ΗΥ352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΔΑΣΚΩΝ

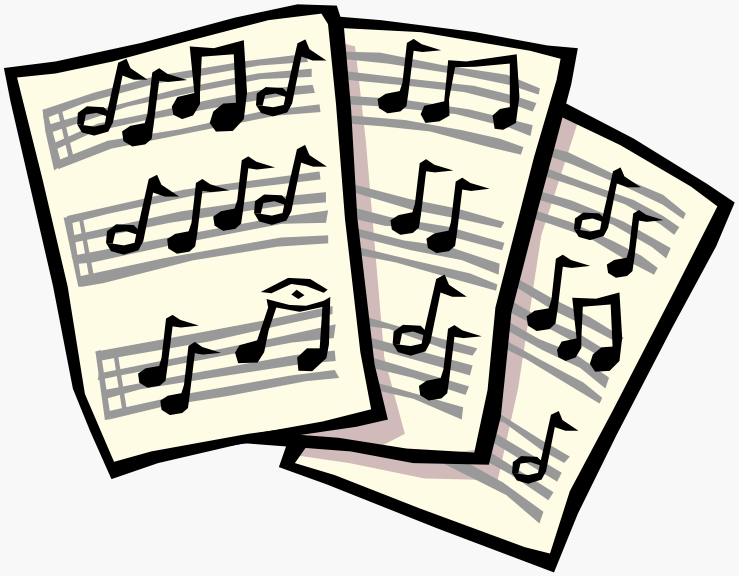
Αντώνιος Σαββίδης

CSD

ΕΝΟΤΗΤΑ 5

ΣΧΕΔΙΑΣΤΙΚΑ ΠΡΟΤΥΠΑ

Αριθμός διαλέξεων 5 – Διάλεξη 5η



Περιεχόμενα

- *Fly weight*
- Command
- Undo / redo

Flyweight (1/9)

- ***Πρόβλημα***
  - Έχουμε ανάγκες δημιουργίας ενός πολύ μεγάλου αριθμού στιγμιότυπων
  - αλλά εάν αναλύσουμε την εσωτερική κατάσταση αυτών, μπορούμε να εντοπίσουμε ένα μικρό σύνολο πραγματικά διαφορετικών στιγμιότυπων με πολύπλοκη κατάσταση (***heavyweight***) με τη διαφοροποιούμενη κατάσταση να είναι πολύ απλή (***flyweight***)
  - Η επανάληψη της κοινής κατάστασης για ένα τόσο μεγάλο αριθμό στιγμιότυπων μπορεί να οδηγήσει σε προβλήματα τόσο μνήμης όσο και απόδοσης

Flyweight (2/9)

- ***Λύση***
  - Αντί να επαναλαμβάνουμε την εσωτερική αυτή κατάσταση με διαφορετικά στιγμιότυπα, η κοινή αμετάβλητη κατάσταση μοντελοποιείται με shared heavyweight objects και χρησιμοποιείται από κοινού από όλα τα αντίστοιχα flyweight objects
  - Η μεταβλητή κατάσταση θα πρέπει να επαναλαμβάνεται ανά στιγμιότυπο και να διαχωρίζεται από την αμετάβλητη κατάσταση
  - Η νέα κλάση τώρα λέγεται flyweight εμπεριέχει ένα δείκτη στο heavyweight στιγμιότυπο, καθώς και τα κατάλληλα μέλη της μεταβλητής κατάστασης

Flyweight (3/9)

- ***Επιπτώσεις***
  - Όταν η κοινή αμετάβλητη κατάσταση εμπλέκει έναν σημαντικό αριθμό δεδομένων, μειώνεται σημαντικά το ποσό απαιτούμενη μνήμης
  - Είναι ευκολότερη η αποθήκευση / φόρτωμα μίας τέτοιας συλλογής στιγμιότυπων (κερδίζουμε χώρο και στο δίσκο)
  - Οι όποιες μεταβολές στο heavyweight έχουν αντίκτυπο στα στιγμιότυπα που το μοιράζονται χωρίς ανάγκη περαιτέρω επεξεργασίας
  - Μπορούμε να έχουμε διαφοροποιήσεις στα flyweights εάν αυτές είναι πολύ μικρές σε σχέση με το memory footprint του heavyweight

