Πανεπιστήμιο Πατρών
Πολυτεχνική Σχολή
Τμήμα Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών
Τομέας Ηλεκτρονικής και Υπολογιστών

Τελική Έκθεση Προόδου
για την εργασία του μαθήματος
Ανάλυση και Σχεδίαση Συστημάτων Λογισμικού

Εργασία 9:
Aspect-Oriented Programming.
Μια νέα προσέγγιση στην ανάπτυξη λογισμικού.

Θωμάς Ρεπαντής
Έτος: Ε΄
Κύκλος Σπουδών: Ηλεκτρονική και Υπολογιστές
Α.Μ.: 4218
E-mail: darkzero@otenet.gr

Πάτρα, 12.2.2002

Εισαγωγή. Σκοπός της εργασίας είναι η παρουσίαση μιας νέας προγραμματιστικής τεχνικής, του Aspect-Oriented Programming (AOP) ή Προγραμματισμού Προσανατολισμένου σε Όψεις (ΠΠΟ), καθώς και η παρουσίαση και μελέτη συγκεκριμένων παραδειγμάτων (case-studies) που χρησιμοποιούν μια επέκταση σε μια γλώσσα προγραμματισμού, ώστε να υποστηρίζεται ο Προγραμματισμός Προσανατολισμένος σε Όψεις. Συγκεκριμένα επιλέχθηκε το AspectJ, που επεκτείνει τη γλώσσα προγραμματισμού Java.

Εισαγωγή στο Aspect-Oriented Programming. Είναι γεγονός πως οι απαιτήσεις από τα έργα λογισμικού ολοένα αυξάνουν, με αποτέλεσμα αυτά να γίνονται όλο και πιο πολύπλοκα. Ένα χαρακτηριστικό της πολυπλοκότητας αυτής είναι η επανάληψη λειτουργιών, όπως έλεγχοι ασφαλείας, διαχείριση μνήμης, διαμοίραση πόρων και χειρισμός λαθών και αποτυχιών, σε όλο το εύρος μιας εφαρμογής. Τέτοιες επαναλήψεις δημιουργούν προβλήματα όταν χρειαστεί να γίνουν αλλαγές στα αντίστοιχα σημεία του κώδικα, μιας και οι ίδιες αλλαγές επιβάλλεται να γίνουν σε πολλά διαφορετικά σημεία, με αποτέλεσμα να διαφύγουν κάποιες. Προκειμένου να ξεπεραστεί το πρόβλημα αυτό, συζητούνται μεθοδολογίες βασισμένες σε ένα νέο προγραμματιστικό στοιχείο, το *aspect* (όψη). Μια όψη είναι ένα κομμάτι κώδικα που περιγράφει μια επαναλαμβανόμενη ιδιότητα ενός προγράμματος [1]. Μια εφαρμογή μπορεί φυσικά να περιλαμβάνει περισσότερες όψεις. Μία όψη επιτρέπει επομένως στον προγραμματιστή να αναφέρεται ενιαία σε όλα τα τμήματα κώδικα που η όψη ορίζει ως συνδεδεμένα.

Βασικές αρχές του Aspect-Oriented Programming. Ο Προγραμματισμός Προσανατολισμένος σε Όψεις βασίζεται στην ιδέα ότι τα υπολογιστικά συστήματα προγραμματίζονται καλύτερα αν καθορίσουμε ξεχωριστά τα διάφορα συμφέροντα (*concerns*) (ιδιότητες ή περιοχές ενδιαφέροντος) ενός συστήματος και κάποια περιγραφή των σχέσεών τους και κατόπιν αφήσουμε σε μηχανισμούς στο υποκείμενο περιβάλλον ΠΠΟ το καθήκον να τα συνθέσουν σε ένα συνεπές πρόγραμμα [2]. Η εστίαση στα διασταυρούμενα συμφέροντα (*cross-scutting concerns*) διαφοροποιεί τον Προγραμματισμό Προσανατολισμένο σε Όψεις από προηγούμενες τεχνολογίες διαχωρισμού συμφερόντων (*separation of concerns*). Λέμε ότι δύο συμφέροντα διασταυρώνονται εάν οι μέθοδοι που σχετίζονται με αυτά επικαλύπτονται. Για παράδειγμα [3] οι μέθοδοι που ορίζουν τη θέση δύο γεωμετρικών σχημάτων ανήκουν στις αντίστοιχες κλάσεις των σχημάτων, θα μπορούσαν όμως να ανήκουν και όλες μαζί σε μια μέθοδο του διαχειριστή οθόνης που ανανεώνει την οθόνη μετά από κάθε μετακίνηση. Ο Προγραμματισμός Προσανατολισμένος σε Όψεις προσπαθεί να παρέχει ενιαία

