ΗΥ352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

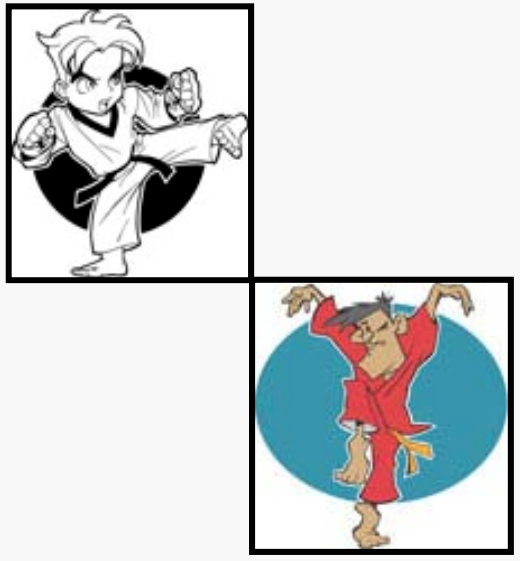
ΔΙΔΑΣΚΩΝ

Αντώνιος Σαββίδης

ΕΝΟΤΗΤΑ 7

ΑΜΥΝΤΙΚΟΣ ΠΡΟΡΓΡΑΜΜΑΤΙΣΜΟΣ

Αριθμός διαλέξεων 1 - Διάλεξη 1η



Περιεχόμενα

- *Εισαγωγή - αμυντικός προγραμματισμός*
- Resource failure (αποτυχία πόρων)
- Bug (προγραμματιστικό σφάλμα)
- Κοινή στρατηγική debugging
- Αυτοέλεγχος προγράμματος
- Ακραίος προγραμματισμός

Εισαγωγή (1/3)

- Ο αμυντικός προγραμματισμός είναι μία τακτική δημιουργίας πηγαίου κώδικα η οποία βασίζεται στην αρχή ότι «*κάθε νέο τμήμα κώδικα φέρει ταυτόχρονα λύσεις αλλά και προβλήματα*»
  - *Η μόνη σωστή γραμμή κώδικα είναι η κενή γραμμή κώδικα*
- Είναι μία προσέγγιση η οποία μπορεί να χρησιμοποιηθεί ανεξαρτήτως του τι κάνει ο κώδικας που λύνει το εκάστοτε πρόβλημα. Συνεπώς μπορεί να συνδυαστεί με τις μεθόδους και τεχνικές διαφόρων πεδίων εφαρμογών
- Βασίζεται στην παρατήρηση ότι οι προγραμματιστές κατά την σύνταξη του κώδικα επικεντρώνονται περισσότερο στο να «*λύσουν το πρόβλημα*», τείνοντας να ξεχάσουν ότι μπορούν εύκολα να «*καταστρέψουν τη λύση*»

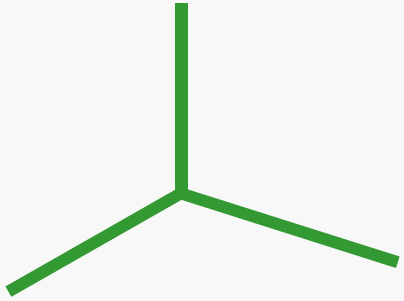
Εισαγωγή (2/3)

- Είναι δεξιότητα η οποία καλλιεργείται με το χρόνο καθώς οι προγραμματιστές έρχονται αντιμέτωποι με καταστροφικές καταστάσεις που οι οποίες προκαλούνται από τους ίδιους, υποφέροντας από:
  - μεταμεσονύκτιες υπερωρίες, αργοπορημένα χρονοδιαγράμματα, προσωρινή απώλεια αυτοπεποίθησης, μειωμένη προσωπική ζωή, πιθανότατα κλονισμένα νεύρα, απώλεια μισθού
- Εξελίσσεται σε ένα ισχυρό οπλοστάσιο το οποίο, καθώς οι προγραμματιστές ωριμάζουν, περιέχει ώριμες διαγνωστικές μεθόδους - *bug detection*, τακτικές εντοπισμού - *bug hunting* - και θεραπείας - *bug repair*.
- Είναι απαραίτητο συστατικό στοιχείο αλλά και κοσμοθεωρία για όσους φιλοδοξούν να ανέβουν στην κλίμακα κατασκευής μεγάλων συστημάτων

Εισαγωγή (3/3)

Οι βασικές προγραμματιστικές δεξιότητες για υψηλού επιπέδου τεχνολογία λογισμικού....
or "the key to programming paradise"

Εξέλιξη σχεδίασης με δυνατότητα αποτελεσματικής διαχείρισης της εντροπίας



Υψηλής ποιότητας αμυντικός προγραμματισμός με εφαρμογή των κατάλληλων τεχνικών και ανάπτυξη διαγνωστικών εργαλείων

Σχεδίαση βασισμένη σε σχεδιαστικά πρότυπα με έμφαση στην επαναχρησιμοποίηση και εξάλειψη επαναλαμβανόμενου κώδικα με γενικό προγραμματισμό

Περιεχόμενα

- Εισαγωγή - αμυντικός προγραμματισμός
- *Resource failure (αποτυχία πόρων)*
- Bug (προγραμματιστικό σφάλμα)
- Κοινή στρατηγική debugging
- Αυτοέλεγχος προγράμματος
- Ακραίος προγραμματισμός

Αποτυχία πόρων (1/4)

