HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

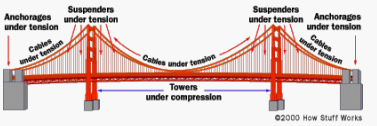
1

CSD

# ΕΝΟΤΗΤΑ 2

## *ΑΡΧΙΤΕΚΤΟΝΙΚΗ ΣΧΕΔΙΑΣΗ ΚΑΙ ΟΙΚΟΓΕΝΕΙΕΣ ΑΡΧΙΤΕΚΤΟΝΙΚΩΝ*

*Αριθμός διαλέξεων 2*






2

CSD

## Ορισμός (1/5)

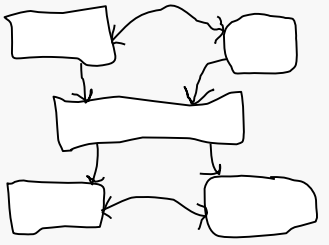
- *Μία αρχιτεκτονική παρέχει*:
  1. σύμπλεγμα αλληλεπιδρώντων τμημάτων με θεμελιώδη ρόλο στη λειτουργία του συστήματος
     - component structure
  2. περιγραφή του λειτουργικού ρόλου κάθε τμήματος
     - functional role description
  3. χαρακτηριστικά αλληλεπίδρασης μεταξύ των τμημάτων
     - inter-component interaction
  4. περιγραφή βασικών κύκλων λειτουργίας και ροής ελέγχου μεταξύ των τμημάτων
     - functional cycles and control flow
  5. σενάρια λειτουργίας που σχετίζονται άμεσα με το τι εξυπηρετεί το σύστημα
     - functional scenarios



3

CSD

## Ορισμός (2/5)



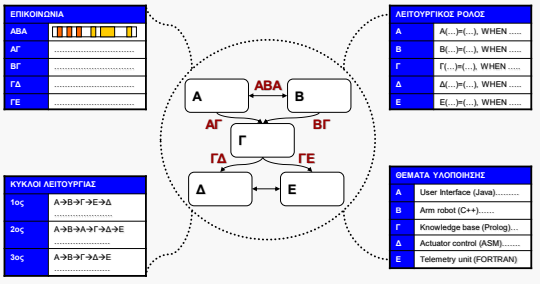
*this is not an architecture...*



4

CSD

## Ορισμός (3/5)




***…this is an architecture***



5

CSD

## Ορισμός (4/5)

- Γενικότερα αρχιτεκτονική είναι:
  - μία λεπτομερής εικόνα της δομής ενός συστήματος ***όταν αυτό λειτουργεί*** αποτελούμενη από αλληλεπιδρώντες οντότητες
    - Αυτές λέγονται τμήματα - ***components***
  - συνήθως ακολουθεί περεταίρω εσωτερική αρχιτεκτονική ανάλυση των τμημάτων
  - αποτυπώνει πάντοτε τους κυρίαρχους ρόλους των τμημάτων χωρίς να αγνοεί κανέναν
    - Τι σκοπό επιτελεί ένα τμήμα - ***roles***
  - καθώς και τις μεταξύ τους σχέσεις, εξαρτήσεις και τρόπο συνεργασίας
    - Πως αλληλεπιδρούν - ***interactions***



6

CSD

## Ορισμός (5/5)

- Η επικέντρωση στην αρχιτεκτονική σχεδίαση οδηγείται από την ανάγκη να μειώσουμε την πολυπλοκότητας κατασκευής ενός συστήματος μέσω:
  - **αφαίρεσης - *abstraction***
    - Δηλαδή αναβάλλουμε την εξειδίκευση κάθε είδους για αργότερα και χρησιμοποιούμε κάποιες γενικά αποδεκτές έννοιες, συμβολισμούς ή μεταφορές (πχ, client-server, n-tier, layers)
  - ***διαχωρισμού ευθυνών - separation of concerns***
    - Δηλαδή μοιράζουμε τις λειτουργικές ευθύνες σε διαφορετικά τμήματα ώστε να είναι εφικτή η ανάθεση καθενός από αυτά σε διαφορετικές ομάδες σχεδίασης και υλοποίησης
- Ωστόσο δεν υπάρχει κάποιος τυποποιημένος και καθολικά αποδεκτός ορισμός της αρχιτεκτονικής
  - …χωρίς αυτό να είναι και ουσιώδες πρόβλημα



7

CSD

## Περιεχόμενα

- Ορισμός
- ***Ρόλος στην σχεδίαση***
- Γρήγορος προσδιορισμός
- Επίπεδα αρχιτεκτονικής
- Βασικά αρχιτεκτονικά μοντέλα
- Στοιχεία αρχιτεκτονικής σχεδίασης



