הרצאה 11 - Structural Design Patterns

יום שני 28 דצמבר 2020 21:53

12.1 Intro to Design Patterns

בהרצאות האחרונות עסקנו בניתוח מוכוון עצמים, לאחר מכן עיצוב מונחה עצמים בקרוב נגיע לשלב התכנות המונחה עצמים ממש. רגע לפני, נרצה לעסוק עדיין בעולם העיצוב, בנושא שנקרא תבניות עיצוב (design pattern). למה שנרצה להשתמש בתבניות עיצוב?

- תבניות עיצוב הן ברות שימוש חוזר עבור בעיות דומות
- חוסכות זמן ובעיות מימוש
- ממש את כל העקרונות שלמדנו בהקשר של עיצוב מונחה עצמים (GRASP וכד')
- עוזרות ל -
  - להפריד חלקים בקוד שיכולים להשתנות לכאלו שלא ישתנו
  - תחזוק של הקוד
  - השגת עיצוב שימושי בצורה מהירה

ניתן לחלק את תבניות העיצוב ל3 משפחות עיקריות -

- **Creational patterns-** עוסק במנגנון יצירה של אובייקטים
- **Structural patterns-** עוסק ביחסים בין מחלקות וישיות בקוד (לדוגמא -הכלה \ הורשה וכד')
- **Behavioral patterns-** עוסק בהתקשרות בין עצמים בזמן ריצה
- **Concurrency patterns-** עוסק בתוכניות שרצות במקביל (multi-threading)

12.2 Structural Patterns: Adapter

התבנית הראשונה אותה נלמד נקראת Adapter (הסתגלות). נשתמש בתבנית זו כאשר נצטרך לעשות התאמה בין אובייקט מסוים לאובייקט אחר. נתבונן בדוגמא על מנת להבין את הבעיה שעליה עונה תבנית עיצוב זו. בדוגמא יש לנו ממשק של Sorter, מחלקה מופשטת common Sorter שמממשת את הממשק ומחלקה bubble Sorter שיורשת את המחלקה המופשטת.

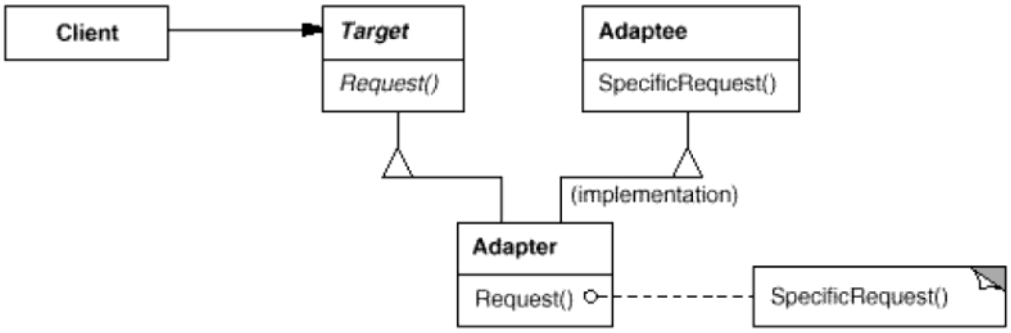
נניח ויש לנו ממשק נוסף, Task, ונרצה שהמחלקה bubble Sorter תהיה גם מסוג ממשק זה (כדי שנוכל להשתמש במתודות של הממשק וכד').

מהי הדרך הנכונה לעשות את זה?

פתרון אפשרי ופשוט יחסית הוא לדאוג לכך ש- bubble Sorter תממש גם את הממשק של Task. פתרון זה לא יעיל מכמה סיבות-

- תפקיד המחלקה bubble Sorter היא למיין לא לבצע משימות. אם היא תממש את הממשק task נצטרך לממש את כל המתודות שלה למרות שהן לא בהכרח קשורות לתפקיד המחלקה. בעיה זו עלולה ליצור בעיה אף יותר גדולה כאשר נרצה להעביר את המחלקה bubble Sorter לפרויקט אחר, במצב כזה נצטרך להעביר גם את הממשק של Task ואת המימוש שלו, למרות שאין לכך תועלת בפרויקט החדש.
- יכול להיות שבעתיד נרצה שהמחלקה bubble Sorter תהיה מסוג אובייקט אחר וחדש, ואם כל פעם שנרצה זאת, נוסיף ממשק שנצטרך לממש נקבל שהמחלקה שלנו תממש הרבה ממשקים שלא בטוח שרלוונטיים במלואם.

