HY351 - Ανάλυση και Σχεδίαση Πληροφοριακών Συστημάτων
CS351 - Information Systems Analysis and Design

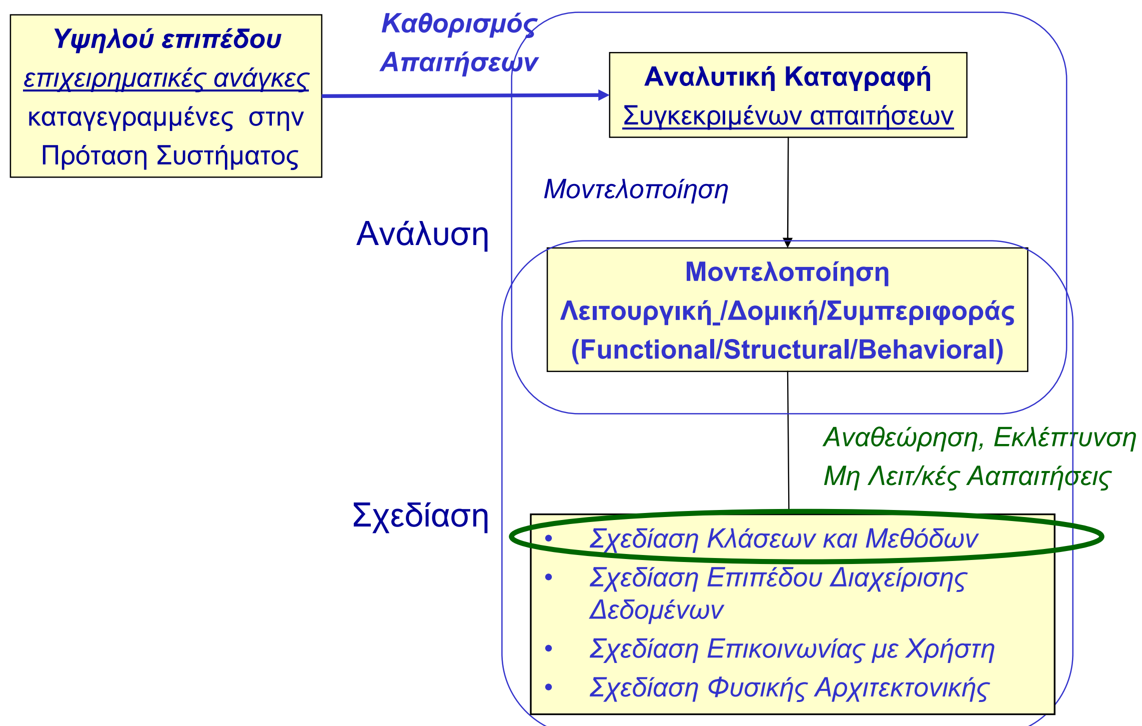
HY351:
Ανάλυση και Σχεδίαση Πληροφοριακών Συστημάτων
Information Systems Analysis and Design

Σχεδίαση Κλάσεων και Μεθόδων
(Class and Method Design)

Γιάννης Τζίτζικας

Διάλεξη : 14
Ημερομηνία :
Θέμα :

Από τα Μοντέλα Ανάλυσης στα Μοντέλα Σχεδίασης



Διάρθρωση

- Γιατί να κάνουμε αναλυτική σχεδίαση κλάσεων και μεθόδων;
- Σχεδιαστικά Κριτήρια
  - σύζευξη (coupling), συνοχή (cohesion)
- Αναδομώντας και Βελτιώνοντας το Σχέδιο (Factoring and Optimizing)
- Αντιστοίχηση κλάσεων προβλήματος σε κλάσεις υλοποίησης
- Προδιαγραφή μεθόδων
- [Περιορισμοί και Συμβόλαια (Constraints and Contracts)]
- Δυνατότητες Επαναχρησιμοποίησης
  - Σχεδιαστικά Μοτίβα (Design Patterns)

Σχεδίαση Κλάσεων και Μεθόδων: Κίνητρο
Class and Method Design: Motivation

- Ένα από τα σημαντικότερα βήματα στη φάση της σχεδίασης είναι η σχεδίαση των κλάσεων και των μεθόδων
- Οι αναλυτές πρέπει να δώσουν οδηγίες και συμβουλές στους προγραμματιστές που να εξηγούν ξεκάθαρα τι πρέπει να κάνει στο σύστημα

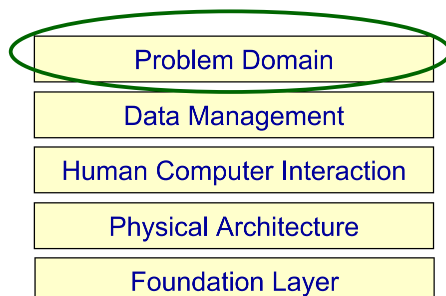
Γιατί να σχεδιάσουμε (πιο λεπτομερώς) τις κλάσεις και μεθόδους;

- Κάποιοι υποστηρίζουν ότι αφού έχουμε επαναχρησιμοποιούμενες βιβλιοθήκες κλάσεων και έτοιμα εξαρτήματα (off-the-shelf components), ο λεπτομερής σχεδιασμός των κλάσεων είναι σπατάλη χρόνου (και άρα δεν πρέπει να χρονοτριβούμε αλλά να ξεκινήσουμε αμέσως την κωδικοποίηση).
- Παρά ταύτα, η εμπειρία έχει δείξει ότι ο αναλυτική σχεδίαση είναι χρήσιμη παρά τις έτοιμες βιβλιοθήκες
  - Ακόμα και οι προϋπάρχουσες κλάσεις πρέπει να κατανοηθούν, οργανωθούν και συναρμολογηθούν σωστά
  - Επίσης η ομάδα θα πρέπει να δημιουργήσει και τις δικές της κλάσεις (application logic of the system) οι οποίες απαιτούν προσεκτική σχεδίαση

Τι θα μπορούσε να πάει στραβά αν δεν κάνουμε προσεχτικά τη σχεδίαση;
What could go wrong without careful design?

- Τα μοντέλα που προέκυψαν από την φάση ανάλυσης μπορεί να μην μπορούν να υλοποιηθούν από την επιλεγμένη γλώσσα προγραμματισμού
- Τα αντικείμενα δεν θα μπορούν να επικοινωνούν σωστά, άρα το σύστημα δεν θα λειτουργεί σωστά
- Μια απρόσεχτη χρήστη στρωμάτων (layers) μπορεί να δημιουργήσει μεγάλη επιβάρυνση στην επικοινωνία (communication overhead) που μπορεί να επιβραδύνει πολύ το σύστημα
- Μια αλλαγή σε ένα τμήμα του συστήματος μπορεί να απαιτήσει αλλαγές σε πάρα πολλά άλλα σημεία του συστήματος
- Θα μπορούσαμε να ολοκληρώσουμε/βελτιώσουμε τη σχεδίαση επαναχρησιμοποιώντας σχεδιαστικά μοτίβα.

Η σπουδαιότητα των κλάσεων προβλήματος
The importance of problem domain classes



- Έχουμε ήδη αρχίσει να σχεδιάζουμε τη δομή και την αλληλεπίδραση των κλάσεων του πεδίου προβλήματος (domain model classes).
- Οι κλάσεις των άλλων επιπέδων (system architecture, HCI, data management) θα εξαρτώνται από τις κλάσεις του στρώματος πεδίου εφαρμογής (problem domain layer).
- Άρα είναι σημαντικό να σχεδιάσουμε σωστά τις κλάσεις του πεδίου προβλήματος (problem domain classes).

Τι είναι η λεπτομερής σχεδίαση κλάσεων και μεθόδων;
What is detailed class and method design?

- Η Μοντελοποίηση της **Δομής** και της **Συμπεριφοράς** (που είδαμε στα προηγούμενα μαθήματα) εντάσσεται πράγματι στη «σχεδίαση κλάσεων και μεθόδων»
- ***Τι πρέπει να κάνουμε παραπάνω;***
- Πρέπει να θέσουμε στον ευατό μας ερωτήματα της μορφής:
  - ***Είναι όλες οι κλάσεις που έχουμε ορίσει απαραίτητες; Μήπως μας λείπουν κάποιες;***
  - ***Είναι πλήρως ορισμένες; Μήπως λείπουν γνωρίσματα ή λειτουργίες;***
  - ***Μήπως έχουν περιττά γνωρίσματα ή λειτουργίες;***
  - ***Μήπως υπάρχουν συγκρούσεις κληρονομικότητας;***
  - ***Υπάρχει κάποια αναποτελεσματικότητα (inefficiency) στο σχέδιο, και πως θα μπορούσαμε να τη διορθώσουμε;***
  - ***Μπορούμε να αντιστοιχίσουμε τις κλάσεις στη γλώσσα προγραμματισμού που θα χρησιμοποιήσουμε;***
  - ***Πως μπορούμε να επαναχρησιμοποιήσουμε κώδικα;***

Δραστηριότητες Λεπτομερούς Σχεδίασης
Detailed Design Activities

Πέραν των προηγουμένων πρέπει να

- ***ελέγξουμε ότι τίποτα δεν λείπει από το μοντέλο του προβλήματος***
- ***οριστικοποιήσουμε την ορατότητα των γνωρισμάτων και λειτουργιών σε κάθε κλάση***
- ***αποφασίσουμε την υπογραφή της κάθε λειτουργίας κάθε κλάσης***
- ***ορίσουμε περιορισμούς που πρέπει να σέβονται τα αντικείμενα***

Υπενθυμιστικό: Αντικειμενοστρεφισμός, Διαγράμματα Κλάσεων
Refresher: Object Orientation, Class Diagrams

Encapsulation
- Hiding the content of the object from outside view
- Communication only through object's methods
- Key to reusability

Polymorphism
- Same message triggers different methods in different objects
- Dynamic binding means specific method is selected at run time
- Implementation of dynamic binding is language specific
- Need to be very careful about run time errors
- Need to ensure semantic consistency

Inheritance
- Single inheritance -- one parent class
- Multiple inheritance -- multiple parent classes
- Inheritance conflict

There are 3 perspectives for the design of a class diagram (of a conceptual model in general)
- ***Conceptual***
  - Independent of implementation. This is often called **domain model.**
- ***Specification***
  - Based on **interfaces of the software**, not the implementation
- ***Implementation***
  - Here we model the **implementation classes**.