8

CSD

## Ρόλος στην σχεδίαση (1/6)

- ***Ο όγκος των συστημάτων και κατά συνέπεια η πολυπλοκότητα τους αυξάνεται συνεχώς***
- Η πρόκληση έχει μετατεθεί από το παραδοσιακό πεδίο των αλγορίθμων και των δομών δεδομένου
- Το νέο απαιτητικό πρόβλημα είναι η βέλτιστη ***σχεδίαση και οργάνωση της κατασκευής*** ενός συστήματος

$$\text{Programs} = \int_{\text{architecture}} \underbrace{\text{algorithms} + \text{data structures}}_{\text{any component } i}$$



9

CSD

## Ρόλος στην σχεδίαση (2/6)

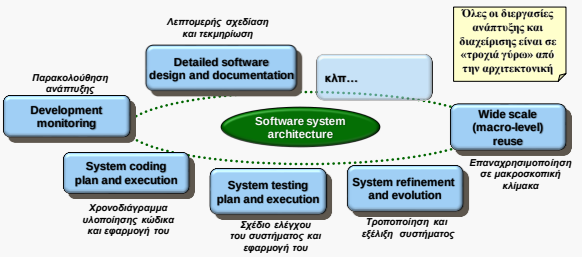
- Η αρχιτεκτονική αποτελεί το ***σχεδιαστικό χάρτη*** και επηρεάζει όλες τις ενέργειες ανάπτυξης που πρόκειται να ακολουθήσουν
- Μόλις γεννηθεί, αποτελεί το αρχικό ***κοινό όραμα μεταξύ των διαφόρων παράλληλων φάσεων***, πάνω στο οποίο οργανώνονται οι επιμέρους στρατηγικές
- Ο τρόπος με τον οποίο ***γεννιέται βέλτιστα*** δεν είναι τεκμηριωμένος η οροθετημένος
- Υπάρχουν όμως τεχνικές για γρήγορη μελέτη και προσδιορισμό που είναι καλύτερα να ακολουθεί κάποιος



10

CSD

## Ρόλος στην σχεδίαση (3/6)



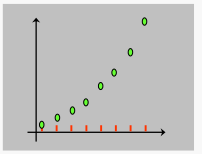



11

CSD

## Ρόλος στην σχεδίαση (4/6)

- Η εμβέλεια και το σχετικό ***κόστος των μετατροπών στο λογισμικό*** ποικίλει ανάλογα με το επίπεδο στο οποίο εφαρμόζονται
- Στην διπλανή λίστα, το κόστος αυξάνεται με μη γραμμικό ρυθμό



1. Line of code
2. Block
3. Function (internal)
4. .Function (API)
5. Main data type
6. Class
7. Package / Component
8. Architecture

Μη γραμμική αύξηση του κόστους τροποποιήσεων

***Η αιτία είναι ότι κάθε γραμμή του πίνακα αντιπροσωπεύει και ένα πολλαπλάσιο σύνολο γραμμών κώδικα από την προηγούμενη της***

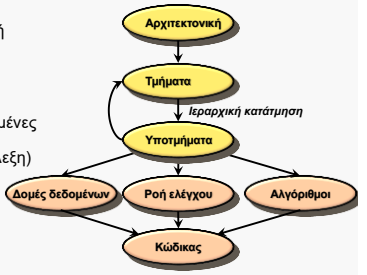


12

CSD

## Ρόλος στην σχεδίαση (5/6)

- Η βασική αρχιτεκτονική σχεδίαση έπεται από οργανωμένα, διαδοχικά ή παράλληλα, χαμηλότερα επίπεδα σχεδίασης
- Στην διαδικασία αυτή ακολουθούνται συγκεκριμένες, εξειδικευμένες ανά περίπτωση τεχνικές σχεδίασης (επόμενη διάλεξη)
- **Στο bottom-up development η αρχιτεκτονική εξειδικεύεται παράλληλα με τον κώδικα**






13

CSD

## Ρόλος στην σχεδίαση (6/6)

- Η αρχιτεκτονική σχεδίαση θεμελιώνει τα εξής:
  - δομή και ονομασία φακέλων και αρχείων
    - *Να ακολουθούν την ονοματολογία των τμημάτων με ιεραρχία αντίστοιχη της αρχιτεκτονικής*
  - αριθμό packages / projects / modules και την εσωτερική τους οργάνωση σε folders / filters του IDE
    - *Να είναι επίσης αντίστοιχη της αρχιτεκτονικής*
  - εξαρτήσεις υλοποίησης μεταξύ των packages
    - *Εξάρτηση στην αρχιτεκτονική ⇒ Εξάρτηση στον κώδικα*



