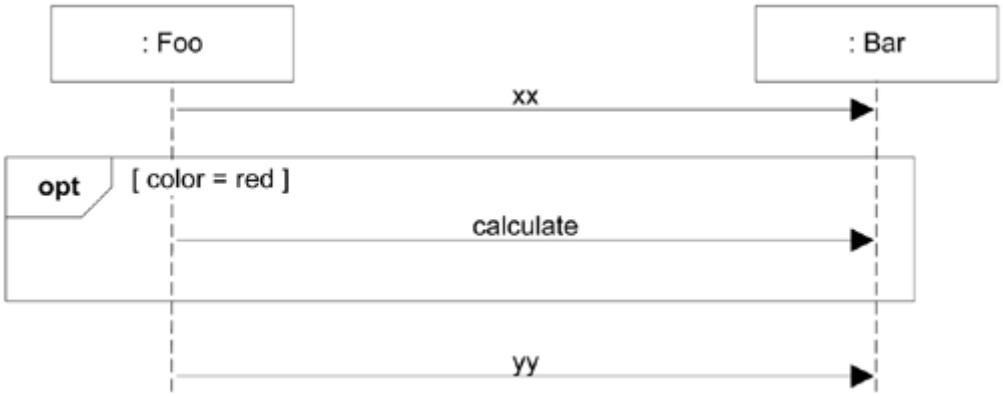


איור 15: דוגמא של תרשים רצף [Mill]

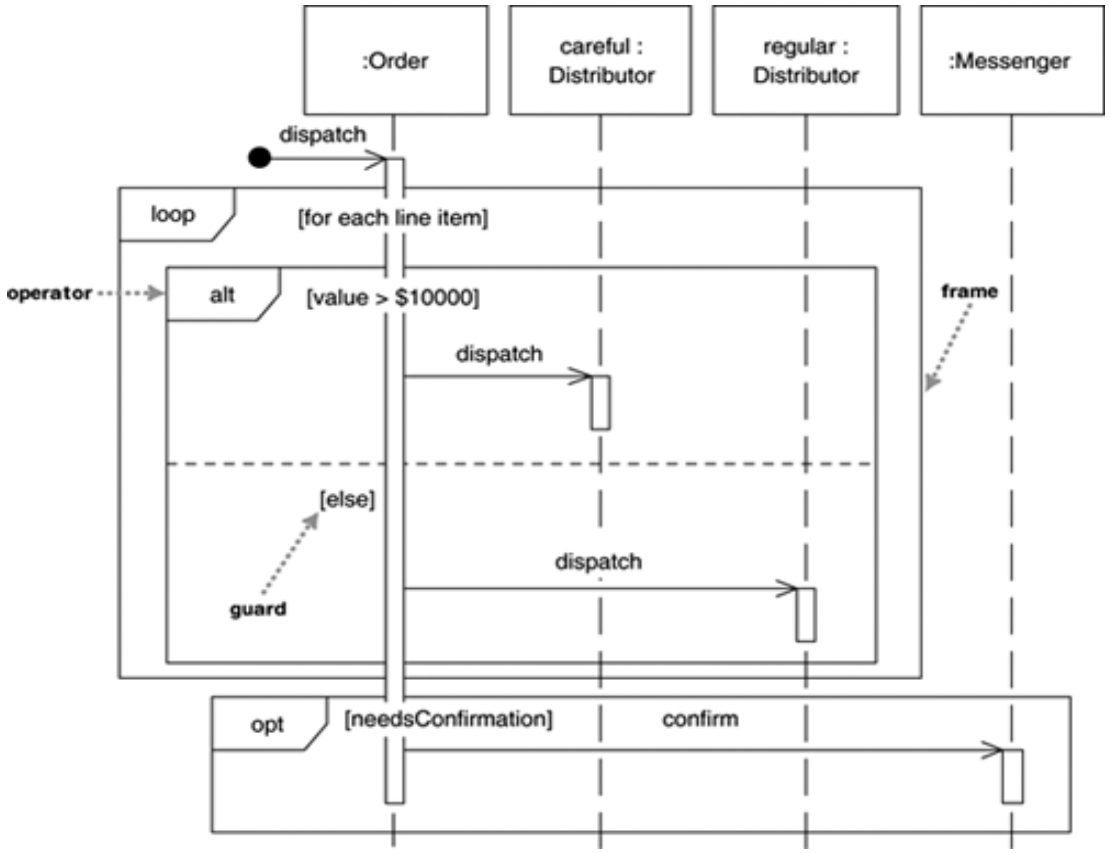
ב - UML-2 נוספה האפשרות להוסיף לולאות, הסתעפויות ועוד. את זאת נעשה בעזרת מסגרות (Interaction Frames). להלן מסגרות שניתן להוסיף לתרשים רצף [Info]. לאחר הרשימה הבאה אציג שני תרשימים לדוגמא.

- Alt - כמה חלקים אלטרנטיביים. רק החלק שהתנאי שלו מתקיים יבוצע.
- Opt – אופציה. דומה ל-Alt אלא שכאן מדובר בחלק אחד בלבד שיבוצע רק אם התנאי שלו מתקיים.
- Par – מקבילי. כל חלק מבוצע במקביל.
- Loop – לולאה. החלק חוזר על עצמו וסימון ה-Guard מגדיר את בסיס האיטראציה.
- Region – אזור קריטי. רק Thread אחד יכול להתבצע בחלק זה בו זמנית.
- Neg – שלילי. מציג פעילות לא אפשרית.
- Ref – התייחסות. סימון שמפנה את הקורא לתרשים אחר.
- Sd – תרשים רצף. אם רוצים אפשר פשוט למסגר את כל תרשים הרצף בסימון זה.

עתה אציג שני תרשימי רצף להדגמת השימוש במסגרות:



איור 16: דוגמא לשילוב המסגרת Opt בתרשים רצף [Info]



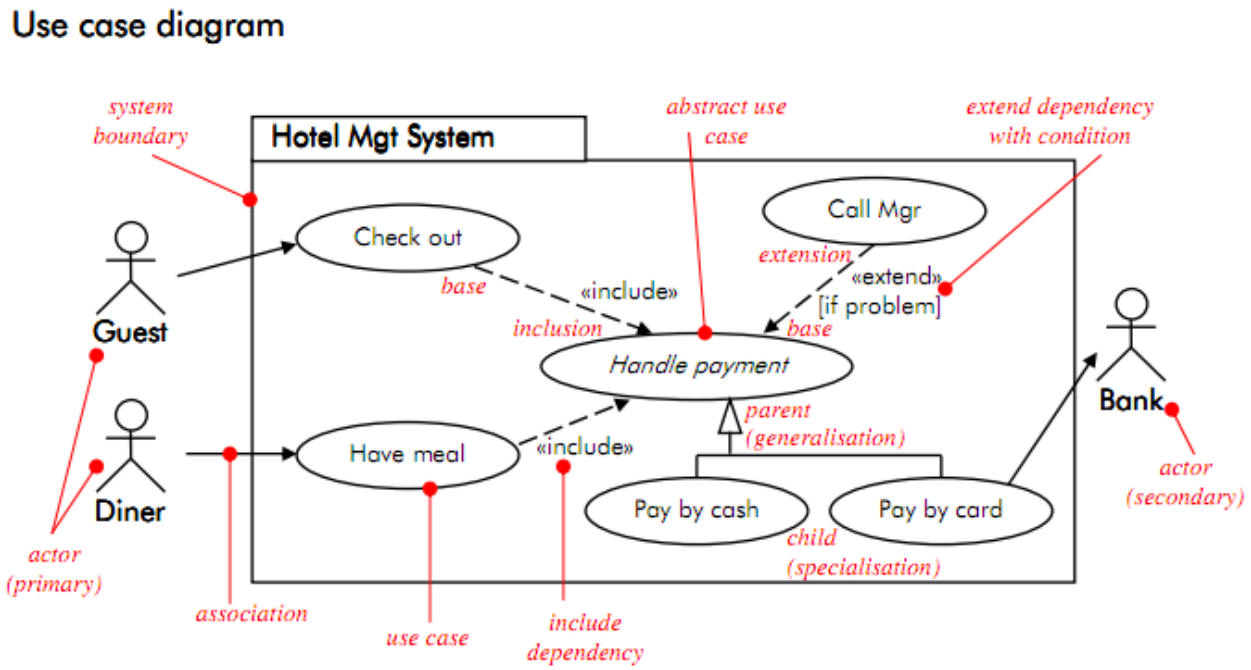
איור 17: דוגמא לשילוב מסגרות בתרשים רצף [Info]

תרשים מקרי שימוש (Use Case Diagram)

האינטראקציה של השחקנים החיצוניים (למשל רופא, חולה, פקיד קבלה) עם המערכת מוצגת מנקודת המבט שלהם. ניתן דגש על מה המערכת עושה ופחות על איך. כל אליפסה בתרשים מייצגת מקרה שימוש (Use Case) שהינו משימה שהמערכת מבצעת. במסמך עיצוב המערכת ייכתב פירוט של כל מקרה שימוש שזה כולל: תנאים מקדימים (preconditions), פירוט הפעולות, תנאים שמתקיימים בסיום (post conditions), תנאים שמתקיימים בכישלון והתמודדות המערכת איתם (exceptions), נושאים שלא ניתן להם עדיין פיתרון.

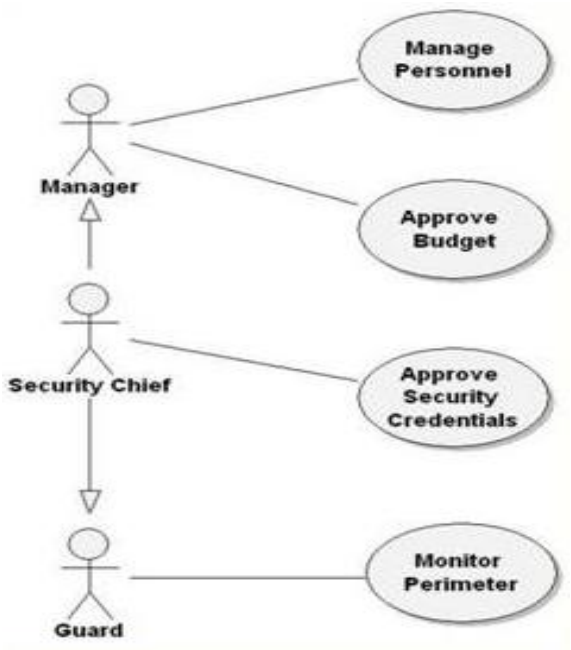
לפעמים נרצה להציג פעולה של המערכת שמתרחשת ללא קשר לשחקן חיצוני כלשהו ואז נציג שחקן ששמו Timer ותפקידו לתזמן מקרי שימוש שמתרחשים ברקע.

איור 18 הבא מציג שחקנים שמתקשרים עם ה-Use Cases, Use Uase שכוללים Use Cases אחרים או שמרחיבים Use Cases אחרים וכן מסגרת שתוחמת את המערכת ונותנת לה שם.



איור 18: דוגמא של תרשים מקרי שימוש [Puff]

קיימות הרחבות לתרשים כמו למשל Actor שיורש את תפקידיו של Actor אחר (generalization) בהקשר של המערכת, כפי שניתן לראות באיור 19 הבא.



איור 19: דוגמא של תרשים מקרי שימוש [Devx1]

7 סקירת דפוסי תכן נבחרים

בפרק זה אסקור דפוסי תכן נבחרים לפי קטגוריות. בחרתי להציג מדגם של דפוסי תכן נפוצים בתעשייה מתוך מדגם של קטגוריות.
כל דפוס תכן נוצר, והתפתח בהמשך, לאחר מחשבה רבה וכל פרט בעיצובו עבר שיקולי עיצוב רבים.
אציג חלק מהשיקולים לעיצוב דפוסי התכן שאציג.
אתחיל בתאור דפוס התכן Observer החביב עליי.

7.1 דפוסי תכן בקטגוריה Behavioral

הקטגוריה Behavioral מכילה דפוסי תכן שעוסקים בהתנהגות של עצמים או מחלקות והתקשורת ביניהם.
דפוסי התכן בקטגוריה זאת משתמשים ביחס של הורשה בין מחלקות או ביחס של הרכבה בין עצמים לצורך החבאה של אופן ההתקשרות בין המחלקות או העצמים [GoF].

7.1.1 דפוס התכן Observer

1) שמות שניתנו לדפוס תכן זה

Observer - השם Observer מרמז על כך שהעצמים "צופים" בשינויי המידע שנעשים בעצם "מקור המידע" [GoF]. להלן שמות אחרים שניתנו לדפוס תכן זה.

Dependents – שם שמרמז על התלות של העצמים בעצם "מקור המידע" [GoF].

Publish-Subscribe- זהו שם שמתאר את הפעילות בין העצמים בדפוס תכן זה [GoF] .
עצמים נרשמים לקבלת עדכונים על שינויי מידע ואז המידע מתפרסם אליהם.

Model-View – Model הינו שם כללי למקור-מידע ואילו View הינו שם כללי לאופן ייצוג אותו מידע. מצב ה-View תלוי במצב המידע שמוחזק ב-Model [GoF] .

2) הצגת ההקשר והבעיה

אציג את הבעיה ע״י כך שאתן דוגמא פשוטה מחיי היומיום. באיור הבא מוצגת מערכת רדיו ישנה עם רמקול מובנה וללא שקעים ליציאת שמע. קנינו מערכת זאת והיינו מרוצים במשך תקופה קצרה – אז מה הבעיה?

איור 20: מערכת שמע לא מודולארית [dev]

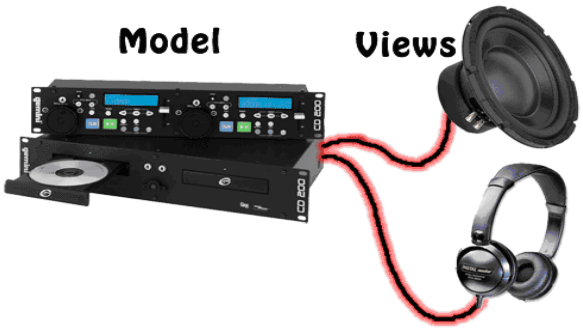
בעיות שזיהינו:

- לפעמים הקול שבוקע מהרדיו נעים לנו אבל לא לאנשים מסביבנו.
- מצאנו בחנות רמקול באיכות טובה יותר וקשה להחליף את הרמקול.

3) הפיתרון

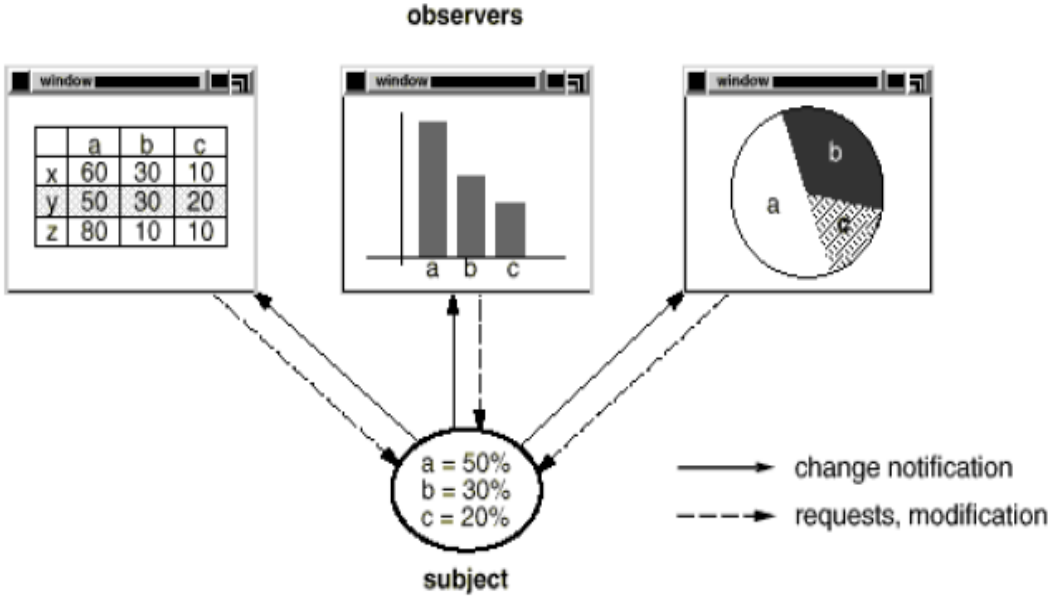
אז החלטנו לקנות מערכת מודולארית חדישה במקום הרדיו הישן שלנו. באיור 23 להלן, מוצגת מערכת שמע מודולארית כזו שאפשר לחבר אליה רמקול ואוזניות (או כל אמצעי שמע אחר). המערכת מייצרת רק אותות חשמליים ואת השמע מפיקים רק הרמקול או האוזניות.

אז קודם כל יש לחבר את אמצעי השמע למערכת. דפוס התכן יקרא לחיבור זה בשם ״הרשמה״ (Registration). ואז באופן אוטומטי נשמע מהם את כל מה שמפיקה המערכת. המערכת שמפיקה את אותות השמע, שדפוס התכן יקרא לה מקור הנתונים (Model או Subject), לא מודעת לקיומם של אמצעי שמע אלו שדפוס התכן יקרא להם אמצעי התצוגה (Views או Observers) ולכן ניתן להחליפם בקלות לכל אמצעי אחר, למשל למערכת הגברה חדשה שנקנה. המערכת שמפיקה את אותות השמע יודעת רק ״לדבר״ עם עצמים בעלי מנשק שמתאים למנשק שלה.



איור 21: מערכת שמע מודולארית [dev] (עריכה — א.א.)

דוגמא נוספת קרובה יותר לעולם התוכנה מובאת באיור 22 הבא. בכל חלון גראפי (שנקרא באיור Observer), שרשמנו אותו לקבלת המידע (registration) ממקור הנתונים (שנקרא באיור Subject), ישנה הצגה שונה של הנתונים.



איור 22: דוגמא לשימוש בדפוס התכן Observer לצורך מוטיבציה [Gof]

שאלה: מה משותף לשתי הדוגמאות לעיל?

תשובה: מנענו צימוד חזק בין מקור הנתונים (שנקרא Model או Subject) לבין האמצעים להצגת הנתונים (שנקראים Views או Observers).

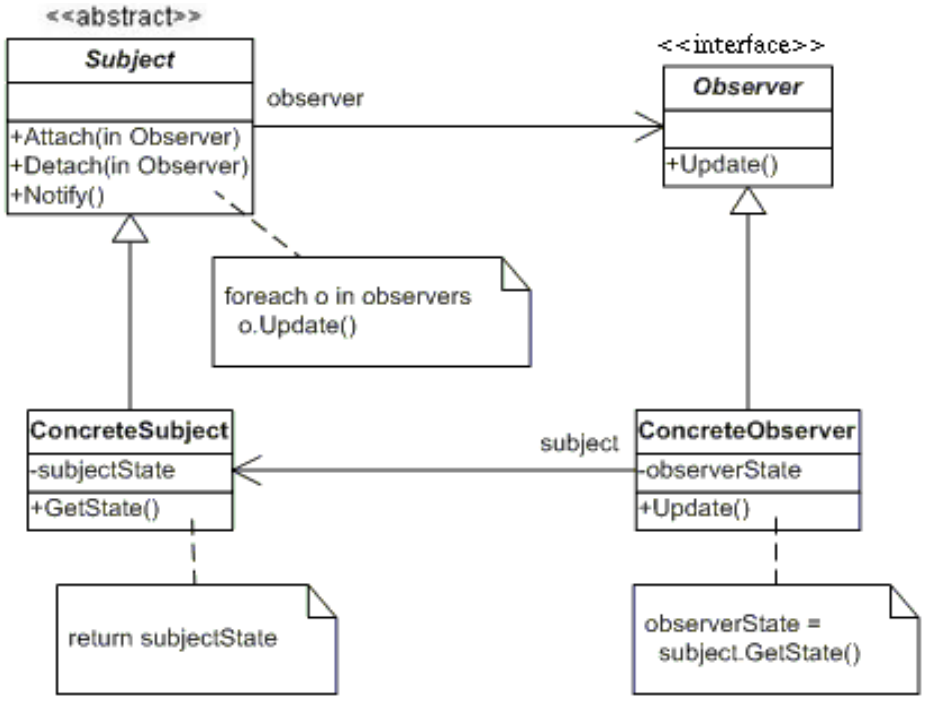
חשובה האפשרות הדינאמית לחבר/לנתק את ה-Observers וגם להחליפם מבלי להחליף או לשנות את ה-Subject ואף אפילו מבלי לכבות את המערכת. באופן דומה נוכל להחליף גם את ה-Subject אם נרצה. מה שלא משתנה זה רק המנשק שהוגדר בין מקור הנתונים לאמצעי התצוגה.

כפי שראינו, נרצה להשתמש בדפוס תכן זאת כאשר נרצה להמעיט את הצימוד בין עצם שהינו מקור הנתונים לבין העצמים שמצבם תלוי במצבו (זה נכון לא רק עבור אמצעי שמע או תצוגה). כך נוכל לנתק/לחבר/לשנות/להחליף כל חלק (עצם) במערכת שלנו ללא פגיעה בפעולת המערכת כולה.