Σχεδιαστικά Κριτήρια
Design Criteria

Σχεδιαστικά Κριτήρια
Design Criteria

A good design is one that balances trade-offs to minimize the total cost of the system over its lifetime [Yourdon'91]

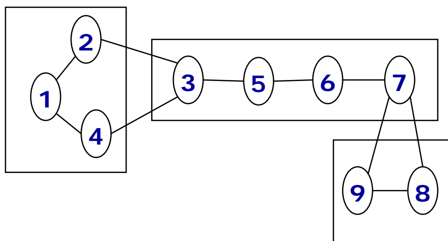
Γενικά σχεδιαστικά κριτήρια

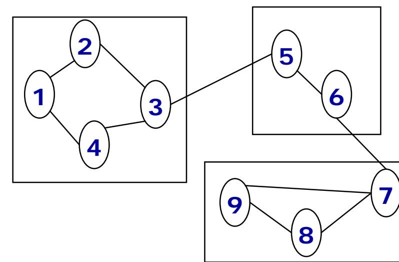
[A] Coupling (σύζευξη)

[B] Cohesion (συνεκτικότητα / συνοχή)

…. [C] Connascence

Έχουμε ήδη συζητήσει αυτά τα κριτήρια (coupling, cohesion) στο μάθημα 12 (διαστρωμάτωση/πακετοποίηση)
Σήμερα θα συζητήσουμε αυτά τα κριτήρια στο επίπεδο των κλάσεων και των μεθόδων.





Σύζευξη και Συνοχή
Coupling and Cohesion

- **Σύζευξη (Coupling)**: μετρά το πόσο <u>εξαρτημένες/ανεξάρτητες είναι οι μονάδες</u> (κλάσεις, αντικείμενα, λειτουργίες) του συστήματος
- **Συνοχή (Cohesion)**: μετρά το πόσο <u>προσηλωμένη (single-minded)</u> είναι μια μονάδα (κλάση, αντικείμενο, λειτουργία)

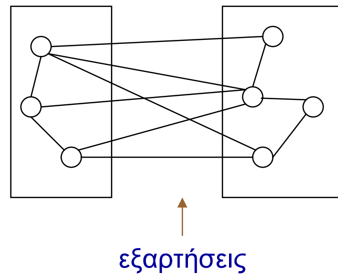
Πλευρές της σύζευξης και της συνοχής:

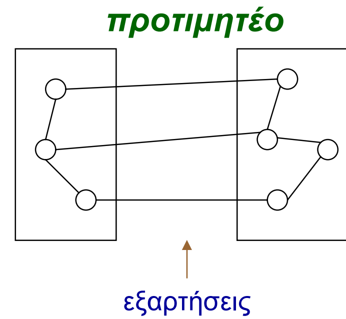
Σύζευξη	Συνοχή
αλληλεπίδρασης	μεθόδου
κληρονομικότητας	κλάσης
	γενίκευσης

Coupling	Cohesion
Interaction	method
Inheritance	class
	generalization

[A] Σύζευξη
[A] Coupling

- Όσο μεγαλύτερη είναι η αλληλεξάρτηση τόσο πιθανότερο είναι αλλαγές σε ένα τμήμα του σχεδίου να απαιτούν αλλαγές και σε άλλα τμήματα του σχεδίου
- Άρα θα θέλαμε να την μειώσουμε





εξαρτήσεις

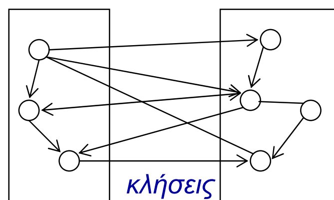
εξαρτήσεις

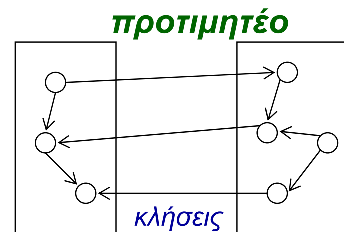
Τύποι σύζευξης:

- Αλληλεπίδρασης (Interaction)
- Κληρονομικότητας (Inheritance)

Σύζευξη> Αλληλεπίδρασης
Coupling> Interaction

Η **σύζευξη αλληλεπίδρασης** αφορά την ανταλλαγή μηνυμάτων (message passing)





Law of Demeter [Lieberherr & Holland 89] («*Only talk to your immediate friends*”)

- **minimize the number of objects that can receive messages from a given object**
- an object could send a message to:
  - itself
  - an object that is contained in an attribute of the object (or one of its superclasses)
  - an object that is passed as a parameter to a method
  - an object that is created by the method
  - an object that is stored in a global variable

Law of Demeter for Functions/Methods

An object A can request a service (call a method) of an object instance B, but object A cannot "reach through" object B to access yet another object to request its services. Doing so would mean that object A implicitly requires greater knowledge of object B's internal structure. Instead, B's class should be modified if necessary so that object A can simply make the request directly of object B, and then let object B propagate the request to any relevant subcomponents. If the law is followed, only object B knows its internal structure. More formally, the Law of Demeter for functions requires that a method *M* of an object *O* may only invoke the methods of the following kinds of objects:

- ***O* itself**
- ***M*'s parameters**
- **any objects created/instantiated within *M***
- ***O*'s direct component objects**

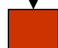
In particular, an object should **avoid** invoking methods of a member object returned by another method.

Advantages: The resulting software tends to be more maintainable and adaptable (as objects are less dependent on the internal structure of other objects, object containers can be changed without reworking their callers).

Disadvantage: Sometimes it requires writing a large number of small "wrapper" methods (sometimes referred to as Demeter *Transmogrifiers*) to propagate method calls to the components. Furthermore, a class's interface can become bulky as it hosts methods for contained classes resulting in a class without a cohesive interface.

6 τύποι Αλληλεπιδραστικής Σύζευξης
6 types of Interaction Coupling

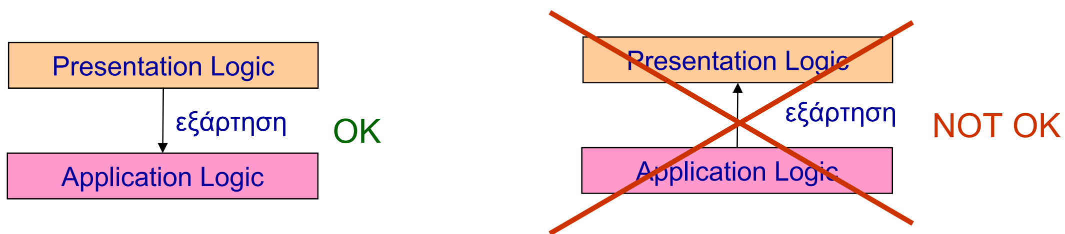
Type	Description
No direct coupling	The methods do not call one another
Data	The calling method passes a variable to the called method. If the variable is an object, the entire object is used by the called method to perform its function
Stamp	The calling method passes a composite variable to the called method, but the called method only uses a portion of the object to perform its function
Control	The calling method passes a control variable whose value will control the execution of the called method
Common or Global	The methods refer to a "global data area" that is outside the individual objects
Content or Pathological	A method of one object refers to the inside (hidden parts) of another object. This violates the principles of encapsulation (C++ allows this with "friends")



Adapted from Dennis et al. 2005

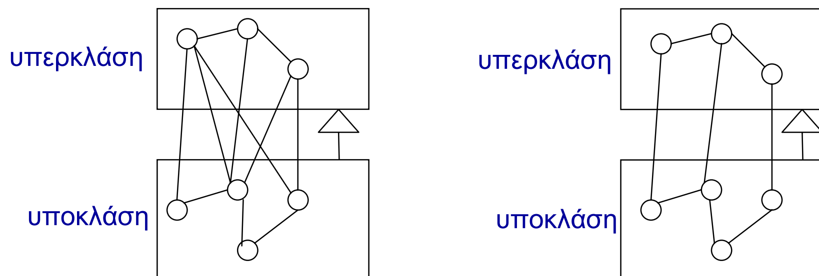
Σύζευξη > Αλληλεπίδρασης: *Οδηγίες*
Coupling> Interaction: *Guidelines*

- Πρέπει να την περιορίσουμε
- Εξαίρεση:
  - Ορισμένες φορές κλάσεις που δεν ανήκουν στο μοντέλο προβλήματος πρέπει να ζευγμένες με κλάσεις του μοντέλου προβλήματος
    - e.g. UIPerson can be coupled to Person
      - for optimization the UIPerson could be pathologically coupled to Person class
- **Παρά ταύτα, οι κλάσεις του μοντέλου προβλήματος δεν πρέπει ποτέ να είναι ζευγμένες με κλάσεις που δεν ανήκουν στο μοντέλο προβλήματος**



Σύζευξης>Κληρονομικότητας
Coupling>Inheritance

Αφορά τις κλάσεις σε μια ιεραρχία γενίκευσης/εξιδείκευσης



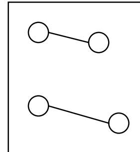
- Μερικοί πιστεύουν ότι η υψηλή ζεύξη δεν είναι άσχημη, άλλοι την θεωρούν άσχημη (λόγω των διαφόρων συγκρούσεων κληρονομικότητας που μπορεί να προκύψουν)
- Σχετικά ερωτήματα
  - *Πρέπει να μπορεί μια μέθοδος ορισμένη σε μια υποκλάση να καλέσει μια μέθοδο της υπερκλάσης;*
  - *Πρέπει μια μέθοδος μιας υποκλάσης να αναφέρεται σε ένα γνώρισμα της υπερκλάσης;*
- Αυτό βέβαια συχνά εξαρτάται από τη γλώσσα προγραμματισμού
- *Συμβουλή*: Να βεβαιωθούμε ότι η κληρονομικότητα χρησιμοποιείται μόνο με σημασία γενίκευσης/εξειδίκευσης (generalization/specialization semantics), δηλαδή υποσυνόλου, και ότι ικανοποιείται η αρχή της υποκατάστασης (substitutability) που θα δούμε αργότερα.

