Πρότυπα Σχεδίασης

Design Patterns

Bridge (Γέφυρα)

- Κατηγορία: Structural
- Σκοπός: *Η αποσύνδεση μιας αφαίρεσης από την υλοποίησή της, ώστε να μπορούν να μεταβάλλονται ανεξάρτητα .*
- Συνώνυμα: Handle/Body

Bridge (Γέφυρα)

• Όταν μία αφαίρεση μπορεί να έχει περισσότερες από μία υλοποιήσεις, ο συνήθης τρόπος οργάνωσης είναι με τη χρήση κληρονομικότητας.

• Με τον όρο αφαίρεση νοείται μία αφηρημένη κλάση που ορίζει μία διασύνδεση (ένα σύνολο υπογραφών), ενώ υλοποιήσεις είναι οι συγκεκριμένες παράγωγες κλάσεις οι οποίες υλοποιούν τις μεθόδους της αφηρημένης κλάσεις.

• Πρόβλημα: Αφαίρεση και υλοποιήσεις συνδέονται μόνιμα, καθιστώντας δύσκολη την επέκταση, τροποποίηση και επαναχρησιμοποίηση αφαιρέσεων και υλοποιήσεων ανεξάρτητα.

Bridge (Γέφυρα)

Το πρότυπο "Γέφυρα" είναι σχετικά δύσκολο στην κατανόησή του· Χρησιμοποιείται ωστόσο σε πληθώρα περιπτώσεων όπου εντοπίζονται:

- Μεταβολές στην αφαίρεση μιας έννοιας
- Μεταβολές στον τρόπο υλοποίησης της έννοιας αυτής

Η Γέφυρα αντίκειται επίσης στη συνήθη τάση χειρισμού αντίστοιχων καταστάσεων μόνο με κληρονομικότητα. Ικανοποιεί όμως δύο από τους βασικούς κανόνες της αντικειμενοστραφούς κοινότητας: "Εντοπίστε αυτό που μεταβάλλεται και ενσωματώστε το", και "προτιμήστε σύνθεση αντικειμένων από κληρονομικότητα κλάσεων".

Bridge (Γέφυρα)

Θεωρούμε μία εφαρμογή η οποία χειρίζεται τις πληρωμές τεχνικών υπαλλήλων (TechEmployees).

Η πληρωμή των υπαλλήλων είναι δυνατόν να πραγματοποιείται

είτε με βάση το μηνιαίο μισθό,

είτε με βάση τον αριθμό ωρών εργασίας ανά μήνα και την αντίστοιχη αμοιβή ανά ώρα.

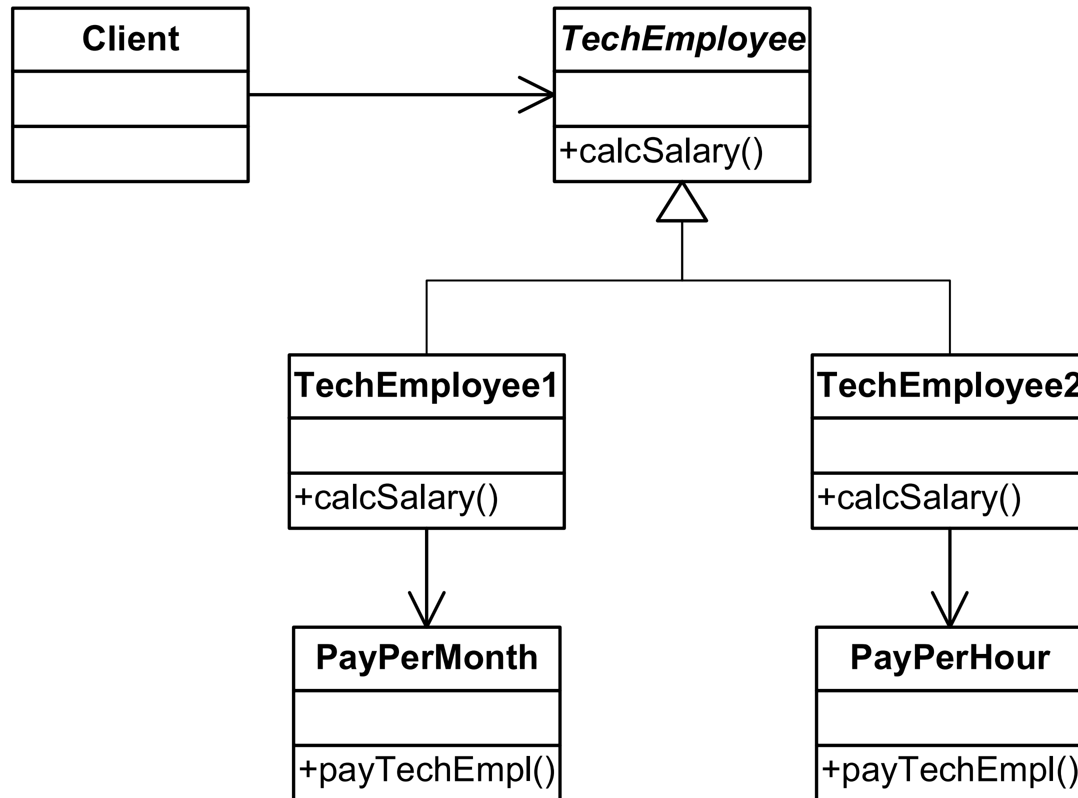
Ο υπολογισμός του μισθού, κρατήσεων κλπ πραγματοποιείται από δύο ανεξάρτητα προγράμματα, που αντιστοιχούν στις κλάσεις Pay1 και Pay2.

Bridge (Γέφυρα)

Η εφαρμογή θα πρέπει να είναι σε θέση να λειτουργήσει με υπάλληλο οποιασδήποτε κατηγορίας πληρωμής, αλλά δεν οφείλει να γνωρίζει εκ των προτέρων τον τρόπο πληρωμής κάθε υπαλλήλου.

Οι υπάλληλοι θα σχετίζονται με συγκεκριμένο τρόπο πληρωμής κατά την υλοποίηση των αντικειμένων τους, και κατά συνέπεια είναι λογικό η εφαρμογή να διατηρεί έναν δείκτη προς μία αφηρημένη κλάση *Τεχνικός Υπάλληλος* από την οποία θα κληρονομούν οι δύο συγκεκριμένες κατηγορίες με βάση τον τρόπο πληρωμής. Η κάθε κατηγορία διατηρεί δείκτη προς την αντίστοιχη κλάση υπολογισμού του μισθού.

Bridge (Γέφυρα)

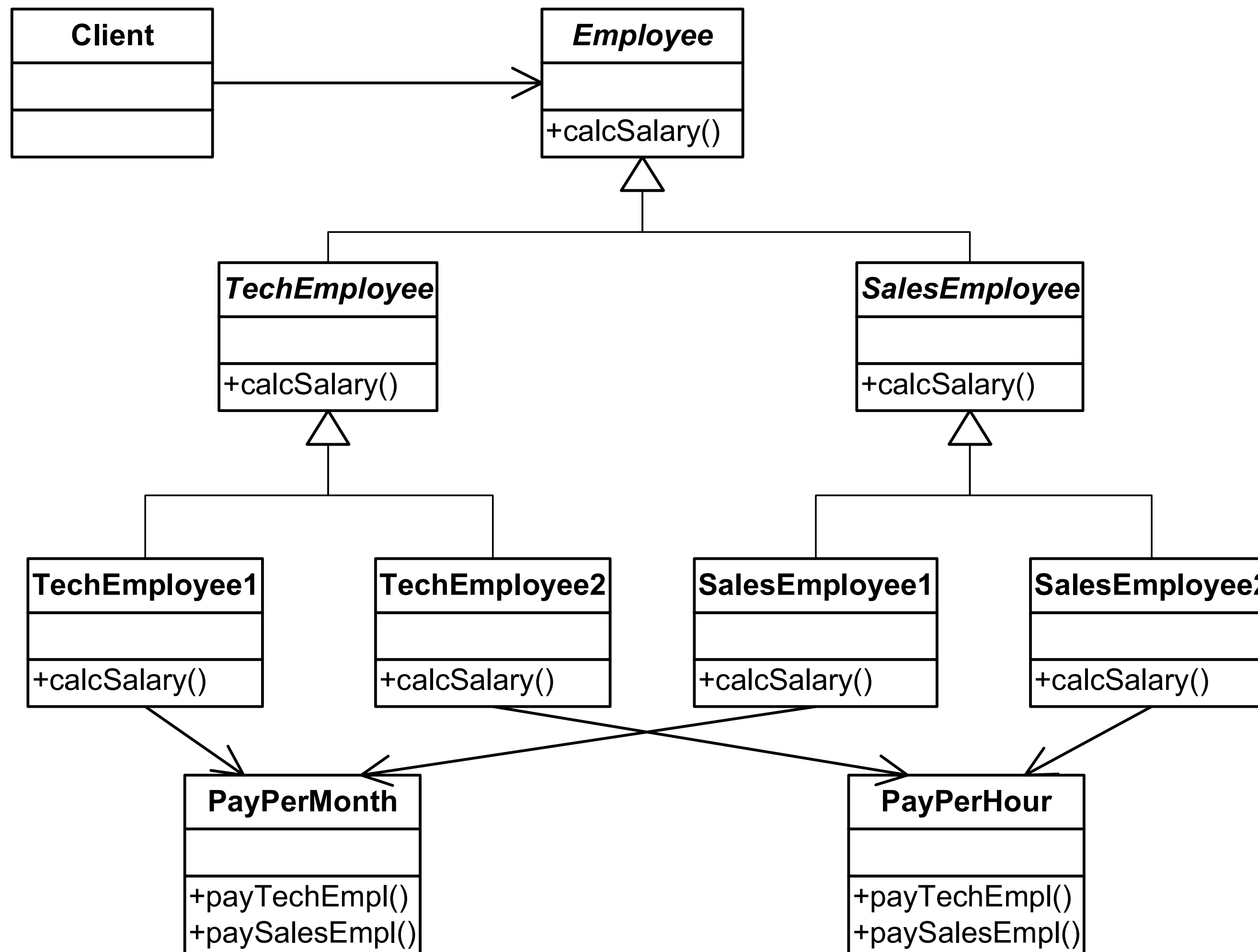


Bridge (Γέφυρα)

Η τροποποίηση των απαιτήσεων είναι αναπόφευκτη: Ζητείται ο εμπλουτισμός της εφαρμογής, ώστε να μπορεί να χειρίζεται και την πληρωμή πωλητών (SalesEmployees), οι οποίοι είναι επίσης δυνατόν να πληρώνονται με οποιονδήποτε από τους δύο τρόπους.