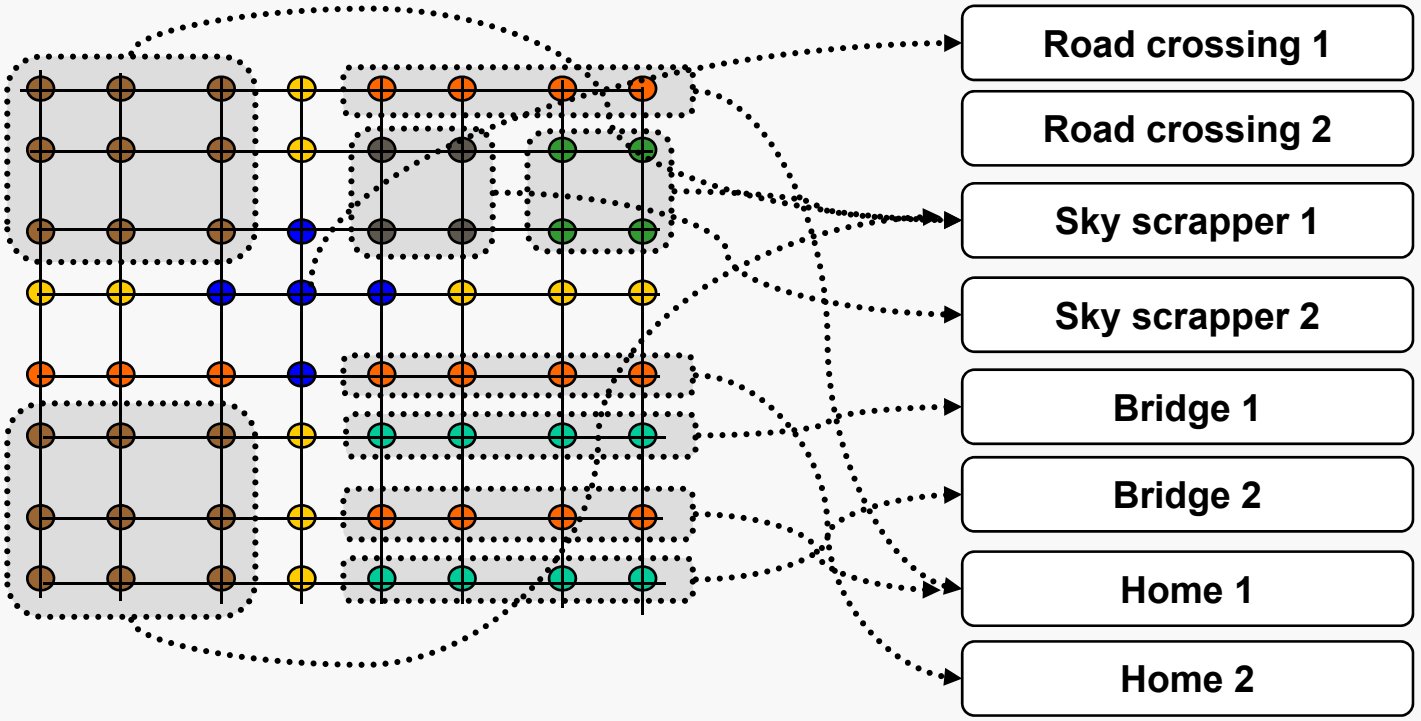
Flyweight (4/9)

- *Παράδειγμα*
  - Κατασκευή ενός *city simulator*. Ο εξομοιωτής πρέπει να είναι ικανός να αντεπεξέλθει στην κατασκευή πόλεων από 20 χιλ. έως και 20 εκατ. κατοίκους
  - Ο εξομοιωτής παρέχει γραφικά δομικά στοιχεία όπως: “road crossing”, “block”, “traffic lights”, “sky scrapper”, “bridge”, “road segment”, “building”, “theatre”, “school”, “church”, etc
  - Κάθε δομικό στοιχείο έχει την δική του εσωτερική κατάσταση, μέθοδο γραφικής παρουσίασης, και πολιτική διασύνδεσης με άλλα στοιχεία
  - Αναμένουμε να έχουμε δεκάδες χιλιάδες στιγμιότυπα από τέτοια δομικά στοιχεία

Flyweight (5/9)

Παράδειγμα (1/5)

City grid plan - μικρό τμήμα



Flyweight distinct objects

Heavyweight shared objects

Flyweight (6/9)

