HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

CSD

ΕΝΟΤΗΤΑ 4

ΣΤΟΙΧΕΙΑ ΟΝΤΟΚΕΝΤΡΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Αριθμός διαλέξεων 7, Διάλεξη 6η

CSD

Περιεχόμενα

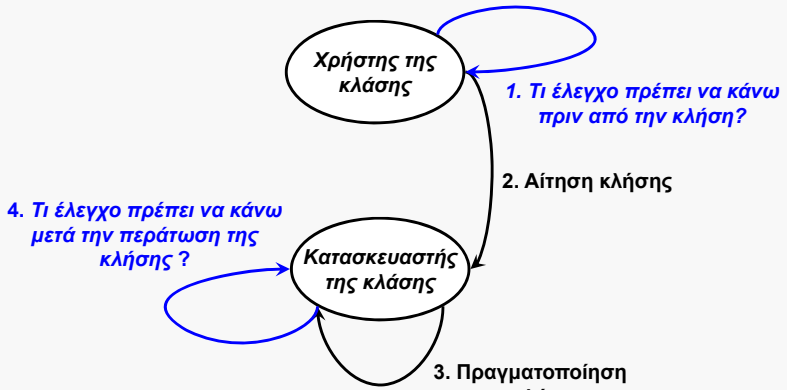
- *Εισαγωγή – design by contract*
- Preconditions
- Postconditions
- Invariants
- Representation invariants
- Πολυμορφισμός και design by contract
- LSP
- Application invariants
- Πρακτική εφαρμογή

CSD

Ένθετο

- Η «αγία τριάδα» των σχεδιαστικών τεχνικών
  - ***Design by Contract***
    - Defensive programming
    - Self-checked and self-documented code
  - ***Design (with) Patterns***
    - Generic solutions
    - Reusable solutions
  - ***Design for Change***
    - Extreme programming and refactoring
    - Best programming practices

Εισαγωγή – design by contract (1/5)



Εισαγωγή – design by contract (2/5)

- Ορίστηκε το 1986 από τον Bertrand Meyer, είναι θεμελιώδες στοιχείο της οντοκεντρικής γλώσσας Eiffel, ενώ απαιτεί ειδικό τρόπο προγραμματισμού σε άλλες γλώσσες
- Ορίζει τη λογική ορθότητας λειτουργίας των μεθόδων μίας κλάσης με τρόπο που ενσωματώνεται στην ίδια την κλάση
- Βασίζεται στην έννοια του assertion:
  - **Μία boolean έκφραση που πρέπει πάντα να είναι true και ουσιαστικά δηλώνει κανόνες ορθότητας του προγράμματος**
  - Εάν ένα assertion είναι false, «κάτι»στην κατάσταση προγράμματος, το οποίο σημασιολογικά ελέγχεται από το assertion, είναι λαθεμένο
  - Ελέγχονται μόνο σε «debug mode», και όχι σε κανονική εκτέλεση
  - Η λειτουργική σημασιολογία του προγράμματος δεν πρέπει ποτέ να βασίζεται στο γεγονός ότι μπορεί να υπάρχουν assertions εμβόλιμα στον κώδικα (δηλ. ο έλεγχος των assertions είναι «ορθογώνιος» με τον πραγματικό κώδικά του προγράμματος)

Εισαγωγή – design by contract (3/5)

- *Assertions στη C++ (1/1)*

```
#include <assert.h>
class Shape {
  private:
    static ShapeList shapeInstances;
    float x, y;
  public:
    bool  Verify (void* ptr) { return shapeInstances.In((Shape*) ptr); }
    virtual void Move (float dx, float dy) {
       assert(Verify(this));
       x += dx, y += dy;
    }
    Shape (...) { shapeInstances.Add(this);   ... }
    ~Shape () { shapeInstances.Remove(this);   ... }
};
class Triangle : public Shape {
  public:
  void Rotate (float angle) { assert(Verify(this));   . }
};
```

επικύρωση νομιμότητας κλήσης πριν την εκτέλεση του εκάστοτε αλγορίθμου

Εισαγωγή – design by contract (4/5)