Η προφανής αντιμετώπιση του προβλήματος από έναν αναλυτή αντικειμενοστραφών συστημάτων είναι η χρήση κληρονομικότητας. Ορίζοντας μία επιπλέον αφηρημένη κλάση, τύπου Υπάλληλος, είναι δυνατόν να κληρονομούν οι δύο ειδικές κατηγορίες υπαλλήλων, ώστε η εφαρμογή να συνεχίσει να μπορεί να χειρίζεται οποιονδήποτε υπάλληλο. Η ανάλυση αυτή, οδηγεί στο διάγραμμα κλάσεων του σχήματος

Bridge (Γέφυρα)



Bridge (Γέφυρα)

Πολυπλοκότητα του σχεδίου:

Αν υπάρχει μεταβολή στην υλοποίηση, όπως για παράδειγμα αν προστεθεί ένας επιπλέον τρόπος πληρωμής (π.χ. ανά τεμάχιο), ο αριθμός των συγκεκριμένων κλάσεων που αναπαριστούν κατηγορίες υπαλλήλων/τρόπων πληρωμής, θα αυξηθεί σε έξι.

Αν μεταβληθεί η αφαίρεση, όπως για παράδειγμα αν προστεθεί μία επιπλέον κατηγορία υπαλλήλου, ο αριθμός των κλάσεων επίσης θα αυξηθεί σημαντικά.

Η εκρηκτική αυτή αύξηση του αριθμού των κλάσεων στο σύστημα, προκαλείται διότι η αφαίρεση (οι κατηγορίες των Υπαλλήλων) και οι υλοποιήσεις (οι διαφορετικοί τρόποι πληρωμών) έχουν υψηλή σύζευξη. Κάθε τύπος Υπαλλήλου πρέπει να γνωρίζει την κλάση που υπολογίζει την πληρωμή του.

Ο στόχος του προτύπου σχεδίασης "Γέφυρα" είναι ο διαχωρισμός των μεταβολών στην αφαίρεση από τις μεταβολές στην υλοποίηση, ώστε ο αριθμός των κλάσεων να αυξάνει γραμμικά.

Bridge (Γέφυρα)

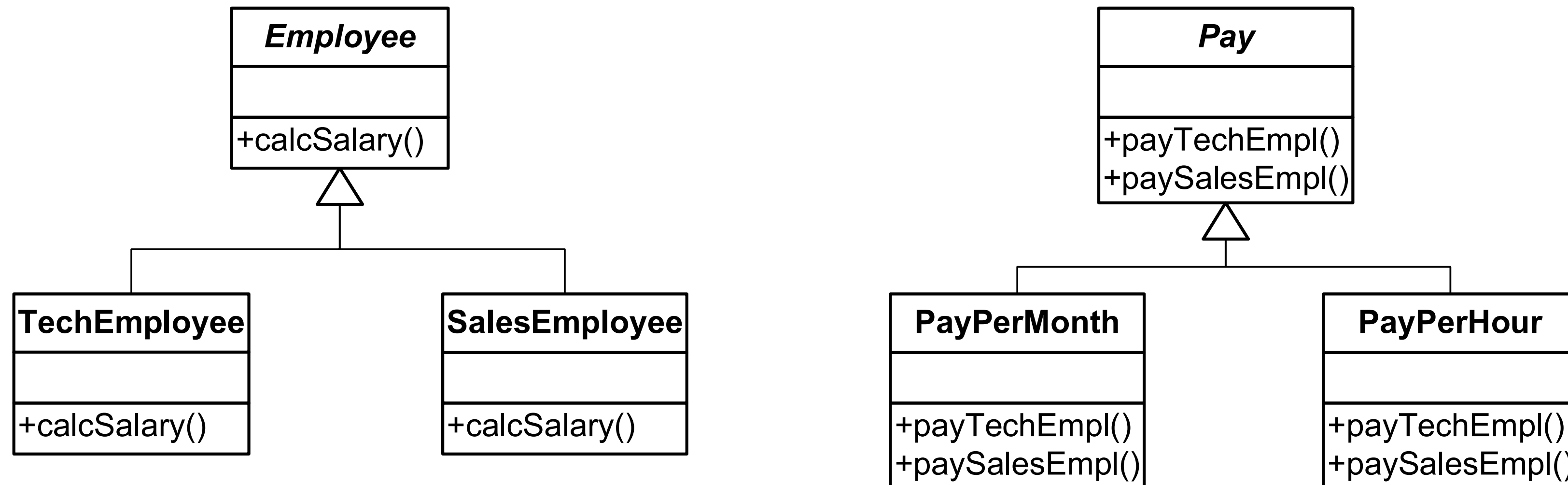
Στρατηγική:

• εντοπισμός των εννοιών που μεταβάλλονται και ενσωμάτωσή τους

• επιλογή σύνθεσης στη θέση της κληρονομικότητας.

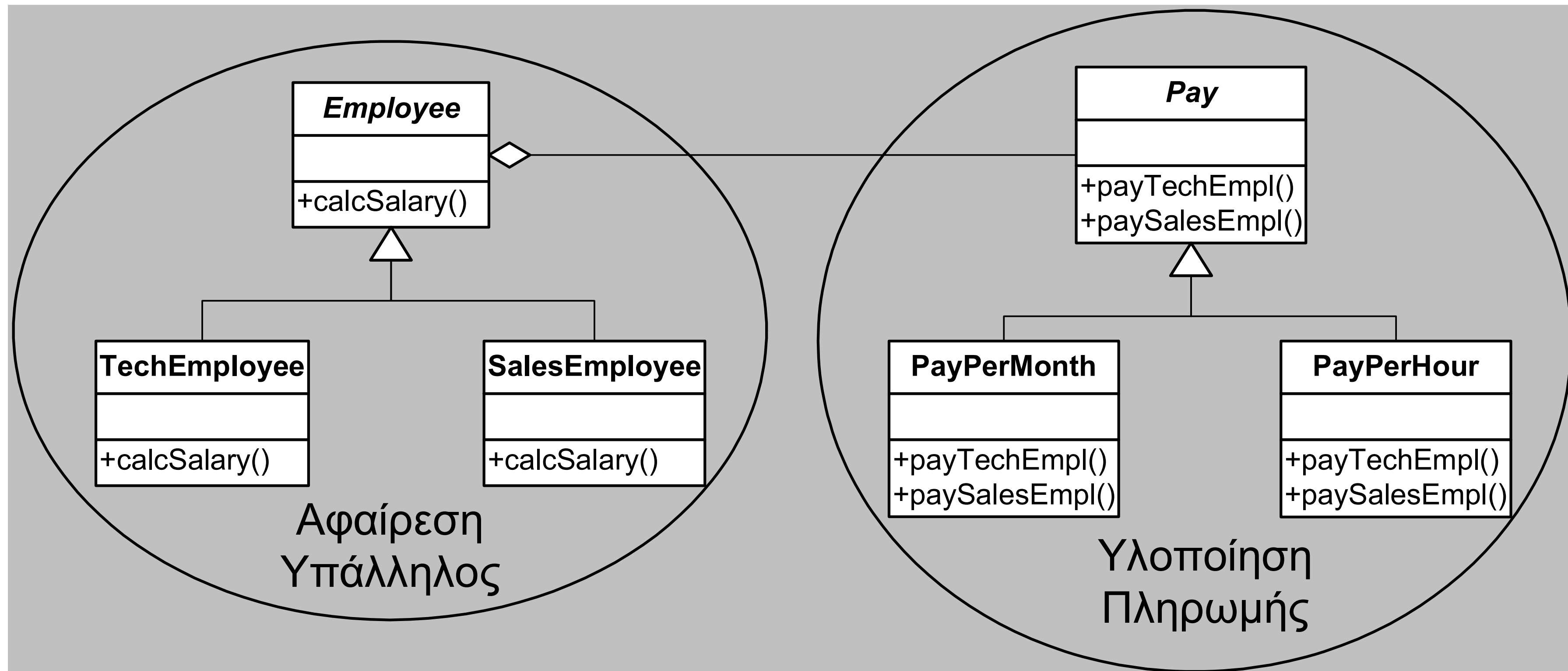
Στη συγκεκριμένη εφαρμογή μεταβάλλονται οι έννοιες του Υπαλλήλου (*Employee*) και του Τρόπου Πληρωμής (*Pay*). Η ενσωμάτωση των εννοιών που μεταβάλλονται επιτυγχάνεται με τη χρήση αφηρημένων κλάσεων για κάθε έννοια. Οι διάφορες έννοιες πλέον αναπαριστώνται ως παράγωγες κλάσεις αυτών των αφηρημένων κλάσεων.

