גירסה 1.00 – 12.11.2004

תכנות מונחה אספקטים

ניר אדר

Nir Adar

Email: underwar@hotmail.com

Home Page: http://underwar.livedns.co.il

אנא שלחו תיקונים והערות אל המחבר.

תוכן עניינים

הקדמה והגדרות בסיסיות

concern זהו מטרה, רעיון או תחום עניין. בתחום הטכנולוגי, מערכת תוכנה מכילה מספר **core concern**, שאלו המטרות העיקריות אותן מבצעת התוכנה, וכן **system concerns** שהם פעולות שפחות קשורות ישירות ל-business logic של התוכנה – כגון logging, ניהול כניסת משתמשים למערכת, אבטחה וכדו'. System concerns רבים נוגעים בחלקים שונים של המערכת, ולא רק במודול אחד, ולכן הם מכונים **crosscutting concerns**.

בפרדיגמות התכנות הקיימות כיום crosscutting concerns נמצאים בתוך הקוד של המודולים השונים במערכת, וכתוצאה מכך מתווסף סיבוך למערכת, המתבטא בתכנון, בהבנה, במימוש ובשיפור של המערכת.

תכנות מונחה אספקטים הוא פרדיגמה חדשה המפרידה crosscutting concerns טוב יותר מהמתודולוגיות הקיימות, על ידי הוספת מודולריות ל-crosscutting concerns.

תכנות מונחה עצמים משמש כיום כשיטת הפיתוח לרוב פרויקטי התוכנה החדשים. תכנות מונחה עצמים הוכיח שימושיות רבה במידול בעיות המכילים אובייקטים בעלי התנהגות דומה. עם זאת, תכנות מונחה עצמים אינו מספק מענה מספיק כאשר אנו באים למדל התנהגות המשותפת למודולים רבים, שאינם בהכרח קשורים אחד לשני. תכנות מונחה אספקטים בא כדי לסגור חלל זה.

ראית המערכת כאוסף של concerns

ניתן להתייחס למערכות תוכנה מורכבות כמימוש של מספר רב של concerns. את ה-concerns השונים ניתן לחלק לקטגוריות: business logic, שמירת נתוני המערכת, אבטחה, ביצועים, בדיקת שגיאות ועוד.

במערכות קיימות כל מודול תוכנית מכיל חלקים השייכים לכל אחת מהמטרות: המודול יכול להכיל קוד השייך ל-business logic, קוד הקשור לאבטחה, קוד הקשור לבדיקת שגיאות, קוד לשמירת נתונים וכו'. ביחד כל המודולים השונים עומדים בדרישות הפרויקט כפי שהוגדרו.

המטרה אותה נרצה להשיג: לאחר הסתכלות על הדרישות והגדרת ה-concerns של מערכת התוכנה, נרצה לממש את ה-concerns בנפרד, בלי שישתלבו בקוד אחד של השני.

Crosscutting concerns במערכת תוכנה

מפתח יוצר מערכת שתעמוד בדרישות רבות. עבור כל מודול, ניתן לסווג דרישות אלו לשתי קטגוריות גדולות: דרישות המהוות את מטרת המודול – core module level requirements, ודרישות ברמת המערכת – system level requirements. דרישות ברמת המערכת הן לרוב דרישות שאינן תלויות אחת בשניה או בדרישות ברמת המודול. עם זאת הדרישות משפיעות על המימוש של מודולים רבים.

נביט בקוד ה-Java הבא:

```
public class SomeBusinessClass extends OtherBusinessClass
{
    // Core data members

    // Other data members: Log stream, data-consistency flag

    // Override methods in the base class

    public void performSomeOperation(OperationInformation info)
    {
        // Ensure authentication

        // Ensure info satisfies contracts

        // Lock the object to ensure data-consistency in case other
        // threads access it

        // Ensure the cache is up to date

        // Log the start of operation

        // ==== Perform the core operation ====

        // Log the completion of operation

        // Unlock the object
    }

    // More operations similar to above

    public void save(PersitanceStorage ps)
    {
    }

    public void load(PersitanceStorage ps)
    {
    }
}
```

נשים לב למספר נקודות:

- Other data members הם משתנים שאינם קשורים למטרה העיקרית של המודול.
- performSomeOperation מבצעת הרבה יותר מאת הפעולה העיקרית עצמה. היא מבצעת בדיקת חוקיות המשתמש, בדיקת נתונים, ניהול cache, ניהול multithreading ועוד. פעולות אלו יכולות להיות רלוונטיות גם עבור חלקים רבים אחרים של התוכנית.
- הפונקציות save ו-load המשמשות לשמירת המחלקה או טעינתה – לא ברור למה הן צריכות להיות חלק עיקרי מהמחלקה עצמה.