Παράδειγμα (2/5)

```
class SkyScrapper_Heavy_Style1 {
   public:
   void Draw (Position&, Attributes&, unsigned floors) const;
};
class Bridge_Heavy_Style2 {
   public:
   void Draw (Position& start, Position& end, Attributes&) const;
   static const Bridge_Fly_Style2* Get (void);
};
class Bridge_Fly_Style2 {
   private:
   const Bridge_Heavy_Style2* heavy;
   Position                   start;
   Position                   end;
   Attributes                 attrs;
   public:
   void Draw (void) const { heavy->Draw(start, end, attrs); }
};
```

Parameterized rendering

Κοινό στιγμιότυπο (singleton)

Κοινή αμετάβλητη κατάσταση

Μεταβλητή κατάσταση

Χρήση του heavyweight στιγμιότυπου με κλήση observer methods and delegation calls

Flyweight (7/9)

Παράδειγμα (3/5)

- Η heavyweight κλάση δεν περιέχει μεταβλητές σχετικές με το «περιβάλλον» στο οποίο το στιγμιότυπο χρησιμοποιείται (όπως π.χ. Position), ή μεταβλητές που μπορεί να μην είναι στοιχεία της κατάστασης, αλλά μπορεί να ποικίλουν στη χρήση (π.χ. Attributes ζωγραφικής).

```
class Bridge_Heavy_Style2 {
   void Draw (const Position&, const Position&, const Attributes&) const;
};
```

- Τέτοιου είδους μεταβλητά κατά τη χρήση χαρακτηριστικά ορίζονται ως χαρακτηριστικά της απλής κλάσης. Ως αποτέλεσμα, ορισμένα μέλη του heavyweight class χρειάζονται παραμετροποίηση (π.χ. η Draw συνάρτηση).

```
class Bridge_Fly_Style2 {
   const Bridge_Heavy_Style2* heavy;
   Position                   start;
   Position                   end;
   Attributes                 attrs;
};
```

Flyweight (8/9)

- Το flyweight pattern μπορεί να εφαρμοστεί και σε περιπτώσεις που όλα τα στιγμιότυπα ανήκουν στην ίδια κλάση, αλλά χρειαζόμαστε μόνο ένα μικρό σύνολο πραγματικά διαφορετικών καταστάσεων, αν και έχουμε εκατοντάδες χιλιάδες run-time στιγμιότυπα.
- Π.χ., σε έναν εξομοιωτή πολέμου έχουμε 100.000 στρατιώτες, από μία συλλογή περίπου 100 διαφορετικών συμπεριφορών, δεξιοτήτων και εμφάνισης. Κατά την εκτέλεση, κάθε στιγμιότυπο χρειάζεται απλά να επαναλαμβάνει: θέση, εντολές, κινηματική κατάσταση, και φυσική κατάσταση. Ο πυρήνας AI συμπεριφοράς, τα γραφικά δεδομένα, ο οπλισμός, κλπ., μπορούν να βελτιστοποιηθούν σε ένα μικρό σύνολο από heavyweight στιγμιότυπα.

Παράδειγμα (4/5)

```
class Soldier {
   PhysicalState          physicalState;
   Position               position;
   Command                currCommand;
   const Soldier_Heavy*   heavy;
};
class Soldier_Heavy_Pool {
   static const Soldier_Heavy* Get (Type type);
};
```

All pictures are a courtesy of Microsoft Corp., title "Age of Empires"

Flyweight (9/9)

- Τα heavyweight στιγμιότυπα μπορεί να δημιουργούνται κατά την εκτέλεση, ή, για να πετύχουμε καλύτερη απόδοση, και για να προσθέσουμε πλέον ευελιξία, μπορεί να φορτώνονται από το δίσκο. Σε περισσότερο ανοικτά συστήματα ειδικοί configuration editors.
- Επιπλέον, μπορεί να χρειαστεί να επιτρέπεται η δυναμική επέκταση των heavyweights. Τότε, θα πρέπει να επιστρέφεται ένα δυναμικό αναγνωριστικό τύπου με το οποίο ο προγραμματιστής θα εισάγει το νέο heavyweight στιγμιότυπο.