דפוס התכן Observer בא לממש תלות של כמה עצמים במצבו הפנימי של עצם מסוים שנקרא לו "נושא" או "מקור המידע". הוא עושה זאת ע"י כך, שכאשר מצבו הפנימי של העצם הזה משתנה, זה יגרום לשינוי אוטומטי במצבם של העצמים האחרים.

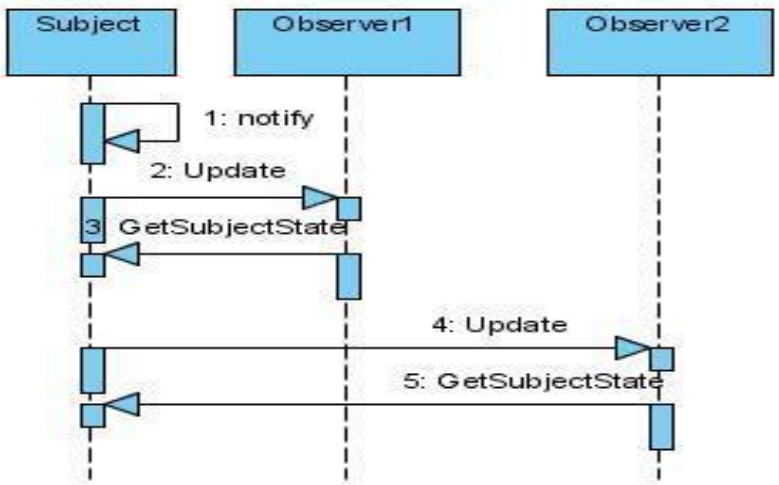
בתרשים המחלקות שבאיור 23 הבא, ניתן לראות כיצד ניתן לרשום (attach/register) עצמים שממממשים מנשק Observer בתוך רשימה שנמצאת בתוך עצם מסוג Subject.

לאחר רישום זה, העצם מסוג Subject יעדכן את כל ה-Observers על שינויים במצבו. למשל, המפתח של המחלקה מסוג ConcreteSubject יעשה זאת ע"י קריאה למתודה Notify מתוך מתודה שגורמת לשינוי במצב העצם (הלקוח קורא לה).



איור 23: תרשים מחלקות של דפוס התכן Observer [Sks]

תרשים הרצף באיור 24 להלן, מתאר את התגובה של דפוס התכן לשינוי במקור הנתונים לאחר שה-Observers נרשמו כבר אצל ה-Subject.



איור 24:תרשים רצף של דפוס התכן Observer [Vetr]

עתה אני מעוניין להרחיב קצת כאן [GoF] . אם נתעמק מעט בפיתרון ובעיקר בתרשים המחלקות של דפוס התכן Observer לעיל, נרצה לשאול למה עיצובו לא הרבה יותר פשוט.

- הפיתרון הנאיבי היה משתמש ב-2 מחלקות בלבד - המחלקה שהיא מקור הנתונים והמחלקה שמציגה את הנתונים.
- למה כל עדכון של ה-Observers במידע שקיים ב-Subject מורכב משתי פעולות לפחות.

נענה על כך בעזרת השאלות והתשובות הספציפיות הבאות.

- שאלה : למה צריך את המחלקה המופשטת Subject ?
  תשובה : זה נעשה לצורך שימוש חוזר. המחלקה Subject יכולה לשמש כמחלקה כללית מתוך ספריית מחלקות לשימוש כללי – היא מכירה רק את המנשק Observer שהינו מנשק כללי מתוך אותה ספרייה וכל עיסוקה הוא בשיוך/הסרה של עצמים שממממשים את Observer מרשימה פנימית ושליחת הודעה (הודעה חסרת פרמטרים) שחל שינוי. לעומת זאת, המחלקה ConcreteSubject היא מחלקה ששייכת לאפליקציה הספציפית.

- שאלה : למה צריך את המנשק Observer ?
  תשובה : באופן זה נוכל לפתח מחלקות שונות שיממשו את המנשק Observer. נוכל באופן דינמי בזמן ריצה לרשום עצם מסוג מחלקה אחרת/נוספת שממממשת את Observer.

- שאלה : למה ConcreteSubject שולח הודעה חסרת פרמטרים (המתודה update) על שינוי מצב המידע שגורר משיכת מידע ע"י ה-ConcreteObserver וזאת במקום שה- ConcreteSubject יבצע דחיפה פשוטה של המידע, כלומר במקום לשלוח הודעה עצם זה יכול פשוט לספר על השינוי במצבו ?
  תשובה : נרצה שמקור הנתונים לא יכיר את ה-Observers. אם הוא ישלח להם נתונים מסויימים אז הוא מגביל אותם לשימוש במנשק מסוים. לעומת זאת, ה-Observers יכולים להכיר את מקור הנתונים כי הם צריכים לשנות את מצבם בהתאמה אליו. כלומר, ניתן לומר שה-ConcretSubject מוגבל לעבודה עם הנתונים שהוא אחראי עליהם ובניגוד אליו ה-Observers תפורים לעבודה מולו.

4) התוצאות של הפיתרון ותאור היתרונות והחסרונות.

דפוס התכן Observer יוצר מערכת הרשמה והפצת מידע מודולארית, דינאמית וחסינה לשינויים. בזמן ריצה ניתן שנות/להוסיף/להסיר מימושים שונים של ה-Observers וניתן גם לשנות מימוש של ה-Subject. והשינויים הללו לא יגרמו לצורך לשנות את הקוד במחלקות האחרות.

יתרונות:

1. דפוס התכן Observer תומך בשידור של אחד לרבים (multicast). ה-Subject שולח הודעה אחת פשוטה מאוד שמשהו השתנה אצלו מבלי לציין למי מיועדת ההודעה. הודעה זאת מגיעה באופן אוטומאטי לכל ה-Observers שנרשמו אצלו אי פעם. זהו הקסם של מנגנון ה - Publish-Subscribe.

2. ה-Subject וה-Observers יכולים להימצא בשכבות שונות של הפשטה במערכת התוכנה בגלל הצימוד החלש ביניהם. למשל, ה-Subject יכול ליהיות בשכבת שנקרא לה DBLayer ואילו ה-Observers יהיו בשכבת התצוגה וישמשו להצגת הנתונים ב-GUI.

3. דפוס תכן זה מדגים שימוש טוב בעקרונות התכנות מונחה העצמים הבאים.

- הסתרה של אספקטים בעיצוב שסביר שישתנו – ה-Observers הספציפיים מוסתרים מפני ה-Subject.

- Open-Close Principle (מחלקה צריכה להיות פתוחה להרחבות אך סגורה לשינויים) - זה נעשה ע"י כך ש-Subject לא עוסקת גם בהצגת המידע, כך נוכל להרחיב אותה להצגה של המידע בצורות שונות בלא שנצטרך לשנות אותה.

- יש לשאוף לצימוד חלש בין עצמים שמשתפים פעולה – Subject לא מודע ל-Observers הספציפיים איתם הוא עובד אלא רק דרך מנשק כללי.

- צריכה להיות רק סיבה אחת לשינוי של מחלקה – המימוש של Subject אמור להשתנות רק אם נרצה לשנות את אופן הטיפול במידע שהוא מחזיק.

חסרונות:

1. מבנה יחסית מורכב – דורש לימוד ורענון ידע, דורש תיעוד טוב בתכנון ובקוד.

2. צריך ניסיון כדי לדעת מתי עדיף לא להשתמש.

3. ה-Observers השונים נאלצים לחקור איזה מידע השתנה ב-Subject. כל מה שהם מקבלים ממנו הוא הודעה שמשהו השתנה.

4. ה-Observers יכולים להיות מימושים שונים שמפותחים בזמנים שונים ואף ע״י גורמים שונים. הם יכולים לשמש בעצמם כ-Subjects ואז שינוי במידע שבמערכת יכול לגרום לפעולה ממושכת ויקרה של עדכונים. ה- Observers לא יטפלו בבעיה כזאת (למשל ע״י התעלמות מהודעות מסוימות) כי הם לא מודעים ל-Observers האחרים במערכת ולכן הם לא מודעים למחיר הכולל של פרסום השינוי [Gof].

5. יש הטוענים ש-ConcreteSubject לא צריך לרשת מ-Subject. הטענה היא שנושא ניהול ה-Observers צריך להיות נפרד מ-ConcreteSubject שתפקידו צריך להיות רק לנהל את המידע השמור בו. לשם כך הם מציעים את דפוס תכן White Board שלא נדון בו כאן. מצד שני, ניהול ה-Observers נעשה בקוד פשוט וקצר שברוב המקרים לא נרצה לשנות לכן צריך לשקול בזהירות אם אנו מעוניינים בתקורה הנוספת.

6. במערכות תוכנה שדורשות זמני תגובה מהירים מאוד נעדיף שלא לשלם את מחיר ההאטה שגורם דפוס תכן זה [Zer].

7.1.2 דפוס התכן Strategy

1) שמות שניתנו לדפוס תכן זה

Strategy ו- Policy – שמות אלו מספרים שדפוס תכן זה עוסק במדיניות/אסטרטגיה/שיטת העבודה [GoF] .

2) הצגת ההקשר והבעיה

אציג את הבעיה ע"י דוגמא פשוטה.

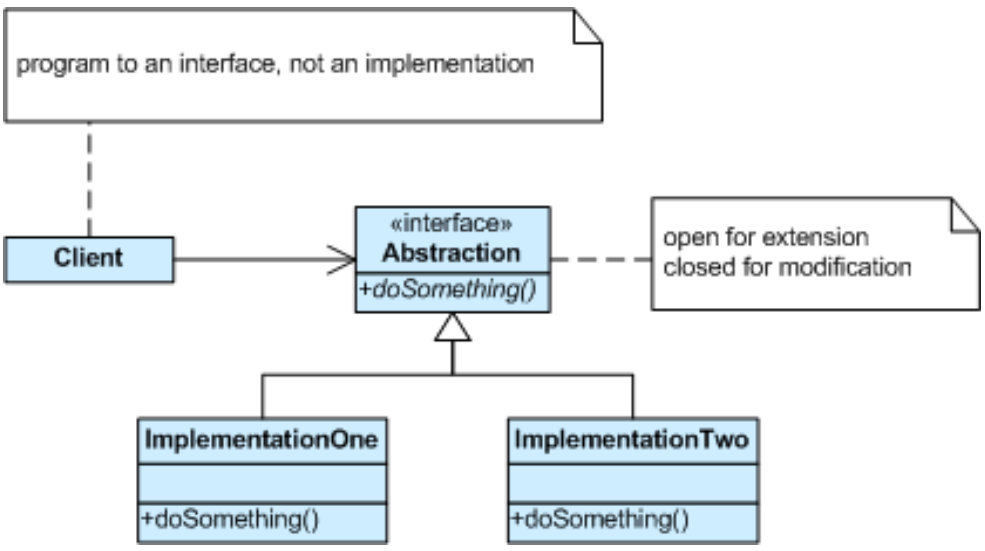
אנו כותבים מחלקה למיון מערך של מספרים אבל אנחנו לא יודעים מראש מגבלות והעדפות. בהתאם לאופי מערך המספרים (למשל, האם הוא כבר ממוין חלקית?) ובהתאם להעדפות כמו העדפה למיון מהיר יותר או העדפה למידת השימוש בזיכרון הפנוי, נרצה לתת ללקוח להחליט על שיטת המיון המיטבית.

למשל, הלקוח יוכל לבחור בין: מיון בועות, מיון בסיס (Radix Sort), מיון מנייה, מיון מהיר, מיון מיזוג.

אז בקוד של הלקוח נוכל לכתוב מתודה עבור כל מימוש ולכתוב מערכת של if... else if... else if... שתקרא למתודה הרצויה לפי העדפת הלקוח אבל אז נצטרך לשנות את הקוד של הלקוח בכל פעם שנרצה לעדכן את שיטת המיון. כמו כן, הקוד של הלקוח יגדל ויהיה מורכב ולכן גם קשה יותר לתחזוקה ופגיע לטעויות.

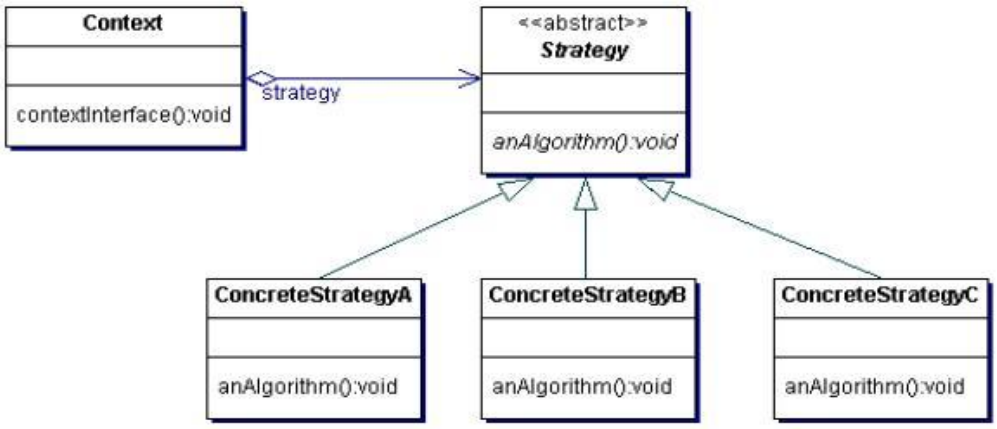
3) הפיתרון

נרצה להחביא מהקוד של הלקוח את המדיניות של בחירת האלגוריתם. הלקוח מבקש פעולה מסוימת אבל הוא לא יודע באיזה דרך היא היא תתבצע. באופן כללי הפיתרון יהיה כפי שמתואר באיור 25 הבא.



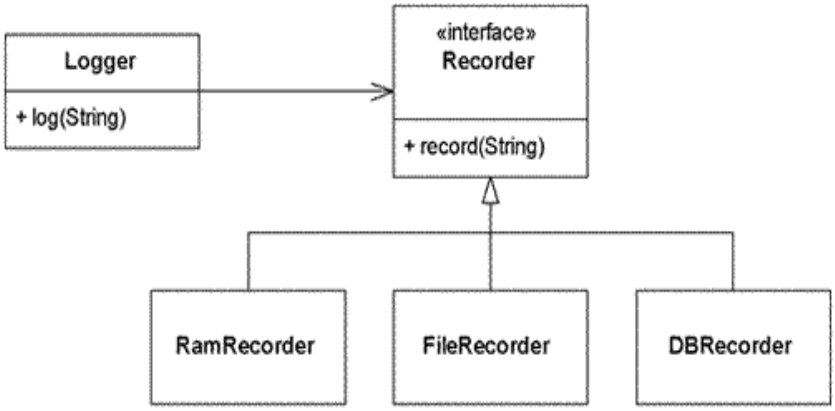
איור 25: תרשים מחלקות שמתאר את העיקרון של דפוס התכן Strategy [Mak]

נותר לנו להוסיף עוד שכבה אחת של הפשטה שתחצוץ בין הקוד של הלקוח לבין המימוש. כפי שניתן לראות באיור 26 הבא, נוספה המחלקה Context. הקוד של הלקוח צריך לעבוד מול המחלקה Context, שרק בה קיים הידע באיזה אלגוריתם ספציפי יש לבחור, ולא ישירות מול המנשק Strategy.



איור 26: תרשים מחלקות של דפוס התכן Strategy [Rav]

איור 27 הבא מדגים שימוש יפה ומעשי בדפוס תכן זה. המנשק Logger משמשת כאן בתפקיד המנשק Context שהוזכר לעיל.



איור 27: תרשים מחלקות שמדגים שימוש בדפוס התכן Strategy [Tod]

כאשר נמצא בקוד שלנו הסתעפות if else if (או בצורתה הסינטקטית האחרת switch) שמתחילה ליהיות גדולה ומורכבת אז זה המקום לחשוב על שימוש בדפוס התכן Strategy.

4) התוצאות של הפיתרון ותאור היתרונות והחסרונות.

דפוס התכן Strategy עוזר לנו להוציא מהקוד של הלקוח את הבחירה באסטרטגיה הרצויה ואת מימוש האסטרטגיה. בסופו של דבר מישהו צריך להחליט באיזה אסטרטגיה (אלגוריתם) לבחור אבל ההחלטה הזאת מוחבאת מהלקוח למשל היא נכתבת מראש בקובץ קונפיגורציה ואז המחלקה Context תקרא את ההחלטה שנעשתה בקובץ הקונפיגורציה (באופן הפשוט ביותר ניתן גם אפילו שההסתעפויות if... else... if... יהיו בתוך הקוד של המחלקה Context).

יתרונות:

1. כמו שראינו בדוגמא לעיל, הלקוח יכול עתה לוותר על הוספת הסתעפויות של if…else if… (או switch) בקוד כשנרצה לבחור באסטרטגיה לפיתרון של בעיה. כך התוכנה תיהיה קלה יותר לתחזוקה.
2. ניתן ליצור עץ הורשה מכל מחלקת Strategy. כך ניתן ליצור משפחות של אלגוריתמים לבחירה.
3. דפוס תכן זה מדגים שימוש טוב בעקרונות התכנות מונחה העצמים, למשל.

- מבנה פשוט ואינטואיטיבי, קל ללמוד ולזכור.
- עיצוב/תכנות מול מנשק ולא מול מימוש – עוזר לפיתוח של קוד מודולארי.
- הסתרה של אספקטים בעיצוב שסביר שישתנו – ההחלטה באיזה אלגוריתם ספציפי לבחור מוסתרת מהלקוח.
- Open-Close Principle (OCP) - מחלקה צריכה להיות פתוחה להרחבות אך סגורה לשינויים. את ההרחבה אנו מבצעים ע״י הוספת מחלקה שמכילה מימוש של כל אלגוריתם חדש וזאת במקום להרחיב את מחלקת הלקוח בכל פעם שנרצה להוסיף/לשנות אלגוריתם.
- יש לשאוף לצימוד חלש בין עצמים שמשתפים פעולה – מחלקת הלקוח לא מכירה את העצמים שמספקים את המימוש.
- צריכה להיות רק סיבה אחת לשינוי של מחלקה – כל מחלקה מטפלת באלגוריתם אחד בלבד.
- העדפת Composition והאצלה על פני שימוש בהורשה – עיקרון זה ממומש ע״י ההוספה של המחלקה Context. זה מאפשר לקוד של הלקוח לא להכיר את המימוש (לפני זמן ריצה).

חסרונות:

1. דפוס תכן זה מגדיל את מספר המחלקות בקוד – מחלקה נפרדת לכל אלגוריתם ספציפי.
2. קיים מנשק קשיח בין המחלקה Context לבין המנשק Strategy. זה אומר שאותו מספר של פרמטרים ואותו סוג של פרמטרים יגיע לבסוף לכל מחלקה ConcreteStrategy גם אם היא מממשת

אלגוריתם פשוט מאוד שאמנם שייך לאותה משפחה של אלגוריתמים אבל הוא אינו זקוק לכל הפרמטרים.

7.1.3 דפוס התכן Iterator

1) שמות שניתנו לדפוס תכן זה

Iterator – שם זה מורה שדפוס תכן זה מאפשר לנו גישה למערך של נתונים (to iterate over)[GoF] .

Cursor – שם זה קיים במשמעות דומה גם בבסיסי נתונים. שם זה מורה שדפוס התכן משמש כסמן שבעזרתו סורקים את מערך הנתונים [GoF].

2) הצגת ההקשר והבעיה

אציג את הבעיה ע״י דוגמא פשוטה.
נניח שאנו מעוניינים לגשת לערכים ששמורים ברשימה מסוימת של עצמים שהינם מופע של אותה מחלקה. ניתן לממש זאת בהתחלה באופן נאיבי באמצעות כך שהלקוח שלנו יודע שהעצמים נשמרים במערך של עצמים ולכן הוא ניגש בעזרת לולאה לכל עצם במערך בצורה ישירה ע״י שימוש באינדקס שמציין את מיקום כל תא במערך. או אפילו גרוע יותר, ציון הכתובת בזיכרון של כל תא במערך - אפילו שפת ה-Assembler מספקת כלים סינטקטיים להימנע מכך (שהרי מערך הוא מבנה נתונים מופשט שלא צריך ליהיות תלוי בטכנולוגיית מחשוב מסויימת).

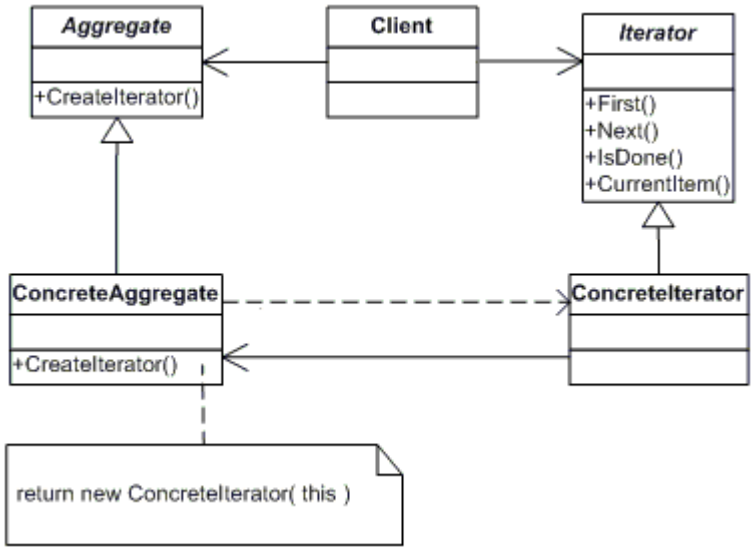
עתה נציג דרישה חדשה, אנו מעוניינים לשנות את אופן שמירת העצמים משיקולי יעילות שימוש בזיכרון (לצורך התגברות על בעיית הפרגמנטציה [fragmentation]) ע״י כך שהם יישמרו ברשימה מקושרת. במקרה זה הלקוח שלנו צריך לשנות את הקוד שלו (קיים צימוד חזק בין הקוד שלו לקוד שלנו).

