Ian Sommerville's

# Software Engineering

5th edition

Παρουσίαση κεφαλαίων

18, 19, 20

Δημήτρης Καλαμαράς

A.M. 137

2002

# 1. Αξιοπιστία Λογισμικού (ΑΛ)

- Στόχοι του κεφαλαίου
  - Εισαγωγή έννοιας αξιοπιστίας. Προβλήματα καθορισμού και μέτρησης της αξιοπιστίας
  - Περιγραφή μετρικών αξιοπιστίας που χρησιμοποιούνται στην ποσοτικοποίηση της αξιοπιστίας
  - Περιγραφή της διαδικασίας στατιστικού ελέγχου για την αποτίμηση της αξιοπιστίας
  - Επίδειξη του πως μπορούμε να βγάλουμε συμπεράσματα για την αξιοπιστία από τα αποτελέσματα στατιστικού ελέγχου

# 1.1 Τι είναι η ΑΛ;

- Η αξιοπιστία είναι το πιο σημαντικό ζητούμενο χαρακτηριστικό κάθε λογισμικού
- Άτυπα, η ΑΛ είναι το μέτρο του πόσο καλά νομίζουν οι χρήστες του συστήματος ότι τους παρέχει τις υπηρεσίες που απαιτούν
- Τυπικά, η ΑΛ συνήθως ορίζεται ως η πιθανότητα για ελεύθερη από αποτυχίες λειτουργία για καθορισμένο χρόνο σε ένα καθορισμένο πλαίσιο για ένα δεδομένο σκοπό
- Παράδειγμα: Λογισμικού αεροπλάνου, αξιόπιστο κατά 99.99% σε μια μέση πτήση 5 ωρών σημαίνει ότι πιθανή βλάβη θα υπάρξει 1 φορά στις 10000 πτήσεις
- Ο ορισμός της ΑΛ συνεπάγεται διαφορετικά πράγματα που εξαρτώνται από το σύστημα και τους χρήστες του

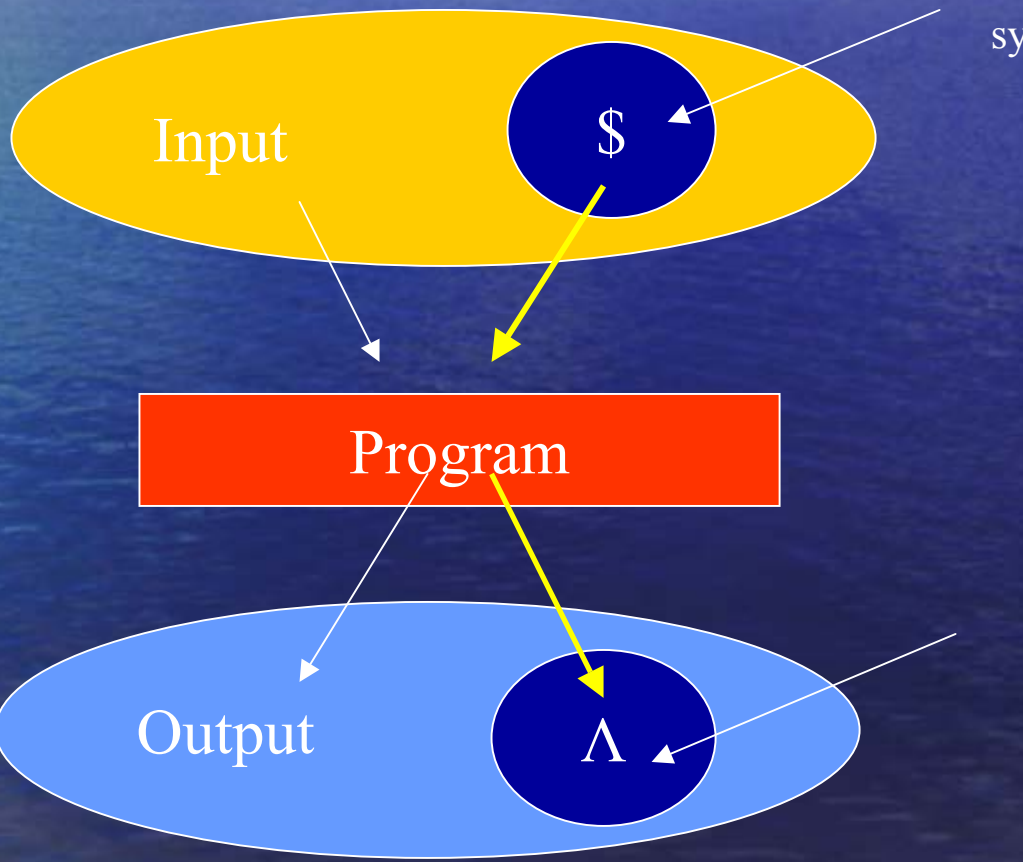
# 1.2 Σημεία άξια προσοχής στον ορισμό ΑΛ

- Δεν έχουν αξία γενικές δηλώσεις περί της ΑΛ
  - Δηλαδή, η έκφραση της ΑΛ σαν πιθανότητα έχει νόημα μόνο σε ένα συγκεκριμένο πλαίσιο χρήσης του λογισμικού, π.χ. διαφορετικά δεδομένα εισόδου μπορούν να προκαλούν αποτυχίες στο λογισμικό με διαφορετικούς τρόπους.
- Στον ορισμό της απαιτείται να λαμβάνεται υπόψη το επιχειρησιακό προφίλ, ο αναμενόμενος τρόπος χρήσης του λογισμικού
  - Π.χ. είναι δυνατό να ορίσουμε ΑΛ για ένα σύστημα που ελέγχει πάντα το ίδιο υλικό, αλλά είναι ανούσιο να ορίσουμε ΑΛ σε ένα αλληλεπιδραστικό σύστημα που θα χρησιμοποιείται με πολλούς και διαφορετικούς τρόπους
- Πρέπει να συνυπολογίζει τις συνέπειες των σφαλμάτων
  - Γιατί οι χρήστες δεν θεωρούν όλα τις υπηρεσίες εξίσου σημαντικές. Το σύστημα θεωρείται σαν περισσότερο αναξιόπιστο αν ενυπάρχουν περισσότερο κριτικής σημασίας σφάλματα

# 2.1 Αποτυχίες και σφάλματα

- Η ΑΛ είναι μια συνάρτηση του πλήθους των *αποτυχιών* του λογισμικού που έγιναν αντιληπτές από ένα χρήστη του.
  - Μια *αποτυχία* είναι η κατάσταση κατά την οποία το λογισμικό δεν παρέχει την αναμενόμενη από τον χρήστη υπηρεσία κατά την διάρκεια εκτέλεσης του
  - Ένα *σφάλμα* μπορεί να είναι ένα προγραμματιστικό ή σχεδιαστικό λάθος στο λογισμικό (όταν δεν συμφωνεί με τις προδιαγραφές του) ή λάθος στις προδιαγραφές ή στην τεκμηρίωση του λογισμικού που προκαλεί την εμφάνιση μιας αποτυχίας. Άρα τα σφάλματα είναι στατικά και είναι χαρακτηριστικά του κώδικα
  - Όμως τα σφάλματα δεν προκαλούν απαραίτητα αποτυχίες. Αυτό συμβαίνει μόνο όταν χρησιμοποιηθεί το εσφαλμένο κομμάτι του λογισμικού.

# 2.2 Παράδειγμα: Αντιστοίχηση Εισόδου/Εξόδου



# 3.1 Βελτίωση ΑΛ

- Η ΑΛ σχετίζεται λοιπόν με την πιθανότητα εμφάνισης λάθους κατά την χρήση
  - Η αφαίρεση x% των σφαλμάτων δεν οδηγεί απαραίτητα σε βελτίωση της αξιοπιστίας κατά x%
  - Για παράδειγμα, σε μια μελέτη η αφαίρεση του 60% των σφαλμάτων οδήγησε σε βελτίωση της αξιοπιστίας μόνο κατά 3%
  - Η ΑΛ δεν βελτιώνεται σημαντικά παρά μόνο όταν αφαιρούνται τα σφάλματα που βρίσκονται στα πλέον χρησιμοποιούμενα μέρη του λογισμικού
- Ο κυριότερος στόχος πρέπει να είναι η αφαίρεση των σφαλμάτων που έχουν σοβαρές συνέπειες

# 4.1 ΑΛ & τυπικές μέθοδοι

- Θεωρείται μερικές φορές ότι η χρήση τυπικών μεθόδων στην ανάπτυξη οδηγεί σε περισσότερο αξιόπιστα συστήματα καθώς το σύστημα θα ανταποκρίνεται στις προδιαγραφές του.
- Στο κεφ. 9, είδαμε πως η ανάπτυξη τυπικών προδιαγραφών οδηγεί σε μια λεπτομερή ανάλυση του συστήματος η οποία είναι χρήσιμη για την ανακάλυψη λαθών
- Παρόλα αυτά, μπορεί οι τυπικές μέθοδοι να μην βελτιώνουν πραγματικά την αξιοπιστία

# 4.2 ΑΛ & τυπικές μέθοδοι

- Αιτίες για τις οποίες οι τυπικές μέθοδοι μπορεί να μην εγγυώνται την ΑΛ στην πράξη:
  - Οι προδιαγραφές μπορεί να μην ανταποκρίνονται στις πραγματικές απαιτήσεις των χρηστών. Μπορεί οι αποτυχίες να είναι συνέπειες λαθών στις προδιαγραφές που οι τυπικές μέθοδοι δεν δύνανται να βρουν. Ή οι τυπικές προδιαγραφές μπορεί να κρύβουν τα προβλήματα διότι οι χρήστες δεν τις κατανοούν
  - Οι αποδείξεις προγραμμάτων μπορεί να περιέχουν λάθη, λόγω μεγέθους και πολυπλοκότητας
  - Η απόδειξη μπορεί να περιέχει υποθέσεις για το περιβάλλον και τη χρήση του συστήματος που είναι λανθασμένες, όπου η απόδειξη ακυρώνεται στην πράξη

# 5.1 ΑΛ & αποδοτικότητα

- Το τίμημα για την αύξηση της ΑΛ είναι η αποδοτικότητα του συστήματος που μειώνεται
- Γιατί, το αξιόπιστο λογισμικό πρέπει να συμπεριλαμβάνει πλεονάζων κώδικα προκειμένου να γίνονται έλεγχοι κτλ. Αυτό μειώνει την ταχύτητα εκτέλεσης του προγράμματος και αυξάνει συνήθως τον ζητούμενο χώρο αποθήκευσης

# 5.2 ΑΛ & αποδοτικότητα

- Λόγοι για τους οποίους η αξιοπιστία είναι συνήθως σημαντικότερη από την αποδοτικότητα:
  - Οι υπολογιστές είναι φτηνοί και γρήγοροι, άρα μπορούμε να βασιστούμε κάπως στην ταχύτητα των Η/Υ για την αποδοτικότητα
  - Το μη αξιόπιστο λογισμικό δεν προτιμάται από τους χρήστες
  - Το κόστος της αποτυχίας του λογισμικού μπορεί να είναι μεγάλο και συχνά υπερβαίνει το κόστος του λογισμικού
  - Τα αναξιόπιστα συστήματα είναι δύσκολο να βελτιωθούν

# 6.1 Μετρικές αξιοπιστίας

- Πρόκειται για μεγέθη είτε πιθανοθεωρητικά είτε χρονικά που προδιαγράφουν την ΑΛ
- Προέρχονται από τις μετρικές hardware
- Οι μονάδες χρόνου για τις μετρικές αξιοπιστίας θα πρέπει να επιλέγονται προσεκτικά. Δεν είναι ίδιες για όλα τα συστήματα
  - Απλός χρόνος εκτέλεσης (για συνεχούς-λειτουργίας συστήματα)
  - Ημερολογιακός χρόνος (για συστήματα που έχουν ένα τακτικό πρόγραμμα χρήσης, π.χ. Μια φορά την ημέρα)
  - Αριθμός πράξεων (για συστήματα που χρησιμοποιούνται κατ' αίτηση)

# 6.2 Είδη Μετρικών αξιοπιστίας

- Πιθανότητα αποτυχίας σε αίτηση (ΠΑΑ):
  - Είναι το μέτρο της πιθανότητας να αποτύχει το σύστημα όταν γίνεται μια αίτηση για μια υπηρεσία
  - ΠΑΑ=0.001 σημαίνει ότι 1 στις 1000 αιτήσεις για υπηρεσία θα οδηγήσουν σε αποτυχία
  - Η μετρική αυτή είναι χρήσιμη σε κριτικής ασφάλειας ή αδιάκοπης λειτουργίας συστήματα
  - Γίνεται με μέτρηση του αριθμού των αποτυχιών για έναν αριθμό δεδομένων εισόδου

# 6.3 Είδη Μετρικών αξιοπιστίας

- Ρυθμός εμφάνισης αποτυχίας (PEA)
  - Είναι η συχνότητα εμφάνισης αναπάντεχης συμπεριφοράς
  - PEA=0.02 σημαίνει ότι είναι πιθανόν να συμβούν 2 σφάλματα σε 100 μονάδες χρόνου λειτουργίας
  - Είναι χρήσιμη για λειτουργικά συστήματα, συστήματα διενέργειας πράξεων κτλ
  - Με μέτρηση του χρόνου ή αριθμού πράξεων μεταξύ δύο αποτυχιών

# 6.4 Είδη Μετρικών αξιοπιστίας

- Μέσος χρόνος μέχρι αποτυχία (MXA)
  - Μέτρο του χρόνου μεταξύ δύο παρατηρούμενων αποτυχιών
  - MXA=500 σημαίνει ότι ο χρόνος μεταξύ δύο αποτυχιών είναι κατά μέσο όρο 500 μ. Χρόνου
  - Χρήσιμη σε συστήματα όπου ο χρόνος είναι σημαντικός π.χ. CAD systems*
  - Με μέτρηση του χρόνου ή αριθμού πράξεων μεταξύ δύο αποτυχιών

# 6.5 Είδη Μετρικών αξιοπιστίας

- Διαθεσιμότητα (ΔΙΑ)
  - Μέτρο του πόσο συχνά είναι διαθέσιμο το σύστημα. Λαμβάνει υπόψη το χρόνο επισκευής/επανεκκίνησης
  - ΔΙΑ=0.998 σημαίνει ότι στις 998 από τις 1000 μ. χρόνου το λογισμικό είναι διαθέσιμο.
  - Χρήσιμο για συστήματα που λειτουργούν συνεχώς, π.χ. Τηλεφωνικά κέντρα
  - Μέτρηση χρόνου επανεκκίνησης μετά από αποτυχία

