ΕΚΔΟΣΕΙΣ ΚΛΕΙΔΑΡΙΘΜΟΣ

Ο Πρακτικός Προγραμματιστής

μάθετε την τέχνη σας
στο έπακρο

Andrew Hunt
David Thomas

Πρόλογος από τον Ward Cunningham,
δημιουργό του πρώτου Wiki

Περιεχόμενα

Παραρτήματα

Κεφάλαιο 6

Την ώρα της συγγραφής κώδικα

Η κοινή λογική λέει ότι από τη στιγμή που ένα έργο θα φτάσει στη φάση της συγγραφής κώδικα, η εργασία είναι κυρίως μηχανική, καθώς αφορά απλώς τη μεταφορά τού σχεδιασμού σε εκτελέσιμες εντολές. Πιστεύουμε ότι τούτη η άποψη αποτελεί την κυριότερη αιτία για το ότι πολλά προγράμματα είναι κακογραμμένα, μη αποδοτικά, ανεπαρκώς δομημένα, μη συντηρήσιμα, και, με μία λέξη, λάθος.

Η συγγραφή κώδικα δεν είναι μηχανική εργασία. Αν ήταν, όλα τα εργαλεία CASE στα οποία βάσισαν τις ελπίδες τους οι άνθρωποι στις αρχές της δεκαετίας του 1980 θα είχαν αντικαταστήσει τους προγραμματιστές εδώ και πολύ καιρό. Κάθε λεπτό λαμβάνονται αποφάσεις, οι οποίες απαιτούν προσεκτική σκέψη και κρίση αν θέλουμε το πρόγραμμά μας να απολαύσει μια μακρά, συνεπή, και παραγωγική ζωή.

Οι προγραμματιστές που δεν σκέφτονται ενεργά για τον κώδικά τους προγραμματίζουν στην τύχη — ο κώδικας μπορεί να λειτουργεί, αλλά δεν υπάρχει κάποιος συγκεκριμένος λόγος γι' αυτό. Στην ενότητα *Προγραμματισμός στην τύχη*, υποστηρίζουμε έναν πιο θετικό τρόπο συμμετοχής στη διαδικασία συγγραφής κώδικα.

Ενώ το μεγαλύτερο μέρος του κώδικα που γράφουμε εκτελείται γρήγορα, περιστασιακά αναπτύσσουμε αλγορίθμους οι οποίοι έχουν τη δυνατότητα να "γονατίσουν" ακόμη και τους γρηγορότερους επεξεργαστές. Στην ενότητα *Ταχύτητα αλγορίθμων*, περιγράφουμε τρόπους υπολογισμού της ταχύτητας του κώδικα, και δίνουμε ορισμένες συμβουλές για το πώς μπορούμε να εντοπίζουμε πιθανά προβλήματα πριν αυτά συμβούν.

Ως Πρακτικοί Προγραμματιστές προσεγγίζουμε κάθε κομμάτι κώδικα με κριτική σκέψη, συμπεριλαμβανομένων εκείνων που έχουμε γράψει οι ίδιοι. Διακρίνουμε συνεχώς περιθώρια για βελτίωση στα προγράμματά μας και τους σχεδιασμούς μας. Στην ενότητα *Ανακατασκευή*, εξετάζουμε τεχνικές οι οποίες μας βοηθούν να διευθετήσουμε υπάρχοντα κώδικα ακόμη και ενώ βρισκόμαστε στη μέση ενός έργου.

Κάτι που πρέπει να κρατάτε στο πίσω μέρος του μυαλού σας κάθε φορά που δημιουργείτε κώδικα είναι ότι κάποια μέρα θα πρέπει να τον ελέγξετε. Δη-

μιουργώντας κώδικα ο οποίος μπορεί να ελεγχθεί εύκολα, αυξάνετε την πιθανότητα να πραγματοποιηθεί τελικά ο εν λόγω έλεγχος, μια σκέψη που αναπτύσσουμε στην ενότητα *Κώδικας που ελέγχεται εύκολα*.

Τέλος, στην ενότητα *Κακοί μάγοι* σάς προτείνουμε να είστε προσεκτικοί με εργαλεία τα οποία γράφουν κώδικα για λογαριασμό σας, εκτός και αν καταλαβαίνετε τι κάνουν.

Οι περισσότεροι από μας μπορούν να οδηγήσουν ένα αυτοκίνητο σε μεγάλο βαθμό ασυναίσθητα — δεν δίνουμε ρητή εντολή στο πόδι μας να πιέσει κάποιο πεντάλ, ή στο χέρι μας να στρίψει το τιμόνι — σκεφτόμαστε απλώς "κόψε ταχύτητα και στρίψε δεξιά". Ωστόσο, οι καλοί, προσεκτικοί οδηγοί αξιολογούν συνεχώς την κατάσταση, ελέγχουν για πιθανά προβλήματα, και είναι προετοιμασμένοι για την περίπτωση που θα συμβεί κάτι απροσδόκητο. Το ίδιο ισχύει για τη συγγραφή κώδικα — ίσως αποτελεί ρουτίνα σε μεγάλο βαθμό, αλλά η επαγρύπνηση θα μπορούσε να σας βοηθήσει να αποφύγετε μια καταστροφή.

Προγραμματισμός στην τύχη

31

Βλέπετε ποτέ παλιά ασπρόμαυρα πολεμικά έργα; Ο κουρασμένος στρατιώτης προχωράει προσεκτικά μέσα στους θάμνους. Μπροστά του ξεπροβάλλει ένα ξέφωτο: υπάρχουν άραγε νάρκες, ή είναι ασφαλές να προχωρήσει; Δεν υπάρχουν ενδείξεις ότι πρόκειται για ναρκοπέδιο — κανένα σημάδι, συρματόπλεγμα, ή κρατήρες. Ο στρατιώτης κεντρίζει το έδαφος μπροστά του με την ξιφολόγχη και μορφάζει, περιμένοντας την έκρηξη. Δεν γίνεται τίποτα. Έτσι, προχωράει αργά-αργά στο ξέφωτο για λίγο, ελέγχοντας το έδαφος στο διάβα του. Τελικά, πεπεισμένος ότι το πεδίο είναι ασφαλές, ισιώνει το κορμί του και προελαύνει υπερήφανα εμπρός, με αποτέλεσμα να γίνει κομμάτια.

Οι αρχικοί έλεγχοι του στρατιώτη για νάρκες δεν αποκάλυψαν τίποτα, αλλά αυτό ήταν απλώς συμπτωματικό. Οδηγήθηκε σε ένα ψευδές συμπέρασμα — με καταστρεπτικά αποτελέσματα.

Ως προγραμματιστές, εργαζόμαστε επίσης σε ναρκοπέδια. Υπάρχουν εκατοντάδες παγίδες οι οποίες παραμονεύουν κάθε μέρα. Παίρνοντας παράδειγμα από την ιστορία του στρατιώτη, πρέπει να είμαστε ιδιαίτερα προσεκτικοί ώστε να μη συνάγουμε ψευδή συμπεράσματα. Πρέπει να αποφεύγουμε τον προγραμματισμό στην τύχη — δηλαδή, το να βασιζόμαστε στις συμπτώσεις και τις τυχαίες επιτυχίες. Αντίθετα, είναι απαραίτητο να *προγραμματίζουμε σκοπίμως*.

