# ΑΥΤΟΜΑΤΗ ΑΝΑΚΑΤΑΣΚΕΥΗ ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΩΝ ΔΙΕΠΑΦΩΝ

Η
ΜΕΤΑΠΤΥΧΙΑΚΗ ΕΡΓΑΣΙΑ ΕΞΕΙΔΙΚΕΥΣΗΣ

Υποβάλλεται στην

ορισθείσα από την Γενική Συνέλευση Ειδικής Σύνθεσης
του Τμήματος Πληροφορικής
Εξεταστική Επιτροπή

από τον

Σπυρίδωνα Κρανά

ως μέρος των Υποχρεώσεων

για τη λήψη

του

ΜΕΤΑΠΤΥΧΙΑΚΟΥ ΔΙΠΛΩΜΑΤΟΣ ΣΤΗΝ ΠΛΗΡΟΦΟΡΙΚΗ

ΜΕ ΕΞΕΙΔΙΚΕΥΣΗ ΣΤΟ ΛΟΓΙΣΜΙΚΟ

Ιούνιος 2012

# ΕΥΧΑΡΙΣΤΙΕΣ

Με το πέρας της παρούσας διατριβής, θα ήθελα να εκφράσω τις ευχαριστίες μου στον επιβλέποντά, κ. Ζάρρα Απόστολο, Επίκουρο Καθηγητή του τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων, για την εποικοδομητική και αρμονική συνεργασία που είχαμε, καθώς επίσης και για το ενδιαφέρον που επέδειξε, καθ' όλη τη διάρκεια εκπόνησής της. Οι συμβουλές και η καθοδήγησή του, υπήρξαν καταλυτικές στην επιτυχή ολοκλήρωση της εργασίας. Επίσης, θα ήθελα να ευχαριστήσω τον κ. Παπαπέτρου Ευάγγελο και τον κ. Βασιλειάδη Παναγιώτη, Επίκουρους Καθηγητές του τμήματος Πληροφορικής του Πανεπιστημίου Ιωαννίνων και μέλη της εξεταστικής επιτροπής. Κλείνοντας, θα αποτελούσε παράλειψη, αν δεν ευχαριστούσα όλα τα αγαπημένα μου πρόσωπα, τα οποία με στήριξαν στην έως τώρα ακαδημαϊκή μου πορεία.

# ΠΕΡΙΕΧΟΜΕΝΑ


| | Σελ |
|---|---|
| ΕΥΧΑΡΙΣΤΙΕΣ | ii |
| ΠΕΡΙΕΧΟΜΕΝΑ | iii |
| ΕΥΡΕΤΗΡΙΟ ΠΙΝΑΚΩΝ | iv |
| ΕΥΡΕΤΗΡΙΟ ΣΧΗΜΑΤΩΝ | v |
| ΕΥΡΕΤΗΡΙΟ ΑΛΓΟΡΙΘΜΩΝ | vii |
| ΠΕΡΙΛΗΨΗ | viii |
| EXTENDED ABSTRACT IN ENGLISH | ix |
| ΚΕΦΑΛΑΙΟ 1. ΕΙΣΑΓΩΓΗ | 10 |
| 1.1 Εισαγωγή | 10 |
| 1.2 Παράδειγμα | 11 |
| 1.3 Συνεισφορά | 15 |
| 1.4 Δομή της διατριβής | 16 |
| ΚΕΦΑΛΑΙΟ 2. ΘΕΩΡΗΤΙΚΟ ΥΠΟΒΑΘΡΟ | 17 |
| 2.1 Ανακατασκευή λογισμικού | 17 |
| 2.2 Βασικές Σχεδιαστικές Αρχές | 20 |
| 2.3 Κακοτεχνίες κώδικα | 22 |
| 2.4 Τεχνικές ανακατασκευής κώδικα | 25 |
| 2.4.1 Κατασκευή Μεθόδων | 26 |
| 2.4.2 Μετακινήσεις Συστατικών | 27 |
| 2.4.3 Απλοποιήσεις Συνθηκών | 30 |
| 2.4.4 Απλοποιήσεις Μεθόδων | 32 |
| 2.4.5 Χειρισμός Γενίκευσης | 34 |
| 2.4.6 Οργάνωση Δεδομένων | 37 |
| 2.5 Σχετική έρευνα | 39 |
| ΚΕΦΑΛΑΙΟ 3. ΠΡΟΤΕΙΝΟΜΕΝΟΙ ΑΛΓΟΡΙΘΜΟΙ | 45 |
| 3.1 Εισαγωγή | 45 |
| 3.2 Clustering Based Approach (CBA) | 45 |
| 3.2.1 Θεωρητικό υπόβαθρο | 45 |
| 3.2.2 Αλγόριθμος | 48 |
| 3.3 Refactoring Based Approach (RBA) | 61 |
| 3.3.1 Θεωρητικό υπόβαθρο | 61 |
| 3.3.2 Αλγόριθμος | 64 |
| ΚΕΦΑΛΑΙΟ 4. ΕΡΓΑΛΕΙΟ ΑΥΤΟΜΑΤΗΣ ΑΝΑΚΑΤΑΣΚΕΥΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΩΝ ΔΙΕΠΑΦΩΝ | |
| 4.1 Αρχιτεκτονική | 69 |
| 4.2 Εγκατάσταση του εργαλείου | 78 |
| ΚΕΦΑΛΑΙΟ 5. ΠΕΙΡΑΜΑΤΑ | 81 |
| ΚΕΦΑΛΑΙΟ 6. ΣΥΜΠΕΡΑΣΜΑΤΑ | 89 |
| ΠΑΡΑΡΤΗΜΑ Α | 92 |
| ΠΑΡΑΡΤΗΜΑ Β | 125 |
| ΣΥΝΤΟΜΟ ΒΙΟΓΡΑΦΙΚΟ | 158 |

# ΕΥΡΕΤΗΡΙΟ ΠΙΝΑΚΩΝ

Πίνακας Σελ
Πίνακας 2.1 Κακοτεχνίες κώδικα. 22
Πίνακας 2.2 Τεχνικές ανακατασκευής: Κατασκευή Μεθόδων. 26
Πίνακας 2.3 Τεχνικές ανακατασκευής: Μετακίνηση Συστατικών. 28
Πίνακας 2.4 Τεχνικές ανακατασκευής: Απλοποίηση Συνθηκών. 30
Πίνακας 2.5 Τεχνικές ανακατασκευής: Απλοποίηση Μεθόδων. 32
Πίνακας 2.6 Τεχνικές ανακατασκευής: Χειρισμός Γενίκευσης. 35
Πίνακας 2.7 Τεχνικές ανακατασκευής: Οργάνωση Δεδομένων. 37

# ΕΥΡΕΤΗΡΙΟ ΣΧΗΜΑΤΩΝ

Σχήμα Σελ
Σχήμα 1.1 Αρχική σχεδίαση. 11
Σχήμα 1.2 Η σχεδίαση με εφαρμογή της ISP. 12
Σχήμα 1.3 Η σχεδίαση με εφαρμογή της κληρονομικότητας. 13
Σχήμα 3.1 Ιεραρχική Ενωτική Ομαδοποίηση. 45
Σχήμα 3.2 Κανόνας του κοντινότερου γείτονα. 47
Σχήμα 3.3 Κανόνας του μακρινότερου γείτονα. 47
Σχήμα 3.4 Κανόνας του μέσου όρου των αποστάσεων. 47
Σχήμα 3.5 Σχεδίαση μετά την εφαρμογή της CBA με το κριτήριο single linkage. 53
Σχήμα 3.6 Σχεδίαση μετά την εφαρμογή της CBA με το κριτήριο complete linkage. 57
Σχήμα 3.7 Παράδειγμα εφαρμογής της τεχνικής ανακατασκευής: Εξαγωγή Διεπαφής. 61
Σχήμα 3.8 Παράδειγμα εφαρμογής της τεχνικής ανακατασκευής: Εξαγωγή Υπερκλάσης. 63
Σχήμα 4.1 Αρχιτεκτονική του συστήματος αυτόματης ανακατασκευής προγραμματιστικών διεπαφών. 68
Σχήμα 4.2 Η γραφική διεπαφή του εργαλείου. 70
Σχήμα 4.3 Παράδειγμα επιλογής φακέλου για την αποθήκευση των αρχείων. 70
Σχήμα 4.4 Παράδειγμα απλού αρχείου .gml και η αντίστοιχη οπτικοποίηση. 72
Σχήμα 4.5 Η οπτικοποίηση του αρχείου εξόδου της CBA με το κριτήριο του μακρινότερου γείτονα που αφορά την κλάση Provider του Σχήματος 1.1. 72
Σχήμα 4.6 Η οπτικοποίηση του αρχείου εξόδου της RBA που αφορά την κλάση Provider του Σχήματος 1.1 73

Σχήμα 4.7 Επάνω: τα στατιστικά δεδομένα που παράγει η CBA με το κριτήριο του μακρινότερου γείτονα για το παράδειγμα της ενότητας 1.2. Κάτω: τα αντίστοιχα στατιστικά δεδομένα της RBA. 74
Σχήμα 4.8 Πηγαίος κώδικας της κλάσης Test 74
Σχήμα 4.9 Το αφηρημένο συντακτικό δένδρο για την κλάση Test .76
Σχήμα 4.10 Το plug in AST Viewer. 76
Σχήμα 4.11 Βήμα 1 της διαδικασίας εγκατάστασης του εργαλείου. 78
Σχήμα 4.12 Βήμα 2 της εγκατάστασης του εργαλείου. 78
Σχήμα 4.13 Βήμα 3 της εγκατάστασης του εργαλείου. 79
Σχήμα 4.14 Εκτέλεση του εργαλείου με είσοδο το παράδειγμα της ενότητας 1.2. 79
Σχήμα 5.1 Μέση τιμή ACD για το JUnit 85
Σχήμα 5.2 Μέση τιμή CBO για το JUnit. 85
Σχήμα 5.3 Μέση τιμή Max Tree Height για το JUnit. 85
Σχήμα 5.4 Μέση τιμή Tree Height για το JUnit. 86
Σχήμα 5.5 Total Interfaces για το JUnit. 86
Σχήμα 5.6 Μέση τιμή Total Interfaces για το JUnit. 86
Σχήμα 5.7 Μέση τιμή ACD για το Concordion. 87
Σχήμα 5.8 Μέση τιμή CBO για το Concordion. 87
Σχήμα 5.9 Μέση τιμή Max Tree Height για το Concordion. 87
Σχήμα 5.10 Μέση τιμή Tree Height για το Concordion. 88
Σχήμα 5.11 Total Interfaces για το Concordion. 88
Σχήμα 5.12 Μέση τιμή Total Interfaces για το Concordion. 88

# ΕΥΡΕΤΗΡΙΟ ΑΛΓΟΡΙΘΜΩΝ


| Αλγόριθμος | Σελ |
|---|---|
| Αλγόριθμος 1. Πρώτη φάση της CBA. | 52 |
| Αλγόριθμος 2. Δεύτερη φάση της CBA. | 60 |
| Αλγόριθμος 3. Ο αλγόριθμος της RBA. | 67 |

# ΠΕΡΙΛΗΨΗ

Σπυρίδων Κρανάς του Ιωάννη και της Κωνσταντίνας. MSc, Τμήμα Πληροφορικής, Πανεπιστήμιο Ιωαννίνων, Ιούνιος, 2012. Αυτόματη Ανακατασκευή Προγραμματιστικών Διεπαφών. Επιβλέποντας: Απόστολος Ζάρρας.

Στην παρούσα εργασία, ασχολούμαστε με τη βελτίωση της ποιότητας του λογισμικού, μέσω της αυτοματοποιημένης ανακατασκευής προγραμματιστικών διεπαφών (interfaces) για αντικειμενοστραφή προγράμματα. Συγκεκριμένα, χρησιμοποιείται η Αρχή Διαχωρισμού Διεπαφών (Interface Segregation Principle – ISP). Η σχεδιαστική αυτή αρχή εφαρμόζεται όταν διαφορετικά υποσύνολα μεθόδων της διεπαφής (interface) μιας κλάσης (provider), αφορούν διαφορετικές κλάσεις (clients) που χρησιμοποιούν τα υποσύνολα αυτά. Η ISP προτείνει τον ορισμό επιμέρους διεπαφών της κλάσης provider, που να περιλαμβάνουν αντίστοιχα υποσύνολά της. Με τον τρόπο αυτό, ο κώδικας των clients εξαρτάται μόνο από τις αντίστοιχες διεπαφές που χρησιμοποιεί. Στην εργασία, προτείνουμε δύο προσεγγίσεις για την βελτίωση της ποιότητας του λογισμικού που βασίζονται στην ISP. Η πρώτη, βασίζεται σε θεμελιώδεις τεχνικές αφαίρεσης για την ανακατασκευή λογισμικού (Refactoring Based Approach - RBA) και αποτελείται από μια επαναληπτική διαδικασία, δύο φάσεων. Η πρώτη φάση περιλαμβάνει την εξαγωγή των διεπαφών (Extract Interfaces) και η επόμενη, την εξαγωγή υπερκλάσεων (Extract Superclass). Η δεύτερη προσέγγιση, εδράζεται σε παραδοσιακές τεχνικές ιεραρχικής ομαδοποίησης (Clustering Based Approach – CBA). Επιπλέον, προχωρήσαμε στην αυτοματοποίηση των δύο προσεγγίσεων και την υλοποίηση ενός εργαλείου για τον σκοπό αυτό. Για να εκτιμηθεί η απόδοση των προτεινόμενων προσεγγίσεων, εξετάσθηκε η εφαρμογή τους σε δύο περιβάλλοντα ανοιχτού λογισμικού, το JUnit και το Concordion.

# EXTENDED ABSTRACT IN ENGLISH

Kranas, Spyridon, MSc, Computer Science Department, University of Ioannina, Greece. June, 2012. Automatic Refactoring of Program Interfaces. Thesis Supervisor: Apostolos Zarras.

A software rarely remains unchanged after its delivery. Usually what happens is that, at some point, developers will need to modify the software either to meet new customer needs or to respond to technological developments. But these interventions in the software code, apart from creating additional complexity, they will probably cause the initial design that the software was based on, to decline. This leads to difficult understanding of the code and that increases the cost of software maintenance. There is, therefore, the problem of maintaining the quality of the software. To address the above problem, there is a need for techniques that reduce the complexity and software maintenance costs, by improving its internal structure. The scientific field that deals with the issue, is called software refactoring, i.e. the process that transforms a software in such a way as not to alter its behavior to the external environment, while, at the same time, improving its internal structure. In this thesis, we deal with improving software quality through automatic programming interfaces' rebuild for object-oriented programs. Specifically, the Interface Segregation Principle (ISP) is employed. This design principle is applied when different subsets of a class' (provider) methods, relate to different classes (clients) that use these subsets. ISP suggests the definition of individual interfaces of the provider class, that include the respective subsets. In this way, the clients' code depends only on the respective interfaces it uses. In this thesis, we propose two approaches to improve the quality of software which are based on ISP. The first one, is based on fundamental abstraction refactoring techniques (Refactoring Based Approach - RBA) and consists of an iterative process with two phases. The first phase, involves extracting interfaces (Extract Interfaces technique) and the second, extracting superclasses (Extract Superclass technique). The second approach is based on traditional techniques of hierarchical clustering (Clustering Based Approach - CBA). Furthermore, we automate the two approaches and implement a tool for this purpose. To assess the performance of the proposed approaches, we examine their application in two open-source environments, JUnit and Concordion.

# ΚΕΦΑΛΑΙΟ 1. ΕΙΣΑΓΩΓΗ

1.1 Εισαγωγή
1.2 Παράδειγμα
1.3 Συνεισφορά
1.4 Δομή της διατριβής

## 1.1 Εισαγωγή

Στη σημερινή εποχή, σπάνια ένα λογισμικό παραμένει αμετάβλητο μετά την παράδοσή του. Συνήθως, εκείνο που συμβαίνει είναι ότι, κάποια στιγμή, οι προγραμματιστές θα χρειαστεί να τροποποιήσουν το ήδη υπάρχον λογισμικό είτε για να ικανοποιήσουν νέες ανάγκες των πελατών είτε για να ανταποκριθούν στις τεχνολογικές εξελίξεις. Οι επεμβάσεις όμως αυτές στον κώδικα του λογισμικού, εκτός του ότι δημιουργούν προστιθέμενη πολυπλοκότητα, πιθανότατα θα έχουν ως αποτέλεσμα, η αρχική σχεδίαση πάνω στην οποία βασίστηκε η ανάπτυξή του, να παρακμάζει. Το γεγονός αυτό οδηγεί στο να δυσχεραίνει η κατανόηση του κώδικα και κατά συνέπεια, να αυξάνεται το κόστος συντήρησης του λογισμικού. Παρατηρείται επομένως, το πρόβλημα της διατήρησης της ποιότητας του λογισμικού. Για να αντιμετωπιστεί το παραπάνω πρόβλημα, υπάρχει η ανάγκη για τεχνικές που μειώνουν την πολυπλοκότητα και το κόστος συντήρησης του λογισμικού, βελτιώνοντας την εσωτερική δομή του. Το επιστημονικό πεδίο που ασχολείται με το ζήτημα, ονομάζεται ανακατασκευή λογισμικού *(software refactoring)*, δηλαδή η διαδικασία η οποία μετασχηματίζει ένα λογισμικό με τέτοιον τρόπο, ώστε να μη μεταβάλλει την συμπεριφορά του στο εξωτερικό περιβάλλον του και ταυτόχρονα να βελτιώνει την εσωτερική δομή του. Στην παρούσα εργασία, ασχολούμαστε με τη βελτίωση της ποιότητας του λογισμικού, μέσω της αυτοματοποιημένης

ανακατασκευής προγραμματιστικών διεπαφών (interfaces) για αντικειμενοστραφή προγράμματα. Συγκεκριμένα, χρησιμοποιείται η Αρχή Διαχωρισμού Διεπαφών (Interface Segregation Principle – ISP). Η σχεδιαστική αυτή αρχή εφαρμόζεται όταν διαφορετικά υποσύνολα μεθόδων της διεπαφής (interface) μιας κλάσης (provider), αφορούν διαφορετικές κλάσεις (clients) που χρησιμοποιούν τα υποσύνολα αυτά. Διευκρινίζεται ότι με τον όρο διεπαφή εδώ, εννοούμε οτιδήποτε ορίζεται ως public στην κλάση provider και με τον όρο client, κώδικα του συστήματος που χρησιμοποιεί τις public μεθόδους της κλάσης provider. Η ISP προτείνει τον ορισμό επιμέρους διεπαφών της κλάσης provider, που να περιλαμβάνουν αντίστοιχα υποσύνολά της. Με τον τρόπο αυτό, ο κώδικας των clients εξαρτάται μόνο από τις αντίστοιχες διεπαφές που χρησιμοποιεί. Η βασική ιδέα και λειτουργία της ISP αναλύεται στην ενότητα 1.2.

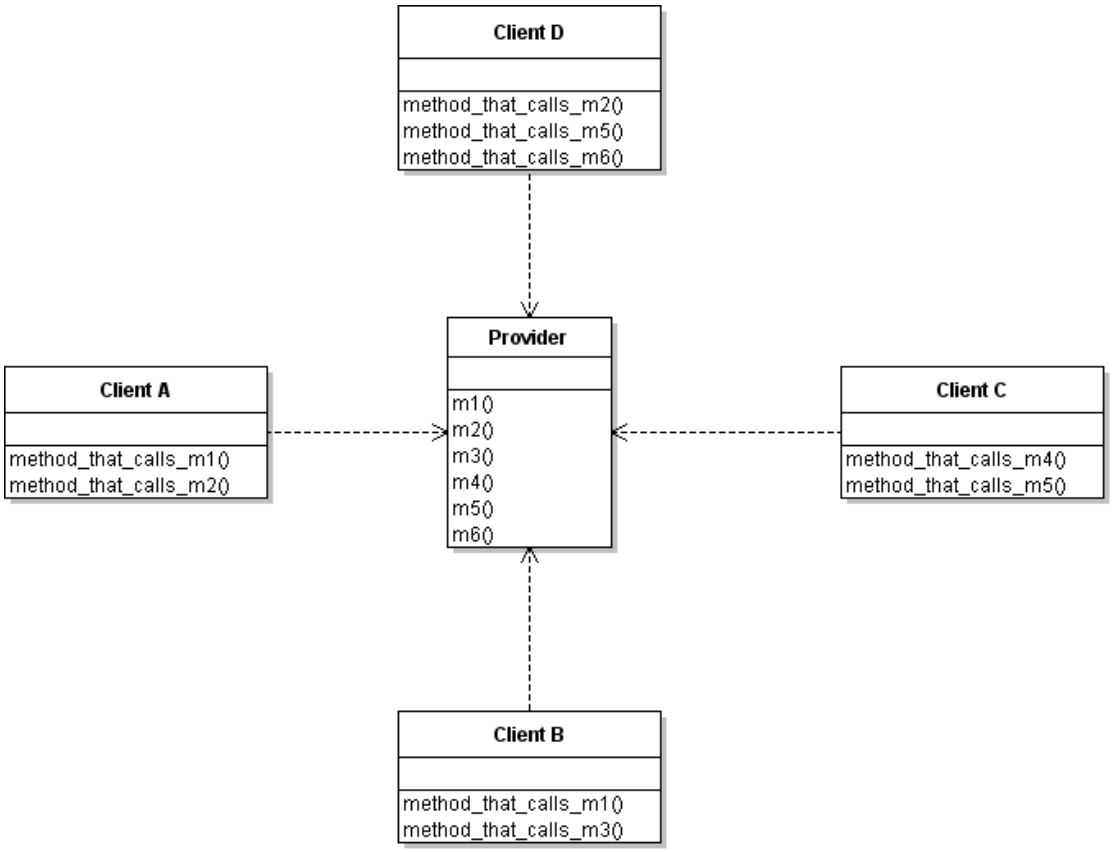
### 1.2 Παράδειγμα

Για να γίνει πιο κατανοητή η δομή και λειτουργία γύρω από την ISP, στην οποία βασίζεται η εργασία, θα εξηγήσουμε τη συλλογιστική πορεία μέσα από ένα παράδειγμα.

Η εφαρμογή της σχεδιαστικής αυτής αρχής προτείνεται, όταν μια κλάση (provider) επιτελεί διαφορετικούς ρόλους, ανάλογα με τις κλάσεις οι οποίες την χρησιμοποιούν (clients). Κάθε client χρησιμοποιεί ένα υποσύνολο των μεθόδων που προσφέρει η κλάση provider. Δηλαδή, η κλάση provider εξυπηρετεί έναν διαφορετικό ρόλο για κάθε κλάση client. Η ISP προτείνει τον ορισμό επιμέρους διεπαφών της κλάσης provider, μία για κάθε διαφορετικό ρόλο της. Με τον τρόπο αυτό, κάθε client θα εξαρτάται μόνο από την αντίστοιχη διεπαφή που χρησιμοποιεί.

Έστω η σχεδίαση του Σχήματος 1.1. Στη σχεδίαση αυτή, υπάρχουν πέντε (5) κλάσεις: μια κλάση provider και τέσσερις (4) κλάσεις clients. Η κλάση Provider περιέχει έξι (6) μεθόδους: τις m1(), m2(), m3(), m4(), m5() και m6(). Η κλάση Client A περιέχει δύο (2) μεθόδους, τις method_that_uses_m1() και method_that_uses_m2(), οι οποίες χρησιμοποιούν τις μεθόδους m1() και m2() της κλάσης Provider, αντίστοιχα. Η κλάση Client B περιέχει, επίσης δύο (2) μεθόδους, τις method_that_uses_m1() και

method_that_uses_m3(), οι οποίες χρησιμοποιούν τις μεθόδους m1() και m3() της κλάσης Provider, αντίστοιχα. Η κλάση Client C περιέχει, επίσης δύο (2) μεθόδους, τις method_that_uses_m4() και method_that_uses_m5(), οι οποίες χρησιμοποιούν τις μεθόδους m4() και m5() της κλάσης Provider, αντίστοιχα. Τέλος, η κλάση Client D, περιέχει τρεις (3) μεθόδους, τις method_that_uses_m2(), τη method_that_uses_m5() και τη method_that_uses_m6(), οι οποίες χρησιμοποιούν τις μεθόδους m2(), m5() και m6() της κλάσης Provider, αντίστοιχα.

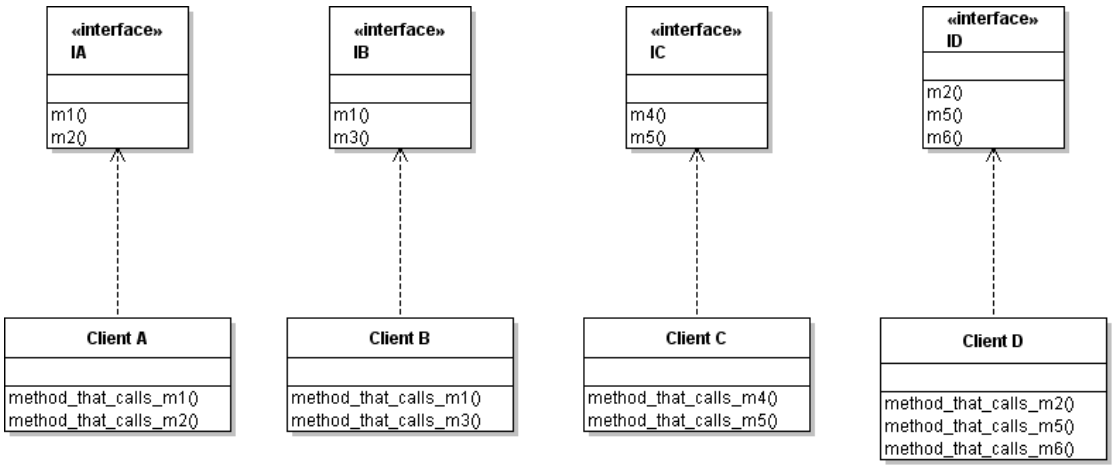



Σχήμα 1.1 Αρχική σχεδίαση.

Έστω ότι με την πάροδο του χρόνου, εξαιτίας μιας αλλαγής στις απαιτήσεις του συστήματος ή της προσθήκης μιας νέας δυνατότητας, η λειτουργία της μεθόδου m6() της κλάσης Provider, πρέπει να τροποποιηθεί. Τότε, σύμφωνα με το μοντέλο εξαρτήσεων του Σχήματος 1.1, όλες οι υπόλοιπες κλάσεις του συστήματος θα πρέπει να ελεγχθούν για ενδεχόμενες αλλαγές που θα πρέπει να υποστούν. Παρόλα αυτά,

στο παράδειγμά μας, με βάση την υλοποίηση του συστήματος, μόνο η κλάση Client D θα πρέπει να ελεγχθεί, αφού είναι η μόνη που χρησιμοποιεί τη μέθοδο που τροποποιήθηκε. Ο άσκοπος έλεγχος κλάσεων, τις οποίες οι αλλαγές δεν αφορούν, κοστίζει σε χρόνο και πόρους. Όπως γίνεται αντιληπτό, ο έλεγχος αυτός επαναλαμβάνεται κάθε φορά που συντελούνται αλλαγές στο σύστημα, με αποτέλεσμα το κόστος να αυξάνεται.

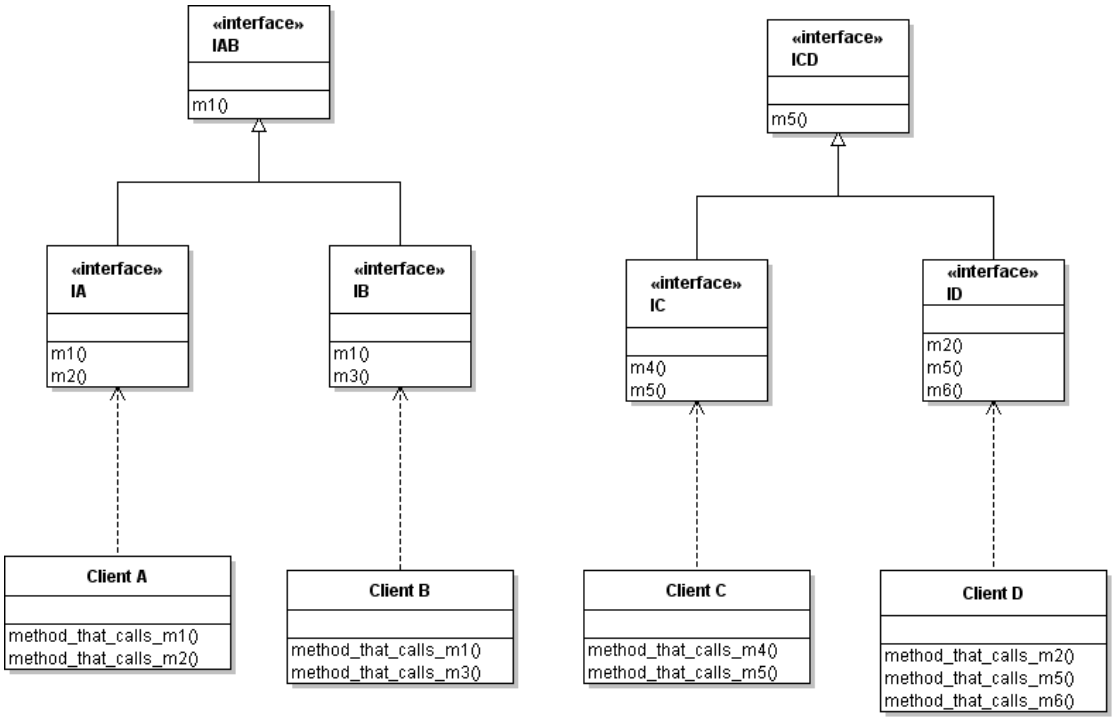
Με μια καλύτερη σχεδίαση του συστήματος θα μπορούσε να αποφεύγεται ο περιττός και δαπανηρός έλεγχος. Η σχεδίαση που προκύπτει με βάση την ISP φαίνεται στο Σχήμα 1.2.




Σχήμα 1.2 Η σχεδίαση με εφαρμογή της ISP.

Σύμφωνα με τη σχεδίαση που προκύπτει, μετά τις αλλαγές στην υλοποίηση της μεθόδου m6() της κλάσης Provider, μόνο η κλάση Client D θα πρέπει να ελεγχθεί εάν και κατά πόσο επηρεάζεται η υλοποίηση της μεθόδου της, method_that_uses_m6(), αφού πλέον είναι η μόνη κλάση που εξαρτάται από διεπαφή που περιέχει τη μέθοδο m6(). Επομένως, οι συνέπειες μιας αλλαγής στην κλάση Provider, δηλαδή το ποιοι clients επηρεάζονται, μπορούν να εκτιμηθούν εύκολα με ένα γρήγορο έλεγχο στο μοντέλο σχεδίασης του λογισμικού μας.

Παρόλα αυτά, ενώ λύσαμε το πρόβλημα που προέκυπτε αρχικά, παρατηρούμε ότι υπάρχει επανάληψη των μεθόδων ανάμεσα στις διεπαφές που δημιουργούνται. Για να μειώσουμε, όσο είναι δυνατόν, την επανάληψη αυτή, θα μπορούσαμε να χρησιμοποιούμε την ιδέα της κληρονομικότητας στις διεπαφές. Η κληρονομικότητα εφαρμόζεται όταν δύο ή περισσότερες κλάσεις έχουν παρόμοια πεδία ή/και μεθόδους και προτείνει τον ορισμό μιας βασικής κλάσης και μετακίνηση των κοινών πεδίων ή/και μεθόδων σε αυτήν. Παρατηρούμε, ότι οι διεπαφές IA και IB έχουν ως κοινή μέθοδο τη m1(), επομένως μπορούμε να ορίσουμε μια νέα διεπαφή στην οποία μετακινούμε τη μέθοδο m1(). Κατ' αντιστοιχία, παρατηρούμε ότι οι διεπαφές IC και ID έχουν ως κοινή μέθοδο τη m5(), άρα μπορούμε να ορίσουμε μια νέα διεπαφή στην οποία μετακινούμε τη μέθοδο m5(). Με τον τρόπο αυτό, καταλήγουμε στη σχεδίαση που φαίνεται στο σχήμα 1.3.




Σχήμα 1.3 Η σχεδίαση με εφαρμογή της κληρονομικότητας.

### 1.3 Συνεισφορά

Αν και τα πλεονεκτήματα της ανακατασκευής λογισμικού είναι αναμφισβήτητα, στην πράξη δεν εφαρμόζεται τόσο συχνά, όσο θα ήταν αναμενόμενο. Σύμφωνα με τους [6] υπάρχει μια σειρά λόγων που εξηγούν το φαινόμενο αυτό. Ανάμεσά τους, συγκαταλέγεται η ανάγκη για την προσθήκη επιπλέον χαρακτηριστικών στο λογισμικό κατά τη διάρκεια ανάπτυξής τους, που σε συνδυασμό με την χρονική πίεση για την παράδοσή του, κάνουν την ανακατασκευή να φαντάζει ως δευτερεύουσας σημασίας. Επιπλέον, ο φόβος ότι εφαρμόζοντας μια τεχνική ανακατασκευής στο λογισμικό, ελλοχεύει ο κίνδυνος να καταστούν μη λειτουργικές ορισμένες λειτουργίες ζωτικής σημασίας. Παρατηρούμε λοιπόν ότι τόσο οι επικεφαλής των ομάδων σχεδίασης, όσο και οι προγραμματιστές, εμφανίζονται διστακτικοί στην εφαρμογή ανακατασκευής του λογισμικού χειροκίνητα. Για το λόγο αυτό, θεωρήσαμε ότι θα ήταν πιο ωφέλιμο να στραφούμε σε μια αυτόματη διαδικασία ανακατασκευής. Ο συνολικός στόχος της διατριβής είναι να παρέχουμε μια αυτοματοποιημένη λύση στο πρόβλημα που περιγράφηκε στην ενότητα 1.1.

Στην παρούσα εργασία προτείνουμε δύο προσεγγίσεις για την βελτίωση της ποιότητας του λογισμικού που βασίζονται στην ISP. Η πρώτη, βασίζεται σε θεμελιώδεις τεχνικές αφαίρεσης για την ανακατασκευή λογισμικού (Refactoring Based Approach - RBA) και αποτελείται από μια επαναληπτική διαδικασία, δύο φάσεων. Η πρώτη φάση περιλαμβάνει την εξαγωγή των διεπαφών (Extract Interfaces) και η επόμενη, την εξαγωγή υπερκλάσεων (Extract Superclass). Η δεύτερη προσέγγιση, εδράζεται σε παραδοσιακές τεχνικές ιεραρχικής ομαδοποίησης (Clustering Based Approach – CBA). Επιπλέον, προχωρήσαμε στην αυτοματοποίηση των δύο τεχνικών και την υλοποίηση ενός εργαλείου για τον σκοπό αυτό. Το εργαλείο προσφέρεται ως επιπρόσθετη λειτουργία (plug-in) στο ολοκληρωμένο περιβάλλον ανάπτυξης (Integrated Development Environment - IDE), Eclipse. Τέλος, εντάξαμε στο εν λόγω εργαλείο τη δυνατότητα παραγωγής μετρήσιμων δεδομένων για την αξιολόγηση της ποιότητας ανακατασκευής των δύο προσεγγίσεων.

### 1.4 Δομή της διατριβής

Η εργασία αποτελείται από, συνολικά, έξι (6) Κεφάλαια, από τα οποία το πρώτο είναι η Εισαγωγή. Στο Κεφάλαιο 2 παρουσιάζεται το θεωρητικό υπόβαθρο. Συγκεκριμένα, γίνεται μια εισαγωγή στην ανακατασκευή λογισμικού και στις βασικές σχεδιαστικές αρχές. Ακόμη, παρουσιάζονται οι πιο συνηθισμένες κακοτεχνίες κώδικα αλλά και οι τρόποι επίλυσής τους, δηλαδή τεχνικές ανακατασκευής κώδικα. Τέλος, παρατίθεται η σχετική έρευνα που μελετήθηκε. Στο Κεφάλαιο 3 αναλύονται οι προτεινόμενες προσεγγίσεις οι οποίες είναι η Clustering Based Approach (CBA) και η Refactoring Based Approach (RBA). Στο Κεφάλαιο 4 γίνεται λόγος για το εργαλείο αυτόματης ανακατασκευής προγραμματιστικών διεπαφών που υλοποιήθηκε για την εφαρμογή των προσεγγίσεων. Το Κεφάλαιο 5 περιέχει τα αποτελέσματα και τις τιμές των πειραμάτων που πραγματοποιήθηκαν στις μελέτες περίπτωσης. Κλείνοντας, στο Κεφάλαιο 6 συνοψίζονται τα συμπεράσματα που απορρέουν από τις δύο προσεγγίσεις. Οι αναφορές και τα παραρτήματα αναγράφονται στο τέλος της εργασίας.

# ΚΕΦΑΛΑΙΟ 2. ΘΕΩΡΗΤΙΚΟ ΥΠΟΒΑΘΡΟ

---

2,1 Ανακατασκευή λογισμικού
2.2 Βασικές σχεδιαστικές αρχές
2.3 Κακοτεχνίες κώδικα
2.4 Τεχνικές ανακατασκευής κώδικα
2.5 Σχετική έρευνα

---

## 2.1 Ανακατασκευή λογισμικού

Όπως αναφέρει η [4], η ανακατασκευή λογισμικού ορίζεται ως η διαδικασία τροποποίησης του λογισμικού, έτσι ώστε μετά το τέλος της να βελτιώνεται η εσωτερική δομή και ταυτόχρονα να μην μεταβάλλεται η συμπεριφορά του προς το εξωτερικό περιβάλλον. Στην ουσία λοιπόν, όταν εφαρμόζεται η ανακατασκευή λογισμικού, βελτιώνεται η σχεδίαση του προγράμματος αφού έχει ολοκληρωθεί η υλοποίησή του. Κάτι τέτοιο ίσως φαίνεται ασυνήθιστο, αφού τις περισσότερες φορές προηγείται η διαδικασία σχεδίασης του συστήματος, το οποίο θα υλοποιηθεί και ύστερα ακολουθεί η διαδικασία υλοποίησης. Παρόλα αυτά, με την πάροδο του χρόνου ο κώδικας των προγραμμάτων αλλάζει και σταδιακά, η σχεδίαση δεν θα αντικατοπτρίζει τη δομή και τις λειτουργίες που επιτελεί ο κώδικας. Η συντήρησή του λογισμικού θα έχει μεγαλύτερο κόστος, η πολυπλοκότητα θα αυξηθεί, πιθανότατα διαφορετικά κομμάτια κώδικα θα επαναλαμβάνονται ή θα εξυπηρετούν την ίδια λειτουργία και η διόρθωση σφαλμάτων θα γίνει πιο επίπονη διαδικασία. Η ανακατασκευή έχει ακριβώς την αντίθετη λογική. Εφαρμόζοντάς την σε μια σχεδίαση χαμηλής ποιότητας, μπορεί να μετατρέψει τον κώδικα έτσι ώστε να αντιστοιχεί πλέον σε μια πολύ υψηλότερης ποιότητας σχεδίαση. Το γεγονός αυτό, φυσικά, δεν αναιρεί

την ανάγκη αρχικής σχεδίασης του συστήματος. Απλώς, η διαδικασία ανακατασκευής λογισμικού επιτρέπει να πραγματοποιούνται οι αλλαγές που απαιτούνται στον κώδικα, και κατ' επέκταση στη σχεδίαση, με πολύ μικρότερο κόστος σε σχέση με την περίπτωση κατά την οποία θα πραγματοποιούνταν οι ίδιες αλλαγές, χωρίς να έχει προηγηθεί η διαδικασία αυτή. Θα πρέπει να σημειωθεί ότι η ανακατασκευή λογισμικού δεν εξαντλείται σε μια τυπική «τακτοποίηση» του κώδικα, διότι παρέχει τυποποιημένες και αποδοτικές τεχνικές βελτίωσης του κώδικα.

Με την ανακατασκευή λογισμικού πετυχαίνουμε τα εξής:

- Βελτίωση της σχεδίασης του συστήματος.
- Βελτίωση της αναγνωσιμότητας των προγραμμάτων.
- Ευκολότερος εντοπισμός προγραμματιστικών λαθών.
- Ταχύτερη ανάπτυξη λογισμικού.

Χωρίς ανακατασκευή, η σχεδίαση του λογισμικού θα φθίνει με τον καιρό, όσο οι προγραμματιστές θα αλλάζουν τον ήδη υπάρχοντα κώδικα για να πετύχουν βραχυπρόθεσμους στόχους, χωρίς να έχουν μια γενικότερη αντίληψη των συνεπειών που θα επιφέρουν οι αλλαγές στη σχεδίαση, με αποτέλεσμα να αλλοιώνεται η δομή της. Αναμενόμενα, καθίσταται δυσδιάκριτη η σχέση μεταξύ σχεδίασης και κώδικα. Και όσο πιο δυσδιάκριτη είναι αυτή η σχέση, τόσο πιο δύσκολο είναι να για τους προγραμματιστές να συντηρήσουν το σύστημα και τόσο πιο ραγδαία θα είναι η επιδείνωσή της ποιότητάς του. Η ανακατασκευή επεμβαίνει, επιτελώντας κάθε φορά μικρές βελτιώσεις. Συχνή εφαρμογή τεχνικών ανακατασκευής βοηθά ώστε να διατηρηθεί η ποιότητα σε υψηλό επίπεδο.

Επιπρόσθετα, μια σχεδίαση χαμηλής ποιότητας, συνήθως, μεταφράζεται σε κώδικα, μεγάλο σε έκταση, που εκτελεί τις ίδιες λειτουργίες, επειδή ακριβώς επιτελεί τις ίδιες διαδικασίες σε διαφορετικά σημεία του προγράμματος. Συνεπώς, μια σημαντική πτυχή στη βελτίωση της σχεδίασης αποτελεί η εξάλειψη επαναλαμβανόμενου κώδικα. Η σημασία της βελτίωσης αυτής, έγκειται στις μελλοντικές επεκτάσεις ή τροποποιήσεις του συστήματος. Η μείωση του κώδικα μπορεί να μην επιφέρει αισθητά αποτελέσματα στην ταχύτητα του συστήματος, εντούτοις, θα επιφέρει

σημαντική βελτίωση στη συντηρησιμότητά του. Είναι προφανές ότι όσο περισσότερος κώδικας υπάρχει, τόσο πιο δύσκολα αλλάζει σωστά. Εξαλείφοντας όμως, τον επαναλαμβανόμενο κώδικα, είναι βέβαιο ότι κάθε λειτουργία εμφανίζεται σε ένα μόνο σημείο, δείγμα ποιοτικής σχεδίασης.

Μια άλλη σημαντική παράμετρος που τείνουν να ξεχνούν οι προγραμματιστές, είναι ότι οι αλλαγές που θα πρέπει να γίνουν σε ένα λογισμικό, αργά ή γρήγορα, θα υλοποιηθούν από κάποιον τρίτο προγραμματιστή. Για να είναι όμως σε θέση ο τελευταίος να προβεί στις απαιτούμενες αλλαγές, θα πρέπει πρώτα από όλα να μελετήσει τον κώδικα και να κατανοήσει τη λειτουργία του. Το πρόβλημα προκύπτει από το γεγονός ότι εκείνο που συμβαίνει συνήθως κατά την ανάπτυξη του λογισμικού, είναι ότι οι προγραμματιστές ενδιαφέρονται πρωτίστως για τη λειτουργικότητα του κώδικα και όχι τόσο πολύ για την αναγνωσιμότητά και την εύκολη κατανόηση του. Η ανακατασκευή του λογισμικού βοηθάει στην βελτίωση της αναγνωσιμότητας του κώδικα. Η εφαρμογή λίγων μόνο τεχνικών ανακατασκευής μπορεί να έχει ως αποτέλεσμα κώδικα πολύ πιο ευανάγνωστο.

Η βελτίωση της αναγνωσιμότητας του κώδικα, συνεισφέρει στον γρηγορότερο εντοπισμό προγραμματιστικών λαθών. Έχοντας εφαρμόσει κάποιος ανακατασκευή λογισμικού, έχει βαθιά κατανόηση της σχεδίασης και γνώση του κώδικα, εργαλεία που μπορεί να χρησιμοποιήσει στον εντοπισμό και διόρθωση σφαλμάτων.

Όλα τα προηγούμενα σημεία, συντελούν στην αύξηση της ταχύτητας ανάπτυξης του λογισμικού. Η μείωση των σφαλμάτων και η βελτίωση στη σχεδίαση και την αναγνωσιμότητα του κώδικα βελτιώνουν την ποιότητα του λογισμικού, χωρίς να μειώνουν την ταχύτητα ανάπτυξής του. Χωρίς σχεδίαση υψηλής ποιότητας, η πρόοδος στην ανάπτυξη του κώδικα μπορεί να είναι πρόσκαιρα σημαντική, ωστόσο, σύντομα, ο εντοπισμός σφαλμάτων και η διόρθωσή τους, που θα προκύπτουν εξαιτίας της σχεδίασης χαμηλού επιπέδου, θα την επιβραδύνει. Η προσθήκη νέων λειτουργιών θα καθυστερεί, όσο οι προγραμματιστές θα προσπαθούν να κατανοήσουν τον πολύπλοκο, επομένως δυσνόητο κώδικα και να εξαλείψουν τα σημεία όπου επαναλαμβάνεται.

Συγκεντρωτικά, η διαδικασία της ανακατασκευής λογισμικού αποτελείται από τα εξής διακριτά στάδια, όπως αναφέρει η [9]:

- Αναγνώριση των σημείων του κώδικα τα οποία χρήζουν εφαρμογή ανακατασκευής.
- Προσδιορισμός των τεχνικών ανακατασκευής που θα εφαρμοστούν στα σημεία που έχουν καθοριστεί.
- Παροχή εγγύησης ότι η συμπεριφορά του συστήματος παραμένει αμετάβλητη ύστερα από την εφαρμογή της ανακατασκευής.
- Εκτίμηση της αποτελεσματικότητας της ανακατασκευής βασισμένη σε ποιοτικά χαρακτηριστικά του λογισμικού όπως η πολυπλοκότητα, η κατανόηση και η συντηρησιμότητά του.
- Διατήρηση της συμβατότητας μεταξύ του ανακατασκευασμένου λογισμικού και των συστατικών με τα οποία συσχετίζεται, όπως τα έγγραφα τεκμηρίωσης, η αρχιτεκτονική και τα έγγραφα καθορισμού και περιγραφής απαιτήσεων.

## 2.2 Βασικές Σχεδιαστικές Αρχές

Στον αντικειμενοστραφή προγραμματισμό, υπάρχουν πέντε βασικές αρχές σχεδίασης (Object Oriented Design Principles) ενός συστήματος [8]. Στόχος των αρχών είναι, όταν εφαρμόζονται να δίνουν τη δυνατότητα στους προγραμματιστές και τους σχεδιαστές να υλοποιούν και να σχεδιάζουν συστήματα που είναι εύκολα στη συντήρηση και την επεκτασιμότητα. Ουσιαστικά, πρόκειται για κατευθυντήριες γραμμές των οποίων η χρησιμοποίηση, σε συνδυασμό με τις τεχνικές ανακατασκευής, εξυπηρετεί στην διόρθωση κακοτεχνιών κώδικα.

Η Αρχή Μοναδική Αρμοδιότητας (Single Responsibility Principle - SRP) προτείνει ότι κάθε κλάση ενός συστήματος πρέπει να έχει μία και μοναδική αρμοδιότητα, δηλαδή να εξυπηρετεί ένα μοναδικό σκοπό. Στην πράξη κάτι τέτοιο είναι δύσκολο να επιτευχθεί, καθώς τις περισσότερες φορές ανατίθενται περισσότερες από μια

αρμοδιότητες σε κάποια κλάση. Στην περίπτωση αυτή, συστήνεται η διάσπαση της κάθε αρμοδιότητας σε ξεχωριστή κλάση βάση μετρικών συνεκτικότητας.

Η Αρχή Αντικατάστασης της Liskov (Liskov Substitution Principle - LSP) απαντά στο ερώτημα του πότε είναι σωστή η χρήση κληρονομικότητας μεταξύ δύο κλάσεων. Προτείνει τον κανόνα που αναφέρει ότι είναι ορθό μια κλάση Β να υλοποιηθεί σαν υποκλάση μια κλάσης Α, αν κάθε κώδικα Ρ, ο οποίος λειτουργεί με αντικείμενα της κλάσης Α, συμπεριφέρεται με τον ίδιο τρόπο και με αντίστοιχα αντικείμενα της κλάσης Β. Επειδή, ο παραπάνω έλεγχος είναι αρκετά κοπιώδης και εξαντλητικός και όσο μεγαλύτερο είναι το σύστημά, τόσο πιο πολύ αυξάνεται το κόστος του ελέγχου, ακολουθείται η τεχνική της Σχεδίασης βάσει Συμβολαίου (Design by contract).

Η Αρχή της Ανοιχτής/Κλειστής Σχεδίασης (Open/Closed Principle – OCP) προβλέπει ότι τα επιμέρους συστατικά ενός συστήματος (π.χ. κλάσεις, μέθοδοι, κτλ.) πρέπει να είναι ανοιχτά για επέκταση, αλλά κλειστά σε αλλαγές στην υλοποίησή τους. Αν για οποιονδήποτε λόγο απαιτηθούν αλλαγές, τότε θα πρέπει να υλοποιηθούν επεκτείνοντας τα ήδη υπάρχοντα στοιχεία και όχι αλλάζοντας την υλοποίηση.

Η Αρχή Αντιστροφής Εξαρτήσεων (Dependency Inversion Principle – DIP) ορίζει ότι τα υψηλού επιπέδου στοιχεία δεν πρέπει να εξαρτώνται από χαμηλού επιπέδου στοιχεία. Ο όρος στοιχεία υψηλού επιπέδου αναφέρεται σε στοιχεία που χρησιμοποιούν λειτουργίες άλλων στοιχείων, τα οποία αποτελούν τα χαμηλού επιπέδου στοιχεία. Επιπλέον, σύμφωνα με την DIP τόσο τα υψηλού, όσο και τα χαμηλού επιπέδου στοιχεία θα πρέπει να εξαρτώνται από «ενδιάμεσα» στοιχεία αφαίρεσης.

Η Αρχή Διαχωρισμού Διεπαφών (Interface Segregation Principle – ISP) έχει περιγραφεί στην ενότητα 1.2.

### 2.3 Κακοτεχνίες κώδικα

Οι κακοτεχνίες κώδικα (bad smells in code) αναφέρονται σε μη αποδοτικές τεχνικές κατά την ανάπτυξη κώδικα, που πιθανόν να αποτελούν ένδειξη ύπαρξης κάποιου βαθύτερου προβλήματος. Συνήθως δεν αποτελούν σφάλματα (bugs) στον κώδικα, τα οποία οδηγούν το πρόγραμμα σε ανεπιθύμητη συμπεριφορά. Αντίθετα, υποδεικνύουν αδυναμίες της σχεδίασης του λογισμικού που ενδεχομένως θα επιβραδύνουν την ταχύτητα ανάπτυξής του ή θα αυξήσουν την πιθανότητα εμφάνισης σφαλμάτων στο μέλλον, σε περίπτωση τροποποίησής (π.χ. επέκτασής) του.

Αρκετές από τις κακοτεχνίες κώδικα είναι δυνατό να εντοπιστούν, από το γεγονός ότι αντιβαίνουν κάποια ή κάποιες από τις σχεδιαστικές αρχές. Συχνά, μια κακοτεχνία αποκρύπτει ένα σοβαρότερο πρόβλημα, το οποίο αποκαλύπτεται όταν η σχεδίαση είναι δυνατόν να βελτιωθεί με την εφαρμογή κάποιας τεχνικής ανακατασκευής. Επομένως, από την οπτική γωνία του προγραμματιστή, θα λέγαμε ότι οι κακοτεχνίες κώδικα είναι εμπειρικοί κανόνες (heuristics) που αποτελούν ενδείξεις για το πότε ο κώδικας θα έπρεπε να ανακατασκευαστεί και για το ποιες τεχνικές ανακατασκευής θα έπρεπε να εφαρμοστούν. Ωστόσο, αξίζει να σημειωθεί ότι είναι αναπόφευκτη η ύπαρξη υποκειμενικότητας, ως έναν βαθμό, στην κρίση του ποιες περιπτώσεις χαρακτηρίζονται κακοτεχνίες κώδικα και ποιες όχι. Συχνά, η απόφαση αυτή, εξαρτάται από διάφορους παράγοντες, όπως η γλώσσα προγραμματισμού, ο προγραμματιστής και η μεθοδολογία ανάπτυξης του λογισμικού. Παρόλα, αυτά έχει σταχυολογηθεί ένα, κοινά αποδεκτό, σύνολο από κακοτεχνίες [4]. Στον Πίνακα 2.1 έχουμε συγκεντρώσει τις κακοσμίες αυτές, μαζί με τα συμπτώματα που εμφανίζουν, αλλά και τις προτεινόμενες τεχνικές επίλυσής τους.

Πίνακας 2.1 Κακοτεχνίες κώδικα.

| Όνομα | Συμπτώματα | Προτεινόμενη τεχνική Refactoring |
|---|---|---|

| | | |
|---|---|---|
| Επαναλαμβανόμενος Κώδικας (Duplicated Code) | Ο επαναλαμβανόμενος κώδικας είναι ένα από τα πιο συνηθισμένα προβλήματα. Μπορεί να εμφανιστεί σε διάφορες μορφές | Extract Method<br>Pull Up Field<br>Form Template Method<br>Substitute Algorithm<br>Extract Class |
| Εκτενής Μέθοδος (Long Method) | Οι μεγάλες μέθοδοι είναι, επίσης, ένα συνηθισμένο πρόβλημα. Η κατανόηση, ο έλεγχος και η συντήρηση έχουν μεγάλο κόστος | Extract Method<br>Replace Temp with Query<br>Introduce Parameter Object<br>Preserve whole Object<br>Replace Method with Method Object |
| Εκτενής Κλάση (Long Class) | Συνηθισμένο φαινόμενο είναι η κατασκευή πολύ μεγάλων κλάσεων που παραβιάζουν την αρχή SRP | Extract Class<br>Extract Subclass<br>Extract Interface<br>Duplicate Observed Data |
| Εκτενής Λίστα Παραμέτρων (Long Parameter List) | Μέθοδοι με πολλές παραμέτρους δυσκολεύουν την κατανόηση και τη χρήση τους | Replace Parameter with Method<br>Preserve Whole Object<br>Introduce Parameter Object |
| Αποκλίνουσα Αλλαγή (Divergent Change) | Διάφορα τμήματα κώδικα μιας κλάσης αλλάζουν ποικιλοτρόπως για διαφορετικούς λόγους. Ένδειξη παραβίασης της SRP | Extract Class |
| Shotgun Surgery | Αλλαγή στο σύστημα επιφέρει πολλές μικρές αλλαγές σε διάφορες κλάσεις, που κοστίζουν. Ένδειξη παραβίασης της SRP | Move Method<br>Move Field<br>Inline Class |

| | | |
|---|---|---|
| Feature Envy | Μέθοδος μιας κλάσης χρησιμοποιεί πιο πολύ συστατικά (μεθόδους και πεδία) μιας άλλης κλάσης από ό,τι συστατικά της κλάσης που ανήκει. Ένδειξη μη αποδοτικής ανάθεσης αρμοδιοτήτων στις κλάσεις. Αύξηση δυσκολίας κατανόησης και συντήρησης | Move Method<br>Extract Method |
| Lazy Class | Μια κλάση δεν επιτελεί πλέον χρήσιμα πράγματα, πιθανόν λόγω προηγούμενων ανακατασκευών. | Collapse Hierarchy<br>Inline Class |
| Εναλλακτικές Κλάσεις με Διαφορετικές Διεπαφές (Alternative Classes with Different Interfaces) | Κλάσεις υλοποιούν ίδιες λειτουργίες, ενώ έχουν διαφορετικά πρωτότυπα μεθόδων. Ένδειξη παραβίασης των DIP, LCP και OCP | Rename Method<br>Move Method<br>Extract Superclass |
| Εντολές Switch/If (Switch/If Statemetnts) | Η ύπαρξη εντολών switch που αντιστοιχούν σε κώδικα που εκτελειται για διαφορετικές κλάσεις αντικειμένων, είναι μια ένδειξη μη σωστής χρήσης του πολυμορφισμού, της DIP, της LCP και της OCP | Extract Method<br>Move Method<br>Replace Type Code with Subclasses<br>Replace Type Code with State/Strategy<br>Replace Conditional with Polymorphism |
| Αλυσίδα Μηνυμάτων (Message Chains) | Μια κλάση χρησιμοποιεί μια άλλη κλάση, αποκλειστικά, για της απόκτηση πρόσβασης σε μια τρίτη κλάση. Άσκοπη αύξηση της σύζευξης | Hide Delegate<br>Extract Method<br>Move Method |

| | | |
|---|---|---|
| Άρνηση Κληροδότησης (Refused Bequest) | Κάποια πεδία και/ή μέθοδοι μιας κλάσης δεν είναι απαραίτητα για μια υποκλάση. Ένδειξη μη ορθής χρήσης κληρονομικότητας και κακοσχεδιασμένης ιεραρχίας | Push Down Method<br>Push Down Field<br>Replace Inheritance with Delegation |

**2.4 Τεχνικές ανακατασκευής κώδικα**

Η βασική διαδικασία όσον αφορά τις τεχνικές ανακατασκευής περιλαμβάνει, αρχικά την εφαρμογή τους, βήμα-βήμα και τον έλεγχο ορθής λειτουργίας του προγράμματος ύστερα από κάθε βήμα. Δεν αποκλείεται η εφαρμογή μιας τεχνικής, να οδηγήσει το πρόγραμμα σε μια μορφή τέτοια, ώστε να είναι δυνατή και ωφέλιμή η εφαρμογή επιπλέον τεχνικών ανακατασκευής. Ορισμένες από τις τεχνικές είναι κατάλληλες για εφαρμογή σε ειδικές περιστάσεις και θα πρέπει να προσαρμοστούν στην ιδιαιτερότητα του κάθε προγράμματος.