# 6.6 Μετρικές & συνέπειες αποτυχίας

- Οι μετρικές αξιοπιστίας βασίζονται όλες στην πιθανότητα μιας αποτυχίας αλλά δεν λαμβάνουν υπόψη τις συνέπειες μιας αποτυχίας
- Κάποια σφάλματα τα οποία είναι *παροδικά* μπορεί να μην έχουν πραγματικές συνέπειες ενώ άλλα σφάλματα μπορεί να προκαλέσουν απώλεια δεδομένων ή της ίδιας της λειτουργίας του συστήματος
- Έτσι όταν καθορίζουμε την ΑΛ είναι απαραίτητες διαφορετικές μετρικές για διαφορετικές κλάσεις αποτυχιών

# 7.1 Ταξινόμηση αποτυχιών

| ΑΠΟΤΥΧΙΑ | ΚΛΑΣΗ |
|---|---|
| Παροδική | Συμβαίνει μόνο με συγκεκριμένη είσοδο |
| Μόνιμη | Συμβαίνει με όλες τις εισόδους |
| Ανακτήσιμη | Το σύστημα επανέρχεται από μόνο του |
| Μη-ανακτήσιμη | Η επέμβαση του χρήστη είναι απαραίτητη για την την ανάνηψη από την αποτυχία |
| Μη-καταστροφική | Δεν καταστρέφει δεδομένα ή την κατάσταση του συστήματος |
| Καταστροφική | Καταστρέφει δεδομένα ή την κατάσταση του συστήματος |

# 7.2 Βήματα για προδιαγραφές αξιοπιστίας σε ένα σύστημα

- Για κάθε υπο-σύστημα, ανάλυσε τις συνέπειες των πιθανών αποτυχιών
- Από την ανάλυση των αποτυχιών του συστήματος, διαμέρισε τις αποτυχίες σε κατάλληλες κλάσεις
- Για κάθε μια κλάση που ταυτοποιείται, περιέγραψε την αξιοπιστία χρησιμοποιώντας την κατάλληλη μετρική. Διαφορετικές μετρικές μπορούν να χρησιμοποιηθούν για διαφορετικές απαιτήσεις αξιοπιστίας.

# 7.3 Παράδειγμα σε ATM

- Κάθε ATM σε ένα δίκτυο χρησιμοποιείται 300 φορές την ημέρα
- Ο κύκλος ζωής του λογισμικού είναι 2 χρόνια
- Κάθε ATM χειρίζεται περίπου 200.000 πράξεις στα δύο χρόνια της ζωής του λογισμικού του
- Η τράπεζα έχει συνολικα 1000 ATM
- Περίπου 300.000 πράξεις ΒΔ γίνονται συνολικά την ημέρα σε όλο το δίκτυο

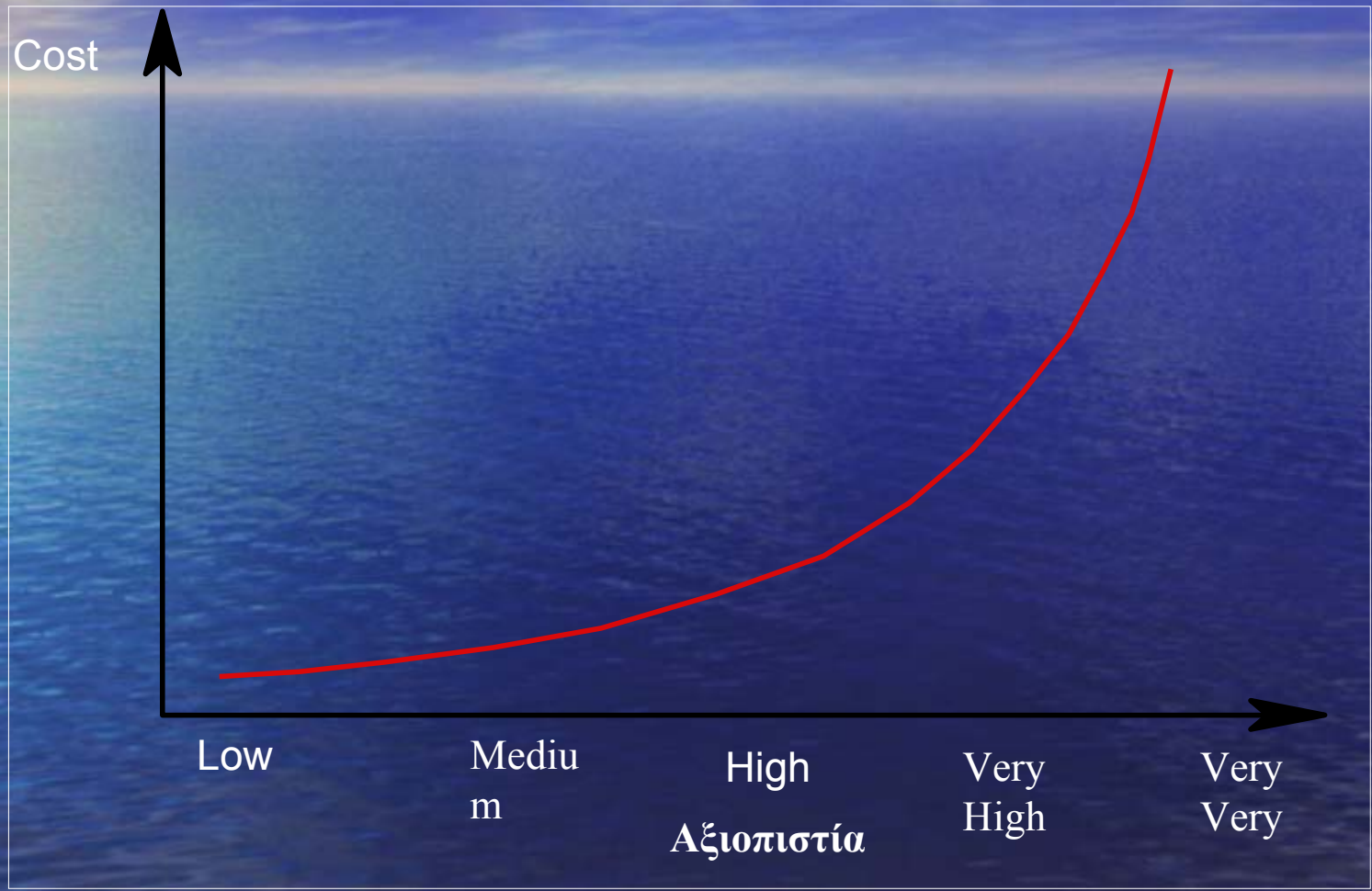
# 7.4 ATM: Ταξινόμηση αποτυχιών - πιθανών προδιαγραφών αξιοπιστίας

| Κλάση | Παράδειγμα | Μετρική Αξιοπιστίας |
|---|---|---|
| Μόνιμη, μη καταστρεπτική | Το σύστημα αποτυγχάνει να λειτουργήσει με κάθε κάρτα. Επανεκκίνηση | POFOD<br>1 ανά 1000 μέρες=3 χρόνια |
| Παροδική, μη καταστρεπτική | Η μαγνητική ταινία δεν μπορεί να αναγνωστεί ή προβληματική κάρτα | ΠΑΑ<br>1 ανά 1000 πράξεις |
| Παροδική, καταστρεπτική | Ένα σχέδιο πράξεων σε όλο το δίκτυο προκαλεί καταστροφή στη ΒΔ | Μη μετρήσιμη – δεν πρέπει να συμβεί ποτέ στη ζωή ενός συστήματος |

# 7.5 ATM: Επικύρωση προδιαγραφής ΑΛ

- Ιδεατά, αποτυχίες της τρίτης κλάσης δεν θα έπρεπε να συμβαίνουν. Μη καταστροφή της ΒΔ σημαίνει απαίτηση αξιοπιστίας ΠΑΑ < 1 / 200 εκατομμύρια (όλος ο χρόνος ζωής του λογ)
- Όμως, είναι αδύνατο να επικυρώσουμε εμπειρικά τόσο υψηλές προδιαγραφές αξιοπιστίας
- Εάν μια πράξη γίνεται σε ένα δευτερόλεπτο, τότε η προσομείωση μιας ημέρας πράξεων θα απαιτούσε 3.5 ημέρες. Ότι και να κάνουμε είναι δύσκολο να επικυρώσουμε αυτήν την προδιαγ. αξιοπιστίας
- Επιπλέον, η αύξηση στην ΑΛ, η ανάπτυξη και επικύρωση μιας προδιαγραφής ΑΛ, συνεπάγεται συνήθως εκθετική αύξηση στο κόστος του λογισμικού.

# Το κόστος της αυξανόμενης ΑΛ



# 8.1 Στατιστικός έλεγχος

- Είναι η διαδικασία ελέγχου του λογισμικού για την μέτρηση της αξιοπιστίας του παρά για ανίχνευση σφαλμάτων
- Χρησιμοποιούνται δοκιμ. δεδομένα, η επιλογή των οποίων θα πρέπει να ακολουθεί την προβλεπόμενη χρήση του λογισμικού
- Μπορεί να συνδυαστεί με την μοντελοποίηση ανάπτυξης ΑΛ. Μπορούν να γίνουν προβλέψεις της ΑΛ και του πότε θα επιτευχθεί και καθώς ανακαλύπτονται & διορθώνονται τα σφάλματα η ΑΛ θα βελτιώνεται κατά την διάρκεια του ελέγχου
- Πρέπει να καθορίζεται ένα αποδεκτό επίπεδο αξιοπιστίας και το λογισμικό να ελέγχεται και να τροποποιείται μέχρι να πετύχει αυτό το επίπεδο.

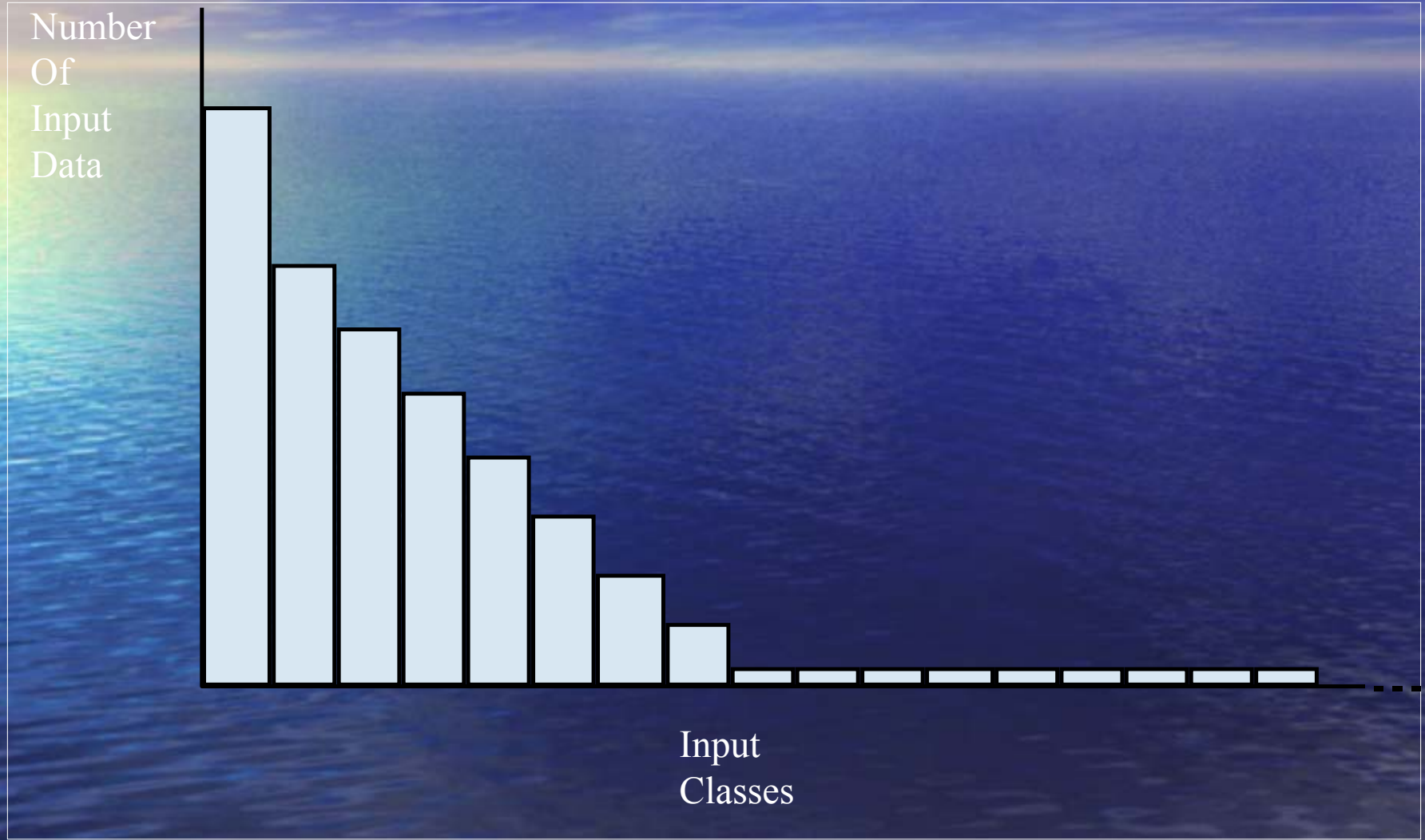
# 8.2 Διαδικασία στατιστικού ελέγχου

- Προσδιορισμός του επιχειρησιακού προφιλ του λογισμικού
- Δημιουργία ενός δοκιμαστικού συνόλου δεδομένων που να αντιστοιχούν σε αυτό το προφίλ
- Εφαρμογή δοκιμών, π.χ. με μέτρηση του χρόνου εκτέλεσης που μεσολαβεί μεταξύ αποτυχιών (προσοχή στις μονάδες)
- Μετά από ένα στατιστικά έγκυρο αριθμό δοκιμών μετράται η αξιοπιστία (μετρικές)

# 8.3 Δυσκολίες στατιστικού ελέγχου

- Αβεβαιότητα στο επιχειρησιακό προφιλ
  - Ιδιαίτερα για νέα συστήματα που δεν έχουν επιχειρησιακή ιστορία.
- Μεγάλο κόστος δημιουργίας του επιχειρησιακού προφιλ
  - Το κόστος εξαρτάται από τις πληροφορίες περί τη χρήση που συλλέγονται από τον οργανισμό που θέλει το προφιλ
- Στατιστική αβεβαιότητα όταν απαιτείται υψηλή αξιοπιστία
  - Δύσκολο να εκτιμηθεί ο βαθμός εμπιστοσύνης στο επιχειρησιακό προφίλ
  - Ο τρόπος χρήσης του λογισμικού μπορεί να αλλάξει με την πάροδο του χρόνου.