αντιμετώπιση των διασταυρούμενων συμφερόντων (*crosscutting modularity*). Με άλλα λόγια οι προγραμματιστές μπορούν να χρησιμοποιούν όψεις για να φτιάχνουν μονάδες λογισμικού (software modules) για αντικείμενα ενασχόλησης που τέμνουν διάφορα σημεία μιας εφαρμογής [1]. Έτσι μπορεί να μπει κάποια τάξη στο μπλέξιμο (*tangling*) που προκαλούν τα διασταυρούμενα συμφέροντα όταν αντιμετωπίζονται με μία παραδοσιακή γλώσσα προγραμματισμού. Προκειμένου να ομαδοποιήσουν (modularize) τα διασταυρούμενα συμφέροντα οι γλώσσες Προγραμματισμού Προσανατολισμένου σε Όψεις χρησιμοποιούν σημεία συνένωσης (*join points*), για να προσδιορίζουν τις θέσεις στον κώδικα όπου ανήκουν οι όψεις. Μία βασική διάκριση συστημάτων που υλοποιούν τον Προγραμματισμό Προσανατολισμένο σε Όψεις έγκειται στην τεχνολογία που χρησιμοποιείται για το συνδυασμό προγράμματος και όψεων. Οι προσεγγίσεις τύπου clear-box εξετάζουν το εσωτερικό του προγράμματος και των όψεων, παράγοντας ένα μείγμα αυτών. Οι προσεγγίσεις τύπου black-box καλύπτουν components με aspect wrappers [2].

Πλεονεκτήματα από τη χρήση του Aspect-Oriented Programming. Θα έλεγε κανείς πως οι ήδη υπάρχουσες γλώσσες προγραμματισμού μπορούν να καλύψουν την ανάγκη για διαχωρισμό των συμφερόντων με τη δημιουργία και την κλήση υποπρογραμμάτων. Ο Προγραμματισμός Προσανατολισμένος σε Όψεις σίγουρα δεν αρνείται την υπάρχουσα αυτή τεχνολογία. Ωστόσο τη συμπληρώνει, μιας και συχνά η έκφραση ενός συμφέροντος δεν μπορεί να υλοποιηθεί με ωραίο τρόπο με την κλήση μιας υπορουτίνας. Ένα συμφέρον του οποίου ο κώδικας μπλέκεται μέσα σε άλλα δομικά στοιχεία δημιουργεί αταξία. Οι όψεις είναι ένας μηχανισμός πέρα από τις υπορουτίνες και την κληρονομικότητα που επιτρέπει να περιορισθεί σε ένα συγκεκριμένο σημείο του κώδικα η έκφραση ενός διασταυρούμενου συμφέροντος [2].

Ένα επιπλέον μειονέκτημα των υποπρογραμμάτων είναι ότι απαιτούν γνώση και συνεργασία από την πλευρά των προγραμματιστών που τα καλούν. Πιο συγκεκριμένα αυτοί θα πρέπει να γνωρίζουν ότι πρέπει να καλέσουν ρητά μια συγκεκριμένη υπορουτίνα και να γνωρίζουν και πως να την καλέσουν (π.χ. να γνωρίζουν τη διεπαφή (ορίσματα κτλ.) που αυτή προσφέρει), απαιτήσεις που μπορεί να δυσκολεύουν την ανάπτυξη μεγάλων έργων λογισμικού που γίνεται από ξεχωριστές ομάδες για ξεχωριστά μέρη του κώδικα. Συστήματα που βασίζονται στον Προγραμματισμό Προσανατολισμένο σε Όψεις προσφέρουν υπονοούμενους μηχανισμούς κλήσης (implicit invocation mechanisms) για να τίθεται σε λειτουργία συμπεριφορά σε κώδικα του οποίου οι προγραμματιστές δεν είχαν υπ' όψιν τους τα επιπλέον συμφέροντα. Για παράδειγμα μπορούμε εντός μιας όψης που υλοποιεί π.χ. καταγραφή συμβάντων (logging) να δηλώ-