- Σε αντίθεση με τα σφάλματα, τέτοιου δυσλειτουργίες αναμένονται φυσιολογικά και μπορεί να προκαλούνται από:
  - προβλήματα εξωτερικών πόρων:
    - *hard disc overflow*
    - *no more memory*
    - *loss of network connection*
  - λάθος είσοδο σε λειτουργίες του συστήματος από εξωτερικές πηγές:
    - *«χαλασμένα» αρχεία δεδομένων*
    - *λάθος είσοδος από τον χρήστη*
    - *λάθος requests από clients*
- *Η αδυναμία του προγράμματος να να αντιδράσει ορθά σε τέτοιού είδους αναμενόμενα προβλήματα είναι μία πολύ κοινή περίπτωση προγραμματιστικού σφάλματος*

Αποτυχία πόρων (2/4)

- Όλες οι πιθανές περιπτώσεις αποτυχίας πόρων πρέπει να ταξινομηθούν και να τεκμηριωθούν
  - Ορισμένες τέτοιες αποτυχίες μπορεί να χαρακτηριστούν ως «καταληκτικές» - *fatal* – εάν αποτυχαίνουν ζωτικοί πόροι, που σημαίνει ότι το πρόγραμμα πρέπει αναγκαστικά να τερματίσει, όπως π.χ.
    - *λάθος δίσκου αποθήκευσης σε μία βάση δεδομένων*
    - *stack overflow (program execution)*
    - *αδυναμία επικοινωνίας με την κάρτα γραφικών σε εφαρμογή multimedia*
    - *αποτυχία δικτύου σε μία ftp εφαρμογή*
  - Για τις «μη καταστροφικές» αποτυχίες θα πρέπει ορίσουμε ακριβώς τον τρόπο με τον οποίο το σύστημα συμπεριφέρεται ώστε να αντεπεξέλθει, π.χ. την όποια ειδική κατάσταση στην οποία υπεισέρχεται
  - Το λογισμικό πρέπει να σχεδιάζεται και να υλοποιείται ώστε όλες αυτές οι περιπτώσεις να αντιμετωπίζονται «ανηλεώς» και συστηματικά

Αποτυχία πόρων (3/4)

- Μία αποτυχία πόρων μπορεί να συμβεί ενώ ο έλεγχος βρίσκεται στη μέση κάποιας εσωτερικής επεξεργασίας. Για να απεμπλακεί το πρόγραμμα θα πρέπει να:
  - *επιστρέψει από αρκετές συναρτήσεις*
  - *απελευθερώσει όσους πόρους παραχωρήθηκαν ενδιάμεσα*
  - *να μεταδώσει τον κωδικό και την πληροφορία της αποτυχίας*
  - *να περιέλθει σε μία τέτοια κατάσταση στην οποία η λειτουργία που απέτυχε να έχει πλήρως ακυρωθεί*
- Όταν συναρτήσεις εμπλέκονται σε τέτοιες καταστάσεις θα πρέπει να υλοποιούνται με τρόπο που όλες οι περιπτώσεις λαθών ελέγχονται με κατάλληλες συνθήκες ενώ τα λάθη μεταδίδονται στον caller είτε με τη μορφή επιστρεφόμενων τιμών (παλαιός τρόπος;) η μέσω διαχείρισης exceptions (νέος τρόπος;)

Αποτυχία πόρων (4/4)

- Όταν επιστρέψουμε με επιτυχία μετά τον εντοπισμό αποτυχίας πόρων, θα πρέπει να γίνει περαιτέρω αναφορά του συμβάντος. Αυτό μπορεί να σημαίνει ότι θα επιστρέψουμε όλη την πληροφορία στον αρχικά αιτούμενο την λειτουργία, που μπορεί να είναι:
  - *ένα άλλο sub-system*
  - *μία άλλη process*
  - *ο χρήστης*
- Η C++ προσφέρει ένα σχετικά απλό εγγενή μηχανισμό για exception handling, με κλάσεις exceptions που ορίζονται από τον προγραμματιστή

Περιεχόμενα

- Εισαγωγή - αμυντικός προγραμματισμός
- Resource failure (αποτυχία πόρων)
- *Bug (προγραμματιστικό σφάλμα)*
- Κοινή στρατηγική debugging
- Αυτοέλεγχος προγράμματος
- Ακραίος προγραμματισμός

Προγραμματιστικό σφάλμα (1/6)

- *Ένα σφάλμα (bug) είναι μία εντολή (offensive stmt) που φέρει το πρόγραμμα σε κατάσταση διαφορετική από αυτές που συνεπάγονται ορθή λειτουργία*
- Μετά την γέννηση ενός bug η παρατηρούμενη συμπεριφορά του προγράμματος *ενδέχεται* να είναι λανθασμένη
- Ένα bug μπορεί να οφείλεται σε:
  - *αποτυχίες πόρων που δεν αντιμετωπίζονται*
  - *λάθη αλγοριθμικής σχεδίασης (π.χ. δεν προβλέπονται κάποια σενάρια εκτέλεσης)*
  - *λάθη πληκτρολόγησης (ακούσια αλλαγή κώδικα)*
  - *κακή εκτίμηση ότι ο κώδικας κάνει κάτι το οποίο στην πράξη δεν κάνει (χειρότερα, κάνει κάτι το οποίο δεν περιμένουμε)*

Προγραμματιστικό σφάλμα (2/6)