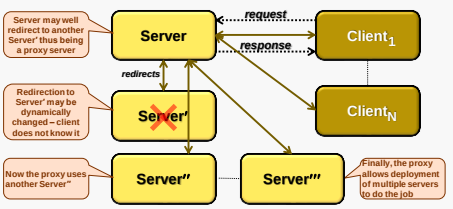
14

CSD

## Παράδειγμα

- *Client – Server* architecture

Φαίνεται πόσο η αρχιτεκτονική επηρεάζεται και αλλάζει με νέες η μεταβαλλόμενες λειτουργικές απαιτήσεις






15

CSD

## Περιεχόμενα

- Ορισμός
- Ρόλος στην σχεδίαση
- *Γρήγορος προσδιορισμός*
- Επίπεδα αρχιτεκτονικής
- Βασικά αρχιτεκτονικά μοντέλα
- Στοιχεία αρχιτεκτονικής σχεδίασης



16

CSD

## Γρήγορος προσδιορισμός (1/5)

- Το ποιο κοινό ερώτημα είναι: «*πως αρχίζουμε να μελετάμε και να προσδιορίζουμε την αρχιτεκτονική*»
- Συνήθως ο τρόπος χειρισμού είναι οπτικός με πρώτη έμφαση στη συνδεσμολογία ως τρόπο αναπαράστασης
- Η συνδεσμολογία αυτή προσφέρει εποπτικό έλεγχο των τμημάτων ενώ αποτυπώνει εύκολα τα δύο θεμελιώδη χαρακτηριστικά:
  - *ιεραρχική κατάτμηση - hierarchical decomposition*
  - *λειτουργική συνέργια - operational synergy*



17

CSD

## Γρήγορος προσδιορισμός (2/5)

- Στα πρώτα στάδια η αρχιτεκτονική αποτελείται από βασικά τμήματα σε μακροσκοπική αποτύπωση και είναι αρκετά ρευστή
- Η σχεδίαση βασίζεται πάντα στην ανάλυση γενικών λειτουργικών ρόλων και όχι σε κλάσεις ή συναρτήσεις ή οτιδήποτε σχετικό με γλώσσα

☒ ***δε σκεφτόμαστε βάση κάποιας γλώσσας και δε σχεδιάζουμε την υλοποίηση του κώδικα***

☑ ***αλλά αποτυπώνουμε τη δομή ενός συστήματος σε λειτουργία ως συνεργαζόμενα αντικείμενα***



18

CSD

## Γρήγορος προσδιορισμός (3/5)

- Ως σχεδιαστικό στοιχείο θα υποστεί σημαντικές αλλαγές και βελτιώσεις μέχρι να παγιωθεί, ενώ θα πρέπει πάντοτε να εξασφαλίζεται:
  - Η ύπαρξη τεκμηριωμένης λογικής των μετατροπών - όχι απλώς αλλαγές επειδή δεν μας ικανοποιεί «οπτικά» το αποτέλεσμα
  - Η χρήση κάποιας μεθόδου που επιτρέπει γρήγορες και εύκολες μετατροπές
- Μία τεχνική που μπορείτε να χρησιμοποιήσετε είναι μία τροποποίησης της τεχνικής CRC cards (*extreme programming*) για αρχιτεκτονική σχεδίαση
  - Classes - Responsibilities - Collaborators *CRC cards*
  - ***Roles – Responsibilities – Collaborators***
  - Διαβάστε το http://c2.com/doc/oopsla89/paper.html

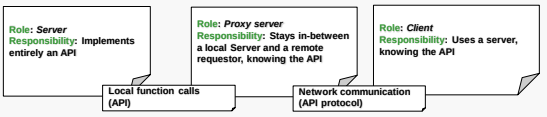


19

CSD

## Γρήγορος προσδιορισμός (4/5)

- η τεχνική αυτή δεν είναι αυστηρή και τυποποιημένη
- η μεγάλη αξία της είναι η ευελιξία της και η ταχύτητα σχεδίασης
- βασίζεται σε σχεδίαση με κάρτες που «συνδέονται» μεταξύ τους βάσεις αρχιτεκτονικών κανόνων
- δε χρειάζεται όλες οι κάρτες να είναι αυστηρά CRC style

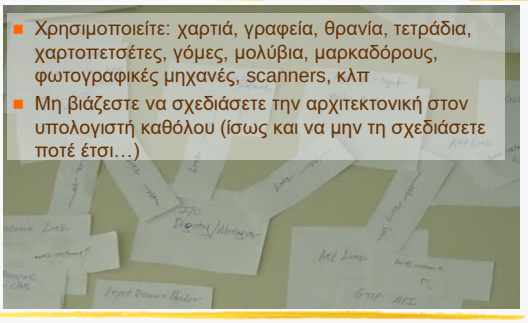





20

CSD

## Γρήγορος προσδιορισμός (5/5)



- Χρησιμοποιείτε: χαρτιά, γραφεία, θρανία, τετράδια, χαρτοπετσέτες, γόμες, μολύβια, μαρκαδόρους, φωτογραφικές μηχανές, scanners, κλπ
- Μη βιάζεστε να σχεδιάσετε την αρχιτεκτονική στον υπολογιστή καθόλου (ίσως και να μην τη σχεδιάσετε ποτέ έτσι…)



