

HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

**ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ**



ΔΙΔΑΣΚΩΝ

Αντώνιος Σαββίδης

ΕΝΟΤΗΤΑ 4

ΣΤΟΙΧΕΙΑ ΟΝΤΟΚΕΝΤΡΙΚΟΥ ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΥ

Αριθμός διαλέξεων 7, Διάλεξη 5η



Περιεχόμενα

- *Κατηγοριοποίηση μεθόδων*
- Κατηγορίες constructor
- Const-ness στιγμιότυπου σε κλήσεις μεθόδων
- Κατηγορίες μεθόδων

Κατηγοριοποίηση μεθόδων

- *Γιατί κατηγοριοποιούμε τις συναρτήσεις – μέλη*
 - ➔ Εντοπισμός κοινών λειτουργικών ρόλων, με ενδεχόμενες κοινές δομές υλοποίησης
 - ➔ Τυποποίηση ονομασίας, υπογραφής, τεκμηρίωσης, και χρήσης, σε πεπερασμένο σύνολο προτύπων
 - ➔ Βοηθά στην απομνημόνευση και στον καλύτερο έλεγχο της υλοποίησης μεγάλων ή πολλών κλάσεων, διότι αναγνωρίζεται ο ρόλος της κάθε συνάρτησης μόνο από την δήλωσή της
 - ➔ Υποστηρίζει την καλύτερη επιθεώρηση του κώδικα και τον ευκολότερο εντοπισμό λογικών σφαλμάτων:
 - ➔ κάθε συνάρτηση ελέγχεται ως προς τη δήλωση, υλοποίηση και χρήση που αρμόζει ανάλογα με την κατηγορία στην οποία ανήκει

Περιεχόμενα

- Κατηγοριοποίηση μεθόδων
- *Κατηγορίες constructor*
- Const-ness στιγμιότυπου σε κλήσεις μεθόδων
- Κατηγορίες μεθόδων

Κατηγορίες constructor (1/13)

- *Empty (default)*
 - Κενός
- *Parameterized*
 - Παραμετροποιημένος
- *Copier (copy)*
 - Αντιγραφέας
- *Move*
 - Μεταφορέας
- *Converter*
 - Μετατροπέας
- *Decoder*
 - Αποκωδικοποιητής

Κατηγορίες constructor (2/13)

■ *Empty (default) constructor*

- Δεν έχει καμία παράμετρο με υπογραφή `<class>(void)`
- Είναι μόνο ένας (μοναδικός)
- Ο σκοπός του είναι η αρχικοποίηση του στιγμιότυπου (των τοπικών μεταβλητών και των base objects) σε μία συγκεκριμένη αρχική κατάσταση
- αυτή η κατάσταση μπορεί να αντιπροσωπεύει
 - ◆ ένα «παράνομο» στιγμιότυπο (η χρήση δεν επιτρέπεται)
 - ◆ ένα ατελές στιγμιότυπο (χρήση με προϋποθέσεις)
 - ◆ ένα πλήρες και έτοιμο προς χρήση στιγμιότυπο

Κατηγορίες constructor (3/13)

■ *Parameterized constructor*

- Έχει υπογραφή `<class> (T1, ..., Tn)`
- Μπορεί να έχουμε πολλούς τέτοιους (overloaded) constructors
- Οι εναλλακτικές εκδοχές των τυπικών παραμέτρων αντιστοιχούν στους διαφορετικούς τρόπους αρχικοποίησης στιγμιότυπων της κλάσης
- Συνήθως, όταν ένας παραμετροποιημένος constructor καλείται με τα σωστά ορίσματα, ένα έτοιμο και απολύτως λειτουργικό στιγμιότυπο δημιουργείται
- Τις περισσότερες φορές αρχικοποιούμε από τα ορίσματα τις εσωτερικές τοπικές και τα base objects.