Πώς να προγραμματίζετε στην τύχη

Υποθέστε ότι ανατίθεται στον Κώστα μια προγραμματιστική εργασία. Εκείνος πληκτρολογεί λίγο κώδικα, τον δοκιμάζει, και φαίνεται να λειτουργεί. Ο Κώστας πληκτρολογεί λίγο ακόμα κώδικα, τον δοκιμάζει, και ξανά όλα φαίνονται εντάξει. Μετά από αρκετές εβδομάδες συγγραφής κώδικα κατ' αυτό τον τρόπο, το πρόγραμμα ξαφνικά σταματάει να λειτουργεί, και μετά από ώρες προσπαθειών για να το διορθώσει, ο Κώστας δεν έχει καταλάβει ακόμα το λόγο. Ο Κώστας ίσως αφιερώσει πολύ χρόνο σε τούτο το κομμάτι κώδικα, χωρίς να καταφέρει να το διορθώσει ποτέ. Ό,τι και να κάνει, το πρόγραμμα δεν φαίνεται να λειτουργεί σωστά.

Ο Κώστας δεν γνωρίζει γιατί παρουσιάζει πρόβλημα ο κώδικας, επειδή *δεν ήξερε γιατί λειτούργησε εξαρχής*. Φαινόταν να λειτουργεί, με δεδομένο τον περιορισμένο "έλεγχο" που πραγματοποιήθηκε, αλλά αυτό δεν ήταν παρά μια σύμπτωση. Ακολουθώντας μια ψεύτικη αίσθηση αυτοπεποίθησης, ο Κώστας οδηγήθηκε στην καταστροφή. Βέβαια, οι περισσότεροι έξυπνοι άνθρωποι ίσως έχουμε ακουστά κάποιον σαν τον Κώστα, αλλά όσον αφορά τη δική μας δουλειά προσέχουμε περισσότερο. Δεν στηριζόμαστε στις συμπτώσεις — ή μήπως το κάνουμε;

Μερικές φορές μπορεί και να το κάνουμε. Μερικές φορές είναι αρκετά εύκολο να μπερδέψεις μια ευτυχή σύμπτωση με ένα σκόπιμο σχέδιο. Ας δούμε ορισμένα παραδείγματα.

Ατυχήματα της υλοποίησης

Ατυχήματα στην υλοποίηση συμβαίνουν απλώς και μόνο επειδή αυτός είναι ο τρόπος με τον οποίο γράφεται ο κώδικας επί του παρόντος. Καταλήγετε να βασίζεστε σε ατεκμηρίωτες συνθήκες σφαλμάτων ή ορίων.

Υποθέστε ότι καλείτε μια ρουτίνα με ακατάλληλα δεδομένα. Η ρουτίνα αποκρίνεται με έναν συγκεκριμένο τρόπο, και εσείς γράφετε κώδικα με βάση τη συγκεκριμένη απόκριση. Ο δημιουργός, όμως, της ρουτίνας δεν την προόριζε να λειτουργήσει κατ' αυτό τον τρόπο — δεν είχε σκεφτεί καν μια τέτοιου είδους χρήση. Όταν η ρουτίνα "διορθωθεί", ο κώδικάς σας μπορεί να καταρρεύσει. Στην πιο ακραία περίπτωση, η ρουτίνα που καλέσατε ίσως να μην έχει καν σχεδιαστεί για να κάνει αυτό που θέλετε, αλλά να *φαίνεται* απλώς ότι λειτουργεί εντάξει. Ένα σχετικό πρόβλημα αποτελεί η κλήση πραγμάτων με λανθασμένη σειρά, ή σε λανθασμένο πλαίσιο.

```
paint(g);
invalidate();
validate();
revalidate();
repaint();
paintImmediately(r);
```

Εδώ ο Κώστας μοιάζει να προσπαθεί απελπισμένα να τυπώσει κάτι στην οθόνη. Αλλά αυτές οι ρουτίνες δεν ήταν ποτέ σχεδιασμένες ώστε να καλούνται με αυτό τον τρόπο· αν και φαίνονται να λειτουργούν, στην πραγματικότητα αυτό είναι μια απλή σύμπτωση.

Χειροτερεύοντας τα πράγματα, όταν το συστατικό στοιχείο τελικά εμφανιστεί στην οθόνη, ο Κώστας δεν θα προσπαθήσει να απομακρύνει τις προβληματικές κλήσεις. "Τώρα δουλεύει, οπότε καλύτερα να μην το πειράξουμε..."

Είναι εύκολο να ξεγελαστείς από τούτο τον τρόπο σκέψης. Γιατί να κινδυνεύσεις τροποποιώντας κάτι που λειτουργεί; Μπορούμε να σκεφτούμε διάφορους λόγους:

- Ίσως να μην λειτουργεί πραγματικά — ίσως να φαίνεται ότι λειτουργεί.
- Η οριακή συνθήκη στην οποία βασίζεστε μπορεί να είναι απλώς ένα ατύχημα. Σε διαφορετικές περιστάσεις (λόγου χάρη, με μια διαφορετική ανάλυση οθόνης), ίσως συμπεριφερθεί διαφορετικά.
- Στην επόμενη έκδοση της βιβλιοθήκης μπορεί να έχουν αλλάξει μη τεκμηριωμένες συμπεριφορές.
- Πρόσθετες και περιττές κλήσεις καθιστούν τον κώδικά σας πιο αργό.
- Οι πρόσθετες κλήσεις αυξάνουν επίσης τον κίνδυνο για παρουσία νέων σφαλμάτων.

Για κώδικα που γράφετε εσείς και ο οποίος θα καλείται από άλλους, μπορούν να βοηθήσουν οι βασικές αρχές της καλής τμηματοποίησης (modularization) και της απόκρυψης της υλοποίησης πίσω από μικρές, καλά τεκμηριωμένες διασυνδέσεις. Ένα σαφώς καθορισμένο συμβόλαιο (δείτε την ενότητα *Σχεδιασμός βάσει συμβολαίου*, σελίδα 149) θα μπορούσε να σας βοηθήσει να εξαλείψετε τις παρανοήσεις.

Όσον αφορά ρουτίνες που καλείτε, βασιστείτε μόνο στην τεκμηριωμένη συμπεριφορά τους. Αν δεν μπορείτε, για οποιονδήποτε λόγο, τεκμηριώστε κατόπιν τις παραδοχές σας με σαφήνεια.

