



Εθνικό και Καποδιστριακό Πανεπιστήμιο Αθηνών  
Τμήμα Πληροφορικής & Τηλεπικοινωνιών

# Προηγμένες Μέθοδοι Προγραμματισμού

ΠΜΣ 2022-23 (M135.CS1E, M135.CS23B, M135.IC1E, παλαιό:  
M117)

Εισαγωγή

Δρ. Κώστας Σαΐδης ([saiko@di.uoa.gr](mailto:saiko@di.uoa.gr))

# Στόχος του μαθήματος

- Κατανόηση και εξοικοίωση με τις προηγμένες μεθόδους και τεχνικές προγραμματισμού που χρησιμοποιούνται στην πράξη

# Το πλαίσιο

- Έμφαση στα δύο από τα πλέον διαδεδομένα runtime environments που χρησιμοποιούνται από τη βιομηχανία
  - JVM (~Android, Java, Scala, Kotlin, Groovy, etc.)
  - V8 (Web Browser, Node.js, Javascript/ECMAScript, TypeScript, etc.)

# Κίνητρο

- Ποιος θα πάρει τις σημαντικές σχεδιαστικές αποφάσεις (major design decisions) για το λογισμικό;
- Που θα επηρεάσουν και θα καθορίσουν την επιτυχία του
  - Θα αντιμετωπίσουν, δηλαδή, την ουσιώση πολυπλοκότητα (essential complexity) του ζητουμένου
  - Θα επιτύχουν τη μέγιστη δυνατή παραγωγικότητα της ομάδας
  - Θα εγγυηθούν τη βιωσιμότητα, επεκτασιμότητα, διαχειρισιμότητα του αποτελέσματος

# Σχεδιαστικές αρχές λογισμικού

- Κατά την ανάλυση ασχολούμαστε με το **τι** θα κάνει το λογισμικό (ποια είναι η επιθυμητή συμπεριφορά;).
- Στη σχεδίαση, η προσοχή μεταφέρεται στο **πώς** θα το κάνει (πώς θα επιτύχουμε την επιθυμητή συμπεριφορά;).

# Σχεδίαση λογισμικού

Θα πρέπει να εντοπίσουμε:

- Ποια συστατικά αποτελούν την εφαρμογή μας.
- Πώς αυτά συσχετίζονται μεταξύ τους.
- Πώς διατάσσονται.
- Ποια είναι η εσωτερική τους δομή.

Οι απαντήσεις στα ερωτήματα αυτά, αποτελούν το **σχέδιο του λογισμικού (software design)**.

The most difficult design task is to find the most appropriate decomposition of the whole into a module hierarchy, minimizing function and code duplications.

N. Wirth

# Κι η αρχιτεκτονική του λογισμικού;

## Περί τίνος πρόκειται

Η λήψη των θεμελιωδών δομικών και σχεδιαστικών αποφάσεων για το λογισμικό που είναι δύσκολο (ακριβό) να αλλάξουν αφού υλοποιηθούν.

Συνοδεύει:

- Ανάλυση απαιτήσεων
- Σχεδιασμό



# Σχεδιασμός λογισμικού

Ένα σύνθετο πρόβλημα:

- Ποιος είναι ο καλύτερος τρόπος μετάβασης από τις προδιαγραφές στο σχέδιο του λογισμικού;
- Με πόσα σχέδια μπορεί να υλοποιηθεί κάθε προδιαγραφή και ποιο είναι το «καλύτερο»;
- Πόσο λεπτομερής είναι μια «καλή» αποτύπωση του σχεδίου και τι περιλαμβάνει αυτή;
- Πώς διασφαλίζεται η ποιότητα του λογισμικού;

# Ποιοτικά ζητούμενα σχεδιασμού

- Performance - Scalability
- Maintainability - Extensibility
- Security - Safety
- Robustness - Fault-tolerance
- Usability - Reliability