Κατηγορίες constructor (4/13)

■ *Copier / copy constructor (1/2)*

- Έχει υπογραφή `<class>(const <class>&);`
- Είναι μόνο ένας (μοναδικός)
- Λαμβάνει ως όρισμα ένα στιγμιότυπο της ίδιας κλάσης από το οποίο και συνήθως αντιγράφει, η απλά αρχικοποιεί, την εσωτερική του κατάσταση.
- Κατά την αντιγραφή της εσωτερικής κατάστασης χρειάζεται προσοχή ώστε να εξασφαλιστεί πως τα δυναμικά δεδομένα δεν εκχωρούνται απλώς (shallow copy), αλλά αντιγράφονται (deep copy)
- Σε μερικές φορές ενδέχεται να είναι σχεδιαστική επιλογή η απλή εκχώρηση ακόμη και για δυναμικά δεδομένα.

Κατηγορίες constructor (5/13)

Copier / copy constructor (2/2)

```
class X {  
    private:  
        int a;  
    public:  
        X (void)          : a(0) {}      ← empty constructor  
        X (int _a)        : a(_a) {}     ← parameterized constructor  
        X (const X& x)    : a(x.a){}     ← copy constructor  
};  
class Y {  
    private:  
        int* arr;  
    public:  
        Y (const Y& y) : arr(y.arr){} ← Assignment instead of copy  
        ~Y() { delete[] arr; }        ← Crash most-likely here  
};
```

Κατηγορίες constructor (6/13)

■ *Move constructor* (1/2)

- Έχει υπογραφή `<class>(<class>&&);`
- Είναι μόνο ένας (μοναδικός)
- Λαμβάνει ως όρισμα ένα *xvalue* της ίδιας κλάσης
 - ◆ Expiring value, νέα κατηγορία εκτός των *rvalue* και *lvalue*
 - ◆ Όλα τα returned temporary objects
 - ◆ Το αποτέλεσμα της `std::move(val)` για ένα *value*
 - ◆ Το αποτέλεσμα της `static_cast<T&&>(val)` ή `(T&&) val` για ένα *value*
- Συνήθως εκχωρεί δυναμικά δεδομένα (ΟΧΙ copy) από το *xvalue* ενώ το κάνει ταυτόχρονα null

Κατηγορίες constructor (7/13)

Move constructor (2/2)

```
struct X {  
    int* arr; int n;  
    X (int _n = 0) { arr = new int[n = _n]; } // note: in C++ new T[0] is safe  
    X (const X& x) { arr = new int[n = x.n]; }  
    X (X&& x) { arr = x.arr; n = x.n; x.arr = nullptr; x.n = 0; }  
    ~X() { delete []arr; } // note: in C++ delete null is safe  
};  
const X f (void) { return X(10); }  
X x1(1);  
X x2(x1); // calls X(X&)  
X x3((X&&) x1); // calls X(X&&)  
X x4(f()); // calls X(X&&)  
X x5(std::move(x4)); // calls X(&&)
```

Κατηγορίες constructor (8/13)

