תכנות אדפטיבי דינאמי

הפרדת מטרות על ידי שימוש בתורת הגרפים

ניר אדר

תוכן עניינים

תקציר

תכנות מונחה עצמים יוצר תלות חזקה בין עץ המחלקות למימוש התוכנית. פתרון לבעיה זו מושג על ידי *החוק של Demeter*. אולם, פתרון זה גורם להיווצרות מספר גדול של מתודות קצרות המפוזרות ברחבי התוכנית, שבעקבותם קשה להבין את התמונה הגדולה של פעולת התוכנית. *תכנות אדפטיבי* מציע פתרון לבעיה זו וכן נותן תמיכה טובה בהפרדת concerns. מספר כלים קיימים על מנת לאפשר למתכנת להשתמש בתכנות אדפטיבי. הכלי DemeterJ מספק למתכנת סביבת עבודה התומכת בתכנות אדפטיבי. הכלי DJ הינו חבילת JAVA המאפשרת תכנות אדפטיבי דינמי הנבנה בזמן ריצה. במסמך זה נציג את התכנות האדפטיבי באמצעות כלי זה.

החוק של Demeter

הגדרות ומבוא

הגדרה: המושג **Loosely coupled** מתייחס להגדרת ממשקים בין חלקי התוכנה, בצורה כזו שחלקים שונים בתוכנה בלתי תלויים במימוש של חלקים אחרים. במערכות Loosely coupled, ניתן לקחת רכיב ולהחליף אותו ברכיב אחר ללא השפעה על המערכת.

החוק של Demeter (**"Law of Demeter"** או בקיצר LoD) הוא כלל אצבע שמטרתו להוריד את התלות של חלקי תוכנית במימוש ובממשק של חלקי התוכנית האחרים. כלל זה הוצג לראשונה בצורה המוכרת כיום במאמר בשם "Assuring good style for object-oriented programs" [9] שפורסם ב-1989. המוטיבציה מאחורי חוק זה היא שמפתחים ומתחזקים של רכיב לא יצטרכו להכיר את כל המערכת בה הרכיב משתלב, וכאשר הם באים לשנות אותו, אלא כמה שפחות מחלקי המערכת.

החוק המוכלל של Demeter:

רכיבי התוכנית השונים צריכים להכיר רק את רכיבי התוכנית האחרים הקשורים אליהם באופן קרוב ("closely related").

עבור תכנות מונחה עצמים, הרכיבים הקשורים למתודה כלשהי באופן קרוב הם השדות של האובייקט בו היא נמצאת, וכן המתודות השייכות לפרמטרים שלה.

נוכל לנסח את **החוק של Demeter עבור תכנות מונחה עצמים:**

מתודה M השייכת למחלקה C צריכה להשתמש אך ורק במתודות השייכות לארגומנטים שלה (כולל this), במתודות השייכות לאובייקטים הנוצרים על ידי M וברכיבים הישירים של המחלקה C.

דוגמא

הדוגמא הבאה לקוחה מהמאמר "The Paperboy, The Wallet, and The Law Of Demeter". [4] עם נציין כי המאמר אינו מציג את הבעיות של חוק זה, שיוצגו להלן, אלא רק את יתרונות השימוש בו.

בעולם שלנו קיימים *מחלק העיתונים* (**Paperboy**), *לקוח* (**Customer**) *וארנק* של הלקוח (**Wallet**). אנו רוצים לייצג את הסיטואציה בה מחלק העיתונים מצפה לקבל את התשלום שלו מן הלקוח. נבנה מחלקות פשוטות למימוש מצב זה.

ללקוח שלנו יש שם, שם משפחה וכן ארנק באמצעותו הוא משלם:

```
public class Customer {
	private String firstName;
	private String lastName;
	private Wallet myWallet;

	public String getFirstName() {
		return firstName;
	}
	public String getLastName() {
		return lastName;
	}
	public Wallet getWallet() {
		return myWallet;
	}
}
```

הארנק הוא בעצמו מחלקה. נציע מימוש אפשרי לה, המאפשר פעולות ניהול של הכסף שבארנק.
מימוש זה הוא פשוט לצורך הדגמת הפונקציונליות. יש לזכור כי ניתן להרחיב את הארנק באופנים רבים, כגון אפשרות הוספת פריטים לארנק, כרטיס אשראי וכדו'.

```
public class Wallet {
	private float value;

	public float getTotalMoney() {
		return value;
	}
	public void setTotalMoney(float newValue) {
		value = newValue;
	}
	public void addMoney(float deposit) {
		value += deposit;
	}
	public void subtractMoney(float debit) {
		value -= debit;
	}
}
```

נניח כעת כי מחלק העיתונים הגיע לבית הלקוח, מסר ללקוח את העיתון וכעת הוא מחכה לקבל את התשלום המגיע לו. הקוד יכול להראות כך:

```
Paying the Paperboy
	// code from some method inside the Paperboy class
	payment = 200; // "I want my two dollars!"
	Wallet theWallet = myCustomer.getWallet();
	if (theWallet.getTotalMoney() > payment)  {
		theWallet.subtractMoney(payment);
	}
	else {
		// come back later and get my money
	}
```

הקוד מקבל את הארנק של המשתמש, בודק שיש בו מספיק כסף ומעביר את הכסף אל מחלק העיתונים.