Bridge (Γέφυρα)



Το ερώτημα που τίθεται είναι με ποιο τρόπο οι δύο αυτές ομάδες κλάσεων θα συσχετιστούν μεταξύ τους.
Η κληρονομικότητα ωστόσο πρέπει να αποφευχθεί διότι η εισαγωγή ενός επιπλέον επιπέδου δημιουργεί τα προβλήματα που συζητήθηκαν προηγουμένως.
Ακολουθώντας τον κανόνα που επιβάλλει να προτιμηθεί η σύνθεση, επιλέγεται η επιλογή μιας σχέσης περιεκτικότητας του τρόπου πληρωμής (Pay) στην έννοια του Υπαλλήλου (Employee).

Bridge (Γέφυρα)



Bridge (Γέφυρα)

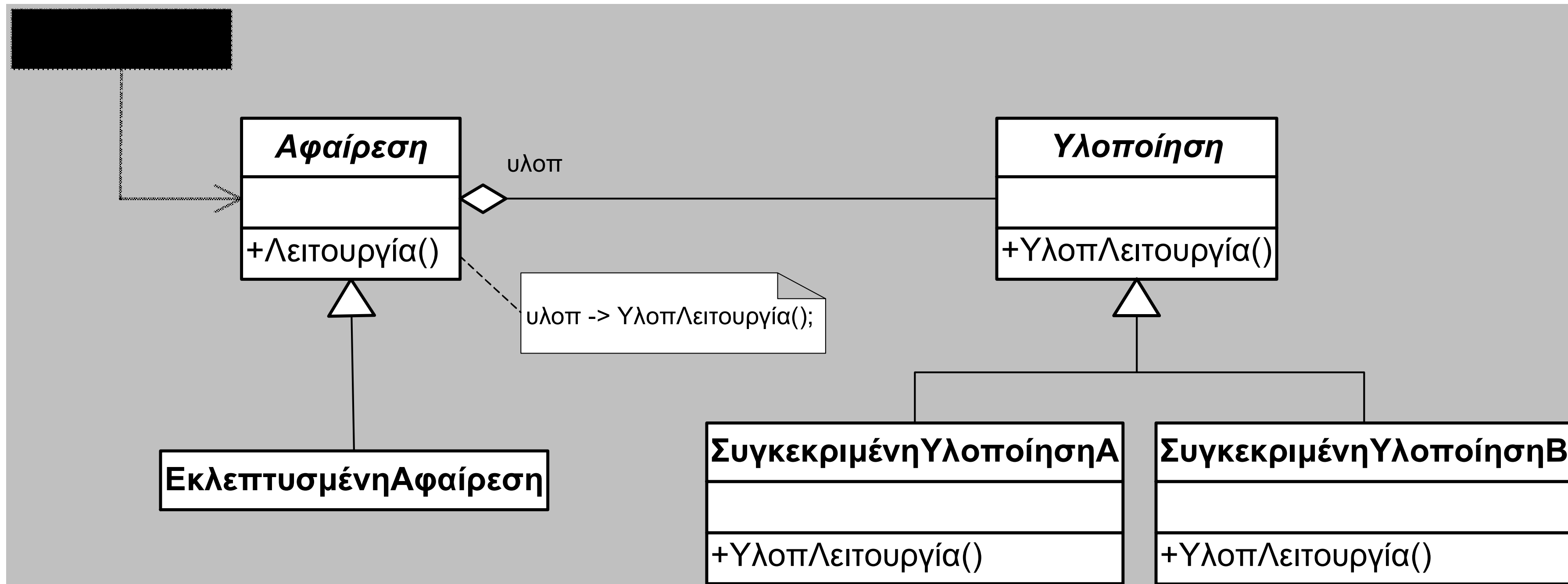
Πλεονεκτήματα:

• Οι υλοποιήσεις δεν είναι μόνιμα συσχετισμένες με μία διασύνδεση. Η υλοποίηση μιας αφαίρεσης μπορεί να διαμορφώνεται ή και να τροποποιείται κατά το χρόνο εκτέλεσης. Στο παράδειγμα, ένα αντικείμενο τύπου TechEmployee μπορεί κατά την εκτέλεση του προγράμματος να συσχετιστεί με τον τρόπο πληρωμής ανά ώρα και στη συνέχεια να τροποποιηθεί ο τρόπος πληρωμής του σε πληρωμή ανά μήνα.

• Οποιαδήποτε αλλαγή στις υλοποιήσεις, δεν απαιτεί μεταγλώττιση των κλάσεων στην ιεραρχία της αφαίρεσης ή των προγραμμάτων που τις χρησιμοποιούν.

• Οποιαδήποτε από τις δύο ιεραρχίες κλάσεων (Αφαίρεση και Υλοποίηση) μπορεί να επεκταθεί (προσθέτοντας περισσότερα επίπεδα) με τρόπο ανεξάρτητο.

Bridge (Γέφυρα)



Singleton (Μοναδιαίο)

• Κατηγορία: Creational

• Σκοπός: *Η εξασφάλιση ότι μία κλάση θα έχει μόνο ένα στιγμιότυπο και παρέχει ένα καθολικό σημείο πρόσβασης σε αυτό .*

• Συνώνυμα: -

Singleton (Μοναδιαίο)

Κλάσεις και στιγμιότυπα: Σχέση ένα-προς-πολλά. Τα αντικείμενα δημιουργούνται δεσμεύοντας χώρο στη μνήμη όποτε κρίνεται σκόπιμο και διαγράφονται από τη μνήμη ότι τερματιστεί η χρήση τους.

Ορισμένες φορές, απαιτείται η ύπαρξη κλάσεων από τις οποίες παράγεται ένα μόνο αντικείμενο. Το αντικείμενο αυτό συνήθως δημιουργείται κατά την έναρξη της εφαρμογής και διαγράφεται με το πέρας της.

Ο ρόλος του μοναδικού αυτού αντικειμένου είναι η διαχείριση των υπολοίπων αντικειμένων της εφαρμογής και για το λόγο αυτό, αποτελεί λογικό σφάλμα να δημιουργηθούν περισσότερα του ενός τέτοια αντικείμενα-διαχειριστές (managers ή controllers).

Singleton (Μοναδιαίο)

Το πρότυπο σχεδίασης "Μοναδιαίο", εξασφαλίζει τη δημιουργία ενός και μόνο αντικειμένου, περιλαμβάνοντας μία ειδική μέθοδο κατασκευής στιγμιοτύπων:

- Όταν καλείται αυτή η μέθοδος, ελέγχει αν κάποιο αντικείμενο έχει ήδη δημιουργηθεί. Αν ναι, η μέθοδος επιστρέφει απλώς έναν δείκτη προς το υπάρχον αντικείμενο. Αν όχι, η μέθοδος δημιουργεί ένα νέο αντικείμενο και επιστρέφει δείκτη προς αυτό.

- Για να εξασφαλισθεί ότι αυτός είναι ο μοναδικός τρόπος δημιουργίας αντικειμένων από αυτή την κλάση, ο κατασκευαστής της κλάσης δηλώνεται ως προστατευμένος (protected) ή ιδιωτικός (private). Με τον τρόπο αυτό, δεν είναι δυνατόν να δημιουργηθεί ένα αντικείμενο, χωρίς να κληθεί η παραπάνω ειδική μέθοδος.

Singleton (Μοναδιαίο)

Παράδειγμα: Μοντέλο μιας CPU.

Πολλοί καταχωρητές γενικής χρήσης, μόνο ένας συσσωρευτής (Accumulator).

Η εφαρμογή πρέπει να εξασφαλίσει ότι δεν θα δημιουργηθούν πέραν του ενός τέτοιοι συσσωρευτές και ότι οι λειτουργίες του καταχωρητή θα είναι καθολικά προσπελάσιμες.

Singleton (Μοναδιαίο)

Εφαρμογή του προτύπου "Μοναδιαίο" και ορίζοντας στην κλάση `Accumulator` μία στατική λειτουργία `getInstance` η οποία δημιουργεί ένα αντικείμενο μόνο όταν αυτό δεν υφίσταται.

Η λειτουργία είναι δηλωμένη στατική, έτσι ώστε να μπορεί να κληθεί και πριν από την κατασκευή του μοναδικού αντικειμένου.

Η κλάση περιλαμβάνει ένα στατικό μέλος `instance`, δείκτη προς το μοναδικό αντικείμενο (όσο αυτό δεν υφίσταται, η τιμή του δείκτη είναι NULL).

• Η λειτουργία `getInstance` σε περίπτωση που η τιμή του δείκτη `instance` είναι NULL δημιουργεί ένα αντικείμενο της κλάσης..

• Σε περίπτωση που ο δείκτης έχει ήδη μία τιμή, η λειτουργία απλώς επιστρέφει την τιμή του.

Singleton (Μοναδιαίο)

```
class Accumulator {
public:
    static Accumulator* getInstance();
protected:
    Accumulator();
private:
    static Accumulator* instance;
};

Accumulator* Accumulator::instance = NULL;

Accumulator* Accumulator::getInstance()
{
    if(instance == NULL)
        instance = new Accumulator;

    return instance;
}
```

Singleton (Μοναδιαίο)

Lazy Initialization: Η τιμή που επιστρέφει η μέθοδος, δεν δημιουργείται μέχρι την πρώτη κλήση της. Αν επομένως ένα μοναδιαίο αντικείμενο δεν χρησιμοποιηθεί, δεν δημιουργείται.

