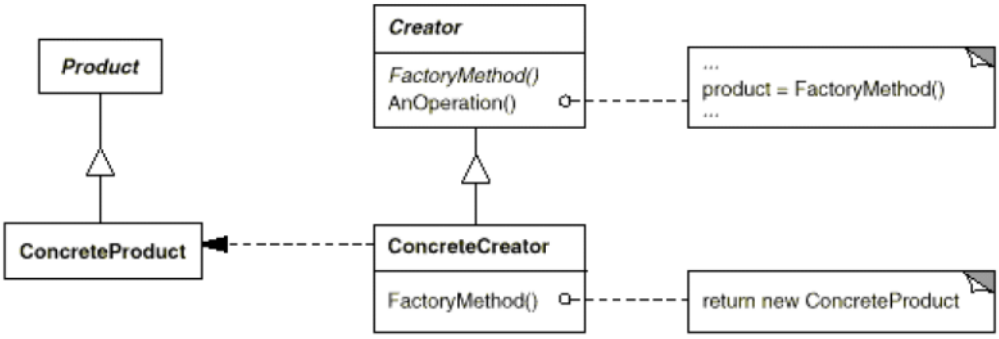
הרצאה 12 - Creational Patterns

יום שלישי 05 ינואר 2021 10:47

13.1 Creational Patterns: Factory

לאחר שעסקנו ב-Structural Patterns , כעת נעבור לעסוק ב-Creational Patterns, תבניות עיצוב ליצירת אובייקטים.
התבנית הראשונה שנראה נקראת Factory. נתבונן בתרשים של תבנית העיצוב -



ישנה מחלקה הנקראת creator ובתוכה קיימת מתודת ה-factory. זהו הממשק של תבנית העיצוב והוא יקרא בד"כ בשם Xfactory, כאשר X מייצג את הקבוצה (לדוג' עובדים, סטודנטים ועוד) שניתן לייצר בעזרת הממשק והמחלקות היורשות.
המחלקה concrete Creator היא מחלקה יורשת של הממשק והיא מייצרת אובייקט ספציפי (concrete Product).

ניתן להסתכל ולהגיד שע"י זה שכל אובייקט יצטרך עכשיו יוצר משלו, הוספנו עוד מחלקות רבות, שלכאורה הן מיותרות! למה לא להשתמש ב-New x מבלי להשתמש במחלקה יוצרת?
התשובה לכך היא, שקודם כל ע"י תבנית העיצוב Factory, אנו מפרידים בין היצירה לבין המוצר הסופי (decoupling). בנוסף, זה מאפשר לנו להשתמש בטכניקות של תכנות שלא היו מתאפשרות אחרת. כאשר יש לנו מחלקות יוצרות, יש לנו אובייקטים, ואם יש אובייקטים אז ניתן להשתמש בהם בכל מיני צורות, לדוגמא להכניס אותם למבנה נתונים ועוד.
נתבונן בדוגמא הממחישה את היכולת להשתמש בטכניקה תכנותית בעזרת תבנית העיצוב הזו. נניח ויש לנו n טיפוסים של עובדים (workers). אנו רוצים, שבהינתן קלט מהמשתמש, נייצר מופע לעובד הרלוונטי לפי הקלט. כמובן שלהשתמש ב-n תנאים של if, זה לא כל כך יישומי ועלול לקחת O(n) זמן. נשתמש בתבנית העיצוב Factory ובעזרת טריק נוסף נחזיר את האובייקט ב-O(1).
ראשית נאתחל את הממשק היוצר שלנו, הממשק יקרא Worker Factory ויהיה בעל מתודה אחת - create -

```
public class WorkerFactory {
 private interface Creator{
  public Worker create();
 }
```

עבור כל סוד של עובד ניצור מחלקה היוצרת אותו. לדוגמא עבור העובד מרצה, ניצור Lecturer Creator שמחזיר אובייקט מסוג מרצה -

```
 private class LecturerCreator implements Creator{
  public Worker create() {
   return new Lecturer();
  }
 }
```

