ΑΛΕΞΑΝΔΡΕΙΟ Τ.Ε.Ι ΘΕΣΣΑΛΟΝΙΚΗΣ
ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΚΩΝ ΕΦΑΡΜΟΓΩΝ
ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ανάπτυξη εργαλείου ελέγχου κώδικα για ευέλικτες εφαρμογές

ΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ

Του φοιτητή
Χαλιάσου Στέφανου
Αρ. Μητρώου: 05/2806

Επιβλέπων καθηγητής
Σφέτσος Παναγιώτης

ΘΕΣΣΑΛΟΝΙΚΗ, 2013

Περιεχόμενα

Πρόλογος

Refactoring είναι η διαδικασία αλλαγής ενός λογισμικού συστήματος με τέτοιον τρόπο ώστε να μην αλλάζει η εξωτερική συμπεριφορά του κώδικα, βελτιώνοντας παράλληλα την εσωτερική του δομή. Είναι ένας πειθαρχημένος τρόπος 'συμμαζέματος' του κώδικα που έχει σαν αποτέλεσμα την μείωση των πιθανοτήτων να δημιουργηθούν σφάλματα (bugs). Στην ουσία όταν κάνουμε refactoring βελτιώνουμε την σχεδίαση του κώδικα αφού έχει ήδη γραφτεί.

Η «βελτίωση της σχεδίασης του κώδικα» είναι μία κάπως περίεργη έννοια. Σήμερα, όλοι πιστεύουμε ότι η ανάπτυξη λογισμικού ξεκινά με την σχεδίαση και έπειτα με την συγγραφή του κώδικα. Πρώτα ξεκινάμε με έναν καλό σχεδιασμό του συστήματος και έπειτα αρχίζουμε να γράφουμε τον κώδικα. Με τον καιρό, ο κώδικας θα τροποποιηθεί, έτσι η ακεραιότητα του συστήματος και η δομή του σύμφωνα με την αρχική σχεδίαση σταδιακά θα εξασθενούν.

Το Refactoring είναι το αντίθετο αυτής της πρακτικής. Με αυτό μπορούμε να πάρουμε μία κακή σχεδίαση, ακόμα και έναν χαώδη κώδικα που δεν στηρίχθηκε σε κάποιο αρχικό σχεδιασμό, και να τον 'επαναδιατυπώσουμε' σε έναν καλοσχεδιασμένο κώδικα. Κάθε βήμα αυτής της διαδικασίας είναι απλό. Μεταφέρουμε ένα πεδίο μίας κλάσης σε μία άλλη, παίρνουμε ένα κομμάτι κώδικα μίας μεθόδου και δημιουργούμε μία ξεχωριστή μέθοδο και μεταφέρουμε κώδικα προς τα πάνω ή προς τα κάτω στην ιεραρχία. Η επίδραση όλων αυτών των μικρών αλλαγών μπορούν να βελτιώσουν σημαντικά την σχεδίαση. Είναι το ακριβώς αντίθετο της έννοιας της «αποσύνθεσης του λογισμικού».

Με το Refactoring βρίσκουμε την ισορροπία ανάμεσα στις αλλαγές που επιφέρουν οι νέες προσθήκες στον αρχικό κώδικα. Καταλαβαίνουμε ότι ο σχεδιασμός δεν μπορεί να δημιουργηθεί εξολοκλήρου στην αρχή, αλλά προκύπτει σιγά σιγά κατά την διάρκεια ανάπτυξης του συστήματος. Μαθαίνουμε πως να βελτιώνουμε την σχεδίαση ενώ χτίζουμε και προγραμματίζουμε το σύστημα μας. Η προκύπτουσα αλληλεπίδραση οδηγεί σε ένα πρόγραμμα με μία σχεδίαση που παραμένει καλή καθώς η ανάπτυξη συνεχίζεται.

1 Ένα πρώτο παράδειγμα

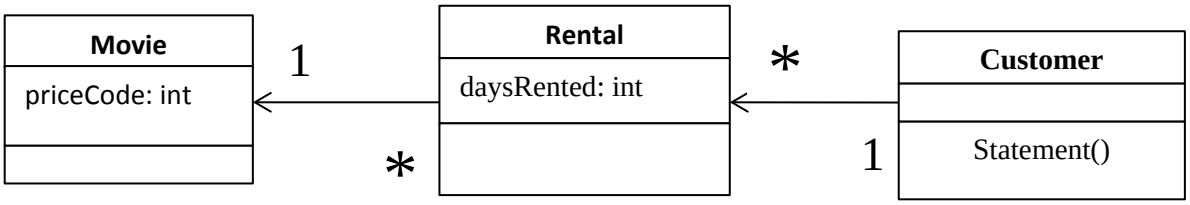
Ο παραδοσιακός τρόπος όταν ξεκινάμε να μιλάμε για κάτι είναι να ψάξουμε την ιστορία του και να αναφέρουμε όλες τις βασικές αρχές που το διέπουν. Αν και ο τρόπος αυτός είναι ιδιαίτερα σωστός και αποδεκτός, συνήθως κατανοούμε καλύτερα τη φύση του προβλήματος μέσω παραδειγμάτων. Θα ξεκινήσουμε λοιπόν με ένα μικρό παράδειγμα.

1.1 Το σημείο εκκίνησης

Το πρόγραμμα που θα αναφέρουμε είναι αρκετά απλό. Είναι ένα πρόγραμμα που υπολογίζει και εκτυπώνει την κατάσταση των χρεώσεων ενός πελάτη (*statement*) σε ένα κατάστημα ενοικίασης ταινιών. Το πρόγραμμα μας ενημερώνει για το ποιες ταινίες έχει νοικιάσει ο πελάτης και για πόσο διάστημα. Έπειτα υπολογίζει τις χρεώσεις, το οποίο εξαρτάται από το πόσο καιρό είναι νοικιασμένη η ταινία και τον τύπο της ταινίας. Υπάρχουν τρία είδη ταινιών: απλή, παιδική και νέα παραλαβή. Επιπλέον, για κάθε χρήστη υπολογίζονται και οι πόντοι συχνότητας ενοικίασης ταινιών (*frequentRenterPoints*) οι οποίοι εξαρτώνται από το αν μία ταινία είναι νέα παραλαβή η όχι.

Στην εικόνα 1.1 βλέπουμε τις βασικές κλάσεις του συστήματος.

Εικόνα 1.1 Διάγραμμα κλάσεων του συστήματος



Ο κώδικας των κλάσεων είναι ο εξής

Η κλάση Movie είναι απλή κλάση δεδομένων.

```
public class Movie {
	public static final int CHILDRENS = 2;
	public static final int REGULAR = 0;
	public static final int NEW_RELEASE = 1;

	private String _title;
	private int _priceCode;

	public Movie(String title, int priceCode) {
		_title = title;
		_priceCode = priceCode;
	}


	public int getPriceCode() {
		return _priceCode;
	}

	public void setPriceCode(int arg) {
		_priceCode = arg;
	}

	public String getTitle (){
		return _title;
	};
}
```

Η κλάση Rental αναπαριστά την ενοικίαση μίας ταινίας από έναν πελάτη.

```
class Rental {
	private Movie _movie;
	private int _daysRented;

	public Rental(Movie movie, int daysRented) {
		_movie = movie;
		_daysRented = daysRented;
	}

	public int getDaysRented() {
		return _daysRented;
	}

	public Movie getMovie() {
		return _movie;
	}
}
```

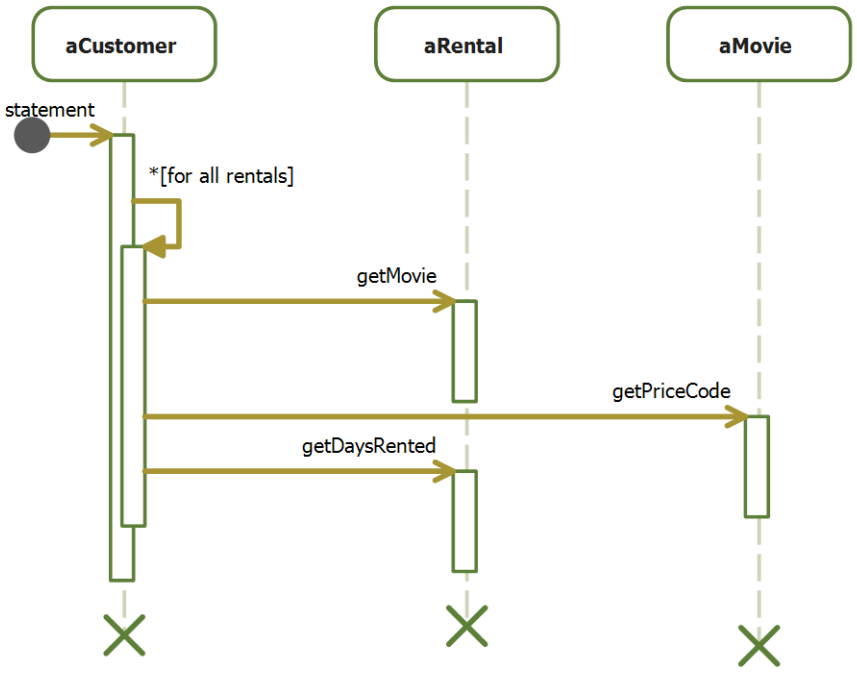
Customer

Η κλάση Customer αναπαριστά τον πελάτη του καταστήματος.

```
class Customer {
      private String _name;
      private Vector _rentals = new Vector();

      public Customer (String name){
            _name = name;
      };

      public void addRental(Rental arg) {
            rentals.addElement(arg);
      }

      public String getName (){
            return _name;
      };
}
```

Η κλάση Customer έχει επίσης μία μέθοδο που παράγει ένα *statement* (κατάσταση των χρεώσεων του πελάτη). Η Εικόνα 1.2 δείχνει τις αλληλεπιδράσεις για κάθε μέθοδο.

Εικόνα 1.2. Οι αλληλεπιδράσεις της μεθόδου statement



```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amounts for each line
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
        }

        // add frequent renter points
        frequentRenterPoints++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1)
                  frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
            String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
             " frequent renter points";

    return result;
}
```

Αν και το παραπάνω πρόγραμμα με μία πρώτη ματιά φαίνεται καλά δομημένο θα δούμε ότι η δομή του μπορεί να βελτιωθεί σημαντικά. Σίγουρα το πρόγραμμα δεν ακολουθεί την αρχή της αντικειμενοστρέφειας. Για ένα απλό πρόγραμμα σαν αυτό, αυτό δεν παίζει σημαντικό ρόλο. Δεν είναι λάθος το να χτίζουμε ένα γρήγορο και αδόμητο απλό πρόγραμμα. Ωστόσο, αν αυτό είναι ένα αντιπροσωπευτικό κομμάτι ενός πιο σύνθετου συστήματος τότε η δομή του σίγουρα θα δημιουργήσει

προβλήματα. Η μεγάλη μέθοδος *statement* της κλάσης *Customer* περιέχει πολύ λειτουργικότητα και λογική. Με απλά λόγια, κάνει πάρα πολλά. Ένα μεγάλο μέρος της λειτουργικότητάς της θα μπορούσε να μοιραστεί σε άλλες κλάσεις ή μεθόδους.

Ακόμα και έτσι όμως το πρόγραμμα δουλεύει. Ο *compiler* δεν ενδιαφέρεται για την δομή του προγράμματος και το κατά πόσο αυτό είναι καλά δομημένο ή όχι. Αλλά οι προγραμματιστές που κάποια στιγμή θα θελήσουν να προσθέσουν νέα κομμάτια στο σύστημα ενδιαφέρονται. Ένα κακά δομημένο σύστημα είναι πολύ δύσκολο να αλλάξει. Αυτό γιατί είναι δύσκολο να κατανοήσουμε τις αλλαγές που χρειάζονται να γίνουν. Αν λοιπόν είναι δύσκολο να κατανοήσουμε τις αλλαγές που πρέπει να γίνουν, τότε ο προγραμματιστής που θα τις κάνει είναι πολύ πιθανόν να οδηγηθεί σε λάθη και να παράξει *bugs*.

Στην περίπτωσή μας έχουμε μία αλλαγή που οι χρήστες θα ήθελαν να γίνει. Αρχικά, θα ήθελαν ο λογαριασμός του πελάτη να εκτυπώνεται σε HTML μορφή ώστε να είναι συμβατό με Web εφαρμογές. Σκεφτείτε την επίδραση που θα είχε αυτή η αλλαγή. Κοιτάζοντας τον κώδικα βλέπουμε ότι είναι αδύνατο να καταλάβουμε την συμπεριφορά της μεθόδου *statement* για να παράξουμε ένα *statement* σε HTML μορφή. Η μόνη επιλογή είναι να δημιουργήσουμε μία καινούρια μέθοδο η οποία κατά ένα μεγάλο βαθμό θα είναι ίδια με την προηγούμενη. Αυτό, αρχικά μπορεί να μην ακούγεται και τόσο δύσκολο. Απλά αντιγράφουμε την μέθοδο *statement*, δημιουργούμε μία καινούρια *htmlStatement* και τροποποιούμε τα σημεία που θέλουμε.

Τι γίνεται όμως όταν αλλάξουν οι χρεώσεις; Τότε θα πρέπει να διορθώσουμε τόσο την *statement* όσο και την *htmlStatement* και να βεβαιωθούμε ότι οι διορθώσεις είναι σωστές. Το πρόβλημα της αντιγραφής κώδικα δημιουργείται όταν κάποια στιγμή ο κώδικας πρέπει να αλλάξει. Αν γράφουμε ένα πρόγραμμα που είναι δύσκολο να αλλάξει τότε δεν υπάρχει πρόβλημα. Αν όμως το πρόγραμμά μας είναι ένα συνεχώς αναπτυσσόμενο πρόγραμμα με πολλές πιθανότητες αλλαγών τότε η αντιγραφή κώδικα θα είναι πάντα μία ‘απειλή’.

Η αλλαγή των χρεώσεων είναι μόνο μία από τις αλλαγές που ενδέχεται να γίνουν. Οι χρήστες του συστήματος θέλουν να κάνουν αλλαγές στον τρόπο που κατηγοριοποιούν τις ταινίες, αλλά δεν έχουν αποφασίσει ακόμα ποιες θα είναι αυτές οι αλλαγές. Οι αλλαγές αυτές θα επηρεάσουν και τον τρόπο με τον οποίο οι πελάτες θα χρεώνονται για τις ταινίες αλλά και τον τρόπο με τον οποίο θα υπολογίζονται οι πόντοι ενοικίασης τους (*frequentRenterPoints*). Σαν προγραμματιστές μπορούμε να καταλάβουμε πως όποιο και να είναι το αρχικό σχήμα που θα επιλέξουν οι χρήστες του συστήματος είναι πολύ πιθανόν να αλλάξει στην συνέχεια.

Η μέθοδος *statement* είναι εκείνη που πρέπει να αλλάξει για να αντιμετωπίσουμε τις αλλαγές στην κατηγοριοποίηση των ταινιών και των χρεώσεων. Αν, παρόλα αυτά αντιγράψουμε την μέθοδο *statement* σε μία νέα *htmlStatement* θα πρέπει να βεβαιωθούμε ότι οι αλλαγές είναι πλήρως συνεπείς. Επιπλέον, καθώς οι κανόνες χρέωσης μεγαλώνουν σε πολυπλοκότητα θα είναι ακόμα πιο δύσκολο να βρούμε που χρειάζονται να γίνουν αλλαγές και να τις πραγματοποιήσουμε χωρίς λάθη.

Ίσως κάποιος εδώ να μπει στον πειρασμό να κάνει τις λιγότερες δυνατές αλλαγές στο πρόγραμμα, έτσι κι αλλιώς θα δουλεύει μια χαρά. Το πρόγραμμα μπορεί να μην χαλάσει, αλλά θα μας κάνει την ζωή δύσκολη γιατί θα είναι δύσκολο να κάνουμε τις αλλαγές που θα θέλουν οι χρήστες. Εδώ ακριβώς έρχεται το refactoring.

1.2 Το πρώτο βήμα για το Refactoring

Το πρώτο βήμα για το Refactoring είναι να χτίσουμε ένα σταθερό σετ από τεστ για το τμήμα του κώδικα που μας ενδιαφέρει. Τα τεστ είναι σημαντικά, γιατί ακόμα και αν ακολουθήσουμε καλές πρακτικές refactoring για την αποφυγή δημιουργίας σφαλμάτων, εξακολουθούμε να είμαστε άνθρωποι και ενδέχεται να κάνουμε λάθη. Για αυτό ακριβώς χρειαζόμαστε σταθερά και συμπαγή τεστ.

Επειδή η μέθοδος *statement* επιστρέφει *String*, δημιουργούμε μερικούς πελάτες (*Customers*), δίνουμε σε κάθε πελάτη μερικές ενοικιάσεις (*Rentals*) με διάφορα είδη ταινιών και δημιουργούμε τα *statements* τα οποία είναι τύπου *String*. Στην συνέχεια κάνουμε μία σύγκριση (*string comparison*) μεταξύ των αποτελεσμάτων της μεθόδου *statement* και κάποιων αναφορών σε *String* τα οποία έχουμε φτιάξει μόνοι μας και γνωρίζουμε ότι είναι σωστά. Είναι το αποτέλεσμα που περιμένουμε να πάρουμε από την μέθοδο *statement* για κάθε έναν από τους πελάτες που έχουμε δημιουργήσει. Η διαδικασία αυτή μπορεί πολύ εύκολα να αυτοματοποιηθεί με Java (Ant & Junit) και χρειάζεται μόλις μερικά δευτερόλεπτα για να εκτελεστούν όλα τα τεστ.

Το σημαντικό κομμάτι αυτών των τεστ είναι ο τρόπος με τον οποίο παρουσιάζονται τα αποτελέσματα. Είτε αναφέρουν το «OK» που σημαίνει ότι όλα τα *strings* που επέστρεψε η μέθοδος *statement* είναι όμοια με τις αναφορές τους, είτε τυπώνουν μία λίστα από λάθη. Εξαρτάται από εμάς το πως θέλουμε να τρέξουν τα τεστ μας και το τι θα εκτυπώνουν στο τέλος. Το σημαντικό είναι ότι αφού δημιουργήσουμε αυτά τα τεστ μπορούμε πολύ εύκολα και γρήγορα να ελέγχουμε κατά πόσο το refactoring που εκτελούμε έχει επηρεάσει την εξωτερική συμπεριφορά του προγράμματός μας.

1.3 Αποσύνθεση και Αναδιαμόρφωση της μεθόδου Statement

Ο πρώτος προφανής στόχος μας είναι η μεγάλη μέθοδος *statement*. Κοιτώντας την μέθοδο, αντιλαμβανόμαστε ότι πρέπει να την αποσυνθέσουμε σε μικρότερα κομμάτια. Τα μικρότερα κομμάτια κώδικα τείνουν να είναι περισσότερο διαχειρίσιμα και κατανοητά. Είναι πολύ πιο εύκολο για τους προγραμματιστές να δουλεύουν με αυτά.

Η πρώτη φάση του refactoring είναι να ‘σπάσουμε’ την μέθοδο και να μετακινήσουμε κάποια κομμάτια της σε νέες κλάσεις. Ο στόχος μας είναι να γίνει όσο το δυνατόν ευκολότερο να γράψουμε μία μέθοδο HTML statement με τη λιγότερη δυνατή αντιγραφή κώδικα από την αρχική μέθοδο.

Το πρώτο βήμα λοιπόν, είναι να βρούμε λογικά κομμάτια κώδικα και να χρησιμοποιήσουμε την μέθοδο Εξαγωγής (βλέπε *Extract Method (Εξαγωγή* Μεθόδου)). Ένα προφανές κομμάτι είναι ο *switch* βρόχος. Φαίνεται να είναι ένα αρκετά μεγάλο κομμάτι το οποίο θα μπορούσε να αποτελέσει από μόνο του μία μέθοδο.

Όταν κάνουμε εξαγωγή μίας μεθόδου πρέπει να γνωρίζουμε τι θα μπορούσε να πάει στραβά αν η εξαγωγή γίνει λανθασμένα. Θα μπορούσε να δημιουργήσει κάποιο bug στο πρόγραμμά μας. Έτσι, πριν κάνουμε το refactoring πρέπει να εξετάσουμε πως θα το κάνουμε με ασφάλεια.

Αρχικά, πρέπει να κοιτάξουμε αν στο κομμάτι μας γίνεται αναφορά σε κάποια μεταβλητή που έχει οριστεί τοπικά μέσα στη μέθοδο. Όπως βλέπουμε χρησιμοποιούνται δύο μεταβλητές: `each` και `thisAmount`. Από αυτές η `each` δεν τροποποιείται από τον κώδικα όμως η `thisAmount` τροποποιείται. Κάθε μη τροποποιήσιμη μεταβλητή μπορούμε να την περάσουμε ως παράμετρο. Οι μεταβλητές όμως που η τιμή τους αλλάζει χρειάζονται περισσότερη προσοχή. Αν υπάρχει μόνο μία μπορούμε να την επιστρέφουμε από την μέθοδο. Η `thisAmount` αρχικοποιείται στο 0 κάθε φορά που ξεκινάει ένας νέος γύρος μέσα στον βρόχο επανάληψης και δεν αλλάζει παρά μόνο μέσα στο *switch*. Άρα επιστρέφουμε απλά την τιμή της από την νέα μέθοδο.

Παρακάτω βλέπουμε την μέθοδο *statement* πριν και μετά το refactoring. Με έντονο χρώμα φαίνονται τα κομμάτια που μεταφέραμε από την αρχική μέθοδο.

Πριν το Refactoring

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();

        //determine amounts for each line
        switch (each.getMovie().getPriceCode()) {
            case Movie.REGULAR:
                thisAmount += 2;
                if (each.getDaysRented() > 2)
                    thisAmount += (each.getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                thisAmount += each.getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                thisAmount += 1.5;
                if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
        }

        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1)
                  frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
            String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";

    return result;
}
```

Μετά το Refactoring

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        double thisAmount = 0;
        Rental each = (Rental) rentals.nextElement();
        thisAmount = amountFor(each);
        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1)
                  frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
            String.valueOf(thisAmount) + "\n";
        totalAmount += thisAmount;
    }

    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
             " frequent renter points";

    return result;
}
```

// Η νέα μέθοδος amountFor

```
private int amountFor(Rental each) {
    int thisAmount = 0;
    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
        }
    return thisAmount;
}
```

Κάθε φορά που κάνουμε μία τέτοια αλλαγή πρέπει να κάνουμε compile των κώδικα και να τρέξουμε τα τεστ. Εάν λοιπόν είχαμε ένα σετ από τεστ και τα τρέχαμε είναι πολύ πιθανόν τα περισσότερα από αυτά να αποτύγχαναν. Με μία γρήγορη ματιά ίσως να μην εντοπίσουμε το λάθος. Αν όμως κοιτάξουμε λίγο καλύτερα την νέα μέθοδο *amountFor* θα δούμε ότι ο τύπος επιστροφής της είναι *int* αντί για *double*.

```
private double amountFor(Rental each) {
    double thisAmount = 0;

    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
        }

    return thisAmount;
}
```

Αυτό ήταν ένα πολύ συνηθισμένο λάθος όταν εκτελούμε refactoring. Σε αυτή την περίπτωση η Java μετατρέπει τους δεκαδικούς σε ακεραίους χωρίς να δηλώνει κάποιο σφάλμα κατά την μεταγλώττιση του κώδικα. Βλέπουμε λοιπόν πόσο σημαντικά είναι τα τεστ κατά την διάρκεια του refactoring. Παρόλα αυτά, αν και είναι πολύ εύκολο να κάνουμε κάποιο λάθος, είναι επίσης πολύ εύκολο να βρούμε γρήγορα το λάθος μας. Αυτό γιατί οι αλλαγές που εκτελούμε είναι αρκετά μικρές και έτσι δεν ξοδεύουμε πολύ ώρα για να κάνουμε αποσφαλμάτωση (debugging).

Επειδή δουλεύουμε με Java έπρεπε αρχικά να αναλύσουμε τον κώδικά μας για να δούμε τι θα κάνουμε με τις τοπικές μεταβλητές. Με ένα αυτόματο εργαλείο refactoring ωστόσο, αυτό θα μπορούσε να γίνει πάρα πολύ εύκολα. Σήμερα υπάρχουν πολλά τέτοια εργαλεία όπως είναι το refactoring tool του Eclipse. Απλά, επιλέγουμε τον κώδικα που θέλουμε, επιλέγουμε “Extract Method” από το μενού επιλογών και πληκτρολογούμε το όνομα της νέας μεθόδου. Επιπλέον, τα αυτόματα εργαλεία δεν πρόκειται να κάνουν ανόητα λάθη όπως το προηγούμενο.

Τώρα λοιπόν που σπάσαμε την αρχική μέθοδο σε δύο μεθόδους, μπορούμε να δουλέψουμε με την κάθε μία ξεχωριστά. Η επόμενη αλλαγή είναι να αλλάξουμε τα ονόματα κάποιων μεταβλητών. Θα μετονομάσουμε την μεταβλητή *thisAmount* σε *result* και την *each* σε *aRental*.

Ο αρχικός κώδικας

```
private double amountFor(Rental each) {
    double thisAmount = 0;

    switch (each.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            thisAmount += 2;
            if (each.getDaysRented() > 2)
                thisAmount += (each.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            thisAmount += each.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            thisAmount += 1.5;
            if (each.getDaysRented() > 3)
                thisAmount += (each.getDaysRented() - 3) * 1.5;
            break;
    }

    return thisAmount;
}
```

Και ο μετονομασμένος

```
private double amountFor(Rental aRental) {
    double result = 0;

    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2)
                result += (aRental.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.getDaysRented() > 3)
                result += (aRental.getDaysRented() - 3) * 1.5;
            break;
    }

    return result;
}
```

Μόλις τελειώσουμε με την μετονομασία των μεταβλητών ξανακάνουμε compile και τεστάρουμε ξανά για να βεβαιωθούμε πως δεν χάλασε τίποτα.

Άξιζε λοιπόν η μετονομασία των μεταβλητών; Σίγουρα. Ο καλός κώδικας θα πρέπει να μπορεί να δείχνει ξεκάθαρα τι ακριβώς κάνει, και οι μεταβλητές είναι το κλειδί για τον καθαρό κώδικα. Δεν χρειάζεται να φοβόμαστε να αλλάζουμε τα ονόματα των μεταβλητών αν αυτό βοηθά στην σαφήνεια. Και με τα εργαλεία refactoring που αναφέραμε πριν, αυτή η διαδικασία γίνεται πάρα πολύ εύκολα.

1.4 Μετακίνηση του υπολογισμού του ποσού

Καθώς κοιτάμε την μέθοδο `amountFor`, βλέπουμε ότι περιέχει πληροφορίες για την ενοικίαση (*rental*) αλλά δεν περιέχει πληροφορία για τον πελάτη (*Customer*).

```
private double amountFor(Rental aRental) {
    double result = 0;

    switch (aRental.getMovie().getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (aRental.getDaysRented() > 2)
                result += (aRental.getDaysRented() - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += aRental.getDaysRented() * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (aRental.getDaysRented() > 3)
                result += (aRental.getDaysRented() - 3) * 1.5;
            break;
    }

    return result;
}
```

Αυτό αμέσως μας βάζει σε υποψία ότι η μέθοδος βρίσκεται σε λάθος κλάση. Στις περισσότερες περιπτώσεις η μέθοδος θα πρέπει να βρίσκεται στην κλάση της οποίας τα δεδομένα χρησιμοποιεί. Για αυτό η μέθοδος πρέπει να μετακινηθεί στην κλάση *Rental*. Για να το κάνουμε αυτό χρησιμοποιούμε την τεχνική Μετακίνησης Μεθόδου (βλέπε *Move Method (Μετακίνηση Μεθόδου)*). Αντιγράφουμε τον κώδικα, δηλαδή την μέθοδο *statement* στην κλάση *Rental*, κάνουμε μερικές τροποποιήσεις για να ‘ταιριάζει’ στην νέα κλάση και μετά κάνουμε ξανά compile.

```
class Rental {
    ...
    double getCharge() {
        double result = 0;
        switch (getMovie().getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (getDaysRented() > 2)
                    result += (getDaysRented() - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += getDaysRented() * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (getDaysRented() > 3)
                    result += (getDaysRented() - 3) * 1.5;
                break;
    }

    return result;
}
```

Στην περίπτωσή μας, το να 'ταιριάζει' στην νέα κλάση σημαίνει ότι πρέπει να αφαιρέσουμε την παράμετρο `Rental aRental` και να μετονομάσουμε το όνομα της μεθόδου, από `amountFor` σε `getCharge`.

Χρειαζόμαστε όμως μία ακόμα αλλαγή ώστε να ολοκληρωθεί αυτή η μετατροπή. Η κλάση *Customer* εξακολουθεί να υπολογίζει το ποσό με τον δικό της τρόπο. Πρέπει η μέθοδος `amountFor` να αλλάξει ώστε να υπολογίζει το ποσό μέσω της νέας μεθόδου.

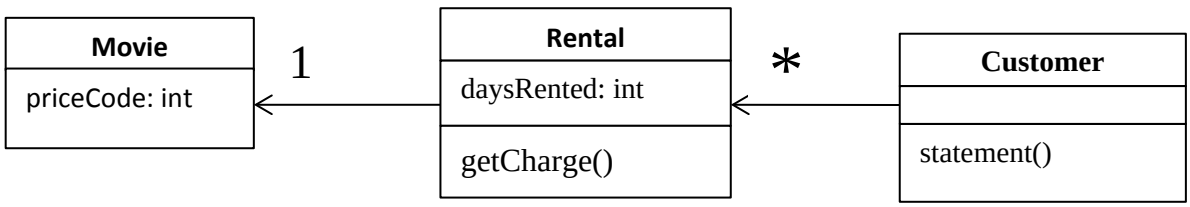
```
class Customer {
    ...
    private double amountFor(Rental aRental) {
        return aRental.getCharge();
    }
}
```

Μόλις κάνουμε compile, τρέξουμε τα τεστ και βεβαιωθούμε ότι δεν υπήρξε κάποιο λάθος, περνάμε στο επόμενο βήμα. Είναι φανερό πλέον ότι η `amountFor` δεν κάνει τίποτα άλλο από το να καλεί την `getCharge`. Άρα μας είναι άχρηστη. Θα αλλάξουμε λοιπόν κάθε αναφορά στην μέθοδο `amountFor` να καλεί την νέα μέθοδο. Στην συγκεκριμένη περίπτωση αυτό δεν είναι κάτι δύσκολο. Η `amountFor` δημιουργήθηκε πριν λίγο και γνωρίζουμε ότι μόνο η μέθοδος `statement` έχει αναφορά σε αυτή. Αν όμως η αλλαγή αυτή δεν ήταν στα πλαίσια ενός γενικότερου refactoring όπως αυτό που εκτελούμε τώρα, θα πρέπει να είμαστε αρκετά προσεκτικοί γιατί η μέθοδος μπορεί να καλείται από διάφορα σημεία του κώδικα.

```
class Customer {
    ...
    public String statement() {
        double totalAmount = 0;
        int frequentRenterPoints = 0;
        Enumeration rentals = _rentals.elements();
        String result = "Rental Record for " + getName() + "\n";

        while (rentals.hasMoreElements()) {
            double thisAmount = 0;
            Rental each = (Rental) rentals.nextElement();

            thisAmount = each.getCharge();

            // add frequent renter points
            frequentRenterPoints ++;
            // add bonus for a two day new release rental
            if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
                each.getDaysRented() > 1)
                frequentRenterPoints ++;

            //show figures for this rental
            result += "\t" + each.getMovie().getTitle()+ "\t" +
                      String.valueOf(thisAmount) + "\n";
            totalAmount += thisAmount;
        }

        //add footer lines
        result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
        result += "You earned " + String.valueOf(frequentRenterPoints) +
                  " frequent renter points";

        return result;
    }
}
```

Ένας εύκολος τρόπος για να βρούμε όλες τις αναφορές στην μέθοδο που θέλουμε να διαγράψουμε είναι να σβήσουμε την μέθοδο και να κάνουμε compile. Τότε ο compiler θα μας ενημερώσει για όλες τις κλήσεις στην μέθοδο ως σφάλματα, αφού πλέον δεν υπάρχει.

Εικόνα 1.3. Διάγραμμα κλάσεων μετά την μετακίνηση της μεθόδου charge



Κάποιες φορές ωστόσο, είναι προτιμότερο να μην σβήνουμε την παλιά μέθοδο. Αυτό είναι ιδιαίτερα χρήσιμο στην περίπτωση που η μέθοδος είναι public γιατί έτσι δεν θα αλλάξει το interface της κλάσης.

Η επόμενη αλλαγή που χρειάζεται να κάνουμε αφορά την μεταβλητή `thisAmount` η οποία είναι πλέον περιττή. Το μόνο που κάνει είναι να παίρνει το αποτέλεσμα που επιστρέφει η μέθοδος `getCharge` και χωρίς να αλλάζει η τιμή της στη συνέχεια. Οπότε θα την αφαιρέσουμε με την μέθοδο της Αντικατάστασης Προσωρινής Μεταβλητής με Μέθοδο (βλέπε *Replace Temp with Query (Αντικατάσταση Προσωρινής Μεταβλητής με Μέθοδο)*) και στην θέση της θα καλούμε την `getCharge`.