σουμε πως αυτή θα γίνεται σε κάθε κλήση π.χ. public μεθόδων από το package printers. Ο μεταγλωττιστής της γλώσσας Προγραμματισμού Προσανατολισμένου σε Όψεις θα ενσωματώσει την όψη αυτή σε όλες τις αντίστοιχες μεθόδους [1]. Έτσι με τον Προγραμματισμό Προσανατολισμένο σε Όψεις ένα τμήμα πηγαίου κώδικα δεν αντιστοιχεί γραμμικά σε ένα τμήμα μεταγλωττισμένου κώδικα, αλλά αντίθετα αντιστοιχεί σε όλες τις εμφανίσεις μιας όψης. Το κέρδος είναι πως όχι μόνο γράφουμε τον αντίστοιχο κώδικα μια φορά, αλλά επιπλέον ορίζουμε σε ένα μόνο σημείο που θα καλείται αυτός, αντί να τον καλούμε κάθε φορά που χρειάζεται από διάφορα σημεία του κώδικα.

Ερευνητική δραστηριότητα στο Aspect-Oriented Programming. Ο Karl Lieberherr ένας από τους πρώτους ερευνητές επί του θέματος ξεκινώντας από το *Law of Demeter*, σύμφωνα με τον οποίο τα αντικείμενα θα πρέπει να έχουν γνώση μόνο στενά συσχετιζόμενων με αυτά αντικειμένων και θα πρέπει να επικοινωνούν μόνο με αυτά [4], ανέπτυξε την ιδέα του adaptive programming, η οποία βρήκε πρακτική υλοποίηση στη βιβλιοθήκη για Java DJ [5].

Στα τέλη της δεκαετίας του ΄80 οι William Harrison και Harold Ossher μίλησαν για Subject-Oriented Programming , το οποίο επεκτάθηκε και στο ερευνητικό πρόγραμμα MDSOC της IBM Research. Στην περίπτωση αυτή τα συμφέροντα υλοποιούνται ως αυτοδύναμα κομμάτια κώδικα (hyperslices) και όχι ως επιπρόσθετως κώδικας όπως οι όψεις. Οι ιδέες αυτές υλοποιούνται στο εργαλείο Hyper/J, που επεκτείνει τη Java [6].

Ο Mehmet Aksit έχει ασχοληθεί με την ενσωμάτωση aspects σε composition filters [7]. Τα φίλτρα προσπαθούν να εκφράσουν ένα γενικό μηχανισμό αφαίρεσης στα δεδομένα.

Επεκτάσεις για την υποστήριξη AOP στις C και C++ επίσης ετοιμάζονται από διάφορες ερευνητικές ομάδες. Η AspectC μπορεί να χρησιμοποιηθεί για τη βελτίωση της συνεκτικότητας των δομών σε ένα λειτουργικό σύστημα [8], ενώ αντίστοιχες επεκτάσεις και στη C++ μπορούν να βοηθήσουν στον προγραμματισμό χαμηλού επιπέδου [9].

Η ομάδα που ασχολείται με τον Προγραμματισμό Προσανατολισμένο σε Όψεις στο Xerox PARC οδηγήθηκε στην ιδέα αυτή ασχολούμενη με πρωτόκολλα αντανάκλασης και μετααντικειμένων [10]. Με πρωτοπόρο τον Gregory Kiczales ανέπτυξε το AspectJ που επεκτείνει τη Java και -ξεφεύγοντας από τα στενά πλαίσια μιας ερευνητικής εργασίας- είναι αρκετά ώριμο για να χρησιμοποιηθεί σε πραγματικές εφαρμογές.