Ατυχήματα θεματικού πλαισίου

Ίσως επίσης συναντήσετε "ατυχήματα θεματικού πλαισίου". Υποθέστε ότι γράφετε μια βοηθητική λειτουργική μονάδα. Πρέπει η εν λόγω μονάδα να βασιστεί σε κάποιο υπάρχον GUI απλώς και μόνο επειδή τώρα γράφετε κώδικα για ένα τέτοιο περιβάλλον; Θεωρείτε ότι απευθύνεστε σε αγγλόφωνους χρήστες; Σε εγγράμματους χρήστες; Σε τι άλλο στηρίζεστε το οποίο δεν είναι εγγυημένο;

Υπονοούμενες παραδοχές

Οι συμπτώσεις μπορούν να παραπλανήσουν σε όλα τα επίπεδα — από την παραγωγή απαιτήσεων μέχρι τον έλεγχο. Ιδιαίτερα ο έλεγχος είναι γεμάτος με ψευδείς αιτιότητες και συμπτωματικές εκβάσεις. Είναι εύκολο να υποθέσουμε ότι το Χ προκαλεί το Υ, αλλά όπως είπαμε στην ενότητα *Αποσφαλμάτωση*, σελίδα 127: μην υποθέτεις, απόδειξέ το.

Σε κάθε επίπεδο, οι άνθρωποι λειτουργούν έχοντας πολλές παραδοχές στο μυαλό τους — αλλά τούτες οι παραδοχές σπανίως είναι τεκμηριωμένες, και συχνά διαφέρουν από προγραμματιστή σε προγραμματιστή. Οι παραδοχές που δεν είναι βασισμένες σε καλά εδραιωμένα γεγονότα αποτελούν τον όλεθρο κάθε έργου.

ΣΥΜΒΟΥΛΗ 44

Μην προγραμματίζετε στην τύχη

Πώς να προγραμματίζετε σκοπίμως

Θέλουμε να αφιερώνουμε λιγότερο χρόνο στη συγγραφή κώδικα, να εντοπίζουμε και να διορθώνουμε τα σφάλματα όσο το δυνατόν νωρίτερα στον κύκλο ανάπτυξης, και να κάνουμε λιγότερα σφάλματα εξαρχής. Ο σκόπιμος προγραμματισμός μπορεί να βοηθήσει:

- Πάντα να έχετε συνείδηση αυτού που κάνετε. Ο Κώστας άφησε σιγά-σιγά τα πράγματα να ξεφύγουν από τον έλεγχο, ώσπου κατέληξε βραστός, όπως ο βάτραχος στην ενότητα *Πετρόσουπες και βραστοί βάτραχοι*, σελίδα 33.
- Μην γράφετε κώδικα με κλειστά τα μάτια. Το να προσπαθείς να κατασκευάσεις μια εφαρμογή την οποία δεν κατανοείς πλήρως, ή να χρησιμοποιήσεις μια τεχνολογία με την οποία δεν είσαι εξοικειωμένος, είναι σαν να πηγαίνεις γυρεύοντας να παραπλανηθείς από συμπτώσεις.

- Προχωρείτε βάσει σχεδίου, είτε το σχέδιο αυτό είναι στο μυαλό σας, είτε επάνω σε μια χαρτοπετσέτα, είτε σε κάποια εκτύπωση από ένα εργαλείο CASE η οποία καταλαμβάνει ολόκληρο τον τοίχο.
- Να βασίζεστε μόνο σε αξιόπιστα πράγματα. Μην εξαρτάστε από ατυχήματα ή παραδοχές. Αν δεν μπορείτε να αποφασίσετε σε κάποιες περιστάσεις, υποθέστε το χειρότερο.
- Τεκμηριώστε τις παραδοχές σας. Η ενότητα *Σχεδιασμός βάσει συμβολαίου*, σελίδα 149, μπορεί να σας βοηθήσει να αποσαφηνίσετε τις παραδοχές στο μυαλό σας, καθώς επίσης και να τις μεταβιβάσετε σε άλλους.
- Μην ελέγχετε μόνο τον κώδικά σας, αλλά και τις παραδοχές σας. Μην μαντεύετε· δοκιμάστε το στην πραγματικότητα. Γράψτε έναν ισχυρισμό για να ελέγξετε τις παραδοχές σας (δείτε την ενότητα *Προγραμματισμός με ισχυρισμούς*, σελίδα 164). Αν ο ισχυρισμός σας είναι σωστός, έχετε βελτιώσει την τεκμηρίωση στον κώδικά σας. Αν διαπιστώσετε ότι η παραδοχή σας είναι λανθασμένη, τότε θεωρήστε τον εαυτό σας τυχερό.
- Καθορίστε προτεραιότητες στις προσπάθειές σας. Αφιερώστε χρόνο σε σημαντικές πλευρές· πιθανότατα αυτές θα αποτελούν και τα πιο δύσκολα κομμάτια. Αν δεν έχετε καθορίσει σωστά τις βασικές αρχές ή την υποδομή, οι μετέπειτα καλλωπισμοί δεν πρόκειται να σας βοηθήσουν.
- Μην είστε σκλάβοι της ιστορίας. Μην αφήνετε τον υπάρχοντα κώδικα να υπαγορεύει τον μελλοντικό. Αν το σώμα του κώδικα δεν είναι πλέον κατάλληλο, μπορείτε να το αντικαταστήσετε όλο. Ακόμη και στα πλαίσια ενός μοναδικού προγράμματος, μην αφήνετε ό,τι έχετε ήδη κάνει να περιορίζει το τι θα κάνετε στη συνέχεια — να είστε έτοιμοι για ανακατασκευή (δείτε την ενότητα *Ανακατασκευή*, σελίδα 231). Μια τέτοια απόφαση ίσως επηρεάσει το χρονοδιάγραμμα του έργου. Η ιδέα είναι ότι η επιβάρυνση θα είναι μικρότερη από το κόστος που θα προκύψει αν *δεν* γίνει η αλλαγή[1].

Έτσι, την επόμενη φορά που κάτι φαίνεται να λειτουργεί, αλλά δεν ξέρετε γιατί, σιγουρευτείτε ότι δεν είναι μόνο μια σύμπτωση.

[1] Και εδώ όμως μπορείτε να το παρακάνετε. Κάποτε γνωρίζαμε έναν προγραμματιστή ο οποίος ξανάγραψε όλον τον κώδικα που του δόθηκε επειδή ήθελε να εφαρμόσει τις δικές του συμβάσεις ονομασίας.