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
            each.getDaysRented() > 1)
            frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
                  String.valueOf(each.getCharge()) + "\n";
        totalAmount += each.getCharge();
    }

    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";

    return result;
}
```

Είναι αρκετά χρήσιμο να απαλλαγούμε από τις τοπικές προσωρινές μεταβλητές όσο το δυνατόν περισσότερο. Οι μεταβλητές αυτές είναι πρόβλημα γιατί δημιουργούν πολλές παραμέτρους χωρίς να χρειάζονται πραγματικά. Μπορούμε πολύ εύκολα να χάσουμε τον λόγο ύπαρξή τους. Είναι ιδιαίτερα ‘ύπουλες’ σε μεγάλες μεθόδους. Βέβαια, από την άλλη, υπάρχει το τίμημα της απόδοσης. Εδώ η μέθοδος `getCharge` υπολογίζεται δύο φορές. Αυτό όμως μπορεί πολύ εύκολα να βελτιστοποιηθεί στην κλάση `Rental`.

1.5 Εξαγωγή των πόντων ενοικίασης

Το επόμενο βήμα είναι να κάνουμε κάτι παρόμοιο με τους πόντους ενοικίασης (`frequentRenterPoints`). Οι κανόνες ποικίλουν ανάλογα με το είδος της ταινίας. Φαίνεται λογικό να αναθέσουμε και πάλι τον υπολογισμό στην κλάση `Rental`. Πρώτα, πρέπει να χρησιμοποιήσουμε την μέθοδο Εξαγωγής ξανά για το κομμάτι υπολογισμού των πόντων ενοικίασης.

```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        // add frequent renter points
        frequentRenterPoints ++;
        // add bonus for a two day new release rental
        if ((each.getMovie().getPriceCode() == Movie.NEW_RELEASE) &&
             each.getDaysRented() > 1)
            frequentRenterPoints ++;

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
                  String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
    }

    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";

    return result;
}
```

Κοιτάμε ξανά για χρήση τοπικά ορισμένων μεταβλητών. Και πάλι χρησιμοποιείται η μεταβλητή `each` την οποία μπορούμε να την περάσουμε ως παράμετρο. Η άλλη τοπική μεταβλητή είναι η `frequentRenterPoints`. Σε αυτή την περίπτωση όμως η μεταβλητή έχει ήδη μία τιμή. Το σώμα της εξαγόμενης μεθόδου ωστόσο δεν διαβάζει την τιμή, οπότε δεν χρειάζεται να την περάσουμε ως παράμετρο.

Η νέα μέθοδος στην κλάση `Rental`

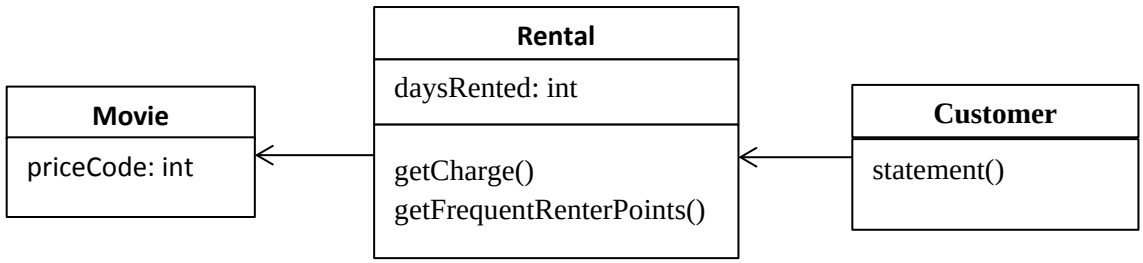
```
class Rental {
    ...
    double getFrequentRenterPoints() {
        int points = 1;

        if((getMovie().getPriceCode() == Movie.NEW_RELEASE &&
            each.getDaysRented() > 1)
            points++;
    }

    return points;
}
```

Η μέθοδος `statement` μετά την αλλαγή

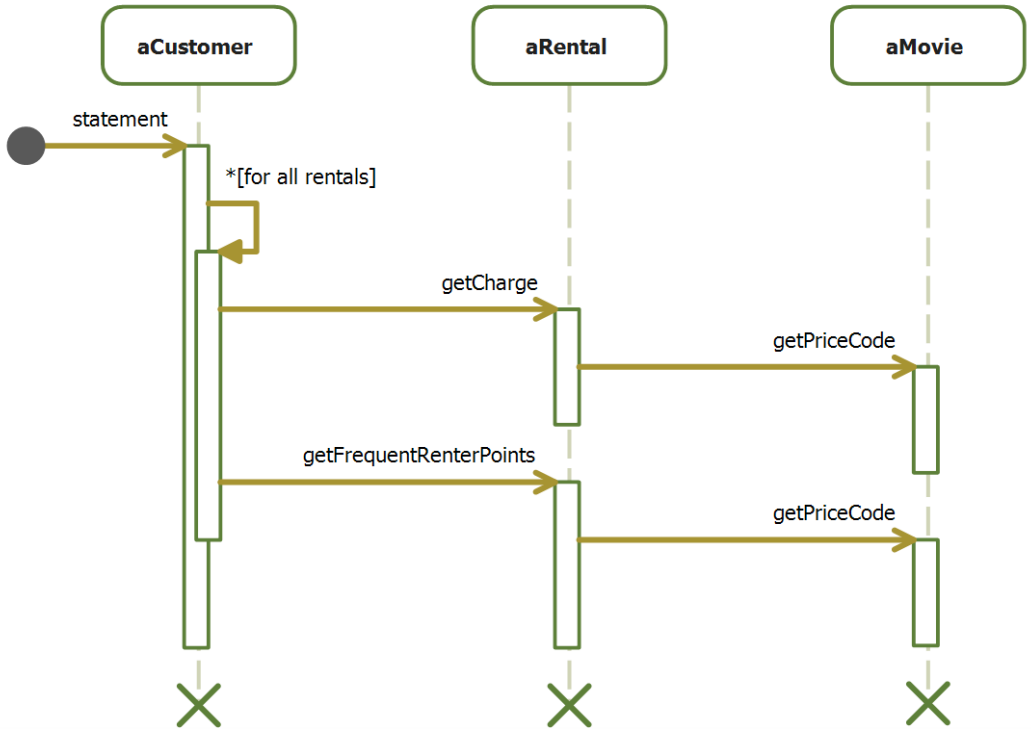
```
public String statement() {
    double totalAmount = 0;
    int frequentRenterPoints = 0;
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        frequentRenterPoints += each.getFrequentRenterPoints();

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
                  String.valueOf(each.getCharge()) + "\n";
            totalAmount += each.getCharge();
    }

    //add footer lines
    result += "Amount owed is " + String.valueOf(totalAmount) + "\n";
    result += "You earned " + String.valueOf(frequentRenterPoints) +
              " frequent renter points";

    return result;
}
```

Ας δούμε μετά και την τελευταία αλλαγή πως έχουν διαμορφωθεί τα διαγράμματα κλάσεων και ροής.

Εικόνα 1.4. Διάγραμμα κλάσεων μετά την εξαγωγή και μετακίνηση του υπολογισμού των πόντων ενοικίασης



Εικόνα 1.5. Διάγραμμα ροής μετά την εξαγωγή και την μετακίνηση του υπολογισμού των πόντων ενοικίασης



1.6 Απαλλαγή από τις προσωρινές μεταβλητές

Όπως είπαμε και πριν, οι προσωρινές τοπικές μεταβλητές μπορεί να είναι πρόβλημα. Είναι χρήσιμες μόνο μέσα στην ρουτίνα που έχουν οριστεί, και για αυτό τείνουν να δημιουργούν μεγαλύτερες και πολυπλοκότερες μεθόδους. Σε αυτή την περίπτωση, έχουμε δύο προσωρινές μεταβλητές, οι οποίες και οι δύο χρησιμοποιούνται για να πάρουμε ένα σύνολο από τις ενοικιάσεις του χρήστη. Και οι δύο εκδόσεις της μεθόδου, η απλή και η HTML, χρησιμοποιούν αυτές τις μεταβλητές. Όπως αναφέραμε προηγουμένως, με ένα αυτόματο εργαλείο refactoring μπορούμε πολύ γρήγορα να αντικαταστήσουμε τις μεταβλητές `totalAmount` και `frequentRentalPoints` δημιουργώντας καινούριες μεθόδους, οι οποίες θα είναι διαθέσιμες σε κάθε άλλη μέθοδο της κλάσης. Αντικαθιστούμε λοιπόν τις μεταβλητές αυτές με τις μεθόδους `getTotalCharge()` και `getTotalFrequentRenterPoints()`.

```
public String statement() {
    Enumeration rentals = _rentals.elements();
    String result = "Rental Record for " + getName() + "\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        //show figures for this rental
        result += "\t" + each.getMovie().getTitle()+ "\t" +
                  String.valueOf(each.getCharge()) + "\n";
    }

    //add footer lines
    result += "Amount owed is " + String.valueOf(getTotalCharge()) + "\n";
    result += "You earned " + String.valueOf(getTotalFrequentRenterPoints())
              + " frequent renter points";

    return result;
}
```

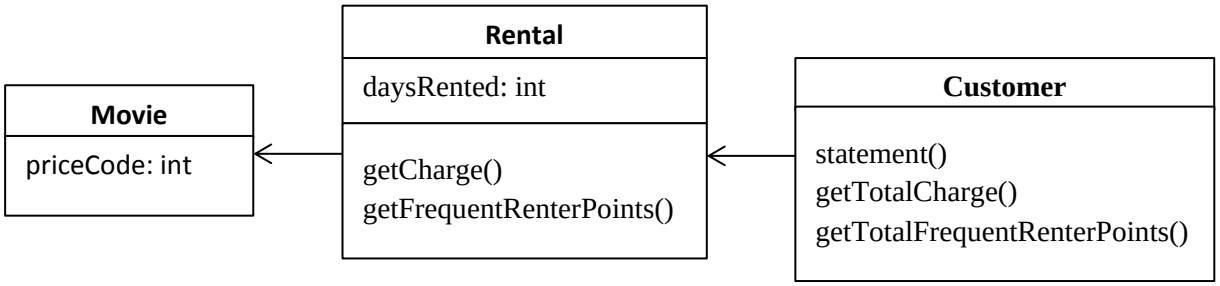
Η νέα μέθοδος `getTotalCharge`

```
private double getTotalCharge() {
    double result = 0;
    Enumeration rentals = _rentals.elements();

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += each.getCharge();
    }

    return result;
}
```

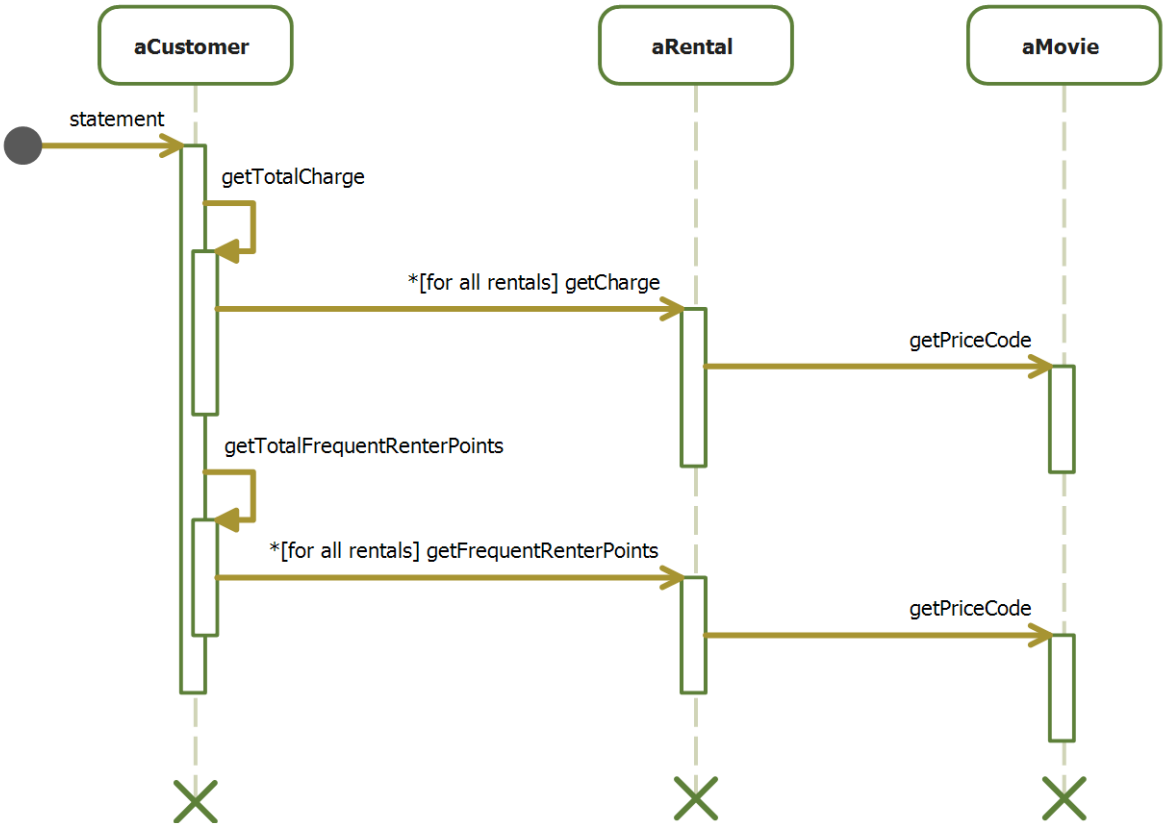
Η νέα μέθοδος `getTotalFrequentRenterPoints`

```
private int getTotalFrequentRenterPoints(){
    int result = 0;
    Enumeration rentals = _rentals.elements();

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();
        result += each.getFrequentRenterPoints();
    }

    return result;
}
```

Εικόνα 1.6. Διάγραμμα κλάσεων την αλλαγή των μεταβλητών



Εικόνα 1.7. Διάγραμμα ροής μετά την αλλαγή των μεταβλητών



Έχουμε αναφέρει ότι το refactoring μειώνει των κώδικα μας. Με την τελευταία κίνηση όμως έγινε το αντίθετο. Επίσης φαίνεται ότι και απόδοση μειώθηκε. Πριν την αλλαγή ο κώδικας χρειαζόταν μόνο ένα βρόχο while, τώρα χρειάζεται τρεις. Κάθε βρόχος χρειάζεται επιπλέον χρόνο για να εκτελεστεί και αυτό μπορεί να βλάψει την απόδοση. Πολλοί προγραμματιστές δεν θα έκαναν αυτή την αλλαγή για αυτό τον λόγο. Ωστόσο δεν μπορούμε να είμαστε απόλυτα σίγουροι ότι η απόδοση έχει μειωθεί, διότι δεν μπορούμε να ξέρουμε πόσο χρόνο χρειάζεται κάθε ένας από τους βρόχους για να εκτελεστεί και πόσο συχνά καλείται ώστε να επηρεάζει δραστικά την απόδοση.

Οι νέες μέθοδοι είναι τώρα διαθέσιμες στην κλάση Customer. Μπορούν πολύ εύκολα να προστεθούν στο interface της κλάσης ώστε και άλλες κλάσεις του συστήματος να μπορούν να τις χρησιμοποιούν. Χωρίς αυτές, άλλες μέθοδοι πρέπει να γνωρίζουν τις ενοικιάσεις (rentals) και να δημιουργήσουν εκ νέου τους βρόχους επανάληψης που απαιτούνται. Σε ένα πολύπλοκο σύστημα αυτό θα οδηγήσει σε πολύ περισσότερο κώδικα που θα είναι πιο δύσκολο να συντηρηθεί.

Ήρθε λοιπόν η ώρα να δούμε τις θετικές επιπτώσεις που είχαν οι παραπάνω αλλαγές δημιουργώντας την μέθοδο `htmlStatement`.

Η νέα μέθοδος `htmlStatement`

```
public String htmlStatement() {
    Enumeration rentals = _rentals.elements();
    String result = "<H1>Rentals for <EM>" + getName() +
                    "</EM></H1><P>\n";

    while (rentals.hasMoreElements()) {
        Rental each = (Rental) rentals.nextElement();

        //show figures for each rental
        result += each.getMovie().getTitle()+ ": " +
                  String.valueOf(each.getCharge()) + "<BR>\n";
    }

    //add footer lines
    result += "<P>You owe <EM>" + String.valueOf(getTotalCharge()) +
              "</EM><P>\n";
    result += "On this rental you earned <EM>" +
              String.valueOf(getTotalFrequentRenterPoints()) +
              "</EM> frequent renter points<P>";

    return result;
}
```

Με την εξαγωγή όλων των υπολογισμών σε διαφορετικές μεθόδους, μπορούμε να δημιουργήσουμε την μέθοδο `htmlStatement` και να επαναχρησιμοποιήσουμε των κώδικα των υπολογισμών που αρχικά βρισκόταν στην μέθοδο `statement`. Δεν κάναμε καμία αντιγραφή από την αρχική μέθοδο, οπότε αν κάποιος από τους υπολογισμούς αλλάξει, η αλλαγή θα πρέπει να γίνει σε ένα μόνο σημείο.

1.7 Αλλαγή του κώδικα υπολογισμού με συνθήκες σε πολυμορφισμό

Το πρώτο μέρος αυτού του προβλήματος είναι ο βρόχος switch. Είναι κακή ιδέα να κάνουμε switch πάνω στο χαρακτηριστικό ενός άλλου αντικειμένου. Αν πρέπει να χρησιμοποιήσουμε ένα switch βρόχο, θα πρέπει να είναι στα δικά μας δεδομένα, όχι στα δεδομένα κάποιου άλλου αντικειμένου.

Αυτό συνεπάγεται ότι η μέθοδος `getCharge` θα πρέπει να μεταφερθεί στην κλάση `Movie`:

```
class Movie {
    ...
    double getCharge(int daysRented) {
    double result = 0;

    switch (getPriceCode()) {
        case Movie.REGULAR:
            result += 2;
            if (daysRented > 2)
                result += (daysRented - 2) * 1.5;
            break;
        case Movie.NEW_RELEASE:
            result += daysRented * 3;
            break;
        case Movie.CHILDRENS:
            result += 1.5;
            if (daysRented > 3)
                result += (daysRented - 3) * 1.5;
            break;
    }

    return result;
}
```

Για να δουλέψει αυτό, πρέπει να περάσουμε ως παράμετρο τις μέρες ενοικίασης, οι οποίες ασφαλώς είναι δεδομένα της κλάσης `Rental`. Η μέθοδος χρησιμοποιεί αποτελεσματικά δύο κομμάτια δεδομένων, τις μέρες ενοικίασης και τον τύπο της ταινίας. Γιατί όμως είναι προτιμότερο να περάσουμε ως παράμετρο τις μέρες ενοικίασης από την `Rental` στην `Movie` αντί τον τύπο της ταινίας από την `Movie` στην `Rental`; Επειδή οι προτεινόμενες αλλαγές έχουν να κάνουν με την προσθήκη νέων τύπων ταινίας. Αν αλλάξουμε τον τύπο της ταινίας, θέλουμε να έχει τις λιγότερες δυνατές επιπτώσεις, έτσι είναι προτιμότερο να υπολογίζουμε την χρέωση μέσα στην `Movie`.

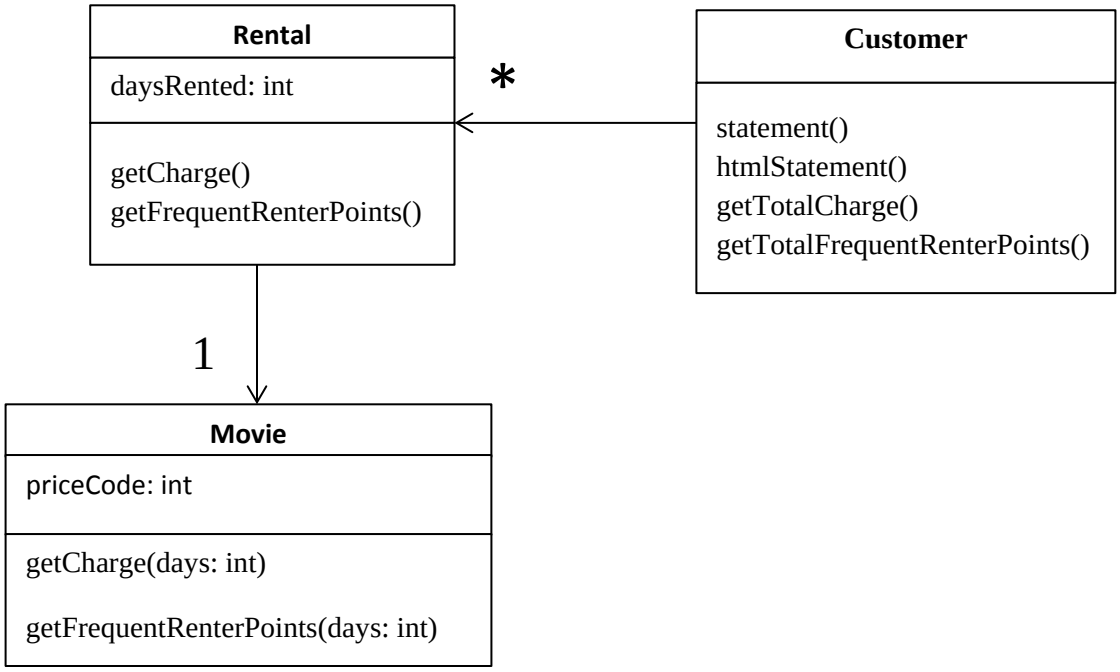
Αφού μετακινήσαμε την μέθοδο `getCharge` κάνουμε το ίδιο και με τον υπολογισμό των πόντων συχνότητας ενοικίασης. Έτσι μεταφέρουμε και τους δύο υπολογισμούς που ποικίλουν ανάλογα με τον τύπο της ταινίας, μέσα στην κλάση που γνωρίζει τον τύπο.

```
class Movie {
    ...
    int getFrequentRenterPoints(int daysRented) {
    if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
        return 2;
    else
        return 1;
}
```

Η κλάση `Rental` μετά τις αλλαγές έχει ως εξής:

```
class Rental {
    ...
    double getCharge() {
        return _movie.getCharge(_daysRented);
    }

    int getFrequentRenterPoints() {
        return _movie.getFrequentRenterPoints(_daysRented);
    }
}
```

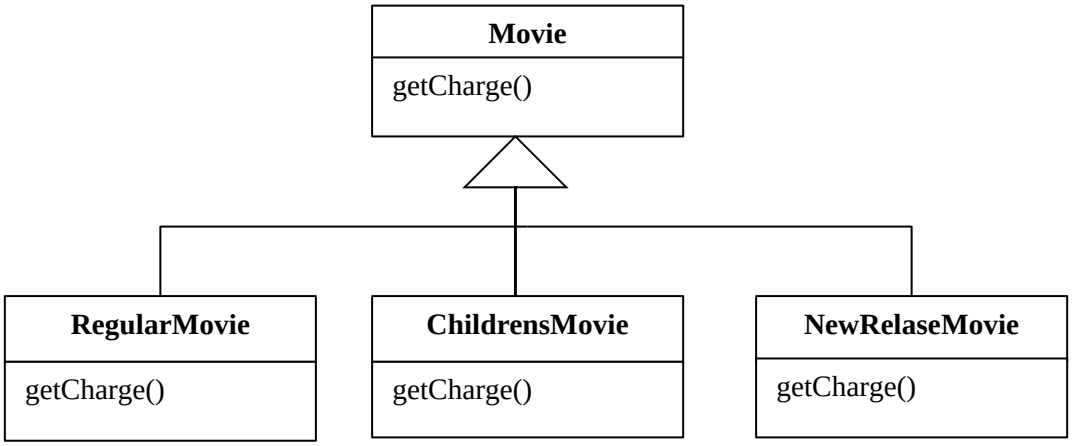
Εικόνα 1.8. Διάγραμμα κλάσεων μετά την μετακίνηση των μεθόδων στην κλάση Movie



1.8 Υπολογισμός των χρεώσεων μέσω Κληρονομικότητας

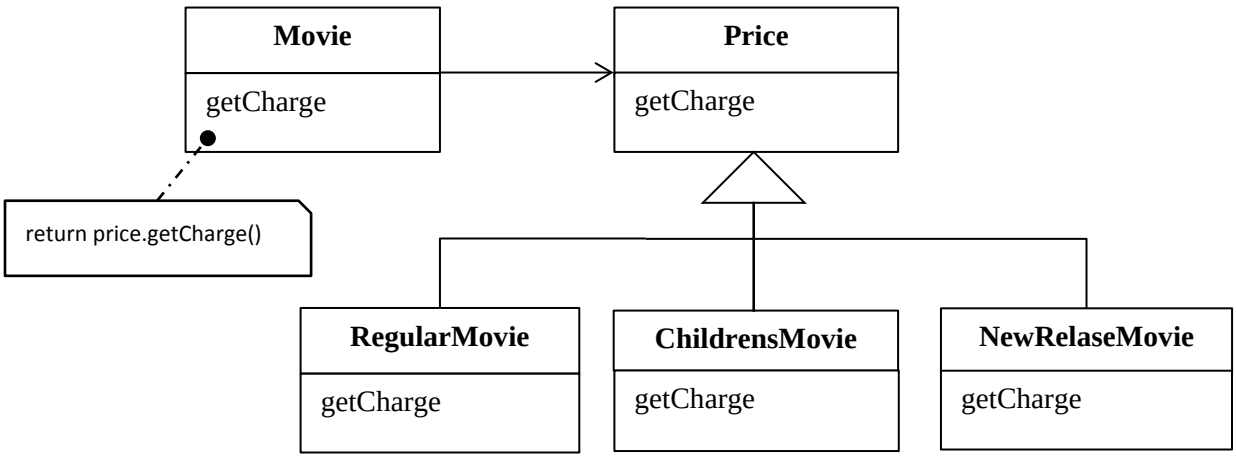
Έχουμε αρκετούς τύπους ταινιών ο καθένας από τους οποίους έχει ένα διαφορετικό τρόπο να 'απαντάει' στην ίδια ερώτηση. Αυτό ακούγεται σαν δουλεία για υποκλάσεις. Μπορούμε να έχουμε τρεις υποκλάσεις της κλάσης Movie, κάθε μία από τις οποίες θα έχει την δική της έκδοση χρεώσεων.

Εικόνα 1.9. Χρήση κληρονομικότητας στην κλάση Movie



Αυτό μας επιτρέπει να αντικαταστήσουμε τον βρόχο switch χρησιμοποιώντας πολυμορφισμό. Δυστυχώς όμως, ακόμα και έτσι δεν θα δουλέψει. Μία ταινία μπορεί να αλλάξει κατηγορία κατά την διάρκεια της ζωής της. Ένα αντικείμενο όμως, δεν μπορεί να αλλάξει κλάση κατά την διάρκεια της δικής του ζωής. Η λύση είναι να χρησιμοποιήσουμε το Πρότυπο Κατάστασης (Design Patterns, 1994). Με το πρότυπο κατάστασης οι κλάσεις θα είναι όπως στην εικόνα 1.10.

Εικόνα 1.10. Χρήση του Προτύπου Κατάστασης στην κλάση Movie



Προσθέτοντας αυτή την έμμεση κλήση στην μέθοδο getPrice μπορούμε να χρησιμοποιούμε τις υποκλάσεις του αντικειμένου priceCode, το οποίο πριν ήταν τύπου Int, όπου το χρειαζόμαστε.

Κάποιος θα μπορούσε πολύ εύκολα να αναρωτηθεί αν η νέα κλάση Price αντιπροσωπεύει έναν αλγόριθμο για να υπολογίζουμε το κόστος ή αν αντιπροσωπεύει τον τύπο (κατάσταση) της ταινίας. Σε

αυτό το σημείο η επιλογή του προτύπου αντανακλά το πως σκεφτόμαστε την δομή. Προς το παρών μας δείχνει την κατάσταση της ταινίας.

Για να εισάγουμε το Πρότυπο Κατάστασης στον κώδικά μας θα χρησιμοποιήσουμε τρεις μεθόδους refactoring. Αρχικά θα μετακινήσουμε τον Κώδικα Τύπου (πεδίο που δηλώνει τον τύπο της ταινίας) μέσα στο Πρότυπο Κατάστασης με την χρήση της μεθόδου Αντικατάσταση Τύπου Κωδικού με Κατάσταση (*SourceMaking, 2013,Replace Type Code with State/Strategy*). Έπειτα θα χρησιμοποιήσουμε την μέθοδο Μετακίνησης Μεθόδου, για να μετακινήσουμε την τον βρόχο switch στην κλάση Price. Τέλος θα κάνουμε χρήση της μεθόδου Αντικατάσταση Συνθήκης Πολυμορφισμό (*SourceMaking, 2013, Replace Conditional with Polymorphism*), για να αφαιρέσουμε τον βρόχο switch.

Στο πρώτο βήμα πρέπει να χρησιμοποιήσουμε την τεχνική της Αυτό-ενθυλάκωσης Πεδίου (*SourceMaking, 2013, Self-Encapsulate Field*) στον Τύπο Κωδικού (`priceCode`) για να βεβαιωθούμε ότι όλες οι κλήσεις αυτού του πεδίου περνάνε μέσα από τις μεθόδους ανάθεσης/απόκτησης (set/get). Επειδή ο περισσότερος κώδικας προέρχεται από άλλες κλάσεις, οι περισσότερες μέθοδοι κάνουν ήδη χρήση αυτών των μεθόδων. Ωστόσο, οι δομητές έχουν απευθείας πρόσβαση στο πεδίο priceCode της κλάσης Movie.

```
class Movie {
    ...
    public Movie(String name, int priceCode) {
        _name = name;
        _priceCode = priceCode;
    }
}
```

Αντί λοιπόν για την απευθείας ανάθεση τιμής στο πεδίο priceCode χρησιμοποιούμε την μέθοδο set.

```
class Movie {
    ...
    public Movie(String name, int priceCode) {
        _name = name;
        setPriceCode(priceCode);
    }
}
```

Τώρα θα προσθέσουμε τις νέες κλάσεις. Βάζουμε την λογική του Τύπου Κωδικού μέσα στην κλάση Price. Αυτό θα γίνει με μία αφηρημένη μέθοδο στη κλάση Price και με κανονικές μεθόδους στις υποκλάσεις.

```
abstract class Price {
    abstract int getPriceCode();
}

class ChildrensPrice extends Price {
    int getPriceCode() {
        return Movie.CHILDRENS;
    }
}

class NewReleasePrice extends Price {
    int getPriceCode() {
        return Movie.NEW_RELEASE;
    }
}

class RegularPrice extends Price {
    int getPriceCode() {
        return Movie.REGULAR;
    }
}
```

Τώρα πρέπει να δημιουργήσουμε τις μεθόδους get/set στην νέα κλάση Price για το πεδίο priceCode για να χρησιμοποιεί η κλάση Movie.

```
public int getPriceCode() {
    return _priceCode;
}
public setPriceCode (int arg) {
    _priceCode = arg;
}
private int _priceCode;
```

Αυτό σημαίνει ότι θα πρέπει να αντικαταστήσουμε το πεδίο priceCode στην Movie με ένα νέο πεδίο price που θα είναι αντικείμενο της νέας κλάσης.

```
class Movie {
    ...
    public int getPriceCode() {
        return _price.getPriceCode();
    }

    public void setPriceCode(int arg) {
        switch (arg) {
            case REGULAR:
                _price = new RegularPrice();
                break;
            case CHILDRENS:
                _price = new ChildrensPrice();
                break;
            case NEW_RELEASE:
                _price = new NewReleasePrice();
                break;
            default:
                throw new IllegalArgumentException("Incorrect Price Code");
        }
    }

    private Price _price;
}
```

Τώρα θα χρησιμοποιήσουμε την μέθοδο ‘Μετακίνησης Μεθόδου’ (βλέπε *Move Method (Μετακίνηση Μεθόδου)*), στην μέθοδο getCharge για να την μεταφέρουμε στην κλάση Price. Η μέθοδος πριν την μετακίνηση είχε ως εξής:

```
class Movie {
    ...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }

        return result;
    }
}
```

Μετά την μετακίνηση:

```
class Movie {
    ...
    double getCharge(int daysRented) {
        return _price.getCharge(daysRented);
}

class Price {
    ...
    double getCharge(int daysRented) {
        double result = 0;
        switch (getPriceCode()) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDRENS:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }

        return result;
    }
}
```

Μετά την μετακίνηση της getCharge θα κάνουμε την Αντικατάσταση Συνθήκης με Πολυμορφισμό (SourceMaking, 2013, Replace Conditional with Polymorphism). Αυτό θα γίνει παίρνοντας κάθε φορά μία από τις συνθήκες που βρίσκονται στο switch και δημιουργούμε με αυτή μία καινούρια μέθοδο στις υποκλάσεις της Price.

```
class RegularPrice {
    ...
    double getCharge(int daysRented){
        double result = 2;

        if (daysRented > 2)
            result += (daysRented - 2) * 1.5;

        return result;
    }
}
```

Η παραπάνω μέθοδος ‘υπερβαίνει’ (overrides) την αρχική συνθήκη (case), την οποία προς το παρών αφήνουμε ως έχει. Κάνουμε το ίδιο και για τις άλλες δύο συνθήκες του βρόχου.