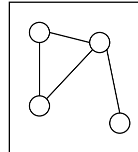
[B] Συνοχή
[B] Cohesion

Συνοχή (Cohesion): μετρά το πόσο προσηλωμένη (single-minded) είναι μια μονάδα (κλάση, αντικείμενο, λειτουργία)

Πρέπει να την μεγιστοποιήσουμε

προτιμητέο





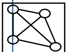
Τύποι Συνοχής:

- Μεθόδων
- Κλάσεων
- Γενίκευσης/Εξειδίκευσης

Συνοχή>Μεθόδων
Cohesion>Method

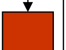
Μια μέθοδος πρέπει να κάνει **μια εργασία** (a method should solve a **single task)**

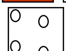
- Μια μέθοδος που κάνει πολλές λειτουργίες είναι πιο δύσκολο να κατανοηθεί και να επαναχρησιμοποιηθεί (a method performing multiple functions is more difficult to understand and reuse)



7 types of method cohesion:

Functional	A method performs one single task
Sequential	The method combines two functions: the output from the first is used as input to the second
Communicational	The method combines two functions that use the same attributes to execute
Procedural	The method supports multiple weakly related functions
Temporal or Classical	The method supports multiple related functions in time (e.g. initialize all attributes)
Logical	The method supports multiple related functions but the choice of the specific function is chosen based on a control variable that is passed as parameter
Coincidental	The method supports multiple unrelated functions





Adapted from Dennis et al. 2005

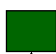
Συνοχή>Κλάσεων
Cohesion>Class

- Μια κλάση πρέπει να αναπαριστά ένα «πράγμα» (π.χ. πρόσωπο, αυτοκίνητο)
- Όλα τα γνωρίσματα και λειτουργίες της κλάσης πρέπει να είναι απαραίτητα για αναπαραστήσουν ένα πράγμα. Δεν πρέπει να υπάρχουν περιττά γνωρίσματα
- Η συνοχή μιας κλάσης είναι ο βαθμός συνοχής μεταξύ των γνωρισμάτων και των λειτουργιών της κλάσης

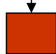
Guidelines [G. Meyers 78]

- a class should contain multiple methods that are visible outside of the class and that each visible method only performs a single function (I.e. functional cohesion)
- a class should have methods that only refer to attributes or other methods defined with the class or its superclasses
- a class should not have any control-flow couplings between its methods

4 τύποι συνοχής Κλάσεων
4 types of Class Cohesion

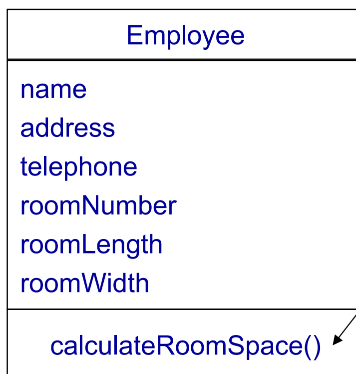


• **Ideal**	• The class has no mixed cohesions
• **Mixed-Role**	• The class has one or more attributes that relate objects of the class to other objects on the same layer (e.g. the problem domain layer), but the attribute(s) have nothing to do with the underlying semantics of the class
• **Mixed-Domain**	• The class has one or more attributes that relate objects of the class to other objects on a different layer. So these attributes have nothing to do with the underlying semantics of the thing that the class represents.
• **Mixed-Instance**	• The class represents different types of objects meaning that different instances only use a portion of the full definition of the class. The class should be decomposed into separate classes.



Adapted from Dennis et al. 2005

Παράδειγμα: Συνοχή μεθόδων έναντι συνοχής κλάσης
Example: Method vs Class Cohesion



Υψηλή συνοχή μεθόδου αλλά
Χαμηλή συνοχή κλάσης

Συνοχή>Γενίκευση/Εξειδίκευση
Cohesion>Generalization/Specialization

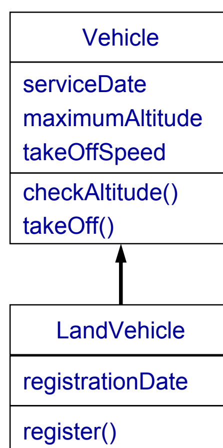
- *Για ποιο λόγο οι κλάσεις μιας ιεραρχίας κληρονομικότητας συνδέονται;*
- *Η σύνδεση τους έχει πράγματι σημασία γενίκευσης/εξιδείκευσης (generalization/specialization semantics);*
- *Ή μήπως συνδέονται μόνο για λόγους επαναχρησιμοποίησης;*
- *Σε ποιο βαθμό μια υποκλάση πράγματι χρειάζεται αυτά που κληρονομεί;*

Για ποιο λόγο οι κλάσεις μιας ιεραρχίας κληρονομικότητας συνδέονται;

Uses of Inheritance for

- *Specialization*: The child class is a special case of the parent class; in other words, the child class is a subtype of the parent class
- *Specification*: The parent class defines behavior that is implemented in the child class but not in the parent class
- *Construction*: The child class makes use of the behavior provided by the parent class, but is not a subtype of the parent class
- *Extension*: The child class adds new functionality to the parent class, but does not change any inherited behavior
- *Limitation*: The child class restricts the use of some of the behavior inherited from the parent class
- *Combination*: The child class inherits behavior from two or more parent classes

Συνοχή>Γενίκευση/Εξειδίκευση
Cohesion>Generalization/Specialization

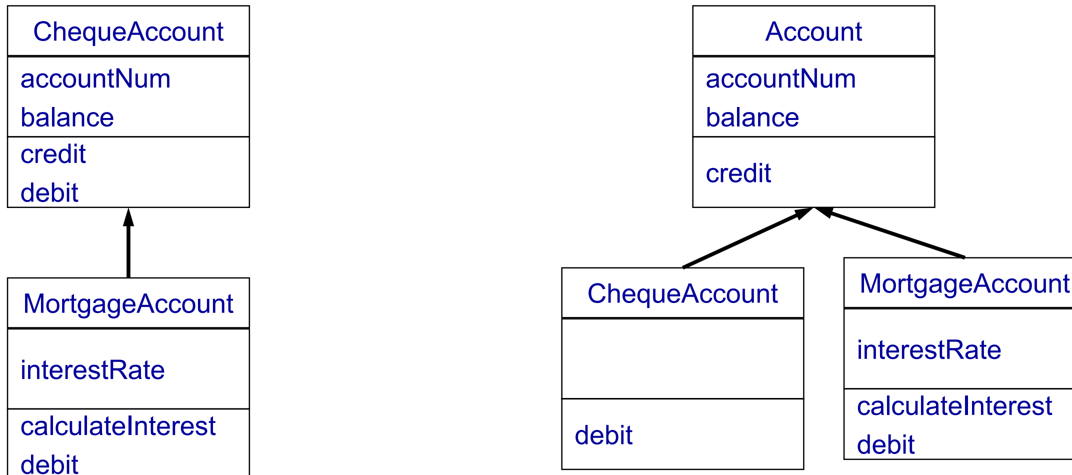


Πολύ Χαμηλή Συνοχή Κληρονομικότητας

Παράδειγμα: Αναδόμηση για ικανοποίηση της αρχής LSP
Example: Restructuring for satisfying LSP

Αρχή Υποκατάστασης του Liskov (Liskov Substitution Principle, LSP)

– Σε μια ιεραρχία κλάσεων πρέπει να είναι δυνατόν να μεταχειριστούμε ένα εξειδικευμένο αντικείμενο ωσάν να ήταν ένα βασικό (γενικό) αντικείμενο



Παράδειγμα (1/3)

```
class Rectangle {
public:
   void setWidth(double w) {itsWidth=w;}
   void setHeight(double h){itsHeight=h;}
   double getArea() {return itsHeight * itsWidth;}
private:
  double itsWidth;
  double itsHeight;
};
```

```
class Square: public Rectangle {
...
};
```

```
void Square::setWidth(double w)
{
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
};
void Square::setHeight(double w)
{
  Rectangle::setWidth(w);
  Rectangle::setHeight(w);
};
```

Παράδειγμα(2/3)

```
void f(Rectangle* r)
{
   r->setWidth(32);
}
```

Παραβίαση της LSP:
Η συνάρτηση f δεν θα λειτουργήσει σωστά εάν το r είναι τετράγωνο (αν είναι στιγμυότυπο της Square)

```
class Rectangle {
public:
   virtual void setWidth(double w) {itsWidth=w;}
   virtual void setHeight(double h){itsHeight=h;}
   double getArea() {return itsHeight * itsWidth;}
private:
  double itsWidth;
  double itsHeight;
};
```

Θα μπορούσαμε να διορθώσουμε αυτό το πρόβλημα επιτρέποντας πολυμορφισμό:
We could fix this problem by allowing polymorphism

Παράδειγμα (3/3)

```
void g(Rectangle* r)
{
   r->setWidth(5);
   r->setHeight(4);
  assert(r->getArea()==20);
}
```

Θα λειτουργήσει σωστά εάν το r είναι παραλληλόγραμμο
Δεν θα λειτουργήσει σωστά εάν το r είναι τετράγωνο

Αρα η g() είναι **εύθραυστη** (fragile) διότι παραβιάσει την LSP

Η κλάση Square παραβιάζει την «αναλλοίωτη συνθήκη» της κλάσης Rectangle, ήτοι την ανεξαρτησία μεταξύ πλάτους και ύψους .

Αυτή η αναλλοίωτη θα μπορούσε να εκφραστεί στην OCL (που θα δούμε σε επόμενο μάθημα). Συγκεκριμένα θα εκφραζόταν ως μια μετα-συνθήκη (post-condition) της setWidth που λέει το height διατηρεί την τιμή του.