Σχετικές ενότητες:

- *Πετρόσουπες και βραστοί βάτραχοι*, σελίδα 33
- *Αποσφαλμάτωση*, σελίδα 127
- *Σχεδιασμός βάσει συμβολαίου*, σελίδα 149
- *Προγραμματισμός με ισχυρισμούς*, σελίδα 164
- *Χρονική σύζευξη*, σελίδα 194
- *Ανακατασκευή*, σελίδα 231
- *Όλα για τη συγγραφή*, σελίδα 301

Ασκήσεις

31. Μπορείτε να αναγνωρίσετε μερικές συμπτώσεις στο ακόλουθο τμήμα κώδικα C; Υποθέστε ότι ο κώδικας είναι βαθιά θαμμένος σε μια ρουτίνα βιβλιοθήκης.

```
fprintf (stderr, "Σφάλμα, συνέχεια;");
gets(buf); (Απάντηση στη σελ. 359)
```

32. Το επόμενο κομμάτι κώδικα C ίσως να λειτουργήσει για λίγο χρόνο σε κάποιες μηχανές. Από την άλλη, ίσως όχι. Ποιο είναι το πρόβλημα;

```
/* Περικοπή αλφαριθμητικού στους τελευταίους maxlen χαρακτήρες */
void string_tail(char *string, int maxlen) {
  int len = strlen(string);
  if (len > maxlen) {
    strcpy(string, string + (len - maxlen));
  }
} (Απάντηση στη σελ. 359)
```

33. Ο παρακάτω κώδικας προέρχεται από ένα πακέτο παρακολούθησης (tracing) γενικής χρήσης σε Java. Η συνάρτηση γράφει ένα αλφαριθμητικό σε ένα αρχείο ημερολογίου. Περνάει τον έλεγχο υπομονάδας της, αλλά αποτυγχάνει όταν χρησιμοποιείται από κάποιον προγραμματιστή Ιστού. Σε ποια σύμπτωση βασίζεται;

```
public static void debug(String s) throws IOException {
  FileWriter fw = new FileWriter("debug.log", true);
  fw.write(s);
  fw.flush();
  fw.close();
} (Απάντηση στη σελ. 360)
```

Ταχύτητα αλγορίθμων

32

Στην ενότητα *Εκτίμηση*, σελίδα 97, μιλήσαμε για την εκτίμηση πραγμάτων, όπως πόσο χρόνο θα πάρει σε κάποιον να διασχίσει την πόλη ή πόσο χρόνο θα χρειαστεί η ολοκλήρωση ενός έργου. Ωστόσο, υπάρχει ένα άλλο είδος εκτίμησης που οι Πρακτικοί Προγραμματιστές χρησιμοποιούν σχεδόν καθημερινά: η εκτίμηση των πόρων οι οποίοι χρησιμοποιούνται από αλγόριθμους — χρόνος, επεξεργαστής, μνήμη, κ.ο.κ.

Τούτο το είδος εκτίμησης είναι συχνά κρίσιμο. Όταν σας παρέχονται δύο τρόποι για να κάνετε κάτι, ποιον επιλέγετε; Γνωρίζετε πόσο χρόνο χρειάζεται το πρόγραμμά σας για 1.000 εγγραφές, αλλά τι γίνεται όταν οι εγγραφές φτάσουν τις 1.000.000; Ποια μέρη του κώδικα χρειάζονται βελτιστοποίηση;

Τέτοιες ερωτήσεις μπορούν συχνά να απαντηθούν με την κοινή λογική, λίγη ανάλυση, και έναν τρόπο έκφρασης προσεγγίσεων ο οποίος ονομάζεται "συμβολισμός O" (big O notation).

Τι εννοούμε με τη φράση "εκτίμηση αλγορίθμων";

Οι περισσότεροι σύνθετοι αλγόριθμοι χειρίζονται κάποιο είδος μεταβλητής εισόδου — ταξινομούν n αλφαριθμητικά, αναστρέφουν μια μήτρα $m \times n$, ή αποκρυπτογραφούν ένα μήνυμα με κλειδί n bit. Συνήθως, το μέγεθος αυτής της εισόδου θα επηρεάσει τον αλγόριθμο: όσο μεγαλύτερη η είσοδος, τόσο πιο μεγάλος ο χρόνος εκτέλεσης ή τόσο περισσότερη η μνήμη που χρησιμοποιείται.

Αν η σχέση ήταν πάντα γραμμική (έτσι ώστε ο χρόνος να αυξάνεται ευθέως ανάλογα με την τιμή τού n), η παρούσα ενότητα δεν θα χρειαζόταν. Ωστόσο, οι πιο σημαντικοί αλγόριθμοι δεν είναι γραμμικοί. Τα καλά νέα είναι ότι πολλοί είναι υπογραμμικοί. Μια δυαδική αναζήτηση, για παράδειγμα, δεν χρειάζεται να εξετάζει κάθε υποψήφιο κατά την αναζήτηση μιας ταύτισης. Τα κακά νέα είναι ότι άλλοι αλγόριθμοι είναι αρκετά χειρότεροι από τους γραμμικούς· οι χρόνοι εκτέλεσης ή οι απαιτήσεις μνήμης αυξάνονται πολύ γρηγορότερα από το n. Ένας αλγόριθμος ο οποίος χρειάζεται ένα λεπτό για να επεξεργαστεί δέκα στοιχεία, ίσως χρειαστεί μια ολόκληρη ζωή για να επεξεργαστεί 100.

Διαπιστώνουμε ότι όποτε γράφουμε κάτι που περιέχει βρόχους ή αναδρομικές κλήσεις, ελέγχουμε υποσυνείδητα τις απαιτήσεις σε χρόνο εκτέλεσης και μνήμη. Η διαδικασία αυτή σπανίως είναι τυποποιημένη, αλλά αποτελεί μάλλον μια γρήγορη επιβεβαίωση ότι αυτό που κάνουμε έχει νόημα με βάση τις περιστάσεις. Ωστόσο, μερικές φορές θέλουμε να πραγματοποιήσουμε μια πιο λεπτομερή ανάλυση. Τότε μας είναι χρήσιμος ο συμβολισμός $O()$.

Ο συμβολισμός *O*()

