HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

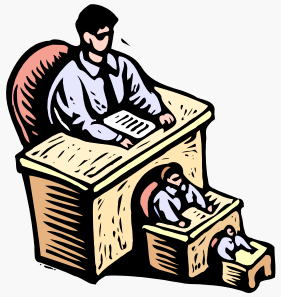
ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

CSD

ΕΝΟΤΗΤΑ 3

ΣΧΕΔΙΑΣΤΙΚΕΣ ΠΡΟΟΠΤΙΚΕΣ ΚΑΙ ΔΟΜΗΜΕΝΟΣ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ

Αριθμός διαλέξεων 2 – Διάλεξη 1η



CSD

Περιεχόμενα

- Σχεδίαση λογισμικού: αρχές και μέθοδοι
  - Αρχές σχεδίασης λογισμικού (σύντομη εισαγωγή)
  - Σχεδιαστικές προοπτικές
    - Data modeling – *μοντελοποίηση δεδομένων*
    - Structural design – *δομική σχεδίαση*
    - Functional design – *λειτουργική σχεδίαση*
    - Behavioral analysis – *συμπεριφεριολογική ανάλυση*

CSD

Αρχές σχεδίασης λογισμικού

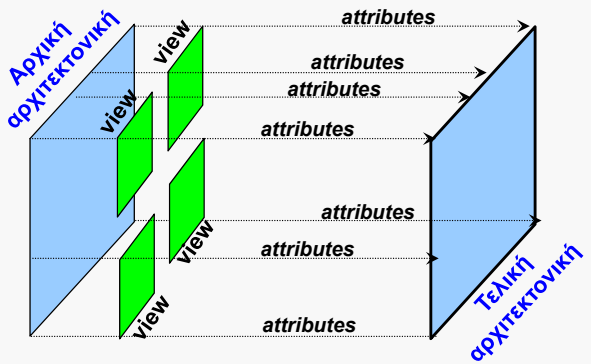
- Δεν υπάρχει συνταγολόγιο επιτυχίας
- Δεν υπάρχει κεντρικά συγκεντρωμένη σοφία
- Δεν υπάρχει ένας μόνο κώδικας καλής πρακτικής
- Δεν υπάρχουν καθολικά βέλτιστες λύσεις

- Do the right thing
  - δηλ. ακολουθείτε όλους τους κανόνες που μαθαίνετε
- Provide primitives, not solutions
  - δηλ. έμφαση στο «ψάρεμα» και όχι στο «ψάρι»
- KISS (keep it simple stupid)
  - ο ανθρώπινος εγκέφαλος δεν μπορεί να χειριστεί τίποτε μετά από κάποια πολυπλοκότητα, αλλά πολλά με περιορισμένη πολυπλοκότητα
- ***Η τελειοποίηση της σχεδίασης δεν έρχεται όταν δεν έχεις κάτι άλλο να προσθέσεις, αλλά κάτι άλλο να αφαιρέσεις***

Σχεδιαστικές προοπτικές (1/4)

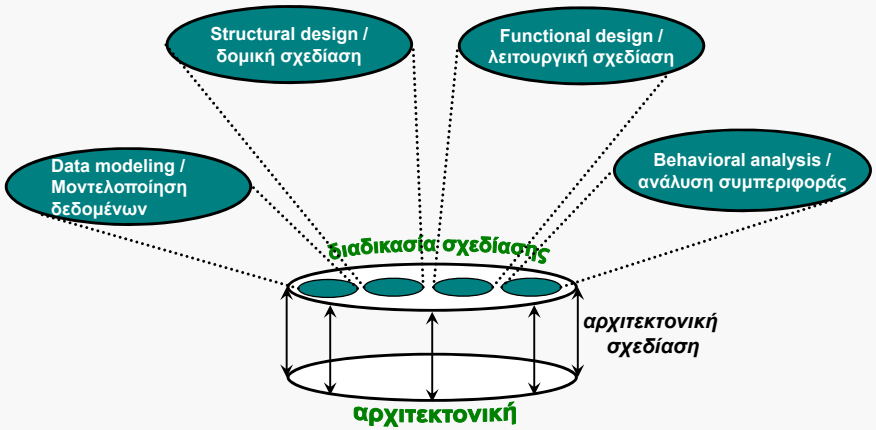
- Αμέσως μετά την σκιαγράφηση της αρχιτεκτονικής ακολουθεί η λεπτομερής σχεδίαση λογισμικού, η οποία παρουσιάζει εναλλακτικές συμπληρωματικές προοπτικές του συστήματος, ενώ συνεχώς παρέχει χρήσιμη πληροφορία στην διαδικασία αρχιτεκτονικής σχεδίασης
- Η διαδικασία αυτή χαρακτηρίζεται από τρεις βασικές ιδιότητες:
  - *use case driven – βασισμένη σε σενάρια χρήσης*
  - *architecture centric – έχει ως κέντρο την αρχιτεκτονική*
  - *iterative and incremental – είναι επαναληπτική και αυξητική*