21

CSD

## Περιεχόμενα

- Ορισμός
- Ρόλος στην σχεδίαση
- Γρήγορος προσδιορισμός
- ***Επίπεδα αρχιτεκτονικής***
- Βασικά αρχιτεκτονικά μοντέλα
- Στοιχεία αρχιτεκτονικής σχεδίασης



22

CSD

## Επίπεδα αρχιτεκτονικής (1/3)

- *Macro-architecture*
  - Αποτελεί την συνολική / ευρύτερη αρχιτεκτονική του συστήματος και χαρακτηρίζει δομικά το σύστημα
  - Η εμβέλειά τους περιορίζεται κυρίως σε συγκεκριμένες κάθε φορά κατηγορίες συστημάτων
  - Δεν υπάρχει κριτήριο ως προς το μέγεθος των συστημάτων που αντιπροσωπεύει μία αρχιτεκτονική
  - Κάθε αρχιτέκτονας του λογισμικού πρέπει να γνωρίζει όλες τις σχετικές macro-architectures
  - Μπορεί ωστόσο να εμφανιστεί στην ανάλυση ενός συγκεκριμένου τμήματος (δηλ. όχι σε macro επίπεδο)



23

CSD

## Επίπεδα αρχιτεκτονικής (2/3)

- *Micro-architecture*
  - Αποτελούν αρχιτεκτονικές λύσεις για κατηγορίες αρχιτεκτονικών τμημάτων
  - Συχνά πολλά στιγμιότυπα μίας micro-architecture εμφανίζονται και υλοποιούνται σε ένα σύστημα
  - Κάθε αρχιτέκτονας και σχεδιαστής πρέπει να γνωρίζει όλες τις σχετικές micro-architectures



24

CSD

## Επίπεδα αρχιτεκτονικής (3/3)

- Παραδείγματα όπου η macro-architecture εσωτερικά εμπλέκει πολλές micro-architectures – *για κάθε ένα από τα τμήματα στην δεξιά στήλη υπάρχει εξειδικευμένη αρχιτεκτονική.*

| Σύστημα με γνωστή μακρό-αρχιτεκτονική | Τμήματα με τις δικές τους ειδικές μικρό-αρχιτεκτονικές |
|---|---|
| ***Operating system*** | ▪Process management<br>▪Memory management<br>▪Device driver management |
| ***Compiler*** | ▪Lexical analyzer<br>▪Intermediate code generator<br>▪Symbol table |
| ***Windowing system*** | ▪Input control<br>▪Display composition<br>▪Event dispatcher |



25

CSD

## Περιεχόμενα

- Ορισμός
- Ρόλος στην σχεδίαση
- Γρήγορος προσδιορισμός
- Επίπεδα αρχιτεκτονικής
- ***Βασικά αρχιτεκτονικά μοντέλα***
- Στοιχεία αρχιτεκτονικής σχεδίασης



26

CSD

## Βασικά αρχιτεκτονικά μοντέλα (1/2)

- *Layered* architectures
  - Επιπέδων / στρωμάτων
- *Sequential* architectures
  - Ακολουθιακής επεξεργασίας
- *Event-based* architectures
  - Βασισμένων σε γεγονότα
- *Agent-based* architectures
  - Βασισμένων σε λογισμικούς πράκτορες
- *Component-based* architectures
  - Βασισμένων σε ανεξάρτητα λογισμικά τεμάχια
- *Plug-in* architectures
  - Βασισμένων σε δυναμικά τεμάχια



27

CSD

## Βασικά αρχιτεκτονικά μοντέλα (2/2)

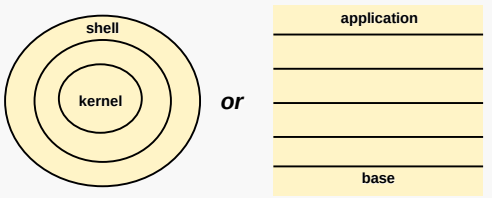
- Δεν περιορίζονται σε συγκεκριμένα λογισμικά συστήματα, αλλά αντικατοπτρίζουν κάποιες θεμελιώδεις δομές οργάνωσης και ροής ελέγχου.
- Ουσιαστικά πρόκειται για μετά-αρχιτεκτονικές, ή αλλιώς οικογένειες αρχιτεκτονικών
  - **meta architectures, architecture families**
- Σήμερα χρησιμοποιείται και ο όρος αρχιτεκτονικά πρότυπα (**architectural patterns**) για αρχιτεκτονικές λύσεις που μπορούν να χρησιμοποιηθούν στην κατασκευή ενός συστήματος