לכאורה – תוכנית זו היא תוכנית תקינה הממומשת בעזרת תכנות מונחה עצמים.
כעת נשאל – מה היא הבעייתיות בקוד זה?
נחשוב לרגע במונחים של העולם האמיתי – לרוב לקוח לא ייתן את הארנק שלו למחלק העיתונים, כדי שיבדוק אם יש בו מספיק כסף וייקח משם את המגיע לו. הלקוח אינו בהכרח סומך על מחלק העיתונים. כמו כן, בארנק יכולים להיות כרטיסי אשראי ופריטים אחרים, שאינן מעניינו של מחלק העיתונים. אם נכליל את הבעיה – מחלק העיתונים בדוגמא זו חשוף ליותר מידע מזה שצריך להיות לו.

מחלקת מחלק העיתונים מודעת לכך שקיים למחלקה לקוח עצם מסוג ארנק, וכן היא יכולה לבצע עליו פעולות שונות. אם נבצע שינויים כעת במחלקת הארנק, נצטרך להדר מחדש (ואולי אף לשנות) את מחלקת מחלק העיתונים. קוד זה הפך למעשה את המחלקות קשורות אחת לשניה בקשר חזק.

נעלה בעיה נוספת: נניח כעת שבעולם שלנו ישנו גם גנב, המסוגל לגנוב את ארנקו של הלקוח. במקרה זה מודל הגיוני יהיה כזה שיקבע את ערכו של אובייקט הארנק להיות null:

```
victim.setWallet(null);
```

כמו שנכתב למעלה, קוד מחלק העיתונים יגרום לשגיאת זמן ריצה – הוא מניח כי ללקוח קיים ארנק. ניתן כמובן להוסיף בדיקה כזו לקוד, אולם כפי שאנחנו רואים, הקוד של כל מתודה המשתמשת בארנק מתחיל להסתבך יותר ויותר.

הפתרון לבעיה זו ניתן לנו על ידי החוק של Demeter.
נשנה את המחלקות כך שכאשר מחלק העיתונים מגיע אל דלתו של הלקוח, הוא ייבקש מהלקוח את התשלום. מחלק העיתונים לא יתעסק עם הארנקים של הלקוחות, ויותר מכך, הוא אפילו אינו צריך לדעת שיש להם ארנקים.
נציג מימוש חדש למחלקת הלקוח ולקטע הקוד של מחלק העיתונים:

```
public class Customer {
	private String firstName;
	private String lastName;
	private Wallet myWallet;
	public String getFirstName() {
		return firstName;
	}
	public String getLastName() {
		return lastName;
	}
	public float getPayment(float bill) {
		if (myWallet != null) {
			if (myWallet.getTotalMoney() > bill) {
				theWallet.subtractMoney(payment);
				return payment;
			}
		}
		else {
			// come back later and get my money
		}
	}
}
```

נשים לב שללקוח אינו חושף יותר כי יש ברשותו ארנק. במקום הפונקציה getWallet() המחזירה את הארנק כתבנו בדוגמא זו את הפונקציה getPayment() הדואגת לתשלום, ואינה חושפת את אובייקט הארנק הפנימי.

נביט כעת בקוד של מחלק העיתונים:

```
// code from some method inside the Paperboy class
payment = 200; // "I want my two dollars!"
paidAmount = myCustomer.getPayment(payment);
if (paidAmount == payment) {
      // say thank you and give customer a receipt
} else {
      // come back later and get my money
}
```

מדוע קוד זה יותר טוב?

ראשית, אנחנו מציגים במקרה זה מודל הקרוב יותר לתרחיש בעולם האמיתי - מחלק העיתונים מבקש מהלקוח לשלם - ואינו ניגש ישירות אל הארנק של הלקוח.

סיבה שניה היא שכעת נוכל לשנות את מחלקת הארנק ללא שום צורך לשנות את מחלקת מחלק העיתונים. התלות בין מחלקות אלו נעלמה.

הסיבה השלישית (והחשובה ביותר מבחינת תכנות מונחה העצמים), היא שנוכל לשנות את מימוש מחלקת הלקוח, למשל להחליף את ההתנהגות של getPayment(), מבלי לשנות את מחלקת מחלק העיתונים.

