HY352 : ΤΕΧΝΟΛΟΓΙΑ ΛΟΓΙΣΜΙΚΟΥ

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΡΗΤΗΣ,
ΣΧΟΛΗ ΘΕΤΙΚΩΝ ΕΠΙΣΤΗΜΩΝ,
ΤΜΗΜΑ ΕΠΙΣΤΗΜΗΣ ΥΠΟΛΟΓΙΣΤΩΝ

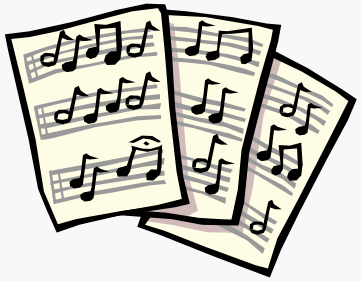
ΔΙΔΑΣΚΩΝ
Αντώνιος Σαββίδης

CSD

ΕΝΟΤΗΤΑ 5

ΣΧΕΔΙΑΣΤΙΚΑ ΠΡΟΤΥΠΑ

Αριθμός διαλέξεων 5 – Διάλεξη 2η



CSD

Περιεχόμενα

- *Iterator*
- Visitor
- Factory
- Prototype
- Singleton
- State

CSD

Iterator (1/7)

- ***Πρόβλημα***
  - Παρέχουμε σειριακούς τρόπους πρόσβασης στα περιεχόμενα σύνθετων συλλογών στοιχείων ***C*** (*container classes*) μη απαραίτητα γραμμικά δομημένων, χωρίς να εκτίθεται η εσωτερική τους αναπαράσταση
- ***Λύση*** **(μία από τις διάφορες που υπάρχουν)**
  - Παρέχουμε έναν αφηρημένο τύπο iterator (ADT) ο οποίος και υλοποιείται πλήρως μέσα στις κλάσεις ***C***, ο οποίος και προσφέρει όλες τις απαραίτητες συναρτήσεις για την πρόσβαση στα περιεχόμενα στοιχεία. Ο αφηρημένος τύπος iterator δεν πρέπει να εμπεριέχει συναρτήσεις οι οποίες βασίζονται σε κάποια συγκεκριμένη υλοποίηση των κλάσεων ***C***.
- ***Επιπτώσεις***
  - Διαφορετικοί αλγόριθμοι πρόσβασης από διαφορετικές κλάσεις iterator
  - Το API της κάθε ***C*** κλάσης απλουστεύεται
  - Πολλαπλές παράλληλες προσβάσεις είναι εφικτές (ένας iterator διατηρεί την κατάσταση πρόσβασης σε ειδικές τοπικές μεταβλητές)
  - Η κλάσεις ***C*** παράγουν / παρέχουν τα στιγμιότυπα / τύπους iterator

Iterator (2/7)

Iterator abstract class

Για το ρόλο των συναρτήσεων **Bwd**, **Fwd**, και **Get** συνήθως χρησιμοποιούνται υπερφορτωμένοι οι τελεστές **--**, **++** και το μοναδιαίο ***

Ο τύπος T του value είναι παράμετρος στο template

```
template <class T> class Iterator {
 public:
 virtual T&      Get (void) const = 0;
 virtual void    Fwd (void) = 0;
 virtual void    Bwd (void) = 0;
 virtual bool    MoreFwd (void) const = 0;
 virtual bool    MoreBwd (void) const = 0;
 virtual void    SetAtBegin (void) = 0;
 virtual void    SetAtEnd (void) = 0;
 virtual bool    operator==(const Iterator&) const = 0;
 virtual bool    operator!=(const Iterator&) const = 0;
 virtual void    operator=(const Iterator&) = 0;
 virtual bool    IsValid (void) const = 0;
};
```

Container abstract class

```
template <class T> class List {
  virtual const Iterator<T>&  Begin (void) const=0;
  virtual const Iterator<T>&  End (void) const=0;
  virtual void                Erase (const Iterator<T>&) =0;
};
```


Iterator (3/7)

Iterator concrete class

```
class Iterator : public ::Iterator<T> {
      Ο ειδικός αυτός iterator έχει εσωτερικές μεταβλητές
      για το στιγμιότυπο της συγκεκριμένης λίστας στην οποία
      αναφέρεται καθώς και μεταβλητές για την εκάστοτε θέση
      μέσα στη λίστα, ώστε να επιτρέπονται ανεξάρτητες
      προσβάσεις.
};
```

Container concrete class

```
template <class T>
class ListByArray : public List<T> {

      Εδώ υπάρχει η υλοποίηση της λίστας μέσω
      πίνακα.

      Εδώ μέσα τοποθετείται ο ορισμός
      της παραπάνω κλάσης iterator (associated type)
};
```