```
class ChildrensPrice {
    ...
    double getCharge(int daysRented){
        double result = 1.5;

        if (daysRented > 3)
            result += (daysRented - 3) * 1.5;

        return result;
    }
}

class NewReleasePrice {
    ...
    double getCharge(int daysRented){
        return daysRented * 3;
    }
}
```

Αφού τελειώσουμε με όλα τις συνθήκες του βρόχου switch, δηλώνουμε την `Price.getCharge` ως abstract.

```
class Price {
    ...
    abstract double getCharge(int daysRented);
}
```

Συνεχίζουμε με τα ίδια βήματα να κάνουμε το ίδιο για την μέθοδο `getFrequentRenterPoints`. Πρώτα μετακινούμε την μέθοδο στην κλάση Price.

```
Class Movie {
   ...
   int getFrequentRenterPoints(int daysRented) {
      return _price.getFrequentRenterPoints(daysRented);
   }

Class Price {
   ...
   int getFrequentRenterPoints(int daysRented) {
      if ((getPriceCode() == Movie.NEW_RELEASE) && daysRented > 1)
         return 2;
      else
         return 1;
   }
}
```

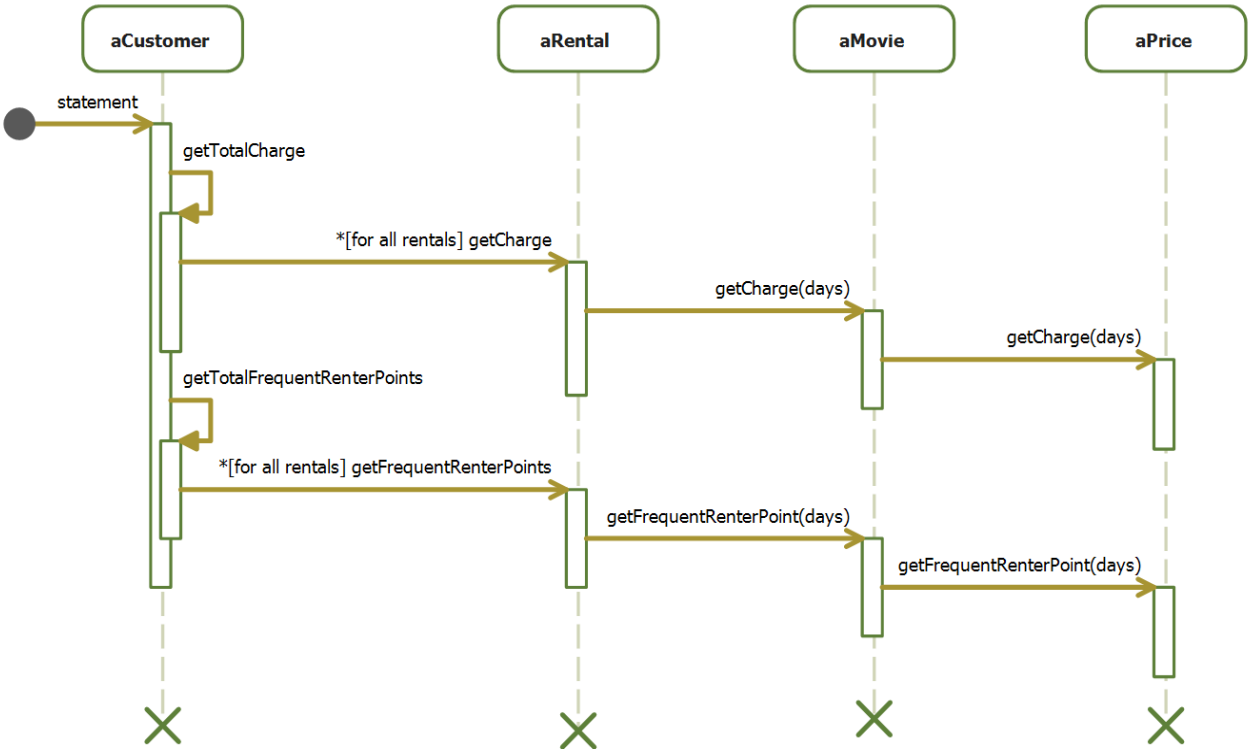
Σε αυτή την περίπτωση ωστόσο, δεν δηλώνουμε την μέθοδο στην υπερκλάση abstract. Αντίθετα δημιουργούμε μία μέθοδο που την κάνει override για τις νέες παραλαβές (new releases) και αφήνουμε ως προεπιλογή (default) την μέθοδο στην υπερκλάση.

```
Class NewReleasePrice {
    ...
    int getFrequentRenterPoints(int daysRented) {
        if (daysRented > 1)
            return 2;
        else
            return 1;
    }
}