פתרון יעיל יותר הוא ליצור מחלקה חדשה בשם class adapter, שתירש את bubble Sorter ותממש את task. כך, נשמר עיקרון ה-open\close, פתוח להרחבה ע"י ירושה אך סגור לשינויים. נצרף תרשים שממחיש את התבנית הזו -

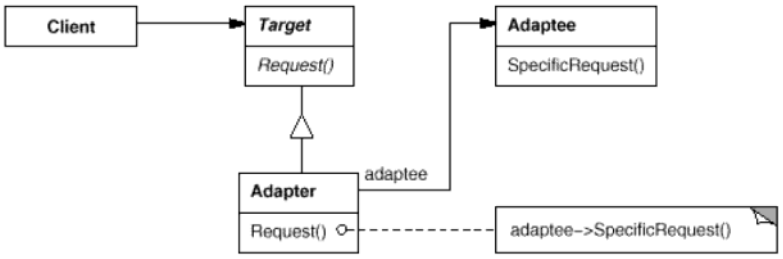


כמו שניתן לראות בתרשים מחלקת ה-adapter (bubble Sorter Task) יורשת את מחלקת ה-target (Task) וממש את מחלקת ה-adaptee (bubble sorter) ובכך מונעת מאיתנו את הצורך לממש המון ממשקים שלא בהכרח רלוונטים למחלקת ה-adaptee.

הבעיה בפתרון הזה הוא שלכל מחלקה אחרת שממשת את הממשק sorter, לדוגמא quick Sortet, נצטרך ליצור מחלקת adapter חדשה.

המחלקות האלה יהיו דומות מאד ובפועל נקבל הרבה קוד כפול. בעיה זו נובעת מכך שהירושה היא ברמה נמוכה יותר וכבל שהפתרון נמוך יותר ככה הוא פרטי יותר. נרצה להעביר את התלות (ירושה) למעלה במעלה ההיררכיה.

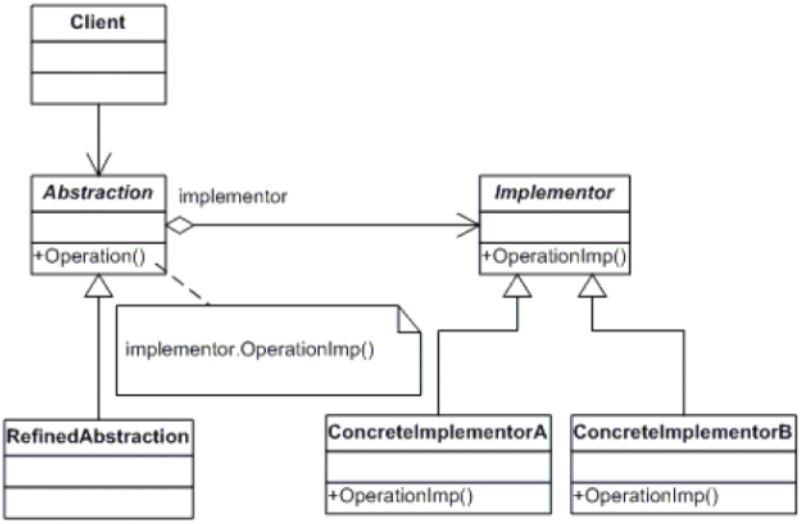
במקום להשתמש ב-class adapter נשתמש ב- object adapter - ניצור מחלקה שהיא תהיה הadapter, תכיל אובייקט מסוג Sorter ולא תירש מחלקה ספציפית. כך נוכל להזריק לsorter Task איזה מיון שרק נרצה ולא נצטרך לשכפל את הקוד בצורה חוזרת. נצרף תרשים שממחיש את התבנית הזו -



כעת מחלקת ה-adapter מכילה את מחלקת ה-adaptee. כלומר נוכל לקבל כל סוג של adaptee מבלי לדעת באופן ספציפי איזה סוג אובייקט ספציפי הוא.
אם כך, מה היתרון לclass adapter? מה היתרון של ירושה על הכלה?
בהכלה, יש לנו גישה רק שדות ומתודות שמוגדרות כ- public. בירושה יש לנו גישה גם לשדות ומתודות שהן protect. לכן, אם נצטרך שיהיה לנו גישה לדברים שהם protected נהיה חייבים להשתמש במחלקה יורשת (class adapter). אם נשמור על עקרונות התכונות object adapter יספיק לנו וזה יהיה הפתרון בו נשתמש.

