ΗΥ352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

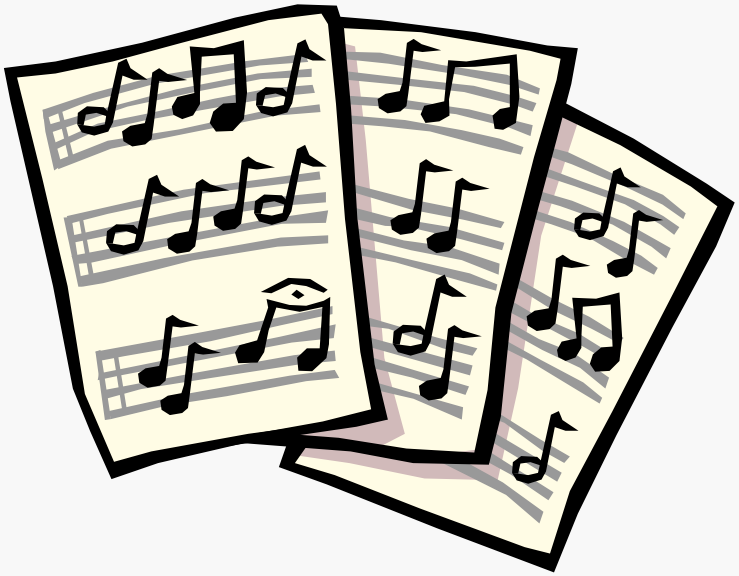
ΔΙΔΑΣΚΩΝ

Αντώνιος Σαββίδης

ΕΝΟΤΗΤΑ 5

ΣΧΕΔΙΑΣΤΙΚΑ ΠΡΟΤΥΠΑ

Αριθμός διαλέξεων 5 – Διάλεξη 3η



Περιεχόμενα

- *Adapter*
- Proxy
- Dispatch table
- Blackboard

Adapter (1/9)

- ***Πρόβλημα***
  - Θέλουμε να χρησιμοποιήσουμε την *αυθεντική λειτουργικότητα* μίας κλάσης αλλά μέσα από ένα *διαφορετικό API*.
  - Δεν μπορούμε να αλλάξουμε την αυθεντική κλάση.
  - Μπορεί να υπάρχει ήδη κώδικας ο οποίος χρειάζεται το διαφορετικό API.

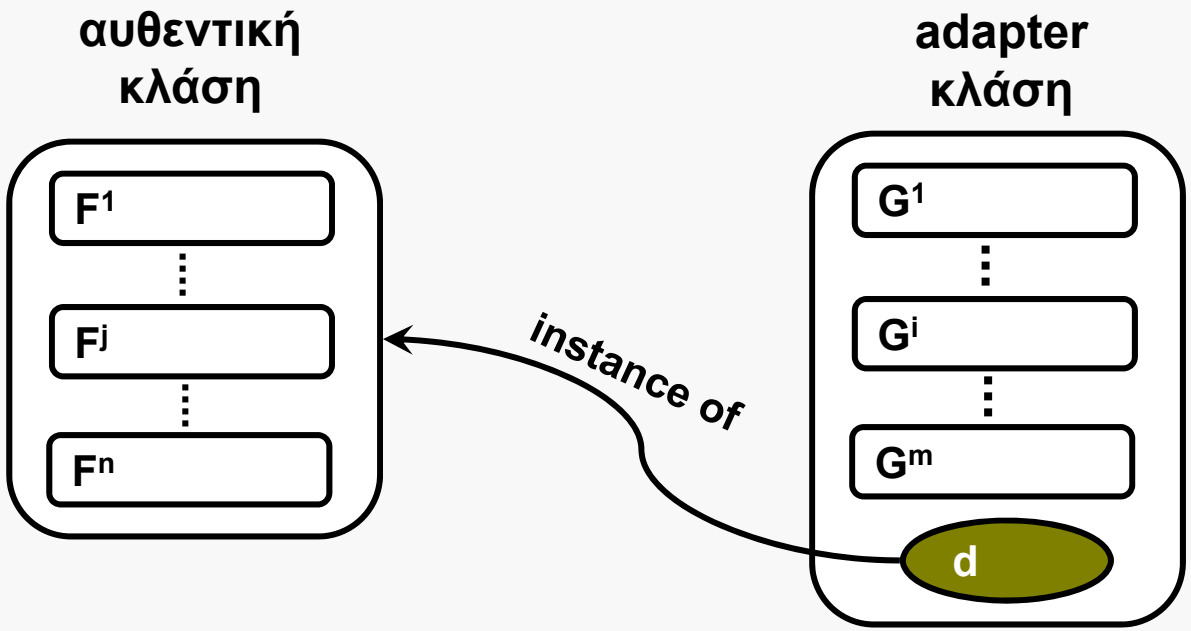
Adapter (2/9)

