ΑΝΩΤΑΤΟ ΤΕΧΝΟΛΟΓΙΚΟ ΕΚΠΑΙΔΕΥΤΙΚΟ ΙΔΡΥΜΑ ΔΥΤΙΚΗΣ ΕΛΛΑΔΟΣ

ΣΧΟΛΗ ΔΙΟΙΚΗΣΗΣ ΚΑΙ ΟΙΚΟΝΟΜΙΑΣ

ΤΜΗΜΑ:

ΔΙΟΙΚΗΣΗ ΕΠΙΧΕΙΡΗΣΕΩΝ

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

''ΘΕΜΑΤΟΣΤΡΕΦΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ''

ΒΕΛΛΗ ΜΑΡΙΑ

ΚΟΥΤΡΟΥΜΑΝΟΥ ΙΩΑΝΝΑ

Εισηγητής:Επίκ.Καθηγητής Μπακάλης Αριστείδης

Πάτρα 9 Απριλίου 2014

Πρόλογος

Ένα μεγάλο μέρος προγραμματιστών έχουν γαλουχηθεί στην ανάπτυξη συστημάτων λογισμικού με τη χρήση τεχνικών αντικειμενοστρεφούς προγραμματισμού. Η μέθοδος αυτή είναι η πλέον διαδεδομένη με κύριο χαρακτηριστικό τον κατακερματισμό ενός προβλήματος σε αντικείμενα που χαρακτηρίζονται από ιδιότητες (μέθοδους) και δεδομένα (μεταβλητές).

Παρά το γεγονός ότι ο αντικειμενοστρεφής προγραμματισμός έχει μεγάλη επιτυχία στη διαμόρφωση και υλοποίηση πολύπλοκων συστημάτων λογισμικού, έχει και τα προβλήματά του. Η πρακτική εμπειρία με μεγάλα έργα έχει δείξει ότι οι προγραμματιστές ενδέχεται να αντιμετωπίσουν προβλήματα με τη διατήρηση του κώδικα τους, δεδομένου ότι όσο μεγαλύτερο το λογισμικό που υλοποιείται τόσο και πιο δύσκολος γίνεται ο ξεκάθαρος διαχωρισμός του έργου σε ενότητες (αντικείμενα), πράγμα και το οποίο αποτελεί τη βάση του **αντικειμενοστρεφούς προγραμματισμού**. Για παράδειγμα, μια μικρή αλλαγή σε μία επαναχρησιμοποιούμενη ενότητα μπορεί τελικά να προκαλέσει πολλές αλλαγές σε άλλες, ανεξάρτητες, ενότητες του κώδικα.

Τέτοιου είδους προβλήματα και πολλές άλλες ανησυχίες έρχεται να επιλύσει μία διαφορετική τεχνική, ο θεματοστρεφής προγραμματισμός. Αυτό που προσπαθεί να κάνει ο θεματοστρεφής προγραμματισμός είναι να προωθήσει τα επιθυμητά χαρακτηριστικά του αντικειμενοστρεφούς προγραμματισμού που είναι δύσκολο να εφαρμοστούν στις τρέχουσες τεχνολογίες.

Στην παρούσα εργασία θα μελετήσουμε διεξοδικά την νέα αυτή προσέγγιση προγραμματισμού, θα δούμε τα κύρια χαρακτηριστικά της, ενώ θα ερευνήσουμε τι πλεονεκτήματα προσφέρει και τι μειονεκτήματα τη χαρακτηρίζουν.

Περιεχόμενα

Βιβλιογραφία

1. Κλ. Θραμπουλίδης, *Διαδικαστικός Προγραμματισμός – C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα, 2002.

2. *Java 2: A beginner's Guide*, Schildt

3. *Η βίβλος της JAVA 2*, Aaron Walsh, Justin Couch, Daniel Steinberg, ΓΚΙΟΥΡΔΑΣ

4. *Thinking in Java*, Bruce Eckel, Prentish Hall

5. *Aspect-Oriented Programming*, Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

6. *An Overview of AspectJ***,** Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, in Proceedings of the 15th European Conference on Object-Oriented Programming Pages 327-353

7. http://docs.oracle.com/javase/tutorial/

8. *Programming paradigms and an overview of C:* http://cs.anu.edu.au/student/comp3610/lectures/Paradigms.pdf

9. http://en.wikipedia.org/wiki/Object-oriented_programming

10. http://en.wikipedia.org/wiki/Aspect-oriented_programming

11. http://en.wikipedia.org/wiki/Assembly_language

12. http://www.j2ee.gr/2010/06/10/j2ee-%CE%BA%CE%B1%CE%B9-aspect-oriented-programming/

1.Εισαγωγή

Στην παρούσα εργασία μελετάται διεξοδικά το μοντέλο του Θεματοστρεφούς Προγραμματισμού (Aspect Oriented Programming – AOP), το οποίο φαίνεται να αποτελεί τον διάδοχο του πολύ επιτυχημένου μοντέλου του Αντικειμενοστρεφούς Προγραμματισμού (Object Oriented Programming – OOP).

Για να κατατοπισθεί πλήρως ο αναγνώστης και μελετητής της παρούσας εργασίας, γίνεται μια προσπάθεια έτσι ώστε να εισαχθεί σύντομα αλλά περιεκτικά στον κόσμο του προγραμματισμού των Η/Υ. Αυτό επιτυγχάνεται παρουσιάζοντας την ιστορική εξέλιξη των προγραμματιστικών μοντέλων και καταλήγοντας στο μοντέλο του AOP.

Θα πρέπει να σημειωθεί πως όλα τα προγραμματιστικά μοντέλα που παρουσιάζονται στα κεφάλαια της παρούσας εργασίας χρησιμοποιούνται στις μέρες μας από τους προγραμματιστές ανάλογα με την εφαρμογή που αναπτύσσουν. Στεκόμαστε σε αυτό, διότι θα πρέπει να αποτραπεί η παρεξήγηση ότι αφού, για παράδειγμα, το προγραμματιστικό μοντέλο Β διαδέχθηκε το προγραμματιστικό μοντέλο Α, το προγραμματιστικό μοντέλο Α έχασε την ισχύ του και έπαψε να εξελίσσεται. Κάτι τέτοιο, θα πρέπει να είναι ξεκάθαρο πως δεν ισχύει! Κάθε μοντέλο έχει και παίζει τον ρόλο που του αναλογεί, και οδηγούμαστε σε νέα προγραμματιστικά μοντέλα όχι για να αναπτύξουμε τις παλιού τύπου εφαρμογές με νέου τύπου μοντέλα, αλλά για να μπορέσουμε να αναπτύξουμε νέες εφαρμογές αποδοτικότερα και γρηγορότερα, ενώ τις παλιού τύπου εφαρμογές συνεχίζουμε συνήθως να τις αναπτύσσουμε με τα “παλιά” μοντέλα. Κλασσικό παράδειγμα είναι η ανάπτυξη λειτουργικών συστημάτων, όπου παρά το πέρας των χρόνων και την άφιξη νέων προγραμματιστικών μοντέλων και γλωσσών προγραμματισμού, τα περισσότερα από τα υπάρχοντα λειτουργικά συστήματα γράφονται κατά κύριο λόγω χρησιμοποιώντας συμβολική γλώσσα (assembly) και C (διαδικαστικός προγραμματισμός).

Το Κεφάλαιο 2 ξεκινά με την παρουσίαση του μοντέλου του συμβολικού προγραμματισμού (assembly programming), το οποίο και αποτελεί τον πιο πολύπλοκο τρόπο προγραμματισμού ενός ηλεκτρονικού υπολογιστή καθώς κάθε επεξεργαστής έχει το δικό του ξεχωριστό σύνολο εντολών και άρα τη δική του συμβολική γλώσσα. Έπειτα παρουσιάζεται το μοντέλο του προστακτικού προγραμματισμού. Στη συνέχεια στο ίδιο κεφάλαιο, παρουσιάζεται το μοντέλο του διαδικαστικού προγραμματισμού το οποίο βασίζεται στην ανάπτυξη προγραμμάτων με γλώσσες υψηλού επιπέδου όπως η C και η Fortran και τον κατακερματισμό των προγραμμάτων σε συναρτήσεις/διαδικασίες. Εν τέλει, στο Κεφάλαιο 2, γίνεται μια αναλυτική εισαγωγή και στο μοντέλο του αντικειμενοστρεφούς προγραμματισμού και στις αρχές στις οποίες αυτός βασίζεται έτσι

ώστε να παρουσιάσουμε στη συνέχεια τα επόμενα κεφάλαια τα οποία θεωρούν ως δεδομένη τη βασική γνώση των αρχών του αντικειμενοστρεφούς προγραμματισμού.

Στο Κεφάλαιο 3, μελετάται διεξοδικά η γλώσσα προγραμματισμού Java η οποία αποτελεί μια μοντέρνα υλοποίηση του μοντέλου του αντικειμενοστρεφούς προγραμματισμού. Γίνεται ιδιαίτερη αναφορά στη Java καθώς το μοντέλο του θεματοστρεφούς προγραμματισμού βασίζεται σε μεγάλο βαθμό στις εξελίξεις που αφορούν αυτή τη γλώσσα. Επίσης, τα Κεφάλαια 4 και 5 τα οποία αφορούν μεν τον θεματοστρεφή προγραμματισμό, όλος όμως ο κώδικας και τα παραδείγματα που παρατίθενται βασίζονται στη γλώσσα προγραμμτισμού Java.

Στο Κεφάλαιο 4, γίνεται μια προσπάθεια έτσι ώστε ο αναγνώστης να κατανοήσει για ποιους πρακτικούς λόγους το μοντέλο του αντικειμενοστρεφούς προγραμματισμού πλέον θεωρείται παρωχημένο, και για ποιους λόγους έχει προκύψει η ανάγκη να αναπτύσσονται κάποιες εφαρμογές με βάση τις τεχνολογίες του θεματοστρεφούς προγραμματισμού. Στο ίδιο κεφάλαιο παρουσιάζονται όλα τα βασικά χαρακτηριστικά του μοντέλου σε θεωρητική και πρακτική βάση, ενώ παρουσιάζονται και τα πλεονεκτήματα και τα μειονεκτήματα αυτού του τρόπου προσέγγισης.

Στο Κεφάλαιο 5 παρουσιάζεται η γλώσσα προγραμματισμού AspectJ η οποία αποτελεί επέκταση της Java και θεωρείται ως η πιο επιτυχημένη υλοποίηση του μοντέλου του θεματοστρεφούς προγραμματισμού. Στα πλαίσια αυτού του κεφαλαίου, δίνουμε πληροφορίες για τη σύνταξη της AspectJ καθώς και κάποια βασικά προγραμματιστικά παραδείγματα ώστε να φανεί στην πράξη πως διαχωρίζεται η AspectJ (θεματοστρεφής προγραμματισμός) από την Java (αντικειμενοστρεφής προγραμματισμός).

Τέλος, στο Κεφάλαιο 6 γίνεται σύνοψη της εργασίας και παρουσιάζονται τα συμπεράσματα που εξήχθησαν μετά το πέρας της συγγραφής των 5 πρώτων κεφαλαίων. Επίσης, στο Κεφάλαιο 6 παρατίθενται και κάποιες βασικές ιδέες για μελλοντική εργασία η οποία θα μπορούσε να βασισθεί και να επεκτείνει τη δουλειά που έγινε στα πλαίσια της παρούσας εργασίας.

2. Ιστορική Αναδρομή

2.1 Συμβολικός Προγραμματισμός

Το μοντέλο του συμβολικού προγραμματισμού βασίζεται στην συγγραφή προγραμμάτων στη συμβολική γλώσσα (assembly language) του εκάστοτε επεξεργαστή. Η συγγραφή προγραμμάτων σε assembly αποτελεί συνήθως μια ιδιαιτέρως κοπιαστική και χρονοβόρα διαδικασία καθώς απαιτείται από τον προγραμματιστή να είναι πολύ καλά εξοικειωμένος με όλες τις εντολές του επεξεργαστή πάνω στον οποίο δουλεύει ενώ δεν έχει στη διάθεσή του συνήθως καθόλου βρόχους επανάληψης ή/και ελέγχου. Απ' την άλλη βέβαια, πολλές φορές η χρήση της γλώσσας assembly κρίνεται απαραίτητη, όταν πρέπει να γίνουν βελτιστοποιήσεις ταχύτητας, διαχείρισης μνήμης και διαχείρισης ισχύος στην υπό ανάπτυξη εφαρμογή είτε γιατί οι προδιαγραφές της εφαρμογής είναι ιδιαιτέρως υψηλές είτε γιατί οι δυνατότητες του επεξεργαστή είναι περιορισμένες, είτε γιατί η συσκευή στην οποία τρέχει το εκάστοτε πρόγραμμα λειτουργεί με μπαταρία.

Μεγάλα κομμάτια από τους πυρήνες των λειτουργικών συστημάτων που έχουμε ακόμα και σήμερα στους προσωπικούς μας υπολογιστές αλλά και στις φορητές και τις οικιακές μας συσκευές είναι γραμμένα σε γλώσσα assembly. Τα τελευταία χρόνια δε, που έχει γιγαντωθεί η αγορά ενσωματωμένων συστημάτων (έξυπνες οικιακές συσκευές, κινητά τηλέφωνα, φορητοί υπολογιστές, υπολογιστές χειρός κλπ) η γλώσσα προγραμματισμού assembly για πολλές αρχιτεκτονικές επεξεργαστών είναι μονόδρομος.

Ο προγραμματισμός με συμβολική γλώσσα, θεωρείται πως είναι το πιο κοντινό μοντέλο προγραμματισμού στο υλικό του υπολογιστή, καθώς ο προγραμματιστής έχει πλήρη αίσθηση των χρησιμοποιούμενων μερών του επεξεργαστή σε κάθε χρονική στιγμή. Σημειώνεται πως αυτό το χαρακτηριστικό είναι που αυξάνει κατά πολύ την πολυπλοκότητα των προγραμμάτων assembly, αλλά είναι και αυτό που δίνει τη δυνατότητα στον προγραμματιστή να αναπτύξει βέλτιστα προγράμματα ως προς συγκεκριμένες απαιτήσεις.

Ένα προγραμματιστικό παράδειγμα στη γλώσσα assembly του επεξεργαστή MIPS ο οποίος είναι ένας πολύ διαδεδομένος επεξεργαστής σε εφαρμογές κινητής τηλεφωνίας φάινεται παρακάτω:

```
    lw      $t0, 4($gp)       # fetch N
   mult     $t0, $t0, $t0     # N*N
   lw       $t1, 4($gp)       # fetch N
   ori      $t2, $zero, 3     # 3
   mult     $t1, $t1, $t2     # 3*N
   add      $t2, $t0, $t1     # N*N + 3*N
   sw       $t2, 0($gp)       # i = ...
```

Αν θέλαμε να γράψουμε την αντίστοιχη εφαρμογή σε κάποια γλώσσα όπως η C ή η C++ που μελετούνται στη συνέχεια του κεφαλαίου, θα αρκούσε να γράφαμε:

```
i = N*N + 3*N
```

Είναι εμφανές λοιπόν πως κάτι που στις γλώσσες υψηλού επιπέδου είναι προφανές, στη συμβολική γλώσσα αποκτά ιδιαίτερη πολυπλοκότητα.

2.2 Προστακτικός Προγραμματισμός

Στην επιστήμη της πληροφορικής, ο προστακτικός προγραμματισμός είναι ένα παράδειγμα προγραμματισμού που περιγράφει τον υπολογισμό σε όρους δηλώσεων που αλλάζουν την κατάσταση ενός προγράμματος. Με τον ίδιο τρόπο που η προστακτική διάθεση στις φυσικές γλώσσες εκφράζει εντολές που θα πρέπει να εκτελεστούν, τα προστακτικά προγράμματα ορίζουν ακολουθίες εντολών που θα πρέπει να εκτελέσει ο υπολογιστής.

Ο όρος αυτός χρησιμοποιείται σε αντίθεση με τον «δηλωτικό» προγραμματισμό, οποίος δηλώνει *τι* θα πρέπει να επιτύχει το πρόγραμμα χωρίς να περιγράφει ακριβώς πώς να το κάνει, δίνοντας ακολουθίες ενεργειών που θα πρέπει να εκτελεστούν. Ο «συναρτησιακός» και ο «λογικός» προγραμματισμός είναι παραδείγματα αυτής της προσέγγισης.

Για παράδειγμα, η δήλωση ανάθεσης

```
x:=5
```

είναι μια εντολή προς τον υπολογιστή να αποθηκεύεσει την τιμή 5 σε μια συγκεκριμένη θέση. Οι γλώσσες προγραμματισμού περιέχουν επίσης κατασκευές δήλωσης, όπως η δήλωση συνάρτησης

```
function f(int x) { return x+1; }
```

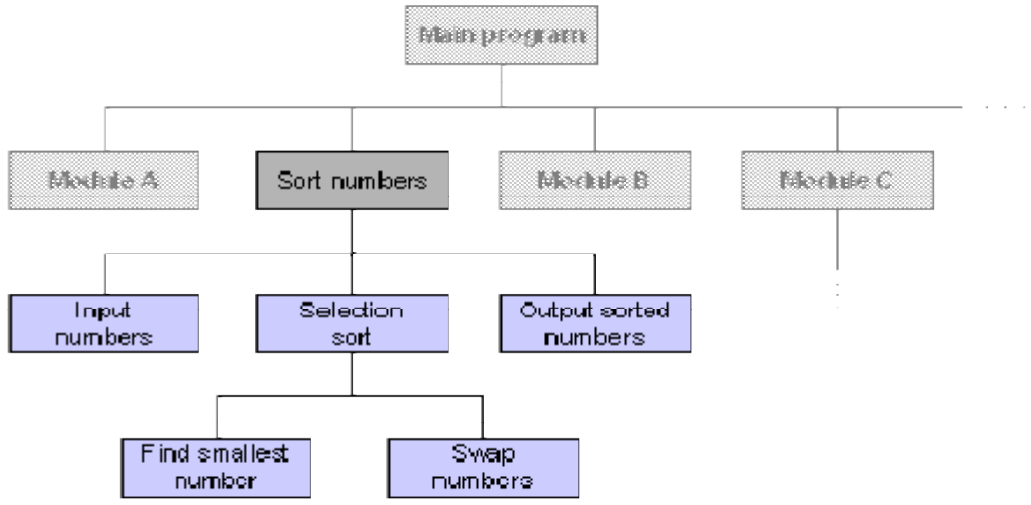
η οποία δηλώνει μία πραγματικότητα.

2.2.1 Διαδικαστικός προγραμματισμός

Πιθανώς το πιο γνωστό παράδειγμα ανάπτυξης λογισμικού είναι το διαδικαστικό παράδειγμα. Ο Brookshear λέει ότι το διαδικαστικό πρότυπο «αντιπροσωπεύει την παραδοσιακή προσέγγιση στη διαδικασία του προγραμματισμού. Πράγματι, ο διαδικαστικός προγραμματισμός είναι εκείνος στον οποίο βασίζεται ο κύκλος «φορτώνω-αποκωδικοποιώ-εκτελώ» μιας εντολής που πραγματοποιεί η CPU. Όπως υποδηλώνει το

όνομα του, ο διαδικαστικός προγραμματισμός ορίζει τη διαδικασία του προγραμματισμού να είναι η ανάπτυξη μιας ακολουθίας εντολών που, όταν ακολουθείται, χειρίζεται τα δεδομένα ώστε να παραγθεί το επιθυμητό αποτέλεσμα». Ας παρατηρήσουμε ότι το διαδικαστικό μοντέλο προσεγγίζει την ανάπτυξη λογισμικού με τρόπο που να ταιριάζει στο υποκείμενο υλικό ενός τυπικού προσωπικού υπολογιστή. Ο επεξεργαστής ενός υπολογιστή λειτουργεί λαμβάνοντας μια απλή εντολή, ερμηνεύοντάς τη, και τελικά εκτελώντας τη. Το διαδικαστικό μοντέλο ταιριάζει με αυτή την αρχιτεκτονική καθοδηγώντας τη ανάπτυξη του λογισμικού έτσι ώστε να έχει ακολουθιακή λογική. Ως αποτέλεσμα, τα διαδικαστικά προγράμματα μπορούν να σχεδιαστούν από την αρχή έως το τέλος, έτσι ώστε η όλη λογική του προγράμματος να είναι μια σειρά από οδηγίες. Το ταίρισμα μεταξύ του υλικού και του λογισμικού οδηγεί συνήθως σε προγράμματα με υψηλή απόδοση εκτέλεσης.

Για να κατανοήσουμε το διαδικαστικό παράδειγμα, ας σκεφτούμε το εξής πρόβλημα σχεδιασμού: Ας υποθέσουμε ότι ένας σχεδιαστής λογισμικού θέλει να δημιουργήσει μια σύντομη ρουτίνα που δέχεται ως είσοδο μια λίστα με αριθμούς και δίνει ως έξοδο τους αριθμούς σε ταξινομημένη σειρά. Σύμφωνα με το διαδικαστικό παράδειγμα, πώς πρέπει να αναπτυχθεί αυτό το σχέδιο; Πρώτον, το πρόβλημα διασπάται σε διάφορες δευτερεύουσες εργασίες που απαιτούνται για την ταξινόμηση: είσοδο, ταξινόμηση, έξοδο. Στη συνέχεια, κάθε μία από αυτές τις εργασίες διαιρείται περαιτέρω σε σχετικές δευτερεύουσες εργασίες.