3) הפיתרון

נרצה שהלקוח לא יכיר את המימוש שמכיל את רשימת העצמים ואת המימוש של סקירת העצמים השמורים. למשל נחשוף ללקוח שלנו רק את האפשרות לעבור לעצם הבא (העצם הבא והמעבר אליו מוגדר ע"י המימוש שמוחבא מהלקוח).

איור 28 להלן ממחיש את החבאת המימוש שנעשית בדפוס התכן Iterator.

המחלקה ConcreteAggregate מכילה את רשימת הנתונים והלקוח צריך לקרא למתודה שלה CreateIterator בכדי שתחזיר ללקוח את העצם מסוג ConcreteIterator שבעזרתו הלקוח יוכל לסרוק את הנתונים (ע"י קריאה למתודה Next למשל שתחזיר לו את הנתון הבא שמוחזק בעצם מסוג ConcreteAggregate).



איור 28: תרשים מחלקות של דפוס התכן Iterator [Dof2]

4) התוצאות של הפיתרון ותאור היתרונות והחסרונות

דפוס התכן Iterator עוזר ללקוח לעבוד ברמת הפשטה גבוהה יותר, למשל עם מבני נתונים (אין הגבלה לשימוש רק במבני נתונים). הלקוח לא צריך לעסוק במימוש של סריקת הנתונים זאת צריכה ליהיות הבעיה של מבנה הנתונים עצמו.

יתרונות:

1. מבנה פשוט ואינטואיטיבי, קל ללמוד ולזכור.
2. דפוס תכן זה מסתיר את המימוש מהלקוח וזה פותר את הבעיות הבאות שיוצרות לולאות איטראציה כמו for ו-while:

- האם להתחיל את האיטרציה מ-0 או מ-1 ? דפוס תכן זה מסתיר את מימוש האיטראציה מהלקוח.
- כאשר משתמשים ב-Index, שהינו מספר, אז לפעמים מפתח התוכנה משנה את ערכו בטעות במהלך ריצת האיטראציה. דפוס תכן זה מסיר את הצורך בשימוש באינדקס.
- כאשר המימוש משתנה אז צריך לשנות את הקוד של האיטראציה אצל הלקוח. דפוס תכן זה מסתיר את מימוש האיטראציה מהלקוח.

3. דפוס תכן זה מדגים שימוש טוב בעקרונות התכנות מונחה העצמים, להלן.

- הסתרה של אספקטים בעיצוב שסביר שישתנו – המימוש של מבנה הנתונים וכן סוג הנתונים מוחבא מהלקוח.
- יש לשאוף לצימוד חלש בין עצמים שמשתפים פעולה – המחלקות בחלק המנשק שהלקוח עובד איתן ישירות לא מכירות את חלק המימוש.
- עיצוב מול מנשק ולא מול מימוש.
- העדפת הרכבה והאצלה על פני שימוש בהורשה – המתודה CreateIterator מונעת תלות של הלקוח בחלק של המימוש ע"י שימוש בהאצלה.

חסרונות:

1. עיצוב מורכב יותר לעומת הדרך הישירה והפשוטה לעבוד עם מבנה הנתונים.
2. ריבוי מחלקות - עבור כל ConcreteAggregate נוספת מחלקה ConcreteIterator.
למשל, בשפת Java, עבור המחלקה ArrayList נצטרך מחלקה ConcreteIterator משלה ועבור המחלקה LinkedList נצטרך גם כן מחלקה ConcreteIterator משלה.

7.2 דפוסי תכן בקטגוריה Creational

הקטגוריה Creational מכילה דפוסי תכן שעוסקים ביצירה של עצמים או איתחול של מחלקות.
דפוסי התכן בקטגוריה זאת משתמשים ביחס של הורשה בין מחלקות או ביחס של הרכבה בין עצמים לצורך החבאה של אופן איתחול המחלקות או יצירת העצמים. קטגוריה זאת מאפשרת גמישות רבה בהחלטות של מה נוצר, מי היוצר, איך נוצר ומתי נוצר (למשל בזמן קומפילציה או בזמן ריצה) [GoF].

7.2.1 דפוס התכן Factory Method

1) שמות שניתנו לדפוס תכן זה

Factory Method – שם זה מרמז שדפוס התכן מספק מתודה שמייצרת ללקוח עצמים [GoF].

Factory – שם זה הוא קיצור לשם המפורט יותר Factory Method. בשימוש למשל בספר [Fbs].

Virtual Constructor – שפת C++ משתמשת בשם זה בכדי לתאר מתודה של מחלקת-אב שמייצרת עצמים שהם מסוג-העל של אותה מחלקה אבל ההחלטה מאיזה סוג-ספציפי יהיה כל עצם תלויה במחלקה היורשת שתממש את ה-Constructor [GoF].
בדומה לשימוש בשם זה בשפה C++, שם זה מתייחס לכך שהלקוח לא מקושר ישירות לסוג העצם שנוצר עבורו אלא רק לייצוג שלו ובעצם ההחלטה הסופית איזה סוג של עצם ליצור תיהיה רק בזמן הריצה של התוכנה.

2) הצגת ההקשר והבעיה

אציג את הבעיה ע"י שאביא דוגמא פשוטה שאציג כאן בקיצור ועם שינויים קלים מהמאמר [Tony]. כמו אנשי מקצוע רבים וטובים אחרים (וכמוני לפני שכתבתי את העבודה המסכמת הזאת), גם האדם שכתב את הדוגמא הזאת טעה בהצגת דפוס התכן Factory Method. לכן אציג את הדוגמא שלו ואתקן אותה ולאחר מכן אסביר את הטעות ואת מהות התיקון.
- הצגת טעות נפוצה ולימוד איך לתקן את הטעות היא דרך נפלאה ללמוד.

נרצה לכתוב קוד שיעזור לתוכנה שלנו לכתוב דוח מעקב של פעולותיה (log/trace). נתחיל בהגדרת מנשק.

```
public interface Trace {

    public void debugTitle( );
    public void debug( String message );
}
```

עתה נממש את המנשק בשני אופנים. מימוש אחד יכתוב לקובץ והמימוש השני יכתוב למסך.

המימוש הראשון כותב לקובץ:

```
public class FileTrace implements Trace {

    private java.io.PrintWriter pw;

    public FileTrace() throws java.io.IOException {
       pw = new java.io.PrintWriter( new java.io.FileWriter( "c:\trace.log" ) );
    }

     public void debugTitle( ) {
           pw.println( "Log started on date:" + new Date( ) );
    }

     public void debug( String message ) {
           pw.println( "DEBUG: " + message );
    }
}
```

המימוש השני כותב למסך:

```
public class SystemTrace implements Trace {

    private boolean enableDebug;

     public void debugTitle( ) {
           pw.println("Log started on date:" + new Date( ) );
    }

    public void debug( String message ) {
            System.out.println( "DEBUG: " + message );
    }
}
```

הלקוח יכול לעבוד ישירות מול המימוש:

```
SystemTrace log = new SystemTrace();
log.debugTitle( );
log.debug( "entering first log" );
```

אבל כך הקוד של הלקוח תלוי ישירות במימוש מסויים. עתה נראה איך כותב הדוגמא פותר זאת ולאחר מכן אתקן את הפיתרון שלו.

3) הפיתרון

נשפר במקצת את הקוד ע"י הוספת מחלקה שתכמס את ההחלטה באיזה מימוש נשתמש:

```
public class TraceFactory {

    public static Trace getTrace() {
        return new SystemTrace();
    }
}
```

כך בחירת המימוש מוסתרת מהלקוח. עתה הקוד של הלקוח ייראה כך:

```
Trace log = new TraceFactory().getTrace();
log.debugTitle( );
log.debug( "entering first log" );
```

כותב הדוגמא הזאת מסביר שכך אנחנו לא נצטרך לשנות את הקוד של הלקוח בכל פעם שנרצה לשנות את המימוש של המנשק Trace. זאת בגלל שהמימוש מוכמס בתוך המחלקה TraceFactoy.

אז אמנם הכימוס הזה של יצירת העצם המסויים, בכדי להסתיר את הפעולה מהלקוח, הוא צעד חיובי אבל זה לא מספיק. הדוגמא לעיל מציגה בעצם פתגם תכנותי (programming idiom). פתגם תכנותי הוא שיטה פשוטה שמתכנתים נוהגים להשתמש בה. הדוגמא לעיל מציגה את פתגם התכנות שנקרא "Static Factory", אנשים מציגים אותו בטעות כאילו הוא דפוס התכן "Factory Method" [Tony].

עתה אציג תיקון לדוגמא לעיל בכדי להפוך אותה לדפוס התכן "Factory Method" ואתן הסברים.

אנסח שתי בעיות שעולות מהדוגמא לעיל:

1. הקוד של הלקוח תלוי ישירות במחלקה TraceFactory (למרות שהשינוי כמוס בתוך המחלקה).

2. ניתנת ללקוח גמישות גדולה מידי לאופן השימוש בעצם החדש מסוג Trace שנוצר עבורו. בדוגמא לעיל, הלקוח יכול לקרא למתודה debugTitle או שלא לקרוא לה בכלל. נרצה לחייב את הקריאה למתודה debugTitle בשלב האיתחול.

בכדי לפתור את שתי הבעיות לעיל, נהפוך את TraceFactory למחלקת אב וגם נוסיף לה מתודה בשם initialize שהלקוח יידרש לקרוא לה ממחלקה שיורשת את TraceFactory:

```
public class TraceFactory {

   //Factory Method pattern calls this method:  AnOperation
    public Trace initialize( ) {
          Trace  trace = getTrace( );
          trace.debugTitle ( );
          return trace;
    }

    //Factory Method pattern calls this method:  FactoryMethod
    public Trace getTrace( ) {
          //Implemented in sub classes only.
    }
}
```

אחת המחלקות היורשות של TraceFactory תיראה כך (היא לא אמורה לדרוס את המתודה initialize):

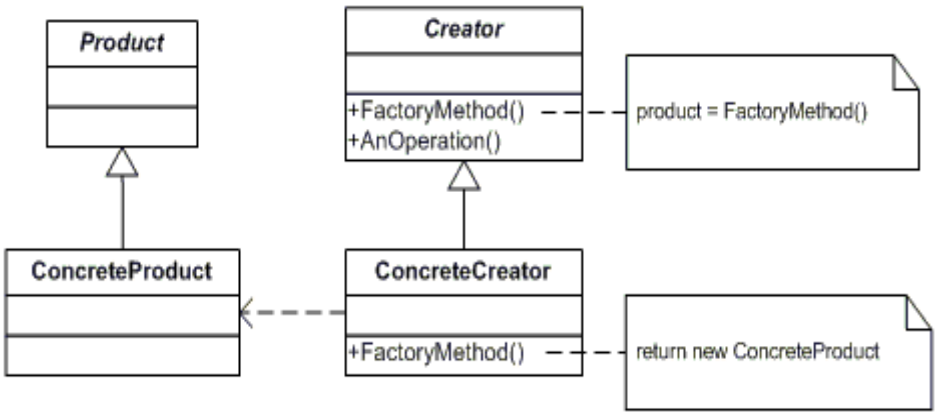
```
// Factory Method pattern calls this class:  ConcreteCreator
public class SystemTraceFactory extends TraceFactory {
    public Trace getTrace( ) {
        return new SystemTrace();
    }
}
```

ואז הקוד של הלקוח יראה כך:

```
  TraceFactory traceFactory = new SystemTraceFactory( );
  Trace trace = traceFactory.initialize( );
  Trace.debug( "Entering first log" );
```

עתה אציג את דפוס התכן "Factory Method" ואוסיף הסברים בכדי להבין את מהות התיקון לעיל.

איור 29 להלן, מראה את המבנה הכללי של דפוס התכן Factory Method.



איור 29: תרשים מחלקות של דפוס התכן Factory Method [Dof3]

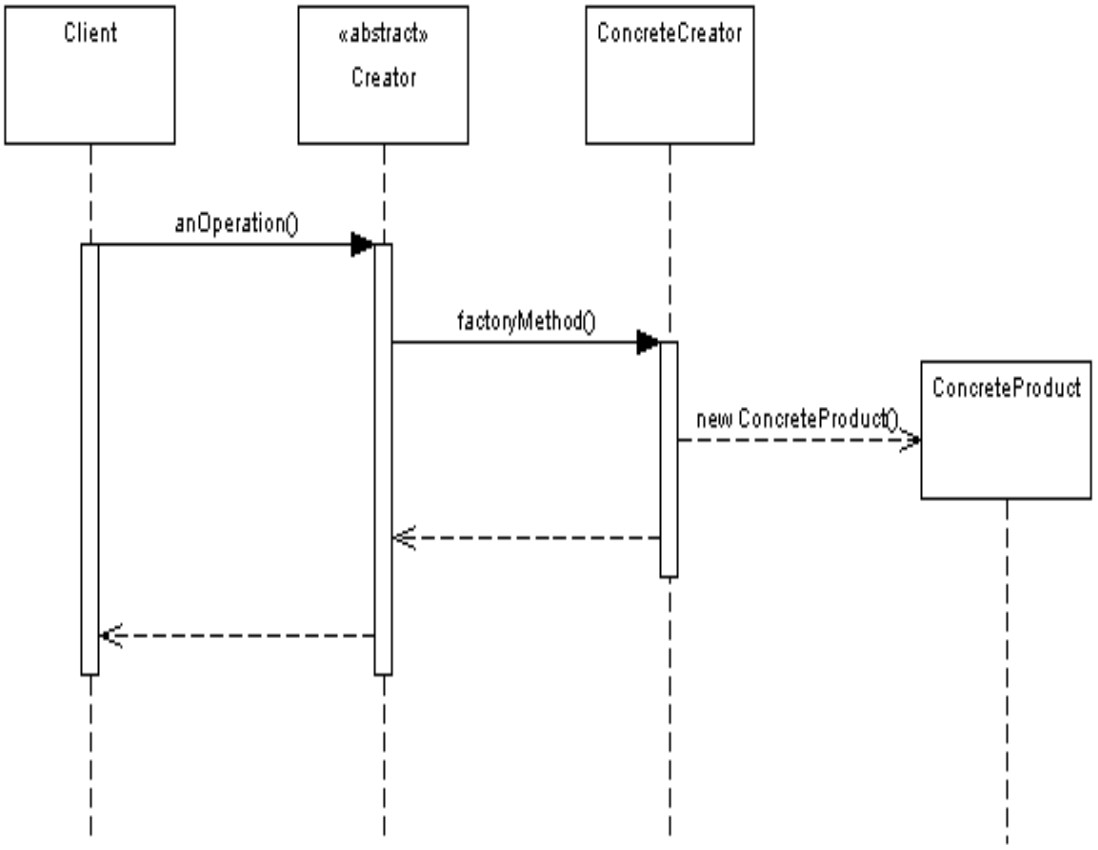
כפי שניתן לראות באיור 29 לעיל, דפוס התכן Factory Method מאפשר שימוש בשתי היררכיות מקבילות של הורשה. אחת עוסקת בהגדרת המוצרים (products) והשנייה עוסקת בדרך הייחודית ליצירת כל מוצר (creators).

נגדיר את דפוס התכן Factory Method באופן הבא [GoF]:

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses." (p. 107)

כלומר, רק המחלקות שיורשות את Creator יחליטו מאיזה מחלקה ליצור את העצם מסוג Product. כפי שראינו בדוגמא לעיל, הלקוח לא מכיר בכלל את המחלקות שיורשות את Product ואפילו לא את Product.

איור 30, להלן, מראה את התקשורת בין העצמים בדפוס התכן Factory Method.

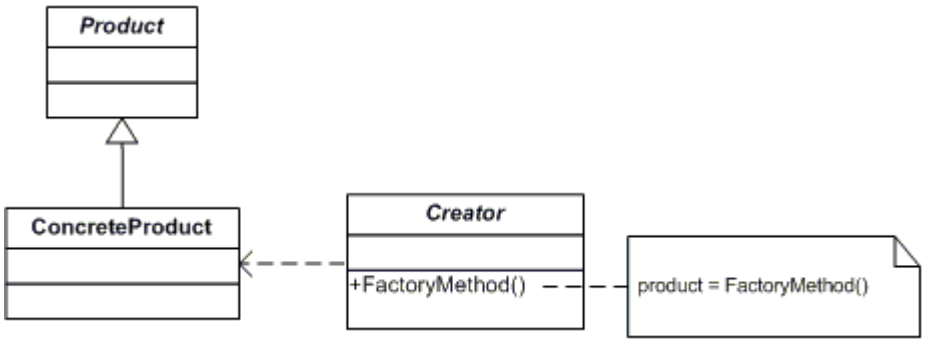


איור 30: תרשים רצף של דפוס התכן Factory Method [Jam]

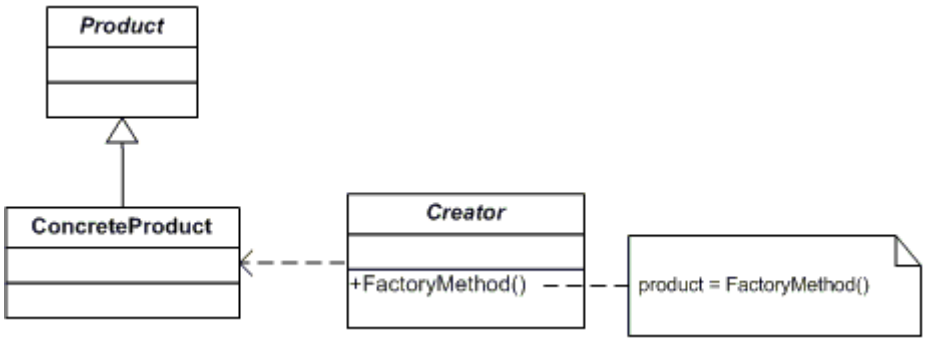
מסגרות-עבודה נוהגות ליצור את המחלקות המופשטות Creator ו–Product, תוך מתן אפשרות למפתחים לרשת מחלקות אלו לצורך הכרת סוג חדש של עצמים למערכת וכן לספק את אופן היצירה הייחודי לסוג החדש של העצמים [GoF].

עתה נתעמק קצת יותר בנושא. אציג עתה את פתגמי התכנות (Programming Idioms) שנקראים "Simple Factory" ו-"Static Factory". אנשים רבים בתעשייה קוראים להם בטעות "Factory Pattern" למרות שהם אינם דפוס תכן כלל אלא רק סגנון תכנות [Fbs]. נראה במה שונה מהם דפוס התכן "Factory Method".

כאשר בוחנים את תרשים המחלקות שבאיור 29 לעיל, יכולה לעלות השאלה האם המחלקה ConcreteCreator (שצריכה לרשת את המחלקה Creator) הכרחית? הרי אנחנו מעוניינים בסך הכל לאפשר להוסיף בקלות ConcreteProduct חדש למערכת בכל פעם, כך שהקוד של הלקוח לא ידע על כך ולא ישתנה. אז אולי פשוט המחלקה Creator תטפל לבדה בנושא, כלומר היא תיצור עצמים מסוג Product מבלי לדחות את ההחלטה איך ליצור אותם למחלקה שתירש אותה. ואכן עושים שימוש במבנה מחלקות כזה בתעשייה וזה נקרא בשם "Simple Factory" אבל זהו לא דפוס תכן [Fbs]. "Simple Factory" איננו נחשב לדפוס תכן מכיוון שהוא מפר את עיקרון ה-"Open-Close Principle" כפי שנראה כאן בהמשך.



איור 31 להלן, מציג את תרשים המחלקות של סגנון התכנות Simple Factory.



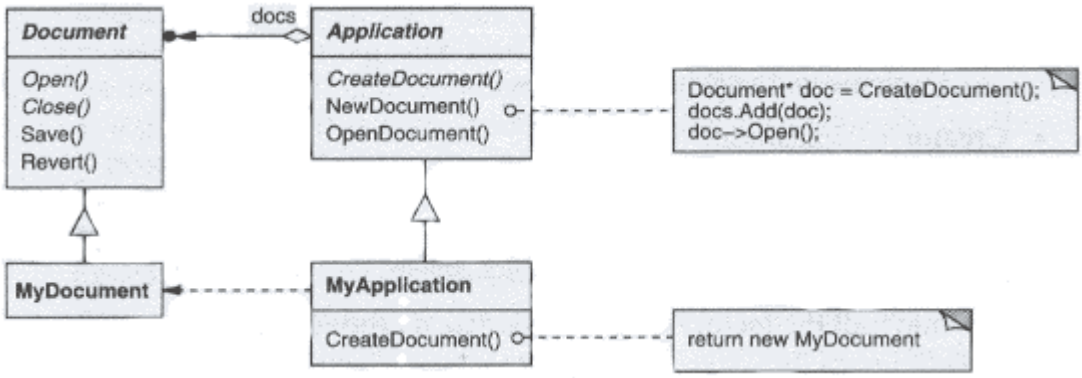
איור 31: תרשים מחלקות של סגנון התכנות Simple Factory [Dof3] (עריכה — א.א.)