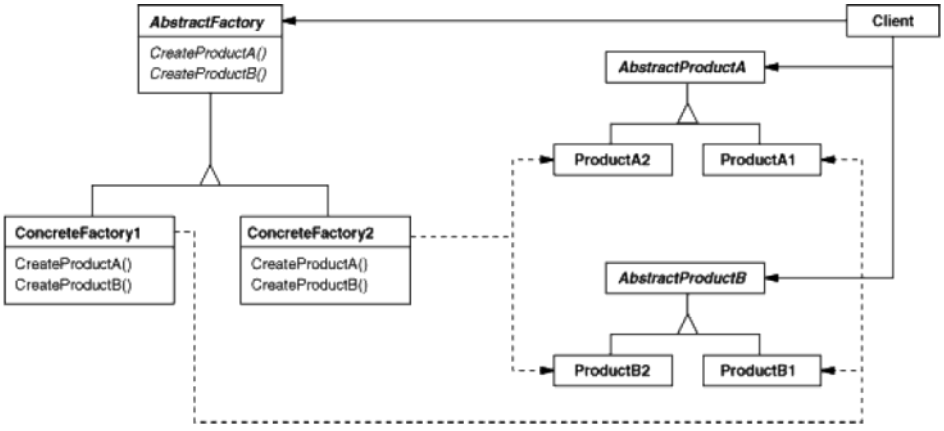
בשלב הבא נשתמש בטבלת גיבוב (HashMap), הטבלה תקבל מחרוזת (שם העובד), ויוצר של העובד (בהתאמה למחרוזת).
נזין את כל סוגי העובדים בטבלה (את הזנת הערכים ניתן להכניס למתודה שנקרא לה Worker Factory). לאחר שהכנסנו את הערכים אל טבלת הגיבוב, נרצה שבהינתן הקלט מהמשתמש נחזיר לו אובייקט מהסוג הנדרש. לשם כך נשתמש במתודה נוספת, שנקרא לה create Worker , המחזירה אובייקט מסוג עובד. במתודה נחפש את העובד שהמשתמש הכניס בקלט בטבלת הגיבוב, במידה והוא קיים נשתמש במתודה create של האובייקט -

```java
HashMap<String,Creator> workersCreators;

public WorkerFactory() {
 workersCreators=new HashMap<String, Creator>();
 workersCreators.put("admin", ()->new Admin());
 workersCreators.put("ta", ()->new TA());
 workersCreators.put("lecturer",()->new Lecturer());
 // notice, takes O(n) memory
}

public Worker createWorker(String type){
 Creator c=workersCreators.get(type);
 // takes O(1) time!
 if(c!=null) return c.create();
 return null;
}
}
```

כך, בעזרת תבנית העיצוב Factory, וטכניקות תכנות, ב-O(1), החזרנו למשתמש את האובייקט הרצוי.

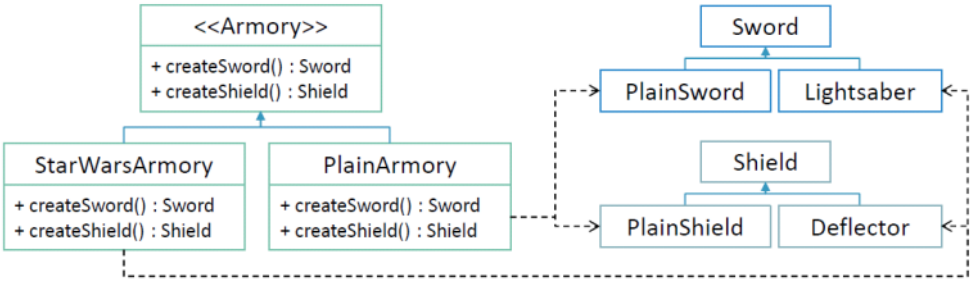
13.2 Creational Patterns: Abstract Factory

זוהי תבנית השונה מתבנית ה-Factory במופשטותה (Abstraction), כלומר קיימים מימושים שונים (יתבהר בדוגמא). נתבונן בתרשים תבנית העיצוב -



בתבנית זו אנו רואים את הלקוח שמכיר את הממשק Abstract Factory ואת Abstract Product A, B. בממשק Abstract Factory קיימות שתי מתודות יוצרות, מתודה יוצרת למוצר A ולמוצר B. נשים לב שלמתודות היוצרות (Abstract Product A, B) יש שתי סוגי של מוצרים לכל אחד. אנו נרצה לוודא שלא יהיה מצב שהלקוח יקבל מוצר מסוג A1 ו-B2 או A2 ו-B1 אלא רק מוצרים מאותו האינדקס (דהיינו A1-B1 או A2-B2). לשם כך קיימות שתי מחלקות יורשות לממשק, כאשר כל מחלקה יורשת יוצרת מוצר של A ו-B, אך שומרות על האחידות (כמו שניתן לראות בתרשים).