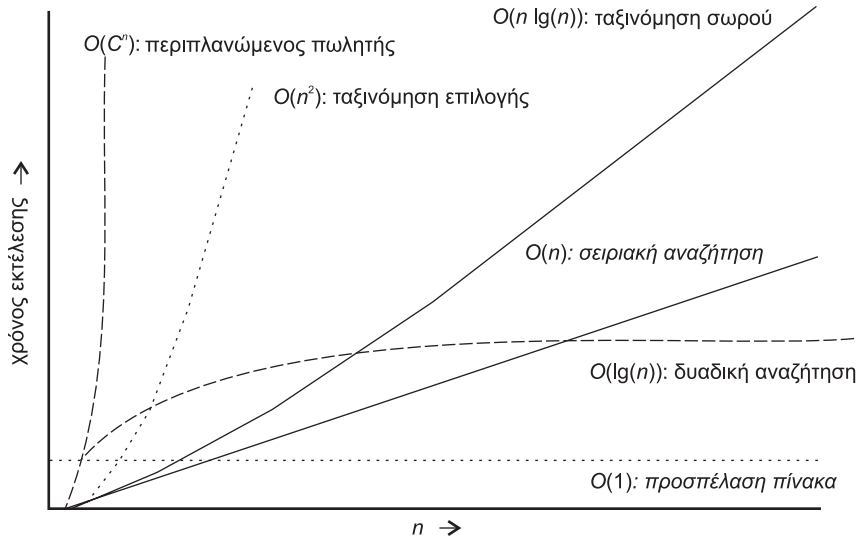
Ο συμβολισμός *O*() είναι ένας μαθηματικός τρόπος για το χειρισμό των προσεγγίσεων. Όταν γράφουμε ότι μια συγκεκριμένη ρουτίνα ταξινόμησης ταξινομεί n εγγραφές σε χρόνο $O(n^2)$, απλώς δηλώνουμε ότι στη χειρότερη περίπτωση ο απαιτούμενος χρόνος θα μεταβάλλεται ως το τετράγωνο του n. Αν διπλασιάσουμε το πλήθος των εγγραφών, ο χρόνος θα τετραπλασιαστεί. Σκεφτείτε το συμβολισμό *O* με την έννοια "*της τάξης τού*". Ο συμβολισμός *O*() θέτει ένα ανώτατο όριο στην τιμή της πλευράς που μετράμε (χρόνος, μνήμη, κ.ο.κ.). Αν πούμε ότι μια συνάρτηση χρειάζεται χρόνο $O(n^2)$, γνωρίζουμε ότι το ανώτατο όριο του απαιτούμενου χρόνου δεν θα αυξηθεί γρηγορότερα από n^2. Μερικές φορές καταλήγουμε με αρκετά σύνθετες συναρτήσεις *O*(), αλλά επειδή ο όρος της υψηλότερης τάξης θα κυριαρχεί στην τιμή καθώς αυξάνει το n, η σύμβαση είναι να αφαιρούνται όλοι οι όροι κατώτερης τάξης, και να μην παρουσιάζονται τυχόν σταθεροί παράγοντες πολλαπλασιασμού. Έτσι, η συνάρτηση $O(n^2/2 + 3n)$ είναι ίδια με την $O(n^2/2)$, η οποία είναι ισοδύναμη με την $O(n^2)$. Αυτό αποτελεί μια αδυναμία του συμβολισμού *O*() — ένας αλγόριθμος $O(n^2)$ μπορεί να είναι 1.000 φορές γρηγορότερος από έναν άλλο αλγόριθμο $O(n^2)$, αλλά αυτό δεν φαίνεται από το συμβολισμό.

Η Εικόνα 6.1 στην επόμενη σελίδα παρουσιάζει διάφορους συνηθισμένους συμβολισμούς *O*() που θα συναντήσετε, μαζί με ένα γράφημα το οποίο συγκρίνει χρόνους εκτέλεσης αλγορίθμων σε κάθε κατηγορία. Σαφώς, τα πράγματα βγαίνουν γρήγορα εκτός ελέγχου μετά το $O(n^2)$.

Για παράδειγμα, υποθέστε ότι έχετε μια ρουτίνα η οποία χρειάζεται 1 δευτερόλεπτο για να επεξεργαστεί 100 εγγραφές. Πόσο θα χρειαστεί για να επεξεργαστεί 1.000; Αν ο κώδικάς σας είναι στο $O(1)$, θα χρειαστεί και πάλι μόνο ένα 1 δευτερόλεπτο. Αν είναι στο $O(\lg(n))$, θα περιμένετε πιθανότατα γύρω στα 3 δευτερόλεπτα. Με $O(n)$ θα αυξηθεί γραμμικά σε 10 δευτερόλεπτα, ενώ με $O(n \lg(n))$ θα πάρει περίπου 33 δευτερόλεπτα. Αν είστε αρκετά άτυχοι ώστε η ρουτίνα σας να είναι στο $O(n^2)$, καθίστε αναπαυτικά και περιμένετε για 100 δευτερόλεπτα μέχρι να τελειώσει η εργασία. Και αν χρησιμοποιείτε κάποιον εκθετικό αλγόριθμο στο $O(2^n)$, ίσως προλαβαίνετε να φτιάξετε έναν καφέ — η ρουτίνα σας θα πρέπει να ολοκληρωθεί σε περίπου 10^{263} χρόνια. Ενημερώστε μας για την κατάληξη του σύμπαντος.

Ο συμβολισμός *O*() δεν ισχύει μόνο για το χρόνο· μπορείτε να τον χρησιμοποιείτε για να αναπαριστάτε οποιουσδήποτε άλλους πόρους οι οποίοι χρησιμοποιούνται από έναν αλγόριθμο. Για παράδειγμα, συχνά είναι χρήσιμο να μπορείτε να μοντελοποιήσετε κατανάλωση μνήμης (δείτε την Άσκηση 35 στη σελίδα 231).

Εικόνα 6.1. Χρόνοι εκτέλεσης διαφόρων αλγορίθμων



Ορισμένοι συνηθισμένοι συμβολισμοί *O*()

$O(1)$	Σταθερός (προσπέλαση στοιχείου πίνακα, απλές εντολές)
$O(\lg(n))$	Λογαριθμικός (δυαδική αναζήτηση) *[Ο συμβολισμός lg(n) είναι συντόμευση του $\log_2(n)$]*
$O(n)$	Γραμμικός (σειριακή αναζήτηση)
$O(n \lg(n))$	Χειρότερος από γραμμικό, αλλά όχι πολύ (μέσος χρόνος εκτέλεσης γρήγορης ταξινόμησης και ταξινόμησης σωρού)
$O(n^2)$	Τετραγωνικός (ταξινομήσεις επιλογής και παρεμβολής)
$O(n^3)$	Κυβικός (πολλαπλασιασμός 2 πινάκων $n \times n$)
$O(C^n)$	Εκθετικός (πρόβλημα περιπλανώμενου πωλητή, διαμερισμός συνόλου)

Εκτίμηση με την κοινή λογική

Μπορείτε να εκτιμήσετε την τάξη πολλών βασικών αλγορίθμων χρησιμοποιώντας κοινή λογική.

