HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

CSD

ΕΝΟΤΗΤΑ 4

ΣΤΟΙΧΕΙΑ ΟΝΤΟΚΕΝΤΡΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Αριθμός διαλέξεων 7, Διάλεξη 7η

CSD

Περιεχόμενα

- *Προηγμένες τεχνικές*
  - *Το πρόβλημα της εντροπίας λογισμικού*
    - Software entropy
  - Re-factoring
    - Δημιουργική αναδιάρθρωση
  - Θεωρητική απόδειξη λειτουργικότητας
  - Προγραμματιστικά ιδιώματα
  - Άμεση κλήση constructor / destructor
  - new placement syntax
    - Συντακτικό τοποθέτησης σε ειδική μνήμη για new
  - Μετατροπή των C unions σε C++ κλάσεις

CSD

Εντροπία λογισμικού (1/7)

- *Η γένεση του προβλήματος*
  - → Το σύστημα αρχικά παράγεται με μία καλά σχεδιασμένη κατάσταση
  - → καθώς νέα λειτουργικότητα προστίθεται, η εικόνα της συνολικής δομής αρχίζει σταδιακά να «θολώνει»
  - → τελικά, η δομή «αποσυντίθεται» σε ένα ανοργάνωτο λογισμικό πλέγμα
  - → μετά από μία «κρίσιμη μάζα», οι προγραμματιστές χάνουν πρακτικά κάθε εποπτεία και έλεγχο
  - → οι οποιεσδήποτε τοπικές εμβολές / ενέσεις λειτουργικότητας προκαλούν περισσότερο αρνητικές καθολικές συνέπειες
  - → το λογισμικό σύστημα είναι ουσιαστικά «νεκρό»,και είναι οικονομικά αποτελεσματικότερη η σχεδίαση και υλοποίηση εξαρχής, παρά η επαναχρησιμοποίηση του υπάρχοντος κώδικα.

Εντροπία λογισμικού (2/7)

- *Κρίσιμες ερωτήσεις*
  - Ποιο είναι το ποσοστό του προστιθέμενου πηγαίου κώδικα, επί της αρχικής καλά σχεδιασμένης μάζας, με το οποίο αρχίζει να «θολώνει» η σχεδιαστική εικόνα ?
  - Ποιες είναι οι συγκεκριμένες περιπτώσεις στις οποίες ο επιπλέον κώδικας αυξάνει την εντροπία ?
  - Πότε ποσοτικά μπορούμε να πούμε ότι προσεγγίζεται η κρίσιμη μάζα του πηγαίου κώδικα ?
  - Πως μπορούμε να ελέγχουμε εάν κινούμαστε σε πορεία αύξησης της εντροπίας ώστε να μπορέσουμε να αντιδράσουμε ?
  - → *Καλή μεταφορά για τα λογισμικά συστήματα είναι η αντιστοιχία με την αύξηση της εντροπίας σε μία πόλη όταν γίνεται άναρχη δόμηση και επέκταση εκτός του αυθεντικού σχεδίου*

Εντροπία λογισμικού (3/7)

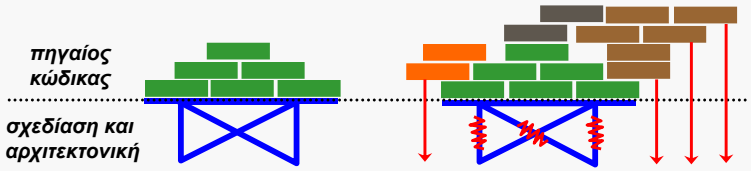
- *Πιθανές απαντήσεις ?*
  - → Δεν υπάρχει σχετικό τυποποιημένο μαθηματικό μοντέλο το οποίο να δίνει εξισώσεις αποτίμησης και πρόβλεψης της συνάρτησης αύξησης με καλά ορισμένες παραμέτρους.
  - ← Ωστόσο γνωρίζουμε ότι η αύξησή της επηρεάζεται θετικά από συγκεκριμένες ενέργειες και τακτικές ανάπτυξης.
  - ← Η δυνατότητα των προγραμματιστών να χειρίζονται και να κατασκευάζουν συστήματα μεγάλης κλίμακας είναι άμεσα εξαρτημένη από την δεξιότητά τους να αντιμετωπίζουν τη λογισμική εντροπία
    - Συνήθως οι προγραμματιστές αγνοούν ότι η ανάπτυξη συστημάτων δεκαπλάσιου μεγέθους απαιτεί «εκατονταπλάσιες» γνώσεις και δεξιότητες.

Εντροπία λογισμικού (4/7)

- *Είναι η «ρίζα του κακού»*
  - Όποτε προσθέτουμε ένα νέο χαρακτηριστικό σε ένα υπάρχον σύστημα, ουσιαστικά χτίζουμε πάνω στην αυθεντική του σχεδίαση
    - → συχνά με τρόπο στον οποίο η αρχική σχεδίαση δεν αποσκοπούσε.
  - Αυτή η ενέργεια δεν έπεται μίας προσεκτικής αξιολόγησης βασισμένη στην *αντοχή της αυθεντικής σχεδίασης*
    - → γεγονός που οδηγεί στην διαδοχική καταστροφή της σχεδίασης, η οποία είναι καταδικασμένη τελικώς να καταρρεύσει
  - Ο χρόνος απλώς καταναλώνεται για αλγοριθμική αντιμετώπιση του εκάστοτε νέου προβλήματος / χαρακτηριστικού
    - → αλλά δεν αφιερώνεται χρόνος για την εξασφάλιση της ομαλής και βέλτιστης ενσωμάτωσης στην υπάρχουσα βάση πηγαίου κώδικα

Εντροπία λογισμικού (5/7)

- *Η αρχή της σχεδιαστικής αντοχής και ισορροπίας*



- → Έλλειψη σταθερότητας κώδικα
- → Έλλειψη σταθερότητας σχεδίασης
- → Κατάρρευση σχεδίασης

Εντροπία λογισμικού (6/7)