מה החסרונות של פתרון זה?

- מחלקת הלקוח נהייתה יותר מסובכת. הפונקציה getPayment() שנוספה לה מסובכת הרבה יותר מ-getWallet() שהיתה שם קודם. אם זאת לא הוספנו סיבוך למערכת. הסיבוך כבר קודם היה במערכת וכעת הוא רוכז בתוך מחלקת הלקוח. לפיכך, נושא זה אינו ממש חסרון. על ידי ריכוז הסיבוך במקום אחד, אנו דווקא מרוויחים.
- הבעיתיות של פתרון זה מתגלה במערכות המספקות יותר פונקציונליות. אם נרצה לחשוף פונקציונליות נוספת של מחלקת הארנק או של רכיבים אחרים של מחלקת הלקוח, נצטרך לכתוב מתודות נוספת במחלקת הלקוח. למשל, נניח שוטר עוצר את הלקוח ומבקש את רשיון הנהיגה הנמצא בארנק, יתכן שנוסיף מתודה בשם getDrivingLicense(). תרחיש אופייני למערכות המשתמשות בקוד של Demeter: למחלקות השונות קיימות מתודות רבות בנות שורות בודדות, המשמשות ככיסוי לפונקציונליות של האובייקטים הפנימיים שלהן.

מסקנות

החוק של Demeter אכן מאפשר תחזוקה קלה יותר של תוכנית על ידי הפחתת התלות בין חלקי התוכנית השונים. אולם, בזמן שחוק זה מאפשר תחזוקה קלה של התוכנית ונותן לה עמידות לשינויים של מחלקות, החוק גורם לכך שבסופו של דבר יהיו בתוכנית שלנו הרבה מאוד wrapper methods המשמשות להעביר הודעות אל הרכיבים של האובייקט. כתיבתן של מתודות אלו לוקחת זמן, וכן מסבכת את הקוד. החוק גורם להיווצרות מספר גדול של מתודות קצרות המפוזרות ברחבי התוכנית, שבעקבותם קשה להבין את התמונה הגדולה של פעולת התוכנית. על מנת לפתור בעיה זו, נציג פרדיגמה חדשה הבאה להפוך תהליך זה של יצירת wrapper methods לאוטומטי ושקוף למתכנת.

Adaptive Programming

מוטיבציה

החוק של Demeter עבור תכנות מונחה עצמים נראה הגיוני ותורם. עם זאת, כאשר אנחנו משתמשים בו מתעוררות בעיות אחרות. המסקנה אליה אנו מגיעים: האבסטרקציה של "קופסה שחורה", הניתנת על ידי תכנות מונחה עצמים, איננה מספיק טובה! "קופסה שחורה" מפרידה בין הממשק למימוש. היא מאפשרת למתכנת לשנות מימוש של מתודה מסויימת בלי לפגוע בשלמות התוכנית. אם זאת, היא אינה מפרידה בין ממשק המחלקות לאלו המשתמשים בו.
נחפש אבסטרקציה אחרת שתאפשר לנו לבצע הפרדה זו. ההפרדה שנציע תאפשר לנו לשנות את ממשק המשתמש ללא שינוי קוד המשתמש באותה מחלקה.

Adaptive Traversal

נציג כעת הסתכלות חדשה על תוכנית, ובעזרתה נפתח מינוח בו שוב נוכל לראות את הבעיה, אולם גם את הפתרון לה.
מצב התוכנית בזמן ריצה, בייחוד מצב של תוכניות מונחות עצמים, יכול להיות מיוצג על ידי גרף מכוון, בו האובייקטים הקיימים בתוכנית מיוצגים על ידי צמתים, והשדות מיוצגים על ידי קשתות.
אם נרחיב את גישה זו, נוכל אף לראות מהלך ריצת כל התוכנית כטיול על אותו הגרף.

הגדרות אפשרויות **לטיול על הגרף**:

1. קבוצת האובייקטים אליהם ניתן להגיע מאובייקט נתון שלהם מאפיין מסוים, נצפו במהלך הטיול.
2. בוצעה פעולה כלשהי על תת קבוצה של קבוצת האובייקטים הישיגים מאובייקט נתון.

בטכניקות התכנות הסטנדרטיות (ללא חוק Demeter) ביטוי של טיול על גרף מעורב בהסתמכות חזקה על מבנה המחלקות שבכל המסלול (מכיוון שכל קפיצה בין צמתים במסלול מבוטאת בקוד מהצורה "a.b"). עובדה זו נכונה אפילו במקרים בהם כל מה שמעניין אותנו במסלול זוהי נקודת ההתחלה והאובייקטים שהם המטרה.