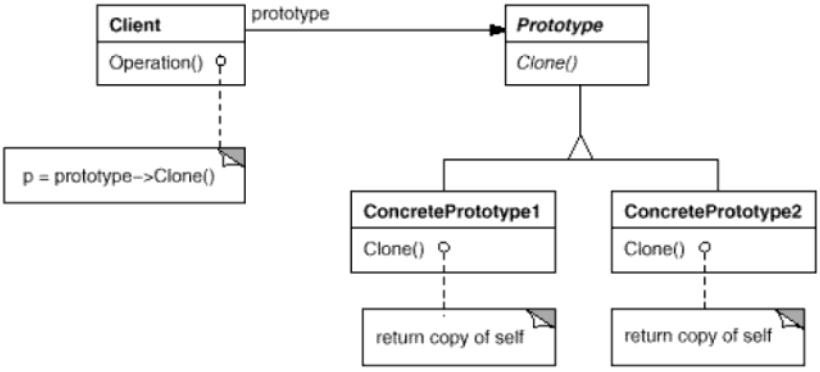
נמחיש את הרעיון בעזרת דוגמא. נניח שקיימים שתי דמויות, אחת משר הטבעות ואחת ממלחמת הכוכבים. לכל דמות יש שריון וחרב, אך לדמות אחת השריון הוא מגן מעץ והחרב היא מברזל ועבור הדמות השנייה המגן הוא deflector והחרב היא lightsaber. נרצה לדאוג שכאשר מאתחלים דמות היא תקבל את החרב והמגן שלה ולא יווצרו ערבובים של חרב מברזל עם deflector וכד'.
לכן נשתמש בממשק מופשט שנקרא לו Armory. בממשק זה יהיו מתודות create Sword ו- create Shield. למחלקה זו יהיו שתי יורשים Star Wars Armory ו- Plain Armory . כל יורש יוצר את הכלים אך ורק לפי הצירוף המתאים לו ובצורה זו נמנע ערבובים לא נכונים של כלי המלחמה. נצרף תרשים להמחשה -



נעיר שתבנית זו אינה שומרת על עיקרון הפתוח/סגור (open/closed) שלמדנו. כיוון שאם נרצה להוסיף כלי נשק בממשק, נצטרך לעדכן את המחלקות היורשות גם כן. לכן עלינו להכיר ולהבין את תבניות העיצוב על מנת שנוכל להתאים אותם אל צרכינו בדיוק.

13.2 Creational Patterns: Prototype

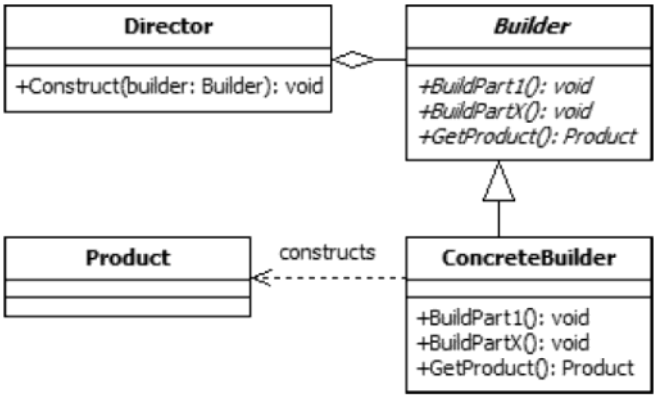
ראשית נמחיש את הצורך שגרם ליצירת תבנית עיצוב זו. נניח וקיים ממשק הנקרא shape. לממשק זה יש מספר מחלקות יורשות - Circle, Rectangle, Triangle ועוד. אנו רוצים ליצור רשימה עם העתק של כל הצורות הקיימות במקום מסוים. לשם כך יצרנו מתודה הנקראת add Copy המקבלת אובייקט מסוג s, ומבצעת העתק. אך נשים לב שהמתודה לא יכולה לעבוד שכן היא מקבלת אובייקט ממשק והיא לא יודעת איזה אובייקט ספציפי ולכן לא תוכל לבצע העתקה שלו! לשם כך הגיעה תבנית העיצוב prototype. נתבונן בתרשים של תבנית העיצוב -



בתבנית זו הלקוח מכיר את הממשק Prototype המכיל מתודת clone. היורשים השונים של הממשק יכילו גם הם את מתודת clone.
בצורה כזו, במידה ונצטרך לשכפל אובייקטים מסוג הממשק (כלומר מחלקות יורשות מבלי לדעת איזו מחלקה ספציפית) נוכל להפעיל את מתודת ה-clone ולקבל שכפול מתאים.
היתרונות של שיטה זו הן היכולת לבצע שכפול בלי תלות במחלקה הספציפית והמהירות שלה (אין צורך לברר איזה אובייקט ספציפי וכד'). החסרונות הן שהמתודה clone לא יכולה לקבל פרמטרים, ואם היא כן מקבלת פרמטרים זה פרמטרים שאמורים להתאים **לכל** מי שממש את הממשק. כמו כן עלינו לשים לב שמבצעים את ההעתק המתאים - העתק שטחי או עמוק.