Στο παρακάτω πρόγραμμα, οι δύο εντολές `printf` επιστρέφουν την ίδια διεύθυνση για τους δείκτες `A` και `A1`.

```
int main() {
    Accumulator* A;

    A = Accumulator::getInstance();

    Accumulator* A1;

    A1 = Accumulator::getInstance();

    printf("A  address: %p\n", A);
    printf("A1 address: %p\n", A1);

    return 0; }
```

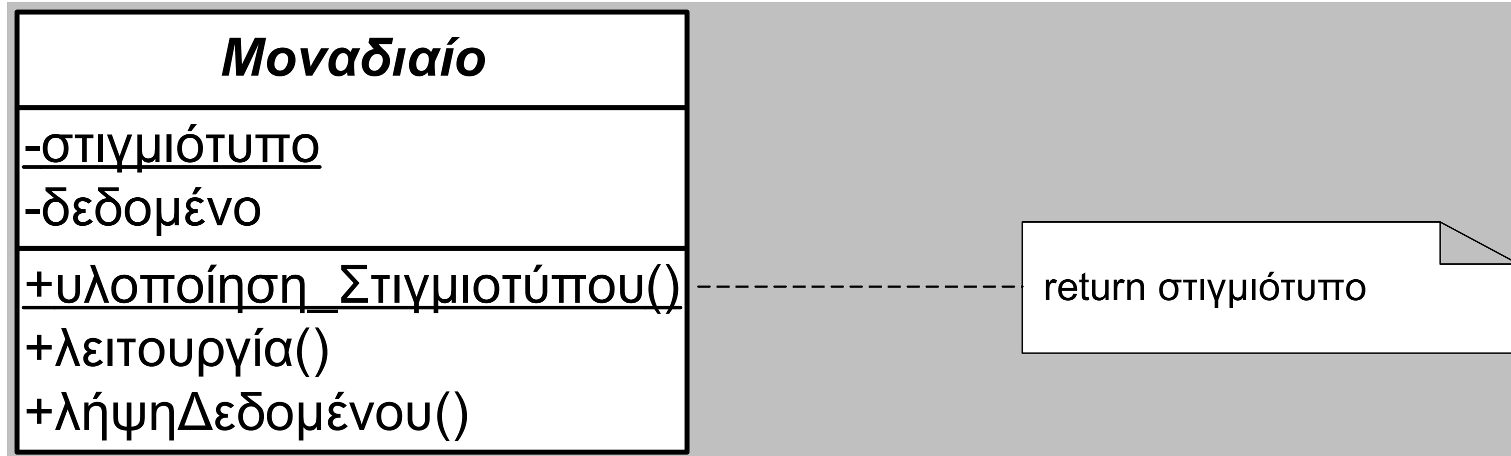
Singleton (Μοναδιαίο)

Απλότητα: οποιαδήποτε κλάση να μετατραπεί σε μοναδιαία.

Πλεονέκτημα: σε περίπτωση δημιουργίας παράγωγων κλάσεων από μία μοναδιαία κλάση, κάθε απόγονος μπορεί να είναι επίσης μοναδιαία κλάση αν προστεθούν και σε αυτές η στατική ιδιότητα και μέθοδος.

Για παράδειγμα, αν υπάρχουν ειδικές κατηγορίες συσσωρευτών από τις οποίες μπορεί να επιλέξει ο σχεδιαστής, κάθε κατηγορία μπορεί να κληρονομεί από την κλάση `Accumulator`. Καθώς η ιδιότητα instance είναι δείκτης προς αντικείμενα τύπου `Accumulator`, η μέθοδος `getInstance` μπορεί να αναθέσει στην `instance` την τιμή ενός δείκτη προς οποιαδήποτε παράγωγη κλάση.

Singleton (Μοναδιαίο)



Visitor (Επισκέπτης)

• Υλοποιεί τη λεγόμενη "διπλή αποστολή" (**double dispatch** ή **dual dispatch**).

• Τα μηνύματα στον OOP κατά κανόνα επιδεικνύουν συμπεριφορά που αντιστοιχεί στην "απλή αποστολή":

-η λειτουργία που εκτελείται εξαρτάται:

- από το όνομα της αίτησης

- τον τύπο του (μοναδικού) αποδέκτη

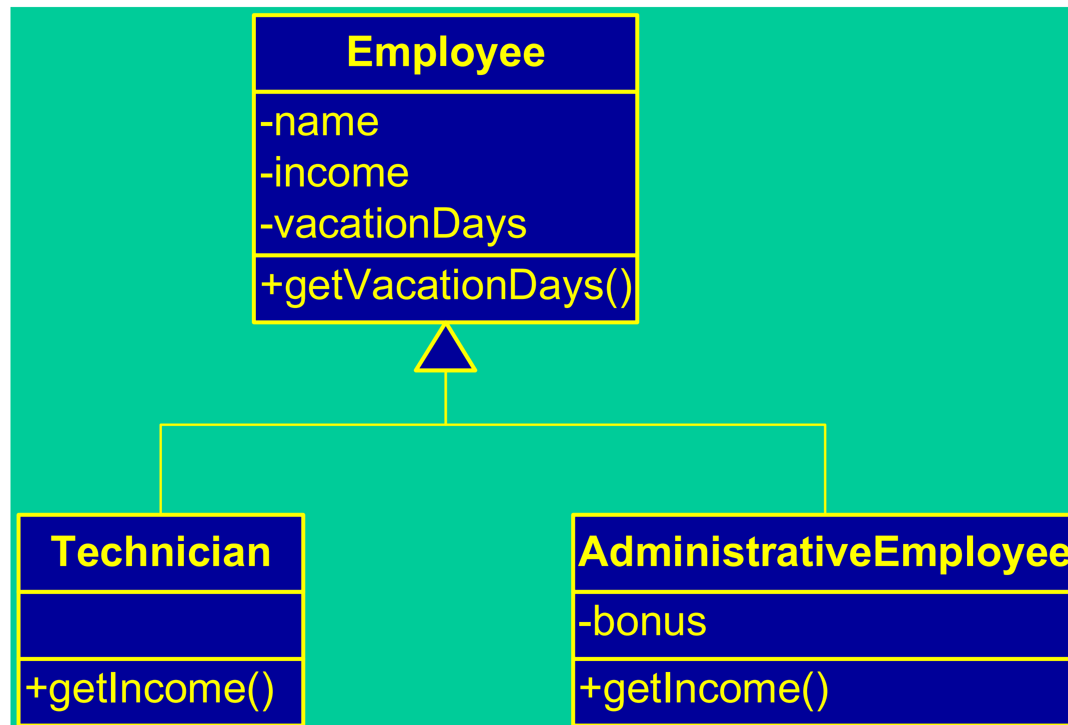
• Στη "διπλή αποστολή" η λειτουργία που εκτελείται εξαρτάται:

- από το όνομα της αίτησης και

- τον τύπο **δύο αποδεκτών** (στο συγκεκριμένο πρότυπο από τον τύπο του "Επισκέπτη" και τον τύπο του στοιχείου που επισκέπτεται).

Visitor (Επισκέπτης)

• Παράδειγμα: εφαρμογή η οποία διαχειρίζεται το εισόδημα (Income) και τις μέρες αδείας (Vacation Days) των υπαλλήλων μιας εταιρείας



donwload αρχικό κώδικα από macedonia.uom.gr

Visitor (Επισκέπτης)

• Νέες απαιτήσεις: Υποθέτουμε ότι η εταιρεία εφαρμόζει μία πολιτική τροποποίησης των μισθών και αδειών σε τακτά χρονικά διαστήματα. Η τροποποίηση των κλάσεων είναι υπαρκτή δυνατότητα, αλλά δεν είναι ρεαλιστική.

•Φιλοσοφία: προσθήκη λειτουργικότητας χωρίς την αλλαγή του κώδικα υπαρχόντων κλάσεων.

• Υλοποίηση: Δημιουργία μίας ιεραρχίας κλάσεων Επισκέπτη (Visitor class hierarchy) η οποία ορίζει αμιγώς υπερβατές μεθόδους `visit()` στην αφηρημένη κλάση βάσης. Κάθε μέθοδος `visit()` λαμβάνει μία μοναδική παράμετρο – έναν δείκτη ή αναφορά προς μία κλάση της αρχικής ιεραρχίας.

Visitor (Επισκέπτης)

Σχεδίαση:

• 1ο Στάδιο: Κάθε νέα λειτουργία μοντελοποιείται ως παράγωγος κλάσης της ιεραρχίας των Επισκεπτών. Οι αφηρημένες μέθοδοι visit() υλοποιούν την επιθυμητή λειτουργικότητα για τη συγκεκριμένη κλάση που δέχονται ως παράμετρο.

• Πώς λαμβάνει κάθε κλάση "Επισκέπτης" το δείκτη ή την αναφορά προς το στοιχείο που καλείται να εμπλουτίσει με νέα λειτουργικότητα?

• 2ο Στάδιο: προσθήκη αφηρημένης μεθόδου `accept`() στη βασική κλάση της πρωτότυπης ιεραρχίας. Η `accept`() λαμβάνει έναν δείκτη προς την αφηρημένη κλάση βάσης της ιεραρχίας των Επισκεπτών.

• Κάθε στοιχείο της πρωτότυπης ιεραρχίας, υλοποιεί την `accept()` καλώντας τη μέθοδο `visit()` του στιγμιοτύπου των Επισκεπτών που της δόθηκε ως παράμετρος, περνώντας τον "this" pointer.