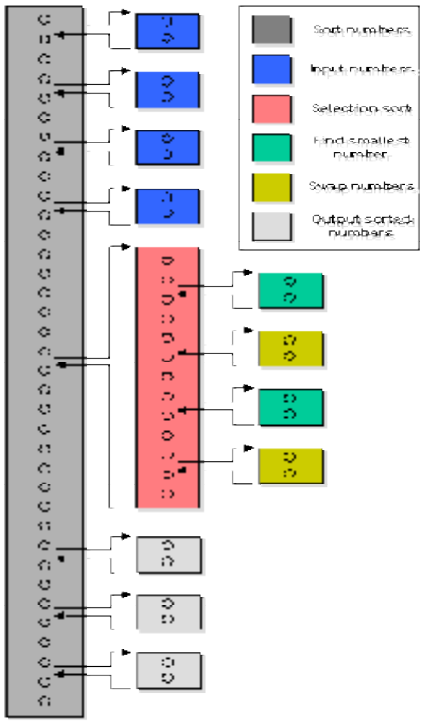
Εικόνα 2.1 Ένας ολοκληρωμένος σχεδιασμός για τη ρουτίνα ταξινόμησης

Για την ταξινόμηση, κάποιες σχετικές εργασίες αναλαμβάνουν να βρουν το μικρότερο αριθμό και να κάνουν την εναλλαγή των αριθμών. Μετά από αυτή τη διαδικασία της αποσύνθεσης εργασιών, το σύνολο του σχεδιασμού χωρίζεται σε λογικές μονάδες οι οποίες αντιπροσωπεύουν λειτουργικές οντότητες και υπορουτίνες. Το διάγραμμα της Εικόνα 2.1 Ένας ολοκληρωμένος σχεδιασμός για τη ρουτίνα ταξινόμησηςπαρουσιάζει ένα ολοκληρωμένο σχέδιο για το επιθυμητό είδος ρουτίνας. Οι λειτουργικές οντότητες παρουσιάζονται ως γκρίζα κουτιά, ενώ τα υποπρογράμματα με μπλε κουτιά. Το διάγραμμα απεικονίζει επίσης ένα ευρύτερο πλαίσιο όπου η σύντομη ρουτίνα ενσωματώνεται. Αυτό το πλαίσιο θα μπορούσε να είναι ένα πρόγραμμα υπολογιστικών φύλλων, μια βάση δεδομένων ή οποιοδήποτε άλλο προϊόν λογισμικού που χρησιμοποιεί ταξινόμηση.

Ένα από τα κύρια χαρακτηριστικά του λογισμικού σχεδιασμένα με τη διαδικαστικό προγραμματισμό είναι η ακολουθιακή λογική, δηλαδή, η ροή της λογικής του προγράμματος να μπορεί να ακολουθηθεί από την αρχή μέχρι το τέλος. Για τη ρουτίνα ταξινόμησης που περιγράφεται παραπάνω, η λογική μπορεί να εντοπιστεί μέσω των διαφόρων κλήσεων προς τις υπορουτίνες.

Το διάγραμμα της

Εικόνα 2.2 αντιπροσωπεύει τη ροή του ελέγχου για ένα *Selection Sort* όταν ταξινομούνται τρεις αριθμοί. Κάθε ένα από τα έγχρωμα πλαίσια αντιστοιχεί σε μια λειτουργική οντότητα ή υποπρόγραμμα. Η κύρια μονάδα, *Sort numbers*, διατρέχει όλο το μήκος του διαγράμματος και είναι υπεύθυνη για την κλήση όλων των άλλων υποπρογραμμάτων. Οι κύκλοι μέσα στα πλαίσια αντιπροσωπεύουν τις εκτελέσιμες δηλώσεις της μονάδας ή του υποπρογράμματος. Τα πρώτα τέσσερα μπλε κουτιά είναι οι κλήσεις προς το υποπρόγραμμα *Input numbers* και αντιστοιχούν στην είσοδο του μεγέθους ταξινόμησης (3) και τους αριθμούς που πρέπει να ταξινομηθούν (3, 1, 5). Κάτω από αυτά, το κόκκινο κουτί αντιπροσωπεύει το υποπρόγραμμα Selection Sort που πραγματοποιεί δύο κλήσεις στα *Find smallest number* και *Swap numbers* για να ταξινομήσουν τους αριθμούς. Τα τρία τελευταία γκρίζα κουτιά αντιπροσωπεύουν το υποπρόγραμμα *Output sorted numbers* που καλείται τρεις φορές για την έξοδο των αριθμών με τη σειρά.



Εικόνα 2.2 **Η ροή ελέγχου για** Selection Sort **όταν ταξινομούνται τρεις αριθμοί**

Παρατηρήστε ότι η σχέση μεταξύ των *Sort numbers* και *Input numbers* είναι ακολουθιακή. Όταν η *Sort numbers* καλεί την *Input numbers*, η ροή του ελέγχου αλλάζει αμέσως στο υποπρόγραμμα *Input numbers*. Ο κώδικας της *Sort numbers* σταματάει να εκτελείται μέχρι η κλήση ολοκληρωθεί (δηλαδή, όταν τελειώσει η εκτέλεση της *Input numbers*). Όταν συμβεί αυτό, η ροή του ελέγχου επιστρέφει στη *Sort numbers* στην ακριβή θέση όπου είχε γίνει η κλήση του υποπρογράμματος. Αυτή η σειριακή σχέση μεταξύ της καλούντας οντότητας και του καλούμενου υποπρογράμματος είναι πάντα η ίδια στο διαδικαστικό παράδειγμα. Ως εκ τούτου, το διάγραμμα αντιπροσωπεύει κάθε κλήση με ένα ζευγάρι από βέλη που προέρχονται και τερματίζουν στο ίδιο σημείο. Ακολουθώντας αυτά τα βέλη μέσα από το διάγραμμα, είναι δυνατό να εντοπίσει την ακριβή πορεία εκτέλεσης της ρουτίνας Selection Sort.

Κάποιες από τις πιο αντιπροσωπευτικές γλώσσες προγραμματισμού ακολουθιακής λογικής είναι οι Pascal, Basic, C.

2.2.2 **Η γλώσσα προγραμματισμού** C

Η C επινοήθηκε το 1972 από τον Dennis Ritchie, στα εργαστήρια Bell. Δημιουργήθηκε για να εξυπηρετήσει το λειτουργικό σύστημα Unix, το οποίο έως τότε ήταν γραμμένο σε assembly. Ο δημιουργός του Unix, Ken Thompson, φίλος και
συνεργάτης του Ritchie, είχε δημιουργήσει την πρόγονο της C, τη γλώσσα B. Και οι δύο γλώσσες έχουν κοινή καταγωγή από τη γλώσσα BCPL, η οποία είχε αναπτυχθεί από τον Martin Richards κατά το πέρασμά του απο το Τεχνολογικό Ινστιτούτο της Μασσαχουσέττης (MIT) το 1967, στηριζόμενη στη γλώσσα CPL (Cambridge Programming Language) του πανεπιστημίου του Cambridge. Και οι τρεις γλώσσες κατασκευάστηκαν στα πλαίσια του προγράμματος MAC και του απόγονού του Multics, τα οποία στόχευαν στην κατανομή των πόρων των υπολογιστών σε πολλούς χρήστες. Τα δύο αυτά προγράμματα, στα οποία συνέπραξαν το MIT, η General Electric και τα εργαστήρια Bell, απετέλεσαν τη θερμοκοιτίδα πολλών προγραμμάτων λογισμικού, που κυριαρχούν από τη δεκαετία του 1960 έως σήμερα. Για ενδελεχή μελέτη της ιστορίας του λογισμικού, ο αναγνώστης μπορεί να ανατρέξει στην αναφορά [12].

Η C, ούσα ευέλικτη και αποδοτική χρησιμοποιήθηκε αρχικά για τον προγραμματισμό συστημάτων στο Unix. Το 1974 εμφανίσθηκε από τον Brian Kernighan το πρώτο γραπτό κείμενο για τη γλώσσα, υπό τον τίτλο “Programming in C: A Tutorial”. Το 1977 έγινε η πρώτη επίσημη τεκμηρίωση της γλώσσας με το βιβλίο “The C Programming Language” από τους Kernighan και Ritchie. Το βιβλίο αυτό απετέλεσε το «ευαγγέλιο» των προγραμματιστών της C, αποκαλούμενο «Λευκή Βίβλος» ή «πρότυπο K&R».

Με την πάροδο του χρόνου η γλώσσα C άρχισε να χρησιμοποιείται και σε άλλα πεδία εφαρμογών, πέραν του προγραμματισμού συστημάτων. Η εμφάνιση των μεταγλωττιστών της γλώσσας στο MS–DOS και ο μεγάλος αριθμός προγραμμάτων βιβλιοθήκης που κατασκευάστηκαν, οδήγησαν τη γλώσσα στο απόγειό της στα τέλη της δεκαετίας του 1980. Βέβαια η γλώσσα γνώρισε πολλές αλλαγές, οδηγούμενη τελικά στην επονομαζόμενη **ANSI έκδοση**. Η τελευταία ενημέρωση της γλώσσας έγινε το 1999.

Πλεονεκτήματα της C

- Είναι σχετικά μικρή και εύκολη στην εκμάθηση υποστηρίζοντας top down/modular σχεδιασμό και δομημένο προγραμματισμό

- Υπάρχει μεγάλη εγκατεστημένη βάση εφαρμογών που αναπτύχθηκαν με τη γλώσσα αυτή.

- Μπορεί να διαχειριστεί: δυαδικά ψηφία (bits), ψηφιολέξεις (bytes), συμβολοσειρές (words), δείκτες (pointers).

- Μπορεί να χρησιμοποιηθεί για χαμηλού επιπέδου προγραμματισμό επιτρέποντας άμεση πρόσβαση στους πόρους του υπολογιστή. Συνεπώς είναι κατάλληλη γλώσσα για ανάπτυξη προγραμμάτων συστήματος (systems programs) όπως είναι: λειτουργικά συστήματα (operating systems), διερμηνευτές (interpreters), συντάκτες (editors), συμβολομεταφραστές (assemblers), μεταγλωττιστές (compilers), διαχειριστές βάσεων δεδομένων (database managers).

Η δομή ενός προγράμματος σε C

Ένα πρόγραμμα γραμμένο σε γλώσσα C αποτελείται από τα εξής βασικά στοιχεία: τις εντολές του προεπεξεργαστή, τις δηλώσεις των συναρτήσεων, τις δηλώσεις των μεταβλητών, το κυρίως πρόγραμμα και τον ορισμό των συναρτήσεων. Το κυρίως πρόγραμμα με τη σειρά του, το κυρίως πρόγραμμα είναι αυτό που δέχεται την είσοδο των δεδομένων, πραγματοποιεί την επεξεργασία τους καλώντας άλλες συναρτήσεις και δίνει στην έξοδο τα αποτελέσματα. Η διαδικασία αυτή αποτελείται από «προτάσεις της γλώσσας».

Ένα απλό παράδειγμα φαίνεται παρακάτω.

```
#include <stdio.h>          ⟹   εντολή προεπεξεργαστή
int k=0; float f; int n=5;  ⟹   δηλώσεις μεταβλητών
int a;
int main ( )                ⟹   κυρίως πρόγραμμα
{
     int i;
     /* this is a comment */

      for (i=1;  i<n;  i++)
      {   scanf("%d", &a);  ⟹   εισαγωγή δεδομένων
          k=k+a;            ⟹   επεξεργασία δεδομένων
      }
      f=k/n;
      printf("%f/n",f);     ⟹   έξοδος δεδομένων
}
```

Όπως φαίνεται από το παραπάνω παράδειγμα, όλες οι εντολές γράφονται η μία μετά την άλλη και η εκτέλεσή τους θα είναι ακολουθιακή. Το κυρίως πρόγραμμα είναι η βασική συνάρτηση της εφαρμογής, η οποία μπορεί να καλέσει άλλες εξωτερικές συναρτήσεις. Στο παράδειγμα, η εκτέλεση της main() θα σταματήσει έτσι ώστε να εκτελεστεί η συνάρτηση scanf(). Με το που τελειώσει η εκτέλεση της scanf() συνάρτησης θα συνεχιστεί η εκτέλεση των εντολών της main() από εκεί που είχαν σταματήσει. Το ίδιο συμβαίνει και με τη συνάρτηση printf().

2.3 Αντικειμενοστρεφής προγραμματισμός

Στην επιστήμη υπολογιστών **αντικειμενοστρεφή προγραμματισμό** (object-oriented programming), ή ΑΠ, ονομάζουμε ένα προγραμματιστικό υπόδειγμα το οποίο εμφανίστηκε στα τέλη της δεκαετίας του 1960 και καθιερώθηκε κατά τη δεκαετία του 1990, αντικαθιστώντας σε μεγάλο βαθμό το παραδοσιακό υπόδειγμα του δομημένου προγραμματισμού. Πρόκειται για μία μεθοδολογία ανάπτυξης προγραμμάτων, υποστηριζόμενη από κατάλληλες γλώσσες προγραμματισμού, όπου ο χειρισμός σχετιζόμενων δεδομένων και των διαδικασιών που επενεργούν σε αυτά γίνεται από κοινού, μέσω μίας δομής δεδομένων που τα περιβάλλει ως αυτόνομη οντότητα με ταυτότητα και δικά της χαρακτηριστικά. Αυτή η δομή δεδομένων καλείται *αντικείμενο* και αποτελεί πραγματικό στιγμιότυπο στη μνήμη ενός σύνθετου, και πιθανώς οριζόμενου από τον χρήστη, τύπου δεδομένων ονόματι *κλάση*. Η κλάση προδιαγράφει τόσο δεδομένα όσο και τις διαδικασίες οι οποίες επιδρούν επάνω τους· αυτή υπήρξε η πρωταρχική καινοτομία του ΑΠ.

Έτσι μπορεί να οριστεί μία προδιαγραφή δομής αποθήκευσης (π.χ. μία κλάση "τηλεόραση") η οποία να περιέχει τόσο ιδιότητες (π.χ. μία μεταβλητή "τρέχον κανάλι") όσο και πράξεις ή χειρισμούς επί αυτών των ιδιοτήτων (π.χ. μία διαδικασία "άνοιγμα της τηλεόρασης"). Στο εν λόγω παράδειγμα κάθε υλική τηλεόραση (κάθε αντικείμενο αποθηκευμένο πραγματικά στη μνήμη) αναπαρίσταται ως ξεχωριστό, "φυσικό" στιγμιότυπο αυτής της πρότυπης, ιδεατής κλάσης. Επομένως μόνο τα αντικείμενα καταλαμβάνουν χώρο στη μνήμη του υπολογιστή ενώ οι κλάσεις αποτελούν απλώς "καλούπια". Οι αιτίες που ώθησαν στην ανάπτυξη του ΑΠ ήταν οι ίδιες με αυτές που οδήγησαν στην ανάπτυξη του δομημένου προγραμματισμού (ευκολία συντήρησης, οργάνωσης, χειρισμού και επαναχρησιμοποίησης κώδικα μεγάλων και πολύπλοκων εφαρμογών), όμως τελικώς η αντικειμενοστρέφεια επικράτησε καθώς μπορούσε να αντεπεξέλθει σε προγράμματα πολύ μεγαλύτερου όγκου και πολυπλοκότητας.

2.3.1 Ιστορικό

Οι περισσότερες αντικειμενοστρεφείς έννοιες εμφανίστηκαν αρχικά στη γλώσσα προγραμματισμού Simula 67, η οποία ήταν προσανατολισμένη στην εκτέλεση

προσομοιώσεων του πραγματικού κόσμου. Οι ιδέες της Simula 67 επηρέασαν κατά τη δεκαετία του '70 την ανάπτυξη της Smalltalk, της γλώσσας που εισήγαγε τον όρο αντικειμενοστρεφής προγραμματισμός. Η Smalltalk αναπτύχθηκε από τον Άλαν Κέι της εταιρείας Xerox στο πλαίσιο μίας εργασίας με στόχο τη δημιουργία ενός χρήσιμου, αλλά και εύχρηστου, προσωπικού υπολογιστή. Όταν η τελική έκδοση της Smalltalk έγινε διαθέσιμη το 1980 η έρευνα για την αντικατάσταση του δομημένου προγραμματισμού με ένα πιο σύγχρονο υπόδειγμα ήταν ήδη εν εξελίξει. Στη γλώσσα αυτή όλοι οι τύποι δεδομένων ήταν κλάσεις (δεν υπήρχαν δηλαδή πια παραδοσιακές δομές δεδομένων παρά μόνο αντικείμενα).

Την ίδια περίπου εποχή, και επίσης με επιρροές από τη Simula, ολοκληρωνόταν η ανάπτυξη της C++ ως μίας ισχυρής επέκτασης της δημοφιλούς γλώσσας προγραμματισμού C στην οποία είχαν "μεταμοσχευθεί" αντικειμενοστρεφή χαρακτηριστικά. Η επιρροή της C++ καθ' όλη της δεκαετία του '80 ήταν καταλυτική με αποτέλεσμα τη σταδιακή κυκλοφορία αντικειμενοστρεφών εκδόσεων πολλών γνωστών διαδικαστικών γλωσσών προγραμματισμού. Κατά το πρώτο ήμισυ της δεκαετίας του '90 η βαθμιαία καθιέρωση στους μικροϋπολογιστές των γραφικών διασυνδέσεων χρήστη (GUI), για την ανάπτυξη των οποίων ο ΑΠ φαινόταν ιδιαιτέρως κατάλληλος, και η επίδραση της C++ οδήγησαν στην επικράτηση της αντικειμενοστρέφειας ως βασικού προγραμματιστικού υποδείγματος.

Το 1995 η εμφάνιση της Java, μίας ιδιαίτερα επιτυχημένης, πλήρως αντικειμενοστρεφούς γλώσσας που έμοιαζε συντακτικώς με τη C/C++ και προσέφερε πρωτοποριακές για την εποχή δυνατότητες, έδωσε νέα ώθηση στον ΑΠ. Παράλληλα εμφανίστηκαν ποικίλες άτυπες βελτιώσεις στο βασικό προγραμματιστικό υπόδειγμα, όπως οι αντικειμενοστρεφείς γλώσσες μοντελοποίησης λογισμικού, τα σχεδιαστικά πρότυπα κλπ. Το 2001 η Microsoft εστίασε την προσοχή της στην πλατφόρμα .NET, μία ανταγωνιστική της Java πλατφόρμα ανάπτυξης και εκτέλεσης λογισμικού η οποία ήταν εξολοκλήρου προσανατολισμένη στην αντικειμενοστρέφεια.

2.3.2 **Βασικές έννοιες του αντικειμενοστρεφούς προγραμματισμού**

Κεντρική ιδέα στον αντικειμενοστρεφή προγραμματισμό είναι η **κλάση** (class), μία αυτοτελής και αφαιρετική αναπαράσταση κάποιας κατηγορίας αντικειμένων, είτε φυσικών αντικειμένων του πραγματικού κόσμου είτε νοητών, εννοιολογικών αντικειμένων, σε ένα περιβάλλον προγραμματισμού. Πρακτικώς είναι ένας τύπος δεδομένων, ή αλλιώς το προσχέδιο μίας δομής δεδομένων με δικά της περιεχόμενα, τόσο μεταβλητές όσο και διαδικασίες. Τα περιεχόμενα αυτά δηλώνονται είτε ως *δημόσια* (public) είτε ως *ιδιωτικά* (private), με τα ιδιωτικά να μην είναι προσπελάσιμα από κώδικα εκτός της κλάσης. Οι διαδικασίες των κλάσεων συνήθως καλούνται *μέθοδοι* (methods) και οι μεταβλητές τους *γνωρίσματα* (attributes) ή *πεδία* (fields). Μία κλάση πρέπει

ιδανικά να είναι εννοιολογικά αυτοτελής, να περιέχει δηλαδή μόνο πεδία τα οποία περιγράφουν μία κατηγορία αντικειμένων και δημόσιες μεθόδους οι οποίες επενεργούν σε αυτά όταν καλούνται από το εξωτερικό πρόγραμμα, χωρίς να εξαρτώνται από άλλα δεδομένα ή κώδικα εκτός της κλάσης, και επαναχρησιμοποιήσιμη, να αποτελεί δηλαδή μαύρο κουτί δυνάμενο να λειτουργήσει χωρίς τροποποιήσεις ως τμήμα διαφορετικών προγραμμάτων.

Αντικείμενο (object) είναι το στιγμιότυπο μίας κλάσης, δηλαδή αυτή καθαυτή η δομή δεδομένων (με αποκλειστικά δεσμευμένο χώρο στη μνήμη) βασισμένη στο «καλούπι» που προσφέρει η κλάση. Παραδείγματος χάρη, σε μία αντικειμενοστρεφή γλώσσα προγραμματισμού θα μπορούσαμε να ορίσουμε κάποια κλάση ονόματι BankAccount, η οποία αναπαριστά έναν τραπεζικό λογαριασμό, και να δηλώσουμε ένα αντικείμενο της με όνομα MyAccount. Το αντικείμενο αυτό θα έχει δεσμεύσει χώρο στη μνήμη με βάση τις μεταβλητές και τις μεθόδους που περιγράψαμε όταν δηλώσαμε την κλάση. Έτσι, στο αντικείμενο θα μπορούσε να περιέχεται ένα γνώρισμα Balance (=υπόλοιπο) και μία μέθοδος GetBalance (=επέστρεψε το υπόλοιπο). Ακολούθως θα μπορούσαμε να δημιουργήσουμε ακόμα ένα ή περισσότερα αντικείμενα της ίδιας κλάσης τα οποία θα είναι διαφορετικές δομές δεδομένων (διαφορετικοί τραπεζικοί λογαριασμοί στο παράδειγμα). Ας σημειωθεί εδώ πως τα αντικείμενα μίας κλάσης μπορούν να προσπελάσουν τα ιδιωτικά περιεχόμενα άλλων αντικειμένων της ίδιας κλάσης.

Ενθυλάκωση δεδομένων (data encapsulation) καλείται η ιδιότητα που προσφέρουν οι κλάσεις να «κρύβουν» τα ιδιωτικά δεδομένα τους από το υπόλοιπο πρόγραμμα και να εξασφαλίζουν πως μόνο μέσω των δημόσιων μεθόδων τους θα μπορούν αυτά να προσπελαστούν. Αυτή η τακτική παρουσιάζει μόνο οφέλη καθώς εξαναγκάζει κάθε εξωτερικό πρόγραμμα να φιλτράρει το χειρισμό που επιθυμεί να κάνει στα πεδία μίας κλάσης μέσω των ελέγχων που μπορούν να περιέχονται στις δημόσιες μεθόδους της κλάσης.