- **Απλοί βρόχοι.** Αν ένας απλός βρόχος εκτελείται από το 1 έως το n, ο αλγόριθμος πιθανότατα είναι τάξης $O(n)$ — ο χρόνος αυξάνει γραμμικά με το n. Στα παραδείγματα περιλαμβάνονται εξαντλητικές αναζητήσεις, η εύρεση της μέγιστης τιμής ενός πίνακα, και η δημιουργία αθροισμάτων ελέγχου.
- **Ένθετοι βρόχοι.** Αν τοποθετήσετε ένα βρόχο μέσα σε κάποιον άλλο, ο αλγόριθμός σας γίνεται $O(m \times n)$, όπου τα m και n είναι τα όρια των δύο βρόχων. Αυτό εμφανίζεται συνήθως σε απλούς αλγορίθμους ταξινόμησης, όπως η ταξινόμηση φυσαλίδας (bubble sort), όπου ο εξωτερικός βρόχος σαρώνει κάθε στοιχείο του πίνακα με τη σειρά, και ο εσωτερικός βρόχος εντοπίζει τη θέση όπου πρέπει να τοποθετηθεί το συγκεκριμένο στοιχείο στο ταξινομημένο αποτέλεσμα. Τέτοιοι αλγόριθμοι ταξινόμησης είναι συνήθως τάξης $O(n^2)$.
- **Δυαδική περικοπή.** Αν ο αλγόριθμός σας μειώνει στο μισό το σύνολο των πραγμάτων που εξετάζει σε κάθε επανάληψη ενός βρόχου, τότε πιθανότατα είναι λογαριθμικός, στο $O(\lg(n))$ (δείτε την Άσκηση 37 στη σελίδα 231). Η δυαδική αναζήτηση μιας ταξινομημένης λίστας, η διάσχιση ενός δυαδικού δέντρου, και η εύρεση του πρώτου ενεργοποιημένου bit σε μια λέξη μηχανής, μπορούν όλα να είναι στο $O(\lg(n))$.
- **Διαίρει και βασίλευε.** Οι αλγόριθμοι που διαμερίζουν την είσοδό τους, επεξεργάζονται ανεξάρτητα τα δύο μισά, και κατόπιν συνδυάζουν το αποτέλεσμα, είναι πιθανώς στο $O(n \lg(n))$. Κλασικό παράδειγμα αποτελεί ο αλγόριθμος γρήγορης ταξινόμησης (quick sort), ο οποίος χωρίζει τα δεδομένα σε δύο μισά και ταξινομεί το καθένα τους με αναδρομικό τρόπο. Αν και τεχνικά είναι στο $O(n^2)$, καθώς η συμπεριφορά του υποβαθμίζεται όταν τροφοδοτείται με ταξινομημένη είσοδο, ο μέσος χρόνος εκτέλεσης του αλγόριθμου γρήγορης ταξινόμησης είναι $O(n \lg(n))$.
- **Συνδυαστικές διαδικασίες.** Όταν οι αλγόριθμοι αρχίζουν να εξερευνούν συνδυασμούς πραγμάτων, οι χρόνοι εκτέλεσής τους μπορούν να τεθούν εκτός ελέγχου. Αυτό συμβαίνει επειδή οι συνδυασμοί περιλαμβάνουν παραγοντικά (υπάρχουν $5! = 5{\times}4{\times}3{\times}2{\times}1 = 120$ συνδυασμοί των ψηφίων από 1 έως 5). Χρονομετρήστε έναν συνδυαστικό αλγόριθμο για πέντε στοιχεία: θα χρειαστεί έξι φορές περισσότερο χρόνο

για να εκτελεστεί για έξι, και 42 φορές περισσότερο χρόνο για επτά. Στα παραδείγματα περιλαμβάνονται αλγόριθμοι για πολλά από τα γνωστά δισεπίλυτα προβλήματα — το πρόβλημα του περιπλανώμενου πωλητή, τη βέλτιστη συσκευασία πραγμάτων σε ένα δοχείο[2], το διαχωρισμό ενός συνόλου αριθμών έτσι ώστε κάθε σύνολο να έχει το ίδιο άθροισμα, κ.ο.κ. Συχνά χρησιμοποιούνται ευρετικές προσεγγίσεις (heuristics) για τη μείωση των χρόνων εκτέλεσης αλγορίθμων τέτοιου τύπου σε συγκεκριμένους τομείς εφαρμογών.

Οι ταχύτητες αλγορίθμων στην πράξη

Δεν πρόκειται να ξοδέψετε πολύ χρόνο στην καριέρα σας γράφοντας ρουτίνες ταξινόμησης. Εκείνες που παρέχονται από τις βιβλιοθήκες θα ξεπεράσουν εύκολα σε απόδοση οτιδήποτε προσπαθήσετε να γράψετε μόνοι σας χωρίς να καταβάλλετε ουσιαστική προσπάθεια. Ωστόσο, τα βασικά είδη αλγορίθμων που περιγράψαμε νωρίτερα εμφανίζονται σταθερά από καιρό σε καιρό. Κάθε φορά που γράφετε έναν απλό βρόχο, γνωρίζετε ότι έχετε έναν αλγόριθμο $O(n)$. Αν αυτός ο βρόχος περιέχει κάποιον εσωτερικό βρόχο, τότε είστε στο $O(m \times n)$. Θα πρέπει να αναρωτηθείτε πόσο μεγάλες μπορούν να γίνουν τούτες οι τιμές. Αν οι αριθμοί είναι προκαθορισμένοι, γνωρίζετε και πόσο χρόνο θα χρειαστεί η εκτέλεση του κώδικα. Αν οι αριθμοί εξαρτώνται από εξωτερικούς παράγοντες (όπως το πλήθος των εγγραφών σε μια νυχτερινή διαδικασία μαζικής ενημέρωσης, ή το πλήθος των ονομάτων σε έναν κατάλογο), ίσως είναι καλό να σταματήσετε και να εξετάσετε την επίδραση που μπορεί να έχουν οι μεγάλες τιμές στο χρόνο εκτέλεσης ή την κατανάλωση μνήμης.

ΣΥΜΒΟΥΛΗ 45

Εκτιμήστε την τάξη των αλγορίθμων σας

Υπάρχουν ορισμένες προσεγγίσεις τις οποίες μπορείτε να ακολουθήσετε για να αντιμετωπίσετε πιθανά προβλήματα. Αν έχετε έναν αλγόριθμο στο $O(n^2)$, προσπαθήστε να βρείτε μια προσέγγιση "διαίρει και βασίλευε" η οποία θα μειώσει το χρόνο στο $O(n \lg(n))$.

Αν δεν είστε βέβαιοι για το πόσο χρόνο απαιτεί ο κώδικάς σας, ή για το πόση μνήμη θα χρησιμοποιήσει, προσπαθήστε να τον εκτελέσετε, μεταβάλλοντας το πλήθος των εγγραφών εισόδου ή οτιδήποτε άλλο είναι πιθανό να επηρεάζει το χρόνο εκτέλεσης. Κατόπιν σχεδιάστε ένα γράφημα με τα αποτελέσματα.

[2] *Σ.τ.Μ.* Το γνωστό πρόβλημα του σακιδίου (knapsack problem).

Σύντομα θα πρέπει να έχετε μια καλή ιδέα για το σχήμα της καμπύλης. Ποια είναι η συμπεριφορά της καθώς αυξάνεται το μέγεθος της εισόδου; Ανεβαίνει απότομα προς τα επάνω, είναι ευθεία γραμμή, ή εξομαλύνεται; Τρία ή τέσσερα σημεία πρέπει να σας δώσουν μια ιδέα.

Επίσης αναλογιστείτε τι ακριβώς κάνετε στον ίδιο τον κώδικα. Ένας απλός βρόχος στο $O(n^2)$ ίσως να αποδίδει καλύτερα από έναν σύνθετο στο $O(n \lg(n))$ για μικρότερες τιμές του n, ιδιαίτερα αν ο αλγόριθμος στο $O(n \lg(n))$ διαθέτει κάποιον δαπανηρό από άποψη πόρων εσωτερικό βρόχο.