Παράδειγμα (5/5)

```
class Soldier_Heavy_Pool {
   public:
   static bool LoadAll (const char* filePath);
   static bool LoadSingle (Type type, const char* filePath);
   static bool StoreAll (const char* filePath);
   static bool StoreSingle (Type type, const char* filePath);
   static Type GetNewType (void);
   static bool Install (Type type, Soldier_Heavy* heavy);
};
```

Περιεχόμενα

- Fly weight
- *Command*
- Undo / redo

Command (1/7)

- ***Πρόβλημα***
  - Έχουμε λειτουργίες (***commands***) που πρέπει να εφαρμοστούν κάποια στιγμή αργότερα από άγνωστα αντικείμενα (***invokers***)
  - Τόσο οι λειτουργίες, όσο και οι πραγματικοί παράμετροι πρέπει να αποθηκεύονται με κάποιο τρόπο
  - Οι λειτουργίες μπορεί να χρησιμοποιούν άλλα αντικείμενα στα οποία προωθούν τις κλήσεις (***receivers***)
  - Οι invokers δεν έχουν καμία ιδέα (είναι ανεξάρτητοι) για τους receivers

Command (2/7)

- ***Λύση***
  - Οι λειτουργίες μοντελοποιούνται μέσω μίας αφαιρετική κλάσης Command που περιέχει μία abstract method με το όνομα ***execute*** και signature void (void)
  - Μπορεί να υποστηρίζει copy construction και cloning εάν αυτό ορίζεται σύμφωνα με τη λειτουργία
  - Κληρονόμοι κλάσεις (derived) υλοποιούν τις λειτουργίες και τις εξαρτήσεις με τους receivers

Command (3/7)

```
class Command {
public:
    virtual void Execute (void) = 0;
    virtual ~Command(){}
};

class EditorCommand : public Command {
protected:
    Editor* editor;
public:
    // extra abstract methods with meaning for
    // editing
    virtual void Undo (void) = 0;
    virtual void Redo (void) =0;
    EditorCommand (Editor* _editor) : editor (_editor){}
};
```

Command (4/7)

```
class DragAndDropCommand : public EditorComannd {
protected:
    Item        dragedObj;
    Position    dragSrc, dragDest;
public:
    virtual void Execute (void) {
        -   move draggedObj at dragDest
    }
    virtual void Undo (void) {
        -   move draggedObj at dragSrc
    }
    virtual void Redo (void)
            { Execute(); }
    DragAndDropCommand (
        Editor*     _editor,
        Position    _src,
        Position    _dest,
        Item        _obj
    );
};
```

Command (5/7)

```
class DeleteCommand : public EditorCommand {
protected:
    Item        removedObj;
    Data        restoreData;    // data to undo removal
public:
    virtual void Execute (void) {
        if (obj) {
        -   set / copy / create restoreData
        -   remove obj from the editor and clear obj
        }
    }
    virtual void Undo (void) {
        -   reintroduce obj via restoreData
        -   clear restoreData
    }
    virtual void Redo (void)
            { Execute(); }
    DeleteCommand (Editor* _editor, Item _obj);
};
```

Command (6/7)

```
class CommandHistory {
private:
    std::list<EditorCommand*> history;
    std::list<EditorCommand*>::iterator i;

public:
    // i: is where PREVIOUS undo took place
    void Undo (void) {
        if (history.empty() || i == history.begin())
            return;
        else
            (*--i)->Undo(); // notice pre-decrement
    }

    // i: is where NEXT redo will take place
    void Redo (void) {
        if (history.empty() || i == history.end())
            return;
        else
            (*i++)->Redo(); // notice post-increment
    }
```

Command (7/7)

Με κάθε *editing action* δε μπορούμε πλέον να κάνουμε *redo* (για το λόγο αυτό κάνουμε clear όλα τα commands μετά τον current iterator)

```
    // with every editing action we add the respective
    // command here - in case we have undone, the redone
    // actions are cleared
    void Edit (EditorCommand* cmmd) {
        if (i != history.end()) // redos have been applied
            while (i != history.end()) {    // clear all redos
                delete *i;
                i = history.erase(i);
            }
        history.push_back(cmmd);
        cmmd->Execute();    // and apply the editor command
    }

    CommandHistory  (void)
        { i = history.end(); }
};
```

Περιεχόμενα

- Fly weight
- Command
- *Undo / redo*

Undo / redo (1/8)

- ***Πρόβλημα***
  - *Undo*. Έχουμε στιγμιότυπα τα οποία υπόκεινται σε διάφορες επεξεργασίες / λειτουργίες από το χρήστη (π.χ. text editor, graphic editor, κλπ). Θέλουμε να μπορούμε να ακυρώνουμε τέτοιες λειτουργίες, επαναφέροντας τα στιγμιότυπα στην ακριβώς προηγούμενη κατάσταση (αυτό να ισχύει αναδρομικά)
  - *Redo*. Θέλουμε επίσης να μπορούμε να ακυρώνουμε τις ακυρώσεις, εφόσον δεν έχει γίνει κάποια άλλη επεξεργασία ενδιάμεσα, επιστρέφοντας στην προηγούμενη κατάσταση πριν την ακύρωση (αυτό ισχύει επίσης αναδρομικά)