Ορισμός

Η τάση κατά την οποία για το λογισμικό με την πάροδο του χρόνου καθίσταται δύσκολη και ακριβή η συντήρηση και διατήρηση. Ένα σύστημα το οποίο υφίσταται συνεχείς μεταβολές, όπως η προσθήκη νέας λειτουργικότητας με βάση την αυθεντική σχεδίαση, καταλήγει τελικά να είναι πιο πολύπλοκο και μπορεί γίνει ανοργάνωτο καθώς διευρύνεται, χάνοντας την αυθεντική σχεδιαστική του δομή. Θεωρητικά, ενδέχεται να είναι καλύτερο να σχεδιαστεί από την αρχή το σύστημα για την υποστήριξη των τροποποιήσεων παρά να χτίσει κάποιος στην υπάρχουσα δομή, όμως η επανασχεδίαση ενός υπάρχοντος συστήματος απαιτεί περισσότερη δουλειά καθώς θα οδηγήσει αναπόφευκτα σε νέα σφάλματα και προβλήματα.

Εντροπία λογισμικού (7/7)

"Broken Window Theory" *(excerpt from Pragmatic Programmers)*

Initial state In inner cities, some buildings are beautiful and clean, while others are rotting hulks. Why? Researchers in the field of crime and urban decay discovered a fascinating trigger mechanism, one that very quickly turns a clean, intact, inhabited building into a smashed and abandoned derelict.

Triggering event One broken window, left un-repaired for any substantial length of time, instills in the inhabitants of the building a sense of abandonment- a sense that the powers don't care about the building. So another window gets broken. People start littering. Graffiti appears. Serious structural damage begins.

Final state In a relatively short space of time, the building becomes damaged beyond the owner's desire to fix it, and the sense of abandonment becomes reality.

Περιεχόμενα

- *Προηγμένες τεχνικές*
  - Το πρόβλημα της εντροπίας λογισμικού
    - Software entropy
  - *Refactoring*
    - Δημιουργική αναδιάρθρωση
  - Θεωρητική απόδειξη λειτουργικότητας
  - Προγραμματιστικά ιδιώματα
  - Άμεση κλήση constructor / destructor
  - new placement syntax
    - Συντακτικό τοποθέτησης σε ειδική μνήμη για new

Δημιουργική αναδιάρθρωση (1/7)

- *Η θεραπεία της λογισμικής εντροπίας ακούει στο όνομα refactoring*
  - Θεμελιώδης τεχνική του ακραίου προγραμματισμού (extreme programming) και των ευκίνητων διαδικασιών ανάπτυξης (agile development)
  - Εάν είναι «αποδεκτή» η επανασχεδίαση, ή εάν δεν μπορεί να αποφευχθεί, τότε ναι σχεδίασε από την αρχή
    - Αυτό δουλεύει είτε για πολύ μικρά συστήματα / υποσυστήματα, ή αναγκαστικά για συστήματα που είναι ήδη καταδικασμένα
  - Αλλά στην πλειονότητα, δεν μπορούμε να αλλάξουμε όλη τη σχεδίαση, ούτε καν να εφαρμόσουμε απευθείας δραστικές τροποποιήσεις, διότι
    - θα χάσουμε πολύ χρόνο σε αναδιοργάνωση και γράψιμο εξαρχής πηγαίου κώδικα,
    - και όταν τελειώσουμε, αναμένουμε την εμφάνιση πολλών νέων bugs

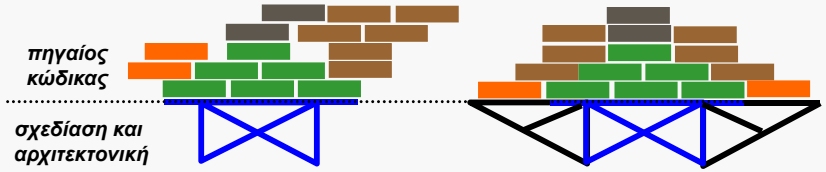
Δημιουργική αναδιάρθρωση (2/7)

- *Τι ρόλο παίζει το refactoring (1/2)*
  - Μία τεχνική εξέλιξης της σχεδίασης, χωρίς ωστόσο να συνιστά σχεδιαστική τεχνική
  - Επεκτείνει τη σχεδίαση πάντα ένα βήμα μακρύτερα από το σημείο στο οποίο θα σταματήσουν οι ενδεχόμενες προσθήκες πηγαίου κώδικα
    - ενώ είναι περισσότερο μία «θεραπευτική» τεχνική, υποθέτει ότι μάλλον κάτι παρόμοιο θα ξαναγίνει και προετοιμάζει «προληπτικά» την σχεδίαση

Δημιουργική αναδιάρθρωση (3/7)

- *Τι ρόλο παίζει το refactoring (2/2)*
  - Σπρώχνει στα όρια την σχεδίαση, στο μέγιστο ανεκτό μέγεθος κώδικα
  - όμως εάν αδυνατούμε πια να αντιμετωπίσουμε την εντροπία μέσω refactoring, ξέρουμε ότι φτάσαμε στο επικίνδυνο σημείο της "μη επιστροφής", όπου η σχεδίαση πρέπει επαναπροσδιοριστεί
  - Δεν πρέπει να θεωρείται ως μία μέθοδος δυναμικής σχεδίασης, ούτε ως μία τακτική επιβίωσης λογισμικών συστημάτων με «λιγότερο από άριστη» σχεδίαση, αλλά ως *μέθοδος για βέλτιστη διατήρηση μίας βέλτιστης σχεδίασης*
    - Εάν επιμένετε στην εξέλιξη λάθος σχεδίασης, το re-factoring θα βοηθήσει το λογισμικό σας να ζήσει μεν περισσότερο, αλλά θα οδηγήσει αναμφίβολα σε μία πολύ επώδυνη και άδοξη κατάρρευση

Δημιουργική αναδιάρθρωση (4/7)