Στη μέση όλης τούτης της θεωρίας, μην ξεχνάτε ότι υπάρχουν και πρακτικά θέματα που πρέπει να λάβετε υπόψη. Ο χρόνος εκτέλεσης ίσως φαίνεται ότι αυξάνεται γραμμικά για μικρά σύνολα εισόδων. Αλλά αν τροφοδοτήσετε τη ρουτίνα με εκατομμύρια εγγραφές, ο χρόνος ξαφνικά θα υποβαθμιστεί καθώς το σύστημα θα αρχίσει να καταπονείται. Αν ελέγξετε μια ρουτίνα ταξινόμησης με τυχαία κλειδιά εισόδου, την πρώτη φορά που θα αντιμετωπίσει ταξινομημένη είσοδο ίσως να εκπλαγείτε. Οι Πρακτικοί Προγραμματιστές προσπαθούν να καλύψουν τόσο τις θεωρητικές όσο και τις πρακτικές βάσεις. Μετά από όλη αυτή τη θεωρία περί εκτιμήσεων, η μόνη μέτρηση που έχει σημασία είναι η ταχύτητα του κώδικά σας, όταν αυτός εκτελείται στο περιβάλλον παραγωγής με πραγματικά δεδομένα[3]. Αυτό οδηγεί στην επόμενη συμβουλή μας.

ΣΥΜΒΟΥΛΗ 46

Ελέγξτε τις εκτιμήσεις σας

Αν είναι δύσκολο να πάρετε ακριβείς χρόνους, χρησιμοποιήστε εφαρμογές *προφίλ κώδικα* (code profilers) για να μετρήσετε το πλήθος των φορών που εκτελούνται τα διαφορετικά βήματα στον αλγόριθμό σας, και δημιουργήστε ένα γράφημα με τούτα τα νούμερα σε σχέση με το μέγεθος της εισόδου.

Το καλύτερο δεν είναι πάντα το καλύτερο

Πρέπει επίσης να είστε πρακτικοί όσον αφορά την επιλογή του κατάλληλου αλγορίθμου — ο γρηγορότερος αλγόριθμος δεν είναι πάντα ο καλύτερος για μια συγκεκριμένη δουλειά. Με δεδομένο ένα μικρό σύνολο εισόδου, μια απλή ταξινόμηση εισαγωγής (insertion sort) θα αποδώσει εξίσου καλά με έναν αλ-

[3] Για την ακρίβεια, κατά τον έλεγχο των αλγορίθμων ταξινόμησης οι οποίοι χρησιμοποιήθηκαν ως ασκήσεις για την παρούσα ενότητα σε ένα Pentium με 64 MB, οι συγγραφείς εξάντλησαν την πραγματική μνήμη καθώς εκτελούσαν τον αλγόριθμο ταξινόμησης βάσης (radix sort) με περισσότερους από επτά εκατομμύρια αριθμούς. Η ταξινόμηση ξεκίνησε χρησιμοποιώντας χώρο εναλλαγής (swap space) στο δίσκο, και οι χρόνοι υποβιβάστηκαν δραματικά.

γόριθμο γρήγορης ταξινόμησης (quick sort), και θα απαιτήσει λιγότερο χρόνο για τη συγγραφή και την αποσφαλμάτωσή της. Επίσης, πρέπει να είστε προσεκτικοί σχετικά με το αν ο αλγόριθμος που επιλέγετε έχει υψηλό κόστος διευθέτησης. Για μικρά σύνολα εισόδου, η διευθέτηση ίσως να επισκιάζει το χρόνο εκτέλεσης και να καθιστά τον αλγόριθμο ακατάλληλο.

Επίσης, να προσέχετε την *πρόωρη βελτιστοποίηση* (premature optimization). Πριν επενδύσετε τον πολύτιμο χρόνο σας προσπαθώντας να βελτιώσετε έναν αλγόριθμο, είναι πάντα καλή ιδέα να έχετε πρώτα βεβαιωθεί ότι ο αλγόριθμος αποτελεί όντως πηγή καθυστέρησης.

Σχετικές ενότητες:

- *Εκτίμηση*, σελίδα 97

Προκλήσεις

- Κάθε προγραμματιστής πρέπει να έχει μια ιδέα για το πώς σχεδιάζονται και αναλύονται οι αλγόριθμοι. Ο Robert Sedgewick έχει γράψει μια σειρά προσιτών βιβλίων για το θέμα ([Sed83, SF96, Sed92] και άλλα). Συνιστούμε να προσθέσετε ένα από αυτά στη συλλογή σας, και να φροντίσετε να το διαβάσετε.
- Για όσους προτιμούν περισσότερες λεπτομέρειες από εκείνες που παρέχει ο Sedgewick, διαβάστε την απόλυτη σειρά βιβλίων *Art of Computer Programming* (Η τέχνη του προγραμματισμού υπολογιστών) του Donald Knuth, τα οποία αναλύουν ένα μεγάλο εύρος αλγορίθμων [Knu97a, Knu97b, Knu98].
- Στην Άσκηση 34 εξετάζουμε την ταξινόμηση πινάκων με μεγάλους ακέραιους. Ποιο μπορεί να είναι το αποτέλεσμα αν τα κλειδιά είναι περισσότερο σύνθετα, και η επιβάρυνση της σύγκρισής τους υψηλή; Επηρεάζει η δομή των κλειδιών την αποδοτικότητα των αλγορίθμων ταξινόμησης, ή η γρηγορότερη ταξινόμηση είναι πάντοτε η πιο γρήγορη;

Ασκήσεις

34. Έχουμε γράψει ένα σύνολο απλών ρουτινών ταξινόμησης, τις οποίες μπορείτε να "κατεβάσετε" από τον ιστότοπο μας (`http://www.pragmaticprogrammer.com`). Εκτελέστε τις σε διάφορες μηχανές που έχετε διαθέσιμες. Συμφωνούν οι αριθμοί σας με τις αναμενόμενες καμπύλες; Τι

μπορείτε να συναγάγετε για τις σχετικές ταχύτητες των μηχανών σας; Ποια είναι τα αποτελέσματα των διάφορων ρυθμίσεων βελτιστοποίησης του μεταγλωττιστή; Είναι η ταξινόμηση βάσης (radix sort) όντως γραμμική; *(Απάντηση στη σελ. 360)*

35. Η παρακάτω ρουτίνα τυπώνει τα περιεχόμενα ενός δυαδικού δέντρου. Αν θεωρήσουμε ότι το δέντρο είναι ισορροπημένο, πόσο χώρο στοίβας θα χρησιμοποιήσει κατά προσέγγιση η ρουτίνα για να τυπώσει ένα δέντρο 1.000.000 στοιχείων; (Υποθέστε ότι οι κλήσεις υπορουτινών δεν επιφέρουν σημαντική επιβάρυνση στη στοίβα.)

```
void printTree(const Node *node) {
  char buffer[1000];

  if (node) {
    printTree(node->left);
    getNodeAsString(node, buffer);
    puts(buffer);
    printTree(node->right);
  }
}
```
(Απάντηση στη σελ. 361)

36. Μπορείτε να βρείτε έναν τρόπο ώστε να μειώσετε τις απαιτήσεις στοίβας για τη ρουτίνα της Άσκησης 35 (εκτός από το να μειώσετε το μέγεθος του χώρου προσωρινής αποθήκευσης); *(Απάντηση στη σελ. 362)*

37. Στη σελίδα 227 υποστηρίξαμε ότι μια δυαδική περικοπή είναι στο $O(\lg(n))$. Μπορείτε να το αποδείξετε; *(Απάντηση στη σελ. 362)*

Ανακατασκευή

33

Αλλαγή και αποσύνθεση όπου και να κοιτάξω…

► **H. F. Lyte, "Abide With Me"**

Καθώς ένα πρόγραμμα εξελίσσεται, θα δημιουργηθεί η ανάγκη να αναθεωρήσουμε προηγούμενες αποφάσεις μας και να επεξεργαστούμε εκ νέου τμήματα του κώδικα. Αυτή η διαδικασία είναι απόλυτα φυσιολογική. Ο κώδικας πρέπει να εξελίσσεται· δεν είναι κάτι στατικό.