Αφαίρεση δεδομένων καλείται η ιδιότητα των κλάσεων να αναπαριστούν αφαιρετικά πολύπλοκες οντότητες στο προγραμματιστικό περιβάλλον. Μία κλάση αποτελεί ένα αφαιρετικό μοντέλο κάποιας κατηγορίας αντικειμένων. Επίσης οι κλάσεις προσφέρουν και αφαίρεση ως προς τον υπολογιστή, εφόσον η καθεμία μπορεί να θεωρηθεί ένας μικρός και αυτάρκης υπολογιστής (με δική του κατάσταση, μεθόδους και μεταβλητές).

Κληρονομικότητα ονομάζεται η ιδιότητα των κλάσεων να επεκτείνονται σε νέες κλάσεις, ρητά δηλωμένες ως κληρονόμους (*υποκλάσεις* ή 'θυγατρικές κλάσεις'), οι οποίες μπορούν να επαναχρησιμοποιήσουν τις μεταβιβάσιμες μεθόδους και ιδιότητες της γονικής τους κλάσης αλλά και να προσθέσουν δικές τους. Στιγμιότυπα των θυγατρικών κλάσεων μπορούν να χρησιμοποιηθούν όπου απαιτούνται στιγμιότυπα των γονικών (εφόσον η θυγατρική είναι κατά κάποιον τρόπο μία πιο εξειδικευμένη εκδοχή της

γονικής), αλλά το αντίστροφο δεν ισχύει. Παράδειγμα κληρονομικότητας είναι μία γονική κλάση Vehicle (=Όχημα) και οι δύο πιο εξειδικευμένες υποκλάσεις της Car (=Αυτοκίνητο) και Bicycle (=Ποδήλατο), οι οποίες λέμε ότι "κληρονομούν" από αυτήν. Πολλαπλή κληρονομικότητα είναι η δυνατότητα που προσφέρουν ορισμένες γλώσσες προγραμματισμού μία κλάση να κληρονομεί ταυτόχρονα από περισσότερες από μία γονικές. Από μία υποκλάση μπορούν να προκύψουν νέες υποκλάσεις που κληρονομούν από αυτήν, με αποτέλεσμα μία ιεραρχία κλάσεων που συνδέονται μεταξύ τους "ανά γενιά" με σχέσεις κληρονομικότητας.

Υπερφόρτωση μεθόδου (method overloading) είναι η κατάσταση κατά την οποία υπάρχουν, στην ίδια ή σε διαφορετικές κλάσεις, μέθοδοι με το ίδιο όνομα και πιθανώς διαφορετικά ορίσματα. Αν πρόκειται για μεθόδους της ίδιας κλάσης διαφοροποιούνται μόνο από τις διαφορές τους στα ορίσματα και στον τύπο επιστροφής.

Υποσκέλιση μεθόδου (method overriding) είναι η κατάσταση κατά την οποία μία θυγατρική κλάση και η γονική της έχουν μία μέθοδο ομώνυμη και με τα ίδια ορίσματα. Χάρη στη δυνατότητα του *πολυμορφισμού* ο μεταγλωττιστής «ξέρει» πότε να καλέσει ποια μέθοδο, βασισμένος στον τύπο του τρέχοντος αντικειμένου. Δηλαδή πολυμορφισμός είναι η δυνατότητα των αντικειμενοστρεφών μεταγλωττιστών να αποφασίζουν δυναμικά ποια είναι η κατάλληλη να κληθεί μέθοδος σε συνθήκες υποσκέλισης.

Αφηρημένη κλάση (abstract class) είναι μία κλάση που ορίζεται μόνο για να κληρονομηθεί σε θυγατρικές υποκλάσεις και δεν υπάρχουν δικά της στιγμιότυπα (αντικείμενα). Η αφηρημένη κλάση ορίζει απλώς ένα "συμβόλαιο" το οποίο θα πρέπει να ακολουθούν οι υποκλάσεις της όσον αφορά τις υπογραφές των μεθόδων τους (όπου ως υπογραφή ορίζεται το όνομα, τα ορίσματα και η τιμή επιστροφής μίας διαδικασίας). Μία αφηρημένη κλάση μπορεί να έχει και μη αφηρημένες μεθόδους οι οποίες υλοποιούνται στην ίδια την κλάση (αν και φυσικά μπορούν να υποσκελίζονται σε υποκλάσεις). Αντιθέτως οι αφηρημένες μέθοδοί της είναι απλώς ένας ορισμός της υπογραφής τους και εναπόκειται στις υποκλάσεις να τις υλοποιήσουν. Μία αφηρημένη κλάση που δεν έχει γνωρίσματα και όλες οι μέθοδοί της είναι αφηρημένες και δημόσιες καλείται **διασύνδεση** (interface). Οι κλάσεις που κληρονομούν από μία διασύνδεση λέγεται ότι την "υλοποιούν".

Ακολουθεί ένα απλό παράδειγμα σε γλώσσα προγραμματισμού Java:

```
interface Logger
{
   public void log(String msg);
}
```

```
class ConsoleLogger implements Logger
{
   public void log(String msg)
   {
      System.err.println("\nConsole logging..." + msg +          "\n");
   }
}
class FileLogger implements Logger
{
   public void log(String msg)
   {
       System.out.println("\nFile logging..." + msg +
      "\n");
   }
}
public class LogTest
{
    public static void main(String[] args)
    {
       if(args.length != 1)
       {
           System.out.println("\nError. Exiting...");
           return;
       }
       Logger logg;
       String choice = args[0];
       if(choice.equals("FileLogger"))
          logg = new FileLogger();
       else if(choice.equals("ConsoleLogger"))
          logg = new ConsoleLogger();
       else
       {
          System.err.println("\nError. Exiting...");
          return;
       }
       logg.log("Log This!");
    }
}
```

Στο παραπάνω παράδειγμα ορίζουμε μία διασύνδεση Logger η οποία παρέχει την υπογραφή μίας μεθόδου log, που υποθέτουμε πως πρέπει να καταγράφει κάπου πληροφορίες για τα σφάλματα που συναντά η εφαρμογή όταν εκτελείται (οι πληροφορίες αυτές τής μεταβιβάζονται με το αλφαριθμητικό όρισμα msg). Η κλάση ConsoleLogger και η κλάση FileLogger είναι δύο διαφορετικές κλάσεις που υλοποιούν τη διασύνδεση Logger και υποσκελίζουν, η καθεμία με διαφορετικό τρόπο, τη μέθοδο log ώστε η μία να καταγράφει πληροφορίες στην οθόνη και η άλλη σε κάποιο αρχείο.

Το πρόγραμμα LogTest είναι ένα μικρό δοκιμαστικό πρόγραμμα το οποίο δέχεται ως όρισμα γραμμής εντολών το πού επιθυμεί ο χρήστης να γίνεται η καταγραφή και δημιουργεί ένα στιγμιότυπο της αντίστοιχης κλάσης: της ConsoleLogger ή της FileLogger. Το στιγμιότυπο αυτό δηλώνεται με τον γενικότερο τύπο Logger, τον τύπο δηλαδή της διασύνδεσης που υλοποιούν και οι δύο κλάσεις, αλλά χάρη στον πολυμορφισμό καλείται αυτομάτως η κατάλληλη εκδοχή της μεθόδου log.

2.3.3 **Αρχές αντικειμενοστρεφούς σχεδίασης**

Με το πέρασμα του χρόνου κωδικοποιήθηκαν κάποιες ανεπίσημες αρχές για την ορθή σχεδίαση αντικειμενοστρεφών συστημάτων λογισμικού. Οι αρχές αυτές παρουσιάστηκαν κατά καιρούς σε βιβλία και άρθρα ακαδημαϊκών και αναγνωρισμένων μηχανικών λογισμικού.

Οι σπουδαιότερες αρχές είναι οι παρακάτω:

- Αρχή ανοιχτότητας-κλειστότητας (open-closed principle), του δημιουργού της γλώσσας προγραμματισμού Eiffel Μπέρτραντ Μέιερ. Η αρχή αυτή δηλώνει πως τα συστατικά ενός προγράμματος πρέπει να είναι "ανοιχτά" ως προς την επέκταση των δυνατοτήτων του συστήματος αλλά "κλειστά" ως προς αλλαγές στην υλοποίηση του. Πρακτικώς αυτό σημαίνει οι διάφορες κλάσεις και τα υπόλοιπα τμήματα λογισμικού να μη χρειάζεται να τροποποιηθούν σε περίπτωση που προστεθεί νέα λειτουργικότητα στο σύστημα (π.χ. μία νέα κλάση) προκειμένου να την αξιοποιήσουν. Βεβαίως είναι αδύνατο να μη χρειάζεται να τροποποιηθεί τίποτα, οπότε αυτό που επιτάσσει στην πραγματικότητα η εν λόγω αρχή είναι η ελαχιστοποίηση και η συγκέντρωση, κατά προτίμηση σε ένα μικρό τμήμα του κώδικα, των γραμμών που θα πρέπει να αλλάξουν. Αυτό συνήθως επιτυγχάνεται μέσω αφαίρεσης (με αφηρημένες κλάσεις ή διασυνδέσεις και πραγματικές κλάσεις που κληρονομούν από αυτές) και με χρήση του πολυμορφισμού. Έτσι, στο παράδειγμα της προηγούμενης ενότητας αν προσθέσουμε και μία τρίτη υλοποίηση της διασύνδεσης Logger, την PrinterLogger η οποία "καταγράφει" σφάλματα αποστέλλοντας τα προς εκτύπωση, ο κώδικας του προγράμματος γίνεται:

```
public class LogTest
{
    public static void main(String[] args)
    {
       if(args.length != 1)
       {
           System.out.println("\nError. Exiting...");
           return;
       }
```

```
        Logger logg;
        String choice = args[0];
        if(choice.equals("FileLogger"))
           logg = new FileLogger();
        else if(choice.equals("ConsoleLogger"))
           logg = new ConsoleLogger();
        else if(choice.equals("PrinterLogger"))
           logg = new PrinterLogger();
        else
        {
           System.err.println("\nError. Exiting...");
           return;
        }
        logg.log("Log This!");
    }
}
```

Όπως φαίνεται μόνη αλλαγή είναι η προσθήκη μίας ακόμα δήλωσης else if. Η εντολή

```
logg.log("Log This!");
```

μπορεί να λειτουργήσει και με αντικείμενα του νέου τύπου χωρίς καμία τροποποίηση. Μία συνηθισμένη τακτική για διασφάλιση της κλειστότητας του ολικού προγράμματος ως προς την υλοποίηση μίας κλάσης, είναι η συνειδητή προσπάθεια για δήλωση όλων των γνωρισμάτων της ως ιδιωτικών. Έτσι η προσπέλαση των πεδίων της κλάσης μπορεί να ελεγχθεί εξ ολοκλήρου μέσω ειδικών δημόσιων μεθόδων της, γεγονός που διευκολύνει κατά πολύ την αποσφαλμάτωση: στις μεθόδους αυτές συγκεντρώνονται οι έλεγχοι επιτρεπτών τιμών για τα πεδία, έλεγχοι κατάλληλων συνθηκών κλπ.

- Αρχή υποκατάστασης Λίσκοφ (Liskov substitution principle), της επιστήμονα υπολογιστών Μπάρμπαρα Λίσκοφ. Η αρχή αυτή συμπυκνώνεται στον παρακάτω κανόνα για σχηματισμό μίας ορθής ιεραρχίας κλάσεων: μία κλάση Κ1 μπορεί να υλοποιηθεί ως υποκλάση μίας κλάσης Κ2 αν κάθε πρόγραμμα Π το οποίο λειτουργεί με αντικείμενα Κ2 συμπεριφέρεται με τον ίδιο τρόπο και με αντίστοιχα αντικείμενα Κ1. Έτσι με την αρχή υποκατάστασης Λίσκοφ φαίνεται πως για να οριστεί μία κλάση ως θυγατρική μίας άλλης δεν αρκεί να έχουν διαισθητικά μία ανάλογη εννοιολογική σχέση (π.χ. μία κλάση που αναπαριστά όχημα και μία που αναπαριστά αυτοκίνητο) αλλά, στο πλαίσιο του υπό εξέταση προγράμματος, τα αντικείμενα της υποκλάσης να έχουν πάντα την ίδια προγραμματιστική συμπεριφορά με τα αντικείμενα της υπερκλάσης υπό τις ίδιες συνθήκες.

- Αρχή αντιστροφής εξαρτήσεων (dependency inversion principle), του γνωστού μηχανικού λογισμικού Ρόμπερτ Σέσιλ Μάρτιν. Η αρχή αυτή πρακτικά αποτελεί

εκλέπτυνση της αρχής ανοιχτότητας-κλειστότητας, προϋποθέτοντας όμως χρήση και της αρχής υποκατάστασης Λίσκοφ. Αφορά ιεραρχίες κληρονομικότητας κλάσεων και τη χρήση αντικειμένων αυτών των ιεραρχιών από εξωτερικά προγράμματα. Στα πλαίσια της αρχής αντιστροφής εξαρτήσεων ένα τμήμα λογισμικού Α (π.χ. μία κλάση) το οποίο χρησιμοποιεί τις υπηρεσίες που παρέχει ένα άλλο τμήμα λογισμικού Β, καλώντας για παράδειγμα μία μέθοδό του, θεωρείται στοιχείο "υψηλότερου επιπέδου" σε σχέση με το Β. Η αρχή λέει πως τα υψηλού επιπέδου στοιχεία δεν πρέπει να εξαρτώνται από την υλοποίηση χαμηλότερου επιπέδου στοιχείων, αλλά πως και τα δύο πρέπει να βασίζονται σε ενδιάμεσα επίπεδα αφαίρεσης. Στην πράξη αυτή η αφαίρεση είναι μία διασύνδεση (ή αφηρημένη κλάση) την οποία γνωρίζει το υψηλού επιπέδου στοιχείο Α και υλοποιεί το χαμηλού επιπέδου στοιχείο Β. Ακόμα και αν το Β αλλαχθεί με μία κλάση Γ η οποία επίσης υλοποιεί την ίδια διασύνδεση, το Α θα πρέπει να συνεχίσει να λειτουργεί χωρίς καμία τροποποίηση. Η αρχή αντιστροφής εξαρτήσεων δεν είναι παρά ένα απτό παράδειγμα χρήσης ιεραρχικών επιπέδων με τη βοήθεια ενδιάμεσων αφαιρέσεων, μίας πρακτικής που εφαρμόζεται κατά κόρον στην επιστήμη υπολογιστών (για ένα άλλο παράδειγμα βλέπε δίκτυα υπολογιστών).

- Αρχή διαχωρισμού διασυνδέσεων (interface segregation principle), του μηχανικού λογισμικού Ρόμπερτ Σέσιλ Μάρτιν. Η εν λόγω αρχή σημαίνει ότι σε περιπτώσεις όπου διαφορετικά υποσύνολα μεθόδων μίας κλάσης αφορούν διαφορετικές περιπτώσεις χρήσης της κλάσης, σκόπιμο είναι να ορίζουμε επιμέρους διασυνδέσεις τις οποίες η κλάση θα υλοποιεί. Κάθε τέτοια διασύνδεση θα ορίζει μόνο το αντίστοιχο υποσύνολο των μεθόδων.

- Αρχή μοναδικής αρμοδιότητας (single responsibility principle), των Τομ Ντε Μάρκο και Μέιρ Πέιτζ Τζόουνς. Σύμφωνα με την αρχή αυτή κάθε κλάση θα πρέπει να έχει μόνο μία, καλά ορισμένη και διαχωρισμένη από το υπόλοιπο πρόγραμμα αρμοδιότητα, η ύπαρξη της οποίας να εξυπηρετεί έναν συγκεκριμένο σκοπό. Αν μπορούμε να εντοπίσουμε σε μία κλάση Α δύο διαφορετικές αρμοδιότητες, τότε η καλύτερη λύση είναι η διάσπαση της σε δύο κλάσεις Β' και Γ', καθεμία από τις οποίες θα λάβει ένα υποσύνολο των πεδίων και των μεθόδων της Α. Τα υποσύνολα αυτά θα είναι ξένα μεταξύ τους, οπότε με το ανάποδο σκεπτικό αν μπορούμε να διασπάσουμε μία κλάση Α σε δύο άλλες κλάσεις (π.χ. σε περίπτωση που κάποιες μέθοδοι δε χρησιμοποιούν κάποια γνωρίσματα, οπότε οι μεν μπορούν να καταλήξουν στη μία κλάση Β' και τα πεδία στην άλλη κλάση Γ') τότε πιθανώς η κλάση να παραβιάζει την αρχή μοναδικής αρμοδιότητας. Έτσι έχουν προταθεί κάποιες μετρικές οι οποίες επιχειρούν να προσδιορίσουν την έλλειψη *συνοχής* (cohesion) σε μία κλάση, δηλαδή το κατά πόσον οι μέθοδοι της δε σχετίζονται με τα γνωρίσματα της. Συνήθως η συνοχή αντιπαραβάλλεται με τη *σύζευξη* (coupling),

δηλαδή το βαθμό στον οποίον μία κλάση εξαρτάται από κάποια/ες άλλη/ες, και τα δύο αυτά μεγέθη είναι αντιστρόφως ανάλογα.

3.**Η γλώσσα προγραμματισμού** JAVA

3.1 **Ιστορικά στοιχεία**

Στις αρχές του 1991, η ***Sun*** αναζητούσε το κατάλληλο εργαλείο για να αποτελέσει την πλατφόρμα ανάπτυξης λογισμικού σε μικρο-συσκευές (έξυπνες οικιακές συσκευές έως πολύπλοκα συστήματα παραγωγής γραφικών). Τα εργαλεία της εποχής ήταν γλώσσες όπως η C++ και η C. Μετά από διάφορους πειραματισμούς προέκυψε το συμπέρασμα ότι οι υπάρχουσες γλώσσες δεν μπορούσαν να καλύψουν τις ανάγκες τους. Ο "πατέρας" της Java, James Gosling, που εργαζόταν εκείνη την εποχή για την Sun, έκανε ήδη πειραματισμούς πάνω στη C++ και είχε παρουσιάσει κατά καιρούς κάποιες πειραματικές γλώσσες (C++ ++) ως πρότυπα για το νέο εργαλείο που αναζητούσαν στην ***Sun***. Τελικά μετά από λίγο καιρό κατέληξαν με μια πρόταση για το επιτελείο της εταιρίας, η οποία ήταν η γλωσσα ***Oak***. Το όνομά της το πήρε από το ομώνυμο δένδρο (βελανιδιά) το οποίο ο Gosling είχε έξω από το γραφείο του και έβλεπε κάθε μέρα.

Η ***Oak*** ήταν μία γλώσσα που διατηρούσε μεγάλη συγγένεια με την C++. Παρόλα αυτά είχε πολύ πιο έντονο αντικειμενοστρεφή (***object oriented***) χαρακτήρα σε σχέση με την C++ και χαρακτηριζόταν για την απλότητα της. Σύντομα οι υπεύθυνοι ανάπτυξης της νέας γλώσσας ανακάλυψαν ότι το όνομα ***Oak*** ήταν ήδη κατοχυρωμένο οπότε κατά την διάρκεια μιας εκ των πολλών συναντήσεων σε κάποιο τοπικό καφέ αποφάσισαν να μετονομάσουν το νέο τους δημιούργημα σε Java που εκτός των άλλων ήταν το όνομα της αγαπημενης ποικιλιας καφέ για τους δημιουργούς της. Η επίσημη εμφάνιση της ***Java*** αλλα και του ***HotJava*** (πλοηγός με υποστήριξη ***Java***) στη βιομηχανία της πληροφορικής έγινε το Μάρτιο του 1995 όταν η ***Sun*** την ανακοίνωσε στο συνέδριο ***Sun World*** 1995. Ο πρώτος μεταγλωττιστής (***compiler***) της ήταν γραμμένος στη γλώσσα C από τον James Gosling. Το 1994, ο A.Van Hoff ξαναγράφει τον μεταγλωττιστή της γλώσσας σε ***Java***, ενώ το Δεκέμβριο του 1995 πρώτες οι IBM, Borland, Mitsubishi Electronics, Sybase και Symantec ανακοινώνουν σχέδια να χρησιμοποιήσουν τη ***Java*** για την δημιουργία λογισμικού. Από εκεί και πέρα η Java ακολουθεί μία ανοδική πορεία και είναι πλέον μία από τις πιο δημοφιλείς γλώσσες στον χώρο της πληροφορικής. Στις 13 Νοεμβρίου του 2006 η Java έγινε πλέον μια γλώσσα ανοιχτού κώδικα (GPL) όσον αφορά το μεταγλωττιστή (javac) και το πακέτο ανάπτυξης (JDK, Java Development Kit).

3.2 **Τα χαρακτηριστικά της** Java

- Αντικειμενοστρεφής

- Δημιουργία ανεξάρτητων εφαρμογών και applets (applet = προγράμματα που περιλαμβάνονται σε HTML σελίδες και εκτελούνται από τον Web Browser).
- Είναι Interpreted γλώσσα. Αυτό σημαίνει ότι ο java compiler δεν παράγει εκτελέσιμο κώδικα αλλά μια μορφή ψευδοκώδικα (bytecode) το οποίο από μόνο

του δεν τρέχει σε καμία μηχανή. Προκειμένου λοιπόν να εκτελεστεί απαιτείται η χρήση ενός interpreter (=διερμηνέα) για να μετατρέψει το bytecode σε πραγματικό εκτελέσιμο κώδικα. Αυτό το χαρακτηριστικό δίνει τη δυνατότητα στα java bytecodes να μπορούν να τρέξουν σε οποιοδήποτε μηχάνημα, κάτω από οποιοδήποτε λειτουργικό, αρκεί να έχει εγκατασταθεί ένας java interpreter. Επίσης ένα άλλο χαρακτηριστικό του java bytecode είναι το μικρό του μέγεθος, (μόλις λίγα Kilobytes). Αυτό το κάνει ιδανικό για μετάδοση μέσω του δικτύου.