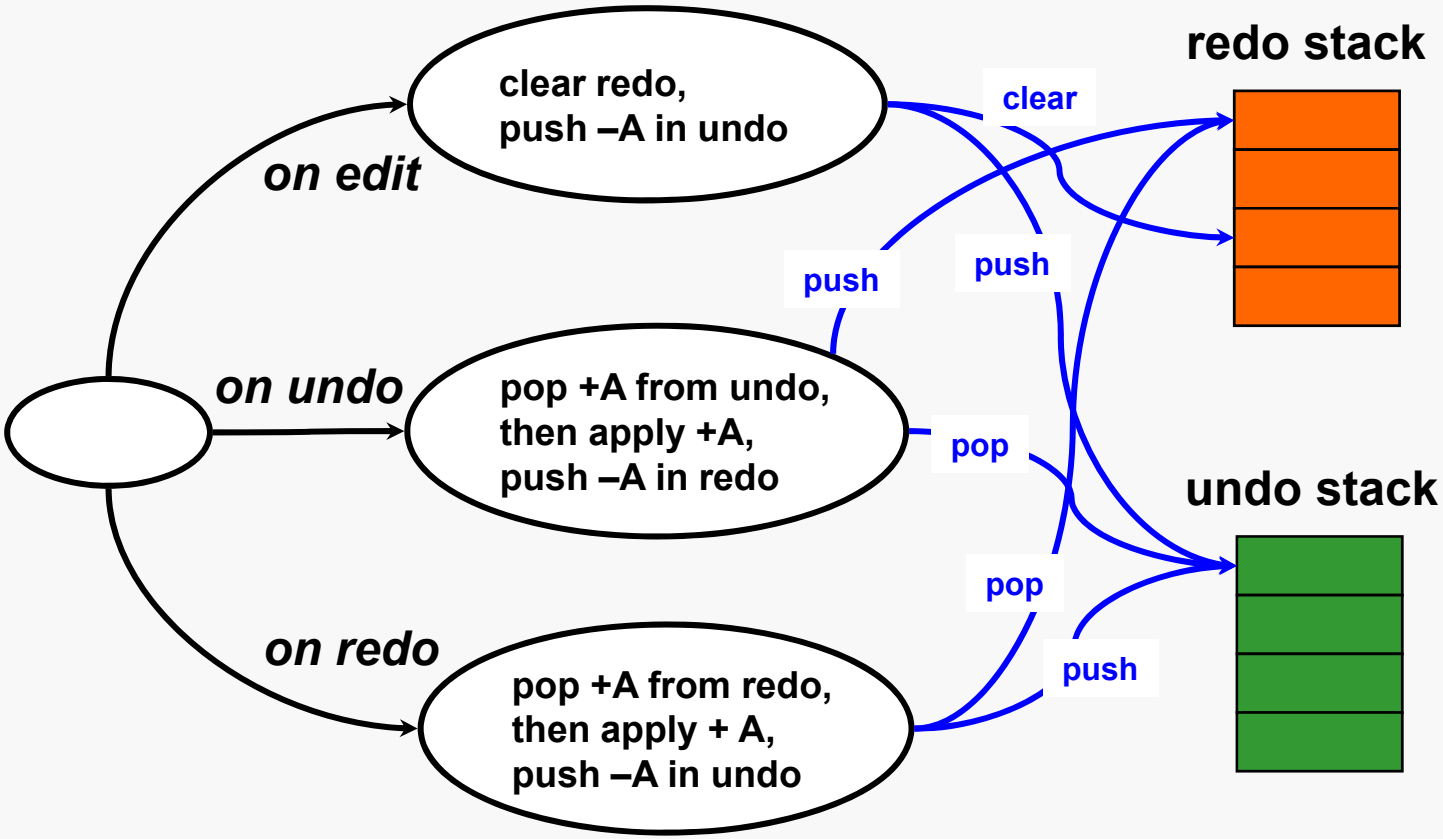
Undo / redo (2/8)