Visitor (Επισκέπτης)

Εφαρμογή:

όταν ο πελάτης χρειάζεται να προσθέσει μία λειτουργία:

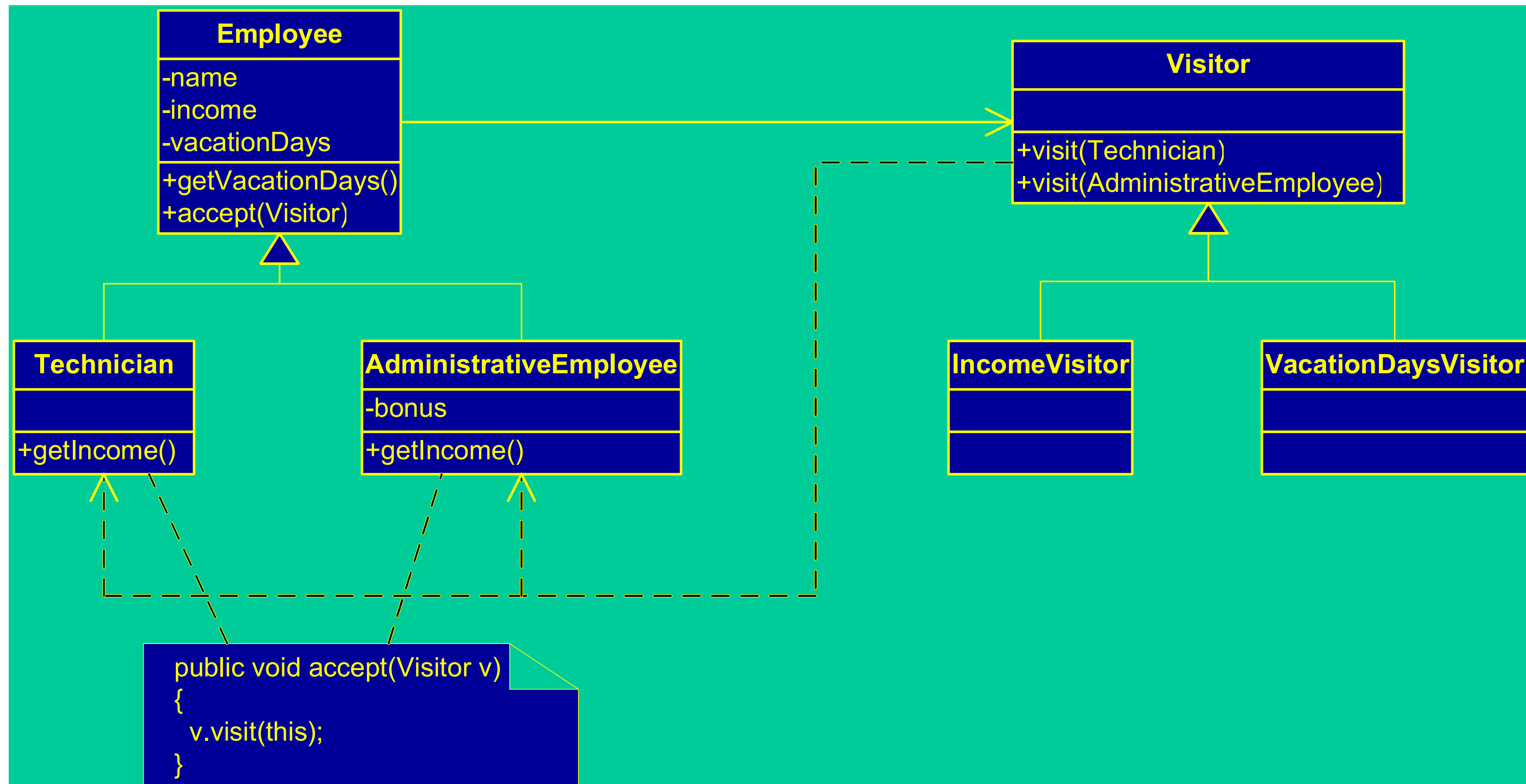
Βήμα 1ο: δημιουργεί ένα στιγμιότυπο του αντικειμένου Επισκέπτη που αφορά στη συγκεκριμένη λειτουργικότητα.

Βήμα 2ο: καλεί τη μέθοδο accept() σε κάθε στοιχείο της πρωτότυπης ιεραρχίας που επιθυμεί να εμπλουτίσει περνώντας ως όρισμα το αντικείμενο Επισκέπτη.

Η μέθοδος accept() μέσω της "διπλής αποστολής" καθοδηγεί τη ροή του ελέγχου στην κατάλληλη κλάση Visitor και στη συνέχεια στην εκτέλεση της λειτουργικότητας στην επιθυμητή αρχική κλάση.

Visitor (Επισκέπτης)

Διάγραμμα UML:



Visitor (Επισκέπτης)

Παρατήρηση:

Η προσθήκη νέων μεθόδων είναι πλέον σχετικά εύκολη – αρκεί η προσθήκη μιας νέας υποκλάσης στην ιεραρχία των Επισκεπτών.

Ωστόσο, αν οι κλάσεις στην αρχική ιεραρχία δεν είναι σταθερές, τότε ο συγχρονισμός μεταξύ της ιεραρχίας των Επισκεπτών και της ιεραρχίας των αρχικών κλάσεων απαιτεί μεγάλη προσπάθεια, που ίσως να μη δικαιολογεί τη χρήση του προτύπου.

Visitor (Επισκέπτης)

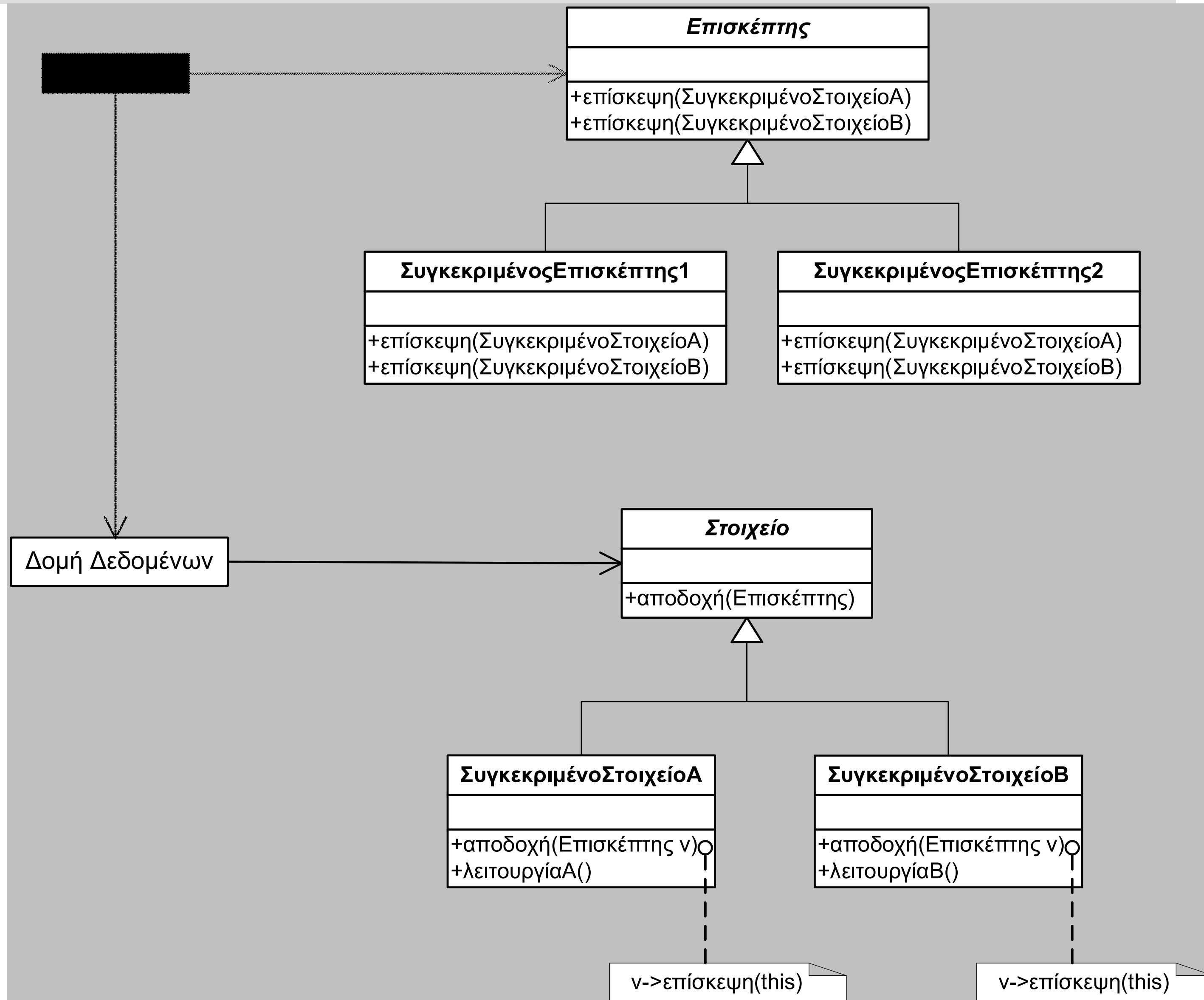
Γενική Δομή - Εφαρμογή:

Χρησιμοποιείστε το πρότυπο σχεδίασης "Επισκέπτης" :

• όταν θέλετε να προσθέσετε λειτουργίες στα αντικείμενα μιας ιεραρχίας αντικειμένων χωρίς να "μολύνετε" τις κλάσεις τους με αυτές τις λειτουργίες. Το πρότυπο "Επισκέπτης" σας επιτρέπει να διατηρείτε σχετιζόμενες λειτουργίες σε ένα μέρος ορίζοντας αυτές σε μία ξεχωριστή κλάση.