Iterator (4/7)

Τμήμα της υλοποίησης

Η συγκεκριμένη κλάση iterator θα πρέπει να έχει πρόσβαση στις εσωτερικές μεταβλητές της κλάσης που την περιέχει. Γι να γίνει αυτό θα πρέπει η κλάση iterator εσωτερικά να δηλωθεί ως friend της περιέχουσας κλάσης.

```
class Iterator : public ::Iterator<T> {
 private:
   ListByArray* myList;
   unsigned     currItem;
 public:
   Iterator (ListByArray* _myList = (ListByArray*) 0) :
     myList(_myList), currItem(_myList && _myList->Total() ? 0 : -1) {}
   bool IsValid (void) const
     { return myList && currItem >= 0 && currItem < myList->Total(); }
   void Fwd (void) { ++currItem; }
   void Bwd (void) { --currItem; }
   void SetAtBegin (void) { assert(IsValid()); currItem = 0; }
   void SetAtEnd (void) { assert(IsValid()); currItem = myList->Total()-1; }
   T& operator*(void) const
     { assert(IsValid()); return myList->Get(currItem); }
};
```


Iterator (5/7)

Παράδειγμα

1ος τρόπος πρόσβασης στα στοιχεία της λίστας με ανακύκλωση. Έστω ότι έχουμε στοιχεία τύπου Object (π.χ. int, string, κλπ).

```
ListByArray<Object> myList;
ListByArray::Iterator iter1(&myList);
iter1.SetBegin();
while (iter1.IsValid()) {
   Object& obj = *iter1;
   iter1.Fwd();
   obj.f(...);
}
```

2ος ευκολότερος τρόπος πρόσβασης στα στοιχεία της λίστας με ανακύκλωση

```
for (      ListByArray<Object>::iterator iter2  = myList.Begin();
           iter2 != myList.End();
           ++iter2    )
           (*iter2).f(...);
```

Iterator (6/7)

```
template <class Titerator, class Tfunctor>
void for_all (Titerator& start, const Titerator& end, const Tfunctor& fun) {
  for (; start != end; ++start)
       fun(*start);
}
```

- Πρέπει **Titerator** να είναι κλάση που κληρονομεί από **Iterator** ή κλάση που να υλοποιεί τους operators που εμφανίζονται στην **for_all** με την ίδια σημασιολογία.
- Πρέπει η **Tfunctor** να είναι ή συνάρτηση με υπογραφή **T1 (*)(T2&);** με **T1** τύπο επιστρεφόμενο που δεν χρησιμοποιείται, και **T2** ο τύπος των στοιχείων του container class για τον iterator **Titerator**, ή στιγμιότυπο κλάσης με υπερφορτωμένο τον τελεστή (), με υπογραφή **T1 operator()(T2&);**

```
ListByArray<int>     l;          // Έστω στοιχεία τύπου int
StackDynamic<string> s;          // Έστω στοιχεία τύπου std::string
void Print (int i) { printf("%d", i); }
struct StoreFunctor {
   FILE* fp;
   void operator()(const string& s) { fprintf(fp, "%s", s.c_str()); }
   StoreFunctor (FILE* _fp) : fp(_fp) {}
};
for_all(l.Begin(), l.End(), Print);
FILE* fp = fopen("output.txt", "wt");
for_all(s.begin(), s.End(), StoreFunctor(fp));
```

Εδώ έχουμε functor με local state

Πλέον μπορούμε να χρησιμοποιούμε ή range-based for loop ή lambda functions

Ένθετο

```
ListByArray<int>     nums;
StackDynamic<string> strs;

for (auto i : nums) printf("%d", i);  ← range-based for
std::for_each(
  nums.begin(),
  nums.end(),
  [](int i){ printf("%d", i); } ← lambda (anonymous) function
);

FILE* fp = fopen("output.txt", "wt");
for (auto& s : strs) fprintf(fp, "%s", s.c_str());  ← range-based for
std::for_each(
  strs.begin(),
  strs.end(),
  [](string& s){ fprintf(fp, "%s", s.c_str());  } ← lambda (anonymous) function
);
fclose(fp);
```


Iterator (7/7)

- *Περίληψη*
  - Εξειδικευμένη κλάση που παρέχει τις απαραίτητες συναρτήσεις επίσκεψης των περιεχομένων στοιχείων σε σύνθετους περιέκτες κρύβοντας της υλοποίησή τους
  - Υλοποιεί τον αφηρημένο τύπο για κάθε διαφορετική αναπαράσταση του περιέκτη
  - Η εσωτερική του υλοποίηση βασίζεται στις λεπτομέρειες υλοποίησης της εκάστοτε κλάσης περιέκτη
  - Περιέχει πληροφορία για το στιγμιότυπο περιέκτη στο οποίο αναφέρεται καθώς και για την κατάσταση (δηλ. εκάστοτε σημείο) πρόσβασης.