תזכורת: **concern** זהו מטרה, רעיון או תחום עניין - תחום כלשהו בתוכנית המעניין את המתכנת.
נכנה concern אשר עוסק בטיול בין אובייקטים לצורך מימוש התנהגות מסוימת על אובייקטים אלו בשם **traversal-related concern**. תוכנית טיפוסית הפועלת על קבוצות גדולות של אובייקטים מכילה concerns רבים מסוג זה. concerns אלו קיימים כבר בשלב התיכון של המערכת, והם תופסים ביטוי גם כאשר אנחנו עוברים מהתכן אל המימוש.
הפתרון המבוצע על ידי מתכנתים כיום כדי לממש traversal concern הוא לכתוב מתודות מעבר בכל מחלקה שאובייקטים שלה משתתפים במסלול הטיול. שיטה זו מאפשרת לממש traversal concern, אולם היא מסבכת את הקוד (*code tangling*) וכן גורמת לכך שקוד המימוש של ה-concern יתפרש על פני מחלקות רבות (*code scattering*). תופעה זו הינה התופעה אותה ראינו בדוגמאות הקודמות.
כעת, כאשר אנו מציגים את מצב התוכנית כגרף, נוכל לחשוב על פתרון חדש לבעיה.

פרדיגמה בשם **traversal strategies** תשמש אותנו על מנת להפוך את ה- traversal related concerns למודולריים. בנוסף, פרדיגמה זו תפריד את הקשר בין ה-concern אל המבנה המדוייק של מחלקות האובייקטים הנמצאים במסלול בו אנו מטיילים.
הרעיון של פרדיגמה זו הוא להגדיר בשפה עילית את מבנה הטיול, כאשר רק מספר נקודות מפתח מצויינות במפורש (נקודת ההתחלה, מה מחפשים בטיול, מה מבצעים כשמוצאים עצם שעונה על הדרישות וכדו'). בזמן ההידור של התוכנית, בהינתן המבנה הקונקרטי של המחלקות בתוכנית, מיוצר קוד המממש את הטיול עם כל הפרטים המתאימים. מכיוון שבצורה זו הטיול קשור בצורה מינימלית למבנה המחלקות, שינויים במבנה המחלקות המשתתפות בטיול לא ידרוש שינוי, או ידרוש שינוי קטן בלבד, במימוש הטיול.

נכנה בשם **Adaptive Programming** תכנות המשתמש ב-traversal strategies.

דוגמא למשפט שתכנות אדפטיבי מאפשר לנו לבטא:

```
Starting from object A, go to object C via all objects with an
attribute named "x".
```

תכנות אדפטיבי מסתמך על התאמה של החוק הכללי של Demeter לתכנות שמרכזו הוא ה-concerns.
נקבל את **חוק Demeter עבור concerns**:

> בעת מימוש של concern, אובייקט צריך לתקשר רק עם אובייקטים הקשורים באופן מובהק אל אותו ה-concern.

לדוגמא:
נניח שברצוננו לבנות מערכת הקשורה לניהול של אוטובוסים והנוסעים בהם. לאוטובוס יש מספר תחנות (הנשמרות ברשימה). בכל תחנה מספר נוסעים ממתינים. נשאל: כמה נוסעים ממתינים ישנם?
לפי גישת התכנות הסטנדטרטית - נממש מתודה העוברת על התחנות, סופרת את מספר הנוסעים בכל תחנה וסוכמת את התוצאה. הגישה תיפול כאשר נשנה את הממשק שמציעה המחלקה אוטובוס. למשל, הנוסעים המחכים לאוטובוס יהיו מסודרים ברשימה משלהם, במקום תחת התחנות השונות בדרך. במקרה כזה של שינוי ממשק מחלקת האוטובוס, כל הפונקציות המניחות שהנוסעים מסודרים לפי התחנות יצטרכו לעבור שינוי.

כאשר נתכנת לפי גישת התכנות האדפטיבי נתייחס לגרף של האובייקטים, כאשר אובייקט השורש הינו האוטובוס, ואובייקטי המטרה הם כל האובייקטים מסוג "נוסע" הישיגים ממנו.
האלגוריתם שלנו יהיה: "התחל באובייקט 'אוטובוס'. סכום את כל האובייקטים מסוג 'נוסע' הישיגים מאובייקט זה והחזר סכום זה". אלגוריתם זה יעמוד גם בשינוי של המסלול בין אוטובוס לנוסע, אם הקשר ביניהם יוגדר סופית רק בזמן ההידור.

הערה: נשים לב שבתכנות האדפטיבי גרף האובייקטים איננו מושג מופשט אלא צורת חשיבה וכלי בו אנו משתמשים כל הזמן.