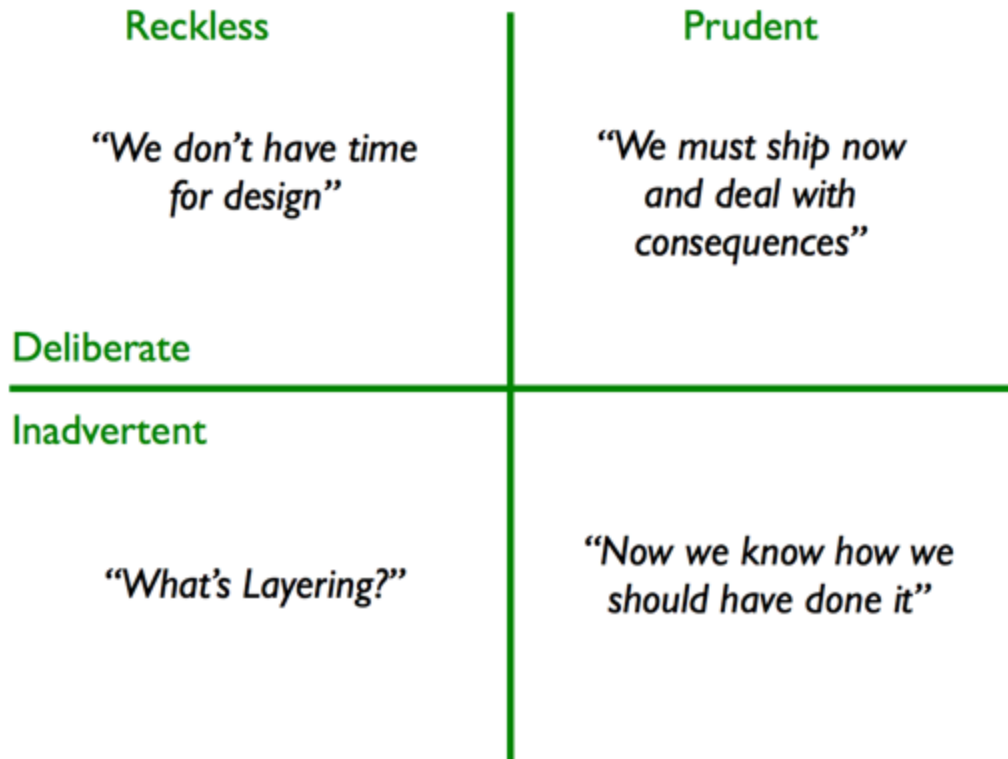
# Σχεδιασμός = Συμβιβασμός

- Συνήθως δεν είναι εφικτό να γίνουν όλα καλά.
- Επιλογή των επιθυμητών trade-offs.

# Τεχνικό χρέος

- Το κόστος της πρόσθετης προσπάθειας / δουλειάς που θα απαιτηθεί
- αν επιλέξουμε μια "εύκολη" και "γρήγορη" λύση (quick & dirty)
- αντί για τη συνολικά καλύτερη λύση
- στο κάθε σχεδιαστικό πρόβλημα.

# Τέσσερα είδη



By [Martin Fowler](#)

# Το κλασικότερο trade-off

## Efficiency vs Abstraction

Programmers have spent far too much time worrying about efficiency in the wrong places at the wrong times; premature optimization is the root of all evil.

Donald Knuth

# Αφαίρεση (Abstraction)

Θεμελιώδης έννοια

- "Η εννοιολογική διαδικασία κατά την οποία προκύπτουν γενικοί κανόνες από την εξέταση επιμέρους παραδειγμάτων." (Wikipedia)
- Χρησιμοποιείται σε πολλές επιστήμες.
- Αποτελεί το κύριο εργαλείο σχεδιασμού.

# Αφαίρεση

Αναπαράσταση ενός ουσιώδους χαρακτηριστικού του λογισμικού χωρίς τις δευτερεύουσες λεπτομέρειες.