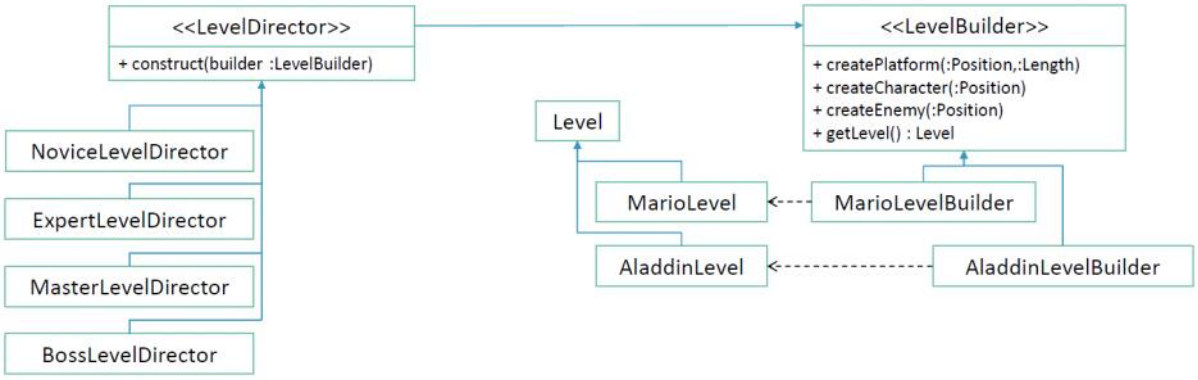
13.3 Creational Patterns: Builder

הבעיה שתבנית עיצוב זו פותרת קשורה ביצירת אובייקט הנוצר בעזרת אלגוריתם. כעת משתמשים באלגוריתם מסוים אבל יכול להיות שבעתיד נרצה לשנות אותו. לכן עלינו להפריד בין האלגוריתם שיוצר את האובייקט לבין החלקים שהאובייקט מורכב מהם. נתבונן בתרשים של תבנית העיצוב -

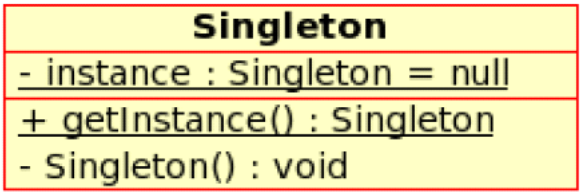


בתרשים אנו רואים שתי ממשקים - Builder, Director. בממשק Builder, נמצאים כל החלקים שהאובייקט מורכב מהם ולכן אובייקטים שונים יכולים לרשת אותו ולממש בצורות שונות. לעומת זאת בממשק Director, נמצא הבנאי, האלגוריתם היוצר של האובייקט. המחלקות שירשו את הממשק הזה, יממשו אלגוריתמים שונים שיוצרים את האובייקט.

נמחיש בעזרת דוגמא. נניח ויש לנו שתי משחקי מחשב- אלאדין ומריו (מאחורי הקלעים הם אותו הדבר, רק הגרפיקה שונה).
נרצה לממש עבור כל משחק שלבים שונים אך נרצה לדאוג שלא נצטרך לממש בכל פעם מחלקה לכל סוג משחק ולכל שלב.
לכן נשתמש בתבנית העיצוב Builder. נפריד בין האלג', סוגי השלבים השונים, לבין החלקים שהאובייקט מורכב מהם, האובייקטים השונים הנמצאים במשחק - לדוגמא: אויבים, מספר השלב, מיקום הדמות ועוד.
לכן ניצור ממשק לאלג' הנקרא Level Director. את הממשק יירשו השלבים השונים - שלב מתחיל, מתקדם, בוס וכד'.
מנגד, ניצור ממשק לאובייקטים הנקרא Level Builder. את הממשק יירשו סוגי המשחקים השונים. נצרף תרשים להמחשה -