Οι τεχνικές ανακατασκευής σχετίζονται με τις βασικές αρχές σχεδίασης [8], καθώς αρκετές από τις τεχνικές, έχουν ως βάση στη φιλοσοφία της λειτουργίας τους κάποια από τις αρχές. Επιπλέον, υπάρχει μια στενή σχέση μεταξύ τεχνικών ανακατασκευής και σχεδιαστικών προτύπων (design patterns) [5]. Συνεπώς, ορισμένες τεχνικές εμπνέονται από τη φιλοσοφία ορισμένων εξ αυτών. Τα σχεδιαστικά πρότυπα θα λέγαμε ότι αποτελούν το στόχο της τελικής μορφής ενός προγράμματος και οι τεχνικές ανακατασκευής τα μέσα με τα οποία επιτυγχάνεται ο στόχος αυτός,

Οι υποενότητες που ακολουθούν επιχειρούν μια συγκεντρωτική καταγραφή των τεχνικών ανακατασκευής, των συμπτωμάτων (bad smells), που όταν εμφανίζονται, αποτελούν ένδειξη για τη χρήση της κάθε τεχνικής, αλλά και τη λύση που προτείνει η κάθε τεχνική.

*2.4.1 Κατασκευή Μεθόδων*

Μεγάλο μέρος του πεδίου που ονομάζεται ανακατασκευή κώδικα περιλαμβάνει τη σύνθεση μεθόδων με στόχο την αποδοτική συγκέντρωση του κώδικα. Πολλές φορές, όμως, τα προβλήματα προέρχονται από μεθόδους που είναι εκτενείς (Large Classes). Οι εκτενείς μέθοδοι είναι προβληματικές διότι συχνά περιέχουν πληροφορία που γίνεται δυσδιάκριτη εξαιτίας της πολύπλοκης λογικής που συνεπάγεται το μεγάλο μέγεθος της μεθόδου. Βασικός στόχος των τεχνικών ανακατασκευής της ενότητας είναι η απλοποίηση μεθόδων που περιλαμβάνουν πολλές γραμμές κώδικα. Στον Πίνακα 2.2 έχουμε συγκεντρώσει τις τεχνικές Κατασκευής Μεθόδων (Composing Methods), τα συμπτώματα, που όταν εμφανίζονται, αποτελούν ένδειξη για τη χρήση της κάθε τεχνικής, αλλά και τη λύση που προτείνει η κάθε τεχνική.

Πίνακας 2.2 Τεχνικές ανακατασκευής: Κατασκευή Μεθόδων.

| Όνομα | Συμπτώματα | Λύση Τεχνικής |
|---|---|---|
| Εξαγωγή Μεθόδου (Extract Method) | Μέθοδοι μεγάλου μεγέθους. Τμήματα κώδικα που χρειάζονται σχόλια για την κατανόησή τους | Δημιουργία νέας μεθόδου Χρήση ονόματος αντιπροσωπευτικού της λειτουργίας της |
| Ενσωμάτωση Μεθόδου (Inline Method) | Μέθοδοι που απλά ανακατευθύνουν στην κλήση μιας άλλης μεθόδου | Κατάργηση της μεθόδου και ενσωμάτωση του κώδικά της στη μέθοδο στην οποία ανακατεύθυνε |
| Ενσωμάτωση Προσωρινής Μεταβλητής (Inline Temp) | Μια προσωρινή μεταβλητή αρχικοποιείται μια φορά μόνο, με το αποτέλεσμα κλήσης μεθόδου | Αντικατάσταση των σημείων που χρησιμοποιείται η προσωρινή μεταβλητή με την κλήση της μεθόδου |
| Αντικατάσταση Προσωρινής Μεταβλητής με Κλήση (Replace Temp with Query) | Μια μεταβλητή χρησιμοποιείται μια φορά και αποθηκεέι την τιμή μιας έκφρασης | Εφαρμογή Εξαγωγής Μεθόδου για την έκφραση και αντικατάσταση των χρήσεων της μεταβλητής με την κλήση στη μέθοδο |

| | | |
|---|---|---|
| Εισαγωγή Επεξηγηματικής Μεταβλητής (Introduce Explaining Variable) | Μια πολύπλοκη και δυσνόητη έκφραση στον κώδικα | Τοποθέτηση του αποτελέσματος της έκφρασης σε μια μεταβλητή με χαρακτηριστικό όνομα που επεξηγεί την έκφραση |
| Διαχωρισμός Προσωρινής Μεταβλητής (Split Temporary Variable) | Μια προσωρινή μεταβλητή χρησιμοποιείται στον κώδικα για διαφορετικού σκοπούς με διαφορετικές αρμοδιότητες | Χρησιμοποίηση διαφορετικής μεταβλητής για κάθε διαφορετικό σκοπό |
| Κατάργηση Αναθέσεων σε Παραμέτρους (Remove Assignments to Parameters) | Ο κώδικας μιας μεθόδου αναθέτει τιμές στις παραμέτρους της μεθόδου | Χρησιμοποίηση προσωρινής μεταβλητής |
| Αντικατάσταση Μεθόδου με Αντικείμενο Μεθόδου (Replace Method with Method Object) | Ο κώδικας μιας μεθόδου είναι πολύ μεγάλος και επιπλέον οι μεταβλητές χρησιμοποιούνται με τέτοιον τρόπο, ώστε είναι δύσκολο να εφαρμοστεί Εξαγωγή Μεθόδου | Κατασκευή νέας κλάσης μόνο για τη μέθοδο, με πεδία τις μεταβλητές που χρησιμοποιεί. Απλοποίηση της μεθόδου στη συν |
| Υποκατάσταση Αλγορίθμου (Substitute) | Αλγόριθμος που υλοποιείται σε μια μέθοδο είναι πολύπλοκος | Αντικατάσταση του αλγορίθμου με νέο, πιο απλό |

*2.4.2 Μετακινήσεις Συστατικών*

Μια από τις πιο θεμελιώδεις, αν όχι η πιο θεμελιώδης, αποφάσεις στον αντικειμενοστραφή προγραμματισμό είναι, η ανάθεση αρμοδιοτήτων μεταξύ των κλάσεων. Ακόμα και έμπειροι προγραμματιστές, ενδεχομένως αντιμετωπίζουν

προβλήματα σχετικά με την παραπάνω απόφαση, τουλάχιστον σε πρώτη φάση. Βασικός στόχος των τεχνικών ανακατασκευής της ενότητας είναι η μετακίνηση συστατικών, δηλαδή πεδίων και μεθόδων, από μια κλάση σε άλλη. Η εν λόγω μετακίνηση, επιφέρει βελτίωση στην κατανόηση, στη σύζευξη (LCP), και στη συνεκτικότητα των κλάσεων (SRP). Στον Πίνακα 2.3 έχουμε συγκεντρώσει τις τεχνικές Μετακίνησης Συστατικών (Moving Features), τα συμπτώματα, που όταν εμφανίζονται, αποτελούν ένδειξη για τη χρήση της κάθε τεχνικής, αλλά και τη λύση που προτείνει η κάθε τεχνική.

Πίνακας 2.3 Τεχνικές ανακατασκευής: Μετακίνηση Συστατικών.

| Όνομα | Συμπτώματα | Λύση Τεχνικής |
|---|---|---|
| Μετακίνηση Μεθόδου (Move Method) | Μια μέθοδος σχετίζεται με περισσότερα συστατικά μιας άλλης κλάσης, από ό,τι της κλάσης στην οποία ανήκει | Μετακίνηση της μεθόδου στην κλάση με τα συστατικά της οποίας συσχετίζεται περισσότερο |
| Μετακίνηση Πεδίου (Move Field) | Ένα πεδίο σχετίζεται με περισσότερα συστατικά μιας άλλης κλάσης, από ό,τι της κλάσης στην οποία ανήκει | Μετακίνηση του πεδίου στην κλάση με τα συστατικά της οποίας συσχετίζεται περισσότερο |
| Εξαγωγή Κλάσης (Extract Class) | Μια κλάση είναι πολύ μεγάλη και έχει πολλές αρμοδιότητες. Συνεπώς, είναι δύσκολη στην κατανόησή της | Διαχωρισμός αρμοδιοτήτων σε περισσότερες κλάσεις |
| Ενσωμάτωση Κλάσης (Inline Class) | Η ύπαρξη μιας κλάσης δεν έχει κάποιο ουσιαστικό όφελος | Μετακίνηση των αρμοδιοτήτων της κλάσης, σε μια άλλη κλάση και κατάργηση της πρώτης |

| | | |
|---|---|---|
| Απόκρυψη Αντιπροσώπου (Hide Delegate) | Μια κλάση αποκτά πρόσβαση μέσω μιας getter μεθόδου σε ένα αντικείμενο που χαρακτηρίζει μια άλλη κλάση και καλεί μεθόδους στο αντικείμενο. Παραβίαση της ενθυλάκωσης | Κατασκευή κατάλληλων μεθόδων στην κλάση που χαρακτηρίζεται από το αντικείμενο και κλήση του αντικειμένου, μέσω των μεθόδων αυτών |
| Κατάργηση Μεσάζοντα (Remove Middle Man) | Αντίθετο φαινόμενο με το Hide Delegate. Μια κλάση δεν επιτελεί πολλές λειτουργίες, αλλά μόνο το ρόλο του ενδιάμεσου. Προωθεί κλήσεις από άλλες κλάσεις σε ένα αντικείμενο που χαρακτηρίζει την κλάση. | Αναπροσαρμογή, των κλάσεων που χρησιμοποιούν την ενδιάμεση κλάση, ώστε να χρησιμοποιούν απευθείας το αντικείμενο που τη χαρακτηρίζει |
| Εισαγωγή Ξένης Μεθόδου (Introduce Foreign Method) | Μια κλάση επεξεργάζεται με τον ίδιο τρόπο, σε διάφορα σημεία, το αντικείμενο μιας άλλης κλάσης, καλώντας διάφορες μεθόδους του αντικειμένου. Επανάληψη κώδικα | Κατασκευή μιας μεθόδου στην κλάση για τον επαναλαμβανόμενο κώδικα. Η μέθοδος έχει ως παράμετρο το αντικείμενο |
| Εισαγωγή Τοπικής Προσθήκης (Introduce Local Extension) | Σε μια κλάση χρειάζεται προσθήκη νέων μεθόδων, αλλά ο κώδικας της κλάσης δεν είναι διαθέσιμος | Κατασκευή νέας κλάσης με τις επιπλέον μεθόδους. Η νέα κλάση μπορεί να συσχετίζεται με την αρχική, είτε μέσω κληρονομικότητας, είτε μέσω της υλοποίησής της ως wrapper |

*2.4.3 Απλοποιήσεις Συνθηκών*

Η λογική με συνθήκες μπορεί να γίνει αρκετά πολύπλοκη. Αν και ο αντικειμενοστραφής προγραμματισμός χρησιμοποιεί λιγότερες δομές συνθήκης από τον διαδικαστικό προγραμματισμό, λόγο του πολυμορφισμού, υπάρχουν περιπτώσεις όπου η ύπαρξη συνθηκών δημιουργεί δαίδαλο συλλογιστικής πορείας. Βασικός στόχος των τεχνικών ανακατασκευής της ενότητας, είναι η απλοποίηση σύνθετων εκφράσεων με συνθήκες. Τέτοιου είδους απλοποιήσεις, επιφέρουν σημαντική βελτίωση στην κατανόηση του κώδικα και κατ' επέκταση, συνδράμουν στην εύκολη και φθηνή συντήρησή του. Στον Πίνακα 2.4 έχουμε συγκεντρώσει τις τεχνικές Απλοποίησης Συνθηκών (Simplifying Conditionals), τα συμπτώματα, που όταν εμφανίζονται, αποτελούν ένδειξη για τη χρήση της κάθε τεχνικής, αλλά και τη λύση που προτείνει η κάθε τεχνική.

Πίνακας 2.4 Τεχνικές ανακατασκευής: Απλοποίηση Συνθηκών.

| Όνομα | Συμπτώματα | Λύση Τεχνικής |
|---|---|---|
| Αποσύνθεση Συνθήκης (Decompose Conditional) | Μια μέθοδος περιλαμβάνει μια πολύπλοκη εντολή συνθήκης | Κατασκευή μεθόδων για τα επιμέρους τμήματα της πολύπλοκης εντολής συνθήκης |

| | | |
|---|---|---|
| Ενοποίηση Συνθήκης (Consolidate Conditional) | Μια μέθοδος περιλαμβάνει μια ακολουθία από διαφορετικές εντολές συνθήκης που έχουν το ίδιο αποτέλεσμα | Συνένωση των επιμέρους εντολών συνθήκης σε μια εντολή συνθήκης και κατασκευή αντίστοιχης μεθόδου |
| Ενοποίηση Επαναλαμβανόμενων Τμημάτων Κώδικα σε Συνθήκη (Consolidate Duplicate Conditional Fragments) | Σε όλα τα επιμέρους τμήματα μιας εντολής συνθήκης, υπάρχει επαναλαμβανόμενος κώδικας | Μετακίνηση του επαναλαμβανόμενου κώδικα εκτός της εντολής συνθήκης |
| Κατάργηση Ελέγχου Σημαίας (Remove Control Flag) | Μια μεταβλητή χρησιμοποιείται για τον έλεγχο μιας επαναληπτικής διαδικασίας. Η διαδικασίας περιλαμβάνει ένα σύνολο εντολών συνθήκης τα οποία καθορίζουν την τιμή της μεταβλητής | Αντικατάσταση των εντολών συνθήκης με εντολές break, continue, return |
| Αντικατάσταση Εμφωλευμένων Συνθηκών με Φρουρούς (Replace Nested Conditional with Guard Clauses) | Μια πολύπλοκη εντολή συνθήκης, διαχωρίζει το βασικό μονοπάτι εκτέλεσης μιας μεθόδου σε μονοπάτια εκτέλεσης Κάθε μονοπάτι αντιστοιχεί σε ειδική περίπτωση | Για κάθε ειδική περίπτωση, χρησιμοποίηση μιας απλής εντολής συνθήκης που παίζει το ρόλο «φρουρού» |
| Αντικατάσταση Συνθήκης με Πολυμορφισμό (Replace Conditional with Polymorphism) | Μια μέθοδος περιλαμβάνει μια εντολή συνθήκης κάθε τμήμα της οποίας αντιστοιχεί σε μια διαφορετική συμπεριφορά της μεθόσου | Κατασκευή υποκλάσης για κάθε διαφορετική συμπεριφορά της μεθόδου |

| | | |
|---|---|---|
| Εισαγωγή Υπόθεσης (Introduce Assertion) | Ένα τμήμα κώδικα υιοθετεί μια υπόθεση για την κατάσταση του προγράμματος, η οποία πρέπει να ισχύει πάντα | Ανάδειξη της υπόθεσης σε εμφανή θέση, χρησιμοποιώντας ένα assertion. Δηλαδή, χρησιμοποίηση μιας εντολής συνθήκης που ελέγχει εάν ισχύει η υπόθεση. Αν δεν ισχύει υποδεικνύει το λάθος |
| Εισαγωγή Null Αντικειμένου (Introduce Null Object) | Εμφάνιση επαναλαμβανόμενων ελέγχων για null τιμή | Αντικατάσταση της null τιμής με null αντικείμενο |

*2.4.4 Απλοποιήσεις Μεθόδων*

Η δημιουργία μεθόδων, εύκολων στην κατανόηση και στη χρησιμοποίησή τους είναι μια βασική ιδιότητα του ποιοτικού αντικειμενοστραφούς λογισμικού. Βασικός στόχος των τεχνικών ανακατασκευής της ενότητας, είναι η απλοποίηση των πρωτότυπων (ονόματα, παράμετροι, κτλ.) των μεθόδων αυτών. Τέτοιου είδους απλοποιήσεις, επιφέρουν σημαντική βελτίωση στην κατανόηση του κώδικα και κατ' επέκταση, συνδράμουν στην εύκολη και φθηνή συντήρησή του. Στον Πίνακα 2.5 έχουμε συγκεντρώσει τις τεχνικές Απλοποίησης Μεθόδων (Making Method Calls Simpler), τα συμπτώματα, που όταν εμφανίζονται, αποτελούν ένδειξη για τη χρήση της κάθε τεχνικής, αλλά και τη λύση που προτείνει η κάθε τεχνική.

Πίνακας 2.5 Τεχνικές ανακατασκευής: Απλοποίηση Μεθόδων.

| Όνομα | Συμπτώματα | Λύση Τεχνικής |
|---|---|---|

| | | |
|---|---|---|
| Μετονομασία Μεθόδου (Rename Method) | Το όνομα μιας μεθόδου δεν αντιστοιχεί στο σκοπό που εξυπηρετεί η μέθοδος | Αλλαγή του ονόματος της μεθόδου, σε κατάλληλο, για το σκοπό που εξυπηρετεί |
| Προσθήκη Παραμέτρου (Add Parameter) | Μια μέθοδος χρειάζεται περισσότερες πληροφορίες σαν είσοδο | Προσθήκη επιπλέον παραμετρων |
| Κατάργηση Παραμέτρου (Remove Parameter) | Η παράμετρος μιας μεθόδου δεν χρησιμοποιείται | Κατάργηση της παραμέτρου |
| Separate Query from Modifier | Μια μέθοδος επιστρέφει ένα αποτέλεσμα και επιπλέον αλλάζει την τρέχουσα κατάσταση του αντικειμένου στο οποίο καλείται | Δημιουργία δύο διαφορετικών μεθόδων. Μία που επιστρέφει το αποτέλεσμα και μία που αλλάζει την κατάσταση του αντικειμένου |
| Παραμετροποίηση Μεθόδου (Parameterize Method) | Διάφορες μέθοδοι κάνουν παρόμοιες λειτουργίες, αλλά με διαφορετικές σταθερές, που περιλαμβάνονται στον κώδικά τους | Κατασκευή παραμετροποιημένης μεθόδου με παραμέτρους, αντίστοιχες με τις σταθερές |
| Αντικατάσταση Παραμέτρου με Μεθόδους (Replace Parameter with Explicit Methods) | Μια μέθοδος επιτελεί διαφορετικές λειτουργίες, ανάλογα με την τιμή μιας παραμέτρου | Ορισμός διαφορετικών μεθόδων για κάθε διαφορετική λειτουργία |
| Preserve Object | Μια μέθοδος καλείται με παραμέτρους που προκύπτουν από κλήσεις σε μεθόδους ενός άλλου αντικειμένου | Αντί για πολλές παραμέτρους, πέρασμα του αντικειμένου, από το οποίο προκύπτουν οι τιμές των παραμέτρων |

| | | |
|---|---|---|
| Αντικατάσταση Παραμέτρου με Μέθοδο (Replace Parameter with Method) | Μια μέθοδος Α καλείται με μια παράμετρο που προκύπτει από την κλήση μιας άλλης μεθόδου Β. Η Β, όμως, μπορεί να κληθεί και από την Χ | Κατάργηση της παραμέτρου και κλήση της μεθόδου Β από την Α |
| Introduce Parameter Object | Μια μέθοδος χρησιμοποιεί ένα σύνολο παραμέτρων που σχετίζονται μεταξύ τους | Αντικατάσταση των παραμέτρων με ένα, κατάλληλα ορισμένο, αντικείμενο |
| Κατάργηση Μεθόδων Αρχικοποίησης (Remove Setting Method) | Ένα πεδίο αρχικοποιείται μια φορά και εν συνεχεία δεν αλλάζει τιμή | Κατάργηση των μεθόδων που θέτουν τιμές σε αυτό το πεδίο |
| Απόκρυψη Μεθόδου (Hide Method) | Η μέθοδος μιας κλάσης δεν χρησιμοποιείται από καμία άλλη κλάση εκτός αυτής στην οποία ανήκει | Δήλωση της μεθόδου ως private |

*2.4.5 Χειρισμός Γενίκευσης*

Βασικός στόχος των τεχνικών ανακατασκευής της ενότητας, είναι η απλοποίηση των ιεραρχιών των κλάσεων. Τέτοιου είδους αλλαγές, επιφέρουν σημαντική βελτίωση στη σωστή χρήση της κληρονομικότητας, στη σύζευξη, στη συνεκτικότητα, στη συντήρηση και στην επεκτασιμότητα των κλάσεων. Στον Πίνακα 2.6 έχουμε συγκεντρώσει τις τεχνικές Χειρισμού Γενίκευσης (Dealing with Generalization), τα συμπτώματα, που όταν εμφανίζονται, αποτελούν ένδειξη για τη χρήση της κάθε τεχνικής, αλλά και τη λύση που προτείνει η κάθε τεχνική.

Πίνακας 2.6 Τεχνικές ανακατασκευής: Χειρισμός Γενίκευσης.

| Όνομα | Συμπτώματα | Λύση Τεχνικής |
|---|---|---|
| Μετακίνηση Πεδίου προς τα Πάνω (Pull Up Field) | Δύο ή περισσότερες υποκλάσεις έχουν το ίδιο πεδίο | Μετακίνηση του πεδίου στη βασική κλάση |
| Μετακίνηση Μεθόδου προς τα Πάνω (Pull Up Method) | Δύο ή περισσότερες υποκλάσεις έχουν την ίδια μέθοδο | Μετακίνηση της μεθόδου στη βασική κλάση |
| Μετακίνηση Κατασκευαστή προς τα Πάνω (Pull Up Constructor Body) | Δύο ή περισσότερες υποκλάσεις έχουν σχεδόν τους ίδιους κατασκευαστές | Ορισμός κατασκευαστή που περιέχει τον κοινό κώδικα στη βασική κλάση και κλήση του στους κατασκευαστές των υποκλάσεων |
| Μετακίνηση Μεθόδου προς τα Κάτω (Push Down Method) | Μέθοδος μιας βασικής κλάσης σχετίζεται μόνο με κάποια/-ες υποκλάση/-εις | Μετακίνηση της μεθόδου στην/στις αντίστοιχη/-ες υποκλάση/-εις |
| Μετακίνηση Πεδίου προς τα Κάτω (Push Down Field) | Πεδίο μιας βασικής κλάσης σχετίζεται μόνο με κάποια/-ες υποκλάση/-εις | Μετακίνηση του πεδίου στην/στις αντίστοιχη/-ες υποκλάση/-εις |
| Εξαγωγή Υποκλάσης (Extract Subclass) | Μια κλάση έχει πεδία και μεθόδους που χαρακτηρίζουν μόνο ένα υποσύνολο των αντικειμένων της κλάσης | Ορισμός μιας υποκλάσης που περιλαμβάνει τα εν λόγω πεδία και μεθόδους |
| Εξαγωγή Υπερκλάσης (Extract Superclass) | Δύο ή περισσότερες κλάσεις έχουν παρόμοια πεδία και μεθόδους | Ορισμός μιας βασικής κλάσης και μετακίνηση σε αυτήν των κοινών πεδίων και μεθόδων |

| | | |
|---|---|---|
| Εξαγωγή Διεπαφής (Extract Interface) | Διάφορες κλάσεις χρησιμοποιούν το ίδιο υποσύνολο public μεθόδων μιας άλλης κλάσης. Ή. δύο ή περισσότερες κλάσεις έχουν public μεθόδους με τα ίδια πρωτότυπα | Δημιουργία μιας διεπαφής που περιλαμβάνει τις εν λόγω μεθόδους |
| Δημιουργία Μεθόδου Πρωτοτύπου (Form Template Method) | Σε δύο ή περισσότερες υποκλάσεις υπάρχουν μέθοδοι που εκτελούν παρόμοια βήματα, με την ίδια σειρά | Δημιουργία μεθόδων, αντίστοιχων με τα βήματα και μετακίνηση των κοινών μεθόδων στη βασική κλάση |
| Αντικατάσταση Κληρονομικότητας με Αντιπροσωπεία (Replace Inheritance with Delegation) | Οι κλάσεις που χρησιμοποιούν μια υποκλάση χρησιμοποιούν λίγες ή καμία, από τις μεθόδους που κληρονομούνται από τη βασική κλάση. Ή, μια υποκλάση δεν έχει νόημα να προσφέρει στις κλάσεις που την χρησιμοποιούν ένα σύνολο ή υποσύνολο των μεθόδων που κληρονομεί από τη βασική κλάση. Ή, μια υποκλάση δεν χρειάζεται να κληρονομεί πεδία από τη βασική κλάση | Κατάργηση της κληρονομικότητας |

| | | |
|---|---|---|
| Αντικατάσταση Αντιπροσωπείας με Κληρονομικότητας (Replace Delegation with Inheritance) | Μια κλάση συσχετίζεται με μια άλλη κλάση και έχει αρκετές delegate μεθόδους οι οποίες, απλώς καλούν αντίστοιχες μεθόδους της κλάσης με την οποία σχετίζεται | Κατάργηση της συσχέτισης και χρήση κληρονομικότητας |
| Κατάρρευση Ιεραρχίας (Collapse Hierarchy) | Μια υποκλάση και η αντίστοιχη υπερκλάση δεν διαφέρουν σημαντικά | Συγχώνευση της υποκλάσης στην υπερκλάση |

*2.4.6 Οργάνωση Δεδομένων*

Η δυνατότητα ορισμού νέων τύπων δεδομένων, προσφέρει περισσότερες προοπτικές σε σχέση με τους θεμελιώδεις τύπους δεδομένων. Ωστόσο, η αποτελεσματική διαχείριση των δεδομένων μέσω αντικειμένων αποτελεί ένα κοπιώδες έργο για τους προγραμματιστές. Βασικός στόχος των τεχνικών ανακατασκευής της ενότητας, είναι η ευελιξία στην οργάνωση των δεδομένων. Τέτοιου είδους τεχνικές, επιφέρουν σημαντική βελτίωση στη σωστή χρήση αντικειμένων, στη σύζευξη, στη συντήρηση και στην απλοποίηση της ιεραρχίας των κλάσεων. Στον Πίνακα 2.6 έχουμε συγκεντρώσει τις τεχνικές Οργάνωσης Δεδομένων (Organize Data), τα συμπτώματα, που όταν εμφανίζονται, αποτελούν ένδειξη για τη χρήση της κάθε τεχνικής, αλλά και τη λύση που προτείνει η κάθε τεχνική.

Πίνακας 2.7 Τεχνικές ανακατασκευής: Οργάνωση Δεδομένων.

| Όνομα | Συμπτώματα | Λύση Τεχνικής |
|---|---|---|
| Αυτό-ενθυλάκωση Πεδίου (Self Encapsulate Field) | Η πρόσβαση σε ένα πεδίο γίνεται απευθείας και έτσι, η σύζευξη αυξάνεται | Δημιουργία setter και getter μεθόδων για την πρόσβαση στο εν λόγω πεδίο |

| | | |
|---|---|---|
| Αντικατάσταση Τιμής Δεδομένου με Αντικείμενο (Replace Data Value with Object) | Μια τιμή δεδομένων χρειάζεται προσθήκη επιπλέον πληροφορίας ή συμπεριφοράς | Μετατροπή της τιμής δεδομένων σε αντικείμενο |
| Αντικατάσταση Πίνακα με Αντικείμενο (Replace Array with Object) | Ένας πίνακας έχει συγκεκριμένα στοιχεία που συμβολίζουν διαφορετικές έννοιες | Αντικατάσταση του πίνακα με ένα αντικείμενο που έχει ένα πεδίο για κάθε εν λόγω στοιχείο του πίνακα |
| Τροποποίηση της Μονόδρομης Συσχέτισης σε Αμφίδρομη (Change Unidirectional Association to Bidirectional) | Υπάρχουν δύο κλάσεις που η μία πρέπει να χρησιμοποιεί τα συστατικά της άλλης, αλλά υπάρχει συσχέτιση μονής κατεύθυνσης | Προσθήκη δεικτών για τη συσχέτιση και προς την αντίθετη κατεύθυνση (back pointers) |
| Τροποποίηση της Αμφίδρομης Συσχέτισης σε Μονόδρομη (Change Bidirectional Association to Unidirectional) | Δύο κλάσεις συσχετίζονται αμφίδρομα, αλλά, μία δεν χρειάζεται συστατική της άλλης | Διαγραφή του άκρου της αμφίδρομης συσχέτισης που δεν χρειάζεται πλέον |
| Αντικατάσταση Συγκεκριμένου Αριθμού με Συμβολική Σταθερά (Replace Magic Number with Symbolic Constant) | Υπάρχει ένας αριθμός με ειδική σημασία για το πρόγραμμα | Δημιουργία σταθεράς με όνομα αντίστοιχο της σημασίας του εν λόγω αριθμού και αντικατάσταση του αριθμού με τη σταθερά |
| Ενθυλάκωση Πεδίου (Encapsulate Field) | Ένα πεδίο είναι ορισμένο ως public | Ορισμός του πεδίου ως private και δημιουργία μεθόδων για την προσπέλασή του εκτός κλάσης |

| | | |
|---|---|---|
| Αντικατάσταση Υποκλάσης με Πεδία (Change Subclass with Fields) | Υποκλάσεις διαφέρουν μόνο σε μεθόδους που επιστρέφουν σταθερές τιμές` | Μετατροπή των μεθόδων σε πεδία της υπερκλάσης και διαγραφή των υποκλάσεων |

**2.5 Σχετική έρευνα**

Αρκετές ερευνητικές ομάδες έχουν ασχοληθεί με την ανακατασκευή λογισμικού έχοντας προσεγγίσει το θέμα από διαφορετικές οπτικές γωνίες τις περισσότερες φορές και προτείνοντας λύσεις με τεχνικές ανακατασκευής που εντάσσονται σε διαφορετικές ενότητες της ανακατασκευής λογισμικού.

Η [9] αποτελεί μια βιβλιογραφική επισκόπηση της υπάρχουσας έρευνας στο πεδίο της ανακατασκευής λογισμικού. Η κατηγοριοποίηση που παρουσιάζεται στην εργασία, πραγματοποιήθηκε με βάση πέντε (5) κριτήρια. Το πρώτο κριτήριο αναφέρεται στις διαδικασίες που απαιτούνται πριν, κατά τη διάρκεια και μετά την ανακατασκευή. Στον τομέα αυτό, περιλαμβάνεται η αναγνώριση των σημείων του λογισμικού στα οποία θα εφαρμοστεί ανακατασκευή, η παροχή εγγύησης ότι η συμπεριφορά του δεν θα αλλάξει, η εκτίμηση της αποτελεσματικότητας στην βελτίωση της ποιότητας και η διατήρηση της συμβατότητας μεταξύ του λογισμικού και των συστατικών με τα οποία συσχετίζεται (π.χ. κώδικας, σχεδίαση αρχιτεκτονικής, έγγραφα τεκμηρίωσης, μοντέλα UML). Το δεύτερο κριτήριο, αφορά στις τεχνικές και τους φορμαλισμούς που χρησιμοποιούνται σε αυτές. Συγκεκριμένα, μελετώνται τα invariants, οι προ-συνθήκες, οι μετασυνθήκες και οι μετασχηματισμοί γράφων που αντιστοιχίζονται σε μετασχηματισμούς λογισμικού, καθώς επίσης και πώς βοηθούν στην ορθότητα και τη διατήρηση της συμπεριφοράς του λογισμικού. Το τρίτο κριτήριο εξετάζει τα διάφορα επίπεδα του λογισμικού στα οποία μπορεί να εφαρμοστεί ανακατασκευή διακρίνοντας τρία (3): το επίπεδο πηγαίου κώδικα, το επίπεδο σχεδίασης του συστήματος (ανακατασκευή π.χ. UML διαγραμμάτων) και το επίπεδο της ανάλυσης απαιτήσεων (ανακατασκευή φυσικής γλώσσας περιγραφής απαιτήσεων). Το τέταρτο κριτήριο ασχολείται με τα εργαλεία που έχουν προταθεί για την ανακατασκευή.

Αναλύεται ο βαθμός αυτοματοποίησης (πλήρως αυτόματα ή ημιαυτόματα), η αξιοπιστία, η επεκτασιμότητα, το εύρος κάλυψης τεχνικών ανακατασκευής, ο βαθμός παραμετροποίησης και ανεξαρτησίας από κάποια συγκεκριμένη γλώσσα προγραμματισμού. Το πέμπτο κριτήριο εξετάζει πώς η ανακατασκευή εμπλέκεται στον ανασχεδιασμό λογισμικού (software reengineering), στην ανάπτυξη ευέλικτων προγραμμάτων (agile software development) και ανάπτυξη λογισμικού βασισμένο σε frameworks. Τα συμπεράσματα στα οποία κατέληξε είναι ότι παραμένουν ανοιχτά προβλήματα σε κάθε κατηγορία και ότι υπάρχει ανάγκη για προσεγγίσεις στην ανακατασκευή με μεγαλύτερη ευελιξία, επεκτασιμότητα και πιο γενικευμένους και συστηματικούς τρόπους.

Η [13] ασχολείται με τον τομέα της κατασκευής μεθόδων (composing methods). Συγκεκριμένα, εστιάζει στην τεχνική της εξαγωγής μεθόδου (extract method). Παρόλο που υπάρχουν πολλές διαθέσιμες μεθοδολογίες και εργαλεία για την εφαρμογή της συγκριμένης τεχνικής, οι περισσότερες, βασίζονται στον υπεύθυνο σχεδίασης του λογισμικού. Για το λόγο αυτό, επικεντρώνεται στην αυτοματοποίηση της διαδικασίας με την αυτόματη αναγνώριση σημείων στα οποία μπορεί να εφαρμοστεί η εξαγωγή μεθόδου και η παρουσίαση των αποτελεσμάτων ως προτάσεις βελτίωσης στον σχεδιαστή. Η βασική ιδέα στηρίζεται στον τεμαχισμό προγράμματος (program slicing), σύμφωνα με την οποία, ένα κομμάτι αποτελείται από όλες τι δηλώσεις του προγράμματος που μπορεί να επηρεάζουν την τιμή μιας μεταβλητής x σε ένα συγκεκριμένο σημείο ενδιαφέροντος p. Το ζεύγος (p,x) ορίζεται ως κριτήριο τεμαχισμού (slicing criterion). Η προτεινόμενη μεθοδολογία προτείνει τη συνένωση όλων των κομματιών του προγράμματος που αφορούν μια μεταβλητή σε μια νέα μέθοδο, μαζί με ένα σύνολο κανόνων που θα διασφαλίζουν τη συμπεριφορά του συστήματος μετά την εφαρμογή της, χωρίς επαναλαμβανόμενο κώδικα.

Η [14] προτείνει μια λύση στο πρόβλημα «Feature Envy», το οποίο παρατηρείται όταν «μια κλάση χρησιμοποιεί μια μέθοδο άλλης κλάσης πιο πολύ από ό,τι η κλάση στην οποία ανήκει η μέθοδος» [14]. Η προσέγγιση περιγράφει την μετακίνηση μεθόδου (move method) και εντάσσεται στην γενικότερη περιοχή των μετακινήσεων χαρακτηριστικών (moving features) της ανακατασκευής. Στην εργασία αυτή,

επιστρατεύεται η έννοια της απόστασης μεταξύ οντοτήτων (πεδία ή μέθοδοι κλάσεων) και κλάσεων για την αυτόματη αναγνώριση πιθανών σημείων εμφάνισης του φαινομένου «Feature Envy». Ο αλγόριθμος που υλοποιήθηκε, εξάγει προτεινόμενες ανακατασκευές μετακίνησης μεθόδων. Για κάθε μέθοδο του συστήματος, ο αλγόριθμος σχηματίζει ένα σύνολο από υποψήφιες κλάσεις, όπου η μέθοδος δύναται να μετακινηθεί, εξετάζοντας τις οντότητες των υπόλοιπων κλάσεων του συστήματος που προσπελαύνει η μέθοδος (εξαιρούνται δηλαδή οι κλάσεις που ανήκουν σε βιβλιοθήκες). Τελικά, επιλέγεται η κλάση η οποία ικανοποιεί ένα σύνολο προσυνθηκών, που εγγυάται τη διατήρηση της συμπεριφοράς του προγράμματος. Στην επιλογή λαμβάνεται υπόψη η μετρική που προτείνεται, «Entity Placement» που βασίζεται σε δύο αρχές:

Αρχικά, στην απόσταση μεταξύ των οντοτήτων που ανήκουν σε μια κλάση και της ίδιας της κλάσης, η οποία θα πρέπει να είναι η ελάχιστη δυνατή (υψηλή συνεκτικότητα).

Επίσης, στην απόσταση μεταξύ των οντοτήτων που δεν ανήκουν σε μια κλάση και της κλάσης αυτής, η οποία θα πρέπει να είναι η μέγιστη δυνατή (χαμηλή σύζευξη). Τελικά, η όλη προσέγγιση χαρακτηρίζεται ως ημιαυτόματη, καθώς η εφαρμογή ή όχι των προτεινόμενων ανακατασκευών είναι απόφαση του σχεδιαστή σε σχέση με εννοιολογικά και ποιοτικά κριτήρια σχεδίασης.

Στη μετακίνηση χαρακτηριστικών εστιάζει και η [10] προτείνοντας ανακατασκευή με τέσσερις τεχνικές: τη μετακίνηση μεθόδου, τη μετακίνηση πεδίων (move attribute), την εξαγωγή κλάσης (extract class) και την ενσωμάτωση κλάσης (inline class). Βασικό κριτήριο αποτελεί η συνεκτικότητα των κλάσεων, η οποία μετράται με την απόσταση Jaccard. Η βασική συνεισφορά της εργασίας είναι ότι μεταφράζει την παραπάνω απόσταση σε Ευκλείδεια απόσταση, απεικονίζοντας τις μεθόδους και τα πεδία των κλάσεων υπό εξέταση με γεωμετρικά σχήματα, μέσω μιας εφαρμογής, στον τρισδιάστατο χώρο. Με τον τρόπο αυτό, για παράδειγμα, αν μια μέθοδος $m_A$ μιας κλάσης Α εμφανίζεται απομακρυσμένη από τις υπόλοιπες μεθόδους και τα υπόλοιπα πεδία της Α και ταυτόχρονα κοντά στις μεθόδους και τα πεδία μιας άλλης κλάσης Β, τότε κάτι τέτοιο αποτελεί ένδειξη ότι θα πρέπει να εφαρμοστεί μετακίνηση μεθόδου και η μέθοδος $m_A$ να μετακινηθεί στην κλάση Β. Και εδώ, εκφράζεται η άποψη ότι ο

ανθρώπινος παράγοντας αποτελεί τον τελικό κριτή για το εάν απαιτείται ανακατασκευή και πού ή όχι. Παρόλα αυτά, επισημαίνεται ότι η υποστήριξη στην απόφαση αυτή μέσω εργαλείων κρίνεται απαραίτητη, αφού όσο μεγαλύτερο σε έκταση είναι ένα λογισμικό τόσο πιο δυσδιάκριτα γίνονται τα υποψήφια προς ανακατασκευή σημεία. Εδώ, η συνεκτικότητα των κλάσεων οπτικοποιείται στις τρεις διστάσεις μέσω σχημάτων σε διάφορες αποστάσεις βοηθώντας με αυτόν τον τρόπο την ανθρώπινη διαίσθηση.

Η [3] θεωρεί ότι η απλή εφαρμογή τεχνικών ανακατασκευής θεραπεύει απλά τα συμπτώματα της υλοποίησης κώδικα χαμηλής ποιότητας. Για το λόγο αυτό προτείνει ως θεραπεία, τη βελτίωση της σύζευξης και της συνεκτικότητας, χαρακτηριστικά τα οποία θεωρεί δείκτες ποιοτικής σχεδίασης. Λαμβάνοντας ως αφετηρία τη δομή και λειτουργία ορισμένων τεχνικών ανακατασκευής που ανήκουν σε διάφορες κατηγορίες, εξάγει έξι (6) γενικότερα συμπεράσματα που τα ονομάζει γενικές κατευθύνσεις, τα οποία αν ακολουθηθούν από κάποιον που εφαρμόζει ανακατασκευή, θα πετύχει μεγάλη συνεκτικότητα και μικρή σύζευξη.

Η [7] παρουσιάζει μια λύση ανακατασκευής στο πρόβλημα της παραμετροποίησης κλάσεων Java. Ειδικότερα, προτείνεται μια λύση στην παραμετροποίηση των κλάσεων, ώστε non generic κλάσεις, να μετατραπούν σε generic, με στόχο να αυξήσουν την εκφραστικότητά τους και το type safety τους. Τα generics είναι μια μορφή παραμετρικού πολυμορφισμού και εισήχθηκαν για πρώτη φορά στην έκδοση της Java 1.5 το 2004. Δίνουν τη δυνατότητα σε έναν τύπο δεδομένων ή μεθόδου να λειτουργήσει με αντικείμενα διάφορων τύπων, παρέχοντας ταυτόχρονα type safety σε χρόνο μεταγλώττισης (compile time). Το πρόβλημα παραμετροποίησης συνίσταται στην προσθήκη ενός τύπου σε μια υπάρχουσα κλάση, έτσι ώστε να μπορεί να χρησιμοποιείται σε διάφορα πλαίσια χωρίς να χάνεται πληροφορία. Για παράδειγμα η δήλωση: class ArrayList {…}, μετατρέπεται σε class ArrayList<T> {…}, με το T να μπορεί να αντικατασταθεί από διάφορους τύπους του Object. Εκτός της προηγούμενης κατηγορίας μετατροπών, ο αλγόριθμός που παρουσιάζεται, αντιμετωπίζει και το πρόβλημα της αρχικοποίησης του τύπου των αντικειμένων. Για παράδειγμα η δήλωση: ArrayList names; μετατρέπεται σε ArrayList<String> names;.

Η εν λόγω πρόταση είναι αυτοματοποιημένη, και μάλιστα με πολύ ενθαρρυντικά αποτελέσματα καθώς, όπως αναφέρεται, η χειρονακτική εναλλακτική είναι επίπονη, χρονοβόρα και επιρρεπής σε σφάλματα.

Η [6] είναι άλλη μια εργασία η οποία προτείνει τον αυτόματο εντοπισμό πιθανών σημείων του λογισμικού που θα μπορούσε να εφαρμοστεί ανακατασκευή. Για να το πετύχει, χρησιμοποιεί την έννοια των invariants, δηλαδή συνθήκες ή εκφράσεις που δηλώνονται ρητά στον κώδικα (στατικά invariants) ή θα πρέπει να εξαχθούν (δυναμικά invariants). Η εμφάνιση ενός συγκεκριμένου μοτίβου από invariants σε κάποιο σημείο του προγράμματος, αποτελεί ένδειξη ότι το συγκεκριμένο σημείο είναι υποψήφιο προς ανακατασκευή. Η μελέτη παραθέτει διάφορες περιπτώσεις στις οποίες διάφορα μοτίβα από invariants συνηγορούν προς ανακατασκευή του κώδικα. Για παράδειγμα, η τεχνική ανακατασκευής Αφαίρεση Παραμέτρου (Remove Parameter) ενδείκνυται να εφαρμοστεί όταν ισχύει τουλάχιστον μία από τις δύο εκφράσεις:

$$p = \text{constant}$$

$$p = f(a, b, \ldots)$$

όπου p είναι η παράμετρος προς διαγραφή, f είναι μια συνάρτηση υπολογισμού και a,b, … είναι είτε παράμετροι, είτε μεταβλητές στην εμβέλεια όμως του παραπάνω τμήματος κώδικα και επομένως η τιμή της p μπορεί να υπολογιστεί από τη διαθέσιμη πληροφορία. Τελικά, η εργασία καταλήγει στο συμπέρασμα ότι ο συνδυασμός της στατικής και της δυναμικής προσέγγισης φέρνει τα καλύτερα δυνατά αποτελέσματα.

Η [12] ασχολείται με την εξέλιξη της σχεδίασης αντικειμενοστραφών εφαρμογών με τεχνικές ανακατασκευής. Στόχος της ήταν να προσφέρει ένα εργαλείο με τη δυνατότητα η εφαρμογή ανακατασκευής να είναι τόσο εύχρηστη, όσο και οι GUI designers στην κατασκευή διεπαφών. Η συνεισφορά της έγκειται στην παρουσίαση γνωστών, αλλά και προτεινόμενων τεχνικών ανακατασκευής σε τρεις βασικούς τομείς: στο σχήμα αντικειμενοστραφών συστημάτων διαχείρισης βάσεων δεδομένων (object oriented database management systems), στα σχεδιαστικά πρότυπα (design patterns) και στην ανάλυση hot-spot, η οποία αναγνωρίζει τα μέρη του λογισμικού

που μπορεί να διαφοροποιούνται από εφαρμογή σε εφαρμογή. Τα αποτελέσματα των δοκιμών σε συστήματα βασισμένα σε C++, έδειξαν ότι οι τεχνικές ανακατασκευής που παρουσιάζονται μπορούν να εφαρμοστούν σε πραγματικά συστήματα με επιτυχία.

# ΚΕΦΑΛΑΙΟ 3. ΠΡΟΤΕΙΝΟΜΕΝΟΙ ΑΛΓΟΡΙΘΜΟΙ

---

3.1 Εισαγωγή
3.2 Clustering Based Approach (CBA)
3.3 Refactoring Based Approach (RBA)

---

## 3.1 Εισαγωγή

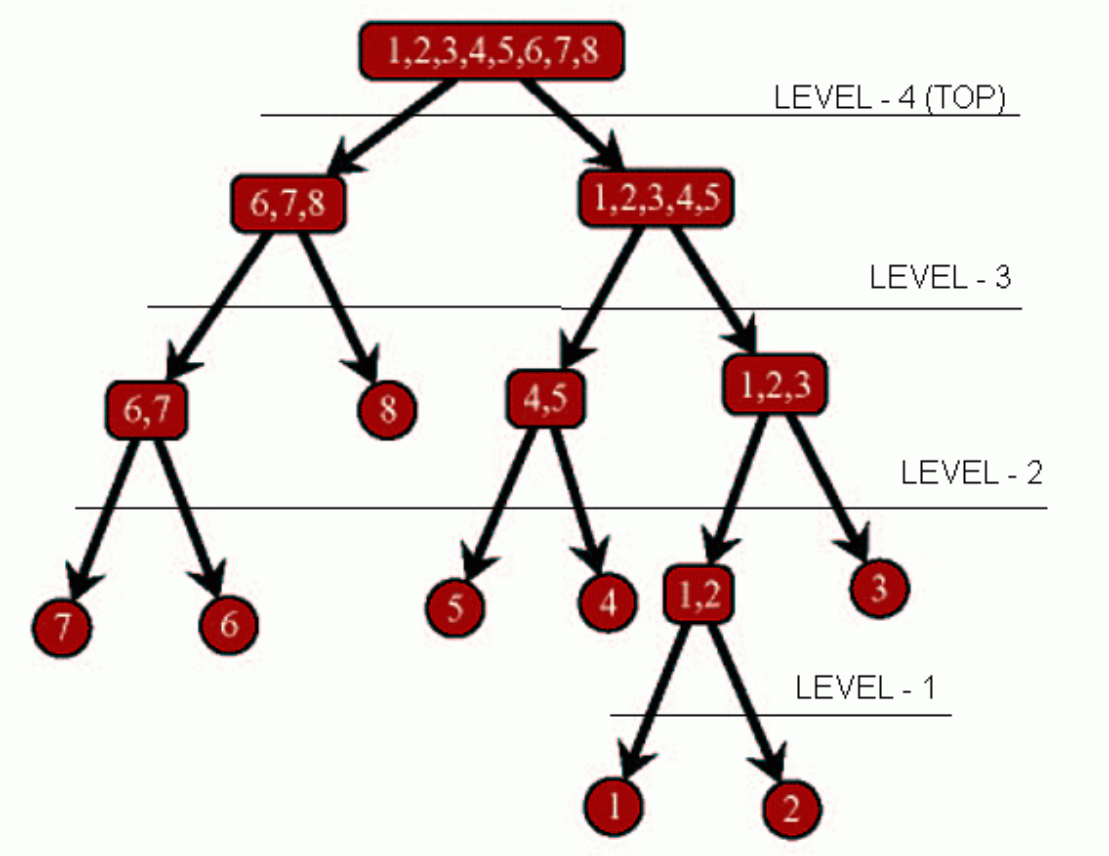
Στο κεφάλαιο αυτό, για να γίνει πιο κατανοητή η γενική ιδέα γύρω από τις προσεγγίσεις που προτείνονται στην εργασία, για κάθε μία, παρουσιάζουμε το θεωρητικό υπόβαθρο, αναλύουμε τη συλλογιστική πορεία μέσα τον αντίστοιχο αλγόριθμο και εξηγούμε τη λειτουργία της με ένα παράδειγμα.

## 3.2 Clustering Based Approach (CBA)

### *3.2.1 Θεωρητικό υπόβαθρο*

Η CBA προσεγγίζει το πρόβλημα που περιγράφηκε στην ενότητα 1.2, χρησιμοποιώντας ιεραρχική ενωτική ομαδοποίηση (Agglomerative Hierarchical Clustering). Ουσιαστικά, πρόκειται για μια επέκταση της τεχνικής που παρουσιάζεται στην [1]. Η βασική ιδέα της ιεραρχικής ενωτικής ομαδοποίησης, είναι ότι ξεκινάει με μια ξεχωριστή ομάδα (cluster) για κάθε οντότητα και, επαναληπτικά, συγχωνεύει τις δύο ομάδες με την μικρότερη απόσταση μεταξύ τους. Τα βήματα που ακολουθεί ο αλγόριθμος της ιεραρχικής ενωτικής ομαδοποίησης (Σχήμα 3.1) είναι:

1. Άρχισε με Ν ομάδες, όπου η κάθε ομάδα περιέχει μία οντότητα.
2. Για κάθε ομάδα, υπολόγισε, με βάση μια μετρική απόστασης, την απόσταση μεταξύ της κάθε οντότητάς της και όλων των οντοτήτων των υπόλοιπων ομάδων.
3. Με βάση τους υπολογισμούς που προέκυψαν από το βήμα 2 και ένα κριτήριο απόστασης, υπολόγισε τις αποστάσεις μεταξύ όλων των ομάδων.
4. Με βάση τους υπολογισμούς που προέκυψαν από το βήμα 3, υπολόγισε το ζευγάρι ομάδων με τη μικρότερη απόσταση.
5. Συγχώνευσε το ζευγάρι που έχει προκύψει από το βήμα 4.
6. Αν υπάρχουν περισσότερες από μία ομάδες στην τρέχουσα ομαδοποίηση (δηλαδή, N>1) πήγαινε στο βήμα 2.

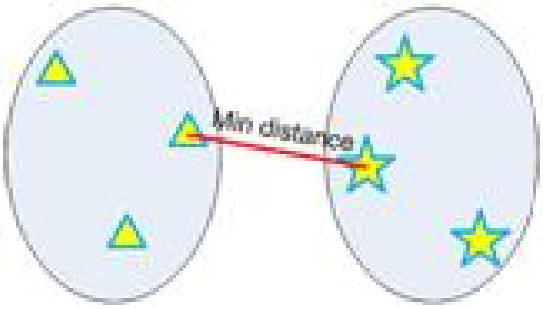



Σχήμα 3.1 Ιεραρχική Ενωτική Ομαδοποίηση.

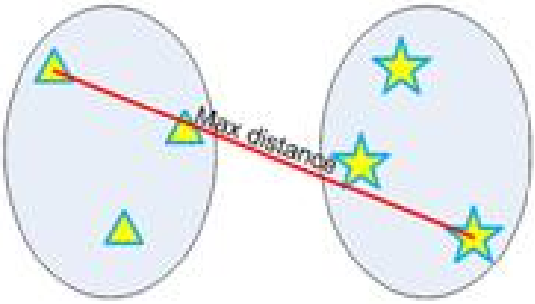
Για το βήμα 2, δηλαδή για τον υπολογισμό της απόστασης μεταξύ οντοτήτων, υπάρχουν διάφορες μετρικές. Στην CBA, χρησιμοποιείται ως μετρική, η απόσταση Jaccard, που αναλύεται στη συνέχεια.

Για το βήμα 3, δηλαδή για τον υπολογισμό της απόστασης μεταξύ των ομάδων, υπάρχουν διάφορα κριτήρια. Στην CBA, χρησιμοποιούνται πέντε (5) κριτήρια:

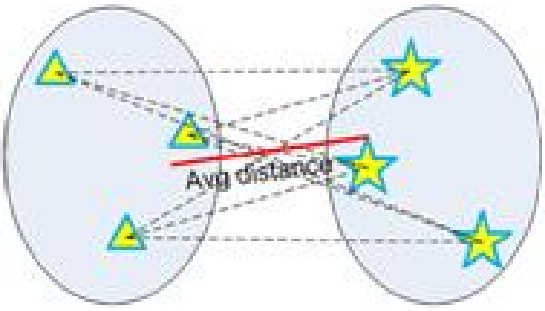
- Το κριτήριο του κοντινότερου γείτονα (Single Linkage Criterion). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο ομάδων ισούται με την ελάχιστη απόσταση μεταξύ των οντοτήτων των δύο ομάδων (Σχήμα 3.2).
- Το κριτήριο του μακρινότερου γείτονα (Complete Linkage Criterion). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο ομάδων ισούται με τη μέγιστη απόσταση μεταξύ των οντοτήτων των δύο ομάδων (Σχήμα 3.3).
- Το κριτήριο του μέσου όρου των αποστάσεων των γειτόνων (Average Linkage Criterion). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο ομάδων ισούται με τη μέση τιμή των αποστάσεων μεταξύ των οντοτήτων των δύο ομάδων (Σχήμα 3.4).
- Το κριτήριο του διάμεσου γείτονα (Median Linkage Criterion). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο ομάδων ισούται με τη διάμεση τιμή των αποστάσεων μεταξύ των οντοτήτων των δύο ομάδων.
- Το κριτήριο της προσαρμοστικής απόστασης (Adaptive Linkage Criterion). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο ομάδων ισούται με μια από τις προηγούμενες αποστάσεις, ανάλογα με την ισχύ κάποιων συνθηκών [1].




Σχήμα 3.2 Κανόνας του κοντινότερου γείτονα.




Σχήμα 3.3 Κανόνας του μακρινότερου γείτονα.




Σχήμα 3.4 Κανόνας του μέσου όρου των αποστάσεων.

*3.2.2 Αλγόριθμος*

Η CBA υλοποιείται από μια επαναληπτική διαδικασία, δύο φάσεων, για όλες τις κλάσεις του συστήματός. Κατά την πρώτη φάση (Αλγόριθμος 1), δημιουργούνται οι διεπαφές της κάθε κλάσης, με βάση τον ιεραρχικό ενωτικό αλγόριθμο. Κατά τη

δεύτερη φάση (Αλγόριθμος 2), επιλέγονται οι διεπαφές της κάθε κλάσης, που αντιστοιχούν σε κάθε χρήστη της κλάσης.

Η πρώτη φάση της CBA, δέχεται ως είσοδο μια δομή δεδομένων, *UPM (UsersPerMethod)*, που περιέχει την πληροφορία, ποιοι είναι οι χρήστες της κλάσης, ανά μέθοδο της κλάσης. Δηλαδή, οι χρήστες της κλάσης είναι ομαδοποιημένοι ανά μέθοδο που χρησιμοποιεί ο καθένας. Η πληροφορία αυτή, έχει παραχθεί από τη συντακτική ανάλυση του προγράμματος η οποία περιγράφεται στο κεφάλαιο 4. Η έξοδος της πρώτης φάσης της CBA, είναι μια λίστα, *L_All_I*, με όλες τις διεπαφές που έχουν δημιουργηθεί.

Κατά την πρώτη φάση της CBA, αρχικά δημιουργούνται τόσες στοιχειώδεις διεπαφές, όσες είναι οι μέθοδοι του *UPM*. Κάθε στοιχειώδης διεπαφή περιέχει μία μόνο μέθοδο. Επιπλέον, η αρχική ομαδοποίηση αποθηκεύεται σε μια λίστα, *L_Init* (γραμμές 3-6). Στη συνέχεια, η λίστα *L_Init* αποθηκεύεται σε μια λίστα μετώπου, *L_Front*. Η λίστα μετώπου με τη σειρά της προστίθεται στη λίστα *L_All_I*. Η λίστα μετώπου περιέχει τις διεπαφές μεταξύ των οποίων θα υπολογίζεται η απόσταση σε κάθε βήμα της επαναληπτικής διαδικασίας που ακολουθεί (γραμμές 9-39). Για τον υπολογισμό της απόστασης κάθε ζεύγους διεπαφών, $I_1$, $I_2$, υπολογίζονται όλες οι αποστάσεις Jaccard, *d*, μεταξύ των μεθόδων των δύο διεπαφών (γραμμές 13-18).

Ορίζουμε την απόσταση *d*, μεταξύ δύο μεθόδων, s και t, ως την απόσταση Jaccard, $J_d$ μεταξύ των συνόλων, $U_s$ και $U_t$, των κλάσεων-χρηστών τους, ως:

$$d(s,t) = J_d(U_s, U_t) = 1 - \frac{|U_s \cap U_t|}{|U_s \cup U_t|} \qquad \text{Εξ. 3.1}$$

Είναι προφανές ότι η απόσταση *d*, παίρνει τιμές από 0, μέχρι 1. Ισχύει *d=0* όταν οι μέθοδοι s και t έχουν τα ίδια σύνολα κλάσεων-χρηστών και *d=1* όταν δεν έχουν καμία κοινή κλάση-χρήστη.