- ***Λύσεις***
  - Δημιούργησε μία νέα κλάση η οποία κληρονομεί από την αυθεντική, αλλά τροποποιεί το API όπως χρειάζεται.
    - *Μέσω κληρονομικότητας (sub-classing style)*
  - Δημιούργησε μία νέα κλάση με το τροποποιημένο API, η οποία δεν κληρονομεί από την αυθεντική, αλλά έχει εσωτερικά ένα στιγμιότυπο αυτής, μέσω του οποίου και πραγματοποιούνται οι κλήσεις στις συναρτήσεις της αυθεντικής κλάσεις
    - *Μέσω εξουσιοδότησης (delegation style) – συνήθης λύση*

Adapter (3/9)

- ***Επιπτώσεις***
  - Χρειάζεται να υλοποιηθούν οι κλάσεις τροποποίησης API (adapters), οι οποίες λέγονται επίσης και *wrappers* (περιτυλίγματα).
  - Μπορεί να βοηθήσει στην μεταφορά παλαιού χρήσιμου κώδικα πολύ γρήγορα,
    - ενώ έπειτα, με την εφαρμογή του re-factoring, να μετασχηματίσουμε σταδιακά την adapter «ελαφριά» κλάση σε μία πλήρη «βαριά» κλάση,
    - αφαιρώντας βήμα-βήμα τις κλήσεις σε απαρχαιωμένες κλάσεις, και αντικαθιστώντας αυτές σε κλήσεις στις νέες αλγοριθμικά συναρτήσεις,
    - μετατρέποντας την adapter κλάση στη εντελώς νέα έκδοση της αυθεντικής κλάσης

Adapter (4/9)



Σχήμα υλοποίησης μέσω εξουσιοδότησης

Οι κλήσεις της μορφής $d \rightarrow F^j$ στην υλοποίηση της συνάρτησης G^i, ονομάζονται κλήσεις εξουσιοδότησης (*delegation calls*).

Adapter (5/9)

παράδειγμα (1/3)

```
class MOTIFNativeWindow : public MOTIFNativeObject {
};
```
Η αυθεντική GUI κλάση

```
class AbstractWindow {
};
```
Μία αφηρημένη κλάση της αφηρημένης οικογένειας

```
class MOTIFAdaptedWindow : public AbstractWindow {
};
```
Μία adapter κλάση της adapted οικογένειας η οποία κληρονομεί από την αφηρημένη κλάση της αφηρημένης οικογένειας

- Χρειάζεται να κατασκευάσουμε ένα GUI factory
- Συνεπώς, χρειαζόμαστε μία αφηρημένη οικογένεια GUI κλάσεων
- Συνεπώς, όλες οι κλάσεις των αυθεντικών GUI οικογενειών πρέπει να κληρονομούν από τις αντίστοιχες κλάσεις της αφηρημένης οικογένειας
- Επειδή όμως δεν μπορούμε να τροποποιήσουμε τις αυθεντικές κλάσεις, δημιουργούμε adapter κλάσεις των αυθεντικών GUI κλάσεων, δηλ. μία adapted GUI οικογένεια

Adapter (6/9)

παράδειγμα (2/3)

```
class MOTIFAdaptedWindow : public AbstractWindow {
private:
   MOTIFNativeWindow* nativeInst;
public:
   virtual void Draw (void) const override { nativeInst->Draw(); }
   virtual void Maximise (void)   override { nativeInst->SetWholeScreen(); }
   virtual void Minimise (void)   override { nativeInst->Iconify(); }

   MOTIFAdaptedWindow(...)
      { nativeInst = new MOTIFNativeWindow(...); }

   ~MOTIFAdaptedWindow()
      { delete nativeInst; }
};
```