# Παράδειγμα

```
interface Item {  
    String getId()  
    String getName()  
    String getType()  
    double getPrice()  
    ...  
}  
  
interface Datastore {  
    Item load(String id)  
    void save(Item item)  
}
```

# Βελτίωση (Refinement)

- Συμπληρωματική διαδικασία της αφαίρεσης
- Τμηματική προσαρμογή των αφαιρέσεων σε νέες απαιτήσεις, περιορισμούς ή αποφάσεις

# Παράδειγμα

```
interface Datastore {
    boolean exists(String id)
    void load(String id, Consumer<Result> itemConsumer)
    void save(Item item, Consumer<Result> resultConsumer)
}

interface Consumer<T> {
    accept(T t);
}

interface Result {
    boolean ok();
    Item item();
    Throwable error();
}
```

Γιατί το αλλάξαμε;

# Refactoring

- Η διαδικασία επαναπροσαρμογής του κώδικα ως τμήμα κάποιας βελτιωτικής απόφασης
- Αυτοματοποιείται από πολλά IDEs
- Σχετικό/συμπληρωματικό με το Refinement

# Τμηματοποίηση (modularity)

- Αναλύουμε ένα πολύπλοκο σύστημα σε επιμέρους απλούστερα τμήματα
- Σχεδιάζουμε ξεχωριστά το ένα τμήμα από το άλλο
- Συνθέτουμε ξανά τα τμήματα και τις αλληλεπιδράσεις τους σε ένα ενιαίο σύνολο
- Με σκοπό τη διευκόλυνση της υλοποίησης, της συντήρησης και της εξέλιξης του λογισμικού

# Απόκρυψη πληροφορίας (information hiding)

- Η απόκρυψη των χαρακτηριστικών που μπορεί να αλλάξουν στο λογισμικό (σχεδιαστικών αποφάσεων, λεπτομερειών υλοποίησης) από τα άλλα τμήματα, ώστε να ελαχιστοποιηθούν οι αλλαγές που απαιτούνται αν/όταν προκύψει η αλλαγή
- Παροχή αφαιρέσεων --π.χ. μοντέλων δεδομένων (data models), διεπαφών (interfaces) κ.ά- για την επικοινωνία μεταξύ των τμημάτων

# Συναφείς έννοιες / αρχές

- Ενθυλάκωση (encapsulation)
- Επιμερισμός αρμοδιοτήτων (separation of concerns)
- Σύζευξη & συνεκτικότητα (coupling & cohesion)

# Ενθυλάκωση (encapsulation)

- Διαχωρισμός της δομής από τη συμπεριφορά ενός συστατικού
- Προστασία ενός συστατικού από τη μετάβασή του σε μη έγκυρη κατάσταση
- Μπορεί να θεωρηθεί ως τεχνική υλοποίησης της αρχής της απόκρυψης πληροφορίας