Class Price {
    ...
    int getFrequentRenterPoints(int daysRented){
        return 1;
    }
}
```

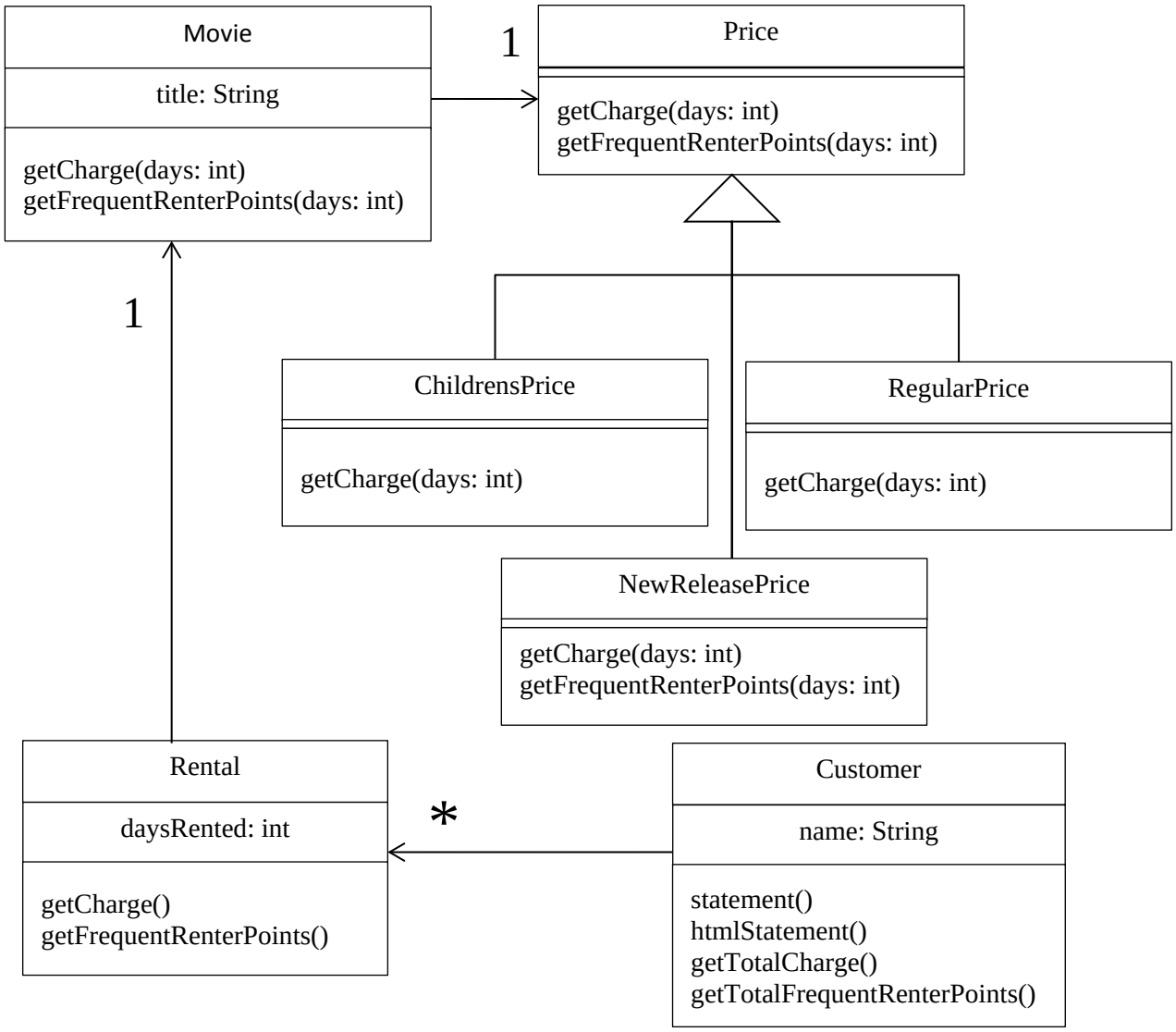
Το να εισάγουμε στον κώδικα το Πρότυπο Κατάστασης χρειάστηκαν αρκετές αλλαγές και πολύ προσπάθεια. Το κέρδος από όλη αυτή την διαδικασία είναι ότι πλέον οποιαδήποτε αλλαγή στον υπολογισμό των πόντων ή των χρεώσεων και η εισαγωγή νέων τύπων ταινιών θα είναι πολύ πιο εύκολη να γίνει. Η υπόλοιπη εφαρμογή δεν γνωρίζει τίποτα για το Πρότυπο Κατάστασης. Για το μικρό σύστημά μας αυτό ίσως να μην έχει μεγάλη επίδραση. Σε ένα πιο πολύπλοκο και μεγάλο σύστημα όμως με δεκάδες μεθόδους που εξαρτώνται από τον τύπο ενός αντικειμένου αυτό θα έκανε μεγάλη διαφορά.

Τώρα που τελείωσε λοιπόν το και το δεύτερο refactoring θα είναι αρκετά πιο εύκολες οι αλλαγές στην κατηγοριοποίηση των ταινιών και στην αλλαγή των κανόνων υπολογισμού των χρεώσεων και των πόντων ενοικίασης. Οι εικόνες 1.11 και 1.12 δείχνουν τις αλλαγές μετά την εισαγωγή του Προτύπου Κατάστασης.

Εικόνα 1.11. Διάγραμμα ροής με την χρήση του Προτύπου Κατάστασης



Εικόνα 1.12. Διάγραμμα κλάσεων με την χρήση του Προτύπου Κατάστασης



2 Αρχές σχετικά με το Refactoring

2.1 Ορισμός του Refactoring

Υπάρχουν δύο ορισμοί σχετικά με το refactoring:

Refactoring (ουσιαστικό): Μια αλλαγή στην εσωτερική δομή του λογισμικού για να γίνει πιο εύκολη η κατανόησή του και λιγότερο δαπανηρή κάθε τροποποίηση χωρίς να αλλαχθεί η εξωτερική του συμπεριφορά.

Refactor (ρήμα): Όταν προσπαθούμε αν αναδιαρθρώσουμε το λογισμικό με μία σειρά από τεχνικές refactoring χωρίς να αλλάζουμε την εξωτερική του συμπεριφορά.

Κάποιοι θεωρούν ότι το refactoring είναι απλά μια προσπάθεια για να ‘καθαρίσουμε’ τον κώδικά μας. Κατά μία έννοια αυτό είναι σωστό. Όμως το refactoring πηγαίνει ένα βήμα παραπέρα επειδή παρέχει τεχνικές για να ‘καθαριστεί’ ο κώδικας με έναν πιο αποτελεσματικό και ελεγχόμενο τρόπο. Όσο περισσότερο χρησιμοποιούμε το refactoring τόσο πιο αποτελεσματικό και καθαρό κώδικα γράφουμε. Καθώς γινόμαστε όλο και πιο οικείοι με τις διάφορες τεχνικές, μαθαίνουμε καλύτερα ποιες πρέπει να χρησιμοποιούμε και πότε. Αυτό έχει σαν αποτέλεσμα την μείωση των πιθανοτήτων να εμφανιστεί κάποιο bug.

Ο σκοπός του refactoring είναι να κάνει το λογισμικό ευκολότερο στην κατανόηση και την τροποποίηση του. Κατά την ανάπτυξη ενός λογισμικού μπορούν να γίνουν αρκετές αλλαγές που μπορούν να επηρεάσουν λίγο ή καθόλου την εξωτερική του συμπεριφορά. Από αυτές τις αλλαγές μόνο όσες γίνονται με σκοπό την ευκολότερη κατανόηση και τροποποίηση του κώδικα είναι refactoring. Ένα καλό παράδειγμα είναι οι αλλαγές για βελτιστοποίηση της απόδοσης του συστήματος (performance optimization). Όπως και το refactoring, η βελτιστοποίηση του συστήματος συνήθως δεν αλλάζει την εξωτερική του συμπεριφορά, αλλάζει μόνο την εσωτερική δομή. Ωστόσο, ο σκοπός είναι διαφορετικός. Οι αλλαγές για βελτιστοποίηση συνήθως κάνουν τον κώδικα δύσκολο στην κατανόησή του.

Το refactoring δεν αλλάζει την εξωτερική συμπεριφορά του συστήματος. Οποιοσδήποτε άλλος χρήστης, είτε είναι ο τελικός χρήστης είτε ένας άλλος προγραμματιστής, δεν μπορεί να γνωρίζει ότι έχουν γίνει αλλαγές.

Σύμφωνα με τον Kent Beck, όταν χρησιμοποιούμε το refactoring για να αναπτύξουμε ένα λογισμικό, εκτελούμε δύο διαφορετικές ενέργειες: εισαγωγή λειτουργίας και refactoring. Όταν γίνεται εισαγωγή νέας λειτουργικότητας δεν πρέπει να αλλάξει ο υπάρχων κώδικας. Νέα τεστ πρέπει να δημιουργηθούν για τις νέες λειτουργίες του συστήματος. Κατά το refactoring, δεν προστίθεται νέα λειτουργικότητα, απλά αλλάζει την δομή του υπάρχοντος κώδικα. Σε αυτή την περίπτωση δεν χρειάζεται να προστεθούν νέα τεστ. Μόνο εάν αλλάξει το interface του κώδικα στον οποίο γίνονται οι αλλαγές θα πρέπει να αλλάξουν και τα τεστ.

Κατά την ανάπτυξη ενός συστήματος είναι πολύ χρήσιμο να γίνεται εναλλαγή των δυο ενεργειών αυτών. Αυτό γιατί, καθώς ξεκινάμε να γράφουμε κώδικα αντιλαμβανόμαστε ότι θα ήταν πιο εύκολο αν

ο κώδικας ήταν δομημένος διαφορετικά. Έτσι σταματάμε για λίγο να προσθέτουμε νέες λειτουργίες και κάνουμε refactoring. Μόλις ο κώδικας έχει καλύτερη δομή συνεχίζουμε να προσθέτουμε νέα χαρακτηριστικά.

2.2 Το refactoring βελτιώνει την σχεδίαση του λογισμικού.

Καθώς οι προγραμματιστές αλλάζουν τον κώδικα, χάνει την δομή του. Αυτό συμβαίνει γιατί οι μικρές αλλαγές με σκοπό να εξυπηρετήσουν έναν βραχυπρόθεσμο στόχο γίνονται εκτός αρχικού σχεδιασμού (design). Γίνεται δυσκολότερη έτσι η κατανόηση του αρχικού σχεδιασμού καθώς κάποιος διαβάζει τον κώδικα. Το refactoring κάνει αυτό ακριβώς, βοηθά στο 'νοικοκύρεμα' του κώδικα. Βοηθά στην αφαίρεση μικρών κομματιών που δεν βρίσκονται ακριβώς στο σωστό σημείο. Η απώλεια της δομής του κώδικα έχει αθροιστική επίδραση. Όσο δυσκολότερο είναι να κατανοήσει κάποιος την δομή του, τόσο δυσκολότερη γίνεται η διατήρησή του και τόσο πιο γρήγορα φθίνει. Το τακτικό refactoring βοηθά ώστε να διατηρηθεί η δομή του κώδικα.

Σε ένα κακά σχεδιασμένο πρόγραμμα συνήθως χρειάζεται περισσότερος χρόνος για να προσθέσουμε νέες λειτουργίες, συχνά γιατί ο κώδικας κάνει παρόμοια πράγματα σε διαφορετικά σημεία. Έτσι ένας σημαντικός τρόπος για την βελτίωση της σχεδίασης είναι η εξάλειψη των διπλοεγγραφών κώδικα. Η σημασία του φαίνεται σε μελλοντικές τροποποιήσεις. Η μείωση του κώδικα δεν θα κάνει το σύστημα να τρέχει γρηγορότερα, γιατί οι αλλαγές αυτές έχουν ελάχιστη επίδραση. Ωστόσο, θα έχει μεγάλη διαφορά στην τροποποίησή του. Όσο περισσότερος είναι ο κώδικας τόσο δυσκολότερη γίνεται η σωστή τροποποίησή του. Υπάρχει περισσότερος κώδικας που πρέπει να κατανοηθεί πριν από οποιαδήποτε αλλαγή. Αλλάζοντας ένα μικρό κομμάτι, το σύστημα δεν κάνει αυτό που θα περιμέναμε γιατί δεν έχει τροποποιηθεί και ένα παρόμοιο κομμάτι που κάνει σχεδόν τα ίδια σε κάποιο άλλο σημείο. Εξαλείφοντας αυτές τις διπλοεγγραφές, εξασφαλίζουμε ότι ο κώδικας θα εκτελεί μία συγκεκριμένη ενέργεια σε ένα μόνο σημείο.

2.3 Το refactoring κάνει το λογισμικό ευκολότερο στην κατανόηση

Ο προγραμματισμός είναι κατά κάποιον τρόπο μια επικοινωνία με τον υπολογιστή. Γράφουμε κώδικα ο οποίος λέει στον υπολογιστή τι πρέπει να κάνει, και αντιδρά κάνοντας ακριβώς αυτό που του λέμε. Ο προγραμματισμός με αυτό τον τρόπο έχει να κάνει με το τι θέλουμε να κάνει ο υπολογιστής. Όμως, υπάρχουν και άλλοι προγραμματιστές στο ίδιο σύστημα. Κάποια στιγμή κάποιος θα προσπαθήσει να διαβάσει τον κώδικα για να κάνει κάποιες αλλαγές. Συνήθως οι προγραμματιστές ξεχνάνε αυτό το κομμάτι και προσπαθούν να γράψουν ένα πρόγραμμα που θα είναι πιο γρήγορο για τον υπολογιστή. Όμως αυτό που έχει σημασία είναι το πόσο γρήγορα θα καταφέρει ένας άλλος προγραμματιστής να κάνει μία αλλαγή στο πρόγραμμα παρά το πόση ώρα θα πάρει στον υπολογιστή να το μεταγλωττίσει.

Το refactoring μας βοηθά να κάνουμε τον κώδικα πιο ευανάγνωστο. Όταν κάνουμε refactoring έχουμε κώδικα που λειτουργεί αλλά δεν είναι ιδανικά δομημένος. Λίγος αφιερωμένος χρόνος στο refactoring μπορεί να κάνει τον κώδικα πολύ πιο κατανοητό. Ο προγραμματισμός με αυτό τον τρόπο έχει να κάνει με το 'λέμε ακριβώς το αυτό που εννοούμε'.

Ο καλύτερος τρόπος για να σκεφτούμε τον προγραμματισμό με αυτόν τον τρόπο είναι να σκεφτούμε ότι θα είμαστε εμείς αυτοί που θα χρειαστεί να κάνουν το refactoring ή να κάνουν κάποιες τροποποιήσεις στον κώδικα μετά από καιρό. Ακόμα και για τον κώδικα που έχουμε οι ίδιοι γράψει θα είναι πολύ δύσκολο να θυμηθούμε ακριβώς τι κάνει. Για αυτό χρειάζεται να γράφουμε τον με τέτοιο τρόπο ώστε να βάζουμε μέσα ότι χρειαζόμαστε για να μην χρειάζεται να το θυμόμαστε.

Αυτή η κατανοησιμότητα του κώδικα προσφέρει ακόμα ένα προτέρημα. Καθώς εκτελούμε το refactoring μας βοηθά να κατανοούμε δύσκολο κώδικα. Στην ουσία αλλάζουμε τον κώδικα ώστε να είναι πιο εύκολος να διαβαστεί. Και καθώς γίνεται πιο ευανάγνωστός μπορούμε να κατανοήσουμε και την αρχική σχεδίαση που στην αρχή δεν ήταν ορατή.

2.4 Το Refactoring βοηθά στον εντοπισμό των bugs

Όσο πιο εύκολα μπορούμε να κατανοήσουμε ένα πρόγραμμα τόσο πιο εύκολο είναι να βρούμε πιθανά bugs. Κάποιοι άνθρωποί μπορούν να διαβάζουν έναν μεγάλο όγκο από κώδικα και να βρουν bugs. Οι περισσότεροι όμως δεν λειτουργούμε έτσι. Για να εκτελέσουμε το refactoring όμως πρέπει να κατανοήσουμε πολύ καλά τι κάνει ο υπάρχων κώδικας και βάζουμε την νέα πιο καθαρή λογική. Αποσαφηνίζοντας την δομή του προγράμματος, κάνουμε διάφορες υποθέσεις για το πως μπορεί να λειτουργήσει και αυτό βοηθά στο να βρεθούν πιθανά bugs.

2.5 Το Refactoring μας βοηθά να προγραμματίζουμε γρηγορότερα

Αυτό ακούγεται κάπως αντιφατικό. Όταν αναφερόμαστε στο Refactoring οι περισσότεροι σκέφτονται ότι βελτιώνει την ποιότητα. Πράγματι, όταν έχουμε έναν καλά σχεδιασμένο και δομημένο πρόγραμμα προγραμματίζουμε πιο γρήγορα. Χωρίς καλή σχεδίαση και δομή μπορεί να προχωρήσουμε γρήγορα για λίγο, αλλά σύντομα η ελλιπής σχεδίαση αρχίζει να μειώνει αυτό τον ρυθμό. Δαπανάται αρκετή ώρα στον εντοπισμό και την διόρθωση των bugs αντί στην περαιτέρω ανάπτυξη του συστήματος. Οι αλλαγές γίνονται πιο χρονοβόρες καθώς προσπαθούμε να κατανοήσουμε τον κώδικα και να βρούμε πιθανές διπλοεγγραφές. Νέες απαιτήσεις (features) χρειάζονται περισσότερο κώδικα να γραφεί καθώς προσπαθούμε να ‘μπαλώσουμε’ προηγούμενα ‘μπαλώματα’.

Μία καλή σχεδίαση είναι ουσιώδης για να διατηρηθεί η ταχύτητα κατά την ανάπτυξη του συστήματος. Το refactoring μας βοηθά να προγραμματίζουμε πιο γρήγορα καθώς αποτρέπει την φθορά του συστήματος.

2.6 Πότε πρέπει να κάνουμε Refactoring;

Όταν μιλάμε για refactoring οι περισσότεροι σκέφτονται πόσο χρόνο πρέπει να αφιερώσουν για αυτό. Συνήθως το refactoring δεν είναι μια ενέργεια για την οποία πρέπει να αφιερώνουμε συγκεκριμένο χρόνο κατά την ανάπτυξη του λογισμικού. Πρέπει να γίνεται συνεχώς σε μικρές δόσεις. Δεν προγραμματίζουμε το πότε θα κάνουμε refactoring, αλλά το κάνουμε επειδή χρειάζεται να κάνουμε κάτι άλλο, και το refactoring μας βοηθά να το κάνουμε.

2.6.1 Ο κανόνας των τριών

Ο κανόνας των τριών έχει ως εξής:

> Την πρώτη φορά που γράφουμε κώδικα, απλά τον γράφουμε. Την δεύτερη που θα γράψουμε κάτι παρόμοιο καταλαβαίνουμε την διπλοεγγραφή, αλλά την κάνουμε. Την τρίτη φορά που θα γράψουμε κάτι παρόμοιο, εκτελούμε το refactoring.

2.6.2 Refactoring όταν προσθέτουμε νέες λειτουργίες

Το πιο κοινό σημείο που συνήθως κάνουμε refactoring είναι όταν προσθέτουμε νέα κομμάτια λογισμικού (features) στο ήδη υπάρχον. Συχνά, ο πρώτος λόγος είναι για να μας βοηθήσει να κατανοήσουμε καλύτερα τον κώδικα που πρέπει να τροποποιήσουμε. Ο κώδικας αυτός ίσως να έχει γραφτεί από κάποιον άλλον ή ακόμα και από εμάς πριν καιρό. Όποτε χρειάζεται να καταλάβουμε τι ακριβώς κάνει ένα κομμάτι, αναρωτιόμαστε αν θα μπορούσαμε να κάνουμε refactoring ώστε να είναι πιο εύκολο και γρήγορο στην κατανόησή του. Αυτό μας βοηθά όχι μόνο για την συγκεκριμένη στιγμή που θα τροποποιήσουμε τον συγκεκριμένο κώδικα, αλλά και για την επόμενη φορά που θα χρειαστεί να ασχοληθούμε με το συγκεκριμένο κομμάτι λογισμικού.

Ο άλλος λόγος που μας ωθεί ώστε να κάνουμε refactoring σε αυτό το σημείο είναι το design του κώδικα ώστε να προστεθεί πιο εύκολα το νέο feature. Καθώς κοιτάμε τον κώδικα σκεφτόμαστε αν με το υπάρχων design μπορούμε να προσθέσουμε εύκολα και γρήγορα το νέο feature. Αν η απάντηση είναι όχι, τότε πρέπει να κάνουμε refactoring. Το refactoring είναι μία αρκετά γρήγορη και απλή διαδικασία οπότε αξίζει να γίνεται πριν την προσθήκη νέων features για να γίνουν και αυτές με την σειρά τους όσο το δυνατόν πιο γρήγορα και εύκολα.

2.6.3 Refactoring όταν διορθώνουμε ένα bug

Όταν διορθώνουμε bugs η χρήση του refactoring έρχεται ως μία βοήθεια για να γίνει ο κώδικας πιο κατανοητός. Καθώς κοιτάμε των κώδικα και προσπαθούμε να τον κατανοήσουμε, κάνουμε αλλαγές ώστε να μπορέσουμε να τον καταλάβουμε πιο εύκολα. Συνήθως το refactoring σε αυτό το σημείο μας βοηθά όχι μόνο να λύσουμε το bug με το οποίο ασχολούμαστε αλλά να βρούμε και να διορθώσουμε και άλλα bugs. Ένας τρόπος για να καταλάβουμε την ανάγκη για refactoring σε αυτό το σημείο, είναι να σκεφτούμε ότι κάθε φορά που πρέπει να διορθώσουμε ένα bug και δεν μπορούμε να εντοπίσουμε που ακριβώς είναι σημαίνει ότι ο κώδικας δεν είναι αρκετά καθαρός και σαφής.

2.6.4 Refactoring όταν κάνουνε επανεξέταση του κώδικα (code review)

Οι περισσότεροι οργανισμοί εκτελούν συχνά code review. Το code review βοηθά στο να μεταδοθεί η γνώση μέσα στην ομάδα των προγραμματιστών. Η τακτική επανεξέταση μας βοηθά ώστε να περάσει η γνώση από τους πιο έμπειρους προγραμματιστές στους λιγότερο έμπειρους. Βοηθά ώστε περισσότεροι άνθρωποι να καταλάβουν τις πτυχές ενός μεγάλου συστήματος. Είναι επίσης πολύ σημαντική για να γράφεται πιο καθαρός κώδικας. Ο κώδικας που γράφουμε μπορεί να φαίνεται καθαρός σε εμάς, αλλά όχι στην υπόλοιπη ομάδα. Αυτό είναι αναπόφευκτο. Επίσης, το code review, δίνει την ευκαιρία σε περισσότερους ανθρώπους να προτείνουν χρήσιμες ιδέες.

Το refactoring μας βοηθά να κάνουμε review τον κώδικα κάποιου άλλου. Πριν ξεκινήσουμε το refactoring, διαβάζουμε τον κώδικα, κατανοούμε το μεγαλύτερο μέρος του και κάνουμε προτάσεις για αλλαγές. Όταν λοιπόν προτείνουμε τις αλλαγές, σκαφτόμαστε αν είναι εύκολο να υλοποιηθούν. Αν όχι, τότε καταφεύγουμε στο refactoring και κάνουμε τις αλλαγές που έχουμε σκεφτεί. Όταν το κάνουμε αρκετές φορές, μπορούμε να δούμε πιο καθαρά πως μοιάζει ο κώδικας μαζί με τις προτεινόμενες αλλαγές. Δεν χρειάζεται να τις φανταστούμε, τις βλέπουμε. Αυτό έχει σαν αποτέλεσμα να προτείνουμε και νέες αλλαγές τις οποίες δεν θα σκεφτόμασταν πριν.

Το refactoring βοηθά επίσης το code review ώστε να έχει πιο συμπαγή αποτελέσματα. Όχι μόνο μας επιτρέπει να έχουμε αρκετές νέες προτάσεις για αλλαγές, αλλά οι αλλαγές αυτές εκτελούνται αμέσως. Έτσι έχουμε μια πιο ολοκληρωμένη εικόνα για τα αποτελέσματα του code review.

Για να πετύχει αυτή η διαδικασία, πρέπει να έχουμε μικρές ομάδες. Συνήθως ο προγραμματιστής που έγραψε τον κώδικα και αυτός που κάνει το review δουλεύουν μαζί. Ο reviewer προτείνει αλλαγές και οι δυο μαζί αποφασίζουν αν οι αλλαγές αυτές μπορούν να γίνουν πιο εύκολα εκτελώντας refactoring.

Όταν γίνεται review ενός μεγάλου μέρους του λογισμικού η ακόμα και της αρχικής σχεδίασής του, τότε είναι προτιμότερο να έχουμε μία ομάδα περισσότερων ατόμων που θα έχουν περισσότερες ιδέες. Το να κοιτάμε τον κώδικα δεν είναι η καλύτερη πρακτική σε αυτό το σημείο. Αντίθετα, είναι προτιμότερο να χρησιμοποιούμε διαγράμματα UML.

Η ιδέα του δραστικού code review έφτασε στα όριά της με τον ‘Ακραίο Προγραμματισμό’ (*Kent Beck, 1999, Extreme Programming Explained*), μία τεχνική Προγραμματισμού σε Ζεύγη (Pair Programming) που διατυπώθηκε από τον Kent Beck. Με αυτή την τεχνική ο σοβαρός προγραμματισμός γίνεται με δύο προγραμματιστές να δουλεύουν ταυτόχρονα στον ίδιο υπολογιστή. Αυτό έχει σαν αποτέλεσμα το συνεχές code review να γίνεται ταυτόχρονα με την διαδικασία ανάπτυξης, και με το refactoring να λαμβάνει και αυτό μέρος κατά την διάρκεια συγγραφής του κώδικα.

2.7 Γιατί το Refactoring είναι αποτελεσματικό

Σύμφωνα με τον Kent Beck, τα προγράμματα έχουν δύο ήδη αξιών: τι μπορούν να κάνουν για εμάς σήμερα και τι μπορούν να κάνουν για εμάς αύριο. Τις περισσότερες φορές όταν προγραμματίζουμε, επικεντρωνόμαστε στο τι θέλουμε το πρόγραμμα να κάνει σήμερα. Είτε διορθώνουμε ένα bug είτε προσθέτουμε ένα νέο feature, κάνουμε το σημερινό πρόγραμμα πιο πολύτιμο καθιστώντας το πιο ικανό. Δεν μπορούμε να συνεχίσουμε να προγραμματίζουμε για πολύ αν δεν αναληφθούμε ότι αυτό που κάνει σήμερα το πρόγραμμα είναι απλά μόνο ένα μέρος της ιστορίας. Πρέπει να προγραμματίζουμε με τέτοιον τρόπο ώστε το πρόγραμμά μας να δουλεύει σήμερα αλλά να είναι ικανό να δουλεύει και αύριο. Ωστόσο, αν και γνωρίζουμε τι θέλουμε να κάνει σήμερα το πρόγραμμά μας, δεν είμαστε αρκετά σίγουροι για το τι θα κάνει αύριο. Γνωρίζουμε αρκετά σχετικά με το τι θα κάνει σήμερα, αλλά σχεδόν τίποτα για το τι θα θέλουμε να κάνει αύριο.

Το refactoring είναι ένας τρόπος να μας αποδεσμεύσει από αυτό που θα κάνει το πρόγραμμά μας αύριο. Αν δούμε ότι η απόφαση που πήραμε χτες δεν έχει νόημα σήμερα, τότε αλλάζουμε την

απόφαση. Η διαδικασία αυτή συνεχίζεται καθ' όλη την διάρκεια ανάπτυξης του λογισμικού και το refactoring μας βοηθά να αλλάξουμε τις αποφάσεις που πήραμε στο παρελθόν.

Τέσσερα πράγματα κάνουν τα προγράμματα δύσκολα στο να δουλέψουμε μαζί τους:

- Προγράμματα που είναι δύσκολα να διαβάσουμε είναι δύσκολα να τροποποιήσουμε.
- Προγράμματα που έχουν διπλοεγγραφές είναι δύσκολα να τροποποιήσουμε.
- Προγράμματα που για να προστεθεί νέα λειτουργία πρέπει να αλλάξει ο υπάρχων κώδικας είναι δύσκολα να τροποποιήσουμε.
- Προγράμματα με πολύπλοκη λογική υπό συνθήκες είναι δύσκολα να τροποποιήσουμε.

Θέλουμε λοιπόν προγράμματα τα οποία είναι εύκολα να διαβάσουμε, δεν έχουν κομμάτια ίδιας λογικής σε διαφορετικά σημεία, που δεν χρειάζονται αλλαγές για να επεκταθεί η υπάρχουσα συμπεριφορά και χρησιμοποιούν την λογική υπό συνθήκες όσο το δυνατόν λιγότερο.

Το refactoring είναι η διαδικασία η οποία παίρνει ένα υπάρχων πρόγραμμα και του προσθέτει 'αξία', χωρίς να αλλάζει την συμπεριφορά του, απλά προσθέτοντας τα χαρακτηριστικά που αναφέραμε. Αυτό μας επιτρέπει να συνεχίσουμε να προγραμματίζουμε γρήγορα.

2.8 Προβλήματα με το Refactoring

Όταν μαθαίνουμε μία νέα τεχνική η οποία βελτιώνει την παραγωγικότητα μας, είναι δύσκολο να δούμε πότε αυτή η τεχνική δεν είναι χρήσιμη. Είναι δύσκολο να καταλάβουμε τι είναι αυτό που κάνει μία τεχνική είναι λιγότερο αποτελεσματική, ακόμα και επιβλαβή. Πριν από καιρό αυτό συνέβαινε με τον αντικειμενοστρεφή προγραμματισμό. Ήταν δύσκολο να βρούμε πότε δεν πρέπει να χρησιμοποιούμε τα αντικείμενα.

Το ίδιο συμβαίνει τώρα με το refactoring. Γνωρίζουμε τα πλεονεκτήματα του. Γνωρίζουμε ότι μπορεί να προσφέρει ουσιαστική διαφορά στην δουλειά μας. Αλλά δεν έχουμε ακόμα αρκετή εμπειρία ώστε να γνωρίζουμε πότε πρέπει να περιοριστεί.

Όσο περισσότερος κόσμος μαθαίνει και χρησιμοποιεί το refactoring τόσα περισσότερα θα μαθαίνουμε όλοι μας. Σίγουρα τα refactoring μπορεί να προσφέρει πολλά, πρέπει όμως να παρακολουθούμε την πρόοδό του και να καταγράφουμε τα προβλήματα που ίσως προκαλέσει.

2.8.1 Βάσεις Δεδομένων

Μία προβληματική περιοχή για refactoring είναι οι βάσεις δεδομένων. Οι περισσότερες εφαρμογές είναι σφικτά δεμένες με ένα σχήμα βάσης που τις υποστηρίζει. Αυτός είναι ένας λόγος για τον οποίο οι βάσεις δεδομένων είναι αρκετά δύσκολο να αλλάξουν. Ένα άλλος λόγος είναι η μεταφορά των δεδομένων (data migration). Ακόμα και αν έχουμε σχεδιάσει προσεκτικά το σύστημά μας σε διαφορετικά επίπεδα (layers) ώστε να μειώσουμε τις εξαρτήσεις μεταξύ σχήματος βάσης και μοντέλου αντικειμένων, μια αλλαγή στο σχήμα της βάσης αναγκάζει να μεταφέρουμε δεδομένα, το οποίο μπορεί να είναι μια χρονοβόρα και επίπονη διαδικασία.

Με τις μη αντικειμενοστρεφείς βάσεις δεδομένων ένας τρόπος για να αντιμετωπιστεί αυτό το πρόβλημα είναι να προσθέσουμε ένα ξεχωριστό επίπεδο λογισμικού ανάμεσα στο μοντέλο αντικειμένων και το μοντέλο της βάσης. Με αυτό τον τρόπο μπορούμε να απομονώσουμε τις αλλαγές στα δύο διαφορετικά επίπεδα. Καθώς ενημερώνουμε το ένα μοντέλο, δεν χρειάζεται να ενημερώσουμε και το άλλο. Ενημερώνουμε απλά το ενδιάμεσο επίπεδο. Ένα τέτοιο επίπεδο προσθέτει πολυπλοκότητα αλλά μας δίνει μεγαλύτερη ευελιξία. Ακόμα και χωρίς refactoring είναι πολύ σημαντικό σε περιπτώσεις που χρησιμοποιούμε πολλές βάσεις δεδομένων ή ένα πολύπλοκο σχήμα βάσης στο οποίο δεν έχουμε έλεγχο.

Οι αντικειμενοστρεφείς βάσεις δεδομένων μπορούν και να βοηθήσουν και να εμποδίσουν. Κάποιες βάσεις παρέχουν αυτόματη μεταφορά δεδομένων από την μία έκδοση στην άλλη. Αυτό μειώνει την προσπάθεια αλλά εξακολουθεί να επιφέρει κάποια καθυστέρηση καθώς γίνεται η μεταφορά. Όταν η μεταφορά δεν είναι αυτόματη, πρέπει να την κάνουμε μόνοι μας. Αυτό απαιτεί μεγάλη προσπάθεια. Σε αυτή την περίπτωση πρέπει να είμαστε πιο προσεκτικοί σχετικά με τις αλλαγές στην δομή των δεδομένων των κλάσεων. Πρέπει να είμαστε προσεκτικοί όσον αφορά την μετακίνηση πεδίων.

2.8.2 Αλλαγή των διαπαφών (Interfaces)

Ένα από τα πιο βασικά πράγματα με τα αντικείμενα είναι ότι μας επιτρέπουν να αλλάζουμε την υλοποίηση τους ξεχωριστά από το interface τους. Μπορούμε εύκολα να αλλάξουμε το εσωτερικό ενός αντικειμένου χωρίς κανείς να καταλάβει τίποτα. Αλλά το interface είναι σημαντικό, η αλλαγή του έχει μεγάλες επιρροές.

Κάτι το οποίο είναι ενοχλητικό σχετικά με το refactoring είναι ότι συνήθως γίνονται αλλαγές και στα interfaces. Κάτι πολύ απλό όπως η μετονομασία μιας μεθόδου χρειάζεται αλλαγές και στο interface.

Δεν υπάρχει κανένα πρόβλημα με το να αλλάζουμε το όνομα μια μεθόδου. Ακόμα και αν η μέθοδος είναι δηλωμένη ως public, αν έχουμε πρόσβαση και μπορούμε να αλλάξουμε όλες τις κλήσεις της, μπορούμε εύκολα να την μετονομάσουμε. Αυτό δημιουργεί πρόβλημα μόνο όταν το interface χρησιμοποιείται από κώδικα στον οποίο δεν έχουμε πρόσβαση και δεν μπορούμε να αλλάξουμε. Όταν συμβαίνει αυτό, λέμε ότι το interface είναι δημοσιευμένο (published), ένα βήμα πέρα από το public. Όταν ένα interface είναι published, δεν μπορούμε να κάνουμε αλλαγές σε αυτό με ασφάλεια διότι κάθε αλλαγή θα επηρεάσει και τις κλήσεις του στις οποίες δεν έχουμε πρόσβαση. Χρειαζόμαστε μία πιο πολύπλοκη διαδικασία.

Αν το Refactoring αλλάζει ένα δημοσιευμένο interface, τότε πρέπει να διατηρήσουμε και το παλιό interface και το καινούριο, τουλάχιστον μέχρις ότου όλοι οι εξωτερικοί χρήστες να μπορέσουν να αλλάξουν τον κώδικα τους. Ευτυχώς αυτό δεν είναι πολύ δύσκολο. Συνήθως διαμορφώνουμε τις αλλαγές ώστε το παλιό interface να εξακολουθεί να δουλεύει. Πρέπει να προσπαθήσουμε να κάνουμε τις αλλαγές ώστε οι κλήσεις στο παλιό interface να καλούν το καινούριο. Δεν χρειάζεται να αντιγράψουμε το σώμα της μεθόδου. Αυτό θα οδηγούσε σε διπλοεγγραφές. Αυτό που πρέπει να κάνουμε είναι να χρησιμοποιήσουμε τον μηχανισμό αντικατάστασης (deprecation) τον οποίο μας παρέχει η Java, ώστε να σημειώσουμε τον κώδικα ως ‘αντικατεστημένο’ (deprecated). Με αυτόν τον

τρόπο οι εξωτερικοί χρήστες του συστήματος θα γνωρίζουν ότι κάτι έχει αλλάξει. Ένα καλό παράδειγμα αυτής της διαδικασίας είναι οι κλάσεις συλλογής (collection classes) της Java.

Η διατήρηση των interfaces είναι εφικτή αλλά επίπονη. Πρέπει να χτίσουμε και να συντηρήσουμε αυτές τις νέες μεθόδους για τουλάχιστον μία φορά. Οι μέθοδοι περιπλέκουν το interface, κάνοντάς πιο δύσκολο στην χρήση. Ο μόνος τρόπος για να αποφευχθεί αυτό είναι να μην δημοσιεύουμε τα interfaces. Βέβαια κάτι τέτοιο είναι αδύνατο αν φτιάχνουμε APIs (Application programming interface - Διεπαφή προγραμματισμού εφαρμογών) για εξωτερική χρήση. Η χρήση published interfaces είναι χρήσιμη αλλά έχει και ένα κόστος. Οπότε δεν χρειάζεται να τα χρησιμοποιούμε εκτός και αν πραγματικά πρέπει να το κάνουμε.

Υπάρχει μία συγκεκριμένη περιοχή όπου δημιουργούνται προβλήματα όταν κάνουμε αλλαγές σε interfaces: η προσθήκη ενός exception στην δήλωση *throws*. Αυτό δεν αλλάζει την υπογραφή (signature) της μεθόδου, οπότε δεν μπορούμε χρησιμοποιήσουμε μία νέα μέθοδο με το ίδιο όνομα που θα την αντικαθιστά. Δεν θα μας το επιτρέψει ο compiler. Είναι αρκετά δύσκολο να αντιμετωπίσουμε αυτό το πρόβλημα. Μπορούμε να επιλέξουμε ένα νέο όνομα για την καινούρια μέθοδο, και να βάλουμε την παλιά να την καλεί. Μπορούμε επίσης να προσθέσουμε το exception στην υπάρχουσα μέθοδο και να ενημερώσουμε όλες τις κλήσεις της ώστε να το ελέγχουν.

2.8.3 Πότε δεν πρέπει να κάνουμε Refactoring

Υπάρχουν περιπτώσεις όπου δεν θα πρέπει να κάνουμε καθόλου refactoring. Το αντιπροσωπευτικό παράδειγμα είναι όταν θα πρέπει να γράψουμε τον κώδικα πάλι από την αρχή. Υπάρχουν φορές που ο υπάρχων κώδικας είναι τόσο άσχημος που αν και θα μπορούσαμε να κάνουμε refactoring, θα ήταν προτιμότερο και ευκολότερο να τον ξαναγράψουμε από την αρχή.

Ένα καθαρό σημάδι ότι ο κώδικας πρέπει να ξαναγραφτεί είναι όταν απλά δεν λειτουργεί. Μπορούμε να το διαπιστώσουμε απλά με το να τον τεστάρουμε και όταν δούμε ότι είναι γεμάτος bugs καταλαβαίνουμε ότι δεν γίνεται να τον διορθώσουμε. Ο κώδικας πρέπει να λειτουργεί σχεδόν σωστά πριν πάμε να κάνουμε refactoring.

2.9 Refactoring και Design

Το Refactoring παίζει ένα σημαντικό ρόλο ως συμπλήρωμα στο design. Όσο πιο έμπειροι γινόμαστε με τον προγραμματισμό τόσο κατανοούμε ότι όταν το design γίνεται εκ των προτέρων αποφεύγουμε επιπλέον δουλειά. Αυτή είναι η αρχή του Upfront Design. Όσο περισσότερος χρόνος αφιερωθεί στο design, τόσο περισσότερος χρόνος θα κερδηθεί στην συνέχεια όταν θα πρέπει να διορθωθούν bugs και να προστεθούν νέα features.

Πολλοί πιστεύουν ότι το αρχικό design είναι απλά το κλειδί για να ξεκινήσουν να προγραμματίζουν. Ένα επιχείρημα προς αυτή την κατεύθυνση είναι ότι το refactoring μπορεί να χρησιμοποιηθεί ως μία εναλλακτική του Upfront Design. Σε αυτή την περίπτωση δεν κάνουμε καθόλου design. Απλά ξεκινάμε να γράφουμε κώδικα όπως μπορούμε αρχικά να σκεφτούμε, τον κάνουμε να λειτουργεί και έπειτα κάνουμε refactoring.

Αν και η παραπάνω προσέγγιση μπορεί εν μέρει να δουλέψει, το να κάνουμε μόνο refactoring δεν είναι και ο πιο αποτελεσματικός τρόπος να δουλεύουμε. Ακόμα και οι πιο καλοί προγραμματιστές κάνουν το design στην αρχή. Μόνο μετά τη δημιουργία ενός αξιόπιστου πρώτου πλάνου ξεκινάνε να προγραμματίζουν και έπειτα να κάνουν refactoring. Η ιδέα είναι ότι το refactoring αλλάζει τον ρόλο του upfront design. Αν δεν κάνουμε refactoring υπάρχει περισσότερη πίεση ώστε το αρχικό design να βγει απόλυτα σωστό. Το νόημα είναι ότι ενδεχόμενες αλλαγές στο design αργότερα θα χρειάζονται αρκετό χρόνο. Άρα αφιερώνουμε αρκετό χρόνο και προσπάθεια στο upfront design για να αποφύγουμε τέτοιες αλλαγές.

Με το refactoring η έμφαση αλλάζει. Εξακολουθούμε να δίνουμε πολύ χρόνο στο αρχικό design αλλά δεν προσπαθούμε να βρούμε μία τέλεια λύση. Καθώς χτίζουμε το σύστημα, κατανοούμε καλύτερα το πρόβλημα και διαπιστώνουμε ότι η καλύτερη λύση είναι διαφορετική από την αρχική. Με το refactoring αυτό δεν είναι πρόβλημα, γιατί οι αλλαγές δεν είναι δύσκολο να γίνουν.

Με την χρήση του refactoring ως συμπλήρωμα στο design κερδίζουμε αρκετό χρόνο. Δεν χρειάζεται να ξοδέψουμε άπειρο χρόνο ώστε να βρούμε ένα design που θα αντιμετωπίζει κάθε πιθανό πρόβλημα που μπορεί να προκύψει με την προσθήκη νέων features. Αφιερώνουμε όσο χρόνο χρειάζεται ώστε να έχουμε ένα καλό αρχικό design και στην συνέχεια με το refactoring μπορούμε να κάνουμε τις αλλαγές που χρειάζονται.

3 Bad Smells στον κώδικα

Το δυσκολότερο πράγμα με το refactoring δεν είναι να το εκτελούμε αλλά να γνωρίζουμε που και πότε να το εφαρμόζουμε. Το να αποφασίζουμε πότε να ξεκινήσουμε και πότε να σταματήσουμε το refactoring είναι εξίσου σημαντικό με το να γνωρίζουμε πως να το εφαρμόζουμε.

Είναι εύκολο να μάθουμε πως να διαγράφουμε μία τοπική μεταβλητή ή πως να δημιουργούμε μία ιεραρχική δομή. Το να μάθουμε όμως πότε να εφαρμόζουμε τις τεχνικές refactoring δεν είναι εύκολο.

Για να βρούμε πότε πρέπει να κάνουμε refactoring δεν χρησιμοποιούμε κάποια τεχνική ή θεωρία. Το μόνο που κάνουμε είναι να ψάχνουμε για Bad Smells (κακές 'οσμές') στον κώδικα. Όσο περισσότερο προγραμματίζομε και όσο περισσότερο κοιτάμε και διαβάζουμε κώδικα τόσο μαθαίνουμε να βρίσκουμε σημεία του κώδικα που καταλαβαίνουμε ότι χρειάζονται refactoring.

3.1 Διπλοεγγραφές κώδικα

Αν δούμε το ίδιο κομμάτι κώδικα να υπάρχει σε περισσότερα από ένα μέρη, τότε μπορούμε να είμαστε σίγουρο ότι θα είναι προτιμότερο να τα ενοποιήσουμε.

Το το πιο συχνό πρόβλημα διπλοεγγραφής κώδικα είναι όταν δύο μέθοδοι της ίδιας κλάσης έχουν ένα κοινό κομμάτι κώδικα. Τότε το μόνο που χρειάζεται να κάνουμε είναι να χρησιμοποιήσουμε την τεχνική της Εξαγωγής Μεθόδου (βλέπε *Extract Method (Εξαγωγή Μεθόδου)*) (SourceMaking teaching IT professionals, 2013, Extract Method) και να βγάλουμε τον κώδικα και από τα δύο σημεία.

Μία άλλη κοινή περίπτωση διπλοεγγραφής είναι όταν κομμάτι κώδικα βρίσκεται σε δύο υποκλάσεις. Και πάλι μπορούμε να εξαλείψουμε την διπλοεγγραφή χρησιμοποιώντας την Εξαγωγή Μεθόδου και στις δύο κλάσεις και μετά την τεχνική Μετακίνηση Πεδίου προς τα πάνω (*SourceMaking, 2013, Pull Up Field*). Αν ο κώδικας είναι παρόμοιος αλλά όχι ο ίδιος πρέπει να χρησιμοποιήσουμε την Εξαγωγή Μεθόδου για να διαχωρίσουμε τα ίδια από τα διαφορετικά κομμάτια κώδικα. Έπειτα μπορούμε να χρησιμοποιήσουμε την τεχνική της Μορφοποίησης Μεθόδου Πρότυπο (*SourceMaking, 2013, Form Template Method*). Αν οι μέθοδοι κάνουν το ίδιο πράγμα με διαφορετικό αλγόριθμο, τότε πρέπει να επιλέξουμε τον καλύτερο από τους δύο αλγόριθμους και να χρησιμοποιήσουμε την τεχνική της Αντικατάστασης Αλγορίθμου (βλέπε *Substitute Algorithm (Αντικατάσταση Αλγορίθμου)*).

Αν έχουμε διπλοεγγραφή σε δύο μη σχετιζόμενες κλάσεις, χρησιμοποιούμε την Εξαγωγή Κλάσης (βλέπε *Extract Class (Εξαγωγή Κλάσης)*) σε μία από τις δύο κλάσεις και μετά να χρησιμοποιήσουμε το καινούριο κομμάτι στην ως κλήση στην άλλη κλάση.

3.2 Μεγάλη μέθοδος

Τα αντικειμενοστρεφή προγράμματα που ζουν περισσότερο είναι αυτά με τις μικρότερες μεθόδους. Οι προγραμματιστές που είναι νέοι στην αντικειμενοστρέφεια συχνά νοιώθουν πως κανένας υπολογισμός δεν λαμβάνει πραγματικά μέρος και ότι τα προγράμματα είναι ατελείωτες ακολουθίες. Όταν όμως χρησιμοποιούμε ένα τέτοιο πρόγραμμα για χρόνια μαθαίνουμε πόσο πολύτιμες είναι όλες αυτές οι μικρές μέθοδοι.

Από τις αρχές του προγραμματισμού οι προγραμματιστές κατάλαβαν ότι όσο μεγαλύτερη είναι μία μέθοδος τόσο πιο δύσκολο είναι να την καταλάβουμε. Οι παλαιότερες γλώσσες προγραμματισμού είχαν μεγάλο βάρος στις κλήσεις ρουτινών οπότε οι περισσότεροι απέφευγαν να χρησιμοποιούν μικρές μεθόδους. Οι σύγχρονες αντικειμενοστρεφείς μέθοδοι όμως έχουν περιορίσει αρκετά αυτό το βάρος. Όμως αυτό δυσκολεύει τους προγραμματιστές να διαβάσουν τον κώδικα γιατί πρέπει να αλλάζουν το περιεχόμενο συχνά ώστε να κατανοήσουν τι κάνει μία υπορουτίνα. Το κλειδί για να γίνει πιο εύκολη αυτή η διαδικασία είναι η καλή ονομασία των μεθόδων. Αν δώσουμε ένα καλό όνομα σε μία μέθοδο δεν χρειάζεται να δούμε το σώμα της για να καταλάβουμε τι κάνει.

Το αποτέλεσμα είναι ότι θα πρέπει είμαστε πιο αποφασιστικοί σχετικά με την αποσύνθεση των μεθόδων. Μία απλή τεχνική είναι ότι όταν χρειαζόμαστε να γράψουμε ένα σχόλιο για κάτι που εκτελεί μία μέθοδος τότε γράφουμε μία καινούρια. Μία τέτοια μέθοδος περιέχει τον κώδικα για τον οποίο θα βάζαμε το σχόλιο και έχει σαν όνομα αυτό που κάνει η μέθοδος παρά το πως το κάνει. Μπορούμε να το κάνουμε αυτό για μία ομάδα από γραμμές κώδικα ή ακόμα και για μία μόνο γραμμή. Μπορούμε να το κάνουμε ακόμα και αν η κλήση της μεθόδου είναι μεγαλύτερη και από τον κώδικα τον οποίο αντικαθιστά. Το κλειδί εδώ δεν είναι το μέγεθος της μεθόδου αλλά η σημασιολογική διαφορά ανάμεσα στο τι κάνει η μέθοδος και στο πως το κάνει.

Στις περισσότερες περιπτώσεις αυτό που χρειάζεται να κάνουμε για να μικρύνουμε μία μέθοδο είναι να χρησιμοποιήσουμε την Εξαγωγή Μεθόδου. Βρίσκουμε τα κομμάτια της μεθόδου που πιστεύουμε ότι ταιριάζουν καλύτερα μαζί και τα βάζουμε σε μία νέα μέθοδο.

Αν έχουμε μία μέθοδο με πολλές παραμέτρους και τοπικές μεταβλητές δεν είναι εύκολο να χρησιμοποιήσουμε την Εξαγωγή Μεθόδου. Αν το προσπαθήσουμε, θα δούμε ότι θα περνάμε τόσες πολλές παραμέτρους και τοπικές μεταβλητές στην νέα μέθοδο που το αποτέλεσμα θα είναι χειρότερο από το αρχικό. Σε αυτή την περίπτωση μπορούμε να χρησιμοποιήσουμε την Αντικατάσταση Προσωρινής Μεταβλητής (βλέπε *Replace Temp with Query (Αντικατάσταση Προσωρινής Μεταβλητής με Μέθοδο)*) για να εξαλείψουμε τις μεταβλητές. Μία μεγάλη λίστα παραμέτρων μπορεί να μικρύνει με την Εισαγωγή Παραμέτρου Αντικειμένου (*SourceMaking, 2013, Introduce Parameter Object*) και την Διατήρηση Ολόκληρου Αντικειμένου (*SourceMaking, 2013, Preserve Whole Object*).

Εάν τα κάνουμε όλα αυτά αλλά και πάλι έχουμε πολλές μεταβλητές και παραμέτρους τότε χρησιμοποιούμε την Αντικατάσταση Μεθόδου με Μέθοδο Αντικειμένου (βλέπε *Replace Method with Method Object (Αντικατάσταση Μεθόδου με Μέθοδο Αντικειμένου)*).

Μία καλή μέθοδος για να εντοπίσουμε τα κομμάτια κώδικα που πρέπει να εξάγουμε είναι να κοιτάξουμε τα σχόλια. Αυτά συχνά μας δείχνουν αυτή την σημασιολογική διαφορά. Ένα κομμάτι κώδικα με ένα σχόλιο που λέει ακριβώς τι κάνει μπορεί να αντικατασταθεί με μία μέθοδο της οποίας το όνομα βασίζεται στο σχόλιο.

Οι συνθήκες και οι βρόχοι επανάληψης αποτελούν επίσης ένα ένδειξη για εξαγωγή. Με την χρήση της Αποσύνθεσης Συνθήκης (*SourceMaking, 2013, Decompose Conditional*) μπορούμε να εξαλείψουμε τις

συνθήκες μέσα σε μία μέθοδο. Για την αντιμετώπιση των βρόχων επανάληψης μπορούμε να εξάγουμε τον βρόχο και τον κώδικα που περιέχεται σε αυτόν σε μία νέα μέθοδο.

3.3 Μεγάλη κλάση

Όταν μία κλάση προσπαθεί να κάνει πάρα πολλά, συχνά χρησιμοποιεί πάρα πολλές μεταβλητές. Αυτό συνήθως οδηγεί σε διπλοεγγραφές κώδικα.

Μπορούμε να κάνουμε Εξαγωγή Κλάσης (βλέπε *Extract Class (Εξαγωγή Κλάσης)*) για να συγχωνεύσουμε μία σειρά από μεταβλητές. Επιλέγουμε αυτές τις μεταβλητές που ταιριάζουν να είναι μαζί. Για παράδειγμα, η 'depositAmount' και η 'depositCurrency' φαίνεται να ανήκουν στο ίδιο κομμάτι κώδικα. Πιο γενικά, τα σύνολα των κοινών μεταβλητών μέσα σε μία κλάση μας δίνουν την δυνατότητα να δημιουργήσουμε μία καινούρια. Αν η νέα κλάση ταιριάζει να είναι υποκλάση της αρχικής τότε χρησιμοποιούμε την Εξαγωγή Υποκλάσης (*SourceMaking, 2013, Extract Subclass*) για μεγαλύτερη ευκολία.

Όπως μία κλάση με πολλές μεταβλητές, έτσι και μία κλάση με πολύ κώδικα είναι πολύ πιθανόν να οδηγήσει σε διπλοεγγραφές. Η πιο απλή λύση είναι να εξαλείψουμε τον πλεονασμό μέσα στην ίδια την κλάση. Αν η κλάση έχει μεθόδους με κάποιες εκατοντάδες γραμμές κώδικα που έχουν πολλά κοινά σημεία, μπορούμε να τις μετατρέψουμε σε μεθόδους κάποιων δεκάδων γραμμών απλά προσθέτοντας λίγες ακόμα μεθόδους λίγων γραμμών τις οποίες εξάγουμε από τις αρχικές.

Όπως με τις κλάσεις που έχουν πολλές μεταβλητές, έτσι και στις κλάσεις με πολύ κώδικα η πιο κοινή λύση είναι η Εξαγωγή Κλάσης ή η Εξαγωγή Υποκλάσης. Ένα έξυπνο κόλπο είναι να καθορίσουμε πως οι πελάτες θα κάνουν χρήση της κλάσης και να χρησιμοποιήσουμε την μέθοδο Εξαγωγής Διεπαφής (*SourceMaking, 2013, Extract Interface*) για κάθε μία από τις προηγούμενες μεθόδους.

3.4 Μεγάλη λίστα παραμέτρων

Στις αρχές του προγραμματισμού περνούσαμε ως παραμέτρους στις ρουτίνες οποιαδήποτε πληροφορία αυτές χρειάζονταν. Αυτό είναι κατανοητό γιατί η εναλλακτική ήταν να έχουμε καθολικά δεδομένα (global data). Τα αντικείμενα το άλλαξαν αυτό γιατί πλέον δεν χρειάζεται να έχουν κάτι που δεν χρειάζονται, απλά ζητάνε από κάποιον άλλον να τους το δώσει. Για αυτό, με τα αντικείμενα δεν περνάμε ως παραμέτρους οτιδήποτε χρειάζεται μία μέθοδος, αλλά περνάμε μόνο αυτά ώστε η μέθοδος να μπορεί να βρει την πληροφορία που χρειάζεται. Τα περισσότερα από αυτά που χρειάζεται μία μέθοδος είναι διαθέσιμα από την κλάση στην οποία ανήκει. Στον αντικειμενοστραφή προγραμματισμό οι λίστες των παραμέτρων των μεθόδων τείνουν να είναι όλο και μικρότερες.

Αυτό είναι καλό γιατί όσο μεγαλύτερη είναι η λίστα των παραμέτρων τόσο πιο δύσκολο είναι να καταλάβουμε την χρησιμότητα της κάθε μιας επειδή είναι δύσκολες στην χρήση τους και επειδή συνεχώς τις αλλάζουμε καθώς χρειαζόμαστε περισσότερα δεδομένα.

Χρησιμοποιώντας την Αντικατάσταση Παραμέτρου με Μέθοδο (*SourceMaking, 2013, Replace Parameter with Method*) μπορούμε να πάρουμε τα δεδομένα που χρειαζόμαστε σε μία μόνο παράμετρο που είναι ένα αντικείμενο και έπειτα να καλούμε τις μεθόδους του αντικειμένου αυτού.

Αυτό το αντικείμενο μπορεί να είναι ένα πεδίο της κλάσης ή μπορεί να είναι και μία άλλη παράμετρος. Με την χρήση της Διατήρησης Ολόκληρου Αντικειμένου μπορούμε να πάρουμε μία σειρά από δεδομένα που προέρχονται από ένα αντικείμενο και να τα αντικαταστήσουμε με το αντικείμενο αυτό καθαυτό.

3.5 Αποκλίνουσα αλλαγή (Divergent Change)

Δομούμε το πρόγραμμά μας για να κάνουμε πιο εύκολες τις αλλαγές. Όταν κάνουμε μία αλλαγή θέλουμε να πάμε σε ένα μοναδικό σημείο του κώδικα και να κάνουμε την τροποποίηση που χρειάζεται.

Η αποκλίνουσα αλλαγή προκύπτει όταν μία κλάση αλλάζει με διαφορετικούς τρόπους για διαφορετικούς λόγους. Όταν για παράδειγμα πρέπει να αλλάξουμε τρεις διαφορετικές μεθόδους μίας κλάσης κάθε φορά που αλλάζει η βάση δεδομένων ή κάθε φορά που προστίθεται ένας νέος τύπος στο σύστημα (π.χ. τρόπος πληρωμής) τότε είναι προτιμότερο να χρησιμοποιούμε δύο αντικείμενα παρά ένα. Με αυτό τον τρόπο κάθε αντικείμενο θα αλλάζει μόνο ως αποτέλεσμα μόνο ενός είδους αλλαγής. Οποιαδήποτε αλλαγή χρειάζεται να χειριστεί μία εξωτερική αλλαγή στο σύστημα θα πρέπει να αλλάζει μόνο μία κλάση, και όλες οι αλλαγές στην κλάση θα πρέπει να εκφράζουν απόλυτα την εξωτερική αλλαγή. Πρέπει να προσδιορίσουμε όλες τις αλλαγές που χρειάζονται να γίνουν σε μία κλάση για μία εξωτερική αλλαγή και να τις βάλουμε όλες μαζί σε μία νέα κλάση με την μέθοδο της Εξαγωγής κλάσης.

3.6 Shotgun Surgery

Το Shotgun Surgery είναι παρόμοιο με την αποκλίνουσα αλλαγή αλλά είναι το ακριβώς αντίθετο. Το καταλαβαίνουμε όταν κάθε φορά που κάνουμε ένα είδος αλλαγής χρειάζεται να κάνουμε επιπλέον πολλές μικρές αλλαγές σε διαφορετικές κλάσεις. Όταν οι αλλαγές αυτές βρίσκονται σε κάθε σημείο του κώδικα είναι πολύ δύσκολο να βρεθούν και είναι εύκολο να ξεφύγει μία σημαντική αλλαγή.

Σε αυτή την περίπτωση χρησιμοποιούμε την Μετακίνηση Μεθόδου (βλέπε *Move Method (Μετακίνηση Μεθόδου)*5.1) και την Μετακίνηση Πεδίου (βλέπε *Move Field (Μετακίνηση Πεδίου)*) για να βάλουμε όλες τις αλλαγές σε μία κλάση. Εάν καμία από τις υπάρχουσες κλάσεις δεν μοιάζει να είναι κατάλληλη τότε χρησιμοποιούμε μία καινούρια.

Η διαφορά μεταξύ της Αποκλίνουσας Αλλαγής και του Shotgun Surgery είναι ότι στην πρώτη περίπτωση έχουμε μία κλάση η οποία υποφέρει από πολλές αλλαγές ενώ στην δεύτερη έχουμε μία αλλαγή η οποία τροποποιεί πολλές κλάσεις.

3.7 Feature Envy

Το όλο νόημα με τα αντικείμενα είναι να ότι παρέχουν μία τεχνική να ‘πακετάρουμε’ τα δεδομένα μαζί με όλες τις λειτουργίες τους. Μία κλασική ‘οσμή’ (smell) είναι μία μέθοδος η οποία φαίνεται να ταιριάζει σε μία άλλη κλάση παρά σε αυτή που βρίσκεται. Συνήθως το κοινό στοιχείο αυτών των μεθόδων είναι το μεγάλο πλήθος των δεδομένων που χρησιμοποιούν. Μπορεί να έχουν δεκάδες κλήσεις σε μεθόδους κάποιου άλλου αντικειμένου για να υπολογίσουν μία τιμή. Η επιλογή εδώ είναι προφανής, η μέθοδος πρέπει να μεταφερθεί σε άλλη κλάση. Χρησιμοποιούμε λοιπόν την Μετακίνηση Μεθόδου για να την μεταφέρουμε στην σωστή κλάση. Κάποιες φορές μόνο ένα μέρος της μεθόδου

υποφέρει από ‘ζήλεια’. Τότε χρησιμοποιούμε την Εξαγωγή Μεθόδου για να εξάγουμε το κομμάτι από την μέθοδο σε μία νέα, και έπειτα την Μετακίνηση Μεθόδου για να μεταφέρουμε την νέα μέθοδο στην κλάση που ταιριάζει περισσότερο.

Συχνά όμως, μία μέθοδος μπορεί να χρησιμοποιεί δεδομένα από πολλές κλάσεις. Σε ποια λοιπόν από αυτές τις κλάσεις πρέπει να μεταφερθεί η μέθοδος; Τότε εξετάζουμε από ποια κλάση ζητάει περισσότερα δεδομένα η μέθοδος και την μεταφέρουμε σε αυτή. Μπορούμε επίσης να εξάγουμε πολλές περισσότερες μεθόδους ανάλογα με τα δεδομένα που χρησιμοποιούνται και έπειτα να τις μεταφέρουμε στις κλάσεις που θα έπρεπε να ανήκουν.

3.8 Data Clumps (Δέσμες Δεδομένων)

Πολλές φορές βλέπουμε μικρές συλλογές από δεδομένα να μεταφέρονται μαζί. Τρία ή τέσσερα πεδία μιας κλάσης μεταφέρονται μαζί σε διάφορες κλάσεις ή αποτελούν όλα παραμέτρους σε μία μέθοδο. Τέτοιες δέσμες δεδομένων που μεταφέρονται συνεχώς όλα μαζί πρέπει να βρίσκονται στο δικό τους ξεχωριστό αντικείμενο.

Το πρώτο που πρέπει να κάνουμε είναι να κοιτάξουμε που αυτά τα δεδομένα δηλώνονται ως πεδία. Μετά χρησιμοποιούμε την Εξαγωγή Κλάσης (βλέπε *Extract Class (Εξαγωγή Κλάσης)*) σε αυτά τα πεδία για να τα μετατρέψουμε σε αντικείμενο. Αφού λοιπόν δημιουργήσαμε την νέα κλάση η οποία περιέχει όλα αυτά τα πεδία, πρέπει να αλλάξουμε και τις μεθόδους στις οποίες τα πεδία περνούσαν ως παράμετροι. Για να γίνει αυτό κάνουμε χρήση της Εισαγωγής Παραμέτρου Αντικειμένου ή της Διατήρησης Ολόκληρου Αντικειμένου για να μειώσουμε την λίστα των παραμέτρων τους. Το άμεσο αποτέλεσμα είναι ότι μικραίνει η λίστα και απλοποιείται η κλήση της μεθόδου.

Ένας πολύ καλός τρόπος για να βρούμε πότε έχουμε μία ‘οσμή’ τύπου Data Clumps είναι να διαγράψουμε ένα από τα πεδία της δέσμης δεδομένων. Αν το κάνουμε και δούμε ότι τα υπόλοιπα πεδία δεν βγάζουν κανένα νόημα από μόνα τους, τότε αυτό είναι ένα ξεκάθαρο σημάδι ότι τα πεδία αυτά πρέπει να μπουν στο δικό τους αντικείμενο.

Η μείωση του αριθμού των πεδίων και των λιστών παραμέτρων στις μεθόδους σίγουρα θα αφαιρέσει κάποιες κακές οσμές από τον κώδικα, αλλά μπορεί να δημιουργήσει κάποιες άλλες. Αφού αφαιρέσουμε τις δέσμες δεδομένων πρέπει να εξετάσουμε μήπως τώρα έχουμε κάποια οσμή τύπου Feature Envy. Μήπως δηλαδή κάποια από τις μεθόδους που χρησιμοποιούσαν όλα αυτά τα δεδομένα πρέπει να μεταφερθεί στην νέα κλάση.

3.9 Primitive Obsession (Εμμονή Πρωταρχικών Τύπων)

Τα περισσότερα περιβάλλοντα προγραμματισμού παρέχουν δύο είδη από δεδομένα. Οι τύποι δεδομένων μας επιτρέπουν να χτίσουμε τα δεδομένα μας σε ομάδες. Οι πρωταρχικοί τύποι αποτελούν τα θεμέλια των τύπων δεδομένων.

Ένα αξιοσημείωτο πράγμα σχετικά με τα αντικείμενα είναι ότι αφήνουν κάπως ασαφή την διαχωριστική γραμμή μεταξύ των πρωταρχικών τύπων και των μεγάλων κλάσεων. Μπορούμε πολύ εύκολα να γράψουμε μικρές κλάσεις οι οποίες δεν διαφέρουν πολύ από τους πρωταρχικούς τύπους. Η

Java παρέχει πρωταρχικούς τύπους για τους ακέραιους, αλλά οι τύποι string και date, που σε πολλά άλλα περιβάλλοντα είναι πρωταρχικοί, εδώ είναι κλάσεις.

Οι προγραμματιστές που είναι νέοι στα αντικείμενα συχνά είναι απρόθυμοι να χρησιμοποιήσουν μικρά αντικείμενα για μικρές εργασίες, όπως μία κλάση Money η οποία θα συνδυάζει ποσά και νομίσματα.

Στις περιπτώσεις που έχουμε μία οσμή όπως στο παράδειγμα που αναφέραμε μπορούμε να κάνουμε Αντικατάσταση Τιμής Δεδομένων με Αντικείμενο (*SourceMaking, 2013, Replace Data Value with Object*) σε μεμονωμένες τιμές δεδομένων. Αν η τιμή των δεδομένων εκφράζει έναν κωδικό (π.χ. enums) και δεν επηρεάζει την συμπεριφορά των δεδομένων τότε χρησιμοποιούμε την Αντικατάσταση Κωδικού Τύπου με Κλάση (*SourceMaking, 2013, Replace Type Code with Class*). Στις περιπτώσεις όπου έχουμε συνθήκες που εξαρτώνται από τον κωδικό του τύπου χρησιμοποιούμε την Αντικατάσταση Τύπου Κωδικού με Υποκλάσεις (*SourceMaking, 2013, Replace Type Code with Subclasses*) ή την Αντικατάσταση Τύπου Κωδικού με Κατάσταση (*SourceMaking, 2013, Replace Type Code with State/Strategy*).

3.10 Switch Statements (Συνθήκες Switch)

Ένα από τα προφανή προτερήματα του αντικειμενοστρεφούς προγραμματισμού είναι η συγκριτική έλλειψη των συνθηκών switch. Το πρόβλημα με τις συνθήκες switch είναι κυρίως οι διπλοεγγραφές. Συχνά βλέπουμε την ίδια συνθήκη switch να υπάρχει σε διάφορα σημεία του προγράμματος. Αν προσθέσουμε μία ακόμα συνθήκη τότε θα πρέπει να βρούμε όλες τις συνθήκες switch μέσα στον κώδικα για να κάνουμε την αλλαγή που χρειάζεται. Η αντικειμενοστρεφής έννοια του Πολυμορφισμού μας δίνει έναν κομψό τρόπο να λύσουμε αυτό το πρόβλημα.

Τις περισσότερες φορές που βλέπουμε μία συνθήκη switch σκεφτόμαστε τον πολυμορφισμό. Το πρόβλημα είναι σε ποιες από αυτές τις περιπτώσεις πρέπει να προκύψει ο πολυμορφισμός. Συχνά οι συνθήκες αυτές ελέγχουν έναν τύπο κωδικού. Άρα θέλουμε να έχουμε μία μέθοδο η μία κλάση η οποία θα εκφράζει τον κωδικό. Για αυτό χρησιμοποιούμε την Εξαγωγή Μεθόδου για να εξάγουμε την συνθήκη switch και έπειτα την Μετακίνηση Μεθόδου για να μεταφέρουμε την μέθοδο στην κλάση από όπου θα προκύψει ο πολυμορφισμός. Σε αυτό το σημείο πρέπει να αποφασίσουμε αν θα χρησιμοποιήσουμε την τεχνική Αντικατάσταση Τύπου Κωδικού με Υποκλάση ή την Αντικατάσταση Τύπου Κωδικού με Κατάσταση. Όταν καταλήξουμε να έχουμε δομή κληρονομικότητας τότε χρησιμοποιούμε την Αντικατάσταση Συνθήκης με Πολυμορφισμό.

Αν έχουμε μόνο λίγες περιπτώσεις στην συνθήκη switch και δεν αναμένουμε να αλλάξουν, τότε ο πολυμορφισμός είναι υπερβολή. Σε αυτή την περίπτωση η Αντικατάσταση Παραμέτρου με Μεθόδους (*SourceMaking, 2013, Replace Parameter with Explicit Methods*) είναι μία καλή επιλογή. Αν μία από τις συνθήκες είναι null, τότε μπορούμε να χρησιμοποιήσουμε την Εισαγωγή Αντικειμένου Null (*SourceMaking, 2013, Introduce Null Object*).

3.11 Parallel Inheritance Hierarchies (Παράλληλες Ιεραρχίες Κληρονομικότητας)

Οι παράλληλες ιεραρχίες κληρονομικότητες είναι μία ειδική περίπτωση Shotgun Surgery. Σε αυτή την περίπτωση, κάθε φορά που δημιουργούμε μία υποκλάση μίας κλάσης, πρέπει επίσης να δημιουργήσουμε μία υποκλάση κάποιας άλλης. Μπορούμε να αναγνωρίσουμε αυτή την οσμή επειδή το προθέματα των ονομάτων της μίας ιεραρχίας είναι ίδια με τα προθέματα της άλλης.

Η γενική στρατηγική που ακολουθείται για να εξαλείψουμε τις διπλοεγγραφές είναι να βεβαιωθούμε ότι τα στιγμιότυπα της μίας ιεραρχίας αναφέρονται σε στιγμιότυπα της άλλης κλάσης. Αν χρησιμοποιήσουμε την Μετακίνηση Μεθόδου και την Μετακίνηση Πεδίου η ιεραρχία στην αναφερόμενη κλάση εξαλείφεται.

3.12 Lazy Class (Βαρετή Κλάση)

Κάθε νέα κλάση που δημιουργούμε χρειάζεται χρόνο για να συντηρηθεί και να κατανοήσουμε. Μία κλάση η οποία δεν κάνει αρκετά για να δικαιολογεί τον χρόνο αυτό θα πρέπει αν αφαιρεθεί. Συχνά αυτή μπορεί να είναι μία κλάση η οποία είχε νόημα αρχικά αλλά έχει συρρικνωθεί μετά από refactoring. Ή ίσως να είναι μία κλάση η οποία δημιουργήθηκε επειδή είχαν προγραμματιστεί να γίνουν κάποιες αλλαγές αλλά ποτέ δεν έγιναν. Και στις δύο περιπτώσεις πρέπει να αφαιρέσουμε την κλάση. Αν έχουμε υποκλάσεις οι οποίες δεν κάνουν αρκετά τις εξαλείφουμε χρησιμοποιώντας την μέθοδο Σύμπτυξη Ιεραρχίας (*Source Making, 2013, Collapse Hierarchy*).

3.13 Speculative Generality

Αυτό το είδος οσμής προκύπτει όταν χρειαζόμαστε την ικανότητα ενός πράγματος να την χρησιμοποιήσουμε μελλοντικά και έτσι συντηρούμε κλάσεις και ειδικές περιπτώσεις τις οποίες δεν χρειαζόμαστε. Το αποτέλεσμα είναι ότι γίνεται αρκετά δύσκολο να καταλάβουμε και να συντηρήσουμε όλο αυτόν τον κώδικα. Αν όλη αυτή η λογική χρησιμοποιούνταν θα άξιζε τον κόπο. Αλλά αν δεν έχει καμία χρήση τότε δεν αξίζει.

Αν έχουμε αφαιρετικές κλάσεις (abstract classes) οι οποίες δεν προσφέρουν αρκετά, χρησιμοποιούμε την Σύμπτυξη Ιεραρχίας. Μέθοδοι με μη χρησιμοποιούμενες παραμέτρους θα πρέπει να υποβληθούν στην μέθοδο Αφαίρεση Παραμέτρων (*SourceMaking, 2013, Remove Parameter*). Μέθοδοι που έχουν περίεργα αφηρημένα ονόματα θα πρέπει να αλλάξουν με την χρήση της Μετονομασίας Μεθόδου (*SourceMaking, 2013, Rename Method*).

Η κερδοσκοπική γενικότητα μπορεί να προσδιοριστεί όταν οι μόνοι χρήστες της κλάσης ή της μεθόδου είναι τα τεστ. Αν βρούμε μία τέτοια μέθοδο ή κλάση την διαγράφουμε.

3.14 Temporary Field (Προσωρινό Πεδίο)

Κάποιες φορές βλέπουμε ένα αντικείμενο στο οποίο μία μεταβλητή λαμβάνει τιμή μόνο σε συγκεκριμένες περιπτώσεις. Τέτοιος κώδικας είναι αρκετά δύσκολος να κατανοηθεί, επειδή περιμένουμε από ένα αντικείμενο να χρειάζεται όλες τις μεταβλητές του. Είναι αρκετά περίεργο και δύσκολο να καταλάβουμε γιατί υπάρχει αυτή η μεταβλητή στο αντικείμενο ενώ δεν χρησιμοποιείται.

Χρησιμοποιούμε την μέθοδο Εξαγωγή Κλάσης για να βάλουμε όλες τις αυτές τις μεταβλητές οι οποίες σπάνια χρησιμοποιούνται σε μία νέα κλάση. Ίσως να χρειαστεί να χρησιμοποιήσουμε και την Εισαγωγή Αντικειμένου Null για να εξαλείψουμε ένα κομμάτι κώδικα από συνθήκες και να δημιουργήσουμε μία εναλλακτική κλάση σε περίπτωση που οι μεταβλητές δεν είναι έγκυρες.

Μία συχνή περίπτωση όπου προκύπτουν προσωρινά πεδία είναι όταν ένας περίπλοκος αλγόριθμος χρειάζεται πολλές μεταβλητές. Επειδή οι προγραμματιστές δεν θέλουν να περνάνε στην μέθοδο που υλοποιεί τον αλγόριθμο πολλές παραμέτρους, εισάγουν τις μεταβλητές αυτές ως πεδία. Αλλά τα πεδία είναι έγκυρα μόνο κατά την διάρκεια που εκτελείται ο αλγόριθμος. Σε κάθε άλλη περίπτωση απλά δημιουργούν σύγχυση. Και σε αυτή την περίπτωση μπορούμε να δημιουργήσουμε μία νέα κλάση για τα πεδία αυτά.

3.15 Message Chains (Αλυσίδες Μηνυμάτων)

Οι αλυσίδες μηνυμάτων δημιουργούνται όταν ένας ‘πελάτης’ ζητάει από μία κλάση πληροφορία για μία άλλη, η όποια με την σειρά της θα ρωτήσει κάποια άλλη και ούτω καθεξής. Αυτό συνήθως το αντικρίζουμε στον κώδικα ως μία μεγάλη γραμμή από κλήσεις σε μέθοδος Get ή ως μία σειρά τοπικών μεταβλητών. Η μεταφορά της πληροφορίας με αυτόν τον τρόπο σημαίνει πως ο ‘πελάτης’ θα πρέπει να γνωρίζει όλη την δομή και τον δρόμο που χρειάζεται για να πάρει την πληροφορία που θέλει. Αν κάποια ενδιάμεση σχέση μεταξύ των αντικειμένων αλλάξει τότε θα πρέπει να αλλάξει και ο τρόπος που ζητείται η πληροφορία.

Αυτό που χρειάζεται εδώ είναι η Απόκρυψη Αντιπροσώπου (βλέπε *Hide Delegate (Απόκρυψη Αντιπροσώπου)*). Μπορούμε να εφαρμόσουμε αυτή την μέθοδο σε διάφορα σημεία της αλυσίδας. Στην ουσία μπορούμε να το κάνουμε για κάθε αντικείμενο της αλυσίδας αλλά συνήθως δεν ενδείκνυται. Το καλύτερο είναι να δούμε ποια είναι η χρήση του επιστρεφόμενου αντικειμένου. Έπειτα χρησιμοποιούμε την Εξαγωγή Μεθόδου για να πάρουμε ένα κομμάτι από τον κώδικα που χρησιμοποιεί το επιστρεφόμενο αντικείμενο και μετά την Μετακίνηση Μεθόδου για να μικρύνουμε την αλυσίδα.

3.16 Middle Man (Ενδιάμεσος)

Ένα από τα κυριότερα χαρακτηριστικά των αντικειμένων είναι η ενθυλάκωση, η απόκρυψη δηλαδή των εσωτερικών γνωρισμάτων τους από τον υπόλοιπο κόσμο. Η ενθυλάκωση (encapsulation) συχνά εμφανίζεται μαζί με την αντιπροσώπευση (delegation). Ένα παράδειγμα της καθημερινής ζωής είναι όταν ζητάμε από κάποιον πότε θα είναι ελεύθερος για μία συνάντηση. Αυτός αναθέτει το μήνυμα στον αντιπρόσωπο που είναι η γραμματέας του, η οποία κρατά σημειώσεις των ραντεβού του και μετά μας δίνει την απάντηση.

Ωστόσο αυτό μπορεί να πάει πολύ μακριά. Καθώς κοιτάμε το interface μιας κλάσης διαπιστώνουμε ότι οι μισές από τις μεθόδους που περιέχει αντιπροσωπεύουν τις μεθόδους μίας άλλης κλάσης. Σε αυτή την περίπτωση πρέπει να αφαιρέσουμε τον ενδιάμεσο (βλέπε *Remove Middle Man (Αφαίρεση Ενδιάμεσου)*) και να ζητήσουμε την πληροφορία από αυτόν που πραγματικά την γνωρίζει. Αν μόνο λίγες μέθοδοι παίζουν τον ρόλο του ενδιάμεσου χωρίς μεγάλη λειτουργικότητα χρησιμοποιούμε την μέθοδο Ενσωμάτωσης Μεθόδου (*SourceMaking, 2013, Inline Method*) για να τις ενσωματώσουμε μέσα

στην κλάση που τις καλεί. Αν υπάρχει επιπλέον λειτουργικότητα στις μεθόδους χρησιμοποιούμε την Αντικατάσταση Αντιπροσώπου με Κληρονομικότητα (*SourceMaking, 2013, Replace Delegation with Inheritance*) για να μετατρέψουμε τον ενδιάμεσο σε μία υποκλάση του πραγματικού αντικειμένου.

3.17 Inappropriate Intimacy (Ακατάλληλη Οικειότητα)

Κάποιες φορές οι κλάσεις γίνονται πάρα πολύ οικείες και ξοδεύουν πολύ χρόνο με το να ασχολούνται η μία με τα προσωπικά δεδομένα της άλλης.

Οι κλάσεις οι οποίες είναι πολύ οικείες μεταξύ τους πρέπει να διαχωριστούν. Με την χρήση των μεθόδων Μετακίνηση Μεθόδου και Μετακίνηση Πεδίου μπορούμε να τις διαχωρίσουμε και να μειώσουμε την οικειότητα. Επίσης μπορούμε να δούμε αν μπορούμε να χρησιμοποιήσουμε την μέθοδο Αλλαγής Αμφίδρομης Συσχέτισης σε Μονόδρομη (*SourceMaking, 2013, Change Bidirectional Association to Unidirectional*). Αν οι κλάσεις έχουν όντως κοινά ενδιαφέροντα τότε πρέπει να κάνουμε Εξαγωγή Κλάσης και να βάλουμε τα κοινά τους σημεία σε μία νέα κλάση.

Η κληρονομικότητα συχνά μπορεί να οδηγήσει σε οικειότητα. Οι υποκλάσεις συνήθως θέλουν να γνωρίζουν περισσότερα σχετικά με τις υπερκλάσεις από ότι αυτές θα ήθελαν.

3.18 Incomplete Library Class (Ελλιπής Βιβλιοθήκη Κλάσεων)

Η επαναχρησιμοποίηση αναφέρεται συχνά ως το προτέρημα των αντικειμένων. Ίσως να θεωρούμε ότι η επαναχρησιμοποίηση είναι υπερεκτιμημένη. Ωστόσο, δεν μπορούμε να αρνηθούμε ότι οι προγραμματιστικές μας ικανότητες βασίζονται σε ένα μεγάλο βαθμό σε έτοιμες βιβλιοθήκες.

Όσοι φτιάχνουν όμως τις βιβλιοθήκες δεν είναι παντογνώστες. Το πρόβλημα είναι ότι συνήθως οι βιβλιοθήκες έχουν άσχημη μορφή και είναι πολύ δύσκολο έως αδύνατον να τροποποιήσουμε μία βιβλιοθήκη για να κάνει αυτό που θα θέλαμε. Αυτό σημαίνει πως οι απλές μέθοδοι refactoring όπως η Μετακίνηση Μεθόδου δεν μπορούν να μας βοηθήσουν.

Παρόλα αυτά έχουμε υπάρχουν και πάλι κάποιες τεχνικές που μπορούν να μας βοηθήσουν. Αν θέλουμε απλά η βιβλιοθήκη να περιείχε λίγες ακόμα μεθόδους μπορούμε να χρησιμοποιήσουμε την Εισαγωγή Ξένης Μεθόδου (βλέπε *Introduce Foreign Method (Εισαγωγή Ξένης Μεθόδου)*). Αν θέλουμε η βιβλιοθήκη να κάνει πολλά περισσότερα από την τωρινή της συμπεριφορά χρησιμοποιούμε την μέθοδο Εισαγωγή Τοπικής Επέκτασης (βλέπε *Introduce Local Extension (Εισαγωγή Τοπικής Επέκτασης)*).

3.19 Data Class (Κλάση Δεδομένων)

Υπάρχουν κλάσεις οι οποίες περιέχουν πεδία, μεθόδους Get/Set για τα πεδία και τίποτα άλλο. Αυτές οι κλάσεις κρατούν απλά δεδομένα και στην ουσία τα δεδομένα τους διαχειρίζονται με περισσότερη λεπτομέρεια από άλλες κλάσεις. Στο αρχικό τους στάδιο αυτές οι κλάσεις περιέχουν μόνο δημόσια (public) πεδία. Αν ισχύει αυτό τότε πρέπει να αμέσως να τα ενθυλακώσουμε χρησιμοποιώντας της μέθοδο Ενθυλάκωσης Πεδίου (*SourceMaking, 2013, Encapsulate Field*). Εάν έχουμε μία συλλογή από πεδία πρέπει να δούμε αν ήδη ενθυλακωμένα αλλιώς κάνουμε χρήση της Ενθυλάκωσης Συλλογής (*SourceMaking, 2013, Encapsulate Collection*). Επίσης πρέπει να αφαιρέσουμε την μέθοδο Set από κάθε πεδίο το οποίο δεν πρέπει να αλλάζει.

Κοιτάζουμε πότε οι μέθοδοι get και set χρησιμοποιούνται από κάποια άλλη κλάση. Με την Μετακίνηση Μεθόδου μετακινούμε την συμπεριφορά από την εξωτερική κλάση μέσα στην κλάση δεδομένων. Αν δεν μπορούμε να μετακινήσουμε μία ολόκληρη μέθοδο τότε κάνουμε Εξαγωγή Μεθόδου για να δημιουργήσουμε μία μέθοδο που μπορεί να μετακινηθεί. Έπειτα μπορούμε να χρησιμοποιήσουμε την μέθοδο Απόκρυψης Μεθόδου (*SourceMaking, 2013, Hide Method*) μέσα στις μεθόδους set και get.

4 Σύνθεση Μεθόδων

Το μεγαλύτερο μέρος του refactoring είναι η σύνθεση μεθόδων για να ομαδοποιήσουμε τον κώδικα. Συνήθως τα περισσότερα προβλήματα προκύπτουν από μεθόδους που είναι πάρα πολύ μεγάλες. Οι μεγάλες μέθοδοι δημιουργούν μπελάδες επειδή συχνά περιέχουν πολλές πληροφορίες, οι οποίες είναι θαμμένες μέσα στην πολυπλοκότητα τους. Το κλειδί για τις μεγάλες μεθόδους είναι η Εξαγωγή Μεθόδου η οποία παίρνει ένα μεγάλο κομμάτι κώδικα και το μετατρέπει σε μία νέα μέθοδο. Η Ενσωμάτωση Μεθόδου είναι ακριβώς το αντίθετο. Παίρνουμε την κλήση μίας μεθόδου και την αντικαθιστούμε με το σώμα της μεθόδου. Χρησιμοποιούμε την Ενσωμάτωση Μεθόδου όταν έχουμε κάνει πολλές εξαγωγές και αντιλαμβανόμαστε ότι οι μέθοδοι που προέκυψαν δεν χρειάζονται πλέον ή ότι ο τρόπος που έγινε η εξαγωγή δεν ήταν σωστός.

Το μεγαλύτερο πρόβλημα με την Εξαγωγή Μεθόδου είναι αντιμετώπιση των τοπικών και προσωρινών μεταβλητών. Όταν εργαζόμαστε πάνω σε μία μέθοδο χρησιμοποιούμε την μέθοδο Αντικατάσταση Προσωρινής Μεταβλητής για να ξεφορτωθούμε όσες προσωρινές μεταβλητές μπορούμε. Αν μία από τις μεταβλητές αυτές χρησιμοποιείται για διάφορα πράγματα σε διάφορα σημεία χρησιμοποιούμε την Διάσπαση Προσωρινής Μεταβλητής (βλέπε *Split Temporary Variable (Διάσπαση Προσωρινής Μεταβλητής)*) αρχικά ώστε να είναι εύκολη η αφαίρεσή της.

Κάποιες φορές, ωστόσο, οι προσωρινές μεταβλητές είναι αρκετά μπερδεμένες για να αφαιρεθούν. Τότε χρειαζόμαστε την Αντικατάσταση Μεθόδου με Μέθοδο Αντικειμένου. Αυτό μας επιτρέπει να σπάσουμε ακόμα και την πιο πολύπλοκη μέθοδο, με το κόστος της εισαγωγής μίας νέας κλάσης.

Οι παράμετροι αποτελούν μικρότερο πρόβλημα από τις προσωρινές μεταβλητές, γιατί απλά παρέχονται στην μέθοδο και δεν γίνεται ανάθεση τιμής σε αυτές. Αν αναθέτουμε τιμή στις παραμέτρους μίας μεθόδου πρέπει να χρησιμοποιήσουμε την Αφαίρεση Ανάθεσης σε Παράμετρο.

Όταν η μέθοδος έχει διασπαστεί, μπορούμε να καταλάβουμε καλύτερα πως δουλεύει. Μπορούμε επίσης να βελτιστοποιήσουμε τον αλγόριθμο της μεθόδου για να τον κάνουμε πιο καθαρό. Τότε χρησιμοποιούμε την Αντικατάσταση Αλγορίθμου για να εισάγουμε έναν πιο καλό αλγόριθμο.

4.1 Extract Method (Εξαγωγή Μεθόδου)

Χρησιμοποιούμε αυτή την τεχνική όταν έχουμε ένα κομμάτι κώδικα που μπορεί να ομαδοποιηθεί.

Μετατρέπουμε το κομμάτι κώδικα σε μία νέα μέθοδο της οποίας το όνομα εξηγεί τον σκοπό της μεθόδου.