Κλήσεις εξουσιοδότησης (delegation calls)

Adapter (7/9)

παράδειγμα (3/3)

```
// MotifFamily.h, all abstract family classes header
class MotifWindow : public Window {
   void* nativeInst;  // use opaque native instance to avoid #include
   public:
   MotifWindow(void);
   virtual ~MotifWindow(void);
};

// MotifFamily.cpp, all abstract family classes implementation
#include "MotifNative.h“ // here we include the native header

MotifWindow::MotifWindow(void)
         { nativeInst = new MOTIFNativeWindow(...); }

MotifWindow::~MotifWindow(void)
         { delete static_cast<MOTIFNativeWindow*>(nativeInst); }

void MotifWindow::Draw (void)
         { static_cast<MOTIFNativeWindow*>(nativeInst)->Draw(); }
```

Adapter (8/9)

- Έχουμε βιβλιοθήκες / τμήματα σε C και πρέπει να μετατραπούν σε ένα οντοκεντρικό API, επιτρέποντας έπειτα κληρονομικότητα, late binding, κλπ.

παράδειγμα (1/2)

Ένα τυπικό C API για μία βιβλιοθήκη GUI αντικειμένων

```
typedef void* Window; // Ο κλασικός στη C αδιαφανής τύπος

extern Window      WindowCreate (void);
extern void        WindowDestroy (Window wnd);
extern void        WindowSetTitle (Window wnd, char* title);
extern char*       WindowGetTitle (Window wnd);
extern void        WindowDraw (Window wnd);
extern bool        WindowIsTopOf (Window wnd);
...
```

Adapter (9/9)

παράδειγμα (2/2)

Μετασχηματισμός σε OOP API με πλήρη επαναχρησιμοποίηση κώδικα μέσω adapter κλάσεων

```
namespace GUI {
   class Window {
   private:
       ::Window wnd;        // native encapsulated instance
   public:
       virtual void   Draw (void) { WindowDraw(wnd); }
       const char*    GetTitle (void) { return WindowGetTitle(wnd); }
       Window (void) { wnd = WindowCreate(); }
       virtual ~Window() { WindowDestroy(wnd); }
    };
}
```

Τώρα μπορούμε να κληρονομούμε από τις μετασχηματισμένες κλάσεις

```
class GUI::AuditoryWindow : public GUI::Window {
  ...
};
```

Περιεχόμενα

- Adapter
- *Proxy*
- Dispatch table
- Blackboard

Proxy (1/5)

- ***Πρόβλημα***
  - Η αυθεντική λειτουργικότητα μίας κλάσης πρέπει να χρησιμοποιηθεί με κάποιου είδους επιπλέον προεργασία και επικύρωση πριν από κάθε κλήση, π.χ.
    - Thread safe έκδοση μίας κλάσης
    - Wrapper για μία κλάση που βρίσκεται σε άλλο process, και πιθανά σε άλλο μηχάνημα (δηλ. είναι service που χρειάζεται RPC wrapper)
  - Μπορεί να απαιτείται εσωτερικά να αλλάξουμε το στιγμιότυπο της αυθεντικής κλάσης που χρησιμοποιείται χωρίς να επηρεάζονται οι clients, π.χ.
    - Restart ενός service σε άλλο μηχάνημα
    - Stop ενός service και start τη νέα του έκδοση
    - Παροχή σταθερού στιγμιότυπου ελέγχου (π.χ. για agent), όταν το ελεγχόμενο αντικείμενο μεταβάλλεται δυναμικά

Proxy (2/5)