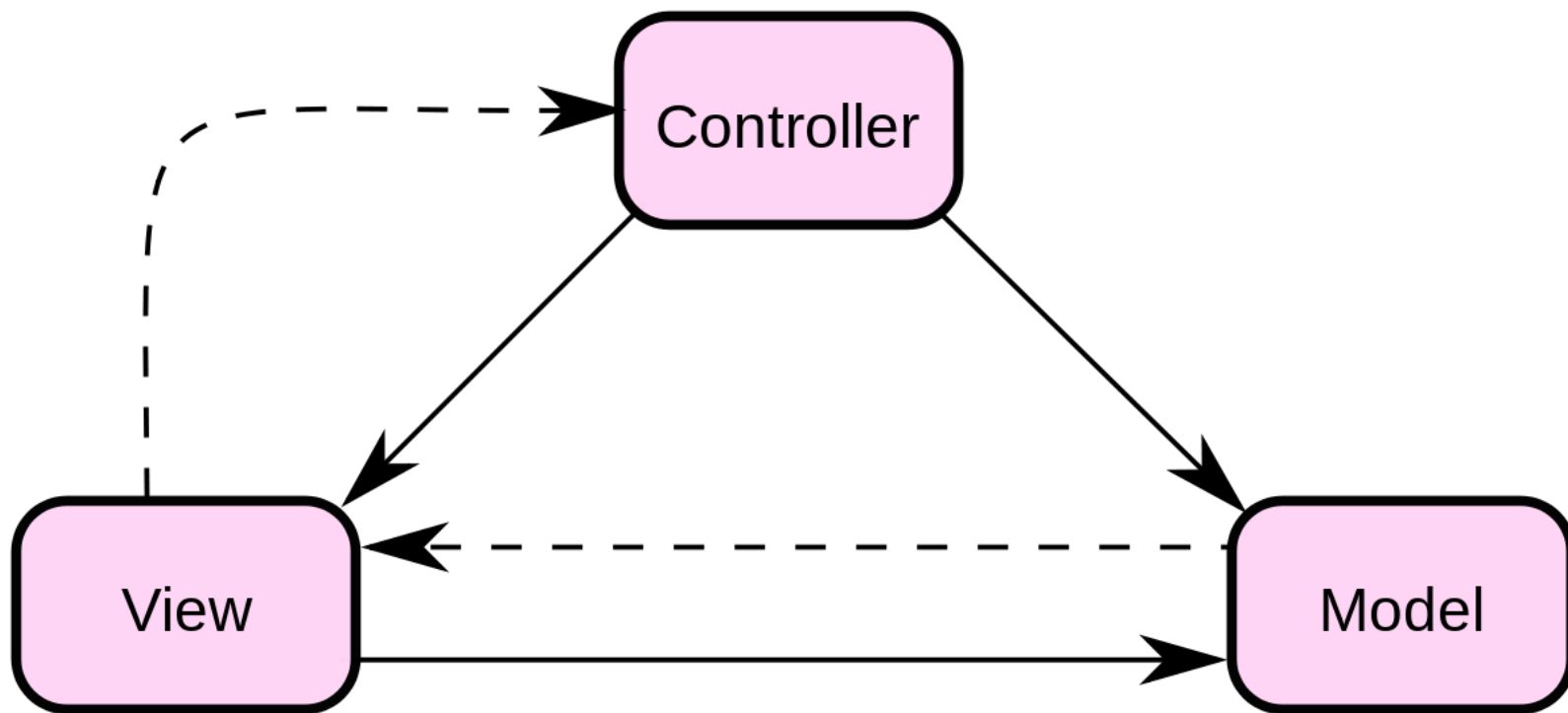
# Παράδειγμα

```
class MySQLDatastore implements Datastore {
    private Connection con //encapsulation
    public boolean exists(String id) {
        ResultSet rs = con.execute("select 1 from items where item = $", id)
        return !rs.isEmpty()
    }
    ...
}
```

# Επιμερισμός αρμοδιοτήτων (separation of concerns)

- Κάθε τμήμα του λογισμικού έχει μία αρμοδιότητα και επικεντρώνεται στην επίλυση ενός ξεχωριστού ζητήματος (concern)
- Αρχιτεκτονικά επίπεδα (επίπεδο παρουσίασης, επίπεδο επιχειρησιακής λογικής, επίπεδο πρόσβασης δεδομένων)

# Παράδειγμα



# Σύζευξη & Συνεκτικότητα

## Σύζευξη (coupling)

Ο βαθμός της αλληλεξάρτησης μεταξύ δύο συστατικών (κλάσεων, modules, κλπ)

## Συνεκτικότητα (cohesion)

Ο βαθμός με τον οποίο τα στοιχεία που ανήκουν σε ένα συστατικό σχετίζονται λειτουργικά (ορθώς συνανήκουν στο ίδιο συστατικό)

# Ζητούμενο

Χαλαρή σύζευξη (loose coupling) μεταξύ των συστατικών

Υψηλή συνεκτικότητα (high cohesion) "μέσα" σε κάθε συστατικό

# Επαναχρησιμοποίηση (reuse)

- Η υλοποίηση του λογισμικού δεν πρέπει να ανακαλύπτει κάθε φορά εκ νέου τον τροχό
- Οι καλές αφαιρέσεις και τα καλώς διαχωρισμένα συστατικά είναι εύκολο να επαναχρησιμοποιηθούν