Σχεδιαστικές προοπτικές (2/4)

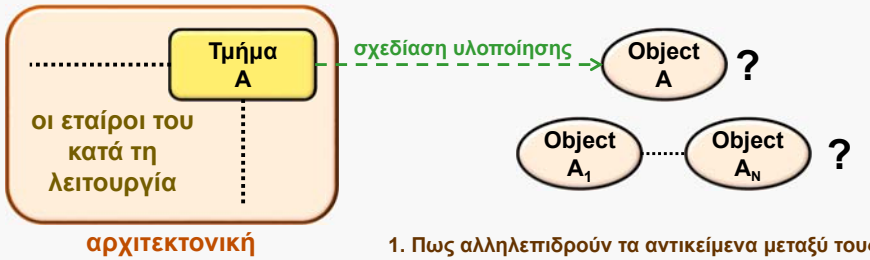


Οι σχεδιαστικές προοπτικές, οι οποίες είναι εξειδικευμένες απόψεις της σχεδίασης, αποκαλύπτουν αρχιτεκτονικές ιδιότητες (attributes) βάσει των οποίων το αρχικό αρχιτεκτονικό σχέδιο: αναθεωρείται, επαληθεύεται, συγκεκριμενοποιείται, και τελειοποιείται.

Σχεδιαστικές προοπτικές (3/4)



Σχεδιαστικές προοπτικές (4/4)



1. Πως αλληλεπιδρούν τα αντικείμενα μεταξύ τους;
2. Πώς ενορχηστρώνεται η συνέργειά τους (control flow);
3. Πως γίνεται η ανάλυση των λειτουργιών τους;
4. Τι δεδομένα χειρίζονται;
5. Ποιες οι βασικές τους απαιτήσεις επεξεργασίας;

Μοντελοποίηση δεδομένων (1/2)

- Τέτοιου είδους μοντέλα *ανακαλύπτουν και αναπαριστούν* τα διαφορετικά είδη δεδομένων τα οποία διαχειρίζεται το λογισμικό, καθώς και τις μεταξύ τους σχέσεις. Τα δεδομένα μπορεί να εμπλέκονται σε τρία επίπεδα:
  - ***Εξωτερικό***: αποτελούν αντικείμενο συναλλαγής κυρίως με το εξωτερικό περιβάλλον – ***δεν αλλάζει συχνά***
    - π.χ. με τον τελικό χρήστη, ή άλλα ανεξάρτητα συστήματα
  - ***Διατμηματικό***: αφορούν την συναλλαγή μεταξύ βασικών αρχιτεκτονικών τμημάτων – ***δεν πρέπει να αλλάζει συχνά***
    - π.χ. μεταξύ φάσεων επεξεργασίας
  - ***Ενδοτμηματικό***: εμπλέκονται αποκλειστικά και μόνο στο εσωτερικό ενός εκάστοτε τμήματος – ***εύκολο να αλλάζει***
    - π.χ. μεταξύ συναρτήσεων ή μικροτμημάτων, ενδοεπικοινωνία

Μοντελοποίηση δεδομένων (2/2)

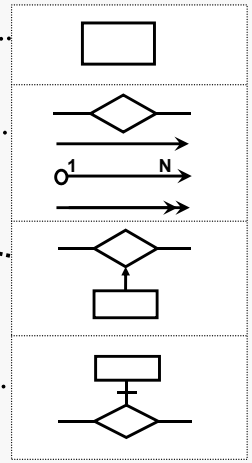
- ***Entity / Relationship diagrams*** (E/R diagrams) – διαγράμματα οντοτήτων / σχέσεων
  - Ένα σχεσιακό / relational μοντέλο, που συνήθως χρησιμοποιείται στην σχεδίαση σχεσιακών βάσεων δεδομένων.
- ***Object hierarchies*** (*isa* / *partof* hierarchies) – ιεραρχίες αντικειμένων
  - Μία απλοποιημένη μορφή οντοκεντρικής σχεδίασης η οποία επικεντρώνεται στη γρήγορη εξαγωγή δομικών και οργανωτικών χαρακτηριστικών των διαφόρων λογισμικών τμημάτων.
    - Αντιμετωπίζοντας όλες τις προγραμματιστικές δομές (δεδομένα και λειτουργίες) ως δομικούς τύπους, εφαρμόζει μοντελοποίηση, που φαινομενικά ταιριάζει σε δεδομένα, πάνω σε λειτουργικές δομές

E/R diagrams (1/5)

- Μία από τις παλαιότερες, αλλά επικρατέστερες μεθόδους μοντελοποίησης δεδομένων που βασίζεται στον καθορισμό των οντοτήτων καθώς και των μεταξύ τους σχέσεων.
  - Συνήθως αναφερόμαστε σε σχέσεις τις οποίες το λογισμικό ενδιαφέρεται να τις καταγράψει και να τις αποθηκεύσει με κάποιο τρόπο
    - και όχι σε σχέσεις οι οποίες έχουν κάποια μικρή χρονική διάρκεια και κυρίως χαρακτηρίζονται ως γεγονότα (events)