אנשים בתעשייה נוהגים אף לעשות זאת יותר פשוט ולתת ללקוח רק מתודות סטטיות במחלקה Creator ולא לאפשר ליצור עצמים ממחלקה זאת (כפי שראינו בדוגמא לעיל). סגנון תכנות זה נקרא בשם "Static

Factory" וגם זה לא נחשב לדפוס תכן [Fbs]. החיסרון של סגנון התכנות Static Factory הוא בכך שהוא מספק פחות גמישות תוך פחות פתיחות להרחבות, זאת הפרה של עיקרון ה-"Open-Close Principle".
כלומר, הלקוח תלוי ישירות במחלקה Creator. כמו כן, זה מונע מאיתנו גמישות למשל בבואנו לכתוב קוד שבודק את ריצת התוכנה שלנו. כלומר, נניח שנרצה שהתוכנה הנבדקת תשתמש בעצם של מחלקה אחרת CreatorMock (התוכנה הנבדקת לא "יודעת" שהיא כבר לא משתמשת במחלקה Creator) שכתבנו רק לצורך הבדיקות (כי לצורך תרחישי הבדיקה אנחנו רוצים להדמות עבור התוכנה הנבדקת סביבת עבודה קבועה וידועה מראש). אז אם המתודות במחלקה Creator הן סטטיות אז לא נוכל לעשות זאת בקלות.

החיסרון של סגנון התכנות Simple Factory הוא בכך שהוא נותן חופש רב מידי לרשת מהמחלקה Creator (כפי שהדגמתי לעיל ואסביר מייד בהמשך). בכך אנו מאבדים שליטה על השימוש בעצמים שמתודת ה-Factory תיצור [Fbs].

בניגוד לסגנונות התכנות שתיארתי לעיל, בדפוס התכן "Factory Method" הלקוח לא קשור למימוש של יצירת המוצרים (Products) ובנוסף לכך בגלל שכל מחלקה חדשה מסוג ConcreteCreator חייבת לרשת את המחלקה Creator אז אנחנו יכולים לחייב את השימוש שנכון בעינינו בכל סוגי המוצרים. זה נעשה באיור 29 לעיל ע"י הוספת המתודה AnOperation למחלקה Creator וכך המחלקה ConcreteCreator יורשת מתודה זאת. מתודה זאת משמשת אותנו כ-Template Method (שאני מציג בקצרה בתת- הפרק "דפוסי תכן ב-JUnit" בעמוד 94) כלומר היא משתמשת במתודה FactoryMethod ועל הלקוח להשתמש בה. נדגים זאת באיור 32 להלן, שבו המתודה NewDocument הינה Template Method שמשתמשת ב-Factory Method שנקראת CreateDocument.



איור 32: תרשים מחלקות של דוגמא לשימוש בדפוס התכן Factory Method [Gof]

4) התוצאות של הפיתרון ותאור היתרונות והחסרונות.

אני מסיק שלקוד שנכתב על פי דפוס תכן זה שותפים מתכנתים שניתן לחלק לשלושה סוגים.

1. המפתח של המחלקות Creator ו-Product. לפעמים מדובר כאן במחלקות שהן חלק מפיתוח של מסגרת-עבודה.

2. המפתח של המחלקות ConcreateCreator ו-Concrete Product. לפעמים מדובר כאן במחלקות שנועדו לשמש כספריה עבור הלקוח הסופי.

3. המפתח שהינו הלקוח הסופי (לצורך הפשטות, כי כמובן שגם לו יכולים ליהיות לקוחות נוספים וכמובן שמפתח אחד גם יכול לחבוש את הכובע של כולם).

דפוס התכן Factory Method מוציא מהקוד של הלקוח הסופי את האחריות לקבוע את הסוג של העצם שהוא רוצה ליצור. ההחלטה על הסוג של העצם שנוצר תתבצע בקוד של המחלקה ConcreteCreator (למשל הקוד הזה יקרא מתוך קובץ קונפיגורציה או מתוך קלט מהמשתמש). ועם זאת דפוס תכן זה משאיר שליטה מראש (למי שכותב את המחלקה Creator – מדובר למשל באדם שמפתח Framewrok) בשימוש עתידי בכל עצם חדש שיווצר [Fbs].

יתרונות:

דפוס תכן זה מדגים שימוש טוב בעקרונות התכנות מונחה העצמים הבאים:

- מבנה פשוט ואינטואיטיבי, קל ללמוד ולזכור.
- הסתרה של אספקטים בעיצוב שסביר שישתנו – המימוש מוחבא מהלקוח.
- יש לשאוף לצימוד חלש בין עצמים שמשתפים פעולה – המחלקות בחלק המנשק שהלקוח עובד איתן ישירות לא מכירות את חלק המימוש.
- עיצוב מול מנשק ולא מול מימוש [Fbs].
- העדפת Composition והאצלה על פני שימוש בהורשה – המתודה factoryMethod מונעת תלות של הלקוח הסופי בחלק של המימוש ע"י שימוש בהאצלה.
- Open-Close Principle - מחלקה צריכה להיות פתוחה להרחבות אך סגורה לשינויים. את ההרחבה אנו מבצעים ע"י הוספת שתי מחלקות. אחת תיהיה סוג חדש לעצם שניצור (ConcreteProduct) והשנייה תיהיה מחלקה שיודעת ליצור אותו בלבד (ConcreteCreator) וזאת במקום להרחיב את מחלקת הלקוח הסופי בכל פעם שנרצה ליצור עצם מסוג חדש [Fbs].

חסרונות:

1. ריבוי מחלקות – עבור כל סוג של Product קיימת מחלקה ConcreteCreator ומחלקה ConcreteProduct.

2. דפוס תכן זה לא כופה על הלקוח להשתמש במתודה AnOperation שהזכרתי לעיל. לכן, הלקוח עלול לעשות שימוש לא רצוי במתודה FactoryMethod מכיוון שהוא יכול שלא להשתמש במתודה AnOperation שמספקת את השימוש הרצוי במתודה FactoryMethod.

7.2.2 דפוס התכן Singleton

1) שמות שניתנו לדפוס תכן זה

Singleton – שם זה מורה שדפוס תכן זה מאפשר לנו ליצור מחלקה שיש לה מופע אחד ויחיד. או באופן יותר כללי, יצירת משאב יחיד לשימושו של הלקוח [GoF].

2) הצגת ההקשר והבעיה

אתן כאן דוגמא מעבודתי כמפתח תוכנה.

כאשר אני מפתח פרויקטים בשפת Java, אז אני נוהג ליצור עצם יחיד שאני קורא לו DbManager שאחראי לפשט את הפעולות של שאר קוד התוכנה מול בסיס הנתונים ולעשות אותן מופשטות (על פי העיקרון של עיצוב מול מנשק ולא מול מימוש).
אני יכול לבקש מעצם זה, למשל, להתחיל טרנזקציה, לסיים טרנזקציה, לטעון מידע מבסיס הנתונים על פי שם של שאילתה בלבד.
וכל זאת מבלי ששאר הקוד, שהינו הלקוח של DbManager, יכיר את הפרטים של העבודה מול בסיס הנתונים המסויים. אני מעוניין שכל מחלקה בקוד, שצריכה לעבוד מול בסיס הנתונים, לא תצטרך ליצור עצם חדש מסוג DbManager, כי הוא עצם שלא צריך לשמור על המצב הפנימי שלו (state) עבור כל שימוש של לקוח מסוים.

שאלה: למה שלא אשתמש רק במחלקה DbManager שתחשוף מתודות-מחלקה לשרות הלקוח ואז לא נצטרך ליצור בכלל מופעים של המחלקה (שהרי ממילא העצם שניצור לא צריך לשמור על מצבו הפנימי)?

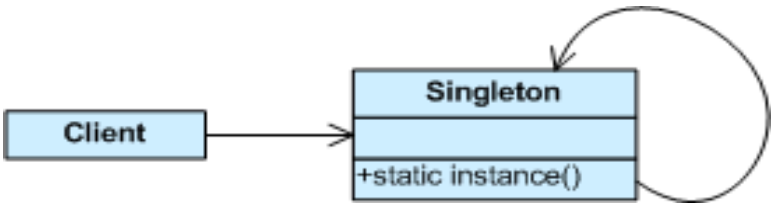
תשובה: בזמן הפיתוח אני עורך בדיקות לקוד שלי ואני מעוניין ליצור תרחישים שונים לבדיקת הקוד. בדיקות אלו נקראות Developer Tests או Unit Tests. אני מדמה (simulate) את בסיס הנתונים וכך אותו קוד שאני בודק אותו (קוד שצורך שירותים מבסיס הנתונים) מקבל בכל תרחיש בדיקה נתונים אחרים ממה שמבחינתו זהו "בסיס הנתונים" על פי מה שאני רוצה לבדוק בהתנהגות של אותו קוד. לכן, בזמן ביצוע הבדיקות, קוד התוכנה יעבוד מול מחלקה אחרת שאני קורא לה DbManagerMock. וזאת מבלי שהקוד יבחין בכך כי DbManagerMock יורשת את DbManager.

3) הפיתרון

ניצור את המחלקה שדפוס התכן קורא לה בשם Singleton ונמנע מהלקוח ליצור מופעים חדשים (עצמים) של אותה מחלקה. את זאת ניתן לעשות ע"י הצהרה מפורשת בקוד של ה-Default Cosnstructor תוך הגדרתו כמתודה פרטית (private). בכך, אנו דורכים על ה-Default Constructor שנוצר אוטומטית כמתודה ציבורית (public) לשימוש הלקוח. כך נעשה גם עבור Constructors אחרים של אותה מחלקה.

ובכל זאת נרצה שהלקוח יוכל פעם אחת ויחידה ליצור את המופע היחיד של מחלקה ובהמשך לגשת לאותו מופע וזאת מבלי לדעת מתי נוצר אותו מופע יחיד. לכן נחשוף מתודת מחלקה ציבורית ששמה יהיה instance. מתודה זאת תיצור רק בפעם הראשונה שנקרא לה את המופע היחיד ובכל הפעמים שנקרא לה היא תחזיר ללקוח את אותו מופע יחיד.

איור 33 להלן, מתאר את דפוס התכן Singleton.



איור 33: תרשים מחלקות של דפוס התכן Singleton [Mak2]

קטע הקוד הבא בשפת Java מממש את המחלקה Singleton באופן הפשוט והנפוץ ביותר.
ניתן להכניס קוד זה כקוד תשתיתי לתוך ספרייה כללית. אך יש לזכור שקוד זה אינו מתאים לעבודה עם ריבוי נימים (Multi-Threaded) – ארחיב על כך עוד מעט.

```
public class Singleton  {
   static private Singleton instance = null;   //Class member that refers to the single object.
    protected Singleton()  { };   //Overrides the Default Constructor, to prevent user access to it.
          static public Singleton instance() {    //This is a class-method.
          if ( null == instance ) {      //Creates a new instance only on the first call to this method.
                instance = new Singleton();
          }
          return instance;    //Returns to the user the one and only instance (object) of this class.
    }
}
```

כדאי לשים לב שמתודת ה-Default Constructor הוגדרה כ-Protected ולא כ-Private.
אם היא היתה מוגדרת כ-Private אז לא היה אפשר לרשת את המחלקה הזאת. ואילו כאשר הגדרנו אותה כ-Protected אז מחלקות שנמצאות באותו ה-Package יכולות לרשת ממנה.

דפוס תכן זה יכול ליהיות ממומש בדרכים רבות אחרות, למשל אפשר להוסיף גמישות לשימוש בהורשה מהמחלקה Singleton כפי שמתואר בקצרה [Gof].

- **Registry of Singletons** - מימוש ע"י שימוש ב-Registry בתוך המחלקה Singleton, כלומר אנו רושמים מראש בתוך המחלקה Singleton עצמים של מחלקות שיורשות מאותה מחלקת Singleton ואז הלקוח יכול לבקש עצם של מחלקה מסויימת עפ"י השם שניתן לו ב-Registry (שהינו רשימה של עצמים יחד עם השם שאנו נותנים לכל עצם כאשר אנו רושמים אותו). נוכל למנוע את הצימוד שמחייב את הלקוח לספק את השם של העצם שהוא צריך למשל ע"י כתיבת שם זה בתוך Environment Variable של מערכת ההפעלה, לשימושו של הקוד במחלקה Singleton.

עבודה עם ריבוי נימים

הבעיה בקוד שהצגתי לעיל היא שעלול להיווצר מצב ששני נימים (threads) של הלקוח יוצרים שני מופעים שונים של מחלקת ה-Singleton וזה קורה כי המתודה Instance לא מתבצעת כיחידה אחת אטומית. נוכל לגרום למתודה Instance להתבצע באופן אטומי אם נגדיר אותה כ- synchronized אבל הגדרת המתודה כולה כ-synchronized גורמת להאטה משמעותית מאוד בעבודתה (סינכרון נימים) וכל זאת רק בשביל שברוב הפעמים נשאל אם המופע כבר קיים. את זה ניתן לפתור למשל ע"י שימוש בדפוס התכן מעולם המקביליות (concurrency) שנקרא "Double Checked Locking" בגלל שהוא עושה שתי בדיקות אם המופע כבר קיים. הוא בודק בדיקה רגילה ומהירה אם המופע קיים ובתוכה בדיקה נוספת זהה אבל מוגדרת הפעם כ-Synchronized ולכן היא אטומית. הבדיקה הפנימית האטומית צורכת הרבה זמן אבל היא מתבצעת רק בהתחלה, בזמן שכמה נימים מנסים ליצור את המופע הראשון. ניתן לקרא על כך בספר [Fbs] ויותר בהרחבה במאמר [Koch] וגם ב - [Hagg].