Περιεχόμενα

Visitor (1/5)

- ***Πρόβλημα***
  - Χρειάζεται να εφαρμόσουμε κάποιες λειτουργίες στα στοιχεία μίας συλλογής ή ενός σύνθετου αντικειμένου όταν τα στοιχεία του είναι διαφορετικών τύπων (αλλά γνωστών στην υλοποίηση του σύνθετου αντικειμένου)
- ***Λύση***
  - Η υλοποίηση της συλλογής παρέχει μία αφηρημένη κλάση ***Visitor*** με μεθόδους για την επίσκεψη κάθε διαφορετικού συστατικού στοιχείου της συλλογής (τα ονόματα των μεθόδων ταιριάζουν με τους τύπους των στοιχείων). Παρέχεται μία μέθοδος ***accept (Visitor*)*** από τη συλλογή.
  - Όλα τα στοιχεία κληρονομούν από έναν κοινό τύπο. Η σειρά επίσκεψης μπορεί να είναι καλά ορισμένη ή όχι αλλά πάντα τεκμηριώνεται τι ισχύει.
- ***Επιπτώσεις***
  - Ο client υλοποιεί μία κατάλληλη subclass του Visitor και καλεί την accept σε ένα κατάλληλο instance της συλλογής
  - Μπορούν να υλοποιηθούν όσες διαφορετικές κλάσεις από visitors επιθυμούμε.
  - Δεν απαιτείται η τροποποίηση της συλλογής εάν θέλουμε να εφαρμόσουμε κάποιες επιπλέον λειτουργίες στα συστατικά στοιχεία.

Visitor (2/5)

```
class Character {}
class Attacker      : public Character { };
class GateKeeper    : public Character { };
class Decoration    : public Character { };
class Helper        : public Character { };
class Trap          : public Character { };
class PowerUp       : public Character { };

class SceneVisitor {
public:
    virtual void VisitAttacker      (Attacker* p) = 0;
    virtual void VisitGateKeeper    (GateKeeper* p) = 0;
    virtual void VisitDecoration    (Decoration* p) = 0;
    virtual void VisitHelper        (Helper* p) = 0;
    virtual void VisitTrap          (Trap* p) = 0;
    virtual void VisitPowerUp       (PowerUp* p) = 0;
};

class Scene {
    public:
    void Accept (SceneVisitor* visitor) {
        for each character in the scene of type do
            invoke appropriate Visit<type> function
    }
};
```

Δεν είναι απαραίτητο εν γένει οι κλάσεις αυτές να σχετίζονται με inheritance.

Πάντα το όρισμα είναι συγκεκριμένου τύπου και όχι κάποιου superclass

Η Accept φροντίζει να καλεί τη σωστή Visit συνάρτηση του Visitor. Εάν επιτρέπεται αλλάζει η συλλογή θέλει ιδιαίτερη προσοχή.

Visitor (3/5)

```
class ReportVisitor : public SceneVisitor {
    std::string report;
    public:
    virtual void VisitAttacker      (Attacker* p)
        { report += <add the necessary description for Attacker p>; }
    virtual void VisitGateKeeper    (GateKeeper* p)
        { report += <add the necessary description for GateKeeper p>; }
    virtual void VisitDecoration    (Decoration* p)
        { report += <add the necessary description for Decoration p>; }
    virtual void VisitHelper        (Helper* p)
        { report += <add the necessary description for Helper p>; }
    virtual void VisitTrap          (Trap* p)
        { report += <add the necessary description for Trap p>; }
    virtual void VisitPowerUp       (PowerUp* p)
        { report += <add the necessary description for PowerUp p>; }
    const std::string GetReport (void) const
        { return report; }
};

const std::string GetSceneReport (Scene* scene) {
    ReportVisitor reportVisitor;
    scene->Accept(&reportVisitor);
    return reportVisitor.GetReport();
}
```

Ένας concrete visitor μπορεί να έχει local data τα οποία δημιουργούνται κατά τη διάρκεια της επίσκεψης.

Χρήση ενός συγκεκριμένου visitor. Φαίνεται πόσο απλουστεύει τα πράγματα.