לסיום, נראה תבנית עיצוב נוספת הנקראת Singleton. תבנית עיצוב זו נועדה להגבלת יצירת אובייקט לפעם אחת. מתי זה שימושי? נניח ויש לנו אובייקט גדול במיוחד ולא נרצה ליצור מופעים נוספים שלו, או לחילופין יש לנו אובייקט אחראי, לדוגמא מנהל משימות, לא נרצה שיהיו כמה אובייקטים כאלו ואז עלול להיווצר בעיות - אז נוכל להשתמש בתבנית העיצוב Singleton.
כיצד עובדת תבנית זו? נצרף המחשה ונסביר -



במחלקה יש בנאי ואובייקט מסוג Singleton פרטיים וכן מתודה סטטית get Instance. הדרך ליצור מופע של האובייקט היא בצורה הבאה -
נקרא למתודת ה-get Instance והיא תפעל כך -
במידה והערך שנמצא ב- Singleton שווה ל-null קרא לבנאי, צור מופע חדש, שמור את המופע ב- singleton והחזר אותו.
במידה והערך ב-singleton לא שווה ל-null, כלומר קיים מופע ב-singleton , החזר אותו.
בצורה זו נוכל לדאוג שאכן נוצר רק מופע אחד של האובייקט. נעיר שזה טוב רק כאשר עובדים בלי תהליכונים (threads), במידה ועובדים עם תהליכונים, ייתכן מצב שבו שתי תהליכונים יבדקו האם singleton שווה ל-null, ובשניהם הוא יהיה שווה ל-null ולכן שניהם יצרו מופע שלו. על כך נלמד בסמסטר ב.

13.4 Creational Patterns: Builder Example

לסיום הרצאה זו, נתבונן בדוגמא מורכבת יותר לשימוש בתבנית העיצוב Builder.
נניח שאנו רוצים לייצר אובייקט מסוג רובוט ואנו רוצים שהוא יהיה בלתי ניתן לשינוי (mutex) לאחר יצירתו. לרובוט יש שדות שהן הכרחיים - name, id. וכן 3 שדות אפשריים (תכונות של סוגי רובוטים שונים) - mass, flyable, autonomous. נרצה שאת השדות האפשריים יהיה ניתן למלא בכל סדר. נשים לב שכל השדות יסומנו בתור final (אנו עובדים ב- java כרגע) על מנת למנוע שינוי. כמו כן לא נוכל לייצר בנאים לכל אפשרות (כלומר רובוט שיכול לעוף עם מסה, שיכול לעוף אוטונומי וכד') מכיוון שאם יש n תכונות נצטרך לכתוב 2^n בנאים.
לכן נרצה להשתמש בתבנית העיצוב builder ולראות כיצד היא פותרת את הבעיה.
ראשית במחלקת הרובוט ניצור בנאי פרטי המקבל אובייקט מסוג Robot Builder -

```
// Java example
public class Robot {

    final String name;   // required
    final int id;        // required
    final double mass;
    final boolean flyable;
    final boolean autonomous;

    private Robot(RobotBuilder rb){
        name=rb.name;
        id=rb.id;
        mass=rb.mass;
        flyable=rb.flyable;
        autonomous=rb.autonomous;
    }
```

כעת בתוך המחלקה רובוט ניצור את המחלקה Robot Builder. במחלקה זו יהיו את אותם שדות בדיוק כאשר השם ו-id מסומנים כ-final אך שאר השדות (השדות האפשריים) לא מסומנים כ-final. הבנאי של המחלקה מקבל שם ו-id ויש לו מתודת set X עבור כל תכונת רצויה. לבסוף מתודה בשם build, הקוראת לבנאי של מחלקת הרובוט (בנאי פרטי) עם this בתור ה-Robot Builder. כך זה יראה בקוד -

```java
// inside Robot class
public static class RobotBuilder{
    final String name;
    final int id;
    double mass;
    boolean flyable;
    boolean autonomous;

    public RobotBuilder(String name, int id) {
        this.name=name;
        this.id=id;
    }
```

```java
public RobotBuilder setMass(double mass){
    this.mass=mass;
    return this;
}
public RobotBuilder setFlyable(boolean flyable){
    this.flyable=flyable;
    return this;
}
public RobotBuilder setAutonmous(boolean automous){
    this.autonomous=automous;
    return this;
}

public Robot build(){
    return new Robot(this);
}
```

נראה דוגמא ל-main אפשרי -

```java
public static void main(String[] args) {
  Robot r1=new Robot.RobotBuilder("RR1",1).build();
  Robot r2=new Robot.RobotBuilder("RR2",2).setAutonmous(true).build();
  Robot r3=new Robot.RobotBuilder("RR3",3).setMass(54.5).setFlyable(true).build();
}
```