- *Βασικά στοιχεία του design by contract*
  - Βασίζεται σε τρία είδη assertions, με κάθε assertion να ορίζεται είτε ως μία boolean έκφραση ή, εάν είναι πολύπλοκη, μέσω μία συνάρτησης που επιστρέφει bool:
    - *Preconditions*
      - **συνθήκες που καθορίζουν κατά την εκτέλεση τη νομιμότητα κλήσης μέλους - ορίζονται για κάθε συνάρτηση μέλος**
    - *Postconditions*
      - **συνθήκες που που καθορίζουν την ορθή περάτωση κλήσης μέλους - ορίζονται για κάθε συνάρτηση μέλος**
    - *Invariants*
      - **η αλλιώς τα αξιώματα της κλάσης, που είναι συνθήκες οι οποίες ορίζουν την ορθότητα και νομιμότητα των στιγμιότυπων μίας κλάσης**

Εισαγωγή – design by contract (5/5)

- Το παράδειγμα πάνω στο οποίο θα μελετήσουμε το design by contract είναι η αφηρημένη κλάση stack, όπως ορίζεται μερικώς παρακάτω:

```
class Stack {
    FUNCTIONS:
            bool      empty (void);
            void      push (type);
            type      top (void);
            void      pop (void);
            int       total (void);
            bool      full (void);
            Stack (void);
            ~Stack();
};
```


Περιεχόμενα

Preconditions (1/2)

- Κάθε precondition ***P*** ορίζεται για μία η περισσότερες συναρτήσεις - μέλη $F_1,\ldots, F_n$
  - και αποτυπώνει με τη μορφή προγραμματιστικής boolean έκφρασης τους κανόνες που πρέπει να ισχύουν, ώστε μία νόμιμη κλήση F_j να μπορεί να πραγματοποιηθεί.
    - Δηλ. περιγράφει τις συνθήκες ορθής λειτουργίας των συναρτήσεων - μελών
- *Η ευθύνη* εξασφάλισης ότι κλήσεις πραγματοποιούνται εάν και μόνο εάν οι αντίστοιχες preconditions ικανοποιούνται, *βαρύνει τον κλητευτή* (δηλ. τον χρήστη της κλάσης).
  - *Οι χρήστες υπόσχονται ότι πριν καλέσουν συναρτήσεις – μέλη, θα ελέγχουν πάντοτε εάν οι αντίστοιχες preconditions επιστρέφουν true.*

Preconditions (2/2)

παράδειγμα

```
class Stack {
  PRECONDITIONS:
    pop:    not FUNCTIONS.empty()
    top:    not FUNCITONS.empty()
    push:   not FUNCTIONS.full()
};
...
if (stackInstance.PRECONDITIONS.pop())
   stackInstance.FUNCTIONS.pop();
else
   Κατάλληλες ενέργειες ανάλογα με την,
   κατά περίπτωση, λογική του προγράμματος !
```

Υλοποίηση των preconditions ως συναρτήσεις - μέλη

Έλεγχος preconditions από τους clients

Το πρόγραμμα είναι έτοιμο για παν ενδεχόμενο

Περιεχόμενα

- Εισαγωγή – design by contract
- Preconditions
- *Postconditions*
- Invariants
- Representation invariants
- Πολυμορφισμός και design by contract
- LSP
- Application invariants
- Πρακτική εφαρμογή

Postconditions (1/2)

- Κάθε postcondition ***P*** ορίζεται συνήθως για μία αντίστοιχη συνάρτηση - μέλος ***F***
  - και αποτυπώνει με τη μορφή προγραμματιστικής boolean έκφρασης τη συνθήκη που χαρακτηρίζει την ορθή πραγμάτωση της νόμιμης κλήσης της συνάρτησης ***F***.
    - Δηλ. περιγράφει τι ιδιότητες θα πρέπει να έχει το στιγμιότυπο μετά την εκτέλεση της αντίστοιχης συνάρτησης
- *Η ευθύνη* εξασφάλισης ότι για κάθε νόμιμη κλήση συνάρτησης, η αντίστοιχη postcondition θα ικανοποιείται πάντοτε, *βαρύνει τον κατασκευαστή της κλάσης*.
  - *Οι κλάσεις υπόσχονται ότι, εφόσον οι χρήστες καλούν συναρτήσεις – μέλη όταν οι αντίστοιχες preconditions είναι true, οι σχετικές postconditions επίσης θα είναι true μετά την κλήση.*