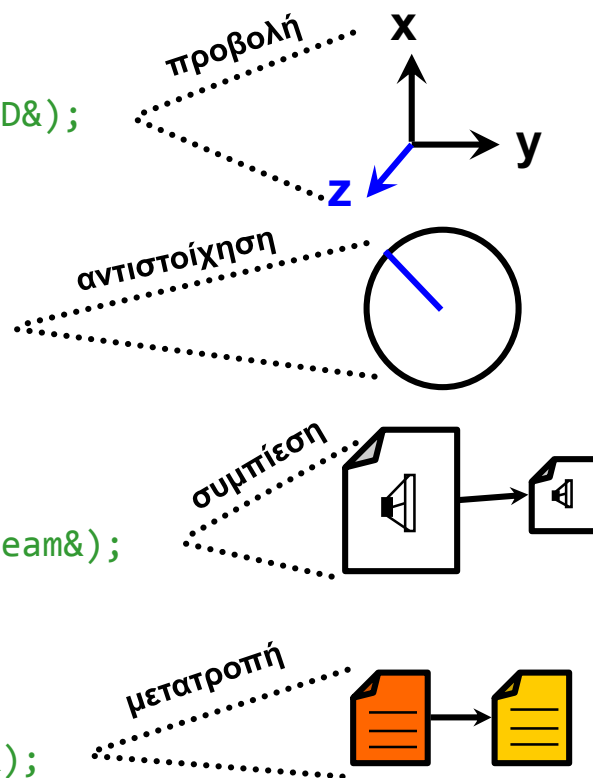
■ *Converter constructor (1/2)*

- Η υπογραφή του είναι `X (const Y&);` για κλάσεις **X** και **Y** που συνήθως δε σχετίζονται με inheritance
- Μπορεί να έχουμε πολλούς τέτοιους constructors για διαφορετικές κλάσεις
- Για την κλάση **X** και **Y** θα πρέπει να ορίζεται αλγοριθμικά η μετατροπή του **Y** σε **X**
 - ◆ μόνο με χρήση public members του **Y**, εκτός και εάν είναι **X** friend of **Y**
- δεν είναι σχεδιαστικά ορθό να θέσουμε με όποιο τρόπο την **Y** εξαρτημένη από την **X**

Κατηγορίες constructor (9/13)

■ Converter constructor (2/2)

```
class Vector2D {  
    public:  
        Vector2D (const Vector3D&);  
};  
class Circle {  
    public:  
        Circle (const Line&);  
};  
class MP3Stream {  
    public:  
        MP3Stream (const WAVStream&);  
};  
class PDFFile {  
    public:  
        PDFFile (const DOCFile&);  
};
```



Κατηγορίες constructor (10/13)

■ *Decoder (deserializer) constructor (1/4)*

- Έχει παραμέτρους που ορίζουν *μία «πακεταρισμένη» κατάσταση στιγμιότυπου*, από την οποία και θα γίνει πλήρης **αναδόμηση** του νέου στιγμιότυπου **X**
- Μπορεί να έχουμε πολλούς τέτοιους constructors
- Συνήθως περιέχει μία μοναδική παράμετρο που ορίζει την πηγή των δεδομένων με τον τύπο να ποικίλει
- Εναλλακτικές μέθοδοι πακεταρίσματος μπορούν να υποστηρίζονται
 - ◆ συμπίεση, αποθήκευση των μεταβλητών με τη σειρά, αποθήκευση σε δυαδική μορφή, κωδικοποίηση σε μορφή κειμένου, κρυπτογράφηση, κλπ

Κατηγορίες constructor (11/13)

■ *Decoder constructor (2/4)*

```
class Agent { STYLE-1, loading once, failure check
private:
    static Agent* root;           // Hierarchy top (root)
    std::string id;               // Unique across a hierarchy
    Agent* parent;               // Parent, null if 'this' is root

public:
    const char* Id (void) const
        { return id.c_str(); }
    bool IsValid (void) const;      // If a valid instance
    static Agent* Get (const char* id); // Depth first search
    static Agent* Root (void)
        { return root; }
    static bool LoadAll (const char* file);
    Agent (FILE* fp);             // Decoder constructor (bin loader)
};
```


Κατηγορίες constructor (12/13)

■ *Decoder constructor (3/4)*

```
bool Agent::LoadAll (const char* file) {  
    FILE* fp = fopen(file, "r");  
    if (!fp)  
        return false;  
    else {  
        delete root;           // Assume recursive destruction  
        while (!feof(fp))  
            new Agent(fp);     // Decoding from a file stream  
        fclose(fp);  
        return true;  
    }  
}
```

Κατηγορίες constructor (13/13)

■ *Decoder constructor (4/4)*

```
class Agent {      STYLE-2, Loading many times, failure check
private:
    bool isValid;
    void Clear (void) { if (isValid) {... isValid = false; } }
    Agent (void) : isValid(false), {...} // Default constructor, hidden.
public:
    bool Load (FILE* fp) { Clear(); ... return result; } // Decoder
    bool IsValid (void) const;
    Agent (FILE* fp) { Agent::Agent(); isValid = Load(fp); }
};
```

Όταν έχουμε στιγμιότυπα που μπορούν να κάνουν load πολλές φορές, τότε έχουμε ένα Load member function το οποίο κάνει πρώτα clear και έπειτα load. Επειδή μπορεί η load να αποτύχει, το στιγμιότυπο πρέπει να μπορεί να είναι και σε ένα *invalid testable state*.