- Ο τρόπος με τον οποίο τελικά αντιλαμβανόμαστε την παρουσία ενός σφάλματος (*bug symptoms*) ποικίλει:
  - Ως αδικαιολόγητη αποτυχία πόρων (το σφάλμα «προσβάλει» κάποιος πόρους, προκαλώντας αποτυχία κατά τη χρήση τους, ακόμη και εάν στην πραγματικότητα θα έπρεπε να είναι διαθέσιμοι, π.χ.
    - *file write error, ενώ το όνομα του αρχείου είναι σωστό και υπάρχει χώρος στο δίσκο*
    - *memory allocation error, ενώ προφανώς η δυναμική μνήμη δεν έχει εξαντληθεί*
  - Μέσω διαγνωστικού ή αμυντικού ενσωματωμένου κώδικα ο οποίος εντοπίζει περιπτώσεις λανθασμένης κατάστασης αντικειμένων, χωρίς ωστόσο να εντοπίζεται και η αιτία του προβλήματος
  - Μέσω αποτυχιών πόρων η διαγνωστικών μηνυμάτων από άλλα sub-systems ή run-time libraries
  - Λόγω λανθασμένης συμπεριφοράς και εξόδου των λειτουργιών του συστήματος
  - Από ένα (αναπάντεχο;) system crash

Προγραμματιστικό σφάλμα (3/6)

- Χρονική απόσταση σφάλματος – *bug time distance*
  - *Ορίζεται ως ο χρόνος που μεσολαβεί κατά την εκτέλεση από την γέννηση του σφάλματος, έως το σημείο που γίνεται αντιληπτή η ύπαρξή του*
    - Όσο μεγαλύτερος είναι αυτός ο χρόνος, τόσο δυσκολότερος είναι ο εντοπισμός και προσδιορισμός της πραγματικής αιτίας

Προγραμματιστικό σφάλμα (4/6)

- Χωρική απόσταση σφάλματος – *bug source distance*
  - *Ορίζεται άτυπα ως η «απόσταση» μεταξύ του σημείου του κώδικα στο οποίο γεννιέται το σφάλμα, και το σημείο στο οποίο γίνεται για πρώτη φορά αντιληπτό – εκεί που χτυπάει το σφάλμα*
    - Αυτή η μετρική δεν έχει ιδιαίτερη αξία πέραν του ότι χρησιμοποιείται σε διαφωνίες μεταξύ των προγραμματιστών για το ποιος ευθύνεται για ένα bug το οποίο μόλις βγήκε στην επιφάνεια:
      - Ενώ το σύμπτωμα εμφανίζεται σε ένα σημείο, δεν είναι απαραίτητο να φταίει ο προγραμματιστής που υλοποιεί αυτόν τον κώδικα
      - Δεν χρειάζεται εφησυχασμός όταν το bug εμφανίζεται σε «αρκετή απόσταση» από τον κώδικά σας
    - Η απόσταση μπορεί να ορίζεται ως μεγαλύτερη καθώς προχωράμε σε: συνεχόμενες εντολές, blocks, συναρτήσεις, τμήματα, υποσυστήματα.

Προγραμματιστικό σφάλμα (5/6)

- Ο τρόπος που ένα σφάλμα αναπαράγεται (*bug reproducibility*) σε διαφορετικές εκτελέσεις του προγράμματος ποικίλει:
  - Μπορεί να εμφανίζεται πάντα μετά από κάποια συγκεκριμένα στάδια επεξεργασίας, όμως η χρονική απόσταση να ποικίλει (δηλ. παρατηρούνται συμπτώματα σε διαφορετικά σημεία κάθε φορά)
  - Μπορεί να εμφανίζεται μόνο σε μία συγκεκριμένη ακολουθία εκτέλεσης εντολών η οποία και πρέπει να αναπαράγεται ακριβώς για την παρατήρηση των συμπτωμάτων
    - σε μερικές περιπτώσεις, η επανάληψη μία ακολουθίας εκτέλεσης μπορεί να είναι πολύ δύσκολο να επιτευχθεί

Προγραμματιστικό σφάλμα (6/6)

- Συνήθως σφάλματα με κανονική (μη μεταβλητή) συμπεριφορά αναπαραγωγής κατά την εκτέλεση είναι ευκολότερο να αντιμετωπιστούν, παρά σφάλματα με μικρή συχνότητα εμφάνισης
- Συνήθως σφάλματα με μεταβλητή χρονική και χωρική απόσταση είναι δυσκολότερο να αντιμετωπιστούν
- Η μεταβλητή χρονική απόσταση με σταθερή χωρική απόσταση μπορεί να δώσει πολύτιμη πληροφορία για το είδος του σφάλματος
  - π.χ. εάν τα συμπτώματα εμφανίζονται πάντα σε ένα υποσύστημα υποθέτουμε ότι ή έχει γίνει πλάγια «καταστροφή» σε αυτό, ή ότι υπάρχει εγγενές στο υποσύστημα λάθος