Postconditions (2/2)

παράδειγμα

```
class Stack {
  POSTCONDITIONS:
     pop():          total equals old_total-1 and
                     not FUNCTIONS.full();
     push(x):        total equals old_total+1 and
                     not FUNCTIONS.empty() and
                     x equals FUNCTIONS.top();
  FUNCTIONS:
     push(x) {
           old_total = total;
        Λογική – αλγόριθμος push του στοιχείου x.
        assert(POSTCONDITIONS.push(x));   Έλεγχος postcondition από την κλάση
     }
};
```


Περιεχόμενα

- Εισαγωγή – design by contract
- Preconditions
- Postconditions
- *Invariants*
- Representation invariants
- Πολυμορφισμός και design by contract
- LSP
- Application invariants
- Πρακτική εφαρμογή

Invariants (1/3)

- Αποτυπώνουν με τη μορφή προγραμματιστικών boolean εκφράσεων τις συνθήκες που χαρακτηρίζουν την ορθότητα των στιγμιότυπων μίας κλάσης.
  - *Θα πρέπει να ικανοποιούνται πριν και μετά* την κλήση public συναρτήσεων – μελών, αλλά μπορεί να παραβιάζονται:
    - είτε προσωρινά, μέσα στην υλοποίηση των μελών, πριν περατωθεί συνολικά η λειτουργία που επιτελούν, ή
    - από τοπικές private βοηθητικές συναρτήσεις- μέλη, οι οποίες χρησιμοποιούνται μόνο στην κατασκευή των public συναρτήσεων.
  - Τα invariants, που ουσιαστικά είναι *εργαλείο επαλήθευσης στιγμιότυπων*, προστίθενται φυσιολογικά, μέσω του λογικής τελεστή σύζευξης - and, σε κάθε precondition και postcondition.
  - Συνηθίζεται να συνθέτουμε όλες τις συνθήκες invariants σε μία, μέσω λογικής σύζευξης των επιμέρους συνθηκών.

Invariants (2/3)

- Μία συνθήκη ***I*** συνιστά αξίωμα (invariant) της κλάσης ***C***, εάν και μόνο εάν τα παρακάτω είναι αληθή:
  1. Ο constructor της κλάσης ***C***, όταν καλείται με πραγματικές παραμέτρους που ικανοποιούν το αντίστοιχο precondition, παράγει ένα στιγμιότυπο το οποίο ικανοποιεί την συνθήκη ***I***.
  2. Για κάθε public συνάρτηση ***F*** της κλάσης ***C***, όταν αυτή καλείται μέσω ενός στιγμιότυπου ***O*** το οποίο ικανοποιεί ταυτόχρονα την συνθήκη ***I*** καθώς και το precondition της ***F***, τότε το ***O*** συνεχίζει ικανοποιεί την συνθήκη ***I*** και μετά την κλήση.

Invariants (3/3)

παράδειγμα

```
class Stack {
   INVARIANTS:
      axiom():        0 ≤ total ≤ MAX_ELEMS and
                      total equals 0 or not FUNCTIONS.empty() and
                      total equals MAX_ELEMS or not FUNCTIONS.full();
   FUNCTIONS:
      push(x) {
            Όπως πριν...
            assert(INVARIANTS.axiom());
      }
      pop() {
            Όπως πριν...
            assert(INVARIANTS.axiom());
      }
};
```

Έλεγχος invariant από την κλάση σε κάθε μέλος

Περιεχόμενα

- Εισαγωγή – design by contract
- Pre-conditions
- Post-conditions
- Invariants
- *Representation invariants*
- Πολυμορφισμός και design by contract
- LSP
- Application invariants
- Πρακτική εφαρμογή