Dynamic Adaptive Programming - DJ בשפת Java

תכנות אדפטיבי דינמי

כפי שהוסבר, כאשר אנחנו מתכנתים בתכנות אדפטיבי, בזמן ההידור המהדר יוצר את הקוד באמצעותו המחלקות מתקשרות אחת עם השניה (וחוסך למשתמש את הצורך להגדיר אותן בעצמו).
תכנות אדפטיבי דינמי הוא תכנות אדפטיבי בו המהדר אינו מייצר מראש קוד עבור המחלקות. ההחלטה על המסלול אותו נצעד בגרף נקבעת בזמן ריצה. כיצד נושא זה ממומש?
שפת Java תומכת ב-*Reflections*. **Reflections** היא טכנולוגיה המאפשרת לנו, בהינתן מחלקה, לזהות בזמן ריצה את המתודות והמאפיינים שלה, ובאופן דינמי לחלוטין להריץ מתודות השייכות לה.
תכונה זו של Java מאפשרת לממש traversal strategy שתפוענח בזמן ריצה.

חבילת DJ - מבוא

החבילה DJ שנכתבה על ידי צוות הפיתוח Demeter מספקת לנו תמיכה בתכנות אדפטיבי.

מטרות העל של החבילה:

- מתמקדת בטיפול ב-concerns מסוג traversal-related concerns.
- ביטול התלות בין מטרות הטיול לבין עץ המחלקות.
- אפשור תכנות אדפטיבי בצורה ידידותית למתכנת.

מימוש המחלקה:

השימוש ב-Reflections מאפשר להוסיף ל-Java תמיכה ב-traversal strategies בקלות – הוא גורם לכך שאין צורך בשינוי תהליך הקומפילציה של התוכנית שלנו, ואף אין צורך בכתיבת כלי מיוחד שיבין את התחביר של strategies אלו. האסטרטגיות לטיול בגרף יכולות להיות מוגדרות כמחרוזות פשוטות בתוך קוד התוכנית. ניתן אפילו להשתמש במחרוזות שנוצרו בזמן הריצה ולא היו ידועות כלל בזמן ההידור לצורך הגדרת האסטרטגיות ועל ידי כך הספריה משיגה דינמיות מרבית.

רכיבי החבילה

הספריה DJ מכילה מספר מחלקות המיועדות להפוך את הגדרת הטיול בגרף לקלה להגדרה, להבנה ולתחזוקה.

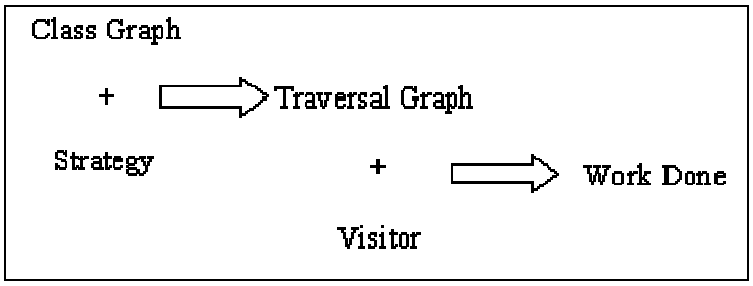
מבחינת תפיסה, אנחנו יכולים לחשוב על שני סוגי גרפים:

- **גרף המחלקות של התוכנית** – גרף זה קיים ללא קשר לגישה החדשה שאנו מציגים – זהו גרף ה-UML המתאים למחלקות התוכנית. בספריית DJ המחלקה **ClassGraph** הולך לייצג גרף זה.
- **גרף האובייקטים של התוכנית** – זהו הגרף עליו דיברנו קודם. מבנהו דומה לגרף המחלקות, אולם זהו הגרף המתאים לאובייקטים הקיימים בזמן הריצה. המחלקה **ObjectGraph** מתאימה לגרף זה.

לגבי הטיול עצמו: האלמנט הראשון המגדיר את הטיול הוא נקודת ההתחלה ונקודת הסיום (ואולי צמתים מיוחדים באמצע). אלמנט זה מכונה האסטרטגיה שלנו, ומיוצג על ידי אובייקט מסוג המחלקה **Strategy**.

חוץ מהאסטרטגיה קיים "מבקר" המטייל על הגרף ומבצע על הצמתים בו פעולות. מבקר זה יוגדר על ידי המחלקה **Visitor**.