12.3 Structural patterns: Bridge

נתחיל בעזרת דוגמא. אם יש לנו מחלקות שונות Straw House, Wood House, Brick House, הממשות את המחלקה המופשטת House. כעת, נרצה להוסיף גם מחלקה מופשטת של big House הממשת את המחלקה המופשטת House. נצטרך שוב לממש את המחלקה המופשטת big House בכל המחלקות השונות. כלומר אנו מכפילים את כמות המחלקות והקוד שלנו. אנו כמובן נרצה להימנע מכך ונרצה לממש זאת בצורה יעילה בעזרת תבנית העיצוב bridge. בתור התחלה בתבנית זו נרצה להפריד בין הפשטה (abstraction) למימוש (implemnetations). תבנית זו בנויה באופן הבאה -

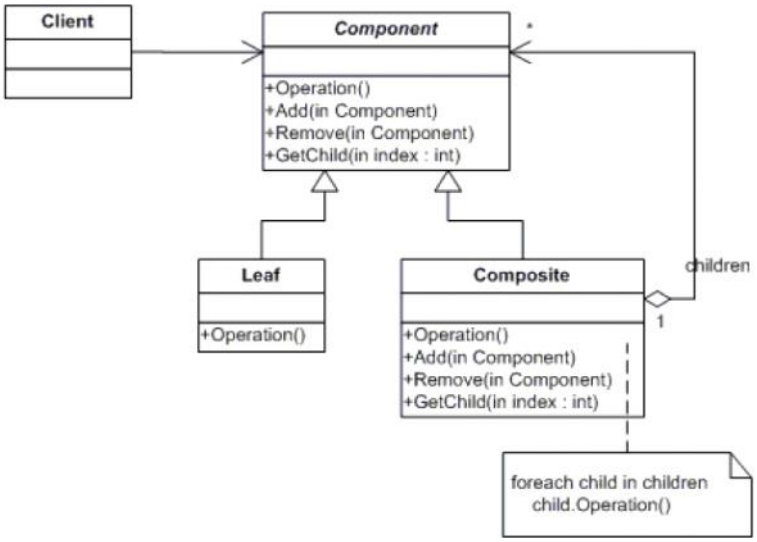


נשים לב שהתלות מתרחשת במיקום גבוה בהיררכיה על מנת למנוע את שכפול הקוד. המחלקה המופשטת Abstraction (house) מכילה אובייקט מסוג המחלקה Implementor (סוג הבית - StrawImp, WoodImp וכד'). את המחלקה המופשטת Implemntor ניתן לממש במחלקות שונות לדוגמא Concrete Implementor A וכך כשנזריק את האובייקט מסוג Concrete Implementor A למחלקה Abstraction נקבל את התוצאות לפי המימוש הספציפי של Implementor. באופן דומה גם המימוש של המחלקה Abstraction יכול להתחלק לכמה מימושים שונים, בדוגמא שלנו Small House, Big House וכד'. כאשר כל אובייקט כזה מכיל Implementor שניתן לממש אותו באופנים שונים כפי שהסברנו לעיל.

בעזרת תבנית זו, נוכל להזריק עצמים שונים למחלקות השונות גם בזמן הריצה. נעודד אצת כתיבת הקוד בצורה של שכבות, כך שהתלות תהיה רק בשכבה המופשטת (במעלה ההיררכיה) וכך נשמור באופן טבעי על עקרונות התכנות. נוכל להוסיף הפשטות או מימושים ללא תלות אחד בשני.