Η κάθε απόσταση *d*, αποθηκεύεται σε μια λίστα, *L_Method_Distances* (γραμμή 16). Η απόσταση, *sd*, μεταξύ των δύο διεπαφών, $I_1$, $I_2$, υπολογίζεται από μια μέθοδο (γραμμή 19), *Select_Distance*, με ορίσματα τη λίστα των αποστάσεων των μεθόδων τους και ένα κριτήριο, *criterion*, που έχει επιλεγεί από τον χρήστη. Η CBA χρησιμοποιεί πέντε (5) κριτήρια:

- Το κριτήριο του κοντινότερου γείτονα (Single Linkage Criterion). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο διεπαφών, $I_1$, $I_2$, ισούται με την ελάχιστη απόσταση μεταξύ κάθε ζεύγους μεθόδων, *s, t*, των δύο διεπαφών: $d_{si}(I_1, I_2) = \min_{s in I_1, t in I_2} d(s,t)$. Επομένως, επιλέγεται το στοιχείο της λίστας *L_Method_Distances* με την ελάχιστη τιμή.
- Το κριτήριο του μακρινότερου γείτονα (Complete Linkage Criterion). ). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο διεπαφών, $I_1$, $I_2$, ισούται με τη μέγιστη απόσταση μεταξύ κάθε ζεύγους μεθόδων, *s, t*, των δύο διεπαφών: $d_{co}(I_1, I_2) = \max_{s in I_1, t in I_2} d(s,t)$. Επομένως, επιλέγεται το στοιχείο της λίστας *L_Method_Distances* με τη μέγιστη τιμή.
- Το κριτήριο του μέσου όρου των αποστάσεων των γειτόνων (Average Linkage Criterion). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο διεπαφών, $I_1$, $I_2$, ισούται με τη μέση τιμή των αποστάσεων μεταξύ κάθε ζεύγους μεθόδων, *s, t*, των δύο διεπαφών: $d_{av}(I_1, I_2) = \frac{\sum_{s in I_1, t in I_2} \delta(s,t)}{|I_1| * |I_2|}$. Επομένως, επιλέγεται η μέση τιμή των στοιχείων της λίστας *L_Method_Distances*.
- Το κριτήριο του διάμεσου γείτονα (Median Linkage Criterion). Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο διεπαφών, $I_1$, $I_2$, ισούται με τη διάμεση τιμή των αποστάσεων μεταξύ κάθε ζεύγους μεθόδων, *s, t*, των δύο διεπαφών: $d_{me}(I_1, I_2) = med\{d(s,t) \mid s in I_1, t in I_2\}$. Επομένως, επιλέγεται η διάμεση τιμή των στοιχείων της λίστας *L_Method_Distances*.
- Το κριτήριο της προσαρμοστικής απόστασης (Adaptive Linkage Criterion) [1]. Με βάση αυτό το κριτήριο, η απόσταση μεταξύ δύο διεπαφών, $I_1$, $I_2$, ισούται με μια από τις προηγούμενες αποστάσεις, ως εξής:

$$d_{ad}(I_1, I_2) = \begin{cases} d_{co}(I_1, I_2), & if\ 0 \le d_{co}(I_1, I_2) < 1 \\ d_{me}(I_1, I_2), & if\ d_{co}(I_1, I_2) = 1 \wedge 0 \le d_{me}(I_1, I_2) < 1 \\ d_{av}(I_1, I_2), & if\ d_{me}(I_1, I_2) = 1 \wedge 0 \le d_{av}(I_1, I_2) < 1 \\ d_{si}(I_1, I_2), & if\ d_{av}(I_1, I_2) = 1 \wedge\ 0 \le d_{si}(I_1, I_2) < 1 \end{cases}$$

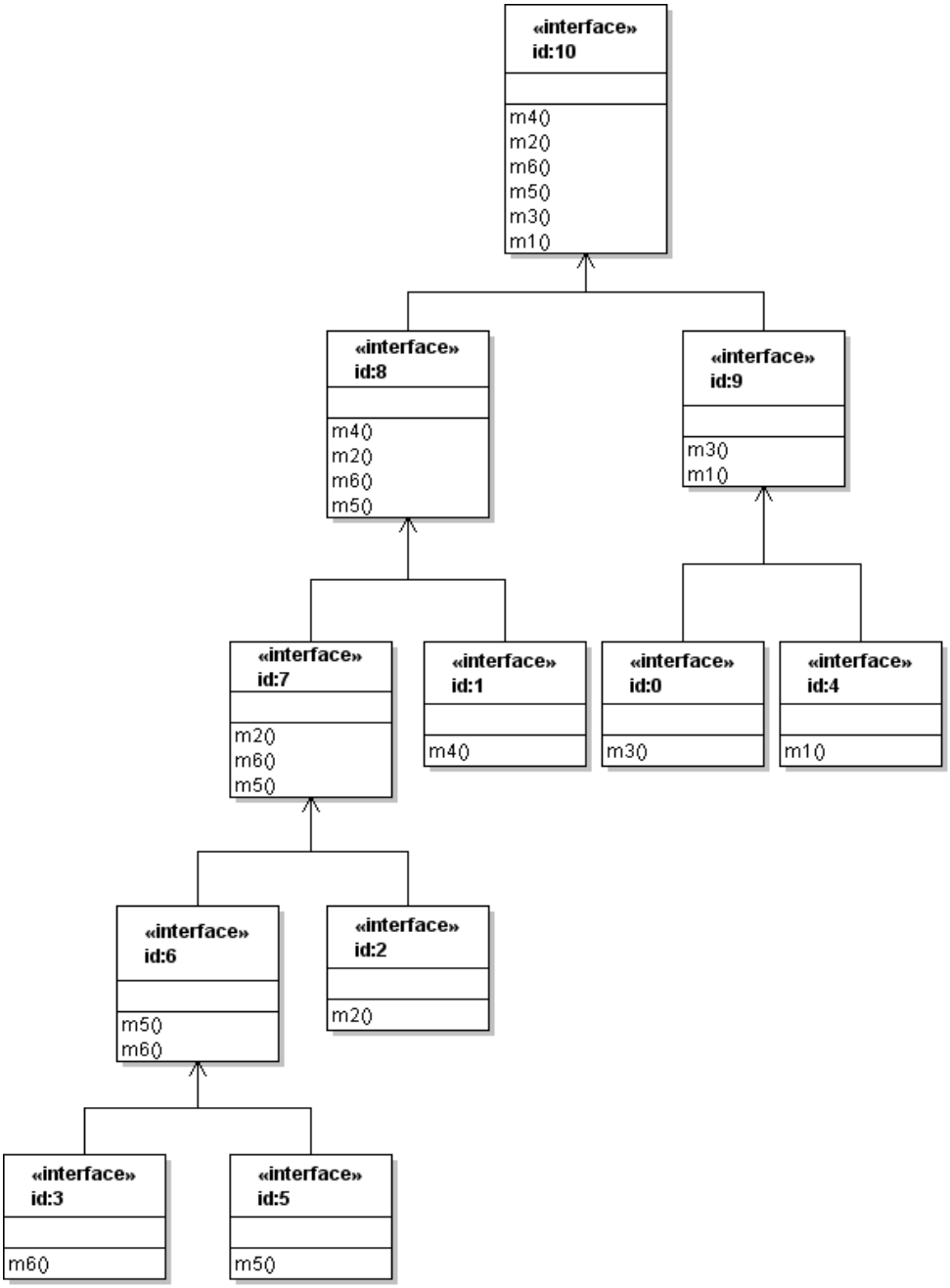
Στη συνέχεια, ο ιεραρχικός ενωτικός αλγόριθμος, υπολογίζει την ελάχιστη απόσταση μεταξύ όλων των διεπαφών της λίστας μετώπου. Επιπλέον, αποθηκεύει το ζεύγος των διεπαφών με την ελάχιστη τιμή απόστασης, *min_I1, min_I2*, καθώς επίσης και τη συγκεκριμένη τιμή, *fd* (γραμμές 20-24). Η αρχική τιμή της μεταβλητής *fd*, ισούται με 1 (γραμμή 10), καθώς είναι η μέγιστη απόσταση μεταξύ δύο διεπαφών, σύμφωνα με την Εξίσωση (3.1). Αν η τελική τιμή της *fd*, είναι μικρότερη από 1 (γραμμή 27), γίνονται οι κατάλληλες ενημερώσεις, αλλιώς (*fd=1)*, τερματίζει ο βρόγχος (γραμμές 37-38). Στην πρώτη περίπτωση, δημιουργείται μια νέα διεπαφή, στην οποία εισάγονται οι μέθοδοι των διεπαφών που συγχωνεύονται (γραμμές 28 και 31). Η νέα διεπαφή προστίθεται στη λίστα μετώπου και στη λίστα *L_All_I* (γραμμή 31). Επιπρόσθετα, οι διεπαφές, *min_I1, min_I2,* εξάγονται από τη λίστα μετώπου (γραμμή 36), καθώς, μετά τη συγχώνευσή τους δεν θα πρέπει πλέον να λαμβάνονται υπόψη στον υπολογισμό των αποστάσεων μεταξύ διεπαφών. Η όλη διαδικασία τερματίζει όταν το μέγεθος της λίστας μετώπου γίνει ίσο με 1 (γραμμή 9), αφού δεν έχει νόημα ο υπολογισμός απόστασης μεταξύ ενός πλήθους διεπαφών, μικρότερο του 2. Τελικά, επιστρέφεται η λίστα, *L_All_I*, που περιέχει όλες τις διεπαφές που έχουν δημιουργηθεί κατά την πρώτη φάση της CBA.

```
Input : UPM : Multimap
Output : L_All_I : List < Interface >
1. L_Init : List < Interface >
2. L_Front : List < Interface >
   //Begin with singleton interfaces
3. for all Method_Decl in UPM do
4.         i = Create_Interface()
5.         L_Init.add(i)
6. end for
7. L_Front.addAll(L_Init)
8. L_All_I.addAll(L_Front)
   //Compute distances between interfaces
9.  while (| L_Front |> 1) do
10.        fd = 1
11.        for all I1 in L_Front do
12.                for all I2 in L_Front do
13.                        for all Method_Decl1 in I1 do
14.                                for all Method_Decl2 in I2 do
15.                                        d = Jaccard(Method_Decl1 , Method_Decl2)
16.                                        L_Method_Distances.add(d)
17.                                end for
18.                        end for
19.                        sd = Select_Distance(L_Method_Distances,criterion)
                           // Find interfaces with minimum distance
20.                        if (sd < fd) then
21.                                fd = sd
22.                                min_I1 = I1
23.                                min_I2 = I2
24.                        end if
25.                end for
26.        end for
```

```
27.     if (fd < 1) then // merge interfaces with minimum distance and update L_Front,L_All
28.             m_I = Create_Interface()
29.             m_I.child1= min_I1
30.             m_I.child2= min_I2
31.             m_I.addAll(min_I1.ListOfMethod_Decl , min_I2.ListOfMethod_Decl)
32.             min_I1.parent = m_I
33.             min_I2.parent = m_I
34.             L_Front.add(m_I)
35.             L_All_I.add(m_I)
36.             L_Front.remove(min_I1 , min_I2)
37.     else // stop clustering if fd == 1
38.     break
39. end while
40. return L_All_I
```

Αλγόριθμος 1. Πρώτη φάση της CBA.

Σύμφωνα με τα παραπάνω, η CBA θα έχει ως αποτέλεσμα η αρχική σχεδίαση του Σχήματος 1.1 να μετατραπεί όπως φαίνεται στο Σχήμα 3.5. Στη συγκεκριμένη περίπτωση, έχει επιλεγεί το κριτήριο του κοντινότερου γείτονα.




Σχήμα 3.5 Σχεδίαση μετά την εφαρμογή της CBA με το κριτήριο single linkage.

Η δεύτερη φάση της CBA δέχεται ως είσοδο μια δομή δεδομένων, *MPU (MethodsPerUser)*, που περιέχει την πληροφορία, ποιες είναι οι μέθοδοι της κλάσης που χρησιμοποιεί κάθε κλάση-χρήστης. Η πληροφορία αυτή, έχει παραχθεί από τη συντακτική ανάλυση του προγράμματος η οποία περιγράφεται στο κεφάλαιο 4. Επιπλέον, δέχεται ως είσοδο, τη λίστα, *L_All_I*, που περιέχει όλες τις διεπαφές που έχουν δημιουργηθεί κατά την πρώτη φάση της CBA. Στόχος της δεύτερης φάσης,

είναι να επιλεχθούν για κάθε κλάση-χρήστη μιας κλάσης οι διεπαφές που χρειάζεται. Δηλαδή, να επιλεχθούν οι διεπαφές που περιέχουν τις μεθόδους της κλάσης που χρησιμοποιεί η κάθε κλάση-χρήστης. Η επιλογή αυτή μπορεί να πραγματοποιηθεί με δύο τρόπους.

Ο πρώτος τρόπος είναι να επιλεγεί ο ελάχιστος αριθμός διεπαφών που περιέχουν τις μεθόδους που χρησιμοποιεί η κάθε κλάση-χρήστης. Στην περίπτωση αυτή όμως, είναι πιθανόν οι διεπαφές που επιλέγονται, να περιέχουν, εκτός από τις παραπάνω μεθόδους, μεθόδους που δεν χρησιμοποιεί η εκάστοτε κλάση-χρήστης. Με αυτόν τον τρόπο επιλογής, ελαχιστοποιείται η σύζευξη της κάθε κλάσης με τις αντίστοιχες κλάσεις-χρήστες της, αλλά με το πιθανό κόστος ανάθεσης περισσότερων μεθόδων από όσες χρειάζεται η κάθε κλάση-χρήστης.

Ο δεύτερος τρόπος είναι να επιλεγεί το απαιτούμενο πλήθος διεπαφών μιας κλάσης, έτσι ώστε το σύνολό τους, να περιέχει ακριβώς όσες μεθόδους χρησιμοποιεί η κάθε κλάση-χρήστης. Στην περίπτωση αυτή όμως, είναι πιθανόν το πλήθος των διεπαφών που επιλέγονται να είναι μεγαλύτερο από ό,τι με τον πρώτο τρόπο επιλογής. Με τον δεύτερο τρόπο επιλογής, ελαχιστοποιείται (για την ακρίβεια μηδενίζεται) το πλήθος των ανατιθέμενων μεθόδων μιας κλάσης χωρίς να τις χρησιμοποιεί η κάθε κλάση-χρήστης, αλλά με το πιθανό κόστος της μεγαλύτερης σύζευξης σε σχέση με τον πρώτο τρόπο επιλογής.

Επομένως, παρατηρούμε ότι σε κάθε τρόπο επιλογής των απαιτούμενων διεπαφών εμφανίζεται ένας συμβιβασμός (trade-off) μεταξύ σύζευξης και πλήθους περιττών ανατιθέμενων μεθόδων, για κάθε κλάση-χρήστη. Για να ποσοτικοποιήσουμε τα εν λόγω μεγέθη, θα περιγράψουμε τις μετρικές ACD και CBO που χρησιμοποιούνται για την επιλογή των διεπαφών για κάθε κλάση-χρήστη.

Καταρχήν, με τον όρο σύζευξη (coupling) εννοούμε το βαθμό εξάρτησης μεταξύ κλάσεων και γενικότερα συστατικών ενός λογισμικού που μπορεί να προκύπτει από διάφορες αιτίες, όπως κλήση μεθόδων και πέρασμα παραμέτρων από το ένα συστατικό, σε ένα άλλο. Υπάρχει ένα εύρος μετρικών για την ποσοτικοποίηση της

σύζευξης, των οποίων χρησιμοποιείται στην παρούσα διατριβή η Coupling Between Objects classes (CBO) [2]. Για μια κλάση-χρήστη Α, ισχύει ότι η τιμή της CBO ισούται με το πλήθος των κλάσεων που χρησιμοποιεί η Α. Όσο μεγαλύτερη είναι η τιμή της CBO για μια κλάση, τόσο φθίνει μια σειρά από παράγοντες που επηρεάζουν την ποιότητα του λογισμικού. Το κόστος της συντήρησης της κλάσης αυξάνεται, διότι εξαρτάται από πολλές άλλες κλάσεις και επηρεάζεται συχνά από αλλαγές. Το κόστος για τον έλεγχό της αυξάνεται, εξαιτίας της εξάρτησής της από τις άλλες κλάσεις. Τέλος, η επαναχρησιμοποίηση της γίνεται πιο δύσκολη, καθώς η λειτουργία της είναι πολύ συγκεκριμένη και εξαρτάται και από τη λειτουργία των άλλων κλάσεων. Επομένως, είναι λιγότερο πιθανό να μπορεί να χρησιμοποιηθεί εύκολα για την επίλυση και κάποιου άλλου προβλήματος. Συμπεραίνουμε λοιπόν ότι όσο μικρότερη είναι η τιμή της CBO, τόσο καλύτερη είναι η σχεδίαση, άρα και η ποιότητα του λογισμικού.

Για μια κλάση Α που χρησιμοποιεί μεθόδους μιας κλάσης Β, η Actual Context Distance (ACD) [11] ορίζεται ως εξής:

$$ACD_{(A)} = \frac{p_{(B)} - i_{(A)}}{p_{(B)}} \qquad \text{Εξ. 3.2}$$

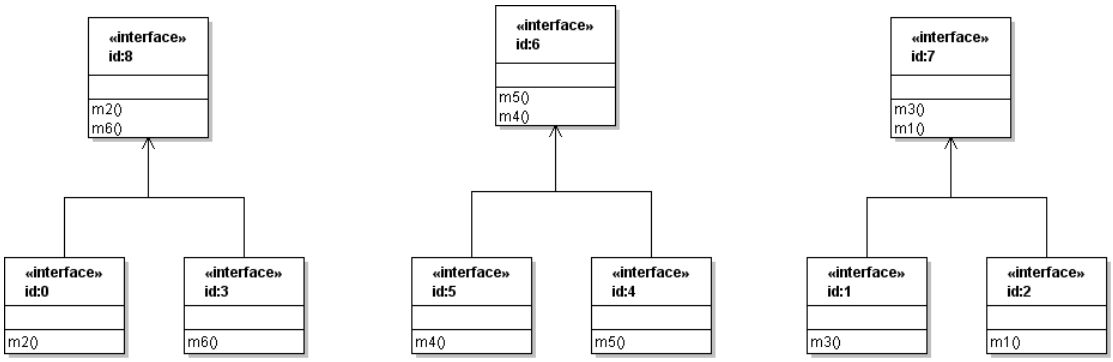
Όπου, ο όρος $p_{(B)}$ εκφράζει το πλήθος των μεθόδων της κλάσης Β που ανατίθενται στην κλάση Α, πριν την επιλογή των διεπαφών για την κλάση Α, δηλαδή το σύνολο όλων των μεθόδων της κλάσης Β. Ο όρος $i_{(A)}$ εκφράζει το πλήθος των μεθόδων που ανατίθενται στην κλάση Α, μετά την επιλογή των διεπαφών για την κλάση Α. Δηλαδή, η ACD δείχνει την ποσοστιαία μείωση των μεθόδων που ανατίθενται στην κλάση Β, με βάση τις επιλεχθέντες διεπαφές. Συμπεραίνουμε λοιπόν ότι, γενικά, όσο μεγαλύτερη είναι η τιμή της ACD, τόσο καλύτερη είναι η ποιότητα του λογισμικού.

Κατά τη δεύτερη φάση της CBA, αρχικά, για κάθε κλάση-χρήστη μιας κλάσης, δημιουργείται μια διεπαφή, στην οποία προστίθενται οι μέθοδοι της κλάσης που χρησιμοποιεί η κλάση-χρήστης (γραμμές 1-7). Δηλαδή, δημιουργούνται οι διεπαφές που αντιστοιχούν σε κάθε κλάση-χρήστη της κλάσης. Κάτι τέτοιο είναι εφικτό, μέσω

της δομής δεδομένων, *MPU*. Η λίστα, *L_All_I*, έχει δενδρική μορφή, εξαιτίας των συγχωνεύσεων των διεπαφών. Για το λόγο αυτό αποθηκεύονται όλες οι διεπαφές, $I_1$, της εν λόγω λίστας, που είναι ρίζες υπο-δένδρων (θεωρούμε μια διεπαφή-απομονωμένο κόμβο ως υπο-δένδρο μηδενικού ύψους) στη λίστα *L_RootsAndIsolated*. Συνεχίζοντας την ανάλυση της δεύτερης φάσης του αλγορίθμου της CBA, διακρίνουμε τις δύο περιπτώσεις όσον αφορά την επιλογή των διεπαφών για κάθε χρήστη. Στην πρώτη περίπτωση (γραμμές 14-28), η επιλογή γίνεται με τέτοιον τρόπο ώστε η μεγάλη τιμή της CBO να αντισταθμίζεται από μεγάλη τιμή της ACD. Επίσης, υπολογίζεται το συνολικό πλήθος, *p*, των μεθόδων που προσφέρει η κλάση (γραμμή 14). Για κάθε κλάση που χρησιμοποιεί μια άλλη κλάση, πραγματοποιείται η επιλογή των διεπαφών που της αντιστοιχούν με τη μέθοδο *Select_Max_Interfaces* (γραμμή 19). Η μέθοδος αυτή, ξεκινά την αναζήτηση των κατάλληλων διεπαφών από τη ρίζα κάθε υπο-δένδρου και καλείται αναδρομικά για το δεξί και το αριστερό παιδί της. Η μέθοδος επιστρέφει όταν εντοπίσει διεπαφές που περιέχουν μόνο μεθόδους που χρησιμοποιεί η κλάση-χρήστης. Η διαδικασία αυτή, είναι δυνατό να συνεχιστεί μέχρι τις τετριμμένες ομάδες-φύλλα. Η λίστα *L_Max_Selected_I*, έχει ως αποτέλεσμα τη λίστα των διεπαφών που αντιστοιχούν στην κλάση-χρήστη. Οι διεπαφές της λίστας *L_Max_Selected_I* προστίθενται στη λίστα *totalMax_I* και χρησιμοποιείται για την περίπτωση κατά την οποία το σύνολο των μεθόδων της κλάσης χρήστη καλύπτεται από διεπαφές που ανήκουν σε διαφορετικά υπο-δένδρα. Εν συνεχεία, υπολογίζονται τα $ACD_{max}$ και $CBO_{max}$ (γραμμές 26-27), αφού υπολογιστεί το πλήθος των μεθόδων των διεπαφών που έχουν επιλεγεί για τον κάθε χρήστη, *i*. Γίνεται αντιληπτό ότι, με τον τρόπο που έχει γίνει η επιλογή των εν λόγω διεπαφών η τιμή του *i* θα ισούται με το πλήθος των μεθόδων της κλάσης, που χρησιμοποιεί η κλάση-χρήστης. Αυτός είναι ο λόγος που η τιμή της ACD αναμένεται να είναι μεγαλύτερη από την περίπτωση που ακολουθεί, καθώς από το συνολικό πλήθος των μεθόδων της κλάσης provider αφαιρούνται μόνο τόσες μέθοδοι, όσες χρησιμοποιεί η κλάση client.

Για να γίνει ξεκάθαρη η παραπάνω διαδικασία, θα την εξετάσουμε με ένα παράδειγμα. Στο Σχήμα 3.6 φαίνεται η μετατροπή της αρχικής σχεδίασης του Σχήματος 1.1 μετά την εφαρμογή της CBA με το κριτήριο complete linkage.

Γνωρίζουμε ότι στο παράδειγμά μας, με βάση την υλοποίηση του συστήματος του Σχήματος 1.1 η κλάση Client D χρησιμοποιεί τις μεθόδους m2(), m5() και m6() της κλάσης Provider. Σύμφωνα με την παραπάνω διαδικασία, η μέθοδος *Select_Max_Interfaces* ξεκινά την αναζήτηση των διεπαφών που περιέχουν μόνο μεθόδους της κλάσης Client D από τη ρίζα κάθε υπο-δένδρου. Οι ρίζες αυτές (διεπαφές με id 6,7,8) ανήκουν στη λίστα *L_RootsAndIsolated*. Επομένως, ξεκινώντας από τη διεπαφή με id 8, θα επιστρέψει τη διεπαφή με id 8 που περιέχει τις μεθόδους m2(), m6() και ξεκινώντας από τη διεπαφή με id 6, θα επιστρέψει τη διεπαφή με id 4 που περιέχει τη μέθοδο m5().




Σχήμα 3.6 Σχεδίαση μετά την εφαρμογή της CBA με το κριτήριο complete linkage.

Στη δεύτερη περίπτωση (γραμμές 29-42), η επιλογή γίνεται με τέτοιον τρόπο ώστε η μικρή τιμή της ACD να αντισταθμίζεται από μικρή τιμή της CBO. Για κάθε κλάση-χρήστη που χρησιμοποιεί μια άλλη κλάση, πραγματοποιείται η επιλογή των διεπαφών που της αντιστοιχούν σύμφωνα με έναν έλεγχο (γραμμή 32). Εάν μια διεπαφή που ανήκει στη λίστα *L_RootsAndIsolated* περιέχει τουλάχιστον μία μέθοδο από κοινού με τις μεθόδους που χρησιμοποιεί η κλάση client, τότε η διεπαφή αυτή προστίθεται στη λίστα *L_Min_Selected_I*. Όπως γίνεται αντιληπτό, πιθανότατα, η διεπαφή ή οι διεπαφές που επιλέγονται με αυτόν τον τρόπο περιέχουν, επιπλέον, μεθόδους που δεν χρησιμοποιούνται από την κλάση client. Αυτός είναι ο λόγος που η τιμή της ACD αναμένεται να είναι μικρότερη ό,τι στην προηγούμενη περίπτωση, καθώς από το συνολικό πλήθος των μεθόδων της κλάσης provider αφαιρούνται τόσες μέθοδοι, όσες

χρησιμοποιεί η κλάση client και ενδεχομένως ορισμένες επιπλέον που δεν χρησιμοποιεί.

Για να γίνει ξεκάθαρη η παραπάνω διαδικασία, θα την εξετάσουμε με το ίδιο παράδειγμα. Στο Σχήμα 3.6 φαίνεται η μετατροπή της αρχικής σχεδίασης του Σχήματος 1.1 μετά την εφαρμογή της CBA με το κριτήριο complete linkage. Σύμφωνα με την παραπάνω διαδικασία, ο έλεγχος (γραμμή 32) θα έχει ως αποτέλεσμα για την κλάση Client D να επιλεγούν οι διεπαφές με id 8 και 6. Παρατηρούμε ότι, από τις διεπαφές που επιλέχθηκαν, εκείνη με id 8 περιέχει μόνο μεθόδους που χρησιμοποιεί η κλάση Client D. Αντίθετα, η διεπαφή με id 6, εκτός από την μέθοδο m5(), που χρησιμοποιεί η κλάση Client D, περιέχει και άλλες, επιπλέον μεθόδους που δεν χρησιμοποιεί (τη μέθοδο m4()).

```
Input : MPU : Multimap. L_All_I : List < Interface >
   // Create interfaces for the users of the class with the methods they use
1. for all User in MPU do
2.         i = Create_Interface()
3.         for all Method_Inv in User do
4.                   i.add(Method_Inv)
5.         end for
6.         L_UsersOfClass.add(i)
7.  end for
   // Add root and isolated interfaces in a list
8.  L_RootsAndIsolated :List < Interfaces >
9. for all I_1 in L_All_I do
10.        if (I_1.Parent == null) then
11.                  L_RootsAndIsolated.add(I_1)
12.        end if
13.  end for
     // Select max no.of interfaces but max improvement
14.     p = | Method_DeclOfClass |
15.  for all I_1 in L_UsersOfClass do
16.        L_Max_Selected_I : List < Interface >
17.        totalMax_I : List < Interface >
18.        for all I_2 in L_RootsAndIsolated do
19.                  L_Max_Selected_I = Select_Max_Interfaces(I_1, I_2)
20.                  totalMax_I.addAll(L_Max_Selected_I)
22.        end for
22.        i = 0
23.        for all I_1 in totalMax_I do
24.                  i = i + |I_1.ListOfMethod_Decl |
25.        end for
26.        ACD_max = (p - i) / p
27.        CBO_max = | totalMax_I |
28.  end for
```

```
// Select min number of interfaces but min imporovement
29. for all I1 in L_UsersOfClass do
30.     L_Min_Selected_I : List < Interface >
31.     for all I2 in L_RootsAndIsolated do
32.             if (|I1.ListOfMethod_Decl ∩ I2.ListOfMethod_Decl |>0) then
33.                     L_Min_Selected_I.add(I2)
34.             end if
35.     end for
36.   i = 0
37.   for all I1 in L_Min_Selected_I) do
38.             i = i+ |I1.ListOfMethod_Decl |
39.   end for
40.   ACDmin = (p − i) / p
41.   CBOmin =| L_Min_Selected_I |
42. end for
```

Αλγόριθμος 2. Δεύτερη φάση της CBA.

## 3.3 Refactoring Based Approach (RBA)

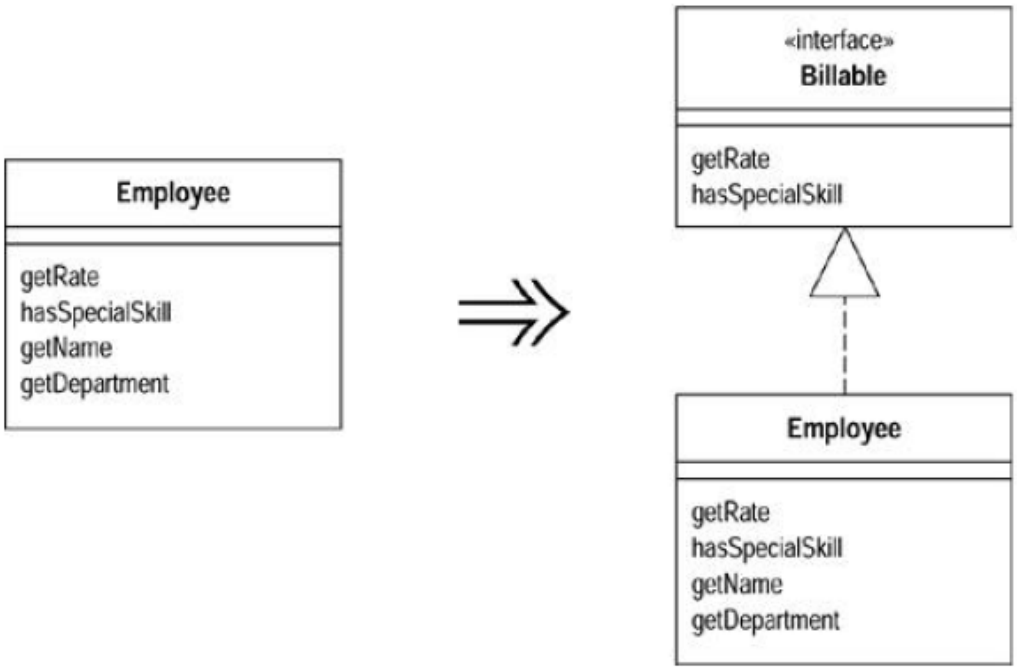
*3.3.1 Θεωρητικό υπόβαθρο*

Η RBA αποτελείται από δύο φάσεις που κάθε μία βασίζεται σε μία τεχνική ανακατασκευής. Η πρώτη φάση της εμπνέεται από την Εξαγωγή Διεπαφής (Extract Interface) και η δεύτερη από την Εξαγωγή Υπερκλάσης (Extract Superclass). Έχει γίνει ήδη αναφορά στις δύο τεχνικές στην υποενότητα 2.4.5, ωστόσο εδώ θα τις μελετήσουμε πιο αναλυτικά εξαιτίας της σημασίας τους στον αλγόριθμο της RBA.

Η Εξαγωγή Διεπαφής λύνει το πρόβλημα της χρησιμοποίησης του ίδιου συνόλου public μεθόδων μιας κλάσης από διάφορες άλλες κλάσεις. Επίσης, αντιμετωπίζει την περίπτωση που δύο ή περισσότερες κλάσεις έχουν public μεθόδους με τα ίδια πρωτότυπα. Η συγκεκριμένη τεχνική, προτείνει τη δημιουργία μιας διεπαφής που περιλαμβάνει τις εν λόγω μεθόδους. Τα βήματα που ακολουθεί είναι τα εξής:

- Δημιουργία μιας διεπαφής.
- Ορισμός των κοινών μεθόδων στη διεπαφή.
- Μετατροπή των αρχικών κλάσεων με τέτοιον τρόπο, ώστε να υλοποιούν τη διεπαφή.
- Μετατροπή των κλάσεων που χρησιμοποιούν μόνο τις κοινές μεθόδους των αρχικών κλάσεων με τέτοιον τρόπο, ώστε να χρησιμοποιούν τη διεπαφή, αντί για τις αρχικές κλάσεις.

Στο Σχήμα 3.7 φαίνεται ένα παράδειγμα εφαρμογής της τεχνικής. Αρχικά, διάφορες κλάσεις-χρήστες χρησιμοποιούν τις μεθόδους getRate() και hasSpecialSkill() της κλάσης Employee. Μετά την εφαρμογή προκύπτει η σχεδίαση του δεξιού τμήματος του Σχήματος 3.7. Πλέον, μετά τις κατάλληλες ενέργειες, οι κλάσεις-χρήστες χρησιμοποιούν τη διεπαφή Billable αντί της κλάσης Employee.

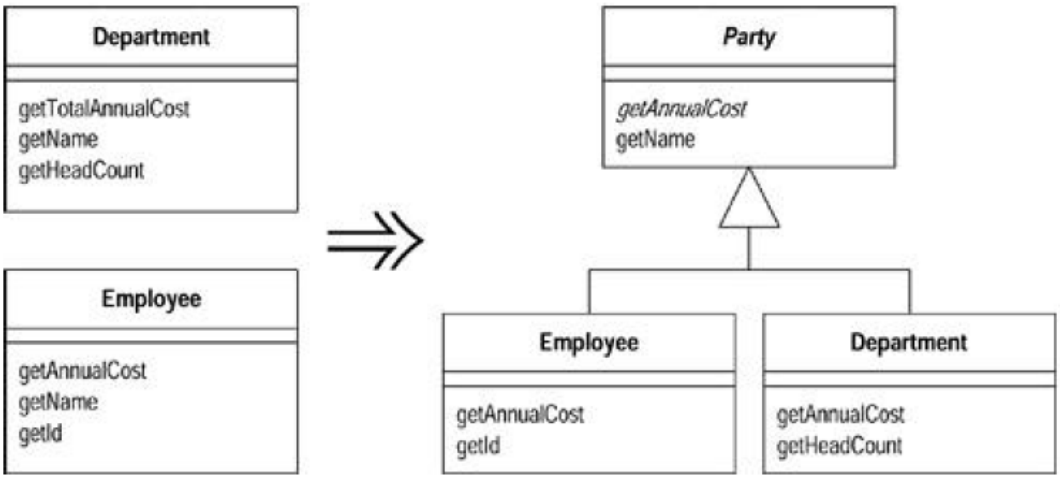



Σχήμα 3.7 Παράδειγμα εφαρμογής της τεχνικής ανακατασκευής: Εξαγωγή Διεπαφής.

Η Εξαγωγή Υπερκλάσης λύνει το πρόβλημα που εμφανίζεται όταν δύο ή περισσότερες κλάσεις έχουν παρόμοια συστατικά (πεδία ή/και μεθόδους). Το πρόβλημα αυτό, είναι πιθανό να προκύψει από την εφαρμογή της Εξαγωγής Διεπαφής. Η συγκεκριμένη τεχνική, προτείνει τον ορισμό μιας βασικής κλάσης και μετακίνηση των κοινών συστατικών σε αυτήν. Τα βήματα που ακολουθεί είναι τα εξής:

- Ορισμός μιας βασικής κλάσης και μετατροπή των αρχικών κλάσεων σε υποκλάσεις της νέας κλάσης.
- Σταδιακή μετακίνηση στη βασική κλάση των κοινών συστατικών με χρήση των τεχνικών ανακατασκευής Pull Up Method και Pull Up Field.
    - Αν στις υποκλάσεις τα συστατικά έχουνε παρόμοια ονόματα και πρωτότυπα, χρησιμοποίηση πρώτα των τεχνικών Rename Field και Rename Method.
    - Αν στις υποκλάσεις οι μέθοδοι έχουν ίδια πρωτότυπα, αλλά διαφορετική υλοποίηση, ορισμός στη βασική κλάση μιας αφηρημένης μεθόδου.
- Εκτέλεση μεταγλώττισης (compile) και ελέγχου (test) μετά από κάθε μετακίνηση.
- Εξέταση αν οι μέθοδοι που παρέμειναν στις υποκλάσεις έχουν κοινά τμήματα κώδικα. Στην περίπτωση που αυτό ισχύει, χρησιμοποίηση των τεχνικών Extract Method και Pull Up Method.
- Εξέταση των κλάσεων που χρησιμοποιούν τις αρχικές κλάσεις. Στην περίπτωση που χρησιμοποιούν μόνο τα κοινά συστατικά που πλέον ανήκουν στη βασική κλάση, χρησιμοποίηση της βασικής κλάσης, αντί των αρχικών κλάσεων.

Στο Σχήμα 3.8 φαίνεται ένα παράδειγμα εφαρμογής της τεχνικής. Αρχικά, οι κλάσεις Department και Employee έχουν κοινές τις μεθόδους getAnnualCost() και getName(). Μετά την εφαρμογή προκύπτει η σχεδίαση του δεξιού τμήματος του Σχήματος 3.8. Πλέον, μετά τις κατάλληλες ενέργειες, οι κλάσεις Department και Employee επεκτείνουν μια βασική κλάση Party που περιέχει τις κοινές μεθόδους τους.




Σχήμα 3.8 Παράδειγμα εφαρμογής της τεχνικής ανακατασκευής: Εξαγωγή Υπερκλάσης.

*3.3.2 Αλγόριθμος*

Η RBA υλοποιείται από μια επαναληπτική διαδικασία, δύο φάσεων, για όλες τις κλάσεις του συστήματός. (Αλγόριθμος 3). Κατά την πρώτη φάση, δημιουργούνται οι διεπαφές που αντιστοιχούν σε κάθε κλάση-χρήστης της κλάσης. Κάτι τέτοιο είναι εφικτό, μέσω μιας δομής δεδομένων, *MPU (MethodsPerUser)*, που περιέχει την πληροφορία, ποιες είναι οι μέθοδοι της κλάσης που χρησιμοποιεί κάθε κλάση-χρήστης. Η πληροφορία αυτή, έχει παραχθεί από τη συντακτική ανάλυση του προγράμματος η οποία περιγράφεται στο Κεφάλαιο 4. Κατά τη δεύτερη φάση, λύνεται το πρόβλημα της επανάληψης μεθόδων που πιθανόν να εμφανίζεται στις διεπαφές που δημιουργήθηκαν. Όταν δύο διεπαφές χρησιμοποιούν κοινές μεθόδους ορίζεται μια νέα βασική διεπαφή, στην οποία μετακινούνται οι κοινές μέθοδοι.

Πιο συγκεκριμένα, η CBA δέχεται ως είσοδο τη δομή δεδομένων, *MPU*, και η έξοδός της, είναι μια λίστα, *L_All_I*, με όλες τις διεπαφές που έχουν δημιουργηθεί.

Αρχικά, κατά την πρώτη φάση της RBA, για κάθε κλάση-χρήστη μιας κλάσης δημιουργείται μια διεπαφή, στην οποία προστίθενται οι μέθοδοι που χρησιμοποιεί η κλάση-χρήστης (γραμμές 3-7).

Κατά τη δεύτερη φάση της RBA, η κάθε διεπαφή προστίθεται σε μια λίστα *L_Init*. Στη συνέχεια, η λίστα *L_Init* αποθηκεύεται σε μια λίστα μετώπου, *L_Front*. Η λίστα μετώπου με τη σειρά της προστίθεται στη λίστα *L_All_I*. Η λίστα μετώπου περιέχει τις διεπαφές μεταξύ των οποίων θα υπολογίζεται η απόσταση σε κάθε βήμα της επαναληπτικής διαδικασίας που ακολουθεί (γραμμές 12-65). Για τον υπολογισμό της απόστασης κάθε ζεύγους διεπαφών, $I_1$, $I_2$, υπολογίζονται οι αποστάσεις Jaccard, *d*, μεταξύ των όλων των διεπαφών (γραμμές 15-17).

Ορίζουμε την απόσταση *d*, μεταξύ δύο διεπαφών, $I_1$ και $I_2$, ως την απόσταση Jaccard, $J_d$ μεταξύ των συνόλων, $U_1$ και $U_2$, των μεθόδων τους, ως:

$$d(I_1,I_2)=J_d(U_1,U_2)=1-\frac{|U_1\cap U_2|}{|U_1\cup U_2|} \qquad \text{Εξ. 3.3}$$

Είναι προφανές ότι η απόσταση *d*, παίρνει τιμές από 0, μέχρι 1. Ισχύει *d=0* όταν οι διεπαφές, $I_1$ και $I_2$, έχουν τα ίδια σύνολα μεθόδων και *d=1* όταν δεν έχουν καμία κοινή μέθοδο.

Στόχος είναι να υπολογιστεί το ζεύγος διεπαφών, *min_I1*, *min_I2*, με τη μικρότερη απόσταση, *fd*, μεταξύ τους (γραμμές 26-31). Σε περίπτωση που υπάρχουν παραπάνω του ενός ζεύγη διεπαφών με απόσταση ίση με την ελάχιστη, επιλέγεται, από αυτά, το ζεύγος που επιπλέον έχει και το μέγιστο πλήθος κοινών μεθόδων (γραμμές 19-25). Η αρχική τιμή της μεταβλητής *fd*, ισούται με 1 (γραμμή 13), καθώς είναι η μέγιστη απόσταση μεταξύ δύο διεπαφών, σύμφωνα με την Εξίσωση (3.3). Αν η τελική τιμή της *fd*, είναι μικρότερη από 1 (γραμμή 34), γίνονται οι κατάλληλες ενημερώσεις, αλλιώς (*fd=1)*, τερματίζει ο βρόγχος (γραμμές 63-64). Στην πρώτη περίπτωση, εξετάζεται το ενδεχόμενο οι επιλεχθείσες διεπαφές, *min_I1*, *min_I2*, να έχουν το ίδιο σύνολο μεθόδων. Εάν αυτό ισχύει, δεν δημιουργείται νέα διεπαφή, αλλά διαγράφεται από τις λίστες *L_Front*, *L_All_I* μία εκ των διεπαφών, *min_I1*, *min_I2* (γραμμές 36-

43). Ένα άλλο ενδεχόμενο που εξετάζεται, είναι αν το σύνολο των κοινών μεθόδων, *intersection*, των επιλεχθέντων διεπαφών να είναι ίδιο με το σύνολο των μεθόδων μιας εκ των δύο διεπαφών. Εάν αυτό ισχύει, για παράδειγμα για τη διεπαφή, $min\_I_1$, δεν δημιουργείται νέα διεπαφή, αλλά η διεπαφή $min\_I_2$ γίνεται παιδί της $min\_I_1$ και διαγράφεται από τη λίστα *L_Front* (γραμμές 44-47). Αντίστοιχα ισχύουν και για τη διεπαφή $min\_I_2$ (γραμμές 48-51). Σε περίπτωση που δεν ισχύει κάποιο από τα παραπάνω ενδεχόμενα, δημιουργείται μια νέα διεπαφή (γραμμή 53). Οι επιλεχθείσες διεπαφές, $min\_I_1$, $min\_I_2$, γίνονται παιδιά της νέας διεπαφής και προστίθενται σε αυτήν, οι κοινές μέθοδοί τους (γραμμές 54-58). Επιπλέον, η νέα διεπαφή προστίθεται στη λίστα μετώπου και στη λίστα *L_All_I* (γραμμή 58). Επιπρόσθετα, οι διεπαφές, $min\_I_1$, $min\_I_2$, εξάγονται από τη λίστα μετώπου (γραμμές 59-60), καθώς, δεν θα πρέπει πλέον να λαμβάνονται υπόψη στον υπολογισμό των αποστάσεων μεταξύ διεπαφών. Η όλη διαδικασία τερματίζει όταν το μέγεθος της λίστας μετώπου γίνει 1 (γραμμή 12), αφού δεν έχει νόημα ο υπολογισμός απόστασης μεταξύ ενός πλήθους διεπαφών, μικρότερο του 2. Τελικά, επιστρέφεται η λίστα, *L_All_I*, που περιέχει όλες τις διεπαφές που έχουν δημιουργηθεί κατά την πρώτη και δεύτερη φάση της RBA.

**Input :** *MPU : Multimap*

**Output :** *L_All_I : List < Interface >*

1. *L_Init : List < Interface >*
2. *L_Front : List < Interface >*

*// Phaze 1: Create interfaces for the users of the class with the methods they use*

3. **for all** *User in MPU* **do**
4. *i = Create_Interface()*
5. **for all** *Method_Inv in User* **do**
6. *i.add(Method_Inv)*
7. **end for**
8. *L_Init.add(i)*
9. **end for**

*// Phaze 2: Extract superclasses from interfaces with common methods*

10. *L_Front.addAll(L_Init)*
11. *L_All_I.addAll(L_Front)*
12. **while** $(|L\_Front| > 1)$ **do**
13. $fd = I$
14. $NCommonMethods = 0$

. *//Compute distances between interfaces*

15. **for all** $I_1$ *in L_Front* **do**
16. **for all** $I_2$ *in L_Front* **do**
17. $d = Jaccard(I_1, I_2)$
18. $intersection = |I_1.ListOfMethod\_Decl \cap I_2.ListOfMethod\_Decl|$

*// Find interfaces with minimum distance and max common methods*

19. **if** $(d == fd)$ **then**
20. **if** $(intersection > NCommonMethods)$ **then**
21. $fd = d$
22. $min\_I_1 = I_1$
23. $min\_I_2 = I_2$
24. $NCommonMethods = intersection$
25. **end if**
26. **if** $(d < fd)$ **then**
27. $fd = d$
28. $min\_I_1 = I_1$
29. $min\_I_2 = I_2$
30. $NCommonMethods = intersection$
31. **end if**
32. **end for**
33. **end for**

```
34.     if (fd < 1) then
35.         intersection = |min_I1.ListOfMethod_Decl ∩ min_I1.ListOfMethod_Decl|
            // If both selected interfaces have the same set of methods delete the one with less children
36.         if (intersection == (|min_I1.ListOfMethod_Decl| && |min_I1.ListOfMethod_Decl|) then
37.             if( |min_I1.ListOfChildren| > |min_I2.ListOfChildren| ) then
38.                 L_All_I.remove(min_I2)
39.                 L_FVront.remove(min_I2)
40.             else
41.                 L_All_I.remove(min_I1)
42.                 L_FVront.remove(min_I1)
43.             end if
44.         else if (intersection == ( |min_I1.ListOfMethod_Decl| ) then
45.             min_I1.ListOfChildren.add(min_I2)
46.             min_I2.parent = min_I1
47.             L_FVront.remove(min_I2)
48.         else if (intersection == (|min_I2.ListOfMethod_Decl|) then
49.             min_I2.ListOfChildren.add(min_I1)
50.             min_I1.Parent = min_I2
51.             L_FVront.remove(min_I1)
52.         else // merge interfaces with minimum distance and update L_FVront, L_All
53.             m_I = Create_Interface()
54.             m_I.child1 = min_I1
55.             m_I.child2 = min_I2
56.             min_I1.parent = m_I
57.             min_I2.parent = m_I
58.             m_I.addAll(intersection)
59.             L_FVront.add(m_I)
60.             L_All_I.add(m_I)
61.             L_FVront.remove(min_I1, min_I2)
62.         end if
63.     else
64.         break // // stop clustering if fd == 1
65. end while
66. return L_All_I
```

Αλγόριθμος 3. Ο αλγόριθμος της RBA.

Σύμφωνα με τα παραπάνω, η πρώτη φάση της RBA θα έχει ως αποτέλεσμα η αρχική σχεδίαση του Σχήματος 1.1 να μετατραπεί όπως φαίνεται στο Σχήμα 1.2. Ακόμη, εφαρμόζοντας τη δεύτερη φάση της RBA, παρατηρούμε ότι η σχεδίαση του Σχήματος 1.2, έχει ως αποτέλεσμα τη σχεδίαση του Σχήματος 1.3.

# ΚΕΦΑΛΑΙΟ 4. ΕΡΓΑΛΕΙΟ ΑΥΤΟΜΑΤΗΣ ΑΝΑΚΑΤΑΣΚΕΥΗΣ ΠΡΟΓΡΑΜΜΑΤΙΣΤΙΚΩΝ ΔΙΕΠΑΦΩΝ

---

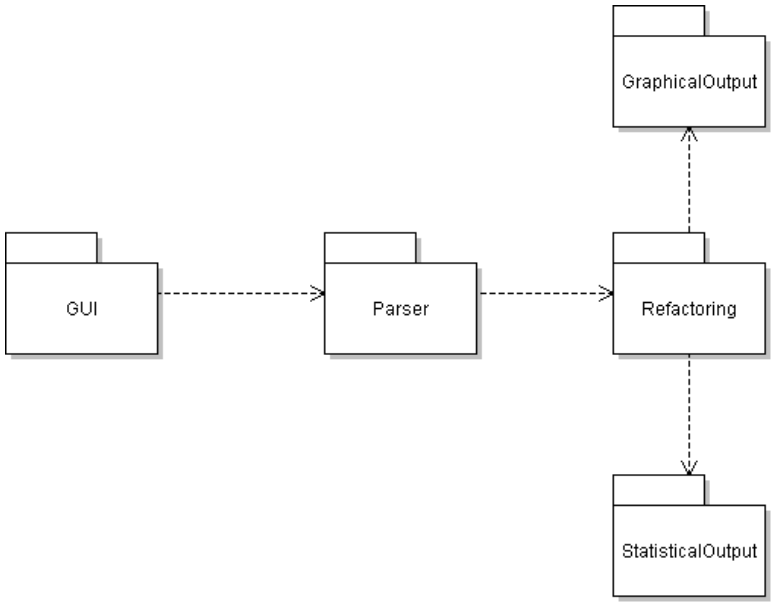
4.1 Αρχιτεκτονική

4.2 Εγκατάσταση του εργαλείου

---

## 4.1 Αρχιτεκτονική

Η αρχιτεκτονική του εργαλείου αυτόματης ανακατασκευής προγραμματιστικών διεπαφών που υλοποιήθηκε φαίνεται στο Σχήμα 4.1.




Σχήμα 4.1 Αρχιτεκτονική του συστήματος αυτόματης ανακατασκευής προγραμματιστικών διεπαφών.

Κάθε υποσύστημα αναλύεται στις επόμενες υποενότητες.

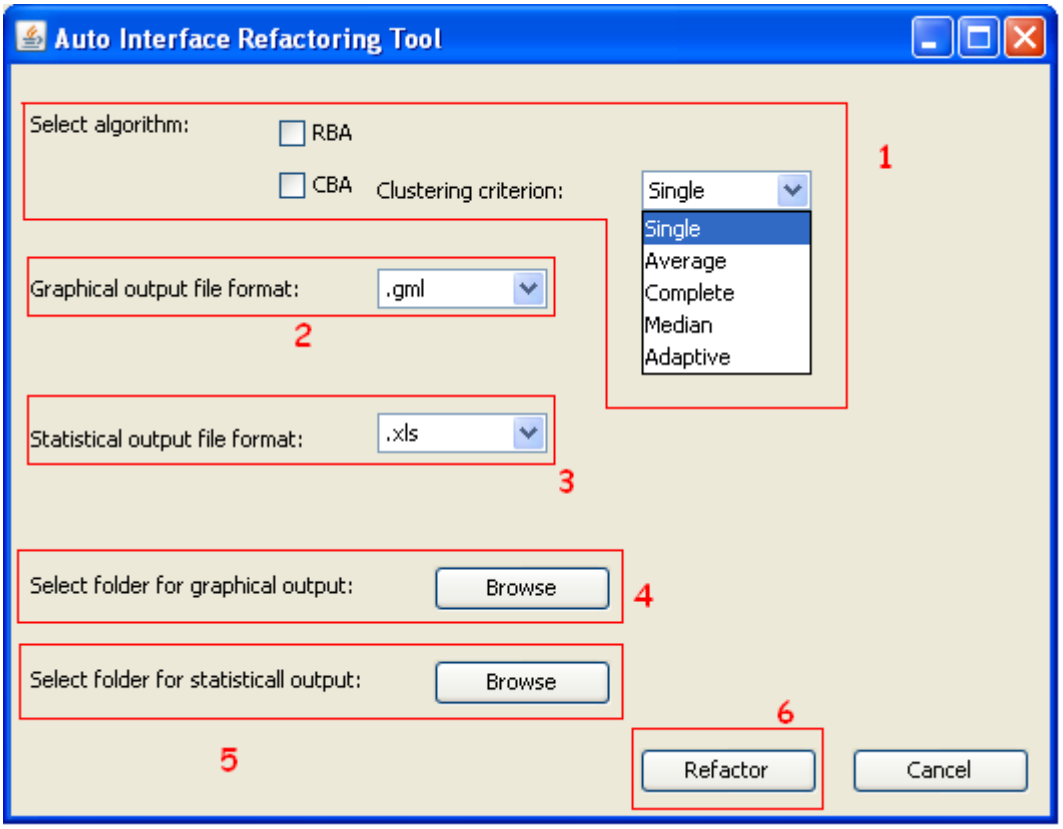
*4.1.1 Υποσύστημα GUI*

Το υποσύστημα GUI (Graphical User Interface) προσφέρει στον χρήστη του συστήματος μια γραφική διεπαφή, στην οποία έχει τη δυνατότητα να ρυθμίσει διάφορες επιλογές. Αρχικά, ο χρήστης μπορεί να επιλέξει τον αλγόριθμό της προτίμησής του. Μια από τις δυνατότητες είναι η επιλογή της ομαδοποίησης που θα χρησιμοποιηθεί για την CBA. Όπως έχουμε αναφέρει στο Κεφάλαιο 3, οι επιλογές είναι πέντε (5):

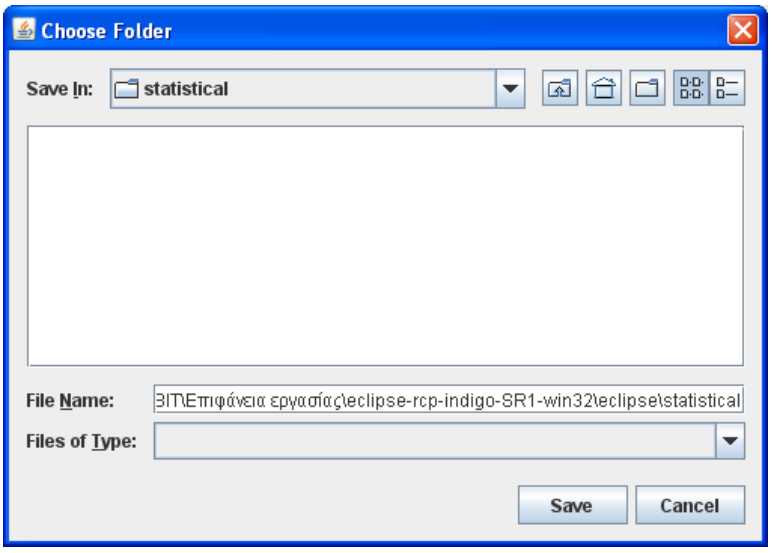
- Το κριτήριο του κοντινότερου γείτονα (Single).
- Το κριτήριο της μέσης τιμής των αποστάσεων των γειτόνων (Average).
- Το κριτήριο του μακρινότερου γείτονα (Complete).
- Το κριτήριο του της διάμεσης τιμής των αποστάσεων των γειτόνων (Median).
- Το κριτήριο της προσαρμοστικής απόστασης (Adaptive).

Οι επιλογές αυτές φαίνονται στο τμήμα 1 του Σχήματος 4.2.

Μια άλλη δυνατότητα που προσφέρει η γραφική διεπαφή του εργαλείου, είναι η αποθήκευση των αρχείων γραφικής απεικόνισης (τμήμα 4 Σχήματος 4.2). Πατώντας το κουμπί «Browse», ανοίγει ένα παράθυρο διαλόγου όπου μπορεί να εισέλθει στο φάκελο που επιθυμεί να αποθηκευτούν και στη συνέχεια να επιλέξει «Save». Το Σχήμα 4.3 παρουσιάζει ένα παράδειγμα αποθήκευσης των αρχείων στο φάκελο «graphical». Αντίστοιχα ισχύουν και για την αποθήκευση των αρχείων στατιστικών δεδομένων (τμήμα 5 του Σχήματος 4.2).




Σχήμα 4.2 Η γραφική διεπαφή του εργαλείου.




Σχήμα 4.3 Παράδειγμα επιλογής φακέλου για την αποθήκευση των αρχείων.

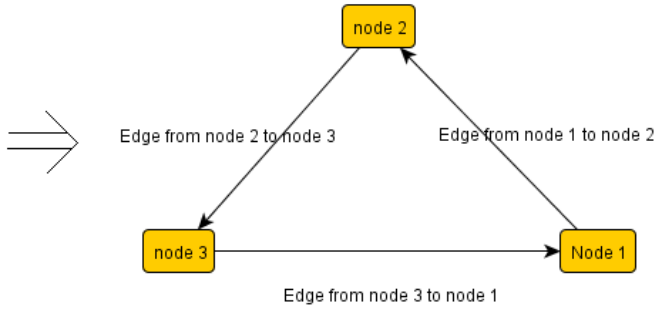
*4.1.2 Υποσύστημα Refactoring*

Η λειτουργία του υποσυστήματος Refactoring περιλαμβάνει την εφαρμογή των διαδικασιών της CBA και RBA, που έχουν περιγραφεί αναλυτικά στο Κεφάλαιο 3, στο υπό ανακατασκευή πρόγραμμα. Ο χρήστης, επιλέγοντας το κουμπί «Refactor», εκκινεί η διαδικασία επεξεργασίας του προγράμματος που δέχεται ως είσοδο το εργαλείο και η ροή ελέγχου περνά στα υπόλοιπα συστατικά της αρχιτεκτονικής του (τμήμα 6 του Σχήματος 4.2).

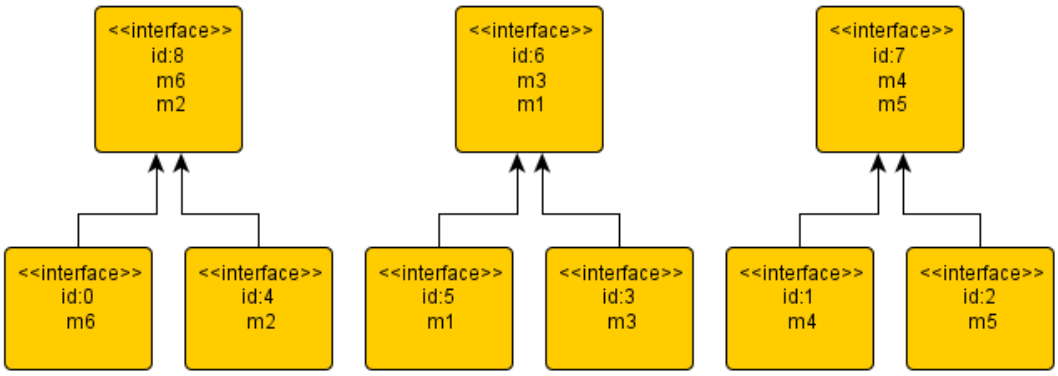
*4.1.3 Υποσύστημα GraphicalOutput*

Υπεύθυνο για την παραγωγή αρχείων γραφικής απεικόνισης των αποτελεσμάτων της ανακατασκευής, είναι το υποσύστημα GraphicalOutput. Το εργαλείο παράγει ως έξοδο για κάθε κλάση του προγράμματος που δέχεται ως είσοδο, ένα αρχείο γραφικής απεικόνισης των διεπαφών που δημιουργούνται κατά τη διαδικασία τόσο της CBA, όσο και της RBA. Ο χρήστης μπορεί να επιλέξει τον τύπο για τα αρχεία εξόδου γραφικής απεικόνισης του εργαλείου (τμήμα 2 του Σχήματος 4.2). Προς το παρόν υποστηρίζεται μόνο ο τύπος αρχείων .gml, ωστόσο για η συγκεκριμένη λειτουργία έχει υλοποιηθεί με το Factory Method Pattern που επιτρέπει την εύκολη επέκταση της λειτουργίας, για υποστήριξη επιπλέον τύπων αρχείων. Τα αρχεία ακολουθούν τη δομή της Graph Modeling Language και έχουν κατάληξη .gml. Το περιεχόμενο ενός αρχείου .gml και η αντίστοιχη οπτικοποίησή του φαίνονται στο Σχήμα 4.4. Η οπτικοποίηση των αρχείων γραφικής απεικόνισης για την κλάση Provider του παραδείγματος της Ενότητας 1.2, που δημιουργούνται από τη διαδικασία της CBA με το κριτήριο του μακρινότερου γείτονα και της RBA, φαίνονται στο Σχήμα 4.5 και στο Σχήμα 4.6, αντίστοιχα. Και στις δύο περιπτώσεις ανατίθεται ένας ξεχωριστός αριθμός (id) ώστε να είναι διακριτή η κάθε διεπαφή. Για την περίπτωση της RBA παρατηρούμε ότι στις διεπαφές οι οποίες αντιστοιχούν στα σύνολα των μεθόδων που χρησιμοποιούν οι κλάσεις-χρήστες της κλάσης Provider, σημειώνεται η πλήρης διαδρομή της κάθε κλάσης-χρήστη. Ένα εργαλείο με το οποίο είναι δυνατή η οπτικοποίηση των αρχείων .gml είναι το yEd Graph Editor, το οποίο χρησιμοποιήθηκε στα πλαίσια της διατριβής.