שלבי הפעולה יכולים להיות מתוארים על ידי השרטוט הבא (הלקוח מאתר הפרויקט Demeter):



אובייקט מסוג **ClassGraph** הוא גרף מכוון בו הצמתים הם מחלקות, והקשתות מבטאות קשרי is-a וקשרי has-a בין המחלקות המתאימות. המחלקה ClassGraph מספקת מתודות ליצירת ותחזוק גרף מחלקות כנ"ל. השימוש הפשוט ביותר במחלקה זו הוא יצירת אובייקט מסוג ClassGraph על ידי קריאה לבנאי ללא פרמטרים. במקרה זה המחלקה תיצור גרף מחלקות המבוסס על כל המחלקות הנמצאות ב-default packet.
המחלקה תומכת בהוספת חלקים נוספים לעץ המחלקות במהלך הריצה. העץ ש-ClassGraph מחזיקה אינו של האובייקטים, אלא גרף המתאר את תרשים ה-UML המתאים למחלקות.

המחלקה **Strategy** מגדירה את הטיול. אנו מסוגלים להגדיר את נקודת ההתחלה, נקודת הסיום ותחנות מעבר בדרך. חבילת DJ מקיימת את הדרישה שבכל מקום בו אנו משתמשים באובייקט מסוג Strategy, נוכל להשתמש גם באובייקט מסוג מחרוזת במקומו על מנת להשיג את אותו אפקט. נביט מיד בשימוש בתכונה זו.

אובייקטים מסוג **Visitor** שומרים את המידע הנדרש ליישום הטיול. האובייקט מגדיר מה לעשות במהלך הטיול כאשר פוגשים עצמים מסוג מסויים. למשל, הוא שומר ייצוג של רעיונות כגון: "אם הגעת לאובייקט מסוג Person – תוסיף אותו לסכום".

נציג כעת דוגמא לשימוש במחלקות אלו.

נניח כי אנחנו מממשים מחלקה המייצגת חברה (Company), ואנו רוצים לחשב את שכר כל העובדים של החברה. הפתרון שנציע אינו תלוי במבנה עץ מחלקות המייצגות את החברה.

נתחיל בטיול על הגרף באובייקט מסוג Company עבורו אנו רוצים לחשב את שכר העובדים. ממנו נמצא את כל האובייקטים מסוג "משכורות" ונסכום אותם.

קוד למימוש האלגוריתם המשתמש בספריית DJ:

```
class Company {
      Double sumSalaries(edu.neu.ccs.demeter.dj.ClassGraph cg) {
            String s = "from Company to Salary";
            edu.neu.ccs.demeter.dj.Visitor v =
              new edu.neu.ccs.demeter.dj.Visitor() {
                  private double sum;

                  public void start() {
                        sum = 0.0;
                  }
                  public void before(Salary host) {
                        sum += host.getValue();
                  }

                  public Object getReturnValue() {
                        return new Double(sum);
                  }
            }

            return (Double) cg.traverse(this, s, v);
      }
}
```

ClassGraph מתאר את מבנה המחלקות והקשרים ביניהן. בדוגמא זו איננו יוצרים אותו. Visitor מגדיר את מהלך הטיול. אנו יוצרים משתנה sum, וקובעים כי בהתחלה (start) ערכו יהיה שווה ל-0. before היא פונקציה המופעלת בכל פעם שאנו מגיעים לאובייקט מסוג Salary, לפני הקריאה למתודה בו. בדוגמא לעיל אנו מוסיפים לסכום הכולל את הערך שבאובייקט. נשים לב שפונקציה זו מזוהה על פי הפרמטר שהיא מקבלת – כלומר – נניח אנחנו רוצים להתייחס לאובייקט מסוג מסויים על ידי Visitor, פשוט נכתוב פונקצית before המקבלת אותו. מכיוון ש-DJ פועל בעזרת Reflections, הוא יזהה פונקציה זו ויקרא לה במקומות המתאימים. כמו כן, באופן מקביל קיימת הפונקציה after, הפועלת על האובייקט אחרי שביצענו פעולה עליו, רגע לפני שעוברים לאובייקט הבא. הפונקציה getReturnValue() קובעת מה הערך שה-Visitor יחזיר בסוף.