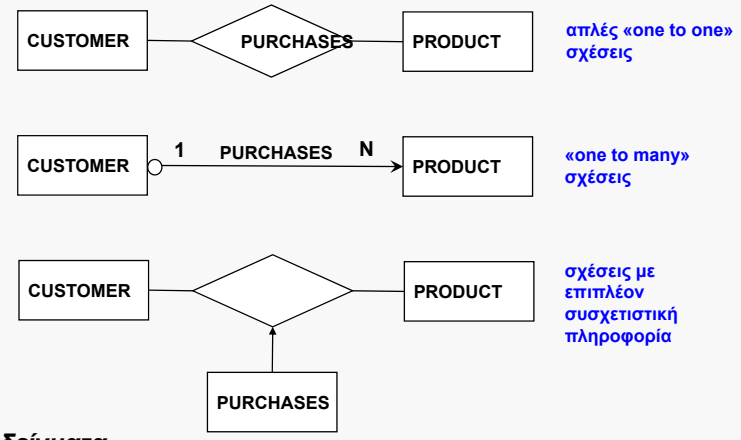
E/R diagrams (2/5)

- *Entity types*
  - τύποι οντοτήτων
- *Relationships*
  - Σχέσεις
- *Associative entity type*
  - Συσχετιστική οντότητα
- *Supertype / subtype*



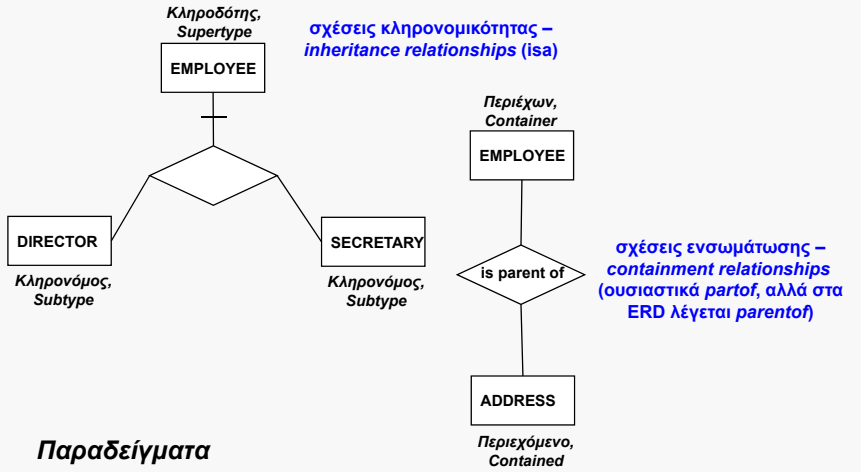
Σύμβολα

E/R diagrams (3/5)



Παραδείγματα

E/R diagrams (4/5)



Παραδείγματα

E/R diagrams (5/5)

Επιρροή στον κώδικα

```
class Product_E {...};      ← Entity
class Customer_E {...};     ← Entity
class Purchases_R {         ← Associative entity
    private:
    Customer_E*   customer;
    Product_E*    products;
    Date_E*       date;
    unsigned      totalProducts;
    ...
};
class Address_E {...};      ← Entity
class Employee_E {          ← Supertype and Container
    private:
    Address_E address;      ← Contained
    ...
};
class Director_E : public Employee_E {...};
class Secretary_E: public Employee_E {...};
```

Βοηθά στην οπτική απεικόνιση των τύπων δεδομένων οι οποίοι έπειτα μετατρέπονται σε δομές δεδομένων στη γλώσσα C++

Κυρίως για βάσεις δεδομένων

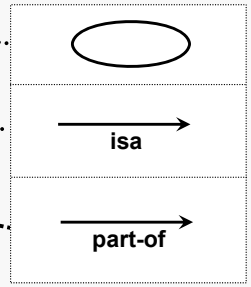
Object diagrams (1/10)

- Αναπαριστούν την ιεραρχία κληρονομικότητας και την εσωτερική διάρθρωση δομών δεδομένων
- Μπορεί να χρησιμοποιηθεί και για δομική σχεδίαση όταν δεν επικεντρώνεται σε δομές δεδομένων αλλά σε λειτουργικές οντότητες
  - ***Μπορείτε να θεωρείτε ως τέτοια οντότητα ένα object ή component***
- Ορίζονται δύο σχέσεις παρόμοιες με τα διαγράμματα οντοτήτων σχέσεων:
  - ***isa*** σχέσεις δηλαδή “is a” (κληρονομικότητα)
  - ***part-of*** σχέσεις (σύνθετα χαρακτηριστικά ή λειτουργικός τεμαχισμός).