- Κατανεμημένη (distributed). Δηλαδή ένα πρόγραμμα σε Java είναι δυνατό να το φέρουμε από το δίκτυο και να το τρέξουμε. Επίσης είναι δυνατό διαφορετικά κομμάτια του προγράμματος να έρθουν από διαφορετικά sites.

- Ασφαλής (secure). Στο δίκτυο όμως ελλοχεύουν πολλοί κίνδυνοι για τον χρήστη - παραλήπτη μιας δικτυακής εφαρμογής, γι' αυτό η Java έχει σχεδιαστεί έτσι ώστε να ελαχιστοποιείται η πιθανότητα προσβολής του συστήματος του χρήστη από κάποιο applet γραμμένο για τέτοιο σκοπό.

- Είναι multithreaded. Η Java υποστηρίζει εγγενώς την χρήση πολλών threads. Προκειμένου να το επιτύχει αυτό σε συστήματα με έναν επεξεργαστή, το Java runtime system (interpreter) υλοποιεί ένα δικό χρονοδρομολογητή (scheduler), ενώ σε συστήματα που υποστηρίζουν πολυεπεξεργασία η δημιουργία των threads ανατίθεται στο λειτουργικό σύστημα. Φυσικά όλα αυτά είναι αόρατα τόσο στον προγραμματιστή όσο και στον χρήστη.

- Υποστηρίζει multimedia εφαρμογές. Με αυτό εννοούμε ότι η Java παρέχει ευκολίες στη δημιουργία multimedia εφαρμογών. Αυτό επιτυγχάνεται τόσο με την ευελιξίας της σαν γλώσσα όσο και με τις πλούσιες και συνεχώς εμπλουτιζόμενες βιβλιοθήκες της.

3.3 **Τα εργαλεία της** JAVA

Ακολούθως παρουσιάζονται εν συντομία όλα τα εργαλεία που έρχονται με το Java 2 Platform Standard Edition (J2SE), της Sun microsystems.

- **javac** Είναι ο compiler της Java. Η χρήση του στο command-line είναι : *javac <όνομα αρχείου>*. Εδώ να σημειώσουμε ότι το javac δεν παράγει ένα αρχείο με όλον τον κώδικα, αλλά χωριστό αρχείο για κάθε κλάση. Τα αρχεία των κλάσεων ονομάζονται : *<όνομα κλάσης>.class.*

- **java** Είναι ο interpreter της Java. Η χρήση του είναι η εξής : *java <κλάση>, πχ java myClass και όχι java myClass.class.*

- **javaw** (MONO στα Windows 95/NT) Είναι παρόμοιο με το java με μόνη την διαφορά ότι δεν χρειάζεται shell για να τρέξει.

- **jdb** Είναι ο Java debuger.

- **javah** Κατασκευάζει C files και stub files για κάποια κλάση. Αυτά τα αρχεία είναι απαραίτητα όταν θέλουμε να υλοποιήσουμε κάποιες από τις μεθόδους της κλάσης σε C, πράγμα πολύ σπάνιο.

- **javap** Είναι ο Java disassembler.

- **javadoc** Είναι ένα πρόγραμμα για αυτόματη κατασκευή documentation. Είναι αρκετά χρήσιμο στην κατασκευή βοηθημάτων και τεχνικών αναφορών για εφαρμογές οποιουδήποτε μεγέθους.

- **appletviewer** Είναι ένα πρόγραμμα το οποίο μας επιτρέπει να τρέχουμε και να χρησιμοποιούμε τα διάφορα applets σε Java. Οι stand-alone εφαρμογές, ωστόσο, δεν τρέχουν στον appletviewer αλλά κατευθείαν στον java ή javaw.

3.3.1 **Δημιουργία μιας** JAVA **εφαρμογής**

Με τα ακόλουθα βήματα μπορεί να κατασκευαστεί μια ανεξάρτητη JAVA εφαρμογή (stand-alone application).
Δημιουργούμε το αρχείο πηγαίου κώδικα HelloWorldApp.java το οποίο περιέχει τις ακόλουθες εντολές:

```
/**
* The HelloWorldApp class implements an application that
* simply displays "Hello World!" to the standard output.
**/
class HelloWorldApp {
     public static void main(String[] args) {
          System.out.println("Hello World!"); //Display
                                              the string.
     }
}
```

Το κάνουμε Compile χρησιμοποιώντας τον Java compiler:

javac HelloWorldApp.java

Ο compiler θα δημιουργήσει το αρχείο HelloWorldApp.class
Εκτελούμε την εφαρμογή καλώντας τον Java interpreter:

java HelloWorldApp

Θα δούμε στην οθόνη το μήνυμα "Hello World!"

3.4 Υλοποίηση αρχών της αντικειμενοστρεφούς σχεδίασης σε JAVA

3.4.1 Κλάσεις και αντικείμενα στη JAVA

Μία κλάση της java μοιάζει αρκετά με ένα structure της C. Δηλαδή έχει την ακόλουθη δομή :

```
class <class_name> {
     <data_type or class_name> <variables or objects>;
     <returned type> <method_name> (<parameter list>) {
          <method body>
     }
     <data_type or class_name> <variables or objects>;
     <returned type> <method_name> (<parameter list>) {
     <method body>
     }
     κ.ο.κ.
}
```

Για παράδειγμα

```
class Apple {
     int num_of_vitamins; // πεδίο

     int countVitamins() { // μέθοδος
          return num_of_vitamins;
     }
```

```
     Vitamins v1, v2, v3; // πεδία

     Vitamin getVitamin(int num) { // μέθοδος
          return ....;
     }
}
```

Στο παραπάνω παράδειγμα δημιουργούμε μια νέα κλάση η οποία περιέχει ως μέλη της μεταβλητές και μεθόδους. Συγκεκριμένα, η μεταβλητή ***num_of_vitamins*** είναι μία κοινή μεταβλητή τύπου **int**. Αντίθετα το ***Vitamins*** είναι όνομα μιας κλάσης και συνεπώς τα ***v1, v2, v3*** θα είναι αντικείμενα της κλάσης ***Vitamin*** ή καλύτερα αναφορές (κάτι σαν δείκτες) στα αντίστοιχα αντικείμενα. Τέλος βλέπουμε ότι οι μέθοδοι μπορούν να έχουν επιστρεφόμενο τύπο τόσο έναν απλό τύπο δεδομένων όσο και μία κλάση.

Όπως ειπώθηκε και παραπάνω τα αντικείμενα περιέχουν δεδομένα και παρέχουν μεθόδους για την επεξεργασία των δεδομένων αυτών καθώς και για την επικοινωνία με άλλα αντικείμενα. Ας εμβαθύνουμε λίγο περισσότερο σε αυτές τις έννοιες.

Κατ` αρχήν τα δεδομένα μπορεί να είναι άλλα αντικείμενα που περιέχονται μέσα σε ένα άλλο αντικείμενο ή μπορεί να είναι κοινές μεταβλητές όπως τις γνωρίζουμε από την C και την PASCAL. Τις μεταβλητές και τα αντικείμενα αυτά τα καλούμε *πεδία ή* ***instance variables*** του αντικειμένου.

Σε αυτό το σημείο να παρατηρήσουμε ότι μία κλάση μοιάζει με ένα structure της C το οποίο μπορεί να περιέχει κοινές μεταβλητές ή μεταβλητές που προέκυψαν από άλλα structures. Επιπλέον μία κλάση (ή ένα αντικείμενο) περιέχει και μεθόδους για την επικοινωνία του με τον *έξω κόσμο*, δηλαδή τα άλλα αντικείμενα. Οι μέθοδοι αυτοί υλοποιούνται σαν συναρτήσεις παρόμοιες με αυτές της C. Όταν λοιπόν κάποιος θέλει να ζητήσει κάτι από ένα αντικείμενο, (ή εναλλακτικά να στείλει ένα μήνυμα - αίτημα), δεν έχει παρά να καλέσει - εκτελέσει την αντίστοιχη μέθοδο του αντικειμένου. Αυτό γίνεται πολύ απλά ως εξής :

<Αντικείμενο>.<μέθοδος>(<Λίστα παραμέτρων>);

Για παράδειγμα

```
myApple.countVitamins();
```

Αυτό το σχήμα μας θυμίζει πολύ τον τρόπο πρόσβασης σε μία μεταβλητή που περιέχεται σε ένα structure της C. Επίσης να σημειώσουμε ότι μια μέθοδος μπορεί να καλεί άλλες μεθόδους του ίδιου αντικειμένου ή να επεξεργάζεται τα πεδία του. Φυσικά μπορεί να καλεί και μεθόδους ξένων αντικειμένων εφόσον έχει κάποιον δείκτη σε αυτά.

Όλα τα παραπάνω αποτελούν ένα πολύ σπουδαίο χαρακτηριστικό του αντικειμενοστρεφούς προγραμματισμού: το να μπορεί να κρατά τα δεδομένα του κρυφά από τα άλλα αντικείμενα αλλά και να προσφέρει μεθόδους με τις οποίες μπορούν άλλα αντικείμενα να επικοινωνούν και να αλληλεπιδρούν μαζί του. Επιπλέον ο κώδικας που υλοποιεί αυτές τις μεθόδους είναι άγνωστος σε άλλες κλάσεις ή αντικείμενα. Έτσι έχουμε την δημιουργία ενός *interface* επικοινωνίας ανεξάρτητου από την υλοποίηση και την εσωτερική δομή του αντικειμένου. Αυτό λέγεται *implementation hiding* που είναι μία μορφή *data abstraction*.

3.4.2 **Ενθυλάκωση** (encapsulation)

Ας ορίσουμε μια κλάση Person η οποία θα περιέχει ορισμένες πληροφορίες για ένα άτομο, όπως π.χ. το όνομά του, την ηλικία του, το τηλέφωνό του και τη διεύθυνση email του. Μια τέτοια κλάση μπορεί να χρησιμοποιηθεί π.χ. σε ένα πρόγραμμα ατζέντας ή ακόμη και ως βάση σε πρόγραμμα πελατολογίου, ασθενών, κλπ.

```
class Person
{
      // οι μεταβλητές της κλάσης
      String Firstname_, Lastname_;
      int Age_;
      String Telephone_;
      String Email_;

      // ο constructor
      Person(String fname, String lname, int age, String
      tel,String email)
      {
            Firstname_ = new String(fname);
            Lastname_ = new String(lname);
            Age_ = age;
            Telephone_ = new String(tel);
            Email_ = new String(email);
      }
}
```

Όπως μπορεί να παρατηρήσει κανείς τα ονόματα των μεταβλητών της κλάσης λήγουν σε "_". Φυσικά, κάτι τέτοιο δεν είναι αναγκαίο, είναι όμως μια συνηθισμένη πρακτική και βοηθάει στην αναγνώριση και διαχωρισμό των απλών μεταβλητών από τις μεταβλητές μέλη μιας κλάσης.

Αφού ορίσαμε την κλάση, μπορούμε να προχωρήσουμε στην δημιουργία κάποιων αντικειμένων της κλάσης:

```
Person bilbo = new Person( “Bilbo”, “Baggins”, 111, “+306970123456”, “bilbobaggins@theshire.net”);
```

Με αυτόν τον τρόπο, δημιουργήσαμε το αντικείμενο bilbo που αντιστοιχεί στο άτομο Bilbo Baggins, 111 ετών με τηλ. 306970123456 και email bilbobaggins@theshire.net.

Όπως είναι οι πληροφορίες που περιγράφουν το άτομο είναι προσβάσιμες σε όλους. Αυτό σημαίνει ότι αν με κάποιον τρόπο αποκτήσουμε πρόσβαση στο αντικείμενο bilbo, θα μπορούμε να αλλάξουμε οποιαδήποτε πληροφορία θελήσουμε και με οποιονδήποτε τρόπο. Δηλαδή, να δώσουμε μια μη έγκυρη διεύθυνση email στο πεδίο email_ ή μια άσχετη πληροφορία στο πεδίο telephone_. Και μάλιστα με πολύ απλό τρόπο:

```
bilbo.Lastname_ = “μπαγκινσόπουλος”;
bilbo.Age_ = 3;
bilbo.Email_ = “this is definitely not a valid email address”;
bilbo.Telephone_ = “yeah, try to call this”;
```

Πρόκειται για τρανή παραβίαση των προσωπικών δεδομένων!!!

Πώς μπορούμε να αποφύγουμε τέτοιου είδους μη προβλεπόμενη μετατροπή των δεδομένων ενός αντικειμένου;

Η Java καθώς και οι περισσότερες γλώσσες προγραμματισμού που έχουν σχεδιαστεί γύρω από το μοντέλο του αντικειμενοστρεφούς προγραμματισμού, προβλέπουν τον περιορισμό της πρόσβασης των δεδομένων σε επίπεδα. Το πρώτο επίπεδο το έχουμε ήδη συναντήσει, κατά τον ορισμό της μεθόδου main() που ορίζεται πάντα public για να είναι προσβάσιμη από το λειτουργικό σύστημα.

Ακριβέστερα, η ***public*** ορίζει μια μέθοδο ή μια μεταβλητή ως προσβάσιμη από οπουδήποτε μέσα στην ίδια κλάση ή εκτός της κλάσης. Το αντίστροφο, δηλαδή ο περιορισμός της πρόσβασης γίνεται με τη χρήση της λέξης ***private***. Η ***private*** περιορίζει την πρόσβαση της μεταβλητής ή της μεθόδου μόνο στην συγκεκριμένη κλάση (και φυσικά σε αντικείμενα αυτής). Οποιαδήποτε πρόσβαση εκτός, θα αποτρέπεται στο επίπεδο της μεταγλώττισης ακόμη. Για παράδειγμα η παραπάνω κλάση Person, με τη χρήση της ***private***, θα μετασχηματιστεί ως εξής:

```
class Person
{
     // οι μεταβλητές της κλάσης
     private String Firstname_, Lastname_;
```

```
private int Age_;
private String Telephone_;
private String Email_;
...
```

Αυτό όμως σημαίνει ότι δεν θα είναι πλέον δυνατή η πρόσβαση σε οποιαδήποτε πληροφορία του ατόμου ακόμη και για απλή ανάγνωση! Κάτι τέτοιο δεν είναι επιθυμητό και πρέπει να βρεθεί τρόπος να επιτραπεί έστω και ελεγχόμενη πρόσβαση στα δεδομένα.

Ακριβώς, αυτό επιτυγχάνουμε με την χρήση των μεθόδων της κλάσης. Ελεγχόμενη πρόσβαση για ανάγνωση αλλά και μετατροπή των δεδομένων. Συνήθως και για τις περισσότερες περιπτώσεις κλάσεων, αρκεί ο ορισμός ενός ζεύγους μεθόδων για κάθε μεταβλητή της κλάσης, μία για ανάγνωση και μια για μετατροπή της τιμής της μεταβλητής (ένα ζεύγος getter/setter όπως λέγονται συχνά).

Με τον ίδιο τρόπο που περιορίζουμε την πρόσβαση σε μεταβλητές μπορούμε να περιορίσουμε την πρόσβαση και σε μεθόδους. Θα μπορούσαμε π.χ. να έχουμε μια μέθοδο που να ελέγχει αν ο αριθμός τηλεφώνου του ατόμου είναι έγκυρος, πραγματοποιώντας αναζήτηση σε κάποια βάση δεδομένων. Οπωσδήποτε, μια τέτοια μέθοδος δεν θα θέλαμε να είναι προσβάσιμη από οποιονδήποτε εκτός της κλάσης, παρά μόνο σε άλλες μεθόδους της ίδιας της κλάσης (π.χ. στην μέθοδο setTelephone()).

3.4.3 **Κληρονομικότητα** (Inheritance)

Είναι ο μηχανισμός εκείνος ο οποίος επιτρέπει σε μια κλάση Β να κληρονομήσει πεδία και μεθόδους από μια κλάση Α. Λέμε ότι η "Β κληρονομεί από την Α". Τα αντικείμενα (instances) της κλάσης Β έχουν πρόσβαση στα δεδομένα και τις μεθόδους της κλάσης Α χωρίς να χρειάζεται να τα ξαναδηλώσουμε. Δηλαδή είναι σαν να αντιγράψαμε τα περιεχόμενα της κλάσης Α στην κλάση Β. Ακόμη, να πούμε ότι για να δηλώσουμε ότι η Β κληρονομεί από την Α χρησιμοποιούμε την δεσμευμένη λέξη **extends.** Ένα παράδειγμα σε JAVA φαίνεται παρακάτω:

```
class A {
     int a;
     void add(int x) {
          a += x;
     }
}

class B extends A {
     // Τα 'a' και 'void add(int x) {...}' υπονοούνται.
     int prev;
     void sub(int x) {
          prev = a;
```

```
            a -= x;
      }
}
```

Βλέπουμε δηλαδή ότι στην κλάση Β δεν ξαναγράφουμε τον κώδικα που αφορά το a και το ***void add(int x).*** Αυτό μας απαλλάσσει από σημαντικό προγραμματιστικό φόρτο. Επίσης βλέπουμε ότι μπορούμε στην Β να δηλώσουμε νέα πεδία και μεθόδους και έτσι να επεκτείνουμε τη λειτουργικότητα της κλάσης αυτής σε σχέση με την Α.

Είναι δυνατό οι μέθοδοι που ορίζονται στη Β να χρησιμοποιούν τα πεδία που δηλώθηκαν στην Α και να καλούν τις μεθόδους που ορίσθηκαν σ' αυτήν (την Α).

```
class B extends A {
      int square() {
            return a*a;
      }
      void increaseBy1() {
            add(1);
      }
}
```

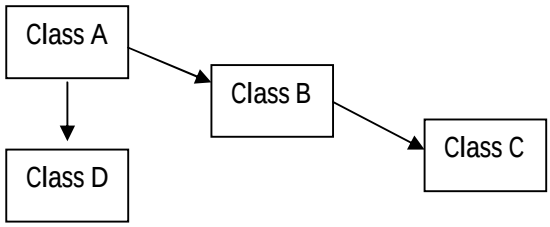
Επιπλέον είναι δυνατό η κλάση Β να μεταβάλλει κάποια από τα ήδη υπάρχοντα πεδία ή μεθόδους. Για παράδειγμα :

```
class B extends A {
      a : float; // Αλλάζει τον τύπο της α.
      void add(int x) { // Αλλάζει την add(int x).
            // Αυτό λέγεται overriding.
            a = a + ((float) x);
            print(a);
      }
}
```

Όπως αναφέρεται και στο παράδειγμα η μεταβολή κάποιου πεδίου ή μεθόδου, που κληρονομήθηκε από την κλάση Α στην κλάση Β, λέγεται ***υπερκάλυψη*** **(overriding)** του πεδίου/μεθόδου της Α από το αντίστοιχο πεδίο/μεθόδο της Β. Ας παρατηρήσουμε ότι η μέθοδος της Β που κάνει override έχει το ίδιο όνομα, λίστα παραμέτρων και

επιστρεφόμενο τύπο με αυτήν της Α. Αν δεν είναι τα ίδια τότε δεν έχουμε υπερκάλυψη αλλά ορισμό μιας νέας μεθόδου.

Η κλάση Β λέγεται υποκλάση (subclass) της Α ή παράγωγη κλάση ή παιδί της Α. Η Α τώρα λέγεται υπερκλάση (superclass) της Β ή παράγουσα κλάση ή πατέρας της Β. Η διαδικασία αυτή λέγεται subclassing. Βέβαια από την Α μπορούν να προκύψουν και άλλες κλάσεις εκτός από την Β. Επιπλέον και από την Β μπορούν να προκύψουν νέες υποκλάσεις. Δηλαδή η κληρονομικότητα δημιουργεί μια δενδρική συσχέτιση μεταξύ των κλάσεων.



Στο παραπάνω σχήμα η Α είναι υπερκλάση των Β και D. Η Β είναι υπερκλάση της C, επομένως η Α είναι υπερκλάση και της C (μεταβατική ιδιότητα). Επίσης οι Β, D είναι υποκλάσεις της Α, ενώ η C είναι υποκλάση τόσο της Β όσο και της Α. Φυσικά η C θα περιέχει τα δεδομένα και τις μεθόδους και της Α και της Β.

Φυσικά όλα όσα αναφέρθηκαν σχετικά με το overriding και τη δυνατότητα πρόσβασης σε πεδία/μεθόδους των υπερκλάσεων ισχύουν τόσο για τον πατέρα όσο και για τον πατέρα του πατέρα κ.ο.κ.. Δηλαδή η C μπορεί να δει όλες τις μεθόδους της Α και φυσικά μπορεί να τους κάνει override εφ' όσον δεν το έχει κάνει η Β.

3.4.4 **Υπερφόρτωση** (overloading)

Έστω το παρακάτω κομμάτι κώδικα:

```
class number {
     int a;
     void add(int x) {
          a += x;
     }
     void add(float f) {
          a += ((int) f);
     }
     void add(number n) {
          a += n.get_a();
     }
     int get_a() {
          return a;
     }
}
```

Βλέπουμε δηλαδή ότι η συνάρτηση *void add(...)* χρησιμοποιείται πολλές φορές (3) αλλά κάθε φορά με διαφορετική λίστα παραμέτρων. Αυτή η επαναχρησιμοποίηση του ίδιου ονόματος λέγεται **υπερφόρτωση** (overloading) του ονόματος αυτού.

Στις αντικειμενοστρεφείς γλώσσες προγραμματισμού επιτρέπεται ο ορισμός μίας μεθόδου με τι ίδιο όνομα δύο ή περισσότερες φορές αρκεί να έχουν διαφορετικές λίστες παραμέτρων:

aMethod(), aMethod(type1), aMethod(type2), aMethod(type1, type2), aMethod(type2, type1).