- ➔ Επεκτείνεται η σχεδίαση χωρίς να διαφοροποιείται από την αρχική στρατηγική και φιλοσοφία
- ➔ Ο πηγαίος κώδικας αναδιοργανώνεται προσεκτικά, ώστε να επηρεάζεται ελάχιστα η προηγούμενη δομή
- ➔ Αναμένουμε επιπλέον εισαγωγές κώδικα, καθιστώντας την σχεδίαση ικανή να αντεπεξέλθει σε μελλοντικές προσθήκες

Δημιουργική αναδιάρθρωση (5/7)

- *Η διαδικασία refactoring*
  1. Εντόπισε τα σημεία στα οποία η σχεδίαση φαίνεται να αποτυχαίνει, οδηγώντας πιθανότατα σε αύξηση εντροπίας
  2. Προσπάθησε να οριοθετήσεις τοπικά το πρόβλημα και αποφάσισε τις απαιτούμενες επεμβάσεις ενδυνάμωσης της σχεδίασης
  3. Πιστοποίησε την καταλληλότητα των επεμβάσεων πριν τις εφαρμόσεις με ανάλυση κώδικα και σενάρια χρήσης
  4. Εφάρμοσε τις μεταβολές βαθμιαία, επικυρώνοντας σε κάθε βήμα την ορθή λειτουργία του συστήματος, έως ότου γίνουν όλες οι αλλαγές
  5. Εάν είναι δυνατό, εισήγαγε την νέα σχεδίαση έως παράλληλα διαθέσιμο κώδικα, χωρίς να διαγράφεται ο αρχικός (τεχνική διαδοχικής παράκαμψης)
  6. Πρόσθεσε τον επιπλέον κώδικα στη νέα δομή, και έλεγξε και πάλι την καταλληλότητα της αναδιάρθρωσης
  7. Τεκμηρίωσε πιθανές προσθήκες για τις οποίες η σχεδίαση είναι ήδη έτοιμη

Δημιουργική αναδιάρθρωση (6/7)

- *Εφαρμόζεται σε μικρά προσεκτικά βήματα*
  - Αλλαγή αναγνωριστικών ονομάτων
  - Μετακίνηση μεταβλητών
  - Μεταβολή τελεστών πρόσβασης σε κλάσεις
  - Συγχώνευση / διάσπαση / τροποποίηση υλοποίησης συναρτήσεων
  - Αλλαγή «υπογραφών»
  - Προσθήκη ελέγχου λαθών και διαγνωστικών τμημάτων
  - Δομική αναδιοργάνωση και εσωτερική τεκμηρίωση
  - «Ενέσεις»καλυτέρευσης απόδοσης, βελτιστοποίηση αλγορίθμων
  - Αναδιοργάνωση αρχείων
  - Εξάλειψη επαναλαμβανόμενου κώδικα
  - Κατάτμηση πολύπλοκης λειτουργικότητας σε πολλές συναρτήσεις
  - Αντικατάσταση κλάσεων και συναρτήσεων με διαδοχική απομόνωση
  - Ανάδυση αρχιτεκτονικών τμημάτων

Δημιουργική αναδιάρθρωση (7/7)

- *Πότε είναι η κατάλληλη στιγμή ?*
  - Ποτέ δεν εφαρμόζουμε re-factoring και προσθήκη νέας λειτουργικότητας ταυτόχρονα: μπορούμε μεν να μεταβαίνουμε από το ένα στο άλλο, αλλά μόνο για αυτοτελείς ενέργειες (no s/w zapping) – ποτέ παράλληλα !
  - Πάντοτε έχουμε έτοιμα tests πριν το re-factoring, ώστε να ελέγχουμε ότι δεν «καταστρέφεται» κάτι: να εφαρμόζετε τα tests όσο το δυνατόν συχνότερα (όχι απλά «re-compile και φύγαμε»)
  - Οργανώστε τις αλλαγές σε μικρά βήματα: μην επιδιώκετε μεγάλες μεταβολές σε ένα βήμα (π.χ. αποφύγετε να αλλάζετε τα μέλη – δεδομένα πολλών κλάσεων με μία «κίνηση»)
  - Ποτέ δεν κάνουμε re-factoring χωρίς προετοιμασία. Σχεδιάστε προσεκτικά την τακτική αναδιάρθρωσης πριν ακουμπήσετε τον κώδικα
  - Ποτέ δεν εφαρμόζουμε re-factoring όταν δεν είμαστε σε καλή κατάσταση. Το μυαλό σας πρέπει να βρίσκεται σε εγρήγορση ώστε να ελέγχει και να διαχειρίζεται κρίσιμες παραμέτρους και εξαρτήσεις (*yes vodka ? no re-factoring!*)

Refactoring techniques (1/7)

Extract Method refactoring
Έχουμε συναρτήσεις που είναι μεγάλες και πρέπει να γίνουν split σε μικρότερες.
Έχουμε συναρτήσεις που εμπεριέχουν ένα επαναλαμβανόμενο μοτίβο κώδικα.
Ο κώδικας μεταφέρεται σε μία ξεχωριστή συνάρτηση.

1. Εντοπίζουμε το τμήμα του κώδικα.
2. Ενοποιούμε τις διαφορές με παραμετροποίηση.
3. Εντοπίζουμε τις εξαρτήσεις με τοπικές μεταβλητές.
4. Μετατρέπουμε τις εξαρτήσεις σε παραμέτρους. Read-only variables γίνονται const references. Modified variables γίνονται pointers. Μερικές φορές εάν έχουμε μοναδικό written variable γίνεται return value.
5. Βρίσκουμε ένα αντιπροσωπευτικό όνομα.
6. Αλλάζουμε τον κώδικα.
7. Rerun your tests and repair.

Field Encapsulation refactoring
Λέγεται και data hiding και είναι πολύ απλό και συνηθισμένο. Έχουμε fields τα οποία χρησιμοποιούνται απευθείας σε κάποιες methods άλλων κλάσεων. Μετατρέπουμε το field σε attribute και αλλάζουμε τη χρήση του σε όλα τα σημεία.