- ***Λύση***
  - Πριν την εφαρμογή μίας λειτουργίας ***+A***, κάνε push την “αντίθετη” λειτουργία ***-A*** στην undo stack
    - Εάν η ***–A*** εφαρμοστεί ακριβώς μετά την ***+A***, η κατάσταση του στιγμιότυπου πρέπει να είναι ακριβώς όπως ήταν πριν η ***+A*** εφαρμοστεί
  - ➔ Η ακύρωση (undo) μίας λειτουργίας ***+A*** υλοποιείται με την εφαρμογή της αντίθετης λειτουργίας***–A*** η οποία λαμβάνεται από την κορυφή της undo stack. Έπειτα, κάνουμε pop την ***–A,*** ενώ η ***+A*** γίνεται push στην redo stack
  - ➔ Η ακύρωση της ακύρωσης (redo) υλοποιείται με την εφαρμογή και pop της λειτουργίας από την κορυφή της redo stack, και με την εισαγωγή της αντίθετής της στην undo stack. Εάν γίνει κάποια κανονική λειτουργία επεξεργασίας, η redo stack καταστρέφεται

Undo / redo (3/8)

- ***Επιπτώσεις***
  - Πρέπει να ορίσουμε και να υλοποιήσουμε τις αλγοριθμικά αντίθετες λειτουργίες
  - Σε μερικές περιπτώσεις, το αντίθετο μίας λειτουργίας (όπως π.χ. της delete) μπορεί να χρειάζεται πολλαπλές εσωτερικά λειτουργίες για την υλοποίησή της
  - Σε αρκετές περιπτώσεις, για την επαναφορά στην προηγούμενη κατάσταση, μπορεί να είναι ευκολότερη η αποθήκευση της εσωτερικής κατάστασης και εσωτερικών δομών (και όχι μόνο αυτών που βλέπει ο client programmer)

Undo / redo (4/8)


clear redo,
push –A in undo
on edit
on undo
pop +A from undo,
then apply +A,
push –A in redo
on redo
pop +A from redo,
then apply + A,
push –A in undo
clear
push
push
pop
pop
push
redo stack
undo stack

Undo / redo (5/8)

- *Προγραμματιστικές απαιτήσεις*
  - Υλοποίηση αλγοριθμικά αντίθετων λειτουργιών.
  - Ορισμός τύπων λειτουργιών και δομών δεδομένων που θα περιέχουν τα ορίσματα των λειτουργιών αυτών
  - Ορισμός ενός ενοποιημένου τύπου, για όλους τους τύπους λειτουργιών, στιγμιότυπα του οποίου θα εισάγονται στις στοίβες
  - Χρειάζεται ένας dispatch table για την εκτέλεση των λειτουργιών βάσει του αντιστοίχου τύπου λειτουργίας

Undo / redo (6/8)

Παράδειγμα (1/3)

```
enum EditOperationId  { Insert = 0, Delete = 1};

class ContentBuffer {...};

struct InsertData {
   unsigned short col, row;
   ContentBuffer* content;
};

struct DeleteData {
   unsigned short fromCol, fromRow, toCol, toRow;
};

struct EditOperation {
   EditOperationId id;
   void*           data; ← Type cast to necessary type
};
```

Undo / redo –(7/8)

Παράδειγμα (2/3)

```
class Editor {
   EditOperation* ProduceOpposite (EditOperation* operation);
   void           Perform (EditOperation* operation);
   void           Clear (EditOperation* operation);
   stack<EditOperation*> undoStack;
   stack<EditOperation*> redoStack;
 public:
   void Insert (char c, unsigned short row, unsigned short col) {
       DeleteData* del = new DeleteData;   ← Opposite operation
       del->fromCol = del->toCol = col;
       del->fromRow = del->toRow = row;
       EditOperation* undo = new EditOperation;
       undo->id = Delete, undo->data = del;
       undoStack.push(undo);
       for (auto i : redoStack) delete i; redoStack.clear();
       - Implement insertion logic here
   }
 };
```

Undo / redo (8/8)

Παράδειγμα (3/3)

```
class Editor {
   public:
   void Undo (void) {
      EditOperation* undo = undoStack.top();
      undoStack.pop();
      redoStack.push(ProduceOpposite(undo));
      Perform(undo);
      Clear(undo);
   }
   void Redo (void) {
      EditOperation* redo = redoStack.top();
      redoStack.pop();
      undoStack.push(ProduceOpposite(redo));
      Perform(redo);
      Clear(redo);
   }
};
```