Χρησιμοποιείται στην πράξη

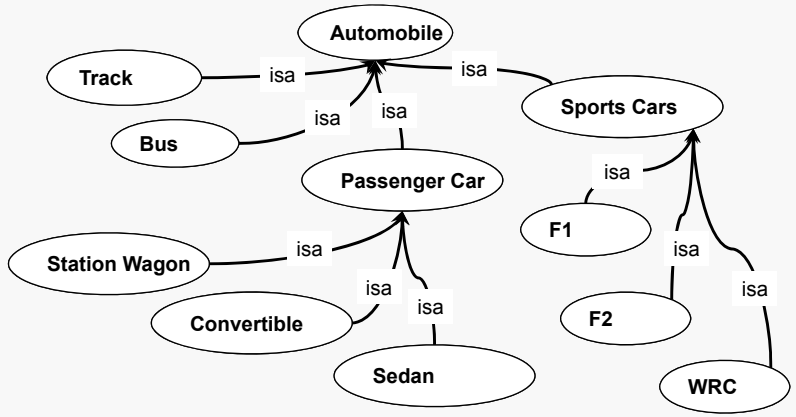
Object diagrams (2/10)

- *Object types*
  - τύποι αντικειμένου
- *Inheritance - isa*
  - Κληρονομικότητα
- *Containment – partof*
  - Ενσωμάτωση



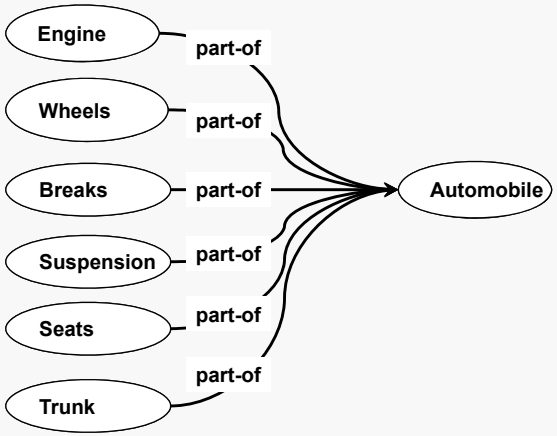
Σύμβολα

Object diagrams (3/10)



Παραδείγματα

Object diagrams (4/10)



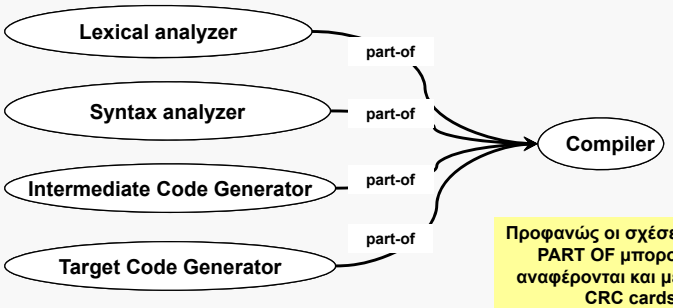
Παραδείγματα

Object diagrams (5/10)

- Απλή τεχνική αντικειμενοστραφούς ανάλυσης χωρίς ιδιαίτερη έμφαση στην εξαντλητική ανάλυση
- Επιτρέπει την *γρήγορη εξαγωγή των δομικών χαρακτηριστικών*, τα οποία μπορούν έπειτα αμέσως να μεταφραστούν σε ένα σχήμα ιεραρχίας η οργάνωσης κλάσεων στον κώδικα
- Σε πολλές περιπτώσεις η ίδια τεχνική εφαρμόζεται για την ιεραρχική ανάλυση λειτουργικών τμημάτων – *structural decomposition*, ή για την σχεδίασης ιεραρχίας κληρονομικότητας λειτουργικών τμημάτων – *inheritance*
  - Εφόσον υπάρχουν πιο ώριμες σχεδιαστικές μέθοδοι ειδικά για την λειτουργική σχεδίαση – *functional design*, προτείνεται να εφαρμόζονται αυτές στην πράξη για σχετικά πολύπλοκες αλγοριθμικά λειτουργίες, παρά τα διαγράμματα αντικειμένων
  - ***Για λεπτομερή σχεδίαση κλάσεων συνιστάται να τη χρησιμοποιείτε σε συνδυασμό με τη μέθοδο γρήγορης σχεδίασης των CRC cards***

Object diagrams (6/10)

- Άλλο παράδειγμα για ιεραρχική ανάλυση κώδικα με επαγωγή λειτουργικών τμημάτων και μεταξύ των σχέσεων



Προφανώς οι σχέσεις ISA και PART OF μπορούν να αναφέρονται και μέσω των CRC cards

Object diagrams (7/10)

```
class TokenList {...};
class LexicalAnalyser {
    public:  TokenList*   Analyse (const char* file);
};
class ParseTree {...};
class SymbolTable {...};
class SyntaxAnalyser {
    public:        ParseTree*   Parse (TokenList*);
                   SymbolTable* Symbols (void);

};
class ByteCode {...};
class IntermediateCodeGenerator {
    public: ByteCode* Generate(ParseTree*, SymbolTable*);
};
class TargetCode {...};
class TargetCodeGenerator {
    public: TargetCode* Generate(ByteCode*, SymbolTable*);
}
```