```
void printOwing(double amount) {
    printBanner();
    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

```
void printOwing(double amount) {
    printBanner();
    printDetails(amount);
}

void printDetails (double amount) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + amount);
}
```

4.1.1 Κίνητρο

Η Εξαγωγή Μεθόδου είναι η πιο κοινή τεχνική refactoring. Απλά βλέπουμε ότι μία μέθοδος είναι μεγάλη και κοιτώντας τον κώδικά της πιστεύουμε ότι πρέπει να προσθέσουμε σχόλιο για να καταλάβουμε τι κάνει. Τότε παίρνουμε αυτό το κομμάτι κώδικα και το βάζουμε στην δική του μέθοδο.

Είναι καλό να προτιμούμε να έχουμε μικρές μεθόδους με καλά ονόματα για διάφορους λόγους. Αρχικά, γιατί αυξάνονται οι πιθανότητες ότι και άλλες μέθοδοι θα θέλουν να χρησιμοποιήσουν αυτές τις μεθόδους όταν τελικά δημιουργηθούν. Δεύτερον, μας επιτρέπει στις μεθόδους ανώτερου επιπέδου να καταλαβαίνουμε τι κάνουν γιατί οι κλήσεις των μικρότερων μεθόδων θα λειτουργούν ως σχόλια.

Χρειάζεται λίγο χρόνο να συνηθίσουμε στις μικρές μεθόδους όταν είμαστε συνηθισμένοι να χρησιμοποιούμε μεγάλες. Οι μικρές μέθοδοι δουλεύουν πραγματικά όταν έχουν σωστά ονόματα, όποτε χρειάζεται να δίνουμε μεγάλη βαρύτητα στην ονοματοδοσία τους. Το κλειδί είναι η σημασιολογική διαφορά ανάμεσα στα ονόματα των μεθόδων και το σώμα τους. Αν η εξαγωγή βελτιώνει την σαφήνεια τότε είναι καλό να την κάνουμε, ακόμα και αν το όνομα της νέας μεθόδου είναι μεγαλύτερο από τον κώδικα που αντικαθιστά.

4.1.2 Μηχανισμοί

- Δημιουργούμε μία νέα μέθοδο και την ονομάζουμε ανάλογα με την πρόθεσή της (την ονομάζουμε με βάση το τι κάνει και όχι πως το κάνει).
- Αντιγράφουμε τον εξαγόμενο κώδικα από τον αρχικό κώδικα στην καινούρια μέθοδο.

- Ψάχνουμε τον εξαγόμενο κώδικα για τυχόν αναφορές σε προσωρινές μεταβλητές που είναι τοπικά ορισμένες στην αρχική μέθοδο. Αυτές θα αποτελέσουν τοπικές μεταβλητές και παραμέτρους της νέας μεθόδου.
- Κοιτάζουμε αν υπάρχουν προσωρινές μεταβλητές που χρησιμοποιούνται μόνο μέσα στον εξαγόμενο κώδικα. Αν ναι, τότε τις ορίζουμε στην νέα μέθοδο ως προσωρινές μεταβλητές.
- Κοιτάζουμε αν υπάρχουν προσωρινές μεταβλητές της αρχικής μεθόδου που να τροποποιείται η τιμή τους από τον εξαγόμενο κώδικα. Αν μία μόνο μεταβλητή τροποποιείται τότε ελέγχουμε αν μπορούμε να επιστρέφουμε την τιμή της από την νέα μέθοδο. Αν αυτό δεν είναι εύκολο ή αν υπάρχουν περισσότερες από μία μεταβλητές που τροποποιούνται, δεν μπορούμε να εξάγουμε την μέθοδο ως έχει. Ίσως χρειαστεί να χρησιμοποιήσουμε την τεχνική Διάσπασης Τοπικής Μεταβλητής και να ξαναπροσπαθήσουμε. Μπορούμε επίσης να εξαλείψουμε τις προσωρινές μεταβλητές χρησιμοποιώντας την Αντικατάσταση Προσωρινής Μεταβλητής.
- Περνάμε στην νέα μέθοδο ως παραμέτρους όλες τις τοπικές μεταβλητές της αρχικής μεθόδου που χρησιμοποιούνται από τον εξαγόμενο κώδικα.
- Όταν τελειώσουμε με τις τοπικές μεταβλητές κάνουμε compile τον κώδικα.
- Αντικαθιστούμε τον εξαγόμενο κώδικα στην αρχική μέθοδο με μία κλήση στην νέα μέθοδο.
- Κάνουνε ξανά compile και τρέχουμε τα τεστ.

4.1.3 Παράδειγμα: Χωρίς Τοπικές Μεταβλητές

Σε αυτό το παράδειγμα η Εξαγωγή Μεθόδου γίνεται αρκετά εύκολα. Ας υποθέσουμε ότι έχουμε την ακόλουθη μέθοδο:

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Είναι αρκετά εύκολο να εξάγουμε τον κώδικα που εκτυπώνει το banner. Απλά κάνουμε αντιγραφή/επικόλληση του κώδικα που μας ενδιαφέρει σε μία νέα μέθοδο και αλλάζουμε τον κώδικα στην παλιά με μία κλήση στην καινούρια.

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}

void printBanner() {
    // print banner
    System.out.println ("**************************");
    System.out.println ("***** Customer Owes ******");
    System.out.println ("**************************");
}
```

4.1.4 Παράδειγμα: Με χρήση Τοπικών Μεταβλητών

Οι παράμετροι της αρχικής μεθόδου και οι μεταβλητές που είναι ορισμένες σε αυτή αποτελούν τις τοπικές μεταβλητές. Οι μεταβλητές αυτές είναι προσβάσιμες μόνο μέσα από αυτή την μέθοδο. Οπότε όταν χρησιμοποιούμε την Εξαγωγή Μεθόδου, οι μεταβλητές αυτές απαιτούν επιπλέον δουλειά. Σε κάποιες περιπτώσεις είναι ικανές ακόμα και να μας αποτρέψουν από το να κάνουμε το refactoring.

Η ευκολότερη περίπτωση τοπικών μεταβλητών είναι όταν αυτές απλά διαβάζονται αλλά δεν αλλάζουν. Σε αυτή την περίπτωση τις περνάμε απλά σαν παραμέτρους.

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    //print details
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Μπορούμε να εξάγουμε τον κώδικα που εκτυπώνει τις λεπτομέρειες σε μία νέα μέθοδο με μία παράμετρο.

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}

void printDetails (double outstanding) {
    System.out.println ("name:" + _name);
    System.out.println ("amount" + outstanding);
}
```

Μπορούμε να χρησιμοποιήσουμε αυτό τον τρόπο για όσες παραμέτρους θέλουμε.