Αναδομώντας το Σχέδιο
Restructuring the Design

Factoring

Optimizing

Translate to Implementation Language

Παραγοντοποίηση
Factoring

Factoring is the process of separating out aspects of a method or class into a new method or class to simplify the overall design.

For example we may realize that some classes of the design share a similar definition. In this case we can factor out the similarities and define a new class. The new class is then related with the existing classes through generalization, aggregation or association.

Βελτιώνοντας το Σχέδιο
Optimizing the Design

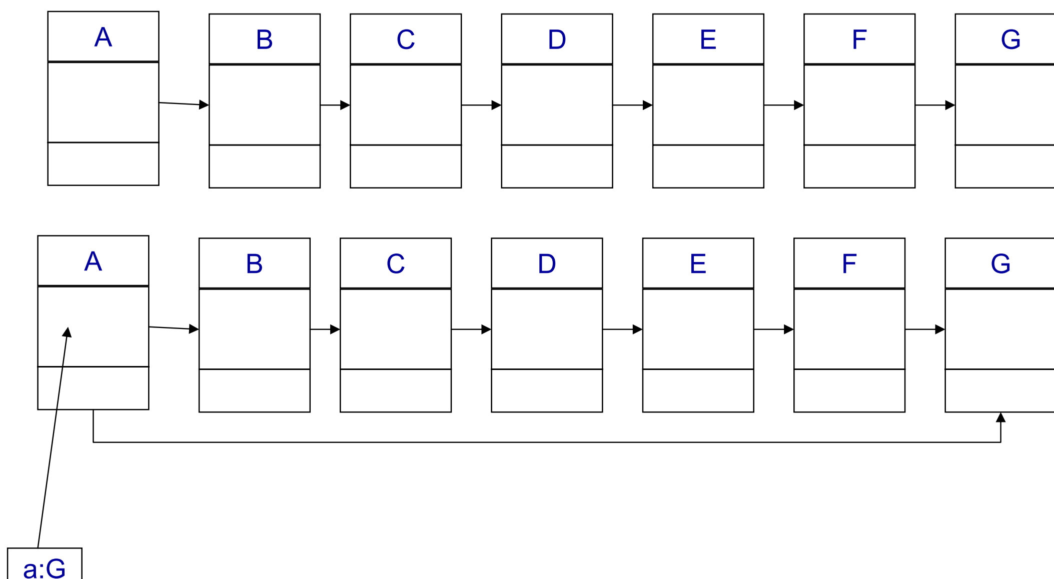
Κατανοησιμότητα έναντι ***Αποδοτικότητας***

- There is a ***trade-off*** between understandability and efficiency
  - Η αύξηση της κατανοησιμότητας ενός σχεδίου συχνά οδηγεί σε μείωση τα αποδοτικότητας (inefficiencies)
  - Αν εστιάσουμε μόνο στην αποδοτικότητα συχνά καταλήγουμε σε σχέδια που είναι δύσκολο να κατανοηθούν από κάποιον άλλο

- Μερικοί τρόποι βελτίωσης της αποδοτικότητας ενός σχεδίου
  - ***Αναθεώρηση Μονοπατιών Πρόσβασης (Review Access Paths)***
  - ***Αναθεώρηση Γνωρισμάτων (Review Attributes)***
  - ***Χρήση παραγόμενων γνωρισμάτων (derived attributes) όπου κρίνεται απαραίτητο (to cache values)***
  - ***Αναθεώρηση της σειράς εκτέλεσης των εντολών σε μεθόδους που χρησιμοποιούνται συχνά (αυτό ανήκει στη «Σχεδίαση Μεθόδων» που θα δούμε αργότερα)***

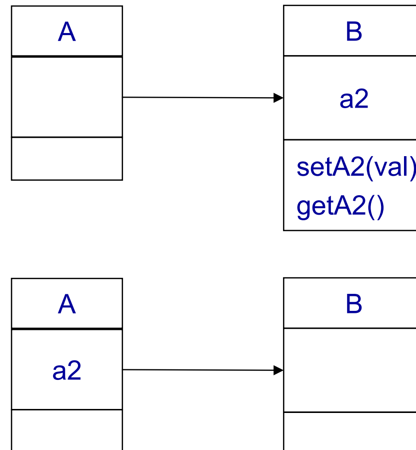
Βελτίωση Σχεδίου: Αναθεωρώντας τα Μονοπάτια Πρόσβασης
Optimizing the Design: Reviewing the Access Paths

Αν ένα μήνυμα πρέπει να διασχίσει ένα μακρύ μονοπάτι και τέτοιου είδους κλήσεις γίνονται συχνά, είναι καλύτερα να δημιουργήσουμε μια πλεονάζουσα άμεση σύνδεση



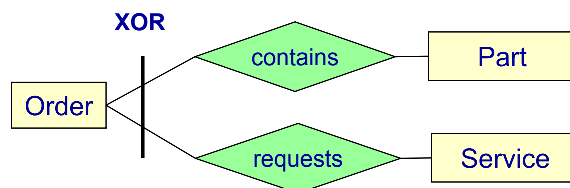
Βελτίωση Σχεδίου: Αναθεωρώντας τα Γνωρίσματα
Optimizing the Design: Reviewing the Attributes

Αν το γνώρισμα a2 της B χρησιμοποιείται μόνο από τις μεθόδους setA2 και getA2 και μόνο η κλάση A χρησιμοποιεί αυτές τις 2 μεθόδους, τότε το a2 καλύτερα να τοποθετηθεί στην A.

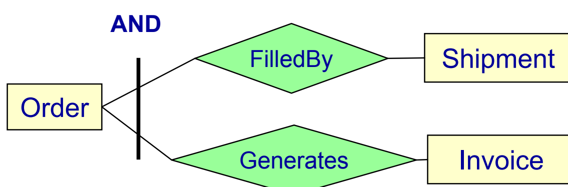


Παρένθεση: AND/XOR για σχέσεις και κλάσεις συσχέτισης
Parenthesis: AND/XOR for Relationships (also for associations in class diagrams)

Εδώ περιγράφουμε αυτούς τους περιορισμούς σε EntityRelationship diagrams, αλλα γενικά μπορούμε να τους έχουμε και σε διαγράμματα κλάσεων της UML



- Μια παραγγελία **είτε** περιέχει ένα Part **ή** ένα Service. **Ποτέ** και τα δύο.



- Για μια παραγγελία, αν υπάρχει τιμολόγιο τότε υπάρχει **και** αποστολή και το αντίστροφο.
- For any given order, whenever there is at least one invoice there is also at least one shipment and vice versa.

Αντιστοίχηση Κλάσεων Πεδίου Εφαρμογής σε Κλάσεις Υλοποίησης

Mapping Problem Domain Classes to Implementation Languages

- Συγκρούσεις Πολλαπλής Κληρονομικότητας
  - (Multiple Inheritance Conflicts)
- Απαλοιφή Πολλαπλής Κληρονομικότητας
  - Αν η υλοποίηση γίνει σε μια γλώσσα προγραμματισμού που δεν υποστηρίζει πολλαπλή κληρονομικότητα, τότε πρέπει να την απαλείψουμε από τα διαγράμματα κλάσεων
- Απαλοιφή Κληρονομικότητας
  - Αν η υλοποίηση γίνει σε μια γλώσσα προγραμματισμού που δεν υποστηρίζει κληρονομικότητα, τότε πρέπει να την απαλείψουμε από τα διαγράμματα κλάσεων
- Υποστηρίζοντας Πολλαπλή και Δυναμική Ταξινόμηση
  - (Handling Multiple and Dynamic Classification)

Κληρονομικότητα στις Γλώσσες Προγραμματισμού
Inheritance in PLs

Διαφορετικές γλώσσες αντιμετωπίζουν την κληρονομικότητα διαφορετικά

- Άρα στη φάση αυτή της σχεδίασης πρέπει να γνωρίζουμε την γλώσσα προγραμματισμού που θα χρησιμοποιήσουμε

Feature	**C++**	**Eiffel**	**Java**
multiple inheritance	yes	yes	no

Η προσβασιμότητα των κληρονομούμενων στοιχείων επίσης εξαρτάται από τη γλώσσα προγραμματισμού

In UML, visibility (private, public, protected) applies to methods and attributes

- Let A be a class with some private and public attributes and methods.
- Let B be a class defined as a subclass of class A.

What B inherits?

- In Java, we can answer this question right away.
- In C++ we should see how B has been declared as subclass of A
  - C++ allows visibility at the class level.

Κληρονομικότητα και Ορατότητα
Inheritance and Visibility

In C++, class B may have been defined as:

class B: public A

class B: protected A

class B: private A

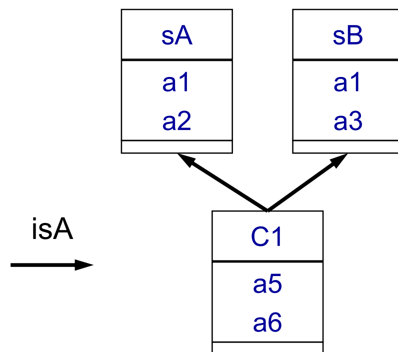
- Accessibility rules (C++)
  - The private properties of A are not visible to class B objects (in every case)
  - If base class A is defined as public, the visibility of inherited properties does not change in derived class B (public are still public and protected are still protected)
  - If base class A is defined as protected, the visibility of inherited public properties changes in derived class B to protected
  - If base class A is defined as private, the visibility of inherited public and protected properties changes in derived class B to private.