Ανάλογα με τη δυνατότητα να εφαρμόσουμε λειτουργική ανάλυση της αρχιτεκτονικής με αυτή τη μέθοδο, καλός τεμαχισμός σε τμήματα μπορεί επιτευχθεί

Object diagrams (8/10)

```
class Compiler {
    private:
    LexicalAnalyser              lexAnalyser;
    SyntaxAnalyser               syntaxAnalyser;
    IntermediateCodeGenerator    icodeGenerator;
    TargetCodeGenerator          tcodeGenerator;
    public:
    TargetCode* Compile (const char* file) {
    return
        tcodeGenerator.Generate(
           icodeGenerator.Generate(
              syntaxAnalyser.Parse(lexAnalyser.Analyse(file)),
              syntaxAnalyser.Symbols()
           ),
           syntaxAnalyser.Symbols()
        );
    }
};
```

Ο κώδικας έχει απλοποιηθεί με το να μη συμπεριλαμβάνει απελευθέρωση της μνήμης μετά το compilation

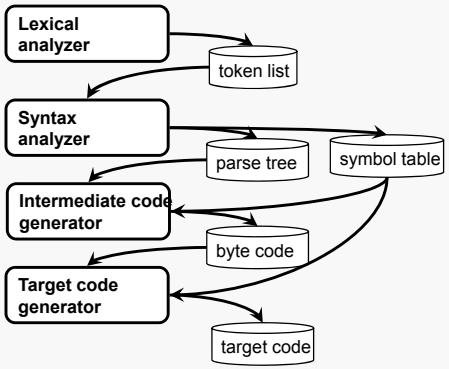
Αυτό που γίνεται προφανές είναι ότι η ροή ελέγχου της ακολουθιακής αρχιτεκτονικής όταν υλοποιείται σε ένα πρόγραμμα μπορεί να σημαίνει και αντίστροφη φορά κλήσης.

Object diagrams (9/10)

```
class Compiler {
    private:
    LexicalAnalyser              lexAnalyser;
    SyntaxAnalyser               syntaxAnalyser;
    IntermediateCodeGenerator    icodeGenerator;
    TargetCodeGenerator          tcodeGenerator;
    SymbolTable                  symbols;
    public:
    TargetCode* Compile (const char* file) {
    return
        TokenList* t = lexAnalyser.Analyse(file);
        ParseTree* p = syntaxAnalyser.Parse(t);
        ByteCode*  b = icodeGenerator.Generate(p, &symbols);
        TargetCode*  c = tcodeGenerator.Generate(b, &symbols);
        delete t; delete p; delete b; return c;
    }
};
```

Αλλά μπορούμε να υλοποιήσουμε και τη ροή ελέγχου ούτως ώστε να είναι «σύννομη» αυτής που αποτυπώνει η αρχιτεκτονική μας.

Object diagrams (10/10)



…και εξειδικεύεται η sequential architecture του compiler

Δομική σχεδίαση (1/2)

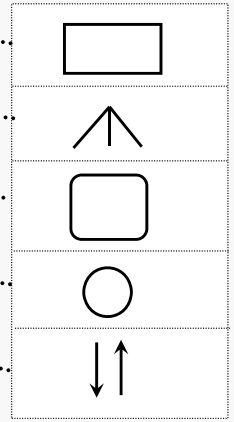
- Διαπραγματεύεται μοντέλα για την ***ανάλυση των λειτουργιών*** (κυρίως των συναρτήσεων) ως προς την διάρθρωση τους σε επιμέρους λειτουργίες – *functional decomposition*.
- Ως αποτέλεσμα, αυτά τα μοντέλα πετυχαίνουν την ***αναπαράσταση κυρίως των στατικών χαρακτηριστικών*** του συστήματος, που έχουν σχέση με την δομή και τις εξαρτήσεις του κώδικα, παρά με τη δυναμική συμπεριφορά του συστήματος και τη ροή ελέγχου.

Δομική σχεδίαση (2/2)

- ***Structured Charts***
  - Γραφική αναπαράσταση της ***ιεραρχικής κατάτμησης*** (hierarchical decomposition) των λειτουργιών (…τμημάτων;) του προγράμματος.
  - Η ***ροή δεδομένων*** (data flow) δεν αναλύεται ιδιαίτερα, ενώ η ***ροή ελέγχου*** (control flow) μερικώς αναπαρίσταται με έμφαση στην αλληλουχία βημάτων εκτέλεσης (execution sequencing).
- ***Dependency Graphs***
  - Παρουσιάζουν τις ***σχέσεις ελέγχου*** (control links) μεταξύ των λειτουργικών τμημάτων, ενσωματώνοντας επίσης πληροφορία για τα χρησιμοποιούμενα δεδομένα.
    - Οι σχέσεις είναι τύπου: (α) call για τις συναρτήσεις,(β) και read / write για τα δεδομένα.
- ***CRC Cards***
  - Αποτυπώνουν γρήγορα και ευέλικτα αρχιτεκτονικό τεμαχισμό σε οποιοδήποτε επίπεδο (macro / micro / … / code). Στέκονται ένα επίπεδο υψηλότερα από τα δύο παραπάνω. Εφαρμόζονται και σε συνδυασμό με object diagrams