4.1.5 Παράδειγμα: Επαναθέτοντας Τοπική Μεταβλητή

Όταν έχουμε ανάθεση τιμών σε τοπικές μεταβλητές η εξαγωγή γίνεται πιο πολύπλοκη. Σε αυτή την περίπτωση αναφερόμαστε μόνο σε προσωρινές μεταβλητές. Αν δούμε ότι γίνεται ανάθεση τιμής σε μία παράμετρο πρέπει να χρησιμοποιήσουμε την μέθοδο Αφαίρεση Ανάθεσης σε Παράμετρο (βλέπε *Remove Assignments to Parameters (Αφαίρεση Ανάθεσης σε Παράμετρο*12).

Όταν η ανάθεση γίνεται σε προσωρινές μεταβλητές έχουμε δύο περιπτώσεις. Η απλούστερη είναι όταν η μεταβλητή στην οποία γίνεται η ανάθεση είναι δηλωμένη μόνο μέσα στο κομμάτι κώδικα που γίνεται η εξαγωγή. Όταν συμβαίνει αυτό, απλά μεταφέρουμε την μεταβλητή στην εξαγόμενη μέθοδο. Η άλλη περίπτωση είναι όταν γίνεται χρήση της μεταβλητής και έξω από τον εξαγόμενο κώδικα. Αν η

μεταβλητή δεν χρησιμοποιείται μετά τον εξαγόμενο κώδικα τότε μπορούμε να κάνουμε την αλλαγή στην τιμή της μόνο μέσα στην νέα μέθοδο. Αν όμως χρησιμοποιείται και σε επόμενο τμήμα του κώδικα της αρχικής μεθόδου, τότε πρέπει να επιστρέφουμε την τιμή της. Μπορούμε να το καταλάβουμε καλύτερα με το επόμενο παράδειγμα.

```
void printOwing() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Τώρα εξάγουμε το κομμάτι του υπολογισμού.

```
void printOwing() {
    printBanner();
    double outstanding = getOutstanding();
    printDetails(outstanding);
}

double getOutstanding() {
    Enumeration e = _orders.elements();
    double outstanding = 0.0;
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }
    return outstanding;
}
```

Η μεταβλητή απαρίθμησης (enumeration) χρησιμοποιείται μόνο μέσα στον εξαγόμενο κώδικα, άρα μπορούμε να μετακινήσουμε όλο το κομμάτι μέσα στην νέα μέθοδο. Η μεταβλητή `outstanding` χρησιμοποιείται σε δύο σημεία, οπότε πρέπει να την δηλώσουμε ξανά μέσα στην νέα μέθοδο.

Στο επόμενο παράδειγμα η μεταβλητή `outstanding` αρχικοποιείται σε μία συγκεκριμένη τιμή. Αν κάτι πιο πολύπλοκο συμβαίνει σε μία μεταβλητή τότε πρέπει να την περάσουμε ως παράμετρο στην νέα μέθοδο. Με αυτόν τον τρόπο θα ξέρουμε ότι αν η αρχική τιμή της `outstanding` αλλάξει, η αλλαγή θα γίνει σε μόνο ένα σημείο.

```
void printOwing(double previousAmount) {
    Enumeration e = _orders.elements();
    double outstanding = previousAmount * 1.2;

    printBanner();

    // calculate outstanding
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        outstanding += each.getAmount();
    }

    printDetails(outstanding);
}
```

Σε αυτή την περίπτωση η εξαγωγή έχει ως εξής.

```
void printOwing(double previousAmount) {
    double outstanding = previousAmount * 1.2;
    printBanner();
    outstanding = getOutstanding(outstanding);
    printDetails(outstanding);
}

double getOutstanding(double initialValue) {
    double result = initialValue;
    Enumeration e = _orders.elements();
    while (e.hasMoreElements()) {
        Order each = (Order) e.nextElement();
        result += each.getAmount();
    }
    return result;
}
```

Αφού δούμε ότι κώδικάς μας λειτουργεί μπορούμε να κάνουμε ακόμα μία βελτίωση στον τρόπο που αρχικοποιούμε την μεταβλητή.

```
void printOwing(double previousAmount) {
    printBanner();
    double outstanding = getOutstanding(previousAmount * 1.2);
    printDetails(outstanding);
}
```

Όταν όμως έχουμε περισσότερες από μία μεταβλητές που πρέπει να επιστραφούν τα πράγματα είναι πιο δύσκολα. Η καλύτερη επιλογή είναι να επιλέξουμε διαφορετικά μικρά κομμάτια κώδικα για εξαγωγή.

4.2 Inline Temp (Ενσωμάτωση Προσωρινής Μεταβλητής)

Χρησιμοποιούμε αυτή την τεχνική όταν έχουμε μία προσωρινή μεταβλητή στην οποία γίνεται ανάθεση μόνο μία φορά, και η μεταβλητή αυτή μας εμποδίζει στο γενικότερο refactoring.

Αντικαθιστούμε όλες τις αναφορές σε αυτή την μεταβλητή με την έκφραση ανάθεσης της τιμής της.

```
double basePrice = anOrder.basePrice();
return (basePrice > 1000)
```

```
return (anOrder.basePrice() > 1000)
```

4.2.1 Κίνητρο

Τις περισσότερες φορές αυτή η τεχνική χρησιμοποιείται ως μέρος της Αντικατάστασης Προσωρινής Μεταβλητής. Η μόνη φορά που χρησιμοποιείται από μόνη της είναι όταν έχουμε μία προσωρινή μεταβλητή στην οποία ανατίθεται η τιμή της κλήσης μίας μεθόδου. Συνήθως η μεταβλητή δεν έχει κάποιο αντίκτυπο οπότε μπορούμε να την αφήσουμε όπως είναι. Αν όμως δημιουργεί εμπόδιο όταν κάνουμε refactoring πρέπει να την αφαιρέσουμε.

4.2.2 Μηχανισμοί

- Δηλώνουμε την μεταβλητή ως final, αν δεν είναι ήδη. Κάνουμε compile τον κώδικα για να βεβαιωθούμε ότι όντως γίνεται ανάθεση τιμής σε αυτή μόνο μία φορά.
- Βρίσκουμε όλες τις αναφορές τις και αντικαθιστούμε τις με την δεξιό μέρος της ανάθεσής της.
- Αφαιρούμε την δήλωση της μεταβλητής.
- Κάνουμε compile και τεστάρουμε τον κώδικα.

4.3 Replace Temp with Query (Αντικατάσταση Προσωρινής Μεταβλητής με Μέθοδο)

Χρησιμοποιούμε αυτή την τεχνική όταν έχουμε μία προσωρινή μεταβλητή για να κρατάμε το αποτέλεσμα μίας έκφρασης.

Εξάγουμε την έκφραση σε μία μέθοδο. Αντικαθιστούμε όλες τις αναφορές της μεταβλητής με μία κλήση στην νέα μέθοδο. Μπορούμε έπειτα να καλέσουμε την νέα μέθοδο και από άλλες μεθόδους.

```
double basePrice = _quantity * _itemPrice;
if (basePrice > 1000)
    return basePrice * 0.95;
else
    return basePrice * 0.98;
```

```
if (basePrice() > 1000)
    return basePrice() * 0.95;
else
    return basePrice() * 0.98;

...

double basePrice() {
    return _quantity * _itemPrice;
}
```

4.3.1 Κίνητρο

Το πρόβλημα με αυτές τις μεταβλητές είναι το ότι είναι προσωρινές και τοπικές. Επειδή υπάρχουν μόνο μέσα στα όρια μίας μόνο μεθόδου τείνουν να δημιουργούν μεγαλύτερες μεθόδους, γιατί αυτός είναι ο μόνος τρόπος να έχουμε πρόσβαση σε αυτές. Αντικαθιστώντας μια τέτοια μεταβλητή με μία μέθοδο, οποιαδήποτε μέθοδο στην κλάση μπορεί να την καλέσει για να πάρει την πληροφορία που χρειάζεται. Αυτό βοηθά αρκετά να έχουμε πιο καθαρό κώδικα μέσα στην κλάση.

Η Αντικατάσταση Τοπικής Μεταβλητής είναι ένα συνηθισμένο βήμα πριν της Εξαγωγή Μεθόδου. Οι τοπικές μεταβλητές κάνουν δύσκολη την εξαγωγή, οπότε τις αντικαθιστούμε με μεθόδους.

Οι απλές περιπτώσεις αυτής της τεχνικής είναι όταν η ανάθεση σε αυτές γίνεται μόνο μία φορά και όταν η έκφραση ανάθεσής τους δεν έχει παράπλευρες επιρροές.

4.3.2 Μηχανισμοί

- Ψάχνουμε για προσωρινές μεταβλητές στις οποίες γίνεται ανάθεση μόνο μία φορά. Αν γίνεται ανάθεση περισσότερες από μία φορές τότε χρησιμοποιούμε την Διάσπαση Προσωρινής Μεταβλητής (βλέπε *Split Temporary Variable (Διάσπαση Προσωρινής Μεταβλητής)*).
- Δηλώνουμε την μεταβλητή ως final.
- Κάνουμε compile για να βεβαιωθούμε ότι όντως γίνεται ανάθεση τιμής σε αυτή μόνο μία φορά.
- Εξάγουμε το δεξιό μέρος της ανάθεσης σε μία νέα μέθοδο.
    - Αρχικά δηλώνουμε την μέθοδο ως private.
    - Βεβαιωνόμαστε ότι η νέα μέθοδος δεν δημιουργεί εξωτερικές αλλαγές, δηλαδή δεν επηρεάζει κανένα αντικείμενο έξω από αυτή.
- Κάνουμε ξανά compile και τεστάρουμε τον κώδικα.
- Χρησιμοποιούμε την Αντικατάσταση Προσωρινής Μεταβλητής με Μέθοδο στην μεταβλητή.

Οι προσωρινές μεταβλητές συχνά χρησιμοποιούνται για να αποθηκεύουν αθροιστική πληροφορία μέσα σε βρόχους επανάληψης. Ολόκληρος ο βρόχος μπορεί να εξαχθεί σε μία μέθοδο, αυτό αφαιρεί από την αρχική μέθοδο αρκετές γραμμές ‘ανόητου’ κώδικα. Κάποιες φορές ένας βρόχος επανάληψης μπορεί να χρησιμοποιείται για να αθροίσει πολλές τιμές. Σε αυτή την περίπτωση, δημιουργούμε ένα αντίγραφο του βρόχου για κάθε μία από τις μεταβλητές. Ο βρόχος θα πρέπει να είναι αρκετά απλός ώστε να μην υπάρχει πρόβλημα να τον αντιγράψουμε.

Ίσως αναρωτηθούμε τι γίνεται σχετικά με την απόδοση σε αυτή την περίπτωση. Προς το παρόν η απόδοση δεν παίζει σημαντικό ρόλο. Αν ένας βρόχος εκτελεστεί δύο ή τρεις φορές περισσότερες δεν θα επηρεάσει αισθητά την απόδοση του προγράμματος.

4.3.3 Παράδειγμα

```
double getPrice() {
    int basePrice = _quantity * _itemPrice;
    double discountFactor;

    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;

    return basePrice * discountFactor;
}
```

Στο παράδειγμά μας έχουμε δύο προσωρινές μεταβλητές. Αν και στην περίπτωσή μας είναι αρκετά εύκολο να τις αντικαταστήσουμε και τις δύο μαζί, είναι προτιμότερο να το κάνουμε για την καθεμία ξεχωριστά.

Πρέπει να ελέγξουμε ότι γίνεται ανάθεση σε αυτές μόνο μία φορά, αν και είναι προφανές. Οπότε τις δηλώνουμε ως final.

```
double getPrice() {
    final int basePrice = _quantity * _itemPrice;
    final double discountFactor;

    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;

    return basePrice * discountFactor;
}
```

Κάνοντας compile τον κώδικα, ο compiler θα μας ειδοποιήσει για το αν γίνεται ανάθεση στις μεταβλητές και άλλη φορά. Κάνουμε την αντικατάσταση σε κάθε μεταβλητή ξεχωριστά. Εξάγουμε αρχικά το δεξιό μέρος της ανάθεσης.

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;

    if (basePrice > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;

    return basePrice * discountFactor;
}

private int basePrice() {
    return _quantity * _itemPrice;
}
```

Αφού κάνουμε ξανά compile και τεστάρουμε τον κώδικα, ξεκινάμε την Αντικατάσταση Προσωρινής Μεταβλητής με Μέθοδο. Αντικαθιστούμε τις αναφορές στην πρώτη μεταβλητή με μία κλήση στην μέθοδο.

```
double getPrice() {
    final int basePrice = basePrice();
    final double discountFactor;

    if (basePrice() > 1000)
        discountFactor = 0.95;
    else
        discountFactor = 0.98;

    return basePrice() * discountFactor;
}
```

Μόλις τελειώσουμε με την πρώτη, κάνουμε το ίδιο και για την δεύτερη και παράλληλα διαγράφουμε την δήλωση της πρώτης.

```
double getPrice() {
    final double discountFactor = discountFactor();

    return basePrice() * discountFactor;
}

private double discountFactor() {
    if (basePrice() > 1000)
        return 0.95;
    else
        return 0.98;
}
```

Βλέπουμε ότι θα ήταν δύσκολο να εξάγουμε την `discountFactor` αν δεν είχαμε αντικαταστήσει την `basePrice` με μέθοδο. Η μέθοδος `getPrice` καταλήγει ως εξής.

```
double getPrice() {
    return basePrice() * discountFactor();
}
```

4.4 Introduce Explaining Variable (Εισαγωγή Βοηθητικής Μεταβλητής)

Χρησιμοποιούμε αυτή την τεχνική όταν έχουμε μία περίπλοκη έκφραση.

> *Βάζουμε το αποτέλεσμα της έκφρασης, ή μέρος της έκφρασης, σε μία προσωρινή μεταβλητή με ένα όνομα το οποίο εξηγεί τον σκοπό της.*

```
if ((platform.toUpperCase().indexOf("MAC") > -1) &&
    (browser.toUpperCase().indexOf("IE") > -1) &&
     wasInitialized() && resize > 0 )
{
        // do something
}
```

```
final boolean isMacOs = platform.toUpperCase().indexOf("MAC") > -1;
final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;
final boolean wasResized = resize > 0;

if (isMacOs && isIEBrowser && wasInitialized() && wasResized) {
    // do something
}
```

4.4.1 Κίνητρο

Οι εκφράσεις μπορεί να γίνουν αρκετά πολύπλοκες και δύσκολες στην ανάγνωση και την κατανόησή τους. Σε τέτοιες περιπτώσεις οι προσωρινές μεταβλητές είναι αρκετά χρήσιμες για να σπάσουν την έκφραση σε κάτι πιο απλό.

Η Εισαγωγή Βοηθητικής Μεταβλητής είναι ιδιαίτερα χρήσιμη σε συνθήκες επιλογής όπου χρησιμοποιείται για να αντικαταστήσει κάθε μέρος της συνθήκης και να εξηγήσει τι εκφράζει κάθε μία από αυτές με μία μεταβλητή.

Η Εισαγωγή Βοηθητικής Μεταβλητής είναι πολύ κοινή τεχνική refactoring, αλλά συνήθως συνίσταται να μην γίνεται υπερβολική χρήση της. Είναι προτιμότερο να χρησιμοποιούμε την Εξαγωγή Μεθόδου όπου μπορούμε. Αυτό γιατί η προσωρινή μεταβλητή είναι χρήσιμη μόνο μέσα στα πλαίσια της μεθόδου της. Μία μέθοδος αντίθετα είναι ορατή μέσα στην κλάση και σε άλλες κλάσεις. Υπάρχουν φορές ωστόσο, που τοπικές μεταβλητές είναι δύσκολο να εξαχθούν. Τότε χρησιμοποιούμε την Εισαγωγή Βοηθητικής Μεταβλητής.

4.4.2 Μηχανισμοί

- Δηλώνουμε μία προσωρινή final μεταβλητή, και αναθέτουμε σε αυτή το αποτέλεσμα μίας πολύπλοκης έκφρασης.
- Αντικαθιστούμε την έκφραση με την μεταβλητή.
    - Αν η έκφραση επαναλαμβάνεται πολλές φορές αντικαθιστούμε τις αναφορές τις μία κάθε φορά.
- Κάνουμε compile και τεστάρουμε τον κώδικα.
- Επαναλαμβάνουμε για άλλες πολύπλοκες εκφράσεις.

4.4.3 Παράδειγμα

Ξεκινάμε με ένα απλό παράδειγμα.

```
double price() {
    // price is base price - quantity discount + shipping
    return _quantity * _itemPrice -
           Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
           Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Πρώτα προσδιορίζουμε την βασική τιμή (base price) ως το αποτελέσματα του πολλαπλασιασμού της ποσότητας (quantity) ενός τεμαχίου επί την τιμή του τεμαχίου (item price).

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;

    return basePrice -
           Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
           Math.min(_quantity * _itemPrice * 0.1, 100.0);
}
```

Η ίδια έκφραση χρησιμοποιείται και παρακάτω, άρα μπορούμε να την αντικαταστήσουμε και αυτή με την προσωρινή μεταβλητή.

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;

    return basePrice -
           Math.max(0, _quantity - 500) * _itemPrice * 0.05 +
           Math.min(basePrice * 0.1, 100.0);
}
```

Τώρα κάνουμε το ίδιο και για την ποσότητα έκπτωσης (quantity discount).

```
double price() {
    // price is base price - quantity discount + shipping
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount = Math.max(0, _quantity - 500) *
                                    _itemPrice * 0.05;

    return basePrice – quantityDiscount +
           Math.min(basePrice * 0.1, 100.0);
}
```

Στο τέλος κάνουμε το ίδιο και για τα έξοδα αποστολής. Τώρα μπορούμε να διαγράψουμε και το σχόλιο στην αρχή, γιατί πλέον δεν εξηγεί κάτι παραπάνω από αυτό που κάνει ο κώδικας.

```
double price() {
    final double basePrice = _quantity * _itemPrice;
    final double quantityDiscount = Math.max(0, _quantity - 500) *
                                    _itemPrice * 0.05;
    final double shipping = Math.min(basePrice * 0.1, 100.0);
    return basePrice – quantityDiscount + shipping;
}
```

4.4.4 Παράδειγμα με Εξαγωγή Μεθόδου

Για το παράδειγμα που αναφέραμε είναι προτιμότερο να χρησιμοποιήσουμε την Εξαγωγή Μεθόδου ώστε οι εκφράσεις που αναθέτονται στις μεταβλητές να είναι ορατές και έξω από το όρια της μεθόδου όπου δηλώνονται.

Άρα το αρχικό παράδειγμα θα ήταν καλύτερα κάπως έτσι.

```
double price() {
    return basePrice() - quantityDiscount() + shipping();
}

private double quantityDiscount() {
    return Math.max(0, _quantity - 500) * _itemPrice * 0.05;
}

private double shipping() {
    return Math.min(basePrice() * 0.1, 100.0);
}

private double basePrice() {
    return _quantity * _itemPrice;
}
```

4.5 Split Temporary Variable (Διάσπαση Προσωρινής Μεταβλητής)

Χρησιμοποιούμε αυτή την τεχνική όταν έχουμε μία προσωρινή μεταβλητή στην οποία αναθέτουμε τιμή περισσότερες από μία φορές, και δεν είναι αθροιστική μεταβλητή ή μεταβλητή σε βρόχο επανάληψης.

Δημιουργούμε μία ξεχωριστή μεταβλητή για κάθε ανάθεση.

```
double temp = 2 * (_height + _width);
System.out.println (temp);
temp = _height * _width;
System.out.println (temp);
```

```
final double perimeter = 2 * (_height + _width);
System.out.println (perimeter);
final double area = _height * _width;
System.out.println (area);
```

4.5.1 Κίνητρο

Οι προσωρινές μεταβλητές δηλώνονται για διάφορες χρήσεις. Κάποιες από αυτές τις χρήσεις αναγκαστικά οδηγούν στην ανάθεση τιμής τους πολλές φορές. Τέτοιες μεταβλητές είναι οι μεταβλητές που δηλώνονται σε ένα βρόχο επανάληψης, όπως η `i` μέσα στον βρόχο `for (int i=0; i<10; i++)`, και αυτές που χρησιμοποιούνται για να συλλέξουν αθροιστικά δεδομένα.

Πολλές άλλες προσωρινές μεταβλητές όμως χρησιμοποιούνται για να κρατήσουν το αποτέλεσμα ενός μακροσκελούς κομματιού κώδικα. Σε αυτές οι μεταβλητές θα πρέπει να ανατίθεται τιμή μόνο μία φορά. Αν γίνεται ανάθεση περισσότερες από μία φορές τότε αυτό είναι σημάδι ότι έχουμε παραπάνω από μία αρμοδιότητες μέσα στην μέθοδο. Κάθε μεταβλητή η οποία έχει πολλές υπευθυνότητες θα πρέπει να αντικατασταθεί με μία μεταβλητή για κάθε μία από αυτές.

4.5.2 Μηχανισμοί

- Αλλάζουμε το όνομα της μεταβλητής στην δήλωση της και στην πρώτη ανάθεση τιμής σε αυτή.
    - Αν οι επόμενες αναθέσεις είναι της μορφής *i = i + έκφραση*, το οποίο δηλώνει ότι είναι μία αθροιστική μεταβλητή, δεν τις αλλάζουμε.
- Δηλώνουμε την νέα μεταβλητή ως final.
- Αλλάζουμε όλες τις αναφορές τις με το νέο όνομα μέχρι την δεύτερη ανάθεσή της.
- Δηλώνουμε μία νέα προσωρινή μεταβλητή στην δεύτερη ανάθεση.
- Κάνουμε compile και τεστάρουμε τον κώδικα.
- Επαναλαμβάνουμε τα βήματα. Αλλάζουμε τις αναφορές της δεύτερης μεταβλητής μέχρι την επόμενη ανάθεση και κάνουμε το ίδιο για όλες τις αναθέσεις της αρχικής μεταβλητής που ακολουθούν.

4.5.3 Παράδειγμα

Σε αυτό το παράδειγμα έχουμε έναν αλγόριθμο οποίος υπολογίζει μία απόσταση. Η τελική απόσταση εξαρτάται από κάποιους ενδιάμεσους σταθμούς.

```
double getDistanceTravelled (int time) {
    double result;
    double acc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * acc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;

    if (secondaryTime > 0) {
        double primaryVel = acc * _delay;
        acc = (_primaryForce + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime + 0.5 * acc *
               secondaryTime * secondaryTime;
    }

    return result;
}
```

Το ενδιαφέρον κομμάτι του παραδείγματός μας είναι η διπλή ανάθεση στην μεταβλητή `acc`. Έχει δύο αρμοδιότητες. Να κρατάει την αρχική και την τελική επιτάχυνση (acceleration).

Ξεκινάμε αρχικά αλλάζοντας το όνομα της μεταβλητής και δηλώνοντας την καινούρια ως final. Μετά αλλάζουμε όλες τις αναφορές στην μεταβλητή μέχρι την δεύτερη ανάθεση. Στην επόμενη ανάθεση κάνουμε εκ νέου δήλωση της αρχικής μεταβλητής.

```
double getDistanceTravelled (int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;

    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        double acc = (_primaryForce + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime + 0.5 * acc *
               secondaryTime * secondaryTime;
    }

    return result;
}
```

Επιλέγουμε ένα νέο όνομα για να αναπαραστήσουμε μόνο το πρώτο κομμάτι χρήσης της μεταβλητής. Την δηλώνουμε final για να βεβαιωθούμε ότι γίνεται ανάθεση σε αυτή μόνο μία φορά.

Συνεχίζουμε με την δεύτερη ανάθεση. Αυτό αφαιρεί τελείως το αρχικό όνομα της μεταβλητής.

```
double getDistanceTravelled (int time) {
    double result;
    final double primaryAcc = _primaryForce / _mass;
    int primaryTime = Math.min(time, _delay);
    result = 0.5 * primaryAcc * primaryTime * primaryTime;
    int secondaryTime = time - _delay;

    if (secondaryTime > 0) {
        double primaryVel = primaryAcc * _delay;
        final double secondaryAcc = (_primaryForce
                                + _secondaryForce) / _mass;
        result += primaryVel * secondaryTime + 0.5 * secondaryAcc *
               secondaryTime * secondaryTime;
    }

    return result;
}
```

4.6 Remove Assignments to Parameters (Αφαίρεση Ανάθεσης σε Παράμετρο)

Χρησιμοποιούμε αυτή την τεχνική όταν κάνουμε ανάθεση νέας τιμής σε παράμετρο μεθόδου.

Χρησιμοποιούμε μία προσωρινή μεταβλητή αντί για την ανάθεση στην παράμετρο.

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50) inputVal -= 2;
```

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;
    if (inputVal > 50) result -= 2;
```

4.6.1 Κίνητρο

Η ανάθεση σε παράμετρο σημαίνει ότι αν περνάμε σε μία μέθοδο ένα αντικείμενο, και στην συνέχεια αναθέτουμε σε αυτό ένα νέο αντικείμενο, αλλάζουμε το αρχικό αντικείμενο με ένα άλλο. Ο λόγος που δεν είναι σωστό να χρησιμοποιούμε ανάθεση σε παράμετρο είναι ότι χάνεται η σαφήνεια και δημιουργείται σύγχυση μεταξύ των εννοιών Τιμή Παραμέτρου (pass by value) και Αναφορά Παραμέτρου (pass by reference).

Όταν περνάμε την τιμή της παραμέτρου, οι αλλαγές που γίνονται στην παράμετρο δεν έχουν κανένα αντίκτυπο στο αρχικό αντικείμενο.

4.6.2 Μηχανισμοί

- Δημιουργούμε μία προσωρινή μεταβλητή για την παράμετρο.
- Αντικαθιστούμε όλες τις αναφορές στην παράμετρο, μετά την ανάθεση στην νέα μεταβλητή, με την νέα μεταβλητή.
- Αλλάζουμε στην ανάθεση ώστε αυτή να γίνεται στη νέα μεταβλητή.
- Κάνουμε compile και τεστάρουμε τον κώδικα.

4.6.3 Παράδειγμα

```
int discount (int inputVal, int quantity, int yearToDate) {
    if (inputVal > 50)
        inputVal -= 2;
    if (quantity > 100)
        inputVal -= 1;
    if (yearToDate > 10000)
        inputVal -= 4;

    return inputVal;
}
```

Η αντικατάσταση με προσωρινή μεταβλητή έχει ως εξής.

```
int discount (int inputVal, int quantity, int yearToDate) {
    int result = inputVal;

    if (inputVal > 50)
        result -= 2;
    if (quantity > 100)
        result -= 1;
    if (yearToDate > 10000)
        result -= 4;

    return result;
}
```

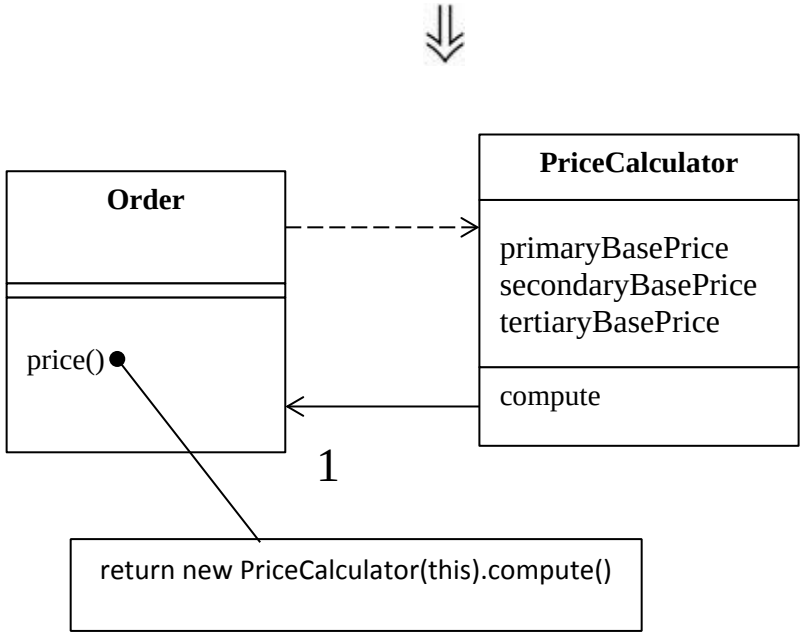
Μπορούμε να αναγκάσουμε την δημιουργία της νέας μεταβλητής δηλώνοντας τις παραμέτρους ως final.

```
int discount (final int inputVal, final int quantity, final int yearToDate) {
    int result = inputVal;

    if (inputVal > 50)
        result -= 2;
    if (quantity > 100)
        result -= 1;
    if (yearToDate > 10000)
        result -= 4;

    return result;
}
```

4.7 Replace Method with Method Object (Αντικατάσταση Μεθόδου με Μέθοδο Αντικειμένου)

Χρησιμοποιούμε αυτή την τεχνική όταν έχουμε μία μεγάλη μέθοδο η οποία χρησιμοποιεί τοπικές μεταβλητές με τέτοιον τρόπο που δυσκολεύει την εφαρμογή της Εξαγωγής Μεθόδου.

```
class Order...
    double price() {
        double primaryBasePrice;
        double secondaryBasePrice;
        double tertiaryBasePrice;
        // long computation;
        ...
    }
    ...
}
```



4.7.1 Κίνητρο

Η δυσκολία στην αποσύνθεση μίας μεθόδου προκύπτει από τις τοπικές μεταβλητές. Αν είναι πάρα πολλές, η αποσύνθεση δεν είναι εύκολη. Η χρήση της Αντικατάστασης Προσωρινής Μεταβλητής με Μέθοδο βοηθά στην μείωση των μεταβλητών, αλλά κάποιες φορές θα διαπιστώσουμε ότι δεν μπορούμε να σπάσουμε μία μέθοδο σε μικρότερες.

Χρησιμοποιώντας την Αντικατάσταση Μεθόδου με Μέθοδο Αντικειμένου μετατρέπουμε όλες αυτές τις παραμέτρους σε πεδία στην μέθοδο του νέου αντικειμένου. Έπειτα μπορούμε να εφαρμόσουμε την Εξαγωγή Μεθόδου στο νέο αντικείμενο για να δημιουργήσουμε νέες μεθόδους και να μειώσουμε το μέγεθος της αρχικής.

4.7.2 Μηχανισμοί

- Δημιουργούμε μία καινούρια κλάση με όνομα σχετικό με την μέθοδο.
- Δηλώνουμε στην νέα μέθοδο ένα final πεδίο για το αντικείμενο στο οποίο βρίσκεται η αρχική μέθοδος (source object) και ένα πεδίο για κάθε προσωρινή μεταβλητή και παράμετρο της αρχικής μεθόδου.
- Δηλώνουμε στην νέα κλάση έναν δομητή οποίος έχει ως παραμέτρους το αντικείμενο (source object) και όλες τις παραμέτρους και μεταβλητές τις μεθόδου.
- Δηλώνουμε στην νέα κλάση μία μέθοδο με όνομα “compute”.
- Αντιγράφουμε τον κώδικα της αρχικής μεθόδου μέσα στην μέθοδο compute. Χρησιμοποιούμε το αντικείμενο (source object) για κάθε κλήση σε άλλες μεθόδους της κλάσης που βρίσκεται η αρχική μέθοδος.
- Κάνουμε compile τον κώδικα.
- Αντικαθιστούμε την παλιά μέθοδο με μία νέα που δημιουργεί ένα αντικείμενο της νέας κλάσης και καλεί την compute.

4.7.3 Παράδειγμα

```
Class Account
    int gamma (int inputVal, int quantity, int yearToDate) {
        int importantValue1 = (inputVal * quantity) + delta();
        int importantValue2 = (inputVal * yearToDate) + 100;

        if ((yearToDate - importantValue1) > 100)
            importantValue2 -= 20;
        int importantValue3 = importantValue2 * 7;
        // and so on.
        return importantValue3 - 2 * importantValue1;
}
```

Για να μετατρέψουμε την παραπάνω μέθοδο σε μέθοδο αντικειμένου ξεκινάμε δηλώνοντας μία νέα κλάση. Εισάγουμε ένα final πεδίο για το αρχικό αντικείμενο και ένα πεδίο για κάθε παράμετρο και τοπική μεταβλητή της μεθόδου.

```
class Gamma {
    ...
    private final Account _account;
    private int _inputVal;
    private int _quantity;
    private int _yearToDate;
    private int _importantValue1;
    private int _importantValue2;
    private int _importantValue3;
}
```

Στην συνέχεια δημιουργούμε τον δομητή.

```
Gamma (Account source, int inputValArg, int quantityArg, int
       yearToDateArg) {
    _account = source;
    _inputVal = inputValArg;
    _quantity = quantityArg;
    _yearToDate = yearToDateArg;
}
```

Τώρα μπορούμε να μεταφέρουμε την αρχική μέθοδο στην νέα κλάση. Χρειάζεται να τροποποιήσουμε κάθε κλήση σε μεθόδους της κλάσης Account ώστε να χρησιμοποιούν το πεδίο `_account`.

```
int compute () {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;

    if ((yearToDate - importantValue1) > 100)
        importantValue2 -= 20;

    int importantValue3 = importantValue2 * 7;
    // and so on.
    return importantValue3 - 2 * importantValue1;
}
```

Τροποποιούμε την παλιά μέθοδο ώστε να χρησιμοποιεί την μέθοδο του νέου αντικειμένου.