# 8.4 Ένα επιχειρησιακό προφιλ



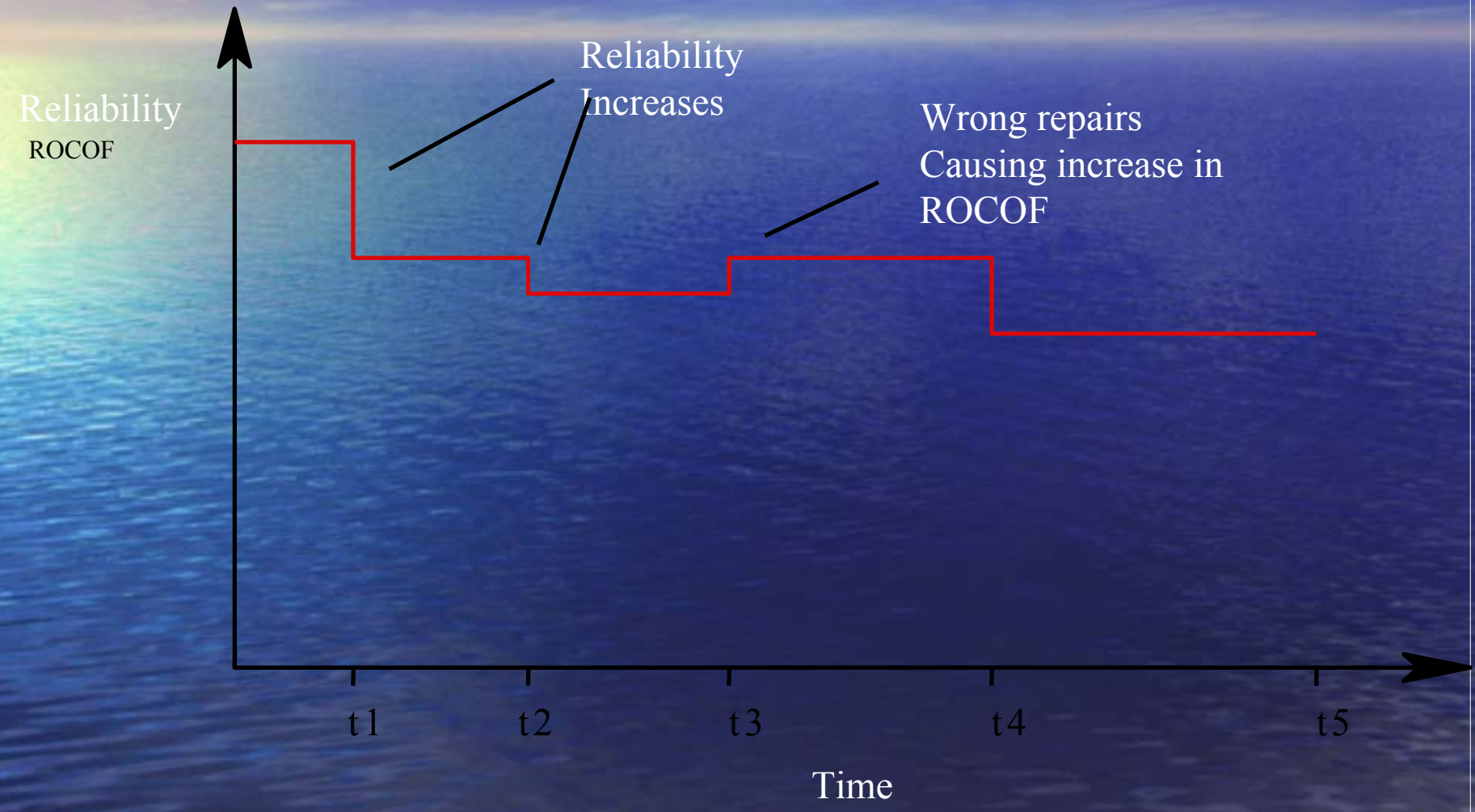
# 8.5 Παραγωγή επιχειρησιακού προφιλ

- Ο στατ. έλεγχος βασίζεται στην χρήση μεγάλου αριθμού δοκ. Δεδομένων. Βασίζεται στην παραδοχή ότι μόνο ένα μικρό μέρος αυτών μπορεί να προκαλέσει αποτυχία.
- Ο καλύτερος τρόπος παραγωγής δδ, όταν είναι δυνατό, είναι μια γεννήτρια τυχαίων δδ (αυτόματη παραγωγή προφίλ)
- Η αυτόματη παραγωγή προφιλ είναι δύσκολη για αλληλεπιδραστικά συστήματα
- Μπορεί να είναι άμεση για «κανονικές» εισόδους αλλά είναι δύσκολο να προβλεφθούν «απίθανες» είσοδοι και να παραχθούν δοκιμαστικά δεδομένα για αυτές.

# 9.1 Μοντελοποίηση αύξησης αξιοπιστίας

- Κατά την επικύρωση του λογισμικού είναι σημαντικό να προβλέψουμε το πότε έχει επιτευχθεί το επιθυμητό επίπεδο ΑΛ και να σταματήσουμε την επικύρωση-τεστάρισμα (λόγοι κόστους)
- Το μοντέλο αύξησης είναι είναι μαθηματικό μοντέλο της ΑΛ που προβλέπει πως η ΑΛ θα βελτιώνεται με το χρόνο καθώς το λογισμικό ελέγχεται και απομακρύνονται τα σφάλματα
- Έχουν προταθεί πολλά μοντέλα αύξησης της αξιοπιστίας
- Το απλούστερο είναι το step function, όπου η αξιοπιστία θεωρείται πως αυξάνεται με κάθε εύρεση σφάλματος
- Το μοντέλο Littlewood&Verral (σχήμα) προσπαθεί να λύσει τα προβλήματα των προηγούμενων
  - Δεν είναι όλες οι επιδιορθώσεις ίδιου αντίκτυπου στην αξιοπιστία
  - Κάποια επιδιόρθωση μπορεί να έχει αρνητικό αποτέλεσμα
  - Καθώς τα σφάλματα επιδιορθώνονται η μέση αύξηση αξιοπιστίας ανα σφάλμα μειώνεται, διότι τα πιο πιθανά σφάλματα ανακαλύπτονται ενωρίς!

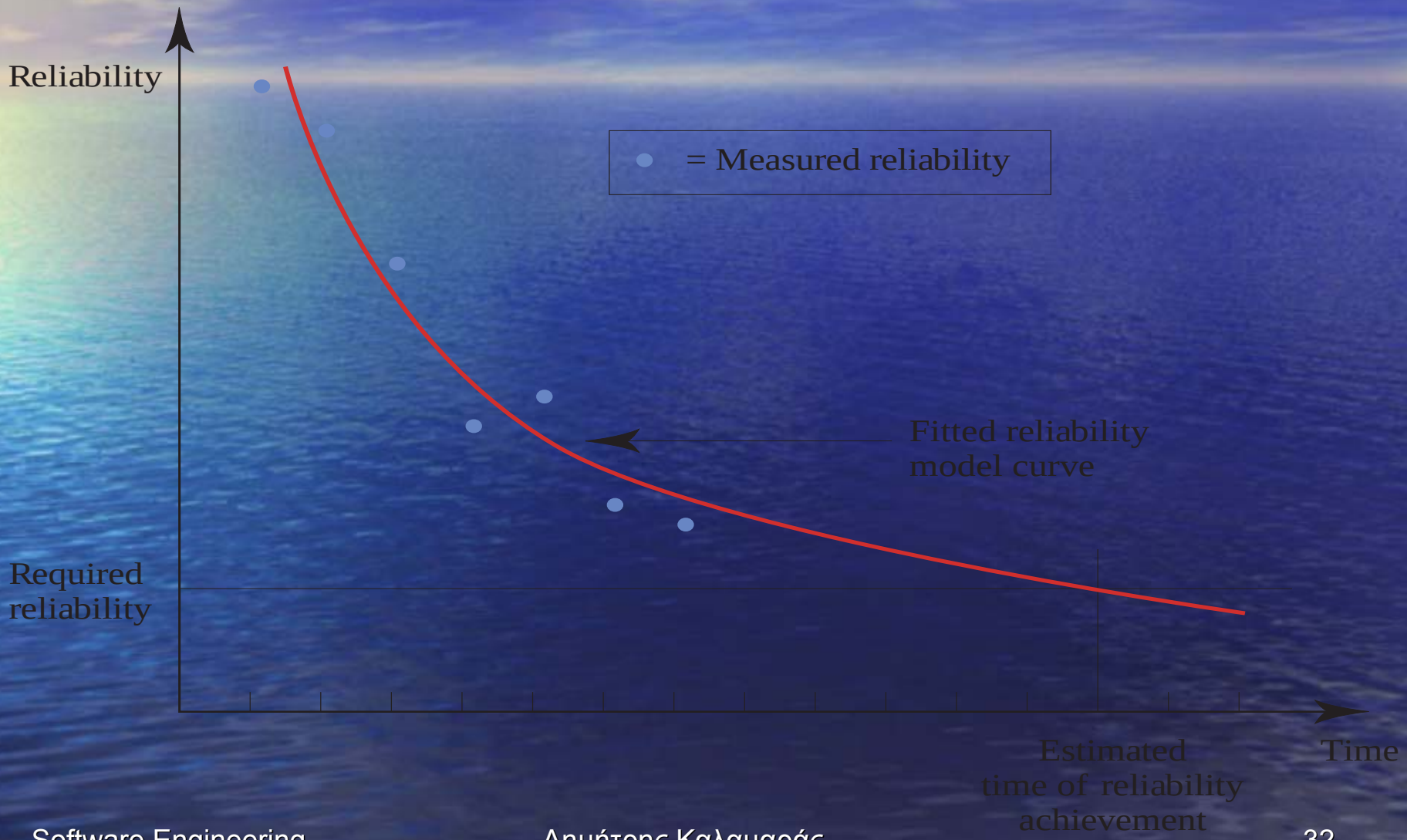
# 9.2 Μοντέλο ανάπτυξης αξιοπιστίας πολλαπλών βημάτων



# 9.3 Επιλογή μοντέλων ανάπτυξης

- Δεν υπάρχει μοντέλο που να εφαρμόζεται σε όλες τις περιπτώσεις, εξαρτάται από το πεδίο της εφαρμογής
- Για την πρόβλεψη της ΑΛ, τα παρατηρούμενα δεδομένα θα πρέπει να χρησιμοποιούνται σε διαφορετικά μοντέλα ανάπτυξης
- Το μοντέλο που ταιριάζει καλύτερα θα πρέπει να χρησιμοποιείται για την πρόβλεψη της αξιοπιστίας

# 9.4 Πρόβλεψη αξιοπιστίας



# 1. Προγραμματίζοντας για αξιοπιστία

- Στόχοι κεφαλαίου
  - Περιγραφή προγραμματιστικών τεχνικών για ανάπτυξη αξιόπιστων συστημάτων
  - Εξήγηση του πως αποφεύγουμε σφάλματα με ελαχιστοποίηση της χρήσης «κατασκευών που είναι επιρρεπείς σε λάθη»
  - Εισαγωγή έννοιας λογισμικών ανεκτικών σε σφάλματα και σχετικές μέθοδοι
  - Επίδειξη του πως ο χειρισμός εξαιρέσεων μπορεί να χρησιμοποιηθεί στη δημιουργία εύρωστων προγραμμάτων
  - Περιγραφή αμυντικής προσέγγισης προγραμματισμού, η οποία σκοπεύει στην ανίχνευση σφαλμάτων και στο ότι δεν οδηγούν αυτά σε αποτυχία του συστήματος

# 1.1 Το rationale των τεχνικών που θα συζητηθούν

- Εν γένει, οι τελικοί χρήστες ενός λογισμικού αναμένουν να είναι αξιόπιστο. Βελτιωμένες προγραμματιστικές τεχνικές, καλύτερες γλώσσες και καλύτερη διαχείριση στην ποιότητα έχουν οδηγήσει σε σημαντική αύξηση αξιοπιστίας στα περισσότερα συστήματα
- Όμως, κάποιες εφαρμογές, π.χ. αυτά που ελέγχουν μηχανήματα που δεν παρακολουθούνται, έχουν ειδικές απαιτήσεις υψηλής αξιοπιστίας και τότε πρέπει να χρησιμοποιηθούν *ειδικές* προγραμματιστικές τεχνικές για να επιτευχθούν αυτές.