12.4 Structural patterns: Composite

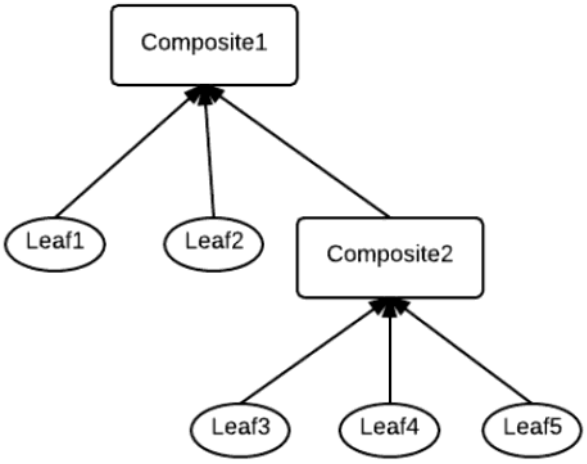
תבנית ה-Composite בשמה מלמדת על הרכבה, אובייקט המורכב ממספר חלקים. נתבונן בתרשים של התבנית -



הלקוח מכיר Component מסוים שיכול להכיל בתוכו שתי סוגי של אובייקטים -

- Composite - בעל operation וכן יכול להכיל תחתיו עוד אובייקט מסוג composite.
- Leaf - בעל operation בלבד.

נשים לב שהגדרה כזו, יוצרת מעין עץ בעזרת שימוש בהגדרה רקורסיבית. הלקוח יכיר את השורש של העץ ויהיה לו גישה לכל מחלקה שהוא ירצה. מעבר על עץ כזה יעשה ע"י DFS, מכיוון שנרצה להמשיך ולרדת אל עומק העץ עד שנגיע לאיבר שהוא עלה. המחשה לעצים שניתן ליצור -

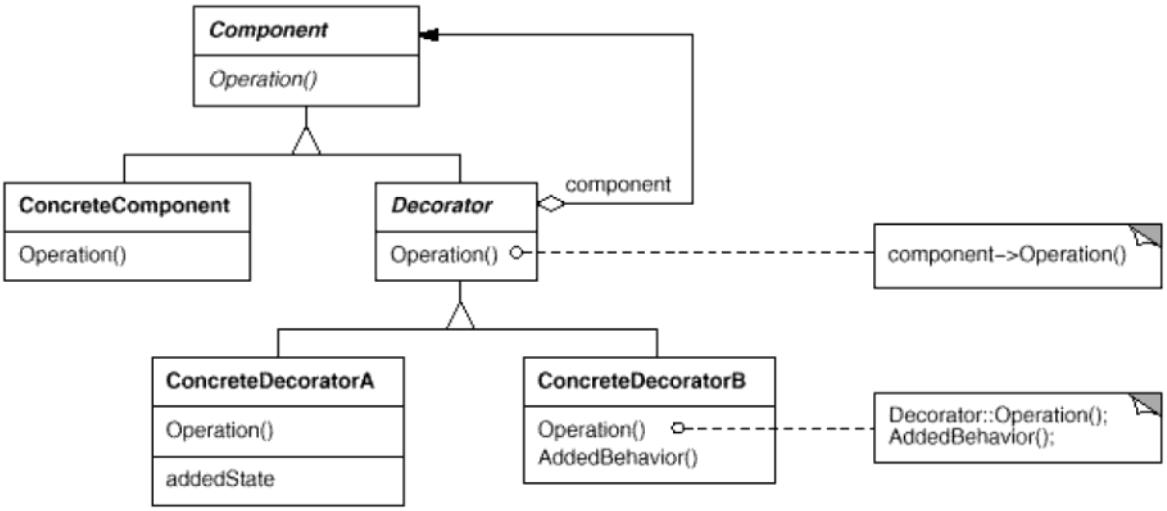


הבעיה העיקרית בתבנית כזאת היא שהיא מאפשרת מעגלים שכן אין מניעה שה-composite יכיל את עצמו כאיבר ה-composite של עצמו. בעיה זו תיצור לולאות אין סופית ורקורסיה ללא עצירה. על מנת למנוע בעיות כאלו נצטרך לעשות בדיקות בזמן ריצה ולראות שלא יצרנו בטעות מעגלים.