Refactoring techniques (2/7)

```
void f (int a, const std::string& b, double c) {

    // Part 1.

    std::string d;
    std::list<unsigned> e;

    // Part 2, uses 'd' and 'a' and produces 'e'.
    // EXTRACT METHOD.

    // Part3, uses 'e'.
}
```

```
void f_new (int a, const std::string& b, double c) {

    // Part 1.

    std::string d;
    std::list<unsigned> e;

    f_part2(a, b, &e);

    // Part3, uses 'e'.
}
```

```
void f_part2 (int a, const std::string& b, std::list<unsigned>* e) {
    // Part 2
}
```

Refactoring techniques (3/7)

Extra Class refactoring
Έχουμε μία κλάση η οποία κάνει δουλειά που θα έπρεπε να γίνεται από δύο ή και περισσότερες κλάσεις. Συνήθως τέτοιες κλάσεις αναγνωρίζονται επειδή έχουν πολλά data members και methods.
Δημιουργούμε μία νέα κλάση και μεταφέρουμε όλα τα data members και methods στη νέα κλάση.

1. Εντοπίζουμε την κλάση
2. Αποφασίζουμε πώς να διασπάσουμε την κλάση σε διαφορετικές κλάσεις (ουσιαστικά κάνουμε role splitting / split of responsibilities)
3. Ενδέχεται η παλιά κλάση να αντιπροσωπεύει πλέον ευθύνη / ρόλο διαφορετικό από ότι το αρχικό όνομα, οπότε αυτό μπορεί να γίνει dropped.
4. Οριοθετούμε ένα link (association) από την παλιά κλάση στη νέα (π.χ. *#include* στην C++).
5. Ορίζουμε ένα νέο field στην αρχική κλάση με τύπο την νέα κλάσης.
6. Μετακινούμε ένα-ένα τα data members και methods που πρέπει να μετακινηθούν στη νέα κλάση, αλλάζοντας τη χρήση τους στην αρχική κλάση να γίνεται μέσω του νέου field.
7. Εάν έχουμε methods στη νέα κλάση που χρησιμοποιούν data members της αρχικής πρέπει να έχουμε link από την νέα κλάση και προς την αρχική μέσω ενός local reference (ή pointer) που θα λαμβάνεται στο construction της αρχική κλάσης.
8. Rerun your tests and repair.

Refactoring techniques (4/7)

```
class B {

    private:
    int a;
    std::string b;
    double c;
    struct D {} d;

    void f1 (double* n) { /* change n */ }
    const D f4 (std::string* s)
        { /* change s*/ f2(&a); return d; }

    public:
    void f2 (int* p) { *p = f3(b); f1(&c); }
    int  f3 (const std::string& s) { /* use s*/ return 1; }
};
```

```
class B1 {

    private:
    int a;
    std::string b;
    struct D {} d;
    B2 b2;

    const D f4 (std::string* s)
        { /* change s*/ f2(&a); return d; }

    public:
    void f2 (int* p) { *p = b2.f3(b); b2.f1(); }
};
```

Extracted class

Modified class

```
class B2 {
    friend class B1;
    private:
    double c;
    void f1 (void) { /* change */ }

    public:
    int  f3 (const std::string& s) { /* use s*/ return 1; }
};
```

Absorb parameter to private field in private method: **Σε private method περνάμε argument ένα private field. Τότε αφαιρούμε την παράμετρο και στον κώδικα χρησιμοποιούμε απευθείας το field.**

Refactoring techniques (5/7)

Pull Up Method refactoring
Έχουμε methods με ίδια αποτελέσματα και συμπεριφορά σε διαφορετικά subclasses. Τις μετακινούμε ενοποιώντας τες στην superclass. Συνήθως αυτό σημαίνει ότι είχαμε συντηρητική σχεδίαση ως προς το τι τοποθετείται στην superclass.

1. Επιθεώρησε τις methods με προσοχή ώστε να σιγουρευτούμε ότι κάνουν ακριβώς τα ίδια (ακόμη και εάν αυτό γίνεται με διαφορετικό τρόπο).
2. Προσέχουμε ιδιαίτερα τις κλήσεις σε άλλα methods που μπορεί να προκαλέσουν διαφορετικά αποτελέσματα.
3. Εάν δεν είμαστε σίγουροι ότι κάνουν το ίδιο πράγμα, εφαρμόζουμε το Algorithm Substitution Refactoring ώστε να έχουν ίδιο source code και κάνουμε testing.
4. Εάν έχουν διαφορετικά signatures, άλλαξε τα signatures ώστε να είναι αυτό που πρόκειται να έχεις στην superclass.
5. Φτιάξε μία νέα method στο superclass, κάνει copy το σώμα method από το subclass, προσάρμοσέ το και κάνε compile.
6. Άλλαξε τις κλήσεις ώστε να είναι προσαρμοσμένες στο superclass method.
7. Εάν η method είχε πρόσβαση σε subclass field, μετακίνησέ τα με abstract attribute.