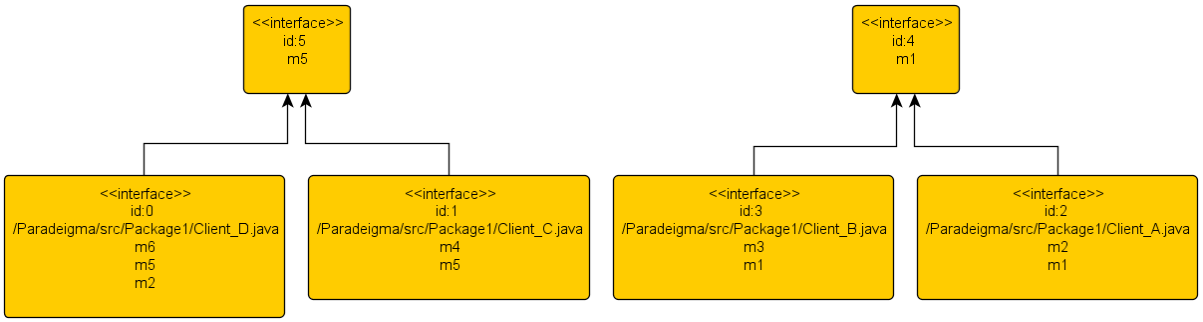
```
graph [

        node [
                id 1
                label "Node 1"
                        ]
        node [
                id 2
                label "node 2"
                        ]
        node [
                id 3
                label "node 3"
                        ]
        edge [
                source 1
                target 2
                label "Edge from node 1 to node 2"
        ]
        edge [
                source 2
                target 3
                label "Edge from node 2 to node 3"
        ]
        edge [
                source 3
                target 1
                label "Edge from node 3 to node 1"
        ]
]
```




Σχήμα 4.4 Παράδειγμα απλού αρχείου .gml και η αντίστοιχη οπτικοποίηση..




Σχήμα 4.5 Η οπτικοποίηση του αρχείου εξόδου της CBA με το κριτήριο του μακρινότερου γείτονα που αφορά την κλάση Provider του Σχήματος 1.1.




Σχήμα 4.6 Η οπτικοποίηση του αρχείου εξόδου της RBA που αφορά την κλάση Provider του Σχήματος 1.1.

### *4.1.4 Υποσύστημα StatisticalOutput*

Υπεύθυνο για την παραγωγή αρχείων στατιστικών αποτελεσμάτων της ανακατασκευής, είναι το υποσύστημα StatisticalOutput. Το εργαλείο παράγει ως έξοδο, για το πρόγραμμα που δέχεται ως είσοδο, ένα αρχείο στατιστικών μετρήσεων με κατάληξη .xls, ανά πακέτο, για τις διεπαφές που δημιουργούνται κατά τη διαδικασία τόσο της CBA, όσο και της RBA. Ο χρήστης μπορεί επίσης, να επιλέξει τον τύπο για τα αρχεία εξόδου στατιστικών μετρήσεων του εργαλείου (τμήμα 3 του Σχήματος 4.3). Προς το παρόν υποστηρίζεται μόνο ο τύπος αρχείων .xls, ωστόσο για η συγκεκριμένη λειτουργία έχει υλοποιηθεί με το Factory Method Pattern που επιτρέπει την εύκολη επέκταση της λειτουργίας, για υποστήριξη επιπλέον τύπων αρχείων. Το περιεχόμενο του αρχείου στατιστικών δεδομένων για το παράδειγμα της Ενότητας 1.2, που δημιουργείται από τη διαδικασία της CBA με το κριτήριο του μακρινότερου γείτονα και της RBA, φαίνεται στο Σχήμα 4.7. Ένα εργαλείο με το οποίο είναι δυνατή η εμφάνιση του περιεχομένου των αρχείων .xls είναι το Microsoft Excel, το οποίο χρησιμοποιήθηκε στα πλαίσια της διατριβής.

Paradeigma Arie Complete and Our.xls

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | Average ACD(min) | Average ACD(max) | Average CBO(min) | Average CBO(max) | Total Interfaces | Average Interfaces | Max Tree Height | Average Tree Height |
| 2 | | | | | | | | | |
| 3 | Package1 | 0,5 | 0,625 | 1,5 | 1,5 | 9 | 1,8 | 1 | 0,2 |
| 4 | | | | | | | | | |
| 5 | | | | | | | | | |
| 6 | | | | | | | | | |

-Paradeigma Arie Complete | -Paradeigma Our

Paradeigma Arie Complete and Our.xls

| | A | B | C | D | E | F | G | H | I |
|---|---|---|---|---|---|---|---|---|---|
| 1 | | Average ACD(min) | Average ACD(max) | Average CBO(min) | Average CBO(max) | Total Interfaces | Average Interfaces | Max Tree Height | Average Tree Height |
| 2 | | | | | | | | | |
| 3 | Package1 | 0,625 | 0,625 | 1 | 1 | 6 | 1,2 | 1 | 0,2 |
| 4 | | | | | | | | | |
| 5 | | | | | | | | | |
| 6 | | | | | | | | | |

-Paradeigma Arie Complete | -Paradeigma Our

Σχήμα 4.7 Επάνω: τα στατιστικά δεδομένα που παράγει η CBA με το κριτήριο του μακρινότερου γείτονα για το παράδειγμα της ενότητας 1.2. Κάτω: τα αντίστοιχα στατιστικά δεδομένα της RBA.

### *4.1.5 Υποσύστημα Parser*

Το υποσύστημα Parser είναι υπεύθυνο για τη συντακτική ανάλυση των κλάσεων του προγράμματος που δέχεται ως είσοδο το εργαλείο αυτόματης ανακατασκευής προγραμματιστικών διεπαφών. Το υποσύστημα Parser κάνει χρήση του ASTParser του Eclipse. Ο ASTParser αναλύει τον πηγαίο κώδικα κάθε κλάσης σε μορφή Αφηρημένου Συντακτικού Δένδρου (Abstract Syntax Tree - AST). Το AST είναι μια δενδρική αναπαράσταση της αφηρημένης συντακτικής δομής του πηγαίου κώδικα γραμμένου σε μια γλώσσα προγραμματισμού. Κάθε κόμβος του δένδρου υποδηλώνει μια λεκτική μονάδα (token) του κώδικα. Το δένδρο είναι αφηρημένο με την έννοια ότι δεν αναπαριστά κάθε λεπτομέρεια που εμφανίζεται στον κώδικα. Για παράδειγμα, η αναπαράσταση της κλάσης Test του Σχήματος 4.8 σε αφηρημένο συντακτικό δένδρο φαίνεται στο Σχήμα 4.9.

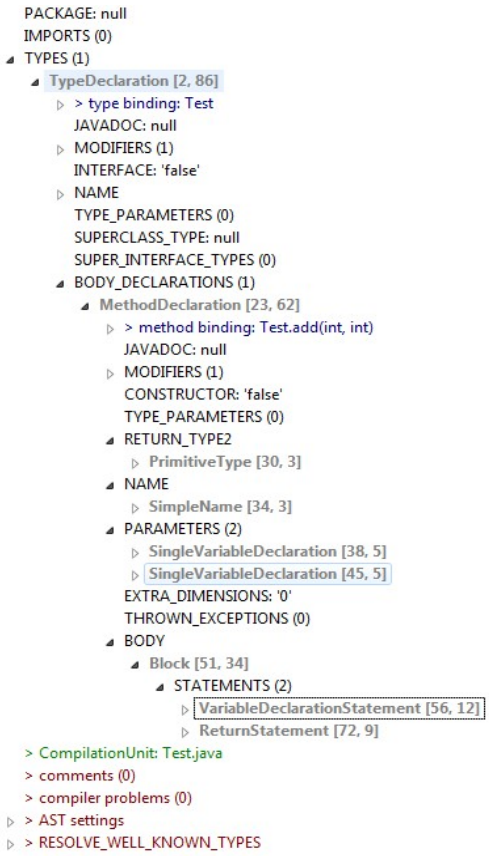
```
public class Test {
        public int add(int a, int b){
                int c = a+b;
                return c;
        }
}
```

Σχήμα 4.8 Πηγαίος κώδικας της κλάσης Test

Για να γίνει πιο εύκολα αντιληπτό στον χρήστη πώς αναπαριστά το Eclipse τον πηγαίο κώδικα ως αφηρημένο συντακτικό δένδρο, μπορεί να εγκαταστήσει το plug-in AST Viewer. Με το εν λόγω plug-in ο χρήστης μπορεί να επιλέξει ένα τμήμα κώδικα και να εμφανιστεί η αντίστοιχη απεικόνισή του στην ιεραρχία (Σχήμα 4.10). Χρησιμοποιώντας το Visitor Pattern, μπορούμε να προσπελάσουμε όλους τους κόμβους που εμφανίζονται στο Σχήμα 4.10 με γκρι γραμματοσειρά (MethodDeclaration, Modifier, SimpleName, SingleVariableDeclaration).

Το υποσύστημα Parser με τη χρήση του ASTParser αποθηκεύει, για το πρόγραμμα προς ανακατασκευή, σε μια λίστα (PackageInfoList) όλα τα πακέτα του προγράμματος, δηλαδή τους κόμβους IPackageFragment. Επιπλέον, σε κάθε πακέτο της λίστας PackageInfoList, προστίθενται σε μια λίστα (PackageCus) οι κλάσεις του, δηλαδή οι κόμβοι ICompilationUnit Επίσης, σε κάθε κλάση της λίστας PackageCus προστίθενται σε μια λίστα (MethodsDeclared) όλες οι μέθοδοι που δηλώνονται σε αυτήν, δηλαδή οι κόμβοι MethodDeclaration. Αντίστοιχα, σε κάθε κλάση της λίστας PackageCus προστίθενται σε μια λίστα (MethodsInvoked) όλες οι μέθοδοι που καλούνται σε αυτήν, δηλαδή οι κόμβοι MethodInvocation. Από την πληροφορία αυτή, εξάγεται η πληροφορία των δομών multimap UPM και MPU (Κεφάλαιο 3, Αλγόριθμος 1 και Αλγόριθμος 2) που αποτελούν είσοδο για τους αλγορίθμους, CBA και RBA αντίστοιχα, του υποσυστήματος Refactoring.


Σχήμα 4.9 Το αφηρημένο συντακτικό δένδρο για την κλάση Test.

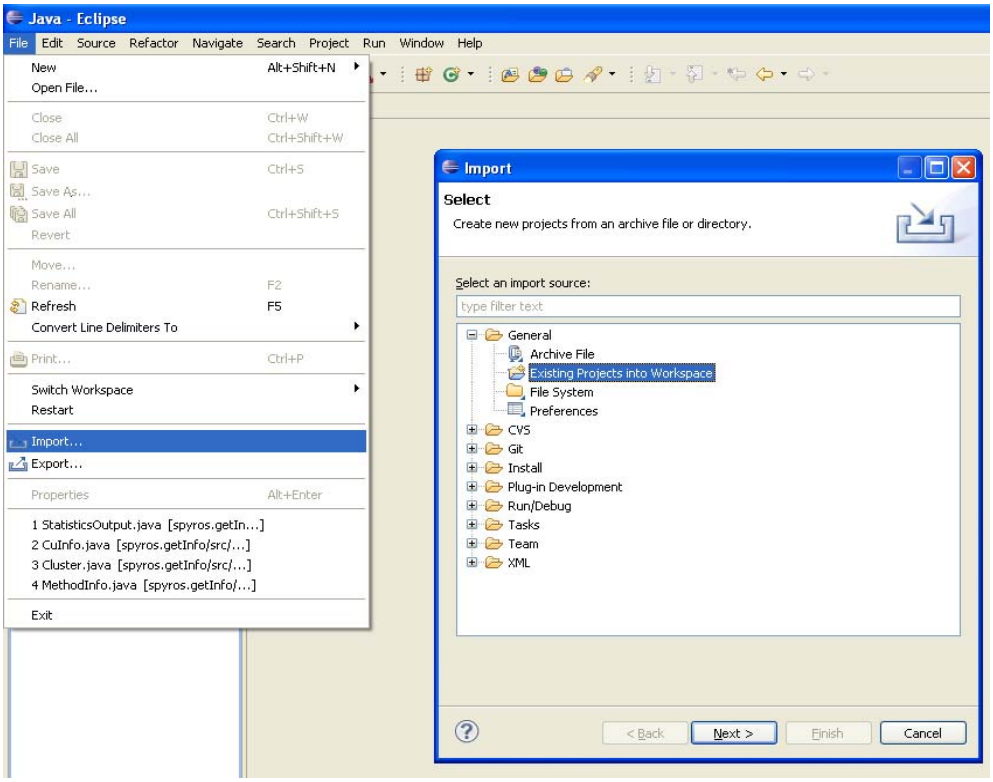



Σχήμα 4.10 Το plug in AST Viewer.

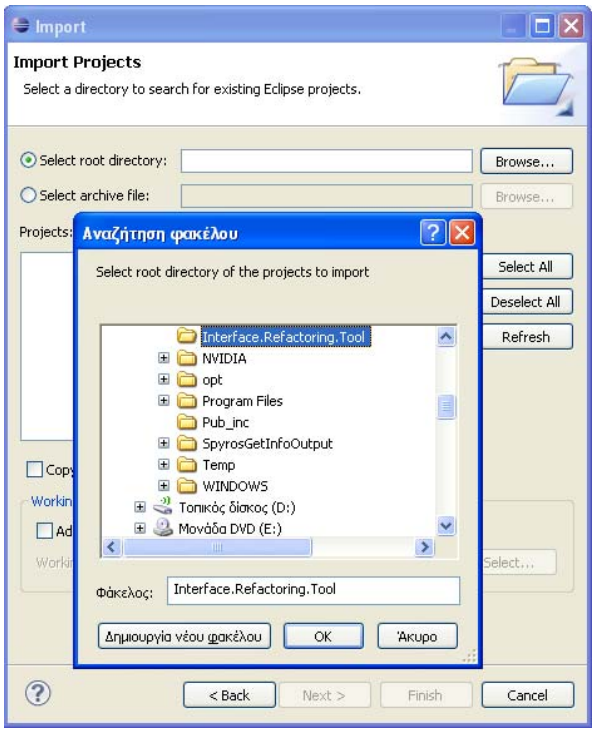
### 4.2 Εγκατάσταση του εργαλείου

Το εργαλείο αυτόματης ανακατασκευής προγραμματιστικών διεπαφών έχει υλοποιηθεί ως πρόσθετη λειτουργία (plug-in) για το ολοκληρωμένο περιβάλλον ανάπτυξης (IDE) Eclipse. Η ανάπτυξη του εργαλείου υλοποιήθηκε στην έκδοση 3.7 του Eclipse RCP (Eclipse Rich Client Platform Indigo). Για την εγκατάσταση και εκτέλεση του προγράμματος που υλοποιεί το εργαλείο αυτόματης ανακατασκευής προγραμματιστικών διεπαφών, ο χρήστης ακολουθεί την εξής διαδικασία:

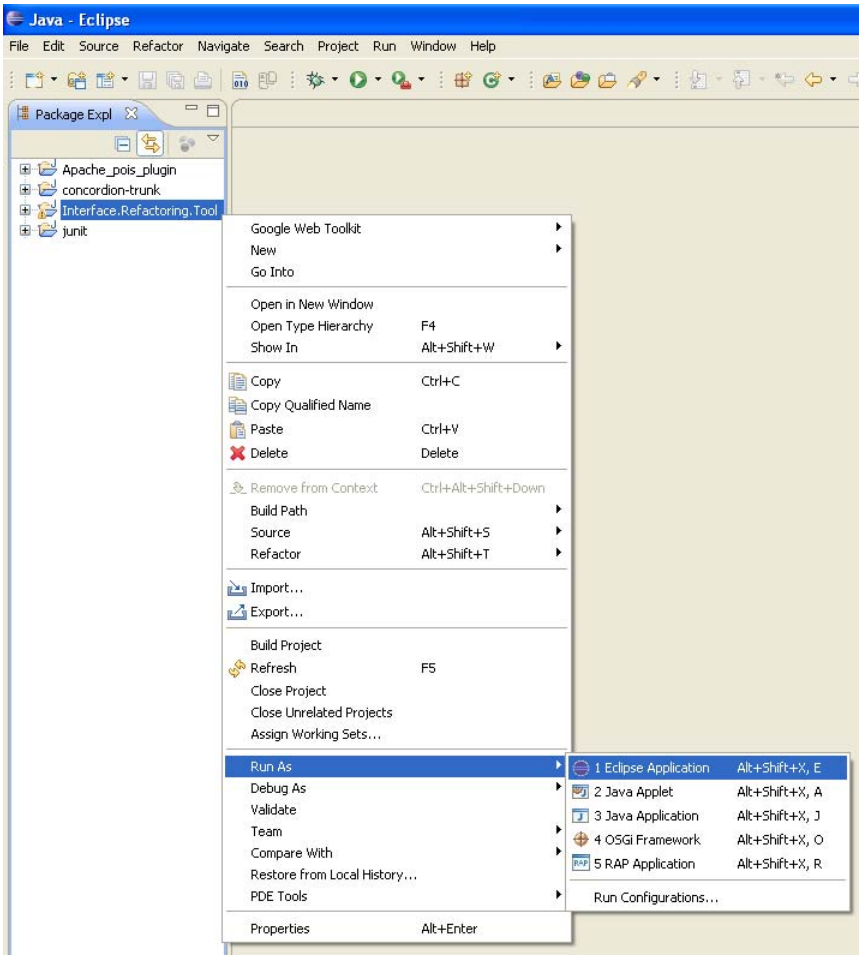
1 Επιλογή File→Import→Existing Projects into Workspace→Next (Σχήμα 4.11).
2 Αναζήτηση και επιλογή του φακέλου «Interface.Refactoring.Tool»→επιλογή «OK» (Σχήμα 4.12).
3 Δεξί κλικ στο πρόγραμμα→επιλογή «Run As»→Eclipse Application (Σχήμα 4.13).
4 Εισαγωγή, με ανάλογο τρόπο του Βήματος 1, του επιθυμητού προς ανακατασκευή προγράμματος στη νέα πλατφόρμα Eclipse που ανοίγει και επιλογή από το μενού Interface Refactoring Tool→CBA/RBA (Σχήμα 4.14). Η επιλογή αυτή μας οδηγεί στην γραφική διεπαφή του εργαλείου (Σχήμα 4.2).



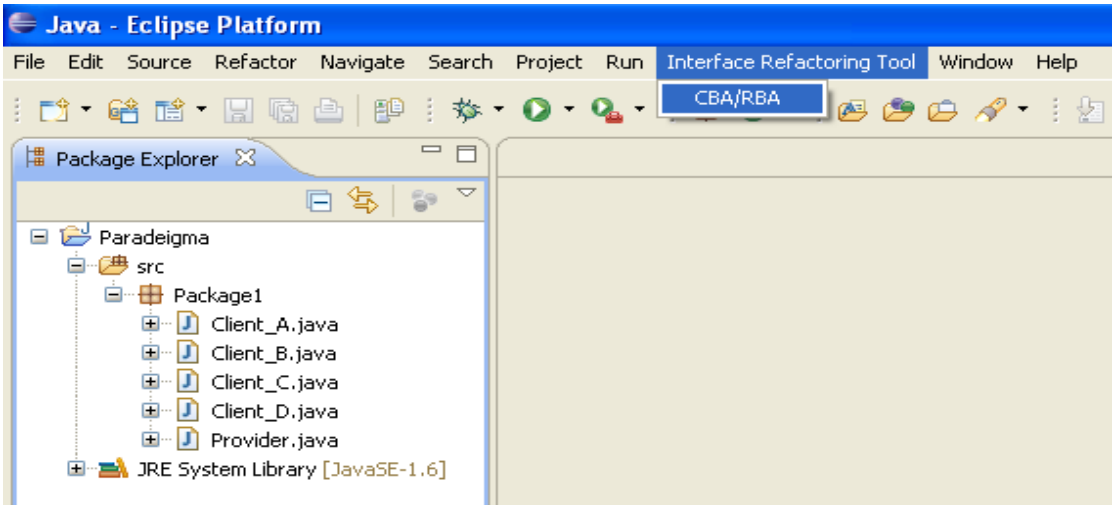

Σχήμα 4.11 Βήμα 1 της διαδικασίας εγκατάστασης του εργαλείου.




Σχήμα 4.12 Βήμα 2 της εγκατάστασης του εργαλείου.




Σχήμα 4.13 Βήμα 3 της εγκατάστασης του εργαλείου.




Σχήμα 4.14 Εκτέλεση του εργαλείου με είσοδο το παράδειγμα της ενότητας 1.2.

# ΚΕΦΑΛΑΙΟ 5. ΠΕΙΡΑΜΑΤΑ

5.1 Μεθοδολογία
5.2 Αποτελέσματα και συμπεράσματα

## 5.1 Μεθοδολογία

Για να εκτιμήσουμε την απόδοση των αλγορίθμων της CBA και της RBA, τις εφαρμόσαμε σε δύο μελέτες περίπτωσης (case studies). Για το σκοπό αυτό επιλέχθηκαν δύο περιβάλλοντα (frameworks), ανοιχτού λογισμικού, που χρησιμοποιούνται στην ανάπτυξη λογισμικού με τακτικούς ελέγχους (test-driven development). Τα περιβάλλοντα αυτά είναι το JUnit και το Concordion. Το JUnit αποτελείται από 58 πακέτα που περιέχουν συνολικά 320 κλάσεις. Το Concordion αποτελείται από 48 πακέτα που περιέχουν συνολικά 231 κλάσεις. Οι τοποθεσίες από τις οποίες μπορούν να ανακτηθούν είναι (HTTP://JUNIT.ORG/) και (HTTP://WWW.CONCORDION.ORG/), αντίστοιχα.

Τα κριτήρια σύγκρισης που χρησιμοποιήσαμε έχουν ως στόχο να υπολογιστεί:

- Η σύζευξη μεταξύ των κλάσεων. Το κριτήριο αυτό, ποσοτικοποιείται μέσω της μετρικής CBO, που ορίστηκε στο Κεφάλαιο 3. Για τη μετρική CBO υπάρχουν δύο παραλλαγές, μία για κάθε τρόπο επιλογής διεπαφών που χρειάζεται η κάθε κλάση-χρήστη μιας κλάσης. Η πρώτη παραλλαγή αντιπροσωπεύει την ελάχιστη σύζευξη ($CBO_{(min)}$) και αντιστοιχεί στην επιλογή του ελάχιστου αριθμού διεπαφών που περιέχουν τις μεθόδους που χρησιμοποιεί η κάθε κλάση-χρήστης. Η δεύτερη παραλλαγή αντιπροσωπεύει τη μέγιστη σύζευξη ($CBO_{(max)}$) και αντιστοιχεί στην επιλογή του απαιτούμενου πλήθους διεπαφών μιας κλάσης, έτσι ώστε το σύνολό τους, να περιέχει ακριβώς όσες μεθόδους χρησιμοποιεί η κάθε κλάση-χρήστης.

- Το ποσοστό μείωσης των ανατιθέμενων μεθόδων για κάθε κλάση. Το κριτήριο αυτό, ποσοτικοποιείται μέσω της μετρικής ACD που, επίσης, ορίστηκε στο Κεφάλαιο 3. Για τη μετρική ACD υπάρχουν, επίσης, δύο παραλλαγές, μία για κάθε τρόπο επιλογής διεπαφών που χρειάζεται η κάθε κλάση-χρήστη μιας κλάσης. Η πρώτη παραλλαγή αντιπροσωπεύει το ποσοστό της ελάχιστης μείωση των ανατιθέμενων μεθόδων για κάθε κλάση ($ACD_{(min)}$) και αντιστοιχεί στην επιλογή του ελάχιστου αριθμού διεπαφών που περιέχουν τις μεθόδους που χρησιμοποιεί η κάθε κλάση-χρήστης. Η δεύτερη παραλλαγή αντιπροσωπεύει το ποσοστό της μέγιστης μείωσης των ανατιθέμενων μεθόδων για κάθε κλάση ($ACD_{(max)}$) και αντιστοιχεί στην επιλογή του απαιτούμενου πλήθους διεπαφών μιας κλάσης, έτσι ώστε το σύνολό τους, να περιέχει ακριβώς όσες μεθόδους χρησιμοποιεί η κάθε κλάση-χρήστης. Οι δύο τρόποι επιλογής διεπαφών που χρειάζεται η κάθε κλάση-χρήστη μιας κλάσης, καθώς και ο συμβιβασμός (trade-off) μεταξύ σύζευξης και ποσοστού μείωσης των περιττών ανατιθέμενων μεθόδων, για κάθε κλάση-χρήστη, έχουν αναλυθεί στο Κεφάλαιο 3.
- Η πολυπλοκότητα των ιεραρχιών από διεπαφές που δημιουργούνται. Το κριτήριο αυτό, ποσοτικοποιείται μέσω των μετρικών που αντιπροσωπεύουν τις συνολικές διεπαφές (Total Interfaces) και τη μέση τιμή των διεπαφών (Average Interfaces). Επίσης, ποσοτικοποιείται μέσω των μετρικών που αντιπροσωπεύουν τη μέση τιμή του μέγιστου ύψους των δένδρων (Max Tree Height) και της μέσης τιμής του ύψους των δένδρων (Average Tree Height).

Ιδανικά, θα θέλαμε το μέγιστο ποσοστό μείωσης των ανατιθέμενων μεθόδων, με την ελάχιστη σύζευξη μεταξύ των κλάσεων και την ελάχιστη πολυπλοκότητα των ιεραρχιών των διεπαφών που δημιουργούνται. Οι τιμές που παρουσιάζονται στα σχήματα της ενότητας 5.2, αποτελούν τη μέση τιμή ανά μελέτη περίπτωσης και ανά κριτήριο σύγκρισης. Ο συγκεκριμένος τρόπος παρουσίασης επιλέχθηκε για πρακτικούς και εποπτικούς λόγους, καθώς το μεγάλο πλήθος των κλάσεων του κάθε περιβάλλοντος δεν θα διευκόλυνε την εξαγωγή των συμπερασμάτων. Εξαίρεση,

αποτελεί το κριτήριο σύγκρισης των συνολικών διεπαφών (Total Interfaces) όπου παρουσιάζεται το συνολικό πλήθος διεπαφών που δημιουργείται, με κάθε αλγόριθμο, για κάθε μελέτη περίπτωσης. Τα αποτελέσματα παρουσιάζονται, συγκρίνοντας κάθε φορά, τις επιδόσεις των αλγορίθμων της CBA και της RBA, για κάθε ένα από τα έξι (6) κριτήρια σύγκρισης που αναφέρθηκαν παραπάνω και κάθε ένα από τα πέντε (5) κριτήρια απόστασης που έχουν παρουσιαστεί στο Κεφάλαιο 3:

- Το κριτήριο του κοντινότερου γείτονα (Single Linkage Criterion).
- Το κριτήριο του μακρινότερου γείτονα (Complete Linkage Criterion).
- Το κριτήριο του μέσου όρου των αποστάσεων των γειτόνων (Average Linkage Criterion).
- Το κριτήριο του διάμεσου γείτονα (Median Linkage Criterion).
- Το κριτήριο της προσαρμοστικής απόστασης (Adaptive Linkage Criterion).

Αναλυτικότερα αποτελέσματα για την κάθε μελέτη περίπτωσης, παρουσιάζονται στο Παράρτημα, όπου παρατίθενται μετρήσεις ανάλογες με αυτές της υποενότητας 5.2, αλλά για κάθε πακέτο του JUnit και του Concordion.

**5.2 Αποτελέσματα και συμπεράσματα**

Τα αποτελέσματα των πειραμάτων για το JUnit παρουσιάζονται από το Σχήμα 5.1, έως και το Σχήμα 5.6 για το Concordion από το Σχήμα 5.7, έως και το Σχήμα 5.12. Οι παρατηρήσεις μας είναι ανάλογες για τις δύο μελέτες περίπτωσης.

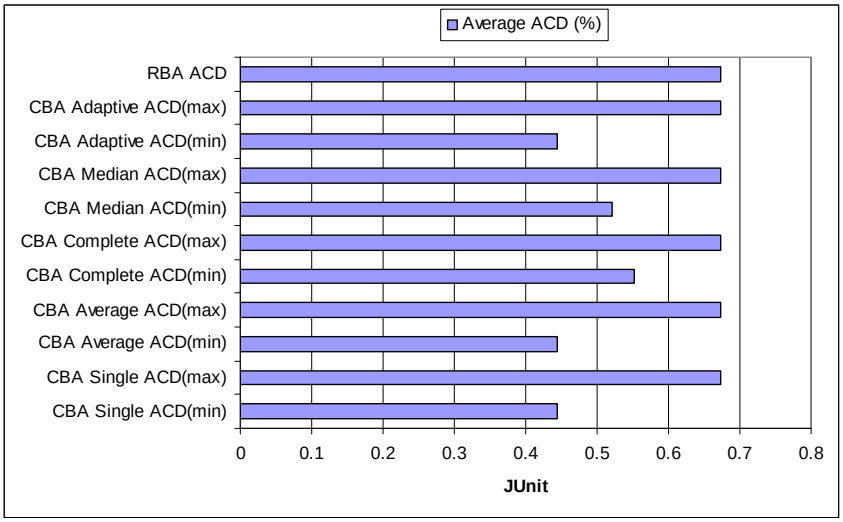
Τα συμπεράσματά μας συνοψίζονται ως εξής:

- Από τα Σχήματα 5.1 και 5.7, παρατηρούμε ότι, σε σχέση με τη μείωση ανατιθέμενων μεθόδων για κάθε κλάση, ο αλγόριθμος της RBA, μαζί με τον αλγόριθμο της CBA όταν επιλέγεται οποιοδήποτε από τα κριτήρια απόστασης με τη μέγιστη μείωση ανατιθέμενων μεθόδων για κάθε κλάση ($ACD_{(max)}$), επιτυγχάνουν την καλύτερη επίδοση. Ακολουθεί ο αλγόριθμος

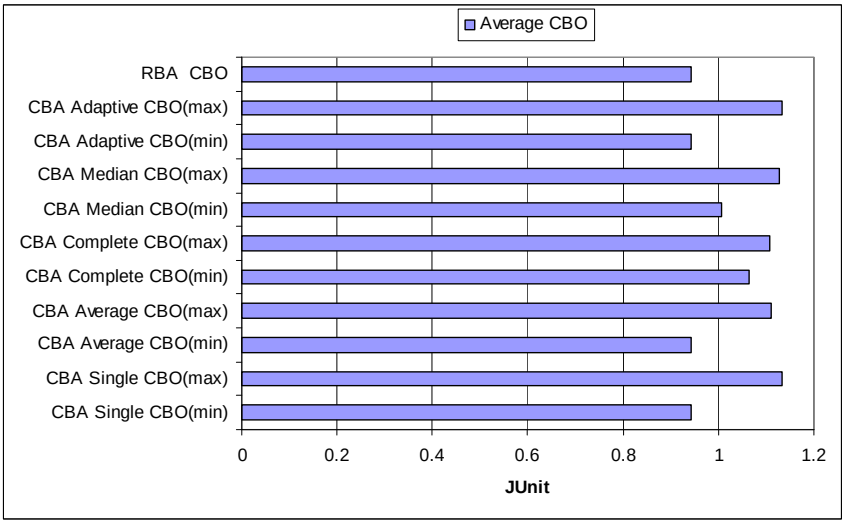
της CBA, όταν επιλέγεται το κριτήριο απόστασης του μακρινότερου γείτονα, με την ελάχιστη μείωση ανατιθέμενων μεθόδων για κάθε κλάση (CBA Complete $ACD_{(min)}$) και εν συνεχεία οι υπόλοιπες περιπτώσεις.

- Από τα Σχήματα 5.2 και 5.8, παρατηρούμε ότι, σε σχέση με τη σύζευξη, ο αλγόριθμος της RBA, μαζί με τον αλγόριθμο της CBA όταν επιλέγεται ως κριτήριο απόστασης ένα από: του μέσου όρου των αποστάσεων των γειτόνων, του κοντινότερου γείτονα ή της προσαρμοστικής απόστασης, με την ελάχιστη σύζευξη ($CBO_{(min)}$) επιτυγχάνουν την καλύτερη επίδοση. Ακολουθεί ο αλγόριθμος της CBA, όταν επιλέγεται το κριτήριο απόστασης του διάμεσου γείτονα, με την ελάχιστη σύζευξη (CBA Median $CBO_{(min)}$) και εν συνεχεία οι υπόλοιπες περιπτώσεις.
- Από το Σχήμα 5.3 έως και το Σχήμα 5.6 και από το Σχήμα 5.9 έως και το Σχήμα 5.12, παρατηρούμε ότι, σε σχέση με την πολυπλοκότητα των ιεραρχιών από διεπαφές που δημιουργούνται, ο αλγόριθμος της RBA, επιτυγχάνει την καλύτερη επίδοση σε όλες τις αντίστοιχες μετρικές: τις συνολικές διεπαφές (Total Interfaces), τη μέση τιμή των διεπαφών (Average Interfaces), τη μέση τιμή του μέγιστου ύψους των δένδρων (Max Tree Height) και της μέσης τιμής του ύψους των δένδρων (Average Tree Height). Ακολουθεί ο αλγόριθμος της CBA, όταν επιλέγεται το κριτήριο απόστασης του μακρινότερου γείτονα και εν συνεχεία οι υπόλοιπες περιπτώσεις.

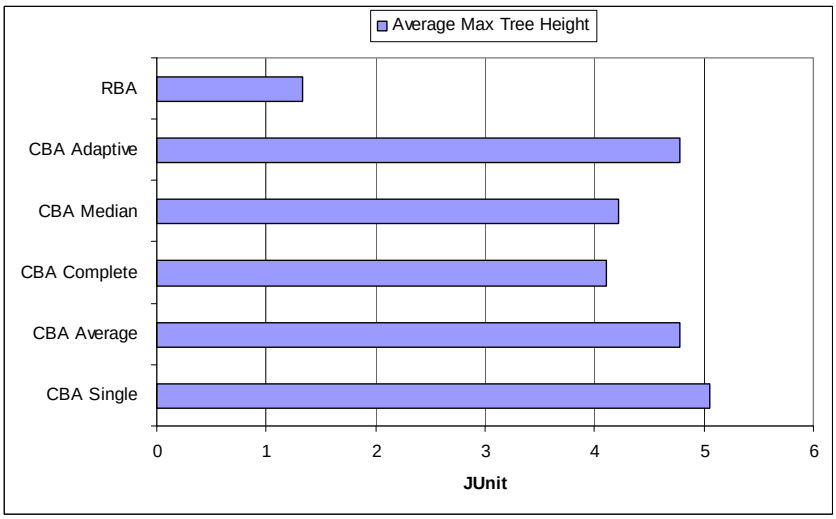
Συγκεντρωτικά, θα μπορούσαμε να αποφανθούμε ότι ο αλγόριθμος της RBA επιτυγχάνει τον βέλτιστο συνδυασμό σε σχέση με το ποσοστό μείωσης των ανατιθέμενων μεθόδων για κάθε κλάση, τη σύζευξη μεταξύ των κλάσεων και την πολυπλοκότητα των ιεραρχιών από διεπαφές που δημιουργούνται.



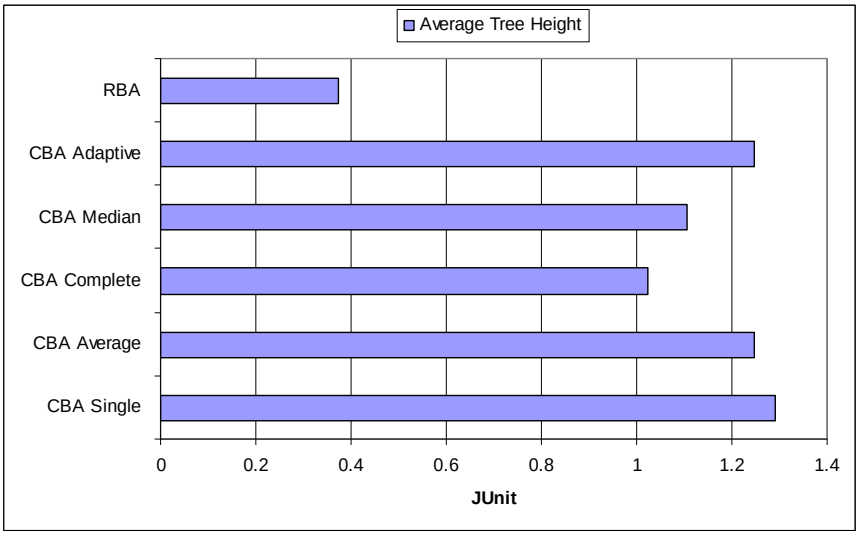

Σχήμα 5.1 Μέση τιμή ACD για το JUnit.



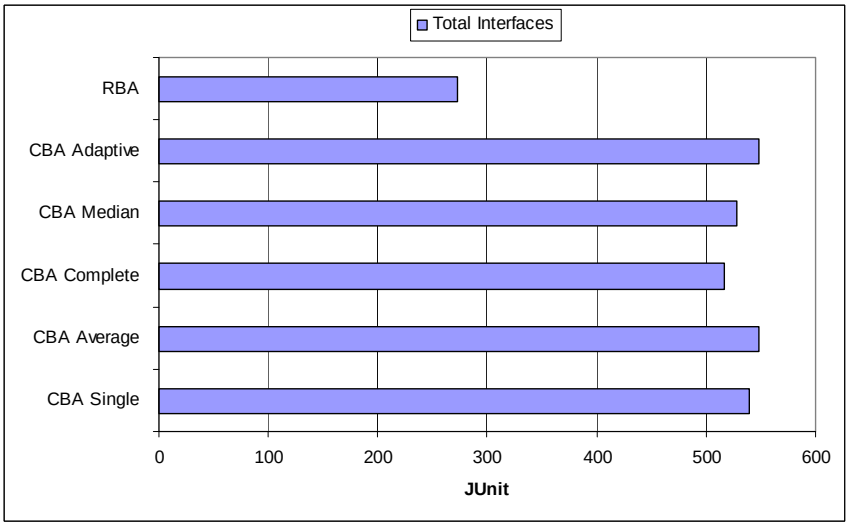

Σχήμα 5.2 Μέση τιμή CBO για το JUnit.



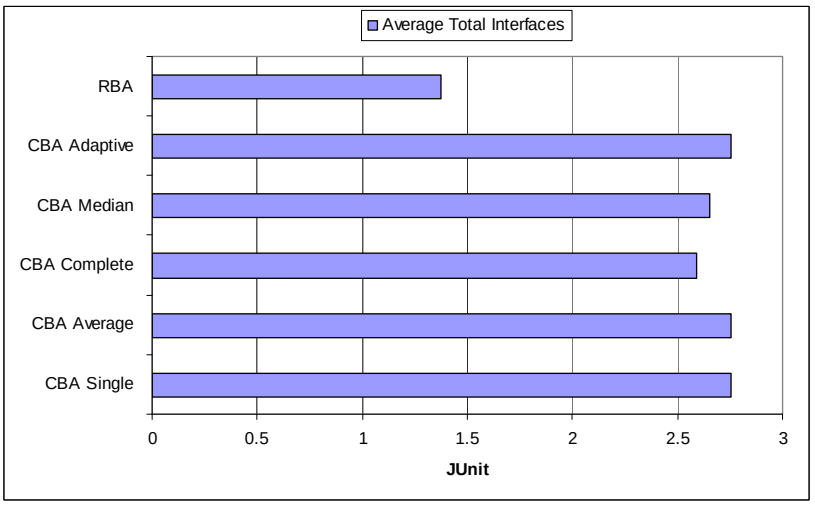

Σχήμα 5.3 Μέση τιμή Max Tree Height για το JUnit.



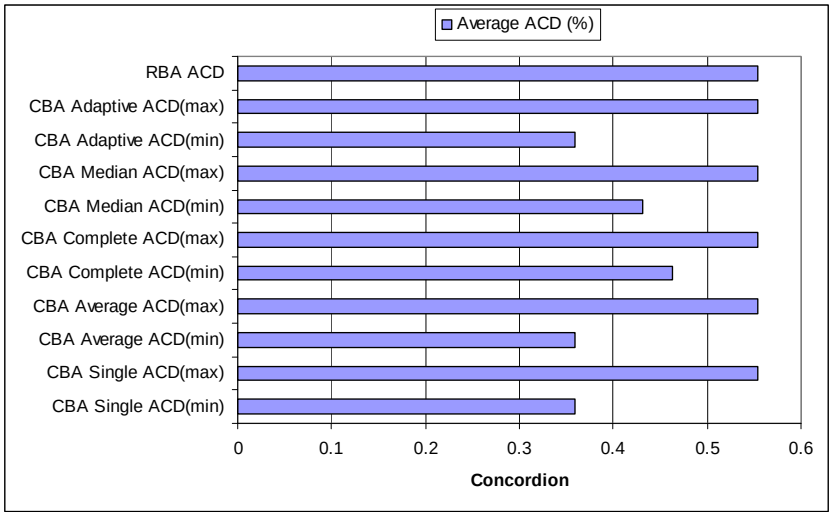

Σχήμα 5.4 Μέση τιμή Tree Height για το JUnit.



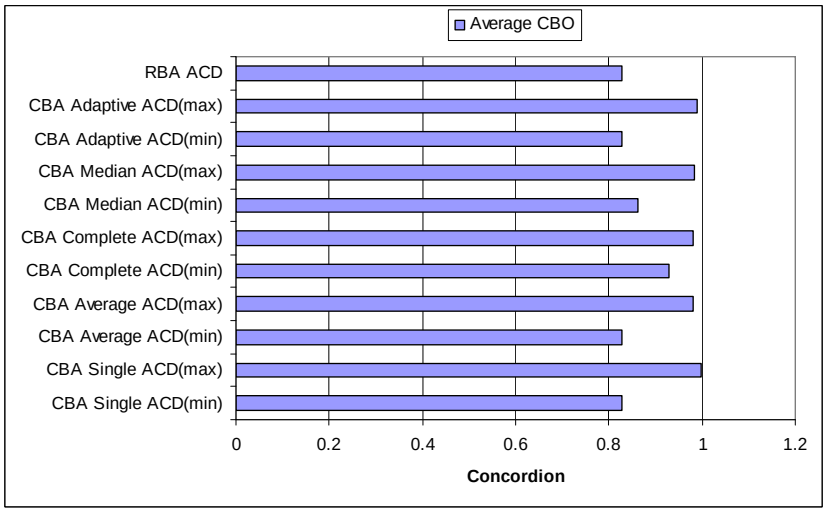

Σχήμα 5.5 Total Interfaces για το JUnit.



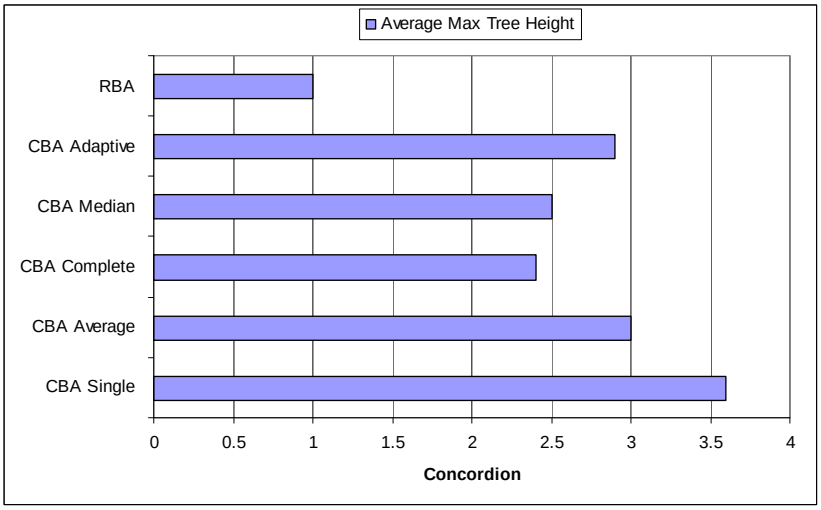

Σχήμα 5.6 Μέση τιμή Total Interfaces για το JUnit.



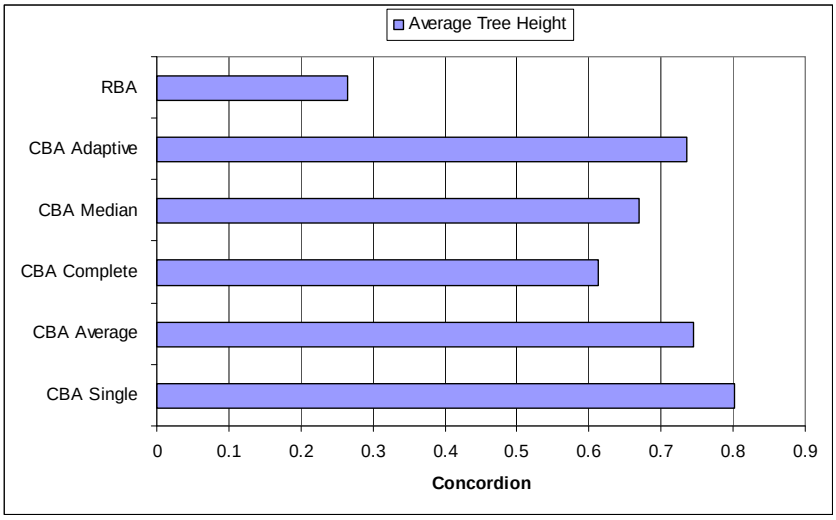

Σχήμα 5.7 Μέση τιμή ACD για το Concordion.



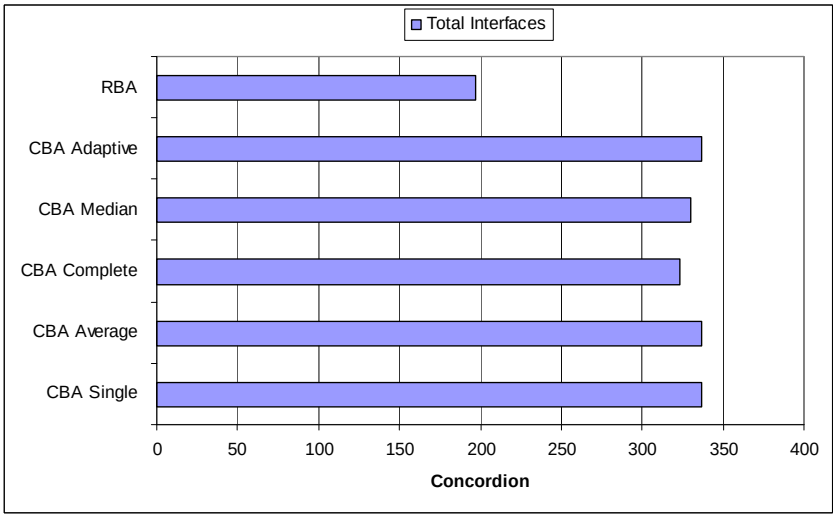

Σχήμα 5.8 Μέση τιμή CBO για το Concordion.



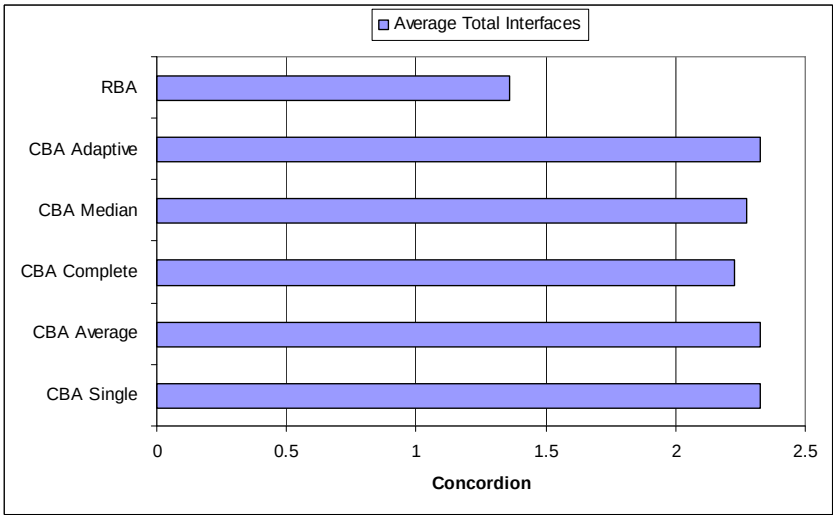

Σχήμα 5.9 Μέση τιμή Max Tree Height για το Concordion.




Σχήμα 5.10 Μέση τιμή Tree Height για το Concordion.




Σχήμα 5.11 Total Interfaces για το Concordion.




Σχήμα 5.12 Μέση τιμή  Total Interfaces για το Concordion.

# ΚΕΦΑΛΑΙΟ 6. ΣΥΜΠΕΡΑΣΜΑΤΑ

Στην παρούσα εργασία, ασχοληθήκαμε με τη βελτίωση της ποιότητας του λογισμικού, μέσω της αυτοματοποιημένης ανακατασκευής προγραμματιστικών διεπαφών (interfaces) για αντικειμενοστραφή προγράμματα. Συγκεκριμένα, χρησιμοποιήθηκε η Αρχή Διαχωρισμού Διεπαφών (Interface Segregation Principle – ISP). Στην εργασία, προτάθηκαν δύο προσεγγίσεις για την βελτίωση της ποιότητας του λογισμικού που βασίζονται στην ISP. Η πρώτη, βασίζεται σε θεμελιώδεις τεχνικές αφαίρεσης για την ανακατασκευή λογισμικού (Refactoring Based Approach - RBA) και αποτελείται από μια επαναληπτική διαδικασία, δύο φάσεων. Η πρώτη φάση περιλαμβάνει την εξαγωγή των διεπαφών (Extract Interfaces) και η επόμενη, την εξαγωγή υπερκλάσεων (Extract Superclass). Η δεύτερη προσέγγιση, εδράζεται σε παραδοσιακές τεχνικές ιεραρχικής ομαδοποίησης (Clustering Based Approach – CBA). Επιπλέον, προχωρήσαμε στην αυτοματοποίηση των δύο προσεγγίσεων και την υλοποίηση ενός εργαλείου για τον σκοπό αυτό. Για να εκτιμηθεί η απόδοση των προτεινόμενων προσεγγίσεων, εξετάσθηκε η εφαρμογή τους σε δύο περιβάλλοντα ανοιχτού λογισμικού, το JUnit και το Concordion. Συμπερασματικά, παρατηρώντας τα αποτελέσματα των πειραμάτων, θα μπορούσαμε να αποφανθούμε ότι ο αλγόριθμος της RBA επιτυγχάνει τον βέλτιστο συνδυασμό των κριτηρίων σύγκρισης που χρησιμοποιήθηκαν, δηλαδή το ποσοστό μείωσης των ανατιθέμενων μεθόδων για κάθε κλάση, τη σύζευξη μεταξύ των κλάσεων και την πολυπλοκότητα των ιεραρχιών από διεπαφές που δημιουργούνται.

# ΑΝΑΦΟΡΕΣ

[1] Adnan, R., Graaf, B., van Deursen, A. and Zonneveld, J. Using Cluster Analysis to Improve the Design of Component Interfaces 2008.

[2] Chidamber, S. and Kemerer, C. A Metrics Suite for Object Oriented Design, IEEE Transactions on Software Engineering, Vol. 20(6). pp 476-493, June 1994.

[3] Du Bois, B., Demeyer, S. and Verelst, J. Refactoring - improving coupling and cohesion of existing code, 2004.

[4] Fowler, M., Beck, K., Brant, J., Opdyke, W. and Roberts, D. Refactoring: Improving the Design of Existing Code, 2002.

[5] Gamma, E., Helm, R., Johnson, R. and Vlissides, J. Design Patterns: Elements of Reusable Object-Oriented Software.

[6] Kataoka, Y., Ernst, M. D., Griswold, W. G. and Notkin, D. Automated support for program refactoring using invariants, 2001.

[7] Kiezun, A., Ernst, M. D., Tip, F. and Fuhrer, R. M. Refactoring for Parameterizing Java Classes, 2007.

[8] Martin, R. Desing Principles and Design Patterns.

[9] Mens, T. and Tourwe, T. A survey of software refactoring, 2004.

[10] Simon, F., Steinbruckner, F. and Lewerentz, C. Metrics based refactoring, 2001.

[11] Steinmann, F. The Infer Type Refactoring and its Use for Interface-Based Programming. Special Issue OOPS Track at SAC, Vol. 6(2), February 2007.

[12] Tokuda, L. and Batory, D. Evolving object-oriented designs with refactorings, 1999.

[13] Tsantalis, N. and Chatzigeorgiou, A. Identification of Extract Method Refactoring Opportunities, 2009

[14] Tsantalis, N. and Chatzigeorgiou, A. Identification of Move Method Refactoring Opportunities, 2009

# ΠΑΡΑΡΤΗΜΑ Α

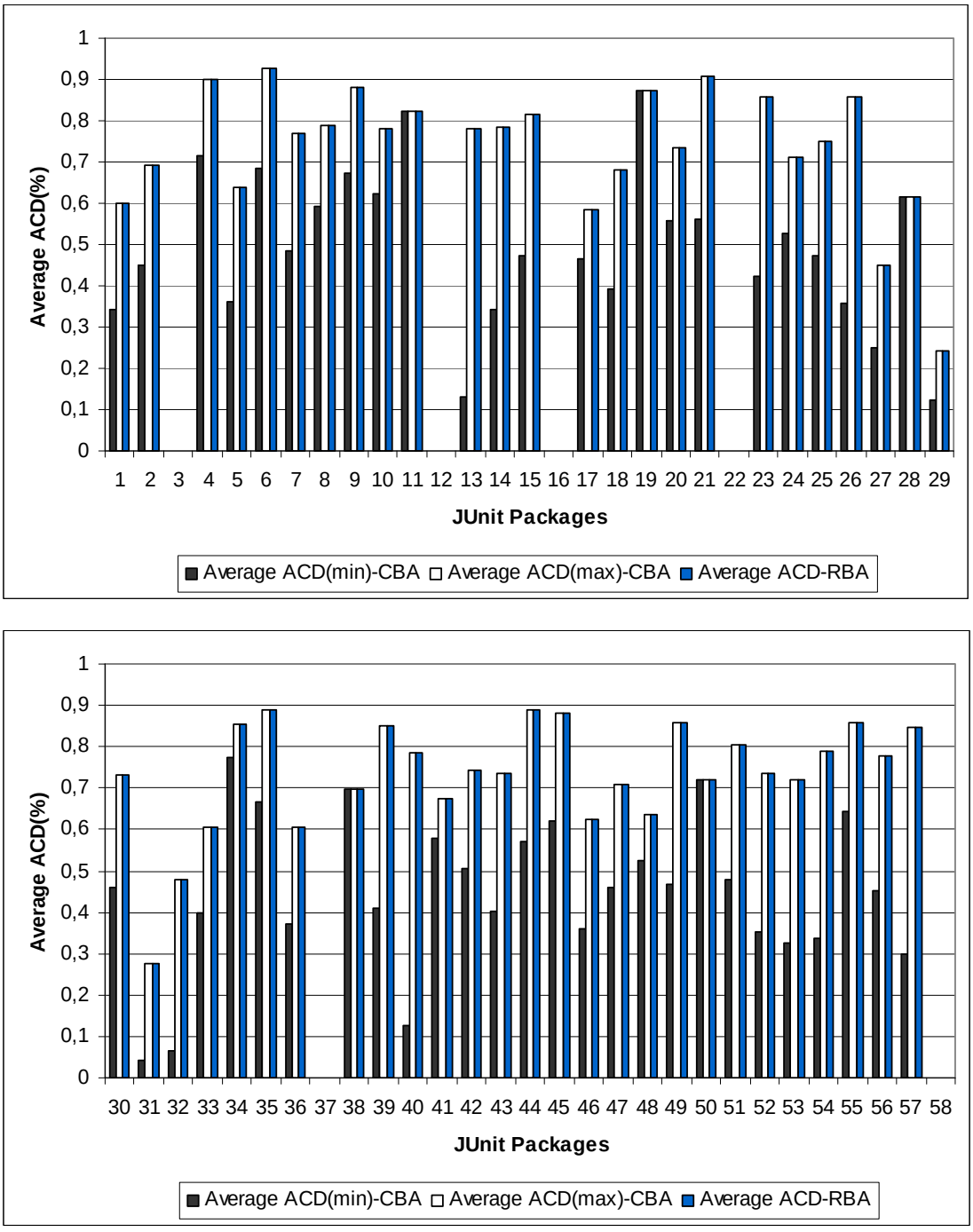
Οι τιμές που παρουσιάζονται στα σχήματα του Παραρτήματος Α, αποτελούν τη μέση τιμή για κάθε πακέτο του JUnit. Ο συγκεκριμένος τρόπος παρουσίασης επιλέχθηκε για πρακτικούς και εποπτικούς λόγους, καθώς το μεγάλο πλήθος των κλάσεων του κάθε περιβάλλοντος δεν θα διευκόλυνε την εξαγωγή των συμπερασμάτων. Εξαίρεση, αποτελεί το κριτήριο σύγκρισης των συνολικών διεπαφών (Total Interfaces) όπου παρουσιάζεται το συνολικό πλήθος διεπαφών που δημιουργείται, με κάθε αλγόριθμο. Τα αποτελέσματα παρουσιάζονται, συγκρίνοντας κάθε φορά, τις επιδόσεις των αλγορίθμων της CBA και της RBA, για κάθε ένα από τα έξι (6) κριτήρια σύγκρισης που αναφέρθηκαν στο Κεφάλαιο 5 και κάθε ένα από τα πέντε (5) κριτήρια απόστασης που έχουν παρουσιαστεί στο Κεφάλαιο 3. Για εποπτικούς λόγους, έχουμε αντιστοιχίσει το κάθε πακέτο του JUnit σε ένα μοναδικό αριθμό (Πίνακας Α.1).