CSD

Representation invariants (1/3)

- *Representation* – αναπαράσταση, δηλ. η εξειδικευμένη υλοποίηση ενός ADT (Abstract Data Type)
  - Μία κλάση η οποία υλοποιεί έναν αφηρημένο τύπο ονομάζεται αναπαράσταση (representation, REP) του τύπου αυτού. Π.χ., για την στοίβα ADT μπορούμε να έχουμε τα δύο παραδείγματα REP:
    - *Δυναμική δομή δεδομένων με δείκτες*
    - *Υλοποίηση μέσω πίνακα σταθερού μεγέθους*
- Ένα REP υλοποιεί όλα τα μέλη ενός ADT, αλλά μπορεί να εισάγει και νέα
  - Θεωρητικά, η σχέση ενός ADT και ενός αντίστοιχου REP είναι μία συνάρτηση αφαίρεσης *ADT = αφαίρεση(REP)* και μία συνάρτηση αναπαράστασης *REP = αναπαράσταση(ADT)*.
    - Για συγκεκριμένους τύπους T, είναι *REP(T) isa ADT(T)*

CSD

Representation invariants (2/3)

- Μερικά assertions μέσα στα invariants ενός REP δεν έχουν καμία αντίστοιχη υπόσταση στον ορισμό του ADT
  - Αυτού του είδους τα assertions εμπλέκουν εσωτερικές μεταβλητές ή συναρτήσεις οι οποίες, εξ ορισμού, δεν θα είχαν κανένα νόημα στο χώρο του ADT.
  - Αυτού του είδους τα εσωτερικά εμπλεκόμενα μέλη υφίστανται μόνο στα πλαίσια ενός συγκεκριμένου REP του ADT (δηλ. σε μία πραγματική υλοποίηση).
- Αυτού του είδους τα invariants εκφράζουν τη συνέπεια και ορθότητα του REP που έχει επιλεγεί για το ADT
  - για το λόγο αυτό ονομάζονται *representation invariants* (αξιώματα αναπαράστασης).

CSD

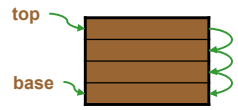
Representation invariants (3/3)

Παράδειγμα

```
class REP(Array-Stack) isa  ADT(Stack) {
   REP INVARIANTS :
   axiom(): 0 ≤ total ≤ ARRAY_SIZE
};

class REP(Dynamic-Stack) isa  ADT(Stack) {
   REP INVARIANTS :
   pointer down (pointer p, unsigned n) {
            return n ≤ 1 ? p : down(p->below, n-1);
   }
   axiom():  not (total equals 0) or (base equals null and top equals null) and
             not (total equals 1) or (base equals top and top ≠ null) and
             not (total > 1)  or (base ≠ top) and
             down(top, total) equals base
};
```

Συνθήκες σωστής αναπαράστασης του αφηρημένου τύπου στα πλαίσια της συγκεκριμένης υλοποίησης



CSD

Περιεχόμενα

- Εισαγωγή – design by contract
- Preconditions
- Postconditions
- Invariants
- Representation invariants
- *Πολυμορφισμός και design by contract*
- LSP
- Application invariants
- Πρακτική εφαρμογή

CSD

Πολυμορφισμός και design by contract (1/3)

- *Preconditions*
  - Μία κληρονόμος κλάση μπορεί μόνο να «αποδυναμώνει» τα preconditions για τα virtual methods της base κλάσης που εξειδικεύει
    - Δηλ. ∀ virtual F στην base B, με precondition BP, όπου F εξειδικεύεται στην derived D, με precondition DP, θα πρέπει να ισχύει: *BP(F) είναι true ⇒ DP(F) είναι true*
    - Ο λόγος είναι ότι θα πρέπει να μπορεί κάποιος να γράφει polymorphic functions σε base objects αρκεί να ικανοποιούνται οι συνθήκες του base class
      - Συνεπώς, δεν μπορεί το derived class να έχει πιο περιοριστικό precondition, αλλά πιο «χαλαρό»
      - *DP(F) = BP(F) or "Επιπλέον στο DP precondition(F)"*