Περιεχόμενα

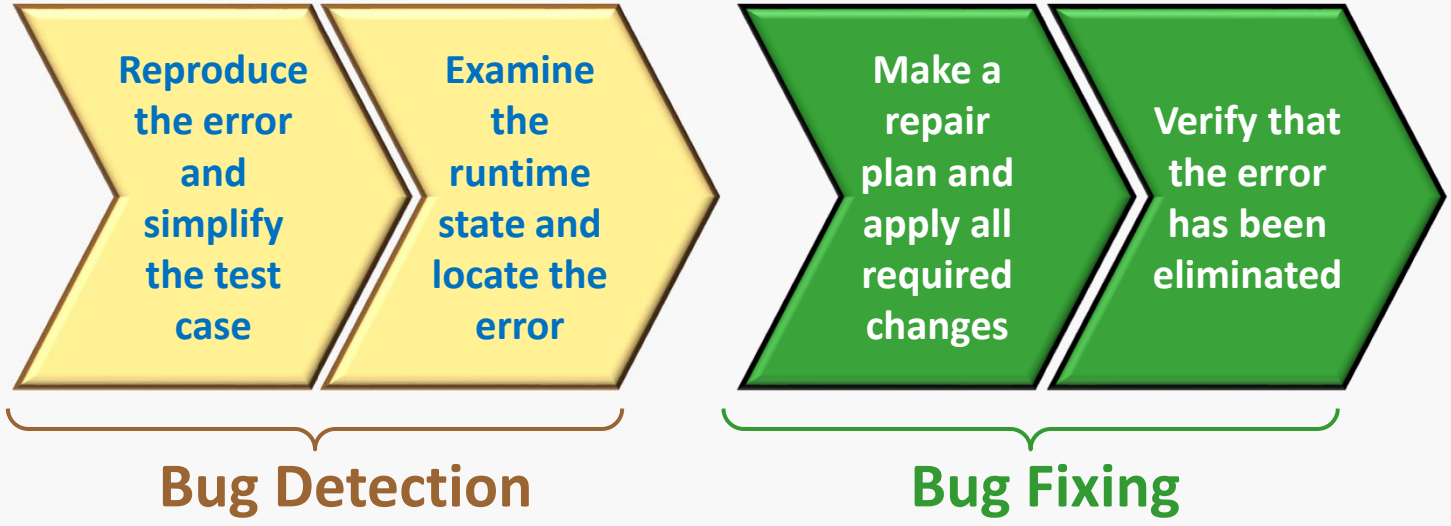
- Εισαγωγή - αμυντικός προγραμματισμός
- Resource failure (αποτυχία πόρων)
- Bug (προγραμματιστικό σφάλμα)
- *Κοινή στρατηγική debugging*
- Αυτοέλεγχος προγράμματος
- Ακραίος προγραμματισμός

Κοινή στρατηγική debugging (1/5)

- Όταν η χρονική απόσταση είναι μηδέν, τότε και η χωρική είναι μηδέν, δηλ. το σφάλμα εμφανίζεται στο σημείο γέννησής του
- Τέτοια σφάλματα *άμεσης εμφάνισης* θεωρούνται ως οι καλύτερες περιπτώσεις (και συνήθως τετριμμένες)
- Οι «καλοί compilers» περιέχουν run-time libraries προστασία μνήμης ώστε να εξασφαλίσουν άμεση εμφάνιση σε περίπτωση παράνομης πρόσβασης μνήμης
  - *Αλλά δεν εντοπίζουν πάντα την προγραμματιστικά λανθασμένη πρόσβαση, αλλά την λανθασμένη πρόσβαση σε επίπεδο διεργασίας (π.χ. writing read only memory)*

Κοινή στρατηγική debugging (2/5)

- Συνεπώς η πρόκληση είναι το «κυνήγι» σφαλμάτων με μη-μηδενική χρονική απόσταση, και δυσκολότερα, με μη κανονική εμφάνιση
- Τα δύο βασικά βήματα για όλες τις συστηματικές στρατηγικές αντιμετώπισης σφαλμάτων είναι:

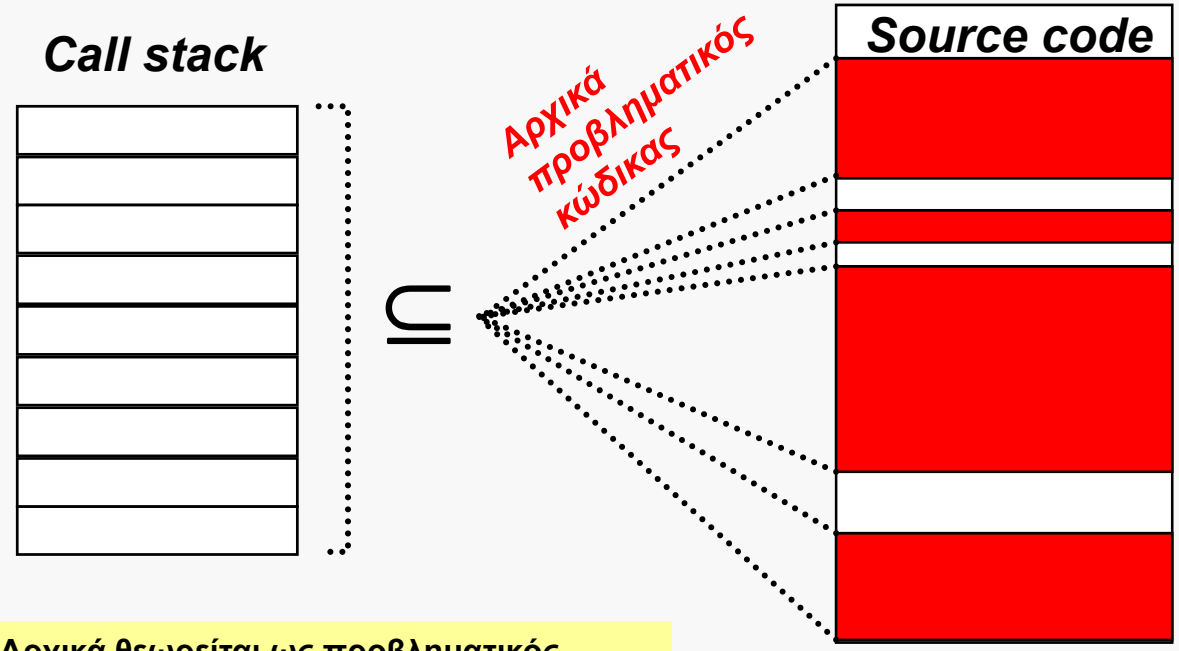


Κοινή στρατηγική debugging (3/5)