Πίνακας Α.1 Ποσοτική Περιγραφή του JUnit.

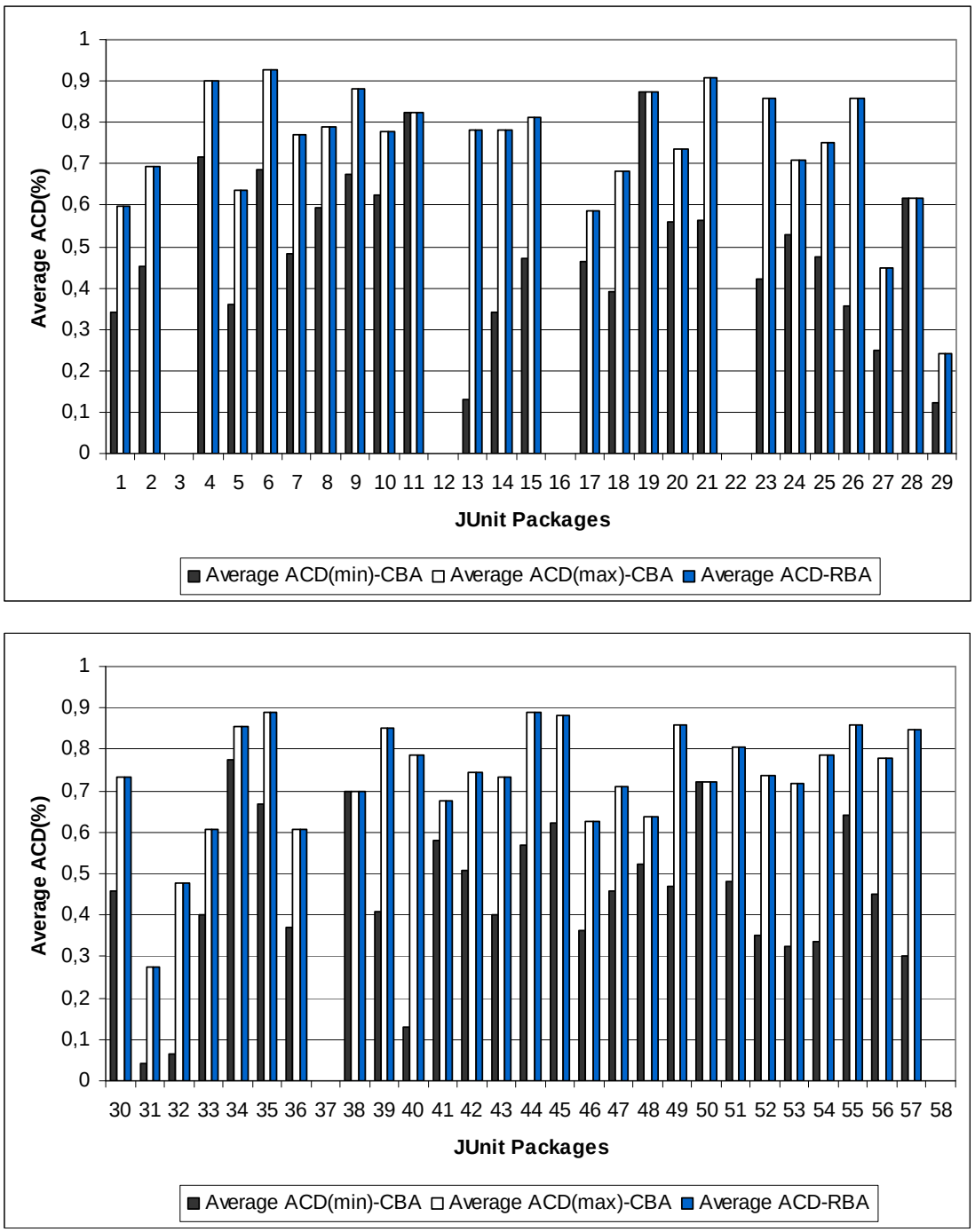
| Πακέτο | Αριθμός αντιστοίχισης | Πλήθος κλάσεων |
|---|---|---|
| junit.extensions | 1 | 5 |
| junit.framework | 2 | 15 |
| junit.runner | 3 | 4 |
| junit.samples | 4 | 4 |
| junit.samples.money | 5 | 5 |
| junit.tests | 6 | 3 |
| junit.tests.extensions | 7 | 5 |
| junit.tests.framework | 8 | 23 |
| junit.tests.runner | 9 | 8 |
| junit.textui | 10 | 3 |

| | | |
|---|---|---|
| org.junit | 11 | 12 |
| org.junit.experimental | 12 | 1 |
| org.junit.experimental.categories | 13 | 2 |
| org.junit.experimental.max | 14 | 3 |
| org.junit.experimental.results | 15 | 3 |
| org.junit.experimental.runners | 16 | 1 |
| org.junit.experimental.theories | 17 | 8 |
| org.junit.experimental.theories.internal | 18 | 3 |
| org.junit.experimental.theories.suppliers | 19 | 2 |
| org.junit.internal | 20 | 8 |
| org.junit.internal.builders | 21 | 8 |
| org.junit.internal.matchers | 22 | 6 |
| org.junit.internal.requests | 23 | 4 |
| org.junit.internal.runners | 24 | 12 |
| org.junit.internal.runners.model | 25 | 3 |
| org.junit.internal.runners.rules | 26 | 1 |
| org.junit.internal.runners.statements | 27 | 6 |
| org.junit.matchers | 28 | 2 |
| org.junit.rules | 29 | 13 |
| org.junit.runner | 30 | 9 |
| org.junit.runner.manipulation | 31 | 6 |
| org.junit.runner.notification | 32 | 5 |
| org.junit.runners | 33 | 7 |
| org.junit.runners.model | 34 | 10 |
| org.junit.samples | 35 | 3 |
| org.junit.samples.money | 36 | 2 |
| org.junit.tests | 37 | 4 |
| org.junit.tests.assertion | 38 | 4 |
| org.junit.tests.deprecated | 39 | 1 |
| org.junit.tests.description | 40 | 3 |
| org.junit.tests.experimental | 41 | 4 |

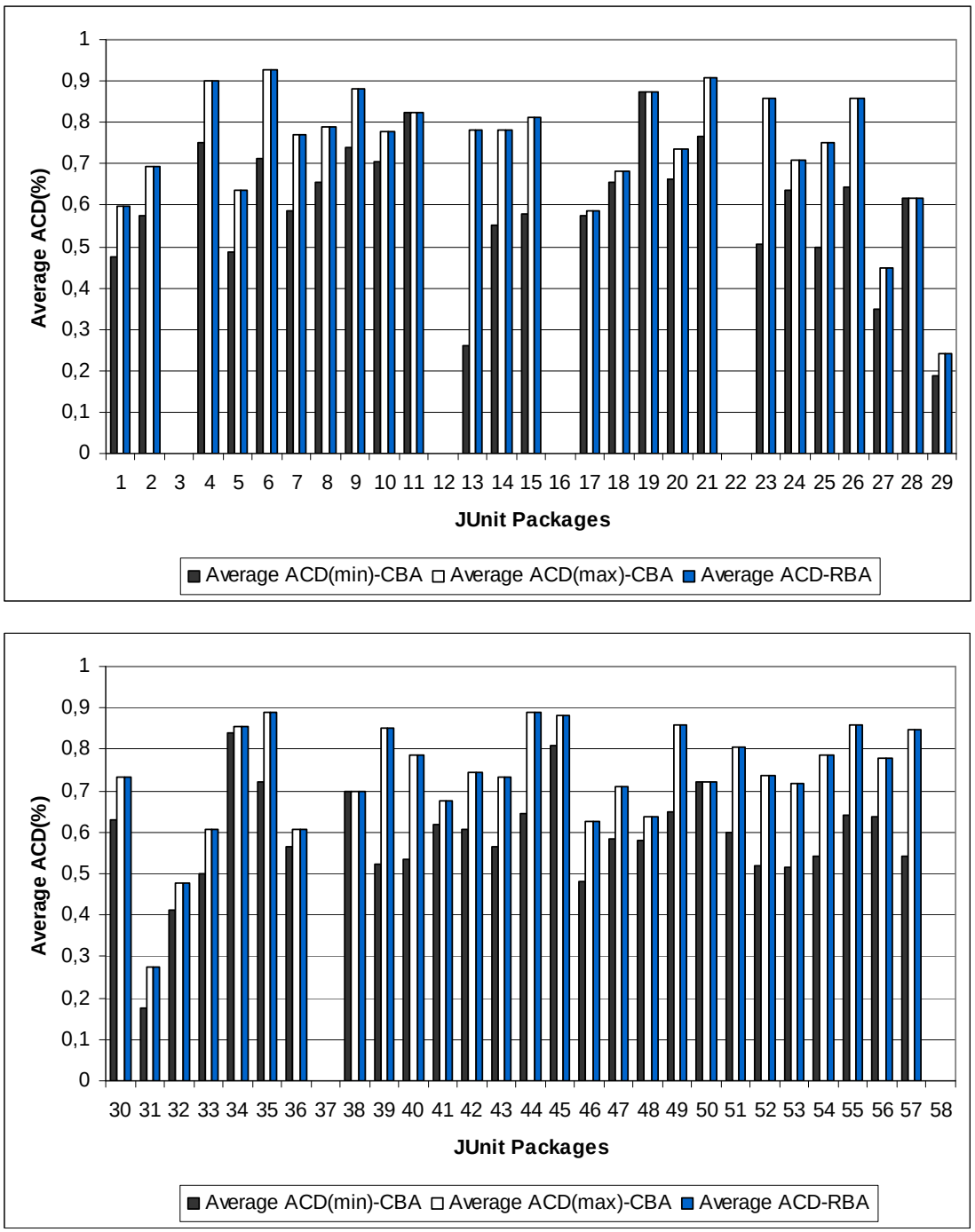
| | | |
|---|---|---|
| org.junit.tests.experimental.categories | 42 | 2 |
| org.junit.tests.experimental.max | 43 | 3 |
| org.junit.tests.experimental.parallel | 44 | 2 |
| org.junit.tests.experimental.results | 45 | 2 |
| org.junit.tests.experimental.rules | 46 | 16 |
| org.junit.tests.experimental.theories | 47 | 3 |
| org.junit.tests.experimental.theories.extendingwithstubs | 48 | 9 |
| org.junit.tests.experimental.theories.runner | 49 | 8 |
| org.junit.tests.internal.runners.statements | 50 | 1 |
| org.junit.tests.junit3compatibility | 51 | 9 |
| org.junit.tests.listening | 52 | 5 |
| org.junit.tests.manipulation | 53 | 4 |
| org.junit.tests.running.classes | 54 | 10 |
| org.junit.tests.running.core | 55 | 3 |
| org.junit.tests.running.methods | 56 | 6 |
| org.junit.tests.validation | 57 | 4 |
| org.junit.tests.validation.anotherpackage | 58 | 2 |



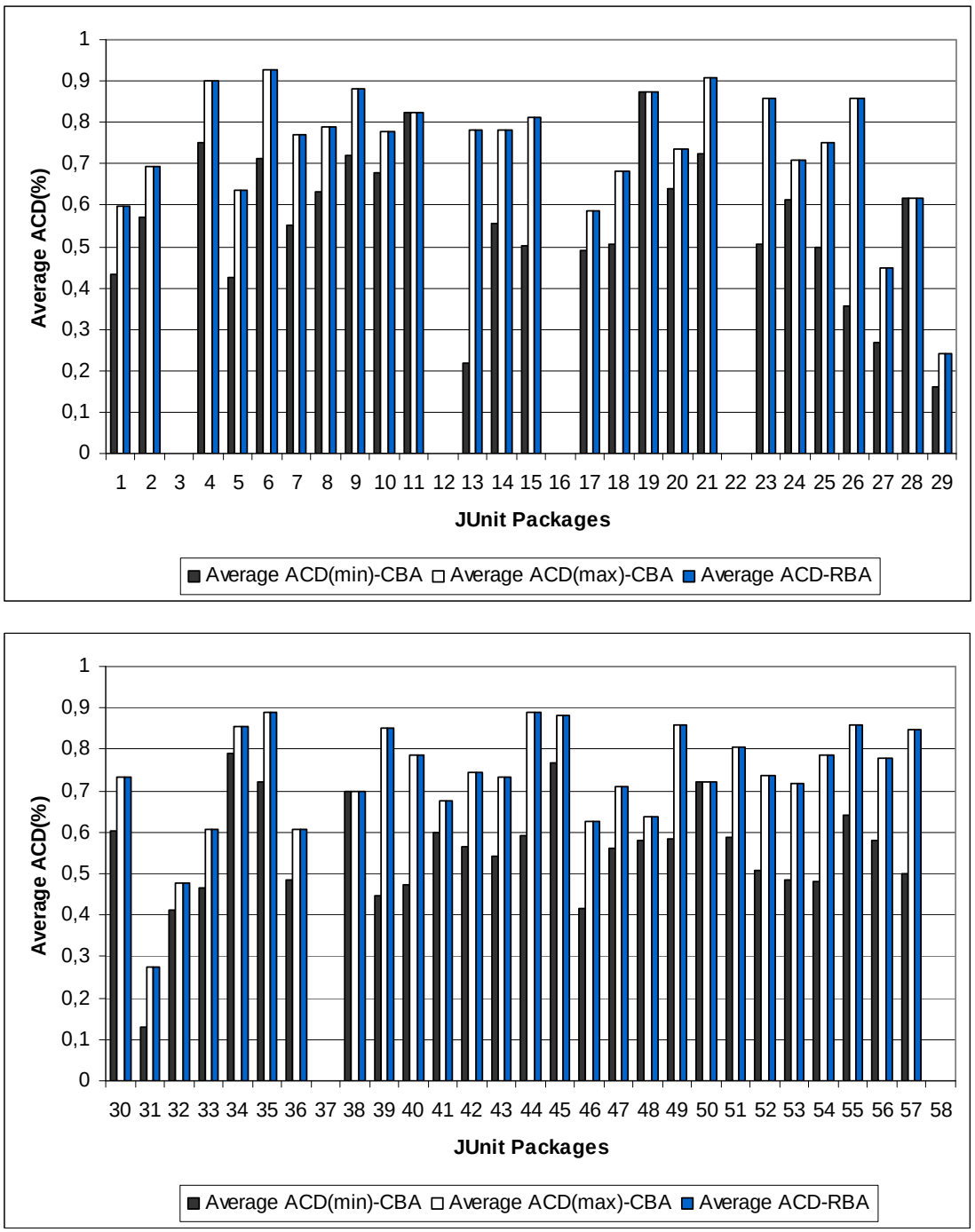

Σχήμα Α.1 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Single και RBA.



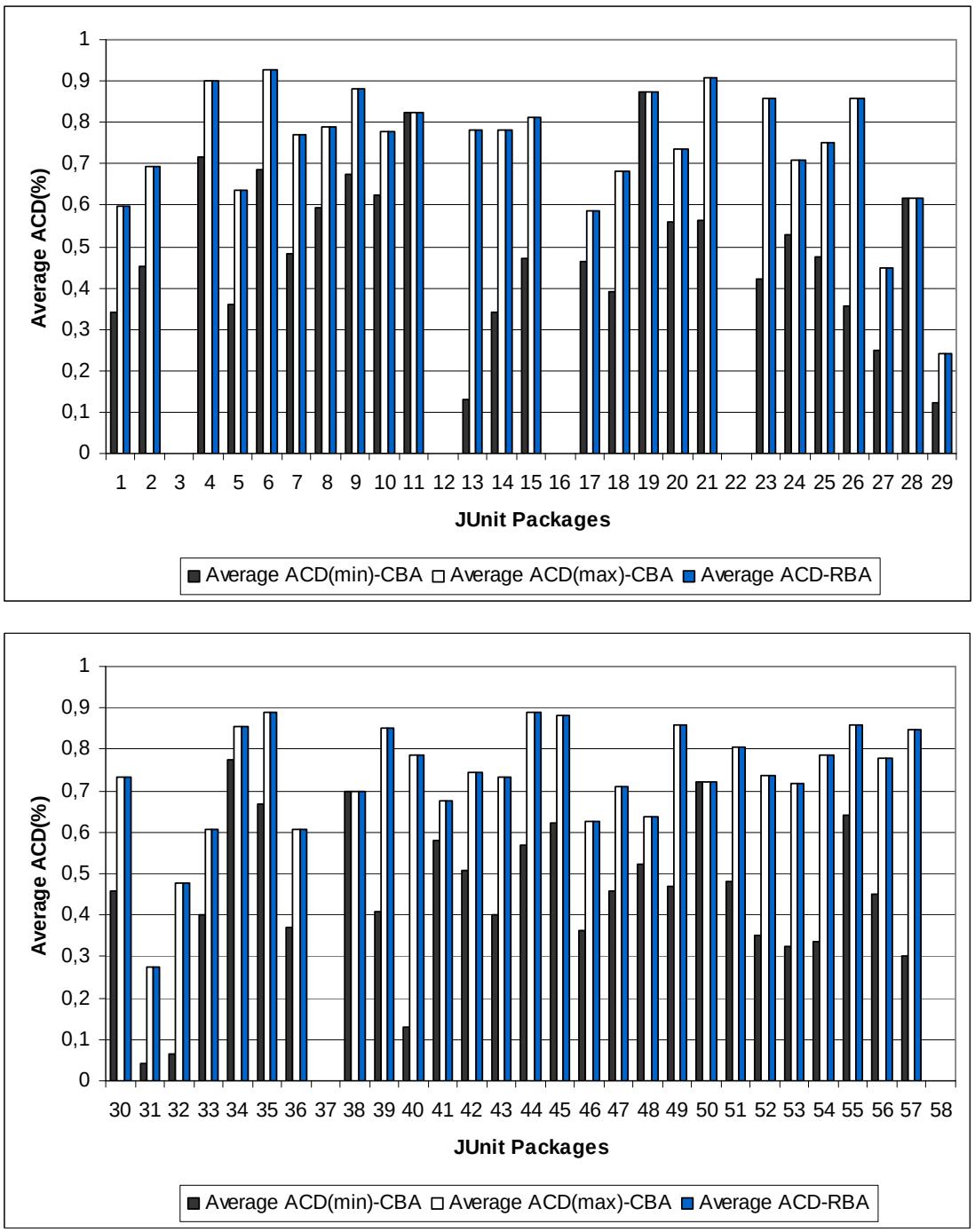

Σχήμα Α.2 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Average και RBA.



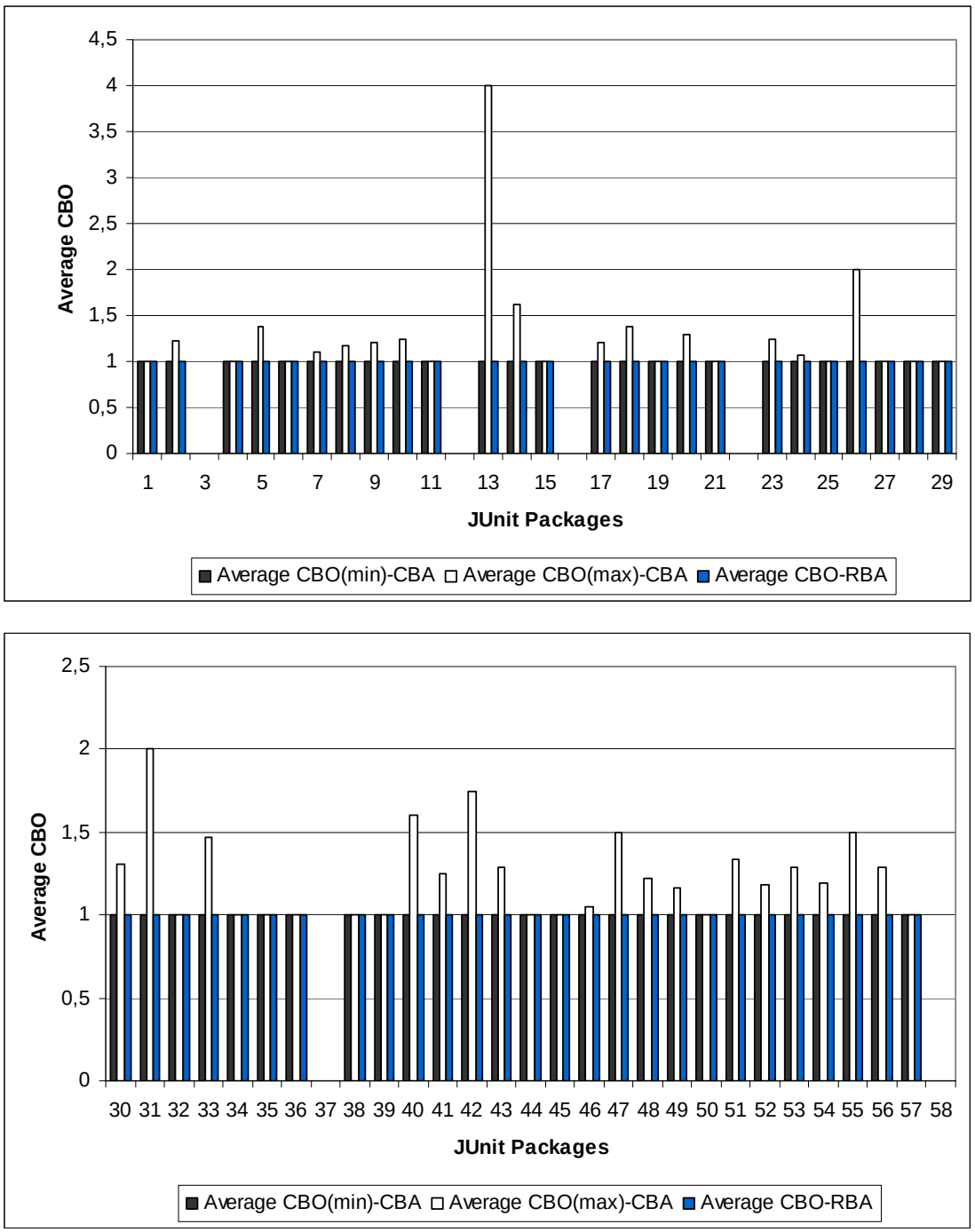

Σχήμα Α.3 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Complete και RBA.



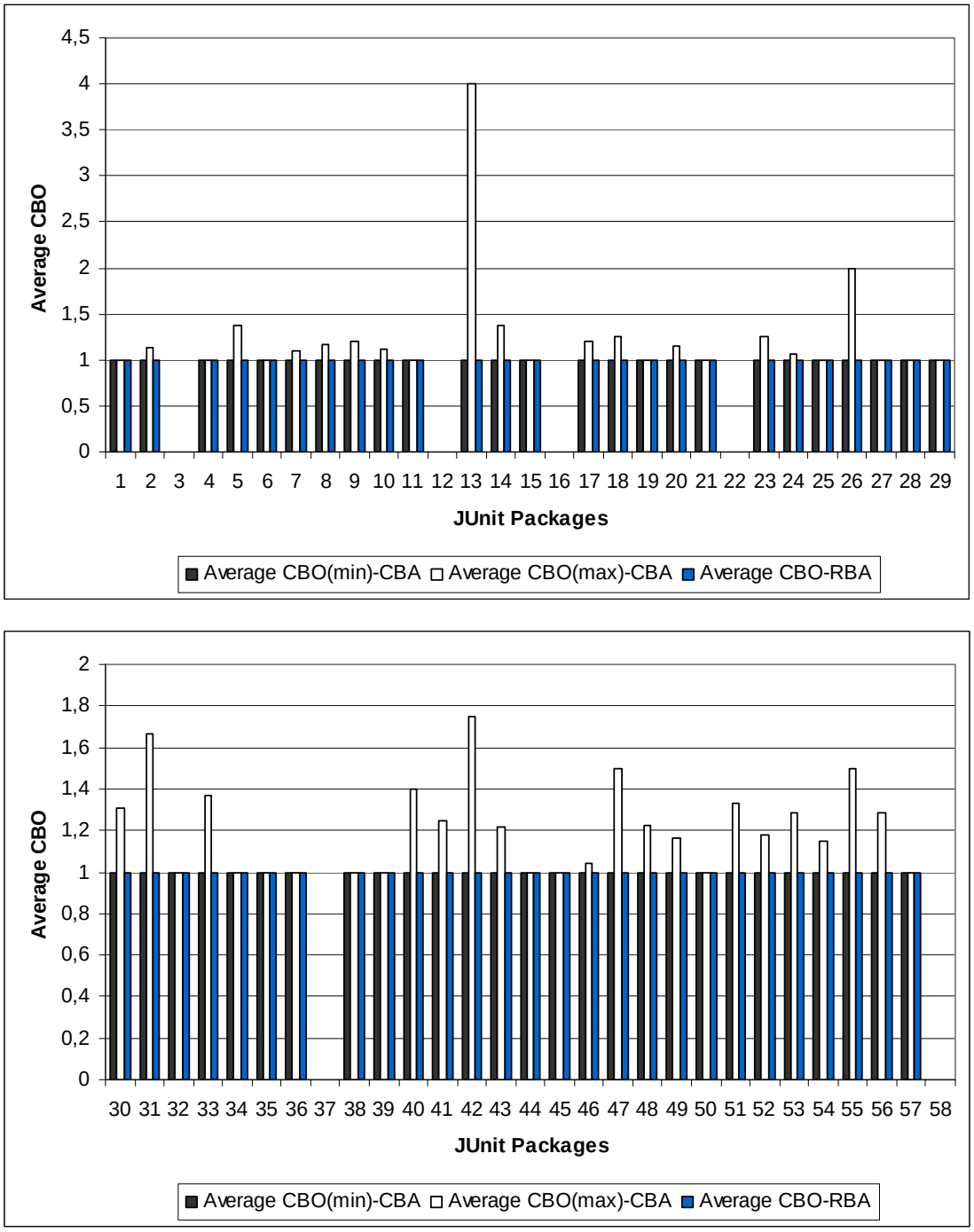

Σχήμα Α.4 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Median και RBA.



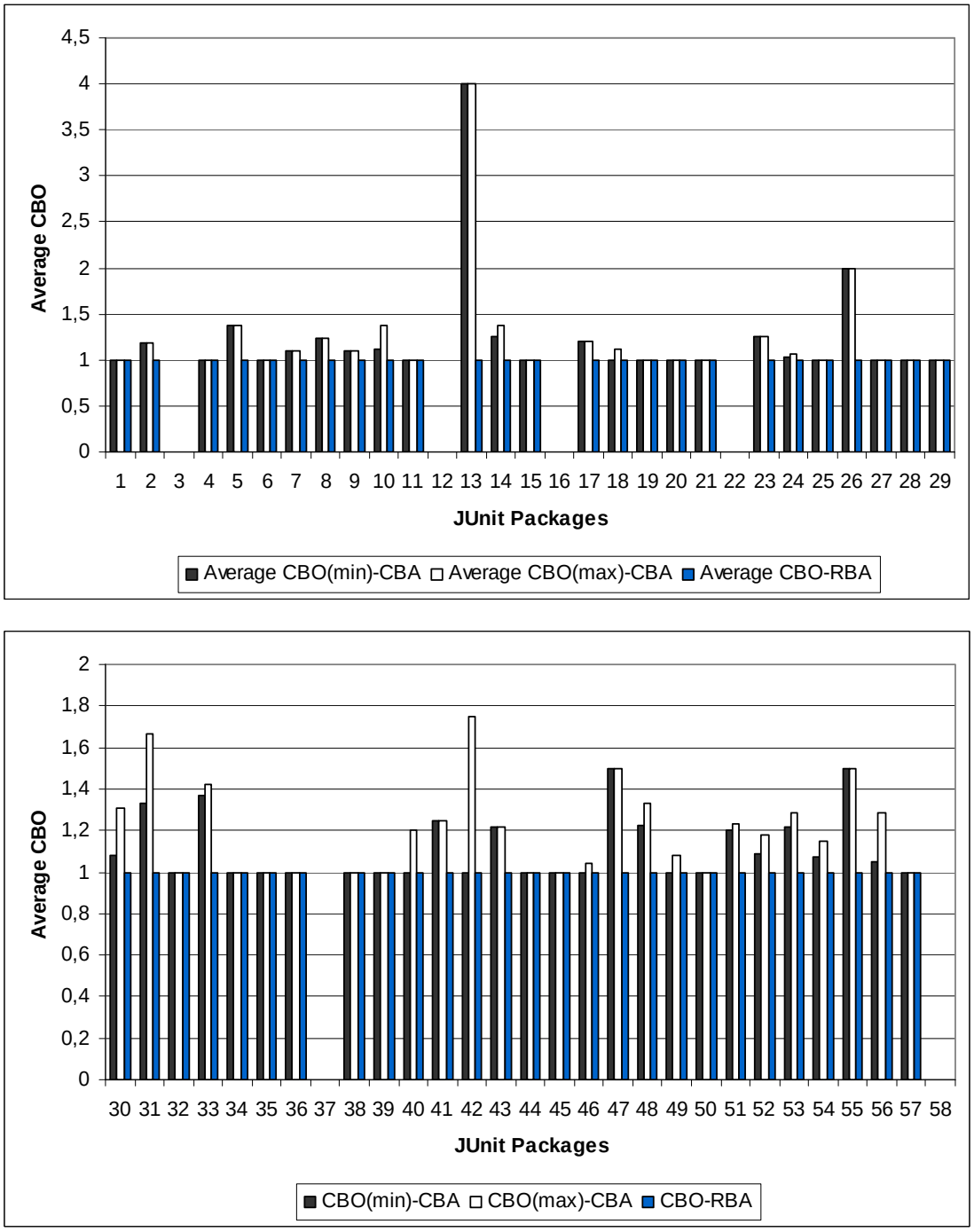

Σχήμα Α.5 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Adaptive και RBA.



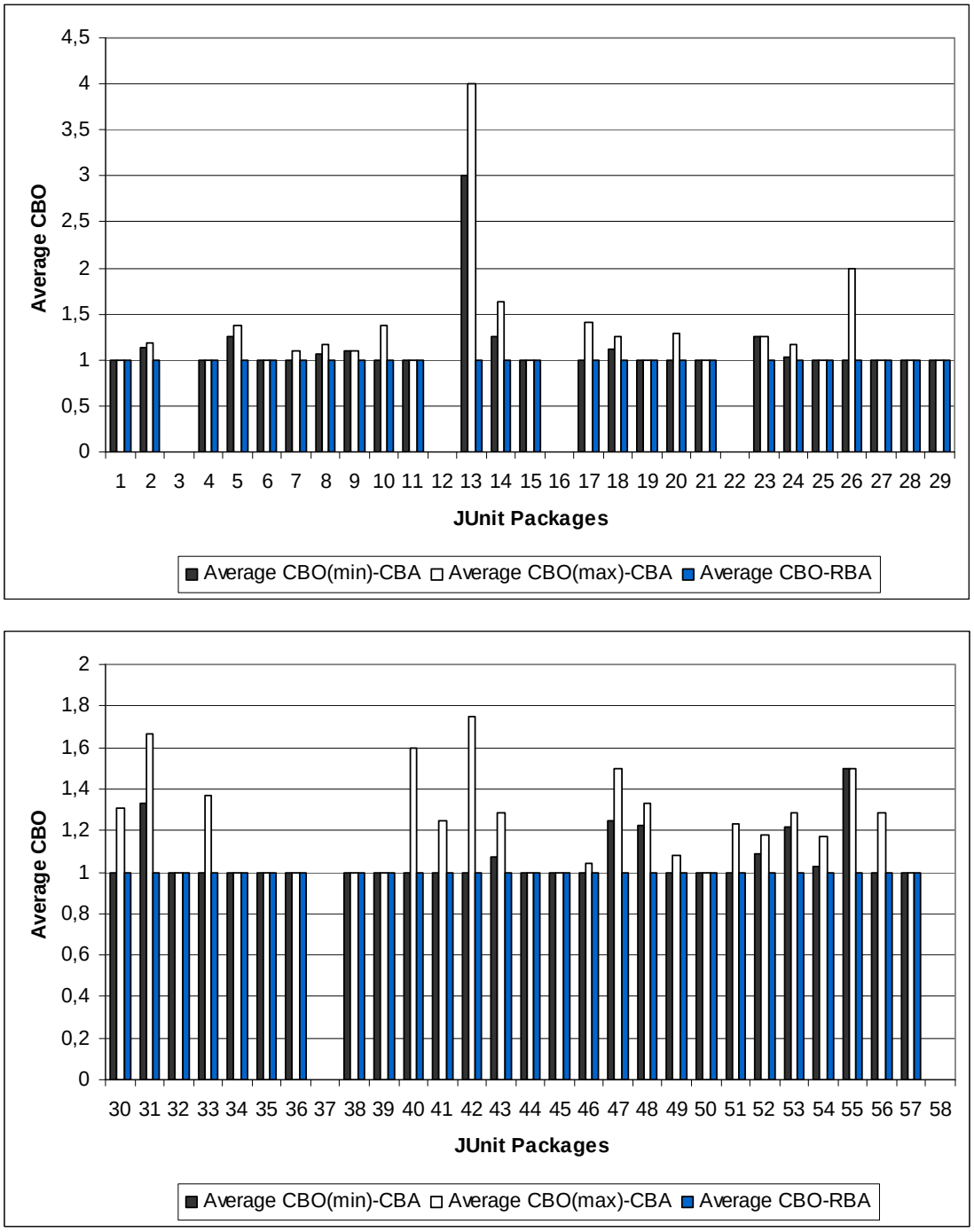

Σχήμα Α.6 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Single και RBA.



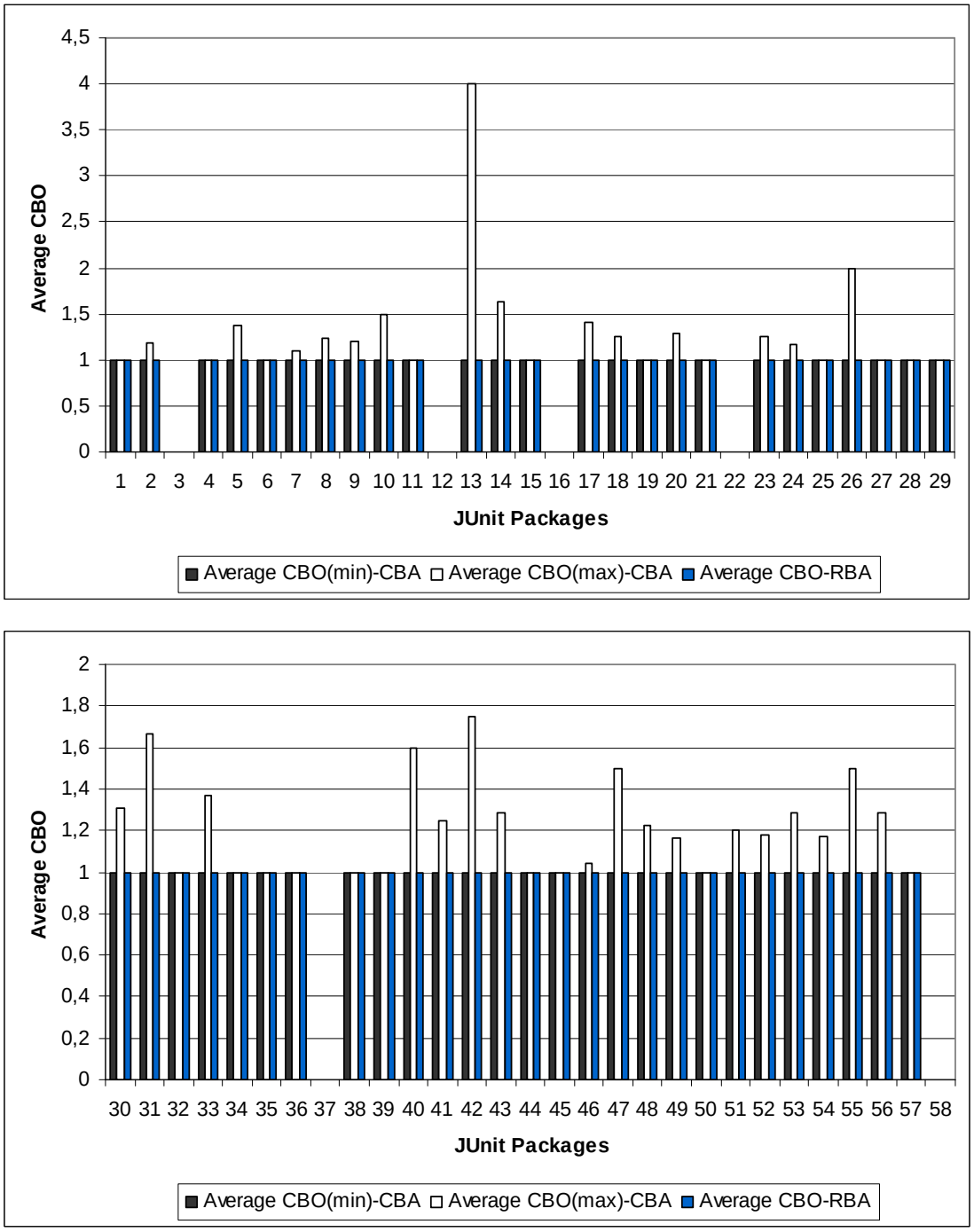

Σχήμα Α.7 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Average και RBA.



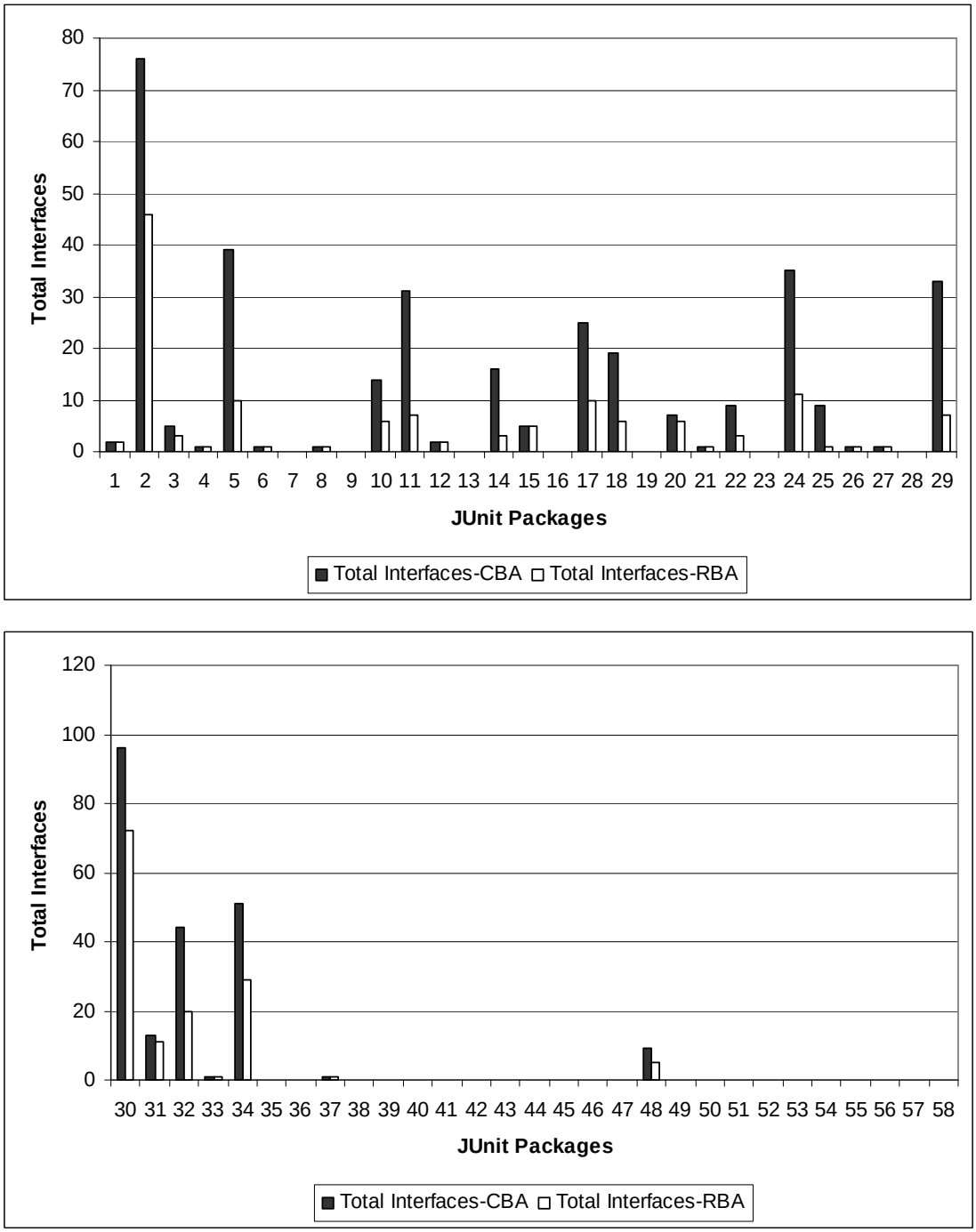

Σχήμα Α.8 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Complete και RBA.



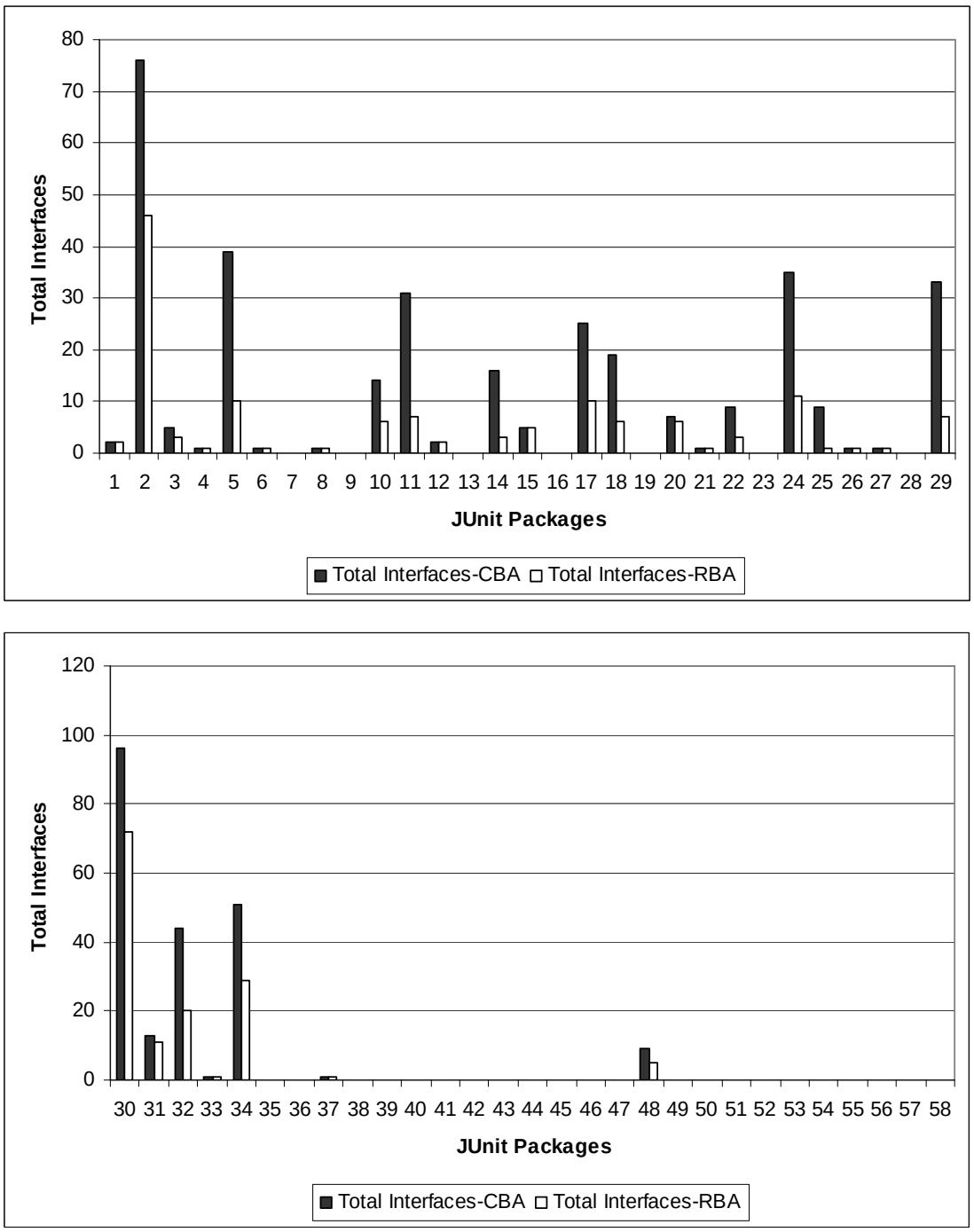

Σχήμα Α.9 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Median και RBA.



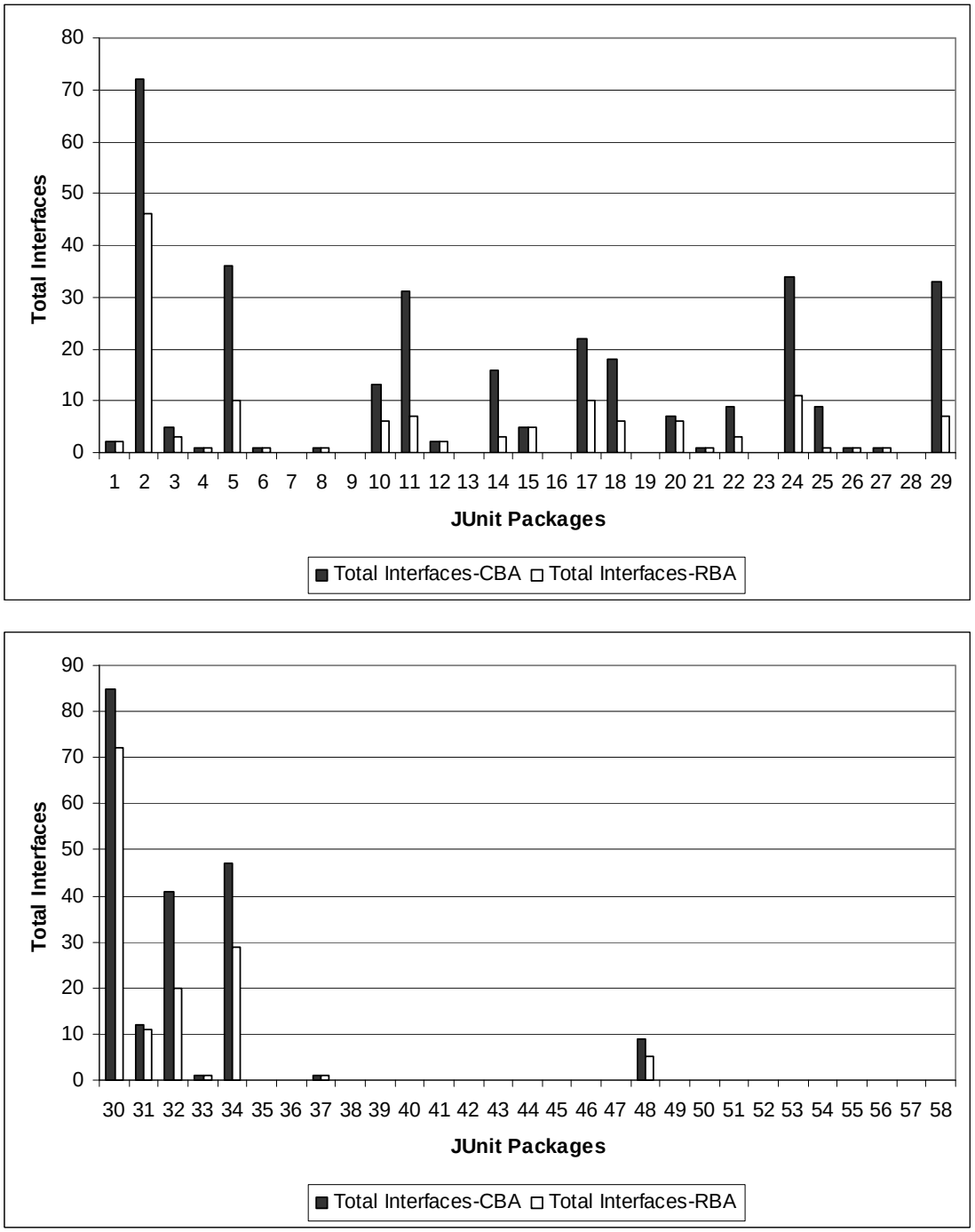

Σχήμα Α.10 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Adaptive και RBA.



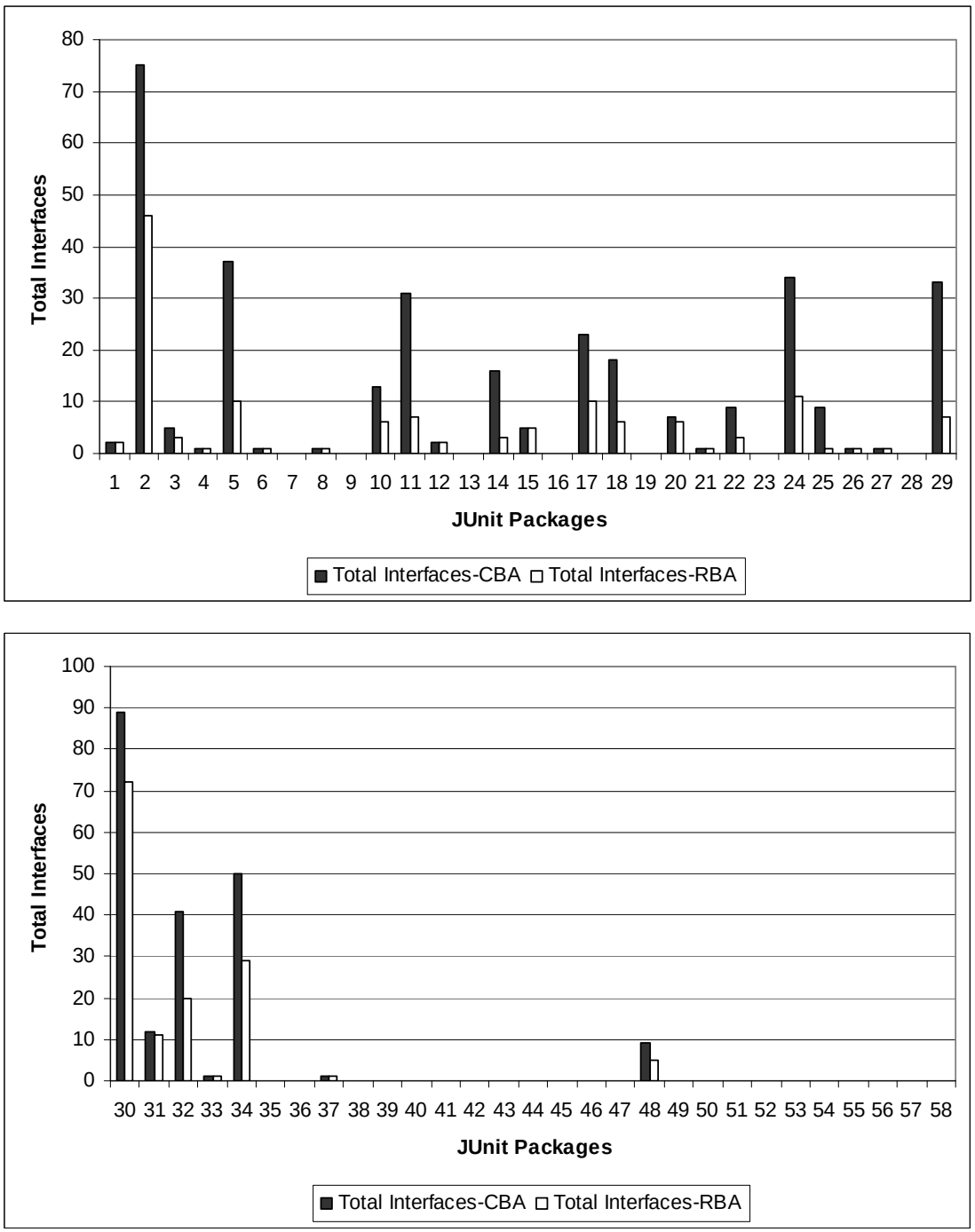

Σχήμα Α.11 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Single και RBA.



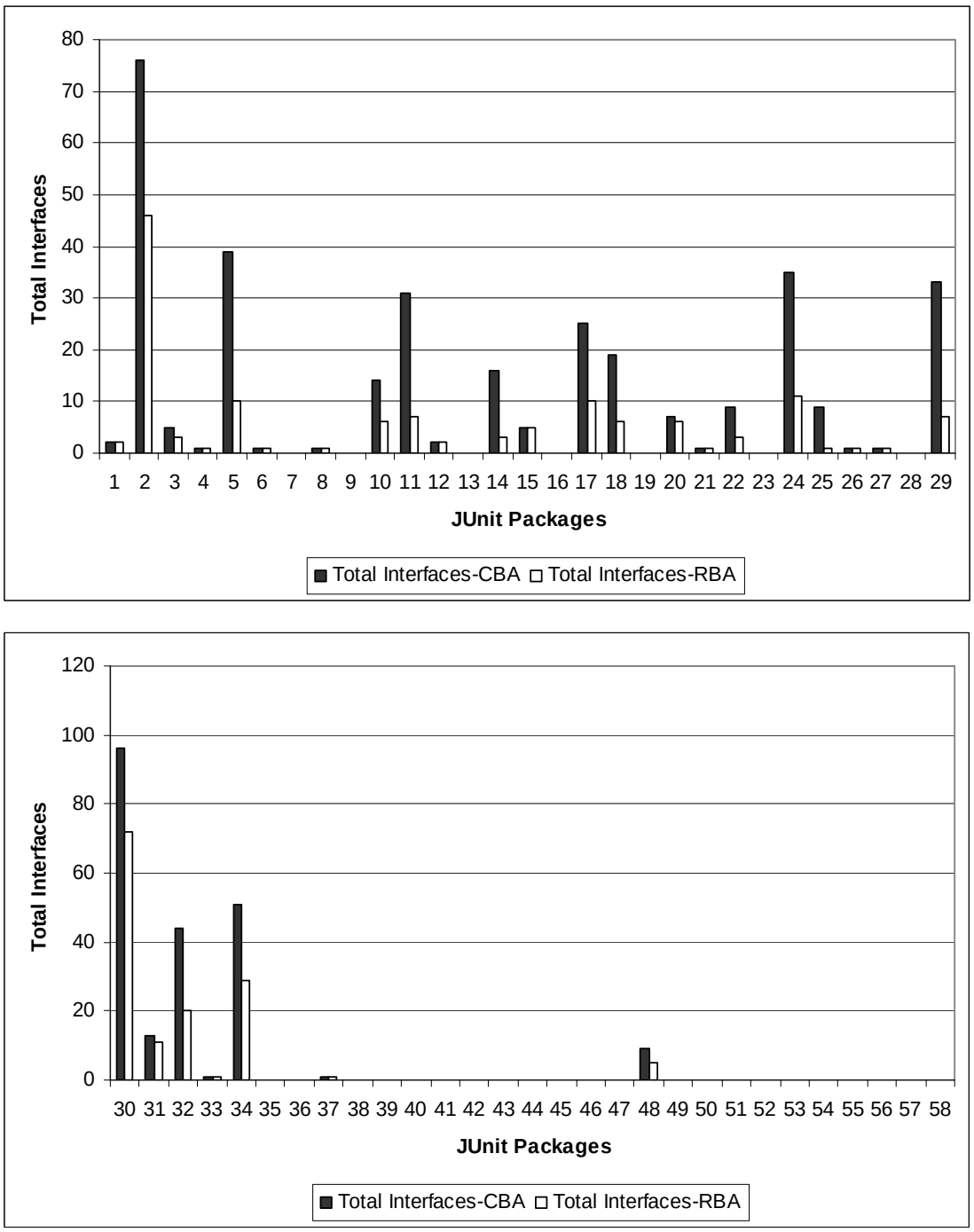

Σχήμα Α.12 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Average και RBA.



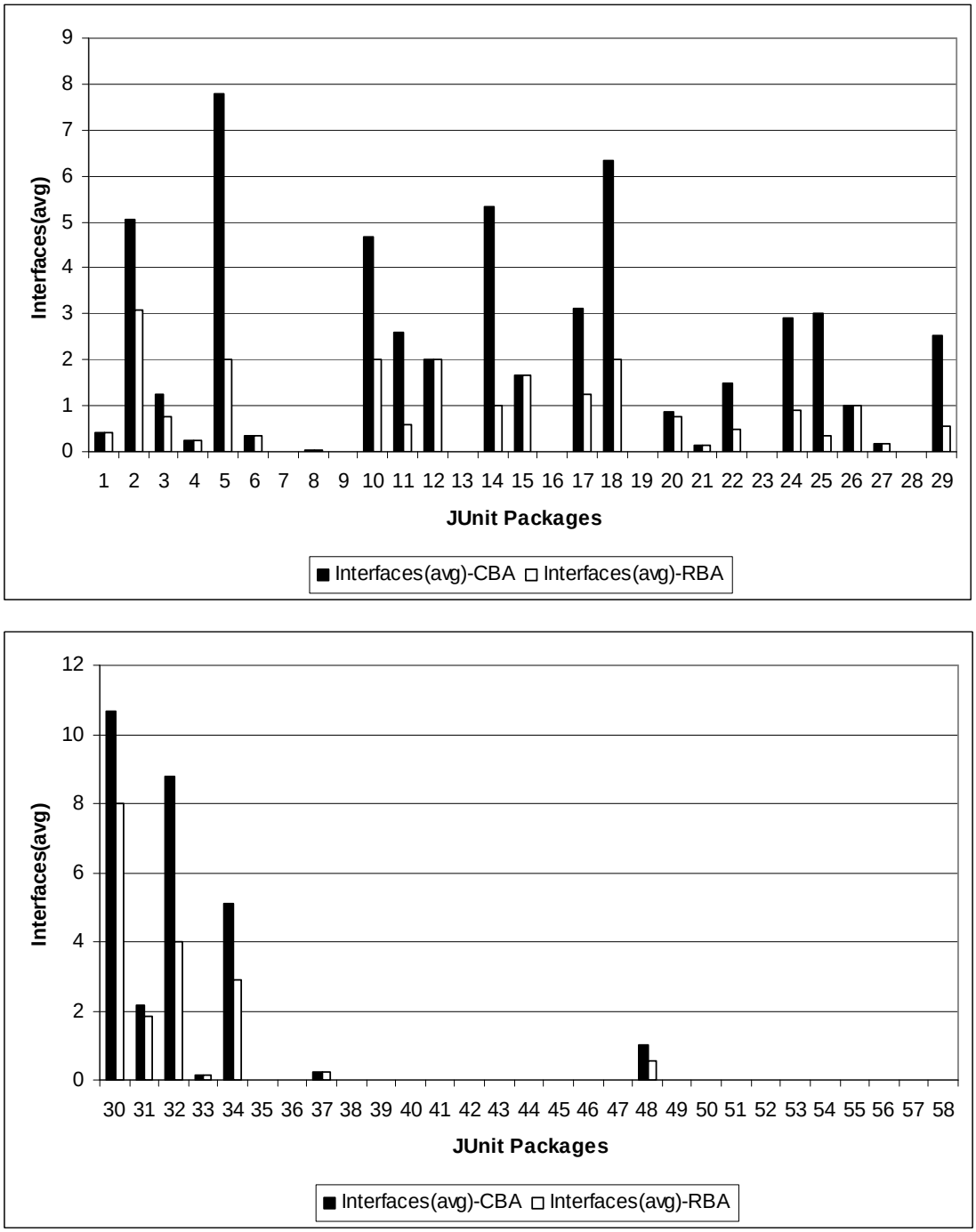

Σχήμα Α.13 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Complete και RBA.