Η ιδέα του Προγραμματισμού Προσανατολισμένου σε Όψεις κερδίζει συνεχώς σε δημοσιότητα. Υπάρχει τοποθεσία στο Διαδίκτυο αφιερωμένη στην ανάπτυξη λογισμικού με προσανατολισμό στις όψεις, που περιέχει αναφορές σε γλώσσες, εργαλεία, εφαρμογές, μεθόδους και συνέδρια [11], ενώ το MIT Technology Review συμπεριέλαβε τον ΠΠΟ στις 10 ανερχόμενες περιοχές της τεχνολογίας που σύντομα θα έχουν βαθιά επίδραση στην οικονομία και στον τρόπο που ζούμε και εργαζόμαστε [12]

Εισαγωγή στο AspectJ. Για την παρουσίαση και μελέτη συγκεκριμένων παραδειγμάτων (case-studies) που χρησιμοποιούν Προγραμματισμό Προσανατολισμένο σε Όψεις επιλέξαμε το AspectJ, μιας και είναι μια από τις πλέον δημοφιλείς και ώριμες υλοποιήσεις. Επιπλέον είναι υλοποίηση ανοικτού κώδικα. Το AspectJ επεκτείνει απρόσκοπτα τη γλώσσα προγραμματισμού Java, ώστε να παρέχεται η δυνατότητα χρήσης όψεων προκειμένου να ομαδοποιούνται με ευανάγνωστο τρόπο τα διασταυρούμενα συμφέροντα [13]. Σε αυτά περιλαμβάνονται ο έλεγχος και ο χειρισμός λαθών, ο συγχρονισμός, βελτιστοποιήσεις επιδόσης, παρακολούθηση και καταγραφή συμβάντων και υποστήριξη εκσφαλμάτωσης. Η κυκλοφορία του AspectJ, στην έκδοση 1.0 πλέον, περιλαμβάνει compiler, structure browser, debugger, javadoc extension, IDE υποστήριξη για τα JBuilder, Forte, Emacs, καθώς και εγχειρίδια και παραδείγματα.

Τρόπος χρήσης του AspectJ. Μπορεί κανείς να χρησιμοποιήσεις το AspectJ είτε βοηθητικά στη διαδικασία ανάπτυξης, προσθέτοντας όψεις που εξυπηρετούν στην εκσφαλμάτωση, στον έλεγχο (π.χ. με βάση τη φιλοσοφία Design By Contract) ή στη ρύθμιση της απόδοσης μιας εφαμογής και οι οποίες δε συμπεριλαμβάνονται στο τελικό προϊόν, είτε κανονικά στην παραγωγή λογισμικού, οπότε και οι όψεις μένουν στο τελικό προϊόν.

Κατά τη διαδικασία της ανάπτυξης χρησιμοποιείται ο μεταγλωττιστής του AspectJ (ajc) αντί του javac και παράγει κώδικα εκτελέσιμο σε οποιαδήποτε πλατφόρμα Java. Εναλλακτικά ο ajc μπορεί να παράγει πηγαίο κώδικα που θα τροφοδοτήσει τον javac.

Το AspectJ Development Environment (AJDE) συμπληρώνει τα IDEs που αναφέρθηκαν, δίνοντας στον προγραμματιστή τη δυνατότητα να κινείται μέσα στη δομή με όψεις του προγράμματος. Παρέχεται έτσι μια καθαρή εικόνα του ποιες όψεις επηρεάζουν ποιες κλάσεις.

Δομές της γλώσσας AspectJ. Οι πιο σημαντικές δομές που προστίθενται είναι:

- Τα *Join points* είναι συγκεκριμένα σημεία στην εκτέλεση ενός προγράμματος, όπως η κλήση ή η εκτέλεση ενός χειριστή εξαιρέσεων, ενός δημιουργού ή μιας μεθόδου.
- Τα *Pointcuts* ορίζονται σε κλάσεις ή όψεις και χρησιμεύουν στο να περιγράφουν join points. Ο ορισμός των join points αυτών γίνεται είτε με ρητή αναφορά στα ονόματα των μεθόδων (name-based crosscutting (call void Point.setX(int))) είτε με περιγραφή των ιδιοτήτων των μεθόδων (property-based crosscutting (call public * Display.*(..))) [14].
- Τα *Advices* είναι κώδικας που εκτελείται πριν, μετά ή πριν και μετά την εκτέλεση ενός join point.
- Τα *Introductions* είναι τα μόνα που επιδρούν στατικά στη δομή του προγράμματος και συγκεκριμένα στόχο έχουν να μεταβάλλουν τα μέλη και τις σχέσεις των κλάσεων.
- Τα *Aspects* μοιάζουν με τις κλάσεις και στόχο έχουν να ενθυλακώνουν συμφέροντα που τέμνουν κλάσεις.