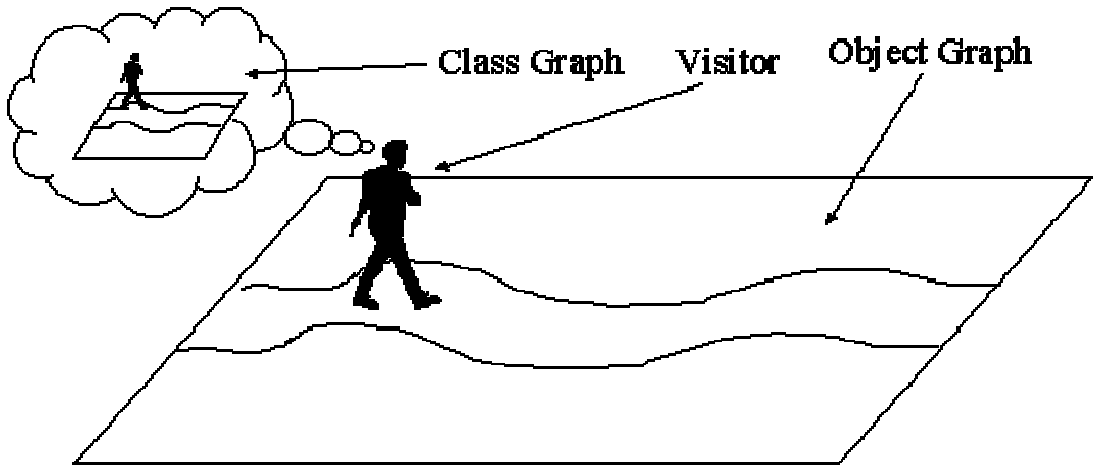
בדוגמא איננו יוצרים אובייקט מסוג Strategy. אנו מגדירים מחרוזת (s) המשמשת לקביעת האסטרטגיה.

המתודות בהן תומכת המחלקה Visitor:

```
public void start();
public void finish();
public void before(<Class>);
public void after(<Class>);
public Object getReturnValue();
```

השימוש ב-Reflections לצורך מימוש הטיול ניכר במחלקה Visitor - אנו יכולים להגדיר מתודות before, after בכמות כלשהי כרצוננו. בזמן ריצה הספריה מנתחת את המחלקה שאנו מעבירים לה ומשתמשת במתודות הקיימות בה.

המחלקה Strategy מגדירה את נקודת המוצא, נקודת היעד, ואופציונאלית גם תחנות ביניים. ניתן להשתמש במחרוזת בכל מקום שמשתמשים במחלקה זו. ניתן להתייחס למחרוזת כאל ביטוי רגולארי המתאים לאסטרטגיה. על ידי השימוש במחרוזת, ניתן לבנות ולהגדיר אסטרטגיות חדשות בזמן ריצה. ניתן להגדיר התנהגויות שונות לצורת הסיור בגרף על ידי האסטרטגיה. מסמך זה לא ייכנס לפרטים אודות התנהגות זו. המתעניינים יוכלו להרחיב בנושא בעזרת התיעוד של מחלקות אלו.



חסרונות הספרייה

החסרון העיקרי והבולט של הספריה כרגע הוא המימוש הלא יעיל שלה. לפי המידע המסופק על ידי הצוות המפתח, פתרון הממומש על ידי הספריה הוא איטי פי 30 מפתרון הממומש בצורה ישירה. צוות המימוש טוען כי בעתיד על ידי שיפור המימוש הם יוכלו להגיע לשפר את הספריה כך שתהיה איטית פי 10 ממימוש בצורה ישירה. כמו כן, מימוש יעיל יותר של reflections בגרסאות עתידיות של Java ישפר את ביצועי הספריה.

סיכום

כאשר הודעה עוברת בין האובייקטים, בייחוד כשאנחנו מסתכלים על הייצוג שלהם כגרף, ניתן להסתכל על מעבר ההודעה כאל מעבר גל.
הכלים שפרויקט Demeter מספק לנו מאפשרים לנו כאשר אנו באים לבעיה לקבוע אך ורק את התנאים המקדימים, התנאים הסופיים והאינווריאנטים של הטיול. בצורה כזו כאשר אנו פותרים בעיה איננו פותרים אותה רק עבור מודל האובייקטים הנוכחי, אלא עבור משפחה שלמה של אובייקטים שלהם ישנה אותה התנהגות לוגית.
הגדרת חוקי המעבר בגרף בעזרת Visitor והאסטרטגיה מאפשרת לנו להגדיר במפורש רק את המעברים המיוחדים, אלו שבהם צריכים לבצע פעולה מיוחדת. המעברים האחרים נעשים באופן שקוף למתכנת.
הרעיון המרכזי: כאשר אנחנו מתעסקים עם פחות מידע כתיבת המערכת תהיה פשוטה יותר, והמערכת תהיה יותר ניתנת לתחזוקה וניתנת לשינויים בקלות.

תכנות אדפטיבי ככלל, ובפרט הכלים של Demeter, מעלים את תכנות מונחה העצים לרמת הפשטה גבוהה יותר בכך שהם מאפשרים למשתמש להתעסק רק במחלקות המפתח בתהליך ובאינווריאנטים לגבי הקשרים ביניהן.