CSD

Πολυμορφισμός και design by contract (2/3)

- *Postconditions*
  - Μία derived class μπορεί μόνο να «δυναμώνει» τα postconditions για τις virtual base methods που εξειδικεύει.
    - Δηλ. ∀ virtual F στην κληροδότη B, με postcondition BP, όπου F εξειδικεύεται στην κληρονόμο D, με postcondition DP, θα πρέπει να ισχύει: *DP(F) είναι true ⇒ BP(F) είναι true*
    - Η εξειδικευμένη συνάρτηση θα πρέπει να εξασφαλίζει ότι επιτελεί τουλάχιστον ότι επιτελεί και η αυθεντική συνάρτηση.
      - Συνεπώς, πρέπει να έχει πιο περιοριστικό postcondition
      - *DP(F) = BP(F) and "Επιπλέον στο DP postcondition(F)"*

CSD

Πολυμορφισμός και design by contract (3/3)

- *Invariant*
  - Προφανώς το invariant της κληροδότη πρέπει να εξασφαλίζεται και από τις κληρονόμους κλάσεις
  - Επομένως οι κληρονόμοι θα πρέπει να «δυναμώνουν» το invariant της κληροδότη
    - *Invariant(κληρονόμου) = Invariant(κληροδότη) and Representation invariant(κληρονόμου)*

CSD

Περιεχόμενα

LSP (1/3)

- *Liskov Substitutability Principle (1988)*
  - *Συναρτήσεις οι οποίες χρησιμοποιούν δείκτες ή αναφορές σε κληροδότες κλάσεις θα πρέπει να μπορούν να χρησιμοποιούν στιγμιότυπα κληρονόμων κλάσεων χωρίς να το γνωρίζουν*
    - + Ουσιαστικά οι συναρτήσεις αυτές συνιστούν κώδικα αμετάβλητο και επαναχρησιμοποιήσιμο στο απεριόριστο σύνολο των κληρονόμων κλάσεων
    - – Αντίθετα, εάν παραβιάσουμε τον κανόνα αυτό, θα υπάρχουν συναρτήσεις οι οποίες θα πρέπει να γνωρίζουν τις κληρονόμους κλάσεις με κάποιο τρόπο, ή να ισχύουν μόνο για συγκεκριμένες από αυτές

LSP (2/3)

Παράδειγμα (1/2)

```
class Shape {
   public:
   virtual void Display (const Surface& at) const = 0;
};

Υποστήριξη του LSP
void Displayer (const Surface& at, const Shape** shapes, unsigned n) {
   while (n--)
    (*shapes++)->Display(at);
}

Παραβίαση του LSP
void Evaluate (Expression* expr, ExrValue* result) {
   if (expr->GetType() == ExprTypeConstInt) {
      result->type = ValueTypeInt;
      result->content.intVal = ((ConstIntExpression*) expr)->GetValue();
   }
   else
      … Ομοίως για κάθε διαφορετικό κληρονόμο τύπο του Expression
}
```


LSP (3/3)

Παράδειγμα (2/2)

```
Υποστήριξη του LSP
class Expression {
   virtual const ExprValue Evaluate (void) = 0;
};

class ConstIntExpr : public Expression {
   int val;
   public:
   const ExprValue Evaluate (void) { return ExprValue(val); }
};

class AddExpr : public Expression {
   Experssion* left;
   Expression* right
   public:
   const ExprValue Evaluate (void)
          { return left->Evaluate() + right->Evaluate(); }
};
```


Περιεχόμενα

CSD

Application invariants (1/8)

- Μία ειδική τεχνική για ορισμό αξιωμάτων που έχουν νόημα μόνο στα πλαίσια ειδικών προγραμμάτων ελέγχου, είναι μια εξειδικευμένη μέθοδος αμυντικού προγραμματισμού
- Η τεχνική αυτή ενώ δεν είναι κομμάτι του αυθεντικού design by contract, αλλά είναι συμπληρωματική ευρεστική μέθοδος εντοπισμού λαθών