Όλες οι παραπάνω μέθοδοι είναι διαφορετικές μεταξύ τους. Έτσι, κατά την κλήση μιας μεθόδου κατ' αρχήν εξετάζεται το όνομά της και έπειτα εξετάζεται και η λίστα των παραμέτρων της, όσον αφορά το πλήθος, τον τύπο και τη σειρά των παραμέτρων. Αυτό είναι μια μορφή πολυμορφισμού.

3.4.5 **Κατασκευαστές** (constructors)

Κάθε κλάση διαθέτει τουλάχιστο μία μέθοδο η οποία εκτελείται κατά τη δημιουργία ενός στιγμιότυπου της, (δηλαδή κατά τη δημιουργία ενός αντικειμένου της κλάσης αυτής). Αυτές οι μέθοδοι λέγονται **κατασκευαστές** (constructors) της κλάσης. Σκοπός των constructors είναι η δέσμευση μνήμης για την κατασκευή του αντικειμένου καθώς και η διενέργεια κατάλληλων αρχικοποιήσεων. Ο προγραμματιστής μπορεί να ορίσει πολλούς constructors για μία κλάση αλλά για τη δημιουργία ενός αντικειμένου (στιγμιότυπου) μπορεί να καλέσει μόνο έναν από αυτούς. Φυσικά η επιλογή είναι ελεύθερη. Αν όμως ο προγραμματιστής **δεν** ορίσει κανένα constructor τότε η γλώσσα καλεί έναν constructor της υπερκλάσης (κάποιον που δεν θέλει παραμέτρους) ή δίνει μήνυμα λάθους αν δεν βρει κάποιον τέτοιο.

Ένα θέμα που προκύπτει είναι το πώς ορίζονται οι constructors. Στη JAVA ορίζονται όπως ακριβώς και οι απλές μέθοδοι με μόνη τη διαφορά ότι έχουν το **ίδιο** όνομα με την κλάση:

```
class A {
     int n;
     A() { // Constructor 1
          n=0;
     }
     A(int x) { // Constructor 2
          n = x;
     }
}
```

Σε αυτήν την περίπτωση οι constructors ξεχωρίζονται μεταξύ τους από τις λίστες των παραμέτρων τους. Επίσης, **οι constructors δεν έχουν επιστρεφόμενο τύπο**, ούτε void.

Η κατασκευή ενός νέου αντικειμένου γίνεται ως εξής:

```
A a;
A = new A();
 ή
a = new A(5);
```

3.4.6 **Καταστροφή Αντικειμένων** (Finalization)

Σε ορισμένες γλώσσες όπως η C++ και η Object PASCAL υποστηρίζεται και η χρήση των **καταστροφέων** (destructors) μιας κλάσης. Προφανώς οι destructors είναι μέθοδοι που τερματίζουν την ύπαρξη ενός στιγμιότυπου της αντίστοιχης κλάσης. Ωστόσο στην JAVA το χαρακτηριστικό αυτό δεν υποστηρίζεται αφού είναι αιτία λαθών. Φαντασθείτε για παράδειγμα ότι δύο δείκτες δείχνουν στο ίδιο αντικείμενο και ότι με χρήση του ενός από αυτούς τερματίζουμε την ύπαρξη του αντικειμένου. Τότε ο άλλος δείκτη θα δείχνει σε άκυρα δεδομένα (dangling pointer).

Η JAVA έχει να αντιπαρατάξει έναν άλλο ασφαλέστερο τρόπο για τερματισμό (finalization) της ύπαρξης ενός αντικειμένου. Ο τρόπος αυτός έχει ως εξής. Το runtime system της java εξετάζει κατά διαστήματα αν για κάθε αντικείμενο υπάρχει τουλάχιστο ένας δείκτης σε αυτό. Αν δεν υπάρχει τότε το αντικείμενο είναι άχρηστο (garbage) και η μνήμη που κατέχει απελευθερώνεται. Όμως προτού γίνει αυτό καλείται η μέθοδος ***void finalize()*** η οποία πρέπει να περιέχει κώδικα για clean-up. Για παράδειγμα αν το αντικείμενο έχει ακόμα ανοιχτές συνδέσεις στο δίκτυο τότε θα τις κλείσει. Μετά το τέλος της finalize() το αντικείμενο καταστρέφεται.

Η διαδικασία του εντοπισμού τον αντικειμένων - σκουπιδιών λέγεται ***garbage collection*** και ο μηχανισμός που το πραγματοποιεί είναι ενσωματωμένος στο JAVA runtime system.

3.4.7 **Μεταβλητές – Αναφορές στη** JAVA

Όσες μεταβλητές έχουν δηλωθεί σαν μεταβλητές ενός απλού τύπου δεδομένων (int, float, short κλπ) είναι ακριβώς όπως αυτές της C.

```
int a;
short b;
```

Οι μεταβλητές όμως που έχουν δηλωθεί σαν μεταβλητές κάποιας κλάσης είναι στην πραγματικότητα δείκτες στον αντικείμενο που τους ανατίθεται. Όμως σε αντίθεση με τη C δεν χρησιμοποιούνται οι τελεστές (* & ->) για να αναφερθούμε στα αντικείμενα. Αντίθετα οι μεταβλητές χρησιμοποιούνται σαν να μην ήταν δείκτες αλλά σαν να ήταν τα ίδια τα αντικείμενα.

Επίσης να σημειώσουμε ότι η JAVA δεν επιτρέπει την αριθμητική διευθύνσεων ή δεικτών. Για παράδειγμα:

```
A a = new A();
a++; // ΛΑΘΟΣ !!!
```

Ωστόσο

```
int a;
a++; // ΣΩΣΤΟ διότι το a εδώ είναι τύπου int και όχι
 δείκτης.
```

Για να είμαστε ακόμη πιο ακριβείς οι μεταβλητές που δείχνουν σε αντικείμενα (πχ A a) δεν είναι δείκτες, με την συμβατική έννοια, στα αντικείμενα, αλλά αναφορές (references) ή χειριστές (handlers) οι οποίοι δεν δείχνουν σε μία διεύθυνση της μνήμης αλλά σε κάποια δομή του runtime system. Έτσι το runtime system έχει τη δυνατότητα να ξέρει ανά πάσα στιγμή ποιο αντικείμενο αντιστοιχεί σε κάθε μεταβλητή και έτσι μπορεί να κάνει το garbage collection και την διαχείριση της μνήμης με μεγαλύτερη αποτελεσματικότητα και ασφάλεια.

Τελειώνοντας να πούμε ότι η αναφορά null μπορεί να ανατεθεί σε μεταβλητές αντικειμένων.

3.4.8 **Πολυμορφισμός** (Polymorfism)

Πολυμορφισμός είναι η δυνατότητα μίας μεθόδου να κάνει διαφορετικές ενέργειες ανάλογα με τον τύπο του αντικειμένου πάνω στο οποίο δρά. Είναι η τρίτη βασική αρχή του αντικειμενοστρεφούς προγραμματισμού.

Οι τύποι πολυμορφισμού είναι:

- Η υπερκάλυψη (overriding)
- Η υπερφόρτωση (overloading)
- Η δυναμική συσχέτιση μεθόδων (dynamic method binding)

Οι δύο πρώτες έχουν αναφερθεί προηγουμένως, η τρίτη φαίνεται στο ακόλουθο παράδειγμα:

Υποθέστε ότι τρεις υποκλάσεις (Cow, Dog και Snake) έχουν δημιουργηθεί χρησιμοποιώντας την αφηρημένη (abstract) κλάση Animal, με κάθε μια από τις οποίες να υλοποιεί την δική της μέθοδο speak().

```
public class AnimalReference
{
     public static void main(String[] args)
          Animal ref; // set up a reference for an Animal

          // make specific objects
          Cow aCow = new Cow("Bossy");
          Dog aDog = new Dog("Rover");
          Snake aSnake = new Snake("Earnie");

          // now reference each as an Animal
          ref = aCow;
          ref.speak();

          ref = aDog;
          ref.speak();

          ref = aSnake;
          ref.speak();
}
```

Μπορεί κανείς να παρατηρήσει ότι αν και η κλήση της μεθόδου speak() έγινε σε αναφορά τύπου Animal, το πρόγραμμα μπορεί να αποφανθεί για την σωστή μέθοδο (ανάλογα με την υποκλάση) και τον χρόνο εκτέλεσης. Αυτό ονομάζεται ***δυναμική συσχέτιση μεθόδων (dynamic method binding).***

3.4.9 **Ροή ενός προγράμματος**

Μέχρι τώρα παρουσιάσαμε ορισμένα παραδείγματα προγραμμάτων σε Java αλλά δεν αναλύσαμε περισσότερο τη λειτουργία τους και πολύ περισσότερο τη ροή τους. Η ροή ενός προγράμματος αναφέρεται στην διαδοχή της εκτέλεσης των εντολών στον επεξεργαστή (είτε είναι πραγματικός είτε εικονικός στη περίπτωση της Java με το JVM). Η κατανόηση και εμπέδωση του προγραμματισμού ως έννοια, προϋποθέτει την αναγνώριση της ροής ενός προγράμματος από τη μελέτη του πηγαίου του κώδικα. Από τη μελέτη αυτή μπορούμε να βγάλουμε τα εξής συμπεράσματα:

- Αναγνώριση των κλάσεων που χρησιμοποιεί το πρόγραμμα και των σχέσεων μεταξύ τους.

- Πληροφορίες για τα δεδομένα και τους τύπους δεδομένων που χρησιμοποιούνται σε κάθε κλάση.

- Πληροφορίες για τις συναρτήσεις/μεθόδους που χρησιμοποιεί η κάθε κλάση.

- Πληροφορίες για τη σειρά κλήσης κάθε μεθόδου των κλάσεων.

- Πληροφορίες για τη δομή της κάθε μεθόδου και πιθανές παραμετροποιήσεις της λειτουργίας τους.

- Από ποιά κλάση ξεκινάει το πρόγραμμα (δηλαδή σε ποιά κλάση περιέχεται η μέθοδος main()).

Απώτερος σκοπός είναι να συλλάβουμε τη γενική εικόνα της λειτουργίας του προγράμματος.
Θα θεωρήσουμε το ακόλουθο παράδειγμα για καλύτερη κατανόηση των προαναφερθέντων σημείων:

```
1 class ArgsExample {
2    public static void main(String args[]) {
3         for (int i=0; i < args.length; i++) {
4              if (args[i].equals(“-file”) == true) {
5                   if (i+1 < args.length)
6                   System.out.println(“FILE“+args[i+1]);
7              }
8         }
9    }
10 }
```

Στις γραμμές 1-3 η διαδικασία είναι γνωστή: ορίζεται η κλάση, δηλώνεται η βασική ρουτίνα main() και ξεκινάει το loop της εντολής for. Στη γραμμή 4 χρησιμοποιούμε την μέθοδο equals() της κλάσης String για να ελέγξουμε μία-μία τις παραμέτρους args[] αν κάποια είναι ίση με “-file”. Αν ισχύει κάτι τέτοιο (δηλαδή αν η equals() επιστρέψει true) τότε βεβαιωνόμαστε ότι υπάρχει επόμενη παράμετρος (το i+1 είναι μικρότερο του μεγέθους του πίνακα, γραμμή 5) και τυπώνουμε την επόμενη παράμετρο (γραμμή 6). Ο έλεγχος συνεχίζεται για τις υπόλοιπες παραμέτρους args[].

Αφού μεταγλωττίσουμε το πρόγραμμά μας με τη javac, έστω ότι το εκτελούμε με κάποιες παραμέτρους στη γραμμή εντολών:

```
# javac ArgsExample.java
# java ArgsExample one -file two three -file
FILE: two
```

Ο πίνακας args[] με την εκκίνηση του προγράμματος θα έχει τις εξής τιμές:

`args[0]`	`one`
`args[1]`	`-file`
`args[2]`	`two`
`args[3]`	`three`
`args[4]`	`-file`

Το μέγεθος του πίνακα απεικονίζεται με τη μεταβλητή args.length και στην περίπτωση αυτή είναι args.length = 5.

Στον ακόλουθο πίνακα, φαίνεται αυτή η ροή του προγράμματος αναλυτικά:

I	***args[i]***	***args[0].equals("-file")***	***i+1 < args.length***	***Print?***
0	`one`	False	-	No
1	`-file`	True	True	Yes
2	`two`	False	-	No
3	`three`	False	-	No
4	`-file`	true	false	no

Το "-" στη στήλη "i+1 < args.length" σημαίνει ότι δεν γίνεται ο έλεγχος για να μας ενδιαφέρει το αποτέλεσμα, δηλαδή το πρόγραμμα δεν εισέρχεται στην εντολή της γραμμής 5 για όλες τις επαναλήψεις του for loop (για όλες τις τιμές του i δηλαδή). Αυτό γιατί η εντολή if στη γραμμή 5 εκτελείται μόνο όταν η εντολή if της εντολής 4 επιστρέψει true.

4.Το μοντέλο του Θεματοστρεφούς Προγραμματισμού

Οι περισσότεροι προγραμματιστές στο μεγαλύτερο μέρος της καριέρας τους έχουν γαλουχηθεί στην ανάπτυξη συστημάτων λογισμικού με τη χρήση τεχνικών αντικειμενοστρεφούς προγραμματισμού (OOP). Η μέθοδος αυτή είναι η πλέον διαδεδομένη με κύριο χαρακτηριστικό τον κατακερματισμό ενός προβλήματος σε αντικείμενα που χαρακτηρίζονται από ιδιότητες (μέθοδους) και δεδομένα (μεταβλητές).

Παρά το γεγονός ότι ο OOP έχει μεγάλη επιτυχία στη διαμόρφωση και υλοποίηση πολύπλοκων συστημάτων λογισμικού, έχει και τα προβλήματά του. Η πρακτική εμπειρία με μεγάλα έργα έχει δείξει ότι οι προγραμματιστές ενδέχεται να αντιμετωπίσουν προβλήματα με τη διατήρηση του κώδικα τους, δεδομένου ότι όσο μεγαλύτερο το λογισμικό που υλοποιείται τόσο και πιο δύσκολος γίνεται ο ξεκάθαρος διαχωρισμός του έργου σε ενότητες (αντικείμενα), πράγμα και το οποίο αποτελεί τη βάση του OOP. Για παράδειγμα, μια μικρή αλλαγή σε μία επαναχρησιμοποιούμενη ενότητα μπορεί τελικά να προκαλέσει πολλές αλλαγές σε άλλες, ανεξάρτητες, ενότητες του κώδικα.

Τέτοιου είδους προβλήματα και πολλές άλλες ανησυχίες έρχεται να επιλύσει μία διαφορετική τεχνική, ο θεματοστρεφής ή κατά άλλους πτυχοστρεφής προγραμματισμός (AOP).

4.1 Τι είναι ο θεματοστρεφής προγραμματισμός

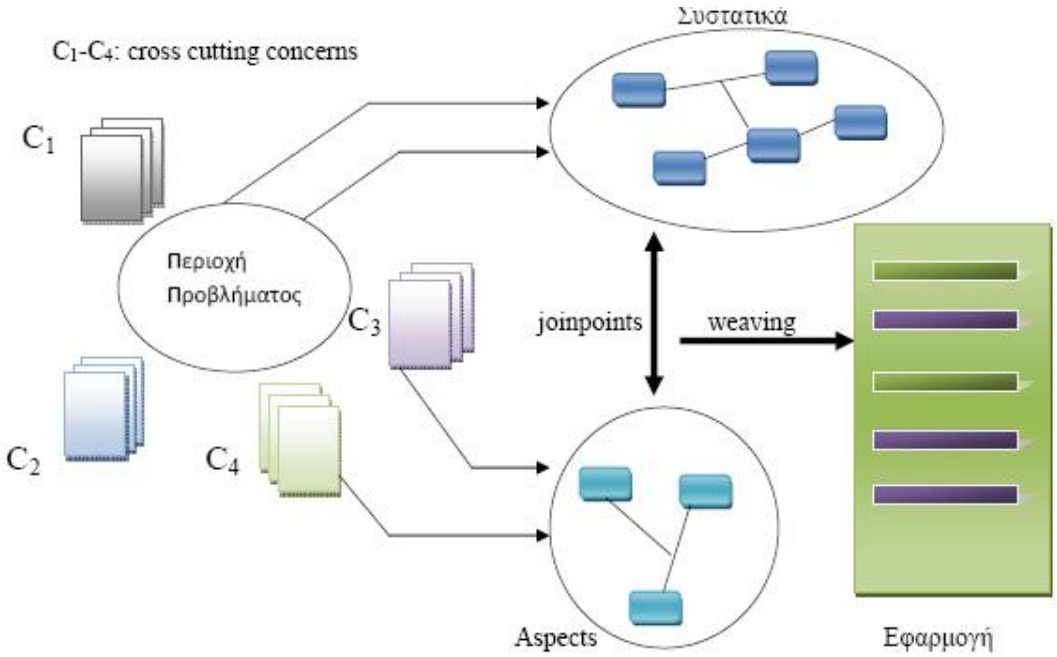
Ο θεματοστρεφής προγραμματισμός είναι μια νέα σχετικά μεθοδολογία για το διαχωρισμό ενός προβλήματος σε μεμονωμένες μονάδες που ονομάζονται *πτυχές* (aspects). Μια *πτυχή* είναι μια μονάδα που «κόβει» εγκάρσια τη λογική ροή μίας εφαρμογής (εξού και η καλύτερη απόδοση στα ελληνικά αντί της λέξης «θέμα»). Έτσι συμπυκνώνει συμπεριφορές (μέθοδους) που επηρεάζουν πολλαπλές κλάσεις και ενότητες του κωδικά μας και είναι επαναχρησιμοποιήσιμες αλλά η αλλαγή τους δεν έχει απολύτως καμία επίδραση στον υπόλοιπο κώδικα.

Αυτό που προσπαθεί να κάνει ο θεματοστρεφής προγραμματισμός είναι να προωθήσει τα επιθυμητά χαρακτηριστικά του αντικειμενοστρεφούς προγραμματισμού που είναι δύσκολο να εφαρμοστούν στις τρέχουσες τεχνολογίες OOP. Τα χαρακτηριστικά αυτά είναι τα εξής:

- 1:1 διαμορφωσιμότητα (modularity)
- Διαχωρισμός των θεμελιωδών concerns (cross-cutting concerns)
- Απόκρυψη πληροφοριών

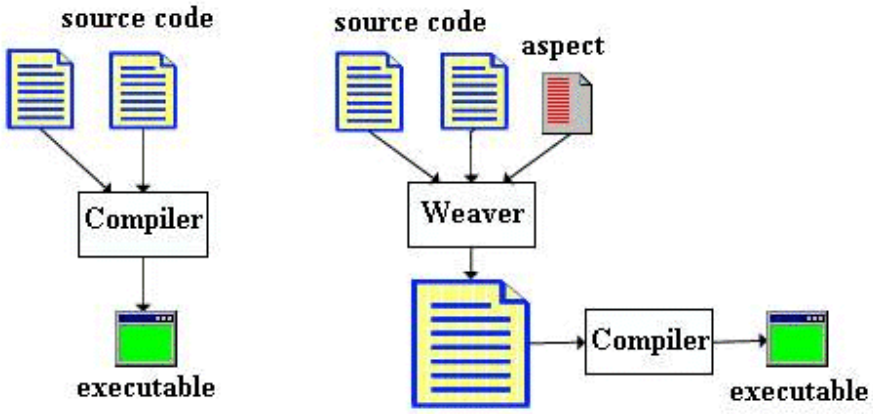
Με τον θεματοστρεφή προγραμματισμό, ξεκινάμε την υλοποίηση του έργου μας, χρησιμοποιώντας την αντικειμενοστρεφή γλώσσα μας (για παράδειγμα, Java), και στη συνέχεια απασχολούμαστε ξεχωριστά με τα θέματα που σχετίζονται στη δημιουργία επαναχρησιμοποιούμενου κώδικα που τέμνει εγκάρσια τη ροή του προγράμματος, με τη δημιουργία πτυχών. Τέλος, τόσο τα αντικείμενα όσο και οι πτυχές συνδυάζονται σε μια τελική εκτελέσιμη μορφή μέσω ενός «υφαντή πτυχών» (Aspect Weaver). Ως εκ τούτου, μια ενιαία πτυχή μπορεί να συμβάλει αποφασιστικά στην υλοποίηση μιας σειράς μεθόδων, ενοτήτων, ή αντικειμένων του λογισμικού, αυξάνοντας τόσο την δυνατότητα επαναχρησιμοποίησης όσο και της συντήρησης του κώδικα.

Με αυτόν τον τρόπο ο θεματοστρεφής προγραμματισμός δεν αντικαθιστά αλλά συμπληρώνει τον αντικειμενοστρεφή προγραμματισμό εισάγοντας ένα είδος λειτουργικότητας που συγκεντρώνει την εφαρμογή μίας cross-cutting concern σε μία μόνο μονάδα, την «πτυχή» (aspect).



Εικόνα 4.3 Προσθήκη πτυχών στο πρόγραμμα

Το παρακάτω σχήμα εξηγεί τη διαδικασία της ύφανσης. Θα πρέπει να σημειωθεί ότι ο αρχικός κώδικας δεν χρειάζεται να έχει επίγνωση για κάθε λειτουργία που έχει προσθέσει μία πτυχή. Για να αποκτήσει ξανά την αρχική του λειτουργικότητα, το μόνο που πρέπει να κάνει κάποιος είναι να ξαναμεταγλωττίσει τον κώδικα χωρίς την «πτυχή».