Δυστυχώς, η πιο συνηθισμένη μεταφορά για την ανάπτυξη λογισμικού είναι η οικοδόμηση κτηρίων (ο Bertrand Meyer [Mey97b] χρησιμοποιεί τον όρο "Οικοδόμηση λογισμικού" — software construction). Αλλά η χρήση της οικοδόμησης ως καθοδηγητικής μεταφοράς υπονοεί τα ακόλουθα βήματα:

"Αν επρόκειτο να ξεκινήσω κάποιο έργο, θα ήθελα να το αναλάβουν οι συγγραφείς αυτού του Βιβλίου... Και αν αυτό δεν γινόταν, θα αναζητούσα ανθρώπους οι οποίοι διάβασαν το βιβλίο τους.

— Ward Cunningham

Βγαλμένο κατευθείαν μέσα από την πράξη, το βιβλίο ***Ο Πρακτικός Προγραμματιστής*** προσπερνάει την όλο και περισσότερη εξειδίκευση και τις τεχνικές λεπτομέρειες της σύγχρονης ανάπτυξης λογισμικού, και εξετάζει τη βασική διαδικασία — τη μεταπροή μιας απαίτησης σε λειτουργικό, συντηρήσιμο κώδικα ο οποίος ικανοποιεί τους χρήστες του. Καλύπτει θέματα που ευμαίνονται από την προσωπική ευθύνη και την ανάπτυξη της σταδιοδρομίας μέχρι τις αρχιτεκτονικές τεχνικές οι οποίες έχουν στόχο να διατηρούν τον κώδικά σας ευέλικτο και εύκολα προσαρμόσιμο και επαναχρησιμοποιήσιμο. Διαβάστε αυτό το βιβλίο, και θα μάθετε πώς:

- Να καταπολεμάτε την αποσύνθεση λογισμικού
- Να αποφεύγετε την παγίδα της επανάληψης γνώσης
- Να γράφετε ευέλικτο, δυναμικό, και προσαρμόσιμο κώδικα
- Να αποφεύγετε τον προγραμματισμό στην τύχη
- Να προστατεύετε τον κώδικά σας με συμβόλαια (contracts), ισχυρισμούς (assertions), και εξαιρέσεις (exceptions)
- Να "συλλαμβάνετε" πραγματικές απαιτήσεις
- Να ελέγχετε "ανελέητα" και αποτελεσματικά
- Να κάνετε χαρούμενους τους χρήστες σας
- Να δημιουργείτε ομάδες πρακτικών προγραμματιστών
- Να κάνετε την εργασία σας ακριβέστερη μέσω της αυτοματοποίησης

Γραμμένο ως μια σειρά ανεξάρτητων ενοτήτων και γεμάτο με διασκεδαστικές ιστορίες, εμπνευσμένα παραδείγματα, και ενδιαφέρουσες αναλογίες, το βιβλίο ***Ο Πρακτικός Προγραμματιστής*** παρουσιάζει τις βέλτιστες πρακτικές και τις σημαντικές παγίδες πολλών διαφορετικών πτυχών της ανάπτυξης λογισμικού. Είτε είστε νέοι προγραμματιστές, είτε πεπειραμένοι, είτε μάναζερ αρμόδιοι για έργα λογισμικού, χρησιμοποιήστε τούτα τα μαθήματα στην καθημερινή σας εργασία, και θα δείτε γρήγορα βελτιώσεις σε προσωπική παραγωγικότητα, ακρίβεια, και επαγγελματική ικανοποίηση. Θα αποκτήσετε δεξιότητες και θα αναπτύξετε συνήθειες και συμπεριφορές οι οποίες θα αποτελέσουν τη βάση για μακροχρόνια επιτυχία στη σταδιοδρομία σας. Θα γίνετε ένας Πρακτικός Προγραμματιστής.

Ο Andy Hunt είναι ενθουσιώδης ξυλουργός και μουσικός, αλλά, περιέργως, έχει μεγαλύτερη ζήτηση ως σύμβουλος επιχειρήσεων. Έχει εργαστεί σε τηλεπικοινωνίες, τραπεζικούς τομείς, οικονομικές υπηρεσίες, και κοινωφελείς οργανισμούς, καθώς επίσης και σε πιο εξωτικούς τομείς, όπως ιατρική απεικόνιση, γραφικές τέχνες, και διαδικτυακές υπηρεσίες. Ο Andy ειδικεύεται στο συνδυασμό δοκιμασμένων τεχνικών με τεχνολογίες αιχμής, δημιουργώντας πρωτοποριακές αλλά εφαρμόσιμες λύσεις. Διαθέτει τη δική του επιχείρηση παροχής συμβουλευτικών υπηρεσιών στο Raleigh της Βόρειας Καρολίνας.

Στον Dave Thomas αρέσουν οι πτήσεις με μονοκινητήρια αεροπλάνα, ένα ακριβό χόμπι το οποίο πληρώνει επινοώντας κομψές λύσεις σε δύσκολα προβλήματα, παρέχοντας συμβουλές σε τομείς τόσο ποικίλους όσο η αεροπορική βιομηχανία, οι τραπεζικές υπηρεσίες, οι οικονομικές υπηρεσίες, οι τηλεπικοινωνίες, τα ταξίδια και οι μεταφορές, και το Διαδίκτυο. Πριν μετακομίσει στις ΗΠΑ το 1994, ο Dave είχε ιδρύσει μια αγγλική εταιρεία λογισμικού με πιστοποίηση ISO9001 η οποία παρέδιδε εξελιγμένα, εξατομικευμένα έργα λογισμικού σε όλο τον κόσμο. Ο Dave σήμερα είναι ανεξάρτητος σύμβουλος στο Ντάλας του Τέξας.

Ο Dave και ο Andy εργάζονται τώρα μαζί στην εταιρεία The Pragmatic Programmers, LLC., προσφέροντας περισσότερα από σαράντα χρόνια συνδυασμένης πείρας σε πελάτες από κάθε γωνιά των ΗΠΑ.

ISBN 978-960-461-135-5

εκδόσεις ΚΛΕΙΔΑΡΙΘΜΟΣ

Δαμασκού 4, Στάθμος Λαμπίρη, 10440 ΑΘΗΝΑ, Τηλ. 210-5237685

Επισκεφθείτε μας στο Internet:
www.klidarithmos.gr