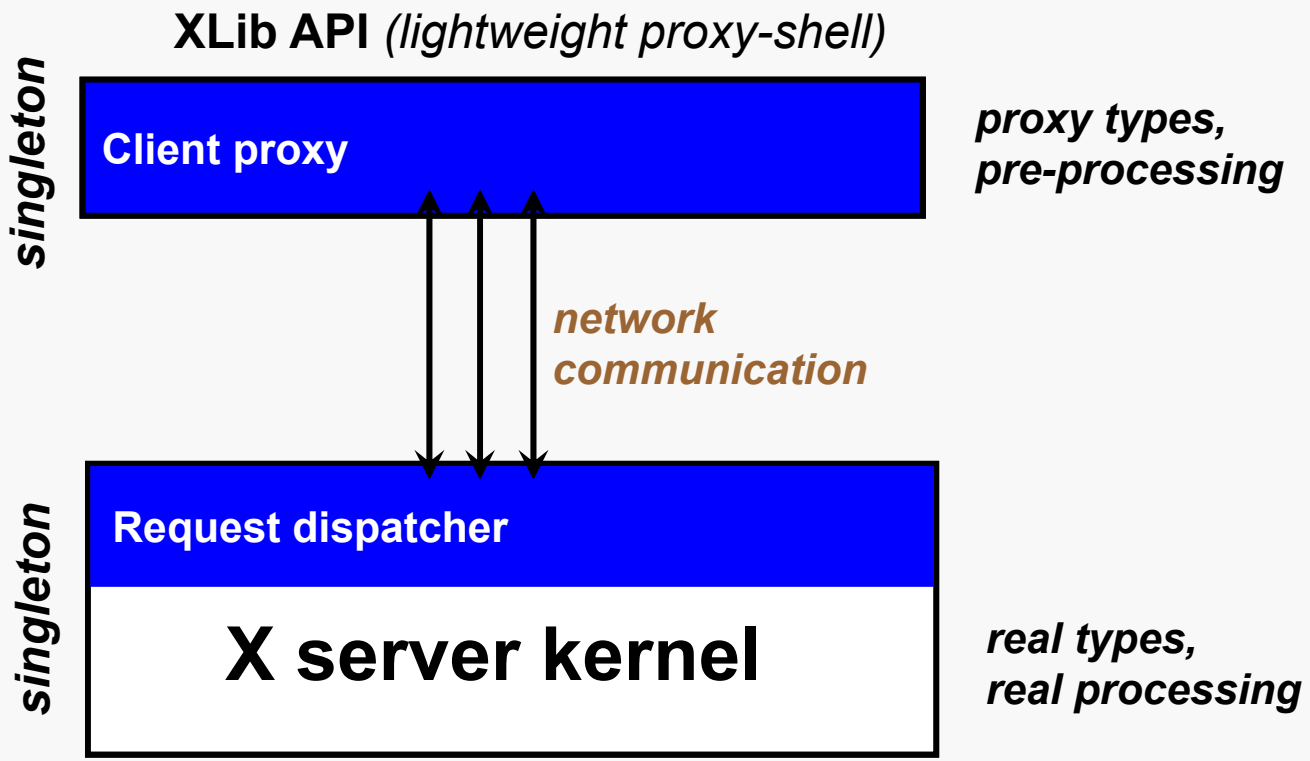
- ***Λύση***
  - όρισε το API που χρειάζεται ο client σε ξεχωριστή proxy κλάση
  - υλοποίησε την αναγκαία προεργασία μέσα στην proxy κλάση
  - ενσωμάτωσε ένα στιγμιότυπο (ή τα απαραίτητα δεδομένα) της αυθεντικής κλάσης (ή του σχετικού αρχικού API)
  - χρησιμοποίησε κλήσεις εξουσιοδότησης
  - έλεγχε την ορθότητα του στιγμιότυπου, και εάν χρειάζεται, δημιούργησε άλλο

  → ***είναι ειδική περίπτωση adapter***

Proxy (3/5)

- ***Επιπτώσεις***
  - Η αυθεντική κλάση ανεξαρτητοποιείται από τις όποιες πολιτικές προεργασίας και επικύρωσης, οι οποίες τώρα υλοποιούνται από τις proxy κλάσεις
  - Οι proxy κλάσεις δεν κληρονομούν ποτέ από τις αυθεντικές κλάσεις
  - Είναι πολύ συχνό να έχουμε proxies πάνω από singletons

Proxy (4/5)


XLib API (lightweight proxy-shell)
singleton
Client proxy
proxy types,
pre-processing
network
communication
singleton
Request dispatcher
X server kernel
real types,
real processing

Proxy (5/5)

... στο παράδειγμα η δομή υλοποίησης του Xlib τροποποιήθηκε για απλότητα παρουσίασης

Server-side

```
class XServer {
    Window CreateWindow (...);
    void   GetNextEvent (Window, Event*);
};
```