- Αυτή είναι η διαδικασία με την οποία προσπαθούμε να χαρακτηρίσουμε τμήματα κώδικα ως επιθετικά (offensive) θεωρώντας ότι το σφάλμα γεννιέται από την εκτέλεσή τους
- Βασίζεται στην τροποποίηση κώδικα και στην παρατήρηση με συστηματικό τρόπο, ο οποίος μπορεί να δουλεύει καλά για αρκετές περιπτώσεις
  - *αλλά δεν μπορεί να εντοπίσει όλων των ειδών τα σφάλματα χωρίς την εφαρμογή ειδικών διαγνωστικών τεχνικών*

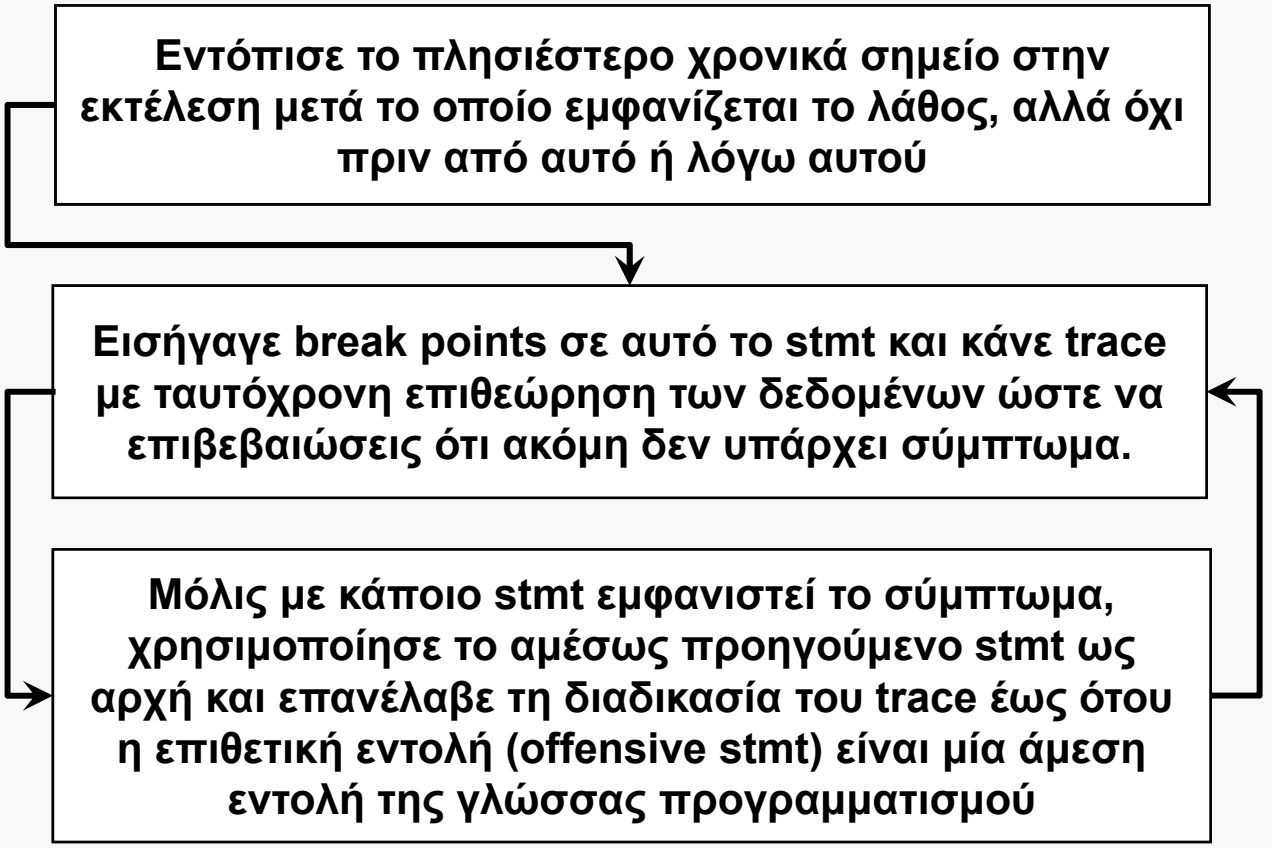
Κοινή στρατηγική debugging (4/5))

ΒΗΜΑ 1



Αρχικά θεωρείται ως προβληματικός όλος ο κώδικας που έχει εκτελεστεί μέχρι την εμφάνιση του σφάλματος (και όχι μόνο οι ενεργές κλήσεις της στοίβας). Κυρίως όμως ο κώδικας που τελευταία είχε πρόσβαση στα προσβεβλημένα δεδομένα.

Κοινή στρατηγική debugging (5/5)