Περιεχόμενα

- Κατηγοριοποίηση μεθόδων
- Κατηγορίες constructor
- *Const-ness σε κλήσεις μεθόδων*
- Κατηγορίες μεθόδων

Constness στιγμιότυπου (1/4)

- Μία μέθοδος με τον χαρακτηρισμό **const** δεν επιτρέπεται να τροποποιεί τις μεταβλητές δεδομένων της κλάσης της (συμπεριλαμβανομένων και αυτών που κληρονομούνται)
- Τέτοιου είδους μέθοδοι λειτουργούν με πρόσβαση ανάγνωσης - *read only*, και είναι οι μόνες συναρτήσεις που μπορούν να κληθούν μέσω αναφορών της κλάσης που έχουν τον χαρακτηρισμό **const**
- ➔ Ωστόσο, ο κανόνας μπορεί να παραβιάζεται ανάλογα με την ανάγκη:
 - ➔ μπορούμε να χαρακτηρίσουμε τοπικές μεταβλητές με τρόπο που να επιτρέπει ακόμη και σε **const** συναρτήσεις να τις τροποποιούν
 - ➔ η να αφαιρούμε τον **const** χαρακτηρισμό από τις αναφορές μέσω του **const_cast** και να καλούμε συναρτήσεις που δεν είναι **const**
 - ➔ Εδώ το standard λέει πως αυτό μπορεί να οδηγήσει σε undefined behavior
 - ➔ Δε θα είναι αυτό επικίνδυνο σε objects αλλά να μη το κάνετε ποτέ σε μεμονωμένες μεταβλητές

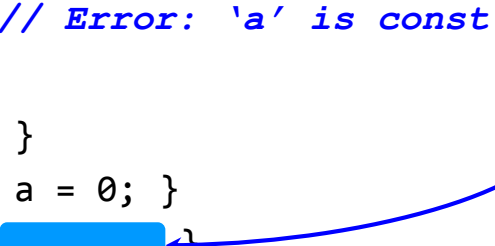
Constness σιγμιότυπου (2/4)

Παραδείγματα (1/2)

```
enum X_Call { f1_Call = 0, f2_Call = 1, f3_Call = 2 };

class X {
private:
    int a;
    mutable X_Call lastCall; // Allowed to change even in const methods

public:
                                // Error: 'a' is const
    int  Get_a (void) const { return a; }
    void f1(void) const    { lastCall = f1_Call; }
    void f2(void)          { lastCall = f2_Call; a = 0; }
    void f3(void) const    { lastCall = f3_Call; }
};
```



Constness στιγμιότυπου (3/4)

Παραδείγματα (2/2)

```
X x;  
const X& xRef = x;           // Ένα const reference, μπορεί να καλεί  
                              // μόνο const συναρτήσεις - μέλη.  
  
xRef.f1();                   // Ok, η f1() είναι const.  
xRef.f2();                   // Error, η f2() δεν είναι const.  
const_cast<X&>(xRef).f2();   // Εξουδετέρωση του χαρακτηρισμού const,  
                              // άρα νόμιμα καλείται η non-const f2.
```

Constness στιγμιότυπου (4/4)

■ Οδηγίες χρήσεις

- Χρησιμοποιήστε `const` συναρτήσεις όταν ο ρόλος τους είναι περισσότερο να «παρατηρούν» παρά να «τροποποιούν»
- Υιοθετήστε τον χαρακτηρισμό *mutable* με φειδώ και μόνο σε προσεκτικά σχεδιασμένες εξαιρέσεις
- Αποφύγετε την συχνή χρήση του τελεστή `const_cast` για την εξουδετέρωση του χαρακτηρισμού `const`
- Εάν είναι ο μόνος τρόπος, οπισθοχωρήστε ένα βήμα στη σχεδίαση και ελέγξτε γιατί καταλήξατε στο συγκεκριμένο σημείο του κώδικα να έχετε `const` αναφορά, όταν απαιτείται να καλέσετε non-`const` συναρτήσεις
- Συνήθως οι μέθοδοι που χρειάζεστε μπορούν να οριστούν ως `const` ειδάλλως λάθος ορίσατε ότι χρειάζεστε `const object`
- Εφαρμόστε τον ειδικό αυτό τελεστή εάν η μεταβολή του κώδικα
 - ◆ είναι αρκετά ακριβή (δε θα γίνει)
 - ◆ πρόκειται να γίνει αργότερα
 - ◆ δεν απλώς δυνατή (δε βρίσκετε τρόπο αλλαγής που να καταλήγει να σας δίνει `const object`)