Συγκρούσεις Κληρονομικότητας: Πολλαπλή Κληρονομικότητα
Inheritance Conflicts: Multiple Inheritance

[1] Two (or more) attributes (or methods) have the <u>same name and semantics</u>

[2] Two (or more) attributes (or methods) have the <u>same name but different semantics</u> (homonyms)

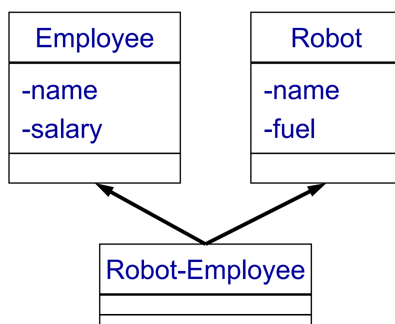
[3] Two (or more) attributes (or methods) have <u>different names but identical semantics</u> (synonyms)



Here we could be in case [1] or [2]

Συγκρούσεις Κληρονομικότητας: Πολλαπλή Κληρονομικότητα
Inheritance Conflicts: Multiple Inheritance

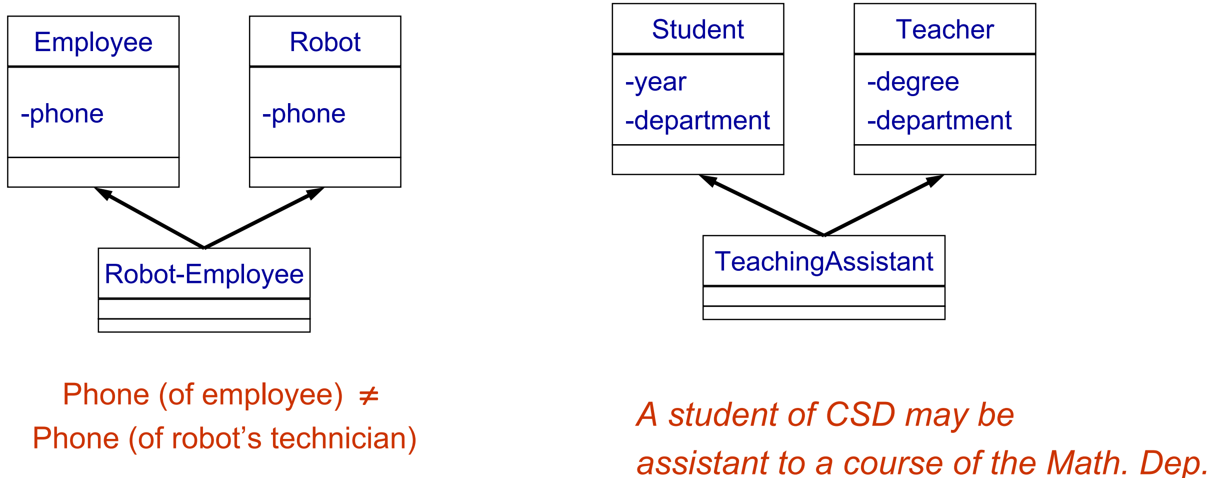
[1] Two (or more) attributes (or methods) inherited from different superclasses have the same name and semantics



name

Συγκρούσεις Κληρονομικότητας: Πολλαπλή Κληρονομικότητα
Inheritance Conflicts: Multiple Inheritance

[2] Two (or more) attributes (or methods) have the same name but different semantics (homonyms)

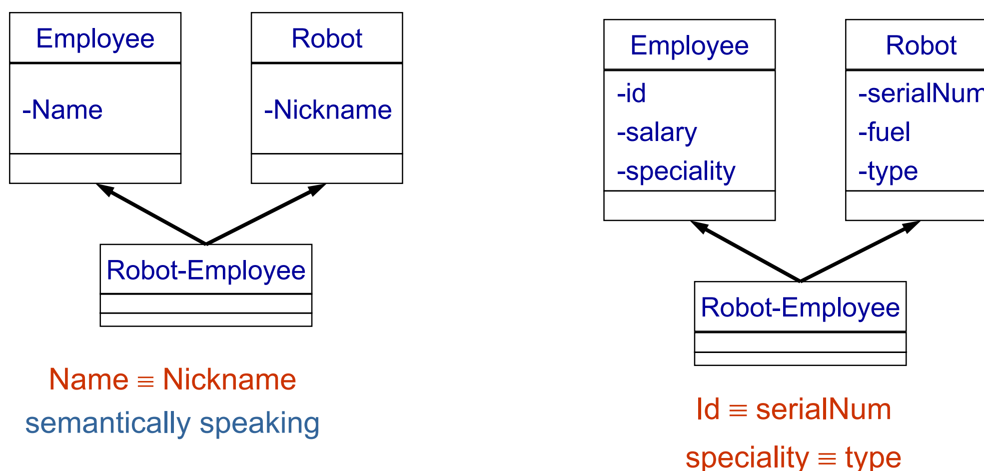


Phone (of employee) ≠ Phone (of robot's technician)

A student of CSD may be assistant to a course of the Math. Dep.

Συγκρούσεις Κληρονομικότητας: Πολλαπλή Κληρονομικότητα
Inheritance Conflicts: Multiple Inheritance

[3] Two (or more) attributes (or methods) inherited from different superclasses have different names but they have identical semantics (synonyms)

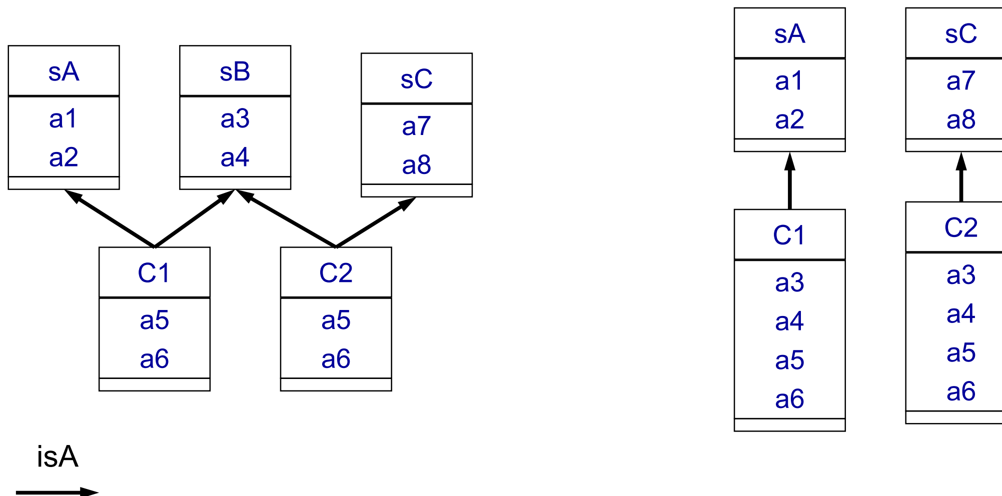


Name ≡ Nickname
semantically speaking

Id ≡ serialNum
speciality ≡ type

Απαλοιφή Πολλαπλής Κληρονομικότητας
Eliminating Multiple Inheritance

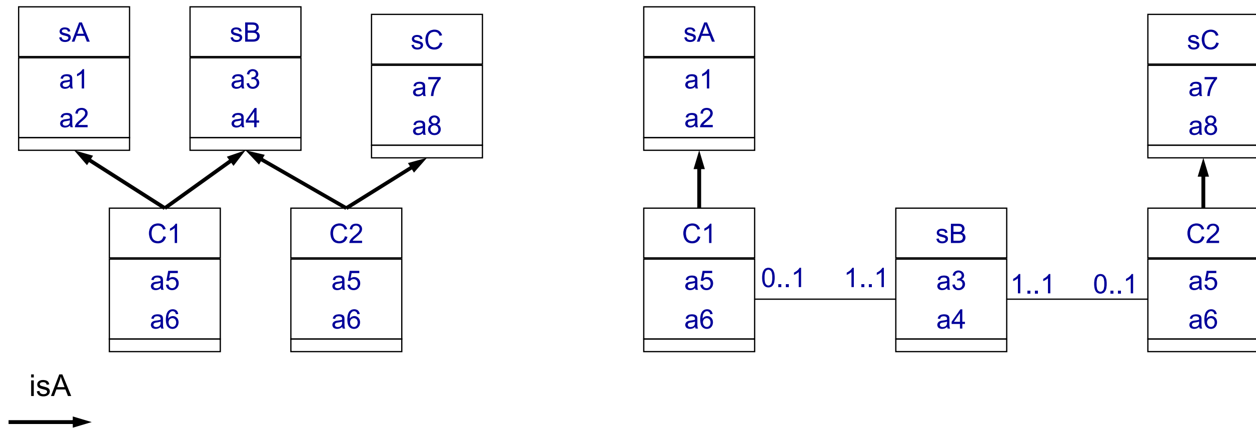
Απαλοιφή Πολλαπλής Κληρονομικότητας
Μέθοδος 1: Ισοπέδωση των Επιπλέον Υπερκλάσεων



Ισοπέδωση της κληρονομικότητας αντιγράφοντας τα γνωρίσματα και τις μεθόδους των επιπλέον υπερκλάσεων σε όλες τις υποκλάσεις τους και τέλος διαγραφή αυτών των υπερκλάσεων από το σχέδιο

Απαλοιφή Πολλαπλής Κληρονομικότητας
Μέθοδος 2: Μετατροπή των Επιπλέον Υπερκλάσεων σε Συσχετίσεις
Convert the Extra Superclasses to Associations

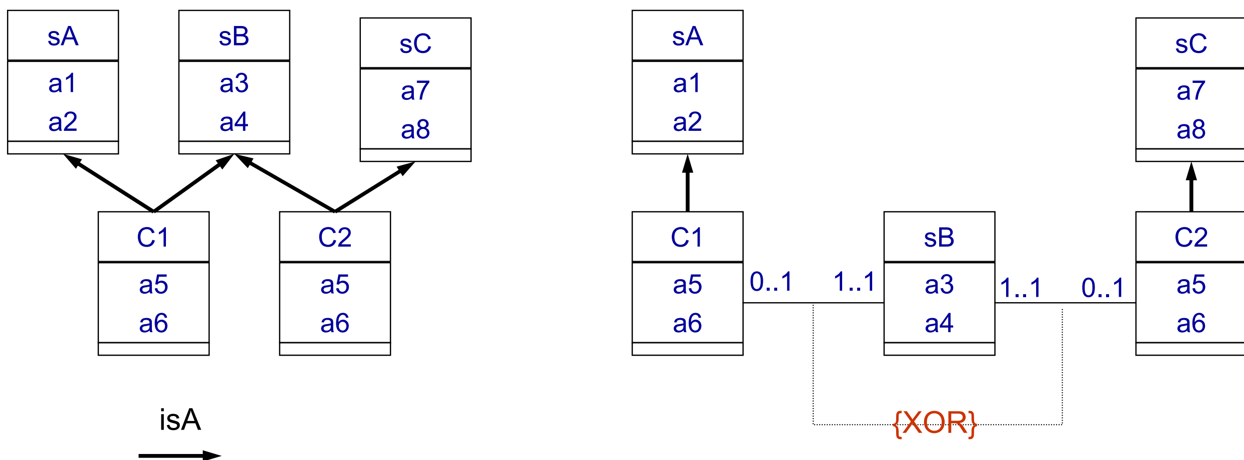
Μετατροπή των επιπλέον υπερκλάσεων σε **συσχετίσεις** με την **κατάλληλη πολλαπλότητα**



Υπόθεση: sB μπορεί να είναι είτε συγκεκριμένη (concrete) ή αφηρημένη (abstract).