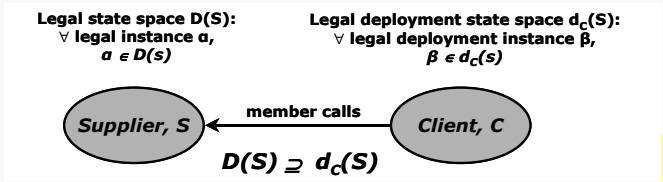
➔ *Ως APP invariant ορίζουμε ένα REP invariant το οποίο είναι ορθό μόνο στα πλαίσια συγκεκριμένης εφαρμογής και σεναρίου χρήσης της κλάσης*

Anthony Savidis: **Application invariants: Design by Contract augmented with deployment correctness logic.** Softw., Pract. Exper. 36(3): 255-282 (2006)

CSD

Application invariants (2/8)

- Η τεχνική αποτυπώνει την ορθότητα λειτουργίας βασισμένη στη γνώση του προγραμματιστή για τις τιμές που μπορεί να λαμβάνουν τα τοπικά δεδομένα μίας κλάσης, στα πλαίσια των εκάστοτε προγραμμάτων ελέγχου
- Μπορεί να εντοπίζει πολύ γρήγορα λάθη πρόσβασης μνήμης *ειδικά εάν στα ελεγκτικά προγράμματα σχεδιάζονται σενάρια που δίνουν τέτοιες τιμές στα ορθά δεδομένα μίας κλάσης, οι οποίες είναι απίθανο να παραχθούν μέσω λάθος χρήσης*



Βασίζεται στην εξής παρατήρηση

CSD

Application invariants (3/8)

```
class REP(Dynamic-Stack) isa  ADT(Stack) {
   APP INVARIANTS :
   ValueList values;          ← Λίστα από τιμές που κάνουμε push στο test
   values = [ 1, 2, 3, 5, 7, 11, 13, 17, 19 ]; ← Επιλέξαμε πρώτους αριθμούς
   bool Check (Pointer p) {   ← Έλεγχος στοίβας από το p και «κάτω»
      if (p equals null)
           return true;        ← Θεωρούμε ότι ο null pointer περνάει το test
      else
      if (not values.Inside(p->Value))
          return false;
      else
         return Check(p->down);
   }
   axiom(): Check(top);
};
```


CSD

Application invariants (4/8)

- Επιπλέον, πέραν των περιορισμών τοπικού χαρακτήρα, τα application invariants οριοθετούν και τον εξής κανόνα:
  - Κάθε instance μίας κλάσης θα πρέπει να υφίσταται μόνο εφόσον: (α) έχει δημιουργηθεί από την εφαρμογή στο πλαίσιο ορθής λειτουργίας, (β) η ύπαρξη του συνεπάγεται από την κατάσταση της εφαρμογής
  - Με λίγα λόγια, τα application invariants ορίζουν και τον λόγο ύπαρξης των στιγμιότυπων ως κριτήριο ορθότητας
- Το παραπάνω κριτήριο δεν καλύπτεται από κανένα άλλο που ήδη γνωρίζουμε στο Design by Contract
  - Π.χ., σύμφωνα με το παραπάνω κριτήριο, το αυθαίρετο copy construction ενός correct instance δεν είναι ποτέ correct

CSD

Application invariants (5/8)

- Τέλος διαχωρίζεται το precondition μίας μεθόδου σε δύο θεμελιώδη σκέλη:
- ***readiness precondition*** – συνθήκη ετοιμότητας που καθορίζει εάν η κατάσταση του αντικειμένου επιτρέπει την κλήση της μεθόδου
  - Όλα τα preconditions που έχουμε μελετήσει έως τώρα είναι αυτού του είδους
  - Συνεπώς το Design by Contract προβλέπει μόνο έλεγχο ετοιμότητας
- ***appropriateness precondition*** - συνθήκη καταλληλότητας που καθορίζει πότε οι πραγματικές παράμετροι κατά την κλήση μίας μεθόδου είναι ορθές
  - Το αυθεντικό Design by Contract δεν προβλέπει μόνο έλεγχο καταλληλότητας αυτού του είδους