Παράδειγμα Aspect-Oriented Programming με το AspectJ. Παρουσιάζουμε ένα από τα παραδείγματα χρήσης του AspectJ. Χρησιμοποιήσαμε το linux-jdk1.3.1 σε FreeBSD 4.2-STABLE, το ajc, το AspectJ Browser και το AspectJ-mode για Emacs που παρέχονται από το AspectJ 1.0. Το παράδειγμα εκτελεί τη λειτουργία της καταγραφής (tracing). Θα δείξουμε πως αυτή επιτυγχάνεται χωρίς και με τη χρήση όψεων.

Περιγραφή της εφαρμογής. Η εφαρμογή μας περιέχει 4 κλάσεις και αφορά γεωμετρικά σχήματα (shapes). (Δε συμπεριλάβαμε διάγραμμα κλάσεων, μιας και υπήρχε ασάφεια σχετικά με την αναπαράσταση των όψεων σε αυτό.)

Η κλάση TwoDShape βρίσκεται στην κορυφή της ιεραρχίας:

```
public abstract class TwoDShape {
    protected double x, y;
```

```
    protected TwoDShape(double x, double y) {
        this.x = x; this.y = y;
    }
    public double getX() { return x; }
    public double getY() { return y; }
    public double distance(TwoDShape s) {
        double dx = Math.abs(s.getX() - x);
        double dy = Math.abs(s.getY() - y);
        return Math.sqrt(dx*dx + dy*dy);
    }
    public abstract double perimeter();
    public abstract double area();
    public String toString() {
        return (" @ (" + String.valueOf(x) + ", " + String.valueOf(y) + ") ");
    }
}
```

Η κλάση TwoDShape έχει δύο υποκλάσεις, τις Circle και Square:

```
public class Circle extends TwoDShape {
    protected double r;
    public Circle(double x, double y, double r) {
        super(x, y); this.r = r;
    }
    public Circle(double x, double y) { this(  x,   y, 1.0); }
    public Circle(double r)           { this(0.0, 0.0,   r); }
    public Circle()                   { this(0.0, 0.0, 1.0); }
    public double perimeter() {
        return 2 * Math.PI * r;
    }
    public double area() {
        return Math.PI * r*r;
    }
    public String toString() {
        return ("Circle radius = " + String.valueOf(r) + super.toString());
    }
}
```

```
public class Square extends TwoDShape {
    protected double s;    // side
    public Square(double x, double y, double s) {
        super(x, y); this.s = s;
    }
    public Square(double x, double y) { this(  x,   y, 1.0); }
    public Square(double s)           { this(0.0, 0.0,   s); }
    public Square()                   { this(0.0, 0.0, 1.0); }
    public double perimeter() {
        return 4 * s;
    }
    public double area() {
        return s*s;
    }
    public String toString() {
        return ("Square side = " + String.valueOf(s) + super.toString());
    }
}
```

Χρησιμοποιούμε τέλος την κλάση ExampleMain που φτιάχνει στιγμιότυπα και καλεί μεθόδους:

```
public class ExampleMain {
    public static void main(String[] args) {
        Circle c1 = new Circle(3.0, 3.0, 2.0);
        Circle c2 = new Circle(4.0);

        Square s1 = new Square(1.0, 2.0);

        System.out.println("c1.perimeter() = " + c1.perimeter());
        System.out.println("c1.area() = " + c1.area());

        System.out.println("s1.perimeter() = " + s1.perimeter());
        System.out.println("s1.area() = " + s1.area());

        System.out.println("c2.distance(c1) = " + c2.distance(c1));
        System.out.println("s1.distance(c1) = " + s1.distance(c1));
```

```
        System.out.println("s1.toString(): " + s1.toString());
    }
}
```

Μεταγλωττίζουμε τις κλάσεις, είτε με τον ajc είτε με τον javac, αφού ακόμη δεν έχουμε χρησιμοποιήσεις όψεις:

```
ajc -argfile tracing/notrace.lst
```

Εκτελούμε το πρόγραμμα:

```
java tracing.ExampleMain
```

και παίρνουμε:

```
c1.perimeter() = 12.566370614359172
c1.area() = 12.566370614359172
s1.perimeter() = 4.0
s1.area() = 1.0
c2.distance(c1) = 4.242640687119285
s1.distance(c1) = 2.23606797749979
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

Tracing χωρίς όψεις. Μπορούμε χωρίς να χρησιμοποιήσουμε όψεις να ενσωματώσουμε δυνατότητες καταγραφής στην εφαρμογή μας, γράφοντας μια κλάση Trace:

```
import java.io.PrintStream;

public class Trace {
    /**
     * There are 3 trace levels (values of TRACELEVEL):
     * 0 - No messages are printed
```

```
 * 1 - Trace messages are printed, but there is no indentation
 *     according to the call stack
 * 2 - Trace messages are printed, and they are indented
 *     according to the call stack
 */
public static int TRACELEVEL = 0;
protected static PrintStream stream = null;
protected static int callDepth = 0;

/**
 * Initialization.
 */
public static void initStream(PrintStream s) {
    stream = s;
}

/**
 * Prints an "entering" message. It is intended to be called in the
 * beginning of the blocks to be traced.
 */
public static void traceEntry(String str) {
    if (TRACELEVEL == 0) return;
    if (TRACELEVEL == 2) callDepth++;
    printEntering(str);
}

/**
 * Prints an "exiting" message. It is intended to be called in the
 * end of the blocks to be traced.
 */
public static void traceExit(String str) {
    if (TRACELEVEL == 0) return;
    printExiting(str);
    if (TRACELEVEL == 2) callDepth--;
}

private static void printEntering(String str) {
    printIndent();
    stream.println("--> " + str);
}
```

```
    private static void printExiting(String str) {
        printIndent();
        stream.println("<-- " + str);
    }

    private static void printIndent() {
        for (int i = 0; i < callDepth; i++)
            stream.print("  ");
    }
}
```

Σε όλες τις μεθόδους και δημιουργούς που θα θέλαμε να κάνουμε trace θα έπρεπε να ενσωματώσουμε κλήσεις στις traceEntry και traceExit και να αρχικοποιούμε το TRACELEVEL και το stream, καλώντας την initStream. Προφανώς ελοχεύει ο κίνδυνος να ξεχάσουμε κάποια κλήση, ενώ μία αλλαγή στη διεπαφή της Trace συνεπάγεται πολλές αλλαγές στον κώδικα σε όλα τα σημεία κλήσης. Επιπλέον ο προγραμματιστής που επιθυμεί να καλέσει μεθόδους της Trace πρέπει όχι μόνο να γνωρίζει πότε πρέπει να κάνει κάτι τέτοιο, αλλά και τη διεπαφή τους.

Tracing με όψεις. Μπορούμε όμως να ενσωματώσουμε δυνατότητες καταγραφής στην εφαρμογή μας χρησιμοποιώντας όψεις. Γράφουμε το ακόλουθο aspect (TraceMyClasses):

```
aspect TraceMyClasses {
    pointcut myClass(): within(TwoDShape) || within(Circle) || within(Square);
    pointcut myConstructor(): myClass() && execution(new(..));
    pointcut myMethod(): myClass() && execution(* *(..));

    before (): myConstructor() {
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myConstructor() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }

    before (): myMethod() {
```

```
        Trace.traceEntry("" + thisJoinPointStaticPart.getSignature());
    }
    after(): myMethod() {
        Trace.traceExit("" + thisJoinPointStaticPart.getSignature());
    }
}
```

Η όψη αυτή εκτελεί τις κλήσεις για tracing στις κατάλληλες στιγμές που ορίζουμε σύμφωνα με τα pointcuts που καθορίζουν τα αντίστοιχα joinpoints. Αυτά είναι η είσοδος και η έξοδος κάθε δημιουργού και κάθε μεθόδου ορισμένων μέσα στην ιεραρχία των σχημάτων. Πριν και μετά από τα σημεία αυτά τυπώνουμε την υπογραφή της μεθόδου που εκτελείται. Μιας και η υπογραφή είναι στατική πληροφορία, μπορούμε να την πάρουμε μέσω του thisJoinPointStaticPart. Προκειμένου να ελέγξουμε το trace aspect του ενσωματώνουμε και μια μέθοδο main:

```
    public static void main(String[] args) {
        Trace.TRACELEVEL = 2;
        Trace.initStream(System.err);
        ExampleMain.main(args);
    }
```

Μεταγλωττίζουμε τις κλάσεις, με τον ajc αφού χρησιμοποιούμε όψεις:

```
ajc -argfile tracing/tracev1.lst
```

Εκτελούμε το πρόγραμμα:

```
java tracing.version1.TraceMyClasses
```

και παίρνουμε:

```
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Circle(double, double, double)
  <-- tracing.Circle(double, double, double)
  --> tracing.Circle(double)
  <-- tracing.Circle(double)
  --> tracing.TwoDShape(double, double)
  <-- tracing.TwoDShape(double, double)
  --> tracing.Square(double, double, double)
  <-- tracing.Square(double, double, double)
  --> tracing.Square(double, double)
  <-- tracing.Square(double, double)
  --> double tracing.Circle.perimeter()
  <-- double tracing.Circle.perimeter()
c1.perimeter() = 12.566370614359172
  --> double tracing.Circle.area()
  <-- double tracing.Circle.area()
c1.area() = 12.566370614359172
  --> double tracing.Square.perimeter()
  <-- double tracing.Square.perimeter()
s1.perimeter() = 4.0
  --> double tracing.Square.area()
  <-- double tracing.Square.area()
s1.area() = 1.0
  --> double tracing.TwoDShape.distance(TwoDShape)
    --> double tracing.TwoDShape.getX()
    <-- double tracing.TwoDShape.getX()
    --> double tracing.TwoDShape.getY()
    <-- double tracing.TwoDShape.getY()
  <-- double tracing.TwoDShape.distance(TwoDShape)
c2.distance(c1) = 4.242640687119285
  --> double tracing.TwoDShape.distance(TwoDShape)
    --> double tracing.TwoDShape.getX()
    <-- double tracing.TwoDShape.getX()
    --> double tracing.TwoDShape.getY()
    <-- double tracing.TwoDShape.getY()
```