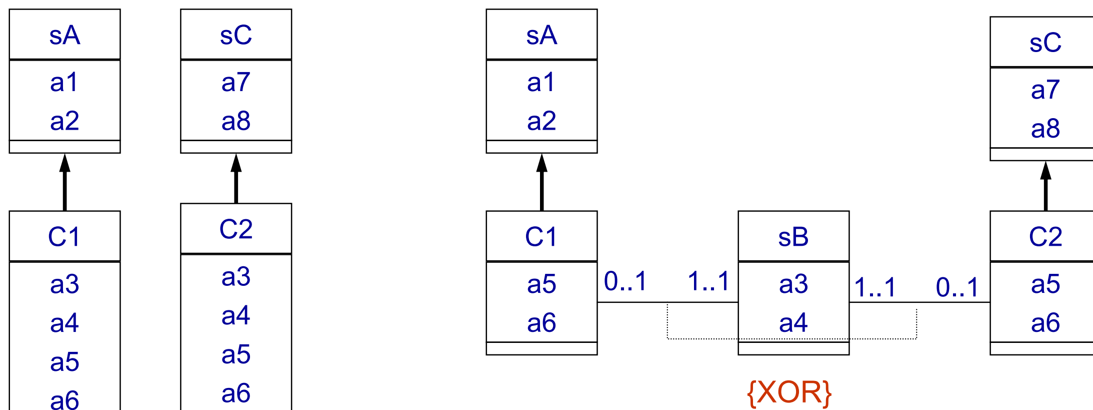
Είναι όλα εντάξει;

Απαλοιφή Πολλαπλής Κληρονομικότητας
Μέθοδος 2: Μετατροπή των Επιπλέον Υπερκλάσεων σε Συσχετίσεις (ΙΙ)



Υπόθεση: sB μπορεί να είναι είτε συγκεκριμένη (concrete) ή αφηρημένη (abstract).

Μέθοδος 1 έναντι Μεθόδου 2



Μέθοδος 2

θετικό: η (εννοιολογική) κλάση (sB) διατηρείται

αρνητικό: αυξάνεται η ανταλλαγή μηνυμάτων και πρέπει να προσέχουμε τον περιορισμό XOR (υπολογιστικά πιο ακριβό)

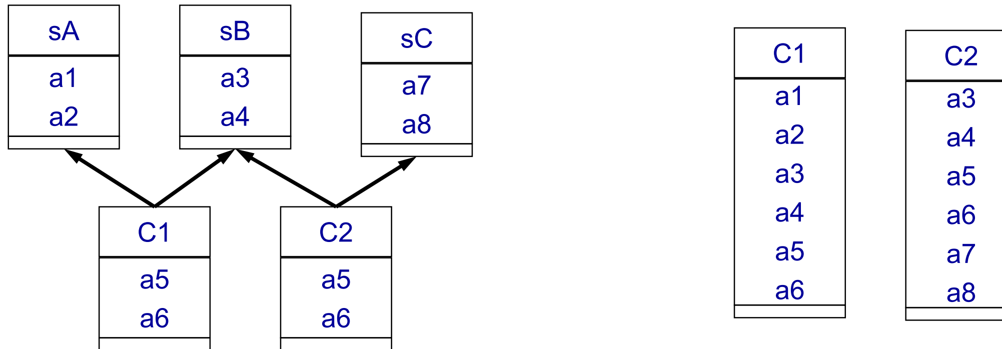
Υπόδειξη: Χρησιμοποιείστε τη μέθοδο 2 μόνο αν η επιπλέον υπερκλάση (sB) είναι concrete (not abstract). Αλλιώς χρησιμοποιείστε τη μέθοδο 1.

Απαλοιφή Κληρονομικότητας

Eliminating Inheritance

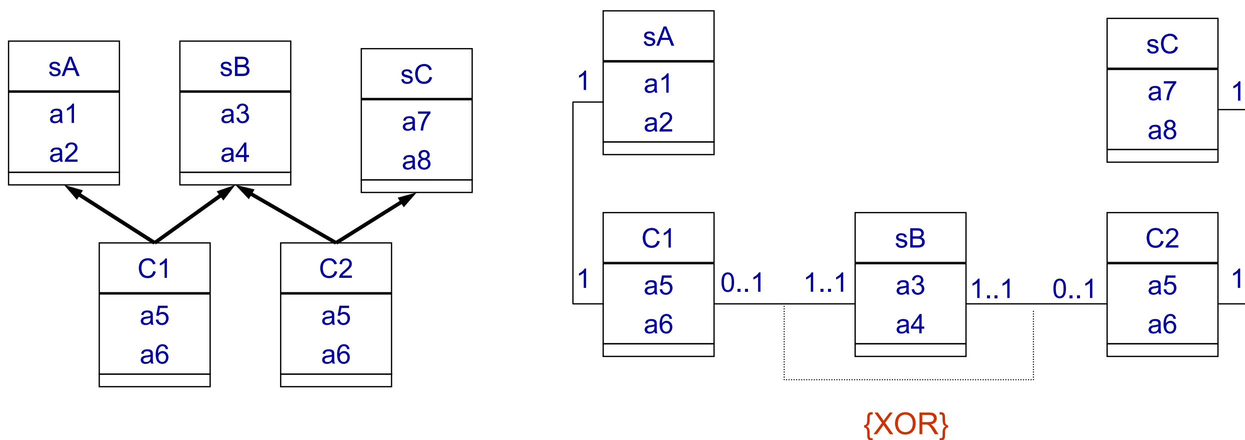
Απαλοιφή Κληρονομικότητας
Μέθοδος 1: Ισοπέδωση (Flattening)

Έστω ότι οι sA, sB και sC είναι όλες αφηρημένες



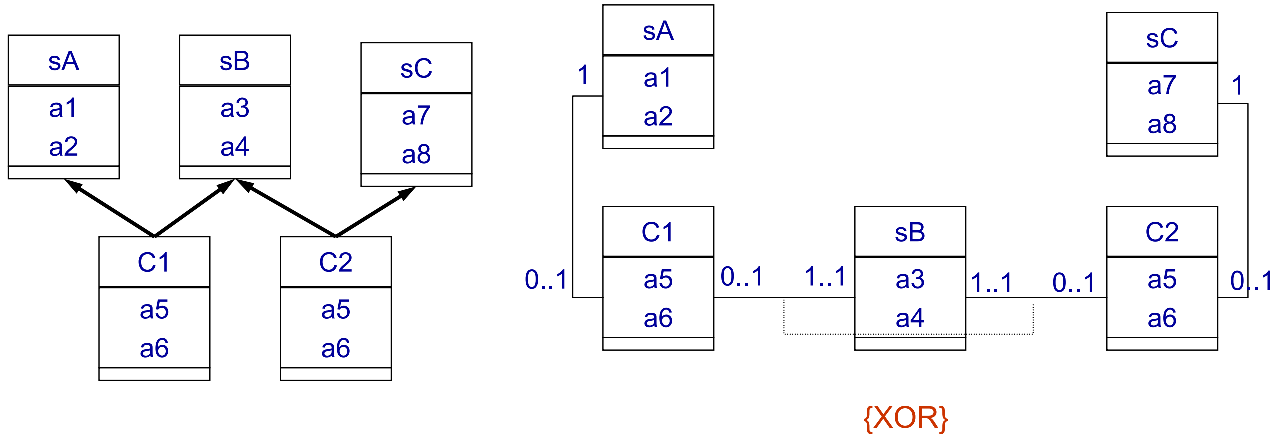
Απαλοιφή Κληρονομικότητας
Μέθοδος 2: Μετατροπή όλων των Υπερκλάσεων σε Συσχετίσεις
Convert all Superclasses to Associations

Έστω ότι οι sA, sB και sC είναι όλες αφηρημένες



Απαλοιφή Κληρονομικότητας
Μέθοδος 2: Μετατροπή όλων των Υπερκλάσεων σε Συσχετίσεις (2)

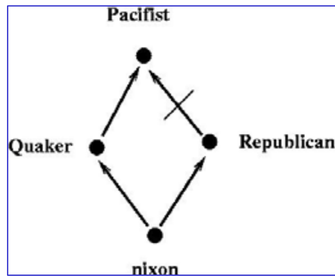
Έστω ότι οι sA, sB και sC είναι όλες συγκεκριμένες (concrete)



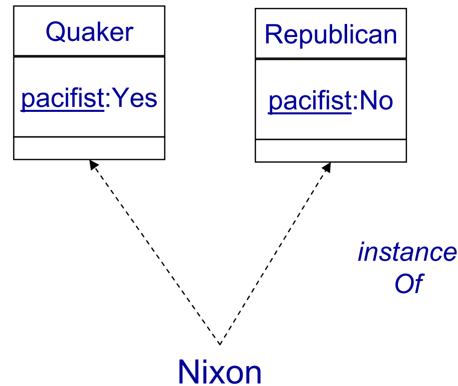
Πολλαπλή Ταξινόμηση
Multiple Classification

Πολλαπλή Ταξινόμηση και Συγκρούσεις
Multiple classification and conflicts

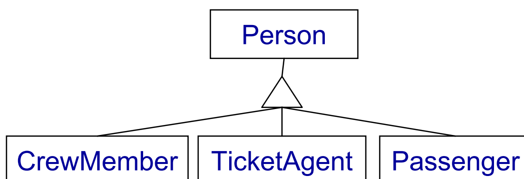
Nixon Diamond



- Nixon is a Quaker.
- Quakers are typically pacifists.
- Nixon is a Republican.
- Republicans are typically not pacifists.
- What to conclude?
  * "Nixon is a pacifist." or
  * "Nixon is not a pacifist."?