CSD

Application invariants (6/8)

```
class Document {
public:
    unsigned        GetSize (void) const;
    virtual bool    CopyPreReady (void) const;
    virtual bool    CopyPreAppr (unsigned startPos, unsigned endPos) const
                        { return startPos < endPos && endPos < GetSize(); }
    virtual void    Copy (unsigned startPos, unsigned endPos) const;
};

Document* doc;
...
doc->Copy(0,100);   // incorrect way
...
if (doc->CopyPreReady() && doc->CopyPreAppr(0,100)) // correct way
    doc->Copy(0,100);
...
if (doc->CopyPreReady() && doc->CopyPreAppr(ComputeStart(),ComputeEnd()))
    doc->Copy(ComputeStart(),ComputeEnd()); // repetition of calls!
```

Η επανάληψη των παραμέτρων είναι αφενός κουραστική, αφετέρου μπορεί να είναι προβληματική όταν εμπλέκει πιο πολύπλοκα expressions με άλλα method invocation ή ακόμη και δημιουργία temporary stack objects

CSD

Application invariants (7/8)

```
class Supplier {
private:
   class f_Args {
       // Supplier needs access to actual arguments.
       friend class Supplier;
   private:
       T1 x1; ... Tn xn;  // Argument value copies.
       Supplier& x;       // Supplier reference.
       bool used;
   public:
       bool operator() const {
            return !used &&
                   (x1,...,xn) is a well formed x.f call;
       }
       f_Args(T1 y1,...,Tn yn, Supplier& _x) : x(_x), used(false)
            { for each j in(0, n]: xj = yj; }
   };

   // Argument class needs access to Supplier.
   friend class f_Args;
   f_Args f_args;

public:
   bool PREREADY_f (void);          // readiness precondition
   bool PREAPPR_f (T1 y1,...,Tn yn){ // appropriateness precondition
        new (&f_args) f_Args(y1,...,yn, *this);
        return f_args();
   }
   void f (void) { // Originally it was f(x1,...,xn)
        assert(f_args());
        f_args.used = true; // Arguments were consumed
        f logic here for arguments: f_args.x1, ... ,f_args.xn
   }
};

Supplier x;
if (x.PREREADY_f() && x.PREAPPR_f(y1,...,yn))
    x.f();
```


CSD

Application invariants (8/8)

Client precondition checking	*Supplier precondition checking*

`X::f() {`
`}`
	`X::f() {`
	`  if (!PRE_f())`
	`  throw` ***Precondition violation***
	`}`

`if (x.PRE_f())`
`  x.f();`
`else`
Logic for failed precondition
	`try { f(); }`
	`catch (`***Precondition violation***`) {`
	Logic for failed precondition
	`}`

Function calls are committed only if their preconditions are satisfied. Otherwise, precondition violation exceptions are raised, which, unless handled, will cause instant program interruption.

While Design by Contract is a genuine defensive programming method, it is not a testing method. Typically, programs written with embedded bug defence always require an appropriate method to perform exhaustive testing of program units. In this context, the two key approaches towards software robustness are revealed: (a) *embedded self-verification* of program units for conformance of their implementation to the corresponding design semantics, i.e. Design by Contract; and (b) *diagnostic functionality tracing* of program units, asserting test requests with expected correct responses, i.e. unit testing.

Περιεχόμενα

- Εισαγωγή – design by contract
- Pre-conditions
- Post-conditions
- Invariants
- Representation invariants
- Πολυμορφισμός και design by contract
- LSP
- Application invariants
- *Πρακτική εφαρμογή*

Πρακτική εφαρμογή (1/5)

Παράδειγμα

```
class Stack {
  protected:
    unsigned int      total;
    unsigned int      oldTotal;
    Thing             pushedArg;
  public:
    virtual bool   IsFull (void) const = 0;
    virtual bool   IsEmpty (void) const = 0;
    virtual const  Thing& Top (void) const = 0;
    virtual bool   Push (const Thing&)=0;
  protected:
    virtual bool   PreconditionPush (void) const
                       { return !IsFull();  }
    virtual bool   PostconditionPush (void) const {
                       return   !IsEmpty() &&
                                total == oldTotal + 1 &&
                                Top() == pushedArg;
                   }
};
```