```
int gamma (int inputVal, int quantity, int yearToDate) {
    return new Gamma(this, inputVal, quantity, yearToDate).compute();
}
```

Το αποτέλεσμα της παραπάνω διαδικασίας είναι ότι τώρα μπορούμε πολύ εύκολα να χρησιμοποιήσουμε την Εξαγωγή Μεθόδου στην μέθοδο compute χωρίς να ανησυχούμε για τις παραμέτρους.

```
int compute () {
    importantValue1 = (inputVal * quantity) + _account.delta();
    importantValue2 = (inputVal * yearToDate) + 100;

    importantThing();

    int importantValue3 = importantValue2 * 7;
    // and so on.
    return importantValue3 - 2 * importantValue1;
}

void importantThing() {
    if ((yearToDate - importantValue1) > 100)
        importantValue2 -= 20;
}
```

4.8 Substitute Algorithm (Αντικατάσταση Αλγορίθμου)

Χρησιμοποιούμε αυτή την τεχνική όταν έχουμε έναν αλγόριθμο ο οποίος δεν είναι αρκετά ξεκάθαρος.

Αντικαθιστούμε το σώμα της μεθόδου με ένα νέο αλγόριθμο.

```
String foundPerson(String[] people){
    for (int i = 0; i < people.length; i++) {
        if (people[i].equals ("Don")){
            return "Don";
        }

        if (people[i].equals ("John")){
            return "John";
        }

        if (people[i].equals ("Kent")){
            return "Kent";
        }
    }

    return "";
}
```

```
String foundPerson(String[] people){
    List candidates = Arrays.asList(new String[] {"Don", "John", "Kent"});

    for (int i=0; i<people.length; i++)
        if (candidates.contains(people[i]))
            return people[i];

    return "";
}
```

4.8.1 Κίνητρο

Αν μπορούμε να βρούμε έναν πιο ξεκάθαρο τρόπο να κάνουμε κάτι, πρέπει να αντικαταστήσουμε τον πολύπλοκο τρόπο με αυτόν. Το refactoring μπορεί να σπάσει κάτι πολύπλοκο σε μικρότερα κομμάτια, αλλά κάποιες φορές φτάνουμε στο σημείο που πρέπει να αφαιρέσουμε έναν ολόκληρο αλγόριθμο και να τον αντικαταστήσουμε με κάτι πιο απλό.

Κάποιες φορές όταν θέλουμε να αλλάξουμε έναν αλγόριθμο για να κάνει κάτι λίγο διαφορετικό, είναι ευκολότερο να αντικαταστήσουμε τον αλγόριθμο με κάτι πιο απλό και έπειτα να κάνουμε την αλλαγή που πρέπει.

4.8.2 Μηχανισμοί

- Ετοιμάζουμε τον εναλλακτικό αλγόριθμο και ελέγχουμε ότι κάνει compile.
- Τεστάρουμε τον νέο αλγόριθμο. Αν τα αποτελέσματα είναι ίδια με τον προηγούμενο τότε έχουμε τελειώσει.
- Αν τα αποτελέσματα είναι διαφορετικά, χρησιμοποιούμε τον παλιό αλγόριθμο για σύγκριση κάνοντας τεστ και debug.

5 Μετακίνηση Χαρακτηριστικών μεταξύ των Αντικειμένων

Μία από τις πιο θεμελιώδεις αποφάσεις στον σχεδιασμό των αντικειμένων είναι το πως και που θα αναθέσουμε τις αρμοδιότητες. Συνήθως η αρχική μας απόφαση δεν είναι η πιο σωστή. Με το refactoring μπορούμε να αλλάξουμε αυτή την απόφαση.

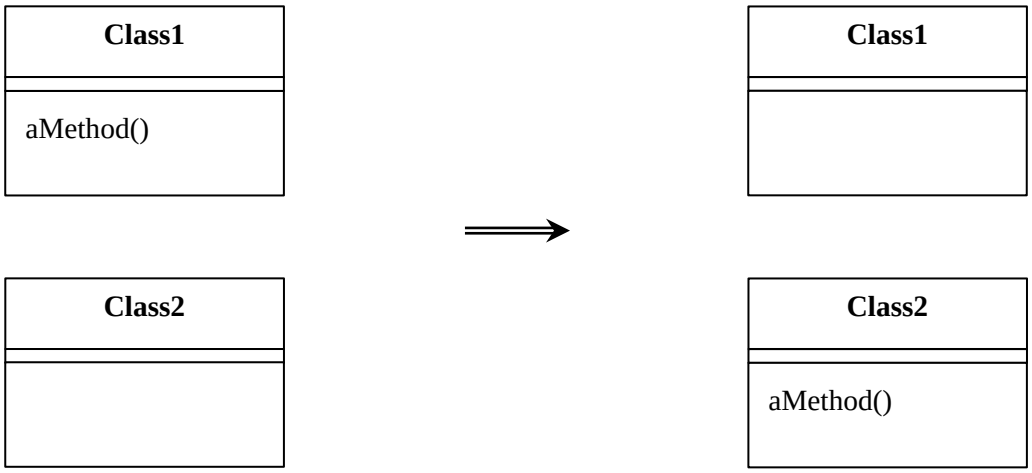
Συνήθως μπορούμε να λύσουμε αυτά τα προβλήματα με την χρήση της Μετακίνησης Μεθόδου και της Μετακίνησης Πεδίου για να μεταφέρουμε την συμπεριφορά των αντικειμένων από το ένα στο άλλο.

Συχνά οι κλάσεις μεγαλώνουν αρκετά και έχουν πάρα πολλές αρμοδιότητες. Σε αυτή την περίπτωση χρησιμοποιούμε την Εξαγωγή Μεθόδου για να ξεχωρίσουμε μερικές από τις αρμοδιότητες. Αν μία κλάση γίνεται υπερβολικά 'ανεύθυνη' μπορούμε να την ενσωματώσουμε σε μία άλλη με την εφαρμογή της Ενσωμάτωσης Κλάσης. Αν μία άλλη κλάση χρησιμοποιείται από αυτήν, είναι χρήσιμο να την κρύψουμε με την Απόκρυψη Αντιπροσώπου. Κάποιες φορές το να κρύβουμε τον αντιπρόσωπο μίας κλάσης έχει ως αποτέλεσμα το να αλλάζομε συνεχώς το interface της, και σε αυτή την περίπτωση χρησιμοποιούμε την Αφαίρεση Ενδιάμεσου.

5.1 Move Method (Μετακίνηση Μεθόδου)

Χρησιμοποιούμε αυτή την τεχνική όταν μία μέθοδος χρησιμοποιείται περισσότερο από μία άλλη κλάση παρά από αυτή στην οποία ανήκει.

> *Δημιουργούμε μία νέα μέθοδο με παρόμοιο σώμα στην κλάση η οποία την χρησιμοποιεί περισσότερο. Έπειτα, είτε τροποποιούμε την παλιά μέθοδο ώστε να χρησιμοποιεί την καινούρια, είτε την αφαιρούμε τελείως.*



5.1.1 Κίνητρο

Η μετακίνηση μεθόδου είναι από τις πιο βασικές τεχνικές refactoring. Μετακινούμε μία μέθοδο όταν μία κλάση έχει υπερβολική λειτουργικότητα ή όταν κάποιες κλάσεις συνεργάζονται πάρα πολύ και δημιουργούν μεγάλη αλληλεξάρτηση μεταξύ τους. Με την μετακίνηση μεθόδων από κλάση σε κλάση

μπορούμε να κάνουμε τις κλάσεις πιο απλές και καταλήγουν να έχουν ένα πιο καθαρό σύνολο ευθυνών.

Συνήθως ψάχνουμε για μεθόδους μίας κλάσης οι οποίες φαίνεται να έχουν περισσότερες αναφορές σε ένα άλλο αντικείμενο παρά σε αυτό που ανήκουν. Ένα καλό σημείο να το κάνουμε αυτό είναι αφού έχουμε μετακινήσει κάποια πεδία. Όταν βρούμε μία μέθοδο που ίσως να χρειάζεται μετακίνηση κοιτάμε τις μεθόδους που την καλούν και αυτές που καλεί και εκτιμούμε αν πρέπει να προχωρήσουμε με βάση το αντικείμενο με το οποίο η μέθοδος φαίνεται να έχει μεγαλύτερη αλληλεπίδραση.

Δεν είναι πάντα μία εύκολη απόφαση. Αν δεν είμαστε σίγουροι που πρέπει να μετακινήσουμε μία μέθοδο συνεχίζουμε κοιτώντας και άλλες μεθόδους. Μετακινώντας άλλες μεθόδους πρώτα συχνά κάνουμε την απόφασή μας πιο εύκολη.

5.1.2 Μηχανισμοί

- Εξετάζουμε όλα τα χαρακτηριστικά που χρησιμοποιεί η μέθοδος και είναι ορισμένα μέσα στην ίδια κλάση. Αποφασίζουμε αν θα πρέπει να μετακινηθούν και αυτά.
    - Αν τα χαρακτηριστικά χρησιμοποιούνται μόνο από την μέθοδο την οποία πρόκειται να μεταφέρουμε, τότε μεταφέρουμε και αυτά.
    - Αν χρησιμοποιούνται και από άλλες μεθόδους, εξετάζουμε αν μπορούμε να μετακινήσουμε και αυτές.
- Ελέγχουμε τις υποκλάσεις και υπερκλάσεις της κλάσης στην οποία ανήκει η μέθοδος για δηλώσεις της μεθόδου.
    - Αν υπάρχουν και άλλες δηλώσεις τότε ίσως να μην μπορούμε να κάνουμε την μετακίνηση.
- Δηλώνουμε την μέθοδο στην κλάση που πρόκειται να μεταφερθεί.
    - Ίσως χρειαστεί να επιλέξουμε νέο όνομα που να έχει περισσότερο νόημα στην νέα κλάση.
- Αντιγράφουμε τον κώδικα από την αρχική μέθοδο στην καινούρια. Κάνουμε τις απαραίτητες τροποποιήσεις ώστε να δουλεύει στην νέα κλάση.
    - Αν η μέθοδος χρησιμοποιεί την αρχική κλάση, πρέπει να ορίσουμε πως θα χρησιμοποιήσουμε τις αναφορές σε αυτήν από την νέα κλάση. Αν δεν υπάρχει αναφορά σε αυτή μέσα από την κλάση πρέπει να περάσουμε το αντικείμενο της αρχικής κλάσης στην μέθοδο ως παράμετρο.
    - Αν η μέθοδος χρησιμοποιεί handlers για exceptions αποφασίζουμε ποια από τις δύο κλάσεις θα πρέπει λογικά να χειρίζεται τα exceptions.
- Κάνουμε compile την κλάση στην οποία μεταφέρθηκε η μέθοδος.
- Ορίζουμε πως η αρχική κλάση θα αναφέρεται στην άλλη.
    - Ίσως να υπάρχει ήδη μία μέθοδος ή ένα πεδίο με το οποίο μπορούμε να έχουμε αναφορά στην κλάση. Αν όχι, ελέγχουμε αν μπορούμε εύκολα να δημιουργήσουμε μία μέθοδο που θα το κάνει. Αν δεν γίνεται, πρέπει να δηλώσουμε ένα νέο πεδίο στην αρχική κλάση που θα είναι ένα αντικείμενο της άλλης.

- Αλλάζουμε την αρχική μέθοδο ώστε να χρησιμοποιεί την καινούρια.
- Κάνουμε compile και τεστάρουμε τον κώδικα.
- Αποφασίζουμε αν θα πρέπει να αφαιρέσουμε την αρχική μέθοδο ή να την διατηρήσουμε καλώντας την καινούρια.
    - Την αφήνουμε την ως έχει αν υπάρχουν πολλές αναφορές σε αυτή.
- Αν αφαιρέσουμε την αρχική μέθοδο, αντικαθιστούμε όλες τις αναφορές σε αυτή με κλήσεις στην νέα μέθοδο.
- Κάνουμε compile και τεστάρουμε ξανά τον κώδικα.

5.1.3 Παράδειγμα

```
class Account {
    ...
    double overdraftCharge() {
        if (_type.isPremium()) {
            double result = 10;
            if (_daysOverdrawn > 7)
                result += (_daysOverdrawn - 7) * 0.85;

            return result;
        }
        else
            return _daysOverdrawn * 1.75;
    }

    double bankCharge() {
        double result = 4.5;
        if (_daysOverdrawn > 0)
            result += overdraftCharge();

        return result;
    }

    private AccountType _type;
    private int _daysOverdrawn;
}
```

Αν υποθέσουμε ότι έχουμε πολλούς διαφορετικούς τύπους λογαριασμών (account types) , τότε για κάθε ένα από αυτούς θα έχουμε και ένα διαφορετικό κανόνα για να υπολογίσουμε το ποσό υπερανάληψης (overdraft charge). Άρα είναι προτιμότερο να μετακινήσουμε την μέθοδο `overdraftCharge` στην κλάση που δηλώνει τον τύπο του λογαριασμού (account type).

Το πρώτο βήμα είναι να κοιτάξουμε τα χαρακτηριστικά τα οποία χρησιμοποιεί η `overdraftCharge` και να δούμε αν αξίζει να μετακινήσουμε πολλές μεθόδους μαζί. Σε αυτή την περίπτωση χρειαζόμαστε το πεδίο `_daysOverdrawn` να παραμείνει στην κλάση Account επειδή αυτό θα είναι διαφορετικό για κάθε λογαριασμό.

Μετακινούμε λοιπόν την μέθοδο στην κλάση AccountType και κάνουμε τις απαραίτητες τροποποιήσεις.

```
class AccountType {
    ...
    double overdraftCharge(int daysOverdrawn) {
        if (isPremium()) {
            double result = 10;
            if (daysOverdrawn > 7)
                result += (daysOverdrawn - 7) * 0.85;

            return result;
        }
        else
            return daysOverdrawn * 1.75;
    }
}
```

Σε αυτή την περίπτωση, για να ταιριάζει η μέθοδος στην νέα κλάση έπρεπε να αφαιρέσουμε το `_type,` που χρησιμοποιείται για τις αναφορές σε χαρακτηριστικά της κλάσης AccountType, και να περάσουμε ως παράμετρο ένα από τα χαρακτηριστικά της κλάσης Account το οποίο εξακολουθεί να χρειάζεται η μέθοδος.

Όταν χρειαζόμαστε ένα από τα χαρακτηριστικά της αρχικής κλάσης τότε έχουμε τέσσερις επιλογές:

1. Μεταφέρουμε και το χαρακτηριστικό αυτό στην νέα κλάση.
2. Δημιουργούμε η χρησιμοποιούμε μία αναφορά στην αρχική κλάση.
3. Περνάμε το όλο αρχικό αντικείμενο ως παράμετρο στην μέθοδο.
4. Αν το χαρακτηριστικό είναι μία μεταβλητή την περνάμε ως παράμετρο.

Αφού κάνουμε compile και τεστάρουμε την νέα μέθοδο, μπορούμε να αντικαταστήσουμε το σώμα της αρχικής μεθόδου με μία κλήση στην καινούρια.

```
class Account {
    ...
    double overdraftCharge() {
        return _type.overdraftCharge(_daysOverdrawn);
    }
}
```

Μπορούμε να αφήσουμε την μέθοδο όπως είναι ή να την αφαιρέσουμε. Για να την αφαιρέσουμε πρέπει να βρούμε όλες τις κλήσεις σε αυτήν και να τις αλλάξουμε ώστε να καλούν την νέα μέθοδο.

```
class Account {
    ...
    double bankCharge() {
        double result = 4.5;
            if (_daysOverdrawn > 0)
                result += _type.overdraftCharge(_daysOverdrawn);

        return result;
    }
}
```

Αφού αντικαταστήσουμε όλες τις κλήσεις μπορούμε να διαγράψουμε την μέθοδο. Αν η μέθοδος δεν είναι private πρέπει να κοιτάξουμε και άλλες κλάσεις που ίσως την χρησιμοποιούν.

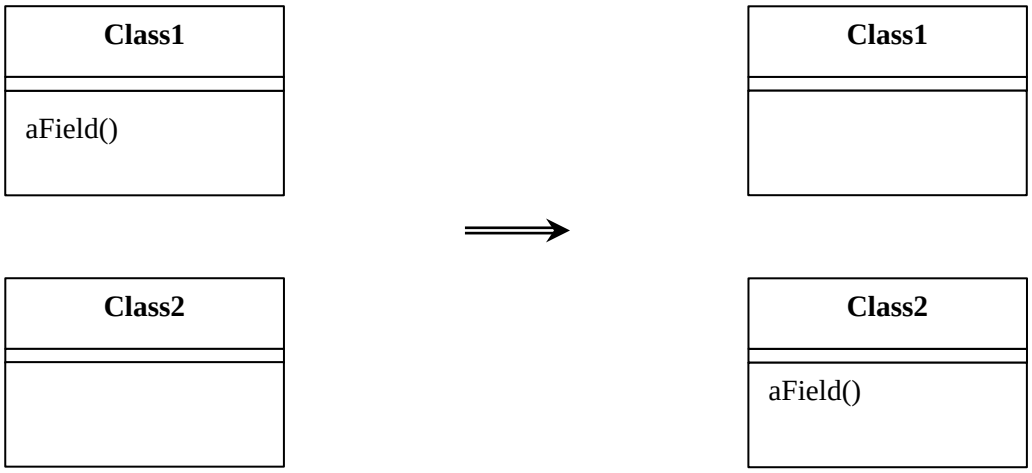
Στο παράδειγμά μας, η μέθοδος χρησιμοποιούσε μόνο ένα πεδίο της κλάσης Account και περάσαμε το πεδίο αυτό ως παράμετρο στην νέα μέθοδο. Αν όμως η μέθοδος καλούσε μία άλλη μέθοδο της κλάσης Account δεν θα μπορούσαμε να κάνουμε το ίδιο. Σε τέτοιες περιπτώσεις πρέπει να περνάμε ως παράμετρο όλο το αρχικό αντικείμενο.

```
class AccountType {
    ...
    double overdraftCharge(Account account) {
        if (isPremium()) {
            double result = 10;
            if (account.getDaysOverdrawn() > 7)
                result += (account.getDaysOverdrawn() - 7) * 0.85;

            return result;
        }
        else
            return account.getDaysOverdrawn() * 1.75;
    }
}
```

5.2 Move Field (Μετακίνηση Πεδίου)

Χρησιμοποιούμε αυτή την τεχνική όταν ένα πεδίο χρησιμοποιείται περισσότερο από μία άλλη κλάση παρά από αυτή στην οποία ανήκει.

Δημιουργούμε ένα νέο πεδίο στην κλάση που το χρησιμοποιεί περισσότερο και αλλάζουμε όλες τις χρήσεις του.



5.2.1 Κίνητρο

Η μεταφορά κατάστασης και συμπεριφοράς μεταξύ των κλάσεων είναι πολύ ουσιώδης στο refactoring. Καθώς αναπτύσσουμε το σύστημά μας έχουμε ανάγκη για περισσότερες κλάσεις και για μεταφορά συμπεριφοράς από την μία στην άλλη. Μία σχεδιαστική απόφαση που μπορεί να είναι σωστή σήμερα μπορεί να είναι λανθασμένη μια βδομάδα αργότερα.

Μετακινούμε ένα πεδίο όταν βλέπουμε πολλές μεθόδους μίας άλλης κλάσης να το χρησιμοποιούν περισσότερο από την ίδια την κλάση. Αυτή η χρήση μπορεί να είναι έμμεση μέσω των μεθόδων set και get. Ίσως επιλέξουμε να μετακινήσουμε και τις μεθόδους αυτές. Αν όμως φαίνεται λογικό να ανήκουν οι μέθοδοι εκεί που είναι, τότε μετακινούμε μόνο το πεδίο.

Ένας άλλος λόγος για την μετακίνηση πεδίου είναι όταν κάνουμε Εξαγωγή Κλάσης. Σε αυτή την περίπτωση μετακινούμε πρώτα τα πεδία και μετά τις μεθόδους.

5.2.2 Μηχανισμοί

- Αν το πεδίο είναι public, χρησιμοποιούμε την μέθοδο Ενθυλάκωσης Πεδίου.
- Κάνουμε compile και τεστάρουμε τον κώδικα.
- Δημιουργούμε ένα πεδίο στην νέα κλάση μαζί με τις μεθόδους get/set.
- Κάνουμε compile την κλάση.
- Ορίζουμε πως θα έχουμε πρόσβαση στο νέο πεδίο από την αρχική κλάση.
    - Ένα υπάρχων πεδίο ή κλάση μπορεί να μας δίνει πρόσβαση στο πεδίο. Αν όχι, ελέγχουμε αν μπορούμε να δημιουργήσουμε εύκολα μία μέθοδο για αυτό τον σκοπό. Αν δεν γίνεται, πρέπει να δημιουργήσουμε ένα νέο πεδίο στην αρχική κλάση που θα κρατά όλο το αντικείμενο της άλλης κλάσης.
- Αφαιρούμε το πεδίο από την αρχική κλάση.
- Αντικαθιστούμε όλες τις αναφορές στο αρχικό πεδίο με αναφορές στην κατάλληλη μέθοδο της άλλης κλάσης.
    - Για να πάρουμε την τιμή του πεδίου αντικαθιστούμε την αναφορά με μία κλήση στην μέθοδο get.
    - Για να αναθέσετε τιμή στο πεδίο αντικαθιστούμε την αναφορά με μία κλήση στην μέθοδο set.
- Κάνουμε compile και τεστάρουμε τον κώδικα.

5.2.3 Παράδειγμα

```
class Account {
    ...
    private AccountType _type;
    private double _interestRate;

    double interestForAmount_days (double amount, int days) {
        return _interestRate * amount * days / 365;
    }
}
```

Θέλουμε να μετακινήσουμε το πεδίο `_interestRate` στην κλάση AccountType. Υπάρχουν όμως αρκετές μέθοδοι που έχουν αναφορά σε αυτό, μία από τις οποίες είναι η `interestForAmount_days`. Δημιουργούμε το πεδίο και τις μεθόδους πρόσβασης στην κλάση AccountType.

```
class AccountType {
    ...
    private double _interestRate;

    void setInterestRate (double arg) {
        _interestRate = arg;
    }

    double getInterestRate () {
        return _interestRate;
    }
}
```

Μετά αλλάζουμε τις μεθόδους της κλάσης Account ώστε να χρησιμοποιούν την νέα μέθοδο και αφαιρούμε το πεδίο `_interestRate` από την Account. Πρέπει να αφαιρέσουμε το πεδίο για να είμαστε σίγουροι ότι πλέον όλες οι μέθοδοι της Account χρησιμοποιούν το νέο πεδίο στην AccountType. Σε αυτό μας βοηθά και ο compiler ο οποίος θα εμφανίσει ως σφάλματα τυχών αναφορές στο αρχικό πεδίο.

```
class Account {
    ...
    private AccountType _type;

    double interestForAmount_days (double amount, int days) {
        return _type.getInterestRate() * amount * days / 365;
    }
}
```

5.2.4 Παράδειγμα Αυτοενθυλάκωσης Πεδίου

Αν πολλές μέθοδοι χρησιμοποιούν το πεδίο `_interestRate`, είναι προτιμότερο να χρησιμοποιήσουμε την αυτοενθυλάκωση Πεδίου.

```
class Account {
    ...
    private AccountType _type;
    private double _interestRate;

    double interestForAmount_days (double amount, int days) {
        return getInterestRate() * amount * days / 365;
    }

    private void setInterestRate (double arg) {
        _interestRate = arg;
    }

    private double getInterestRate () {
        return _interestRate;
    }
}
```

Με αυτό τον τρόπο χρειάζεται να αλλάξουμε μόνο τις μεθόδους πρόσβασης σε αυτό.

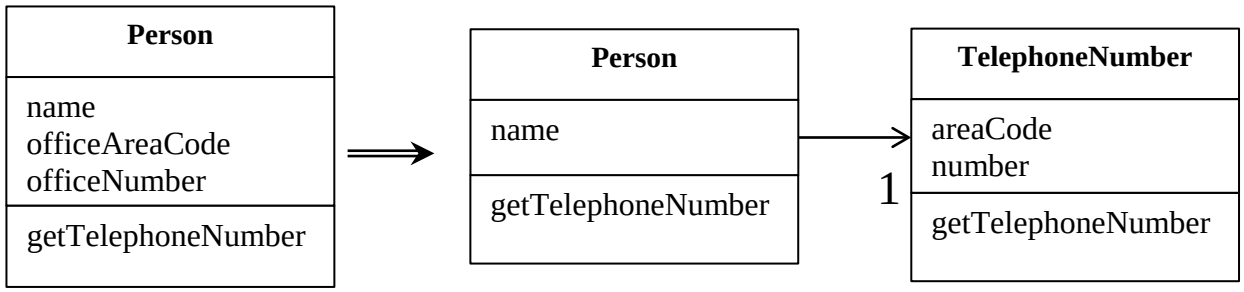
```
class Account {
    ...
    private AccountType _type;

    double interestForAmount_days (double amount, int days) {
        return getInterestRate() * amount * days / 365;
    }

    private void setInterestRate (double arg) {
        _type.setInterestRate = arg;
    }

    private double getInterestRate () {
        return _type.getInterestRate;
    }
}
```

Η χρήση της αυτοενθυλάκωσης Πεδίου είναι ιδιαίτερα χρήσιμη όταν εκτελούμε την Μετακίνηση Μεθόδου γιατί μας επιτρέπει να κάνουμε μικρές αλλαγές στην κλάση και μειώνουμε τις πιθανότητα να κάνουμε λάθη.

5.3 Extract Class (Εξαγωγή Κλάσης)

Χρησιμοποιούμε αυτή την τεχνική όταν έχουμε μία κλάση η οποία κάνει πράγματα που θα έπρεπε να γίνονται από δύο.

Δημιουργούμε μία νέα κλάση και μετακινούμε σε αυτά τα σχετικά πεδία και μεθόδους από την αρχική κλάση.



5.3.1 Κίνητρο

Έχουμε σίγουρα ακούσει ότι μία κλάση θα πρέπει να είναι αφαιρετική (abstract), και να χειρίζεται μόνο λίγες αρμοδιότητες. Στην πραγματικότητα όμως οι κλάσεις μεγαλώνουν αρκετά. Συνεχώς προσθέτουμε νέες αρμοδιότητες σε μία κλάση πιστεύοντας ότι δεν αξίζει να δημιουργήσουμε μία νέα που θα χειρίζεται μία ή και δύο αρμοδιότητες. Οι νέες λειτουργίες που προσθέτουμε όμως στις κλάσεις μεγαλώνουν και κάνουν την κλάση αρκετά πολύπλοκη.

Μία τέτοια κλάση περιέχει πολλές μεθόδους και πεδία. Οι μεγάλες κλάσεις είναι δύσκολες στην κατανόησή τους. Σε τέτοιες κλάσεις πρέπει να δούμε αν μπορούμε να τις χωρίσουμε. Ένα καλό σημάδι είναι όταν ένα υποσύνολο δεδομένων ή ένα υποσύνολο μεθόδων φαίνεται να ταιριάζουν να είναι μαζί. Ένα άλλο σημάδι είναι όταν ένα υποσύνολο δεδομένων αλλάζουν μαζί ή εξαρτώνται το ένα από το άλλο. Ένας εύκολος τρόπος είναι να αναρωτηθούμε τι θα γινόταν αν αφαιρούσαμε ένα συγκεκριμένο πεδίο ή μέθοδο. Ποια πεδία και μέθοδοι θα έπαυαν να είχαν νόημα;

5.3.2 Μηχανισμοί

- Αποφασίζουμε πως και ποιες αρμοδιότητες μίας κλάσης θα διαχωρίσουμε.
- Δημιουργούμε μία νέα κλάση που θα εκφράζει τις αρμοδιότητες αυτές.
    - Αν οι εναπομείνασες αρμοδιότητες της παλιάς κλάσης δεν εκφράζονται από το όνομά της, τότε μετονομάζουμε την κλάση.
- Δημιουργούμε μία σύνδεση από την παλιά στην νέα κλάση.
    - Ίσως χρειαζόμαστε αμφίδρομη σύνδεση μεταξύ των κλάσεων, αλλά δεν την δημιουργούμε ακόμα μέχρι να βεβαιωθείτε ότι πραγματικά χρειάζεται.
- Χρησιμοποιούμε την Μετακίνηση Πεδίου για κάθε ένα από τα πεδία που θέλουμε να μετακινήσουμε.
- Κάνουμε compile και τεστάρουμε τον κώδικα μετά από κάθε μετακίνηση.
- Χρησιμοποιούμε την Μετακίνηση Μεθόδου σε κάθε μία από τις μεθόδους που θέλουμε να μετακινήσουμε στην νέα κλάση. Ξεκινάμε από μεθόδους χαμηλού επιπέδου (αυτές που περισσότερο καλούνται παρά καλούν) και συνεχίζουμε με τις υψηλότερου επιπέδου.
- Κάνουμε compile και τεστάρουμε τον κώδικα μετά από κάθε μετακίνηση.
- Επανεξετάζουμε και μειώνουμε τα interface των κλάσεων.
    - Εάν τελικά δημιουργήσαμε αμφίδρομη σχέση μεταξύ των κλάσεων ελέγχουμε αν μπορούμε να την κάνουμε μονόδρομη.
- Αποφασίζουμε αν θα κάνουμε ορατή ή όχι την νέα κλάση προς άλλες κλάσεις. Αν την κάνουμε ορατή, αποφασίζουμε αν θα την χρησιμοποιήσουμε ως Αντικείμενο Αναφοράς (reference object) ή ως Αντικείμενο Αμετάβλητης Τιμής (immutable value object).

5.3.3 Παράδειγμα

Ξεκινάμε με μία απλή κλάση Person.

```
class Person {
    ...
    public String getName() {
        return _name;
    }

    public String getTelephoneNumber() {
        return ("(" + _officeAreaCode + ") " + _officeNumber);
    }

    String getOfficeAreaCode() {
        return _officeAreaCode;
    }

    void setOfficeAreaCode(String arg) {
        _officeAreaCode = arg;
    }

    String getOfficeNumber() {
        return _officeNumber;
    }

    void setOfficeNumber(String arg) {
        _officeNumber = arg;
    }

    private String _name;
    private String _officeAreaCode;
    private String _officeNumber;
}
```

Σε αυτή την περίπτωση μπορούμε να ξεχωρίσουμε την συμπεριφορά του τηλεφωνικού αριθμού σε μία νέα κλάση. Ξεκινάμε δημιουργώντας την νέα κλάση.

```
class TelephoneNumber {
}
```

Έπειτα δημιουργούμε μία σύνδεση από την κλάση Person στην κλάση TelephoneNumber.

```
class Person {
    ...
    private TelephoneNumber _officeTelephone = new TelephoneNumber();
}
```

Τώρα χρησιμοποιούμε την Μετακίνηση Πεδίου σε κάθε ένα από τα πεδία που μας ενδιαφέρουν.

```
class TelephoneNumber {
    String getAreaCode() {
        return _areaCode;
    }

    void setAreaCode(String arg) {
        _areaCode = arg;
    }

    private String _areaCode;
}

class Person {
    ...
    public String getTelephoneNumber() {
        return ("(" + getOfficeAreaCode() + ") " + _officeNumber);
    }

    String getOfficeAreaCode() {
        return _officeTelephone.getAreaCode();
    }

    void setOfficeAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);
    }
}
```

Αφού μετακινήσουμε το πεδίο `_officeAreaCode` κάνουμε το ίδιο και για το `_officeNumber` και παράλληλα, με την Μετακίνηση Μεθόδου μεταφέρουμε τις μεθόδους που σχετίζονται με το πεδίο αυτό.