Visitor (4/5)

```
class FindClosestCharacterVisitor : public SceneVisitor {
    Location    target;
    Distance    dist;
    Charcter*   closest;
    void CheckDistance (Character* p) {
        Distance d = GetDistance(p->GetLocation(), target);
        if (d < dist) dist = d, closest = p;
    }
    public:
    virtual void VisitAttacker      (Attacker* p)
        { CheckDistance(p); }
    virtual void VisitGateKeeper    (GateKeeper* p)
        { CheckDistance(p); }
    virtual void VisitDecoration    (Decoration* p)
        { CheckDistance(p); }
    virtual void VisitHelper        (Helper* p)
        { CheckDistance(p); }
    virtual void VisitTrap          (Trap* p)
        { CheckDistance(p); }
    virtual void VisitPowerUp       (PowerUp* p)
        { CheckDistance(p); }
    Character* GetResult (void) const { return closest; }
    FindClosestCharacterVisitor (const Location& p) : target(p), closest(NULL){}
};

Character* GetClosestCharacter (Scene* scene, const Location& p) {
    FindClosestCharacterVisitor findVisitor;
    scene->Accept(&findVisitor);
    return findVisitor.GetResult();
}
```

Σε αυτό το παράδειγμα ο visitor έχει σκοπό της αναζήτηση ενός στοιχείου με κάποια καθολική ιδιότητα.

Ωστόσο, ενδέχεται να θέλουμε να βρούμε το πρώτο στοιχείο που έχει μία ιδιότητα και έπειτα να τερματίσει η επίσκεψη.

Visitor (5/5)

```
// TECHNIQUE-1
class SceneVisitor {
public:
    virtual bool VisitAttacker      (Attacker* p) = 0;
    virtual bool VisitGateKeeper    (GateKeeper* p) = 0;
    ... rest as before...
};

class Scene {
    public:
    void Accept (SceneVisitor* visitor) {
        for each character x in the scene of type do {
            if (!Visit<type>(x))
                return; // Break the visit
        }
    }
};
```

Οι Visit methods επιστρέφουν bool με false να σημαίνει τερματισμό της επίσκεψης και true συνέχεια.

```
// TECHNIQUE-2
class SceneVisitor {
    bool stopped;
public:
    void    Stop (void) { stopped = true; }
    bool    IsStopped (void) const { return stopped; }
    ... rest as before...
};

class Scene {
    public:
    void Accept (SceneVisitor* visitor) {
        for each character x in the scene of type do {
            Visit<type>(x);
            if (visitor->IsStopped())
                return; // Break the visit
        }
    }
};
```

Ο Visitor (superclass) έχει ένα γνώρισμα για το εάν έχει τερματίσει η επίσκεψη.

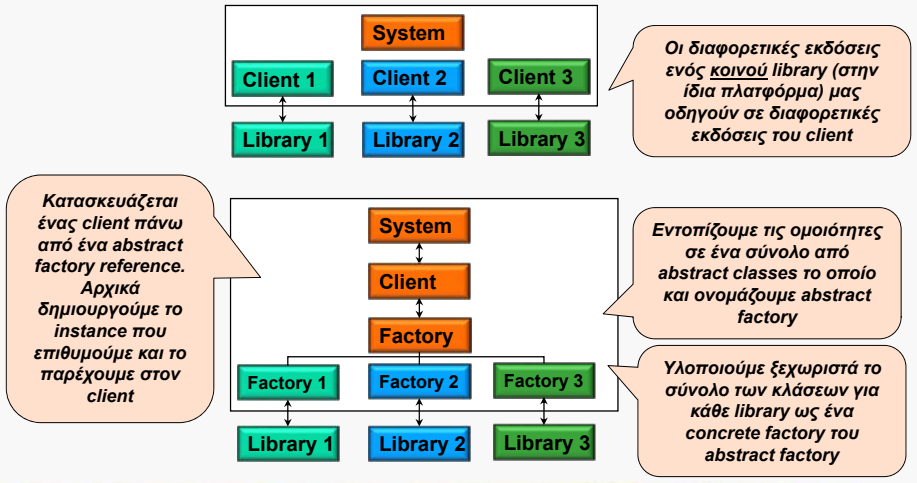
Περιεχόμενα

- Iterator
- Visitor
- *Factory*
- Prototype
- Singleton
- State

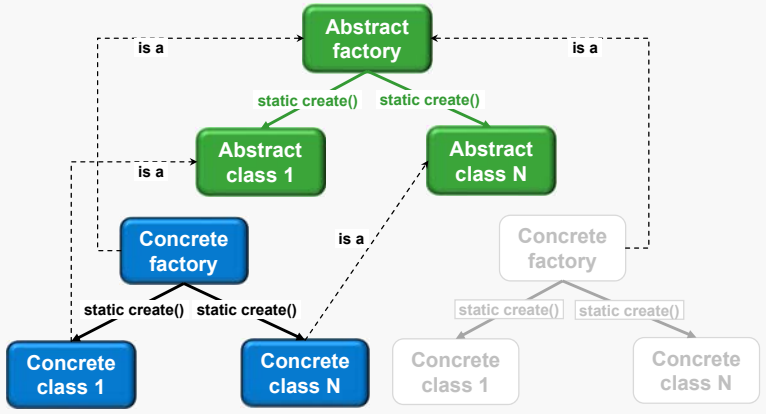
Factory (1/7)