ΒΗΜΑ 2

Περιεχόμενα

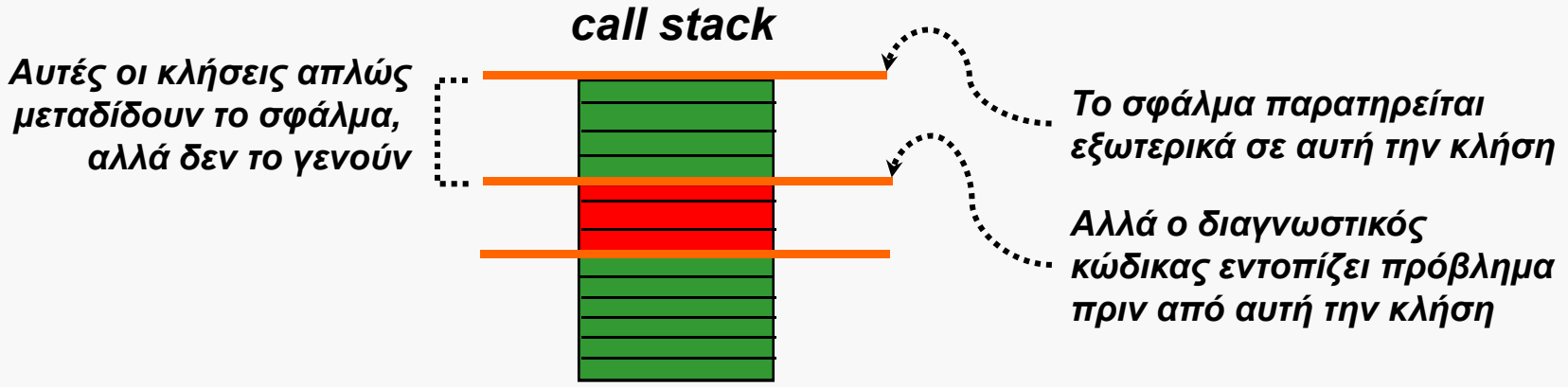
- Εισαγωγή - αμυντικός προγραμματισμός
- Resource failure (αποτυχία πόρων)
- Bug (προγραμματιστικό σφάλμα)
- Κοινή στρατηγική debugging
- *Αυτοέλεγχος προγράμματος*
- Ακραίος προγραμματισμός

Αυτοέλεγχος προγράμματος (1/8)

- Πολύ συχνά ο έλεγχος «με το μάτι» για τυχόν προσβολή δεδομένων δεν είναι πρακτικά εφικτός λόγω της μεγάλης ποσότητας δεδομένων που διαχειρίζεται το πρόγραμμα
- Σε αυτή την περίπτωση είναι αναγκαία η υιοθέτηση ειδικού κώδικα *διάγνωσης και επαλήθευσης*
  - Εάν δεν εφαρμόζετε ήδη τεχνικές «αυτοελέγχου» ή εάν επιδιορθώνετε ξένο κώδικα, θα πρέπει να κάνετε τέτοιες παρεμβολές επιπρόσθετα
    - *ενέσεις επαλήθευσης προγράμματος – program validation injections*

Αυτοέλεγχος προγράμματος (2/8)

- Ο ρόλος τέτοιων αυτόματων έξυπνων ελέγχων είναι να εντοπίζουν την ύπαρξη τους σφάλματος πολύ νωρίτερα από την παρατηρούμενη εμφάνιση του
  - δηλ. να μειώνουν δραστικά την χρονική απόσταση, και συνεπώς τον αρχικά προβληματικό κώδικα
- Αυτοί οι πολύτιμοι διαγνωστικοί έλεγχοι ονομάζονται *αισθητήρες σφαλμάτων – bug sensors*



Αυτοέλεγχος προγράμματος (3/8)

- Το πρόγραμμα θα πρέπει να περιέχει επαλήθευση ορθότητας λειτουργιών και στιγμιότυπων ώστε να εξαλείφεται η μετάδοση του σφάλματος και να εξασφαλίζεται ο *άμεσος εντοπισμός σφαλμάτων - direct defect detection - DDD*
- *Έλεγχος του class invariant*
  - Πιστοποιεί την ορθότητα στιγμιότυπου πριν και μετά την κλήση μελών - συναρτήσεων
- *Έλεγχος των παραμέτρων σε συναρτήσεις*
  - Επαληθεύει τη σημασιολογική ορθότητα των πραγματικών παραμέτρων
- *Έλεγχος του precondition συναρτήσεων*
  - Πιστοποιεί την δυνατότητα κλήσης συνάρτησης

Αυτοέλεγχος προγράμματος (4/8)

- Τα assertions ελέγχουν συνθήκες οι οποίες πρέπει να είναι πάντα true, αλλιώς,
  - *ή κάποιο σφάλμα έχει ήδη γεννηθεί,*
  - *ή ένα σφάλμα θα προκληθεί από τις εντολές που ακολουθούν το assertion*
- Τα assertions πρέπει να χρησιμοποιούνται μόνο για σφάλματα, και όχι για τις φυσιολογικά αναμενόμενες αποτυχίες πόρων (αυτές θα πρέπει να εντοπίζονται και να διαχειρίζονται κατάλληλα).
- Επίσης τοποθετούμε assertions σε σημεία του κώδικα που δεν αναμένουμε ποτέ να εκτελεστούν, ως *φύλακες της ροής ελέγχου - control flow guards*
  - Τα περίφημα ***no mans land*** stmts με `assert(false);`

Αυτοέλεγχος προγράμματος (5/8)

- *Τα assertions*
  - δεν ενσωματώνουν *ποτέ εκφράσεις οι οποίες αλλάζουν την κατάσταση του προγράμματος* (δηλ. βάζουμε μόνο observer calls, που είναι “read only” κώδικας)
  - δεν θεωρούνται *ποτέ ως τμήμα της συνολικής λογικής* του τελικού προγράμματος
  - *απενεργοποιούνται από τον compiler* (δηλ. σαν να «αφαιρούνται») στον κώδικα του production / release mode
- Τοποθετούμε assertions στον κώδικα *όσο το δυνατόν νωρίτερα*, την στιγμή που ακριβώς υλοποιείται, και όχι σε μεταγενέστερο στάδιο ως προσθήκη
  - αλλιώς θα πρέπει να τα εισάγετε αναγκαστικά κατά τη διάρκεια του debugging, όταν θα έχετε ήδη λησμονήσει τις λεπτομέρειες του κάθε τμήματος κώδικα

Αυτοέλεγχος προγράμματος (6/8)