Client-side

```
class XClient { // proxy
    bool           Connect (const char* host);
    ExternalWindow CreateWindow (...) {
                       Στείλε ένα δικτυακό μήνυμα για την συνάρτηση αυτή
                       και περίμενε για το αποτέλεσμα
                   }
};
```

Server-side

```
class XServer {
    void    MainLoop(void) {
                Εδώ είναι το κλασικό server loop, ελέγχοντας για την παρουσία μηνυμάτων, δηλ.
                requests, και καλώντας τις κατάλληλες συναρτήσεις υλοποίησης, δηλ. τις service
                functions, ενώ στέλνει πίσω τα αποτελέσματα, δηλ. results, στον αιτούμενο client.
    }
};
```

Περιεχόμενα

- Adapter
- Proxy
- *Dispatch table*
- Blackboard

Dispatch table (1/6)

- ***Πρόβλημα***
  - Έχουμε μία κλάση που παρέχει ένα μεγάλο αριθμό από συναρτήσεις, που επιτελούν βασικές λειτουργίες,
    - ενώ οι αναγκαίες κλήσεις σε αυτές τις συναρτήσεις δεν είναι γνωστές κατά την ανάπτυξη, αλλά αποφασίζονται πάντα κατά την εκτέλεση
  - Ενώ η κλάση επεκτείνεται με την προσθήκη νέων τέτοιων συναρτήσεων, θέλουμε οι χρήστες της κλάσης να μπορούν χα χρησιμοποιούν ένα σταθερό σύνολο συναρτήσεων,
  - Επιθυμούμε να διατηρήσουμε την απόδοση αμετάβλητη σχετικά ανεξάρτητη του συνολικού αριθμού των συναρτήσεων

Dispatch table (2/6)

- ***Λύση***
  - Οι λειτουργίες ορίζονται ως αντίστοιχες σταθερές τιμές τους ίδιου enumerated τύπου, ή ως strings (όταν δυναμικές λειτουργίες μπορεί να προστίθενται)
  - Όλες οι λειτουργίες συλλέγονται σε ένα πίνακα ή hash table, στη θέση της αντίστοιχης τιμής του enumerated ή string
  - Η κλήση μίας συνάρτησης μπορεί να γίνει μόνο με τη χρήση της enumerated σταθερής τιμής, με την εξαγωγή της διεύθυνσης της συνάρτησης από τον πίνακα
  - ➔ ***Οι συναρτήσεις πρέπει να έχουν πανομοιότυπες υπογραφές, αλλιώς πρέπει να κάνουμε functor wrapper για κάθε συνάρτηση***

Dispatch table (3/6)

- ***Επιπτώσεις***
  - Αποφεύγουμε την χρήση μίας μεγάλης “switch”, ή πολλών συνεχόμενων “if then else” εντολών
  - Μπορούμε να τροποποιούμε αλγοριθμικά τις λειτουργίες ανεξάρτητα, αφού κάθε μία αντιστοιχεί σε ξεχωριστή συνάρτηση
  - Η απόδοση βελτιστοποιείται (εάν είναι πίνακας) και σταθεροποιείται

Dispatch table (4/6)

Παράδειγμα (1/3)

```
class VirtualMachine {
  public:
  void  ADD (const InstructionArgs& args);
  void  SUB (const InstructionArgs& args);
  void  MOV (const InstructionArgs& args);
  void  PUSH (const InstructionArgs& args);
  void  CALL (const InstructionArgs& args);
  void  RET (const InstructionArgs& args);
};
```

Για να πετύχουμε επεκτασιμότητα στην κατασκευή του virtual machine, η χρήση switch ή if then else" πρέπει να αποφεύγεται

Dispatch table (5/6)

Παράδειγμα (2/3)

```
enum OpCode {       // we need successive integers when using a table
  ADD_OpCode        = 0,
  SUB_OpCode        = 1,
  MOV_OpCode        = 2,
  PUSH_OpCode       = 3,
  CALL_OpCode       = 4
};
class VirtualMachine {
   private:
   void ADD (const InstructionArgs&);
   void SUB (cost InstructionArgs&);
   ...
   typedef void (VirtualMachine::*OpFunc)(const InstructionArgs&);
   std::map<OpCode, OpFunc> dispatchTable;
};
```

Dispatch table (6/6)