- ***Πρόβλημα***
  - Το σύστημά μας κατασκευάζεται πάνω από εναλλακτικές παρόμοιες βιβλιοθήκες μέσω των οποίων δημιουργεί στιγμιότυπα διαφορετικών κλάσεων. Θέλουμε να μην εμφανίζεται στον κώδικα εξάρτηση από κάποια τέτοια οικογένεια κλάσεων, με δυνατότητα χρήσης όποιας επιθυμούμε σε διαφορετικές εκδόσεις του συστήματος.
- ***Λύση***
  - Ενοποίησε τις διάφορες κλάσεις της κάθε οικογένειας κάτω από μία οικογένεια αφηρημένων κλάσεων, έπειτα όρισε ένα αφηρημένο εργοστάσιο (factory) στιγμιότυπων, και έπειτα υλοποίησε τα εξειδικευμένα ανά οικογένεια factories.
- ***Επιπτώσεις***
  - Ο κώδικας χρήσης μπορεί να διαλέγει μεταξύ των εναλλακτικών factories, καθιστώντας τον εφαρμόσιμο σε διαφορετικές οικογένειες κλάσεων απ' ευθείας.
  - Μπορούν να επεκταθούν οι οικογένειες χωρίς να επηρεάζεται ο αρχικός κώδικας.

Factory (2/7)



Factory (3/7)



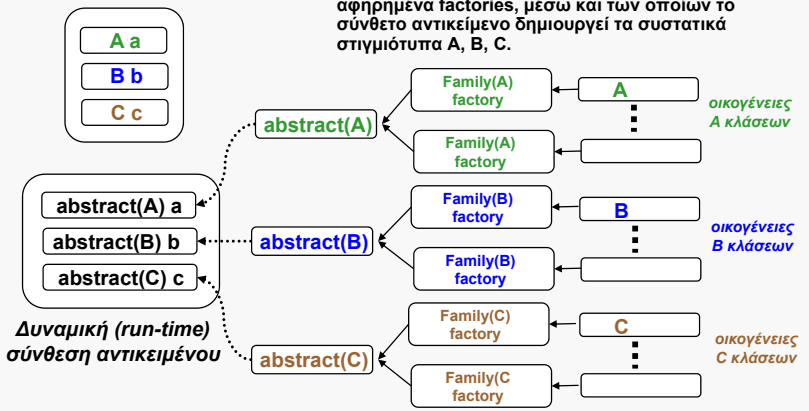
Factory (4/7)

- Σε περίπτωση που το factory pattern πρόκειται να εφαρμοστεί σε υπάρχουσες οικογένειες συγκεκριμένων κλάσεων, τότε απαιτείται:
  - είτε οι κλάσεις αυτές να τροποποιηθούν (εάν είναι δυνατόν), αφού πρέπει να κληρονομούν από τις αφηρημένες κλάσεις που θα ορίσουμε
  - ή ειδικές κλάσεις προσαρμογής (wrappers) πρέπει να οριστούν, πάνω στις αυθεντικές κλάσεις
  - ➔ Και στις δύο περιπτώσεις, πρέπει να γνωρίζουμε την ανάγκη πολύ προσεκτικής σχεδίασης της αφηρημένης οικογένειας ώστε να συνιστά επιτυχή αφαίρεση πάνω στις συγκεκριμένες οικογένειες
  - ➔ Το factory pattern μπορεί να εφαρμοστεί για την υλοποίηση σύνθετων αντικειμένων που μπορούν να περιέχουν στιγμιότυπα από διαφορετικές κάθε φορά οικογένειες.

Factory (5/7)

Δύσκαμπτο (compile-time) σύνθετο αντικείμενο

- Παρέχουμε τα factories για τις A, B, και C ως παραμέτρους στον constructor του σύνθετου αντικειμένου, τύπου αναφορών / δεικτών σε αφηρημένα factories, μέσω και των οποίων το σύνθετο αντικείμενο δημιουργεί τα συστατικά στιγμιότυπα A, B, C.