Τοπικές μεταβλητές ειδικά για την υποστήριξη των assertions

Πρακτική εφαρμογή (2/5)

παράδειγμα

```
class StackBYARRAY : public Stack {
  private:
    Thing  array[MAX_ARRAY_SIZE];
  public:
    bool   IsFull (void) const
              { return total== MAX_ARRAY_SIZE; }
    const  Thing& Top (void) const
              { return array[total-1]; }
    bool   Push (const Thing& item) {
               array[oldTotal = total++] = pushedArg = item;
               return PostconditionPush();
           }
};
StackBYARRAY myStack;
if (myStack.PreconditionPush()) {
    if (!myStack.Push(...))
          Κάποιο bug υπάρχει, αφού με true precondition έχω false postcondition !
}
else
    Λογική του προγράμματος σε περίπτωση που δεν είναι εφικτό το push()
```


Πρακτική εφαρμογή (3/5)

παράδειγμα

```
class StackDYNAMIC : public Stack {
  private:
    struct Element {
           Thing      value;
           Element*   below;
    };
    Element* top;
    Element* base;
    Element Down (Element* p, unsigned int n)
             { return n <=1 ? p : Down(p->below, n-1);
    bool Invariant (void) {
          return  Stack::Invariant()        &&
                  total || (!base && !top)  &&
                  total != 1 || base==top   &&
                  total <= 1 || base != top &&
                  Down(top, total)== base;
         }
  public:
    const Thing& Top (void) { return top->value; }
    bool          PreconditionTop (void) { return Invariant() && !!top;  }
};
```

REP invariant της StackDYNAMIC

Πρακτική εφαρμογή (4/5)

- *Θετικά χαρακτηριστικά*
  - Γενικά, ένας πολύ καλός τρόπος προγραμματισμού με επικύρωση των σημασιολογικών πλευρών του προγράμματος μέσα από τον ίδιο τον κώδικα - *self checking code*, εφαρμόζονται πρακτικές προτεραιότητας ελέγχου - *test-first programming*.
  - Βοηθά στον πρώιμο εντοπισμό λαθών, εφόσον ορίσουμε και προγραμματίσουμε όλες τις συνθήκες ούτως ώστε το οποιοδήποτε λάθος να σηματοδοτεί πάντα ένα false assertion.
  - Οδηγεί σε πιο αναγνώσιμο και αυτό-τεκμηριωμένο πηγαίο κώδικα (διότι περιέχει και τη σημασιολογία του).

Πρακτική εφαρμογή (5/5)

- *...αλλά*
  - Η απαίτηση να ευθύνεται αποκλειστικά για την κλήση των preconditions ο caller, μπορεί να αποδειχθεί ως μη βέλτιστη απόφαση:
    - Ένα bug πρέπει να εντοπίζεται και να αναφέρεται όσο το δυνατό πλησιέστερα του σημείου και χρόνου γεννήσεως του.
    - Βάσει ισχυρισμού στην αυθεντική τεχνική, επειδή οι κλάσεις δεν μπορούν να γνωρίζουν πως θα παρουσιάσουν μηνύματα στον χρήση, μόνο ο caller μπορεί να έχει αυτή την ευθύνη,
    - ...όμως οι κλάσεις μπορεί να βασίζονται σε άλλες αφηρημένες κλάσεις οι οποίες και θα είναι υπεύθυνες για παρουσίαση λαθών.
    - Εάν ο caller λησμονήσει την κλήση κάποιου precondition, τότε το bug μπορεί να «ταξιδέψει» αρκετά μακριά προκαλώντας συρροή από άλλα λάθη!
    - ➔ *Όλα τα assertions σηματοδοτούν μη νόμιμη κατάσταση στιγμιότυπου και πρέπει να σταματούν αυτόματα την εκτέλεση !*