Εικόνα 4.4 Η διαδικασία της ύφανσης

Για να καταλάβουμε καλύτερα τον τρόπο με τον οποίο λειτουργεί ο θεματοστρεφής προγραμματισμός, θα παραθέσουμε ένα παράδειγμα:

```
public class TestClass {

     public void sayHello () {
          System.out.println ("Hello, AOP");
     }

     public void sayAnyThing (String s) {
          System.out.println (s);
     }

     public static void main (String[] args) {
          sayHello ();
          sayAnyThing ("ok");
     }
}
```

Στο παράδειγμά μας έχουμε μία κλάση με όνομα *TestClass* η οποία τυπώνει το μήνυμα “Hello AOP” και μία συνάρτηση *sayAnyThing* η οποία τυπώνει ένα μήνυμα που δίνει ο χρήστης.

Ας υποθέσουμε ότι θέλουμε να κάνουμε τις εξής αλλαγές στον παραπάνω κώδικα:

- Θέλουμε να τυπώνεται ένα μήνυμα πριν και μετά από κάθε κλήση της συνάρτησης TestClass.sayHello() και

- Με κάποιον τρόπο να ελέγχουμε ότι το όρισμα της συνάρτησης TestClass.sayAnyThing() είναι τουλάχιστον 3 χαρακτήρες. Σε περίπτωση που οι χαρακτήρες είναι λιγότεροι από 3 τότε ο κώδικας να τυπώνει ένα μήνυμα λάθους.

Ο κώδικας τώρα γίνεται ως εξής:

```
public aspect MyAspect {

      public pointcut sayMethodCall ():
      call (public void TestClass.say*() );

      public pointcut sayMethodCallArg (String str):
      call (public void TestClass.sayAnyThing(String))
      && args(str);
      before(): sayMethodCall() {
            System.out.println("\n TestClass." +
            thisJoinPointStaticPart.getSignature().getName() +
            "start..." );
      }
      after(): sayMethodCall() {
            System.out.println("\n TestClass." +
            thisJoinPointStaticPart.getSignature().getName() +
            " end...");
      }
      before(String str): sayMethodCallArg(str) {
            if (str .length() < 3) {
                  System.out.println ("Error: I can't say
                  words less than 3 characters");

                  return;
            }
      }
}
```

Στην πρώτη γραμμή ορίζεται μία πτυχή με το όνομα MyAspect με τον ίδιο τρόπο που ορίζεται και μία κλάση στη Java. Στις γραμμές 2 και 3 διευκρινίζεται πού ακριβώς μέσα στην TestClass θα γίνουν οι αλλαγές. Έχουμε εισάγει δύο jointpoints:

- Η κλήση στη συνάρτηση TestClass.sayHello και
- Η κλήση στη συνάρτηση TestClass.sayAnyThing

Το pointcut στη δεύτερη γραμμή:

```
public pointcut sayMethodCall ():
      call (public void TestClass.say*() );
```

το οποίο ονομάζεται `sayMethodCall` και διαλέγει οποιαδήποτε κλήση της συνάρτησης TestClass.sayHello. Επίσης, διαλέγει οποιαδήποτε public συνάρτηση TestClass με μηδέν ορίσματα και οποιαδήποτε λέξη που ξεκινάει με “say” (για παράδειγμα TestClass.sayBye) .

Τα pointcuts που χρησιμοποιούμε μας βοηθούν στον ορισμό των *advice*. Οι *advice* χρησιμοποιούνται για να καθορίσουν επιπλέον κώδικα που εκτελείται πριν, μετά ή γύρω από τα join points.

Την πρώτη αλλαγή την υλοποιούμε με την προσθήκη δύο advice, μία τύπου ***before*** και μία τύπου ***after***. Η before advice που βρίσκεται στην έβδομη γραμμή τυπώνει πριν από κάθε κλήση της συνάρτησης TestClass.sayHello() τη λέξη "start…" και η after advice που βρίσκεται στη δωδέκατη γραμμή τυπώνει μετά από κάθε κλήση της συνάρτησης TestClass.sayHello() τη λέξη "end…".

Τη δεύτερη αλλαγή που θέλαμε να κάνουμε στον κώδικα την υλοποιούμε ξανά με την προσθήκη μίας advice η οποία είναι τύπου before η οποία βρίσκεται στη δέκατη έβδομη γραμμή του κώδικά μας και η οποία εκτελεί έναν έλεγχο πριν από κάθε κλήση της συνάρτησης TestClass.sayAnyThing() με μία εντολή ελέγχου if. Σε περίπτωση που η λέξη που εισάγει ο χρήστης δεν έχει 3 χαρακτήρες και πάνω τότε τυπώνει ένα μήνυμα λάθους.

4.2 Πέρασμα από τον αντικειμενοστρεφή προγραμματισμό στο θεματοστρεφή προγραμματισμό

Στην ενότητα αυτή θα παρουσιάσουμε δύο παραδείγματα ώστε να γίνει πιο κατανοητή η ανάγκη του θεματοστρεφούς προγραμματισμού.

Στο *πρώτο* παράδειγμα υποθέτουμε ότι έχουμε μία κλάση γραφικών, η οποία έχει πολλές μεθόδους «set». Μετά από την εκτέλεση κάθε μεθόδου ***set***, τα χαρακτηριστικά των γραφικών αλλάζουν με αποτέλεσμα να αλλάζουν και τα αντίστοιχα γραφικά που εμφανίζονται στην οθόνη. Για να γίνει κάτι τέτοιο είναι απαραίτητη η προσθήκη της συνάρτησης "update" μετά από κάθε μέθοδο ***set***.

```
void set…(…){
            :
            :
           Display.update();

}
```

Αν ο αριθμός των μεθόδων ***set*** που υπάρχουν στο πρόγραμμα είναι μικρός, τότε δε δημιουργείται πρόβλημα. Ωστόσο θα δημιουργηθεί πρόβλημα, αν υπάρχει μεγάλος αριθμός μεθόδων ***set***. Θα πρέπει να είμαστε απολύτως σίγουροι ότι δεν έχει ξεχαστεί κάποια από τις ***set*** μεθόδους. Η διαδικασία προσθήκης τόσο μεγάλου όγκου νέου κώδικα είναι αρκετά επίπονη και χρονοβόρα.

Ο θεματοστρεφής προγραμματισμός έρχεται για να λύσει το παραπάνω πρόβλημα, χωρίς την ανάγκη προσθήκης μεγάλου αριθμού γραμμών κώδικα. Αρκεί η προσθήκη της παρακάτω πτυχής:

```
after():set() {
          Display.update () ;

}
```

Με τον τρόπο αυτό, είναι εγγυημένο ότι δεν έχει ξεχαστεί κάποια *set* μέθοδος, αλλά και η προσθήκη του νέου απαραίτητου κώδικα γίνεται εύκολα και γρήγορα. Αυτό που κάνει η πτυχή είναι να δίνει στο σύστημα την εντολή να εκτελέσει τον κατάλληλο κώδικα μετά από την εκτέλεση του *pointcut set*.

Το pointcut *set* θα έχει ως εξής:

```
pointcut set (): execution (* set*(*)) &&
this(MyGraphicsClass) && within (com.company.*);
```

Ο παραπάνω κώδικας σημαίνει ότι μία μέθοδος είναι pointcut[1] τύπου *set* εάν είναι μέθοδος της κλάσης MyGraphicsClass, αν αυτή η κλάση είναι μέρος του πακέτου com.company και αν το όνομά της είναι set*, ανεξαρτήτως από τι ακολουθεί (αστερίσκος δίπλα στη λέξη set), τι επιστρέφει (πρώτος αστερίσκος) και τι παραμέτρους δέχεται (τρίτος αστερίσκος).

Το δεύτερο παράδειγμα αφορά κάποιον κώδικα που ακολουθεί το μοντέλο του αντικειμενοστρεφούς προγραμματισμού και τον τρόπο με τον οποίο μπορούμε να επέμβουμε σε αυτόν χρησιμοποιώντας τις τεχνικές του θεματοστρεφούς προγραμματισμού.

Έστω λοιπόν ο παρακάτω αντικειμενοστρεφής κώδικας (σε JAVA):

```
public class MessageCommunicator{
      public static void deliver(String message){
            System.out.println(message);
      }
      public static void deliver(String person,String
      message){
```

[1] Οι έννοιες *πτυχή*, *pointcut*, *advice* που αναφέρονται στην παρούσα ενότητα θα παρουσιαστούν με μεγαλύτερη λεπτομέρεια στην επόμενη ενότητα.

```
                System.out.println(person + "," + message);
        }
}
public class Test{
        public static void main(String[] args){
                MessageCommunicator .deliver("Wanna learn
        AspectJ?");
                MessageCommunicator .deliver("Yes", "AspectJ
                has fun!!!");
        }
}
```

Αν τρέξουμε τον παραπάνω κώδικα, η έξοδος που θα πάρουμε θα είναι η εξής:

```
> javac MessageCommunicator.java Test.java
> java test
Wanna learn AspectJ?
Yes, AspectJ has fun!!!
```

Ας υποθέσουμε ότι στο παράδειγμά μας θέλουμε να προσθέσουμε τη λέξη “Hello” στην αρχή κάθε απάντησης. Για να το πετύχουμε αυτό θα μπορούσαμε να τροποποιήσουμε την κλάση MessageCommunicator, αλλά κάτι τέτοιο θα ήταν πολύ χρονοβόρο, καθώς θα έπρεπε να ελέγξουμε όλα τα απαραίτητα αρχεία ψάχνοντας το σχετικό κώδικα.

Ένας πιο απλός τρόπος είναι να χρησιμοποιηθεί μια *πτυχή*. Έτσι λοιπόν, θα έπρεπε να δηλώσουμε ένα ***pointcut*** το οποίο θα καταχωρούσε όλες τις κλήσεις στη συνάρτηση ***deliver***. Επιπλέον, θα χρειαζόταν να δηλώσουμε μία ***advice*** η οποία θα εκτελούνταν πριν από κάθε εκτέλεση της συνάρτησης ***deliver***.

Ας δούμε λοιπόν τον αντίστοιχο κώδικα.

Κώδικας της πτυχής

```
public aspect MannersAspect{
        pointcut deliverMessage(): call(*MessageCommunicator.deliver);
        before() : deliverMessage(){
                System.out.print("Hello!");
        }
}
```

Αν τώρα εκτελέσουμε το παραπάνω πρόγραμμα, η έξοδος που θα προκύψει θα είναι η εξής:

```
> ajc MessageCommunicator.java MannersAspect.java
Test.java
> java test
Hello! Wanna learn AspectJ?
Hello! Yes, AspectJ has fun!!!
```

Θα πρέπει στο σημείο αυτό να τονίσουμε ότι οι προγραμματιστές δε θα πρέπει να διχάζονται από το δίλημμα χρήσης θεματοστρεφούς ή αντικειμενοστρεφούς προγραμματισμού. Ο θεματοστρεφής προγραμματισμός δεν είναι κάτι τελείως διαφορετικό από τον αντικειμενοστρεφή. Δε δημιουργήθηκε για να υπερκαλύψει τον αντικειμενοστρεφή προγραμματισμό ή για να τον θέσει στο περιθώριο, αλλά για να χρησιμοποιηθεί σε συνδυασμό με αυτόν, να συμπληρώσει τυχόν αδυναμίες του και να βελτιώσει τη λειτουργικότητά του. Στην ουσία, κάποια συγκεκριμένη λειτουργία μπορεί να επιτευχθεί και μόνο με τη χρήση αντικειμενοστρεφούς προγραμματισμού. Αυτό που επιτυγχάνεται με τις τεχνικές του θεματοστρεφούς προγραμματισμού είναι η αποφυγή της χρονοβόρας διαδικασίας πολλαπλών γραμμών κώδικα για την προσθήκη κάποιας επιπλέον λειτουργικότητας.

4.3 Βασικές έννοιες του θεματοστρεφούς προγραμματισμού

4.3.1 Πτυχές (Aspects)

Μία πτυχή είναι μια μονάδα που μπαίνει εγκάρσια στη λογική ροή μίας εφαρμογής διακόπτωντάς την για να εκτελέσει μία λειτουργία. Οι πτυχές καθορίζονται από τμηματικές πληροφορίες άλλων μονάδων και υπάρχουν στο σχεδιασμό αλλά και στην εφαρμογή.

Αυτό που κάνει ουσιαστικά μία πτυχή είναι να αλλάζει τη συμπεριφορά κάποιων κλάσεων αλλά και αντικειμένων ανεξάρτητα από την ιεραρχία κληρονομικότητας ενθυλακώνοντας συμπεριφορές μέσα σε επαναχρησιμοποιήσιμες μονάδες.

Οι πτυχές μπορούν να εφαρμοστούν σε δύο περιπτώσεις:

- Κατά τη διάρκεια του χρόνου εκτέλεσης και
- Κατά τη διάρκεια του ελέγχου για συντακτικά λάθη (compile)

Για να καταλάβουμε καλύτερα το τι είναι μία πτυχή και πώς ορίζεται παραθέτουμε το παρακάτω παράδειγμα:

```
aspect DisplayUpdating {
     pointcut move():
          call(void Line.setP1(Point)) ||
          call(void Line.setP2(Point));
          after() returning: move() {
               Display.update();
          }
}
```

Στο παράδειγμα αυτό ορίζουμε μία πτυχή (aspect) με τα εξής χαρακτηριστικά:

- Το όνομά της είναι `DisplayUpdating`
- Έχει σαν μέλη ένα *pointcut* και μία *advice*
- Το όνομα του pointcut είναι `move()`
- Η advice είναι τύπου `after returning`

Το πρόβλημα που υπάρχει είναι ότι οι cross cutting concerns δεν ενθυλακώνονται κατάλληλα στις ενότητές τους, με αποτέλεσμα να αυξάνεται η πολυπλοκότητα του συστήματος και να κάνει ιδιαίτερα δύσκολη την εξέλιξή του.

Ο θεματοστρεφής προγραμματισμός προσπαθεί να λύσει αυτό το πρόβλημα επιτρέποντας στον προγραμματιστή να εκφράζει τις cross cutting concerns σε ξεχωριστές ανεξάρτητες μονάδες. Αυτές οι ξεχωριστές και ανεξάρτητες μονάδες είναι οι πτυχές.

4.3.2 Jointpoint

Ένα jointpoint σημαίνει δύο πράγματα:

- Γενικά, σημείο στον έλεγχο ροής του προγράμματος

- Ειδικά για το θεματοστρεφή προγραμματισμό, σημείο που καθορίζει πού πρέπει να εκτελεστεί ο κώδικας που περιέχει μία *advice*.

Ένα joinpoint είναι ένα σημείο εκτέλεσης στον βασικό κώδικα στον οποίο εφαρμόζονται οι advice που καθορίζονται σε ένα pointcut. Στην ουσία, τα jointpoints είναι σημεία συνάντησης του κώδικα του κυρίου προγράμματος με τον κώδικα των aspects. Είναι ευκόλως αναγνωρίσιμα μέσα σε ένα πρόγραμμα. Για παράδειγμα, jointpoints μπορεί να είναι τα παρακάτω:

- Κλήση σε μία μέθοδο
- Εκτέλεση μιας μεθόδου
- Ανάθεση τιμής σε μία μεταβλητή
- Βρόγχος επανάληψης
- Χειριστής εξαιρέσεων

4.3.3 Pointcut

Ένα pointcut είναι ένα σύνολο από joinpoints. Ελέγχει εάν ένα joinpoint ταιριάζει στον κώδικα. Για οποιαδήποτε εκτέλεση προγράμματος τα pointcuts υποδεικνύουν ένα σύνολο από joinpoints χωρίς αυτό να σημαίνει ότι θα επιλεχθούν από ένα pointcut όλα τα joinpoints του που υπάρχουν στον κώδικα. Μπορεί να επιλεχθούν ορισμένα ή ακόμα και ένα μόνο joinpoint.

Τα pointcuts έχουν τη δυνατότητα να συνδυάζονται μεταξύ τους με τη βοήθεια των δυαδικών τελεστών &&, || και !.

Παράδειγμα pointcut:

```
pointcut move():
     call(void Line.setP1(Point)) ||
     call(void Line.setP2(Point));
```

Το συγκεκριμένο pointcut έχει όνομα move ενώ ανάμεσα στις δύο παρενθέσεις που ακολουθούν εισάγονται οι παράμετροι του pointcut. Επίσης, παρατηρούμε ότι στο κυρίως σώμα του pointcut υπάρχει ο δυαδικός τελεστής || που σημαίνει ότι αρκεί να εκτελεστεί μία από τις δύο κλήσεις.

Γενικά, η μορφή που έχει ένα pointcut είναι η ακόλουθη:

```
Pointcut όνομα (παράμετροι): προσδιοριστής[2]

        (ένα joinpoint);
```

[2] Ο προσδιοριστής θα καθορίσει αν το joinpoint που ακολουθεί ταιριάζει στον κώδικα.

Ορισμένα παραδείγματα υπαρχόντων pointcuts είναι τα εξής:

- **execution(**συνάρτηση ή κατασκευαστής**)**

  execution(public void Foo-> συνάρτηση**()**
  execution(public Foo->new())

  Χρησιμοποιείται όταν θέλουμε να συμβεί μία παρεμβολή κάθε φορά που καλείται μία συνάρτηση ή ένας κατασκευαστής.

- **get (**δήλωση πεδίου**)**
  get(public int Foo->όνομα πεδίου**)**

  Χρησιμοποιείται όταν θέλουμε να συμβεί μία παρεμβολή κάθε φορά που ένα συγκεκριμένο πεδίο είναι προσβάσιμο για επεξεργασία.

4.3.4 Advice

Μία advcice είναι μία λειτουργία, μέθοδος ή διαδικασία που εφαρμόζεται σε ένα συγκεκριμένο joinpoint του προγράμματος το οποίο πρέπει να έχει επιλεχθεί από κάποιο pointcut. Ο κώδικας της advice πρέπει να παρεμβληθεί πριν, μετά ή ακόμα και αντί του υπάρχοντος κώδικα για κάποιο σύνολο από joinpoints.

Η πρακτική χρήση των advice είναι γενικά να τροποποιήσουν ή να επεκτείνουν τη συμπεριφορά λειτουργιών που δε μπορούν εύκολα να τροποποιηθούν ή να επεκταθούν.

Υπάρχουν τρία είδη advice: *before, after, around*.

- Before advice: εκτελείται όταν φτάσει σε ένα joinpoint πριν το πρόγραμμα το προχωρήσει στην εκτέλεσή του και δεν έχει την ικανότητα να αποτρέψει τη ροή εκτέλεσης προς το joinpoint:

```
before(): log(){
        System.out.println("logging before method call");
}
```

Η λέξη before δηλώνει το είδος της advice. Αυτό που ακολουθεί μετά από το όνομα της advice είναι το pointcut στο οποίο αναφέρεται η advice, στην περίπτωσή μας είναι το log(). Μέσα στις αγκύλες βρίσκεται το κυρίως σώμα της advice, δηλαδή αυτό που πρέπει να εκτελεστεί πριν το pointcut log().

- **After advice:** εκτελείται ανεξάρτητα από τον τρόπο με τον οποίο τελειώνει ένα joinpoint και χωρίζεται σε δύο κατηγορίες:

- *after returning advice* η οποία εκτελείται μετά την κανονική ολοκλήρωση του joinpoint

```
after() returning: log() {
     System.out.println("logging after returning");

}
```

- *after throwing advice* η οποία εκτελείται όταν μία μέθοδος σταματήσει εξαιτίας μίας εξαίρεσης

```
after() throwing: log() {
  System.out.println("logging after throwing");

}
```

Η after advice εκτελείται σε ένα συγκεκριμένο joinpoint μόλις το πρόγραμμα τελειώσει την εκτέλεσή του

```
after(): log() {
       System.out.println("logging after returning/throwing");

}
```

- **Around advice:** αποτελεί το πιο ισχυρό είδος advice. Η around device μπορεί να εκτελέσει μία συμπεριφορά πριν και μετά την κλήση της μεθόδου και έχει τη δυνατότητα να αποφασίσει αν το πρόγραμμα θα συνεχίσει με την εκτέλεση του joinpoint ή όχι.

```
around(): log() {
     System.out.println("logging 'around' method call");
     proceed();

}
```

Από τα παραπάνω μπορούμε να συμπεράνουμε ότι οι μορφές σύνταξης των advice είναι οι ακόλουθες:

- Before advice

```
before(παράμετροι) : pointcut(παράμετροι)
     {κυρίως σώμα}
```

- After advice
  - after returning advice

```
after(παράμετροι) returning []: pointcut(παράμετροι)
      {κυρίως σώμα}
```

  - after throwing advice

```
after(παράμετροι) throwing []: pointcut(παράμετροι)
      {κυρίως σώμα}
```

  - around advice

```
around(παράμετροι) [throws typelist] :
pointcut(παράμετροι)
      {κυρίως σώμα}
```

4.3.5 Cross cutting concerns

Οι cross cutting concerns είναι δευτεροβάθμιες απαιτήσεις που μοιράζονται μεταξύ των μονάδων ενός προγράμματος. Αποτελούν και οι ίδιες πτυχές του κώδικα του προγράμματος που επηρεάζουν άλλες concerns.

Στις περισσότερες των περιπτώσεων, οι cross cutting concerns είναι δύσκολο να διαχωριστούν από το υπόλοιπο πρόγραμμα. Αυτό έχει ως αποτέλεσμα το διπλασιασμό του κώδικα ή έναν κώδικα πολύπλοκο και μπερδεμένο.

Συνεπώς, με τις cross cutting concerns υφίσταται το πρόβλημα της συνοχής του κώδικα:

- Όταν μία concern διαδίδεται σε ένα μεγάλο αριθμό κλάσεων ή μεθόδων, το αποτέλεσμα είναι ότι ο κώδικας που εφαρμόζει η concern να είναι διεσπαρμένος μεταξύ πολλών συστατικών του προγράμματος, κάτι που τον κάνει δύσκολο στην κατανόηση και συντήρησή του.

- Όταν σε ένα πρόγραμμα οι concerns που υπάρχουν διαδίδονται σε πολλά συστατικά του προγράμματος, τότε τα συστατικά αυτά καταλήγουν πλεγμένα με πολλές concerns, όπως ασφάλεια, καταγραφή, κλπ. Για να αλλαχθεί επομένως κάποιο από αυτά τα συστατικά, προϋποθέτει ότι έχουν κατανοηθεί πλήρως όλες οι πλεγμένες concerns.