נציג את הקוד הבא בשפת Java [Fbs], שגורם למופע היחיד להיווצר עוד כאשר המחלקה נטענת מהדיסק לזיכרון. זה פותר את בעיה של שימוש בנימים עם הקוד שהצגתי לעיל.

```
public class Singleton {
    private static final Singleton uniqueInstance = new Singleton();
    private Singleton() { }
    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

אבל כאשר אנו משתמשים בקוד זה אז נפסיד את האיתחול העצל (lazy initialization). ואז גם אם אף לקוח אינו צריך להשתמש ב-Singleton במהלך ריצת התוכנית עדיין נעשות כל הפעולות שכרוכות ביצירת המופע (יכולות ליהיות הרבה כאלו שצורכות זמן ומשאבי זיכרון). כמו כן, נפסיד בגלל כך את האפשרות לאתחל את ה-Singleton רק כאשר יש בו צורך.

4) התוצאות של הפיתרון ותאור היתרונות והחסרונות

דפוס התכן Singleton נותן לנו את הדרך לכפות על הלקוח שימוש במופע (עצם) אחד בלבד של מחלקה. הלקוח כבר לא אחראי ליצית המופע והוא אינו יודע מתי נוצר המופע ואיך.

יתרונות:

1. מבנה פשוט מאוד ואינטואיטיבי, קל ללמוד ולזכור.
2. דפוס תכן זה מדגים שימוש טוב בעקרונות התכנות מונחה העצמים, למשל.

- הסתרה של אספקטים בעיצוב שסביר שישתנו – מסתירים מהלקוח מתי נוצר המופע היחיד, איך הוא נוצר, של איזה מחלקה ספציפית הוא (שיורשת את Singleton).
  כך, ניתן לשנות בהמשך בקלות את מדיניות יצירת העצמים שהלקוח יקבל מבלי שזה ישפיע על הקוד שלו – למשל , ניתן להחליט בהמשך שהלקוח יקבל גישה ליותר ממופע אחד ואף לשלוט במספר המופעים שהלקוח יקבל [Gof].

- עיצוב מול מנשק ולא מול מימוש – המחלקה Singleton נותנת את המנשק הכללי ואפשר לרשת ממנה.

- כפי שהדגמתי (developer tests), דפוס התכן Singleton מוסיף גמישות לעומת שימוש במתודות-מחלקה בלבד. למשל, קל לעדן את המימוש ע"י ירושה מהמחלקה Singleton [Gof].

- ניתן לשנות בהמשך בקלות את מדיניות יצירת העצמים שהלקוח יקבל מבלי שזה ישפיע על הקוד של הלקוח.

- אפשר להחליט שרוצים להשתמש רק במופע אחד של ה-Singleton אבל עדיין יהיה ניתן לשנות בהמשך את מספר העצמים שהלקוח יקבל גישה אליהם וזאת ללא ידיעתו [Mak3].

חסרונות:

1. מפתחים נוטים להשתמש במחלקה זאת כתרוץ לשימוש רב במשתנים גלובאליים, שכידוע פוגעים במודולאריות של התוכנה. שמירת נתונים בעצם מסוג Singlton לא פותרת את בעיית המשתנים הגלובאליים אלא רק מחליפה את שם הבעיה [Mak3].

2. בהרבה מקרים לא כדאי להשתמש ב-Singleton. אפשר למשל, ליצור את העצם היחיד מראש ולספק את הגישה אליו לעצם שצריך את שירותיו (כך עושה למשל מסגרת-העבודה שאני משתמש איתה בעבודתי, שנקראת Spring).

3. בעבודה עם ריבוי נימים (multy-threaded) שדורשים שירותים מעצם ה-Singleton נצטרך להוסיף קוד נוסף שמסנכרן את העבודה של הנימים מול המשאב היחיד שאותו הוא מייצג.

7.3 דפוסי תכן בקטגוריה Structural

הקטגוריה Creational מכילה דפוסי תכן שעוסקים בהרכבה של עצמים או של מחלקות ליצירת מבנים גדולים ומורכבים יותר (למשל, יצירת מבנים רקורסיביים בזמן ריצה).
דפוסי התכן בקטגוריה זאת משתמשים ביחס של הורשה בין מחלקות או ביחס של הרכבה בין עצמים לצורך החבאה של אופן ההרכבה של המחלקות או של העצמים [GoF].

7.3.1 דפוס התכן Adapter

1) שמות שניתנו לדפוס תכן זה

Adapter – שם זה מורה שדפוס התכן בא לתאם בין מחלקות שונות שלא נרצה לשנות אותן אך נרצה שיוכלו לתקשר ביניהן [GoF].

Wrapper – שם זה מתייחס לכך שאנו עוטפים מחלקה במנשק שונה כך שמחלקות אחרות יוכלו לעבוד מולה מבלי להשתנות [GoF].

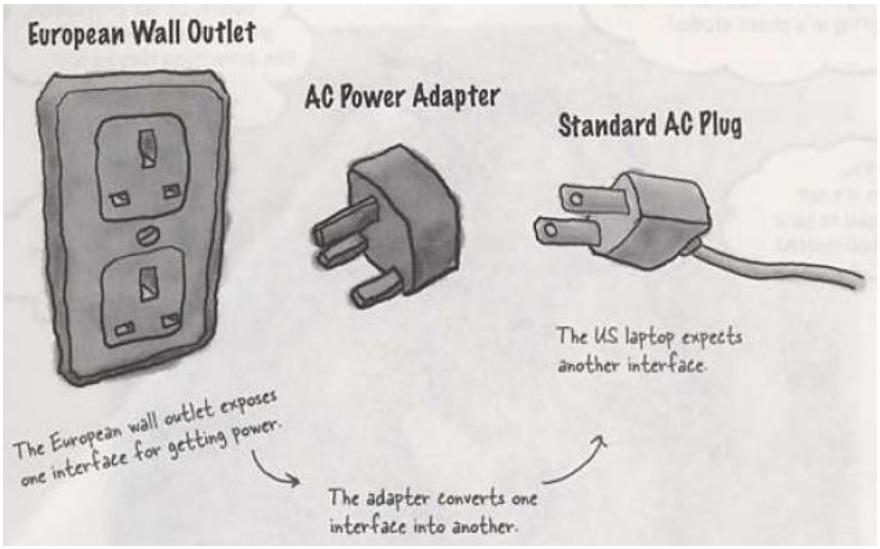
Patcher – זה השם שבחרתי להעניק לדפוס תכן זה. בגלל שבמקום לתקן מערכת תוכנה קיימת, משתמשים בו כדי להוסיף טלאים שיעזרו להאריך את חייה תוך חיסכון במשאבים כמו זמן פיתוח, כסף, תיקון התיעוד של המערכת, בדיקות נסיגה (regression tests) כלומר בדיקה שתת-המערכת ששונתה עדיין עובדת נכון כלומר עפ״י הדרישות המקוריות או לפחות כפי שהיא עבדה לפני השינוי באופן שהסכמנו שהיא תעבוד.

2) הצגת ההקשר והבעיה

נציג את הבעיה ע״י דוגמא פשוטה מחיי היומיום המובאת בספר [Fbs].
יש לנו בקיר שקע חשמל שהחורים שלו עשויים לפי תקו אירופאי אבל קנינו באמריקה מחשב עם תקע לחשמל שמיוצר לפי תקן אמריקאי. כלומר יש לנו שני עצמים שלא יכולים ליצור קשר ביניהם.

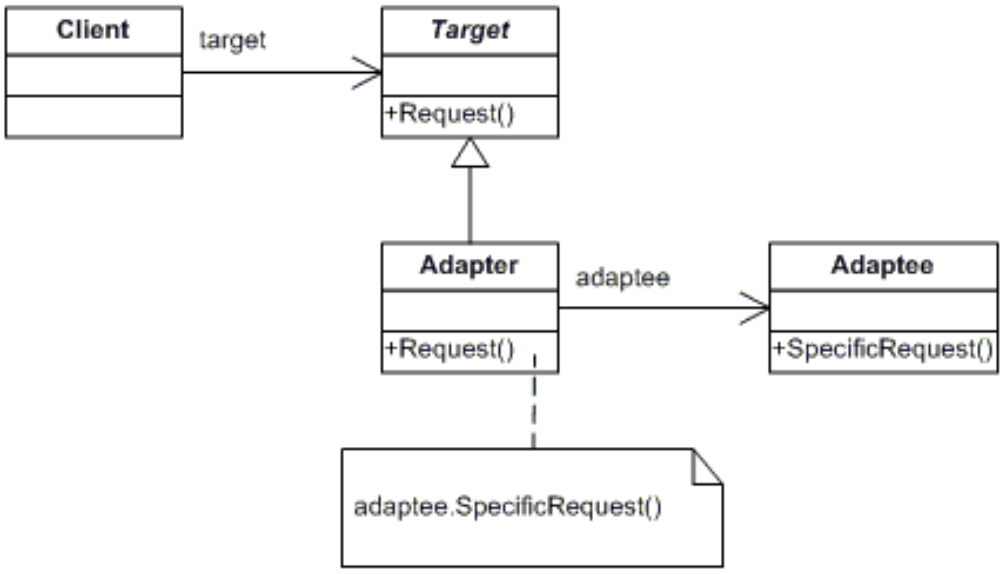
3) הפיתרון

ניצור מתאם שיעזור לשני העצמים לעבוד זה עם זה.



איור 34: שימוש בדוגמא מחיי היומיום לתיאור התבנית Adapter [Fbs]

באיור 35 להלן, ניתן לראות כיצד המחלקה Adapter מתאמת בין המחלקה Client שרגילה לקרא למתודה Request לבין המחלקה Adaptee שחושפת את המתודה SpecificRequst.



איור 35: תרשים מחלקות שמתאר את דפוס התכן Adapter [Dof4]

ארחיב כאן קצת.

שאלה: למה המחלקה Adapter צריכה לרשת מהמחלקה Target? אולי פשוט נכתוב מחלקת Target חדשה?

תשובה: יחס ההורשה משמש כאן בשני התפקידים הבאים.

1. קבלת תנאי החוזה שנחתם בזמנו בין המחלקות הותיקות Client לבין Target. המחלקה Client תוכל "להסכים" לשינוי היישות שהיא עובדת מולה בתנאי שהיישות החדשה תספק את כל ההתחייבויות הישנות.

2. המחלקה Client יכולה ליהיות כתובה בתוך קוד שאולי אפילו אינו נגיש יותר והוא תלוי במחלקה Target שגם הקוד שלה אינו נגיש יותר. ע"י הורשה מהמחלקה Target אנו אומרים למחלקה Client שמבחינתה היא עדיין עובד עם מחלקה מסוג Target.

4) התוצאות של הפיתרון ותאור היתרונות והחסרונות.

דפוס תכן זה נותן מענה להתאמה של מנשקים בין שתי מחלקות קיימות שלא נרצה לשנות אותן. זאת בניגוד לדפוס התכן Bridge שמפריד עוד בשלבים הראשונים של עיצוב המערכת בין הקוד שנותן את השירות לבין המנשק אליו [Gof], כפי שאסביר יותר בסקירה של דפוס התכן Bridge (אז נבין יותר את ההבדלים).
התאמת המנשקים נעשית ע"י הוספה של שכבה (layer) חדשה שחוצצת בין המחלקות ובעצם מתרגמת את בקשת השירות של מחלקה מסויימת לבקשה או אוסף של בקשות שהמחלקות שנותנות שרות תומכות בהן.

יתרונות:

1. מבנה פשוט ואינטואיטיבי, קל ללמוד ולזכור.
2. עוזר לחסוך משאבי פיתוח רבים ויקרים (פירטתי חלק מהם בתחילת הסקירה של דפוס תכן זה לעיל) ע"י תפירת טלאים במקום תיקונים של קוד קיים.
3. דפוס תכן זה מדגים שימוש טוב בעקרונות התכנות מונחה העצמים, למשל.

- הסתרה של אספקטים בעיצוב שסביר שישתנו – המימוש מוחבא מהלקוח.
- יש לשאוף לצימוד חלש בין עצמים שמשתפים פעולה – הלקוח תלוי במחלקה המתאמת בלבד.
- העדפת Composition והאצלה על פני שימוש בהורשה – כפי שנכתב בספר [Gof], ניתן לממש את דפוס התכן הזה גם ע"י כך שהמחלקה Adapter תירש את המחלקה Adaptee. אך נעדיף ש-Adapter תאציל את העבודה למחלקה Adaptee ולא תירש אותה.

חסרונות:

1. הברכה שמביא דפוס תכן זה, בכך שהוא עוזר לחסוך משאבי פיתוח ע"י הימנעות משינוי בקוד קיים, הופכת בהמשך לקללה בגלל שנוטים להמשיך להוסיף עוד ועוד טלאים למערכת. באופן תאורטי

הטלאים האלו פותרים את הבעיות אבל במציאות נגרמות חוסר התאמות שקשה לאתר אותן וגם לתחזק אותן.

2. השימוש בקוד נוסף בשכבה החוצצת (שלפעמים צריך לעשות פעולות מורכבות לצורך התאום בין תת-המערכות) יכול לתת פתח לתקלות לא צפויות נוספות (יותר קוד = יותר סיכוי לתקלות) וכן לגרום לפעולה איטית יותר של המערכת.

7.3.2 דפוס התכן Bridge

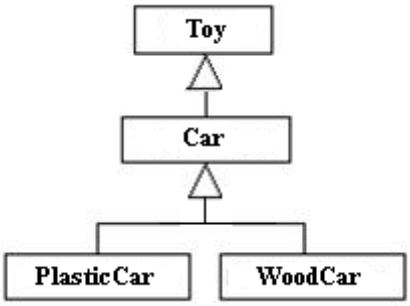
1) שמות שניתנו לדפוס תכן זה

Bridge – כפי שנראה בהמשך, השם Bridge מרמז על ה"גשר" שנוצר בין המנשק שאנו חושפים ללקוח לבין המימוש הפנימי [GoF].

Handle/Body – שם זה מתייחס לשני החלקים שמרכיבים את דפוס התכן. כלומר, שם זה מתייחס למנשק שאנו חושפים ללקוח כידית אחיזה עבורו וכן מתייחס לגוף שהינו המימוש הפנימי [GoF].

2) הצגת ההקשר והבעיה

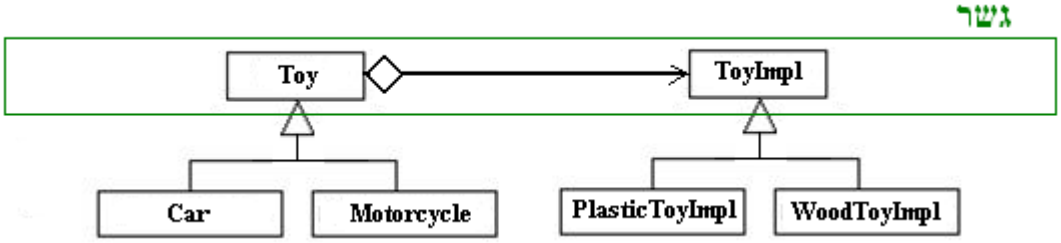
מקור הבעיה הוא שכאשר תוכנה משתמשת במחלקה מסויימת אז היא נעשית תלויה במחלקה הזאת בגלל שהיא משתמשת במנשק של אותה מחלקה.
נציג את הבעיה ע"י דוגמא פשוטה. נניח שקיימים שני מפעלים לייצור צעצועים שהם כלי רכב: מפעל-עץ ומפעל-פלסטיק. מפעל-העץ משתמש רק בעץ ואילו מפעל-פלסטיק משתמש רק בפלסטיק, אבל המוצרים זהים פרט לשימוש בחומר שונה.
נניח ששני המפעלים הזמינו אצלינו מערכת תוכנה שתשלוט על קוי הייצור במפעל שלהם.
בתוכנה שנבנה עבור שניהם, נצטרך ליצור שתי מחלקות עבור כל צעצוע. למשל, לבניית מכונית-צעצוע נצטרך לבנות את המחלקות WoodCar ו- PlasticCar (ראה איור 36 להלן) ובאופן דומה נצטרך עוד שתי מחלקות כאלו לבניית אופנוע-צעצוע WoodMotorcycle ו-PlasticMotorcycle. איך נוכל לצמצם את מספר המחלקות וגם ליצור תוכנה שנוכל להתאים מהר יותר למפעלים נוספים, למשל למפעל-גומי?



איור 36: תרשים מחלקות לדוגמא שמסבירה את דפוס התכן Bridge

3) הפיתרון

נפריד בין המנשק לבין המימוש באופן הבא, כפי שמתואר באיור 37 להלן וכפי שמוסבר אחרי האיור.



איור 37: תרשים מחלקות לדוגמא שמסבירה את דפוס התכן Bridge

- **קודם כל ניצור רק את חלק המנשק שנחשוף ללקוח.**

ניצור מחלקה מופשטת לייצור צעצועים שנקרא לה Toy (דפוס התכן קורא למחלקה כזאת בשם **Abstraction** כי מספקת את הרמה הגבוהה ביותר של ההפשטה). עתה, ניצור מחלקות מופשטות שיורשות את המחלקה Toy ונקרא להן Car ו-Motorcycle (דפוס התכן קורא להן בשם **RefinedAbstraction** כי הן מעדנות/מחדדות את ההפשטה אבל עדיין הן לא מספקות מימוש). המחלקות בחלק זה לא יודעות כיצד לייצר ממש כל צעצוע (באיזה חומר להשתמש וכיצד להשתמש בחומר) אלא רק לקבל מהלקוח בקשות בנייה כלליות כמו למשל בקשה לבניית גלגלים לאופנוע.

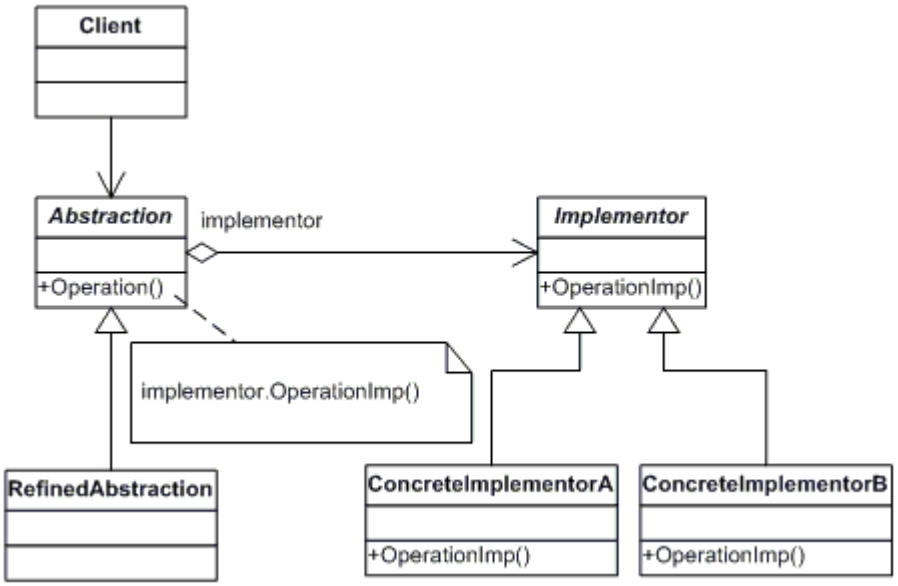
- **עתה ניצור רק את המימוש (שלא חשוף ללקוח).**

ניצור מחלקה מופשטת (למרות שהיא נמצאת בחלק המימוש) לייצור צעצועים שנקרא לה ToyImpl (דפוס התכן קורא למחלקה כזאת בשם **Implementor** כי היא מפרסמת את המנשק של חלק המימוש). עבור כל מפעל ניצור מחלקה שיורשת מהמחלקה ToyImpl, הן ייצרו את הצעצוע הספציפי ויעשו שימוש בחומר הייחודי של אותו מפעל ובטכניקה של אותו מפעל (דפוס התכן קורא למחלקות כאלו בשם **ConcreteImplementor** כי הן מכילות את המימוש הספציפי).

- לבסוף, ניצור "גשר" מהמחלקה Toy אל המחלקה ToyImpl ע"י הוספת יחס של אגרגאציה ביניהן. "גשר" זה הוא הדבר העיקרי בדפוס התכן Bridge מכיוון שהוא זה שגורם להסתרת חלק המימוש מן הלקוח.

כך, הלקוח לא מודע לשימוש במחלקת ה-Implementor ולכל עץ ההורשה שנוצר ממנה. וזה מאפשר למחלקה זאת ולכל המחלקות היורשות ממנה להשתנות ואף לשנות את המנשק שלהן מבלי לגרום לשינוי אצל המחלקה בה משתמש הלקוח.

המבנה הכללי של דפוס תכן זה מובא באיור 38 להלן.



איור 38: תרשים מחלקות של דפוס התכן Bridge [Dof1]

4) התוצאות של הפיתרון ותאור היתרונות והחסרונות.

דפוס התכן Bridge מוסיף שכבה שחוצצת בין הלקוח לבין המימוש.

כפי שמסביר [Gof], מעניין לציין שדפוס התכן Adapter שתיארתי לעיל עושה דבר דומה אז מה ההבדל בין שני דפוסי תכן אלו?

ההבדל הוא בכך ש-Adapter נועד להתאים בין תת-מערכות קיימות בעוד ש-Bridge נועד, עוד בשלב העיצוב, לבנות מערכת שבה כל המחלקות בהיררכיה של המנשק וכל המחלקות בהיררכיה של המימוש יוכלו להשתנות מבלי שהיררכיה אחת תשפיע על חברתה.

יתרונות:

1. העצמה של עיקרון השימוש במנשק. עתה המנשק עצמו יכול להתפתח ללא תלות במימוש.
2. דפוס תכן זה מדגים שימוש טוב בעקרונות התכנות מונחה העצמים הבאים.

- הסתרה של אספקטים בעיצוב שסביר שישתנו – כל ההיררכיה של המימוש מוחבאת מהלקוח.
- Open-Close Principle (מחלקה צריכה להיות פתוחה להרחבות אך סגורה לשינויים) - המחלקות המופשטות שבחלק המנשק ובחלק המימוש לא צריכות להשתנות אך פתוחות להרחבה.
- יש לשאוף לצימוד חלש בין עצמים שמשתפים פעולה – המחלקות בחלק המנשק שהלקוח עובד אין ישירות (ללא המחלקה המופשטת) לא מכירות את חלק המימוש שבדפוס תכן זה.

- עיצוב מול מנשק ולא מול מימוש.
- העדפת Composition והאצלה על פני שימוש בהורשה – ה״גשר״ נוצר ע״י האצלה בכדי למנוע תלות עקיפה של הלקוח בחלק של המימוש.

חסרונות:

מבנה יחסית מורכב – דורש לימוד ורענון ידע, דורש תיעוד טוב בתכנון ובקוד, צריך ניסיון כדי לדעת מתי עדיף לא להשתמש.

7.3.3 דפוס התכן Composite

1) שמות שניתנו לדפוס תכן זה

Composite – דפוס תכן זה מרכיב (**Composes**) עצמים למבנים של עצים בכדי להציג היררכיות של חלק-מכלול [Gof]. אחת המחלקות בדפוס תכן זה נקראת Composite בגלל שכל מופע שלה מייצג מכלול שמורכב ממנה ועוד עצמים אחרים.

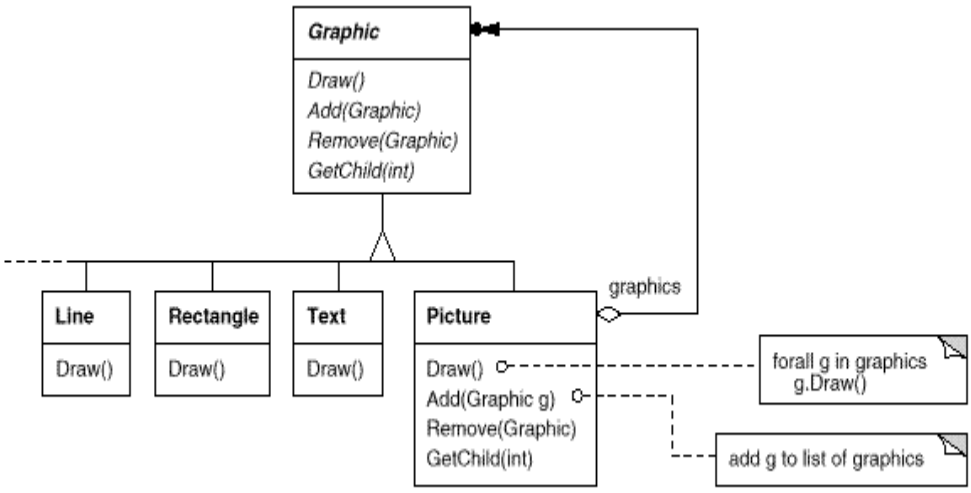
2) הצגת ההקשר והבעיה

בתוכנת הציור שאני מפתח במסגרת הקורס "פרויקט מתקדם במדעי המחשב", אני מעוניין לאפשר ללקוח לסמן צורות שונות ולאחד אותן לקבוצה אחת; לקבוצה הזאת שהוא בחר נקרא Picture-1; ואז אני מעוניין לאפשר לו לצייר ולהזיז את כל הקבוצה כגוף אחד על המסך.
ובאותו אופן הלקוח יוכל לסמן עתה את Picture-1 יחד עם צורה אחרת ואולי גם יחד עם Picture-2 שהלקוח הגדיר מקודם ולאחדן לקבוצה חדשה שנקרא לה בשם Picture-10. זאת התנהגות שאפשר למצוא בהרבה תוכנות שרטוט ווקטורי.

בכדי לממש את זה אצטרך לכתוב קוד די מורכב (שאחר כך יהיה קשה לתחזק ולהרחיב). אצטרך ליצור מיכל (Container) עבור כל קבוצה של צורות. וכדי לאפשר ללקוח להזיז את הקבוצה הזאת, אצטרך לכתוב קוד שעובר על כל הצורות (אוכל להשתמש בדפוס התכן Iterator) שבמיכל ולבקש מהן לזוז.

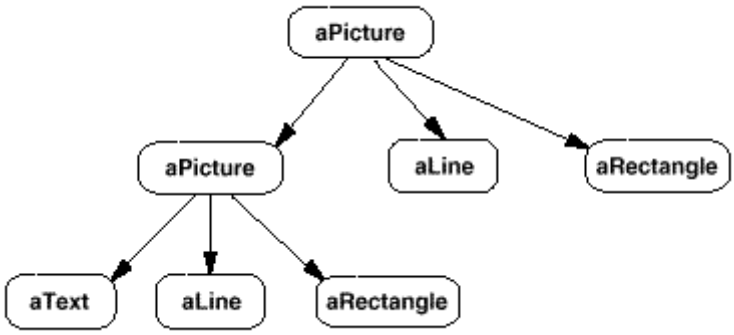
3) הפיתרון

נרצה לא להתעסק עם הטיפול במיכלים. לכן, נסתיר את המיכלים ואת הטיפול בהם בתוך המחלקה המופשטת Graphics כך שמחלקות כמו Picture יוכלו לרשת ממנה את הטיפול במיכלים וגם לדרוך על המתודה Draw ומחלקות כמו Rectangle ידרכו רק על המתודות הפונקציונאליות הבסיסיות כמו Draw, כפי שניתן לראות באיור 39 להלן.



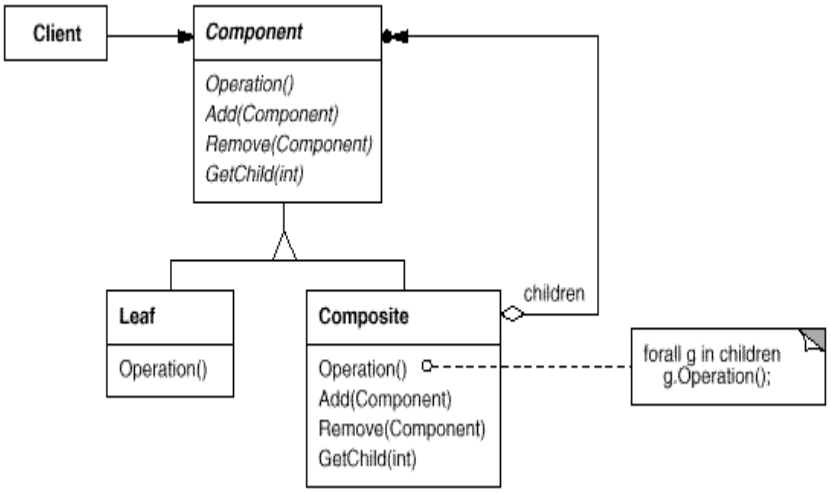
איור 39: תרשים מחלקות - דוגמא לשימוש בדפוס התכן Composite [Gof]

כדאי לשים לב שיחס האגרגאציה בין המחלקה Picture למחלקת-האב המופשטת שלה Graphics יוצר מבנה עצמים רקורסיבי, כלומר עצם מסוג Picture שהינו גם עצם מסוג Graphic יכול לקבץ לתוכו עוד עצמים מסוג Graphics (שזה כולל גם עצמים אחרים מסוג Picture). איור 40 להלן מתאר את המבנה הרקורסיבי שתיארתי.



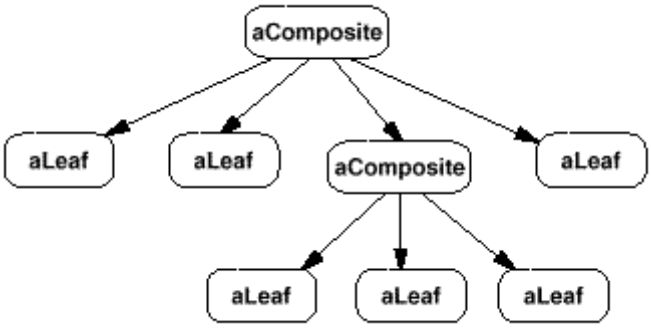
איור 40: דוגמא ספציפית למבנה עצמים רקורסיבי (עץ) שיוצר דפוס התכן Composite [Gof]

באופן כללי, מוצג דפוס התכן באיור 41 להלן.



איור 41: תרשים מחלקות של דפוס התכן Composite [Gof]

מבנה העצמים הרקורסיבי שאפשר ליצור באופן כללי מוצג באיור 42 להלן.



איור 42: מבנה עצמים רקורסיבי (עץ) אפשרי שיוצר דפוס התכן Composite [Gof]

נרחיב כאן. כפי שציינתי בפרק קודם, ישנם קשרים בין דפוסי התכן. למשל, לדפוס התכן Decorator שאני לא מציג בעבודה זאת יש תרשים מחלקות דומה מאוד שמתאר מבנה רקורסיבי של עצמים, אבל יש ביניהם הבדל בשימוש ובמבנה הרקורסיבי שנוצר. Decorator משמש להוספת תפקידים לעצם מסויים בעזרת שרשור רקורסיבי של עצמים אליו. כלומר כל עצם מצביע על העצם הבא בשרשרת ומעביר אליו את בקשת הלקוח וגם כל עצם מוסיף משהו חדש בכדי לשרת את אותה בקשה. כלומר, מדובר כאן בשיפור דינאמי (שניתן להרחיבו בזמן ריצה) של השירות לבקשת הלקוח (למשל, אם הלקוח מבקש להציג חלון אז העצם הראשון יציג את החלון ויבקש מהעצם הבא להציג את חלקו. ואז העצם הבא יציג פסי גלילה לחלון שהוצג). וזאת בניגוד ל - Composite שנותן לנו שירות של יחס הכלה דינאמי בין עצמים [Gof].

כדאי להעיר שדפוס התכן Chain of Responsibility אפילו דומה יותר ל – Decorator (שניהם יצרים שרשרת

רקורסיבית של עצמים) אבל גם כאן מדובר בתפקידים שונים (מטרתו של Chain of Responsibility היא שהלקוח יוכל לבקש שירות ללא ידיעה ברורה איזה עצמים נתנו לו את השירות. זה מודגם ב-[Fbs] ע״י לקוח במסעדה שמזמין מנה מהמלצר שמעביר את ההזמנה לשף).

4) התוצאות של הפיתרון ותאור היתרונות והחסרונות.

דפוס התכן Composite מאפשר לנו ליצור מבנים רקורסיביים של הכלה בין עצמים. לאחר שהמבנה נוצר קל ללקוח לשדר הודעה/בקשה לכל העצמים שבעץ או בתת-עץ מסוים.

יתרונות:

1. המבנה הרקוסיבי שנוצר נותן לנו גמישות גדולה בבניית עצים שונים בזמן ריצה לפי רצונו של הלקוח.
2. אין צורך לבקש שירות בנפרד מכל עצם באופן מפורש. הבקשה של הלקוח לקבלת שירות עוברת במהירות בתוך מבנה העצמים ללא צורך ביצירת ערוץ תקשורת נפרד וזאת ללא תלות בצורת המבנה שיתקבל.
3. דפוס תכן זה מדגים שימוש טוב בעקרונות התכנות מונחה העצמים הבאים:
   - הסתרה של אספקטים בעיצוב שסביר שישתנו – הטיפול במיכלים מוסתר מהלקוח.
   - Open-Close Principle (מחלקה צריכה להיות פתוחה להרחבות אך סגורה לשינויים) - המחלקה המופשטת Component לא צריכה להשתנות אך פתוחה להרחבה ע״י מחלקות Composite ו-Leaf.
   - יש לשאוף לצימוד חלש בין עצמים שמשתפים פעולה – כל עצם מסוג Composite יודע רק שהוא מקבץ עצמים אחרים מסוג Component ולא יותר מזה.
   - עיצוב מול מנשק ולא מול מימוש – למשל, הלקוח עובד מול המחלקה Component ולא חייב להכיר את המחלקות שיורשות ממנה.

חסרונות:

4. נוצר עץ של עצמים ולכן נצטרך להפעיל אלגוריתמים רקורסיביים לפעולות כמו חיפוש של עצם מסויים. אלגוריתמים רקורסיביים נחשבים לאלגוריתמים מורכבים יותר ולכן יותר קשה לתחזק אותם ולתעד אותם.
5. באופן דומה למה שראינו בהצגה של דפוס התכן Observer, כאן בקשה שהלקוח מבקש מעצם שנמצא במעלה העץ תתורגם לבקשות רבות לעצמים במורד העץ. לכן, לפעמים הלקוח לא יהיה מודע למשמעות

(מבחינת עבודת התוכנה) של היענות המערכת לבקשתו (בעיקר בגלל שבהרבה מקרים העץ לא נבנה מראש ע״י המפתח אלא על ידי גורמים אחרים כמו למשל הלקוח עצמו).

8 שימוש בדפוסי תכן במסגרות-עבודה ובסביבות פיתוח בתעשייה

דפוסי תכן נמצאים בשימוש נרחב במסגרות-עבודה ובסביבות פיתוח שונות בתעשיית התוכנה. בעמודים הבאים אסקור כאן בקצרה דוגמאות לכך.

8.1 דפוסי תכן ב-Spring

Spring הינו מסגרת-עבודה לפיתוח תוכנה מבוסס קוד פתוח, שאני משתמש בו בעבודתי והוא בשימוש נרחב מאוד ע"י מפתחי Java בעולם [Spri]. עיקרון מרכזי ב-Spring, כמו בהרבה מסגרות-עבודה אחרות, הוא השימוש בדפוס התכן "הזרקת-תלות" (Depency Injection או בקיצור DI). דפוס התכן DI הינו מימוש של עיקרון "היפוך-השליטה" (Inversion of Control או בקיצור IoC). בקצרה, עיקרון "היפוך השליטה" אומר שבמקום שהקוד שלך יקרא לקוד אחר (שנמצא למשל בספריות של קוד תוכנה), אז קיים קוד אחר (למשל, קוד התוכנה של מסגרת-עבודה) שייקרא לקוד שלך ובנוסף הקוד האחר גם ייקבע את המימוש של הקוד האחר שהקוד שלך תלוי בו. בנספח א' שבעמוד 109, צירפתי מאמר שכתבתי במקום העבודה שלי [Eli] שמרחיב על Spring ועל DI ו-IoC ומתבסס בעיקר על [Mart1].

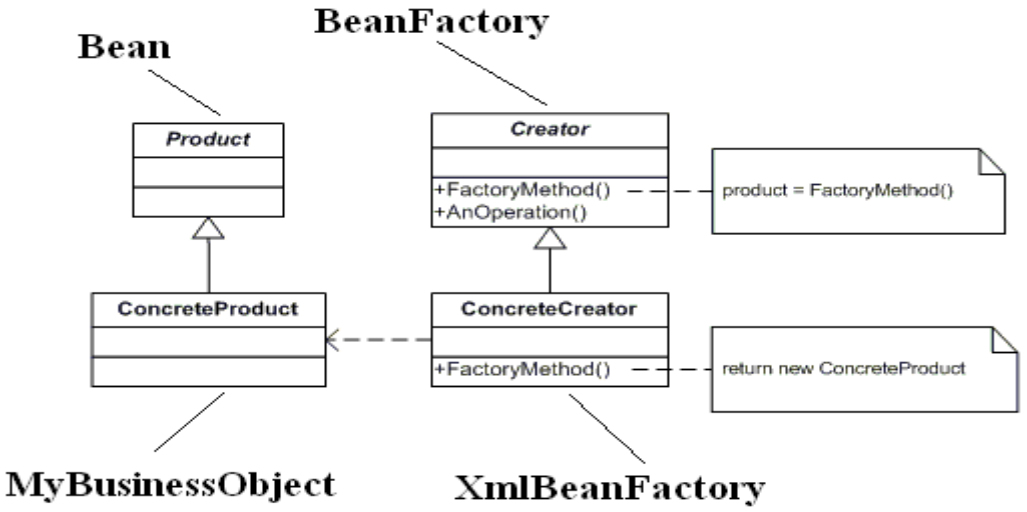
1) Spring משתמש בדפוס התכן Factory Method

Spring משתמש בדפוס התכן Factory Method לטעינת Beans (השם "Bean" מתאר מחלקות לשימוש כללי שנעשה בהן שימוש נרחב בשפת Java) באמצעות BeanFactory (וגם באמצעות ApplicationContext שמרחיב את BeanFactory). להלן דוגמא של קוד [John].