Δυναμική (run-time) σύνθεση αντικειμένου

Factory (6/7)

```
class Window {...};
class Button {...};
```
Αφηρημένη οικογένεια γραφικών αντικειμένων

```
class MOTIFWindow : public Window {...};
class MFCButton   : public Button {...};
```
Window στην οικογένεια MOTIF
Button στην οικογένεια MFC

```
class GUIFactory {
   public:
   virtual Window* CreateWindow (...) = 0;
   virtual Button* CreateButton (...) = 0;
};
```
Αφηρημένο factory γραφικών αντικειμένων

```
class MOTIFFactory : public GUIFactory {
   Window* CreateWindow (...);
   Button* CreateButton (...);
};
```
MOTIF factory γραφικών αντικειμένων

```
class MFCFactory : public GUIFactory {
   Window* CreateWindow (...);
   Button* CreateButton (...);
};
```
MFC factory γραφικών αντικειμένων

Factory (7/7)

```
GUIFactory* guiFactory = GUIFactory::Get();
Window* saveFile = guiFactory->CreateWindow(...);
...
Button* ok = guiFactory->CreateButton(saveFile,...);
ok->AddCallback(GUI_ButtonPressed, SaveFile);
```

- Η υλοποίηση του factory απαιτεί πολύ καλή μοντελοποίηση των κοινών χαρακτηριστικών των αντιστοίχων κλάσεων των διαφορετικών οικογενειών, σε αφηρημένες κλάσεις της αφηρημένης οικογένειας
- Οι αφηρημένες κλάσεις δεν είναι ουσιαστικά αφαιρέσεις, αλλά γενικεύσεις, γιατί υποστηρίζουν όλα τα μέλη των αντιστοίχων κλάσεων, με γενικευμένα αναγνωριστικά, υπογραφές και τύπους
- Πολλές φορές, οι εναλλακτικές διαφορετικές οικογένειες δεν χρειάζεται ή δεν μπορούν να βρίσκονται στο τελικό σύστημα, αλλά έχουμε διαφορετικές εκδόσεις του συστήματος με κάθε μία από αυτές (π.χ. MOTIF / MFC versions).

Περιεχόμενα

- Iterator
- Visitor
- Factory
- *Prototype*
- Singleton
- State

Prototype (1/2)

- ***Πρόβλημα***
  - Χρειάζεται να δημιουργήσουμε ακριβή αντίγραφα στιγμιότυπων κάποιας συγκεκριμένης κατάστασης, παρά να δημιουργούμε στιγμιότυπα και να τα φέρουμε στην επιθυμητή κατάσταση με κλήσεις μελών. Η κατάσταση αυτή μπορεί να μην είναι πάντα γνωστή κατά την κατασκευή του συστήματος (compile-time), αλλά να αποφασίζεται αλγοριθμικά κατά την λειτουργία (run-time).
- ***Λύση***
  - Οι αντίστοιχες κλάσεις παρέχουν έναν αντιγραφέα (π.χ. Clone()). Τα πρωτότυπα είναι στιγμιότυπα είτε της αυθεντικής κλάσης, ή ειδικά κατασκευασμένης κληρονόμου, εάν η αυθεντική κλάση δεν περιέχει αντιγραφέα και επίσης είναι αδύνατο να τροποποιηθεί.
- ***Επιπτώσεις***
  - Τα πρωτότυπα μας σώνουν από αρκετό κώδικα, ειδικά εάν η προσέγγιση της επιθυμητής κατάστασης στιγμιότυπου απαιτεί αρκετές και πολύπλοκες κλήσεις. Επίσης, ελαφρύνεται ο προγραμματιστής από την απομνημόνευση όλων αυτών των μελών που θα εμπλέκονταν μόνο σε τέτοιες κλήσεις.

Prototype (2/2)

```
class Person {
   public:
   bool    Decode (FILE* readProfile);
   void    Encode (FILE* writeProfile) const;
   Person* Clone (void) const;
   Person (const char* path) {
      FILE* fp = fopen(path, "r"); assert(fp);
      bool result = Decode(fp);    assert(result);
      fclose(fp);
   }
};
```

serializer

replicator

decoder constructor

- **Τα A και B είναι αρχεία με σωμένα πρωτότυπα τα οποία φορτώνονται (μέσω του decoder constructor) και έπειτα, μέσω του αντιγραφέα, δημιουργούνται όλα τα επιθυμητά δυναμικά αντίγραφα.**
- **Τα στιγμιότυπα – αντίγραφα, συνήθως μετά την δημιουργία τους δεν είναι απαραίτητο να διατηρούνται ως πανομοιότυπα – απλά θεωρούνται ως πολύπλοκες εναλλακτικές default καταστάσεις στη δημιουργία στιγμιότυπων.**

```
Person a("A_profile.bin", b("B_profile.bin");
Person *john = a.Clone(), *jim = b. Clone(), *george = b.Clone();
```

Περιεχόμενα

- Iterator
- Visitor
- Factory
- Prototype
- *Singleton*
- State

Singleton (1/4)