Περιεχόμενα

- Κατηγοριοποίηση μεθόδων
- Κατηγορίες constructor
- Const-ness στιγμιότυπου σε κλήσεις μεθόδων
- *Κατηγορίες μεθόδων*

Κατηγοριοποίηση μεθόδων (1/14)

- *Producer*
- *Modifier*
- *Observer*
- *Attribute*
- *Replicator*
- *Comparator*
- *Serializer*
- *Verifier*
- *Processor*
- *Notifier*

➤ Αυτή είναι μία αντιπροσωπευτική ταξινόμηση των διαφόρων κατηγοριών στις οποίες μπορεί να ανήκουν οι συναρτήσεις – μέλη μίας κλάσης.

➤ Το σύνολο αυτό δεν πρέπει να θεωρείται ως πλήρες, ούτε οι ρόλοι των συναρτήσεων ως ξένοι μεταξύ τους.

➤ Συναρτήσεις με χαρακτηριστικά παραπάνω της μίας κατηγορίας είναι φυσιολογικά αναμενόμενες.

Κατηγοριοποίηση μεθόδων (2/14)

■ *Producer – παραγωγός στιγμιότυπων (1/3)*

- Member function που δημιουργεί νέα στιγμιότυπα της κλάσης μέσω των στιγμιότυπων κλήσης, τα οποία δεν είναι απαραίτητως αντίγραφα των τελευταίων. Επιστρέφει συνήθως το εκάστοτε δημιουργούμενο στιγμιότυπο.
- Η παραγωγός μπορεί να χρειαστεί να τροποποιήσει το στιγμιότυπο που πραγματοποιεί την κλήση, εάν η διαδικασία της δημιουργίας αλλάζει εσωτερικές μεταβλητές, ή ενδέχεται να μην το επηρεάζει διόλου, οπότε και πρέπει να χαρακτηρίζεται ως const

```
class TreeNode {  
    public:  
    TreeNode* ProduceLeftSibling (void);  
    TreeNode* ProduceRightSibling (void);  
    TreeNode* ProduceRightMostChild (void);  
    TreeNode* ProduceLeftMostChild (void);  
};
```

Κατηγοριοποίηση μεθόδων (3/14)

■ *Producer* – παραγωγός στιγμιότυπων (2/3)

- Ειδική περίπτωση producer αποτελεί static συνάρτηση μίας κλάσης που παράγει δυναμικά στιγμιότυπα με συμπεριφορά ανάλογη του constructor
- Τέτοιες συναρτήσεις λέγονται και *factories*, με πιο συνηθισμένη τον static constructor
- Στο παράδειγμα φαίνεται πώς κάθε normal constructor μπορεί να μετατραπεί σε factory constructor

```
class A {  
private:  
    A (void);  
    A (int a, int b);  
    A (FILE*);  
public:  
    static A* Construct (void)           { return new A; }  
    static A* Construct (int a, int b)   { return new A(a,b); }  
    static A* Construct (FILE* fp)       { return new A(fp); }  
};
```

Κατηγοριοποίηση μεθόδων (4/14)

■ *Producer – παραγωγός στιγμιότυπων (3/3)*

- Μπορεί με factories να δημιουργούνται στιγμιότυπα χωρίς να εκχωρούμε τη διεύθυνση τους κάπου
- Συνηθισμένο σε περιπτώσεις που η χρήση του στιγμιότυπου προκύπτει από εσωτερική διαχείρισή του με άλλο τρόπο
 - ◆ π.χ., window creation, sprite creation, process creation, κλπ
- Τότε μπορούμε να αυτοματοποιήσουμε την κατασκευή με name-based instantiation για παρόμοια constructor signatures

```
class Position {};  
  
class Soldier { public:  
    static void Construct (const Position&);  
    static const char* Id (void) { return "Soldier"; }  
};  
  
class Captain { public:  
    static void Construct (const Position&);  
    static const char* Id (void) { return "Captain"; }  
};  
  
class Tank { public:  
    static void Construct (const Position&);  
    static const char* Id (void) { return "Tank"; }  
};  
  
class Instantiator {  
    private:  
        typedef void (*CtorFunc) (const Position&);  
        std::map<std::string, CtorFunc> ctors;  
  
    public:  
        void Add (CtorFunc f, const char* c) { ctors[c] = f; }  
        void New (const char* c, const Position& p)  
            { (*ctors[c])(p); }  
};
```