Διάγραμμα:

Visitor (Επισκέπτης)



```
abstract class Employee {

    public Employee(String text, double amount, int days)
    {
            name = text;
            income = amount;
            vacationDays = days;
    }

    public void setName(String text) { name = text; }
    public void setIncome(double amount) { income = amount; }
    public void setVacationDays(int days) { vacationDays = days; }

    public String getName() { return name; }
    public int getVacationDays() { return vacationDays; }

    public abstract void accept(Visitor visitor);

    private String name;
    protected double income;
    private int vacationDays;

}
```

```
class Technician extends Employee {

    public Technician(String text, double amount, int days)
    {
      super(text, amount, days);
    }

    public double getIncome() { return income; }

    public void accept(Visitor visitor) {visitor.visit(this);}

}
```

```
class AdministrativeEmployee extends Employee {

    public AdministrativeEmployee(String text, double amount, int days,
    int extra)
    {
      super(text, amount, days);
      bonus = extra;
    }

    public void setBonus(double amount) { bonus = amount; }
    public double getBonus() { return bonus; }

    public double getIncome() { return income; }

    public void accept(Visitor visitor) {visitor.visit(this);}

    private double bonus;
}
```

```
interface Visitor {

    public void visit(Technician employee);
    public void visit(AdministrativeEmployee employee);
}

class IncomeVisitor implements Visitor {

    public void visit(Technician employee)
    {
        //Technicians get an increase 10% of their income
        employee.setIncome(employee.getIncome() * 1.10);
    }

    public void visit(AdministrativeEmployee employee)
    {
        //Administrative Employees get an increase 20% of their bonus
        employee.setBonus(employee.getBonus() * 1.20);
    }
}
```

```
class VacationDaysVisitor implements Visitor {

    public void visit(Technician employee)
    {
        //Technicians get an increase of 3 days
        employee.setVacationDays(employee.getVacationDays() + 3);
    }

    public void visit(AdministrativeEmployee employee)
    {
        //Administrative Employees get an increase of 2 days
        employee.setVacationDays(employee.getVacationDays() + 2);
    }
}
```

```
public class VisitorTestFull {

    public static void main(String[] args) {

        Technician E1 = new Technician("Fred", 2000, 23);

        AdministrativeEmployee E2 = new AdministrativeEmployee("John", 3000, 25, 200);

        System.out.println(E1.getName() + ", Total Income: " + E1.getIncome() +
                ", Vacation Days: " + E1.getVacationDays() );

    System.out.println(E2.getName() + ", Total Income: " + (E2.getIncome()+E2.getBonus()) +
            ", Vacation Days: " + E2.getVacationDays() );

        IncomeVisitor v1 =  new IncomeVisitor();
        VacationDaysVisitor v2 = new VacationDaysVisitor();

        E1.accept(v1);
        E2.accept(v1);
        E1.accept(v2);
        E2.accept(v2);
        System.out.println("After visits have been performed....");

        System.out.println(E1.getName() + ", Total Income: " + E1.getIncome() +
                ", Vacation Days: " + E1.getVacationDays() );
    System.out.println(E2.getName() + ", Total Income: " + (E2.getIncome()+E2.getBonus()) +
            ", Vacation Days: " + E2.getVacationDays() );

    }
}
```

Observer (Παρατηρητής)

- Κατηγορία: Behavioral
- Σκοπός: *Ο ορισμός μιας σχέσης εξάρτησης ένα-προς-πολλά μεταξύ αντικειμένων έτσι ώστε όταν μεταβάλλεται η κατάσταση ενός αντικειμένου, όλα τα εξαρτώμενα αντικείμενα να ενημερώνονται και να τροποποιούνται αυτόματα .*
- Συνώνυμα: Publish - Subscribe

Observer (Παρατηρητής)

• Στόχος της αντικειμενοστραφούς σχεδίασης είναι η δημιουργία ενός συνόλου αλληλεπιδρώντων αντικειμένων.

• Συχνό πρόβλημα: η αναγκαιότητα συνεργασίας μεταξύ κλάσεων, οδηγεί σε υψηλή σύζευξη.

• Ορισμένα από τα πρότυπα, με χαρακτηριστικότερο το πρότυπο "Παρατηρητής", επιδιώκουν να μειώσουν τη σύζευξη μεταξύ των αντικειμένων, παρέχοντας αυξημένη δυνατότητα επαναχρησιμοποίησης και τροποποίησης του συστήματος.

Observer (Παρατηρητής)

• Το συγκεκριμένο πρότυπο, επιτρέπει την αυτόματη ειδοποίηση και ενημέρωση ενός συνόλου αντικειμένων τα οποία "αναμένουν" ένα γεγονός, που εκδηλώνεται ως αλλαγή στην κατάσταση ενός αντικειμένου.

• Ο στόχος είναι η από-σύζευξη των παρατηρητών από το παρακολουθούμενο αντικείμενο, έτσι ώστε κάθε φορά που προστίθεται ένας νέος παρατηρητής (με διαφορετική διασύνδεση ενδεχομένως), να μην απαιτούνται αλλαγές στο παρακολουθούμενο αντικείμενο

• Το συγκεκριμένο πρότυπο είναι από τα πλέον ευρέως χρησιμοποιούμενα και υλοποιείται με σχετική ευκολία σε διάφορες γλώσσες προγραμματισμού.

•Το πρότυπο "Παρατηρητής" εφαρμόζεται για χρόνια στην γνωστή αρχιτεκτονική Μοντέλου-Όψεως-Έλεγκτή (Model-View-Controller)

Observer (Παρατηρητής)

Η εφαρμογή του προτύπου προϋποθέτει τον εντοπισμό των εξής δύο τμημάτων: ενός υποκειμένου και του παρατηρητή. Μεταξύ των δύο υφίσταται μία συσχέτιση ένα-προς-πολλά

Το υποκείμενο διατηρεί το μοντέλο των δεδομένων και η λειτουργικότητα που αφορά στην παρατήρηση των δεδομένων κατανέμεται σε διακριτά αντικείμενα – παρατηρητές.

Οι παρατηρητές αυτό-καταχωρούνται στο υποκείμενο κατά τη δημιουργία.

Οποτεδήποτε το υποκείμενο αλλάζει, "ανακοινώνει" προς όλους τους καταχωρημένους παρατηρητές το γεγονός της αλλαγής.

Κάθε παρατηρητής ερωτά το υποκείμενο για το υποσύνολο της κατάστασης του υποκειμένου που το ενδιαφέρει

Στο ανωτέρω πρωτόκολλο επικοινωνίας η πληροφορία "αντλείται" αντί να αποστέλλεται στους παρατηρητές,

Observer (Παρατηρητής)

Παράδειγμα: Κλάση Timer που υλοποιεί ένα χρονόμετρο

```
class Timer extends Subject {

   private int state;

   public int  getState()           { return state; }
   public void setState( int in )
   {
      state = in;
      Notify();
   }
}
```

Observer (Παρατηρητής)

Αρχή Αντιστροφής των Εξαρτήσεων: Οι παρατηρητές δεν θα πρέπει να "βλέπουν" μία συγκεκριμένη κλάση, για αυτό και η Timer κληρονομεί μία αφηρημένη κλάση βάσης

```
class Subject {
   private ArrayList observers = new ArrayList();
   private int       totalObs  = 0;

   public void attach( Observer o )  {
     observers.add(o);                    //καταχωρηση παρατηρητη
   }

   public void detach( Observer o){
     observers.remove( o );
   }

   public void Notify()  {
      for (int i=0; i < observers.size(); i++)
         ((Observer)observers.get(i)).update();
                           //ανακοινωση αλλαγων στους παρατηρητες
   }
}
```

Observer (Παρατηρητής)

Η τιμή του χρονομέτρου εμφανίζεται στην οθόνη από διάφορους παρατηρητές

Όλοι οι παρατηρητές πρέπει να έχουν την ίδια διασύνδεση.

Λύση: Κληρονομούν μία αφηρημένη βασική κλάση (Observer)

```
abstract class Observer {
   public abstract void update();
}
```

Observer (Παρατηρητής)

Οι παρατηρητές καταχωρούν τον εαυτό τους καλώντας την attach του υποκειμένου (`public void attach (Observer o)`)