- ***Πρόβλημα***
  - Θέλουμε να επιβάλουμε την ύπαρξη ενός μοναδικού στιγμιότυπου μίας κλάσης, το οποίο είναι πάντα διαθέσιμο όταν το χρειάζεται το πρόγραμμά μας.
- ***Λύσεις***
  - Όρισε την κλάση με private constructor και με ένα τοπικό private / protected static στιγμιότυπο της ίδιας της κλάσης.
  - Κάνε μία lightweight κλάση μόνο με static μέλη και private static τοπικά δεδομένα, και με ειδικές *Initialise()* και *CleanUp()* συναρτήσεις
- ***Επιπτώσεις***
  - Η πρώτη λύση δεν έχει μεν καλό στυλ κλήσεων, αλλά επιτρέπει κληρονομικότητα.
  - Η δεύτερη λύση έχει πολύ καλό στυλ κλήσεων,αλλά δεν επιτρέπει κληρονομικότητα.

Singleton (2/4)

Style 1

```
class MemoryManager final {
  private:
  1: static MemoryManager singleton;  // Προσοχή: static instance
  2: static MemoryManager* singleton; // Εναλλακτικά, πολύ καλύτερα
  MemoryManager (void);
  ~MemoryManager();

  public:
  1: static MemoryManager& Singleton (void) { assert(singleton); return *singleton; }
  1: static MemoryManager* Singleton (void) { assert(singleton); return singleton; }
  2: static void Initialise (void) { assert(!singleton); singleton = new MemoryManager; }
  3: static void CleanUp (void)    { assert(singleton); delete singleton; singleton = 0; }
  void*        malloc (unsigned int size);
  void*        free (void* block);
  unsigned int memleft (void) const;
  bool         isvalidblock (void* block) const;
  unisgned int size (void* block) const;
  void         freeall (void);
  void         wipeout (void* block, unsigned char c) const;
};
```

Κατασκευή ειδικού διαχειριστή μνήμης για καλύτερο έλεγχο ορθότητας δεικτών.

Singleton (3/4)

```
void* operator new (size_t size) {
   assert(MemoryManager::Singleton().memleft() >= size);
   return MemoryManager::Singleton().malloc(size);
}
void operator delete (void* block) {
   assert(MemoryManager::Singleton().isvalidblock(block));
   MemoryManager::Singleton().wipeout(block, "ώ");
   MemoryManager::Singleton().free(block);
}

template <class T>
T* dverifyptr (T* ptr) {
   assert(MemoryManager::Singleton().isvalidblock(ptr));
   assert(MemoryManager::Singleton().size(ptr) >= sizeof(T));
   return ptr;
}
int* ip; *dverifyptr(ip) = 10;
```

CSD

Singleton (4/4)

Όλα τα δεδομένα γίνονται private static **Style 2**

```
class MemoryManager final {
  static void*        malloc (unsigned int size);
  static void*        free (void* block);
  static unsigned int memleft (void) const;
  static bool         isvalidblock (void* block) const;
  static unisgned int size (void* block) const;
  static void         freeall (void);
  static void         wipeout (void* block, unsigned char c) const;

  static void           Initialise (void);
  static void           CleanUp (void);
};

MemoryManager::Initialise();
...
C* c = MemoryManager::malloc(sizeof(C));
...
MemoryManager::CleanUp();
```


CSD

Περιεχόμενα

- Iterator
- Visitor
- Factory
- Prototype
- Singleton
- *State*

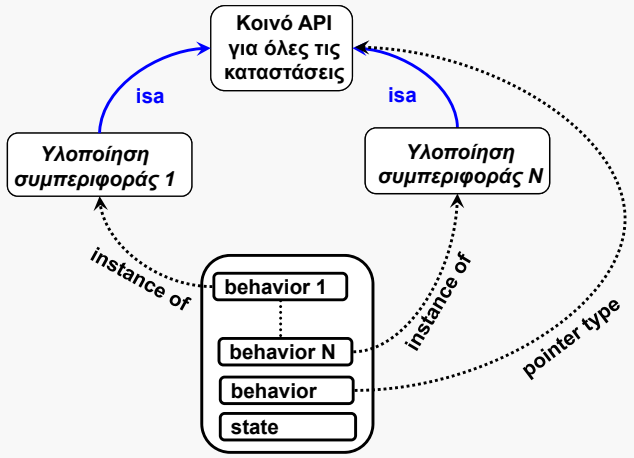
CSD

State (1/7)