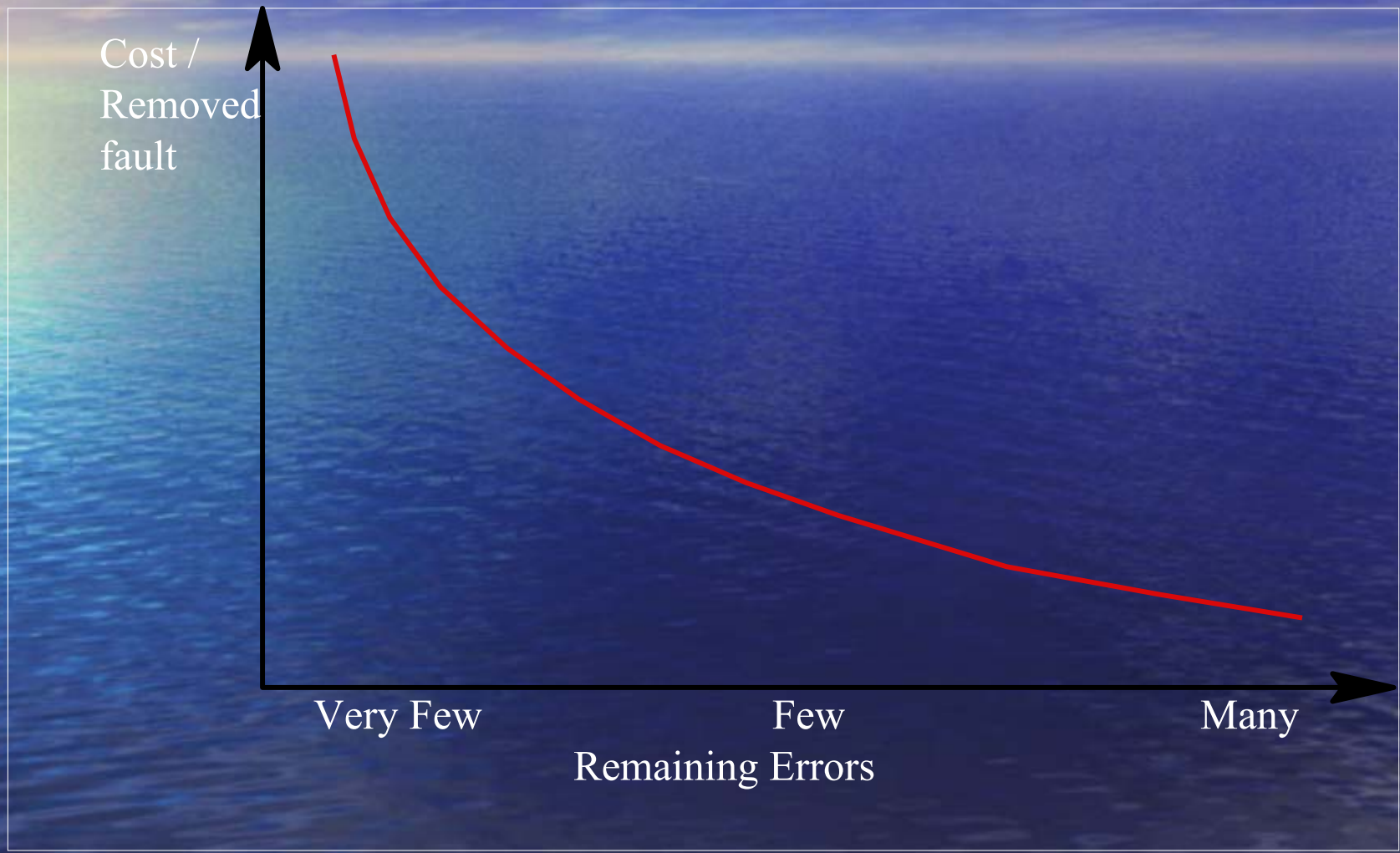
# 1.2 Επίτευξη αξιοπιστίας

- Η ΑΛ σε τέτοια συστήματα μπορεί να επιτευχθεί με τρεις συμπληρωματικές στρατηγικές:
  - Αποφυγή σφαλμάτων
    - Η πιο σημαντική & εφαρμόσιμη! Το λογισμικό σχεδιάζεται & υλοποιείται με τρόπο τέτοιο ώστε να μην περιέχει σφάλματα στον κώδικα
  - Αντοχή σε σφάλματα
    - Η στρατηγική υποθέτει ότι το τελικό προϊόν θα **έχει** σφάλματα και παρέχει στο λογισμικό μηχανισμούς ώστε να συνεχίσει την λειτουργία του όταν τα σφάλματα αυτά οδηγούν σε αποτυχία
  - Εντοπισμός σφαλμάτων * (στο 22ο κεφάλαιο)
    - Η διαδικασία ανάπτυξης & επικύρωσης οργανώνεται έτσι ώστε τα σφάλματα να εντοπίζονται και να επιδιορθώνονται πριν από την παράδοση στον τελικό πελάτη

# 2.1 Αποφυγή σφαλμάτων (ΑΣ)

- Μια καλή διαδικασία ανάπτυξης λογισμικού πρέπει να προσανατολίζεται στην ΑΣ παρά στον εντοπισμό.
- Ως λογισμικό ελεύθερο από σφάλματα εννοούμε στην ουσία εκείνο το λογισμικό που ανταποκρίνεται στις προδιαγραφές του. Όμως **δεν** σημαίνει λογισμικό που πάντα εκτελείται σωστά αφού π.χ. μπορεί να υπάρχουν λάθη στις προδιαγραφές ή να μην ανταποκρίνονται στις ανάγκες του χρήστη
- Το κόστος παραγωγής τέτοιου λογισμικού είναι πολύ υψηλό. Μπορεί τότε να είναι φτηνότερο να αποδεχτούμε κάποια σφάλματα στο λογισμικό

# 2.2 Κόστος – απομάκρυνση σφαλμάτων



# 2.3 Που βασίζεται η ΑΣ

- Τι απαιτείται για ανάπτυξη λογισμικού ελεύθερου από σφάλματα
  - Ακριβείς, προτιμότερα τυπικές, προδιαγραφές λογισμικού
  - Υιοθέτηση οργανωτικής φιλοσοφίας ποιότητας, όπου η ποιότητα να είναι οδηγός στην υλοποίηση
  - Υιοθέτηση μιας προσέγγισης στον σχεδιασμό και υλοποίηση που να βασίζεται στην απόκρυψη πληροφορίας και τη συμπύκνωση* (*encapsulation)* και ενθαρρύνει ευ-αναγνώσιμα προγράμματα
  - Θα πρέπει να χρησιμοποιείται μια γλώσσα που απαιτεί αυστηρότητα στο γράψιμο κώδικα έτσι ώστε τα πιθανά λάθη να εντοπίζονται και από τον compiler
  - Περιορισμούς στην χρήση προγραμματιστικών κατασκευών όπως οι δείκτες που είναι επιρρεπείς στα λάθη. Προσεκτικός και εκτεταμένος έλεγχος του συστήματος

# 2.4 ΑΣ & C++

- Η ΑΣ σε λογισμικό με γλώσσες χαμηλού επιπέδου είναι σχεδόν αδύνατη. Σε strongly-typed γλώσσες, π.χ. Ada, C++, ο compiler μπορεί να βρει κάποια σφάλματα πριν από την εκτέλεση του προγραμματισμού
- Ο ίδιος προτιμά την Ada, αλλά η C++ χρησιμοποιείται όλο και ευρύτερα για ανάπτυξη. Συνδυάζει την αποτελεσματικότητα μιας χαμηλού επιπέδου γλώσσας με δομές αντικειμενοστραφούς προγραμματισμού. Διαθέτει καλύτερο έλεγχο τύπων από την C αλλά όχι τόσο όσο η Ada. Μειονέκτημα: είναι σχεδόν αδύνατο να γραφτεί αποτελεσματικό πρόγραμμα χωρίς την χρήση δεικτών, που όπως ελέχθη πρότερα είναι επιρρεπείς σε λάθη.

# 2.5 Δομημένος προγραμματισμός

- Δομές επιρρεπείς σε σφάλματα, που θα πρέπει να αποφεύγετε* για πρόγραμμα ελεύθερο από σφάλματα:
  - GOTOs
  - Αριθμοί κινητής υποδιαστολής
    - Εμφύτως ανακριβείς. Π.χ. η ανακρίβεια μπορεί να οδηγήσει σε λάθος συγκρίσεις
  - Δείκτες
    - Οι δείκτες που αναφέρονται σε λάθος περιοχές μνήμης μπορεί να καταστρέψουν δεδομένα. Άλλο πρόβλημα τους το aliasing, μπορεί να κάνουν τα προγράμματα δύσκολα στην κατανόηση και αλλαγή.
  - Δυναμική εκχώρηση μνήμης
    - Η κατά την λειτουργία εκχώρηση μνήμης μπορεί να προκαλέσει υπερχείλιση μνήμης
  - Παράλληλος προγραμματισμός
    - Μπορεί να προκαλέσει πολύπλοκα προβλήματα χρονισμού λόγω απρόβλεπτων αλληλεπιδράσεων μεταξύ των παράλληλων διεργασιών

# 2.5 ....συνέχεια

- Αναδρομή
  - Λάθη στην αναδρομή μπορεί να προκαλέσουν υπερχείλιση στην μνήμη. Είναι και δύσκολο να ακολουθηθεί η λογική τους
- Διακοπές (*interrupts)*
  - Οι διακοπές μπορεί να προκαλέσουν μια κριτικής σημασίας λειτουργία να τερματιστεί και να κάνουν ένα πρόγραμμα δύσκολο στην κατανόηση. Συγκρίνονται με τις εντολές GOTO
- *Δεν προτείνεται να μην χρησιμοποιούνται ποτέ οι παραπάνω δομές, απλώς πρέπει να τους δίνεται ιδιαίτερη προσοχή.

# 2.6 Απόκρυψη πληροφορίας

- Τα τμήματα του προγράμματος πρέπει να έχουν πρόσβαση μόνο στα δεδομένα που τους είναι απαραίτητα για να λειτουργήσουν. Αυτό εμπλέκει την κατασκευή αντικειμένων (*objects)* ή αφηρημένων τύπων δεδομένων (ADT) που διατηρούν την κατάσταση και λειτουργίες σε αυτήν
- Έτσι αποφεύγουμε σφάλματα για τρεις λόγους:
  - Την πιθανότητα κατά τύχη καταστροφής πληροφορίας
  - Η πληροφορία περιβάλλεται από *firewalls* έτσι ώστε τα τυχόν προβλήματα είναι δύσκολο να διαδοθούν σε άλλα τμήματα του προγράμματος
  - Καθώς η πληροφορία γίνεται τοπική, ο προγραμματιστής είναι λιγότερο πιθανό να κάνει λάθη και οι testers είναι πιο εύκολο να τα ανακαλύψουν

# 2.7 Κλάσεις Αντικειμένων και ADT

- Υλοποιούνται στην C++ σαν objects ενώ στην Ada σαν packages
- ADT & objects classes δεν είναι το ίδιο ακριβώς. Οι κλάσεις είναι σαν πρότυπα που χρησιμοποιούνται για την δημιουργία αντικειμένων που έχουν λειτουργίες και ιδιότητες που ορίζονται στην κλάση
- Το όνομα ορίζεται μέσα στο αντικείμενο ή ADT
- Οι λειτουργίες ορίζονται σαν procedures ή functions
- Η αναπαράσταση της δομής ορίζεται στο private μέρος
- Οι κλάσεις & οι γενικές ADT μπορούν να παραμετροποιηθούν

# Δήλωση κλάσης ουράς ακεραίων - C++

```
class Queue {
public:
        Queue () ;
        ~Queue () ;
        void Put ( int x ) ; // adds an item to the queue
        int Remove () ; // this has side effect of changing the queue
        int Size( ) ;  // returns number of elements in the queue
private:
        int front, back ;
        int qvec [100] ;
} ;
```

# 2.8 Γενίκευση κλάσεων

- Η συμπεριφορά των αντικειμένων και ADTs που συντίθεται από άλλα αντικείμενα ή ADTs είναι συχνά ανεξάρτητη από το είδος των συνιστωσών τους
- Η γενίκευση είναι ένας τρόπος γραψίματος γενικών, παραμετροποιημένων ADTs και κλάσεων αντικειμένων που μπορούν να υλοποιηθούν αργότερα με συγκεκριμένους τύπους.
- Τόσο η Ada όσο και η C++ διαθέτουν τέτοιες δυνατότητες.

# Γενικευμένη ουρά σε C++

```
template
        <class elem>
class Queue {
public:
        Queue ( int size = 100 ) ; // default to queue of size 100 elements
        ~Queue () ;
        void Put ( elem x ) ;
        elem Remove ( ) ; // this has side effect of changing queue
        int Size ( ) ;
private:
        int front, back ;
        elem* qvec ;
} ;
```

# 2.9 *Instantiation* γενικευμένης ADT

- Τα instants των γενικευμένων ADTs δημιουργούνται κατά την ώρα του compile και όχι κατά την εκτέλεση και έτσι είναι εφικτός ο έλεγχος των τύπων

```
//Assume List has been defined elsewhere as a type
    Queue <int> Int_queue (50) ;
    Queue <List> List_queue (200) ;
```

# 3.1 Αντοχή σε σφάλματα

- Σε καταστάσεις κρίσιμης σημασίας, όπου η απώλεια λειτουργίας θα είναι καταστροφική, τα συστήματα λογισμικού πρέπει να είναι ανεκτικά σε σφάλματα
- Η αντοχή σε σφάλματα σημαίνει ότι το σύστημα συνεχίζει να λειτουργεί παρά το γεγονός μιας αποτυχίας στο λογισμικό
- Ακόμη και εάν το σύστημα είναι ελεύθερο από σφάλματα, πρέπει να έχει επίσης αντοχή σε σφάλματα καθώς μπορεί να υπάρχουν λάθη στις προδιαγραφές ή η επαλήθευση μπορεί να ήταν λανθασμένη. Fault-free σύστημα δεν σημαίνει Failure-free

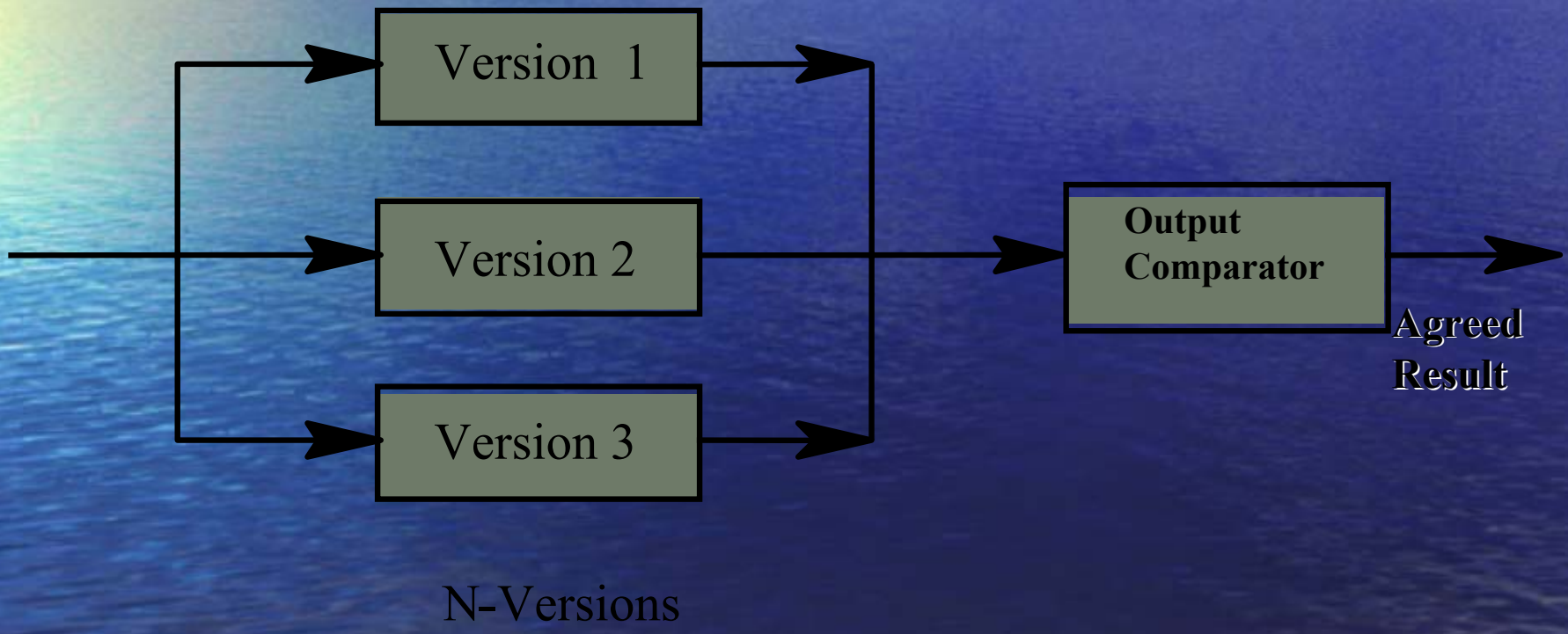
# 3.2 Ενέργειες λογισμικού για αντοχή σε σφάλματα

- Ανίχνευση αποτυχίας
  - Το σύστημα πρέπει να εντοπίζει μια αποτυχία όταν αυτή συμβαίνει
- Αποτίμηση ζημιάς
  - Τα τμήματα της κατάστασης συστήματος που επηρεάζονται από την αποτυχία θα πρέπει να εντοπίζονται
- Ανάνηψη από σφάλμα
  - Το σύστημα θα πρέπει να επαναφέρει τον εαυτό του σε ασφαλή κατάσταση είτε με forward είτε με backward recovery
- Διόρθωση σφάλματος
  - Το σύστημα μπορεί να τροποποιείται ώστε να αποτρέπεται η επανεμφάνιση του σφάλματος. Καθώς πολλά σφάλματα λογισμικού είναι παροδικά αυτό συνήθως είναι μη αναγκαίο.