בעיות הקשורות ל-crosscutting concerns

דרישות Crosscutting concerns הן דרישות המשפיעות על מודולים רבים בתוכנית. בעיתיות הקיימת במתודולוגיות הנוכחיות היא שלמרות עובדה זו, אנו צריכים לטפל בכל ה-crosscutting concerns בכל מודול בנפרד. כתוצאה מכך, המעקב "איזה דרישות כבר מומשו" הופך להיות מסובך (כל מודול צריך מעקב משלו עבור כל אחת מה-crosscutting concerns).

מספר מאפיינים יכולים לגלות לנו מימוש גרוע של crosscutting concerns במתודולוגיות הקיימות.

- **Code tangling ("סיבוך קוד")**: מודולים במערכת התוכנה מתעסקים בו זמנית עם מספר דרישות. למשל: business logic, ביצועי המערכת, סנכרון ואבטחה. דרישות רבות גורמות להופעת אלמנטים הקשורים לכל אחד מה-concerns בקוד, ולהפיכתו למסובך. ("הקוד המטפל ב-consern מעורב עם קוד אחר").
- **Code Scattering ("פיזור הקוד")**: הקוד המטפל ב-concern מפוזר בכל המערכת (בכל המחלקות הרלוונטיות).

השפעות של crosscutting concerns על פרוייקטים קיימים:

- קשיים במעקב אחרי התקדמות הפרויקט ומילוי הדרישות.
- התקדמות איטית בפרויקט – המתכנתים מבזבזים זמן רב על מילוי כל הדרישות המשניות, ומתרכזים פחות במטרות הפרויקט העיקריות.
- קוד שקשה לקחת אותו לשימוש חוזר – הותאם ל-crosscutting concerns הספציפיים לבעיה.
- קשיים בהוספת דרישות חדשות ושינויים למערכת הקיימת.

יסודות התכנות מונחה האספקטים

עד כה ציינו שהפיכת ה-Crosscutting concerns למודולריים יותר יכולה לעזור בעת מימוש התוכנית. מספר מחקרים נעשו בתחום הפרדת ה-concerns מהקוד. תכנות מונחה אספקטים זהו אחד מהפתרונות שהוצעו.

AOP נותן למתכנת את הכלים לממש concerns באופן נפרד, ולאחד מימוש אלו כדי ליצור את המערכת הסופית. על ידי AOP, אנו מממשים את ה-Crosscutting concerns בצורה מודולרית. יחידת המידול של AOP מכונה aspect - בהקבלה לכך שיחידת המידול בתכנות מונחה עצמים היא class.

שלבי פיתוח בתכנות מונחה אספקטים:

1. זיהוי האספקטים השונים של המערכת.
2. מימוש מודול נפרד עבור כל אספקט.
3. שזירת האספקטים למערכת אחת – **Weaving** - מתבצעת על ידי כלי אוטומטי המשלב את האספקטים השונים ליצירת המוצר הסופי.

AOP שונה מ-OOP בדרך בה הוא מתייחס אל ה-crosscutting concerns. בתכנות מונחה אספקטים, המימוש של כל אספקט נכתב באופן בלתי תלוי באחרים – איננו מסתמכים במימוש של concerns אחד על כך שקיימים concerns אחרים. למשל, קטע הקוד המטפל, למשל, במשיכת כספים מהבנק יטפל רק בנושא המשיכה, כשהוא פועל באופן בלתי מודע לכך שקיימים logging ובדיקת חוקיות לפעולות שהוא מבצע במערכת.

זוהי למעשה <u>הנחת עבודה בסיסית</u> של תכנות מונחה אספקטים: התוכנית הבסיסית (ללא האספקטים) אינה מתייחסת לעובדה שקיימים כאלו).

ההגיון של הנחה זו: אם הפתרון שנציע מקיים את הנחה זו, הרי שבעתיד כשיתגלו דרישות נוספות עבור הפרויקט נוכל לשלב גם אותן באותה קלות. אספקטים הם נקודות שראינו לאחר כתיבת הקוד המקורי, שאם לא כך והיינו יודעים את כל הבעיות והדרישות מלכתחילה, היינו מסוגלים למשל למדל את כל המערכת על ידי OOP, למשל.

מסקנה הנובעת מכך היא ששפה התומכת ב-AOP צריכה לתמוך בתחביר שיאפשר להגדיר היכן בקוד אספקטים יכולים להשתלב.

סוגי אספקטים:

- Spectative - צופים או מקליטים את המידע, אך אינם משנים אותו.
- Regulative - משפיעים על זרימת ריצת התוכנית/סיום התוכנית.
- Invasive - משנים ערך של שדות קיימים בתוכנית.