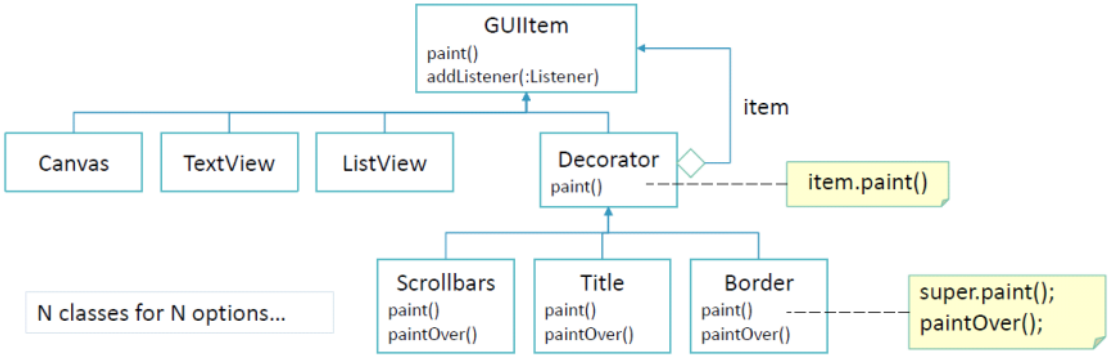
12.5 Structural patterns: Decorator

נניח נרצה לכתוב רכיב GUI - Graphic User Interface שמציג טקסט. יש מספר אפשרויות למימוש שלו - עם פס גלילה, עם פס גלילה ומסגרת, או בלי מסגרת, עם כותרת או בלי. אם נממש כל קומבינציה אפשרית של שילובים נצטרך 2^n מחלקות. נרצה להחליט את המימוש בזמן הריצה מבלי ליצור מחלקה לכל קומבינציה אפשרית.

לשם כך נשתמש בתבנית העיצוב Decorator : ניצור מחלקה decorator שיורש את component ומכילה component. את ה operationשל מחלקת ה-decorator נממש לפי הרכיב (component) שהוא מכיל. כעת, ניצור עבור כל מאפיין שנרצה לממש מחלקה שיורשת את decorator. כעת נוכל "לעטוף" כל אובייקט בעזרת המחלקה decorator, כל אובייקט שיורש את decorator יכול בעצמו להיעטף ע"י אובייקט אחר הממש את decorator ובצורה כזו ניתן לשרשר הרבה מאפיינים מבלי לייצר מחלקות לכל האפשרויות השונות. נצרף את התרשים לתבנית זו -



כעת נתבונן בדוגמא על מנת להבין יותר טוב את התבנית. נחזור אל רכיב ה-GUI שהזכרנו לעיל -



נניח ואנו רוצים ליצור קנבס עם פס גלילה וכותרת - אזי הקנבס הוא concrete Component ופס הגלילה והכותרת הן Concrete Decorator. בקוד זה יראה כך -

GUI Item g2 = New Scrollbar (new Title (New Canvas()))

נשים לב שפס הגלילה עוטף את הכותרת שעוטפת את הקנבס. מכיוון שהקנבס הוא concrete component הוא לא עוטף אף אובייקט אחר. נניח וכעת אנו רוצים לקרוא למתודה paint של g2 -

g2.paint();

הקריאות יתבצעו בצורה רקורסיבית, ראשית נקרא ל-paint ב-scrollbar. מכיוון שמתודת ה-paint שלו מוגדר super.paint, נלך את האובייקט שהוא עוטף, title ונבצע בו את מתודת ה-paint. גם בו נצטרך ללכת לאובייקט שהוא עוטף (בעקבות קריאת ה-super.paint במתודה שלו), אל ה-canvas. באובייקט ה-canvas המתודה paint היא ללא super.paint לכן נפעיל את מתודה ה-paint של ה-canvas.

כעת נתחיל להפעיל את המתודות בצורה רקורסיבית (באופן דומה להוצאה ממחסנית), כלומר את מתודת ה-paint של title ולאחר מכן את scrollbar.

בצורה כזו ניתן לעטוף את כל האפשרויות אחד בתוך השני תוך כדי שאנו שומרים על n מחלקות ולא 2^n .

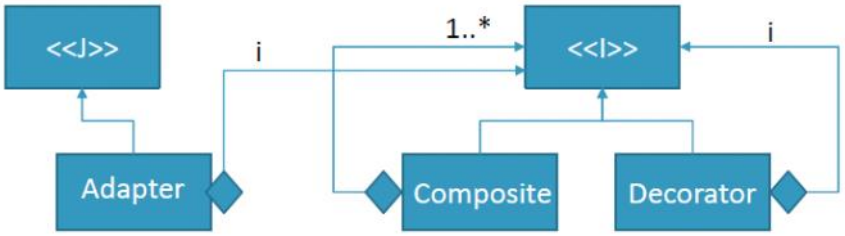
ניתן לראות שיש כאן שמירה על העקרונות של solid. כמו כן, מחלקת הdecorator חשובה שכן היא מאפשרת לממש פעם אחת את המתודות המשותפות לכל המחלקות שיורשות אותו ולהשאיר מתודות מופשטות עבור המימושים הספציפיים של המחלקות היורשות. אם אחת המחלקות היורשת תרצה לממש את אחת המתודות שממשה מחלקת ה-decorator היא תוכל לדרוס אותה. שאר המחלקות, שתיספקו במימוש של decorator, לא יצטרכו לממש את המתודות בעצמן.