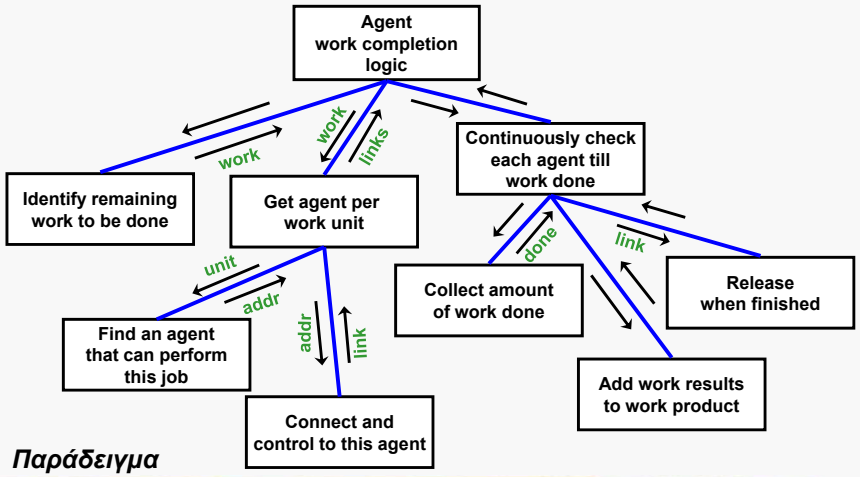
Structured Charts (1/6)

- *Functional unit*
  - Λειτουργικό τμήμα
- Decomposition
  - Κατάτμηση
- *Data item*
  - Δεδομένα
- *Input / Output*
  - Είσοδος / έξοδος
- *Control / data flow*
  - Ροή ελέγχου / δεδομένων



Σύμβολα

Structured Charts (2/6)



Παράδειγμα

Structured Charts (3/6)

- *Πότε εφαρμόζονται*
  - Η ροή ελέγχου / δεδομένων εξάγεται διαβάζοντας το διάγραμμα, το οποίο είναι δέντρο, με depth-first σάρωση
  - Εκτός της λειτουργικής κατάτμησης, *λογική ελέγχου* δεν αναπαρίσταται (π.χ. loops, branches)
  - Το βάρος δίνεται στις *λειτουργίες (how)*, παρά στα αντικείμενα (*what*). Για το λόγο αυτό η τεχνική χαρακτηρίζεται ως procedure-oriented
  - Είναι μία από τις κλασικές τεχνικές δομημένης σχεδίασης (*structured design)*, η οποία εφαρμόζει σταδιακή εξειδίκευση (*step-wise refinement)*.
    - Είναι κατάλληλη για τον κατακερματισμό των διαφόρων τμημάτων σε ανεξάρτητα τεμάχια (black boxes), τα οποία και δίνονται σε διαφορετικούς προγραμματιστές για υλοποίηση

Χρησιμοποιείται στην πράξη, αλλά πιο «χαλαρά»

Structured Charts (4/6)

```
void agent_work_completion_logic (Agent* agent) {
   Work*  work  = identify_remaining_work_to_be_done(agent);
   Links* links = grant_agent_per_work_unit(agent, work);
   continuously_check_each_agent_till_work_done(links);
}

Links* get_agent_per_work_unit (Work* work) {
  Links* links = create_agent_connection_links();
  Unit*  unit  = get_first_work_unit(work);
  while (unit) {
     Agent* agent = find_an_agent_that_can_perform_this_job(unit);
     Link*  link  = connect_and_control_this_agent(agent);
     add_to_agent_links(links, link);
     unit = get_next_work_unit(work, unit);
  }
  return links;
}
```

- Μόνο συναρτήσεις εξάγονται, ενώ λεπτομερής λογική
- ελέγχου δεν αναπαρίσταται
- Καλή κατάτμηση σε συναρτήσεις (procedural analysis) παράγεται
- ***Αυτή η τακτική είναι επί της ουσίας δομημένος προγραμματισμός***

Επιρροή στον κώδικα

Structured Charts (5/6)

```
και_κάποια_έξοδο ← μία_λειτουργία_από_την_ανάλυση_απαιτήσεων (με_κάποια_είσοδο) {
    κάνε_πρώτα_αυτό
    if κάποια_συνθήκη_ισχύει then
        προχώρα_σε_αυτά_τα_βήματα
    else
        προχώρα_σε_διαφορετικά_τα_βήματα
    συνέχισε_με_αυτό
    while κάποια_άλλη_συνθήκη_ισχύει do
        εκτέλεσε_αυτά_τα_βήματα
    return υπολόγισε_το_τελικο_αποτέλεσμα
}
```