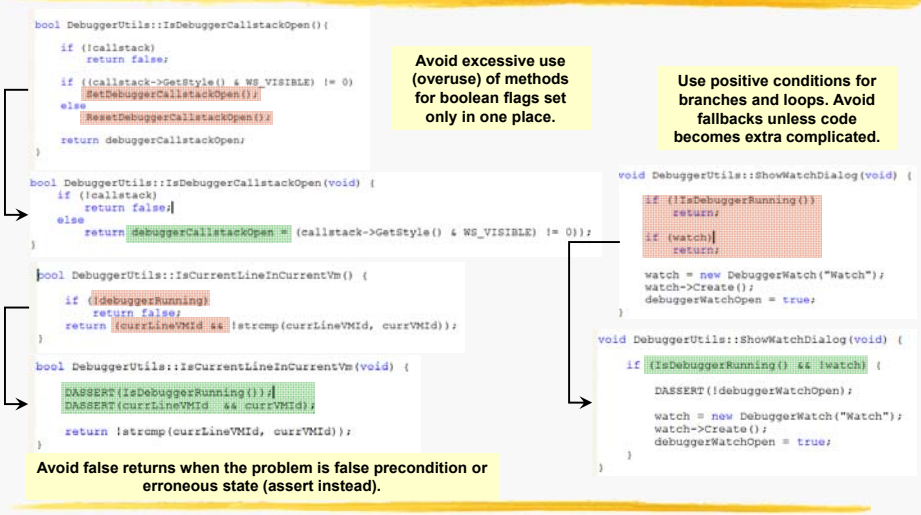
Refactoring techniques (6/7)

```
class Base {
    public:
    virtual void F (void);
};

class Derived1 : public Base {
    int a;
    public:
    virtual void F (void);
    virtual void G (void){ a = 10; }
};

class Derived2 : public Base {
    int a;
    public:
    virtual void G (void){ a = 10; }
};
```

```
class Base_new {
    public:
    virtual int Get_a (void) const = 0;
    virtual void Set_a (int a) = 0;
    virtual void G (void){ Set_a(10); }

    virtual void F (void);
};

class Derived1_new : public Base_new {
    public:
};

class Derived2_new : public Base_new {
    public:
    virtual void F (void);
};
```

***Field Encapsulation and Abstract Field*: Θέλουμε να υλοποιήσουμε μία method σε ένα superclass με πρόσβαση σε fields τα οποία θα τα παρέχει το derived class.**

Refactoring techniques (7/7)



Περιεχόμενα

- *Προηγμένες τεχνικές*
  - Το πρόβλημα της εντροπίας λογισμικού
    - Software entropy
  - Refactoring
    - Δημιουργική αναδιάρθρωση
  - *Θεωρητική απόδειξη λειτουργικότητας*
  - Προγραμματιστικά ιδιώματα
  - Άμεση κλήση constructor / destructor
  - new placement syntax
    - Συντακτικό τοποθέτησης σε ειδική μνήμη για new

Θεωρητική απόδειξη λειτουργικότητας (1/6)

- Κατά τη σχεδίαση και υλοποίηση μίας συνάρτησης συνήθως σκεφτόμαστε άτυπα ως εξής:
  - εάν τα πραγματικά ορίσματα είναι αυτά που προϋποθέτει η συνάρτηση
  - εάν η συνάρτηση «δουλεύει» σωστά και παράγει τα σωστά αποτελέσματα
  - ➔ τότε αυτή η συνάρτηση δεν πρόκειται να θεωρηθεί υπαίτια για παραγωγή λαθών
    - ➔ δηλαδή, μπορεί να χαρακτηριστεί ως πιστοποιημένη για ορθή λειτουργία

Θεωρητική απόδειξη λειτουργικότητας (2/6)

- Ο κύριος λόγος για τον οποίο ο προηγούμενος συλλογισμός αδυνατεί να πετύχει τον στόχο μας, δηλ. την απόδειξη ορθότητας, είναι η απουσία κεντρικής τεκμηρίωσης της λογικής ορθότητας
  - Συνήθως σκεφτόμαστε τι σημαίνει «κάτι» να είναι ορθό, είτε όταν ακριβώς το υλοποιούμε, ή αφού έχει αποδειχθεί ότι κάτι πήγε στραβά
    - Αλλά ποτέ σχεδόν δεν περιγράφουμε τυπικά το ορθό και το σφάλμα εκ των υστέρων κατά τη διαδικασία σχεδίασης
- ➔ Έχουμε δει ότι *ένας πολύ καλός τρόπος να καλύψουμε το κενό της έλλειψης σημασιολογίας ορθότητας* είναι η τεχνική design by contract.

CSD

Θεωρητική απόδειξη λειτουργικότητας (3/6)

- Η τεκμηρίωση και υλοποίηση της λογικής βάσει της τεχνικής design by contract, οδηγεί στην δημιουργία πηγαίου κώδικα που αντιπροσωπεύει την λογική αυτή και ελέγχει την λειτουργική ορθότητα

➔ *Ακολουθώντας την μέθοδο αυτή,η απόδειξη της ορθότητας γίνεται φυσιολογικά με επαγωγή.*

CSD

Θεωρητική απόδειξη λειτουργικότητας (4/6)

- Έστω ένα στιγμιότυπο ***O*** της κλάσης ***C***, πιστοποιημένο ως προς την ορθότητά του μέσω του invariant ***I***, το οποίο και υλοποιεί την σχετική προς την κλάση ***C*** λογική ορθότητας. Τότε:
  - Η συνάρτηση ***F*** έχει ορθή υλοποίηση ⇔ σε όλες τις περιπτώσεις τις οποίες η precondition της ***F*** είναι *true*, και η postcondition είναι *true* μετά από οποιαδήποτε κλήση της ***F***
  - Για κάθε τέτοια ορθά υλοποιημένη συνάρτηση ***F,*** η κλήση της ***F*** μέσω του στιγμιότυπου ***O*** έχει ως αποτέλεσμα το ***O*** να παραμένει ορθό, χωρίς την παραγωγή οιονδήποτε λανθασμένων πλαγίων αποτελεσμάτων

CSD

Θεωρητική απόδειξη λειτουργικότητας (5/6)

- Οι προηγούμενοι ορισμοί ουσιαστικά λένε ότι:
  - εάν προγραμματίσουμε τη λογική ορθότητας ως assertions
  - εάν χρησιμοποιήσουμε αυτά τα assertions όπως υποδεικνύεται από την τεχνική design by contract
  - εάν υλοποιήσουμε κάθε συνάρτηση αποδεικνύοντας ότι κάνει αυτό που πρέπει να κάνει
  - ➔ *τότε η υλοποίηση του προγράμματος είναι αποδεδειγμένα ορθή*

CSD

Θεωρητική απόδειξη λειτουργικότητας (6/6)