- ***Πρόβλημα***
  - Τα στιγμιότυπα πρέπει να αλλάζουν δραστικά την συμπεριφορά τους, χωρίς ωστόσο να σημαίνει αυτό αλλαγή του API, ανάλογα με διαφορετικές τιμές των μεταβλητών κατάστασης, πρακτικά απαιτώντας λειτουργικές διαφοροποιήσεις οι ο οποίες δεν ταιριάζουν καλά μέσα στην ίδια την κλάση
- ***Λύση***
  - Ενσωμάτωσε τις λειτουργικές διαφορές σε εναλλακτικές κλάσεις, όλες ως κληρονόμους του ίδιου αφηρημένου API. Δημιούργησε τοπικά αντίστοιχα στιγμιότυπα, και δήλωσε ένα δείκτη στο αφηρημένο API. Όταν η κατάσταση αλλάζει, ο δείκτης αυτός εκχωρείται τη διεύθυνση του αντίστοιχου στιγμιότυπου. Η αρχική κλάση εφαρμόζει την κλήση των μελών μόνο μέσω του δείκτη στο αφηρημένο API (late binding).
- ***Επιπτώσεις***
  - Πρέπει να οριστούν οι αντίστοιχες κλάσεις ανά κατάσταση, οποίες και περιέχουν όλα τα δεδομένα σχετικά με την κατάσταση. Η ταχύτητα είναι καλύτερη και πάντα σταθερή (late binding), ενώ η επέκταση των καταστάσεων δεν αλλάζει τον αρχικό κώδικα (λίγες γραμμές μόνο).

CSD

State (2/7)



CSD

State (3/7)

Παράδειγμα (1/3)

```
enum SoldierState {
  Attacking =  0,
  Defending =  1,
  Patrolling = 2,
  Retreating = 3
};
#define MAX_BEHAVIORS 4
class Soldier : public Agent {
  private:
  SoldierState state;
  Behavior*    currBehavior;

  public:
  void Act (Command cmmd)
         { currBehavior->Act(cmmd); }
  void SetState (SoldierState newState);
};
```


CSD

State (4/7)

Παράδειγμα (2/3)

```
class Behavior {
   public:
   virtual void Act (Command cmmd) = 0;
};

class AttackingBehavior  : public Behavior {...};

class DefendingBehavior  : public Behavior {...};

class PatrollingBehavior : public Behavior {...};

class RetreatingBehavior : public Behavior {...};
```


CSD

State (5/7)

Παράδειγμα (3/3)

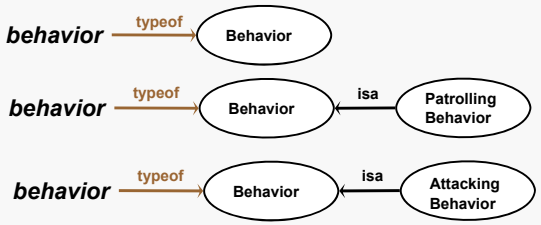
```
class SoldierAgent {
  private:
    Behavior* behaviors[MAX_BEHAVIORS];
  public:
    void SetState (SoldierState newState)
       { currBehavior = behaviors[(unsigned) state = newState]; }
    SoldierAgent(...) {
       behaviors[(unsigned) Attacking]  = new AttackingBehavior;
       behaviors[(unsigned) Defending]  = new DefendingBehavior;
       behaviors[(unsigned) Patrolling] = new PatrollingBehavior;
       behaviors[(unsigned) Retreating] = new RetreatingBehavior;
       SetState(Patrolling];
    }
   ~SoldierAgent()
      { Δεν ξεχνάμε να κάνουμε delete τα στιγμιότυπα  συμπεριφοράς }
};
```


CSD

State (6/7)

- Το state pattern υποδεικνύει ένα πολύ ενδιαφέρον χαρακτηριστικό που περνά απαρατήρητο
  - το *behaviour* variable αρχικά είναι reference σε *PatrollingBehaviour* instance
  - έπειτα μπορεί να έχω κάποιο άλλο τύπο π.χ. *AttackingBehaviour*
- Αυτό δείχνει ότι θα θέλαμε να έχουμε μία μεταβλητή της οποίας ο τύπος να μεταβάλλεται κατά την εκτέλεση
  - πάντα όμως σε ένα σύνολο derived κλάσεων από ένα κοινό super class
  - αυτό μοιάζει με την επόμενη εικόνα που αποτυπώνει την ανάγκη δυναμικής κληρονομικότητας

State (7/7)




```
Behavior behavior;
behavior inherit PatrollingBehaviour;
...
Behaviour uninherit PatrollingBehaviour;
behavior inherit AttackingBehaviour;
...
Behaviour uninherit AttackingBehaviour;
behavior inherit DefendingBehaviour;
```

Υπάρχουν γλώσσες οι οποίες υποστηρίζουν δυναμική κληρονομικότητα, δηλ. ο τύπος ενός αντικειμένου να μεταβάλλεται κατά την εκτέλεση μέσω δυναμικής κληρονομικότητας (π.χ. Self, Delta)