# 3.3 Πρόβλεψη αντοχής σε σφάλματα

- Δύο είναι οι βασικές προσεγγίσεις. Και οι δύο προέκυψαν από αντίστοιχα hardware μοντέλο όπου ένα στοιχείο ή και ολόκληρο το σύστημα αντιγράφεται.
- Προγραμματισμός v-εκδόσεων
  - Παράγεται ένας αριθμός διαφορετικών εκδόσεων του λογισμικού που όλες ακολουθούν τις ίδιες προδιαγραφές. Όλες οι εκδόσεις τίθενται σε ταυτόχρονη λειτουργία και επιλέγονται αυτές που δίνουν στην πλειοψηφία τους την ίδια έξοδο. Παράδειγμα εφαρμογής: Airbus A320. Μειονέκτημα: δεν παρέχει αντοχή σε σφάλματα όταν υπάρχουν λάθη στις προδιαγραφές
- Μπλοκ επαναφοράς
  - Οι εκδόσεις τρέχουν σε ακολουθία. Επιλέγεται η έξοδος που επιτυγχάνει σε κάποιο «τεστ αποδοχής». Η αδυναμία της είναι ότι είναι δύσκολο να γραφτεί κατάλληλο τεστ.

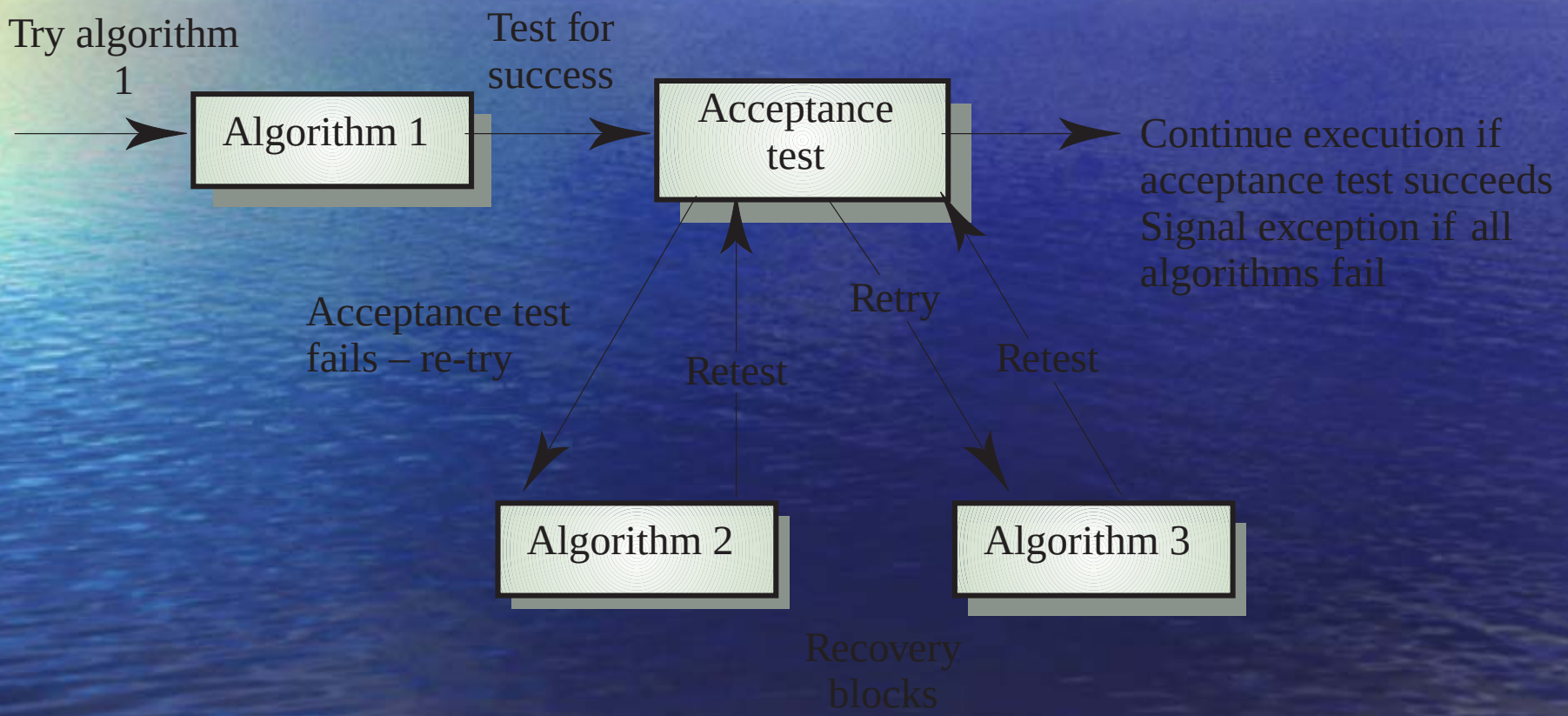
# 3.4 Προγραμματισμός Ν-εκδόσεων



# 3.5 Προγραμματισμός ν-εκδόσεων

- Οι διαφορετικές εκδόσεις του συστήματος σχεδιάζονται και υλοποιούνται από διαφορετικές ομάδες. Θεωρείται έτσι ότι υπάρχει μικρή πιθανότητα να κάνουν τα ίδια λάθη
- Η εμπειρία όμως δείχνει το αντίθετο. Οι ομάδες παρανοούν τις προδιαγραφές με τον ίδιο τρόπο και τελικά καταλήγουν να χρησιμοποιούν τους ίδιους αλγορίθμους στις εκδόσεις τους.

# 3.6 Μπλοκ επαναφοράς



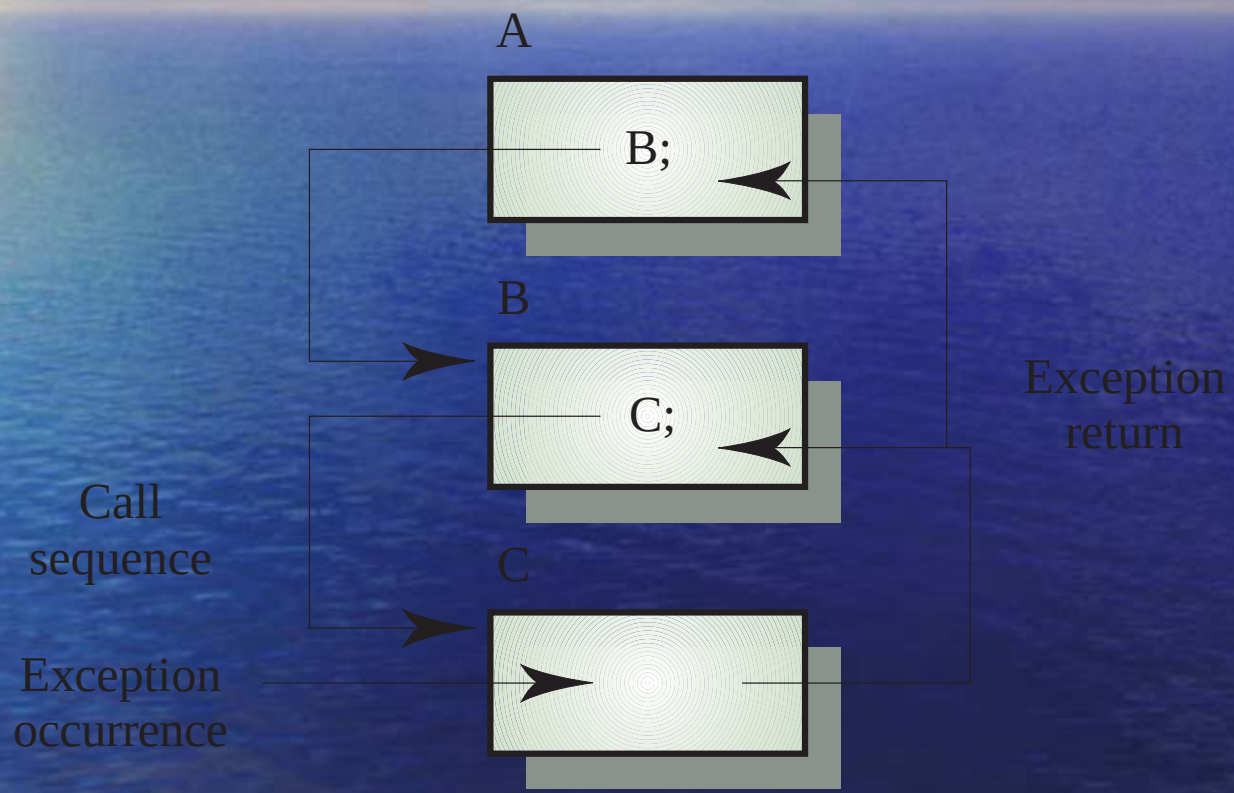
# 3.7 Μπλοκ επαναφοράς

- Απαιτούν διαφορετικό αλγόριθμο να χρησιμοποιείται σε κάθε έκδοση ώστε να μειώνεται η πιθανότητα κοινών ή επαναλαμβανόμενων λαθών
- Όμως είναι δύσκολη η δημιουργία κατάλληλου «τεστ αποδοχής»
- Ακόμη, όπως και στον προγραμματισμό με ν-εκδόσεις, η μέθοδος είναι ευάλωτη σε σφάλματα που προέρχονται από τις προδιαγραφές

# 4.1 Χειρισμός εξαιρέσεων

- Μια εξαίρεση σε ένα πρόγραμμα είναι είτε ένα λάθος ή κάποιο απροσδόκητο γεγονός, π.χ. πτώση τάσης
- Οι δομές χειρισμού εξαιρέσεων επιτρέπουν την πρόβλεψη και αντιμετώπιση τέτοιων καταστάσεων χωρίς να είναι απαραίτητος ο συνεχής έλεγχος της τρέχουσας κατάστασης για τον εντοπισμό τέτοιων εξαιρέσεων
- Οι περισσότερες γλώσσες προγραμματισμού δεν παρέχουν χειρισμό εξαιρέσεων, οπότε αν χρησιμοποιούμε κανονικές δομές ελέγχου για την ανίχνευση εξαιρέσεων π.χ. σε μια ακολουθία nested κλήσεων υπο-ρουτινών οδηγούμαστε στην πρόσθεση πολλών επιπλέον δηλώσεων στο πρόγραμμα που με τη σειρά τους προκαλούν αύξηση του χρόνου εκτέλεσης του προγράμματος
- Η C++ παρέχει χειρισμό εξαιρέσεων.

# Παράδειγμα: nested proc. calls



# 4.2 Χειρισμός εξαιρέσεων στην C++

- Το keyword **throw** σημαίνει την εμφάνιση μιας εξαίρεσης. Ο χειρισμός της εξαίρεσης γίνεται μέσω του keyword **catch**
- Οι εξαιρέσεις ορίζονται σαν κλάσεις ώστε να μπορούν να κληρονομούν ιδιότητες από άλλες κλάσεις εξαιρέσεων
- Κανονικά, οι εξαιρέσεις αντιμετωπίζονται απόλυτα μέσα στο μπλοκ όπου εμφανίστηκαν, παρά να διαδίδονται μέσα στον υπόλοιπο κώδικα.
- Όλες οι εξαιρέσεις είναι καθοριζόμενες από τον χρήστη. Δεν υπάρχουν built-in εξαιρέσεις.