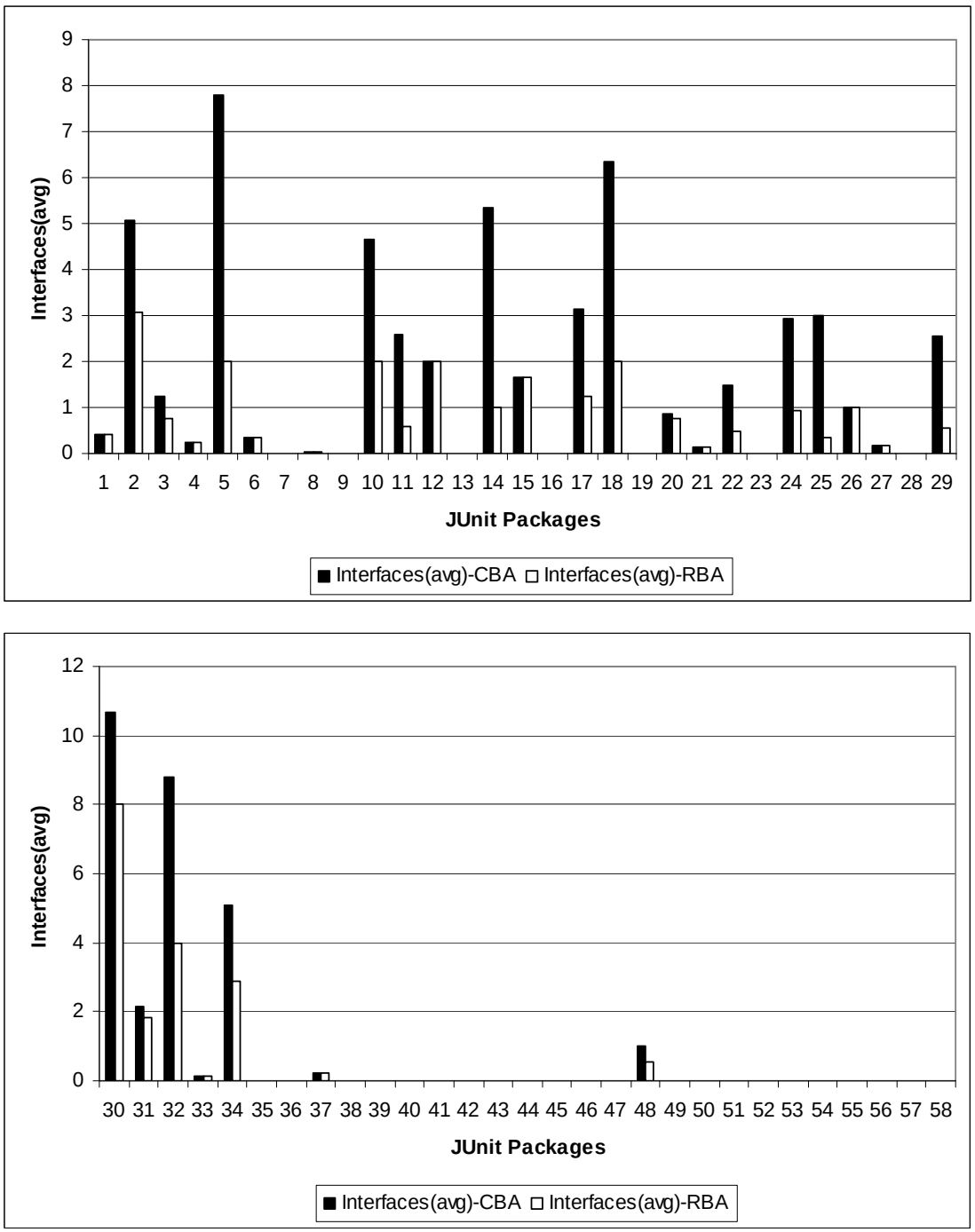

Σχήμα Α.14 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Median και RBA.



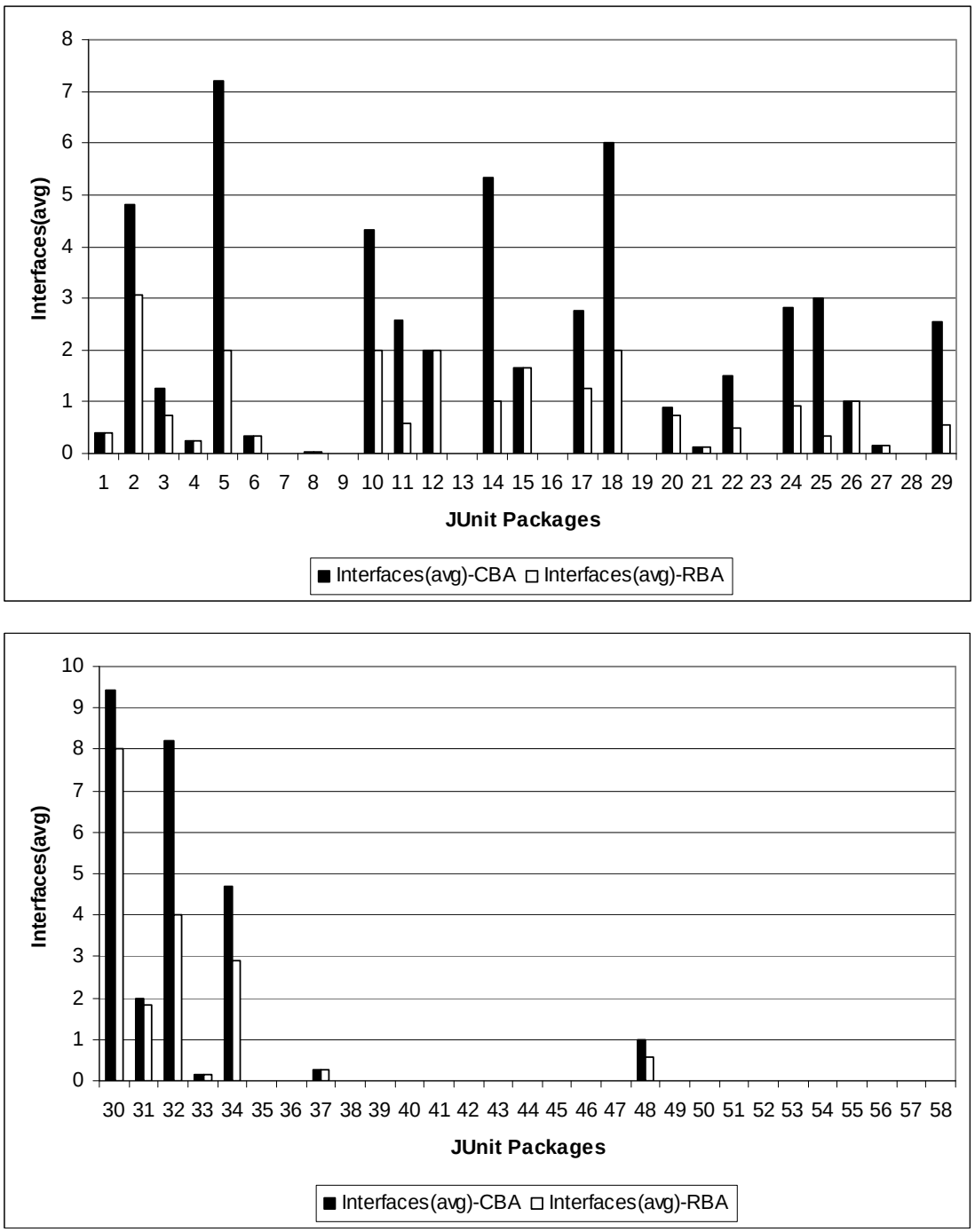

Σχήμα Α.15 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Adaptive και RBA.



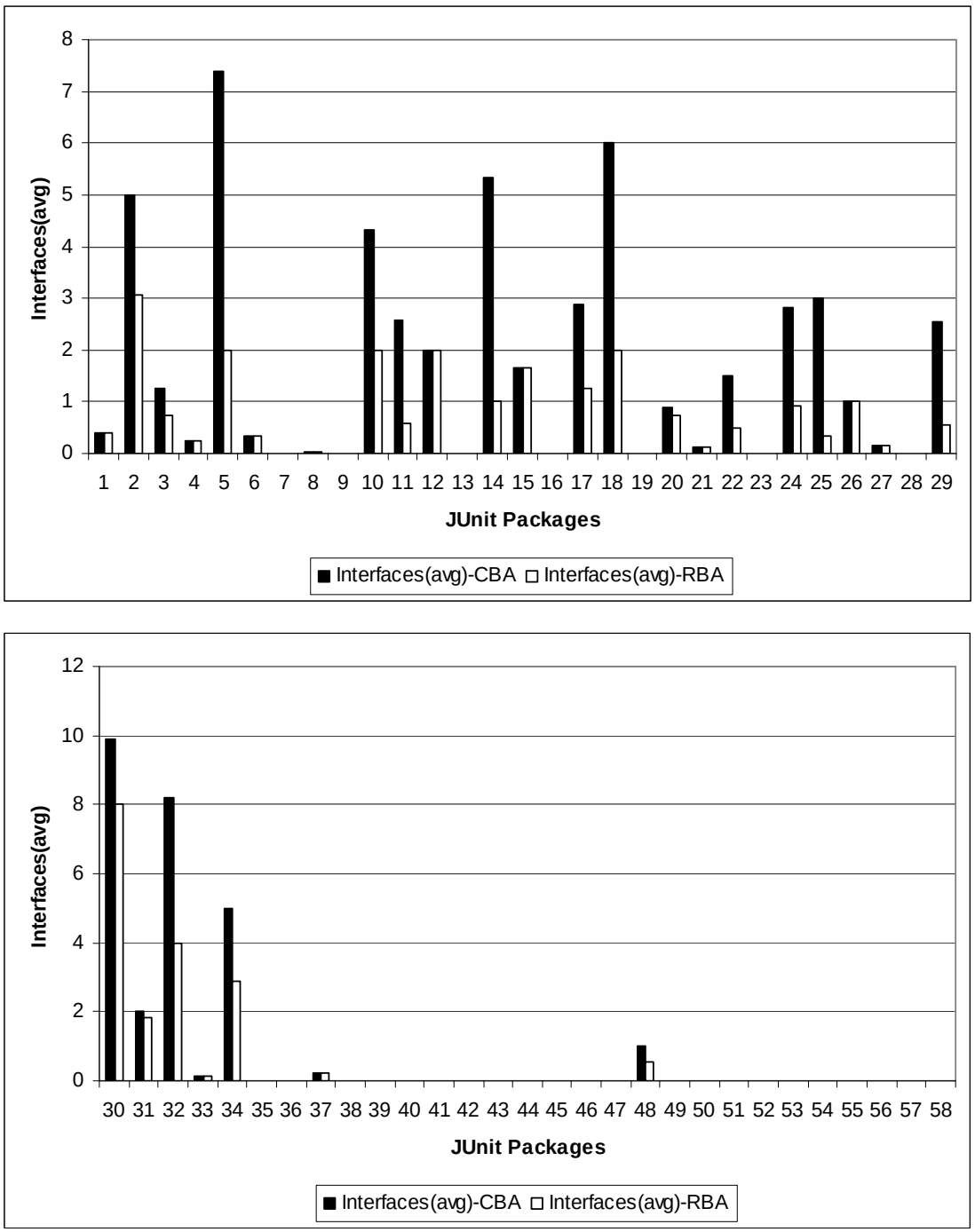

Σχήμα Α.16 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Single και RBA.



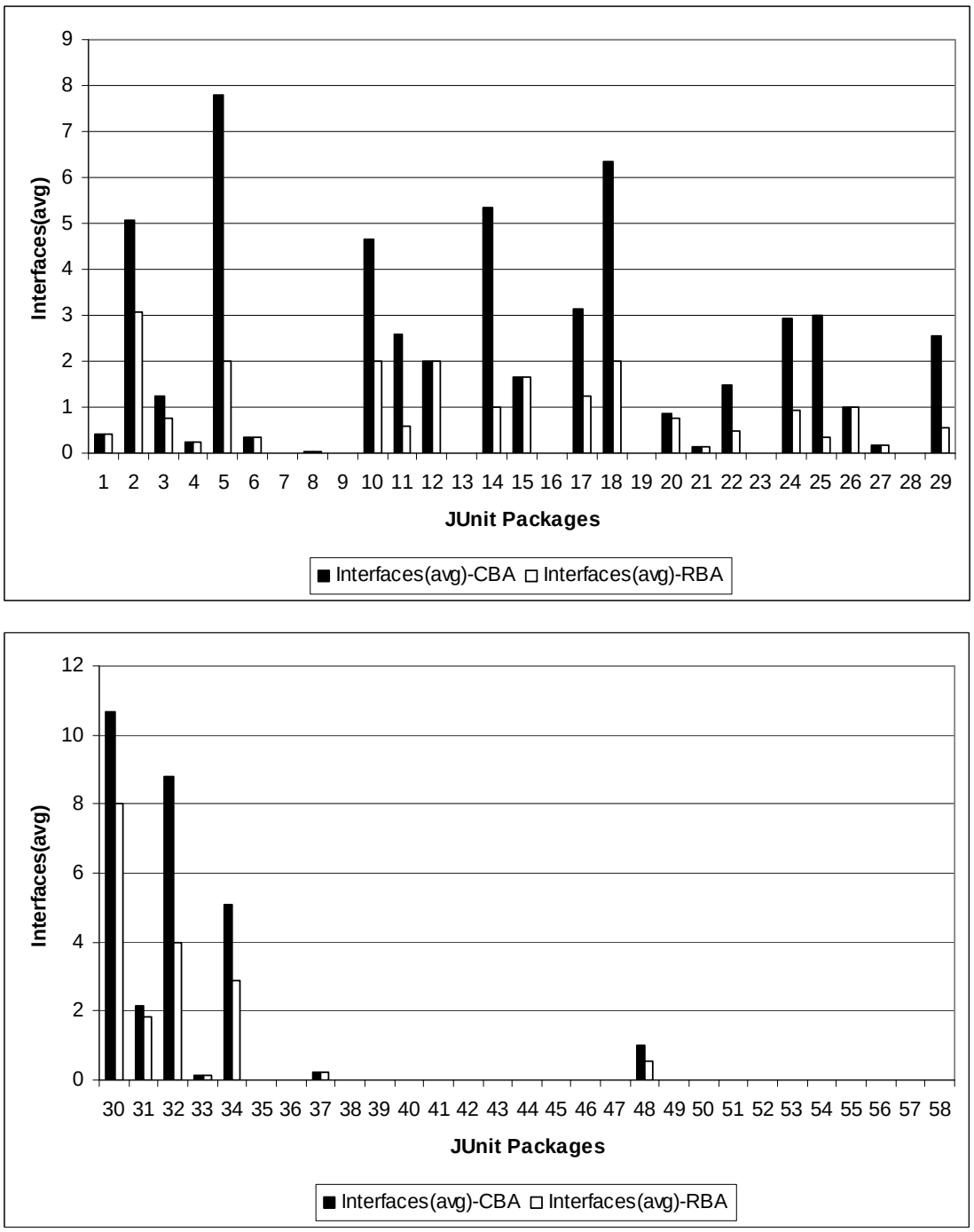

Σχήμα Α.17 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Average και RBA.



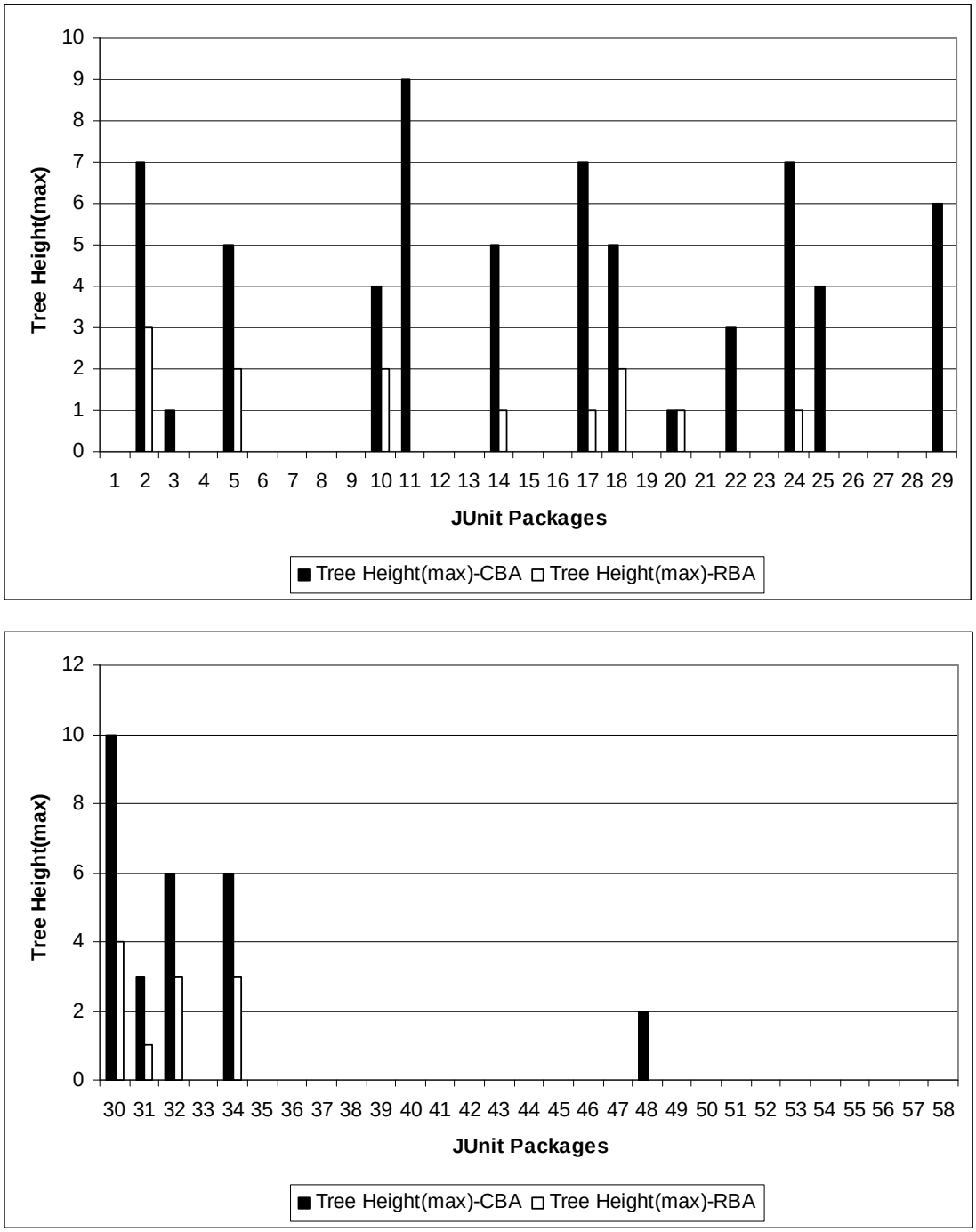

Σχήμα Α.18 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Complete και RBA.



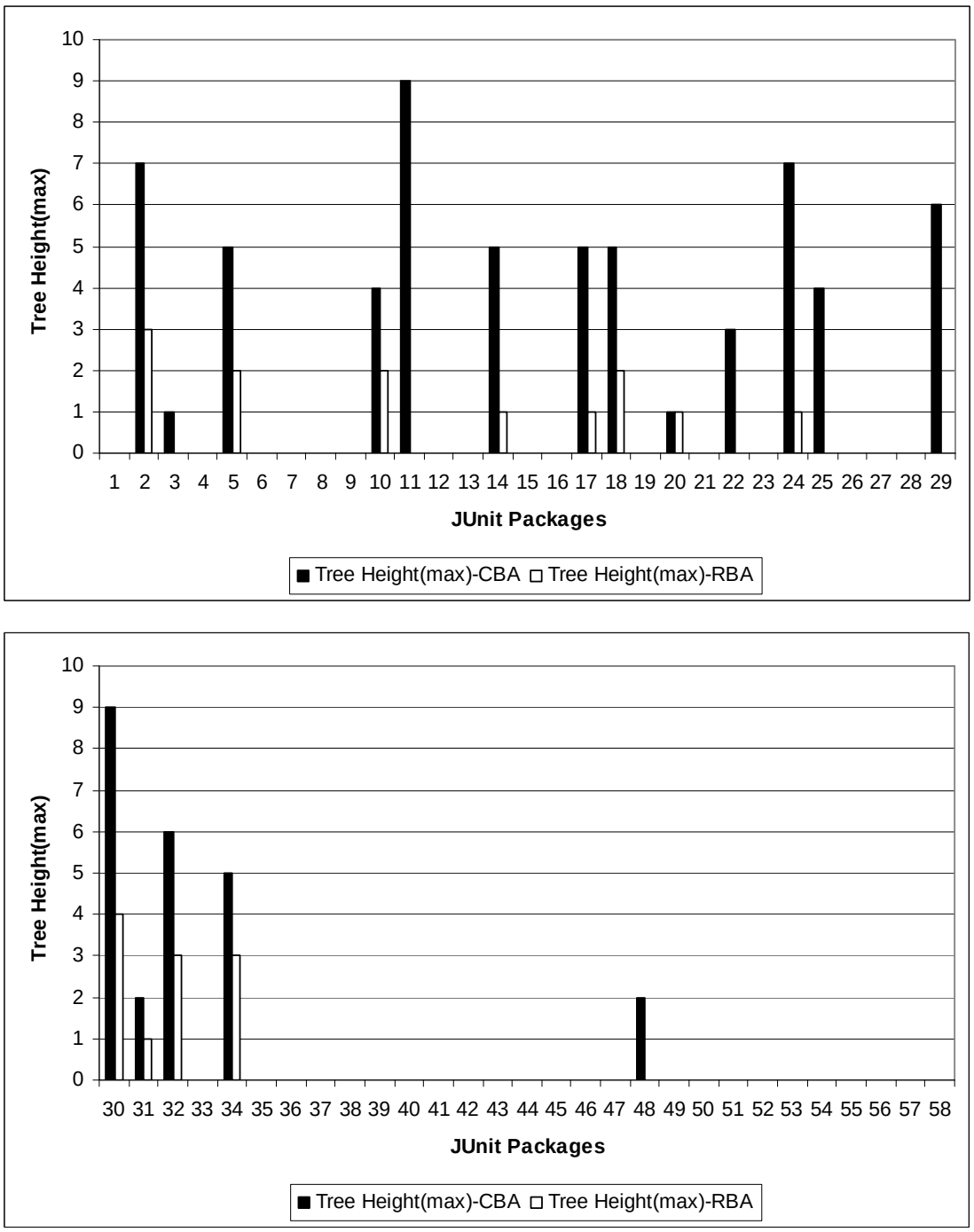

Σχήμα Α.19 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Median και RBA.



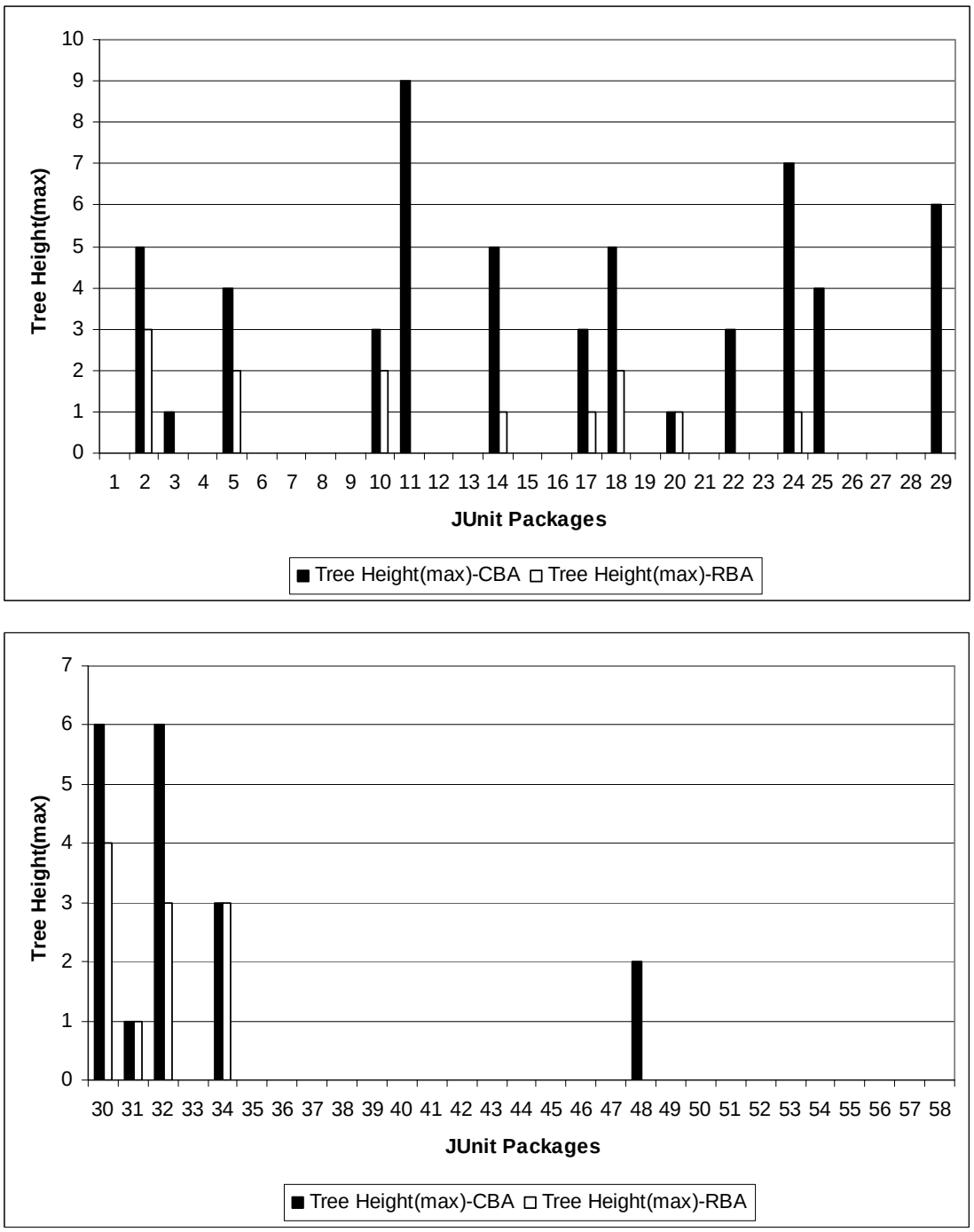

Σχήμα Α.20 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Adaptive και RBA.



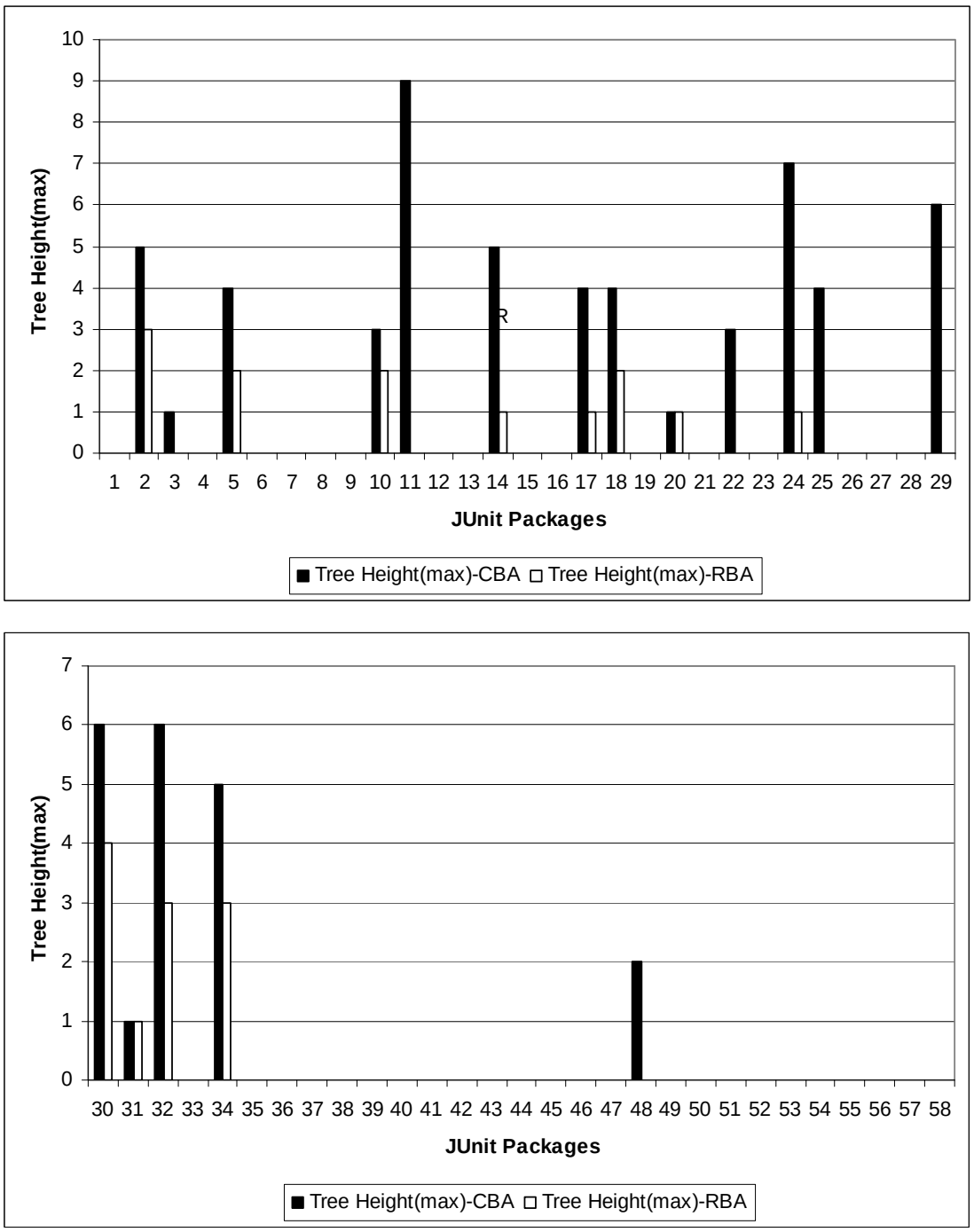

Σχήμα Α.21 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Single και RBA.



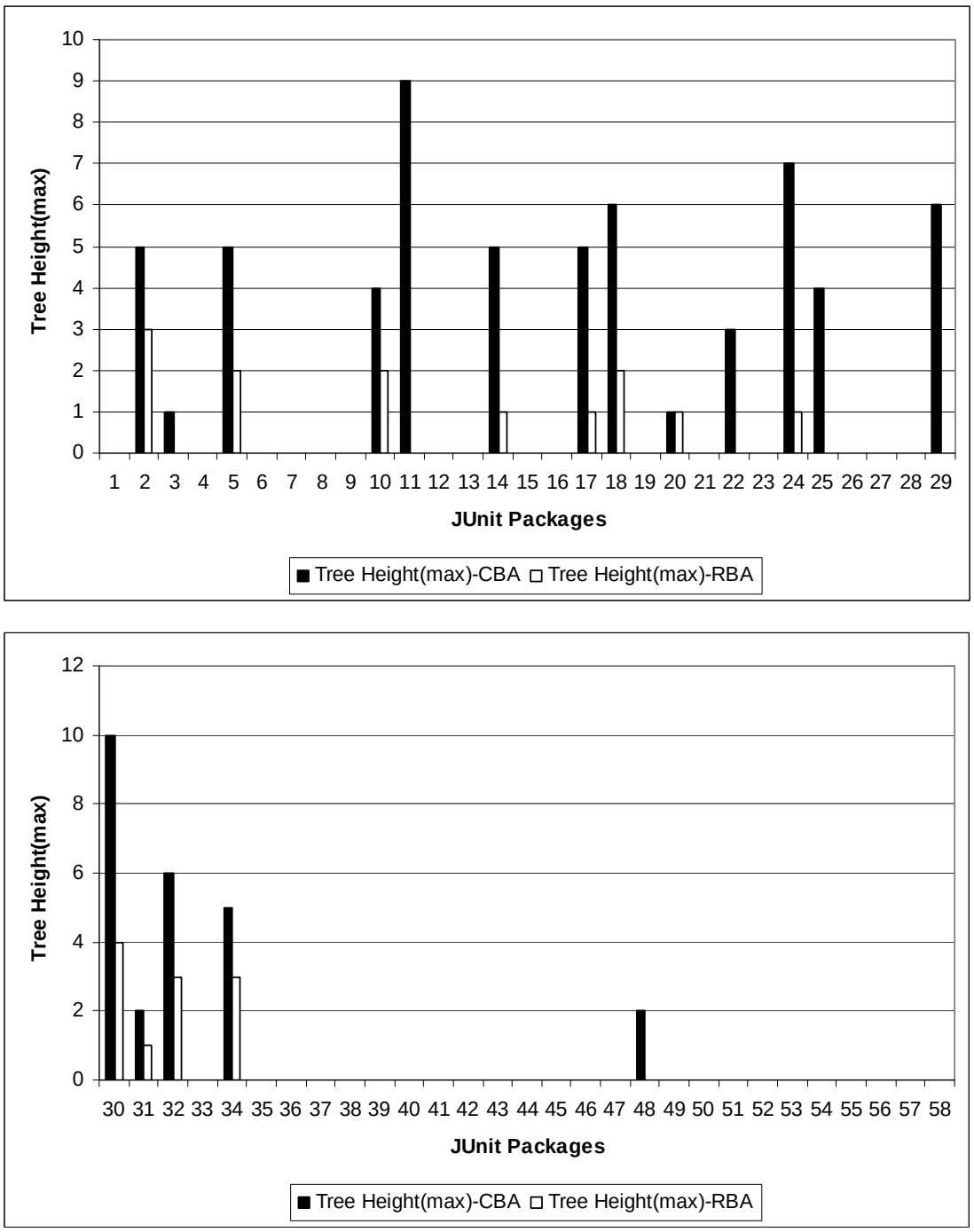

Σχήμα Α.22 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Average και RBA.



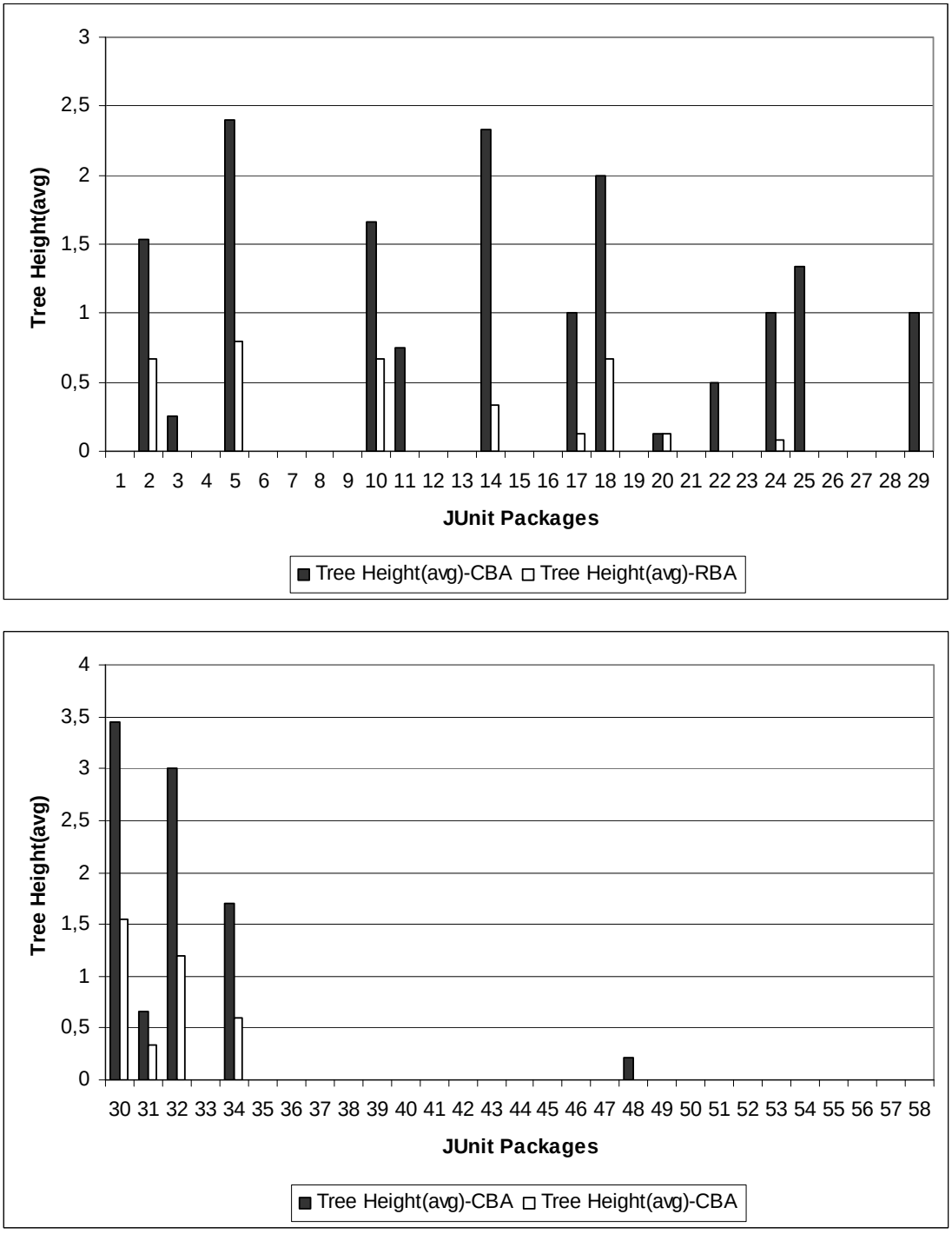

Σχήμα Α.23 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Complete και RBA.



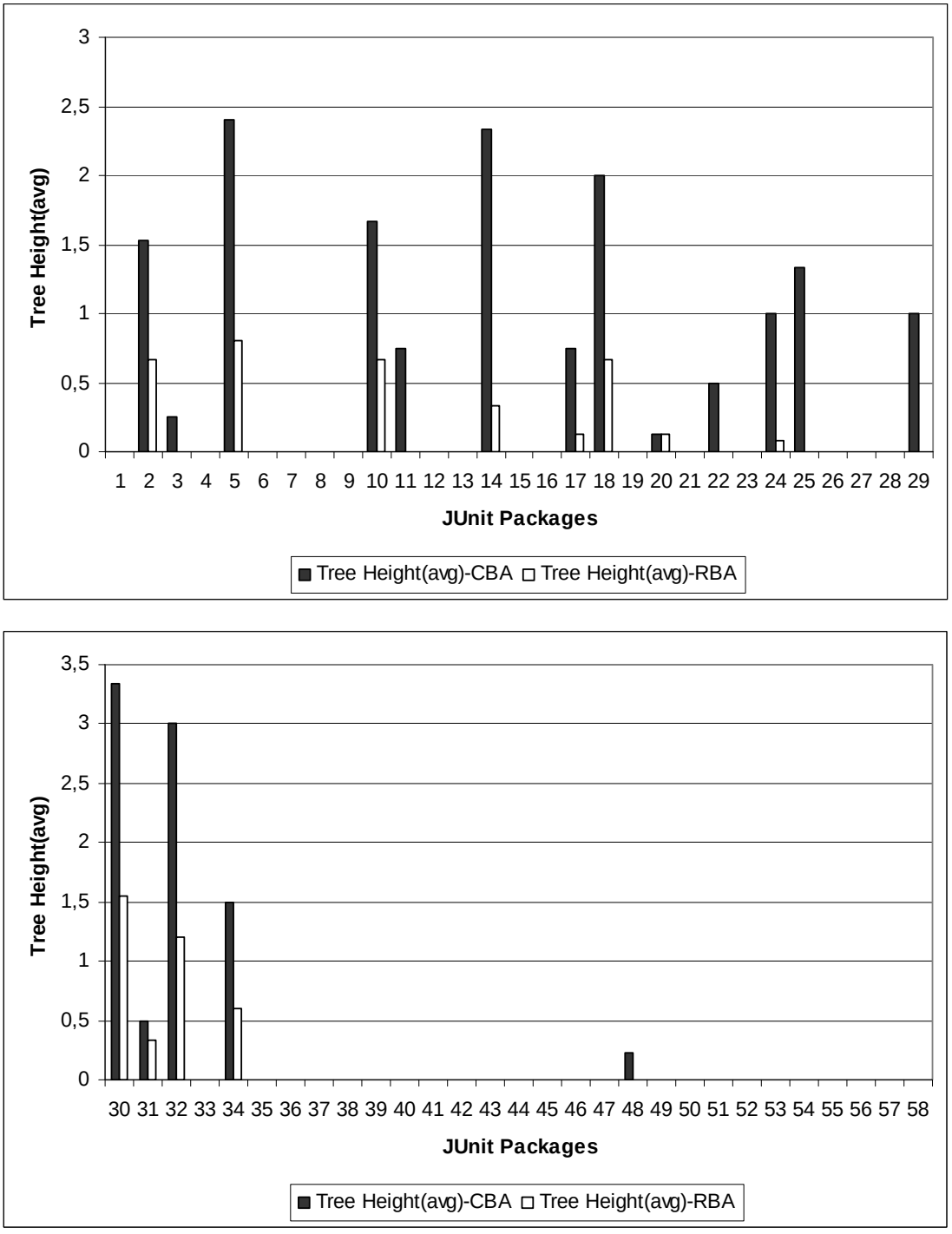

Σχήμα Α.24 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Median και RBA.



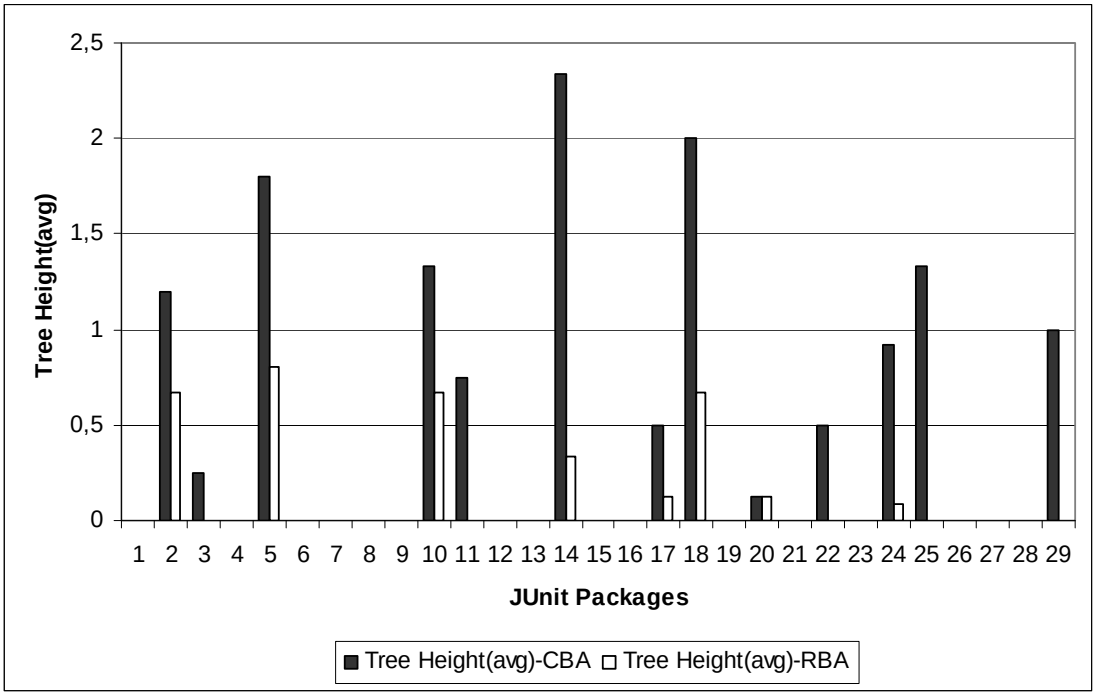

Σχήμα Α.25 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Adaptive και RBA.

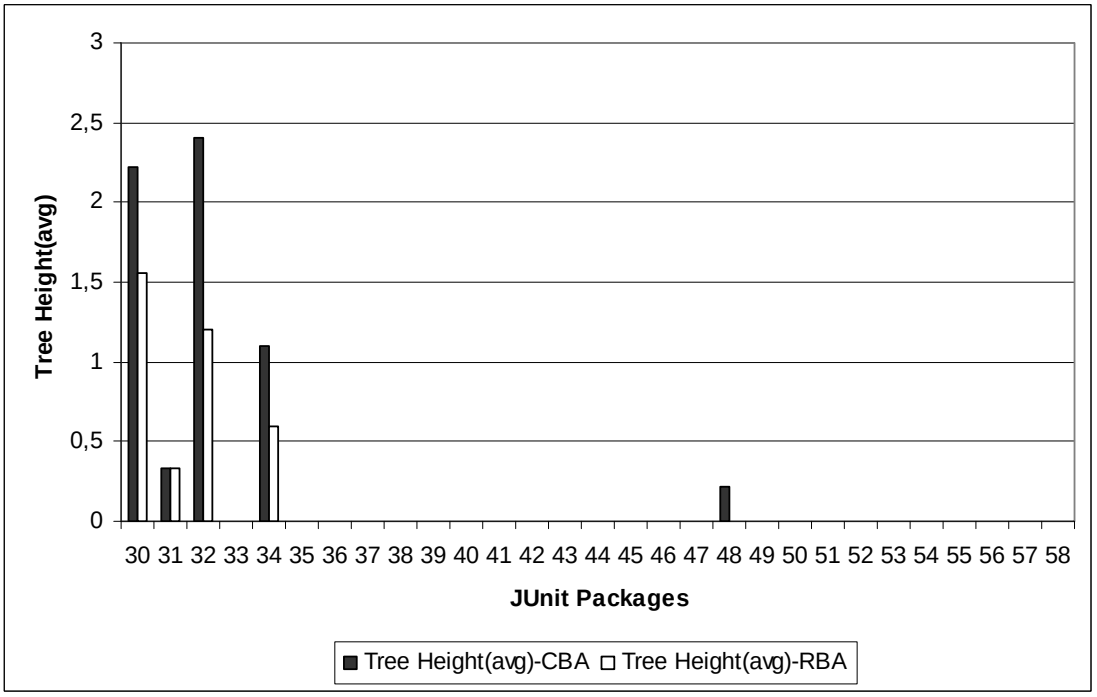



Σχήμα Α.26 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Single και RBA.

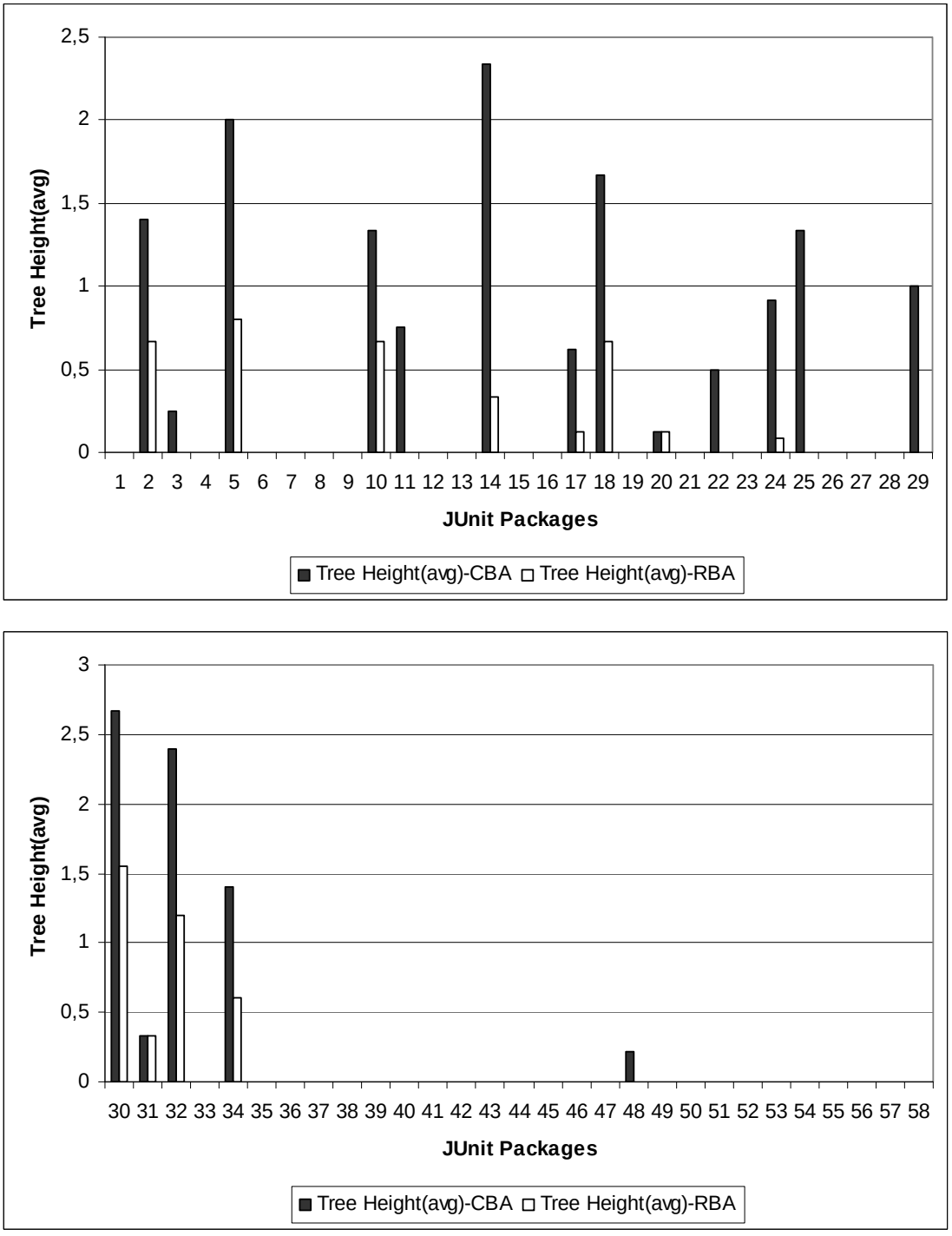



Σχήμα Α.27 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Average και RBA.

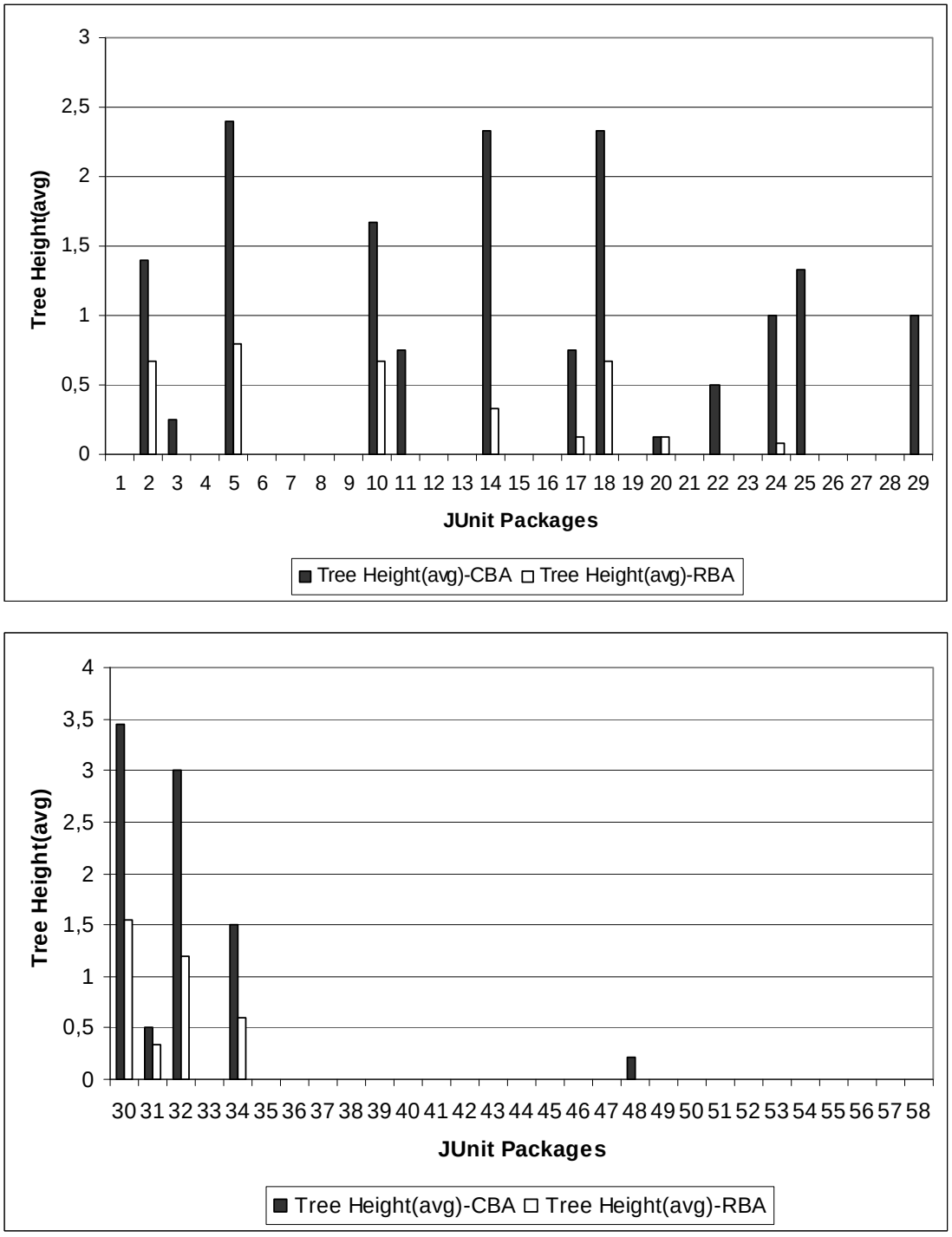



Σχήμα Α.28 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Complete και RBA.




Σχήμα Α.29 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Median και RBA.




Σχήμα Α.30 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Adaptive και RBA.

# ΠΑΡΑΡΤΗΜΑ Β

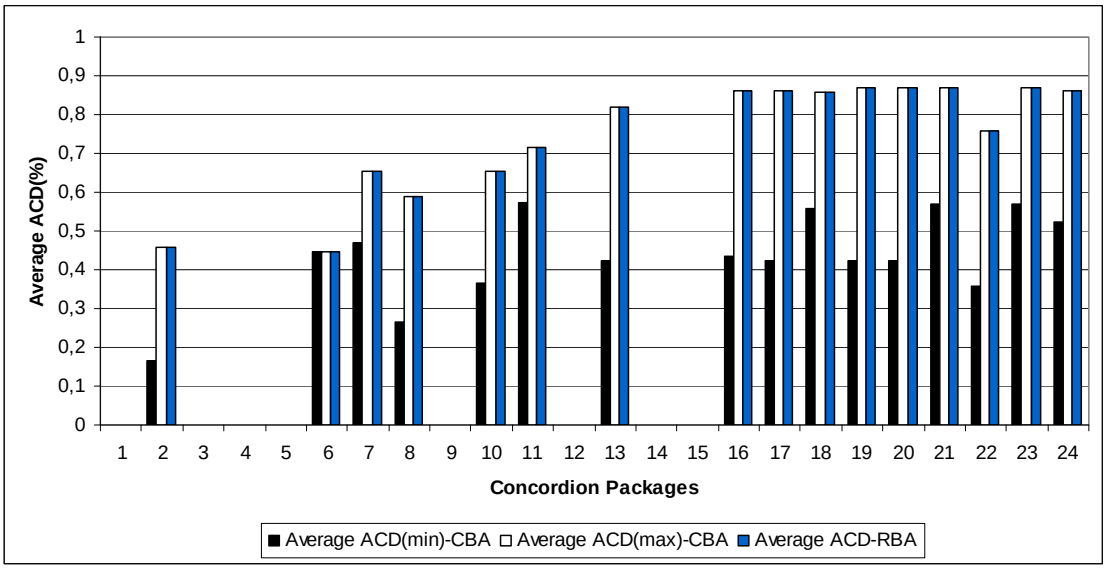
Οι τιμές που παρουσιάζονται στα σχήματα του Παραρτήματος Β, αποτελούν τη μέση τιμή για κάθε πακέτο του Concordion. Ο συγκεκριμένος τρόπος παρουσίασης επιλέχθηκε για πρακτικούς και εποπτικούς λόγους, καθώς το μεγάλο πλήθος των κλάσεων του κάθε περιβάλλοντος δεν θα διευκόλυνε την εξαγωγή των συμπερασμάτων. Εξαίρεση, αποτελεί το κριτήριο σύγκρισης των συνολικών διεπαφών (Total Interfaces) όπου παρουσιάζεται το συνολικό πλήθος διεπαφών που δημιουργείται, με κάθε αλγόριθμο. Τα αποτελέσματα παρουσιάζονται, συγκρίνοντας κάθε φορά, τις επιδόσεις των αλγορίθμων της CBA και της RBA, για κάθε ένα από τα έξι (6) κριτήρια σύγκρισης που αναφέρθηκαν στο Κεφάλαιο 5 και κάθε ένα από τα πέντε (5) κριτήρια απόστασης που έχουν παρουσιαστεί στο Κεφάλαιο 3. Για εποπτικούς λόγους, έχουμε αντιστοιχίσει το κάθε πακέτο του Concordion σε ένα μοναδικό αριθμό (Πίνακας Β.1).