- *Επαγωγική απόδειξη ορθότητας υλοποίησης*
  - Σε αλγοριθμικό επίπεδο
    - Ότι ο αλγόριθμος πραγματικά κάνει αυτό που δηλώνει ότι κάνει
      - Χρειάζεται τεκμηρίωση του αλγορίθμου
  - Ανά εντολή
    - Ότι κάθε εντολή, ή σειρά εντολών, αντιστοιχούν σε ένα καλά ορισμένο βήμα του αλγορίθμου, παράγοντας μία ενέργεια σε πλήρη συμφωνία με τον αντίστοιχο αλγόριθμο
      - Πρέπει να τεμαχίσουμε λογικά τον κώδικα σε κατάλληλα βήματα, με εισαγωγή σχολίων, βάσει της αλγοριθμικής τεκμηρίωσης
  - ➔ Η τεκμηρίωση του αλγορίθμου και τα σχόλια στον κώδικα βοηθούν την επαγωγική σκέψη
    - ➔ με την οποία αποδεικνύουμε ότι ο κώδικας υλοποιεί ορθά και σε πλήρη αντιστοιχία ότι έχουμε σχεδιάσει και επιδιώξει μέσω του αυθεντικού αλγορίθμου

Περιεχόμενα

- *Προηγμένες τεχνικές*
  - Το πρόβλημα της εντροπίας λογισμικού
    - Software entropy
  - Re-factoring
    - Δημιουργική αναδιάρθρωση
  - Θεωρητική απόδειξη λειτουργικότητας
  - *Προγραμματιστικά ιδιώματα – πρόσθετο υλικό*
  - Άμεση κλήση constructor / destructor
  - new placement syntax
    - Συντακτικό τοποθέτησης σε ειδική μνήμη για new

Προγραμματιστικά ιδιώματα (1/6)

- Πρόκειται για ειδικές τεχνικές οι οποίες βασίζονται σε χρήση σημασιολογικών χαρακτηριστικών της γλώσσας με διόλου προφανή τρόπο
- Πρέπει να γνωρίζετε καλά τη σχετική σημασιολογία για να τα χρησιμοποιείτε χωρίς κίνδυνο
- Δεν είναι πάντα δείγματα καλής σχεδίασης, αλλά σε ορισμένες περιπτώσεις μπορεί να φανούν πολύ χρήσιμα

Προγραμματιστικά ιδιώματα (2/6)

Προσωρινά αντικείμενα

```
const X f (const X& x1, const X& x2)
{
  X x;
  x.g(x1);
  x.h(x2);
  return x;
}
```

Ένα προσωρινό αντικείμενο X δημιουργείται, με χρόνο ζωής αυτό της εντολής στην οποία παράγεται, ως εξής:

- Εάν υπάρχει move ή copy constructor (με αυτή την προτεραιότητα) χρησιμοποιείται, αλλιώς
- ένα μη αρχικοποιημένο στιγμιότυπο δημιουργείται, και μία διαδικασία αντιγραφής μνήμης (bit copy) από το **x** προς το προσωρινό στιγμιότυπο λαμβάνει χώρα.

```
const X g (const X& x1, const X& x2) {
  return x1.a() + x2.b();
}
X x = 10; x = g(5,6);
```

Όπου `X (int);` ορισμένη.
Όπου `int a();` ορισμένη.
Όπου `int b();` ορισμένη.

- Ένα προσωρινό στιγμιότυπο επιστρέφεται,το οποίο δημιουργείται με αυτόματη κλήση του converter constructor **X(int)**.
- Εάν δεν επιθυμούμε ο compiler να εφαρμόζει τέτοιες αυτόματες μετατροπές με τη χρήση των constructors, πρέπει να χαρακτηρίσουμε τον converter constructor ως explicit.

```
explicit X (int);
```


Προγραμματιστικά ιδιώματα (3/6)

Κώδικας κατά την επιστροφή από συνάρτηση (1/2)

```
... f() {
  if (...)
     return ...;
  while (...) {
     if (...)
        return ...;
   }
}
```

- Χρειάζεται να κάνουμε log την είσοδο / έξοδο σε / από συνάρτηση, μόνο σε debugging mode.
- Πρέπει να μπορεί αυτό να γίνεται εύκολα, ακόμη και για μη παγιωμένες αλγοριθμικά συναρτήσεις (δηλ. θα αλλάζουν συχνά).

```
debug.Log(<format>, <args>);
```

- Ακολουθεί η κοινή πρακτική. Ως αποτέλεσμα ο κώδικας γίνεται αρκετά είναι υπερφορτωμένος, ενώ σε κάθε νέο return πρέπει να βάλουμε μία κατάλληλη κλήση logging.

```
... f() {
  debug.Log("Enter:f()\n");
  if (...) {
     debug.Log("Exit:f()\n");
     return ...;
  }
  while (...) {
     if (...) {
        debug.Log("Exit:f()\n");
        return ...;
     }
   }
   debug.Log("Exit:f()\n");
}
```

Προγραμματιστικά ιδιώματα (4/6)

Κώδικας κατά την επιστροφή από συνάρτηση (2/2)

- Μια προχωρημένη και πιο κομψή μέθοδος. Βασιζόμαστε στη σημασιολογία κλήσης των destructors για τοπικά στιγμιότυπα.
- *Το μοναδικό μειονέκτημα είναι η αδυναμία πρόσβασης σε τοπικές μεταβλητές της συνάρτησης μέσα στον destructor της βοηθητικής κλάσης.*

```
#define ON_BLOCKEXIT(code) \      ← Μόνο κώδικας χωρίς κόμμα !
   class _Temp { public: ~_Temp(){ code } }; _Temp __temp;

... f() {
  debug.Log("Enter: f()");
  ON_BLOCKEXIT(debug.Log("Exit: f()");)
  if (...)
     return ...;
  while (...) {
     if (...)
        return ...;
   }
}
```


Προγραμματιστικά ιδιώματα (5/6)

Σειρά αρχικοποίησης static στιγμιότυπων (1/2)

```
// x.cpp
static X x;
X& Xref(void) { return x; }

// y.cpp
static Y y;
Y& Yref(void) { return y; }

// init.cpp
class Init {
  public:
  Init(void) {
    Xref().Init();
    Yref().Init();
  }
};
static Init init;
```