# 4.3 Παράδειγμα: Θερμοστάτης

- Ελέγχει ένα ψυγείο με σκοπό την διατήρηση της θερμοκρασίας εντός συγκεκριμένων πλαισίων
- Ανοίγει/Κλείνει μια αντλία ψυκτικού υγρού
- Στέλνει ένα σήμα (alert) εάν η μέγιστη προβλεπόμενη θερμοκρασία –18 ξεπεραστεί
- Χρησιμοποιεί εξωτερικές οντότητες: Pump, Temp_dial, Sensor, Alarm

# 4.4 Θερμοστάτης

```
void Control_freezer ( const float Danger_temp)
{
      float Ambient_temp ;
      // try means exceptions will be handled in this block
      // Assume that Sensor, Temperature_dial and Pump are
      //objects which have been declared elsewhere
      try {
            while (true) {
                  Ambient_temp = Sensor.Get_temperature () ;
                  if (Ambient_temp > Temperature_dial.Setting () )
                        if (Pump.Status () == off)
                        {
                              Pump.Switch (on) ;
                              Wait (Cooling_time) ;
                        }
                        else
                              if (Pump.Status () == on)
                                    Pump.Switch (off) ;
                  if ( Ambient_temp > Danger_temp )
                              throw Freezer_too_hot ( ) ;
            }  // end of while loop
      }  // end of exception handling try block
      // catch indicates the exception handling code.
      catch ( Freezer_too_hot )
            Alarm.Activate () ;
}
```

# 5.1 Αμυντικός προγραμματισμός

- Είναι μια προσέγγιση στην ανάπτυξη προγραμμάτων (fault-tolerant) στην οποία θεωρείται ότι μπορεί να υπάρχουν σφάλματα που δεν έχουν ανιχνευθεί
- Έτσι τέτοια προγράμματα περιέχουν κώδικα ανίχνευσης και ανάνηψης από τέτοια σφάλματα
- Οι τεχνικές που χρησιμοποιεί είναι θεμελιώδεις για την διαδικασία αντοχής σε σφάλματα:
  - Παρεμπόδιση αποτυχιών
  - Αποτίμηση ζημιάς
  - Ανάνηψη από σφάλμα

# 5.2 Παρεμπόδιση αποτυχιών

- Γλώσσες όπως η C++ επιτρέπουν τον εντοπισμό κατά την ώρα του compile πολλών πιθανά καταστροφικών σφαλμάτων λόγω strict type rules
- Επίσης, μια ομάδα πιθανών αποτυχιών μπορεί να εντοπίζεται είτε με range checking είτε με χειρισμό εξαιρέσεων την ώρα της εκτέλεσης του προγράμματος
- Επιπλέον μπορούν να αναπτυχθούν κάποιες δηλώσεις κατάστασης (*state assertions*) και να συμπεριληφθούν στο πρόγραμμα, π.χ. λογικά κατηγορήματα πάνω στις μεταβλητές κατάστασης του συστήματος (κεφ. 9), αυτά ελέγχονται πριν από την ανάθεση σε αυτές τις μτβ και εάν θα προέκυπτε μια ανώμαλη τιμή για αυτήν την μτβλ τότε θα υπήρχε κάποιο λάθος

# 5.3 Αποτίμηση ζημιάς

- Η αποτίμηση ζημιάς περιλαμβάνει ανάλυση της κατάστασης συστήματος για την εκτίμηση της έκτασης της καταστροφής που προκλήθηκε από μια αποτυχία του
- Πρέπει να εκτιμάται το ποια τμήματα της κατάστασης του λογισμικού έχουν επηρεαστεί από την αποτυχία
- Γενικά, βασίζονται σε "validity functions", δηλαδή συναρτήσεις που κατά κάποιο τρόπο εφαρμόζονται σε στοιχεία/μεταβλητές κατάστασης και αξιολογούν αν βρίσκονται εντός καθορισμένου εύρους

# 5.4 Κλάση C++ με αποτίμηση ζημιάς

```
template <class elem> class Robust_array {
public:
      Robust_array (int size = 20) ;
      ~Robust_array () ;
      void Assign ( int Index, elem Val) ;
      elem Eval  (int Index) ;

      // Damage assessment functions
      // Assess_damage takes a pointer to a  function as a parameter
      // It sets the corresponding element of Checks if a problem is
      // detected by the function Test
      void Assess_damage ( void (*Test ) (boolean*)) ;
      boolean Eval_state (int Index) ;
      boolean  Is_damaged () ;
private:
      elem* Vals ;
      boolean* Checks ;
} ;
```

# 5.5 Τεχνικές αποτίμησης ζημιάς

- Τεχνικές για εντοπισμό σφαλμάτων και αξιολόγησης της ζημιάς:
  - Checksums (τιμή που υπολογίζεται με κάποια συνάρτηση από τα δεδομένα από τον αποστολέα) στην μετάδοση δεδομένων και έλεγχος ψηφίων στα αριθμητικά δεδομένα
  - Όταν χρησιμοποιούνται συνδεδεμένες δομές δεδομένων, μπορούν να συμπεριληφθούν επιπλέον «προς τα πίσω» δείκτες. Π.χ. για κάθε αναφορά από το Α στο Β, υπάρχει μια συγκρίσιμη αναφορά από το Β στο Α.
  - Χρονομετρητές για τον έλεγχο διεργασιών που δεν λένε να τελειώσουν. Π.χ. αν δεν υπάρχει απόκριση μετά από κάποιο διάστημα θεωρείται ότι υπάρχει πρόβλημα

# 5.6 Ανάνηψη από σφάλμα

- Γενικά είναι η διαδικασία τροποποίησης της κατάστασης του συστήματος ώστε να ελαχιστοποιηθούν τα αποτελέσματα ενός σφάλματος. Έχουμε:
- Ανάνηψη προς τα εμπρός
  - Προσπάθεια για επιδιόρθωση μιας καταστραμμένης κατάστασης συστήματος
  - Συνήθως είναι συγκεκριμένη για κάθε εφαρμογή. Απαιτείται πλήρης γνώση της κατάστασης για τον υπολογισμό πιθανών επιδιορθώσεων
- Ανάνηψη προς τα πίσω
  - Επιστροφή σε μια πρότερη κατάσταση που ήταν ασφαλής
  - Είναι πιο απλή. Διατηρούνται οι λεπτομέρειες μιας προηγούμενης ασφαλούς κατάστασης συστήματος και γίνεται προσπάθεια επιστροφής σε αυτή.

# 5.7 Ανάνηψη προς τα εμπρός

- Που χρησιμοποιείται συνήθως:
- Σε καταστροφή του κώδικα δεδομένων
  - Τεχνικές που επισυνάπτουν πλεονάζων κώδικα στα δεδομένα, ο οποίος μπορεί να χρησιμοποιηθεί για την εντοπισμό και διόρθωση λαθών
- Όταν συνδεδεμένες δομές έχουν καταστραφεί
  - Όταν περιλαμβάνονται «προς τα πισω» δείκτες σε μια δομή δεδομένων, π.χ. μια λίστα, τότε μια κατεστραμμένη λίστα μπορεί να ανακατασκευαστεί εάν ένας ικανός αριθμός από τέτοιους δείκτες είναι άθικτος. Συνήθως χρησιμεύει σε διορθώσεις βάσεων δεδομένων και συστημάτων αρχείων.

# 5.8 Ανάνηψη προς τα πίσω

- Μια συνήθης μέθοδος ανάνηψης προς τα πίσω είναι οι transactions*. Εδώ δεν εφαρμόζονται αλλαγές, αν δεν έχει ολοκληρωθεί ο υπολογισμός. Έτσι εάν συμβεί ένα λάθος το σύστημα παραμένει στην κατάσταση που ήταν πριν από την transaction.
- Περιοδικά σημεία ελέγχου (checkpoints) τίθενται και επιτρέπουν στο σύστημα να «επιστρέψει» σε μια προηγούμενη ασφαλή κατάσταση, πχ Win ME

# 1. Επαναχρησιμοποίηση Λογισμικού

- Στόχοι κεφαλαίου
  - Εξήγηση μειονεκτημάτων και πλεονεκτημάτων της επαναχρησιμοποίησης λογισμικού
  - Περιγραφή διαδικασιών που εμπλέκονται στην ανάπτυξη με και για επαναχρησιμοποίηση
  - Περιγραφή μεθόδων ανάπτυξης εφαρμογών ώστε να είναι portable*, να μπορούν να επαναχρησιμοποιηθούν σε διαφορετικές πλατφόρμες

# 1.1 Είδη & πρακτικές επαναχρησιμοποίησης

- Ολόκληρο σύστημα
  - Όλο το σύστημα μπορεί να επαναχρησιμοποιηθεί σε κάποια άλλη πλατφόρμα. Συνήθως αναφέρεται σαν *portability*
  - Είναι ευρέως διαδεδομένη πρακτική. Χρησιμοποιείται από πολλούς κατασκευαστές π.χ. Micro$oft για να διαθέσουν τα προϊόντα τους σε άλλες πλατφόρμες
- Υπο-σύστημα
  - Ένα ή περισσότερα τμήματα του όλου λογισμικού, π.χ. ένα pattern-matching subsystem επαναχρησιμοποιείται σε ένα άλλο σύστημα
  - Χρησιμοποείται άτυπα αφού πολλοί μηχανικοί επαναχρησιμοποιούν προηγούμενη δουλειά τους
- Άρθρωμα-αντικείμενο
  - Μια συλλογή συναρτήσεων ή διαδικασιών, π.χ. ένα C++ object
- Συνάρτηση
  - Μια μόνο συνάρτηση
  - Είναι πολύ συνηθισμένη σε συγκεκριμένα πεδία, όπου συγκεκριμένες «βιβλιοθήκες» επαναχρησιμοποιούμενων συναρτήσεων είναι πολύ διαδεδομένες. Π.χ. Fortran

# 1.2 Οι 4 όψεις επαναχρησιμοποίησης

- Ανάπτυξη λογισμικού **με** επαναχρησιμοποίηση ψηφίδων
  - Δημιουργία λογισμικού δεδομένων κάποιων ήδη υπαρχόντων ψηφίδων
- Ανάπτυξη λογισμικού **για** επαναχρησιμοποίηση
  - Σχεδίαση και δημιουργία γενικών ψηφίδων που να είναι επαναχρησιμοποιήσιμες
- Επαναχρησιμοποίηση βασισμένη σε γεννήτορες
  - Επαναχρησιμοποίηση που είναι συγκεκριμένη για κάποιο πεδίο μέσα από γεννήτορες εφαρμογών
- Επαναχρησιμοποίηση ολόκληρου συστήματος
  - Ανάπτυξη συστημάτων ώστε να είναι δυνατό να μεταφερθούν από μια πλατφόρμα σε μια άλλη.

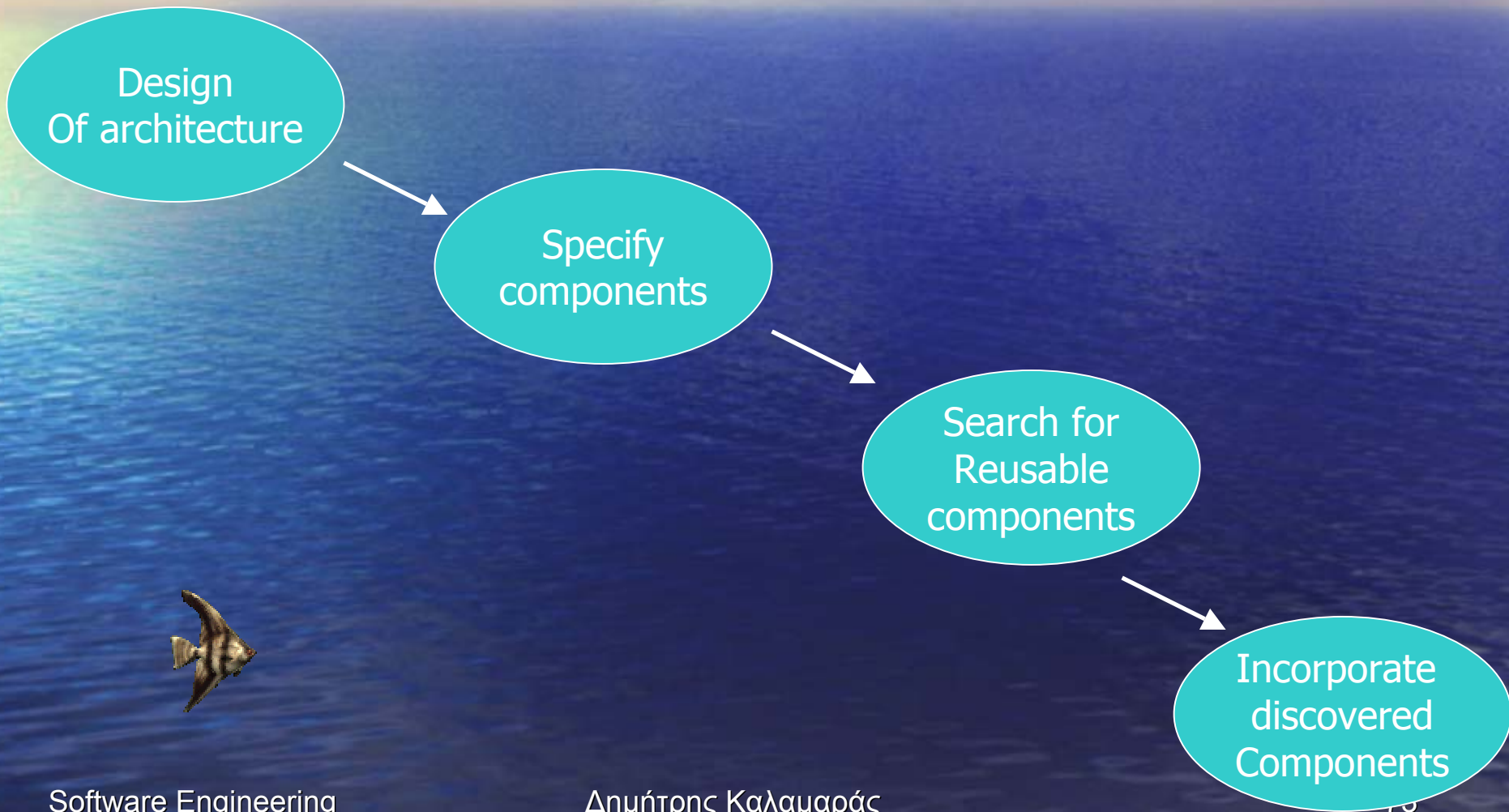
# 2.1 Ανάπτυξη λογισμικού **με** επαναχρησιμοποίηση ψηφίδων

- Είναι μια προσέγγιση στην ανάπτυξη που αποσκοπεί στην μεγιστοποίηση της επανάχρησης ψηφίδων που υπάρχουν. Το πλεονέκτημα είναι ότι το συνολικό κόστος του λογισμικού συνήθως μειώνεται αφού χρειάζεται να φτιαχτούν λιγότερα πράγματα εξαρχής.
- Συνήθως, όμως, να πρέπει να προσαρμοστούν οι ψηφίδες στην νέα εφαρμογή

# 2.2 Επιπρόσθετα πλεονεκτήματα

- Η αξιοπιστία του συστήματος αυξάνεται
  - Επαναχρ. ψηφίδες που έχουν δοκιμαστεί αλλού θα πρέπει να είναι λογικά πιο αξιόπιστες
- Το συνολικό ρίσκο μειώνεται
  - Εάν μια ψηφίδα υπάρχει είναι πιο εύκολο να βρεθεί το κόστος χρήσης της παρά το κόστος ανάπτυξης της
- Αποδοτική χρησιμοποίηση ειδικών (specialists)
  - Ειδικοί μπορούν να αναπτύσσουν επαναχρησιμοποιούμενες ψηφίδες που περικλείουν την γνώση τους
- Κάποια πρότυπα μπορούν να ενσωματωθούν στις ψηφίδες
  - Π.χ. U.I. Standards, για την υλοποίηση μενού. Η χρήση τους αυξάνει την αξιοπιστία καθώς είναι πιο «γνωστά» στον χρήστη
- Ο χρόνος ανάπτυξης μειώνεται