Κατηγοριοποίηση μεθόδων (5/14)

■ *Modifier – τροποποιητής*

- Method που τροποποιεί την κατάσταση του στιγμιότυπου. Δεν αναμένεται ποτέ να χαρακτηριστεί ως `const`, μπορεί να μην έχει παραμέτρους, ενώ το όνομά της ίσως να μην είναι πάντα αντιπροσωπευτικό των εσωτερικών μεταβλητών που μπορεί να επηρεάζει.
- Εκτός από το να τροποποιεί δεδομένα, μπορεί να κάνει επιπλέον κλήσεις εσωτερικών συναρτήσεων, ώστε να εφαρμοστούν οι όποιες εσωτερικές μεταβολές.
- Πλην του `void`, ο επιστρεφόμενος τύπος ενός `modifier` μπορεί να φέρει πληροφορία για την έκβαση της μεταβολής ή να κάνει `throw` κάποιο `exception`

<pre>class Robot { private: Map currMap; Target currTarget; ... }</pre>	<pre>public: void UpdateMission (const Map& m, const Target& t) { StopMission(); currMap = m; currTarget = t; Reschedule(); StartMission(); } };</pre>
---------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Κατηγοριοποίηση μεθόδων (6/14)

■ *Observer (accessor) – παρατηρητής στιγμιότυπων*

- Επιστρέφει πληροφορία σχετικά με την κατάσταση και τα περιεχόμενα του στιγμιότυπου ενώ συνήθως δεν αλλάζει το state και είναι const
- Εάν κάτι αλλάζει εσωτερικά το οποίο επηρεάζει την *παρατηρούμενη συμπεριφορά και κατάσταση*, θα πρέπει οπωσδήποτε να τεκμηριώνεται

```
class DocumentEditor {  
    public:  
    unsigned    Totallines (void) const; ← μη χρησιμοποιείτε τέτοια ονόματα  
    unsigned    GetTotalLines (void) const; ← απλώς επιστρέφει αποθηκευμένη τιμή  
    unsigned    ComputeTotalLines (void) const; ← το υπολογίζει κάθε φορά  
    unsigned    ComputeFileSizeInBytes (void) const;  
    void        GetCursorPos (unsigned* line, unsigned* col) const;  
    bool        SearchText (  
        const std::string& text,  
        const SearchParms& parms,  
        SearchResults* results  
    ) const;  
};
```

Κατηγοριοποίηση μεθόδων (7/14)

■ *Attribute – χαρακτηριστικό γνώρισμα στιγμιότυπου*

- Είναι ειδική περίπτωση ενός ζευγαριού modifier και observer, όταν και οι δύο σχετίζονται με τα ίδια εσωτερικά δεδομένα, κατάσταση ή παραμέτρους λειτουργίας της κλάσης
- Είθισται ο observer να έχει το πρόθεμα **Get**, και ο modifier το πρόθεμα **Set**, με επίθεμα ένα αναγνωριστικό όνομα αντιπροσωπευτικό του γνωρίσματος, ενώ συνήθως ο τύπος παραμέτρου του mutator είναι ίδιος με τον επιστρεφόμενο τύπο του observer

```
class Window {  
    public:  
    unsigned      GetX (void) const;  
    bool          SetX (unsigned _x);  
    const Metrics& GetMetrics (void) const; ← To σωστό style  
    void          GetMetrics (Metrics* metrics) const; ← Don't do this  
    bool          SetMetrics (const Metrics& metrics);  
};
```

Κατηγοριοποίηση μεθόδων (8/14)