Παράδειγμα (3/3)

```
class VirtualMachine {
   public:
   VirtualMachine (void) {
      dispatchTable[ADD_OpCode]  = &VirtualMachine::ADD;
      dispatchTable[SUB_OpCode]  = &VirtualMachine::SUB;
      dispatchTable[MOV_OpCode]  = &VirtualMachine::MOV;
      dispatchTable[PUSH_OpCode] = &VirtualMachine::PUSH;
      dispatchTable[CALL_OpCode] = &VirtualMachine::CALL;
   }
   void Execute (OpCode opcode, const InstructionArgs& args)
     { (this->*dispatchTable[opcode])(args); }
   ...
};
```

Αρχικοποίηση του dispatch array στον constructor

Κλήση της κατάλληλης συνάρτησης σε ένα βήμα

Περιεχόμενα

- Adapter
- Proxy
- Dispatch table
- *Blackboard*

Blackboard (1/5)

- ***Πρόβλημα***
  - Υπάρχουν διάφορα ανεξάρτητα τμήματα τα οποία πρέπει να επικοινωνούν μεταξύ τους και να εκθέτουν (exporter) πληροφορία που μπορεί να ενδιαφέρει άλλα τμήματα (importer)
  - Τέτοιου είδους πληροφορία μπορεί να “ζει” για λίγο, και όχι να απλά να εμφανίζεται με τη μορφή ενός στιγμιαίου γεγονότος
  - Τα τμήματα μπορεί να εμφανίζονται δυναμικά, ενώ ακόμη και οι τύποι της πληροφορίας μπορεί να επεκτείνονται δυναμικά
  - Οι κανονισμοί πρόσβασης στην πληροφορία μπορεί να ποικίλουν: ημερομηνία λήξης, ο πρώτος που διαβάζει την πληροφορία την αφαιρεί, περιορισμένη χρήση σε ειδικούς αποδέκτες, πρόσβαση με κωδικό, κλπ

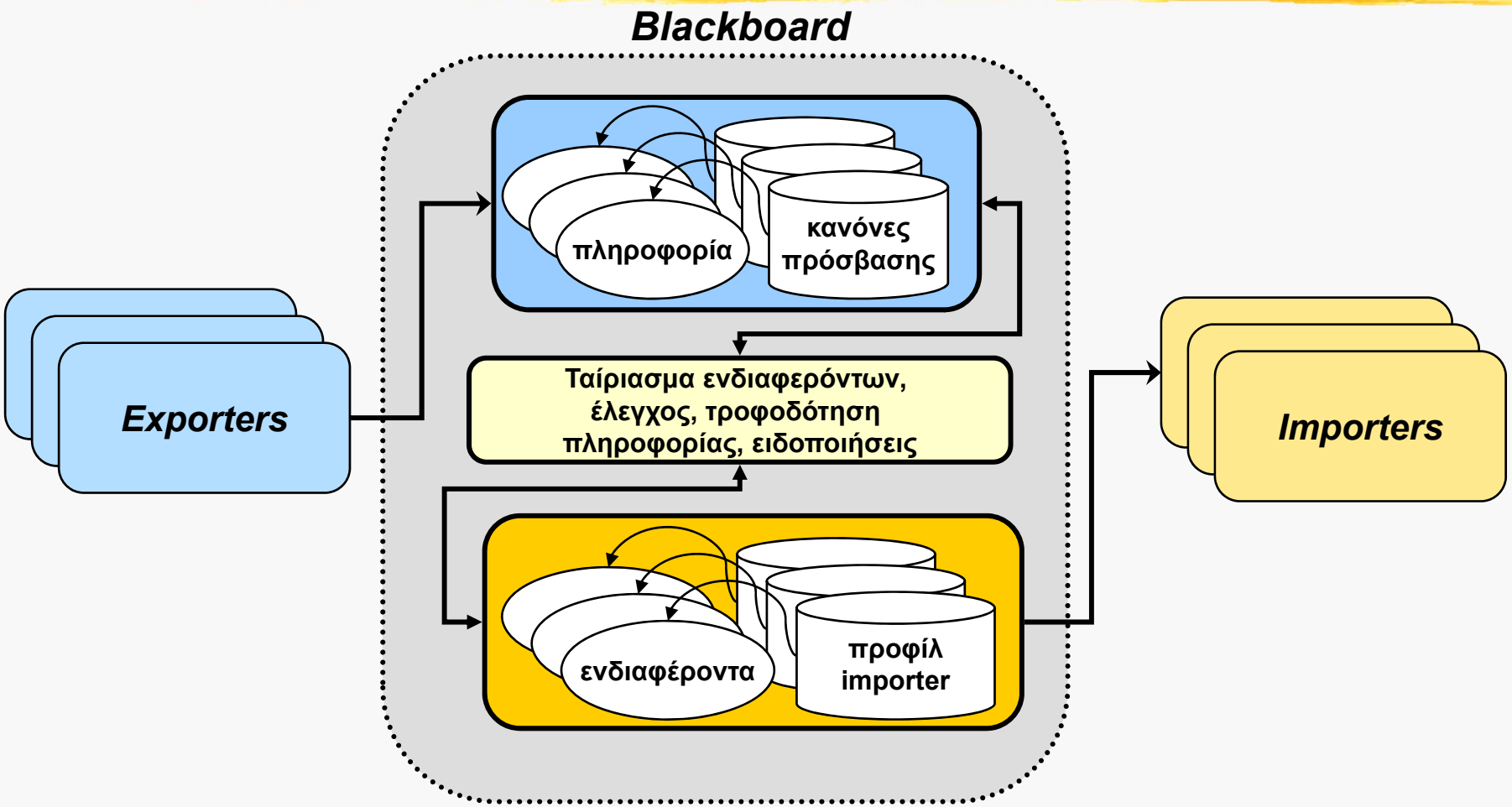
Blackboard (2/5)