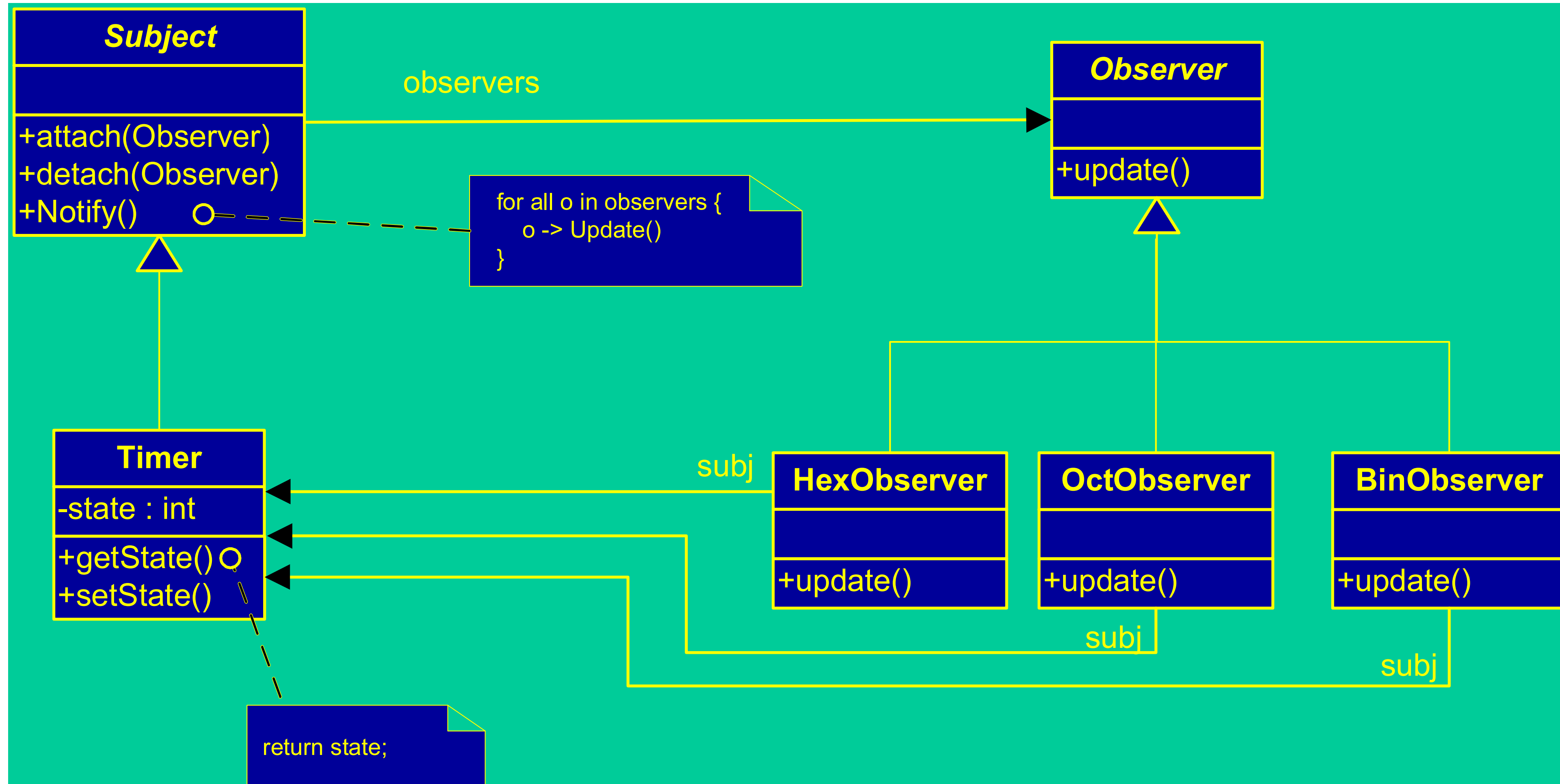
Αν ο παρατηρητής χρειάζεται περισσότερη πληροφορία, την "αντλεί" από το υποκείμενο

```
class HexObserver extends Observer {

   private Timer subj;

   public HexObserver( Timer s ) {
      subj = s;  subj.attach( this );
                         //Οι παρατηρητές καταχωρούν τον εαυτό τους
   }
   public void update() {
    System.out.print("  "+Integer.toHexString(subj.getState()) );
   }                     //Οι παρατηρητές αντλούν πληροφορία
}
```

Observer (Παρατηρητής)



Observer (Παρατηρητής)

Η εφαρμογή του προτύπου έχει νόημα μόνο όταν η λίστα των παρατηρητών αλλάζει δυναμικά

Ειδάλλως (αν για παράδειγμα το υποκείμενο επικοινωνεί πάντοτε με ένα συγκεκριμένο παρατηρητή) η σύζευξη μεταξύ τους κωδικοποιείται ως απλή συσχέτιση

Στρατηγική (Strategy)

- Ορίζει μια οικογένεια αλγορίθμων, τους ενσωματώνει και επιτρέπει την εναλλαγή μεταξύ αυτών.
- Το χρησιμοποιούμε όταν θέλουμε να υλοποιήσουμε έναν κοινό γενικό αλγόριθμο (π.χ. ταξινόμηση) αλλά και πολλές διαφοροποιήσεις του (π.χ. φυσαλίδας, εισαγωγής κ.τ.λ.)

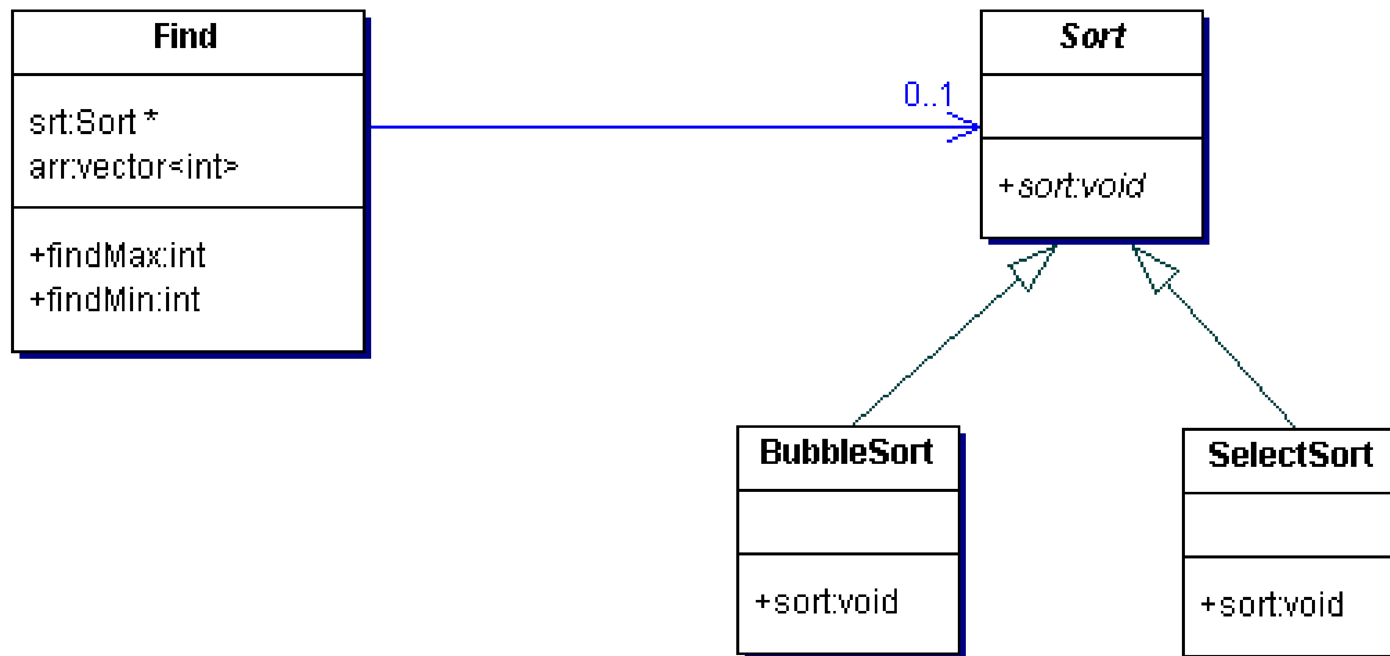
Στρατηγική (Strategy)

- Έστω ότι θέλουμε να δημιουργήσουμε ένα πρόγραμμα το οποίο να βρίσκει την μέγιστη και την ελάχιστη τιμή σε έναν πίνακα

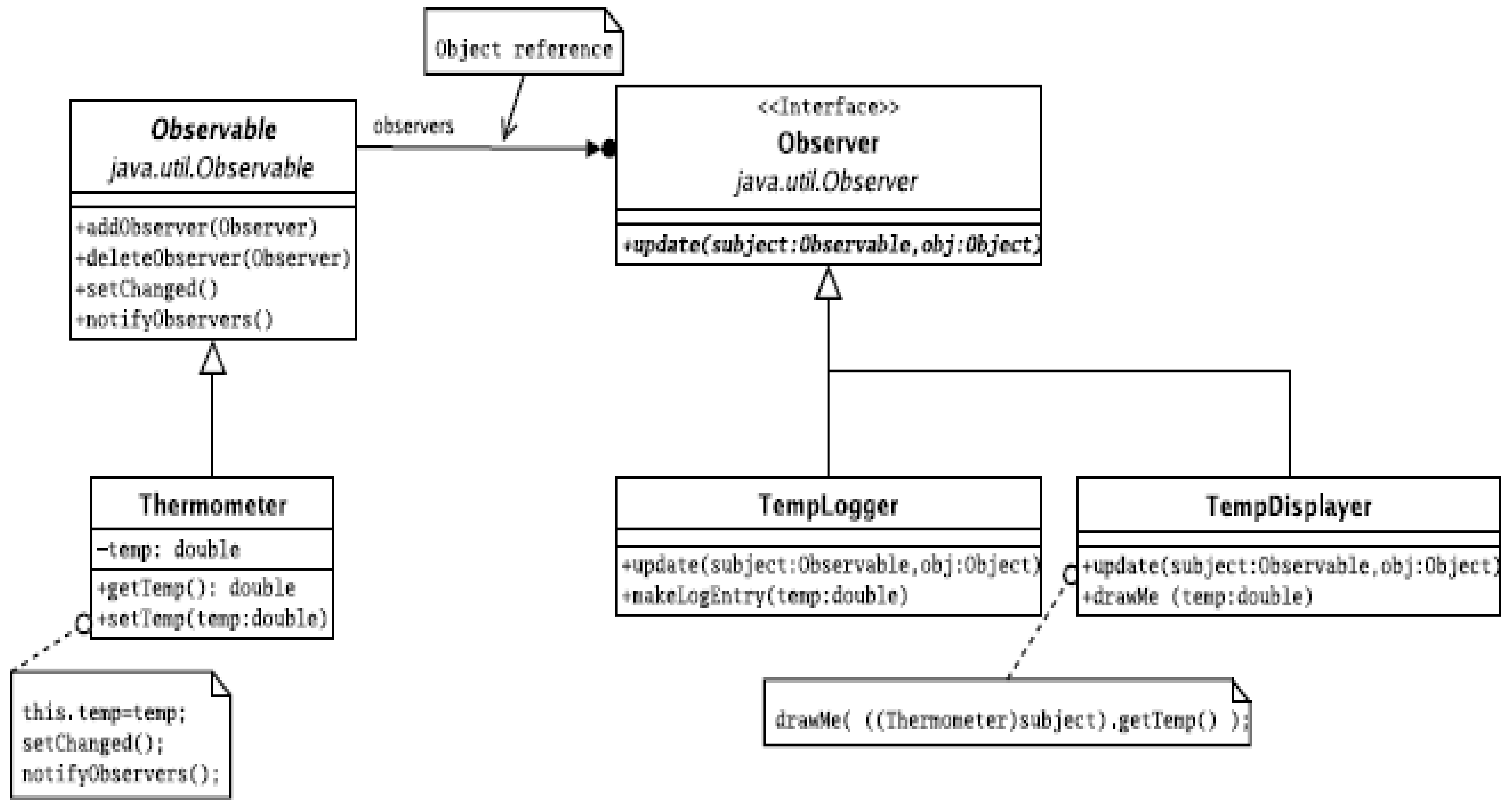
- Επιπλέον, επιθυμούμε για την υλοποίηση του προγράμματος να εκτελείτε αρχικά μια ταξινόμηση του πίνακα.