# Παράδειγμα

## Map/Reduce pattern

```
var animals = ["dog", "cat", "fish"]
var len = function(s) { return s.length; }
var sum = function(a, b) { return a+b; }
animals.map(len).reduce(sum, 0); //10
```

```
var records = //list of person records;
var getAge = function(person) { return currentYear - person.birthYear; }
records.map(getAge).reduce(sum, 0) / records.length; //avg age of people
```

# Πρόσθετες σχεδιαστικές αρχές (design principles)

Κριτήρια για να δομήσουμε και να υλοποιήσουμε τις αφαιρέσεις (abstractions)



# Μόνο μια φορά (Once and Only Once)

- Αρχή του Extreme Programming (XP)
- Each and every declaration of behavior should appear Once and Only Once.

# Μην επαναλαμβάνεσαι (DRY - Don't Repeat Yourself)

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."

A. Thomas, D. Hunt

# DRY vs WET

- WET: We Enjoy Typing
- WET: Waste Everybody's Time

# Σύνολο αρχών S.O.L.I.D.

- Single responsibility (S)
- Open/closed (O)
- Liskov substitution (L)
- Interface segregation (I)
- Dependency inversion (D)

# Μοναδική ευθύνη (Single responsibility)

- Κάθε κλάση/συστατικό πρέπει να έχει μία και μοναδική ευθύνη.
- A class should have only one reason to change.

# Ανοικτό/κλειστό (Open/closed)

- Κάθε κλάση/συστατικό πρέπει να είναι ανοικτή σε επεκτάσεις (π.χ. προσθήκη νέων πεδίων ή μεθόδων).
- Κάθε κλάση/συστατικό πρέπει να είναι κλειστή και οριοθετημένη (έτοιμη προς χρήση από τρίτα συστατικά).

# Δυνατότητα αντικατάστασης (Liskov substitution)

- Αν το **S** είναι υποτύπος του **T** τότε όλα τα αντικείμενα του δεύτερου θα πρέπει να μπορούν να αντικατασταθούν με αντικείμενα του πρώτου χωρίς να αλλοιωθεί κανένα από τα επιθυμητά χαρακτηριστικά του συστήματος.
- Strong behavioral subtyping.

# Επιμερισμός διεπαφών (Interface segregation)

- Κανένα συστατικό δεν πρέπει να εξαρτάται από μεθόδους που δε χρησιμοποιεί.
- Χρήση μικρών και διαφορετικών interfaces για τη θέσπιση των διεπαφών μεταξύ συστατικών.



# Ανιστροφή εξαρτήσεων (Dependency inversion)

- Τα υψηλού επιπέδου συστατικά δεν πρέπει να εξαρτώνται από τα χαμηλού επιπέδου συστατικά. Και τα δύο θα πρέπει να εξαρτώνται από κοινές αφαιρέσεις.
- Οι αφαιρέσεις δεν πρέπει να εξαρτώνται από λεπτομέρειες. Οι λεπτομέρειες θα πρέπει να εξαρτώνται από τις αφαιρέσεις.

# Παράδειγμα

## Strong coupling (bad)

```
class DataAccessLayer {  
    // BAD - strong coupling of DataAccessLayer to MySQLDatastore  
    private MySQLDatastore store = new MySQLDatastore(...)  
}
```

## Dependency inversion (injection)

```
class DataAccessLayer {  
    // The Datastore implementation (whatever that is) is "injected" to the component by the DI mechanism  
    @Inject private Datastore store  
}
```

# Περιεχόμενα διαλέξεων (I)

- Object-Oriented Programming
  - Class-based / prototype-based
  - Encapsulation
  - Inheritance
  - Polymorphism
- Functional Programming (lambdas, closures)
- Design Patterns
- Memory management & Gargage collection

# Περιεχόμενα διαλέξεων (II)

- Meta-programming
  - Compile-time / Runtime
  - Reflection
  - Proxies
  - Mocks & Stabs
  - Meta-Object Protocols
  - Aspect-Oriented Programming
- Concurrency (Threads/Workers, Futures, Promises)
- Flows/Streams, Events, Observables