28

CSD

## *Layered* architectures (1/7)






29

CSD

## *Layered* architectures (2/7)

- *Ιδιότητες*
  - Το κάθε επίπεδο χτίζεται πάνω στο προηγούμενο, βάσει συγκεκριμένων application programming interfaces (APIs)
    - Π.χ. σύνολο από functions ή classes που παρέχει το κατώτερο επίπεδο στο ανώτερο
  - Διαδοχική ανεξάρτητη ανάπτυξη από χαμηλότερα επίπεδα προς ανώτερα, ακόμη και παράλληλα εάν τα APIs είναι ήδη παγιωμένα
  - Υποστηρίζει εύκολη επέκταση και τροποποίηση εσωτερικά σε κάθε επίπεδο, χωρίς να επηρεάζονται τα ανώτερα,
    - εφόσον δεν μεταβάλλεται τόσο το API που παρέχεται στο αμέσως επόμενο επίπεδο, όσο και η συμπεριφορά (semantics).
  - Υποστηρίζει επαναχρησιμοποίηση ενός επιπέδου μέσα σε συμβατή αντίστοιχη αρχιτεκτονική (δηλ. τα APIs)
    - η ακόμη και αντικατάσταση

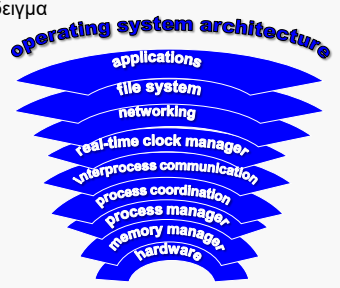


30

CSD

## *Layered* architectures (3/7)

- Παράδειγμα






31

CSD

## *Layered* architectures (4/7)

- *Δομή κώδικα – βλέπουμε μόνο τα exported APIs*

***Layer k, top layer. Πιο πρόσφορο για τροποποιήσεις.***

| | | | | |
|---|---|---|---|---|
| Functions | $F_k1,\ldots,F_kn_k$ | and data types | $T_k1,\ldots,T_km_k$ | K |
| Functions | $F_{k-1}1,\ldots,F_{k-1}n_{k-1}$ | and data types | $T_{k-1}1,\ldots,T_{k-1}m_{k-1}$ | K-1 |
| …. | | | | |
| Functions | $F_21,\ldots,F_2n_2$ | and data types | $T_21,\ldots,T_2m_2$ | 2 |
| Functions | $F_11,\ldots,F_1n_1$ | and data types | $T_11,\ldots,T_1m_1$ | 1 |

***Layer 1, bottom layer. Συνήθως το πιο επισφαλές σε τροποποιήσεις (η αλλιώς «άβατο»)***

***ΚΑΝΟΝΑΣ***. *Στην υλοποίηση κώδικα σε οποιοδήποτε layer j, επιτρέπεται να κληθεί μία συνάρτηση F εάν και μόνο εάν ισχύει:*

$F \in \{ F_{j-1}1,\ldots,F_{j-1}n_{j-1} \} \cup \{ F_j1,\ldots,F_jn_j \} \cup \{$ *inner functions of layer j* $\}$



32

CSD

## *Layered* architectures (5/7)

- *Υλοποίηση (1/3)*

```
// Layer_k.h
class Layer_k_Iface {
    public:
    virtual void f (void) = 0;
    virtual void g (void) = 0;
    virtual void h (void) = 0;
};

class Layer_k {
    public:
    static void             Create (void);
    static void             Destroy (void);
    static Layer_k_Iface*   Get (void);
};
```

Διαχωρίζουμε το API του Layer-k από την κλάση που το υλοποιεί, κάτι που μας δίνει τη δυνατότητα για εναλλακτικές υλοποιήσεις

Αυτό το API βλέπει ο client του Layer-k που είναι πάντοτε το Layer-k+1



CSD

## *Layered* architectures (6/7)

- *Υλοποίηση (2/3)*

```
// Layer_k_Impl.cpp (some implementation of Layer-k)

class Layer_k_impl : public Layer_k_Iface {
    // implement here all exported methods
    public:
    virtual void f (void) { /* code for f */ }
    virtual void g (void) { /* code for g */ }
    virtual void h (void) { /* code for h */ }
};

static Layer_k_Iface* layer_k = nullptr;

void Layer_k::Create (void)
    { assert(!layer_k); layer_k = new Layer_k_impl; }

void Layer_k::Destroy (void)
    { assert(layer_k); delete layer_k; layer_k = nullptr; }

Layer_k_Iface*  Get (void)
    { assert(layer_k); return layer_k; }
```