- *Πριν το λογισμικό παραδοθεί προς πραγματική χρήση:*
  - Αφαιρούμε τους εντατικούς ελέγχους που μπορεί να μειώσουν σοβαρά την απόδοση
    - `DEBUG_ONLY_ASSERT(exprt), RETAINED_ASSERT(expr)`
  - Μετατρέπουμε τα assertions (με χρήση ειδικών macros) σε κωδικοποιημένη πληροφορία με ενσωματωμένο αριθμό αρχείου, και γραμμή κώδικα
  - Αντικαθιστούμε τα μηνύματα που βγάζει ο compiler για τα assertions με φιλικά μηνύματα με αριθμητικούς κώδικες λαθών που θα επιστραφούν από τους πελάτες σε εσάς:
    - *πληροφορούμε τον χρήστη για την ύπαρξη «καταληκτικού» σφάλματος*
    - *ένα υπάρχουν μη σωσμένα δεδομένα, προσπαθούμε να τα αποθηκεύσουμε σε προσωρινή περιοχή*
    - *και κάνουμε exit*

Αυτοέλεγχος προγράμματος (7/8)

Παράδειγμα

```
#ifdef DEBUG_MODE
#define RETAINED_ASSERT assert
#else
#define RETAINED_ASSERT(expr) \
   FatalError(__FILE__, __LINE__)
extern void FatalError (const char* file, unsigned line);
#endif

// FatalError.cpp
void FatalError (const char* file, unsigned line) {
   const char* msg = MakeEncodedErrorMessage(file, line);
   ShowMessage(msg);
   SaveRemainingData();
   UrgentExit();
}
```

Αυτοέλεγχος προγράμματος (8/8)

- Αντιπροσωπευτικές τακτικές αυτοελέγχου είναι:
  - Κατασκευή *ειδικού memory manager*
    - Απαιτεί υπερφόρτωση των τελεστών new και delete για ειδική διαχείριση μνήμης με στόχο:
      - Την επαλήθευση δεικτών και διευθύνσεων δυναμικής μνήμης
      - Την επαλήθευση της νόμιμης χρήσης δυναμικής μνήμης
  - Πλήρεις ή «μερικές» υλοποιήσεις του *design by contract* για τις κλάσεις του προγράμματος
  - Χρήση *έξυπνων δεικτών - smart pointers*
    - ειδικές κλάσεις για δείκτες με ενσωματωμένη δυνατότητα garbage collection, και με συνήθως πλήρη και προστατευμένη υποστήριξη όλων των εγγενών τελεστών που επιτρέπονται σε δείκτες.

Περιεχόμενα

- Εισαγωγή - αμυντικός προγραμματισμός
- Resource failure (αποτυχία πόρων)
- Bug (προγραμματιστικό σφάλμα)
- Κοινή στρατηγική debugging
- Αυτοέλεγχος προγράμματος
- *Ακραίος προγραμματισμός*

Ακραίος προγραμματισμός (1/10)

- Βασίζεται στην φιλοσοφία του test first programming, σύμφωνα με την οποία:
  - το λογισμικό υλοποιείται αυξητικά μέσω μικρών τμημάτων,
  - για κάθε νέο τμήμα κώδικα κατασκευάζουμε εξειδικευμένα ελεγκτικά προγράμματα (tests) για την πιστοποίηση της προστιθέμενης λειτουργικότητας
  - κάθε φορά που προστίθεται νέος κώδικας
    - έχουμε έτοιμα tests ενσωμάτωσης (integration) για να πιστοποιείται ότι το αυξημένο σύστημα συμπεριφέρεται βάσει των προδιαγραφών
    - και επίσης ελέγχουμε ότι όλα τα προηγούμενα tests περατώνονται επιτυχώς
- Δίνεται έμφαση στον έλεγχο τμημάτων - *unit testing*, παρά στο γενικό έλεγχο ενός ολοκληρωμένου συστήματος

Ακραίος προγραμματισμός (2/10)

- *Είναι μια προσέγγιση στην ανάπτυξη λογισμικού η οποία επιτρέπει στο λογισμικό να*:
  - εξελίσσεται παράλληλα με τις μεταβαλλόμενες προδιαγραφές
  - μπορεί να αντεπεξέλθει εύκολα στις αλλαγές
  - διατηρεί τη σχεδίαση όσο το δυνατόν πιο απλή
  - καθιστά τους προγραμματιστές ενεργούς και ενήμερους για την διαδικασία και τα παράγωγά της
  - υποστηρίζει την ευρεία διάχυση της γνώσης μέσα στην ομάδα ανάπτυξης
  - επικεντρώνεται στην ομαδική εργασία
  - εστιάζεται από πολύ νωρίς στη συνεχή διαδικασία ελέγχου

Ακραίος προγραμματισμός (3/10)

- *Extreme programming*
  - `http://www.extremeprogramming.org/`

Ακραίος προγραμματισμός (4/10)

- ***Τμηματικός έλεγχος – unit testing and unit tests***
  - Προγραμματίζουμε γρηγορότερα σε μικρότερα βήματα, ο κώδικας έχει υψηλότερη ποιότητα, και υπάρχει πολύ λιγότερο άγχος και πίεση
  - Μπορεί να εγκλωβίσει τα λάθη ενοποίησης
  - Μεταφέρουν τους στόχους της σχεδίασης ανεξαρτήτως λεπτομερειών υλοποίησης
  - Βοηθούν στο re-factoring, οδηγούν σε απλή σχεδίαση και ενθαρρύνουν την επικοινωνία μεταξύ των προγραμματιστών

Ακραίος προγραμματισμός (5/10)