אספקטים מוגדרים על ידי הגדרת aspect לפי התחביר המתאים. אספקטים כוללים:

- **הצהרות pointcut**: הצהרות המזהות היכן האספקט מוסף/איזה קוד האספקט מחליף.
- **הצהרות advice**: מה לעשות במקום הפעולה המקורית של התוכנית.
- אספקטים יכולים לכלול מתודות חדשות, קוד ומשתנים חדשים.

סוגים של הצהרות advice:

הסוגים השונים נבדלים ביניהם לפי הרגע בו מתבצע ה-advice.

- **before** – ה-advice מתבצע לפני ביצוע הקוד שב-join point.
- **after returning** – ה-advice מתבצע לאחר שקוד ה-join point הסתיים באופן נורמלי.
- **after throwing** – ה-advice מתבצע לאחר שקוד ה-join point הסתיים על ידי חריגה.
- **after** – ה-advice מתבצע לאחר שקוד ה-join point הסתיים (בכל אחת מן הדרכים).
- **around** – ברגע שמגיעים אל ה-join point ה-advice מקבל שליטה מלאה מתי ואם התוכנית תמשיך הלאה.

AspectJ

AspectJ הוא תוסף ל-Java המאפשר תכנות מונחה אספקטים. התוסף מאפשר להשתמש ב-Java כשפת הבסיס לפיתוח, ומגדיר הרחבות לשפה כדי לאפשר הגדרת אספקטים ושזירתם בתוכנית.

על מנת לבצע זאת התוסף מגדיר את המושגים pointcuts, join points, advice ו-aspects.

Join points אלו הגדרות של נקודות ספציפיות בתוכנית. **pointcut** זה תחביר השפה המיועד לתיאור ה-join point. ה-**advice** מגדיר את הקוד שצריך לרוץ בנקודות ה-pointcut.

aspect זהו האלמנט המאחד את האלמנטים הנ"ל תחת מסגרת אחת.

AspectJ מאפשר להגדיר אספקטים המשתלבים על אספקטים אחרים. הוא מאפשר להוסיף Data members למחלקות, שיטות חדשות ואפילו להגדיר מחלקות חדשות.

AspectJ's weaver הוא מהדר המייצר קוד בשפת Java הכולל בתוכו את האספקטים השונים.

דוגמא לקוד AspectJ (לקוחה מהמאמר I want my AOP, Part 1):

```
public aspect LogCreditCardProcessorOperations {
    Logger logger = new StdoutLogger();

    pointcut publicOperation():
        execution(public * CreditCardProcessor.*(..));

    pointcut publicOperationCardAmountArgs(CreditCard card,
                                           Money amount):
        publicOperation() && args(card, amount);

    before(CreditCard card, Money amount):
        publicOperationCardAmountArgs(card, amount) {
        logOperation("Starting",
             thisjoin point.getSignature().toString(), card, amount);
    }

    after(CreditCard card, Money amount) returning:
        publicOperationCardAmountArgs(card, amount) {
        logOperation("Completing",
            thisjoin point.getSignature().toString(), card, amount);
    }

    after (CreditCard card, Money amount) throwing (Exception e):
        publicOperationCardAmountArgs(card, amount) {
        logOperation("Exception " + e,
            thisjoin point.getSignature().toString(), card, amount);
    }

    private void logOperation(String status, String operation,
                              CreditCard card, Money amount) {
        logger.log(status + " " + operation +
                   " Card: " + card + " Amount: " + amount);
    }
}
```

סביבות נוספות

קיימות סביבות נוספות התומכות בתכנות מונחה אספקטים. רוב השפות – ממומשות מעל Java, אם כי גם לשפות אחרות (C#, C++ ועוד) קיימים כלים לתכנות מונחה אספקטים.

שפות כאלו הן למשל Aspect# - תוכנת Open Source לתכנות מונחה אספקטים בסביבת C#, או HyperJ, תוכנה נוספת מבית IBM המציגה דרך ראיה מעט שונה לגבי האספקטים, וכן תוכנות רבות נוספות.

שפות וסביבות נבדלים ביניהם בנושאים כגון:

- Dynamic Aspects – האם ניתן לשלב/להסיר אספקטים תוך כדי ריצת התוכנית, או רק בזמן ההידור? (AspectJ תומך רק באספקטים המוספים בזמן ההידור).
- האם האספקטים קשורים לשפה ולאפליקציה ספציפית או שניתן לעשות בהם שימוש חוזר באפליקציות אחרות?
- איך בודקים את נכונות מימוש האספקטים?
- כושר הביטוי של השפה מול הביצועים שלה.

מקורות

1. Ramnivas Laddad (2002), "**I want my AOP!, Part 1**"
2. Shmuel Katz (2004), "**An Overview of Aspects**"
3. wordIQ.com (2004), "**Definitions of Aspects, Aspect-oriented programming, Core concern, Join points, and pointcuts**"