Εδώ θεωρούμε ότι έχουμε μόνο ένα instance του Layer-k αλλά η αλλαγή για πολλαπλά instances είναι ιδιαίτερα απλή

Τυπική υλοποίηση του Layer-k με το δεύτερο τμήμα να είναι παρόμοιο σε κάθε υλοποίηση



CSD

## *Layered* architectures (7/7)

- *Υλοποίηση (3/3)*

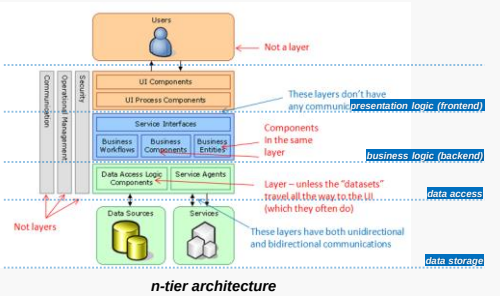
```
class Layer_k_alt { // no more subclass of Layer_k_Ifaces!
    public:
    struct Ctor {   // interface for a constructor of Layer-k interface
        virtual  Layer_k_Iface* New (void) const = 0;
        virtual ~Ctor(){}
    };

    protected:
    static Ctor*            ctor;
    static Layer_k_Iface*   layer_k;

    public:
                    // allows set a Layer-k creator during execution
    static void     Set (Ctor* _ctor)
                        { assert(!ctor); ctor = _ctor; }
    static void     Create (void)
                        { assert(ctor && !layer_k); layer_k = ctor->New(); }
    static void     Destroy (void) {
                        assert(ctor && layer_k);
                        delete  layer_k;
                        delete  ctor;
                        layer_k = nullptr;
                        ctor    = nullptr;
                    }
};
```

The Set() is called in the application main() where we decide which layer k implementation to use. We may even have it configurable and fully dynamic by loading it from a dll file.



CSD

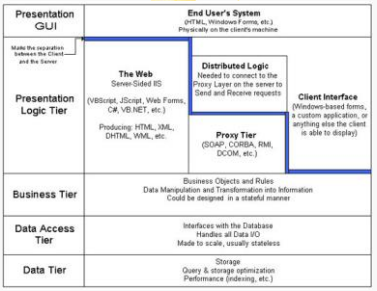
## Examples (1/2)




*n-tier architecture*

CSD

## Examples (2/2)



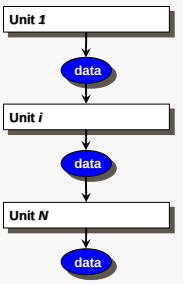

*n-tier architecture*



37

CSD

## *Sequential* architectures (1/8)






38

CSD

## *Sequential* architectures (2/8)

- *Ιδιότητες (1/2)*
  - Αυστηρά ορισμένη αλληλουχία φάσεων επεξεργασίας
  - Κάθε στάδιο επεξεργασίας γίνεται ανεξάρτητο από τα προηγούμενα
  - Συνήθως δεν υπάρχει επικοινωνία, παρά μόνο διάθεση δεδομένων (αποτελεσμάτων επεξεργασίας) από μία φάση προς την αμέσως επόμενη
  - Δυνατότητα επαναχρησιμοποίησης κώδικα και εφαρμογής τροποποιήσεων σε κάθε ξεχωριστή φάση,
    - εφόσον δεν επηρεάζεται η τυπολογία και σημασιολογία των δεδομένων εισόδου και εξόδου



39

CSD

## *Sequential* architectures (3/8)

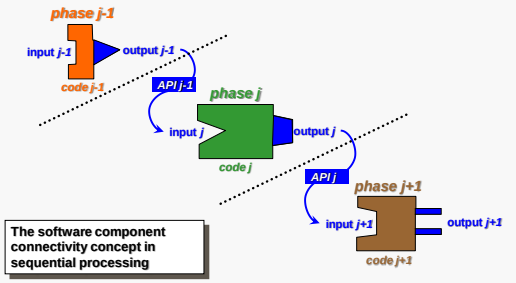
- *Ιδιότητες (2/2)*
  - Η εξάρτηση δύο συνεχόμενων φάσεων επεξεργασίας **A** (*προηγείται*), και **B** (*έπεται*) έγκειται :
    - στην παραγωγή δεδομένων από την A (*producer*) και στην χρησιμοποίηση αυτών από την B (*consumer*)
    - συνήθως στην παροχή ενός API από την A για την πρόσβαση στα δεδομένα αυτά – ***accessor API***, το οποίο και χρησιμοποιείται για την υλοποίηση της φάσης B
  - Με καλή τυποποίηση των δεδομένων και των ***accessor APIs*** (όταν παρέχονται), οι δυνατότητες συνδεσμολογίας προσομοιάζουν αυτές των hardware components.
  - Παραλλαγές στην συνδεσμολογία είναι εφικτές και πολύ συνήθεις – πέρα από την γραμμική σειριακή επεξεργασία