```
class Person {
    ...
    public String getName() {
        return _name;
    }

    public String getTelephoneNumber(){
        return _officeTelephone.getTelephoneNumber();

    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }

    private String _name;
    private TelephoneNumber _officeTelephone = new TelephoneNumber();
}
```

```
class TelephoneNumber {
    ...
    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }

    String getAreaCode() {
        return _areaCode;
    }

    void setAreaCode(String arg) {
        _areaCode = arg;
    }

    String getNumber() {
        return _number;
    }

    void setNumber(String arg) {
        _number = arg;
    }

    private String _number;
    private String _areaCode;
}
```

Η επόμενη απόφαση είναι το κατά πόσο θα κάνουμε ορατή την νέα κλάση προς τις υπόλοιπες. Μπορούμε να την κρύψουμε τελείως προσθέτοντας στο interface της Person μεθόδους αντιπροσώπευσης (delegate methods) προς την TelephoneNumber, ή μπορούμε να την κάνουμε πλήρως ορατή προς όλους. Ίσως να την κάνουμε ορατή μόνο μέσα στο πακέτο που βρίσκεται και όχι προς τα έξω.

Αν την κάνουμε ορατή πρέπει να σκεφτούμε τους κινδύνους που ίσως να δημιουργηθούν. Μπορεί κάποια από τις κλάσεις-πελάτες να αλλάξει το πεδίο `_areaCode`, και είναι πολύ πιθανό αυτή η κλάση να μην έχει άμεση σχέση με την κλάση TelephoneNumber αλλά να την χρησιμοποιεί μέσω μίας άλλης κλάσης που και αυτή με την σειρά της καλεί μία άλλη και ούτω καθεξής.

Έχουμε λοιπόν τις εξής επιλογές:

1. Δεχόμαστε ότι οποιαδήποτε κλάση θα μπορεί να αλλάζει οποιοδήποτε σημείο της κλάσης TelephoneNumber. Αυτό κάνει την κλάση να είναι αντικείμενο αναφοράς, και ίσως πρέπει να χρησιμοποιήσουμε την Αλλαγή Τιμής σε Αναφορά (*SourceMaking, 2013, Change Value to Reference*). Σε αυτή την περίπτωση, η κλάση Person θα είναι η μόνη που θα έχει πρόσβαση στην TelephoneNumber και θα αποτελεί το σημείο πρόσβασης προς όλες τις άλλες.
2. Δεν θέλουμε οποιαδήποτε κλάση να αλλάζει την τιμή της TelephoneNumber χωρίς να περνάει μέσα από την κλάση Person. Μπορούμε είτε να κάνουμε την TelephoneNumber αμετάβλητη είτε να παρέχουμε ένα αμετάβλητο interface για αυτήν.

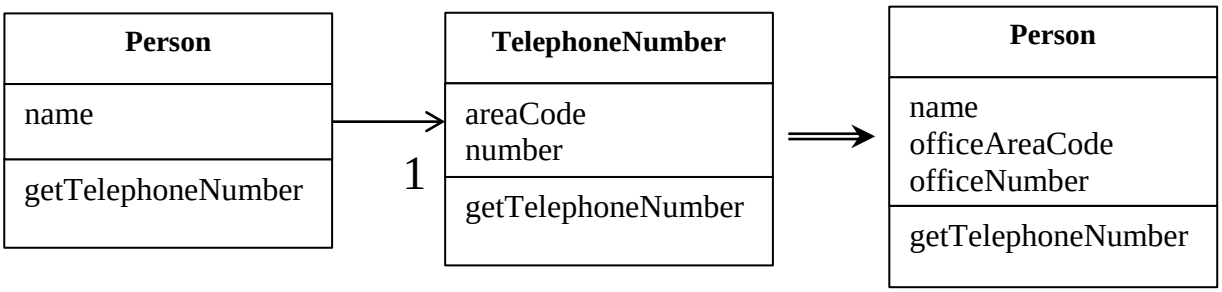
3. Μία άλλη πιθανότητα είναι να κλωνοποιούμε την TelephoneNumber πριν την εμφανίσουμε προς τα έξω. Αλλά αυτό ίσως οδηγήσει σε σύγχυση γιατί κάποιοι μπορεί να σκεφτούν ότι μπορούν να αλλάξουν την τιμή της.

Η Εξαγωγή Κλάσης είναι μία πολύ κοινή τεχνική για να βελτιώσουμε το πρόγραμμα μας γιατί μας επιτρέπει να έχουμε ξεχωριστούς τρόπους πρόσβασης στις δύο κλάσεις που δημιουργούνται. Μπορούμε να κάνουμε αν θέλουμε και τις δύο κλάσεις ορατές ή να αποκρύψουμε την μία μέσα στην άλλη.

5.4 Inline Class (Ενσωμάτωση Κλάσης)

Χρησιμοποιούμε αυτή την τεχνική όταν μία κλάση κάνει πάρα πολλά.

Μετακινούμε όλα τα χαρακτηριστικά της κλάσης σε μία άλλη και την διαγράφουμε.



5.4.1 Κίνητρο

Η Ενσωμάτωση Κλάσης είναι το αντίθετο της Εξαγωγής Κλάσης. Την χρησιμοποιούμε όταν μία κλάση δεν έχει πλέον μεγάλο βάρος και δεν χρειάζεται να υπάρχει. Συχνά αυτό είναι το αποτέλεσμα ενός refactoring που μετακινεί αρμοδιότητες από μία κλάση οπότε πλέον έχουν μείνει πάρα πολύ λίγες σε αυτή. Για αυτό τον λόγο θέλουμε να ενσωματώσουμε αυτή την κλάση σε μία άλλη που φαίνεται να την χρησιμοποιεί περισσότερο.

5.4.2 Μηχανισμοί

- Ορίζουμε το δημόσιο (public) πρωτόκολλο της κλάσης που θα καταργηθεί μέσα στην κλάση που θα την απορροφήσει.
    - Αν είναι λογικό να υπάρχει ξεχωριστό interface για την αρχική κλάση τότε χρησιμοποιούμε την Εξαγωγή Διεπαφής (Extract Interface).
- Αλλάζουμε όλες τις αναφορές στην αρχική κλάσης με αναφορές στην κλάση που θα ενσωματωθεί.
    - Δηλώνουμε την αρχική κλάση private για να αφαιρέσουμε αναφορές σε αυτή έξω από το πακέτο στο οποίο βρίσκεται. Επίσης αλλάζουμε το όνομα της κλάσης ώστε ο compiler να πιάσει όλες τις αναφορές σε αυτή.
- Κάνουμε compile και τεστάρουμε τον κώδικα.
- Χρησιμοποιούμε την Μετακίνηση Μεθόδου και την Μετακίνηση Πεδίου για να μεταφέρουμε τα χαρακτηριστικά από την αρχική κλάση μέσα στην άλλη.

5.4.3 Παράδειγμα

Θα χρησιμοποιήσουμε το προηγούμενο παράδειγμα για να ενσωματώσουμε ξανά την κλάση TelephoneNumber μέσα στην Person.

```
class Person {
    ...
    public String getName() {
        return _name;
    }

    public String getTelephoneNumber(){
        return _officeTelephone.getTelephoneNumber();
    }

    TelephoneNumber getOfficeTelephone() {
        return _officeTelephone;
    }

    private String _name;
    private TelephoneNumber _officeTelephone = new TelephoneNumber();
}
```

```
class TelephoneNumber {
    ...
    public String getTelephoneNumber() {
        return ("(" + _areaCode + ") " + _number);
    }

    String getAreaCode() {
        return _areaCode;
    }

    void setAreaCode(String arg) {
        _areaCode = arg;
    }

    String getNumber() {
        return _number;
    }

    void setNumber(String arg) {
        _number = arg;
    }

    private String _number;
    private String _areaCode;
}
```

Ξεκινάμε δηλώνοντας όλες τις private μεθόδους τις TelephoneNmber μέσα στην Person.

```
class Person {
    ...
    String getAreaCode() {
        return _officeTelephone.getAreaCode();
    }

    void setAreaCode(String arg) {
        _officeTelephone.setAreaCode(arg);
    }

    String getNumber() {
        return _officeTelephone.getNumber();
    }

    void setNumber(String arg) {
        _officeTelephone.setNumber(arg);
    }
}
```

Τώρα πρέπει να αλλάξουμε όλους τους ‘πελάτες’ της TelephoneNumber ώστε να χρησιμοποιούν τις νέες μεθόδους της κλάσης Person. Οπότε το παρακάτω παράδειγμα:

```
Person martin = new Person();
martin.getOfficeTelephone().setAreaCode("781");
```

γίνεται

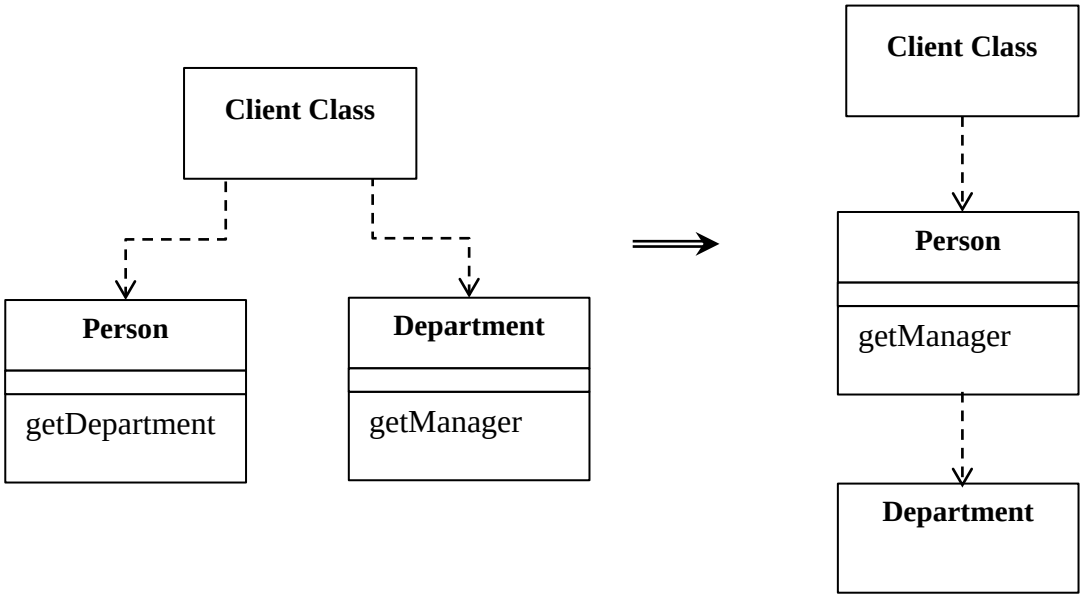
```
Person martin = new Person();
martin.setAreaCode("781");
```

Τέλος χρησιμοποιούμε την Μετακίνηση Μεθόδου και Μετακίνηση Πεδίου μέχρι να μεταφέρουμε όλα τα χαρακτηριστικά της TelephoneNumber και μετά την διαγράφουμε.

5.5 Hide Delegate (Απόκρυψη Αντιπροσώπου)

Χρησιμοποιούμε αυτή την τεχνική όταν μία κλάση πελάτης καλεί μία κλάση μέσω μίας άλλης.

Δημιουργούμε μεθόδους στην κλάση που μας παρέχει πρόσβαση για να κρύψουμε τον αντιπρόσωπο.



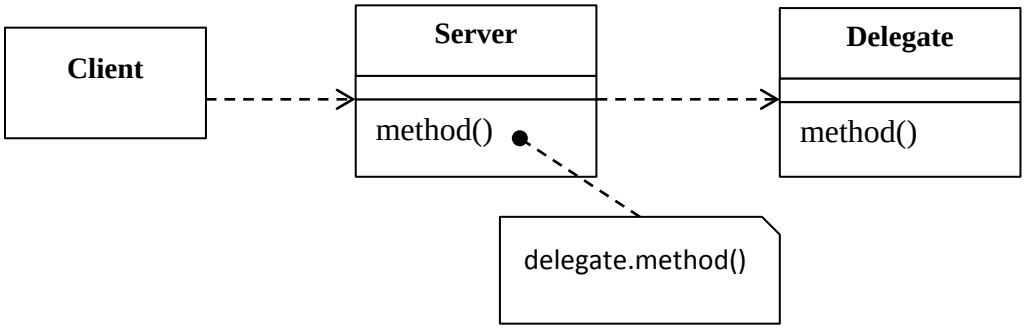
5.5.1 Κίνητρο

Ένα από τα κύρια σημεία των αντικειμένων είναι η ενθυλάκωση. Ενθυλάκωση σημαίνει ότι τα αντικείμενα χρειάζεται να ξέρουν λιγότερα για τα διάφορα μέρη του συστήματος. Έτσι, όταν τα πράγματα αλλάζουν, λιγότερα αντικείμενα χρειάζονται αλλαγές το οποίο κάνει την αλλαγή πιο εύκολη.

Οι έμπειροι προγραμματιστές γνωρίζουν ότι πρέπει να κρύβουν τα πεδία των κλάσεων παρά το γεγονός ότι η Java επιτρέπει public πεδία. Όσο γινόμαστε πιο έμπειροι αντιλαμβανόμαστε ότι υπάρχουν περισσότερα σημεία στον κώδικά μας τα οποία μπορούμε να ενθυλακώσουμε.

Αν μία κλάση πελάτης (client) καλεί μία μέθοδο που είναι ορισμένη μέσα σε ένα από τα πεδία της κλάσης εξυπηρετητής (server), τότε ο client πρέπει να γνωρίζει για την κλάση αντιπρόσωπο (delegate). Αν ο αντιπρόσωπος αλλάξει τότε πρέπει να γίνουν αλλαγές και στον client. Μπορούμε να αφαιρέσουμε αυτή την εξάρτηση προσθέτοντας μία μέθοδο αντιπροσώπευσης (delegating method) μέσα στην κλάση που παίζει τον ρόλο του server, η οποία θα κρύβει τον αντιπρόσωπο. Με αυτό τον τρόπο οι αλλαγές στον αντιπρόσωπο περιορίζονται μόνο σε αλλαγές στον server χωρίς να επηρεάζονται οι clients.

Εικόνα 7.1. Απλή αντιπροσώπευση



Ίσως αξίζει να χρησιμοποιήσουμε την Εξαγωγή Κλάσης για κάποιους ή για όλους τους clients του server. Αν κρύψουμε τον αντιπρόσωπο από όλους τους clients τότε μπορούμε να αφαιρέσουμε όλες αναφορές προς αυτόν από το interface του server.

5.5.2 Μηχανισμοί

- Για κάθε μέθοδο του αντιπροσώπου δημιουργούμε μία απλή μέθοδο αντιπροσώπευσης στον server.
- Προσαρμόζουμε τους clients να καλούν τον server.
    - Αν ο client δεν είναι στο ίδιο πακέτο με τον server τότε αλλάζουμε την πρόσβαση στις μεθόδους αντιπροσώπευσης ως προς την ορατότητα του πακέτου.
- Κάνουμε compile και τεστάρουμε τον κώδικα.
- Αν κανένας από τους clients δεν χρειάζεται πλέον να έχει πρόσβαση στον αντιπρόσωπο, αφαιρούμε τις μεθόδους αντιπροσώπευσης από τον server.
- Κάνουμε ξανά compile και τεστάρουμε τον κώδικα.

5.5.3 Παράδειγμα

Ξεκινάμε με τις κλάσεις Person και Department.

```
class Person {
    Department _department;

    public Department getDepartment() {
        return _department;
    }

    public void setDepartment(Department arg) {
        _department = arg;
    }
}

class Department {
    private String _chargeCode;
    private Person _manager;

    public Department (Person manager) {
    _manager = manager;
    }

    public Person getManager() {
        return _manager;
    }
    ...
}
```

Αν ένας client θέλει να μάθει το όνομα του manager ενός person, πρέπει πρώτα να πάρει το department:

```
manager = john.getDepartment().getManager();
```

Αυτό αποκαλύπτει στον client πως η κλάση Department δουλεύει και ότι αυτή είναι υπεύθυνη να κρατάει πληροφορία για τον Manager. Μπορούμε να μειώσουμε αυτή την εξάρτηση κρύβοντας την κλάση Department από τον client. Αυτό γίνεται εύκολα δημιουργώντας μία απλή μέθοδο αντιπροσώπευσης στην κλάση Person.

```
public Person getManager() {
    return _department.getManager();
}
```

Τώρα πρέπει να αλλάξουμε όλους τους clients της κλάσης Person να χρησιμοποιούν την μέθοδο αντιπροσώπευσης.

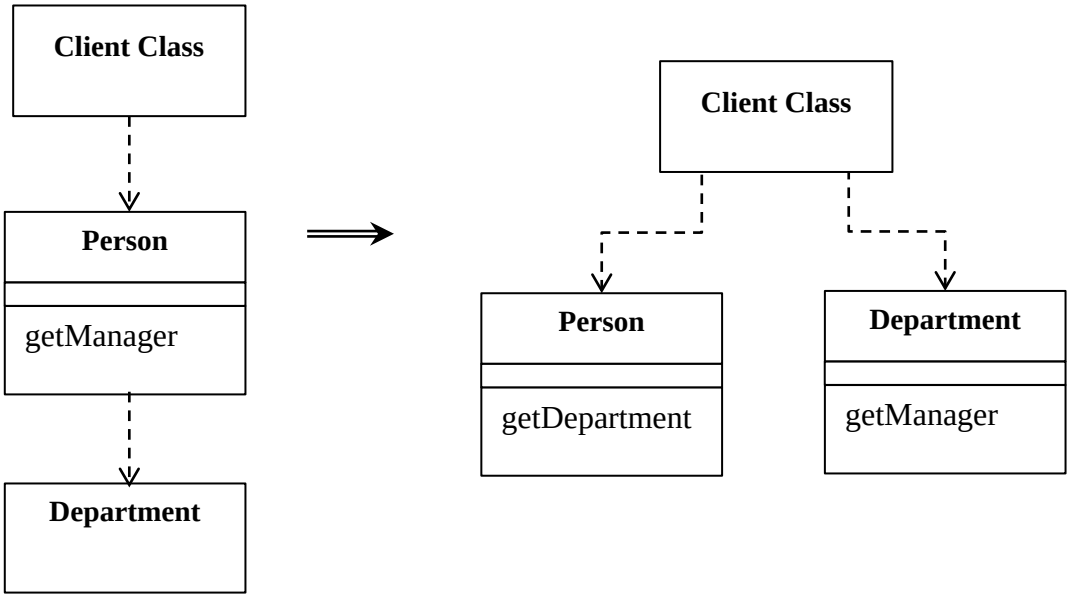
```
manager = john.getManager();
```

Αφού δημιουργήσουμε όλες τις μεθόδους αντιπροσώπευσης για την κλάση Department μέσα στην Person και αλλάξουμε όλους τους clients να καλούν αυτές τις μεθόδους, μπορούμε να διαγράψουμε την μέθοδο `getDepartment` η οποία έδινε πρόσβαση στους clients στον αντιπρόσωπο.

5.6 Remove Middle Man (Αφαίρεση Ενδιάμεσου)

Χρησιμοποιούμε αυτή την τεχνική όταν μία κλάση έχει πάρα πολλές απλές μεθόδους αντιπροσώπευσης.

Αφήνουμε τον client να χρησιμοποιεί απευθείας τον αντιπρόσωπο.



5.6.1 Κίνητρο

Στο κίνητρο για την χρήση της Απόκρυψης Αντιπροσώπου αναφέραμε τα πλεονεκτήματα της ενθυλάκωσης της χρήσης του αντικειμένου αντιπροσώπευσης. Υπάρχει όμως ένα τίμημα για αυτό. Το τίμημα είναι ότι κάθε φορά που ο client θέλει να χρησιμοποιήσει ένα νέο χαρακτηριστικό του αντιπροσώπου, πρέπει να προσθέσουμε μία μέθοδο αντιπροσώπευσης στον server. Όταν αρχίζουν να προστίθενται πολλά νέα χαρακτηριστικά αυτή η διαδικασία γίνεται επίπονη. Η κλάση Server γίνεται τότε ένας ενδιάμεσος και ίσως είναι καιρός οι clients να καλούν απευθείας τον αντιπρόσωπο.

Είναι δύσκολο να ορίσουμε ποιο είναι το ποσοστό των μεθόδων αντιπροσώπευσης που θα καθορίζουν πότε πρέπει η δεν πρέπει να έχουμε απευθείας πρόσβαση στον αντιπρόσωπο. Ευτυχώς, η Απόκρυψη Αντιπροσώπου και η Αφαίρεση Ενδιάμεσου δεν παίζουν μεγάλο ρόλο. Μπορούμε να προσαρμόσουμε το πρόγραμμά μας όπως θέλουμε. Καθώς το σύστημα μας εξελίσσεται και αλλάζει, αλλάζει και η βάση με την οποία χρησιμοποιούμε τους αντιπροσώπους. Μία καλή ενθυλάκωση κάποιους μήνες πριν μπορεί να φαίνεται περίεργη σήμερα.

5.6.2 Μηχανισμοί

- Δημιουργούμε μία μέθοδο πρόσβασης προς τον αντιπρόσωπο στον server.

- Για κάθε έναν από τους πελάτες που χρησιμοποιούν μία μέθοδο αντιπροσώπευσης, αφαιρούμε την μέθοδο από τον server και αντικαθιστούμε την κλήση στον client ώστε να καλεί την μέθοδο πρόσβασης στον αντιπρόσωπο.
- Κάνουμε compile και τεστάρουμε τον κώδικα.

5.6.3 Παράδειγμα

Θα χρησιμοποιήσουμε ξανά το προηγούμενο παράδειγμα με τις κλάσεις Person και Department αντίστροφα.

```
class Person {
    ...
    Department _department;

    public Person getManager() {
        return _department.getManager();
    }
}

class Department {
    ...
    private Person _manager;

    public Department(Person manager) {
        _manager = manager;
    }
}
```

Για να βρει τον manager ενός person, ο client χρησιμοποιεί τον παρακάτω κώδικα:

```
manager = john.getManager();
```

Αυτό είναι αρκετά απλό στην χρήση του και ενθυλακώνει την κλάση Department μέσα στον server. Ωστόσο, αν πολλές μέθοδοι κάνουν το ίδιο, καταλήγουμε να έχουμε πάρα πολλές απλές μεθόδους αντιπροσώπευσης μέσα στην κλάση Person. Τότε είναι ένα καλό σημείο να αφαιρέσουμε τον ενδιάμεσο.

Αρχικά δημιουργούμε μία μέθοδο πρόσβασης προς τον αντιπρόσωπο.

```
class Person {
    ...
    public Department getDepartment() {
        return _department;
    }
}
```

Έπειτα παίρνουμε κάθε κλήση σε μέθοδο αντιπροσώπευσης που χρησιμοποιούν οι clients και την αλλάζουμε ώστε να καλεί πρώτα την μέθοδο πρόσβασης.

```
manager = john.getDepartment().getManager();
```

Τέλος, μπορούμε να αφαιρέσουμε τις μεθόδους αντιπροσώπευσης (`getManager`) από την κλάση Person. Κάνοντας compile βλέπουμε αν μας έχει ξεφύγει κάτι.

5.7 Introduce Foreign Method (Εισαγωγή Ξένης Μεθόδου)

Χρησιμοποιούμε αυτή την τεχνική όταν μία κλάση server χρειάζεται μία επιπλέον μέθοδο, αλλά δεν μπορούμε να τροποποιήσουμε την κλάση.

> *Δημιουργούμε μία μέθοδο στην κλάση client η οποία έχει ως πρώτη παράμετρο ένα στιγμιότυπο της κλάσης server.*

```
Date newStart = new Date (previousEnd.getYear(),
                          previousEnd.getMonth(), previousEnd.getDate() + 1);
```

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
      return new Date (arg.getYear(),arg.getMonth(), arg.getDate() + 1);
}
```

5.7.1 Κίνητρο

Συμβαίνει αρκετά συχνά. Χρησιμοποιούμε μία κλάση η οποία μας προσφέρει πάρα πολλές υπηρεσίες. Υπάρχει όμως μία επιπλέον λειτουργία που θα θέλαμε να κάνει η κλάση, την οποία όμως δεν κάνει. Αν έχουμε πρόσβαση στην κλάση server μπορούμε να την τροποποιήσουμε και να προσθέσουμε την μέθοδο που θέλουμε. Αν όμως δεν έχουμε πρόσβαση, θα πρέπει να εισάγουμε την νέα λειτουργία στον client.

Αν χρησιμοποιούσαμε την μέθοδο που λείπει από τον server στην κλάση client μόνο μία φορά, τότε ο επιπλέον κώδικας δεν είναι κάτι σημαντικό και πιθανόν να μην χρειαζόταν στην κλάση server. Αν όμως χρησιμοποιούσαμε την μέθοδο αρκετές φορές θα πρέπει να επαναλαμβάνουμε τον κώδικα. Επειδή όπως έχουμε αναφέρει οι διπλοεγγραφές κώδικα θα πρέπει να εξαλείφονται, πρέπει να εξάγουμε αυτές τις διπλοεγγραφές σε μία μέθοδο.

5.7.2 Μηχανισμοί

- Δημιουργούμε μία μέθοδο στην κλάση client η οποία κάνει αυτό που θέλουμε.
    - Η μέθοδος δεν πρέπει να έχει πρόσβαση σε κανένα από τα χαρακτηριστικά της κλάσης client. Αν χρειάζεται κάποια από αυτά τότε τα περνάμε ως παραμέτρους.
- Δηλώνουμε ως πρώτη παράμετρο της μεθόδου ένα στιγμιότυπο της κλάσης server.
- Εισάγουμε ένα σχόλιο στην μέθοδο το οποίο θα δηλώνει ότι είναι ξένη μέθοδος και θα έπρεπε να βρίσκεται στον server.
    - Με αυτό τον τρόπο μπορούμε εύκολα να ψάξουμε για ξένες μεθόδους όταν σας δοθεί η ευκαιρία να τις μετακινήσουμε στον server.

5.7.3 Παράδειγμα

Έχουμε ένα κομμάτι κώδικα στο οποίο θέλουμε να δημιουργήσουμε μία νέα ημερομηνία με βάση μία προηγούμενη προσθέτοντας μία επιπλέον μέρα.

```
Date newStart = new Date (previousEnd.getYear(),
                          previousEnd.getMonth(), previousEnd.getDate() + 1);
```

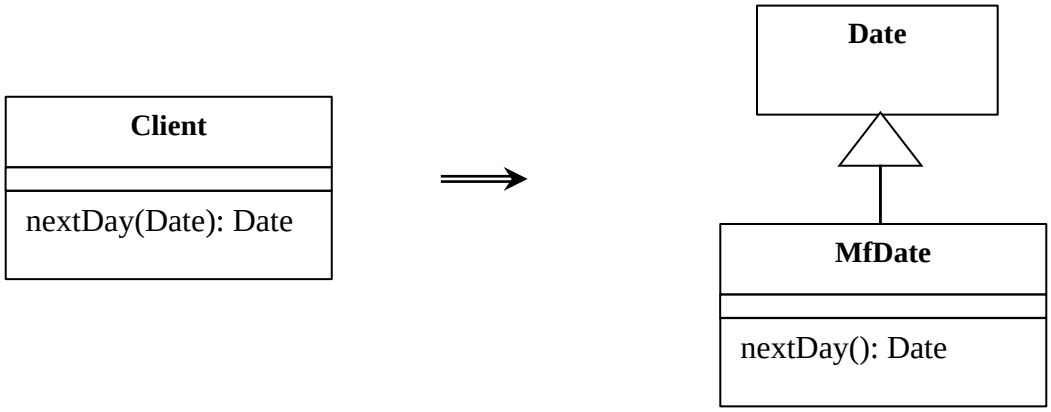
Μπορούμε να εξάγουμε το δεξιό μέρος της ανάθεσης σε μία νέα μέθοδο. Αυτή η μέθοδος είναι μία ξένη μέθοδος προς την κλάση Date.

```
Date newStart = nextDay(previousEnd);

private static Date nextDay(Date arg) {
    // foreign method, should be on date
    return new Date (arg.getYear(),arg.getMonth(), arg.getDate() + 1);
}
```

5.8 Introduce Local Extension (Εισαγωγή Τοπικής Επέκτασης)

Χρησιμοποιούμε αυτή την τεχνική όταν μία κλάση server χρειάζεται πολλές επιπλέον μεθόδους αλλά δεν μπορούμε να την τροποποιήσουμε.

Δημιουργούμε μία νέα κλάση η οποία περιέχει τις επιπλέον μεθόδους που χρειαζόμαστε. Κάνουμε αυτή την κλάση υποκλάση της αρχικής ή περιέχουμε την αρχική σε αυτήν (wrapper).



5.8.1 Κίνητρο

Συνήθως, οι δημιουργοί των κλάσεων δεν μπορούν να προβλέψουν όλες τις πιθανές μεθόδους που θέλουν να χρησιμοποιήσουμε. Αν μπορούμε να τροποποιήσουμε την κλάση, τότε μπορούμε απλά να προσθέσουμε μία να μέθοδο. Ωστόσο, συνήθως δεν έχουμε πρόσβαση σε αυτές. Αν χρειαζόμαστε μόνο μία ή δύο μεθόδους μπορούμε να χρησιμοποιήσουμε την Εισαγωγή Ξένης Μεθόδου. Όταν όμως οι μέθοδοι είναι περισσότερες τότε αυτό δεν είναι βολικό. Δημιουργώντας μία νέα κλάση που θα είναι υποκλάση ή θα περιέχει την αρχική λύνουμε αυτό το πρόβλημα.

Μία τοπική επέκταση είναι μία ξεχωριστή κλάση, αλλά αποτελεί έναν υπο-τύπο της κλάσης την οποία κληρονομεί. Αυτό σημαίνει ότι παρέχει όλες τις υπηρεσίες της αρχικής κλάσης και κάποιες ακόμα που προσθέτουμε εμείς. Αντί να χρησιμοποιούμε την αρχική κλάση, μπορούμε να αναφερόμαστε στην τοπική επέκτασή της και να χρησιμοποιούμε αυτή.

Χρησιμοποιώντας την τοπική επέκταση κρατάμε την αρχή που λέει ότι οι μέθοδοι και τα δεδομένα πρέπει να καλά δομημένα σε οντότητες. Αν βάζουμε κώδικα που θα έπρεπε να βρίσκεται στην επέκταση και σε άλλες κλάσεις καταλήγουμε να έχουμε πολύπλοκες κλάσεις.

5.8.2 Μηχανισμοί

- Δημιουργούμε μία κλάση επέκτασης της αρχικής είτε ως υποκλάση είτε ως wrapper.
- Προσθέτουμε δομητές μετατροπής στην κλάση επέκτασης. Οι δομητές θα παίρνουν ως παράμετρο το αρχικό αντικείμενο.
    - Η υποκλάση καλεί τον κατάλληλο δομητή της υπερκλάσης.
    - Η κλάση wrapper θέτει το αρχικό αντικείμενο που παίρνει ως παράμετρο σε ένα πεδίο αντιπροσώπευσης (delegate field).
- Προσθέτουμε τις νέες μεθόδους στην επέκταση.
- Αντικαθιστούμε τις κλήσεις προς την αρχική κλάση να καλούν την επέκτασή της.
- Μεταφέρουμε τις ξένες μεθόδους (αν υπάρχουν) μέσα στην κλάση επέκτασης.

5.8.3 Παράδειγμα: Με χρήση υποκλάσης

Αρχικά δημιουργούμε μία νέα κλάση η οποία κληρονομεί την κλάση Date.

```
class MfDateSub extends Date {
}
```

Έπειτα δημιουργούμε τον δομητή ο οποίος καλεί τον δομητή της υπερκλάσης.

```
public MfDateSub(String dateString) {
    super (dateString);
};
```

Τώρα χρειαζόμαστε και έναν δομητή μετατροπής ο οποίος παίρνει ως όρισμα τον αρχικό αντικείμενο.

```
public MfDateSub(Date arg) {
    super (arg.getTime());
}
```

Μπορούμε τώρα να προσθέσουμε νέες μεθόδους στην κλάση επέκτασης και να μεταφέρουμε τις ξένες μεθόδους μέσα σε αυτή.

```
client class
    ...
    private static Date nextDay(Date arg) {
        // foreign method, should be on date
        return new Date (arg.getYear(),arg.getMonth(), arg.getDate() + 1);
    }
}
```

Η παραπάνω ξένη μέθοδος μεταφέρεται μέσα στην κλάση επέκτασης:

```
class MfDate {
    ...
    Date nextDay() {
        return new Date (getYear(),getMonth(), getDate() + 1);
    }
}
```

5.8.4 Παράδειγμα: Με χρήση κλάσης wrapper

Ξεκινάμε δηλώνοντας την κλάση wrapper.

```
class mfDate {
    private Date _original;
}
```

Η κλάση wrapper περιέχει και ένα πεδίο αντιπροσώπευσης προς την αρχική κλάση. Με αυτή την προσέγγιση πρέπει να δηλώσουμε τους δομητές μας διαφορετικά. Οι απλοί δομητές υλοποιούνται με μία απλή αντιπροσώπευση.

```
public MfDateWrap(String dateString) {
    _original = new Date(dateString);
};
```

Ο δομητής μετατροπής τώρα απλά αναθέτει το αρχικό αντικείμενο στο πεδίο αντιπροσώπευσης.

```
public MfDateWrap(Date arg) {
    _original = arg;
}
```

Μετά ακολουθεί μία χρονοβόρα διαδικασία δημιουργίας μεθόδων αντιπροσώπευσης προς την αρχική κλάση όπως αυτές στο παράδειγμα.

```
public int getYear() {
    return _original.getYear();
}

public boolean equals(MfDateWrap arg) {
    return (toDate().equals(arg.toDate()));
}
```

Έπειτα, χρησιμοποιούμε ξανά την Μετακίνηση Μεθόδου για να μεταφέρουμε όλες τις ξένες μεθόδους μέσα στην νέα κλάση.

Ένα σημαντικό πρόβλημα με την χρήση των wrappers είναι όταν θέλουμε να χρησιμοποιήσουμε μεθόδους οι οποίες παίρνουν το αρχικό αντικείμενο ως παράμετρο. Επίσης όπως είδαμε όταν χρησιμοποιούμε wrappers πρέπει να δημιουργήσουμε μία μέθοδο αντιπροσώπευσης για κάθε μέθοδο της αρχικής κλάσης.

Βιβλιογραφία

1. Martin Fowler, Kent Beck, John Brant, William Opdyke, don Roberts, 1999. Refactoring: Improving the Design of Existing Code.

2. Kent.Beck, 1999. eXtreme Programming eXplained

3. SourceMaking teaching IT professionals, 2013. *Refactoring*. [online]. Available at: http://sourcemaking.com/refactoring.

4. Refactoring, 2013. [online]. Available at: http://www.refactoring.com/

5. Wikipedia, 2013. Code Refactoring. [online]. Available at: http://en.wikipedia.org/wiki/Code_refactoring