# 2.3 Ανάπτυξη **με** επ. ψηφίδες



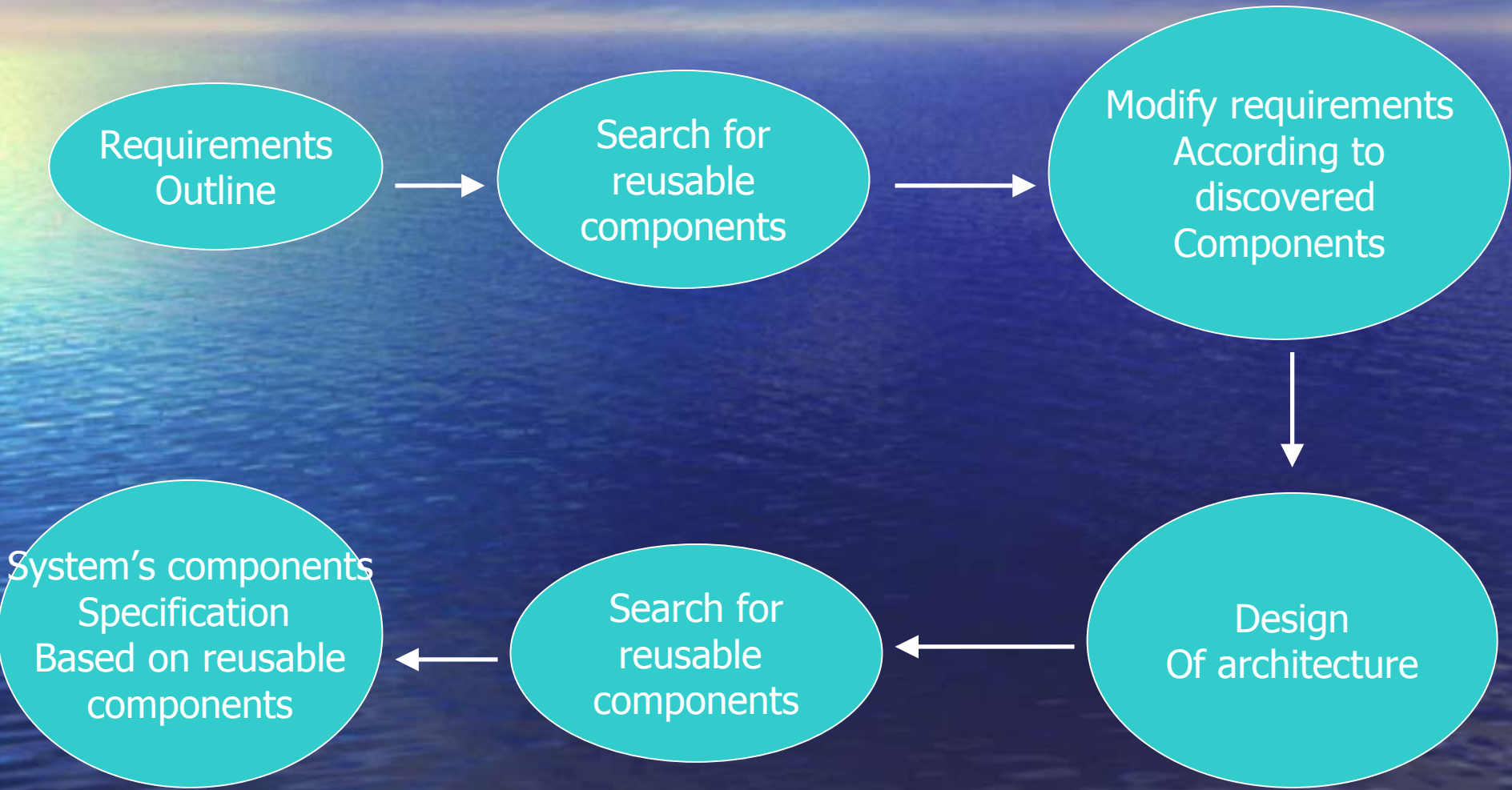
# 2.4 Απαιτήσεις για επαναχρησιμοποίηση

- Πρέπει να είναι δυνατόν να βρεθούν κατάλληλα επαναχρησιμοποιήσιμες ψηφίδες,π.χ. σε κάποια ΒΔ ψηφίδων
- Εκείνοι που πρόκειται να χρησιμοποιήσουν τέτοιες ψηφίδες θα πρέπει να τις καταλαβαίνουν και να πιστεύουν ότι θα καλύψουν τις ανάγκες τους
- Οι ψηφίδες θα πρέπει να συνοδεύονται από τεκμηρίωση που να περιγράφει τον τρόπο επαναχρησιμοποίησής τους καθώς και το πιθανό κόστος αυτής.

# 2.5 Ανάπτυξη οδηγούμενη από την λογική της επαναχρησιμοποίησης

- Αντί να σκεφτεί κανείς την επαναχρησιμοποίηση μετά τις προδιαγραφές του λογισμικού, εδώ προδιαγραφές και σχεδιασμός λαμβάνουν υπόψη την ύπαρξη επαναχρησιμοποιούμενων ψηφίδων και τροποποιούνται ανάλογα. Αυτό ωστόσο θα μπορούσε να προκαλέσει λιγότερο αποδοτικό σχεδιασμό
- Το πλεονέκτημα όμως είναι ότι θα μπορούσε να αυξήσει δραστικά την αναλογία επαναχρησιμοποιούμενων ψηφίδων, μειώνοντας κατακόρυφα το κόστος.
- Είναι κοινός τόπος στον σχεδιασμό ηλεκτρονικών, ηλεκτρικών και μηχανολογικών συστημάτων.

# 2.6 Ανάπτυξη οδηγούμενη από την λογική της επαναχρησιμοποίησης



# 2.7 Προβλήματα στην ανάπτυξη με επαναχρησιμοποίηση

- Δύσκολη η ποσοτικοποίηση κόστους και κέρδους από την ανάπτυξη με επαναχρησιμοποίηση. Οι ψηφίδες πρέπει να βρεθούν, να κατανοηθούν και πιθανώς να τροποποιηθούν. Αυτά έχουν κόστος.
- Τα σετ εργαλείων CASE δεν υποστηρίζουν ανάπτυξη με επαναχρησιμοποίηση
- Κάποιοι μηχανικοί προτιμούν να ξαναγράφουν παρά να επαναχρησιμοποιούν. Αυτό οφείλεται συνήθως στην εκπαίδευσή τους που στρέφεται κυρίως στην αυθεντική ανάπτυξη.
- Οι τρέχουσες τεχνικές ταξινόμησης, αρχειοθέτησης και ανάκτησης είναι ανώριμες. Έτσι το κόστος εύρεσης κατάλληλων επαναχ. ψηφίδων είναι υψηλό.

# 3.1 Ανάπτυξη λογισμικού **για** επαναχρησιμοποίηση

- Το πρόβλημα: Οι ψηφίδες συνήθως δεν είναι άμεσα επαναχρησιμοποιήσιμες αφού συνήθως είναι προσανατολισμένες προς τις απαιτήσεις του συστήματος για το οποίο πρωτοφτιάχτηκαν. Πρέπει να τροποποιηθούν.
- Η ανάπτυξη λογισμικού **για** επαναχρησιμοποίηση είναι μια διαδικασία ανάπτυξης που λαμβάνει υπάρχουσες ψηφίδες και στοχεύει να τις γενικεύσει και να τις τεκμηριώσει ώστε να είναι δυνατόν να επαναχρησιμοποιηθούν σε εφαρμογές

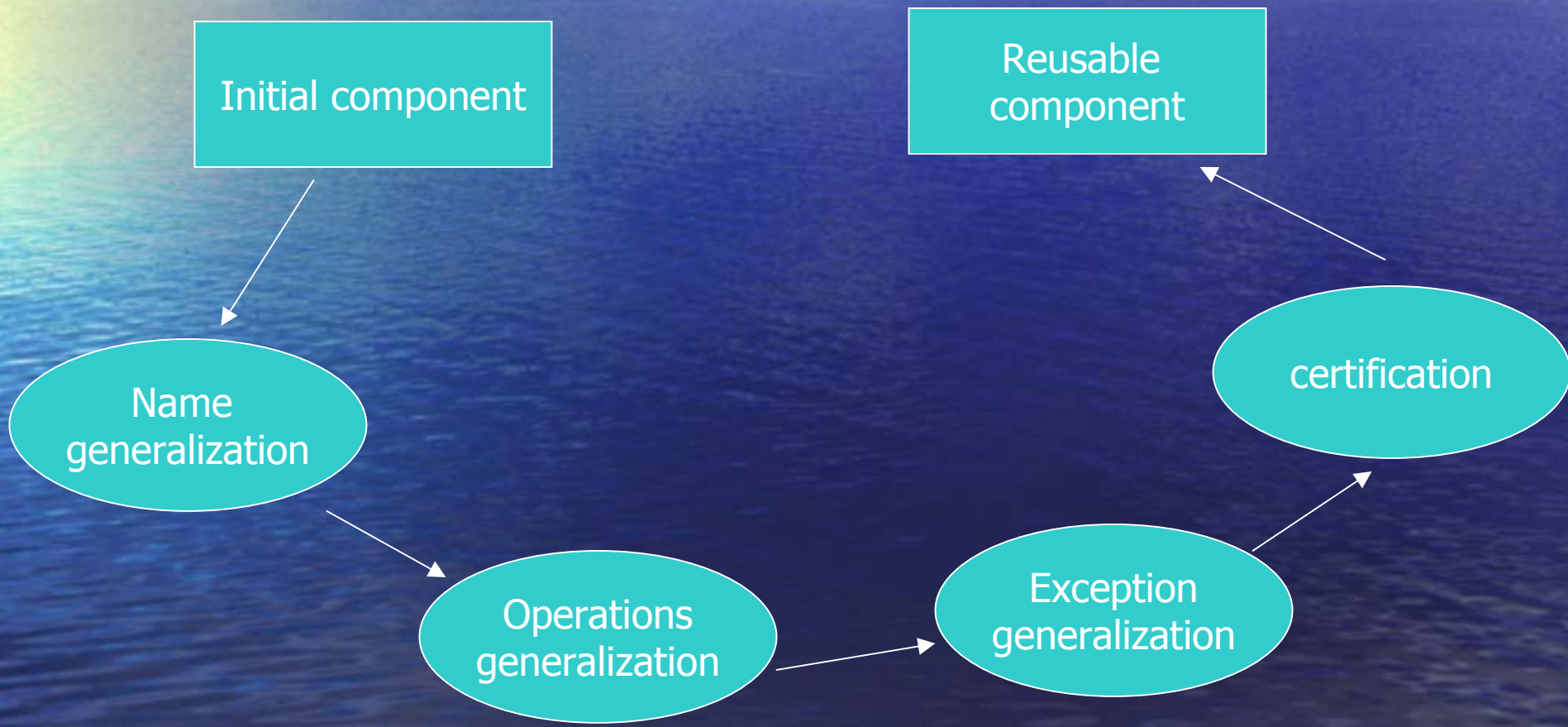
# 3.2 Ανάπτυξη **για** επαναχρησιμοποίηση

- Πιθανά προβλήματα-μειονεκτήματα:
  - Το κόστος ανάπτυξης επαναχρησιμοποιήσιμων ψηφίδων είναι υψηλό.
  - Οι γενικευμένες ψηφίδες μπορεί να λιγότερο αποδοτικές ως προς την χρήση χώρου και μπορεί να έχουν μεγαλύτερους χρόνους εκτέλεσης από αντίστοιχες συγκεκριμένες ψηφίδες.

# 3.3 Βασικά βήματα προσαρμογής ψηφίδων ώστε να είναι επαναχρησιμοποιήσιμες

- Γενίκευση ονομάτων
  - Τα ονόματα (π.χ. μεταβλητών) μιας ψηφίδας τροποποιούνται ώστε να μην αντικατοπτρίζουν αποκλειστικά μια συγκεκριμένη οντότητα της εφαρμογής
- Γενίκευση λειτουργιών
  - Π.χ. πρόσθεση λειτουργιών που προσδίδουν περαιτέρω λειτουργικότητα και αφαίρεση εκείνων που είναι αποκλειστικά για μια εφαρμογή.
- Γενίκευση εξαιρέσεων
  - Π.χ. αφαιρούνται οι εξαιρέσεις που αφορούν μια συγκεκριμένη εφαρμογή και προστίθεται διαχείριση εξαιρέσεων*
- Επιβεβαίωση ψηφίδας
  - Η ψηφίδα επικυρώνεται ως επαναχρησιμοποιήσιμη

# 3.4 Σχηματική αναπαράσταση διαδικασίας



# 3.5 Επαναχρησιμοποίηση για συγκεκριμένο πεδίο

- Πιθανόν, οι ψηφίδες να επαναχρησιμοποιηθούν κυρίως στο πεδίο των εφαρμογών για το οποίο αναπτύχθηκαν αρχικά. Γιατί ενσωματώνουν έννοιες και σχέσεις από το πεδίο αυτό με αποτέλεσμα να είναι cost-effective για αυτό.
- Η ανάλυση πεδίου ασχολείται με την κατανόηση πεδίων:
  - Προσπαθεί να ανακαλύψει τα στοιχειώδη τους χαρακτηριστικά
  - Με αυτά, οι ψηφίδες μπορούν να γενικευθούν για να είναι επαναχρησιμοποιήσιμες σε αυτό το πεδίο γενικά