- **Λύση**
  - Παροχή ειδικής κλάσης κοινού χώρου, blackboard, με δυνατότητα έκθεσης (export) και πρόσβασης δεδομένων (import) από ενδιαφερόμενα ανεξάρτητα τμήματα
  - Ορισμός ρόλων exporter και importer, ως τμήματα που πρέπει να γνωρίζουν τις λεπτομέρειες για encoding και decoding της πληροφορίας
  - Η κλάση blackboard παραχωρεί την απαραίτητη μνήμη για όσο χρόνο η εκτιθέμενη πληροφορία πρέπει να είναι ακόμη διαθέσιμη
  - Η blackboard κλάση μπορεί να ειδοποιεί τους exporters σε κάθε περίπτωση που η πληροφορία χρησιμοποιείται από κάποιον importer

Blackboard (3/5)

- ***Επιπτώσεις***
  - Οι exporters καθίστανται εντελώς ανεξάρτητοι, ως προς την υλοποίηση, από τους importers
  - Η κλάση blackboard μπορεί να ταιριάζει ενδιαφέροντα και να γεφυρώνει, με διαχωρισμό : client / server, consumer / producer, exporter / importer, etc
  - Σε περίπτωση διαχωρισμού των τμημάτων σε ανεξάρτητες διεργασίας, το σύστημα μπορεί να αλλάζει δυναμικά (δηλ. ενώ «τρέχει»)

Blackboard (4/5)



Blackboard (5/5)

```
class Blackboard {
   public:
   unsigned Export (  // Επιστρέφει ένα μοναδικό data stamp
       void*          data,
       unsigned       size,
       const char*    typeId,
       void           (*onImport)(unsigned dataStamp, void* closure),
       unsigned long expiresAfter,
       void           (*onExpire)(void* data, unsigned size)
   );
   void Remove (unsigned dataStamp);
   void SetImportInterest (
       const char*     typeId,
       void*           (*importFunc)(void* data, unsigned size)
   );
};
```

- **Η Export συνάρτηση είναι για την εξαγωγή πληροφορίας, δίνοντας δυνατότητας στον exporter να ειδοποιηθεί εάν κάποιος κάνει import τα δεδομένα, και να μπορεί να δεχθεί δεδομένα από τον importer (μέσω του closure).**
- **Η SetImportInterest είναι μία συνάρτηση εκδήλωσης ενδιαφέροντος, με παράμετρο μία import συνάρτηση, για συγκεκριμένη κατηγορία δεδομένων (encoded δυναμικά ως string). Η import συνάρτηση επιστρέφει δεδομένα που μπορεί να χρειάζεται ο Exporter.**