- Τι λάθος υπάρχει στον κώδικα ? Υπάρχουν οι παρακάτω έξι διαφορετικές ακολουθίες κλήσης των constructors για τα static στιγμιότυπα **x**, **y**, και **init** κατά την έναρξη του προγράμματος:

x → y → init
x → init → y
y → x → init
y → init → x
init → x → y
init → y → x

- Από αυτές, μόνο στις δύο ο κώδικας θα συμπεριφέρονταν ορθά. Όμως, δεν υπάρχει καμία εγγύηση ποια ακολουθία θα επιλέξει το run-time library του compiler.
- Αυτή η συμπεριφορά ορίζεται στο standard ως "implementation dependent" ***και τέτοιος κώδικας πρέπει να αποφεύγεται.*** Μπορεί να είστε άτυχοι και να επιλέγεται η «ευνοϊκή» ακολουθία, και όλα να φαίνονται σωστά.

Προγραμματιστικά ιδιώματα (6/6)

Σειρά αρχικοποίησης static στιγμιότυπων (2/2)

- Για static στιγμιότυπα κλάσης Α, στην δημιουργία τους (κλήση constructor), πρέπει να εξασφαλίζεται ότι ένα static στιγμιότυπο κλάσης Β έχει ήδη δημιουργηθεί, ώστε να είναι νόμιμη η κλήση συναρτήσεων του Β.
- Τα Α και Β βρίσκονται σε διαφορετικά αρχεία. Για να λυθεί αυτό το πρόβλημα, θα πρέπει να μετατραπεί το Β στιγμιότυπο σε δυναμικό (δηλ. να δημιουργηθεί με new) όπως το παρακάτω σχήμα κώδικα:

```
// B.h
class B_Init {
 private:
   static bool initDone;
 public:
   B_Init(void);
};
static B_Init B_InitVar;
extern B& Binstance (void);

// A.h
#include "B.h"
```

→ Να γίνει #include πριν από κώδικα όπου οποιοδήποτε static Α στιγμιότυπο χρησιμοποιεί το Β στιγμιότυπο.

```
// B.cpp
static B* b;
B& Binstance (void) { assert(b); return *b; }
bool B_Init::initDone;

B_Init::B_Init (void) {
   if (!initDone) {
      b = new B;
      initDone = true;
   }
}
```

Μην βάλετε initializer, δηλ. `b=...;` ή `initDone=...;`

→ *Είναι κρίσιμο να μην βάλετε initializer ούτε στο* **b**, *ούτε στο* **initDone**. Η τεχνική βασίζεται στο γεγονός ότι, *πριν την έναρξη του προγράμματος, η μνήμη για static αντικείμενα γράφεται ανά byte με την τιμή 0.*

Περιεχόμενα

- *Προηγμένες τεχνικές*
  - Το πρόβλημα της εντροπίας λογισμικού
    - Software entropy
  - Re-factoring
    - Δημιουργική αναδιάρθρωση
  - Θεωρητική απόδειξη λειτουργικότητας
  - Προγραμματιστικά ιδιώματα
  - *Άμεση κλήση constructor / destructor*
  - new placement syntax
    - Συντακτικό τοποθέτησης σε ειδική μνήμη για new

CSD

Άμεση κλήση constructor / destructor (1/3)

- Όπως είδαμε από τον ορισμό των constructor / destructor συναρτήσεων, επιτρέπεται η κλήση τους, όπως και κάθε άλλης συνάρτησης μέλους
  - Πέραν της αυτόματης κλήσης τους στις περιπτώσεις που αναφέρονται στον πίνακα

	constructor	*destructor*
Static (σε κλάση) ή σε καθολική εμβέλεια	**Ακριβώς πριν την κλήση της main**	**Αμέσως μετά την έξοδο από την main**
Τοπικά, δηλωμένα μέσα σε block	**Ακριβώς στο σημείο της δήλωσης του στιγμιότυπου**	**Μετά την τελευταία εντολή του block, με σειρά αντίστροφη της σειράς δήλωσης των στιγμιότυπων**
Δυναμικά instances, με κλήση new / delete	**Μετά την παραχώρηση της μνήμης, πριν επιτρέψει η new**	**Πριν την απελευθέρωση της μνήμης (η οποία και έπεται)**

CSD

Άμεση κλήση constructor / destructor (2/3)

- Εκτός του ότι προσφέρονται ως συναρτήσεις αρχικοποίησης (οικογένεια *initialize*) ή κάθαρσης (οικογένεια *cleanup*),
  - δίνουν *απεριόριστες* δυνατότητες χρήσης της μνήμης ως στιγμιότυπα

```
class Point {
   private:
      int x, y;
   public:
      Point (void) : x(0), y(0){}
      Point (int _x, int _y) :  x(_x), y(_y) {}
};

unsigned char mem[sizeof(Point)];
Point* pt = (Point*) mem;
pt->Point::Point(3,-5);
```


CSD

Άμεση κλήση constructor / destructor (3/3)

- Δυνατότητες εξελιγμένης επαναχρησιμοποίησης και ανακύκλωσης μνήμης
  - DMR, *Dynamic Memory Recycler* (*από το I-GET UIMS, Savidis 1997*)

```
class X {
    static std::stack<X*> recycler;
    X (int a);    ← Attention, we use private constructors !
    static X* top_and_pop (void)
       { X* x = recycler.top(); recycler.pop(); return x; }
 public:
    static X* New (int a) {
       if (recycler.empty())
          return new X(a);
       else {
          X* x = top_and_pop();
          x->X::X(a); ← Explicit constructor call
          return x;
       }
    }
    void Delete (void)
      { this->~X(); recycler.push(this); }
};
```