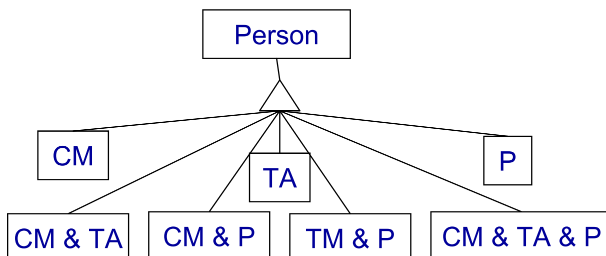
Πολλαπλή Ταξινόμηση
Multiple Classification



Έστω ότι βάσει του μοντέλου προβλήματος, ένα μέλος τους πληρώματος μπορεί να είναι και επιβάτης και έστω ότι περιστασιακά μπορεί να εργαστεί και ως πράκτορας εισιτηρίων

Για να το υλοποιήσουμε αυτό σε Java θα χρειαζόμασταν πολλαπλή και δυναμική ταξινόμηση (multiple and dynamic classification), κάτι το οποίο δεν υποστηρίζεται.

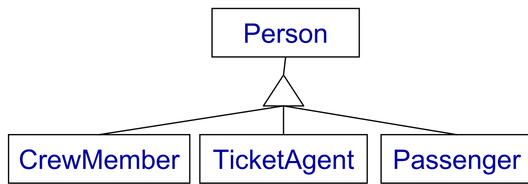
Λύση 1: Για να προσπεράσουμε την πολλαπλή ταξινόμηση, μπορούμε να ορίσουμε “join” classes



Αυτή όμως η λύση μπορεί να απαιτήσει τη δημιουργία εκθετικού πλήθους κλάσεων.

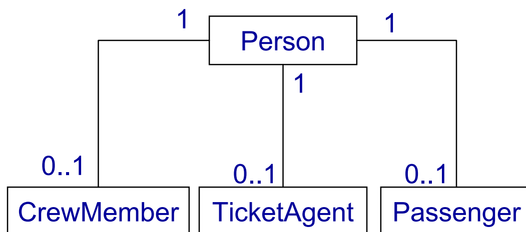
Και δεν δίνει λύση στην ανάγκη *δυναμικής* ταξινόμησης

Πολλαπλή Ταξινόμηση (ΙΙ)
Multiple Classification (II)



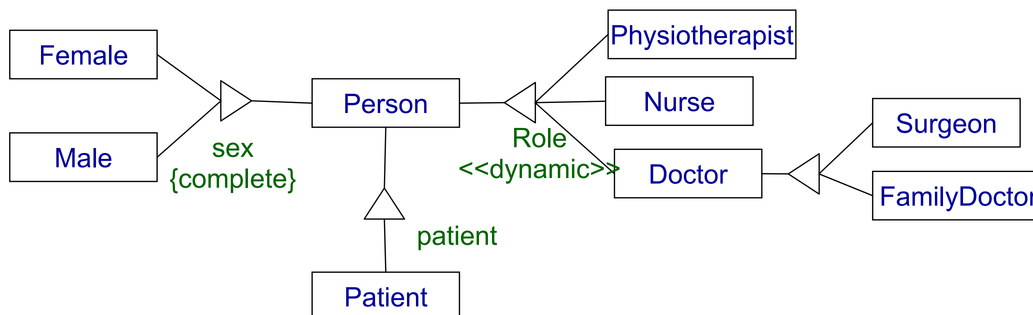
Έστω ότι βάσει του μοντέλου προβλήματος, ένα μέλος τους πληρώματος μπορεί να είναι και επιβάτης και έστω ότι περιστασιακά μπορεί να εργαστεί και ως πράκτορας εισιτηρίων

Λύση 2: **Delegation**



- Delegation είναι ένας τρόπος επέκτασης της συμπεριφοράς μιας κλάσης χωρίς τη χρήση κληρονομικότητας
- Είναι χρήσιμη όταν η πολλαπλή και δυναμική ταξινόμηση δεν υποστηρίζεται από τη γλώσσα προγραμματισμού

Για δική σας εξάσκηση
For homework



Προδιαγραφή Μεθόδων
Method Specification

Προδιαγραφή Μεθόδων
Method Specification

- Σκοπός: Να δώσουμε επαρκείς πληροφορίες στους προγραμματιστές
- Δεν υπάρχει καθιερωμένος μορφότυπος για αυτές τις περιγραφές
- Μερικοί οργανισμοί χρησιμοποιούν φόρμες της μορφής:

Method Name:	Class Name:	ID:
Programmer:	Date due:	
Programming Language:		
Triggers/Events:		
Arguments Received:		
Data Type:	Notes:	
Messages Sent and Arguments Passed:		
ClassName.MethodName:	DataType:	Notes:
Arguments Returned:		
Data Type:	Notes:	
Algorithm Specification:		
Notes:		

Προδιαγραφή Μεθόδων
Method Specification

Method Name: Class Name: ID:
Programmer: Date due:
Programming Language:
Triggers/Events:

Arguments Received:
Data Type: Notes:

Messages Sent and Arguments Passed:
ClassName.MethodName: DataType: Notes:

Arguments Returned:
Data Type: Notes:
Algorithm Specification:

Notes:

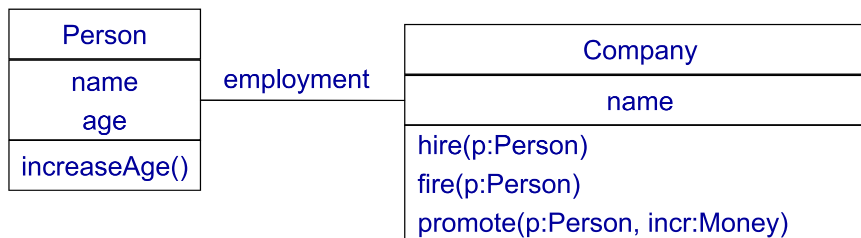
E.g. event-driven programming

Recall behavioural modeling (sequence diagrams, communication diagrams)

Ψευδοκώδικας (Pseudocode)
Δομημένα Αγγλικά (Structured English)
Διάγραμμα Δραστηριοτήτων (Activity Diagr

Περιορισμοί και Συμβόλαια
Constraints and Contracts

Περιορισμοί και Συμβόλαια
Constraints and Contracts



- *Μπορεί ένας ανήλικος να εργαστεί σε μια εταιρία;*
- *Μπορεί μια εταιρία να προσλάβει ένα άτομο που είναι ήδη εργαζόμενος της;*
- *Μπορεί μια προαγωγή να μειώσει το μισθό;*

- Ένα σύνολο περιορισμών που εγγυήσεων για κλάσεις και μεθόδους
- Θα μπορούσαμε να τους εκφράσουμε σε
  - φυσική γλώσσα,
  - δομημένα αγγλικά (ή ελληνικά),
  - ψευδοκώδικα,
  - ή σε μια τυπική γλώσσα (formal language).

Περιορισμοί και Συμβόλαια
Constraints and Contracts

- Ο σχεδιαστής πρέπει να αποφασίσει πως θα διαχειριστεί μια παραβίαση ενός περιορισμού
  - Διακοπή (abort), Ακύρωση/Επανόρθωση (undo), εναπόθεση στο χρήστη (let user handle it)
  - Ο σχεδιαστής πρέπει να σχεδιάσει τα λάθη (errors) που το σύστημα αναμένεται να διαχειριστεί. Είναι καλύτερα να μην αναθέσουμε αυτό το θέμα στους προγραμματιστές
  - Οι παραβιάσεις ενός περιορισμού είναι γνωστές ως **εξαιρέσεις** (exceptions) σε γλώσσες προγραμματισμού όπως η C++/Java.
- *Στο επόμενο μάθημα θα δούμε μια τυπική γλώσσα (που ονομάζεται OCL) για την έκφραση περιορισμών*

Δυνατότητες **Επαναχρησιμοποίησης**
Opportunities to Reuse

Εντοπίζοντας Δυνατότητες Επαναχρησιμοποίησης
Identifying Opportunities for Reuse

Θα μπορούσαμε να εκμεταλευτούμε

- Frameworks
- Class libraries
  - frameworks tend to be more domain specific. Frameworks may be implemented using class libraries
- Design Patterns

Frameworks

- Is a set of implemented classes that can be used as the basis for implementing the system
- Most frameworks allow you to create subclasses
- Frameworks like CORBA and DCOM can be used to specify the physical architecture layer of the system
- Object-persistence frameworks can be used to add persistence to the problem domain classes and thus specify the data management layer of the system. For instance see Spring (http://www.springframework.org/)

Σχεδιαστικά Μοτίβα
Design Patterns

Design patterns

A pattern is a commonly occurring reusable piece in software system that provides a certain set of functionality.

- Using patterns in modeling of systems helps in keeping design standardized and more importantly, minimizes the reinventing of the wheel in the system design.
- *How design patterns relate to UML ?*
  - The patterns need to be captured and documented in a sufficiently descriptive manner so that they can be referred for future use.
  - UML provides the perfect tools to do just this. The class diagram in UML can be used to capture the patterns identified in a system. In addition, UML has a sufficiently extensive and expressive vocabulary to capture the details of patterns.