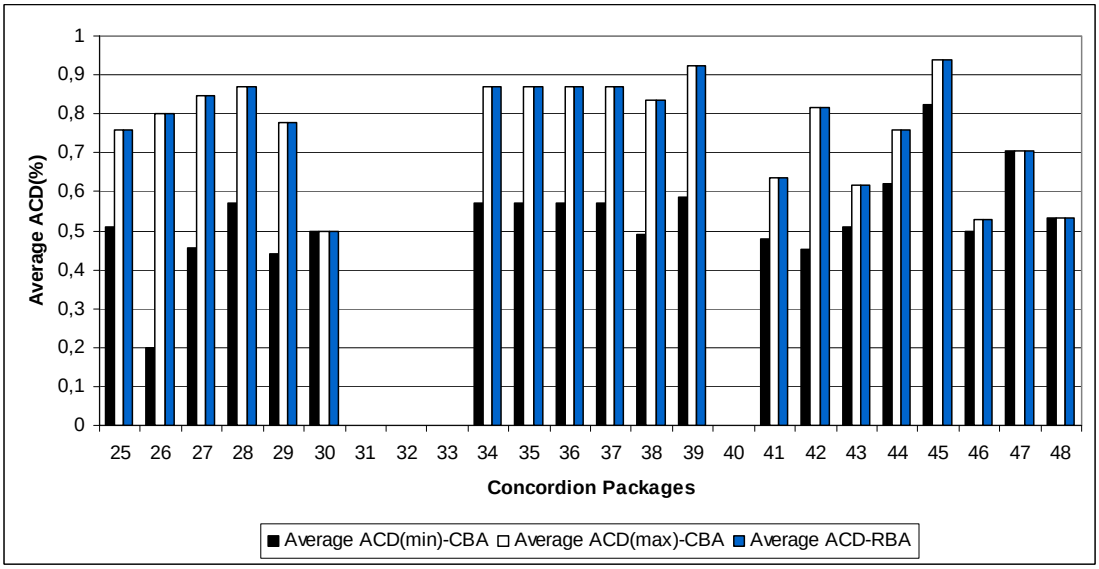
Πίνακας Β.1 Ποσοτική Περιγραφή του Concordion.

| Πακέτο | Αριθμός αντιστοίχισης | Πλήθος κλάσεων |
|---|---|---|
| org.concordion | 1 | 1 |
| org.concordion.api | 2 | 25 |
| org.concordion.api.extension | 3 | 5 |
| org.concordion.api.listener | 4 | 23 |
| org.concordion.integration.junit3 | 5 | 1 |
| org.concordion.integration.junit4 | 6 | 1 |
| org.concordion.internal | 7 | 23 |
| org.concordion.internal.command | 8 | 13 |
| org.concordion.internal.extension | 9 | 2 |

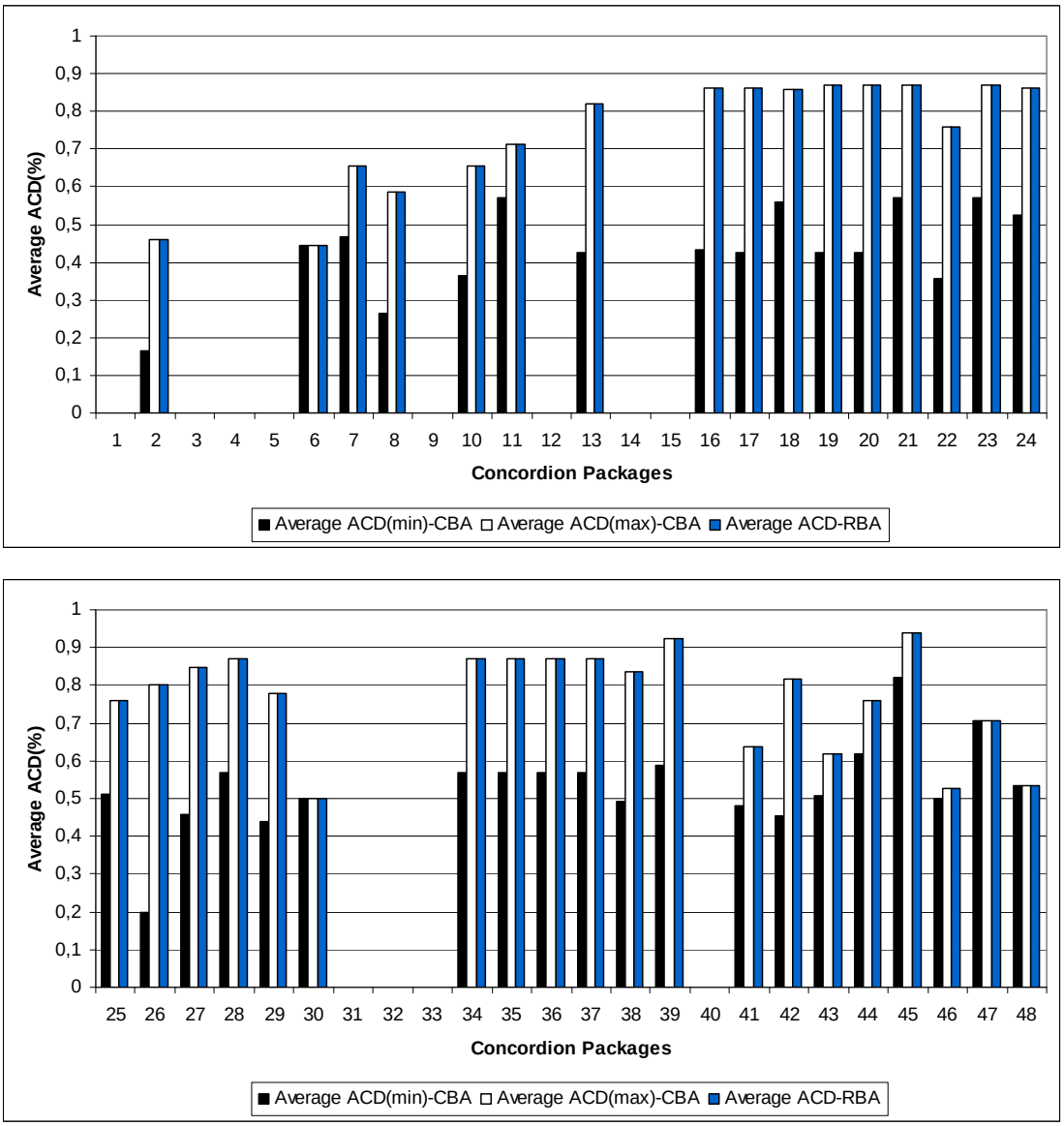
| | | |
|---|---|---|
| org.concordion.internal.listener | 10 | 13 |
| org.concordion.internal.runner | 11 | 1 |
| org.concordion.internal.util | 12 | 3 |
| spec.concordion | 13 | 2 |
| spec.concordion.annotation | 14 | 1 |
| spec.concordion.command | 15 | 1 |
| spec.concordion.command.assertEquals | 16 | 5 |
| spec.concordion.command.assertEquals.nonString | 17 | 3 |
| spec.concordion.command.assertEquals.whitespace | 18 | 3 |
| spec.concordion.command.assertFalse | 19 | 1 |
| spec.concordion.command.assertTrue | 20 | 1 |
| spec.concordion.command.echo | 21 | 3 |
| spec.concordion.command.execute | 22 | 3 |
| spec.concordion.command.results.contentType | 23 | 1 |
| spec.concordion.command.results.stylesheet | 24 | 2 |
| spec.concordion.command.run | 25 | 4 |
| spec.concordion.command.set | 26 | 1 |
| spec.concordion.command.verifyRows | 27 | 3 |
| spec.concordion.command.verifyRows.results | 28 | 2 |
| spec.concordion.extension | 29 | 7 |
| spec.concordion.extension.listener | 30 | 3 |
| spec.concordion.integration | 31 | 1 |
| spec.concordion.integration.junit3 | 32 | 1 |
| spec.concordion.results | 33 | 1 |
| spec.concordion.results.assertEquals.failure | 34 | 4 |
| spec.concordion.results.assertEquals.success | 35 | 4 |
| spec.concordion.results.assertTrue.failure | 36 | 1 |
| spec.concordion.results.assertTrue.success | 37 | 1 |
| spec.concordion.results.breadcrumbs | 38 | 4 |
| spec.concordion.results.exception | 39 | 1 |

| | | |
|---|---|---|
| spec.examples | 40 | 4 |
| test.concordion | 41 | 9 |
| test.concordion.api | 42 | 2 |
| test.concordion.compiler | 43 | 11 |
| test.concordion.extension | 44 | 11 |
| test.concordion.extension.fake | 45 | 6 |
| test.concordion.internal | 46 | 7 |
| test.concordion.internal.listener | 47 | 4 |
| test.concordion.internal.runner | 48 | 2 |

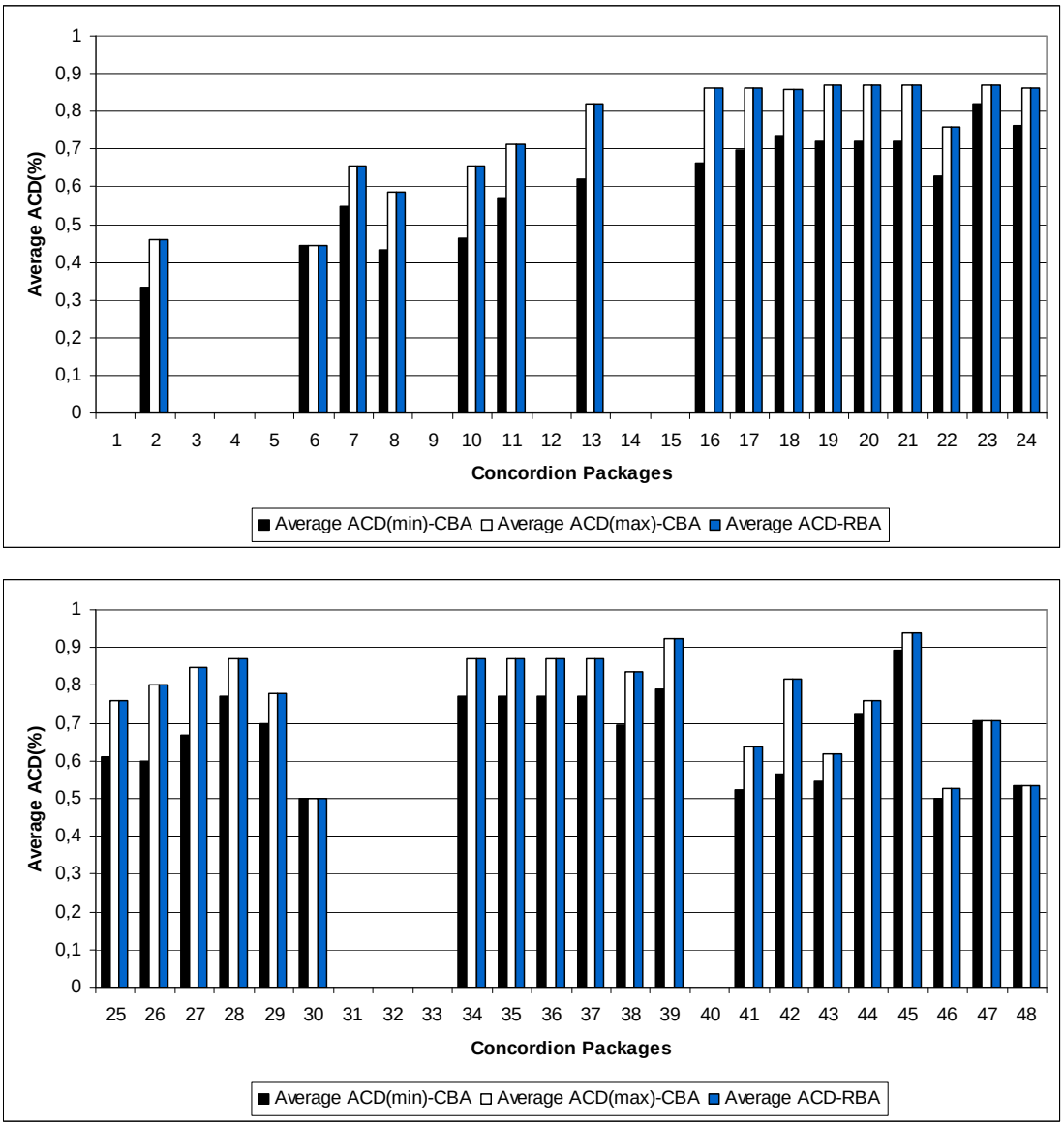


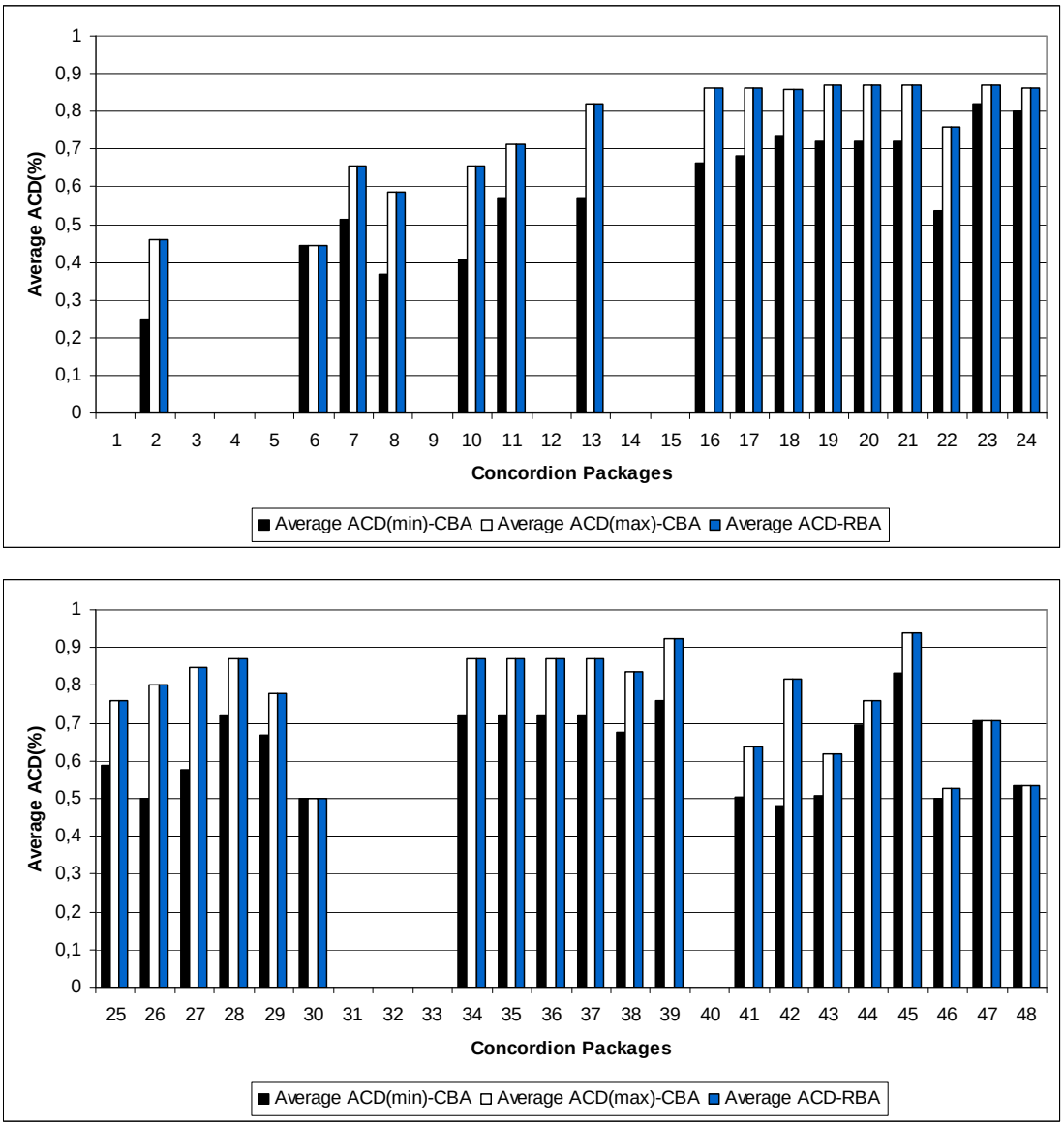

Σχήμα Β.1 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Single και RBA.



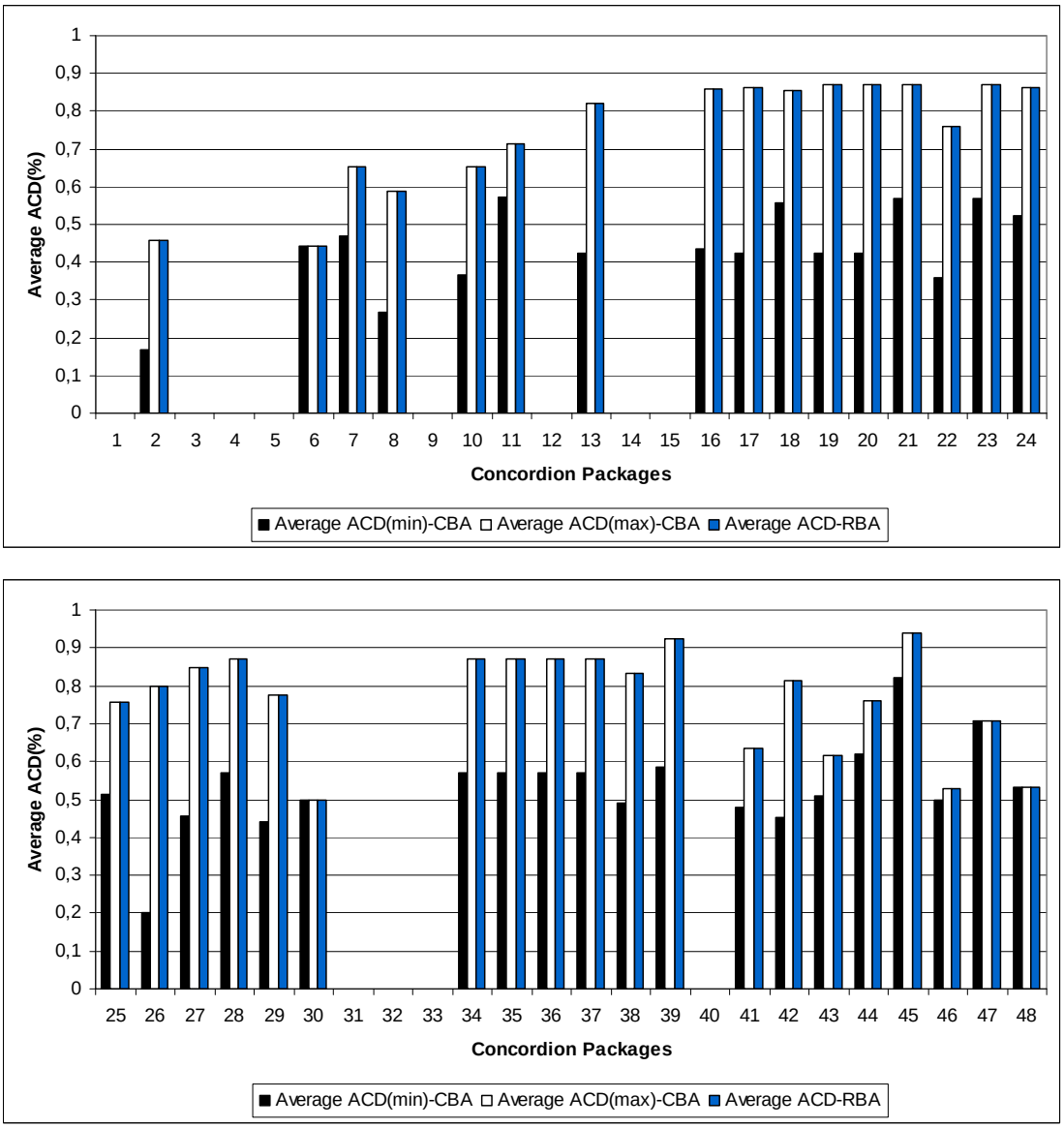

Σχήμα Β.2 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Average και RBA.



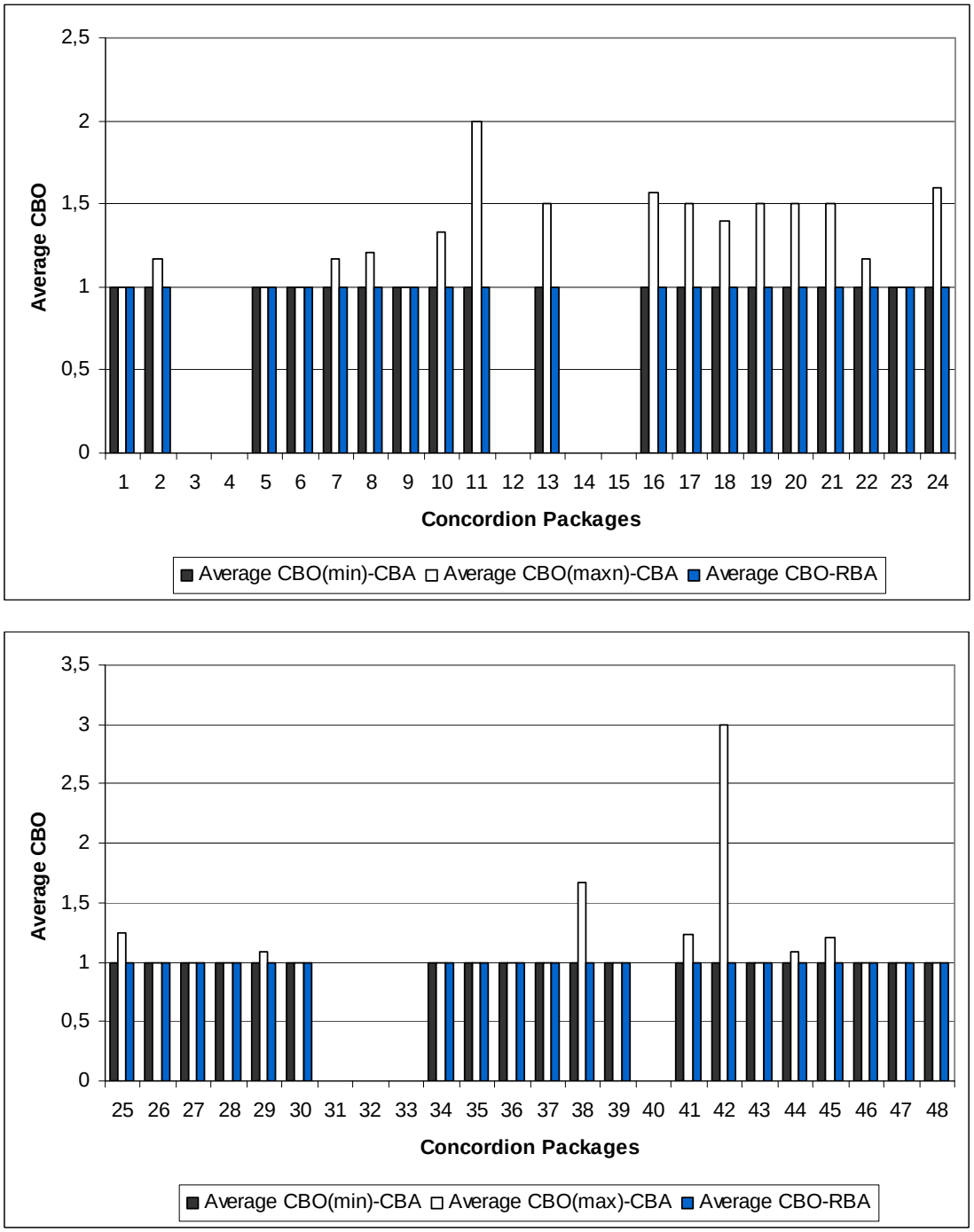

Σχήμα Β.3 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Complete και RBA.



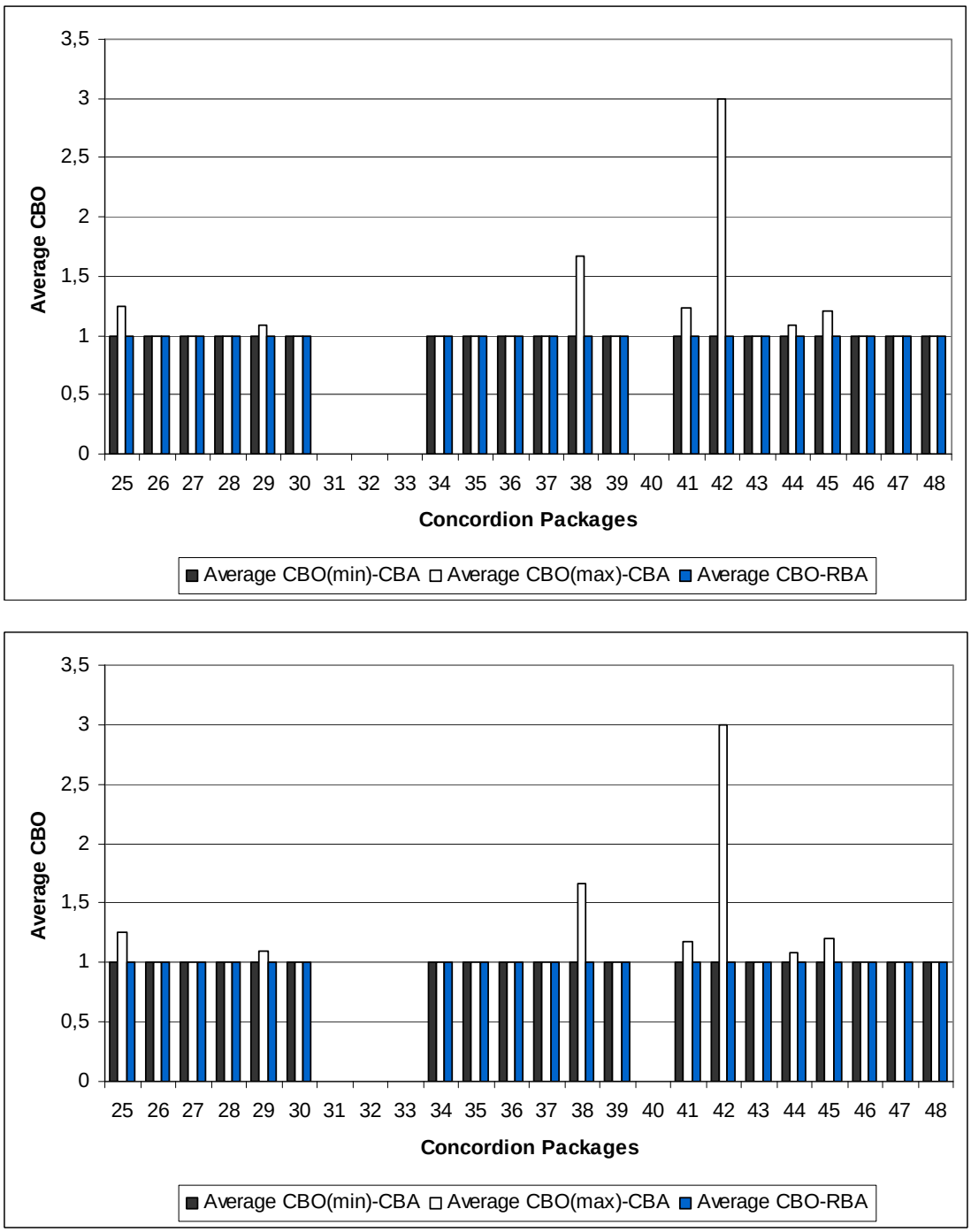

Σχήμα Β.4 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Median και RBA.




Σχήμα Β.5 Σύγκριση Μέσης Τιμής ACD μεταξύ CBA Adaptive και RBA.



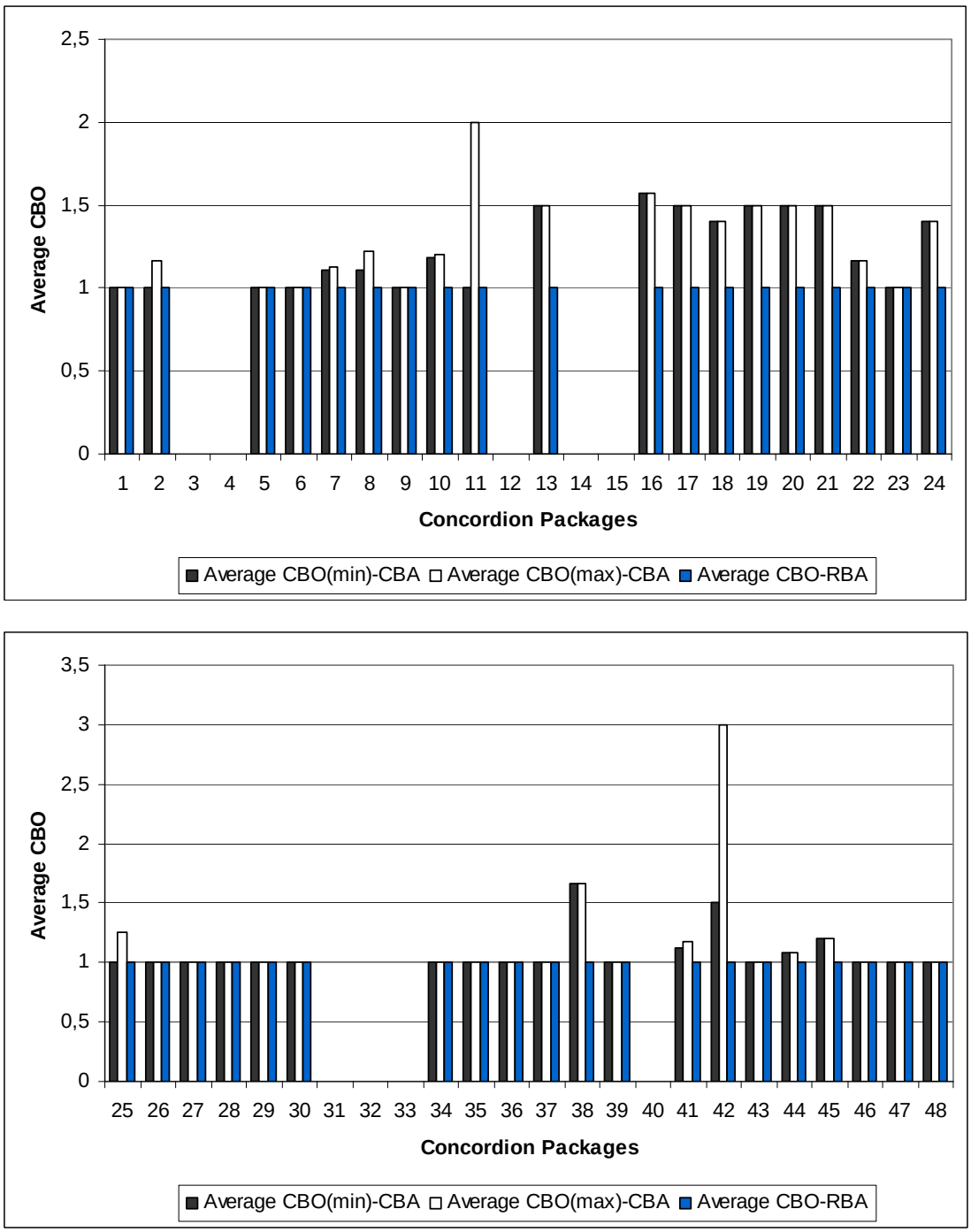

Σχήμα Β.6 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Single και RBA.



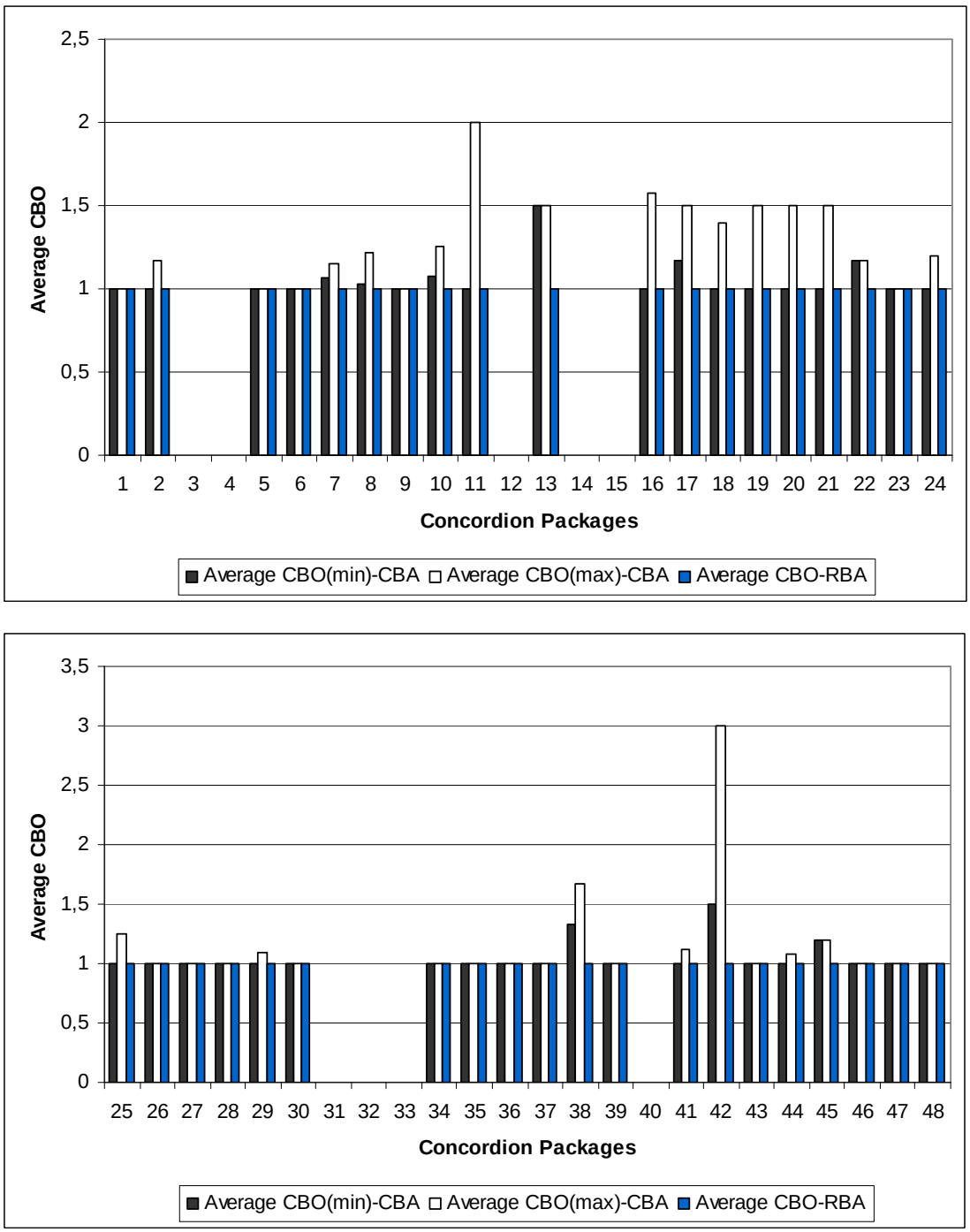




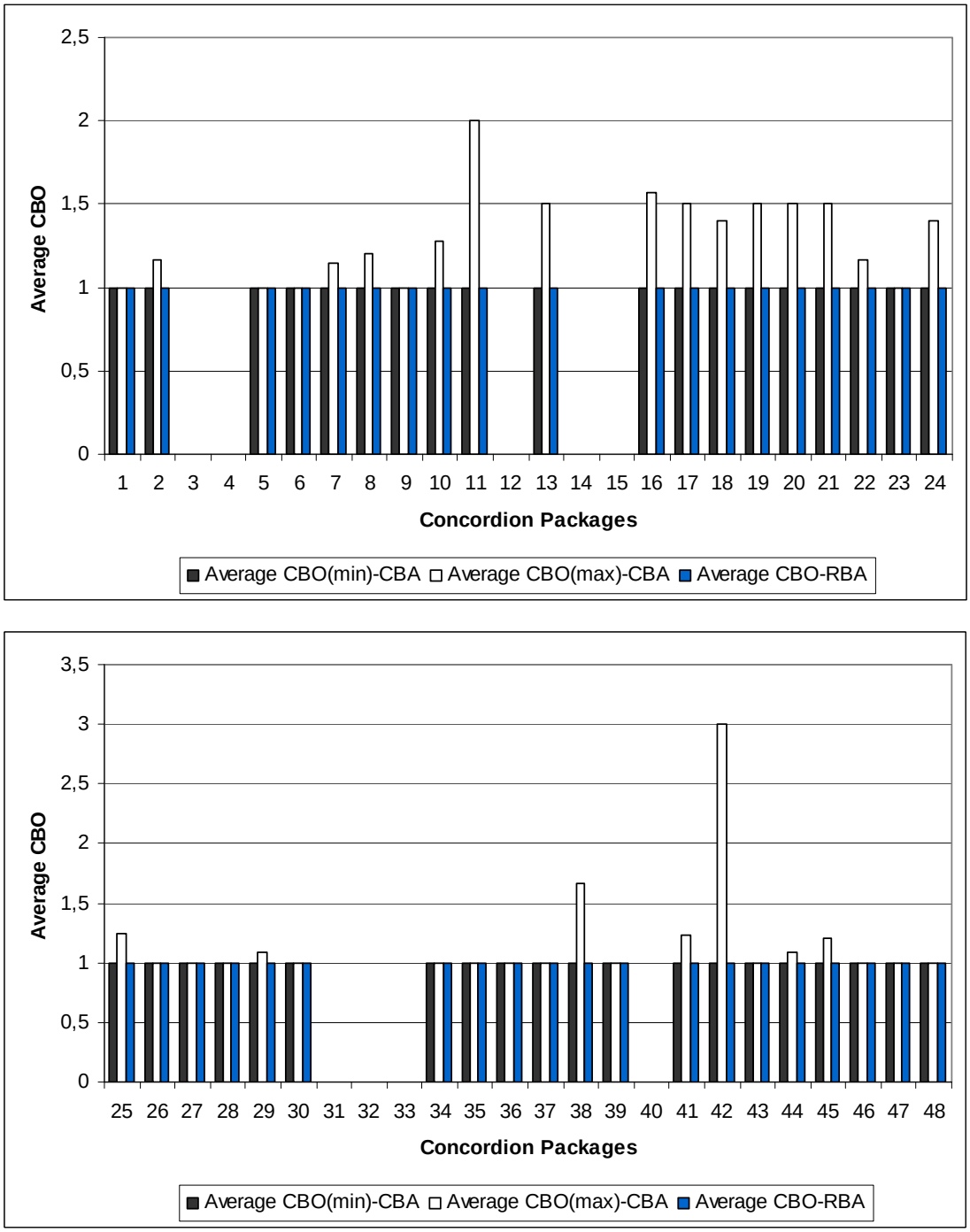
Σχήμα Β.7 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Average και RBA.



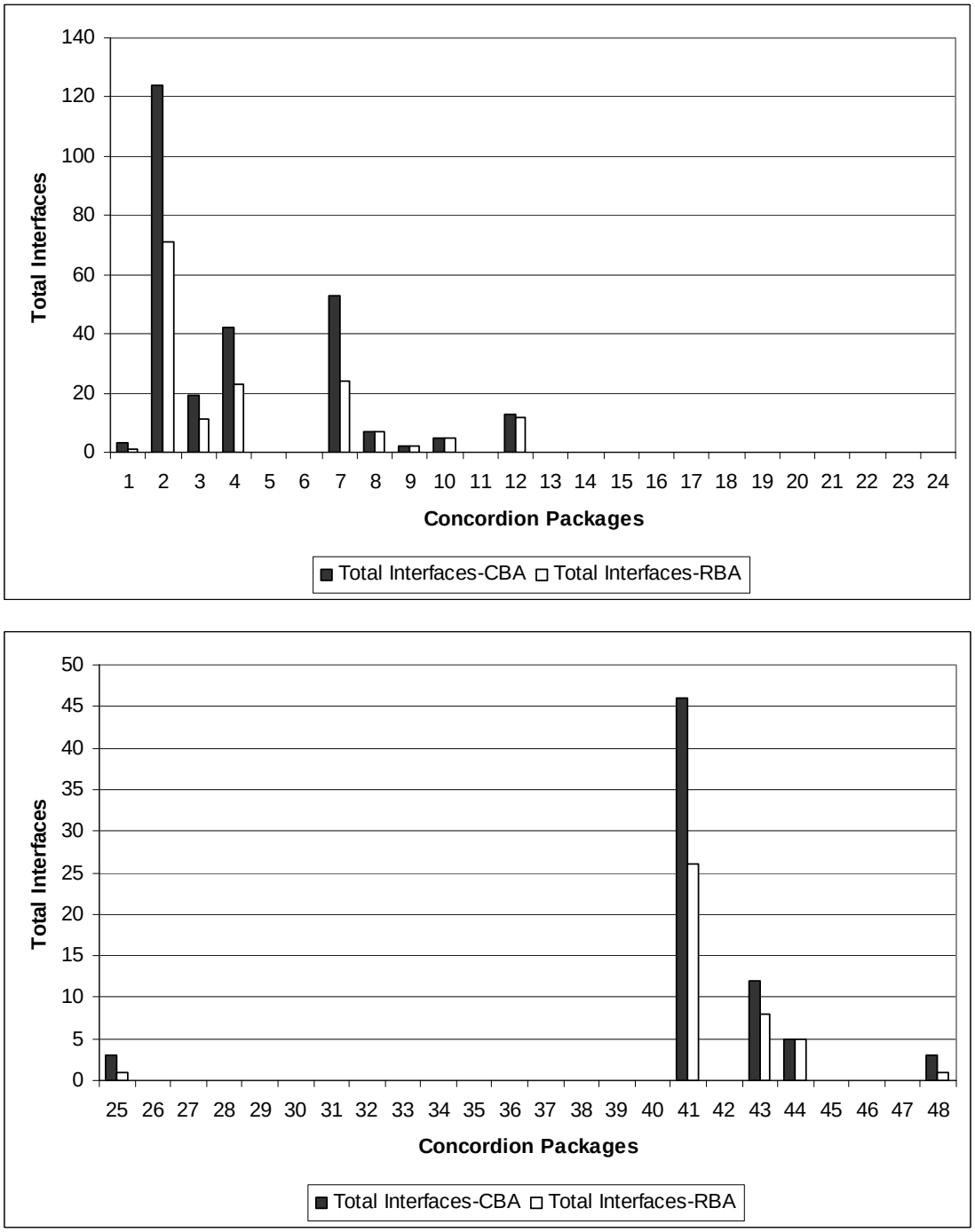

Σχήμα Β.8 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Complete και RBA.



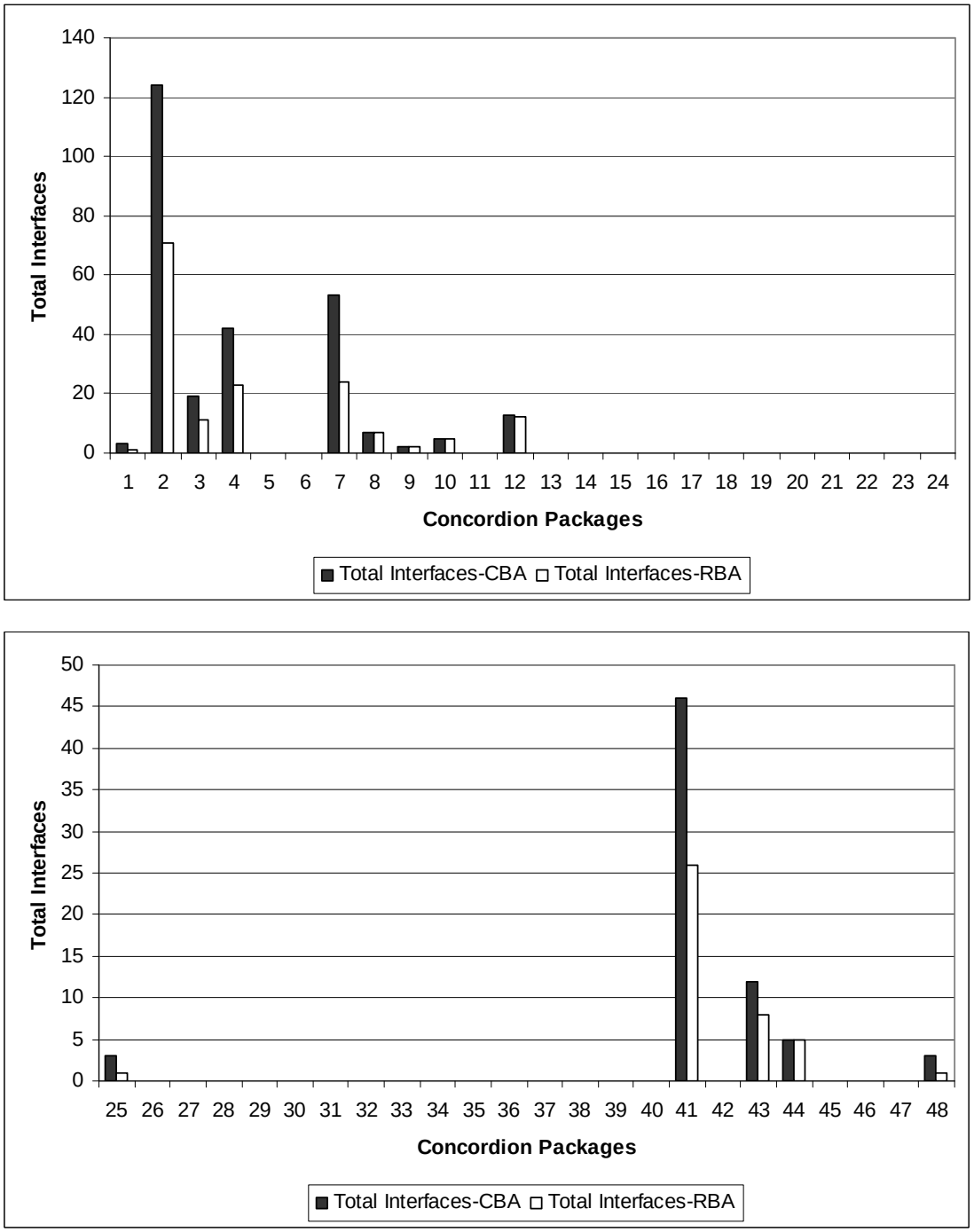

Σχήμα Β.9 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Median και RBA.



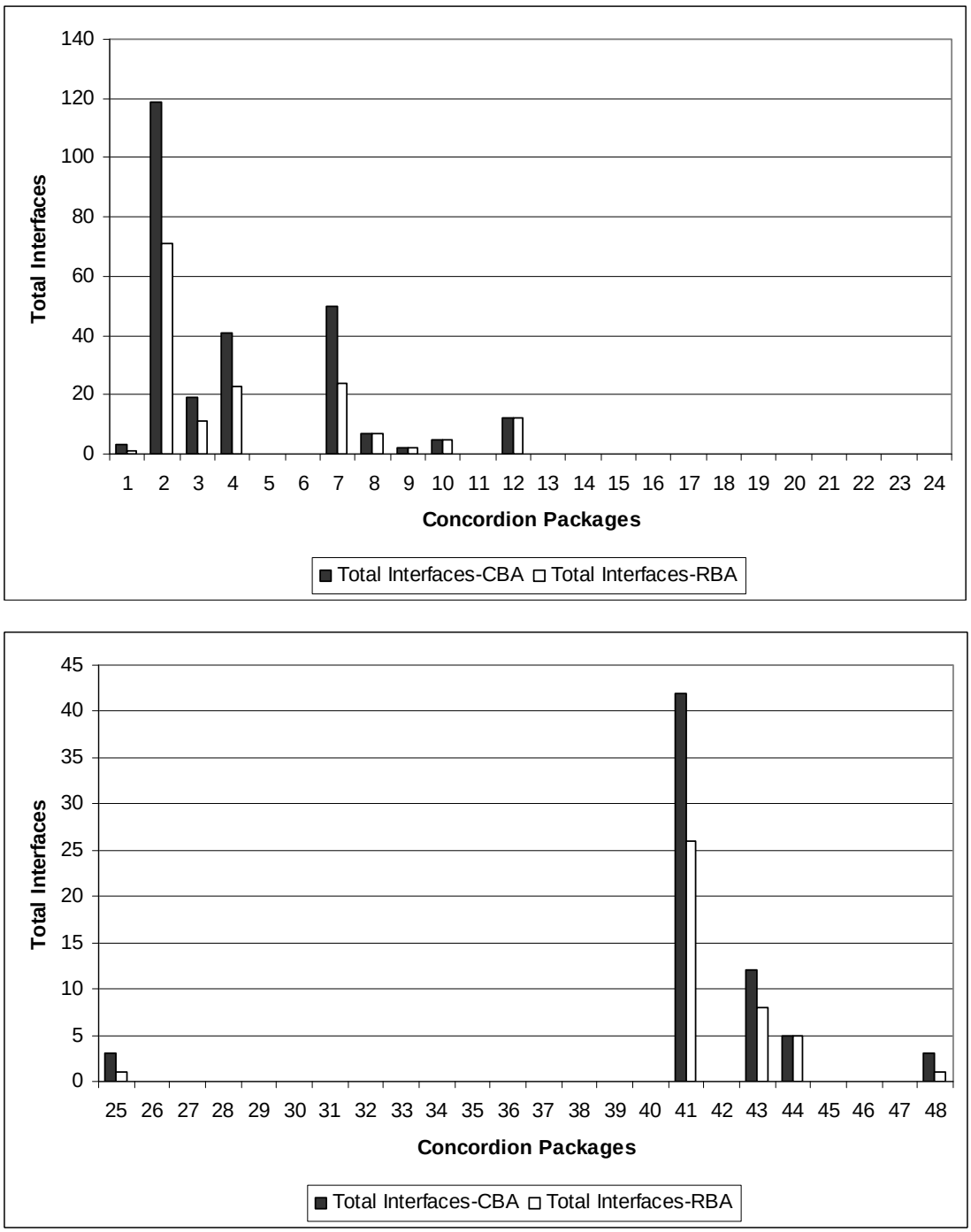

Σχήμα Β.10 Σύγκριση Μέσης Τιμής CBO μεταξύ CBA Adaptive και RBA.



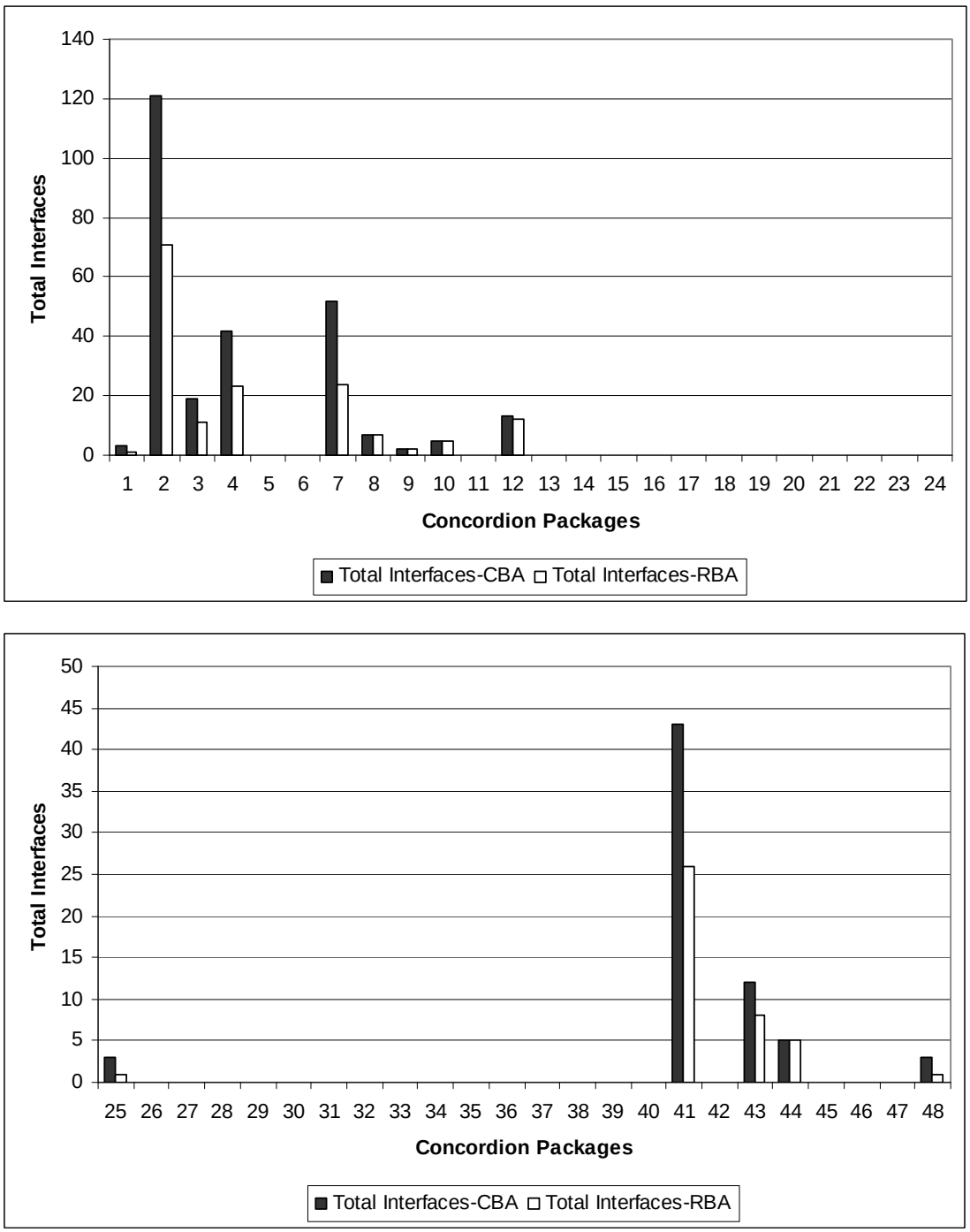

Σχήμα Β.11 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Single και RBA.



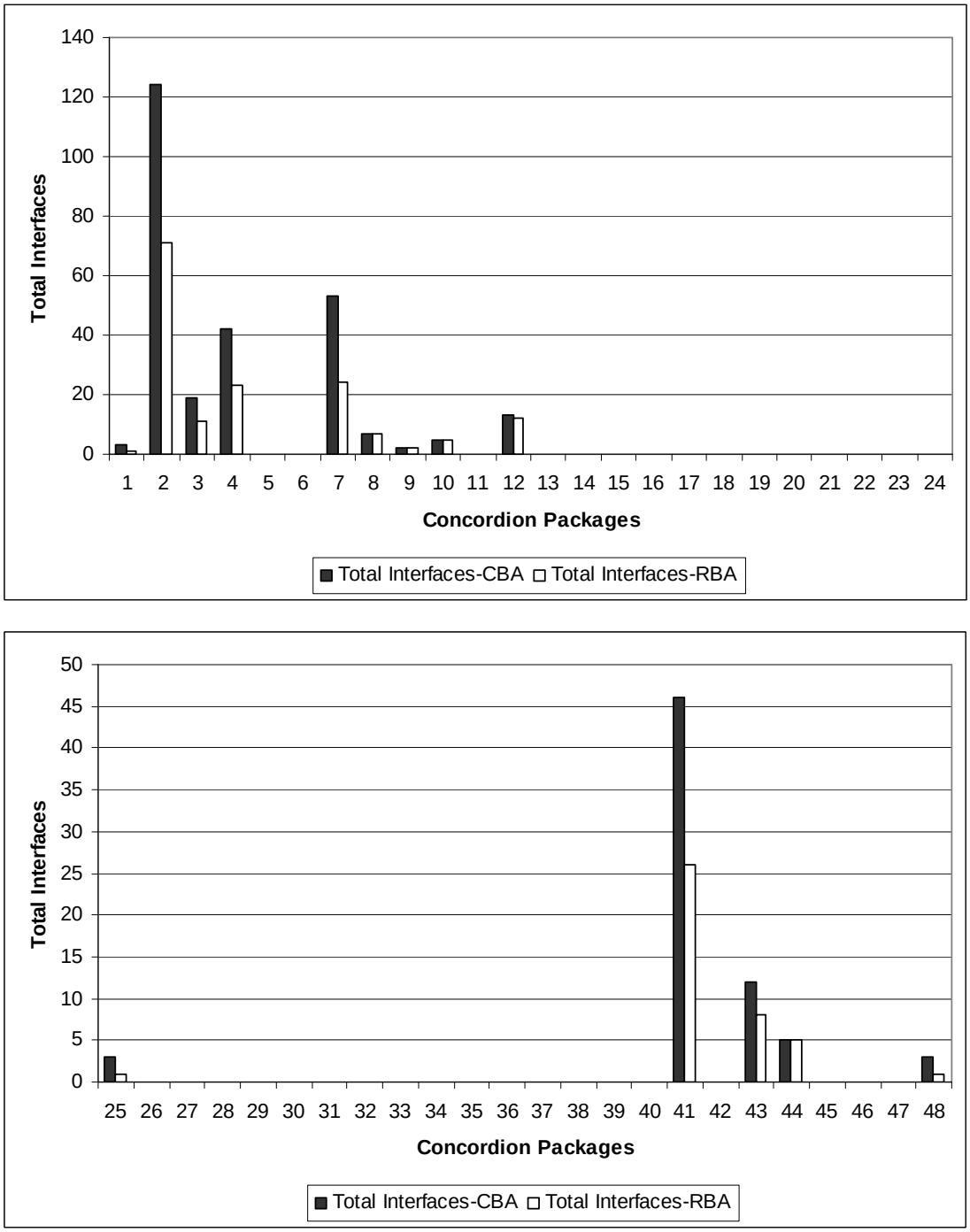

Σχήμα Β.12 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Average και RBA.