} `return new (top_and_pop()) X(a);`

CSD

Περιεχόμενα

- *Προηγμένες τεχνικές*
  - Το πρόβλημα της εντροπίας λογισμικού
    - Software entropy
  - Re-factoring
    - Δημιουργική αναδιάρθρωση
  - Θεωρητική απόδειξη λειτουργικότητας
  - Προγραμματιστικά ιδιώματα
  - Άμεση κλήση constructor / destructor
  - *new placement syntax*
    - Συντακτικό τοποθέτησης σε ειδική μνήμη για new

CSD

new placement syntax (1/3)

- Μία ειδική χρήση της new, ή οποία σας δίνει τη δυνατότητα να μην λάβετε δυναμικά μνήμη για την δημιουργία ενός στιγμιότυπου, αλλά να επαναχρησιμοποιήσετε υπάρχουσα μνήμη
- Η εκδοχή αυτή ισχύει μόνο για τον απλό τελεστή **new** και όχι για τον **new[ ]**
- Όταν δημιουργείτε στιγμιότυπα με αυτό τον τρόπο, συνήθως εάν τα καταστρέφουμε με **delete** προκαλούμε system crash

CSD

new placement syntax (2/3)

Τυπολογία και παραδείγματα

new (*Έκφραση δείκτη*) *Κλάση* (*Ορίσματα constructor*)

- Η μνήμη που αρχίζει από την έκφραση δείκτη θεωρείται ως μνήμη στιγμιότυπου της κλάσης, με αποτέλεσμα να καλείται ο constructor της κλάσης που αντιστοιχεί στα ορίσματα που παρέχονται.
- Το αποτέλεσμα της συνολικής έκφρασης είναι η τιμή του δείκτη μετατρεπόμενη σε δείκτη της κλάσης.

```
X x;
new (&x) X(10);                          ← Χρήση της μνήμης του στιγμιότυπου x
X* xp = new (malloc(sizeof(X)) X(20);    ← Παίρνουμε μνήμη και αλλιώς,
delete xp;                               ← αλλά κάτι τέτοιο είναι system crash.
free(xp);                                ← Παράξενο φαίνεται, αλλά είναι το σωστό !
class Y {
   Y (const Y&);
   void operator=(const Y& y)
     { new (this) Y(y); } ← Υλοποίηση του = μέσω του copier constructor !
};
```


CSD

new placement syntax (3/3)

Γενική υλοποίηση του assignment operator

- Παρακάτω ακλουθεί η γενική υλοποίηση ενός overloaded assignment operator για όλες τις πιθανές περιπτώσεις χρησιμοποιώντας τον move / copy constructor
- Ενδέχεται ο assignment operator να είναι και void, σε όλες τις άλλες περιπτώσεις επιστρέφει μόνο const reference
- Τέλος, προσέχουμε να αποφύγουμε self-assignments καθώς είναι σίγουρο ότι προκαλούν πολύ δύσκολα runtime errors

```
class Y {
   Y (const Y&);
   const Y& operator=(const Y& y) {        ← requires a copy ctor to be present
           if (this != &y) {               ← check self assignment
              this->~Y();                  ← de-initialize
              return * new (this) Y(y);    ← re-initialize via copy ctor
           }
   }
  const Y& operator=(const Y&& y) {        ← requires a move ctor to be present
           this->~Y();                     ← de-initialize
           return * new (this) Y(y);       ← re-initialize via move ctor
   }
};
```


CSD

Μετατροπή των C unions (1/3)

- Τα unions της C, όπως μαρτυρεί και ο τίτλος τους, έχουν σκοπό να ενοποιήσουν πολλούς συγγενείς τύπους σε έναν
  - Που μπορεί να χρησιμοποιηθεί στη θέση οποιουδήποτε από τους εναλλακτικούς
  - Με μέγεθος αυτό του μεγαλύτερου τύπου
  - Παίζοντας έτσι το ρόλο ενός τύπου με πολλές εναλλακτικές μορφές (πολυμορφικός)

```
enum EventType { MouseEvent = 0, KeyboardEvent = 1 };
struct Event {
  EventType type;
  union {
    struct { unsigned x, y; } mouseData;
    struct { char key[16]; } keyboardData;
  } data;
};
```

Μετατροπή των C unions (2/3)

- Όμως υπάρχει καλύτερος και πιο επεκτάσιμος τρόπος για πολυμορφικούς τύπους στην C++
- Μερικές φορές μας βολεύει η χρήση union για πολύ απλές περιπτώσεις
  - αλλά με την πρώτη ευκαιρία καλό είναι να κάνουμε μετατροπή σε αληθινούς πολυμορφικούς C++ τύπους

```
struct Event { Event(void){} virtual ~Event(){} };
class MouseEvent : public Event {
private:
  unsigned x, y;
public:
  unsigned GetX (void) const { return x; }
  unsigned GetY (void) const { return y; }
  MouseEvent (unsigned _x, unsigned _y) : x(_x),y(_y){}
  ~MouseEvent(){}
};
struct KeyboardEvent : public Event { ... }
```


Μετατροπή των C unions (3/3)

- Βλέπουμε ότι η μεταβλητή τύπου (π.χ. EventType type) δεν «επιζεί» τελικά στην υλοποίηση σε C++
  - Αν και αυτός ο κανόνας μπορεί σε σπάνιες περιπτώσεις να παραβιάζεται (και εννοούμε σπάνιες...)
- Αυτό οφείλεται στην ανάγκη υπακοής στο LSP, βάσει του οποίου οι πολυμορφικοί αλγόριθμοι δεν πρέπει να γνωρίζουν τους κληρονόμους τύπους
  - άρα δεν χρειάζεται καμία τέτοια μεταβλητή μέσα στο super-class
- Όλα τα στιγμιότυπα παράγονται βάσει των κληρονόμων κλάσεων
- Όσο και εάν δεν το περιμένατε, *κερδίζουμε σε μνήμη στη C++* διότι κάθε στιγμιότυπο έχει τη ακριβώς τη μνήμη που χρειάζεται
  - ενώ με τα unions απαιτεί πάντα χώρο ίσο με το μέγεθος του μεγαλύτερου σε μνήμη τύπου