Κατηγοριοποίηση Προτύπων
Categorizing Patterns

Based on how they are to be used, patterns are primarily categorized as:

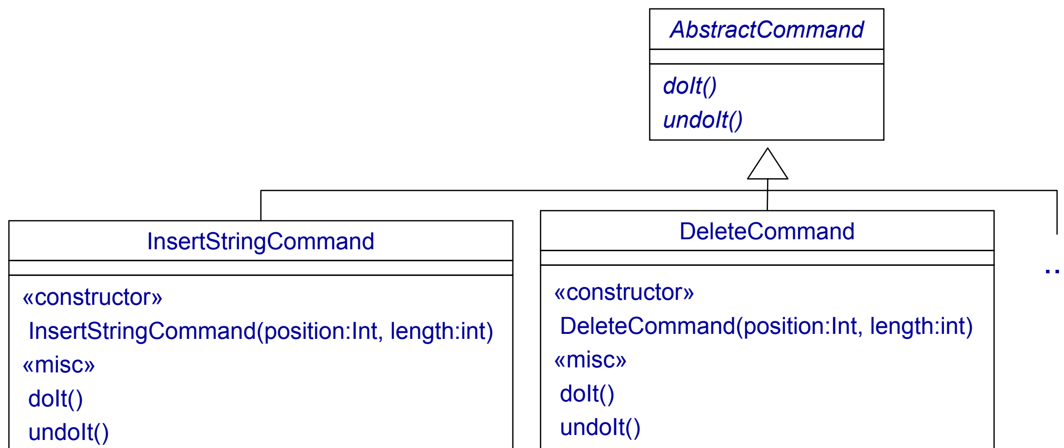
- **Creational**
  - They define mechanisms for instantiating objects. The implementation of the creational pattern is responsible for managing the lifecycle of the instantiated object.
  - Examples: Factory, Singleton
- **Structural**
  - They define compositions of objects and their organization to obtain new and varied functionality.
  - Examples: Adapter, Proxy.
- **Behavioral**
  - They define interaction between different objects.
  - Examples: Command, Iterator

Command

Encapsulate commands in objects so that you control their selection and sequencing, queue them, and othewise manipulate them

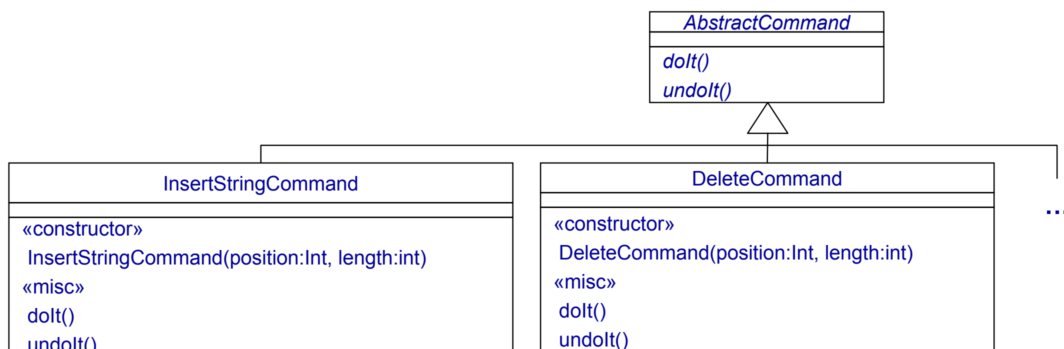
Example:

Design of a Word Processor that supports undoing and redoing commands

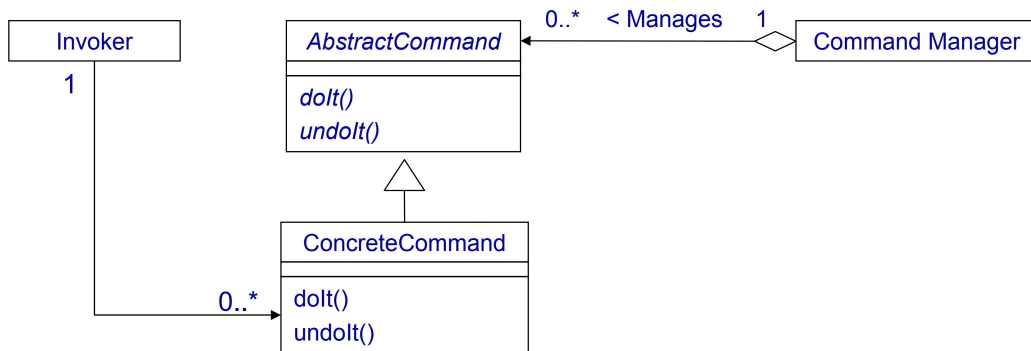


Design of a Word Processor supporting Undo and Redo

- materialize each command as an object with do and undo methods.
- when the user tells the WP to do something instead of performing the command,
  - it creates a new object using the appropriate constructor (e.g, an InsertStringCommand object)
  - it then calls the object's doIt method to execute the command
- The WP also puts the command object in a data structure that allow the WP to maintain a history of what commands have been executed. This allows the WP to undo commands in the reverse order that they were issued by calling their undo methods.

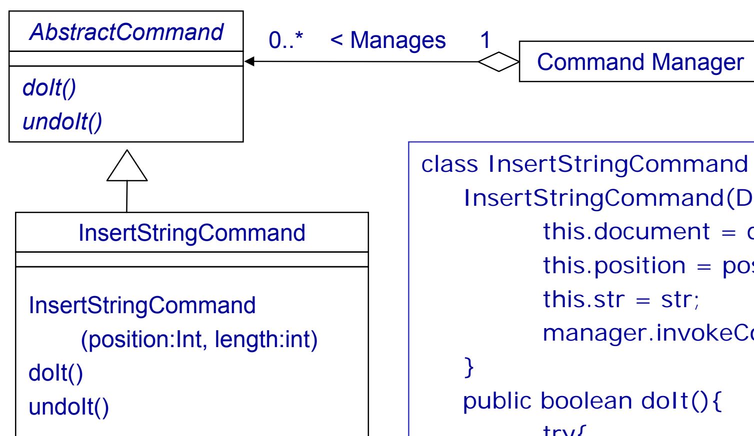


Behavioral Patterns> Command
Design of a Word Processor supporting Undo and Redo




```
public abstract class AbstractCommand{
        public final static CommandManager manager
                =new CommandManager();
        public abstract boolean doIt();  // returns True if successful
        public abstract boolean undoIt();
}
```

Behavioral Patterns> Command
Design of a Word Processor supporting Undo and Redo (2)




```
class InsertStringCommand extends AbstractCommand{
    InsertStringCommand(Document doc, int position, String str){
        this.document = document;
        this.position = position;
        this.str = str;
        manager.invokeCommand(this);
    }
    public boolean doIt(){
        try{
            document.insertStringCommand(position, str);
        } catch (Exception e){
            return false;
        }
        return true;
    }
```

Behavioral Patterns> *Command*
Design of a Word Processor supporting Undo and Redo (3)

```
class CommandManager{
    private int maxHistoryLength = 20;
    private LinkedList history = new LinkedList();
    private LinkedList redoList = new LinkedList();
    public void invokeCommand(AbstractCommand command){
            if (command instanceOf Undo){
                undo(); return;
            }
            if (command instanceOf Redo){
                redo(); return;
            }
             if (command.doIt()) {
                addToHistory(command);
            } else {
                history.clear()
            }
            if (redoList.size()>0)
                redoList.clear();
  }
```

```
interface Undo {
}
interface Redo{
}
```

```
Private void addToHistory(AbstractCommand command){
    history.addFirst(command);
    if (history.size() > maxHistoryLength) {
            history.removeLast();
    }
}
```

Behavioral Patterns> *Command*
Design of a Word Processor supporting Undo and Redo (4)

Command Manager

```
Private void undo(){
    if (history.size() >0) {
            AbstractCommand undoCommand;
            undoCommand = (AbstractCommand) history.removeFirst();
            undoCommand.undoIt();
            redoList.addFirst(undoCommand);
    }
}
```

```
Private void redo(){
    if (redoList.size() >0) {
            AbstractCommand redoCommand;
            redoCommand = (AbstractCommand) redoList.removeFirst();
            redoCommand. doIt();
            history.addFirst (redoCommand);
    }
}
```

Design Patterns

- Design patterns is a useful mechanism to document and learn about common reusable design approaches.
- Design patterns can reduce the designing time for building systems and ensure that the system is consistent and stable in terms of architecture and design.
- The UML class diagrams provide an easy way to capture and document Design patterns
- Suppose that you want to specify a software component. If that component could be based on a design pattern you could just mention it and give a small description. You don't necessarily have to redefine the pattern (i.e. provide the detailed class diagram and the various interaction diagrams)
- Some UML tools support design patterns.
  - They have a pre-built catalog of well-known design patterns. The design patterns can be easily pulled in into your design as templates and then customized for your application design.

More on patterns at

•***CS352 - Software Engineering***

•***You can also see* http://www.research.umbc.edu/~tarr/dp/fall00/cs491.html**

•(it is a university course dedicated to patterns in Java)

Reading and References

- **Systems Analysis and Design with UML Version 2.0** (2nd edition) by A. Dennis, B. Haley Wixom, D. Tegarden, Wiley, 2005. Chapter 10
- **Requirements Analysis and System Design** (2nd edition) by Leszek A. Maciaszek, Addison Wesley, 2005, Chapter 5 and 6
- **Inheritance Hierarchies in KR and Programming Languages**, Lenzerini, Nardi, Simi, 1991
- **Patterns in Java,** Mark Grand, Wiley, 1998
- **Using Design Patterns in UML,** Mandar Chitnis, Pravin Tiwari, & Lakshmi Ananthamurthy
- **Slides of John Mylopoulos, University of Toronto**
- **Αντικειμενοστρεφής Σχεδίαση: UML, Αρχές, Πρότυπα και Ευρετικοί Κανόνες,** Α. Χατζηγεωργίου, Κλειδάριθμος 2005