■ *Replicator (clone creator) - κλωνοποιητής στιγμιότυπων*

- Είναι ειδική περίπτωση producer ο οποίος παράγει αντίγραφο του εαυτού του, όσον αφορά τα εσωτερικά δεδομένα, κατάσταση και λειτουργικότητα
- Επιστρέφει ένα δυναμικό στιγμιότυπο (δείκτη), και συνήθως δεν έχει τυπικές παραμέτρους. Συνήθως ονομάζεται “Replicate”, “Clone”, ή “Reproduce”
- Εσωτερικά, η υλοποίηση μπορεί να βασίζεται σε έναν copier constructor.
- Πολύ χρήσιμος και συνηθισμένος είναι ο *virtual replicator*
- Εάν το object δεν μεταβάλλεται είναι πάντοτε const

```
class Agent {  
    public:  
        virtual Agent* Clone (void) const  
            { return new Agent(*this); } // Via copy constructor  
};
```


Κατηγοριοποίηση μεθόδων (9/14)

■ *Comparator – συγκριτής στιγμιότυπων*

- Ειδική περίπτωση observer, υλοποιεί κάποια λογική τελεστή σύγκρισης μεταξύ στιγμιότυπων μίας κλάσης, ή σπανιότερα μεταξύ στιγμιότυπων διαφορετικών κλάσεων
- Συνήθως επιστρέφει true ή false, ως το αποτέλεσμα της σύγκρισης, ή έναν πιο περιεκτικό τύπο, εάν αποτελέσματα της σύγκρισης πρέπει να αναφέρονται. Ονομάζεται αναλόγως με τη σημασιολογία της σύγκρισης.

```
class Vector {  
    public:  
    bool operator==(const Vector& vector) const; ← Equality if applicable  
    bool IsOrthogonal (const Vector& vector) const;  
};  
class Agent {  
    public:  
    bool IsSuperior (const Agent&, QualityAttrs* outResults) const;  
};
```

Κατηγοριοποίηση μεθόδων (10/14)

■ *Serializer / encoder – μετατροπέας σειριακής μορφής*

- Methods που επιτρέπουν τη μετατροπή στιγμιότυπων σε μία σειριακή μορφή αποθήκευσης (κωδικοποίηση) και αναδόμηση τους από αντίστοιχη πηγή (αποκωδικοποίηση)
- Μπορεί να διαχειρίζεται σωστά ακόμη και τις δυναμικές σχέσεις μεταξύ στιγμιότυπων, με κατάλληλη αναδρομική εφαρμογή και πακετάρισμα των εμπλεκόμενων δυναμικών στιγμιότυπων.
- Χρησιμοποιούνται και τα ονόματα Encode / Decode αντί Serialize / Deserialize
- Εάν έχουμε τετριμμένο serialization σε αρχείο με κάποιο format, τότε μπορούμε να χρησιμοποιούμε τα ονόματα Store / Load ή Write / Read.

```
class Tree {  
    public:  
    void* Serialize (unsigned int* bytesUsed) const;  
    void Serialize (FILE* fp) const;  
    bool Deserialize (const void* buffer);  
    bool Deserialize (FILE* fp);  
    Tree (FILE* fp) // Decoder constructor with deserializer  
        { Tree::Tree(); Deserialize(fp); }  
};
```

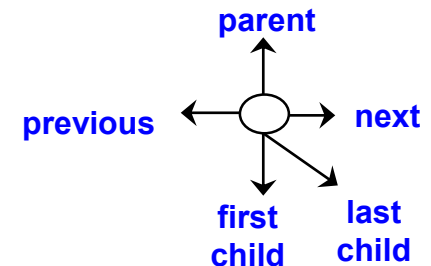
Κατηγοριοποίηση μεθόδων (11/14)

■ **Verifier (validator)– επαληθευτής στιγμιότυπων**

- Ειδική method που μπορεί να επαληθεύει το εκάστοτε στιγμιότυπο ως προς την συνολική ορθότητά του
 - ◆ π.χ., μνήμη, θέση στην εφαρμογή, περιεχόμενα (data members)
- Κάθε στιγμιότυπο επαληθεύεται πριν και μετά τη κλήση μελών, για τον γρήγορο εντοπισμό πιθανών λαθών ή δυσλειτουργιών
- Πρόκειται για κλασική αμυντική τεχνική αντιμετώπισης λαθών, ενώ η λογική επαλήθευσης ποικίλει

```
class Verifiable { public: virtual bool Verify (void) const = 0; };
```

```
class Tree : public Verifiable {  
    private:  
    bool Validate (Node* node) const {  
        return !node || ValidatePointer(node) &&  
            Validate(node->firstChild) // DFS, L→R  
            Validate(node->next);  
    }  
    public:  
    bool Verify (void) const { return Validate(root); }  
};
```