```
  <-- double tracing.TwoDShape.distance(TwoDShape)
s1.distance(c1) = 2.23606797749979
  --> String tracing.Square.toString()
    --> String tracing.TwoDShape.toString()
    <-- String tracing.TwoDShape.toString()
  <-- String tracing.Square.toString()
s1.toString(): Square side = 1.0 @ (1.0, 2.0)
```

Εάν δε συμπεριλάβουμε το TraceMyClasses.java στη μεταγλώττιση η όψη δεν επηρεάζει καθόλου το πρόγραμμα και δεν έχουμε καταγραφή.

Πλεονεκτήματα της υλοποίησης με όψεις. Η υλοποίηση με χρήση του AspectJ προσφέρει αρκετά πλεονεκτήματα [14]:

- Η δομή του διασταυρούμενου συμφέροντος συλλαμβάνεται ρητά και σε ένα σημείο.
- Η εξέλιξη του προγράμματος είναι ευκολότερη, αφού τυχόν αλλαγές γίνονται σε ένα μόνο σημείο του κώδικα.
- Η λειτουργικότητα μπορεί εύκολα να προσθαφαιρείται.
- Η υλοποίηση είναι πιο στιβαρή, αφού για παράδειγμα η προσθήκη και άλλων κλάσεων μπορεί αυτόματα (χωρίς προσθήκη κώδικα κλήσεως μεθόδων) να σημαίνει και την προσθήκη σε αυτές τις αντίστοιχης λειτουργικότητας που προσφέρει η όψη. Αυτό συμβαίνει αν οι νέες κλάσεις υπακούουν στην περιγραφή που έχει δοθεί εντός της όψης.

Προβλήματα που συναντήθηκαν. Δύο ήταν οι βασικές δυσκολίες που συναντήθηκαν και αμφότερες ξεπεράστηκαν. Η πρώτη αφορούσε τη θεωρητική έρευνα. Συγκεκριμένα επειδή το αντικείμενο είναι στην αιχμή της τεχνολογίας δεν υπάρχει ακόμη ελληνική βιβλιογραφία και ελληνική απόδοση των αντίστοιχων όρων και εννοιών. Έτσι αναγκαστήκαμε να κάνουμε τις δικές μας απόπειρες απόδοσης των σύνθετων αυτών συλλήψεων στα ελληνικά.

Η δεύτερη δυσκολία αφορούσε την πρακτική εφαρμογή. Έτσι ενώ η εγκατάσταση του AspectJ κύλισε απρόσκοπτα, αποτύγχανε η μεταγλώττιση προγραμμάτων που χρησιμοποιούσαν τόσο κλήσεις στις κλασσικές βιβλιοθήκες

της Java, όσο και στις βιβλιοθήκες του AspectJ. Η λύση δώθηκε όταν εντοπίστηκε ότι το πρόβλημα βρίσκονταν στον ορισμό του μονοπατιού των κλάσεων βιβλιοθήκης και ορίστηκε το classpath έτσι ώστε να περιλαμβάνει όλες τις απαραίτητες βιβλιοθήκες και της Java και του AspectJ.

Τέλος ως πρόβλημα που συναντήσαμε μπορούμε να σημειώσουμε και την ασάφεια που υπάρχει σχετικά με τον τρόπο αναπαράστασης των όψεων και των υπόλοιπων εννοιών του Προγραμματισμού Προσανατολισμένου σε Όψεις σε διαγράμματα κλάσεων και λοιπούς εποπτικούς μηχανισμούς, οι οποίοι πιθανώς να πρέπει να αναπροσαρμοστούν.

Συμπεράσματα και εμπειρίες που αποκτήθηκαν. Μελετώντας τις βασικές ιδέες του Προγραμματισμού Προσανατολισμένου σε Όψεις, καθώς και πρακτικά παραδείγματα που δείχνουν πως αυτός μπορεί να αποδειχθεί χρήσιμος, πειστήκαμε για την αξία του ως ιδέα και ως εφαρμογή. Επιπλέον είχαμε την ευκαιρία να έλθουμε σε επαφή με προβλήματα που αντιμετωπίζουν όσοι αναπτύσσουν μεγάλα και άρα πολύπλοκα έργα λογισμικού που απαιτούν συστηματική αντιμετώπιση [15], όπως αυτή που όριζει ο ΠΠΟ. Τέλος μας δώθηκε η δυνατότητα να εξοικειωθούμε με το AspectJ, που υλοποιεί όλες τις βασικές έννοιες του Aspect-Oriented Programming και να τις κατανοήσουμε έτσι και στην πράξη, ενώ παράλληλα είδαμε ξανά λεπτομέρειες του Java API.

Αναφορές.

1. Miller, S.K. Aspect-Oriented Programming Takes Aim at Software Complexity. IEEE Computer (Apr. 2001).

2. Elrad, T., Filman R.E. and Bader A. Introduction to Aspect-Oriented Programming. Communications of the ACM 44, 10 (Oct. 2001).

3. Elrad, T., Aksit M., Kiczales G., Lieberherr K. and Ossher H. Discussing Aspects of AOP. Communications of the ACM 44, 10 (Oct. 2001).

4. Holland I. Law of Demeter (General Formulation). http://www.ccs.neu.edu/research/demeter/method/LawOfDemeter/general-formulation.html (1987).

5. Lieberherr K., Orleans D., Ovlinger J. Aspect-Oriented Programming with Adaptive Methods. Communications of the ACM 44, 10 (Oct. 2001).

6. Ossher H. and Tarr P. Using Multidimensional Separation of Concerns to (Re)Shape Evolving Software. Communications of the ACM 44, 10 (Oct. 2001).

7. Bergmans L. and Aksit M. Composing Crosscutting Concerns Using Composition Filters. Communications of the ACM 44, 10 (Oct. 2001).

8. Coaddy Y., Kiczales G., Feeley M, Hutchinson N. and Ong J.S. Structuring Operating System Aspects. Communications of the ACM 44, 10 (Oct. 2001).

9. Netinant P., Elrad T. and Fayad M. A Layered Approach to Building Open Aspect-Oriented Systems. Communications of the ACM 44, 10 (Oct. 2001).

10. XEROX Corporation. Software Design Area. http://www.parc.xerox.com/csl/groups/sda (2000).

11. AOSD Steering Committee. Aspect-Oriented Software Development. http://aosd.net (2002).

12. Tristram C. Untangling Code. Technology Review (Jan./Feb. 2001).

13. XEROX Corporation. AspectJ - Aspect-Oriented Programming for Java. http://aspectj.org (2002).

14. Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J. and Griswold W.G. Getting Started With AspectJ. Communications of the ACM 44, 10 (Oct. 2001).

15. Θραμπουλίδης Κ. Ανάλυση-Σχεδιασμός Συστημάτων Λογισμικού (2001).

Συνημμένα. Σε δισκέττα επισυνάπτονται το παρόν κείμενο σε ηλεκτρονική μορφή, όλοι οι πηγαίοι κώδικες και τα αντίστοιχα εκτελέσιμα.