החסרונות של תבנית זו הן -

- אם יש חשיבות לסדר ההרכבה של העטיפה קשה לאכוף את הסדר הזה. קשה למנוע יצירה של אובייקטים בסדר לא נכון (אם לדוגמא צריך לצייר קודם פס גלילה ואח"כ כותרת אחרת פסי הגלילה יסתירו את הכותרת).
- הרבה מחלקות קטנות לכל תוספת, קשה לתחזק את הקוד רק למי שממש מכיר אותו.

12.6 הבדלים בין תבניות העיצוב השונות עד כה

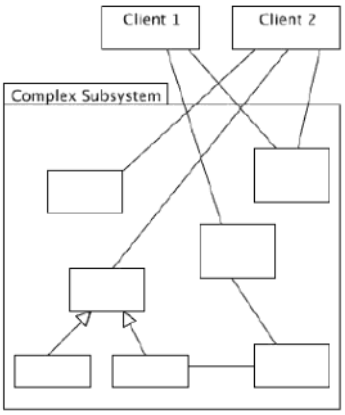
עד כה ראינו ארבע תבניות עיצוב ואנו נרצה להשוות בין שלוש שהן יחסית דומות - Adapter, Composite, Decorator. נתחיל בתרשים שימחיש את ההבדל ולאחר מכן נסביר אותו -



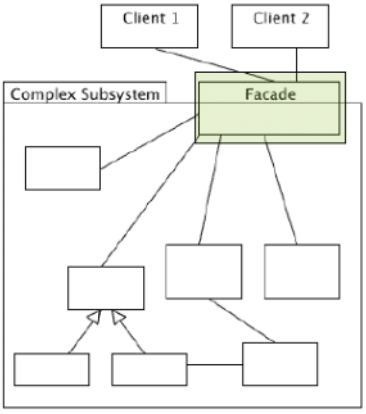
מחלקת ה-Adaptor מבצעת התאמה בין אובייקט מסוג I אותו היא מכילה לבין אובייקט מסוג J אותו היא ממשת. כלומר היא תקבל אובייקט מסוג I ותשתמש בו באובייקט מסוג J.
מחלקת ה- Decorator מכילה בתוכה בדיוק אובייקט אחד מסוג I. לעומת זאת מחלקת ה-Composite מכיל בתוכה לפחות אובייקט אחד מסוג I וייתכן שתכיל הרבה אובייקטים מסוג I שכל אחד מהם בתורו יכול להיות עלה או עוד composite.

12.7 Structural patterns: Facade

תבנית הבאה שנעסוק בה נקראת Facade או בעברית חזית. כמו שלבניין יש חזית שיכולה להיות מרשימה או מוזנחת כך גם בקוד.
נתבונן בתרשים הבא -



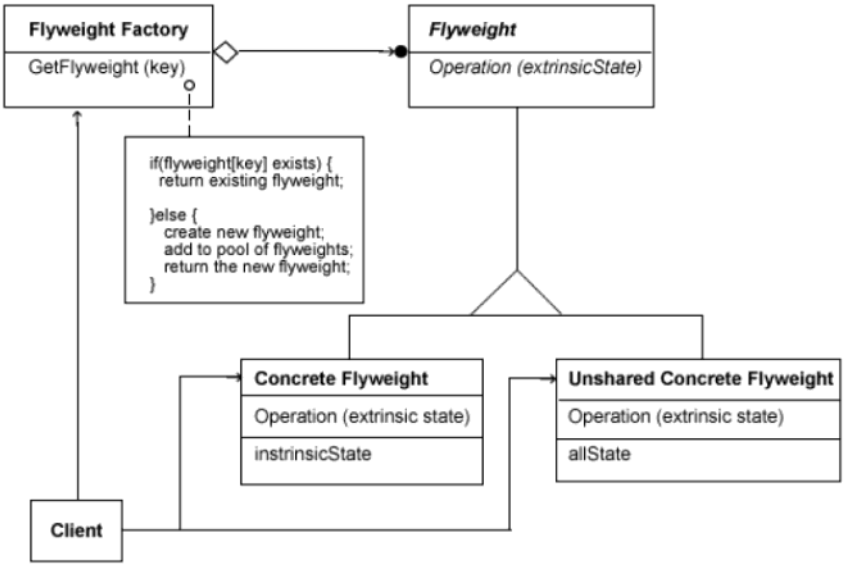
מה אנו רואים? קיימין שתי לקוחות המשתמשים במערכת, כל לקוח יוצר קשרים אל מול המחלקות/מתודות שרלוונטיות אליו. אנו רוצים להימנע מקוד שנראה מבולגן כזה ולאפשר ללקוחות לתקשר מול מחלקה אחת והיא תבצע עבורם את הכל. מחלקה זו תממש את תבנית העיצוב Facade וכך זה יראה בעזרתה -