```
BeanFactory bf = new XmlBeanFactory(new ClassPathResource("myFile.xml", getClass()));
MyBusinessObject mbo = (MyBusinessObject) bf.getBean("exampleBusinessObject");
```

הקוד לעיל קורא את הגדרת ה-Bean מהקובץ myFile.xml (בדוגמא הבאה אדגים תוכן של קובץ כזה) ואז גורם למשתנה mbo להצביע על העצם שקיבלנו (עצם שנוצר עכשיו או נוצר בעבר).

ניזכר בתרשים המחלקות של דפוס התכן Factory. באיור 43 להלן, הוספתי לתרשים מחלקות זה איך דפוס התכן קורא לכל מחלקה שמוזכרת בקוד לעיל.



איור 43: תרשים מחלקות להדגמת שימוש בדפוס התכן Factory Method [Dof3] (עריכה — א.א.)

2) Spring משתמש בדפוס התכן Singleton

לאחר שראינו דוגמא לקבלת Bean בעזרת Spring ע"י שימוש בדפוס התכן Factory Metod, נראה דוגמא נוספת כזאת. אבל הפעם נראה בנוסף שימוש בדפוס התכן Singleton.
דוגמא יפה לשימוש של Spring בדפוס התכן Singleton ניתן לראות בקטע הקוד הבא מתוך [Devx2] שקורא את קובץ ה-XML שתוכנו מובא לאחר קטע הקוד.

```
InputStream is = new FileInputStream("src/examples/spring/beans.xml");
BeanFactory factory = new XmlBeanFactory(is);

DataProcessor dataProcessor = (DataProcessor) factory.getBean("fileDataProcessor");
Result result = dataProcessor.processData();
```

להלן תוכן קובץ ה-XML. הדגשתי את השימוש בדפוס התוכן Singleton.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN" "http://www.springframework.org/dtd/spring-
beans.dtd">

<beans>
   <bean name="fileDataProcessor"
      class="examples.spring.DataProcessor"
      singleton="true">
      <constructor-arg>
         <ref bean="fileDataReader"/>
      </constructor-arg>
   </bean>
   <bean name="fileDataReader"
      class="examples.spring.FileDataReader"
      singleton="true">
      <constructor-arg>
         <value>/data/file1.data</value>
      </constructor-arg>
   </bean>
</beans>
```

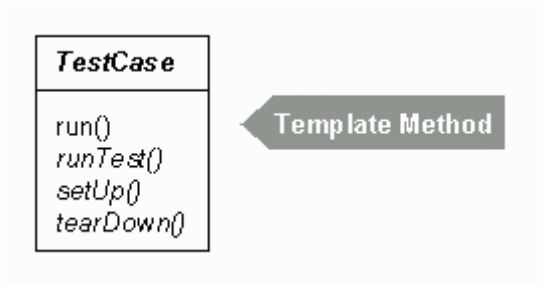
8.2 דפוסי תכן ב-JUnit

Junit משתמש בדפוס התכן Template Method

JUnit הינו מסגרת-עבודה, שמטרתה לעזור למפתח התוכנה ליצור תרחישי בדיקה לקוד שהוא כתב. כלומר לעזור למפתח התוכנה ליצור Developer Tests שנקראים גם Unit Tests ומכאן ניתן השם JUnit. הרעיון הוא לכתוב קוד תוכנה שיריץ תרחישי בדיקה שונים על הקוד הנבדק.

במהלך הדוגמא שאתן כאן, נכיר בקצרה את דפוס התכן Template שנקרא גם "האבא של ה-Frameworks" על כך שהוא נותן את הבסיס ליצירת Hot Spots. כלומר הוא מאפשר למפתח התוכנה שמשתמש במסגרת-העבודה, לכתוב קוד במקומות מסויימים שמסגרת-העבודה תעשה בהם שימוש במהלך עבודתה.
מקומות אלו נקראים Hot Spots. שיטת עבודה זאת לא מחייבת שימוש בעיצוב מונחה עצמים, אפשר למשל להשתמש ב-Callback Functions. כלומר אנחנו מספקים למסגרת-העבודה את הכתובת/מצביע/ייחוס/ידית אחיזה (address/pointer/reference/handle) להפעלת הפונקציה/פרוצדורה שכתבנו ואז במהלך עבודתה מסגרת-העבודה תשתמש בקוד שלנו. עתה נראה דוגמא כיצד זה נעשה בעיצוב מונחה עצמים.

JUnit מספק מחלקה בשם TestCase שחושפת בעיקר את המתודות הבאות כפי שמוצגות באיור 44 להלן.
בהמשך אסביר למה אנו קוראים למתודה run בשם Template Method.



איור 44: תרשים מחלקות שמתאר את המחלקה TestCase של JUnit [Forg]

המחלקה TestCase מספקת מחזור חיים שלם של הקוד הבודק. כלומר בתחילת הבדיקה נרצה לקרוא למתודה setup ואז נרצה להריץ את הבדיקה עצמה ע"י קריאה למתודה runTest ולבסוף נרצה להריץ קוד שמחזיר את המצב שלפני הבדיקה ע"י קריאה למתודה tearDown. המתודה run פשוט קוראת למתודות האחרות לפי הסדר הנכון.

כאשר מפתח התוכנה רוצה לכתוב קוד שיבדוק את הקוד שלו אז הוא כותב מחלקה שיורשת מהמחלקה TestCase לעיל. JUnit מספקת את המימוש הבא [Forg] למתודה run.

```
public void run()  {
    setUp();
    runTest();
    tearDown();
}
```

מפתח התוכנה צריך לממש במחלקה שלו שיורשת מ-TestCase את המתודות setup, runTest ו-TearDown. ואז הוא צריך לקרוא למתודה run של העצם שנוצר מהמחלקה שלו והיא תדאג לקרוא למתודות בסדר הנכון שהוכתב מראש. כלומר, המפתח לא צריך לממש במחלקה שלו את המתודה run אלא רק לרשת אותה ולקרוא לה. המתודה run נקראת Template Method כי היא מספקת תבנית לאופן הקריאה למתודות האחרות.
המתודות setup, runTest ו-TearDown מספקות Hot Spots שיוצר מפתח התוכנה.
מתודות אלה נקראות גם בשם Hook Methods.

זה הרעיון של דפוס התכן Template Method כפי שהוא בא למימוש ב-JUnit.

8.3 דפוסי תכן ב-Java JDK

אתן כאן שלוש דוגמאות לשימוש של Java JDK בדפוסי תכן.

1) Java JDK משתמש בדפוס התכן Observer

המחלקה המופשטת Java.util.Observerable היא בעצם המחלקה המופשטת Subject של דפוס התכן Observer (ראה איור 23 : תרשים מחלקות של דפוס התכן Observer [Sks]) המנשק Java.util.Observer הוא כפי ששמו מעיד עליו המנשק Observer של דפוס התכן Observer.

ה – GUI Event Model של Java מבוסס על דפוס התכן Observer.
למשל כפתור "OK" במנשק הגראפי של האפליקציה שלנו יהיה מיוצג ע"י עצם שישמש כ-Event Source בתיעוד של Java אבל עצם זה בעצם מתפקד כ- ConcreteSubject.

חלון גראפי יהיה מיוצג כעצם שישמש כ-Event Listener בתיעוד של Java אבל עצם זה בעצם מתפקד כ-ConcreteObserver. החלון הגראפי יכול לממש מנשק ActionListener, שבעצם מתפקד כ-Observer. כמו שראינו בדפוס התכן Observer, נצטרך לרשום את החלון הגראפי כ-Event Listener (שזה בעצם Observer) בתוך העצם שמייצג את הכפתור. למשל, נהוג שכאשר החלון הגראפי נוצר הוא יוצר את עצם הכפתור ורושם את עצמו אצל הכפתור כך : okButton.addActionListener(this).

2) Java JDK משתמש בדפוס התכן Iterator

הדוגמא הפשוטה הבאה [Prac1] מציגה שימוש במחלקה Iterator שמספק ה-Java JDK (ראה איור 28 : תרשים מחלקות של דפוס התכן Iterator [Dof2].

הערה : בקוד הבא כתוב "Iterator<String>" שזה שימוש ב-Generics, שדומה מאוד לשימוש ב-Templates בשפת C++. כלומר, אנו מבקשים להשתמש באיטראטור למחרוזות.

```
import java.util.*;

public final class LoopStyles {
      public static void main( String... aArguments ) {
            List<String> flavours = new ArrayList<String>();
            flavours.add("chocolate");
            flavours.add("strawberry");
            flavours.add("vanilla");
            useWhileLoop( flavours );
            useForLoop( flavours );
      }

      private static void useWhileLoop( Collection<String> aFlavours ) {
           Iterator<String> flavoursIter = aFlavours.iterator();
           while ( flavoursIter.hasNext() ){
                 System.out.println( flavoursIter.next() );
           }
      }

      // Note that this for-loop does not use an integer index.
      private static void useForLoop( Collection<String> aFlavours ) {
            for ( Iterator<String> flavoursIter = aFlavours.iterator(); flavoursIter.hasNext(); ) {
            System.out.println( flavoursIter.next() );
      }
   }
}
```

הגרסאות האחרונות של Java מספקות דרך נוחה יותר למפתחי התוכנה להשתמש בצורה מרומזת בלבד בדפוס התכן Iterator. דרך נוחה זאת לשימוש מפתח התוכנה ללא שינוי במהות נקראת "סוכר סינטקטי" (Syntax Sugar). להלן דוגמא של קוד [Prac2].

```
List<String> trees = Arrays.asList("Maple", "Birch", "Poplar");
for (String tree: trees) {
      log(tree);
}
```

3) Java JDK משתמש בדפוס התכן Facade

למדתי את הקורס "תכנות מתקדם בשפת Java" במסגרת לימודיי לתואר שני במדעי המחשב באוניברסיטה הפתוחה. את הקורס הזה ליווה הספר [Phd].

מובאת בספר הדוגמא הבאה לשימוש של ה-Java JDK בדפוס התכן Facade.
המחלקה URL שנמצאת ב-package שנקרא java.net משמשת כ-Facade.
המחלקה מכילה מצביע לעצם מהמחלקה InetAddress שמכיל מידע על כתובת ה-IP של המחשב המארח. כמו כן, המחלקה מכילה מצביע לעצם מהמחלקה URLStreamHandler שתפקידו הוא ליצור את התקשורת לכתובת ה-URL שבמחשב המארח.

הלקוח של המחלקה URL משתמש בשרות של המחלקות שהוזכרו לעיל אבל זאת מבלי שהלקוח יידע כיצד העצמים של המחלקות האלו מבצעים את תפקידם מאחורי העצם URL.

8.4 דפוסי תכן ב-C# SDK

אתן כאן דוגמא לשימוש של C# SDK בדפוסי תכן.