כלים נוספים

הכלי DJ הוא רק אחד מהכלים שצוות הפיתוח של Demeter פיתח.
שני כלים נוספים שנכתבו על ידי מציעים תמיכה בתכנות אדפטיבי:

הכלי DemeterJ מספק סביבת עבודה המרחיבה את Java לצורך תמיכה בתכנות אדפטיבי.
בנוסף לקבצי ה-Java שנכתוב, נשתמש בתוכנית שלנו בקבצים נוספים המשתמשים בתחביר מורחב שהכלי נותן. הגדרת הטיול נעשית בזמן ההידור, ולכן הביצועים של סביבה זו טובים. החסרון העיקרי של הסביבה היא שעקומת הלמידה שלה ארוכה יותר מזו הנדרשת ללימוד DJ, וכן

הכלי DAJ מתחבר עם סביבת העבודה AspectJ המיועדת לתכנות מונחה אספקטים. הוא מציע ממשק דומה לזה של DJ, אך על ידי שימוש באספקטים הוא גורם ליצירת קוד הטיול בזמן ההידור, ולפיכך לביצועים משופרים. כלי זה נהנה למעשה מהיתרונות של DJ וגם מאלו של DemeterJ.
כלי זה נכתב כחלק מעבודת המסטר של John J. Sung שמטרתה היתה להבין כיצד ניתן לשלב תכנות אדטפיבי עם תכנות מונחה אספקטים בצורה הטובה ביותר.

הקשר בין תכנות אדפטיבי לתכנות מונחה אספקטים

תכנות אדפטיבי הופיע מספר שנים לפני תכנות מונחה אספקטים. אם זאת, אנו מתייחסים אל תכנות אדפטיבי כאל מקרה מיוחד של תכנות מונחה אספקטים. לפי ההגדרה המחודשת של הוגה התכנות האדפטיבי, Karl Lieberherr, תכנות אדפטיבי הוא תכנות מונחה אספקטים בו concern אחד לפחות מבוטא במושגים של קשתים, צמתים וטיול על גרף.

תכנות מונחה אספקטים ותכנות אדפטיבי עוסקים באותה בעיה – יצירת הפרדה בין ה-concerns השונים המרכיבים את האפליקציה.
הרעיון המרכזי של תכנות מונחה אספקטים הוא "**מודולריות טובה יותר עבור concerns**", ואילו הרעיון המרכזי של תכנות אדפטיבי הוא "**concern-shyness**" – הסתרת פעולת ה-concerns מ-concerns אחרים. לפני הצגת AOP, התכנות האדפטיבי ניסה למלא את שתי המטרות גם יחד.

תכנות אדפטיבי יכול להיות ממומש בקלות בעזרת תכנות מונחה אספקטים, מאשר על ידי השפות הקיימות כיום. הדוגמא הטובה ביותר לכך היא הכלי DAJ המספק תמיכה בתכנות אדפטיבי בשפת Java, ועושה זאת על ידי שימוש ב-AspectJ.

מקורות

1. Baris Aktemur (2004), "**Aspect Oriented Programming**"
2. Bradford Appleton (1996), "**Introducing Demeter and its Laws**"
3. Bradford Appleton (1995), "**The Law of Demeter for Propagation Patterns**"
4. David Bock (2000), "**The Paperboy, The Wallet, and The Law Of Demeter**"
5. Doug Kaye (2002), "**Web Services Strategies**"
6. John J. Sung (2002), "**DAJ User's Guide V1.1**"
7. Karl Lieberherr, Doug Orleans (2000), "**A Short Introduction to Adaptive Programming**"
8. Karl Lieberherr (2003), "**AOSD and the Law of Demeter: Shyness in Programming**"
9. Karl Lieberherr, and Holland, I. (1989), "**Assuring good style for object-oriented programs**", IEEE Software, pp 38-48
10. Karl Lieberherr, Doug Orleans, Johan ovlinger (2001), "**Aspect-Orient Programming with Adaptive Methods**"
11. Karl Lieberherr's (1997), "**Connections between Demeter/Adaptive Programming and Aspect-Oriented Programming (AOP)**"
12. Karl Lieberherr (2001), "**Coupling Mechanisms in Aspect-Oriented Software**"
13. Karl Lieberherr (2001), "**Computer Science Adapts Techniques from Engineering and Mathematics**"
14. Karl Lieberherr (2004), "**Controlling the Complexity of Software Designs**"
15. Karl Lieberherr, Doug Orleans (2001), "**DJ: Dynamic Adaptive Programming in Java**"
16. Karl Lieberherr, Ian Holland (2002), "**Preventive Maintenance of Object-Oriented Software**"
17. Karl Lieberherr, Boaz Patt-Shamir, Doug Orleans (2004), "**Traversals of Object Structures: Specification and Efficient Implementation**"
18. wordIQ.com (2004), "**Definition of Tree traversal**"
19. www.soaprpc.com (2004), "**Web Service FAQ**"