Κατηγοριοποίηση μεθόδων (12/14)

■ *Processor – επεξεργαστής στιγμιότυπων*

- Είναι μία **κατηγορία** που συγκεντρώνει όλες τις συναρτήσεις οι οποίες περατώνουν κάποιου είδους λειτουργία σχετική με το συνολικό αλγοριθμικό ρόλο της κλάσης
- *Οι processors είναι αυτοί που εξηγούν το λόγο ύπαρξης μίας κλάσης και συνήθως είναι λίγοι, ενώ όλοι σχετίζονται με τον ίδιο λειτουργικό ρόλο (της κλάσης)*
- Οι λειτουργίες ενδέχεται να τροποποιούν το στιγμιότυπο και να επιστρέφουν αποτελέσματα, αλλά δεν είναι απαραίτητο κάτι τέτοιο
- Λόγω της γενικότητας της κατηγορίας, θα πρέπει να γίνεται οργάνωση σε πιο εξειδικευμένες κατηγορίες επεξεργαστών, ανάλογα με την εκάστοτε σημασιολογία των επεξεργαστικών συναρτήσεων

```
class Agent {  
    public:  
    void StartMission (void);  
    void AbortMission (void);  
    void ProgressMission (void);  
};
```

Κατηγοριοποίηση μεθόδων (13/14)

■ *Notifier (callback, event handler)– ειδοποιητής (1/2)*

- Ομάδα συναρτήσεων τις οποίες καλούν οι κλάσεις αυτόματα για διαχείριση συγκεκριμένων κατηγοριών γεγονότων (events) τα οποία λαμβάνουν χώρα σε στιγμιότυπα.
- Υπάρχουν δύο μέθοδοι: (α) registration style, (β) derivation style
- Το πρώτο υποστηρίζει τη δυναμική εκδήλωση ενδιαφέροντος και την παροχή συναρτήσεων ως παραμέτρους, όταν εντοπίζονται ή συμβαίνουν τα αντίστοιχα γεγονότα.

```
#include <algorithm>
#include <functional>
#include <list>
```

registration style

```
class Button {
public:
    typedef void (*OnPress) (Button*, void* closure);
private:
    typedef std::pair<OnPress, void*> PressCallback;
    std::list<PressCallback> onPress;

    struct Notify :
    public std::binary_function<PressCallback, Button*, void> {
        void operator() (PressCallback& c, Button* b) const
        { (*c.first) (b, c.second); }
    };
};
```

```
void NotifyAll (void) {
    std::for_each(
        onPress.begin(), onPress.end(),
        std::bind2nd(Notify(), this)
    );
}

public:
    void AddOnButtonPress (OnPress f, void* c)
    { onPress.push_back(PressCallback(f,c)); }

    void RemoveOnButtonPress (OnPress f, void* c)
    { onPress.remove(PressCallback(f, c)); }
};
```

Κατηγοριοποίηση μεθόδων (14/14)

■ *Notifier (callback, event handler) – ειδοποιητής (2/2)*

- Το δεύτερο υποστηρίζει τον επαναπροσδιορισμό μίας virtual function μέσω derivation
- Είναι type-safe καθώς δεν υπάρχει περίπτωση να περάσουμε callback που δεν ορίζεται για το αντίστοιχο class

```
class Button2 {  
public: virtual void OnPress (void);  
};
```

```
class MyButton : public Button2 {  
public:  
    void OnPress (void) {  
        // Do what needed here.  
        // May also call base handler: augmentation style.  
        Button2::OnPress();  
    }  
};
```

*Derivation
style*

```
// But can turn a derivation style to a registration one.  
class MyButton2 : public Button, public Button2 {  
private:  
    void OnPress (void) { NotifyAll(); } // Hiding derivation style.  
public:  
    // Inheriting registration style.  
};
```