40

CSD

## *Sequential* architectures (4/8)



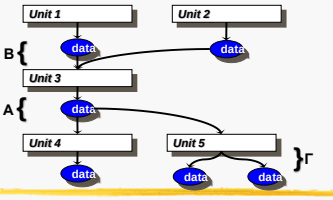



41

CSD

## *Sequential* architectures (5/8)

- ***Συνήθεις οι παραλλαγές ανάλογα με τις ανάγκες επεξεργασίας***
  - Τα δεδομένα μίας φάσης τροφοδοτούνται σε πολλές φάσεις (για άλλου είδους επεξεργασία) - ***A***
  - Μία φάση δέχεται ως είσοδο δεδομένα από πολλές φάσεις- ***B***
  - Μία φάση παράγει πολλές κατηγορίες δεδομένων (λόγω πολλαπλών επεξεργασιών ή απλώς διαφορετικών προοπτικών των δεδομένων) - ***Γ***






42

CSD

## *Sequential* architectures (6/8)

- *Δομή κώδικα (1/2)*

***∀ φάση επεξεργασίας j***

| | | |
|---|---|---|
| ***Εξωτερικές εξαρτήσεις*** | Τύποι δεδομένων εισόδου | ***∀ είσοδο j:*** $T_j1,\ldots, T_jm_j$ |
| | Input Accessor API | ***∀ είσοδο j:*** $A_j1,\ldots,A_jn_j$ |
| ***Εσωτερική υλοποίηση*** | Εσωτερική υλοποίηση | $F_j1,\ldots,Fk_j$ |
| | Τύποι δεδομένων εξόδου | ***∀ έξοδο j:*** $T^*_j1,\ldots,T^*_jm_j$ |
| | Output Accessor API | ***∀ έξοδο j:*** $A^*_j1,\ldots,A^*_jn_j$ |

- ***ΚΑΝΟΝΑΣ***. *Στην υλοποίηση κώδικα επιτρέπεται να κληθεί μία συνάρτηση F εάν και μόνο εάν ισχύει:* $F \in \{ A_j1,\ldots,A_jn_j \} \cup \{ F_j1,\ldots,Fk_j \} \cup \{ A^*_j1,\ldots,A^*_jn_j \}$
  ***Δεν επιτρέπεται να υπάρχουν κλήσεις συναρτήσεων που ανήκουν σε άλλες φάσεις***



43

CSD

## *Sequential* architectures (7/8)

- *Δομή κώδικα (2/2)*
  - ***ΚΑΝΟΝΑΣ***. *∀ F, αν F έχει πρόσβαση στην είσοδο / έξοδο,* $\{ T_j1,\ldots,T_jm_j \}$ / $\{ T^*_j1,\ldots,T^*_jm_j \} \Rightarrow F \in \{ A_j1,\ldots,A_jn_j \}$ / $\{ A^*_j1,\ldots,A^*_jn_j \}$.
    - ***Κάθε συνάρτηση που έχει πρόσβαση στα δεδομένα πρέπει να είναι ορισμένη από πριν σαν accessor function***
  - ***ΤΑΚΤΙΚΗ***. *∀* $F \in \{ A_j1,\ldots,A_jn_j \}$ / $\{ A^*_j1,\ldots,A^*_jn_j \}$, *δεν πρέπει η F να μεταβάλλει τις τιμές των δεδομένων εισόδου ή εξόδου*
    - ***Οι accessor functions καλό είναι να έχουν read only πρόσβαση στα δεδομένα (δηλ. να μην είναι modifiers)***