- Τέλος, επιθυμητό είναι ο χρήστης του προγράμματος να επιλέγει τον αλγόριθμο ταξινόμησης κατά τη διάρκεια της εκτέλεσης

Στρατηγική (Strategy)



- Σχεδιάστε το διάγραμμα κλάσεων για ένα θερμόμετρο το οποίο καταγράφει τη θερμοκρασία του περιβάλλοντος και στη συνέχεια ανάλογα με αυτή μπορεί να αυξομειώνει την ένταση του καλοριφέρ και να αλλάζει τις πληροφορίες (ώρα, βαθμοί C) που απεικονίζονται σε μια οθόνη


Object reference
observers
Observable
java.util.Observable
+addObserver(Observer)
+deleteObserver(Observer)
+setChanged()
+notifyObservers()
<<Interface>>
Observer
java.util.Observer
+update(subject:Observable,obj:Object)
Thermometer
-temp: double
+getTemp(): double
+setTemp(temp:double)
this.temp=temp;
setChanged();
notifyObservers();
TempLogger
+update(subject:Observable,obj:Object)
+makeLogEntry(temp:double)
TempDisplayer
+update(subject:Observable,obj:Object)
+drawMe (temp:double)
drawMe(((Thermometer)subject).getTemp());

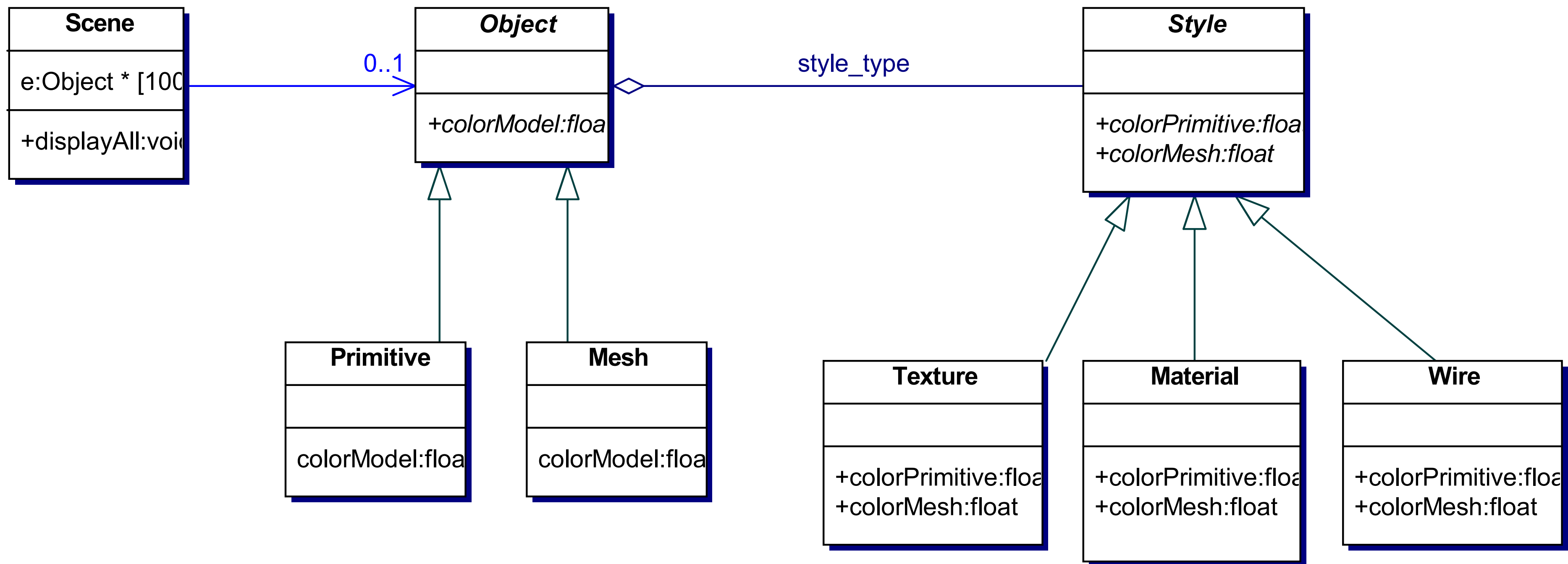
- Θεωρούμε μία εφαρμογή η οποία απεικονίζει τρισδιάστατες σκηνές

- Κάθε αντικείμενο στην σκηνή μπορεί να χρωματιστεί με δύο τρόπους:

  είτε με βάση κάποια φωτογραφία,

  είτε με βάση ένα συγκεκριμένο χρώμα.

- Ο χρωματισμός των αντικειμένων πραγματοποιείται από δύο ανεξάρτητα προγράμματα, που αντιστοιχούν στις κλάσεις Texture και Material.


Scene
e:Object * [100
+displayAll:voi
0..1
Object
+colorModel:floa
style_type
Style
+colorPrimitive:floa
+colorMesh:float
Primitive
colorModel:floa
Mesh
colorModel:floa
Texture
+colorPrimitive:floa
+colorMesh:float
Material
+colorPrimitive:floa
+colorMesh:float
Wire
+colorPrimitive:floa
+colorMesh:float

Περιγραφή του Model- Controller- View framework

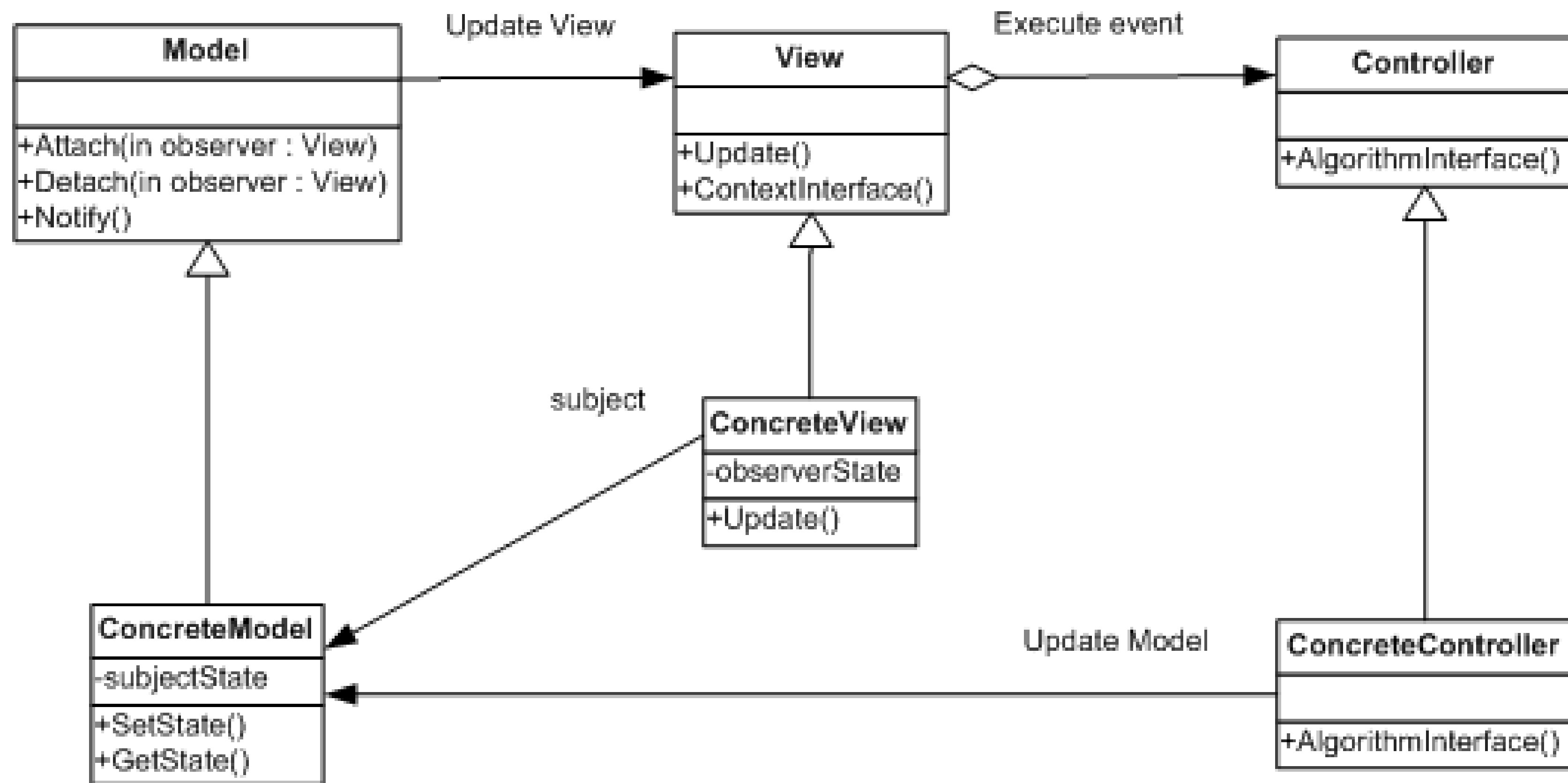


Figure 3: MVC