Ορισμένα παραδείγματα concerns είναι τα ακόλουθα:

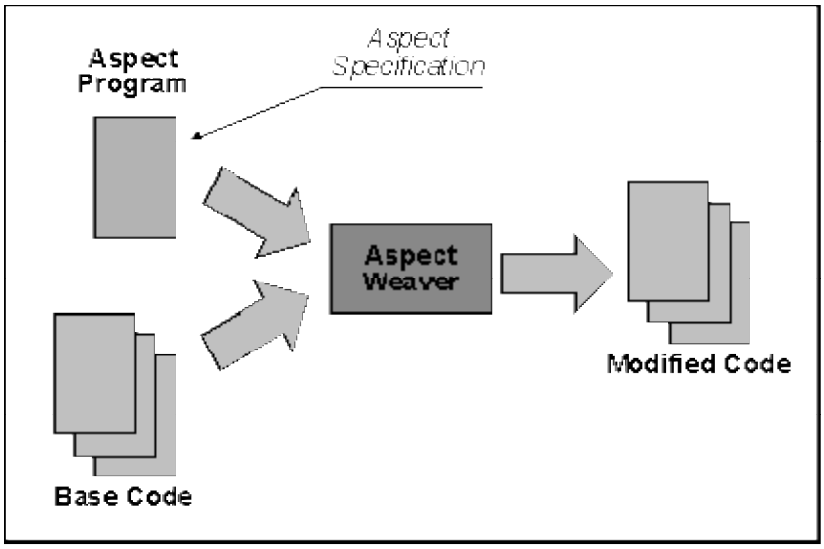
- Διαχείριση εξαιρέσεων
- Καταγραφή

- Επικύρωση
- Ασφάλεια

4.3.6 Aspect weaver

Ο aspect weaver είναι μία μεταπρογραμματική λειτουργία που είναι σχεδιασμένη να παίρνει οδηγίες (οι οποίες δεν είναι τίποτε άλλο από τα advice) από τις πτυχές και να παράγει τον τελικό κώδικα εφαρμογής. Συγχωνεύοντας πτυχές και κλάσεις ο aspect weaver δημιουργεί είτε πλεγμένες κλάσεις είτε πλεγμένο κώδικα.

Για να κατανοήσουμε καλύτερα τη λειτουργία του aspect weaver ας δούμε το παρακάτω σχήμα.



Εικόνα 5 Aspect weaver

Παρατηρούμε ότι ο βασικός κώδικας είναι οργανωμένος σε ενότητες, ενώ υπάρχει και μια πτυχή που καθορίζει έναν ιδιαίτερο κώδικα μετασχηματισμού.

Ο aspect weaver είναι στην ουσία ένα πρόγραμμα παρόμοιο με το μεταγλωττιστή, που διαβάζει τον κώδικα της πτυχής και τον χρησιμοποιεί για να τροποποιήσει τον βασικό κώδικα και ταυτόχρονα να παράγει έναν καινούριο, ο οποίος θα εφαρμόζει την αντίστοιχη πτυχή. Ο βασικός κώδικας και ο τροποποιημένος είναι γραμμένοι στην ίδια γλώσσα.

4.3.7 Dynamic weaving

Το dynamic weaving είναι μία από τις τεχνικές που χρησιμοποιεί ο θεματοστρεφής προγραμματισμός και επιτρέπει στην πτυχή να δεσμευτεί από το πρόγραμμα σε χρόνο

εκτέλεσης (runtime). Είναι μία τεχνική παρόμοια με την τεχνική της δυναμικής σύνδεσης που εφαρμόζει ο αντικειμενοστρεφής προγραμματισμός.

Το dynamic weaving μπορεί να διαχωριστεί σε δύο κατηγορίες: load time weaving και run time weaving.

- **load time weaving**: Οι μεταγλωτισμένες πτυχές και οι κλάσεις αλληλεπιδρούν κατά τη διάρκεια φόρτωσης των κλάσεων

- **run time weaving**: οι πτυχές πλέκονται με κλάσεις που έχουν ήδη φορτωθεί. Αυτό είναι ιδιαίτερα χρήσιμο όταν μία νέα concern πρόκειται να προστεθεί σε πολλές ενότητες (modules) και δε θα δημιουργήσει πρόβλημα στην εφαρμογή.

Υπάρχουν τέσσερις απαιτήσεις για το αποδοτικό dynamic weaving στη JAVA:

- Να υπάρχει αποδοτικότητα του κώδικα κάτω από κανονικές διαδικασίες, δηλαδή να μην υπάρχει πλεγμένος κώδικας
- Ασφάλεια
- Αποδοτική εκτέλεση του κώδικα των advice
- Ευελιξία

4.4 Πλεονεκτήματα και μειονεκτήματα του θεματοστεφούς προγραμματισμου

Τα πλεονεκτήματα που έχει το μοντέλο του θεματοστρεφούς προγραμματισμού είναι τα εξής:

Διαχωρισμός των συστατικών ενός προγράμματος με χρήση των πτυχών: Ο διαχωρισμός αυτός βοηθάει στην καλύτερη κατανόηση του λογισμικού λόγω του υψηλού επιπέδου της αφαίρεσης, συνεισφέρει στην ευκολότερη ανάπτυξη αλλά και διατήρηση του προγράμματος επειδή αποτρέπεται το «πλέξιμο» του κώδικα και αυξάνει τη δυνατότητα επαναχρησιμοποίησης τόσο για τον κώδικα του κυρίως προγράμματος, όσο και για τον κώδικα των πτυχών.

Εφαρμογή των cross cutting concerns: ο θεματοστρεφής προγραμματισμός παρέχει τρόπους καθορισμού και ενθυλάκωσης των cross cutting concerns σε ένα σύστημα.

Ευκολότερη εξέλιξη των συστημάτων: ορισμένες ενότητες του κώδικα μπορεί να μη γνωρίζουν το ποιες ή πόσες cross cutting concerns υπάρχουν στο σύστημα· επομένως είναι εύκολη η προσθήκη επιπλέον λειτουργικότητας με τη δημιουργία νέων πτυχών.

Μεγαλύτερη δυνατότητα επαναχρησιμοποίησης του κώδικα: επειδή ο θεματοστρεφής προγραμματισμός εφαρμόζει κάθε πτυχή ως ξεχωριστή ενότητα, μία μεμονομένη ενότητα δεν συνδέεται τόσο στενά με το υπόλοιπο πρόγραμμα.

Μειωμένο κόστος μελλοντικής εφαρμογής

Δυνατότητα αυτόματου ελέγχου του κώδικα χωρίς να παραποιηθεί.

Παρά τα πολλά και σημαντικά πλεονεκτήματα που εμφανίζει η θεματοστρεφής προσέγγιση, όπως είναι λογικό παρουσιάζει και κάποια μειονεκτήματα, όπως είναι τα παρακάτω:

- Η ροή του προγράμματος είναι δύσκολο να παρακολουθηθεί.
- Η υποστήριξη από εργαλεία είναι σχετικά μικρή.
- Τα τυχόν λάθη στις cross cutting concerns μπορεί να οδηγήσουν σε αποτυχία ολόκληρου του προγράμματος.
- Οι αλλαγές που μπορεί να γίνουν στα joinpoints ενός προγράμματος και δεν τις περιμένει ο χρήστης, μπορεί επίσης να οδηγήσουν σε αποτυχία.
- Η χρήση του θεματοστρεφούς προγραμματισμού για την προσθήκη επιπλέον κώδικα, αν δε γίνει σωστά μπορεί να οδηγήσει σε αναίρεση της ασφάλειας που ίσχυε.

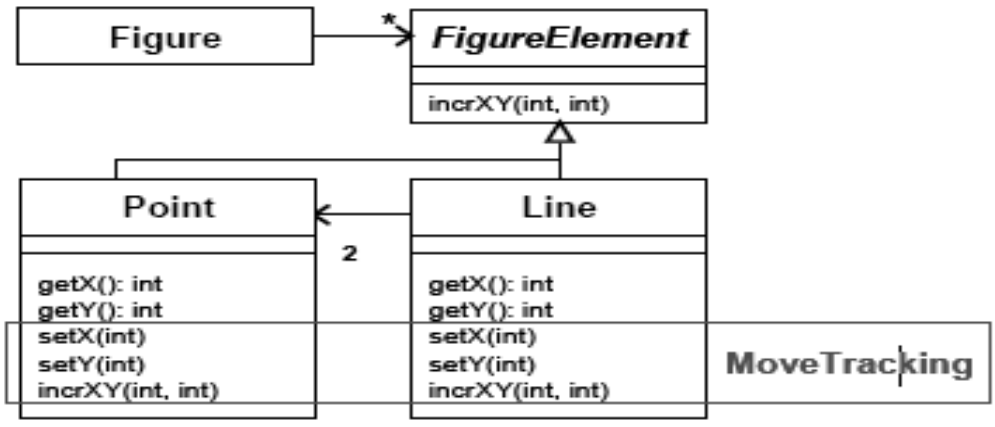
5.Εισαγωγή στην AspectJ

Η AspectJ επεκτείνει τη Java με υποστήριξη για δύο είδη εγκάρσιων υλοποιήσεων. Η πρώτη καθιστά δυνατό τον καθορισμό επιπλέον υλοποίησης που να τρέχει σε ορισμένα καλά ορισμένα σημεία στην εκτέλεση του προγράμματος. Καλούμε αυτή τη διαδικασία *«δυναμικό εγκάρσιο μηχανισμό»*. Η δεύτερη δίνει τη δυνατότητα να οριστούν νέες λειτουργίες για τους υπάρχοντες τύπους. Καλούμε αυτή τη διαδικασία *«στατικό εγκάρσιο μηχανισμό»* διότι επηρεάζει τους στατικούς τύπους του προγράμματος.

Η δυναμική διατομή στην AspectJ βασίζεται σε ένα μικρό αλλά ισχυρό σύνολο κατασκευών. Τα σημεία σύνδεσης (join points) είναι καλώς ορισμένα σημεία στην εκτέλεση του προγράμματος· Τα Pointcuts αποτελούν μέσα αναφοράς σε συλλογές σημείων σύνδεσης και συγκεκριμένων τιμών σε αυτά τα σημεία· advice είναι κατασκευές που μοιάζουν με μεθόδους που χρησιμοποιούνται για τον καθορισμό επιπλέον συμπεριφορών στα σημεία ένωσης· και πτυχές μονάδες τμηματικής υλοποίησης διατομών, αποτελούμενων από pointcuts, advice, και συνηθισμένων στη Java δηλώσεων μελών.

5.1 Το μοντέλο του Join Point

Στην Εικόνα 5.6 παρουσιάζονται κάποια join points χρησιμοποιώντας ως παράδειγμα έναν απλό επεξεργαστή εικόνων. Με βάση αυτές τις κλάσεις, εκτελώντας τις τρεις πρώτες γραμμές κώδικα της Εικόνα 5.7 κατασκευάζονται τα αντικείμενα που παρουσιάζονται στην εικόνα.

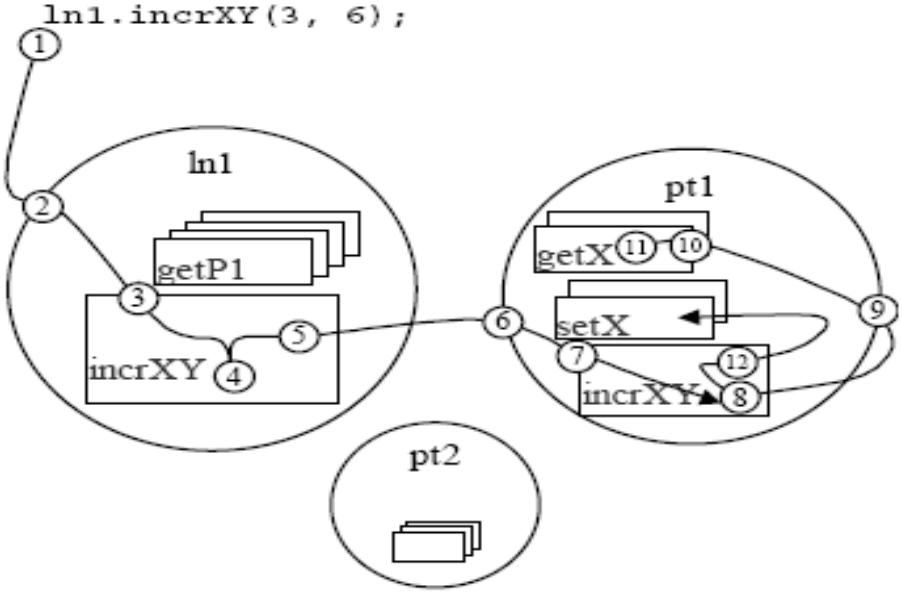


Εικόνα 5.6 Περιγραφή με UML ενός απλού επεξεργαστή εικόνων. Το κουτί με τίτλο "MoveTracking" δείχνει μία πτυχή που τέμνει εγκάρσια μεθόδους στις κλάσεις Point και Line.

Στην εικόνα αυτή, οι μεγάλοι κύκλοι αναπαριστούν αντικείμενα, τα τετράγωνα αναπαριστούν μεθόδους και οι μικροί αριθμημένοι κύκλοι αντιπροσωπεύουν join points. Εκτελώντας την τελευταία γραμμή κώδικα, ξεκινάει ένας υπολογισμός που εξελίσσεται διαμέσου των Join points. Σε κάθε περίπτωση, το join point προσεγγίζεται ακριβώς πριν εκτελεστεί η ενέργεια που περιγράφεται. Ο έλεγχος επιστρέφεται στο πρόγραμμα όταν εκτελεστεί η ενέργεια.

```
Point pt1 = new Point(0, 0);
Point pt2 = new Point(4, 4);
Line  ln1 = new Line(pt1, pt2);

ln1.incrXY(3, 6);
```



Εικόνα 5.7 Οι πρώτες τρεις γραμμές κώδικα κατασκευάζουν τα αντικείμενα που βρίσκονται μέσα στους μεγάλους κύκλους. Τα τετράγωνα αναπαριστούν μεθόδους. Εκτελώντας την τελευταία γραμμή κώδικα, ξεκινάει ένα υπολογισμός που περιλαμβάνει τη σειρά των join points

Τα join points της εικόνας και η διαδικασία περάσματος του ελέγχου:

1. Ένα join point καλέσματος μεθόδου που αντιστοιχεί στη μέθοδο `incrXY` καλείται πάνω στο αντικείμενο `ln1`.
2. Ένα join point αποδοχής καλέσματος μεθόδου στο οποίο το αντικείμενο `ln1` δέχεται το κάλεσμα της `incrXY`.
3. Ένα join point εκτέλεσης μεθόδου στο οποίο η μέθοδος `incrXY` που ορίστηκε στην κλάση `Line` ξεκινάει να εκτελείται.

4. Ένα join point λήψης πεδίου όπου διαβάζεται το `_p1` πεδίο του αντικειμένου `ln1`.

   .....

11. Ένα join point λήψης πεδίου όπου διαβάζεται το `_x` πεδίο του σημείου `pt1`.

Ο έλεγχος επιστρέφει στα join points 11, 10, 9, 8.

12. Ένα join point καλέσματος μεθόδου όπου καλείται η μέθοδος setX πάνω στο p1

    …….

Ο έλεγχος επιστρέφει πίσω μέσω των 3, 2, 1.

Τα διαφορετικά είδη join points που παρέχονται από την AspectJ παρουσιάζονται στον Πίνακας 5.1. Σημειώνουμε ότι ακόμα και αν η AspectJ ορίζει αρκετά είδη Join points, λίγα από αυτά αρκούν για πολλά προγράμματα και όλα τα είδη συμπεριφέρονται το ίδιο ως προς τα άλλα χαρακτηριστικά της γλώσσας, ουσιαστικά μειώνοντας έτσι την πολυπλοκότητα εκμάθησης προγραμματισμού σε AspectJ.

Πίνακας 5.1 Τα δυναμικά join points της AspectJ

Είδος join point
Κάλεσμα μεθόδου Κάλεσμα δημιουργού
Αποδοχή καλέσματος μεθόδου Αποδοχή καλέσματος δημιουργού
Εκτέλεση μεθόδου Εκτέλεση δημιουργού
Λήψη πεδίου
Σετάρισμα πεδίου
Εκτέλεση διαχειριστή εξαιρέσεων
Αρχικοποίηση κλάσης
Αρχικοποίηση αντικειμένου

5.2 **Προσδιοριστές** pointcut

Ένα pointcut είναι ένα σύνολο από join points, συν κάποιες από τις τιμές που παίρνουν κατά την εκτέλεση αυτά τα join points. Η AspectJ περιλαμβάνει αρκετούς στοιχειώδεις προσδιοριστές pointcuts. Οι προγραμματιστές μπορούν να τους συνθέσουν ώστε να ορίσουν προσδιοριστές pointcuts με ή χωρίς όνομα.

Ένας απλός τρόπος για να σκεφτόμαστε τους προσδιοριστές pointcuts είναι ως προς αντιστοιχίζονται συγκεκριμένα join points κατά το χρόνο εκτέλεσης του προγράμματος. Γοα παράδειγμα, ο προσδιοριστής

```
receptions(void Point.setX(int))
```

ταιριάζει με όλα τα join points αποδοχής καλέσματος μεθόδου στα οποία ο JAVA προσδιορισμός του καλέσματος μεθόδου είναι ο

```
void Point.setX(int)
```

Διαισθητικά, αυτό συμβαίνει κάθε φορά που ένα σημείο λαμβάνει μία κλήση για αλλαγή της x συντεταγμένης του. Ομοίως, το

```
receptions(void FigureElement.incrXY(int, int))
```

διαισθητικά συμβαίνει κάθε φορά που οποιοδήποτε είδος εικόνας (για παράδειγμα ένα στιγμιότυπο της κλάσης Point ή Line) λαμβάνει μια κλήση για μετατροπή μιας συγκεκριμένης απόστασης.

Τα pointcuts μπορούν να συνδυαστούν χρησιμοποιώντας τους τελεστές and, or και not (‘&&’, ‘||’ and ‘!’), για παράδειγμα:

```
receptions(void Point.setX(int)) ||
  receptions(void Point.setY(int))
```

5.2.1 **Στοιχειώδεις προσδιοριστές** pointcut

Η AspectJ περιλαμβάνει μια ποικιλία στοιχειωδών προσδιοριστών Pointcut που αναγνωρίζουν τα Join points με διαφορετικούς τρόπους. Κάποιοι βασικοί προσδιοριστές αναγνωρίζουν μόνο pointcuts ενός είδους· για παράδειγμα το Pointcut `receptions` αντιστοιχίζονται μόνο με Join points αποδοχής καλέσματος μεθόδου. Άλλοι προσδιοριστές ταιριάζουν με οποιοδήποτε είδος Join point στο οποίο ισχύει κάποια συγκεκριμένη ιδιότητα. Για πράδειγμα το `instanceof(Point)`ταιριάζει με όλα τα join points στα οποία το αντικείμενο που εκτελείται την τρέχουσα χρονική στιγμή (δηλαδή η τιμή του `this`)είναι αντικείμενο είτε της κλάσης `Point` είτε υποκλάση της `Point`.

Τα δύο παραπάνω είδη προσδιοριστών μπορούν να συνδυαστούν για να αναγνωρίζουν join points με χρήσιμο τρόπο. Για παράδειγμα ο προσδιοριστής

```
!instanceof(FigureElement) &&
calls(void FigureElement.incrXY(int, int))
```

ταιριάζει όλες τις κλήσεις της μεθόδου `incrXY` που δεν έρχονται από κάποιο αντικείμενο που είναι τύπου figure. Θα είναι δηλαδή κλήσεις που έρχονται από κάποιο αντικείμενο διαφορετικού τύπου, καθώς και από κλήσεις που έρχονται από στατικές μεθόδους.

Οι στοιχειώδεις προσδιοριστές pointcuts παρουσιάζονται συνοπτικά στον Πίνακας 5.2.

Πίνακας 5.2 Στοιχειώδεις προσδιοριστές pointcut

`calls(`*signature*`)`
`receptions(`*signature*`)`
`executions(`*signature*`)`
 	
Αντιστοιχίζει τα call/reception/execution join points με τη μέθοδο ή τον δημιουργό με τον οποίο ταιριάζει η υπογραφή. Η σύνταξη της υπογραφής μιας μεθόδου ειναι:
 	
ResultTypeName RecvrTypeName.meth_id`(`*ParamTypeName*`, …)`
 	
Η σύνταξη της υπογραφής ενός δημιουργού είναι:
 	
NewObjectTypeName`.new(`*ParamTypeName*`, …)`

`gets(`*signature*`)`
`gets(`*signature*`)[`*val*`]`
`sets(`*signature*`)`
`sets(`*signature*`)[`*oldVal*`]`
`sets(`*signature*`)[`*oldVal*`][`*newVal*`]`
 	
Αντιστοιχίζει τα join points τύπου get/set πεδίων με τα πεδία με τα οποία ταιριάζει η υπογραφή. Η σύνταξη της υπογραφής ενός πεδίου είναι:
 	
FieldTypeName ObjectTypeName.field_id

`handles(`*ThrowableTypeName*`)` Ταιριάζει join points τύπου διαχειριστή εξαιρέσεων, που έχει τον συγκεκριμένο τύπο.