C# SDK משתמש בדפוס התכן Strategy

המחלקה List משתמשת בדפוס התכן Strategy (ראה איור 25 : תרשים מחלקות שמתאר את העיקרון של דפוס התכן Strategy [Mak]). היא נותנת גמישות ללקוח לבחור את אסטרטגיית המיון ע"י כך שהוא יכול לממש את המנשק IComparer כפי שניתן לראות בקטע הקוד להלן שמציג את המתודה Sort של המחלקה List.

```
public void Sort(IComparer<T> comparer)
```

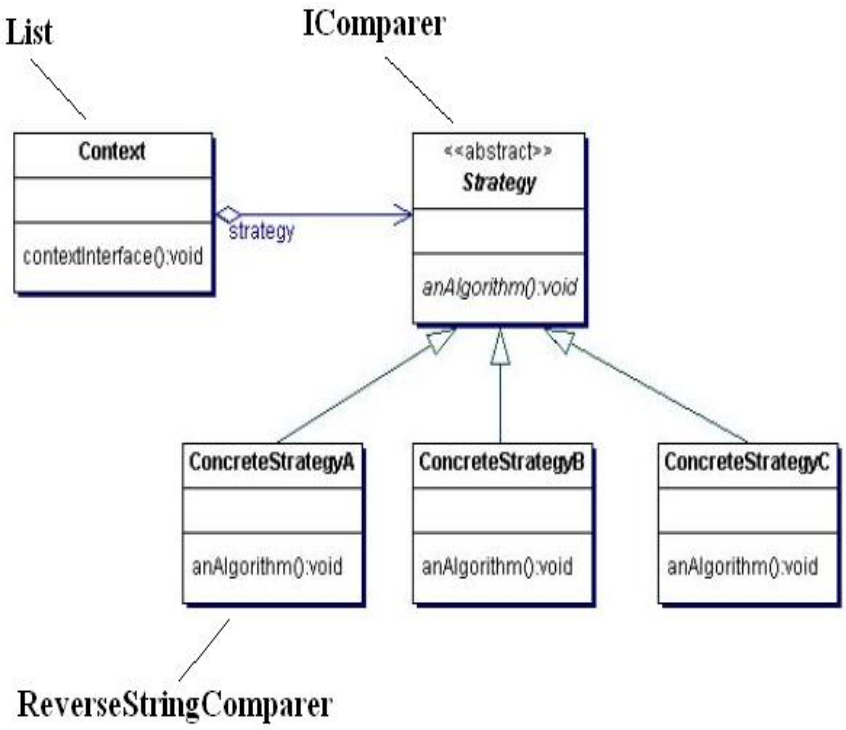
להלן דוגמא של קוד [Dijk].

```
using System;
using System.Collections.Generic;

namespace IComparer {
      public class ReverseStringComparer : IComparer<string> {
            public int Compare(string x, string y) {
            // We reverse the result by flipping the input parameters.
            return String.Compare(y,x);
      }
   }

  class MainClass {
      public static void Main(string[] args) {
            List<string> myList = new List<string>();
            myList.Add("a");
            myList.Add("b");
            myList.Add("c");
            ReverseStringComparer myComparer = new ReverseStringComparer();
            myList.Sort(myComparer);
      }
  }
}
```

ניזכר בתרשים המחלקות של דפוס התכן Strategy. באיור 45 להלן הוספתי לתרשים מחלקות זה איך דפוס התכן קורא לכל מחלקה שמוזכרת בקוד לעיל.



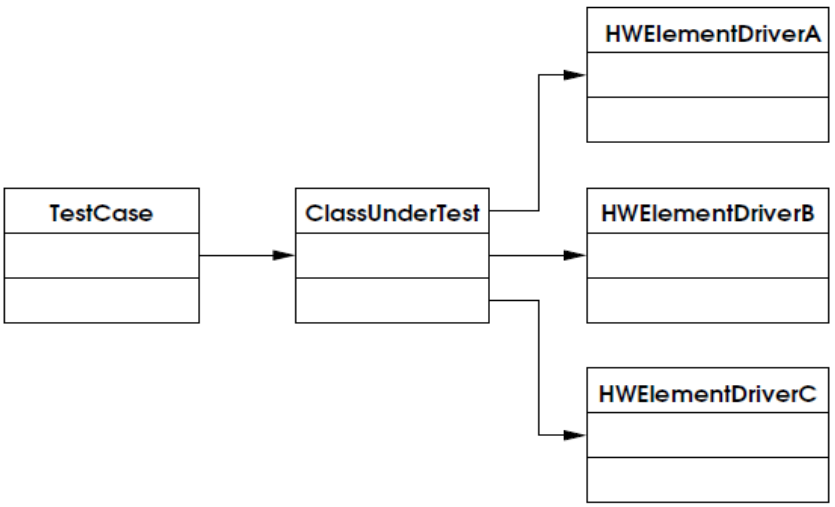
איור 45: תרשים מחלקות להדגמת שימוש בדפוס התכן Strategy [Ravi] (תוספות — א.א.)

9 דוגמאות לשימוש בדפוסי תכן בתהליך פיתוח תוכנה

9.1 תוכנה לבדיקת תוכנה בשילוב עם חומרה

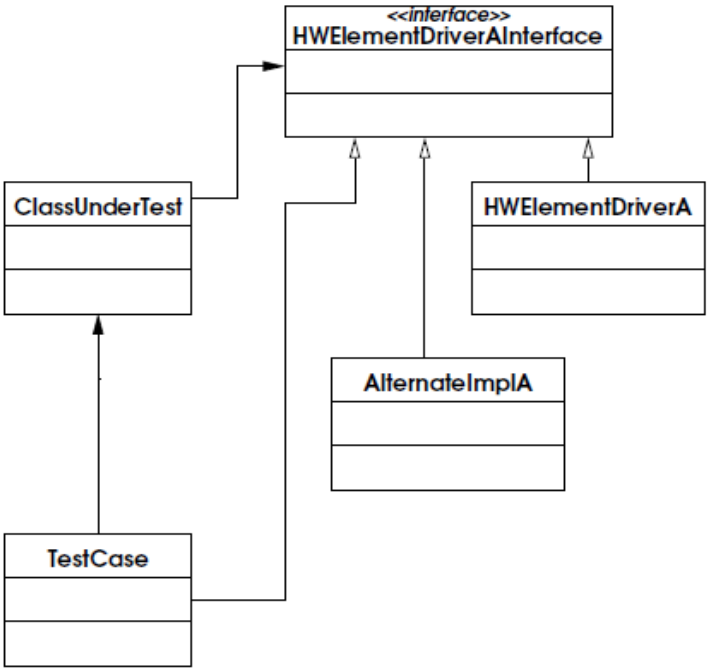
להלן, שיטת פיתוח נפוצה מאוד בתעשיית התוכנה והחומרה [Acd]. לפי שיטה זאת מפתחים מודול של תוכנה שנקרא לו מודול א' שמשתמש במודול ב', אבל לא מחכים עד שהמודול ב' יהיה מוכן אלא מפתחים את שניהם במקביל. באופן דומה, נניח שמודול א' משתמש במודולים A, B, C שהינם מודולים של חומרה. בגלל שהם מודולים של חומרה אז לרוב יצרן החומרה מספק ביחד עם כל אחד מהם תוכנה שמאפשרת להשתמש בהם, תוכנה זאת נקראת Driver. אם החומרה עדיין לא קיימת אז גם במקרה הזה מפתחים את החומרה (יחד עם ה-Driver שלה) במקביל לפיתוח מודול התוכנה א'.

עתה, נרצה לבדוק מחלקה מסוימת בתוכנה של מודול א'. נקרא למחלקה הזאת ClassUnderTest. מחלקה זאת עובדת מול המודולים של החומרה. מימוש נאיבי לבדיקת מחלקה זאת מובא באיור 46 להלן.



איור 46: תרשים מחלקות נאיבי לבדיקות של חומרה [Acd]

אבל המימוש הנאבי שהובא לעיל דורש שה-Drivers יהיו מוכנים לפני שנתחיל לבדוק את המחלקה ClassUnderTest. בכדי לפתור זאת, נשתמש בדפוס התכן Strategy ע"י הוספת מנשק בין ה-Drivers לבין התוכנה שלנו, כפי שניתן לראות באיור 47 להלן. כך נוכל, בזמן הפיתוח כאשר ה-Drivers עוד לא מוכנים, לפתח קוד שידמה עבור הבדיקה שלנו את פעולת ה-Drivers בתרחישי בדיקה שונים (נקרא לקוד זה בשם Mock Driver).



איור 47: תרשים מחלקות לבדיקות של חומרה שמשתמש בדפוס התכן Strategy [Acd]

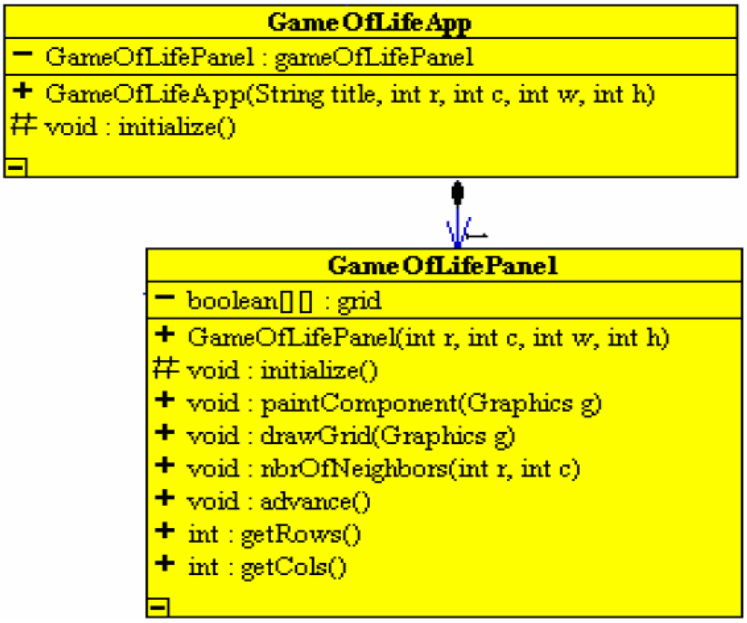
9.2 פיתוח משחק מחשב פשוט

להלן יוצג פיתוח של משחק מחשב פשוט ומוכר שנקרא Game of Life [Wick]. משחק זה פורסם בשנות השבעים ע"י מתמטקאי מפורסם בשם ג'ון קונווי (John H. Conway), שהמציא משחקים מתמטיים [Gab]. המשחק מתנהל על לוח משבצות דו ממדי. המשתמש מגדיר בתחילת המשחק את המשבצות ה"חיות" ואת מספר התורות שהמחשב יריץ את המשחק. בכל תור "נולדות" או "מתות" משבצות בהתאם לחוקים הבאים:

1. משבצת נולדת מחדש כאשר יש לה בדיוק שלוש שכנות חיות.
2. משבצת שורדת כאשר יש לה שתיים או שלוש שכנות חיות.
3. משבצת תמות אם יש לה פחות משתי שכנות חיות או יותר משלוש שכנות חיות.

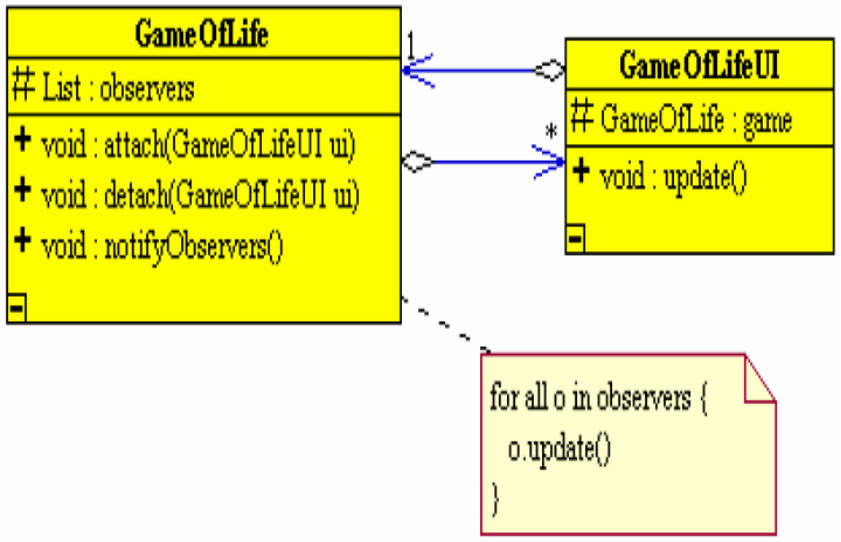
בדרך זאת מתקבלות צורות גאומטריות מפתיעות על לוח המשחק על פי החוקים שנקבעו.

בעיצוב תוכנה נאיבי של המשחק נכניס לאותה מחלקה את רוב האספקטים של המשחק [Wick]. למשל, נכניס לאותה מחלקה את המתודות לציור לוח המשחק (Panel) וגם המתודות המשמשות לחישוב חוקי המשחק (למשל, חישוב מספר השכנים של תא מסוים). כפי שניתן לראות באיור 48 להלן.



איור 48: תרשים מחלקות שמציג עיצוב נאיבי למשחק Game of Life [Wick]

נרצה להפריד בין הקוד שמציג את לוח המשחק לבין הקוד שמפעיל את חוקי המשחק. ואת זה אפשר לממש ע"י שימוש בדפוס התכן Observer. במאמר [Wick] בחרו להשתמש רק באופן חלקי בדפוס התכן Observer על מנת להציג עיצוב תוכנה פשוט ככל האפשר. כמובן, שתוך כדי כך יש ויתור על היתרונות שבשימוש מלא בדפוס התכן Observer כפי שהובאו כאשר הצגתי אותו. איור 49 להלן מציג שימוש חלקי בדפוס התכן Observer שעושה שימוש בשתי מחלקות בלבד לעומת השימוש בארבע מחלקות/מנשקים בדפוס התכן Observer.



איור 49: תרשים מחלקות שמציג שימוש חלקי בדפוס התכן Observer ל - Game of Life [Wick]

10 סקירת דפוסי-נגד (anti-patterns) מונחי עצמים נבחרים

"דפוסי-נגד מונחי עצמים" (object oriented anti patterns) הינם דפוסי תכן שמלמדים אותנו להימנע מעיצוב מונחה עצמים שגוי.

כדאי לציין שקיימות קטגוריות רבות אחרות של דפוסי-נגד, למשל דפוסי-נגד של תכנות (programming anti-patterns) שכולל בין השאר את הנושאים הידועים הבאים.

- **קוד ספגטי** - שיטת התכנות הידועה לשמצה שמתעלמת מהצורך לתכנת באופן מבני (structured programming) דבר שגורם לשימוש בהסתעפויות רבות בקוד בעזרת פקודות כמו Goto וכן שימוש במבני נתונים גלובאליים כאשר בהמשך התפתחות המערכת מתחילים לאבד את ההבנה על אופן זרימת התוכנית ועל מי ניגש ומתי לנתונים (קריאה/כתיבה). <#line>
- **מספרי קסם** - הכנסת מספרים לתוך קוד התוכנה ללא שימוש באף הסבר שמוטמע בקוד (שניתן להוסיפו למשל ע"י הגדרת const ב-C או ע"י הגדרת final ב-Java).

בעמודים הבאים אציג בקצרה מבחר של דפוסי-נגד מונחי עצמים.

Call Supper 10.1

כפי שמתואר ב-[Mart2], המתכנת של מחלקה שיורשת ממחלקת האב דורס (overrides) את המתודה במחלקת האב. עפ"י התיעוד של מחלקת האב, הוא נדרש בפקודה הראשונה שהוא כותב לקרוא למתודה שמומשה במחלקת האב (ושאותה הוא דורס). זה יוצר מצב שהמתכנת עלול לשכוח לקרוא למתודה זאת או להחליט לממש אותה בדרך אחרת במקום לקרוא לה וזה דבר שייצור חוסר אחידות במערכת בין המחלקות היורשות השונות. דוגמא של קוד מובאת להלן.

```
public class EventHandler ...
   //Override this method by first calling this overrided method.
   public void handle (BankingEvent e) {
            housekeeping(e);
   }


public class TransferEventHandler extends EventHandler...
   public void handle(BankingEvent e) {
              super.handle(e); //The developer calls the overrided method.
              initiateTransfer(e);
  }
```

ניתן לפתור בעיה זאת ע"י שימוש בדפוס התכן Template Method שמאפשר למתכנת של המחלקה היורשת רק לממש את המתודה שלו, כאשר המתודה של מחלקת האב (מתודת דפוס התכן) תיקרא לו.
אותה מתודה שהמתכנת של המחלקה היורשת יצטרך לממש נקראת hook method. להלן קטע קוד בשפת Java שמתאר זאת. לבסוף, הלקוח של כל הקוד שלהלן יקרא רק למתודה handle.

```
public class EventHandler ...
   public void handle (BankingEvent e) {
            housekeeping(e);
            doHandle(e); //The super-class calls the hook method.
   }

   protected void doHandle(BankingEvent e) {
   }


public class TransferEventHandler extends EventHandler ...
   //The sub-class developer implements the hook method.
   protected void doHandle(BankingEvent e) {
            initiateTransfer(e);
   }
```

Sequential Coupling 10.2

המתודות של המחלקה חייבות להתבצע בסדר מסויים.
למשל המחלקה מכילה את המתודות : init, do, end.
דבר זה גורם לבעיות רבות בתוכנה שמשתמשת במחלקה בגלל שמפתחי הקוד נוטים לשכוח ולהתעלם מהסדר הנדרש. זה מזכיר בעיה דומה שהזכרתי בדפוס-הנגד Call Super שהבאתי לעיל.

כאשר סקרתי סוגים של צימודים הבאתי את הצימוד הבא.

"צימוד בקרה (control coupling) – מודול אחד שולח נתוני בקרה להשפעה על פעולת מודול אחר."

גם כאן אני מזהה סוג של צימוד בקרה, אמנם לא צימוד בקרה שנמצא במפורש בקוד אבל כזה שמוכתב ע"י תיעוד הקוד.

10.3 Base Bean

שימוש מוטעה בהורשה ממחלקה אחרת במקום להשתמש בהאצלה כאשר רוצים להשתמש בפונקציונאליות מאותה מחלקה. השם "Bean" מתאר מחלקות לשימוש כללי (שנעשה בהן שימוש נרחב בשפת Java).

בכך אנו עוברים על הכלל לעיצוב תוכנה שהזכרתי בפרק של סקירת ספרות, שיש להעדיף Composition והאצלה על פני שימוש בהורשה.

10.4 God Object

הכללת יותר מידי פונקציונאליות במחלקה אחת (מחלקה צריכה להיות ממוקדת). כאן, אנו עוברים על הכלל לעיצוב תוכנה שהזכרתי בפרק של סקירת ספרות, שצריכה להיות רק סיבה אחת לשינוי של מחלקה. ואילו ב-God Object יש פוטנציאל גדול לסיבות רבות לשנות את המחלקה.

11 סיכום

כפי שראינו, מהנדס תוכנה בתעשייה עובד עם דפוסי תכן גם אם זה לא בא מיוזמתו, שהרי דפוסי התכן משולבים בשפות התוכנה המודרניות כמו שפת Java ובמסגרות-העבודה.

ראינו כיצד התפתחו דפוסי התכן במהלך השנים ואת ההתפתחות הטיבעית מדפוסי תכן בתחומים אחרים אל דפוסי תכן מונחי עצמים בתעשיית התכנה.

ראינו כיצד דפוסי תכן מונחי עצמים באים לעזור למהנדס התכנה לכתוב קוד תוכנה איכותי יותר ולתאר את הקוד לעמיתיו באופן ברור ובקצרה. בשלב הזה צריך ליהיות ברור ששימוש בדפוסי תכן מבלי להבין אותם עלול לגרום לכתיבת קוד שהוא גרוע יותר אף מקוד שלא משתמש בדפוסי תכן. ראינו שחשוב להבין לאיזה הקשר מתאים כל דפוס תכן ולהבין מה היתרונות והחסרונות שהוא מביא איתו. חשוב להבין את עקרונות הנדסת התכנה שדפוסי התכן באים לעזור לנו לממש באופן מיטבי.
נתתי רקע הסברים ודוגמאות מעשיות, גם מתעשיית התכנה, לשימוש בדפוסי תכן מונחי עצמים. בכדי לתת זוית ראייה רחבה יותר, נגעתי גם בנושאים קרובים כמו דפוסי תכן של ארכיטקטורת תוכנה ודפוסי-נגד.

תוך כדי כתיבת עבודה זאת למדתי על טעות שמאוד נפוצה בתעשיית התוכנה בהצגת ומימוש דפוס התכן "Factory Method", כפי שהסברתי בגוף העבודה. הצגתי את הנושא והרחבתי את ההסבר עליו כאשר הצגתי את דפוס התכן הזה.

אני מקוה שעבודה זאת תתרום לעוד הרבה אנשים שרוצים לפתח תוכנות איכותיות. עבודה זאת תרמה לי באופן אישי לחדד את ההבנה שלי בנושא דפוסי תכן מונחי עצמים וזה כבר משפיע על אופן פיתוח התוכנה במקום עבודתי.

תודה מיוחדת לד״ר דן אהרוני שעזר לי לתקן את התובנות שהגעתי אליהן בעבודה זאת וללטש את העבודה כך שתעמוד בסטנדרטים האקדמאים.

עיצוב / תָּכְן	Design (vs. planning)
דפוסי תָּכְן / תבניות עיצוב	Design patterns
דפוסי נגד / תבניות נגד	Anti patterns
הֶקְשֵׁר	Context
יכולת של תוכנה להתרחב בעתיד – למשל תמיכה ביותר משתמשים בו זמנית	Scalability
הרכבה	Composition
עיצוב/תָּכְן מונחה עצמים	Object Oriented Design (OOD)
הורשה	Inheritance
האצלה	Delegation
רב תצורתיות	Polymorphism
כימוס	Encapsulation
מחלקה	Class
מחלקת אב	Super class / Base class
מסגרת-עבודה	Framework
עצם / מופע של מחלקה	Object / Class instance
צימוד	Coupling
מקרה שימוש	Use Case
שחקן (במקרה שימוש)	Actor
תרשים רצף (או תרשים רצף העבודה)	Sequence diagram
תרשים מחלקות	Class diagram
מנשק (המילה ממשק השתרשה בשפה בטעות)	Interface
תרשים מצבים	State diagram
תרשים פריסה	Deployment diagram
תרשים רכיבים	Component diagram
תרשים פעילות	Activity diagram
תרשים מקרי שימוש	Use case diagram

טבלה 2: רשימת מונחים ותרגומם

להלן מובא מאמר שלי שמתאר עבודה על פי עיקרון ה-Dependency Injection בעזרת ה- Spring Framework". זהו מאמר שכתבתי במקום עבודתי במאי 2008 אבל תוך כדי כתיבת עבודה מסכמת זאת התעמקתי יותר ומצאתי דברים לא מדוייקים במאמר (למרות שהמאמר עבר בזמנו משוב של עוד ארבעה עמיתים מקצועיים ומוכשרים).