Εξειδικεύουμε σταδιακά τη λειτουργία σε κάποια επιμέρους βήματα προσέχοντας να επιλέγουμε λογικές ενότητες ή υπολειτουργίες και προσδιορίζοντας μία αλληλουχία που αποτυπώνει έναν αρχικό αλγόριθμο

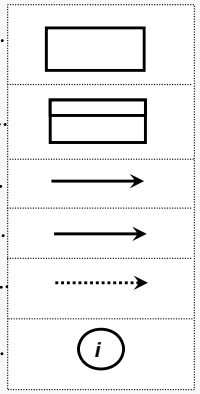
Προκύπτουν νέες λειτουργίες ή γίνεται αναφορά σε υπάρχουσες λειτουργίες (είτε αρχικές ή αποτέλεσμα της δομημένης ανάλυσης άλλων λειτουργιών)

Structured Charts (6/6)

- Προσθέτουμε τις όποιες νέες λειτουργίες προκύπτουν στον κατάλογο λειτουργιών
- Συνεχίζουμε την αρχιτεκτονική ανάλυση έως ότου τις εκχωρήσουμε και αυτές σε κάποιο τμήμα
- Εισαγάγουμε στην αρχιτεκτονική εξαρτήσεις κλήσεων από το τμήμα που περιέχει τη λειτουργία που αναλύσαμε προς τα τμήματα που περιέχουν τις νέες λειτουργίες

Dependency / Call graphs (1/9)

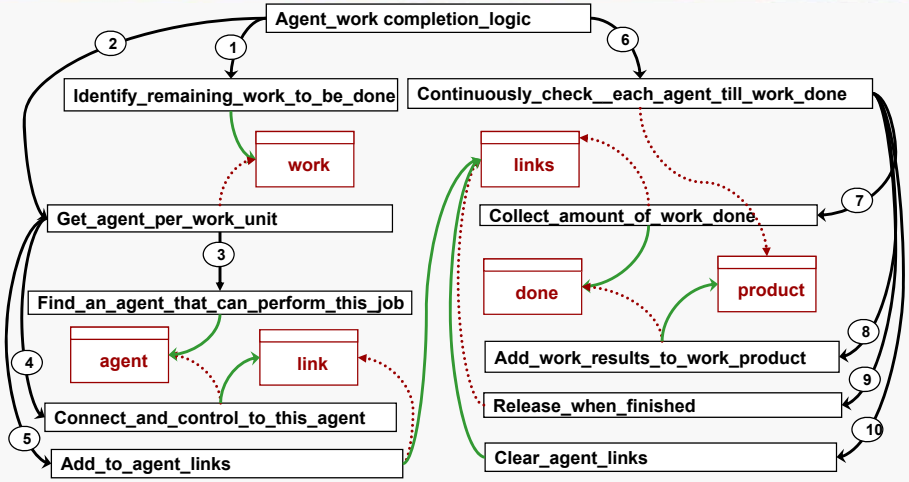
- *Functional unit* }
  - Λειτουργικό τμήμα
- *Data* }
  - Δεδομένα
- *Call* }
  - Κλήση
- *Read / write* }
- *Read* }
- *Order* }
  - Σειρά εκτέλεσης



Dependency / Call graphs (2/9)

- Ο γράφος αυτός συνιστά πολύτιμη σχεδιαστική πληροφορία διότι ***αποκαλύπτει τις εξαρτήσεις υλοποίησης*** των λειτουργικών τμημάτων
- Σε αρχιτεκτονική κλίμακα αποτελεί και μία πρώτη ***ανάλυση των πιθανών σχέσεων και εξαρτήσεων*** της υλοποίησης
- Οι εξαρτήσεις εξειδικεύονται όσο προχωράμε στην δομημένη ανάλυση των λειτουργιών και προκύπτουν υπολειτουργίες
- Το επίπεδο λεπτομέρειας αποφασίζεται από τον σχεδιαστή, ανάλογα με το τι συμπεράσματα θέλουμε να εξάγουμε
- Η τεχνική είναι απολύτως εφαρμόσιμη και σε οντοκεντρική σχεδίαση, διότι οι εξαρτήσεις κλήσης ορίζονται με τον ίδιο τρόπο και για αντικείμενα

Dependency / Call graphs (3/9)

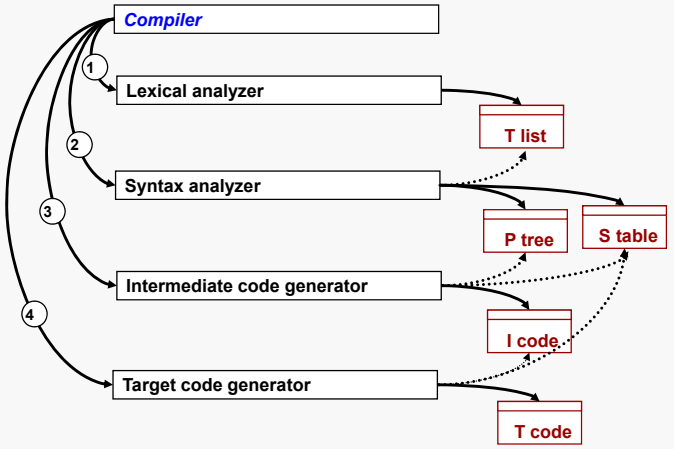


Dependency / Call graphs (4/9)