`instanceof(`*CurrentlyExecutingObjectTypeName*`)`
`within(`*ClassName*`)`
`withincode(`*signature*`)`

Ταιριάζει τα join points οποιοδήποτε τύπου όπου αυτό που εκτελείται την τρέχουσα χρονική στιγμή: • Είναι ένα αντικείμενο τύπου CurrentlyExecutingObjectTypeName • Ο κώδικας περιλαμβάνεται μέσα στην ClassName • Ο κώδικας περιλαμβάνεται μέσα στο μέλος που ορίζεται από τη μέθοδο ή το δημιουργό με το συγκεκριμένο signature
`cflow(`*`pointcut_designator`*`)` Ταιριάζει με τα join points οποιοδήποτε τύπου που συμβαίνουν αυστηρά μέσα στο δυναμικό βαθμό οποιουδήποτε Join point που έχει ταιριάξει με έναν προσδιοριστή pointcut.
`callto(`*`pointcut_designator`*`)` Ταιριάζει με join points τύπου καλέσματος μεθόδου τα οποία σε ένα βήμα οδηγούν στην αποδοχή ή εκτέλεση join points που έχουν αντιστοιχηθεί από προσδιοριστή pointcut.

`staticinitializations(TypeName)`
`initializations(TypeName)`
 	
Ταιριάζει με αρχικοποίηση κλάσης ή αντικειμένου του συγκεκριμένου τύπου.

5.2.2 **Προσδιοριστές** pointcut **ορισμένοι από το χρήστη**

Για να ορίσει ο χρήστης μόνος του ένα pointcut γίνεται με τη χρήση της δεσμευμένης λέξης pointcut, ως εξής:

```
pointcut moves():
     receptions(void FigureElement.incrXY(int, int)) ||
     receptions(void Line.setP1(Point)) ||
     receptions(void Line.setP2(Point)) ||
     receptions(void Point.setX(int)) ||
     receptions(void Point.setY(int));
```

Η παραπάνω δήλωση ορίζει ένα καινούριο pointcut με το όνομα `moves()`, που αναγνωρίζει το πότε ένα στοιχείο εικόνας (figure) λαμβάνει μια κλήση από μια μέθοδο που μπορεί να το μετακινήσει.

5.3 Advice

Ένα Advice είναι ένας μηχανισμός τύπου μεθόδου και χρησιμοποιείται για να ορίσει τον κώδικα που θα πρέπει να εκτελεστεί σε κάθε join point ενός pointcut. Η AspectJ υποστηρίζει τα advice τύπου *before*, *after* και *around*. Επιπλέον, υπάρχουν δύο ειδικές περιπτώσεις του after advice, *after returning* και *after throwing*, που αντιστοιχούν στους δύο τρόπους που ένας υπο-υπολογισμός μπορεί να επιστρέψει από ένα join point. Το around advice έχει την ειδική ικανότητα να προλαμβάνει επιλεκτικά τον κανονικό υπολογισμό στο join point.

Οι δηλώσεις των advice ορίζουν ένα advice συσχετίζοντας το σώμα ενός κώδικα με ένα pointcut, μια χρονική στιγμή σχετική με κάθε join point στο pointcut, για το πότε θα πρέπει να εκτελεστεί ο κώδικας.

Η παρακάτω δήλωση

```
after(): moves() {
     flag = true;
}
```

ορίζει ένα after advice στο pointcut `moves()`. Το "()" ανάμεσα στο "after" και το ":" σημαίνει ότι το advice δεν έχει παραμέτρους. Η επίδραση αυτής της δήλωσης είναι εγγυηθεί ότι η μεταβλητή `flag` έχει την τιμή `true` οποτεδήποτε ένα στοχείο figure σταματάει να διαχειρίζεται μια κλήση στη μέθοδο move.

5.4 Aspects (**Πτυχές**)

Τα Aspects είναι τμηματικές μονάδες της υλοποίησης της εγκάρσιας διατομής του κώδικα. Τα Aspects ορίζονται με την αντίστοιχη δήλωση, η οποία έχει μία μορφή παρόμοια με τη δήλωση μιας κλάσης. Οι δηλώσεις των Aspects μπορεί να περιλαμβάνουν δηλώσεις pointcuts, δηλώσεις adviceκαθώς και άλλα είδη δηλώσεων που επιτρέπονται στη δήλωση μιας κλάσης.

Η παρακάτω δήλωση ορίζει ένα aspect που υλοποιεί την παρακολούθηση του πότε ένα στοιχείο εικόνας έχει μετακινηθεί προσφάτως. Το aspect αυτό μπορεί να χρησιμοποιηθεί από το μηχανισμό ενημέρωσης της οθόνης ώστε να καταλάβει αν κάτι έχει αλλάξει από την τελευταία φορά που ενημερώθηκε η οθόνη.

```
aspect MoveTracking {
     static boolean flag = false;
     static boolean testAndClear() {
          boolean result = flag;
          flag = false;
          return result;
```

```
    }

    pointcut moves():
       receptions(void FigureElement.incrXY(int, int)) ||
       receptions(void Line.setP1(Point)) ||
       receptions(void Line.setP2(Point)) ||
       receptions(void Point.setX(int)) ||
       receptions(void Point.setY(int));

      after(): moves() {
         flag = true;
      }
}
```

Τα advice ενός aspect είναι παρόμοια με τις μεθόδους, με την έννοια ότι έχουν πρόσβαση σε όλα τα μέλη μιας κλάσης. Επομένως, στη συγκεκριμένη περίπτωση το after advice μπορεί να αναφέρεται στη στατική μεταβλητή `flag`.

5.4.1 **Στιγμιότυπα των** Aspect

Στην AspectJ, η προεπιλεγμένη συμπεριφορά των μη αφηρημένων aspects είναι να έχουν μόνο ένα στιγμιότυπο. Τα Advice τρέχουν στα πλαίσια αυτού του στιγμιοτύπου. Η δήλωση του aspect δέχεται έναν τροποποιητή, που καλείται “of” που παρέχει άλλα είδη συμπεριφοράς του στιγμιοτύπου του aspect.

5.5 **Κληρονομικότητα και υπερκάλυψη των** Advice **και των** Pointcuts

Για την υποστήριξη βιβλιοθηκών με aspects, η AspectJ παρέχει έναν απλό μηχανισμό για υπερκάλυψη pointcuts και κληρονομικότητα των advice. Για να χρησιμοποιήσει αυτό το μηχανισμό ένας προγραμματιστής, ορίζει ένα abstract aspect, με ένα ή περισσότερα abstract pointcuts, και με advice στο/στα pointcut(s). Τότε, αυτό είναι ένα είδος βιβλιοθήκης με aspects που μπορεί να παραμετροποιηθεί με aspects που την επεκτείνουν. Για παράδειγμα, ο παρακάτω κώδικας ορίζει μια απλή βιβιοθήκη με λειτουργικότητα παρακολούθησης.

```
abstract aspect SimpleTracing {
     abstract pointcut tracePoints();
     before(): tracePoints() {
          printMessage(“Entering”, thisJoinPoint);
     }
     after(): tracePoints() {
          printMessage(“Exiting”, thisJoinPoint);
     }
     void printMessage(String when, JoinPoint tjp) {
```

```
            code to print an informative message
            using information from the join point
        }
}
```

Χρησιμοποιώντας τη βιβλιοθήκη σε μια συγκεκριμένη κατάσταση απαιτεί απλώς την επέκταση του aspect και την παροχή ενός ειδικού ορισμού για το abstract pointcut.

```
aspect IncrXYTracing extends SimpleTracing {
     pointcut tracePoints():
         receptions(void FigureElement.incrXY(int, int));
}
```

Κάνοντας το abstract pointcut συμπαγές στο υπο-aspect έχει ως αποτέλεσμα της κληρονόμησης της δήλωσης του advice από το υπερ-aspect στο υπο-aspect. Αν το υπο-aspect περιλαμβάνει έναν "`dominates`" τροποποιητή, ο τροποποιητής αυτός επηρεάζει τη σειρά του κληρονομημένου advice.

5.6 **Θέματα Υλοποίησης**

Ο βασικός στόχος οποιασδήποτε θεματοστρεφούς γλώσσας προγραμματισμού είναι να εγγυηθεί ο κώδικας με πτυχές και ο κώδικας χωρίς πτυχές τρέχουν μαζί με ένα σωστα συντονισμένο τρόπο. Αυτή η διαδικασία συντονισμού ονομάζεται ***aspect weaving*** και έχει στόχο να επιβεβαιώσει ότι το advice που εφαρμόζεται τρέχει στα κατάλληλα join points. Όπως συμβαίνει με τις περισσότερες γλώσσες προγραμματισμού, η διαδικασία του aspect weaving μπορεί να πραγματοποιηθεί είτε από έναν ειδικό προ-επεξεργαστή, κατά τη διάρκεια της μεταγλώττισης ή από έναν επεξεργαστή μετά το τέλος της μεταγλώττισης ή κατά τη διάρκεια φόρτωσης ως μέρος μιας εικονικής μηχανής, χρησιμοποιώντας εντολές κατά το χρόνο εκτέλεσης ή με συνδυασμό των παραπάνω προσεγγίσεων.

Ο σχεδιασμός της γλώσσας AspectJ επιδιώκει να είναι ανεπαίσθητο το πότε πραγματοποιείται η διαδικασία του aspect weaving. Παρέχεται μάλιστα μια υλοποίηση βασισμένη σε μεταγλωττιστή που πραγματοποιεί σχεδόν όλο το weaving κατά τη διάρκεια της μεταγλώττισης. Με τον τρόπο αυτό εμφανίζονται όσο το δυνατόν περισσότερα προγραμματιστικά λάθη κατά τη μεταγλώττιση και αποφεύγεται το overhead στη διάρκεια της εκτέλεσης του προγράμματος.
Ο μεταγλωττιστής χρησιμοποιεί τη στρατηγική υλοποίησης "πληρώνω-καθώς-προχωράω". Όσα μέρη του προγράμματος δεν επηρεάζονται από advice μεταγλωττίζονται όπως θα γινόταν με έναν τυπικό JAVA μεταγλωττιστή.

Ο μεταγλωττιστής μετασχηματίζει τον πηγαίο κώδικα με τρεις τρόπους: το σώμα κάθε δήλωσης advice μεταγλωττίζεται σε μια τυπική μέθοδο, τα μέρη του προγράμματος όπου εφαρμόζεται το advice μετασχηματίζονται έτσι ώστε να εισάγουν στατικά σημεία που αντιστοιχούν στα δυναμικά join points και ο υπόλοιπος κώδικας εισάγεται σε αυτά τα στατικά σημεία.

5.6.1 **Μεταγλώττιση του σώματος των** Advice

Κάθε before ή after advice μεταγλωττίζεται σε μια τυπική μέθοδο και το advice εκτελείται καλώντας τη μέθοδο αυτή από τα αντίστοιχα σημεία στον κώδικα. Αυτό πιθανώς σημαίνει ότι η χρήση ενός advice θα προσθέσει το overhead ενός μοναδικού καλέσματος σε μέθοδο. Ωστόσο, οι μέθοδοι αυτοί είναι πάντα είτε στατικές είτε final, επομένως μπορούν εύκολα να ενσωματωθούν από τις περισσότερες JVMs. Επομένως, σε γενικές γραμμές δε θα πρέπει να παρατηρείται μεγάλο overhead στην απόδοση από αυτά τα παραπάνω καλέσματα σε μεθόδους.

Το σώμα ενός around advice μεταγλωττίζεται στο σώμα μίας μεθόδου για κάθε αντίστοιχο στατικό σημείο στον κώδικα. Αυτό μας επιτρέπει μα περάσουμε αποδοτικά την αναγκαία κατάσταση στη στοίβα καλεσμάτων και να υλοποιήσουμε την δήλωση “proceed” χωρίς την ανάγκη των μηχανισμών της JAVA. Η στρατηγική αυτή ισοσταθμίζει την αύξηση του μεγέθους του παραγόμενου κώδικα με τη σημαντική μείωση στο overhead του χρόνου εκτέλεσης.

5.6.2 **Αντίστοιχη μέθοδος**

Ο μεταγλωττιστής μετασχηματίζει τον πηγαίο κώδικα σε μια μορφή όπου υπάρχει μια σαφώς ορισμένη *αντίστοιχη μέθοδος* για κάθε δυναμικό join point που μπορεί να έχει advice κατά την εκτέλεση. Ο μετασχηματισμός αυτός πραγματοποιείται μόνο για join points που μπορεί να έχουν advice και όχι για όλα τα join points. Έτσι για παράδειγμα, σε ένα πρόγραμμα που έχει advice στο pointcut που προσδιορίζεται από το `gets(int Point._x)`, ο μεταγλωττιστής θα μετασχημάτιζε αναφορές της μορφής `p._x` , όπου p είναι στιγμιότυπο της κλάσης `Point`, σε `Point.$jp$0$(p)`και θα πρόσθετε την παρακάτω μέθοδο στην κλάση `Point`:

```
private static int $jp$0(Point obj) {
     return obj._x;
}
```

Μόλις παραχθει η μέθοδος αυτή, τα `before` και `after advice` υλοποιούνται κάνοντας την αντίστοιχη μέθοδο να καλέσει τις advice μεθόδους, όπως απαιτείται.

Υπάρχουν πολλές περιπτώσεις, συμπεριλαμβανομένης αυτής εδώ, όπου ο μεταγλωττιστής θα προσθέσει επιπλέον καλέσματα σε μεθόδους έτσι ώστε να δημιουργήσει αντίστοιχες μεθόδους. Αυτό συμβαίνει για join points καλέσματος μεθόδου, αποδοχής καλέσματος μεθόδου και πρόσβασης πεδίων. Το overhead των μεθόδων αυτών είναι μικρό σε οποιαδήποτε JVM και εφόσον είναι πάλι όλες static ή final θα βελτιστοποιηθούν από καλές JVMs.

5.6.3 **Δυναμική αποστολή**

Η χρήση συγκεκριμένων προσδιοριστών pointcut, όπως `cflow`, `callsto`, και `instanceof`, μπορεί να απαιτήσει έναν έλεγχο κατά το χρόνο εκτέλεση έτσι ώστε να αποφασιστεί αν μια συγκεκριμένη μέθοδος ταιριάζει στην πραγματικότητα με έναν συγκεκριμένο προσδιοριστή join point. Σε τέτοιες περιπτώσεις, η αντίστοιχη μέθοδος περιλαμβάνει υπολειπόμενο έλεγχο κώδικα που προφυλάσσει την εκτέλεση του advice. Το overhead αυτό είναι πολύ μικρό.

5.7 **Παράδειγμα Θεματοστρεφούς Προγραμματισμού με τη γλώσσα προγραμματισμού** AspectJ

Το παράδειγμα που θα παρουσιάσουμε σε αυτή την παράγραφο εκτελεί τη λειτουργία της καταγραφής (tracing). Θα δείξουμε πώς αυτή επιτυγχάνεται χωρίς και με τη χρήση πτυχών.

Περιγραφή της εφαρμογής
Η εφαρμογή περιλαμβάνει τέσσερις κλάσεις που περιγράφουν γεωμετρικά σχήματα (shapes).

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 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 χρησιμοποιήσει πτυχές:

```
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 πτυχές: Μπορούμε χωρίς να χρησιμοποιήσουμε πτυχές να ενσωματώσουμε δυνατότητες καταγραφής στην εφαρμογή μας, γράφοντας μια κλάση Trace:

```
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 πτυχές

Μπορούμε όμως να ενσωματώσουμε δυνατότητες καταγραφής στην εφαρμογή χρησιμοποιώντας πτυχές. 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 πτυχή αυτή εκτελεί τις κλήσεις για tracing στις κατάλληλες στιγμές που ορίζουμε σύμφωνα με τα pointcuts που καθορίζουν τα αντίστοιχα join points. 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 πτυχές:

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)
```

6.Επίλογος

Η παρούσα εργασία αποτελεί μια μελετή πάνω σε βασικά θέματα της ιδέας του θεματοστρεφούς προγραμματισμού καθώς και στο πώς εφαρμόζεται χρησιμοποιώντας τη γλώσσα προγραμματισμού AspectJ.

Έτσι λοιπόν, αρχικά κάναμε μια ιστορική αναδρομή πάνω σε διαφορετικά παραδείγματα προγραμματισμού, όπως ο συμβολικός, ο διαδικαστικός και ο αντικειμενοστρεφής, ώστε να δούμε πώς η εξέλιξη στις τεχνολογικές ανάγκες έχει οδηγήσει στην εξέλιξη των γλωσσών προγραμματισμού. Ιδιαίτερα σταθήκαμε στην πολύ γνωστή και ευρέως χρησιμοποιούμενη γλώσσα προγραμματισμού, τη JAVA, η οποία ακολουθεί τις αρχές της αντικειμενοστρεφούς σχεδίασης. Είδαμε λοιπόν πώς υλοποιούνται οι θεμελιώδεις έννοιες του αντικειμενοστρεφούς προγραμματισμού, οι κλάσεις και τα αντικείμενα, καθώς και οι υπόλοιπες βασικές έννοιες της κληρονομικότητας, της υπερφόρτωσης, των κατασκευαστών και του πολυμορφισμού με τη JAVA.

Είδαμε ότι ο αντικειμενοστρεφής προγραμματισμός έχει μεγάλη επιτυχία στη διαμόρφωση και υλοποίηση πολύπλοκων συστημάτων λογισμικού, ωστόσο έχει και τα προβλήματά του. Η πρακτική εμπειρία με μεγάλα έργα έχει δείξει ότι οι προγραμματιστές ενδέχεται να αντιμετωπίσουν προβλήματα με τη διατήρηση του κώδικα τους, δεδομένου ότι όσο μεγαλύτερο το λογισμικό που υλοποιείται τόσο και πιο δύσκολος γίνεται ο ξεκάθαρος διαχωρισμός του έργου σε ενότητες (αντικείμενα), πράγμα και το οποίο αποτελεί τη βάση του αντικειμενοστρεφούς προγραμματισμού. Για παράδειγμα, μια μικρή αλλαγή σε μία επαναχρησιμοποιούμενη ενότητα μπορεί τελικά να προκαλέσει πολλές αλλαγές σε άλλες, ανεξάρτητες, ενότητες του κώδικα.

Τέτοιου είδους προβλήματα και πολλές άλλες ανησυχίες έρχεται να επιλύσει μία διαφορετική τεχνική, ο θεματοστρεφής προγραμματισμός, που είναι μια νέα σχετικά μεθοδολογία για το διαχωρισμό ενός προβλήματος σε μεμονωμένες μονάδες που ονομάζονται *πτυχές* (aspects). Μια *πτυχή* είναι μια μονάδα που «κόβει» εγκάρσια τη λογική ροή μίας εφαρμογής, έτσι ώστε να συμπυκνώνονται συμπεριφορές (μέθοδοι) που επηρεάζουν πολλαπλές κλάσεις και ενότητες του κωδικά μας και είναι επαναχρησιμοποιήσιμες αλλά η αλλαγή τους δεν έχει απολύτως καμία επίδραση στον υπόλοιπο κώδικα.

Μία από τις γλώσσες προγραμματισμού που υλοποιεί τις έννοιες της θεματοστρεφούς προσεγγίσεως είναι η AspectJ. Συγκεκριμένα είδαμε ότι η AspectJ στην ουσία επεκτείνει τη γλώσσα προγραμματισμού JAVA, αφού διατηρεί την υλοποίηση των μηχανισμών του αντικειμενοστρεφούς προγραμματισμού, αλλά εισάγει και νέες δυνατότητες, όπως τα aspects, join points, pointcuts, ώστε να επιτυγχάνεται ο στόχος του θεματοστρεφούς

προγραμματισμού, που είναι η καλύτερη κατανόηση του λογισμικού και η εύκολη διαχείριση του κώδικα που γράφεται για μια εφαρμογή υψηλού επιπέδου.

Για να αποκτήσουμε μεγαλύτερη εμβάθυνση στον προγραμματισμό με χρήση της γλώσσας AspectJ και γενικότερα για να αποκτήσουμε πλήρη επίγνωση των δυνατοτήτων του θεματοστρεφούς προγραμματισμού, στο μέλλον (και ίσως στα πλαίσια κάποιου μεταπτυχιακού) μπορούμε να αναπτύξουμε μία εφαρμογή σε γλώσσα προγραμματισμού JAVA την οποία κατόπιν θα επεργαστούμε κατάλληλα εισάγοντας όπου απαιτείται τα στοιχεία της AspectJ και εν συνεχεία να συγκρίνουμε την εφαρμογή με άλλες αντίστοιχες εφαρμογές οι οποίες έχουν αναπτυχθεί με άλλες προγραμματιστικές προσεγγίσεις (όπως ο διαδικαστικός προγραμματισμός) και να δούμε στην πράξη πως βοηθά ο θεματοστρεφής προγραμματισμός στην επιτάχυνση της ανάπτυξης λογισμικού και της βελτίωσης της ποιότητάς του.

Βιβλιογραφία

13. Κλ. Θραμπουλίδης, *Διαδικαστικός Προγραμματισμός – C (Τόμος Α)*, 2η έκδοση, Εκδόσεις Τζιόλα, 2002.

14. *Java 2: A beginner's Guide*, Schildt

15. *Η βίβλος της JAVA 2*, Aaron Walsh, Justin Couch, Daniel Steinberg, ΓΚΙΟΥΡΔΑΣ

16. *Thinking in Java*, Bruce Eckel, Prentish Hall

17. *Aspect-Oriented Programming*, Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Videira Lopes, Jean-Marc Loingtier, John Irwin, in proceedings of the European Conference on Object-Oriented Programming (ECOOP), Finland. Springer-Verlag LNCS 1241. June 1997.

18. *An Overview of AspectJ***,** Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm and William G. Griswold, in Proceedings of the 15th European Conference on Object-Oriented Programming Pages 327-353

19. http://docs.oracle.com/javase/tutorial/

20. *Programming paradigms and an overview of C:* http://cs.anu.edu.au/student/comp3610/lectures/Paradigms.pdf

21. http://en.wikipedia.org/wiki/Object-oriented_programming

22. http://en.wikipedia.org/wiki/Aspect-oriented_programming

23. http://en.wikipedia.org/wiki/Assembly_language

24. http://www.j2ee.gr/2010/06/10/j2ee-%CE%BA%CE%B1%CE%B9-aspect-oriented-programming/