בעזרת תבנית זו נוכל להפחית את הצימוד (coupling) ונשמור על עיקרון הכימוס, הלקוחות ישתמשו בקוד אך ורק דרך הfacade ולא יכירו את המימוש של הקוד עצמו. ניצור ממשק (ה-facade) שיודע להשתמש בקוד כדי לבצע פעולות מסוימות, כך הלקוח יכיר רק הממשק. בקוד עצמו ניצור מחלקה שתממש את הממשק ולמחלקה זו תהיה גישה אל המחלקות והמתודות השונות של הקוד. במידה ונרצה לבצע שינוי או עדכון לקוד נצטרך לעדכן רק את המחלקה שמממשת את הfacade (ולא את כל הלקוחות).

12.8 Structural patterns: Flyweight

תבנית Flyweight ובתרגום חופשי משקל נוצה הגיעה במטרה לצמצם את מספר האובייקטים השונים שאנו מחזיקים בזמן נתון. נסתכל על התרשים של תבנית זו -

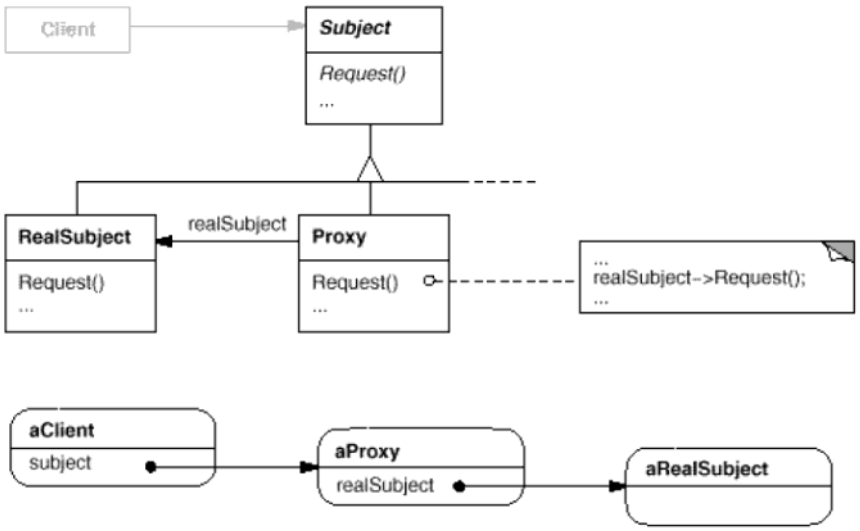


לשם הבנה נסתכל לדוגמא על משחק. במשחק קיימים מחלקות שונות לדוגמא מחלקה של טנקים, חיילים, מטוסים ועוד. לכל מחלקה יש אובייקטים רבים מאותו הסוג, שכן יכול להיות שיש לנו כרגע על הלוח 10 חיילים, 5 טנקים, ו-7 מטוסים. אנו נרצה לצמצם את מספר האובייקטים על מנת לחסוך בזיכרון ומקום. נשים לב שאם לכל דמות יש שדה בו שמור המיקום שלה על הלוח (עמודה, שורה), נצטרך ליצור מופע חדש עבור כל Character במשחק, שכן לכל אחד מהם מיקום שונה. מכיוון שישנן דמיות שהמצב שלהם (state) הוא זהה לחלוטין וההבדל היחיד בין הוא המיקום שלהם, זהו בזבוז ליצור הרבה אובייקטים שונים לאותה מחלקה. כיצד נחסוך ביצירת האובייקטים?
נשנה את הגדרת הממשק שממשים המחלקות של הדמויות השונות. במקום שהמיקום של הדמות תהיה בתור שדה, נשנה את המתודה שמציירת את הדמות על הלוח שתקבל פרמטרים של מיקום הדמות על הלוח. בצורה כזו נוכל לצייר ע"י אובייקט אחד מספר דמויות על הלוח (בכל פעם נקרא למתודה שמציירת את השחקנים עם פרמטרים אחרים). ניצור מערך של אובייקטים וכל אובייקט על הלוח יצביע על האובייקט במערך. כך אובייקטים החולקים מצב זהה (למעט המיקום) יצביעו על אותו אובייקט במערך. מה נעשה כאשר מצב אחד הדמויות על הלוח משתנה וכעת הוא אינו זהה לשאר האובייקטים?