- *Πότε εφαρμόζονται*
  - Μπορεί να φτάσουν σε μεγάλο βαθμό πολυπλοκότητας και να γίνουν «υπερφορτωμένοι», με εκθετική αύξηση των ακμών, καθώς εμπλέκουμε περισσότερα τμήματα η /και αναλύουμε περισσότερο ως προς το βάθος των κλήσεων
  - Το επίπεδο της ανάλυσης ποικίλει, ανάλογα με το τι θέλουμε να αναλύσουμε:
    - Μεθόδους μίας κλάσης ή συναρτήσεις ενός τμήματος
      - Vertex = program function
      - Edge = function call
    - Εξαρτήσεις μεταξύ κλάσεων η τμημάτων
      - Vertex = class / module / component
      - Edge = class / module / component use.

  ➔ ***Με την εφαρμογή σε ανώτερου επιπέδου τμήματα, αρχιτεκτονικές προοπτικές εξάγονται***

Dependency / Call graphs (5/9)



…εξαγωγή της ακολουθιακής αρχιτεκτονικής από τις εξαρτήσεις του κώδικα

Dependency / Call graphs (6/9)

- *Πιο αναλυτικό παράδειγμα (1/4)*
  - Η ανάλυση του γράφου κλήσεων βοηθά στην εξαγωγή της αρχιτεκτονικής που αποτυπώνει ο κώδικας
  - Τα αρχικά τμήματα του συστήματος προκύπτουν ενώ οι εξαρτήσεις κλήσεων των τμημάτων καταγράφονται
    - Αυτού του είδους οι εξαρτήσεις κλήσεως θεωρούνται ως οι αρχικές αρχιτεκτονικές συνδέσεις

  ➔ Το παράδειγμα αναλύει ***εκ των υστέρων*** τον γράφο κλήσεων που φαίνεται ότι ακολουθεί agent-based αρχιτεκτονική

Dependency / Call graphs (7/9)

- *Πιο αναλυτικό παράδειγμα (2/4)*

```
class Soldier_Agent {  ← Αρχικό class design
    private:
    Weapons             weapons;
    Experience          experience;
    Capabilities        capabilities;
    TroupLeader_Agent* leader;  ← Ανεβαίνουμε ιεραρχικά

    public:
    void Attack(AttackTarget* attackTarget);
    void Retreat (FieldPosition* position);
    void Surrender (TroupLeader_Agent* oppositeLeader);
    void Support (Soldier_Agent* soldier);
    void MoveTo (FieldPosition* position);
    void SetLeader (TroupLeader_Agent* leader);
};
```

Dependency / Call graphs (8/9)

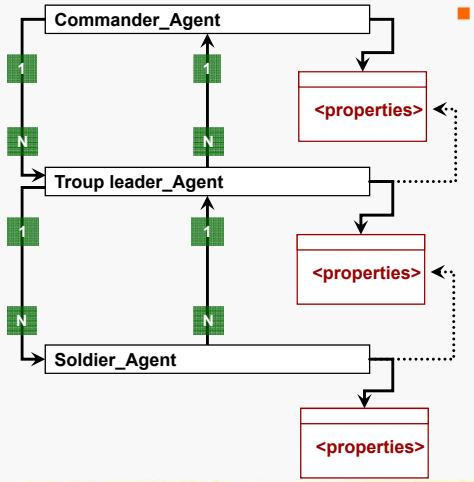
- *Πιο αναλυτικό παράδειγμα (3/4)*

```
class TroupLeader_Agent {   ← Αρχικό class design
    private:
    Strategy*        strategy;
    Mission*         mission;
    Command*         command;
    Commander_Agent* commander;  ← Ανεβαίνουμε ιεραρχικά

    public:
    void SetMission (Mission* mission);
    void SetStrategy (Strategy* strategy);
    void StartMission (void);
    void AbortMission (void);
    void SetCommander (Commander_Agent* commander);
};
```


Dependency / Call graphs (9/9)

- *Πιο αναλυτικό παράδειγμα (4/4)*



Γράφος εξαρτήσεων αντικειμένων, με πληροφορία για τον αριθμό των διαφορετικών αντικειμένων στα οποία μπορεί να γίνει μία κλήση, σε μορφή 1 : N αντιστοιχίας

Επίλογος – δομή και εξαρτήσεις κλήσεων

Tokyo city model

Bottom-level components, heavy dependencies, from A→B in many ways

pico-architecture

Paris city map

Medium-level components, medium-scale and organized dependencies, from A→B in a few ways

micro-architecture

macro-architecture

Top-level components, minimal call dependencies, from A → B practically in one way

Washington state map

API = road, call = route, pico = square, micro= area, macro= city

Πρόταση – ένας καλός τρόπος οργάνωσης με τις τεχνικές αυτές