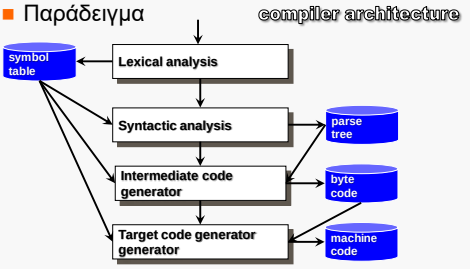


44

CSD

## *Sequential* architectures (8/8)

- Παράδειγμα

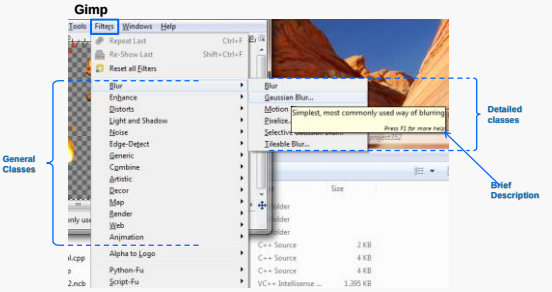





45

CSD

## Example (1/7)

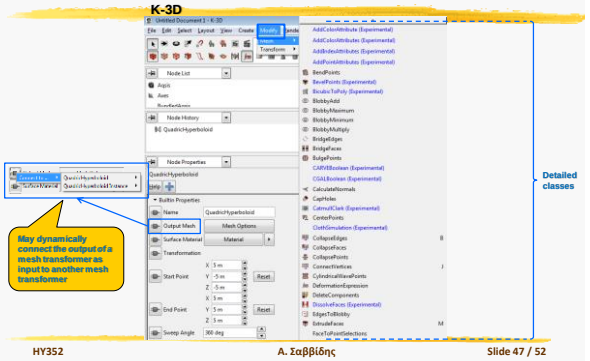





46

CSD

## Example (2/7)

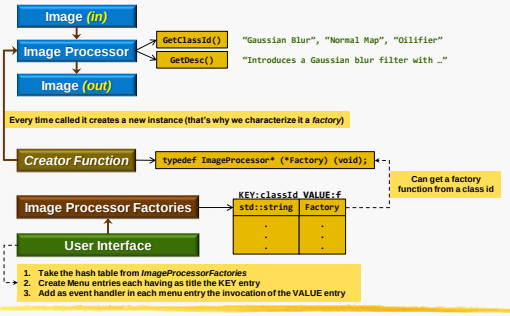





47

CSD

## Example (3/7)






48

CSD

## Example (4/7)

```
struct Image {};

class ImageProcessor {
protected:
    const Image*    in;
    mutable Image*  out;
public:
    typedef ImageProcessor* (*Factory)(void);

    virtual const string    GetClassId (void)    const = 0;
    virtual const string    GetDesc (void)       const = 0;
    virtual Image*          Process (void)       const = 0;
    virtual ImageProcessor* Clone (void)         const = 0;
    void                    Set (Image* _in) {
                                if (in != _in) {
                                    in = _in;
                                    out = (Image*) 0;
                                }
                            }
    Image*                  GetResult (void) const {
                                assert(in);
                                if (!out)
                                    out = Process();
                                return out;
                            }
    ImageProcessor (void) : in((Image*) 0), out ((Image*) 0) {}
    virtual ~ImageProcessor(){}
};
```

Constructor function type

Primary processing method

Processing result is cached



49

CSD

## Example (5/7)

```
class ImageProcessorFactories {
protected:
    typedef ImageProcessor::Factory Factory;
    typedef map<string, Factory>    Factories;
    static Factories                factories;
public:
    static void                 Install (
                                    const string&           classId,
                                    ImageProcessor::Factory f
                                )
                                { factories[classId] = f; }
    static ImageProcessor*      New (const string& classId)
                                    Factories::iterator i = factories.find(classId);
                                    return i != factories.end() ? (*i->second)() : (ImageProcessor*) 0;
                                }
    static const Factories& Get (void)
                                { return factories; }
};
```

Hash table of constructor functions indexed by class id

Dynamic (runtime) installation of an image processor factory

Dynamic (runtime) processor instantiation using its class id

Can get the hash table of processors to create a menu with respective class ids



50

CSD

## Example (6/7)

```
class ImageProcessingSequence {
private:
    Image*                  in;
    list<ImageProcessor*>   processors;
public:
    Image*      Get (void) const {
                    if (processors.empty())
                        return in;
                    else
                        return processors.back()->GetResult();
                }
    void        Push (const std::string& classId) {
                    processors.push_back(
                        ImageProcessorFactories::New(classId)
                    );
                }
    void        Pop (void) {
                    assert(!processors.empty());
                    delete processors.back();
                    processors.pop_back();
                }
    void        Clear (void) {
                    while (!processors.empty())
                        Pop();
                }
};
```

An image processing sequence is an edited image on which stacked transformations are applied

Dynamic insertion (push) and removal (pop) is allowed during editing

More sophisticated editing is easily possible by extending the class: repeating, inserting in-between, or removing all of a class, etc.



51

CSD

## Example (7/7)

```
// sample concrete image processor (subclass); all follow
// a similar pattern, while they are independent to each other

class ImageBlur : public ImageProcessor {
    public:
    static ImageProcessor*  New (void)
                                { return new ImageBlur; }
    virtual const string    GetClassId (void)
                                const { return "blur"; }
    virtual const string    GetDesc (void)  const
                                { return "Simplest, most commonly used way of blurring"; }
    virtual Image*          Process (void) const
                                { return (Image*) 0; }
    virtual ImageProcessor* Clone (void) const
                                { return new ImageBlur(*this); }
    static void             Install (void)
                                { ImageProcessorFactories::Install("blur", &New); }
    ImageBlur (void){}
    ImageBlur (const ImageBlur& p){ /* copy ctor logic */}
    virtual ~ImageBlur(){}
};
```



52