- ***Δημιουργική αναδιάρθρωση - refactoring***
  - Ποτέ δεν αφήνουμε επαναλαμβανόμενο κώδικα
  - Ποτέ δεν εισάγουμε κώδικα εκτός των ορίων και αναγκών της σχεδίασης
  - Ποτέ δεν αφήνουμε αχρησιμοποίητο κώδικα
  - Ποτέ δεν αφήνουμε κώδικα ο οποίος δεν αντικατοπτρίζει βέλτιστη σχεδίαση

Ακραίος προγραμματισμός (6/10)

- ***Απλή σχεδίαση – simple design***, βάσει του κανόνα ότι η σωστή σχεδίαση για το σύστημα σε κάθε χρονική στιγμή είναι αυτή που:
  - τρέχει επιτυχώς όλα tests
  - λέει ότι μόνο χρειάζεται να πει, και μόνο μία φορά
  - λέει τα πάντα μόνο μία φορά
  - βάσει αυτών των περιορισμών περιέχει τις λιγότερες δυνατές κλάσεις και συναρτήσεις
    - αρκεί να μην ξεχνάμε ότι έχουμε βέλτιστη σχεδίαση, με δυνατότητα αντοχής σε αλλαγές, και ελάχιστη αύξηση εντροπίας
  - εν δυνάμει δε χρειάζεται ποτέ σχόλια

Ακραίος προγραμματισμός (7/10)

- ***Σχεδιαστική μεταφορά – design metaphors***
  - Το σύστημα χτίζεται γύρω από μία, ή ένα μικρό σύνολο, συνεργαζόμενων σχεδιαστικών μεταφορών από τις οποίες ονομασίες κλάσεων, μεθόδων, μεταβλητών, και λειτουργικών ρόλων εξάγονται
  - Δεν χρειάζεται κάποιος να εφευρίσκει ονομασίες εφαρμόζοντας αμφιβόλου ποιότητας ευρεστικούς τρόπους
  - Όλοι είναι σίγουροι ότι κατανοούν τα πρώτα πράγματα που πρέπει να γίνουν στην ανάπτυξη
  - Υπάρχει μία δύναμη που τείνει να ενοποιεί το σύστημα, και η οποία καθιστά ευκολότερη την κατανόηση του συστήματος από νέα μέλη της ομάδας ανάπτυξης. Ο τρόπος επέκτασης της σχεδίασης είναι συνήθως ξεκάθαρος

Ακραίος προγραμματισμός (8/10)

- ***Συνεχής ενσωμάτωση – continuous integration***
  - Ελεγμένες προσθήκες στον κώδικα, και μετατροπές ενσωματώνονται στην βασική δομή μετά από μερικές ώρες, το πολύ σε μία μέρα
  - Όταν μία εργασία περατωθεί, περιμένεις τη σειρά σου στη διαδικασία ενσωμάτωσης, έπειτα φορτώνεις τις αλλαγές σου πάνω στην εκάστοτε τρέχουσα δομή (λύνοντας τις όποιες ασυμφωνίες), και τρέχεις όλα τα ελεγκτικά προγράμματα
  - Εάν κάποια tests αποτυχαίνουν, θα πρέπει να εντοπίσεις τα σφάλματα και να κάνεις τις επιδιορθώσεις πριν θεωρήσεις την ενσωμάτωση τελική
  - Εάν δεν μπορούν να λυθούν τα σφάλματα, απορρίπτεις τον κώδικα που ενσωμάτωσες και αρχίζεις από την αρχή

Ακραίος προγραμματισμός (9/10)

- ***Προγραμματισμός σε ζευγάρια – pair programming***
  - Όλος ο κώδικας που πρόκειται να ενσωματωθεί στην έκδοση παραγωγής δημιουργείται από «ζευγάρια» προγραμματιστών που δουλεύουν μαζί στον ίδιο υπολογιστή
  - Τα «ζευγάρια» κινούνται σε διαφορετικά θέματα και ανασυντίθενται πολύ συχνά (δύο, τρεις, και τέσσερις φορές την ημέρα), έτσι ώστε οποιαδήποτε σημαντική πληροφορία να είναι πολύ σύντομα γνωστή σε κάθε μέλος της ομάδας ανάπτυξης
    - Αυτή είναι μία καλή τακτική για τις εταιρείες ώστε στην πράξη να εξασφαλίζεται το «ουδείς αναντικατάστατος»
    - Στο τέλος τείνει να υπάρχει μία ομοιογενής ομάδα με παρόμοιες γνώσεις και μικρό κίνδυνο κλονισμού της ομάδας με την αποχώρηση κάποιου μέλους, καθώς και με μεγάλη ταχύτητα ενσωμάτωσης και εκπαίδευσης νέων μελών
    - *Φαίνεται να είναι καλή τακτική μόνο για βιομηχανική παραγωγή, και λιγότερο για έρευνα και καινοτομία*

Ακραίος προγραμματισμός (10/10)

- ***Εβδομάδα σαράντα ωρών – forty hours week***
  - Πήγαινε σπίτι στις 17:00. Καλό σαββατοκύριακο.
  - Μία ή δύο φορές το χρόνο μπορεί να κάνεις υπερωρίες το σαββατοκύριακο
    - αλλά η ανάγκη για ένα δεύτερο σαββατοκύριακο υπερωριών στη σειρά είναι ένα ξεκάθαρο σημάδι ότι κάτι πάει στραβά με το συνολικό έργο
  - Οι ξεκούραστοι προγραμματιστές είναι πιθανότερο να:
    - σκεφτούν καλύτερες τακτικές για refactoring
    - σκεφτούν αυτό το ένα επιπλέον test το οποίο εμφανίζει κάποιο κρυμμένο σφάλμα στο σύστημα
    - μπορούν να αντεπεξέλθουν στις έντονες διαπροσωπικές δυναμικές της ομάδας ανάπτυξης