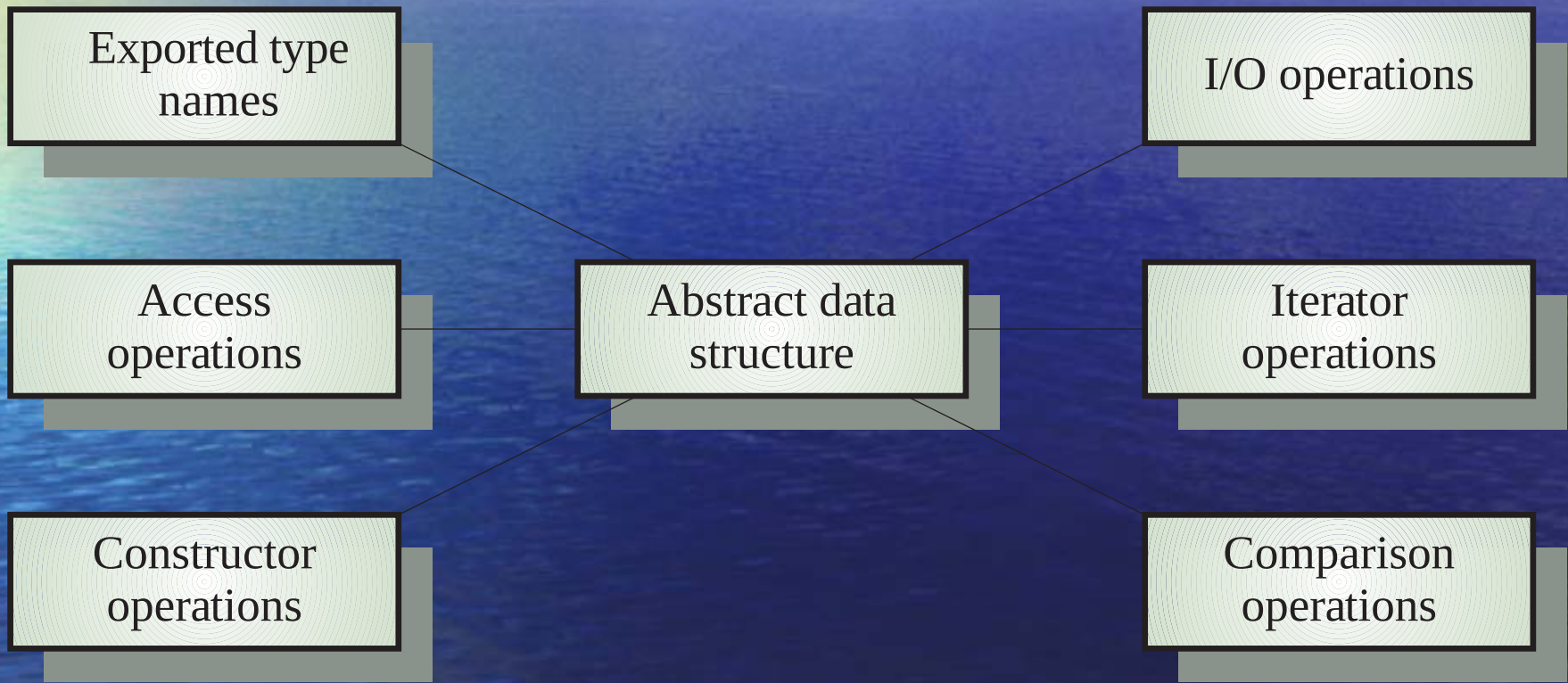
# 3.6 Παράδειγμα: το πεδίο των ADS

- Είναι ένα κατανοητό πεδίο εφαρμογών.
- Σημαντικό σαν υπόβαθρο για πολλά είδη συστημάτων λογισμικού
- Οι απαιτήσεις για επαναχρησιμοποιήσιμες AD structures έχουν εκδοθεί κατ' επανάληψη
- Έτσι έχει ανακαλυφθεί ένα σχήμα ταξινόμησης για τέτοιες επαναχ. Ψηφίδες (Booch, 1987)

# 3.7 Γενίκευση ADS (linked list)

- Για την παραγωγή της γενικ. ADS μπορεί να χρειάζεται πρόσθεση λειτουργιών σε μια τέτοια ψηφίδα για να καλύπτει όλο το πεδίο
- Μερικές από αυτές τις λειτουργίες:
  - Access ops, που ελέγχουν τιμές στοιχείων της λίστας
  - Constructor ops, που προσθέτουν/αφαιρούν στοιχεία
  - I/O ops, που γράφουν/διαβάζουν από μέσα αποθ.
  - Comparison ops, που συγκρίνουν στοιχεία/instances
  - Iterator ops, που επιτρέπουν τον έλεγχο ενός στοιχείου χωρίς την απομάκρυνσή του από την λίστα

# 3.9 Μοντέλο μιας επανα/σιμης ADS



# 3.10 Οδηγίες για επαναχρησιμοποιήσιμες ADS

- Υλοποίηση αυτών σαν κλάσεις (templated), για να μην υπάρχει πρόβλημα με το είδος των στοιχείων
- Παροχή λειτουργιών για δημιουργία και απόδοση instances
- Παροχή μηχανισμών που να δείχνουν αν μια λειτουργία ήταν επιτυχής
- Ελαχιστοποίηση της πληροφορίας που ορίζεται στο πλαίσιο της ψηφίδας
- Υλοποίηση λειτουργιών που μπορεί να αποτύχουν σαν procedures με επιστροφή δείκτη για το αποτέλεσμα
- Παροχή τελεστή ισότητας για σύγκριση structures
- Παροχή iterator που να επιτρέπει την επίσκεψη σε κάθε στοιχείο μιας συλλογής χωρίς την τροποποίηση του

# 3.11 Προσαρμογή ψηφίδων

- Μπορεί να χρειάζεται να προστεθεί επιπλέον λειτουργικότητα σε μια ψηφίδα προκειμένου αυτή να είναι επαναχρησιμοποιήσιμη
- Επίσης μη αναγκαία λειτουργικότητα μπορεί να αφαιρεθεί π.χ. για να βελτιωθεί η απόδοσή της ή να μειωθεί η απαίτηση χώρου
- Η υλοποίηση κάποιων λειτουργιών της μπορεί να χρειάζεται τροποποίηση. Π.χ. όταν οι αρχικές αποφάσεις για την γενίκευσή της ήταν λαθασμένες

# 4.1 Επαναχρησιμοποίηση, αντικείμενα και κληρονομικότητα

- Τα αντικείμενα είναι από φυσικού τους επαναχρησιμοποιήσιμα αφού μπορεί να είναι αυτόνομες οντότητες χωρίς εξωτερικές εξαρτήσεις. Π.χ. εμπεριέχουν την κατάσταση και τις συνοδεύουσες λειτουργίες
- Όσον αφορά την κληρονομικότητα, εκεί εξ' ορισμού έχουμε επαναχρησιμοποίηση όταν μια κλάση κληρονομεί ιδιότητες και λειτουργίες από μια υπερ-κλάση. Αυτές στην ουσία επαναχρησιμοποιούνται
- Το ίδιο συμβαίνει και με την πολλαπλή κληρονομικότητα

# 5.1 Επαναχρησιμοποίηση που βασίζεται σε γεννήτορες

- Οι γεννήτριες προγραμμάτων συμπεριλαμβάνουν επαναχρησιμοποίηση μοτίβων και αλγορίθμων, οι οποίοι είναι ενσωματωμένοι στον γεννήτορα και παραμετροποιούνται απλώς από το χρήστη
- Παράδειγμα: compilers*, είναι γεννήτορες προγραμμάτων όπου τα επαναχρησιμοποιούμενα μοτίβα είναι τεμάχια object-code που αντιστοιχούν σε εντολές της γλώσσας προγραμματισμού
- Αυτό το είδος επαναχρησιμοποίησης είναι πολύ αποδοτικό ως προς το κόστος, αλλά περιορίζεται σε ένα σχετικά μικρό πεδίο εφαρμογών.

# 6.1 Portability συστήματος

- Η μεταφερσιμότητα είναι μια ειδική περίπτωση επαναχ/σης όπου όλο το σύστημα επαναχρησιμοποιείται σε μια διαφορετική πλατφόρμα
- Η μεταφερσιμότητα ενός προγράμματος είναι το μέτρο της εργασίας που απαιτείται για να μπορέσει αυτό το πρόγραμμα να λειτουργήσει σε ένα νέο περιβάλλον
- Δύο όψεις:
  - Φυσική Μεταφορά του κώδικα (και των δεδομένων) σε άλλο σύστημα. Γενικά εύκολο
  - Προσαρμογή του προκειμένου να δουλέψει στο νέο περιβάλλον

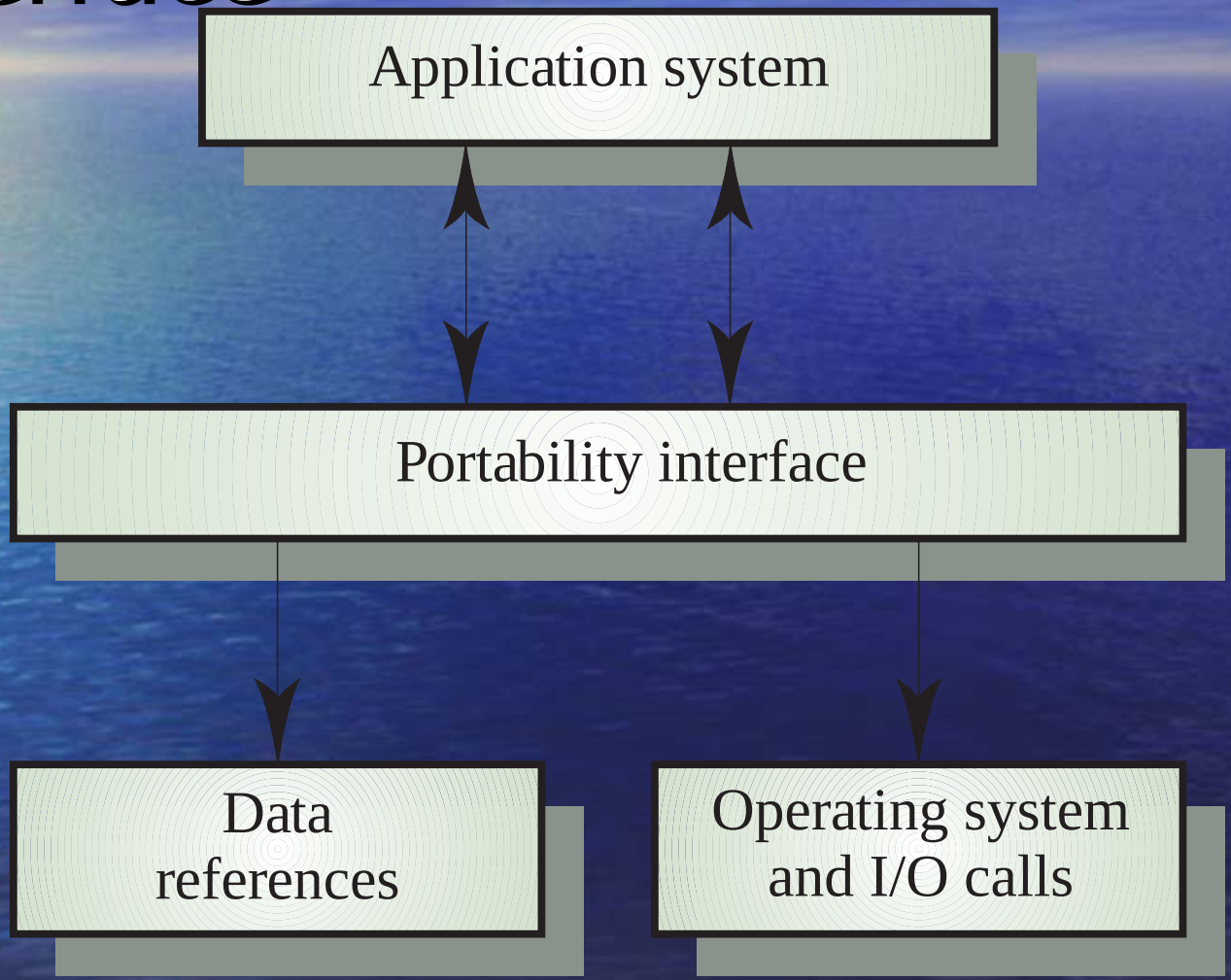
# 6.2 Εξαρτήσεις/προβλήματα στην μεταφερσιμότητα

- Εξάρτηση από την αρχιτεκτονική του υπολογιστή, π.χ. από την αναπαράσταση και οργάνωση της πληροφορίας
- Εξάρτηση από το ΛΣ, π.χ. τα χαρακτηριστικά του
- Εξάρτηση από ένα runtime υποστηρικτικό σύστημα
- Προβλήματα με τις libraries, π.χ. εξάρτηση από συγκεκριμένο σύνολο libraries

# 6.3 Ανάπτυξη με σκοπό την μεταφερσιμότητα

- Απομόνωση εκείνων των τμημάτων του λογισμικού που εξαρτώνται από interfaces με εξωτερικά προγράμματα. Αυτά τα interfaces πρέπει να υλοποιούνται σαν σύνολο ADTs ή αντικειμένων
- Καθορισμός ενός portability interface για την απόκρυψη των χαρακτηριστικών και της αρχιτεκτονικής του ΛΣ
- Για να μεταφέρουμε το πρόγραμμα τότε, είναι απαραίτητο να ξαναγράψουμε μόνο τον κώδικα πίσω από αυτό το portability interface

# 6.4 Παράδειγμα Portability Interface



# 6.5 Εξάρτηση από την αρχιτεκτονική της πλατφόρμας

- Το πρόγραμμα πρέπει να βασίζεται στο σχήμα αναπαράστασης της πληροφορίας που υποστηρίζεται από την συγκεκριμένη αρχιτεκτονική
- Συνήθη προβλήματα:
  - Η ακρίβεια των πραγματικών
  - Το bit ordering στην αναπαράσταση αριθμών
- Αντιμετωπίζουμε την κατάσταση πάλι με ADTs. Πιθανή υποστήριξη διαφορετικών αναπαραστάσεων*.

# 6.6 Εξάρτηση από το ΛΣ

- Το πρόγραμμα βασίζεται στη χρήση συγκεκριμένων κλήσεων του ΛΣ, όπως αυτές που υποστηρίζουν την διαχείριση των διεργασιών
- Επίσης το πρόγραμμα βασίζεται στη συγκεκριμένη οργάνωση του συστ. αρχείων που υποστηρίζει το κάθε ΛΣ

# 6.8 Standards: Ένα τρόπος διευκόλυνσης για portability

- Τα Standards είναι μια είδη συμφωνίας - πρότυπα που μειώνουν την διαφορετικότητα ανάμεσα στα συστήματα λογισμικού
- Η ανάπτυξη τους ειδικά κατά δεκαετία του 80, έκανε την μεταφερσιμότητα πιο εύκολη από πριν
- Καθώς αυτά αναπτύσσονται περαιτέρω, γίνεται δυνατό κομμάτια ενός προγράμματος να μπορούν να τρέχουν σε τελείως διαφορετικές πλατφόρμες

# 6.9 Υπάρχοντα Standards

- Programming language standards
  - Ada, Pascal, C, C++, FORTRAN.
- Operating system standards
  - UNIX, MS-DOS (de-facto standard), MS Windows
- Networking standards
  - TCP/IP protocols, X400, X500, Sun NFS, OSI layered model. HTML, WWW
- Window system standards
  - X-windows. Motif toolkit