נתחיל במה לא נרצה לעשות, לא נרצה לשנות את האובייקט אליו מצביעה הדמות במערך שכן שינוי זה יגרור שינוי אצל כל הדמויות המתאימות בלוח. במקום זאת, ניצור במערך אובייקט חדש של חייל פצוע והדמות שנפצעה תצביע עליו. במידה ועוד חיילים יפצעו יעברו גם הם להצביע על האובייקט "חייל פצוע" במקום להצביע על "חייל".
דוגמא נוספת בהקשר של תבנית זו יכולה להופיע כאשר אנו כותבים מסמך. בסופו של דבר רוב האותיות במסמך מסתכמות בכ-22 אותיות (בעברית) ורוב האותיות חולקות מצב זהה (אותו פנוט, אותו גודל וצבע וכד'). לכן במקום ליצור אובייקט עבור כל אות, ניצור מערך של האותיות וכל אות מתאימה במסמך תצביע אל האות המתאימה במערך. אם נרצה לשנות לאות מסוימת את המצב, צבע, פונט או גודל, נבדוק האם קיים מופע של אובייקט אחר עם מצב דומה, אם כן נשנה את ההצבעה של האות הזו אליה. במידה ואין מופע כזה, ניצור מופע חדש שלה ונכניס אותו אל המערך/זיכרון.

12.9 Structural patterns: Proxy

תבנית העיצוב האחרונה שנראה בהרצאה זו נקראת Proxy או בעברית בא כוח. ניתן דוגמא לשימוש בproxy בהקשר של רשתות - ה-proxy מקבל בקשות מהלקוח, במידה והוא יודע למלא את הבקשה בעצמו הוא ממלא אותו אותה (לדוגמא ע"י זיכרון מטמון פנימי שקיים אצלו) ואם לא הוא מעביר אותה הלאה אל השרת האמיתי שיכול למלא את בקשת הלקוח.
ה- Proxy בהקשר של אובייקטים יתרחש באופן דומה. נצרף את התרשים לתבנית זו -



הלקוח פונה אל אובייקט מסוג subject, הוא איננו מודע כלל אל ה-proxy. בעת הבקשה מגיע אל האובייקט מסוג proxy שמכיל בתוכו אובייקט

מסוג real subject וממש את הממשק של subject. במידה וה-proxy יודע כיצד למלא את בקשת הלקוח הוא יבצע זאת בעצמו מבלי לפנות אל האובייקט real subject. במידה והוא איננו יודע כיצד למלא את הבקשה הוא יעביר את הבקשה הלאה אל ה-real subject.
ההבדל בין proxy ל- object adapter הוא שכאן אנו מכילים את אובייקט מסוג האובייקט שאנו ממשים ואילו ב-object adaptor אנו מכילים אובייקט מסוג מסוים וממשים אובייקט אחר.
ההבדל בין proxy ל-decorator הוא שכאן ההכלה היא של אובייקט ספציפי real subject ואילו ב-decorator ההכלה היא כללית יותר.

ה-proxy יכול לבצע מספר פעולות -

- ניהול זיכרון מטמון
- אם עלות יצור אובייקט היא יקרה, הוא יכול להחליט שהוא יוצר אובייקט חדש רק במקרים מסוימים ומחזיר לך מצביע לאובייקט קיים במקרים אחרים
- ניהול הרשאות גישה
- ניהול זיכרון -smart reference, לספור כמה מצבעים יש לאובייקט מסוים וכד'