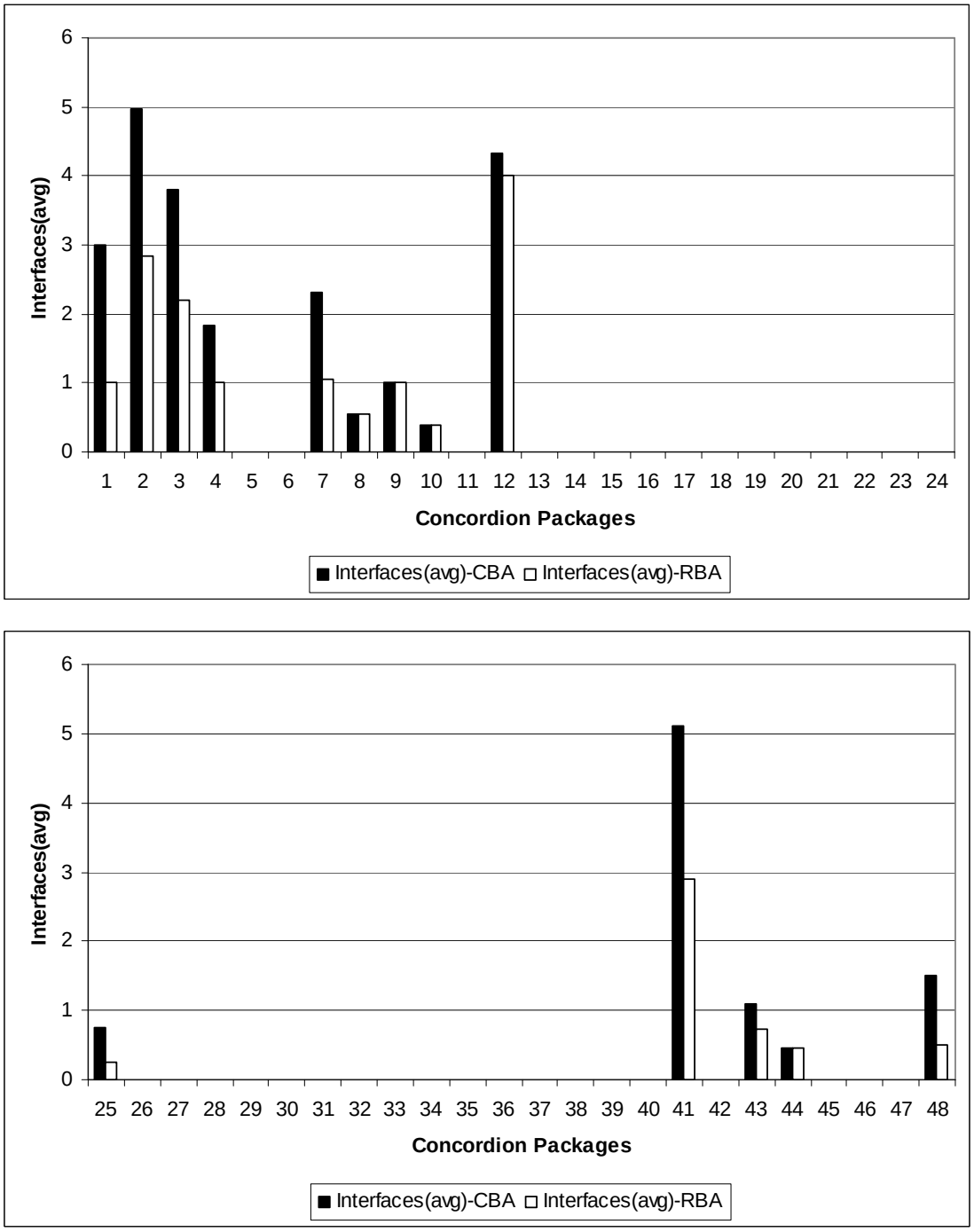

Σχήμα Β.13 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Complete και RBA.



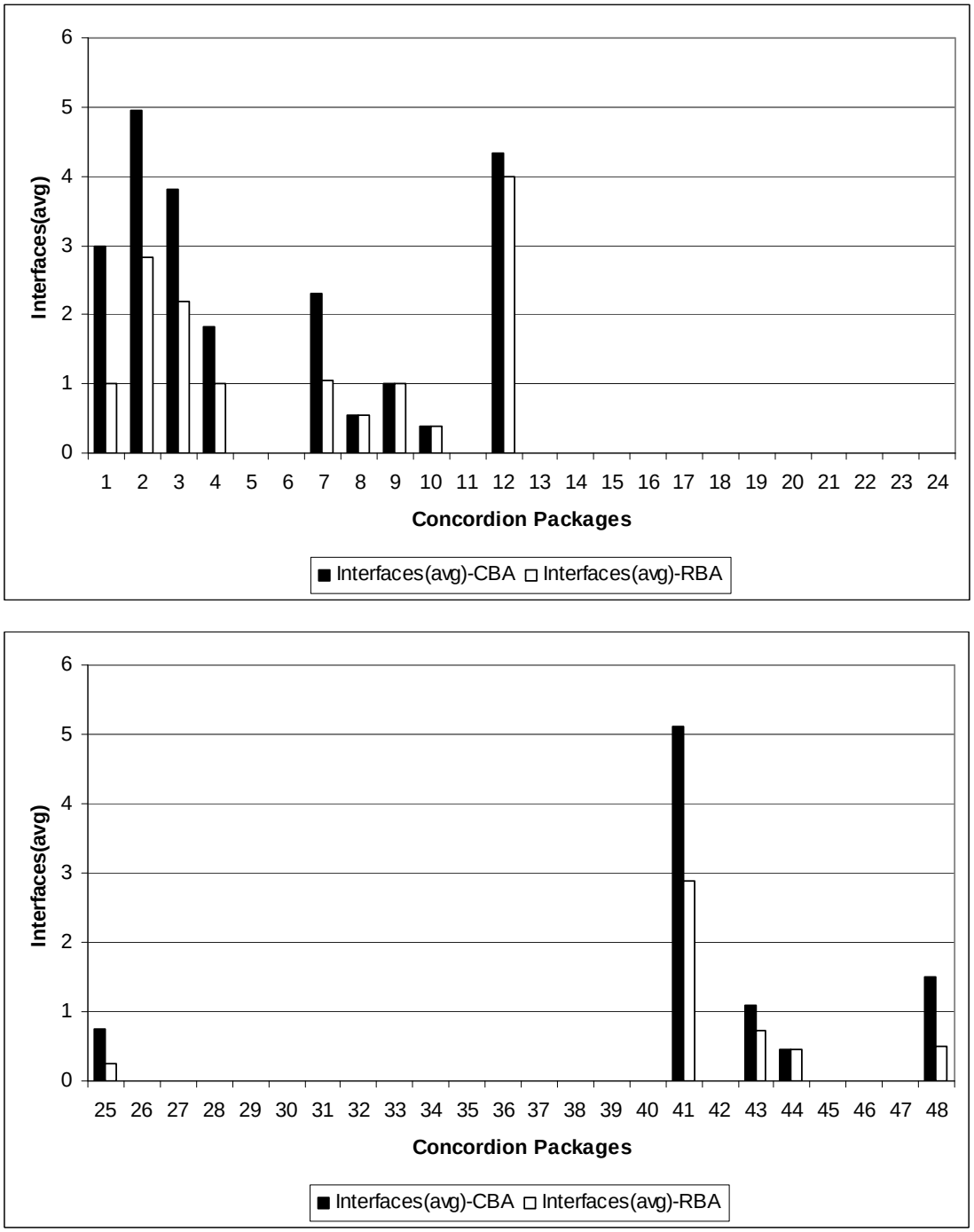

Σχήμα Β.14 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Median και RBA.



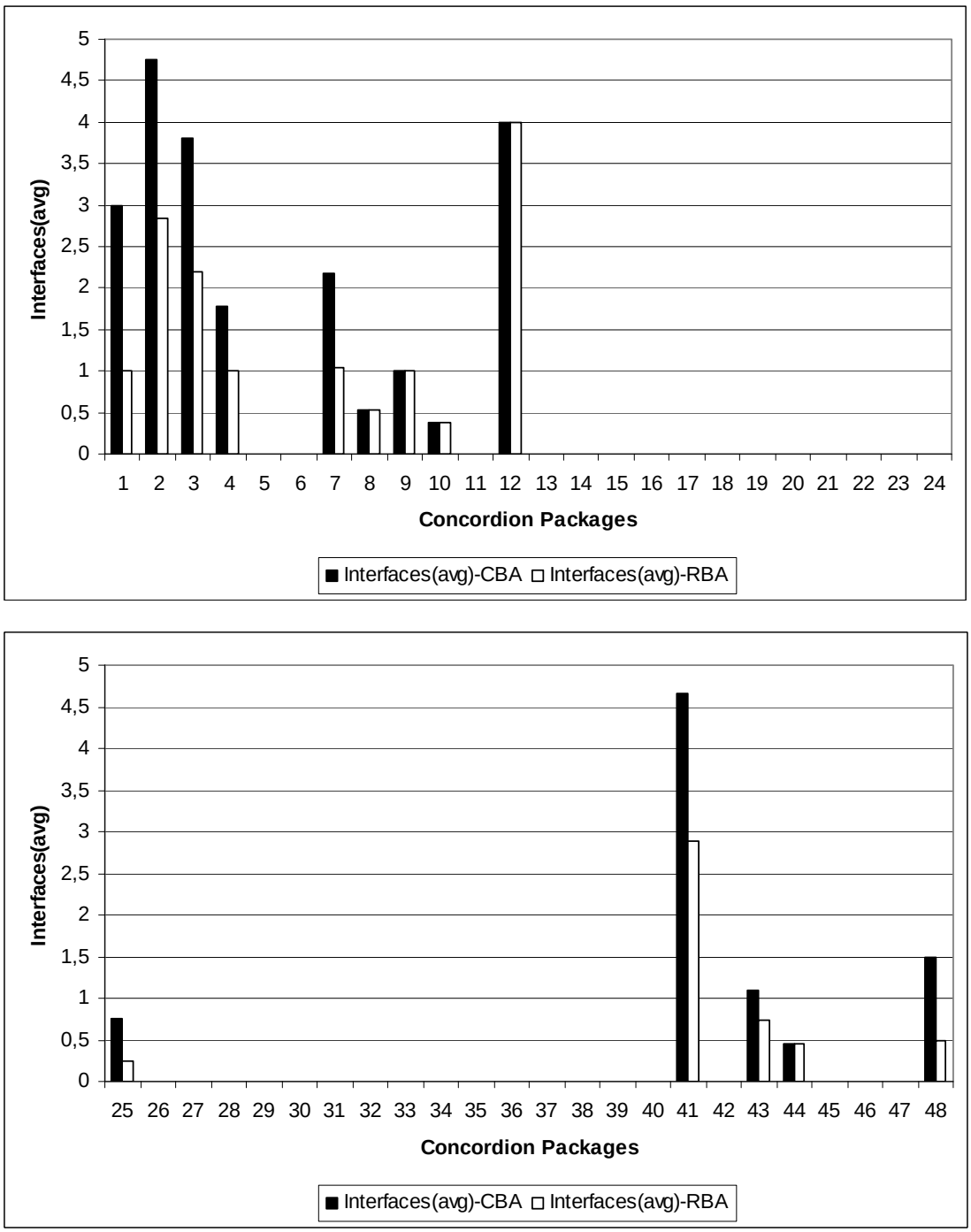

Σχήμα Β.15 Σύγκριση Συνολικού Πλήθους Διεπαφών μεταξύ CBA Adaptive και RBA.



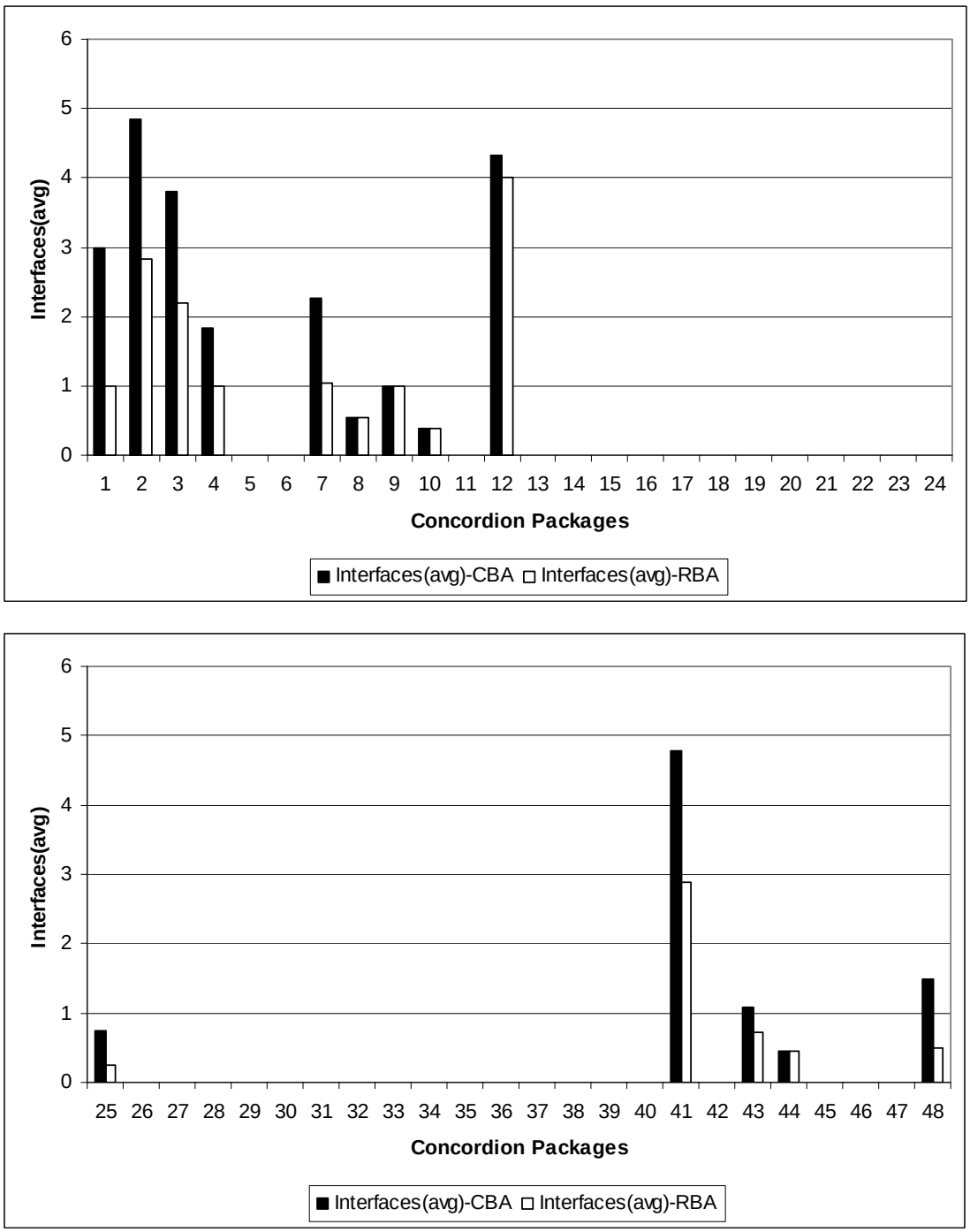

Σχήμα Β.16 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Single και RBA.



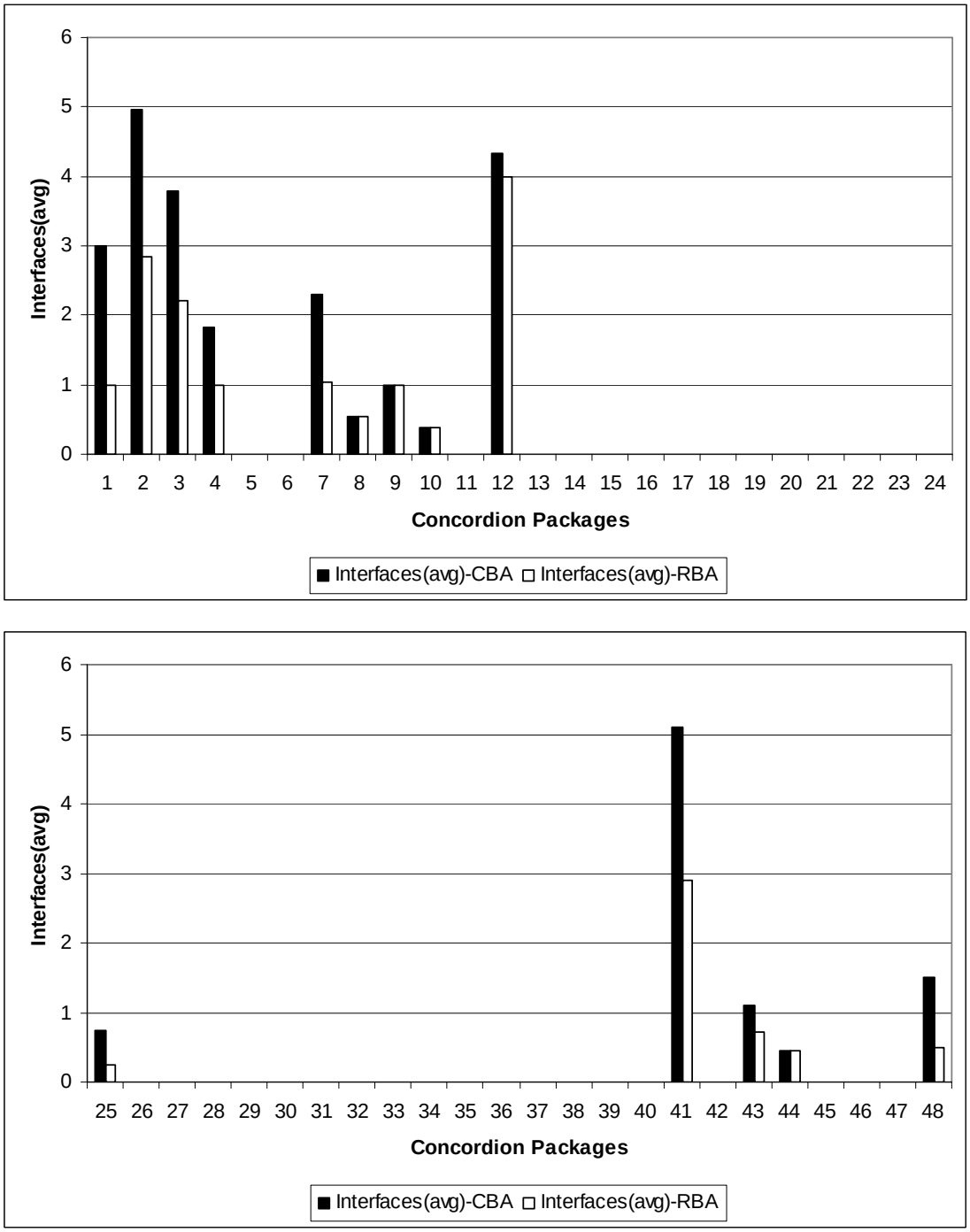

Σχήμα Β.17 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Average και RBA.



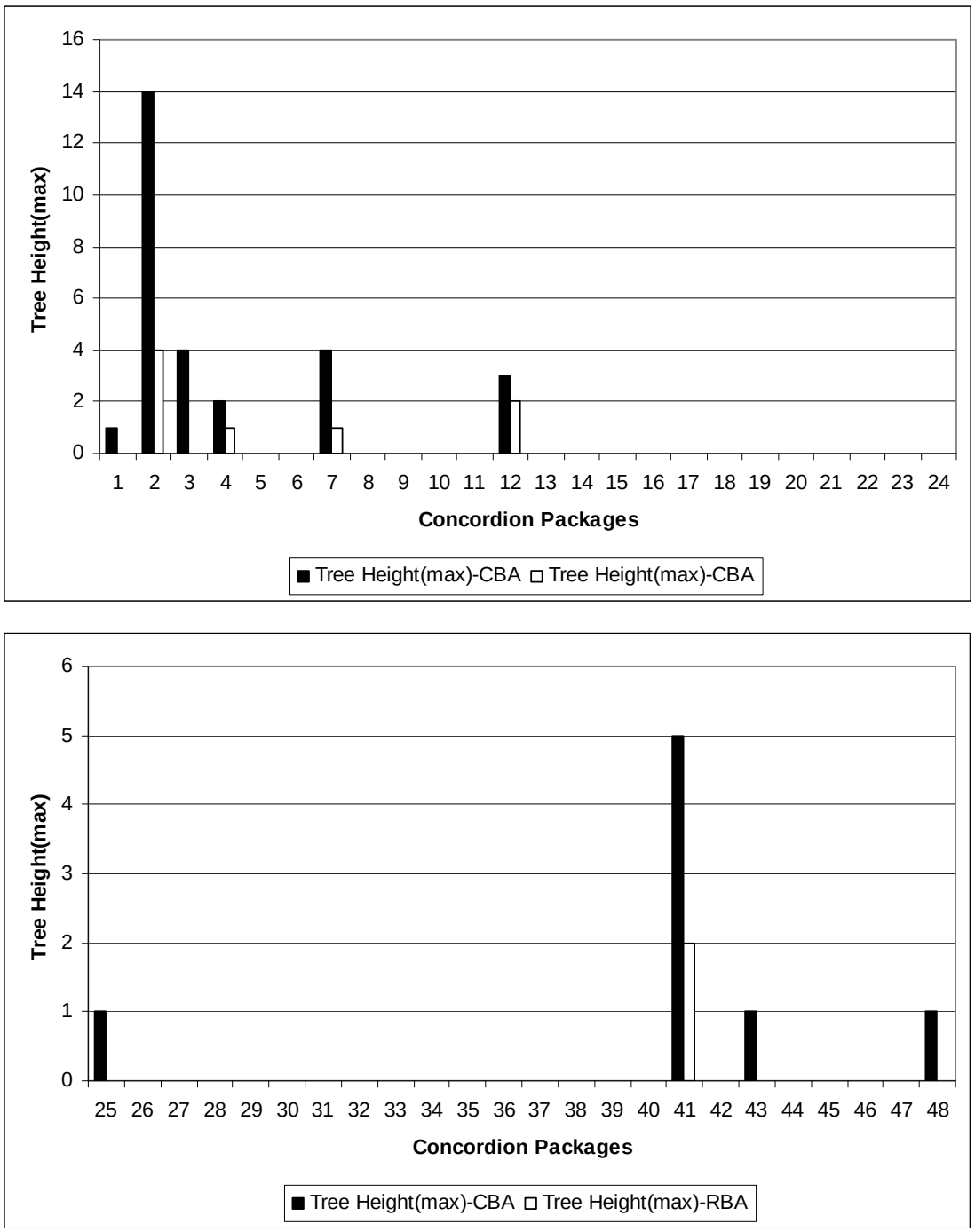

Σχήμα Β.18 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Complete και RBA.



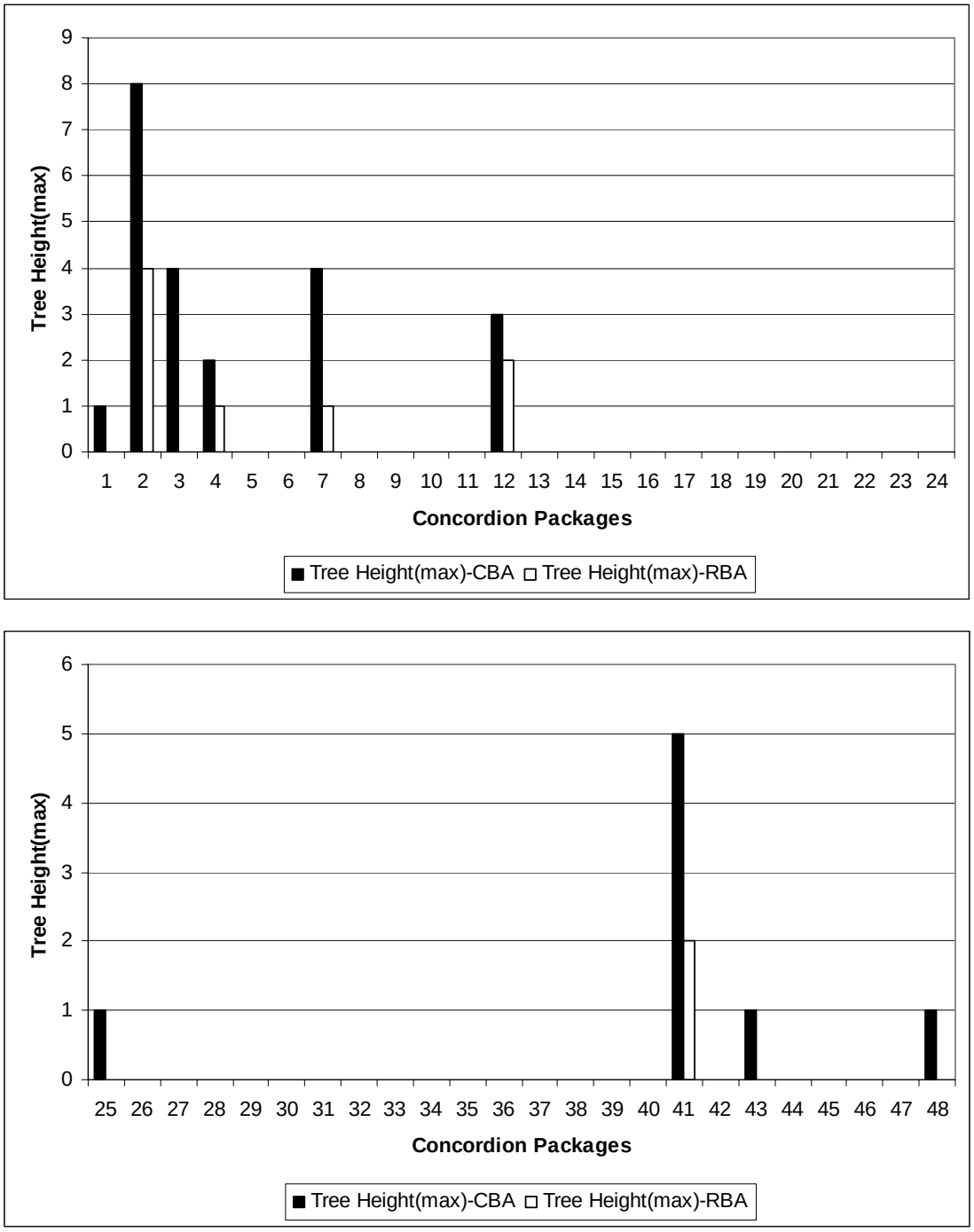

Σχήμα Β.19 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Median και RBA.



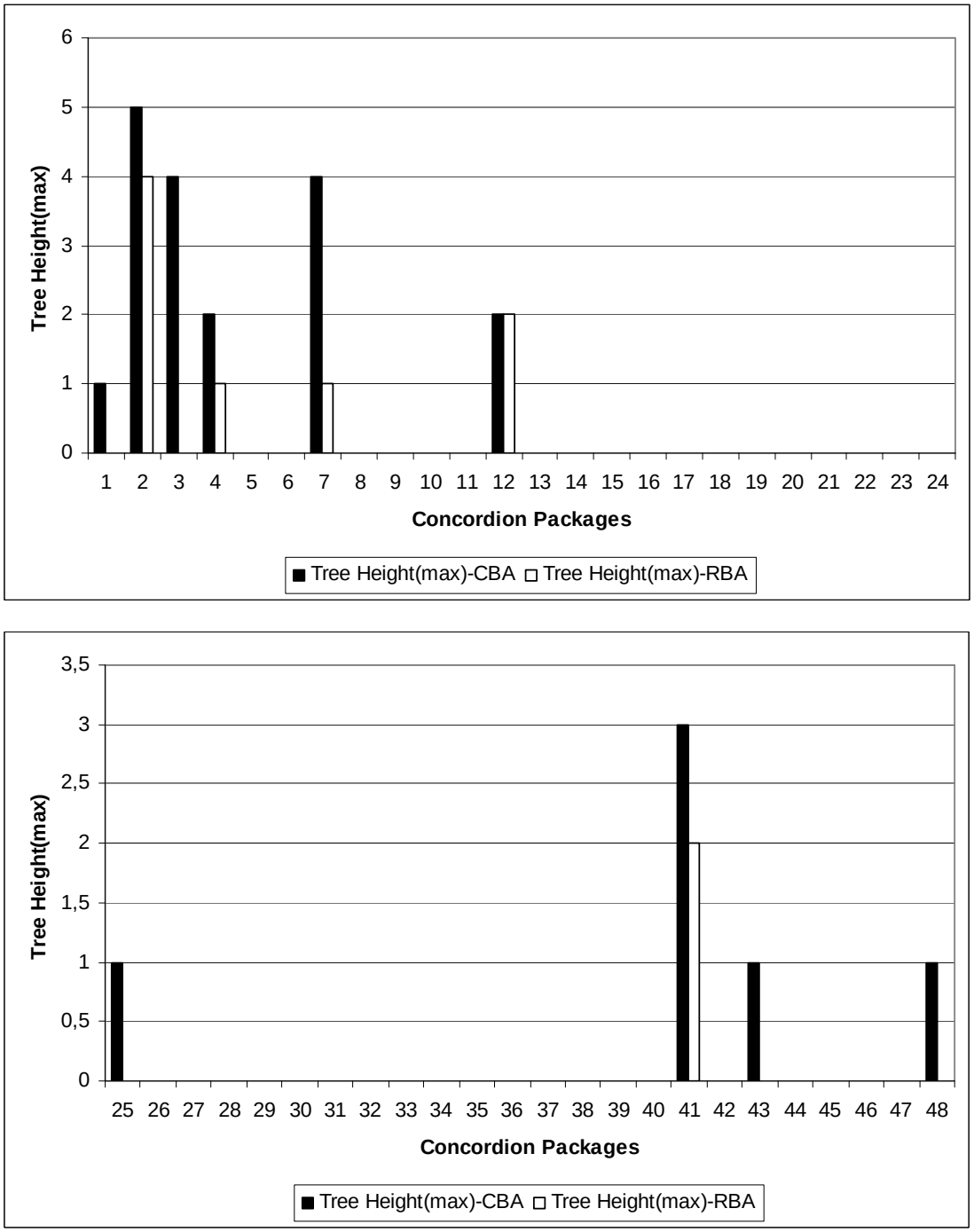

Σχήμα Β.20 Σύγκριση Μέσης Τιμής Πλήθους Διεπαφών μεταξύ CBA Adaptive και RBA.



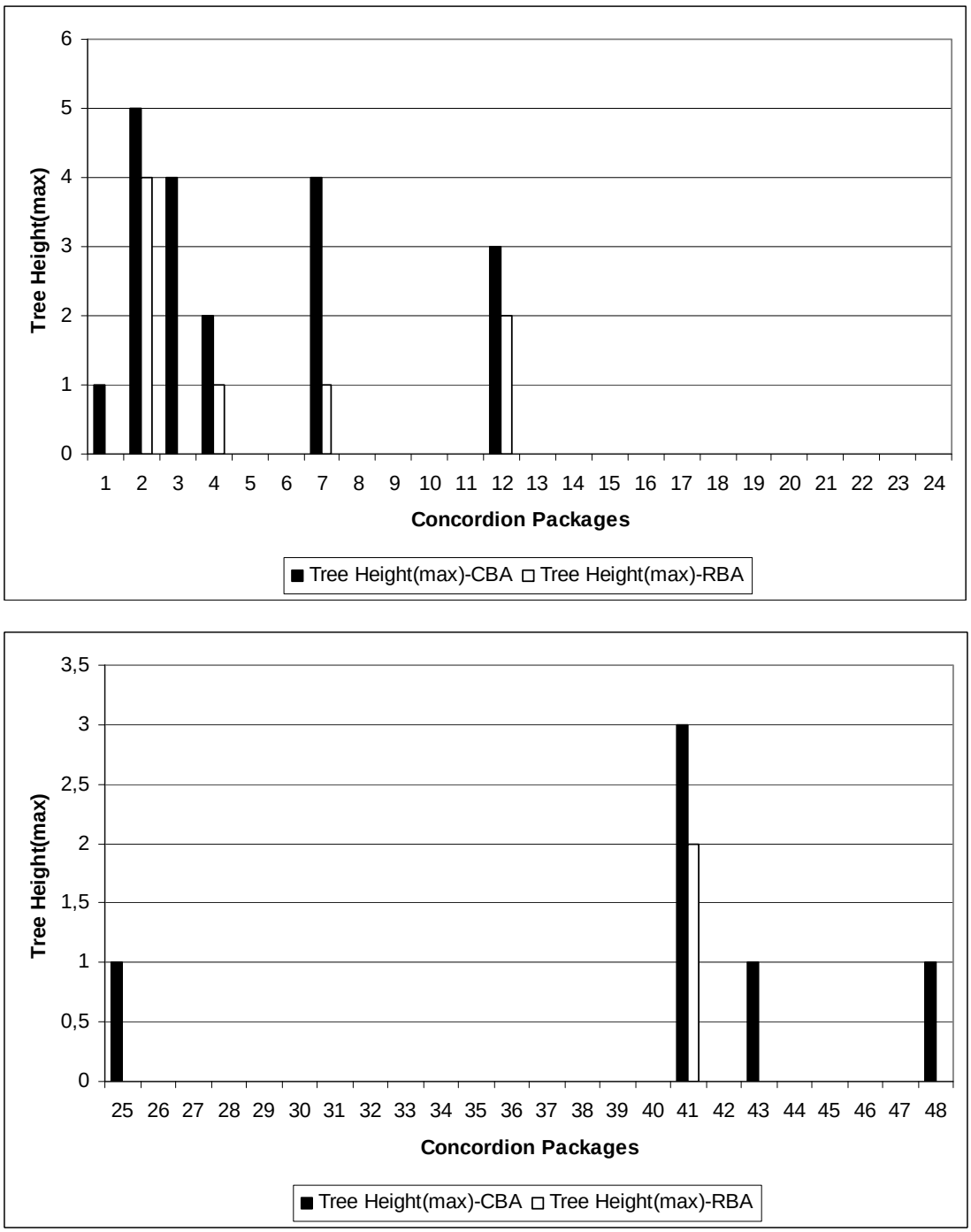

Σχήμα Β.21 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Single και RBA.



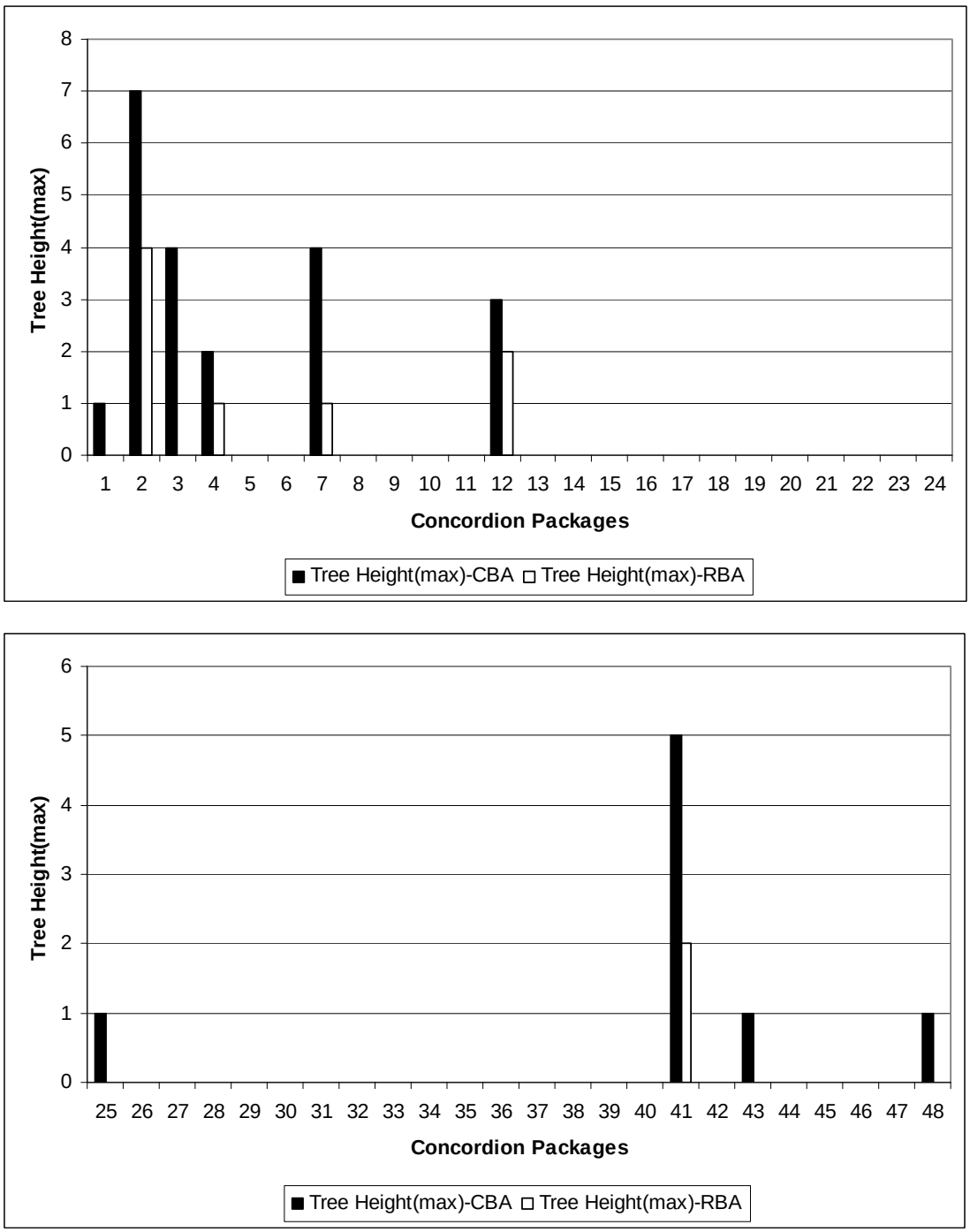

Σχήμα Β.22 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Average και RBA.



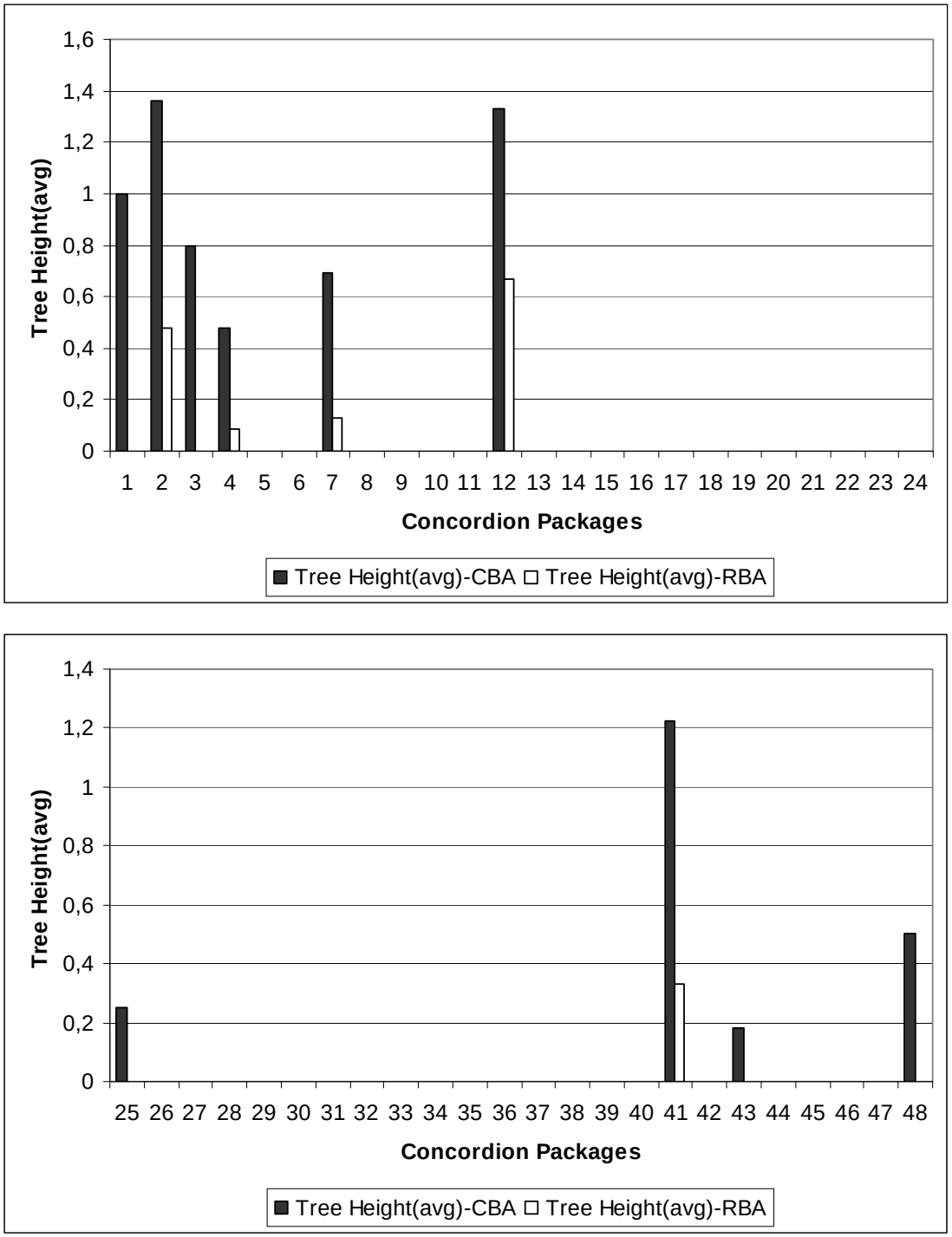

Σχήμα Β.23 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Complete και RBA.



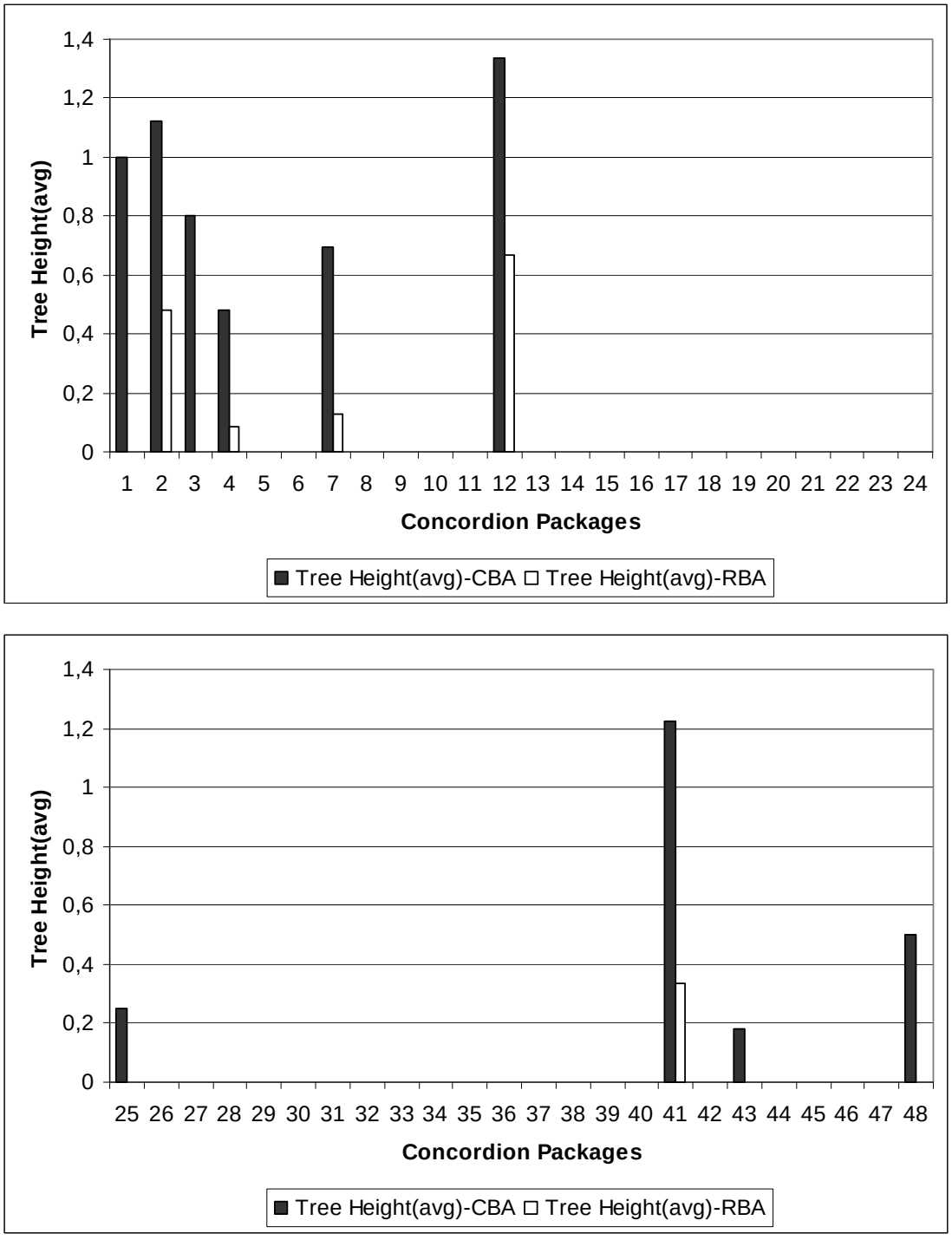

Σχήμα Β.24 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Median και RBA.



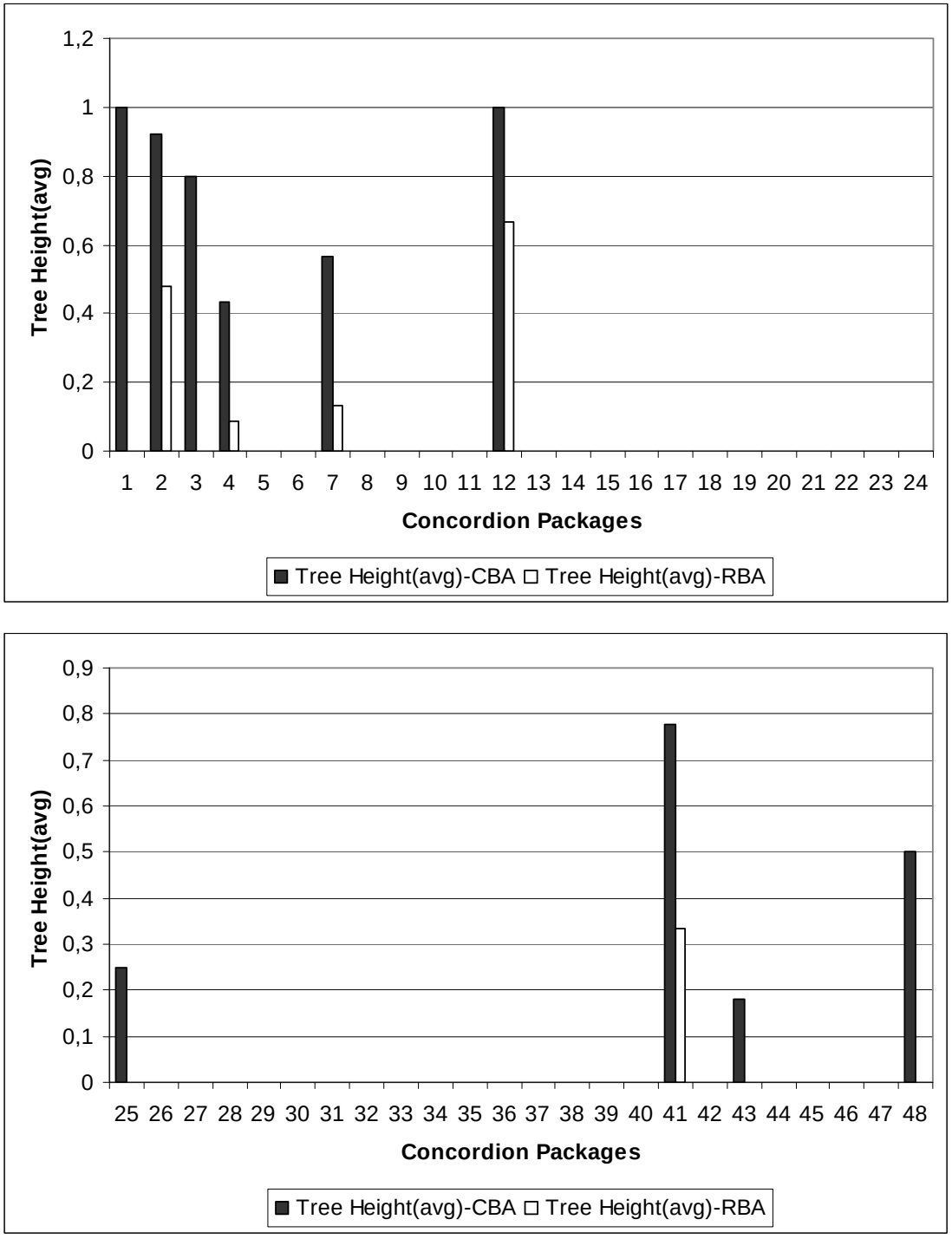

Σχήμα Β.25 Σύγκριση Μέγιστου Ύψους Δένδρων μεταξύ CBA Adaptive και RBA.



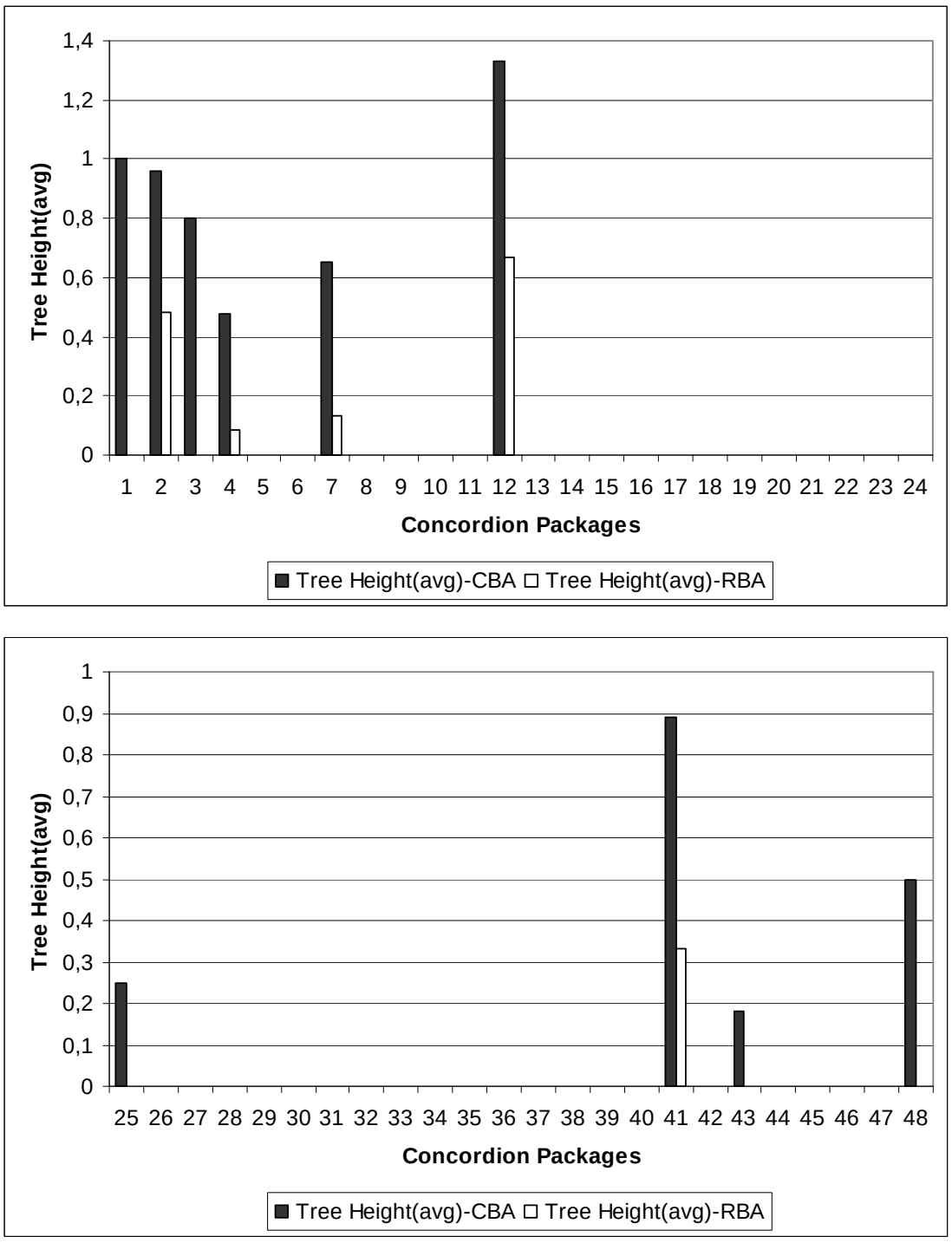

Σχήμα Β.26 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Single και RBA.



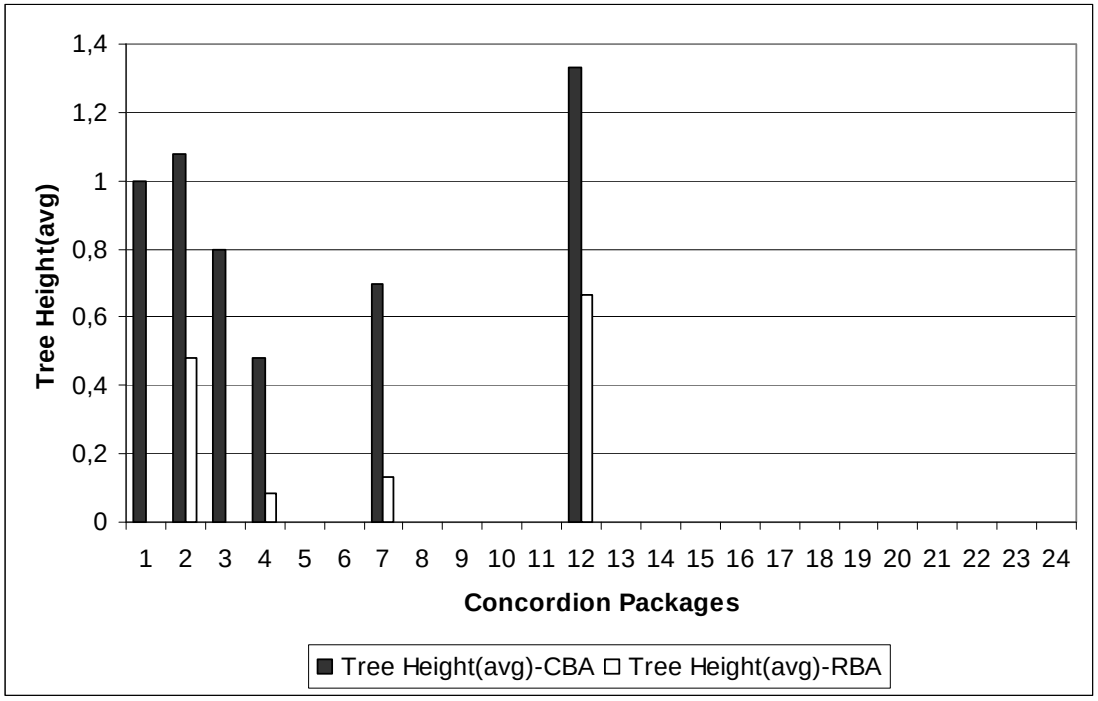

Σχήμα Β.27 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Average και RBA.

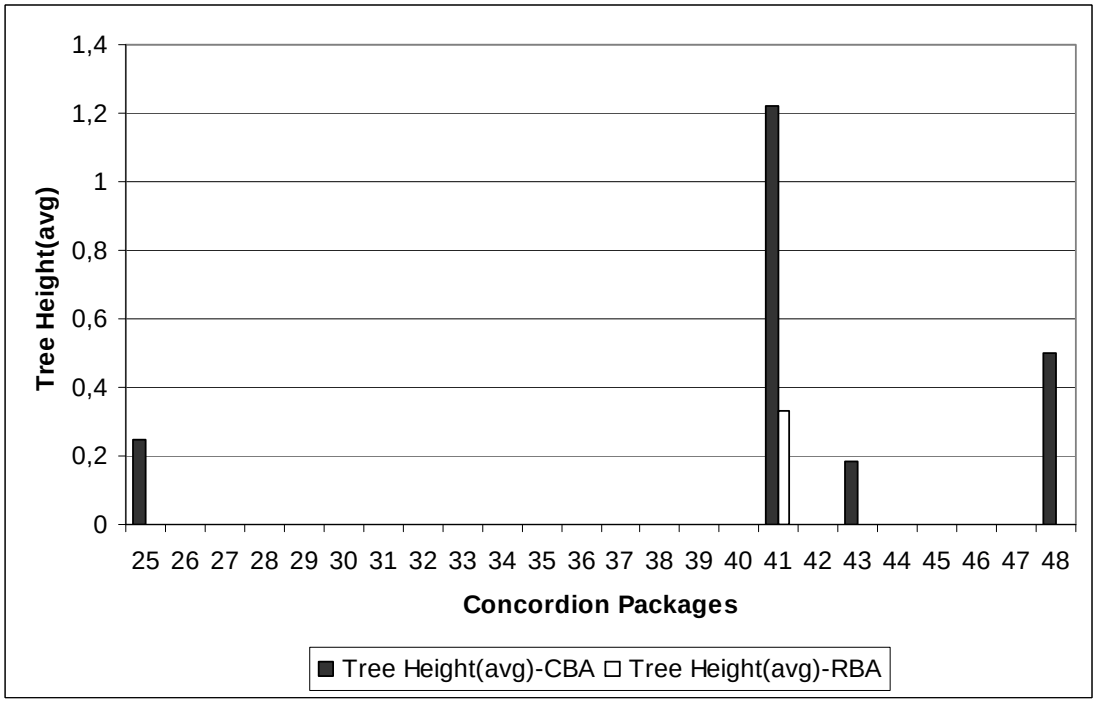



Σχήμα Β.28 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Complete και RBA.




Σχήμα Β.29 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Median και RBA.




Σχήμα Β.30 Σύγκριση Μέσης Τιμής Ύψους Δένδρων CBA Adaptive και RBA.

# ΣΥΝΤΟΜΟ ΒΙΟΓΡΑΦΙΚΟ

Ο Σπυρίδων Κρανάς γεννήθηκε στα Ιωάννινα το 1986. Αποφοίτησε το 2004 από το «Μαρούλειο» Εννιαίο Λύκειο Κατσικά Ιωαννίνων. Οι βασικές σπουδές πραγματοποιήθηκαν στο Τμήμα Πληροφορικής του Πανεπιστημίου Ιωαννίνων, όπου εισήχθη το 2005 και αποφοίτησε το 2010. Συνέχισε για Μεταπτύχιακές Σπουδές στο ίδιο Ίδρυμα και εξειδικεύτηκε στο Λογισμικό. Τα ενδιαφέροντά του περιλαμβάνουν την ανακατασκευή λογισμικού, την τεχνολογία λογισμικού και τις τεχνολογίες διαδικτύου.