- במאמר ניתנת דוגמא לשימוש בדפוס התכן Factory Method אבל בעצם זאת דוגמא לשימוש ב-Static Method שאיננו דפוס תכן כלל. זאת טעות נפוצה מאוד בתעשייה [Fbs]. שהרי Factory Method משתמש בשתי היררכיות של הורשה בכדי לדחות את החלטת המימוש למפתחים שיורשים את המחלקה Product ו-Creator (שיוצר את ה-Product).
- מספר מסופר במאמר שקיימים שני סוגים של צימוד במערכת תוכנה אבל יש יותר, כפי שכתבתי בעבודה מסכמת זאת.

ועתה מובא המאמר כלשונו.

CH Line Development:

Improve Code Modularity using Dependency Injection (DI) via Spring

Author:	Eli Isaak, a developer in NDS Israel/CH Line
Reviewers:	Four names that were omitted here for their privacy sake
Target Audience:	OOP Developers (not only JAVA developers)
Date:	22 May, 2008

What is Spring?

Spring is a Java Framework Open Source library that was developed as a reaction to the difficulty and expense in deploying Java Enterprise Edition (JEE).

JEE is usually deployed using an Application Server which delivers common features needed in Java Enterprise applications (e.g., Persistence, Queues, Scheduler, Thread Pool, etc.). As a lightweight container, Spring enables using most of the JEE features without installing an application server or

dealing with JEE complexity. Each application remains a standalone application, and by using Spring it obtains the JEE features as needed.

For more information, see http://www.springframework.org/.

What Do We Gain By Using Spring?

One of the important advantages of using Spring is improved code modularity in order to reduce complexity of large systems. It allows for loose coupling of modules which keep dependencies to a minimum.

Coupling can appear either as data dependency (i.e., some code sections share the same data) or as flow dependency (i.e., one code section depends on another code section's execution). In this article, we will address flow dependency and how to reduce it.

Spring uses some OOP features more efficiently in order to reduce coupling. For this, Spring provides a simple feature called "Dependency Injection" (DI), which is an implementation of the "Inversion of Control" (IoC) design pattern (a note on DI vs. IoC will follow).

Read on to learn about DI (supported by Spring) and how it is used to improve our code.

Trying to Reduce Coupling

Before illustrating how we use Spring (and DI), let's try to improve our code using the Factory pattern.

At first, we will use a simple code sample that doesn't use the Factory pattern. Below is a sample application (using simple **free style** Java code) that creates and sends messages. It contains the following classes.

- A MessageProvider class that creates a message.
- A MessageSender class that sends a ready message provided to it.
- A main class to run our application.

```
Class MessageProvider {
      public String getMessage() {
            return "Hello";
      }
}

Class  MessageSender  {
      public void sendMessage() {
            //Do you see a problem here? See explanation below.
            MessageProvider messageProvider = new MessageProvider();
            sendMessageUsingTcp( messageProvider.getMessage() );
      }
}

Class Main() {
       public void main( )  {
            //Do you see a problem here? See explanation below.
            MessageSender sender = new MessageSender();
            sender.sendMessage();
      }
}
```

Now, let's improve this code using the Factory pattern.

```
Class  MessageSender  {
      public void sendMessage()  {
            //Can you see a new problem with this fixed line? See explanation below.
            MessageProviderInterface messageProvider = MyFactory.getMessageProvider();
            sendMessageUsingTcp( messageProvider.getMessage() );
      }
```

```
}


Class Main()  {
      public void main( )  {
            MessageSenderInterface sender = MyFactory.getSender();
            sender.sendMessage();
       }
   }
```

What Did We Improve?

The Factory methods (methods of the MyFactory class) decouple the implementation from our code (there is no need to initialize an object of a specific class). For example:

- The Factory methods may read a configuration file (e.g., an XML file) that will guide them in creating mock objects instead of real objects.

> **Note on Mock objects**
>
> We use mock objects in order to test the application using a simulated environment (mock objects simulate the real objects that our application depends on, e.g. they return the same dummy response for any request).

- The objects may be initialized when these Factory methods are called or the objects may be brought from a pool of ready objects.
- The objects implementation may be easily replaced – even at runtime.

However, we still need to further reduce coupling because the MessageSender class now depends on the MyFactory Class. We want to reduce coupling to the minimum. It is preferable that the MessageSender class doesn't care about wiring the MessageProvider to itself.

Further Improving Our Code Using Spring

Let me now introduce a better, yet simpler (simplicity is great), MessageSender class:

```
Class  MessageSender  {
       MessageProviderInterface messageProvider = null;

       //We let another entity to inject this dependency for this class.
       public void setMessageProvider(  MessageProviderInterface messageProvider )  {
            this.messageProvider = messageProvider;
       }

       public void sendMessage()  {
          String message = this.messageProvider.getMessage();  //Using the injected object.
          sendMessageUsingTcp( message );
       }
}
```

Now we can integrate this simple MessageSender class into the Spring framework.

As a matter of fact, Spring supplies us with a Factory which has the following advantages:

- You don't have to write the Factory code yourself – Spring provides it for free.
- As demonstrated in the above simple class, your application code (the business logic) doesn't depend on Spring API.
- You don't have to wire the dependencies into your class (in the sample code above, sender depends on messageProvider) – this is done in an XML file (as illustrated later on this article). This is called Dependency Injection (DI). DI is known a "Don't call us, we will call you" approach, i.e., another entity injects to our class the dependencies that our class relies on.

Note on Dependency Injection (DI) vs. Inversion of Control (IoC)

In many articles and presentations, the difference between "Inversion of Control" (IoC) and "Dependency Injection" (DI) is somehow obscured or doesn't exist.

In short, IoC is a more generic design principle, while DI is implemented using IoC.

"Inversion of Control" (IoC) means that the application code is called by the framework instead of the application calling the framework methods by itself.
On the other hand, "Dependency Injection" (DI) is a programming style where dependee classes are not directly created by the dependent class but rather "injected" by a framework, thus executing IoC.

Compare the following sample code, which uses Spring, and the XML section below it to the previous code samples:

```
main()
{
        BeanFactory factory =
            new XmlBeanFactory ( new FileSystemResource("applicationContext.xml") );
         MessageSenderInterface sender = (MessageSenderInterface) factory.getBean("msgSender");

        // This "wiring" is now remarked out and moves into the bellow XML file:
        //   sender.setMessageProvider(messageProvider);

        sender.sendMessage();
}
```

Below is a section from the **applicationContext.xml** file:

```
  <bean id="msgSender" class="com.nds.MessageSender">
    <property name="messageProvider" ref="messageProvider" />
  </bean>
```

```
<bean id="messageProvider" class="com.nds.MessageProviderMock"/>
```

The object messageProvider is configured to be a mock object (its class is configured to be MessageProviderMock). Note how this object is injected into the msgSender object and may be easily changed by a simple update to this XML file. The msgSender object is not aware of how the messageProvider was initialized and what its specific class is. These "wiring" decisions are left to the integrator, after the development phase is completed.

How Do We Use Spring in Our Projects?

In the CH Line, we develop Spring-based Java projects. A good example of such a project is the successful S4 project which was entirely written in Java.

Each developer develops their part of the project and writes POJO (Plain Old Java Object) classes which are unaware of the Spring framework (we write simple classes, much like the last MessageSender class sample above). Each component is delivered to the development integrator as two Java Archive (JAR) files – one JAR contains the implementation classes and the second JAR contains the interfaces to the implementation classes.

Using Spring, the development integrator wires all the parts of the project to create a final product. This is done by creating the above-mentioned applicationContext.xml file to "wire" the released classes.

At run-time, when the application is initialized, Spring injects the dependencies to each class by calling the setter methods of the class (Spring does this by reading the XML file and using the Java programming language feature called Reflection).

Summary

The advantages of using Spring in our development process are as follows:

- **Gain of efficiency in development time** as we minimize the dependency between the different parts of the product\system during development.
- **Developers only get the interfaces to the other components of the project**, and therefore changes to the implementation of one component won't affect the other components.
- **We clearly separate the business logic** (delivered in the JARs) from the technical (infrastructure) layer, added later by the integrator.

- **The integrator can add several common services** (such as common system configuration and persistency) across the project by injecting them to each delivered JAR using DI.
- **Possibility to simply replace any implementation or infrastructure** in the future without an overall recompilation.
- **Possibility to add more services to the application**, such as changing the application to be a distributed application. This is transparent to the developers and to the business logic implementation.

Note that just like any other tool, Spring can be abused to add complexity to the application instead of reducing it. Dependency injections can be easily abused and cause code to be hidden in large XML files which will need to be debugged. Remember that you must maintain the code together with the XML files in order to get the whole picture.

Bibliography

http://martinfowler.com/articles/injection.html

[Abcm] Astrachan O., Berry G., Cox L., & Mitchener G. (1998), Design Patterns: An Essential Component of CS Curricula, Proceedings of the 29th SIGCSE technical symposium on Computer science education, 30: 1, 153-160.

[Acd] Alphonce C., Caspersen M., & Decker A. (2007), Killer 'Killer Examples' for Design Patterns, Proceedings of the 38th SIGCSE technical symposium on Computer science education, 228-232.

[Ais] Alexander Christopher, S. Ishikawa and Murray Silverstein (1977). A Pattern Language: Towns, Buildings, Construction. Oxford University Press.

[Alex] Alexander Christopher (1979). A Timeless Way of Building, Oxford University Press.

[Alle] Allen H. (2011), Activity Diagram – Exception Handling.
From: http://www.holub.com/goodies/uml. Retrievd: 9/2011.

[Baat] Baat L. (2007), Teaching the Linked List, Open University Seminar Work.

[Bean] NetBeans community, MVC Pattern.
From: http://netbeans.org/kb/docs/javaee/ecommerce/design.html. Retrieved: 10/2011.

[Berg] Bergin, J. (2002), Fourteen Pedagogical patterns for teaching computer science.
From: http://csis.pace.edu/~bergin/PedPat1.3.htm. Retrieved: 11/2011

[Bert] Bertrand M. (1998), Object Oriented Software Construction, Prentice Hall.

[Comm] Wikimedia commons. Composite & Chain of Responsibility patterns.
From: http://commons.wikimedia.org Retrieved: 9/2011

[Czbd] Cheng Z., Budgen D. (2012), What Do We Know about the Effectiveness of Software Design Patterns, IEEE Transactions on Software Engineering. 38: 5, 1213-1231.[Dev] GPWiki, Observer Pattern.
From: wiki.gamedev.net Retrieved: 7/2011.

[Devx1] Developer.com, Use Case Diagram Example.
From: http://www.devx.com/architect/Article/45984/1954. Retrieved: 8/2011.

[Devx2] Developer.com. Spring Uses Singleton.
From: http://www.devx.com/java/Article/21665/0/page/2 Retrieved: 11/2011.

[Dgd] Denzler C., Gruntz D. (2008), Design Patterns: Between Programming and Software Design, roceedings of the 30th international conference on Software engineering, 801-804.

[Dijk] Dijksterhuis M. (1/1/2009), C# Uses Strategy Pattern.
From: http://www.dijksterhuis.org/sorting-generic-lists. Retrieved: 11/2011.

[Dof1] Data & Object Factory, Bridge Pattern.
From: http://www.dofactory.com/Patterns/PatternBridge.aspx. Retrieved: 10/2011.

[Dof2] Data & Object Factory, Iterator Pattern.
From: http://www.dofactory.com/Patterns/PatternIterator.aspx. Retrieved: 10/2011.

[Dof3] Data & Object Factory, Factory Method Pattern.
From: http://www.dofactory.com/Patterns/PatternIterator.aspx. Retrieved: 10/2011.

[Dof4] Data & Object Factory, Adapter Pattern.
From: http://www.dofactory.com/Patterns/PatternAdapter.aspx. Retrieved: 9/2011.

[Eli] Eli I. (2008). CH Line Development: Improve Code Modularity using Dependency Injection (DI) via Spring. From: NDS intranet. Retrieved: 11/2011.

[Fbs] Freeman E., Bates B., Sierra K. (2004), Head First Design Patterns, O' Reilly & Associates .

[Forg] Sourceforge.net, JUnit TestCase class.
From: http://junit.sourceforge.net/doc/cookstour/cookstour.htm Retrieved: 11/2011.

[Gab] Gaby C. The Game of Life.
From: http://www.snunit.k12.il/heb_journals/kimat2000/1308.html. Retrieved: 3/2012.

[Gina] Ginat D. (2003), The greedy trap and learning from mistakes, Proceedings of the 34th SIGCSE technical symposium on Computer science education, 11-15.

[Gof] Gof (Gang Of Four): Gamma E., Helm R., Johnson R., Vlissides J. (1994), Design Patterns: Elements of Reusable Object-Oriented Software, Addison-Wesley.

[Hagg] Haggar P. (1/5/2002), Singleton Pattern.
From: http://www.ibm.com/developerworks/java/library/j-dcl/index.html.
Retrieved: 10/2011.

[Huy] Huy T. (5/2011). Proxy Pattern.
From: http://tndhuy.wordpress.com/page/2. Retrieved: 11/2011.

[Info] InformIT. Sequence Diagram.
From: http://www.informit.com/articles. Retrieved: 8/2011.

[Jam] James S. (2010), Design Patterns Uncovered: The Factory Method Pattern.
From: http://java.dzone.com/articles/design-patterns-factory. Retrieved: 3/2012.

[John] Johnson R. (2005), Spring Uses Factory Method Pattern.
From: http://www.theserverside.com/news/1364527/Introduction-to-the-Spring-Framework. Retieved: 8/2011.

[Jose] Joseph S. (1999), Sams Teach Yourself UML, Sams Publishing .

[Knob] Knobel, E. (2004), Coupling.
From: http://study.eitan.ac.il/sites/index.php?portlet_id=110513&page_id=27.
Retrieved: 9/2011.

[Koch] Kochhar V. (2004), Singleton Pattern.
From: http://www.oaklib.org/docs/oak/singleton.html Retrieved: 10/2011.

[Lisk] Barbara H. Liskov, Jeannette M. Wing (1999),"Behavioral Subtyping Using Invariants and Constraints".
From: http://reports-archive.adm.cs.cmu.edu/anon/1999/CMU-CS-99-156.ps.
Retrieved: 10/2011.

[Mak1] SourceMaking, Strategy Pattern.
From: http://sourcemaking.com/design_patterns/strategy. Retrieved: 9/2011.

[Mak2] SourceMaking, Strategy Pattern.
From: http://sourcemaking.com/design_patterns/singleton. Retrieved: 10/2011.

[Mak3] SourceMaking, Singleton pattern.
http://sourcemaking.com/design_patterns/singleton. Retrieved: 11/2011.

[Mart1] Fowler M. (2004), Dependency Injection Pattern.
From: http://martinfowler.com/articles/injection.html. Retrieved: 8/2011.

[Mart2] Fowler M. (2005), Call Super Anti Pattern.
From: http://www.martinfowler.com/bliki/CallSuper.html Retrived: 11/2005.

[Mgj] Marco A.G., Guillermo J.D., Javier A. (2009), Teaching Design Patterns Using a Family of Games, ITiCSE '09, 41: 3, 268-272.

[Mha] Muller, O., Haberman, B., Averbuch, H. (2004), (An almost) pedagogical pattern for pattern-based problem solving instruction, Proceedings of ITiCSE'04, 36: 3, 102-106.

[Mill] Miller R. (2010), Deployment Diagram Example.
From: http://edn.embarcadero.com/article/31863 Retrieved: 8/2011.

[Mull] Muller, O. (2005), Pattern oriented instruction and the enhancement of analogical reasoning , Proceedings of the first international workshop on Computing education research, 57-67.

[Njga] Neelam S., Jason O. H., Guoqiang S., Adem Delibas. (2008), Patterns: from system design to software testing, Innovations System Software Eng., 4: 71–85

[Part] Partha K. (2004), Software Architecture Design Patterns in Java, CRC Press LLC.

[Pete] Peter L. (2007), Relations Between Classes (9/2011).
From: http://www.peter-lo.com/Teaching/M8748/
Relationships Between Classes.pdf. Retrieved: 9/2011.

[Phd] Paul Deitel, Harvey Deitel (2002), Java How to Program – Introducing OOD With The UML and Design Patterns - Fourth Edition, Prentice Hall.

[Pilo] Pilone D. (2005), UML 2.0 in a Nutshell, O'REILLY

[Ppp] Pecinovsk R., Pavlkov J., Pavlek L. (2006), Let's Modify the Objects-First Approach into Design-Patterns-First, ACM SIGCSE Bulletin. 38: 3, 188-192.

[Prac1] Hirondelle Systems. Java Uses Iterator.
From: http://www.javapractices.com/topic/TopicAction.do?Id=88. Retrieved: 11/2011.

[Prac2] Hirondelle Systems. Java Uses Iterator.
From: http://www.javapractices.com/topic/TopicAction.do?Id=125. Retrieved: 11/2011.

[Puff] Tecademy. Use Case Diagram Example.
From: http://www.puffinonline.com/Tecademy/umlqrg.pdf. Retrieved: 8/2011.

[Ravi] Ravish S. (2010), Strategy Pattern.
From: http://ravish9507.blogspot.com/2010/05/design-pattern.html. Retrieved: 9/2011.

[Robe] Robert C. M. (2004) Strategy Pattern.
From: http://today.java.net/pub/a/today/2004/10/29/patterns.html. Retrieved: 9/2011

[Sagg] Saggu J. (2011), Class Diagram Example.
From: http://www.techpost.info/2011/01/uml-quick-reference-guide.html
Retrieved: 8/2011.

[Sbhs] Shikha B., Harshpreet S. (2012), ANALYZING AND IMPROVING WEB APPLICATION QUALITY USING DESIGN PATTERNS, International Journal of Computers & Technology. 2: 2, 112-116

[SID] Shlezinger G., Iris R. B., Dov D. (2010), Modeling design patterns for semi-automatic reuse in system design., Journal of Database Management 21.1: 29+.

[Sks] Sunil K.S. (2004), Observer Pattern (8/2011).
From: http://codeproject.com. Retrieved: 10/2011.

[Spri] SpringSource. Spring Framework.
From: http://www.springsource.org. Retrieved: 8/2011.

[Tome] Dr. Tomer A. (2011), Software Engineering.
From: http://webcourse.cs.technion.ac.il/236321/Spring2011/ho/WCFiles/SWE-2011-02-development-process.pdf. Retrieved: 11/2011.

[Tony] Tony S. (2001). Factory Method.
From: http://www.javaworld.com/javaworld/javaqa/2001-05/02-qa-0511-factory.html.
Retrieved: 3/2012.

[Vetr] Vetro S. (2006), Observer Pattern.
From: http://bp.blogspot.com. Retrieved: 8/2011.

[Weis] Weiss S. (2005), Teaching Design Patterns By Stealth, Proceedings of the 36th SIGCSE technical symposium on Computer science education, 492-444.

[Wick] Wick M. R. (2005), Teaching Design Patterns in CS1: a Closed Laboratory Sequence based on the Game of Life, Proceedings of the 36th SIGCSE technical symposium on computer science education, 487-491.

[Zer] Zhu G.K., Edward S., Rogers Sr. (2012), Applying Software Design Patterns in Electromagnetic Field Simulators, Antennas and Propagation Magazine, IEEE. 54: 2, 174-179

Abstract

A design pattern can be none-formally described as a general pattern which is used to solve a frequent problem in software design. The same design pattern can be used to design different software codes in many applications.

This work focuses on Object Oriented Design Patterns.
Object Oriented Design patterns are frequently used in the software industry. They are embedded in software frameworks and in modern software languages such as Java.

This work reviews why and how design patterns became important in the software industry. We will see how we should learn and correctly work with design patterns and how design patterns help us using software engineering principles such as:
Single Responsibility Principle (SRP) ,Open Close Principle (OCP), Inversion of Control (IoC) ,Liskov Substitustion Principle (LSP), using composition and deligation instead of inheritance, design to an interface and not to an implementation.

This work also shortly review close topics such as:

- Object Oriented Anti Patterns which are patterns that teach us wrong ways to design our Object Oriented code.
- Architectural Software Design Patterns such as the Model-View-Controller (MVC) which are used by software architects to design complex software systems.

The Open University of Israel

Department of Mathematics and Computer Science

Design Patterns in Software Engineering: From Theory to Production

Final Paper submitted as partial fulfillment of the requirements

towards an M.Sc. degree in Computer Science

The Open University of Israel

Computer Science Division

By

Eli Isaak

ID: 029475175

Email: eisaak123@gmail.com

Prepared under the supervision of Dr. Dan Aharoni

Approved proposal prepared under the supervision of Dr. Tzipi Erlich

January 2013